Paul McJones

A NETWORK OF MICROPROCESSORS TO EXECUTE REDUCTION LANGUAGES

Gyula A. Magó
Department of Computer Science
University of North Carolina
Chapel Hill, N. C.

June, 1976

A NETWORK OF MICROPROCESSORS TO EXECUTE REDUCTION LANGUAGES

The paper describes the architecture of a processor capable of efficiently executing the reduction languages as defined by Backus.

The processor is obtained by interconnecting two different networks of microprocessors: a linear array of identical cells, and a tree-structured network of identical cells. Both types of cells have modest processing and minimal storage requirements.

Since the processor implements a high-level language, and it is designed to be able to move data around easily, its efficient operation is not restricted to any particular data representation, or to any special class of problems. The processor has a modular, easily expandable structure, and it accommodates the unbounded parallelism permitted by reduction languages; hence its performance is essentially proportional to the amount of hardware built into it.

KEYWORDS: reduction languages; network of microprocessors; parallel processing; high-level language processor architecture; computer architecture.

CONTENTS

1. INTRODUCTION

While microelectronics is - or promises to be - spectacularly successful in small-scale and simple applications (digital watches, cameras, pocket calculators, etc.), its impact on the architecture of computers is somewhat harder to predict. Employing large amounts of main memory, and creating systems with distributed processing capabilities (meaning that the processors in the system are very loosely coupled), are both likely to increase the performance of computer systems, and such approaches are being pursued. However, when we ask how to build increased amounts of logic into a CPU to obtain a proportionate amount of increase in performance, we realize that all known methods (e.g. building special functions into the hardware, pipelining, multiprogramming, array processors, etc.) yield rapidly diminishing returns.

In this paper we shall explore the consequences of the hypotheses that (1) the root of the problem just mentioned lies with the languages used to program our computers, and (2) increasing the performance of processors is hard because concurrency is sought on too high a level.

The von Neumann architecture has dominated the computer field for a long time because of its flexibility and simple sequential nature. The essence of this architecture is that a single processor is connected with a passive memory, and the elementary actions of the processor are testing and updating small segments (words) of this memory. A very important aspect of this architecture, rarely stated explicitly, is that it makes data movement an expensive proposition, since everything has to go through the CPU, in small chunks, sequentially. (This is why one of the most important practical complexity measures for programs is the number of memory accesses required.)

Once this point of view is taken, techniques such as indirect references, pointer variables, and list processing can all be considered as ways to avoid moving data.

Despite frequent claims to the contrary, the development of programming languages was always heavily influenced by what could be done efficiently in a von Neumann architecture. A good example (although not the only one) is the way programming languages deal with the problem of run-time storage management. Some try to avoid it (FORTRAN), some try to minimize it (ALGOL 60), and some put it under the direct control of the programmer with a warning that it is expensive (PL/I).

Since today's technology could permit a very different architecture, perhaps it is time to face the fact that moving data is an integral part of any computation, and attempts should be made to design processors that would easily accommodate -- instead of penalizing - massive movements of data.

The obvious next question is: what kinds of data movements should a processor cater for? The suggestion of taking a network of von Neumann computers and connecting them via some very general routing network is, in our opinion, avoiding, rather than facing the problem. The essence of good processor design is knowing what information is needed when, and where, and making sure that everything gets in the right place at the right time. This implies that a processor for a specific programming language, rather than the elusive "general purpose" processor, might be a more promising avenue to follow when trying to explore the potentials of such an approach. The programming language to be implemented would be a specification of the overall behaviour to be synthesized, and it would dictate what kinds of

data movements need be done efficiently.

The choice of programming language is likely to be very important for the success of such a project, and for the significance of the results and conclusions obtained. It is almost obvious that one should avoid programming languages that were designed to be efficiently implementable on a von Neumann computer, since many of their characteristics have been determined (in a more or less well-understood fashion) by the peculiarities of the von Neumann design.

In this paper our choice is the class of reduction languages as defined by Backus[2]. (There are many reduction languages, differing in their choice of primitives, but, as it turns out, our processor is remarkably insensitive to the specific choice. All that matters is the semantic framework of these languages.) Reduction languages are very expressive applicative languages, having some resemblance to LISP and APL. They are extremely demanding from the point of view of storage management (hence totally unsuited for a von Neumann computer), but for reasons to be outlined later they are demanding not in an arbitrary, but rather in a disciplined fashion, making the language an excellent candidate for our purposes.

In carrying out an exploratory design of a processor for a reduction language our main aim will be to accommodate all demands of the language for data movement as efficiently as possible. From the outset we shall try to make all components of the processor cooperate in doing well what is usually the costliest part of processing: moving data around.

The elementary steps of computation in reduction languages are called reductions, and the language definition permits many reductions to take place

simultaneously. To achieve efficient execution we shall accommodate all concurrency permitted by the language definition, but since we have micro-electronic circuits in mind for implementation, we shall go one step further, and move the level of concurrency below the level of language primitives. As a result, the processor will be made up of many identical component processors, implemented by identical cells of logic circuits, whose operation will not be explainable in terms of the source language alone. By uniformly decomposing even the language primitives, and executing the component computations simultaneously, the processor will greatly gain in execution speed, and in simplicity.

Summarizing this brief introduction, we shall in this paper outline the architecture and an implementation of a hardware processor for reduction languages in order to demonstrate that LSI can yield considerable performance increases even for large-scale processors, if it is used primarily to facilitate data movement for the language to be implemented, and if the language requires data movements in some disciplined fashion.

2. DESIGN OBJECTIVES

In the course of designing the processor we shall aim to fulfill the following three objectives:

a. high speed of execution

b. highly regular, cellular structure

c. simplicity.

The most important aim is to achieve a high speed of execution, partly by taking utmost advantage of the possibility of concurrent execution of all reductions as permitted by the language definition. More precisely, we shall require that the execution of any single reduction on this processor be at

least competitive in execution time with the corresponding sequence of machine instructions on any von Neumann computer, and, in addition, accommodation of all permitted concurrency should be effortless, requiring a minimum amount of overhead.

