LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-61

FIRST VERSION OF A DATA FLOW PROCEDURE LANGUAGE

Jack B. Dennis

May 1975

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

First Version of a Data Flow Procedure Language

by

Jack B. Dennis

Abstract: A language for representing computational procedures based on the concept of data flow is presented in terms of a semantic model that permits concurrent execution of noninterfering program parts. Procedures in the language operate on elementary and structured values, and always define functional transformations of values. The language is equivalent in expressive power to a block structured language with internal procedure variables and is a generalization of pure Lisp. The language is being used as a model for study of fundamental semantic constructs for programming, as a target language for evaluating translatability of programs expressed at the user-language level, and as a guide for research in advanced computer architecture.

Introduction

The Computation Structures Group at Project MAC is studying the design of a "common base language" which could serve as a target representation for the translation of important facilities of user programming languages, and therefore could be used as a guide to the conception of future general-purpose computer systems. The objectives and rationale for this work have been presented in [Dennis 1971]. An earlier paper [Dennis 1969] gave some thoughts regarding the sort of computer system organization that may result from pursuing this direction of research. In particular, we envision a highly parallel computer system in which the execution of many program fragments is carried forward simultaneously. But this high level of concurrent activity is to be achieved with no sacrifice in the generality of programming that is possible. In fact, we insist that the base language contribute significantly to the ability of computer users to easily construct correct programs.

This work was supported by the National Science Foundation under research grant GJ-34671.

In [Dennis 1971] we outlined a rudimentary base language using directed graphs having branches labelled with selectors as the basic data structures, and using conventional instruction sequencing. By giving translation rules for a hypothetical block structured language, we have shown that such a base language is sufficiently general to encompass important linguistic features of source languages. Because of our interest in concepts of computer organization that permit possibilities for concurrent execution of program parts to be exploited, and because we are attracted to schemes of representation that expose concurrency while maintaining a guarantee of determinacy, we are seriously studying proposals for a base language founded on the notion of data flow.

In a data flow representation, execution of a test or operation is enabled by availability of the required operand values. The completion of one operation or test makes the resulting value or decision available to the elements of the program whose execution depends on them. A simple data flow model was studied by [Karp and Miller]. A little later [Rodriguez] developed his "program graphs" which have been revised and analyzed as a form of parallel program schema by [Dennis and Fosseen]. Other data flow models include the work of [Adams]; the models of [Bährs] and [Kosinski] are closely related to ours but were independently conceived.

In the present paper we propose a data flow representation for programs that is sufficiently general to encompass the same range of source program semantics as the earlier base language using conventional control flow. This language is a revision and extension of the representation outlined in [Dennis 1969]. We will only consider programs that compute a set of output values from a given set of input values, and that define a functional dependence of output values on input values. In a concluding section we discuss extensions we wish to make so the data flow language will encompass a more complete set of programming constructs.

This document was originally published as Computation Structures Group Memo 93, November 1973, and was revised in August 1974.

It also appeared in the Proceedings of Symposium on Programming, Institut de Programmation, University of Paris, France, April 1974, 241-271.

Elementary Data Flow Programs

We begin with a data flow representation for programs that do not involve data structures or procedure applications. Consider the following program expressed in an Algol-like notation:

```
input (w, x)
    y := x; t := 0;
    while t ≠ w do
    begin
        if y > 1 then y := y ÷ 2
                 else y := y × 3;
        t := t + 1;
    end
output y
```

Variables w and x are input variables of the program and y is the output variable. An elementary data flow program equivalent to the above program is shown in Figure 1. It is a bipartite directed graph where the two types of nodes are called links and actors. The arcs of a data flow program should be regarded as channels through which tokens flow carrying values from each actor to other actors by way of the links.

Certain data links having no incident arcs are specified to be input links of a data flow program and certain data links are specified to be output links. Each link that is not an input link must have exactly one incident arc, and each link that is not an output link must have at least one emanating arc. In Figure 1, x and w are the input links and y is the output link. Figure 2 gives the notation and terminology used for the two kinds of link nodes and the eight kinds of actor nodes from which elementary data flow programs are constructed.

The execution of a data flow program is described by a sequence of snapshots; each snapshot shows the data flow program with tokens and associated values placed on some arcs of the graph of the program. In the case of control arcs, the assocaited values are of type truth = {true, false}; for data arcs, the values are of type integer, real or string. Execution of a data flow program advances from one snapshot to the next through the firing of some link or actor that is enabled in the earlier snapshot. The rules

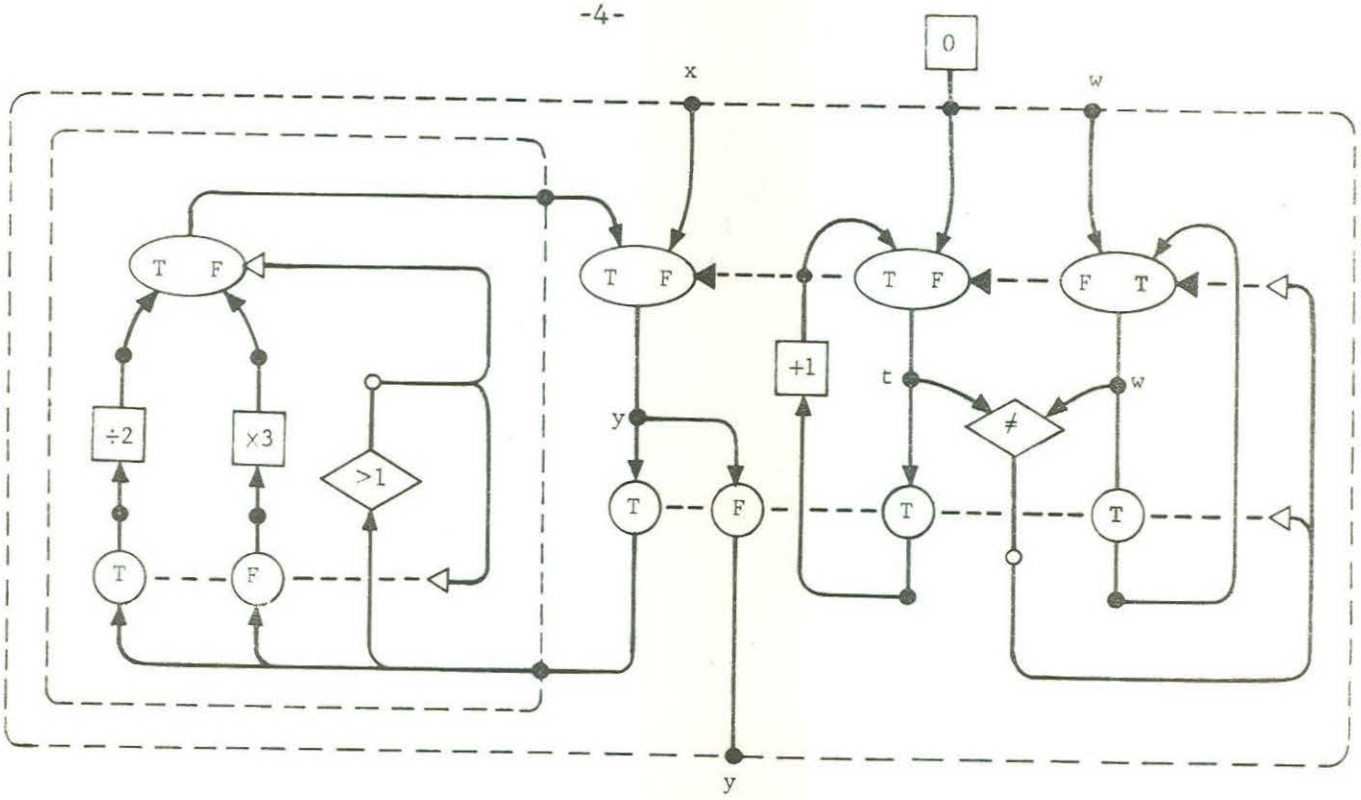


Figure 1. An elementary data flow program.

