International Journal of Computer and Information Sciences, Vol. 8, No. 5, 1979

A Network of Microprocessors to Execute Reduction Languages, Part I

Gyula A. Magó[1]

Received June 1978; revised March 1979

This paper describes the architecture of a cellular processor capable of directly and efficiently executing reduction languages as defined by Backus. The processor consists of two interconnected networks of microprocessors, one of which is a linear array of identical cells, and the other a tree-structured network of identical cells. Both kinds of cells have modest processing and storage requirements. The processor directly interprets a high-level language, and its efficient operation is not restricted to any special class of problems. Memory space permitting, the processor accommodates the unbounded parallelism allowed by reduction languages in any single user program; it is also able to execute many user programs simultaneously.

KEY WORDS: Reduction languages; functional programming languages; network of microprocessors; cellular computer; parallel processing; high-level language-processor architecture; language-directed computer architecture.

1. INTRODUCTION

Though microelectronics is spectacularly successful in small-scale and simple applications, so far it has not had much impact on the architecture of general-purpose computers. This technology can be used to construct systems with large amounts of main memory and distributed processing capabilities, both of which result in an increase in performance of computer systems, and such approaches are being pursued. However, with regard to building increased amounts of logic into a CPU to obtain a proportionate increase in

[1] Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina.

0091-7036/79/1000-0349$03.00/0 © 1979 Plenum Publishing Corporation

828/8/5-1

performance, all known methods, such as building special functions into the hardware, pipelining, or multiprocessing, yield rapidly diminishing returns.

It has been recognized by many that the most promising avenue toward achieving high performance is exploiting the parallelism inherent in user programs. Many approaches have been proposed, the best known of which are associative and array processors. They have been discussed extensively in the literature,[17] and a few machines have actually been built along these lines. On the basis of the experience accumulated with the use of such machines, there is now general agreement that they achieve high performance only in a narrow range of applications, usually at the expense of programmability.

For a long time Dennis[6] has argued that increasing performance by exploiting parallelism must go hand in hand with making the programming of such machines easier. To achieve this goal, his group at MIT developed a programming language, called a *data flow language*,[7] which is intended to express parallelism in a convenient manner. Several attempts have been made to construct machines that efficiently implement this language or similar languages.[1,4,8]

Although the aims of the work reported in this paper are similar to those of the data flow approach, our starting point here is the class of reduction languages. Reduction languages, described recently by Backus,[2,3] are a class of high-level programming languages with several unique and attractive properties. Among currently used programming languages, they can be likened to APL or LISP, but the property of greatest interest to us is that they are capable of expressing parallelism in a natural fashion, not requiring explicit instructions from the programmer to initiate and terminate execution paths.

This paper outlines the architecture of a processor capable of directly executing reduction languages. After an overview of reduction languages in Sec. 2, Sec. 3 and 4 provide a description of the processor, and Sec. 5 offers a brief evaluation of it. (In Sec. 3 and 4, the behavior of a two-dimensional arrangement of processing elements is described along a third dimension, time. Because of the interdependencies among different parts of the description, frequent cross-references were unavoidable, and reading certain sections out of order may be advisable.)

2. REDUCTION LANGUAGES

In this section we give a brief and very informal introduction to reduction languages, to make the paper self-contained. However, one cannot fully appreciate these languages without reading Backus.[3] We use the terminology

and notation of that paper, and deviate from it only when it is absolutely necessary.

Any program written in a reduction language contains a few syntactic markers, and so-called atomic symbols. The latter may serve as data items, primitive operators, or names of defined functions. Only two kinds of composite expressions are allowed. A *sequence* of length n, $n \geqslant 0$, is denoted by $(a1, a2, \ldots, an)$ if $n \geqslant 1$ and by $\varnothing$ otherwise, where ai (called the ith element of the sequence) is an arbitrary well-formed expression. An *application* is denoted by $\langle a, b \rangle$, where a (called the operator) and b (called the operand) are again well-formed expressions.

Of these two forms of composite expressions only applications specify computations. Since the program text at any time may contain many applications, possibly nested, sequencing among them is specified (at least partially) by requiring that only innermost applications can be executed. There is no sequencing requirement among innermost applications—they can be executed in any order. The process of executing an innermost application is called a *reduction*, and so we often refer to an innermost application as a *reducible application* (RA).

A reduction results in replacing an innermost application with the result expression, which may, in turn, contain further applications. The reduction rules relevant to innermost applications can be summarized as follows. If the operator is an atomic symbol, it might be a *primitive operator* (in which case its effect is specified by the language definition), or it might be the name of a *defined function*, i.e., of some well-formed expression containing no applications (in which case the atomic symbol is replaced by that expression). If the operator is a sequence, it is interpreted as a *composite operator*, composed of the elements of the sequence (Backus describes two possible alternatives: regular and meta composition). The computation comes to a halt when there are no more reducible applications left, and the program text so produced is the result of the computation. If the result of a reduction is undefined, the symbol $\perp$ is used to denote it, which is neither a syntactic marker nor an atomic symbol, but a special expression.

The following example should illustrate most of these concepts. Assume that IP (inner product) is a defined operator, representing IP = (+, (AA, *), TR), whereas AA (apply to all), TR (transpose), + (addition), and * (multiplication) are primitive operators of a reduction language. Suppose the initial program text is

$$\langle \text{IP}, ((1, 2, 3, 4), \quad (11, 12. 13, 14)) \rangle$$

First IP is replaced by its definition, resulting in

$$\langle (+, (\text{AA}, *), \text{TR}), \quad ((1, 2, 3, 4), \quad (11, 12, 13, 14)) \rangle$$

Since the operator now is a composite one, and the interpreter can recognize that it is a regular composition of three expressions, after a few reductions we get the following program text:

$$\langle +, \langle (AA, *), \langle TR, \quad ((1, 2, 3, 4), \quad (11, 12, 13, 14))\rangle\rangle\rangle$$

Now TR is the operator of the only reducible application, and applying TR to the two sequences of its operand leads to

$$\langle +, \langle (AA, *), \quad ((1, 11), \quad (2, 12), \quad (3, 13), \quad (4, 14))\rangle\rangle$$

The only reducible application has a composite operator again, but this time it is a meta composition, resulting in

$$\langle +, (\langle *, (1, 11)\rangle, \quad \langle *, (2, 12)\rangle, \quad \langle *, (3, 13)\rangle, \quad \langle *, (4, 14)\rangle)\rangle$$

Now we have four reducible applications, and they can be reduced in any order, but the addition operator cannot be applied until all multiplications are complete, so at some point we must have the program text

$$\langle +, \quad (11, 24, 39, 56)\rangle$$

which finally reduces to the number 130.

3. COMPUTATIONAL REQUIREMENTS OF REDUCTIONS

Before we describe how the processor operates, we should explain what it is supposed to do. That, however, can be done only after describing how the syntactic and semantic aspects of reduction languages are represented in the processor, because this representation determines, to a large extent, the capabilities the processor must have to act as an interpreter for reduction languages.

3.1. Representation of Syntactic Aspects

Expressions are represented as linear strings of symbols, derived from the representation described in Sec. 2 as follows: since any well-formed expression of a reduction language is obtained by nesting sequences and applications in each other, we can associate a *nesting level* with each symbol of an expression; we store the nesting level with every symbol, and then eliminate all closing brackets—) and $\rangle$—from the source text, for they have become superfluous. This representation corresponds to the symbol sequence

obtained by a preorder traversal of the natural tree representation of the original source text, in which the root of each nontrivial subtree is labeled with (or ⟨.

The internal representation of the processor is finally obtained by placing pairs of symbols, each consisting of a program text symbol and its nesting level, into a linear array of identical hardware cells, one pair per cell, preserving the order of the program text symbols from left to right. As a result of placing at most one program text symbol into a cell, the need for explicit separators between symbols vanishes, because now the cell boundaries perform this function. We always assume that there is a sufficient number of cells available to hold our symbols; if there are more than the required number of cells, then some of them will be left empty. From the point of view of the representation, the number and location of these empty cells relative to the symbols of the program text are of no consequence, and the result of a reduction is not influenced by the positions of the empty cells.

As an example, consider one of the intermediate expressions that appeared in the example given in Sect. 2:

$$\langle +, \langle (AA, *), \langle TR, ((1, 2, 3, 4), (11, 12, 13, 14))\rangle\rangle\rangle$$

The tree representation of this expression is shown in Fig. 1, which also shows the (nesting) level number of every symbol. Fig. 2 shows the internal representation of the same expression.

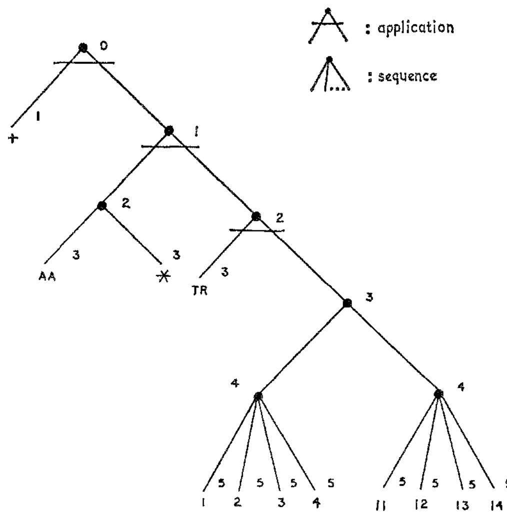

Fig. 1. An expression in tree representation.

	<	+	<	(	AA		*	<	TR	(		(	1		2		3	4	(		11	12			13	14	
	0	1	1	2	3		3	2	3	3		4	5		5		5	5	4		5	5			5	5	

Fig. 2. An expression in internal representation.

3.2. Representation of the Semantic Aspects of Reducible Applications

The semantics of a reduction language is determined by a set of rules prescribing how all the possible reductions should be performed. These include rules specifying the effect of each primitive operator, and rules to decompose composite (regular and meta) operators.

By examining what forms these rules take when using our chosen internal representation for expressions, we can see what kinds of computations the processor will have to be able to perform.