We shall also require that the processor be constructed by interconnecting a large number of simple, identical processors (cells) in a highly regular manner. This we take to imply that any cell can communicate with only a small number of other cells, designated as its neighbours. Some of the resulting advantages of satisfying this requirement are that the processor will be "modular", easily expandable in size, it will be suitable for implementation via LSI technology, and the regular, cellular structure will also facilitate high speed of execution. (If all cells can have all their neighbours placed close to them in space, then delays associated with signal propagation will be reduced, and fast logic can be used for implementation.)

Simplicity is an aim made especially compelling by the simplicity of reduction languages. Beyond that, conceptual simplicity is essential if the user is to understand the operation of the processor in terms of language concepts, which is a requirement for the intelligent use of the processor. Finally, simplicity of details - by facilitating design, manufacture, and maintenance - will contribute to making the processor a viable one.

3. REDUCTION LANGUAGES

In this section we give a brief and very informal introduction to reduction languages, in order to make the paper self-contained. However, one cannot fully appreciate these languages without reading Backus[1],[2]

and Berkling[3]. Although we use the term reduction language for sake of brevity, we always mean a λ - Red or a λ - M - Red language as defined by Backus[2], and extended to sequences.

Apart from a few syntactic markers, these languages contain only two kinds of atomic symbols: variables, and so-called objects. Variables are used to guide substitution only, whereas objects serve all other purposes: they are used as data items, primitive operators, names of defined functions, etc.

Four types of composite expressions are allowed:

a. sequences (a1, a2, ...an) if n ≥ 1, ϕ otherwise

b. lambda expressions (λv, b)

c. applications ⟨a, b⟩

d. formal applications {a, b}.

A sequence can contain an arbitrary number of elements, each ai an arbitrary well-formed expression. In a lambda expression v is a variable, whereas b stands for any well-formed expression called the body of the lambda expression. In an application a (the operator) and b (the operand) are well-formed expressions containing no free variables. In a formal application a and b are again well-formed expressions, but at least one of them must contain a free variable.

Among these composite expressions only applications specify computations. Since the program text at any time can 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[2].

The process of executing an innermost application is called a reduction, and so we shall often refer to an innermost application as a reducible application.

A reduction results in replacing the 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 object, it might be a primitive operator (in which case its effect is defined), or it might be the name of a defined function, i.e. some well-formed expression containing no applications (in which case the object 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). If the operator is a lambda expression, the operand gets substituted for every free occurrence of the variable in the body of the lambda expression. The computation comes to a halt if there are no reducible applications left, and the program text so reduced is the result of the computation. If the result of a reduction is undefined, the symbol $\perp$ is used to denote it.

The following observations are relevant to matters of implementation:

a. if we consider the program text to be a linear string of symbols, then the reducible applications form disjoint segments of that string

b. what happens to a reducible application is determined only by its operator and operand, and nothing outside it influences the reduction

c. the operator and operand can be arbitrarily long

d. the reducible applications can be reduced in any order, including their simultaneous reduction, owing to the so-called extended Church-Rosser

property of these languages.

4. DESCRIPTION OF PROCESSOR

4.1 INTERCONNECTION PATTERN OF CELLS

In this section we describe the structural aspects of the processor, because they play a crucial role in determining the quality of behaviour produced by the processor. Among other aspects we shall consider what kinds of component processors are to be used, what their approximate functions should be, and how they are to be interconnected.

If the processor is to reduce all innermost applications simultaneously, the separation of the processor into a passive memory and one or more active processors is not a workable solution, since no fixed number of processors could carry out the truly simultaneous reduction of an unbounded number of applications. Not only is the number of reducible applications unbounded, but there is no bound on the size of operators and operands either, again strongly suggesting that dedicating component processors to reducible applications is doomed to failure. We are thus led to our first design decision: the processor will be made up of a number of identical processors, each capable of holding a single symbol (variable, object, or syntactic marker), and each capable of some processing. (This immediately implies that the implementation of these component processors will determine the maximum length of symbols allowed.)

The next decision is dictated mainly by our requirement for simplicity: the component processors (implemented by identical hardware cells) will be interconnected as a one-dimensional array. This choice will facilitate setting up a very simple one-to-one correspondence between the most obvious interpretation of the program text, which is a linear, fully

bracketed string of symbols, and the internal representation of it.

The network - as it stands now - is the simplest example of the kind of cellular networks researchers have long been fascinated with, the n-dimensional rectangular arrangement. As such, it suffers from the weakness all of them do: the inability to move data around efficiently. It may seem, indeed, that the structural requirement for cellularity (a network of identical cells interconnected in a regular fashion, each cell communicating with a small number of its immediate neighbours only) cannot be reconciled with the functional requirement of efficient, global control over the operation of all cells.

Fortunately, however, it is possible to find a compromise, by combining the linear array of identical cells (L) with another cellular network of identical cells of a possibly different kind, arranged in the form of a full binary tree (T), as shown in Fig. 1.

Before outlining the many possible uses of the tree network, two comments should be made regarding the structure of the processor. We shall keep on referring to it as a cellular network, although two different kinds of cells, and two different interconnection patterns are used in it. Such a network has the following important and very attractive property: the growth rate of the amount of hardware built into the processor is a linear function of the length of the linear array, L. More precisely, if the length of L is 2^H, then the number of cells in T is 2^H-1, so the amount of hardware is almost exactly $2^H \cdot (\ell+t)$, where ℓ and t represent the amounts of hardware built into a single cell of L and T, respectively.

We can now begin to ponder the possible uses of this network, first in rather broad, general terms only. The operation of the processor is

envisioned as follows: the program text can normally be found in L, and the processor - by using T to propagate control information, move data, and do most of the processing - creates a new, transformed version of the program text in L, corresponding to the effects of the reduction rules applied.

Fig. 2 shows the different patterns of moving information in the processor. By propagating information upwards in the tree simultaneously from all cells of L, the processor is able to evaluate the global situation in L, can do processing on data coming from different cells of L, and can store the results produced into the nodes of T. Since the processor will communicate with the outside world via the root-node of T, producing output fits the same pattern too. We shall often refer to processing activities in which information is moving upwards as an upward cycle, because these activities will usually be executed in a synchronous fashion, at the same time in all nodes of the same level of T, taking H steps to complete them.