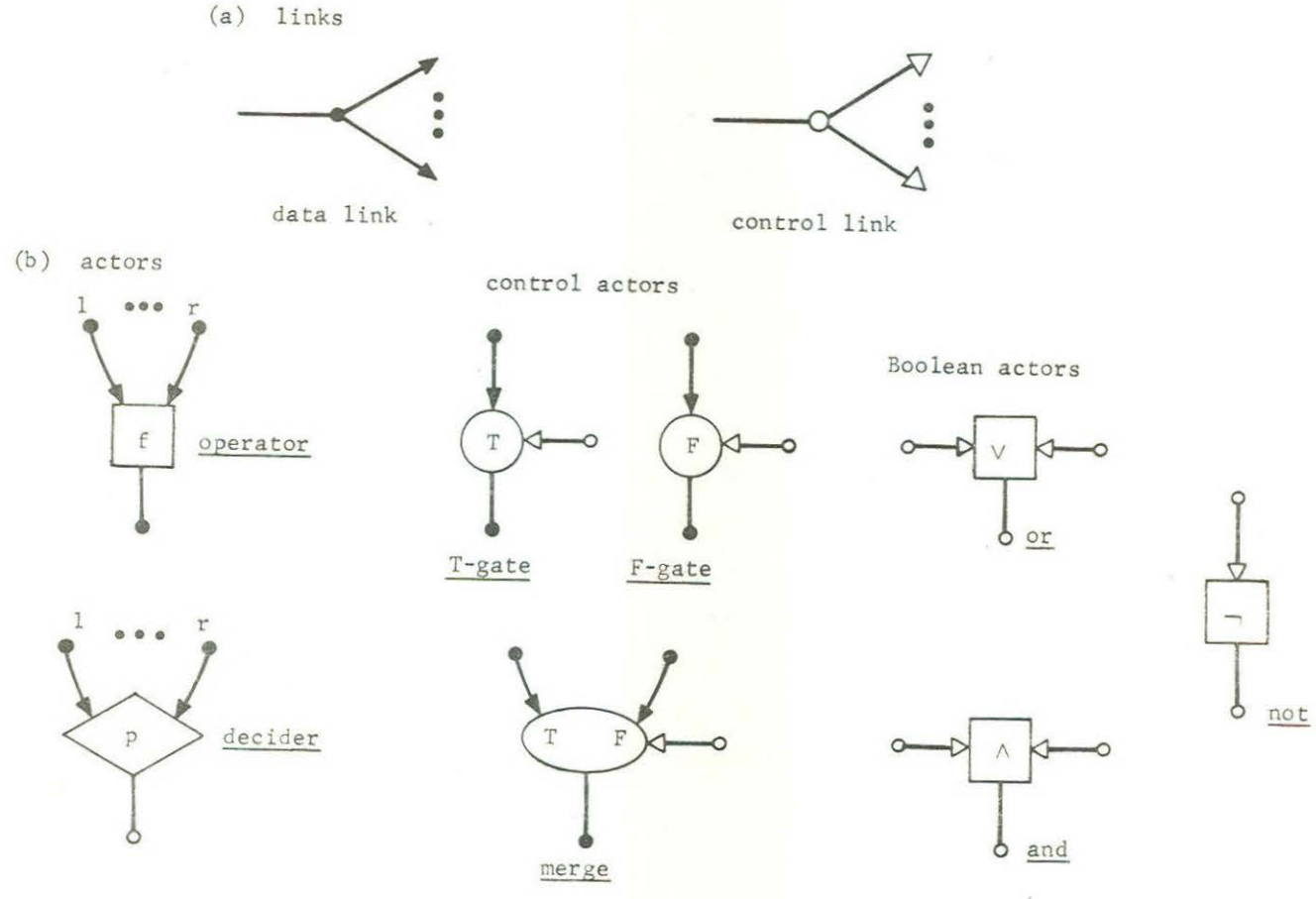


Figure 2. Node types for data flow programs.

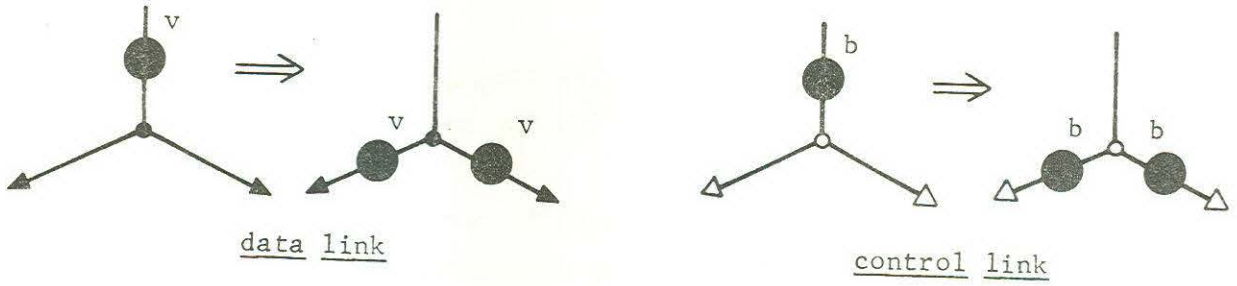


Figure 3. Firing rules for link nodes.

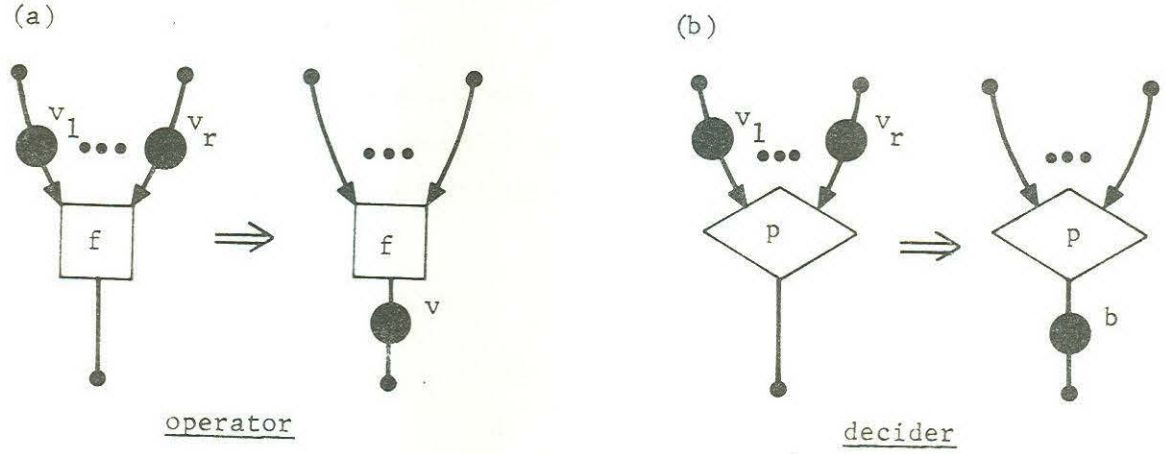


Figure 4. Firing rules for operators and deciders.

governing the enabling and firing of the link nodes are given in Figure 3, and the rules for most of the actors are given in Figures 4 and 5. In each case a condition for which a node is enabled is shown on the left and the new condition that results from firing the node is shown on the right. In addition, a necessary condition for any node to be enabled is that each output arc does not hold a token.