3.2.1. Primitive Operators

A reduction language may have a large number of primitives, but the computational requirements of all of them can be classified into three easily distinguishable categories. They will be explained with the help of the following three operators.[3]

3.2.1.1. AL (Apply to Left Element). AL is a meta operator, and its effect is the following:

$$\langle(\mathrm{AL}, f), (y1, y2, \ldots, yn)\rangle \quad \Rightarrow \quad (\langle f, y1\rangle, y2, \ldots, yn),$$

and if the operand is not in the required form, the result is $\perp$. Here the italicized symbols are metalinguistic variables, and they stand for arbitrary constant expressions, i.e., expressions containing no applications. The arrow $\Rightarrow$ is used to denote that reducing the expression on the left yields the expression on the right.

Fig. 3 shows the effect of (AL, (HEAD, TAIL)) on a particular three-element sequence using tree representation. As the definition prescribes it, (HEAD, TAIL) is applied to the leftmost element of the sequence, and the rest of the sequence is left unchanged.

Fig. 4 depicts the same reduction using the internal representation. Examination of the cells reveals that one of the following things has happened to each one of the symbol–nesting level pairs: the symbol only is rewritten (as in cell 2), the level number only is rewritten (as in cell 16), both symbol and level number are rewritten (as in cell 5), or no change (as in cells 1 and 14).

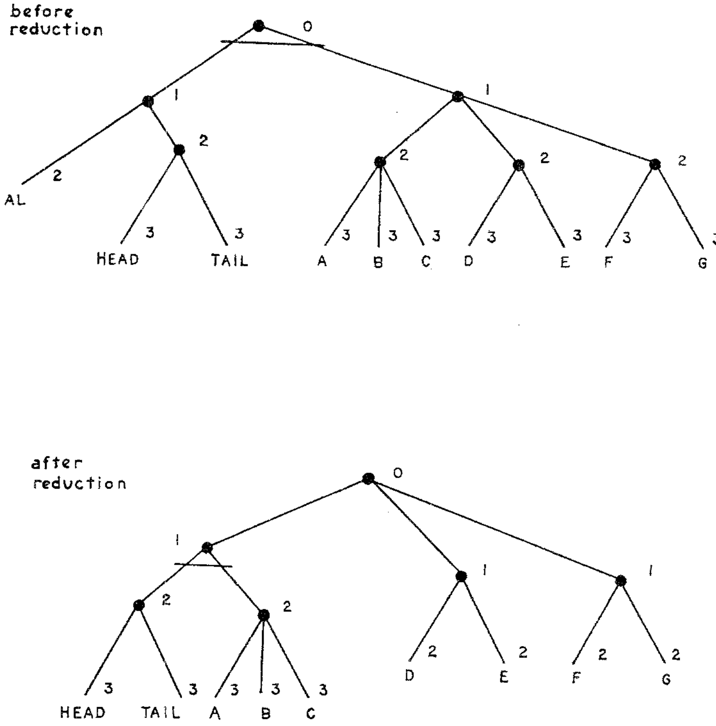

Fig. 3. A reduction involving AL. The program text is shown in tree representation first before, and then after the reduction.

before reduction

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		<	(		AL	(	HEAD	TAIL		(	(	A	B	C	(	D	E	(	F	G
		0	1		2	2	3	3		1	2	3	3	3	2	3	3	2	3	3

after reduction

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		(	<			(	HEAD	TAIL			(	A	B	C	(	D	E	(	F	G
		0	1			2	3	3			2	3	3	3	1	2	2	1	2	2

Fig. 4. A reduction involving AL. The program text is shown in internal representation first before, and then after the reduction.

The processing requirements of this primitive are called *Type A* requirements, and they are characterized by the following: (1) the result expression can be produced in the cells that held the original reducible application, i.e., *there is no need for additional cells;* (2) the processing activities that are to be applied to any symbol of the original RA are known before execution begins, and if the prescription for these activities is placed into the cells before execution begins, they can be performed independently of each other, in any order (possibly simultaneously), and consequently *there is no need for any communication between these cells during execution.* (This conclusion is independent of the expressions that replace the metalinguistic variables of the definition, because the expressions f and $y1$ are left intact, and the only change to expressions $y2$ through yn is their being moved up the tree by one level.)

3.2.1.2. AND. AND is a regular operator, and its effect is defined as follows:

$$\langle \text{AND}, (x, y)\rangle \quad => \quad z$$

where x and y are expected to be atomic symbols, either T (true) or F (false). If both x and y have the value T, then the result z is T; if both of them have Boolean values, but at least one of them is F, then the result is F, and in every other case the result is the undefined expression ($\perp$).

Fig. 5 shows an example of the application of AND to (T, F) in internal representation. (Because of the simplicity of this example, we skip the tree representation.) Although the processing requirements of this primitive include some Type A processing (e.g., the symbol AND and its level number can be erased irrespective of the operand expression), there are some new elements also. They are included in the following, which we call *Type B* requirements: (1) the result expression can be produced in the cells that

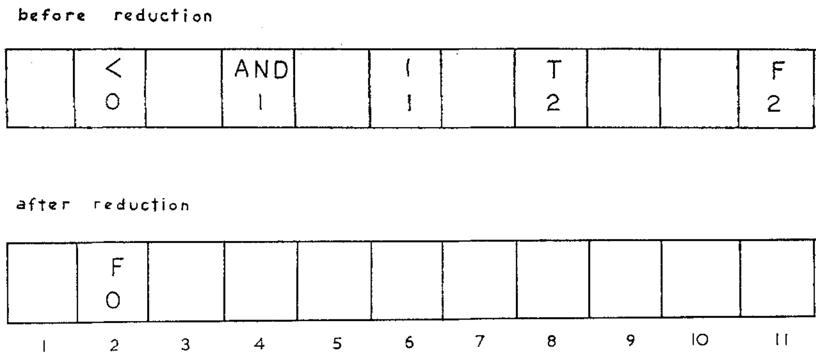

Fig. 5. A reduction involving AND in internal representation.

held the original reducible application, i.e., *there is no need for additional cells;* (2) at least some of the processing activities are data dependent, and as a consequence, *there is a need for communication at least among some of the cells during execution*, and also there are certain timing constraints. (In our example, the two components of the operand determine whether to produce F or T as the result, and this result cannot be produced in cell 2 before bringing together the contents of cells 8 and 11.)

3.2.1.3. AA (Apply to All Elements). AA is a meta operator, and its effect is the following:

$$\langle(\mathrm{AA}, f), (y1, y2, \ldots, yn)\rangle \quad \Rightarrow$$
$$(\langle f, y1\rangle, \langle f, y2\rangle, \ldots, \langle f, yn\rangle)$$

and if the operand is not in the required form, the result is $\perp$. Fig. 6 shows the effect of (AA, *) on a particular four-element sequence using tree

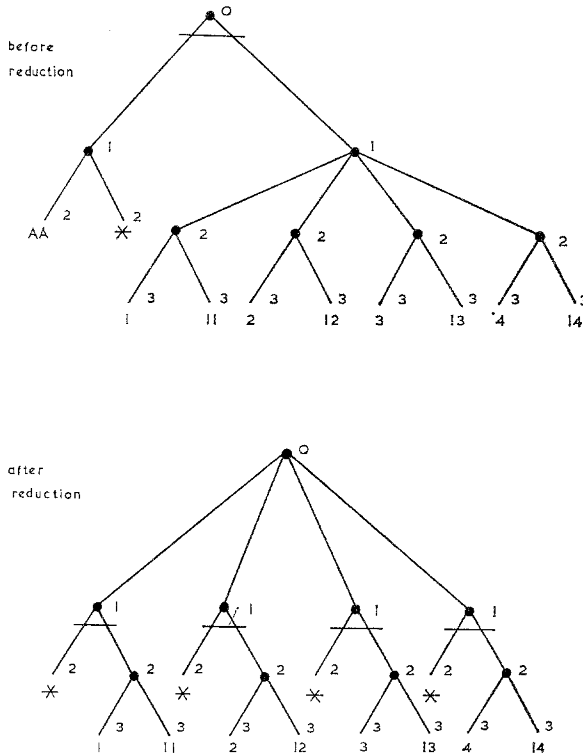

Fig. 6. A reduction involving AA in tree representation.

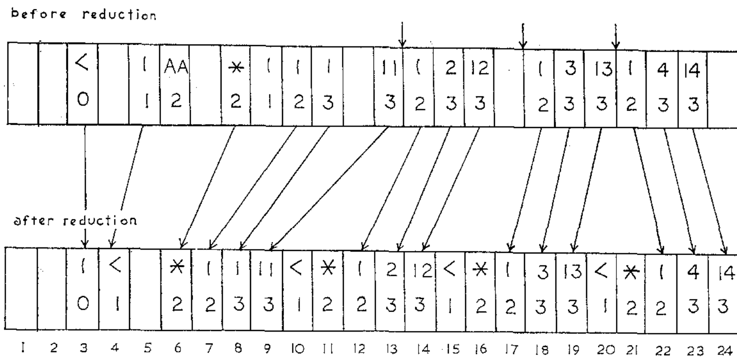

Fig. 7. A reduction involving AA in internal representation.

representation. The same reduction is depicted in Fig. 7 using internal representation.

The processing requirements of this primitive are called *Type C* requirements, and they are characterized by the following property: the result expression cannot be produced in the cells that held the original reducible application, hence *there is a need for additional cells to hold the result.*

Since the number of insertions, and the length of the expressions to be inserted are not generally known before execution begins, a complex rearrangement of the whole RA may be necessary, the details of which must be worked out at runtime. For example, with AA the number of insertions is $n - 1$, where n is the length of the operand. In Fig. 7 there are three insertions, each indicated by an arrow, and each insertion contains the symbols $\langle$ and $*$.

It should be apparent that Type A processing requirements are a special case of Type B requirements, which, in turn, are a special case of Type C processing requirements.