By propagating information downwards in the tree (a downward cycle), the processor can distribute the information produced during a previous upward cycle, can do processing in L based on them, can read in input, and so on.

The contents of any cell in L can be moved into any other cell of L in at most 2H steps (where H is the height of T), by moving it first up towards the root of a subtree of T, then routing it downwards to its destination; furthermore, many such moves can be carried out simultaneously.

The contents of L can be rearranged by shifting different segments in different directions, by differing numbers of positions. Although Fig. 2 seems to suggest that T is not needed for this, these movements really could not be carried out without T making the preparation for them.

We should mention that T can be considered as an interconnection (routing) network of a rather restricted kind[4], but it is also much more than that, since the cells in it are capable of processing in addition to that required for routing functions.

4.2 INTERNAL REPRESENTATION

We have already decided to represent the program text as a linear string of symbols (each symbol held by a different cell of L); now it is time to decide on some further details of the internal representation.

The representation should be chosen to facilitate efficient processing, and as it turns out what we want to do most frequently is to locate component expressions with certain properties, e.g. innermost applications, and their operators and operands. We want to be able to do this kind of processing with the help of the networks T and L, in one cycle. Moreover, since we want to keep the cells of T and L simple, we also require that only a small, constant amount of information be stored in any cell. With these constraints the representation we are about to describe appears to be superior to all other alternatives[5].

In order to avoid the need to recompute the nesting level of certain symbols over and over again, we shall store this information with every symbol, and then eliminate closing brackets of every kind --), >, and } -- from the source text for they have become superfluous, leading to a considerable decrease in the length of the expression. This representation can also be considered as the linear, prefix notation of the tree that naturally corresponds to the original source text (the root, labelled with (, <, or {, followed by the successor trees from left to right in the same

notation).

An example in Fig. 3 should clarify the details, also showing the only additional syntactic change we have made, which is replacing (λ v $\underline{b}$) with λ_v $\underline{b}$, where λ_v is a root node and $\underline{b}$ is its successor tree.

We describe one more interesting aspect of internal representation, which is the representation of variables, because there again the issue of storing vs. recomputing is raised. Since the absence or presence of free variables makes the only difference between applications and formal applications, every time $\langle(\lambda$ v $\underline{b}),\underline{x}\rangle$ is reduced by substituting $\underline{x}$ for every free occurrence of v in $\underline{b}$, every formal application internal to $\underline{b}$ must be checked to see if it is to be changed into an application. Again, a massive amount of processing could be avoided if we store a list of free variables with every formal application, and every time we substitute for one of them (say v) we take note of it (by removing v from the free-variable list of all affected formal applications), because now the only thing we must be able to detect is when the last free variable is removed from the free-variable list of any formal application.

The important question then is: can such a list of free variables for each formal application be efficiently stored, updated, and tested? Fortunately, the structure of reduction languages is such that there is no way to increase the number of free variables in any formal application by performing reductions, since in a reducible application neither the operator, nor the operand can contain free variables. This implies that the following kind of restriction could be the key to a solution: neither the source text, nor any definition can contain formal applications with more than N free variables. If N is large enough, this would mean no serious practical

limitation in the use of the language, and would permit the following internal representation: 1-out-of-N binary codes would stand for variables, N-bit vectors for lists of free variables, and testing and updating such lists would become a very simple and efficient operation.

4.3 PROCESSING REQUIREMENTS OF A REDUCIBLE APPLICATION

In this section we shall consider the problems we are likely to encounter when trying to implement the definition (i.e. the reduction rules) of a reduction language in our network of processors, hence when having to be concerned with the limitations of physical space and time.

We shall consider three primitive operators as examples, all taken from[2]. The first primitive, TAIL, does the obvious:

$(\underline{x1}) \rightarrow \phi$; $(\underline{x1}, \underline{x2}, \ldots \underline{xn}) \rightarrow (\underline{x2}, \ldots \underline{xn})$; $\perp$

i.e. if its operand is a one-element sequence, the result is the null-sequence (ϕ), if it is a sequence of more than one element, then the first element is removed from the sequence, and if the operand is anything else, the result is undefined ($\perp$). (Note that the clauses of the definition are examined from left to right, and the first one applicable to the given input determines the result.) An example for the use of TAIL follows, using our internal representation:

<	TAIL	(	A	B	C	→	(	B	C	
5	6	6	7	7	7		5	6	6	

Obviously, the processor first will have to determine which clause of the definition applies, then it will have to erase the contents of the cells of L holding the symbols <, TAIL, and A, and finally it will have to adjust the integers representing the nesting levels of the remaining symbols in order

that they would keep on representing nesting levels. It is worth noting that when executing this reduction, storage is released, and what happens to any of the cells involved depends only on its position relative to the others (i.e. information does not have to be passed from one cell of L to another).

Another primitive with a well-known effect is AND:

(T, T) → T; (x, y) → F when x,y ∈ {T,F}; ⊥

i.e. if the operand is a two-element sequence, and both elements are equal to the atomic object T (representing the truth-value "true"), then the result is T, if one or both of these atomic objects is F (meaning "false"), then the result is F, and if the operand is anything else, the result is undefined (⊥). An example for the use of AND is

```
< AND ( T F    →    F
7 8   8 9 9         7
```

In addition to finding the applicable clause of the definition (which we shall refer to as "syntax analysis") and erasing, the two symbols T and F will have to be brought together in some cell of T to perform the AND operation on them, then the result will have to be placed into an available cell of L, which in this case should be easy to perform: the result might replace any of the original symbols.