From the figures, we see that, except for the control actors, a node is enabled just when tokens are present on all input arcs and no token is present on any output arc. Then firing the node absorbs the tokens from the input arcs and places tokens on the output arcs. The two kinds of links (Figure 3) replicate data and control values for distribution to several actors. Firing an operator (Figure 4a) applies the function denoted by the symbol written in the operator to the set of values associated with the tokens on the input arcs and associates the resulting value $v = f(v_1, \ldots, v_r)$ with the token placed on the output arc. Firing a decider (Figure 4b) has a similar effect, but the symbol in the decider denotes a predicate, and a control value $b = p(v_1, \ldots, v_r)$ is

associated with the output token. The three Boolean actors shown in Figure 2 act with respect to control values just as an operator acts with respect to data values.

The gate and merge actors (Figure 5) permit the outcomes of tests to affect the flow of values to operators and deciders. A T-gate, for example, passes a value on to its output arc if it receives the value true at its control input arc; the received data value is discarded if false is received. The merge node allows a control value to determine which of two sources supplies a data value to its output arc. If the control value false arrives at the control arc the merge passes on the value present or next to arrive at the F-input arc. A value present at the T-input arc is left undisturbed. The complementary action occurs for the control value true.

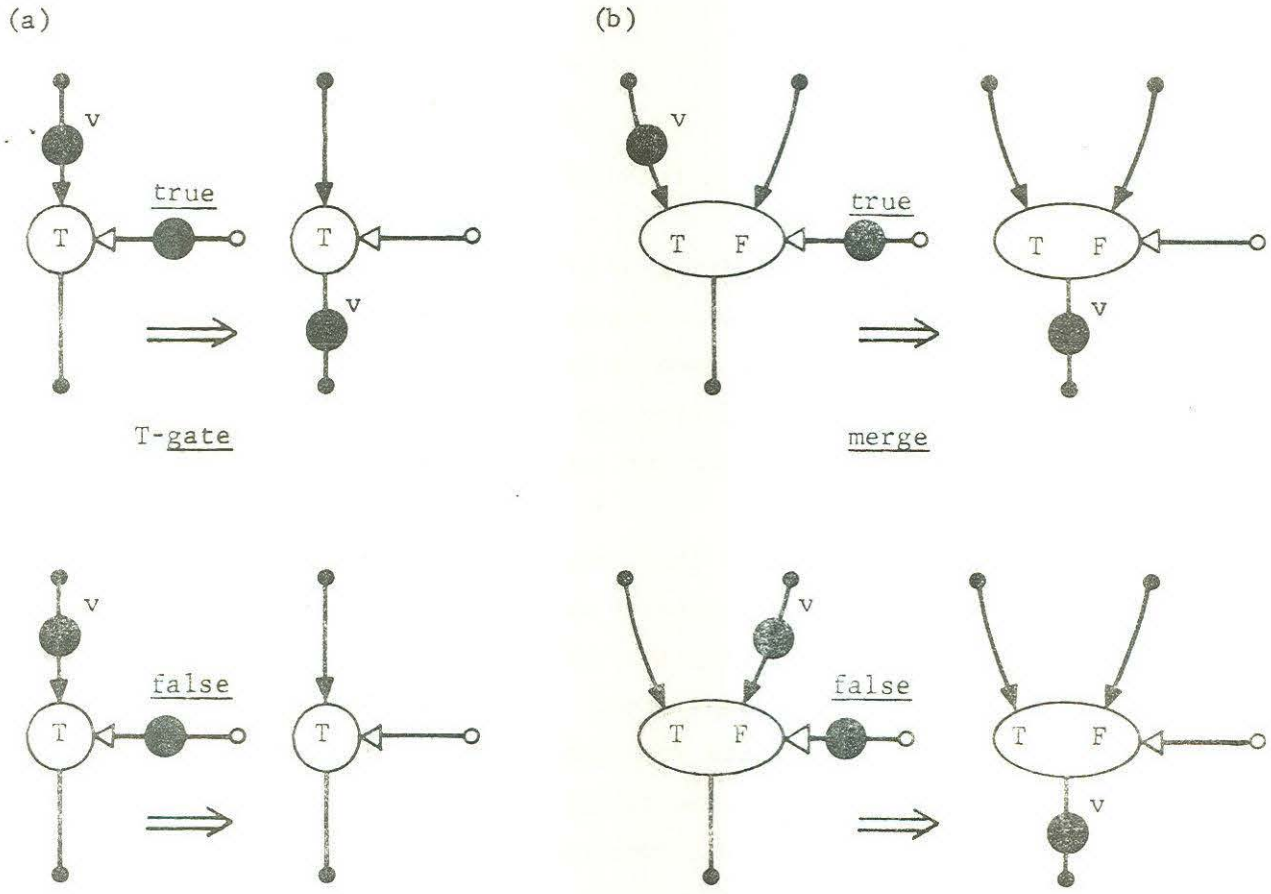


Figure 5. Firing rules for control actors.

For the present, we will consider only data flow programs that produce exactly one set of result values at the output links for each set of values presented at the input links. Such data flow programs are called <u>well behaved</u> when, as in Figure 6a, tokens with associated values are presented at the input links, and the program P has a certain initial distribution of tokens on its arcs, execution will advance until just one token with an associated value has appeared at each output link, as in Figure 6b, and the initial distribution of tokens in P has been reestablished. (We also allow the possibility that P may execute forever without putting tokens on all of its output arcs.)

Well behaved elementary data flow programs are always <u>functional</u> in the sense that a unique set of output values is determined by any set of input values. The functionality of data flow programs follows from work of [Patil], [Lohr], and [Kahn]: The links and actors of data flow programs are determinate systems in the sense of [Patil] and the rules of behavior ensure that the property of determinacy is preserved for the operation of interconnecting links and actors to form a data flow program.

The data flow program in Figure 1 is well behaved. It has specific constructions that correspond to conventional programming constructs for condi-

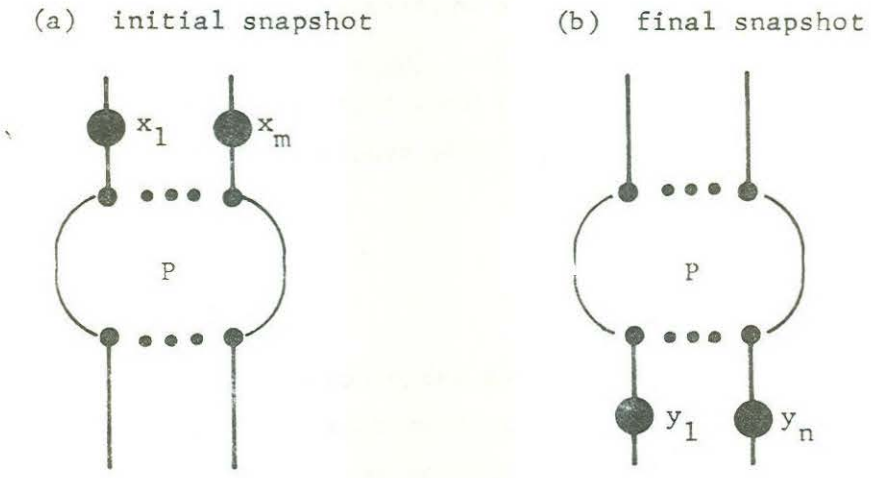


Figure 6. Well behaved data flow programs.

tionals and iteration. The smaller dashed box encloses a data flow subprogram that corresponds to the conditional phrase

```
if y > 1 then y := y ÷ 2
         else y := y x 3
```

This conditional data flow program and the +1 operator constitute a well behaved data flow program that is the body of the iteration data flow subprogram contained within the larger dashed outline. The iteration proceeds by circulating values for y and t through the body until a false result is obtained by the inequality decider. The solid arrowheads on the control arcs that terminate on merge nodes of the iteration program indicate that tokens carrying false are present on these arcs in the initial token distribution for the program. These tokens are required so the merge nodes will pass initial values into the body of the iteration. No other arcs of an elementary data flow program hold tokens in the initial token distribution.