3.2.2. Defined Operators

Whenever an atomic symbol for which a definition exists gets into the operator position of a reducible application, it must be replaced by its definition.[3] Since a nontrivial definition contains more than one symbol, replacing a defined symbol by its definition has Type C processing requirements. It should be noted that definitions must exist before execution begins and cannot be created at runtime.

3.2.3. Composite Operators

When the operator of a reducible application is composite (i.e., a sequence), the way in which the evaluation proceeds depends on whether the

first element of the sequence is regular or meta. If the first element of the sequence is an atomic symbol, then whether it is regular or meta is part of its definition. If the first element of the sequence is a sequence, then it is meta if its first element is ϕ, and otherwise it is regular.

3.2.3.1. Regular Composition. If the first element of a composite operator is regular, we decompose it with the help of the following rule, called *regular composition*[3]:

$$\langle(c1, c2,..., cn), d\rangle \quad => \quad \langle c1, \langle(c2, c3,..., cn), d\rangle\rangle$$

This reduction rule reveals that the processing requirements of regular composition can be characterized as Type C, since we have to create two new symbols between $c1$ and $c2$.

3.2.3.2. Meta Composition. If the first element of a composite operator is meta, it is decomposed with the help of the following rule, called *meta composition*[3]:

$$\langle(c1, c2,..., cn), d\rangle \quad => \quad \langle c1, ((c1, c2,..., cn), d)\rangle$$

Since $c1$ (whatever expression it is) must be duplicated, the processing requirements are to be classified as Type C. (In fact, if $c1$ happens to be a primitive meta operator, there is no need to go through this step: this is demonstrated in Fig. 3 and 6 with the meta operators AL and AA.)

3.3. Order of Evaluation

The definition of reduction languages allows the execution of reducible applications to take place in any order, owing to the so-called Church-Rosser property of these languages.[11] Since reducible applications are disjoint in our chosen internal representation, this representation allows them to be reduced simultaneously. This is because the outcome of the reduction is determined solely by the operator and operand expressions, and nothing from the rest of the program text can influence it.

3.4. Locating Reducible Applications

Before processing of reducible applications can take place, they must be located in the program text. This process is somewhat complicated by the fact that there is no bound on either the number of reducible applications that may exist simultaneously in the program text, or the length of a reducible application.

An application symbol whose level number is i is the left end of a reducible application if

1. it is the rightmost application symbol, or
2. the next application symbol to its right has a level number less than or equal to i, or
3. there exists a symbol with level number less than or equal to i between the application symbol and the next application symbol to its right.

If an application symbol with level number i is known to be the left end of a reducible application, then the entire application consists of the application symbol itself and the sequence of contiguous symbols to its right whose level numbers are greater than i.

4. DESCRIPTION OF PROCESSOR

4.1. Interconnection Pattern of Cells

Fig. 8 shows that the processor is a cellular network containing two kinds of cells interconnected in a highly regular manner. Cells of one kind form a linear array (they are indicated by rectangles in the diagram), and they normally hold the program text as described in Sec. 3. Cells of another kind form a full binary tree (they are indicated by triangles in the diagram), and they perform processing functions, act as a routing network, etc. The linear array of cells is referred to as L, and the tree network as T. Throughout this paper, the root cell of T is assumed to act as the I/O port of the processor (see Sec. 4.7).

Since L holds the program text one symbol per cell, a network of practical size comprises a large number of cells. Because of this, we note here an important and very attractive property such networks have: the total number of cells in the network is a linear function of the length of L. More precisely, if n is the height of the tree of cells, then the length of L is $2^{**}n$, and the number of cells in T is $2^{**}n - 1$, so the total amount of hardware is almost exactly $2^{**}n(l + t)$, where l and t represent the amounts of hardware built into a single cell of L and T, respectively. (Here we ignore problems related to layout and interconnections; these issues will be discussed elsewhere.)

4.2. The Partitioned Processor

In Sec. 3.3 we noted that our internal representation allows all RA's to be reduced simultaneously. To guarantee that the execution of each RA

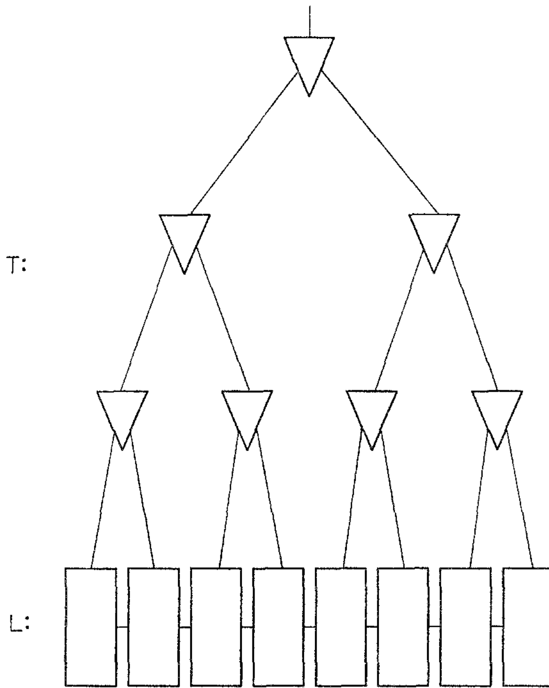

Fig. 8. The pattern of interconnecting cells in the processor.

can proceed independently, different RA's are processed in disjoint portions of T.

We describe a way of partitioning the cells of T so that each RA "sees" a binary tree of processing elements above it, which then processes the RA as if the RA were alone in the processor.

A single cell of T may participate in the (concurrent) processing of more than one RA. But we will show that at any time, any given cell of T must be involved in the processing of at most four RA's, and consequently an arbitrary cell of T must contain at most four processing elements, each belonging to a different binary tree.

Thus each cell of T contains four separate processing elements. Assigning these different processing elements of a cell of T to different RA's is called the *partitioning* of that cell. The process and the result of partitioning all the cells of T, and constructing a binary tree of processing elements for each RA in the program text, is called a *partitioning of the processor.* (Partitioning of the processor occurs repeatedly during the execution of a program, in accordance with the changing program text in L.)

In this section we describe, with the help of a symbolic notation, what

the partitioned processor is like. In Sec. 4.6.1 we present some of the details of the process of partitioning.

At the core of the partitioning process is the execution of the algorithm of Sec. 3.4, done simultaneously for all applications. There are two steps in this process:

1. locating left ends of applications (as described in Sec. 3.4) and subsequently dividing the processor into so-called *areas;*
2. locating right ends of innermost applications (as described in Sec. 3.4), and subsequently transforming some of the areas into so-called *active areas.*

First we describe the processor as partitioned into areas. Assume that we start out with a representation of the processor as shown in Fig. 8. We modify it by erasing all connections shown in Fig. 8 (though we keep implicitly the relation represented by the erased connections), and place symbols of the reduction language program into cells of L. In this symbolic notation an *area* is a binary tree whose leaves are in cells of L and whose nodes which are not leaves are in cells of T. Each ⟨ symbol has associated with it a distinct area. Moreover, the leftmost cell of L has a distinct area associated with it, whether or not it contains an ⟨ symbol.

To give a precise definition for areas, the following terminology is used. The *index* of a cell of L is an integer indicating its position in L from left to right. Let $i(1) = 1$, and let $i(2),..., i(q)$ be the indices of all the cells of L (other than the leftmost one) holding the symbol $<$, with $i(m) < i(n)$, whenever $m < n$. (The ⟨ symbols in $i(k - 1)$ and $i(k + 1)$ are sometimes called the *left* and *right neighbors* of that in $i(k)$ for $2 < k < q$, or $1 < k < q$ if $i(1)$ does contain an ⟨ symbol.) We say that a binary tree B is *embedded* in T and L if

1. every leaf of B is contained in a cell of L (at most one such node being in any cell of L),
2. every node of B that is not a leaf is contained in a cell of T (at most one such node being in any cell of T),
3. if $b1$ and $b2$ are nodes of B, and $b1$ is the father of $b2$, then if $t1$ (a cell of T) contains $b1$, and $t2$ (a cell of T or L) contains $b2$, then $t1$ is the father cell of $t2$.

Definition 1. Depending on the value of j, the *jth area* is a binary tree embedded in T and L such that

for $1 < j < q$ (1) the leaves of the tree are in the cells of L indexed from $i(j)$ to $i(j + 1) - 1$,

(2) the top node (root) of the tree is in the lowest cell of T which has both the L cells $i(j)-1$ and $i(j+1)$ as descendants;

for $j=1$ (1) the leaves of the tree are in the cells of L indexed from $i(1)$ to $i(2)-1$,
(2) the top node (root) of the tree is in the root of T;

for $j=q$ (1) the leaves of the tree are in the cells of L with indices greater than or equal to $i(q)$,
(2) the top node (root) of the tree is in the root of T.

The following method is used to construct the jth area $(1<j<q)$:

1. place a single leaf node into every L cell from $i(j)$ up to and including $i(j+1)-1$;
2. find the T cell that is the lowest common ancestor of the L cells $i(j)$ and $i(j+1)-1$, and call it $t1$ (it can be shown that $t1$ is unique);
3. embed a binary tree in T and L consisting of a minimal set of edges so that there is a path from $t1$ to each of the L cells between $i(j)$ and $i(j+1)-1$ inclusive;
4. find the T cell that is the lowest common ancestor of the L cells $i(j)-1$ and $i(j+1)$, and call it $t2$ ($t2$ is unique, and it is an ancestor of $t1$);
5. if $t1 \neq t2$, then add a path originating in $t1$ and terminating in $t2$ to the tree embedded in T and L in step 3.

The resulting tree is the jth area. The construction for $j=1$ and $j=q$ is similar, except that in step 4 $t2$ is defined to be the root cell of T.

With all the areas constructed in our symbolic notation, the following propositions hold.

Proposition 1. Every cell of T other than the root cell is connected to its parent cell either by two branches of two different areas or by a single branch of one area.

The proof of this proposition can be found in Appendix A.

Proposition 2. Each cell of T holds one, two, three, or four nodes, each belonging to a different area.