The third example is a primitive called DISTL ("distribute from the left"):

(x, (y1, y2, ... yn)) → ((x, y1), (x, y2), ... (x, yn)); ⊥

the use of which is illustrated by the following example

```
< DISTL ( A ( B C D E    →    ( ( A B ( A C ( A D ( A E
2 3     3 4 4 5 5 5 5         2 3 4 4 3 4 4 3 4 4 3 4 4
```

One of the new problems posed by DISTL is that we have to create (n-1) new copies of the string "(x", and we may not be able to find enough empty cells inside the reducible application in question. If this is the case, and there are enough empty cells in L outside the reducible application, then we can make the necessary number of empty cells available in the application by moving (shifting) all or some of the symbols in L by a suitable number of positions, making sure, of course, that the relative positions of all symbols in L will remain the same.

To be able to do this assumes that the agent performing this operation has a "global understanding" of the situation in L, and knows the location of each empty cell of L; fortunately the tree network T will be able to act as such an agent.

Following the creation of the necessary number of empty cells in the appropriate places (in our example, at least two empty cells in front of B, C, D, and E), the string "(x" will have to be moved through the tree network, and (n-1) additional copies of it will have to be created as it moves, so that the formation of the result string could be completed.

As it turns out, all reduction rules can be implemented on our processor by using the few elementary processing activities described in the above examples. Furthermore, they always have to be combined according to a very definite pattern, which can be expressed by saying that every reduction rule is implemented in three phases, as follows:

PHASE 1: locate a reducible application
determine what its operator is
find the applicable clause of its definition ("syntax analysis")
compute the result of the (arithmetic, Boolean, etc.) operation

determine the length of string (or strings) to be moved, and
store this length information into all cells of L around
which empty cells will have to be created

PHASE 2: for each occupied cell of L, determine in which direction, and
by how many positions it will have to move
move every occupied cell of L as specified

PHASE 3: move segments of the reducible application through T to their
destinations
change level numbers to reflect the new structure of the tree
erase symbols not needed any more.

4.4 MICROPROGRAMMING - THE NEED FOR IT

Having briefly outlined the typical, frequently used, elementary processing activities in rather general terms, we would now like to describe more precisely that for any given reduction rule just what kinds of processing activities should apply to any given part of the operand, when should these processing activities take place, and where should all this knowledge reside in the processor (e.g. in hardware, or in software)?

One obvious possibility is to endow every cell of the network with full knowledge of all reduction rules, and since all this information would permanently reside in the cells, they would always be ready for all eventualities of processing (for any occupied cell inside a reducible application, if the cell is told somehow what the operator is and what part of the operand it is holding, it would immediately "know" what to do to it).

A somewhat less obvious alternative to this would be to keep only one copy of this information, in some central place (e.g. outside the processor), and distribute the necessary information whenever it is needed (for

any occupied cell inside a reducible application, the cell would have to be given information as to what to do).

After taking a closer look it is easy to see that the second solution is far superior to the first one. While the second solution requires only one, the first requires storing many (possibly millions of) copies of the same information, which for a not-too-large set of primitives, using a careful encoding, turns out to be many thousands of bits of information. At the same time, very fortunately, the second solution has almost no disadvantage as far as execution time is concerned, because distributing information as to what to do to different parts of a reducible application can be almost fully overlapped with other activities that must be present in both solutions anyway (e.g. telling every cell what part of the operand it is holding).

For these reasons we adopt the second solution, and shall refer to it as the microprogramming solution, because the language in which the relevant instructions are expressed is similar in style to a conventional microprogramming language, and its primitives are much lower-level than those of the reduction language the processor is implementing.

The hardware requirements of the microprogramming solution are: cells of T must be able to distribute the microinstructions, and cells of L must be able to hold all microinstructions relevant to a single symbol (typically 10 ~ 100 bits) and must be able to decode and execute any microinstruction.

By adopting the microprogramming solution we not only drastically reduced the amount of hardware needed (approximately 100 ~ 1000 times), but also make the processor much more flexible with respect to primitives, since they are now much easier to replace, modify, etc.

4.5 A LANGUAGE FOR MICROPROGRAMMING

A set of microinstructions able to bring about the effect of a particular reduction rule will be referred to as a microprogram for that reduction rule. What is unusual about such a microprogram is that it is a set of microinstructions to be executed simultaneously in many different processors (usually in cells of L). Every time a new reducible application is encountered by the processor, a new copy of the corresponding microprogram will have to be brought in from the outside via the root-node of T, moved through T, and distributed to the relevant cells of L, called their destinations. This suggests that the viability of the processor will depend to a large degree on the microprograms being sufficiently short.

In the following we shall give an informal and somewhat incomplete description of a suitable microprogramming language, the design of which was largely determined by two main requirements: it had to produce sufficiently short microprograms for the important primitives, and it had to be flexible and expressive enough to facilitate programming of many, powerful primitives.

Before actually showing some programming examples (for the three primitives described in 4.3), certain interesting aspects of the language will be described that crucially contribute to achieving the design objectives.

(1) A "positional" notation is used to specify implicitly the destination of microinstructions. Since groups of microinstructions apply to constituents of the operator and operand (in our internal representation these constituents can be either single symbols or well-formed expressions), and these constituents of the operator and operand form blocks of a partition of the reducible application in the linear array L, we shall arrange the microprogram into a linear sequence in such a way that the order of the blocks

of microinstructions will match the order of the corresponding constituents in L. This way the only information that has to be attached to any group of microinstructions is the description of the constituent to which it applies (e.g. a single symbol with a given level number), and we shall call such a description a destination expression. The advantage of this arrangement becomes obvious when we consider an alternative, say placing the groups of microinstructions in an arbitrary order, in which case every group has to carry with it not only a description of the constituent to which it applies, but also where that constituent can be found, i.e. an address.

The conciseness of this specification comes from the fact that the position of the group of microinstructions in the microprogram carries a lot of information, and for this reason we can liken it to the positional notation of numbers or the positional notation used in many fixed-format (usually low-level or special-purpose) programming languages. It should be emphasized here that this economy in description relies heavily on the chosen linear representation, and this is one of many places where the efforts of the processor to maintain that representation - by moving data - pay off.

(2) A repetition mechanism is used to avoid the rigid, positional, one-to-one correspondence between the program text and the microprogram, thereby allowing us to send the same microinstruction to many different cells (e.g. to all elements of a sequence or of a subtree, where the number of elements is not known to us), or alternatively to send many different groups of microinstructions to the same cell in an organized fashion.