Elementary data flow programs are obtained from the data flow schemas of [Dennis and Fosseen] by interpreting each function or predicate letter of the schema as some particular primitive operation on elementary values of type integer, real or string.

It is easy to verify [Fosseen] that data flow schemas are equivalent in expressive power to formal programs written using assignment statements, while...do... constructions and if... then...else... constructions. Hence the results of [Ashcroft and Manna] show that data flow schemas are equivalent in power to program schemas of the type studied in [Paterson] and [Luckham, Park and Paterson].

Extended Data Flow Programs

The concepts of data structures and procedure applications both embody possibly unbounded expansion of execution constructs whereas the snapshots of elementary data flow programs are bounded by the size of the program itself. To extend the data flow model to incorporate these important notions, we introduce a form of heap to hold data structures, and we introduce colored tokens to distinguish among concurrent activations of the same data flow procedure.

Data Structures -- The Heap

The domain of values for extended data flow programs includes both elementary values and structured values. The set of elementary values is truth ∪ integers ∪ reals ∪ strings. The set of structured values contain all finite sets of selector-value pairs such as

$$[\langle s_1 : v_1 \rangle, \ldots, \langle s_k : v_k \rangle]$$

where the selectors $s_1, \ldots, s_k$ are distinct elements in integers ∪ strings, and $v_1, \ldots, v_k$ are arbitrary values, either elementary or structured. The empty set is a structured value and is denoted by nil.

In our semantic model for extended data flow programs, values are represented by a heap, which is a finite, acyclic, directed graph having one or more root nodes, and such that each node of the heap may be reached over some directed path from some root node. The branches of the heap are labelled with selectors where no two branches emanating from the same node may bear the same selector. The nodes of a heap are of two types: elementary nodes and structure nodes. An elementary node has no emanating branches, and has an associated elementary value. The root nodes may be of either type.

Each node of the heap represents a value as follows: An elementary node represents its associated elementary value. A structure node represents the structured value

$$[\langle s_1 : v_1 \rangle, \ldots, \langle s_k : v_k \rangle]$$

where $s_1, \ldots, s_k$ are the selectors on branches emanating from the node, and each v_i is the value represented by the node on which the branch labelled s_i terminates. (A node of the heap that has no emanating branches and no associated elementary value represents the empty set nil.)

Pointers are objects that correspond uniquely to nodes of the heap, and will be represented either by Greek letters or by arrows directed to the corresponding heap nodes. For simplicity we suppose that a single node of the heap represents nil, and we will use nil also to denote the pointer for this node.

In extended data flow programs, a data token always has an associated pointer that designates some node of the heap. One may regard the token as carrying the value (elementary or structured) represented by the corresponding node of the heap.

The actors of extended data flow programs are designed so their execution does not change values represented by the heap. Instead, whenever a new value is created, a node is added to the heap to represent the new value. For instance, an operator may fire if its input tokens carry pointers referring to elementary values. Firing an operator creates a new elementary node in the heap with the result of operator execution as its associated (elementary) value. Consequently, one can correctly understand the semantics of data flow programs by regarding tokens as carriers of values (elementary or structured), and ignoring pointers and the heap. We have included the heap in our semantic model only to demonstrate that implementation is possible without need of copying arbitrarily complex values.

A snapshot of a data flow program in execution will now have two parts: a token distribution on the graph of the program, and a heap. For each execution step some enabled link or actor is selected to fire; the result of firing is a new token distribution and, in some cases, a modified heap.

Parts of the heap implicitly disappear when they are no longer accessible, where a node of the heap is <u>accessible</u> in a snapshot if and only if either

1. Some token in the token distribution has a pointer corresponding to the node.
2. The node is reachable by some directed path in the heap starting from an accessible node.

An execution step is performed as follows:

1. Some enabled node of the program is selected for firing.
2. The node is fired yielding a new token distribution and a new heap.
3. Inaccessible nodes in the heap are deleted together with any emanating branches.

The behavior of the new actors for extended data flow programs is specified in Figures 7 through 10. As in the case of elementary data flow programs, an actor is enabled only when its output arc is empty.

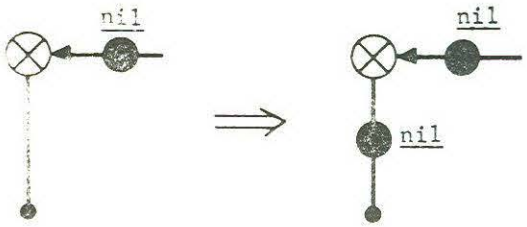


Figure 7. The source actor.

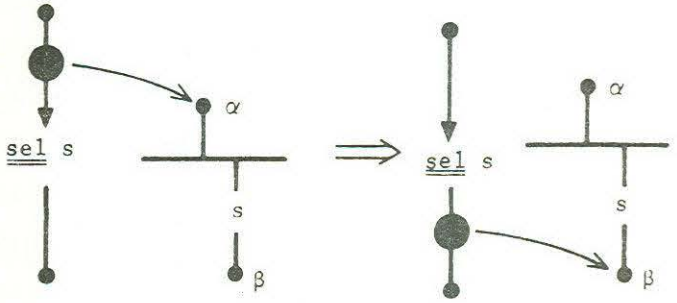


Figure 8. The selector actor.

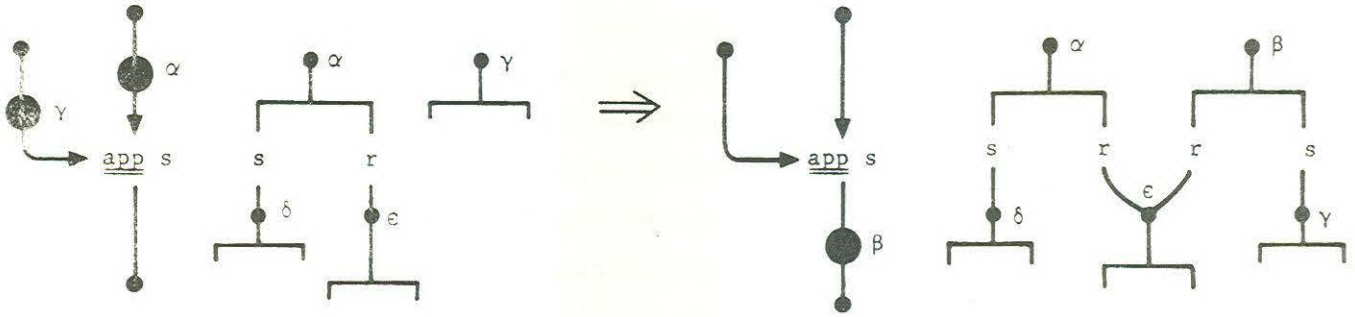


Figure 9. The append actor.