Proof. Consider a cell t of T. Let t' and t'' be the left and right son cells of t, respectively. Proposition 1 states that there are one or two branches between t' and t, and also between t'' and t. The right branch arriving from the left son t', and the left branch arriving from the right son t'' may belong

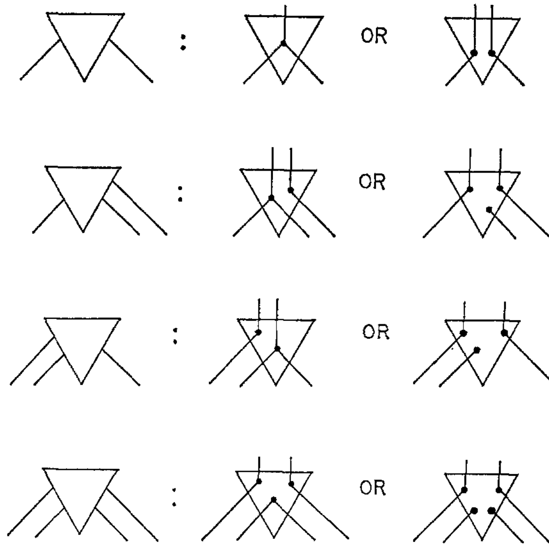

Fig. 9. A cell of T may hold one, two, three, or four nodes, each of a different area.

either to the same area or to two different areas. Fig. 9 shows all the possibilities, completing the proof.

Figure 10 shows a partitioned processor containing six areas. These areas are drawn superimposed on the T and L cells. (The latter can be obtained by starting with a diagram like Fig. 8, and erasing all the connections between cells.) For example, the second area from the left has its leaves in cells 4 through 8 of L. The lowest common ancestor of cells 4 and 8 of L is cell 4 of T and, as Fig. 10 shows, the area contains paths embedded in T and L starting in every L cell between 4 and 8, and ending in cell 4 of T. In addition, since the lowest common ancestor of cells 3 and 9 of L is cell 2 of T, the area also contains the path starting in cell 4 of T and ending in cell 2 of T.

The example of Fig. 10 contains seven of the eight possible partitioning patterns shown in Fig. 9. The elements of the symbolic notation we are using can be interpreted as follows. All branches of areas correspond to communication channels of identical capabilities, capable of carrying information both ways simultaneously. Whenever only one branch is shown between two cells, we may assume that the second channel is idle. Each node of an area corresponds to some fixed amount of processing hardware. Whenever a node of an area has two downward branches, the corresponding node hardware may perform processing that is immediately comprehensible

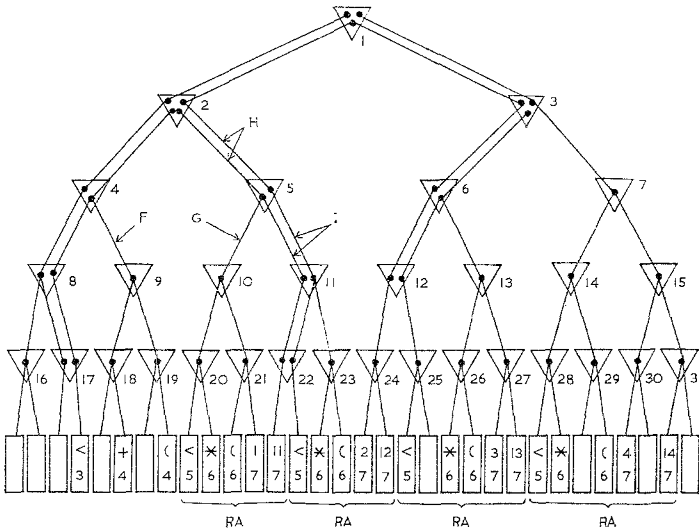

Fig. 10. A processor partitioned into six areas. Cells of T are labeled with integers for identification. The labels attached to branches (F, G, H, and I) are explained in Sec. 4.6.1.

in terms of the reduction language program. (Note that a cell of T may hold at most one node with two downward branches.) For example, in Fig. 10, the node with two downward branches in cell 5 multiplies the symbols 1 and 11 in the program text, 2 and 12 are multiplied in cell 24, 3 and 13 are multiplied in cell 27, and 4 and 14 are multiplied in cell 15. Other functions of such nodes, and the role of nodes with one downward branch are described later. The top of the area serves as its I/O port; the I/O channels with which it connects are not considered here, but are discussed in Sec. 4.7.

Finally, we note that since the root of an area is in a cell of T which has among its descendants the cell of L holding the next $\langle$ symbol on the right (if one exists), all the necessary information can be made available at the root node of each area to determine whether or not the area contains an RA.

Once the processor is divided into areas, the algorithm to locate RA's is executed at the root of each area. Since the leaves of an area holding an RA contain all cells of L up to the next $\langle$ symbol or to the right end of L, some of the rightmost leaves of this area may hold symbols of the reduction language text that are outside the RA. Locating such leaves, and separating them from the area—thereby transforming the area into an *active area*—is the second part of the partitioning process. (The active area is obtained by cutting off certain subtrees of the original area. As a result,

the active area is a binary tree too.) In our example in Fig. 10, there are four RA's, but none of them requires this process, so we postpone showing an example of this until Sec. 4.6.1.

4.3. States, State Changes, and Overall Organization

In a global sense the operation of the processor is determined by the reduction language program placed into L. The operation of each cell is "data-driven," i.e., in response to information received from its neighbors. The activities of cells are coordinated by endowing each cell of the processor with copies of the same finite-state control, which determine how the cell interprets information received from its neighbors. Whenever a cell of T is partitioned, each independent part (there are at most four of them, each corresponding to a distinct node of an area) must have its own finite-state control. On the other hand, a cell of L needs only one such control, since it is never partitioned.

The state of a node of an area changes whenever either its parent or both its children change state. The actual pattern of state changes is the following: the root cell of T initiates the process by the nodes contained in it changing their states; as a result, its son cells in T change state, and these changes, in turn, initiate similar changes on the next lower level of T.

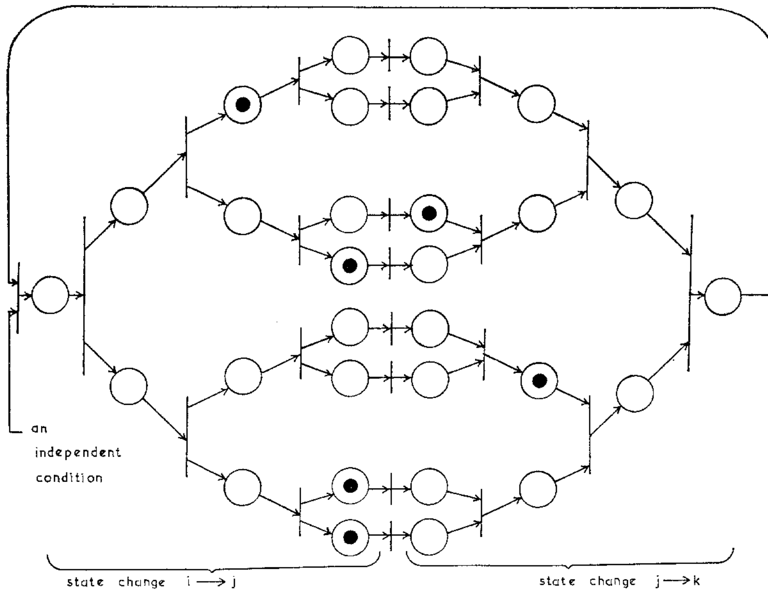

Fig. 11. Petri-net represention of the state changes in a processor, which contains a single area, and eight cells in L.

When this wave of changes reaches L, the state changes in the cells of L initiate changes in the bottom level of T, which, in turn, cause changes in the next higher level of T, and so on.

Figure 11 shows a Petri-net[9,13] representation of the state changes for a processor in which L has eight cells and partitioning produced a single area, hence all the cells go through the same state changes. All conditions (represented by circles) have the following interpretation: "in the given cell state change $p \rightarrow q$ is taking place." The distribution of tokens in the net (showing the holding of certain conditions) illustrates that these state changes can take place at their own pace, but they always get synchronized in the process of approaching the root cell of T.

To simplify our presentation, we assume that the state changes take place simultaneously on any level of T. This allows us to talk about *upward* and *downward cycles*, indicating which way the state changes are propagating. Figure 12 shows a fragment of a processor in the middle of a downward cycle. The cells are partitioned, and four different states—3, 33, 4, and 34—can be found in the diagram. The figure illustrates the following point: since the reason for having the finite-state control in the cells is to coordinate related activities in the processor, all nodes of an area go through the same

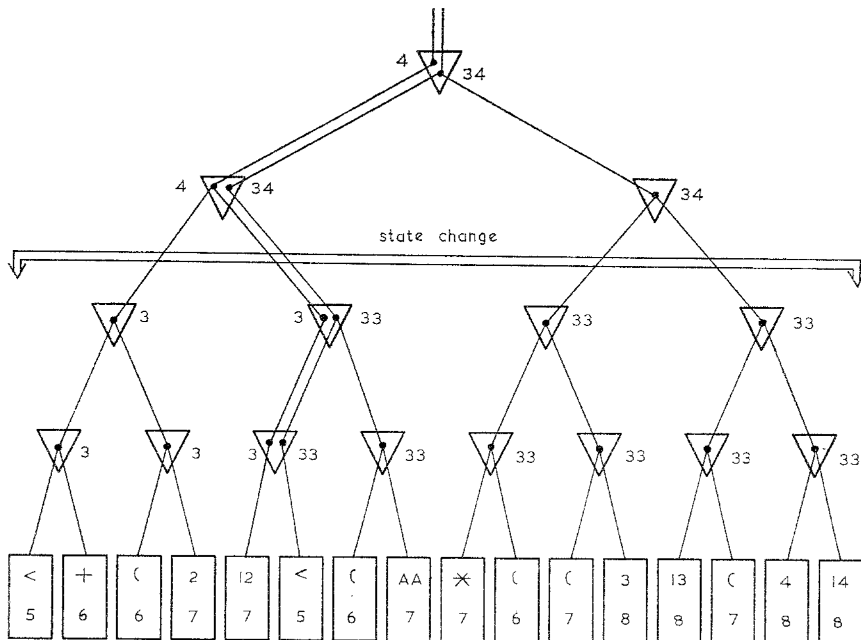

Fig. 12. Snapshot of a fragment of a processor during the downward cycle $k + 3 \rightarrow k + 4$. Labels 3, 4, 33, and 34 indicate states of the nodes of areas.

state changes (e.g., in Fig. 12 all nodes of the active area on the left are going through the change $3 \rightarrow 4$, whereas all nodes of the active area on the right are going through $33 \rightarrow 34$), and, as a result, in general no useful purpose is served by talking about the state of the processor as a whole.