(3) No extra instructions are needed to check the syntactic correctness of the input, if we simply raise some error condition whenever the microprogram does not match the program text, or a specified repetition

condition (e.g. a sequence should have exactly four elements) is violated. The only thing the processor will have to do is to observe if there are any microinstructions that find no destination with the specified description, or if there are any cells receiving no microinstructions in the reducible application.

The first example shown in Table I. is the microprogram for the primitive TAIL.

As the example reveals, we chose a representation for the statements of the microprogramming language with an ALGOL-like appearance to ease communication and to avoid the need to specify many low-level details of the internal representation that are irrelevant here. However, the fact that this is a very low-level language cannot be hidden: only a handful of variables are allowed, there are very limited options to form expressions, etc.

In the destination expressions all level numbers are relative to the current reducible application ('<' having relative level number 0), S identifies a single symbol, while E stands for a well-formed expression (a subtree). In the latter case the enclosed level number is that of the leftmost symbol of the expression (the root of the subtree). The keywords overlap and to specify repetition; S, L and "valency" are variable names, S and L standing for the "symbol" and "level number" fields of the given cell, while "valency" holds the number of successors of any given node (this information can easily be deduced from information held in T, whenever needed). Finally, whenever a conditional instruction applies to a tree, only the root-node is tested, but all nodes and leaves are modified.

With these comments the microprogram should now be readable. The leftmost symbol -- whatever it is -- should have relative level number 0,

and it should be erased (we do not look at the symbol, because by now we know it is "<", since the reducible application had been located on that basis). The next symbol -- whatever it is -- should have relative level number 1, and it should be erased (again, we know it is TAIL, since the microprogram had been brought in because of it). Next we deal with a tree (a well-formed expression), whose root has the relative level number 1: if the root-node holds the symbol "(", then the relative level number of every symbol in this tree is decreased by one, alternatively we signal an error by writing "⊥".

In the next destination expression the keyword overlap indicates that instead of going to the next constituent in the program text on the right, we should distribute another "layer" of microinstructions to the current constituent. The first microinstruction says take the leftmost symbol -- whatever it is -- with relative level number 1 (by now we know that the symbol is "(", since the previous instruction tested for it), and if it has only one successor, replace the symbol with ϕ, otherwise do nothing. After that, we erase the next subtree whose root has relative level number 2, and keep all the others (to last indicates that we distribute as many copies of this microinstruction as needed).

The next example shown in Table II. is the microprogram for the primitive AND.

In this example operand1, operand2, and result are names of variables; the microprogram connects them with the program text, while an internal mechanism, invisible on the level of this language delivers operand1 and operand2 to a processor in T, has the AND operation performed on them, and places the result of this operation into the variable called result.

(Obviously, the timing of these actions is important, but again invisible in the microprogramming language.)

Our third example is the microprogram for the primitive DISTL (see Table III.).

In this example string1 and string2 are names of variables, standing for parts of the reducible application to be moved. The micro-instruction "becomes" triggers an internal mechanism to place consecutive integers (from left to right) into all symbols of string1 and string2, to guarantee their orderly arrival to their destination. The keyword before specifies that string1 and string2 (in that order) have to be inserted on the left end of the program component in question, and its main function is to store the joint length of string1 and string2 in that place to facilitate storage management.

4.6 INTERNAL MECHANISMS NOT MICROPROGRAMMED

The microprograms described in the previous section are made as short as possible by encoding the information contained in them with the greatest economy, and by making sure that microprograms do not carry unnecessary information. To this end, microprograms should only contain information that is specific to the reduction rule (or operator) in question.

Whereas processing activities specific to individual reduction rules, and prescribed in microinstructions, are generally simple, the ones common to all reduction rules, and not even mentioned in microinstructions, tend to be more complex and intricate. Since the main problem in designing this processor is to synthesize the required global behaviour from simple, local activities in the nodes of T, (which might be considered as a programming

problem with very unusual primitives), and to do so efficiently, (which might, for example, mean that a certain result must be produced in a single upward cycle), we must in this paper describe these important mechanisms at a sufficient level of detail to demonstrate that they can be implemented within the bounds of these constraints.

The description of these mechanisms usually amounts to specifying what an arbitrary cell of T does, and to keep the size of this paper within reasonable bounds we shall describe them in an informal style, relying heavily on examples. We present every one of the five major mechanisms at considerable detail, skipping only a few of the less important aspects. All these mechanisms are brought into play automatically in certain states of the processor (i.e. they are not initiated by microprograms), two of them work for certain reduction rules only (4.6.3 and 4.6.5), and two of them use information prepared for them by microprograms (4.6.4 and 4.6.5).

It should be noted that some of the mechanisms triggered by microprograms but not described in this paper (e.g. determining the length of a component of the operand, or laying down consecutive integers for purposes of moving data) can easily be constructed along the lines described here.

4.6.1 LOCATING REDUCIBLE APPLICATIONS

Recall that we decided to do most processing by moving information along paths between the root-node of T and the leaves of T. This implies that most processing related to a reducible application will take place in a subtree of T, whose leaves are the cells of L holding the reducible application in question.

Whereas reducible applications are held by disjoint segments of L, the subtrees of T intended to serve them are not necessarily disjoint, as the

example in Fig. 4 shows. Since the cells of T are required to be identical, the simplest way several reducible applications could use a cell of T without interfering with each other is if the hardware of the cell is divided into parts, and each reducible application uses a different part. The process and the result of separating a cell of T into parts will be called the partitioning of that cell. (Fig. 4 shows cells that have to be separated into two or three parts.) Because of partitioning the nodes of T, the part of T serving a particular reducible application is usually not a subtree of T, so we shall refer to it as an active area of T. All cells, or parts of cells, not included in any active area will remain unused temporarily, and will be said to be inactive.

All in all, locating reducible applications means dividing the processor into many different, disjoint, active areas, one for each reducible application. (This process involves a kind of self-organization: the processor organizes itself so that its structure will suit the current program text to be processed.)