A source actor (Figure 7) has an implicit input arc that always holds a token carrying the pointer nil. It is enabled whenever there is no token on its output arc, and produces a token carrying the pointer nil.

Figure 8 gives the behavior of a select actor in which a fixed selector s is specified. Given a pointer α that refers to a structured value x in the heap, the actor sel s delivers a pointer β that refers to the s-component of x. If x has no s-component, then no action is defined.

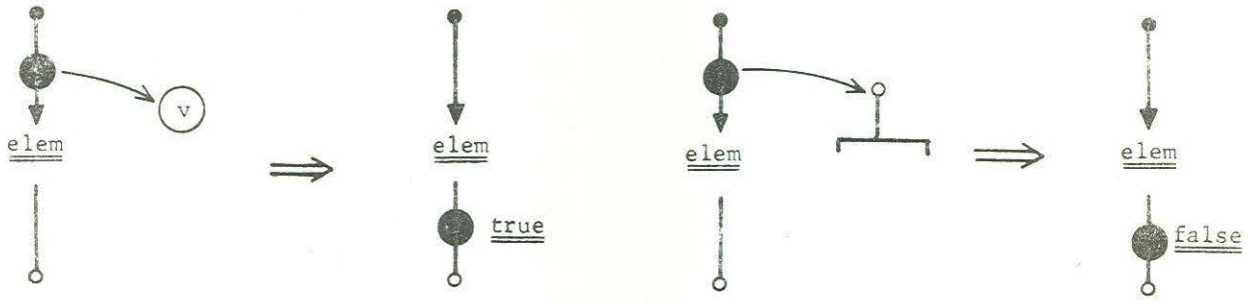
The append actor (Figure 9) is the means for creating new structured values. If pointer α refers to a structured value x and γ refers to any value z, the actor app s produces a pointer β that refers to a new value y such that the s-component of y is z and all other components of y are equal to the corresponding components of x. The value to which α refers remains x in keeping with the principle that values represented by heap nodes never change.

Two actors are provided for testing values represented in the heap. The actor elem (Figure 10a) yields true if the pointer presented refers to an elementary value, or false if the pointer refers to a structured value. The actor exists s (Figure 10b) yields true if the given pointer refers to a structured value having an s-component, and yields false otherwise.

Actors select, append and exists are also provided for use when the selector for a component of a data structure is the result of a computation. These actors have an additional data input arc on which the selector is presented to the actor.

Since only append actors cause branches to be added to the heap, and the added branches always emanate from a new structure node, invariance of the acyclic property of the heap is guaranteed. Furthermore, only operator and append actors have any effect on the heap, and these actors add new nodes without changing the values represented by existing nodes. The behavior is as if values were carried by the tokens and the heap did not exist. Thus well behaved extended data flow programs define functional transformations of values into values for the same reason that elementary data flow programs are functional.

(a) the element actor



(b) the existence actor

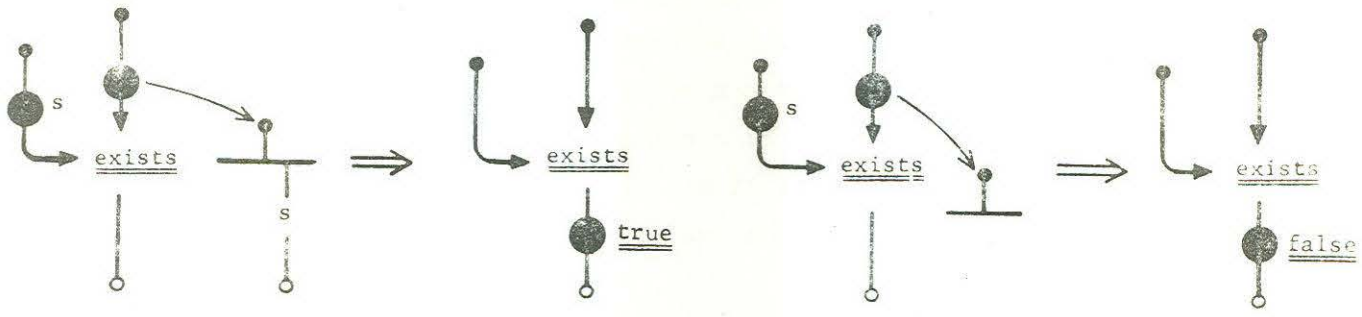


Figure 10. The element and existence actors.

As an example, we give in Figure 11 a data flow program that computes the sum of two vectors. For simplicity, link nodes are not indicated explicitly when they have a single output arc. The corresponding conventional program is the following:

```
input (a, b, n)
    i := 1;
    while i ≤ n do
    begin
        c[i] := a[i] + b[i];
        i := i + 1;
    end
output c
```