Figure 13 shows the state diagram of the nodes of areas for the processor we are describing. The details of this diagram are explained at length in the remainder of Sec. 4. Here only the following observations are made:

1. Although different cells (or parts thereof) may be in different states at any moment, we can always say (thanks to our simplifying assumption) that all cells (or parts thereof) on the same level of T are in states that are in the same column of the state diagram (for an example, see Fig. 12).

2. We use the expression $k + i$ to denote all states in a column of the state diagram, where i is the smallest label in the column and k may

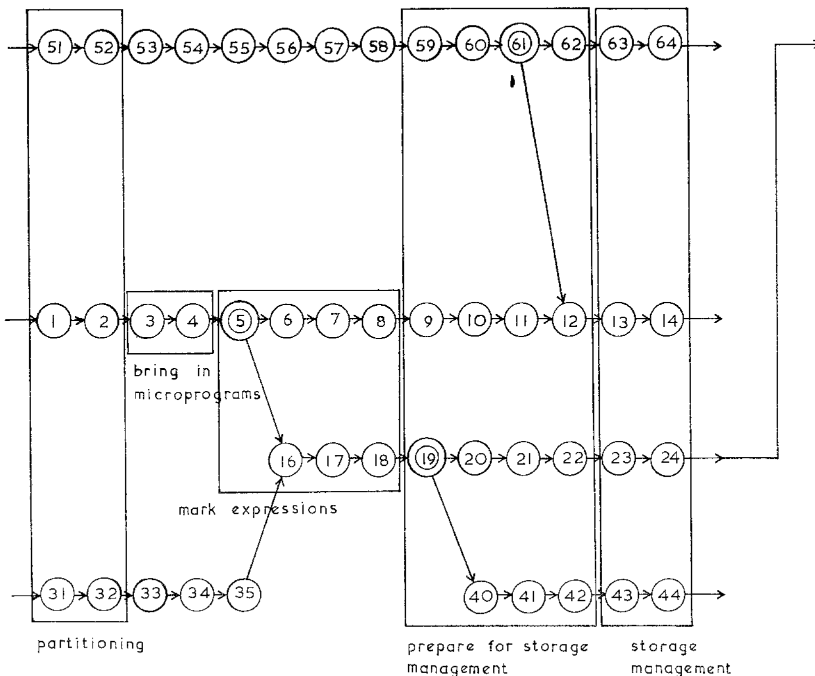

Fig. 13. The state diagram of a node of an area, or that of the contents of a cell of L. (Meaning of certain state sequences: $1 - \cdots - 14$, area contains no RA, or it is executing a Type A or B operation; $1 - \cdots - 5 - 16 - \cdots - 24$, area is executing a Type B or C operation; $1 - \cdots - 5 - 16 - \cdots - 19 - 40 - \cdots - 44$, area is executing a Type C operation, requested storage, and did not receive it; $51 - \cdots - 64$, Type B or C operation, sending messages or moving data.)

assume values of 0, 10, 30, and 50, e.g., $k + 4$ where $k = 0$, 30, 50; $k + 7$ where $k = 0$, 10, 50; or $k + 12$ where $k = 0$, 10, 30, 50.

3. Odd-numbered states are entered in upward cycles, and even-numbered states are entered in downward cycles.

4. The state diagram is cyclic: the successors to states $k + 14$ are states $k + 1$.

5. There are three (specially marked) states in the diagram—states 5, 19, and 61—with more than one successor state; in these states the successors are always chosen deterministically, with the help of conditions that are not visible on the level of the state diagram.

6. In cells of L, the state belongs to the contents of the cell, not to the cell itself—hence the state information moves with the contents of the cell during storage management (see Sec. 4.6.6.2.)

7. Since the state diagram describes the states of the nodes of areas, during state transition $k + 14 \rightarrow k + 1$, when partitioning takes place, hence areas go out of and come into existence, some additional rules are needed: in states $k + 14$ each node of each area changes its state to the undefined state, with the exception of the leftmost cell of L and the cells of L holding an ⟨ symbol, and these cells of L, in the process of repartitioning the processor, determine the states of the newly formed areas (the state transitions to and from the undefined state are not shown in the state diagram of Fig. 13.

8. When the program text is first placed into L, the state of each ⟨ symbol in it is 1.

The state diagram specifies the overall organization of the processor. The organization, hence the state diagram, chosen for description in this paper is just one of many possible alternatives: the main criteria in its selection were that it be easy to describe, yet still able to illustrate well the advantages of a processor of this kind.

Since it is the state of the node of an area which determines what processing activities that node performs, and the states of the nodes of an area are closely coordinated (all nodes of the area go through the same state change in each upward and downward cycle), the processing activities performed by an area in certain states (or in certain groups of states) can be classified as fitting one of several global patterns. We choose to distinguish three such patterns, and call them *modes of operation.*

In both Mode I and Mode II, information is sent along paths between L and the root cell of T, usually inside areas, but possibly also across area boundaries (examples of the latter are partitioning, preparation for storage

management, and detecting the end of storage management). In Mode III, information is sent only along L.

Mode I and Mode II are distinguished because information items moving upward are treated differently. In a Mode I operation, (1) whenever a node of an area (or a cell of T) receives two information items from below, it produces one information item to be sent up to its parent node; (2) the output item is produced by performing some kind of operation on the two input items, such as adding two numbers (see combining messages, Sec. 4.5.2), taking the leftmost three of six arriving items (see partitioning, Sec. 4.6.1), or the considerably more complex operation of preparing the directory (see Sec. 4.6.2); (3) since each subtree of the area (or of T) produces a single value, this value can be stored in the root node of that subtree, and can be used to influence a later phase of processing; (4) if the data paths are wide enough, the node of the area (or the cell of T) is able to receive both its inputs in one step, hence is able to produce its output in one step (we make this simplifying assumption throughout this paper).

In a Mode II operation, (1) whenever a node of an area receives two information items from below, it produces two information items to be sent up to its parent node; (2) the two output items are the same as the two input items, and the order in which they appear on the output may or may not be of consequence; (3) the higher up a node is in the area, the larger the number of information items that pass through it, and as a result, the time required for a Mode II operation is data dependent. Because of this queueing phenomenon, and because the size of information items may also vary considerably, the natural way to control a Mode II operation is with the help of *asynchronous control structures*,[5,12] via ready and acknowledge signals. The Mode II operations are: (1) bringing in microprograms (in state 3, see Sec. 4.6.2), (2) sending messages, data movement, and I/0 (in states 52 through 61, see Secs. 4.6.4, 4.6.5, and 4.7), and (3) preparation for storage management (in states $k + 9$ and $k + 10$, see Sec. 4.6.6.1).

Modes I and II also differ in the ways they treat information items moving downward, but these differences are consequences of the primary distinction between them. In a Mode I operation, (1) the node of the area (or the cell of T) produces both a left and a right output item in response to the one input item, and they may be different, depending on what was left in the node (or cell) in question by the previous upward cycle (an example of this is the process of marking expressions—see Sec. 4.6.3); (2) since during the previous upward cycle the top of the area produced a single item, during the next downward cycle only a single item (not necessarily the same) arrives at each cell of L in the area; (3) the processes of moving information up and down in the area do not overlap in time.

In a Mode II operation, (1) the node of the area produces two output

items in response to the one input item, and they are always identical (the item is being broadcast to each cell of L in the area); (2) every item that passes through the top node of the area is broadcast separately to cells in L, and the latter are free to accept or reject any of the items arriving at them; (3) the processes of moving information up and down in the area overlap in time, hence each branch of the area must be able to carry information items in both directions simultaneously.

In summary, Mode I and Mode II can be compared and contrasted as follows. In a Mode I operation, by propagating information upward in the tree simultaneously from all cells of L, the global situation in L is evaluated, and the partial results of this evaluation are stored into the nodes of the area (or the cells of T). Next, by propagating information downward in the tree and by using the partial results stored in the nodes, each cell of L can be influenced separately and differently. In a Mode II operation, on the other hand, the area functions as a routing, or interconnection, network, and typically delivers information items from L back to L, by passing them through the root node of the area.

Mode III is characterized by the fact that only cells of L participate in the processing, and adjacent cells of L communicate with each other directly. The only Mode III operation is storage management (see Sec. 4.6.6.2).

4.4. Outline of Cell Organization

In this section we outline the processing and storage capabilities that a typical cell of L and T must have. When describing certain components of these cells, we often refer to details that are explained only in subsequent parts of Sec. 4. Thus this section can be fully understood only after reading the rest of Sec. 4.

4.4.1. Cell of L

Figure 14 shows a cell of L. The names of registers appearing in the figure are used in the rest of Sec. 4 to explain how the processor operates.

The component labeled *state* has the ability to store the current state, and compute the next state corresponding to the state diagram of Fig. 13; this state information belongs to the contents of the cell, not to the cell itself.

CPU has the ability to execute segments of microprograms, and perform processing related to storage management (see Sec. 4.6.6.2), which is not explicitly specified in microprograms.