Since partitioning the cells of T implies multiplying the amount of hardware built in, and complicating the overall behaviour, we have to take a careful look at these complications in order to be able to estimate their effects.

We begin with an examination of the global aspects of locating reducible applications. In a single upward cycle the following three activities take place:

(1) all nodes of T get partitioned, based solely on the locations of the application symbols ("<") in L (this is because each "<" symbol is the leftmost symbol of a program segment in L which is potentially a reducible

application)

(2) when certain ways of partitioning a node of T suggest the possibility, it has to be determined if part of that node is the "top" of an active area (we shall give a definition for this shortly)

(3) if part of a node of T turns out to be the top of an active area, it has to be immediately determined what the operator of the corresponding reducible application is, and this information must be propagated up to the root-node of T, so that during the next downward cycle the required micro-program could be sent in.

Algorithms for (2) and (3) can be explained without any reference to T. Using our internal representation the algorithm to locate reducible applications is very simple. To determine if an arbitrary "<" symbol marks the beginning of a reducible application we have to look at its level number (i), at the level number of the nearest "<" symbol on the right (j), and possibly also at the smallest level number of any other symbol in between (k). The application on the left is innermost (hence reducible) if either $i \geq j$, or alternatively $(i < j)$ and $(i = k)$. Example for the first alternative is shown in Fig. 5.a, whereas Fig. 5.b shows example for the second.

The state diagram of Fig. 6 is the basis for determining what a reducible application is supposed to do. The diagram shows the significance of the leftmost symbols of a reducible application. Note, however, that not all of the branches in the diagram correspond to separate symbols of the program text: every object carries with it two bits indicating if it is a primitive or defined operator, and if it is a regular or meta operator[(2)].

We will now use an example to explain how the "<" symbols in L determine the partitioning of all nodes of T. A symbolic representation is

used in Fig. 7 to show different patterns of partitioning the nodes of T. The level numbers have been omitted from the program text contained in L, but all cells of L and T are numbered for ease of reference.

The following process is central to locating reducible applications: in an upward cycle all "<" symbols begin to move towards the root of T, each one causing nodes of T to be partitioned in order to separate an area (active or not) for itself. The details of how this is done (in particular, what happens when several "<" symbols meet in a node of T) are to be explained soon, but in the meantime it might be helpful to imagine that each "<" symbol is building a "wall" through T, which will become the left-hand boundary for that area. (The paths along which the "<" symbols travel need not be identical with the links shown in Fig. 7, and will be considered later, with the help of Fig. 9.)

The node of T in which a "<" symbol meets the nearest "<" symbol on its right in L (called its right neighbour) contains the top of the area for the given "<". For example, the "<" symbol in cell 2 of Fig. 7 moves through the cells 2, 17, 25, and 29, and in 29 it meets its right neighbour coming from 5. Hence the segment of the program text in 2, 3, and 4 has its area in cells 2, 3, 4, 17, 18, 25, and 29, the top of the area being in 29.

Each "<" symbol must also meet the nearest "<" on its left, so that this left neighbour can determine where the top of its area is. It is not possible for an "<" symbol to meet its two neighbours in the same cell of T; consequently, depending on which neighbour is met first, there are two alternatives, as depicted in the following examples:

(1) the "<" symbol in 5 meets it right neighbour (coming from 8) in 26, but has to go beyond that to 29 in order to meet its left neighbour (coming from 2)

(2) the "<" symbol in 8 has to go up to 31 to meet its right neighbour (coming from 11), but meets its left neighbour (coming from 5) earlier, in 26. (The three special cases: the leftmost "<" in L, the rightmost "<" in L, and the only "<" in L must move up to the root-node of T to find out about their special status.)

The diagram in Fig. 7 shows that connections between nodes of T are in the form of two, identical, independent links. This is possible because the mechanism with which an active area communicates with the outside world (in order to bring in microprograms, perform I/O, etc.) is left unspecified in this paper, hence links between the top of an active area and the root of T are not shown in the diagram. The necessity and sufficiency of two links between nodes of T is a consequence of the following considerations:

(1) Since T is a binary tree, each internal node of T that is not partitioned has three links connecting it with other nodes. If a node gets partitioned, each part separated must have at least one independent link connecting it with some part of another node (a part with no outside connections is useless). This implies that there must be at least two independent links between nodes if the node is partitioned.

(2) We can demonstrate with the help of Fig. 8 that three independent links are never needed between two nodes of T. The three hypothetical links -- X, Y, and Z in Fig. 8 -- belong to areas associated with three disjoint, not necessarily adjacent, segments of L. About each of these segments the following statement can be made: "the leftmost symbol of the segment is the "<" symbol." All possible combinations of the truth values for the three statements are listed in Fig. 8 , "<" and "--" representing truth and

falsity, respectively. Since an "<" symbol always separates an area for itself, "--" in Fig. 8 means that the segment of L to which it applies contains no application symbols, whereas "<" means that the segment contains exactly one "<" symbol. In six out of the eight cases two, or all three, links can be merged, as indicated by the groupings of Fig. 8. This is because the segments of L to which they apply are adjacent, and contain no, or only one "<" symbol, hence the links belong to the same area.

In case 4 of Fig. 8 the "<" symbol arriving in link Y has already passed the top of its area, for it has already met its right neighbour (either the "<" symbol in Z, or another "<" symbol in between). We conclude that link Y is superfluous, provided the information about the "<" symbol in Y can be propagated through link X, so that it could meet its left neighbour somewhere higher up in the tree. (Our implementation will do exactly that, as the data structures of Fig. 9 reveal.)

Finally, in case 8 of Fig. 8 the "<" symbol arriving in link Y has already met both its neighbours, so it does not need a separate link any more.

(3) Since all cells of T must be identical, we choose to have exactly two links between any two nodes of T that are connected.

Having examined the global aspects of locating reducible applications, and derived from it the need for two identical links between the nodes of T, we can now describe the mechanism to locate reducible applications, which amounts to specifying what an arbitrary cell of T does during the relevant upward cycle. We shall use Fig. 9 for this purpose, which shows all the possible ways to partition a node of T.