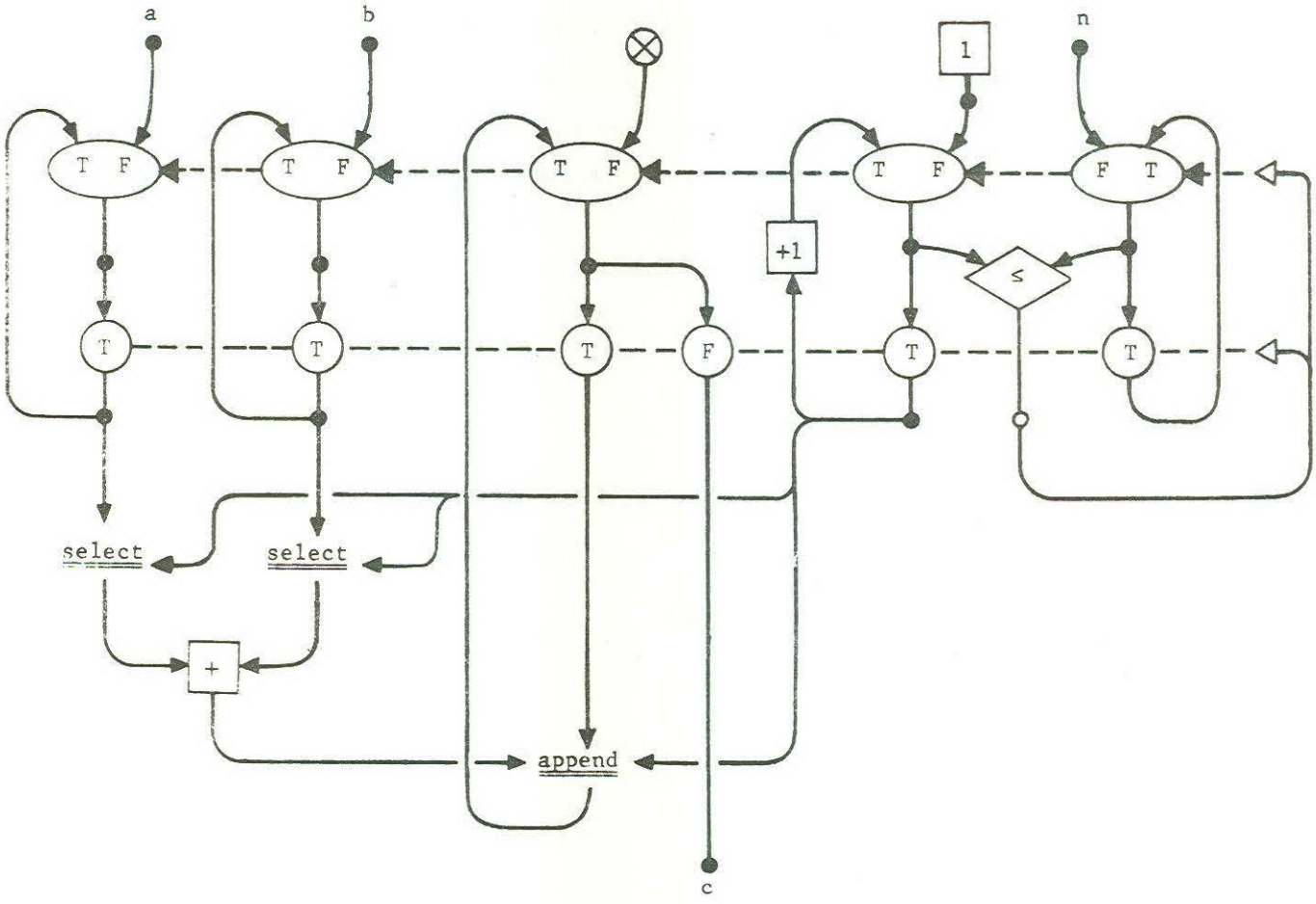


Figure 11. Data flow program to add two vectors.

Data Flow Procedures -- Colored Tokens

A data flow procedure is a data flow program in which the actors apply and proced are allowed as well as the actors already introduced, and which has a single input link designated A (for argument) and a single output link designated R (for result).

To provide for procedure application, we introduce the actors proced and apply, and a third type of node in the heap. The new node type is a text node and has an associated text value that represents some data flow procedure.

Many concurrent activations of a data flow procedure may exist as a result of concurrent or recursive procedure applications. For the meaning of such programs to be clear, we must avoid the possibility of confusing the tokens present in a data flow procedure on account of two or more distinct activations of the procedure. For the purpose of this presentation we solve this problem by using the notion of colored tokens. Every token is tagged with a color that uniquely identifies the procedure activation it is associated with. All tokens belonging to the same procedure activation have the same color.

In using colored tokens, the firing rules for links and actors must be recast, so that tokens of different colors have no interaction in a data flow procedure, and token distributions must be generalized so that arbitrarily many tokens may occupy an arc simultaneously, so long as each has a differ-

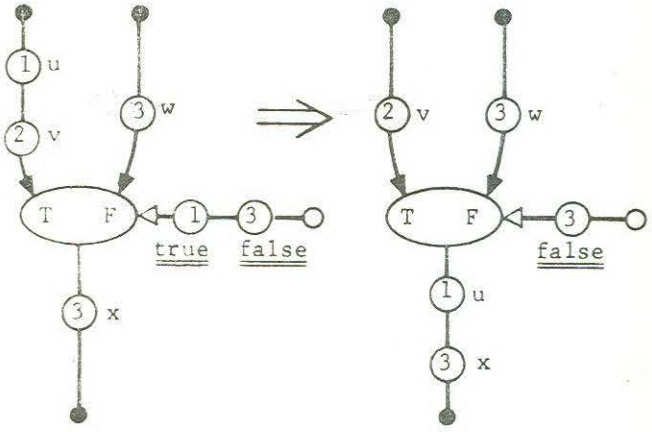


Figure 12. Firing rule for colored tokens.

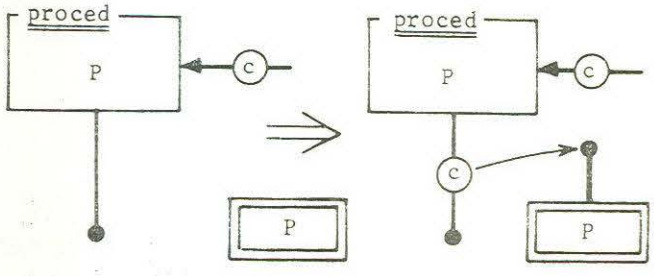


Figure 13. The procedure actor.

ent color. Figure 12 illustrates the revised firing rules for a merge node. A data token and a control token of the same color must be present for the merge node to fire. Also, only the presence of a token of this color on the output arc can prevent the actor from firing.

A procedure actor (Figure 13) contains a text value that represents some data flow procedure. Firing a procedure actor yields a pointer for a new text node in the heap that represents the text value of the procedure actor.

An apply actor (Figure 14) should be regarded as consisting of an activate actor and a terminate actor joined by an internal arc. The left input arc of an apply actor is called the procedure arc and must present a pointer that refers to a text value representing some data flow procedure P. The right input arc is called the argument arc and conveys a pointer to the argument structure for the desired application. As shown in Figure 14a, the first firing acti-

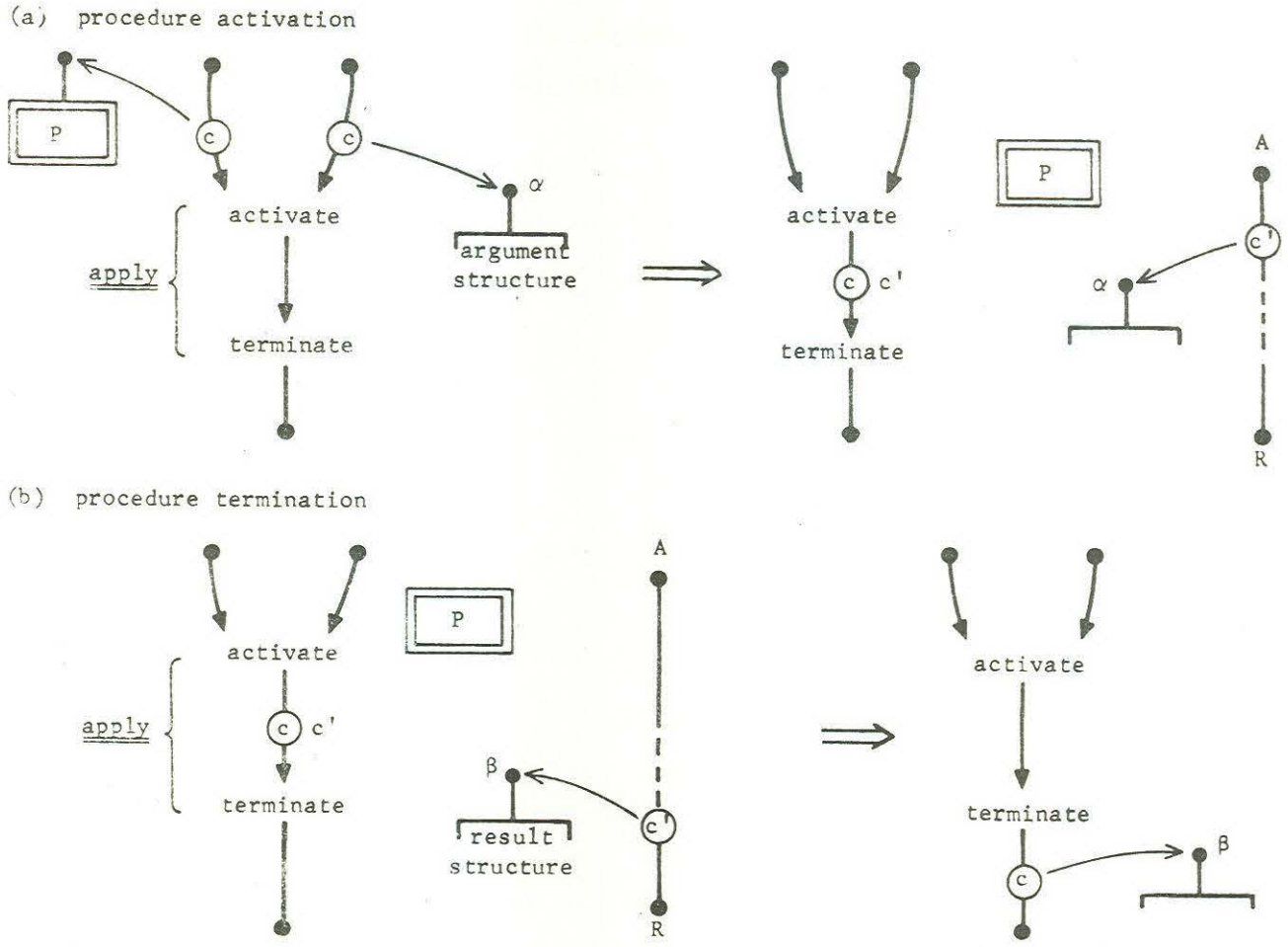


Figure 14. The apply actor.