Microprogram store is capable of storing a certain number of micro-instructions. This is necessary because certain microinstructions cannot be

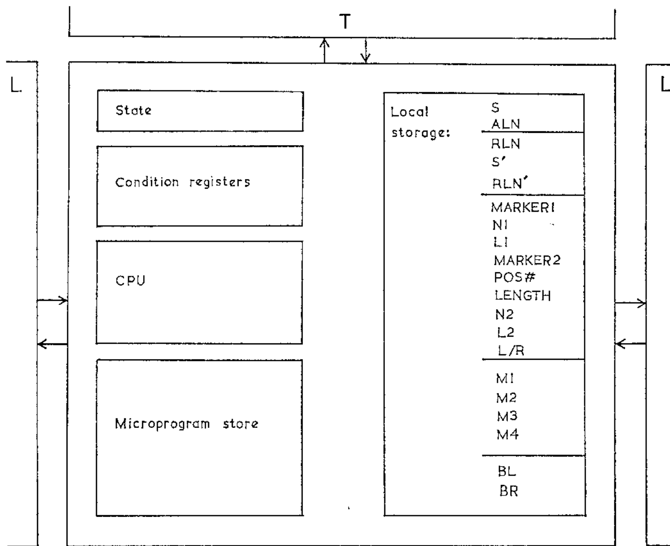

Fig. 14. Schematic representation of a cell of *L*.

executed immediately on receipt, since some of their operands are not yet available.

Condition registers store status information concerning the contents of the cell, e.g., whether the cell is full or empty, and whether the contents of the cell are to move during data movement.

Local storage contains the following registers: *S* holds a single symbol of the reduction language text. *S′* holds another symbol of the reduction language, with which *S* is to be rewritten at the end of processing the RA.

ALN holds the absolute level number of the symbol of the program text; this is obtained by considering the contents of *L* as a single expression of the reduction language and assigning to ALN the nesting level of the symbol in question. RLN is the relative level number of a symbol in an RA. This is obtained by assigning to RLN the nesting level of the symbol with respect to the RA. RLN′ is the value with which RLN is to be rewritten at the end of processing the RA.

MARKER1 and MARKER2 are set by the microprogram and used to mark all symbols of an expression. Whenever a microinstruction "MARK WITH *x*" is executed in a cell of *L*, MARKER1 receives the value "*x*," and if RLN has a certain value, MARKER2 receives "*x*," too. Symbols of the marked expression are indexed, beginning with one, and these index values are placed in N1. The largest index value, which is the total number of symbols in the expression, is placed in L1. When a symbol occurs in a

marked expression, the value of POS# (mnemonic for position number) for each symbol is set as follows: the marked expression is considered a sequence, and all symbols of the expression that is the *i*th element of this sequence receive the value *i* in their POS# register. The largest value of POS#, which is the length of this sequence, is placed in register LENGTH of each symbol. Also, each expression that is an element of the sequence is indexed separately, the index values are placed in N2, and the total number of symbols in the element expression are placed in L2. (N2 and L2 play the same role for the element expressions as N1 and L1 do for the whole marked expression.) The L/R register holds the value "left" (or "right") if the symbol contained in *S* is the leftmost (or rightmost) symbol of one of the elements of a marked sequence.

M1, M2, M3, and M4 are called message registers: SEND statements generate messages that may have one, two, three, or four components, and on arrival at the cell of L, they are placed in M1, M2, M3, and M4, respectively.

BL and BR contain nonnegative integers, and are used during storage management: the cell of *L* in question is entered on the left by the contents of BL cells of *L*, and on the right by the contents of BR cells of *L*.

Of these registers, BL, BR, and those controlling the state of the cell are used in every cell of *L*; *S* and ALN are used in every occupied cell of *L*; and all the others are used only by occupied cells internal to an RA, or by cells reserved during storage management.

The *S* register must be large enough to hold any symbol that might occur in the source text, including numbers. The ALN and RLN registers must be able to hold any nesting level value that might arise. (If *L* has $2^{**}n$ cells, the maximum nesting level of any expression in it is $2^{**}n - 1$, hence n-bit registers will suffice.) Finally, the number of cells in *L* is a bound on the values that BL, BR, N1, L1, POS#, LENGTH, N2, and L2 may have to hold.

4.4.2. Cell of T

A cell of *T* is shown schematically in Fig. 15. The components of this diagram can be explained as follows:

1. R1 through R6 stand for identical groups of registers. In this paper we do not commit ourselves to details, such as how many registers they contain and what the size of each register is, because the choice of such details does not affect the essential nature of the processor. These registers serve two functions, as input and output ports in the process of communicating with other cells of *T* and *L*, and as local storage for P1 through P4.

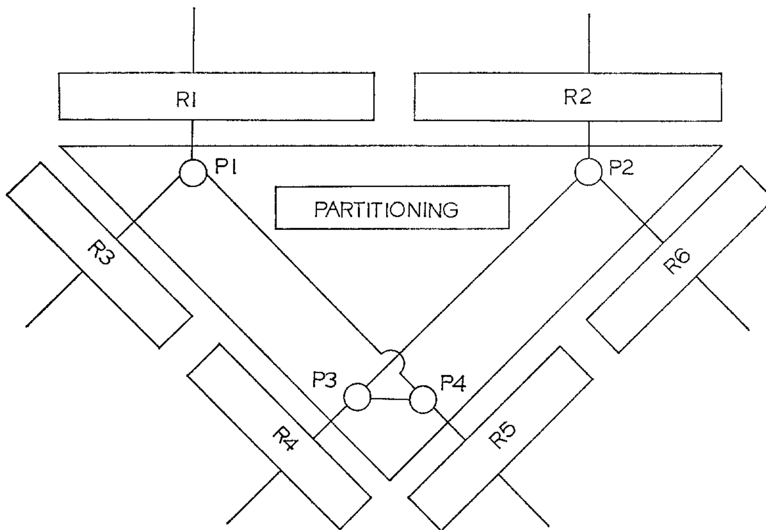

Fig. 15. Schematic representation of a cell of *T*.

2. P1, P2, P3, and P4 are the processing components of the cell; each one may belong to a different area of the processor. All have identical processing capabilities, the same amount of local storage, and identical state control units (similar to the component labeled *state* in Fig. 14). They must be able to perform the processing required by the internal mechanisms, described at length in Sec. 4.6.

3. The lines connecting the register groups and the processing components represent communication channels of identical capabilities, capable of carrying information both ways simultaneously. (Again, we do not specify details, such as channel widths.) Not all channels are used all the time: Fig. 22 specifies the eight possible partitioning patterns of the cell. The component in Fig. 15 labeled *partitioning* determines which partitioning configuration is assumed by the cell.

4. The pairs of lines originating in R1 and R2, in R3 and R4, and in R5 and R6 lead to the parent cell, left son cell, and right son cell, respectively. Each line represents a communication channel capable of carrying information both ways simultaneously. (The need for two separate lines leading to each of the parent, left son, and right son is explained by Proposition 1 of Sec. 4.2.)

In Sec. 4.6 the processing activities performed by cells of *T* are described with the help of temporaries. Again, we do not show an explicit mapping between these temporaries and the components of *T*, because many different mappings are possible.

4.5. Specification of Processing—Microprogramming Language

In this section we describe a simple language capable of specifying all the computational requirements outlined in Sec. 3. Since it is closer in style to a conventional microprogramming language than to a machine language, we refer to it as a *microprogramming language*.

Type A processing, which we described in Sec. 3, can be performed in cells of L alone, and we choose to implement it by executing suitable microprograms in cells of L. Type B and C processing requirements are more complex, and we implement them by executing suitable microprograms in cells of L, which, in turn, may initiate processing activities in cells of T.

Figure 16 is useful in introducing some terminology. It shows the RA already discussed in the context of Fig. 4, and indicates (in plain English) the processing activities that must be performed to bring about the effect of the reduction rule in question.

The totality of processing activities required by an RA, and expressed in the microprogramming language, will be called a *microprogram*. A microprogram is made up of *segments* specifying processing required by single symbols (atomic symbols or syntactic markers) or well-formed expressions of the RA. A segment comprises a sequence of *microinstructions*.

Microprograms specifying the effects of operator expressions reside outside the processor. When a reducible application is located, the appropriate microprogram is sent in via the root cell of T. Section 4.7 describes how it gets from the top of T to the top of the active area in question. In Sec. 4.6.2 it is shown how components of the microprogram find their way from the top of the active area to cells of L holding the reduction language program text. If the RA is well-formed, every symbol of it receives a segment of a microprogram, and only such cells receive microinstructions.

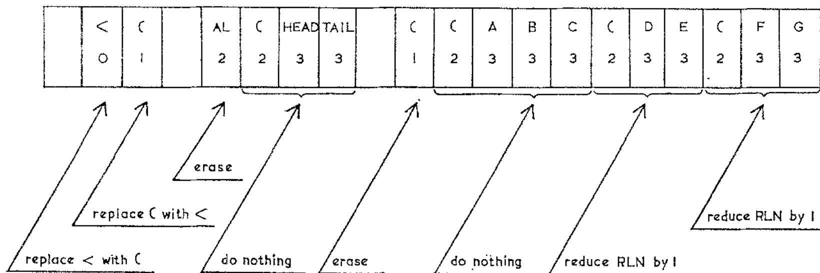

Fig. 16. Processing to be performed in cells of L to execute the reduction shown in Fig. 3 and 4.

4.5.1. Three Microprograms

We now introduce some of the details of the microprogramming language by means of three examples.

For ease of understanding and to avoid the need to specify low-level details of the internal representation that are irrelevant here, we have chosen an ALGOL-like representation for the microinstructions.

Since segments of microinstructions apply to constituents of the operator and operand, and these constituents form blocks of a partition of the RA in L, we arrange the segments of the microprogram in a linear sequence so that the order of the segments matches the order of the corresponding constituents in L. Because of this simple positional correspondence, the only information that has to be attached to any segment of microinstructions is the description of the constituent to which it applies (e.g., a single symbol with a given level number). and we call such a description a *destination expression.*

Our first example, shown in Fig. 17, is a microprogram for the primitive meta operator AL (see also Sec. 3.2.1.1).