On the left of Fig. 9 an arbitrary node of T is shown with its six

identical links. (These links are symbolic representations of two-way communication channels whose organization is left unspecified here.) Since we will be concerned with operation during an upward cycle, we shall refer to the four links coming from the lower levels of T as inputs, and to the two links leading to the higher levels of T as outputs.

There are three four-field data structures associated with the inputs and outputs of the node. The data structures associated with the inputs contain all the information needed (1) to partition the node, (2) to determine if the area whose top has just been found is active (i.e. the corresponding application is reducible), (3) to determine which microprogram to bring in. A field labelled with "<" holds the level number of an application symbol. In a field labelled with "M" two kinds of information is carried: (1) the smallest level number of any symbol in a segment of L that contains no "<" symbol -- that is needed to find reducible applications, (2) information (four bits and a symbol of the program text) needed to determine what a reducible application is supposed to do (i.e. which microprogram to bring in).

There are four different formats for our data structure:

(1) (M, --, --, --)

(2) (--, --, <, M)

(3) (M, <, <, M)

(4) (<, --, <, M)

If both links belong to the same area, either (1) or (2) is used. (In the diagram we connect both ends of the two links to indicate that one of them is superfluous, as, for example, between cells 21 and 27 of Fig. 7.) If the two links belong to different areas, then either (3) or (4) is used. In

these two cases, the left and right halves of the data structure are associated with the left and right links, respectively.

The interpretation of the four data structures can be given in terms of the segments of L with which they are associated.

(M, --, --, --) is associated with a segment of L containing no "<" symbols (as, for example, the right input of cell 25 of Fig. 7).

(--, --, <, M) is associated with a segment of L with a single "<" in it (e.g. the left input of cell 26 in Fig. 7). The "<" symbol in this data structure has not yet found either of its neighbours.

(M, <, <, M) is associated with two, not necessarily adjacent, segments of L, neither of which has been fully located so far (see, for example, the output of cell 27 in Fig. 7). The left(right) half of the data structure is trying to locate the left(right) end of its segment, i.e. the "<" symbol in the left(right) half of the data structure is looking for its left(right) neighbour. It should be noted that between the two "<" symbols contained in this data structure there may be many others in L, but since all of them have already found both their neighbours, they do not have to be propagated further up in T.

(<, --, <, M) is associated with two, not necessarily adjacent, segments of L, such that the one referred to by the left side of the data structure has been fully located, whereas the other has not (see, for example, the right input to cell 30 in Fig. 7). Again, the right half of the data structure is trying to locate the right end of its segment, i.e. the "<" symbol in the right half is looking for its right neighbour. It is important to note that the link on the left is above the top of its area, hence it will not be used for processing. The sole use of this link is to get its "<"

symbol to the left neighbour of that symbol. (This format is decidedly asymmetrical, because the relation of segments and "<" symbols is asymmetrical: we always try to locate a segment with "<" on its left. We have already explained under case 4 of Fig. 8 why we do not need the reverse of this format.)

Since either one of the left and right inputs may be associated with any of these four formats, there are altogether 16 input combinations. Fig. 9 specifies how the formats of the two input data structures determine the partitioning of the node, and the format of the output data structure. With the help of the preceding explanations the diagrams should be easy to understand.

Fig. 9 shows that any node may be required to do any of the following types of processing activities using these data structures:

(1) route fields from input to output (as in Fig. 9.b)

(2) combine two "M" fields (as in Fig. 9.a) by taking the minimum of the two level numbers, and by combining the rest in the obvious manner (information about the three leftmost symbols should be kept)

(3) determine if the area is active (i.e. the application is reducible) whenever neither of the input data structures is (M, --, --, --) (as in Fig. 9.f)

(4) determine what operation is to be performed whenever the top of an active area is found.

At this point we are in the position to estimate the increase in complexity of the cell of T owing to the need to share it among different areas. Our conclusions are:

(1) <u>the processing capabilities of a cell need not be duplicated</u> (only in cases a), c), d), and e) of Fig. 9 does the node receive two

different inputs from the same area, and to handle these the original processing capabilities will be sufficient

(2) the original routing capabilities must be duplicated, since there are two links between nodes of T

(3) additional logic must be included to control partitioning.

Notice that we were able to reach these conclusions without ever having committed ourselves to many details of implementation.

There is one more processing activity we should mention in this section, and that is finding the right ends of reducible applications. So far, by locating a reducible application we meant finding its left end, i.e. the root-node of the corresponding subtree, but because we want to do syntax checking by raising an error condition whenever a symbol in the reducible application does not receive any microinstructions, we must find its right end too, so that we could mark every cell belonging to it.

All this can be done during the downward cycle following the partitioning of the nodes of T, by using the data structures described in Fig. 9. Beginning from the top of the active area, and moving downwards in the tree, we try to locate the leftmost symbol in the segment whose level number is smaller than or equal to the level number of the "<" symbol, and in the process separate from the active area all the subtrees that are not required on the right. Fig. 10 shows an example for this: the upward cycle located a reducible application in the segment "<AB(CD", and the process just described separates from it "D", then "C", and finally "(".

4.6.2 DISTRIBUTING MICROINSTRUCTIONS

In this section we show that a small amount of suitably chosen

information stored in every node of T enables us to distribute the microinstructions to every occupied cell in a reducible application. (We shall, however, describe algorithms neither to set up this "data structure", nor to distribute the microinstructions, because both are reasonably straightforward.)

The whole microprogram is propagated from the outside (i.e. from the root-node of T) down to the root-node of the active area of the reducible application in question by a mechanism that we leave here unspecified. Once the microprogram enters the active area, however, the individual microinstructions must be routed carefully, so that each one of them will arrive only at those cells to which it applies.

An algorithm to distribute microinstructions must answer the following question: in a given node of T, should a given microinstruction be sent into the left subtree, the right subtree, or both? This question will be answered by using the destination expression of the microinstruction (plus the information its position in the microprogram carries), and a data structure called the directory, specifying which segment of the reducible application is in the left and which is in the right subtree of the node in question.