vates P by choosing a new color c' and placing a token of color c' on the input arc of P, with a pointer to the argument structure. Tokens of color c' are also put on arcs as required to initialize the actors of P. Also a token of color c is placed on the internal arc of the apply actor with c' as its associated value. This token serves to identify the site at which the result value is to be delivered when the application of P terminates.

Termination occurs when a token of color c' arrives at the output arc of P with a pointer to a result structure. As shown in Figure 14b, the termination firing of the apply actor removes this token from the output arc of P, removes the token with value c' from the actor's internal arc, and places a token of color c on the output arc with a pointer to the result structure.

Figure 16 gives, as an illustration, a recursive data flow procedure which constructs the reverse of a binary tree. To simplify the figure, the abbreviation shown in Figure 15a is used to represent the combination of source and append actors (Figure 15b) required to form a new structured value from two values. Note that procedure 'rev-1' is made a component of its own argument structure so that no nonlocal references are required.

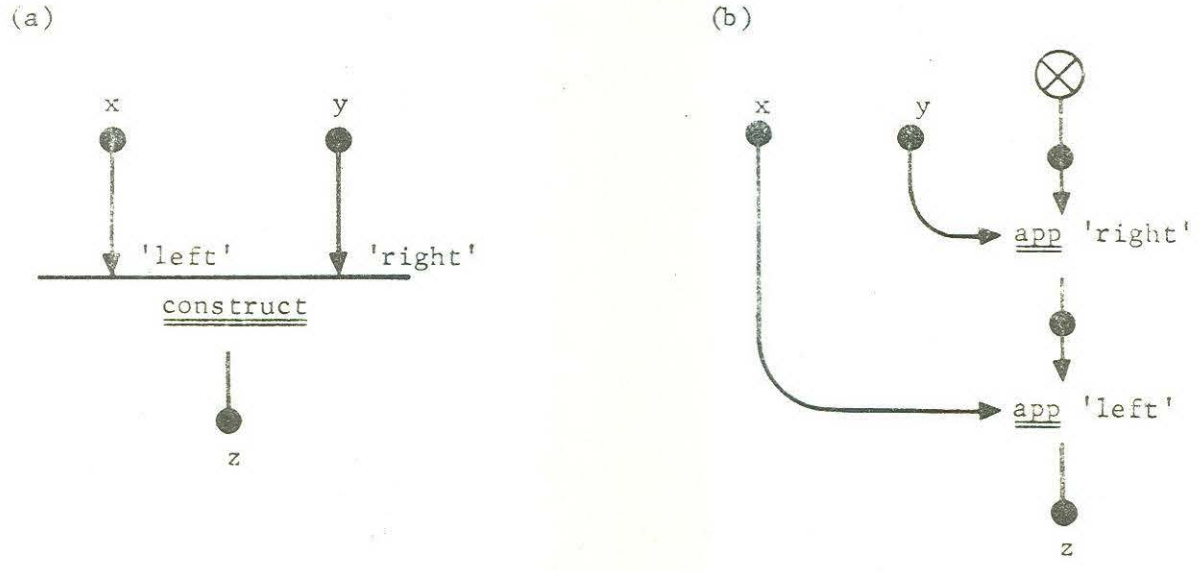


Figure 15. Simplified notation for building data structures.

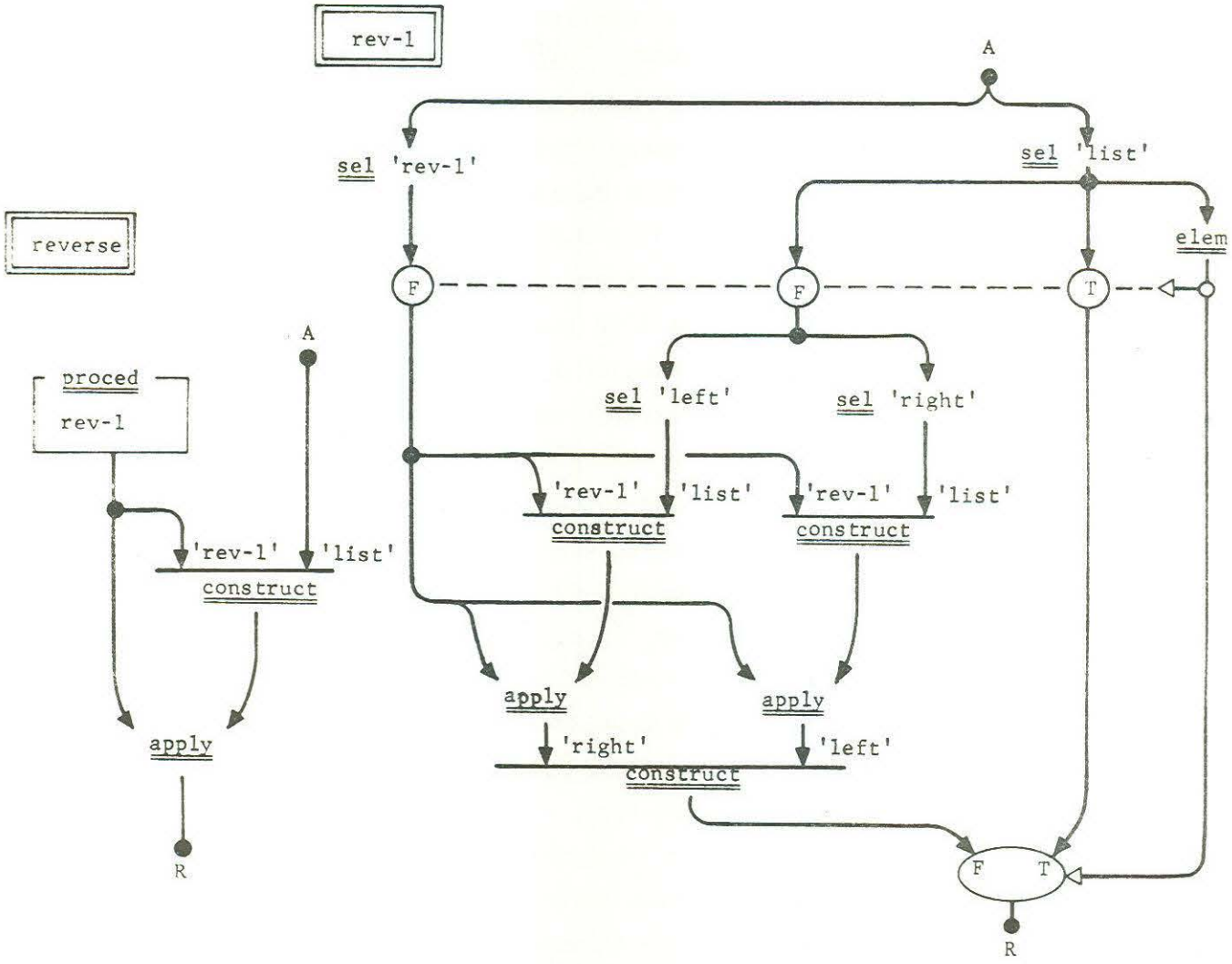


Figure 16. A recursive data flow procedure.