The destination expressions of this example show the ways the microprogramming language deals with those aspects of the RA that become known only at runtime:

Constituent of source text	Destination expression	Segment of microprogram
</0	(S/0):	4: S :="(";
(/1	(S/1):	4: if S ="("
		then S :="<"
		else S :="⊥";
AL/2	(S/2):	ERASE;
f/2	(E/2):	do nothing;
(/1	(E/1):	MARK WITH x;
		4: if (RLN=1) & (S≠"(")
		then S :="⊥";
		8: if (N1=1) then ERASE;
		8: if POS# = 1
		then do nothing
		else RLN:=RLN-1;

Fig. 17. A microprogram for AL.

1. The microprogram is written with the assumption that the level number of the ⟨ symbol is 0 (whenever an RA is located, these so-called relative level numbers—RLN for short—are computed for each symbol by subtracting the true, or absolute, level number—ALN—of the ⟨ symbol from the ALN of the symbol in question).

2. The destination expression (E/i) indicates that the same segment of microprogram is to be sent to all symbols of a well-formed expression whose leftmost symbol has RLN $= i$. (The size of this expression is generally unknown prior to execution. Section 4.6.2 describes how symbols of this expression are located.) The destination expression (S/i) indicates that the segment is to be sent to a single symbol with RLN $= i$.

In the microinstructions, S, RLN, and POS# refer to registers of the cell of L executing the microinstruction in question (see Sec. 4.4.1).

The numeric labels in front of statements indicate the state of the cell of L in which the statement in question should be executed. Statements with the same label are executed in their order of occurrence. Some statements, such as ERASE, need no label, because their time of execution is determined in some other way (ERASE is executed at the end of executing the RA).

The phrase "MARK WITH x" activates the only available mechanism to analyze a sequence into its components. As a result of executing this statement, the whole marked expression is considered a sequence, and symbols of its component expressions receive a number in their POS# register indicating their position in the sequence, and their LENGTH registers receive a number indicating the number of elements of this sequence. (The full effect of the MARK statement is explained in Sec. 4.5.2, whereas the process of marking an expression, which begins in state 5 and ends in state 8, is described in Sec. 4.6.3.)

With these comments the microprogram for AL should now be readable. It says: the leftmost symbol of the RA should have RLN $= 0$, and this symbol should be changed to (. We need not verify that this symbol is ⟨, since the RA was located on the basis of its being a ⟨. The next symbol from left to right must be (with RLN $= 1$; if it is, it should be changed to ⟨; alternatively we signal an error. The next symbol—whatever it is—should have RLN $= 2$, and it should be erased. (Again, we know it is AL, since the microprogram was brought in on that basis.) The next expression, which is the parameter of the AL operator, should be left alone (its leftmost symbol must have RLN $= 2$). Following that is the operand expression, whose leftmost symbol must have RLN $= 1$. We erase this leftmost symbol if it is (, otherwise signal an error. In addition, all component expressions of the operand with the exception of the first have their RLN reduced by one.

Constituent of source text	Destination expression	Segment of microprogram
</0	(S/0):	S := M2(1);
AND/1	(S/1):	ERASE;
(/1	(E/1):	4: if (RLN=1) & (S≠"(")
		then S :="⊥";
		4: if (RLN = 2)
		then SEND1C(AND,S);
		4: if (RLN > 2)
		then S :="⊥";
		ERASE;

Fig. 18. A microprogram for AND.

We introduce some further details of the microprogramming language by showing a microprogram for the primitive regular operator AND in Fig. 18 (see also Sec. 3.2.1.2).

This example introduces what can be called the *message mechanism*, providing a means of communication among cells of L during execution, which is the chief requirement of Type B processing. A variety of SEND commands exists for the purpose of broadcasting information in active areas. For example, a message sent by a SEND1C statement moves up the area simultaneously with the state change $4 \rightarrow 5$. The command in our example has the form "SEND1C (binary operator, operand)," which causes the operands to be combined according to the binary operator as they move up in T. Only one message (containing the result) reaches the top of the active area; that message, again in the form (binary operator, operand), is broadcast down to every cell of L in the active area. Any cell can pick up the result in its M2 register, but in our example only one cell is programmed to do so, by means of the statement $S{:} = M2(1)$.

A microprogram for the primitive meta operator AA, shown in Fig. 19, illustrates how Type C processing is specified (see also Sec. 3.2.1.3).

This microprogram implements AA as shown in Fig. 7: the originally existing copy of f is left in place, and becomes the operator of $y1$, and $n - 1$ additional copies of $\langle/1$ and $f/2$ are created in front of $y2$ through yn.

The operand expression is marked with y; this causes the elements of the operand sequence to be indexed by setting the values of the POS# registers. Thus, if a symbol appears in the ith element of the operand sequence, the POS# register of the cell which holds the symbol is assigned the value i. We insert the symbol $\langle/1$ on the right of yi $(1 \leqslant i < n)$ by writing "INSERTS(right, $\langle$, 1)," and insert the expression $f/2$ on the left of $y2$

Constituent of source text	Destination expression	Segment of microprogram
</0	(S/0):	4: S :="(";
(/1	(S/1):	4: if S ="("
		then S :="<"
		else S :="⊥";
AA/2	(S/2):	ERASE;
f/2	(E/2):	MARK WITH x;
(/1	(E/1):	MARK WITH y;
		4: if (RLN=1) & (S≠'(')
		then S :="⊥";
		8: if (N1=1) then ERASE;
		18: if POS# < LENGTH
		then
		INSERTS(right,<,1);
		18: if POS# > 1
		then
		INSERTE(left,x,+0);

Fig. 19. A microprogram for AA.

through yn by writing "INSERTE(left, x, $+0$)." In the latter case x is the symbol with which we marked every symbol of the expression f, and since the information we are inserting comes from the source text and not from the microprogram, we give an increment ($+0$) to the original RLN instead of a new value for RLN.

The INSERT commands result in insertions adjacent only to the leftmost or rightmost cell of the expression to which they apply. Information to control where the insertion is to be made is in the L/R registers of an expression, placed there in the process of marking the expression. Consider, for example, the statement "if POS# $<$ LENGTH then INSERTS(right, ⟨, 1)." This is received by every symbol of the operand. The condition holds only in cells containing symbols of yi ($1 \leqslant i < n$). Moreover, we do not want to perform insertions next to each symbol of these expressions, only at the right end of their rightmost symbols. The command "INSERTS(right, ⟨, 1)" is executed only in cells whose L/R register contains the value "right."

A microprogram offers a way to specify the result of a reduction in terms of the operator and operand expressions of the original RA. Part II of this paper describes what is involved in executing microprograms, and Appendix B in Part II shows what happens in the processor when the microprograms for AL, AND, and AA are executed.

4.5.2. Description of Microprogramming Language

The microprogramming language described here is capable of expressing the computational requirements of a large number of primitives. It has been used to write microprograms for many primitives, including most of those considered by Backus[3] and Pozefsky.[14] The only primitive considered by Backus that cannot be programmed in this language is TRANSpose; it must be defined in terms of other primitives. (Microprograms cannot be composed: since each RA must be executed in its own area, composing two microprograms, i.e., executing one after the other in the same area, would be of very limited utility, hence not explicitly included in the design.) Although the microprogramming language has an ALGOL-like appearance, the simplicity of the constructs allows a very concise encoding into an internal representation.

A segment of a microprogram is composed of a destination expression, followed by a sequence of labeled or unlabeled statements. The permissible *destination expressions* are S/i and E/i with $0 \leqslant i \leqslant 3$, because beyond relative level number three we cannot distribute different microinstructions to different expressions (the reasons for this restriction are explained in Sec. 4.6.2).

Every statement should be preceded by a *numeric label*, unless (1) it is a MARK, ERASE, or no-op statement, (2) it is one of the arms of a conditional, (3) it is a SEND statement other than SEND1 or SEND1C, or (4) it uses some of the message registers (M1 through M4). Any integer used to designate a state in the state diagram (Fig. 13) can appear as a label of a statement.

The *conditional* has the following form: if ⟨predicate⟩ then ⟨statement⟩ else ⟨statement⟩. Neither arm of a conditional may be another conditional, or a MARK statement. The predicate is formed from relational expressions with the help of Boolean operators, assuming certain reasonable length restrictions. In a relational expression the usual relational operators ($=$, $\neq$, $<$, $\leqslant$, $>$, $\geqslant$) may compare constants, contents of any of the registers of the cell of L, or values of arithmetic expressions formed thereof (again assuming certain length restrictions).

In an *assignment statement* on the lefthand side one can write only S or RLN (all other registers of cells of L are set only in specific contexts, for example, by some of the other statements), whereas on the righthand side one can write a constant, the name of any register of the cell of L, or an arithmetic expression formed thereof assuming that certain length restrictions apply. When all quantities are available, the righthand side of the assignment statement is evaluated and stored in a temporary register (S′ or RLN′); the time of evaluation should be indicated in the statement label,

if possible. The assignment itself, however, is executed only at the final stage of the processing of the RA.

The *ERASE statement* clears all registers of the cell of L at the end of the processing of the RA.

The *SEND statement* is used to send messages to the top of the area, from which they are broadcast to all cells of L that are contained in the area. Sending and processing of different messages can be overlapped in time if the relative order is immaterial. Sequencing is made possible by indexing the messages; a message with index $i+1$ is sent only after all messages with index i have arrived at their destinations. Indexing is done by using SEND statements of the form SENDi, where $i = 1, 2, 3, \ldots$. The parameters of the SEND statements shown above are the messages to be sent. The number of parameters varies, but should not exceed some specified value. (Four parameters allow a large set of primitives to be implemented, so we choose the maximum to be four.) The messages sent by SEND1, SEND2, etc., will not interact with any other message in T. On arrival back at L the parameters of these SEND statements are placed into registers M1, M2, M3, and M4 of each cell of L in the area, ready to be used by the microprogram. Since registers M1 through M4 accept every message arriving at the cell of L in question, whenever their names appear in an expression in the microprogram, that expression is evaluated for every message accepted. (M1 through M4 are used most frequently in conditionals, since usually some part of a particular message is sought depending on some condition.) We can refer to components of a message produced by SENDi by writing M1(i), M2(i), and so on.

As an alternative, we may want the messages to be combined whenever they meet in some node of the area, such as adding them up, or selecting the larger one (see also the microprogram for AND in the preceding section). Such SEND statements are written as SEND1C, SEND2C, etc. (Statements of the forms SENDi and SENDjC must have $i \neq j$.) The first parameter of a SENDiC statement is the operator specifying the rule of combination. The second, third, and fourth parameters are to be combined separately according to the operator specified by the first parameter. When two messages produced by SENDiC statements meet in a node of the active area, the output produced has the same format as the inputs. The final result produced on the top of the area is broadcast down to L, and the components of the final result end up in the registers M2, M3, and M4 of every L cell of the area in question. (For any value of i, only one operator can be used in statements of the form SENDiC.)

All the statements of the form SEND1 or SEND1C should be labeled, each with the same label, chiefly to indicate whether the results of the MARK statement are needed to generate these messages. Other SEND

statements, i.e., SENDi and SENDiC, where $i > 1$, should never be labeled.

MARK statements are used to identify expressions that are to be inserted somewhere else in the program text, and also to identify elements of such marked sequences. Any segment can contain only one MARK statement, and such a statement cannot be either arm of a conditional. As a result, every cell of L receiving the MARK statement will be marked, and furthermore only constituents of the source text that have their own destination expressions can be marked. The full effect of the MARK statement is explained with the help of Fig. 20. Registers N1 and L1 make it possible, for example, to write microprograms to compare two arbitrary expressions for equality, or to insert the whole marked expression somewhere else in the program. Registers POS# (position number) and LENGTH allow us to write microprograms to do different things to different elements of the marked sequence, and they, combined with registers N2 and L2, allow us to insert the component expressions of this sequence at different places in the program. Finally, register L/R is used to locate the left or right end of any of the component expressions, in order to be able to make an insertion there. (The process of assigning values to these registers is described in Sec. 4.6.3.)

The *INSERT statement* has three variants. INSERTS is used whenever a single symbol is to be inserted from the microprogram. Its form is INSERTS (left/right, symbol, RLN). The first parameter specifies whether the symbol is to be inserted on the left or on the right end of the expression holding the INSERT statement in question. The second and third parameters are the symbol to be inserted and its RLN.

S	<	OP	(	(	A	B	C	D	(	E	F	G	(	H	I	(	K
RLN	0	1	1	2	3	3	3	3	2	3	3	3	2	3	3	2	3
MARKER 1			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
N1			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L1			15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
MARKER 2				X					X				X			X	
POS#				1	1	1	1	1	2	2	2	2	3	3	3	4	4
LENGTH				4	4	4	4	4	4	4	4	4	4	4	4	4	4
N2				1	2	3	4	5	1	2	3	4	1	2	3	1	2
L2				5	5	5	5	5	4	4	4	4	3	3	3	2	2
L/R				L				R	L			R	L		R	L	R

Fig. 20. The cells of L shown hold an RA. The operand expression has received the microinstruction "(E/1): MARK WITH x." The contents of all the registers set by this microinstruction are shown.

INSERTE is used whenever an expression (possibly a single symbol) is to be inserted from the program text. Its form is INSERTE(left/right, marker, increment to RLN). The first parameter is the same as in the case of INSERTS. The second parameter identifies the symbol or expression to be inserted, which must have been marked. The third parameter specifies how to adjust RLN of the symbol or expression to be inserted.

INSERTC is used whenever a component of a marked sequence is to be inserted. Its form is INSERTC(left/right, marker, position number, increment to RLN). The third parameter specifies which component of the marked sequence is to be inserted.

Although the microprogramming language described here has some powerful features (especially the SEND, MARK, and INSERT statements), it is basically a low-level language. It can be used to full advantage only if one understands the operation of the processor to a sufficient degree.

This language often allows several different microprograms to be written for the same primitive. The easiest examples to illustrate this involve some rearrangement of the operand. Consider a primitive EXCHANGE, whose effect is

$$\langle \text{EXCHANGE}, (x, y) \rangle \quad \Rightarrow \quad (y, x)$$

It is possible to write a microprogram that leaves the expression x in place, inserts y on its left, and erases the original copy of y from the program text. As an alternative, it is possible to write another microprogram that leaves y in place, inserts a copy of x on its right, and erases the original copy of x from the program text. Since for a short while two copies of y (or two copies of x) must exist in L, it would be desirable to move the shorter one of x and y. Since the lengths of x and y become known only at runtime, a third version of the same microprogram could test the lengths of x and y, and move the shorter one of the two.

One more issue that should be briefly mentioned is testing the syntactic correctness of the whole RA. Since the RA may be an arbitrarily long expression, with arbitrarily deep nesting, its syntactic correctness cannot always be fully tested by the processor. However, the following tools are available:

1. The segments of the microprogram must match the corresponding constituents of the program text, otherwise an error message is generated (when the microprogram is distributed, the only thing the processor has to do is to observe whether there are any segments of the microprogram that find no destination with the specified description or whether there are any occupied cells of L in the active area that received no microinstructions).

2. The microprogram can do some further checking of syntactic correctness with the help of the MARK statement and conditionals.

In fact, experience has convinced us that this kind of syntactic checking, in which syntax errors are discovered only when they prevent further processing, is extremely helpful.

APPENDIX A: PROOF OF PROPOSITION 1

Let t be a cell of T other than the root cell, and let t' be its parent cell. First we show that there is always at least one branch of an area between t and t'. We refer to a cell of L which is a descendant of t as a *leaf under t*.

Case 1. No $\langle$ symbol lies in any leaf under t. In this case, every leaf under t belongs to the same area, namely the jth one, where j is the largest integer for which the cell indexed by $i(j)$ is not to the right of the leaves under t.

Since the root node of the jth area is either in the root cell of T or in a cell of T which, by definition, has both the $i(j) - 1$ and $i(j + 1)$ cells of L as descendants, t must be a proper descendant of that cell, and thus a branch from t to t' must be part of the jth area. (In fact, in this case there is no need for more than one branch between t and t'.)

Case 2. At least one $\langle$ symbol lies in a leaf under t. Assume $i(p)$ is the index of the rightmost occurrence of $\langle$ under t. If $p = 1$ or $p = q$, where q is the total number of areas, then the root of the pth area is in the root cell of T, so we need a branch for this area between t and t'. If, on the other hand, $1 < p < q$, then the root of the pth area must have the $i(p + 1)$ cell of L as a descendant, which is not a descendant of t. Hence t is a proper descendant of the root of the pth area, and therefore there is a branch between t and t'.

Finally, we show that there can never be more than two branches between t and t', by showing that if there were more than two, all but the rightmost and leftmost ones would be in violation of the definition of an area. Assume there are three or more branches between t and t'. Consider a branch other than the leftmost or rightmost one, corresponding to, say, the mth area, and assume further that the leftmost and rightmost branches correspond to the kth and nth areas, respectively. By definition, the top of the mth area is in the lowest cell of T which has both the $i(m) - 1$ and $i(m + 1)$ cells of L as descendants.

The presence of the leftmost branch indicates that t has among its descendants $i(k + 1) - 1$, and the presence of the rightmost branch indicates that t has among its descendants $i(n)$. From $k < m < n$ it follows that $i(k + 1) - 1 \leqslant i(m) - 1$, and $i(m + 1) \leqslant i(n)$. Since both $i(m) - 1$ and $i(m + 1)$ are descendants of t, it follows that either t, or one of its descendants, holds the top of the mth area, and the branch between t and t' assumed to belong to $i(m)$ is in violation of the definition of an area.

REFERENCES

1. Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Information Processing 77* (North-Holland Publishing Co., 1977), pp. 849–853.
2. J. W. Backus, "Reduction Languages and Variable-Free Programming," IBM Research Report RJ1010, Yorktown Heights, New York (April 1972).
3. J. W. Backus, "Programming Language Semantics and Closed Applicative Languages," IBM Research Report RJ1245, Yorktown Heights, New York (July 1973).
4. A. L. Davis, "The Architecture of DDM1: A Recursively Structured Data Driven Machine," Technical Report UUCS-77-113, Department of Computer Science, University of Utah, Salt Lake City, Utah (October 1977).
5. J. B. Dennis, "Computation Structures," *COSINE Committee Lectures*, Princeton University, Department of Electrical Engineering, Princeton, New Jersey (July 1968).
6. J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," *Information Processing* 68 (North-Holland Publishing Co., 1969), pp. 484–492.
7. J. B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science*, Vol. 19 (Springer-Verlag, New York, 1974), pp. 362–376.
8. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Proceedings of the Second Annual Symposium on Computer Architecture* (IEEE, New York, 1975), pp. 126–132.
9. A. W. Holt and F. Commoner, "Events and Conditions," *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation* (ACM, New York, 1970), pp. 3–52.
10. A. Koster, "Execution Time and Storage Requirements of Reduction Language Programs on a Reduction Machine," Ph.D. thesis, Department of Computer Science, University of North Carolina at Chapel Hill (March 1977).
11. P. McJones, "A Church-Rosser Property of Closed Applicative Languages," IBM Research Report RJ1589, Yorktown Heights, New York (May 1975).
12. S. S. Patil and J. B. Dennis, "The description and realization of digital systems," *Rev. Fr. Autom. Inf. Rech. Oper.* 55–69 (February 1973).
13. C. A. Petri, *Kommunikation mit Automaten*, No. 2. (Schriften des Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962).
14. M. Pozefsky, "Programming in Reduction Languages," Ph.D. thesis, Department of Computer Science, University of North Carolina at Chapel Hill (October 1977).
15. D. F. Stanat and G. A. Magó, "Minimizing maximum flows in linear graphs," to appear in *Networks.*
16. D. F. Stanat and G. A. Magó, "A parallel algorithm for minimizing maximum flows in linear graphs," in preparation.
17. K. J. Thurber, *Large Scale Computer Architecture—Parallel and Associative Processors* (Hayden Book Co., Rochelle Park, New Jersey, 1976).