Discussion

Several significant aspects of this data flow representation should be mentioned. The language presented here is sufficient to effectively represent programs in a block structured language. [Amerasinghe] has shown how programs in a simplified block structured language (which allows procedure assignments and procedure returning procedures) can be translated into a base language having conventional instruction sequencing, but without the side effects or nonlocal references permitted by block structure scope rules. We believe it will be possible to construct similar translation procedures using the data flow representation as the target language, although some restrictions on procedure variables may be required.

The data flow language is also consistent with the concepts of modular programming presented in [Dennis 1973]. This is achieved by eliminating side effects: Each procedure application produces an explicit result structure that is functionally related to the argument structure presented to the procedure. Also a sufficiently general concept of data structure is provided by the data flow language so modules represented in the language do not require access to more primitive mechanisms whose use would conflict with program modularity.

An unusual feature of the data flow language is the absence of any primitives for changing the form or content of a data structure. Instead of altering an existing structure, new structured values are assembled using the append primitive from old and newly constructed components. The language is essentially an extension of pure Lisp [McCarthy], in which the restriction to binary trees is removed, computed selectors are permitted, and a specific means for indicating iteration is provided. It is also similar to PAL [Evans].

Several valuable results follow: First, it was easy to specify the data flow language in such a way that functionality of data flow procedures may be guaranteed. Second, it was easy to guarantee that cyclic data structures will not be formed. We believe the possibility of cyclic structures would make it harder to guarantee functionality, and would make the design of efficient memory management schemes a tough problem. Yet cyclic data structures seem to have no sufficiently redeeming advantages to the programmer; where cyclic structures occur, they seem to be used to realize a more suitable class of basic data structures, which replaces the primitive class directly provided by the language being used. Third, in communication between program modules, it becomes unnecessary to distinguish between values which may be modified and values that may only be read -- all values are read-only.

Extensions

Several linguistic issues remain that are not satisfactorily met by the data flow language specified here. We aim to find resolutions of these questions as part of our base language development effort.

1. Parallel for -- The largest source of potential parallel actions in computation arises from program phrases that can be expressed as

for all x in A do S

where A is a set of objects and S is a statement or procedure having x as a parameter. Assuming that the executions of S for the elements of A do not interact, they may all be carried out concurrently without loss of functionality of the procedure in which the phrase resides. At present we do not know how the semantics of the parallel for clause should be provided for in the data flow language.

2. Determinate procedures with memory -- It often happens that a procedure or program module must retain certain information from one activation to the next. In many cases these procedures are functional in the sense that the sequence of argument structures presented, starting with the procedure in a standard initial state, yields a unique sequence of result structures. We say that such a procedure is determinate. Although we believe the key to an appropriate extension of the data flow language lies in the closure construct [Landin], we have not yet developed the details.

3. Function clusters -- We wish our base language to meet the representational requirements of structured programming [Dijkstra]. To a large degree the language specified here meets these requirements: the basic program structures are conditionals, iterations and recursions, and goto's do not exist. Furthermore, no side effects may occur in procedure application.

However, we feel that a language that supports structured programming must deal with the issue of defining data types and encapsulating their implementations within program modules. The notion of a "function cluster" has been proposed by [Liskov and Zilles] as an appropriate linguistic construct for defining data types in structured programming. Function clusters are related to the class construct of Simula 67 [Dahl and Hoare]. We wish to extend the data flow language to provide appropriate means for supporting function clusters.

4. Nondeterminate computation -- There are important applications that require nondeterminate programs. An airline seat reservation system, for example, must give the last seat on a plane to only one agent, even if two agents request it at nearly the same time. We believe nondeterminate programs

can be realized by having just one procedure which is able to update a structure which is a component of its internal state. Yet we want to be able to guarantee a user of our language that his program is determinate if he desires such a guarantee.

Acknowledgement

The development of this version of a data flow language has been strongly affected by discussions with Nimal Amerasinghe, Austin Henderson, and Jim Rumbaugh of Project MAC, and A. Bährs of the Computing Center, Novosibirsk. In particular, the colored token idea was suggested by Nimal Amerasinghe, but had been used earlier by Jim Rumbaugh. The idea of the append operation is old, but its use in our data flow language arose from discussion with A. Bährs, Austin Henderson and Jim Rumbaugh.

References

Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, Calif., December 1968, 130 pp.

Amerasinghe, S. N. The Handling of Procedure Variables in a Base Language. S. M. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., September 1972, 179 pp.

Ashcroft, E., and Z. Manna. The translation of 'goto' programs to 'while' programs. Information Processing 71, North-Holland Publishing Co., 1972, 250-255.

Bährs, A. Operation patterns (an extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972.

Dahl, O. J., and C. A. R. Hoare. Hierarchical program structures. Structured Programming, Academic Press 1972, 175-220.

Dennis, J. B. [1969]. Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., 1969, 484-492.

Dennis, J. B. [1971]. On the design and specification of a common base language. Proceedings of the Symposium on Computers and Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, N. Y., 1971, 47-74.

Dennis, J. B. [1973]. Modularity. Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, 81, Springer-Verlag 1973, 128-182.

Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. [To be published].

Dijkstra, E. W. Notes on structured programming. Structured Programming, Academic Press 1972, 1-82.

Evans, A., Jr. PAL -- a language designed for teaching programming linguistics. Proceedings of the 23rd ACM National Conference, 1968, 395-403.

Fosseen, J. B. Representation of Algorithms by Maximally Parallel Schemata. S. M. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., June 1972, 100 pp.

Kahn, G. A Preliminary Theory for Parallel Programs. Internal Memo, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, Yvelines, France, 1973, 20 pp.

Karp, R. M., and R. E. Miller. Properties of a model for parallel conventions: determinacy, termination, queueing. SIAM J. of Applied Math., 14, 6 (November 1966), 1390-1411.

Kosinski, P. A Data Flow Programming Language. Report RC 4264, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 140 pp.

Landin, P. J. The mechanical evaluation of expressions. The Computer Journal, 6, 4 (January 1964), 308-320.

Lohr, K.-P. Buffered Communication in Abstract Systems. Bericht Nr. 73-03. Fachbereich 20 -- Kybernetik, Technische Universitat Berlin, February 1973, 31 pp.

Liskov, B. H., and S. N. Zilles. Programming with abstract data types. SIGPLAN Notices, 9, 4 (April 1974), 50-59.

Luckham, D. C., D. M. R. Park, and M. S. Paterson. On formalised computer programs. J. of Computer and System Sciences, 4, 3 (June 1970), 220-249.

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

Paterson, M. S. Equivalence Problems in a Model of Computation. Ph.D Thesis, Trinity College, University of Cambridge, August 1967, 154 pp.

Patil, S. S. Closure properties of interconnections of determinate systems. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, 107-116.

Rodriguez, J. E. A Graph Model for Parallel Computation. Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969, 120 pp.