The directory for each node consists of two identical five-field data structures, associated with the left and right subtrees. The interpretation of the five fields is explained with the help of Fig. 11. In the sequence of level numbers in a segment of a reducible application LN3 is the number of 3's before any 2 or 1 is encountered from left to right; LN2 is the number of 2's before any 1 is encountered; N1 is the total number of 1's in the segment; RN2 is the number of 2's after the last 1, and RN3 is the number

of 3's after the last 2. If N1=0, then obviously LN2=RN2=N2, and if, in addition, N2=0, then LN3=RN3=N3, so we have two special cases: (LN3,N2,0,N2,RN3) and (N3,0,0,0,N3).

With the help of an example (Fig. 12) it is easy to see that this directory contains exactly the information needed to route the micro-instructions to their destination. The only limitation of the scheme is that it "sees" the operator and the operand of the reducible application only up to nesting level 3, but it can obviously be extended by adding fields to it.

4.6.3 LOCATING FREE VARIABLES DURING SUBSTITUTION

Whenever the reducible application is in the form $\langle(\lambda v\ \underline{e}),\underline{f}\rangle$, $\underline{f}$ is to be substituted into $\underline{e}$ for every free occurrence of v. Since $\underline{e}$ might contain not only free, but also bound occurrences of v, the processor must be able to tell them apart, which is equivalent to locating all lambda expressions internal to $\underline{e}$ that have v as their bound variable.

The implementation of this will be efficient, and easy to understand: the microprogram for substitution will be written to treat all occurrences of v in $\underline{e}$ equally; on the other hand, a mechanism we are about to outline will "seal off" all lambda expressions with bound variable v and internal to $\underline{e}$, so the microprogram simply will not be able to access the bound occurrences of v, thereby guaranteeing correct operation. "Sealing off" these lambda expressions is quite similar to locating reducible applications: in both cases we are analyzing tree (or nested expressions), and trying to locate certain subtrees. The only important difference is that there we had to locate innermost applications, while here we have to locate outermost lambda expressions. As a result, the data structure to be

stored in the nodes of T is somewhat simpler: it needs three fields instead of four.

The data structure (LN,N,RN) of any node can have one of two interpretations (in both cases we consider only lambda expressions internal to the operator):

(1) if there is at least one "λ_v" in the segment of L under the node, then

N is the level number of the rightmost outermost "λ_v" in the segment

LN is the smallest level number of any symbol (including other "λ_v"'s) on the left of the "λ_v" mentioned above

RN is the smallest level number of any symbol (including other "λ_v"'s) on the right of the "λ_v" mentioned above

(2) alternatively

N is undefined

LN is the smallest level number of any symbol in the left subtree

RN is the smallest level number of any symbol in the right subtree.

An example in Fig. 13 illustrates the workings of this mechanism, and again we omit description of algorithms to set up the data structures, and to "seal off" subtrees, since they are reasonably straightforward.

4.6.4 STORAGE MANAGEMENT

Storage management is an integral part of the operation of this processor. Since the program text is represented as a linear string of symbols in L, most reducible applications lead to either the contraction or expansion of the operand. Contraction is easily done, because empty cells do not interfere with program execution. Expansion, however, may necessitate the intervention of a global agent to make room for the expanded segments by

moving the contents of cells in L appropriately. Just before storage management is to begin, it is already known (from having partially executed the microprograms) where and how much room has to be made, and it is also known where the empty cells are. The problem is to determine how the occupied cells of L will have to be moved to make the necessary room. (For the sake of brevity we shall talk about moving cells, instead of moving the contents of cells.)

The algorithm to determine this uses the following initial information: every empty cell in L is marked by P = 0 and N = -1, and every occupied cell of L is marked by P ≥ 0 and N = 0. P is called an <u>insertion request</u>, meaning that P empty cells must be created immediately adjacent to the cell holding this flag. N = -1 means that an empty cell can satisfy an insertion request with P = 1.

The algorithm produces the following results (jointly referred to as a <u>solution</u>: (1) an integer S (called a <u>shift amount</u>) is stored into every occupied cell of L, meaning the cell should be moved by S positions to the left, or to the right, depending on the sign of S, (2) if an insertion request can be partially or completely satisfied, because it will be possible to create P - P' empty cells around the cell holding P, then P is reduced to P' (P ≥ P' ≥ 0).

We shall always assume that the shift amounts are determined in a consistent fashion, so that the simultaneous shifting of the occupied cells in L -- by one position at a time -- will change only the distribution of the empty cells in L, but will leave the positions of all occupied cells the same relative to each other.

If L contains as many empty cells as requests for insertions, then

the job of the storage management algorithm is unambiguously specified: it has to satisfy all insertion requests by moving occupied cells of L to absorb all empty cells.

If L contains fewer or more empty cells than requests for insertions, then a choice has to be made as to which insertion requests to satisfy, or, alternatively, which empty cells are to be used and which are to remain empty.

Every solution involves moving some of the occupied cells in L in different directions by a different number of positions. We assume that the actual shifting of these cells takes place simultaneously, one position at a time; consequently, the largest shift amount ~~stored into any cell of L~~ will determine the overall time needed to complete storage management.

The solution in which the largest shift amount is as small as possible will be called the minimal solution.

Another desirable property of a solution could be that the empty cells that remain after all insertion requests have been satisfied are distributed evenly in L. Fig. 14 shows why this is desirable. On the diagram the horizontal lines represent L, the incoming arrows and their integer labels represent the positions and amounts of insertion requests, outgoing arrows represent blocks of empty cells. In the first case we need 500 steps, in the second only 50 steps to satisfy all insertion requests.

We shall describe two algorithms for storage management:

(1) the primary algorithm -- in some cases, this algorithm does not produce the minimal solution, but we conjecture that the maximum shift amount produced by it is at most double of that produced by the minimal algorithm

(2) the secondary algorithm -- this algorithm ~~computes~~ shift amounts ~~to~~ that distribute empty cells evenly.

Both algorithms are implemented in four parts, using a four-field data structure (BL,P,N,BR) stored into every cell of T and L. This data structure summarizes information about the segment of L under the cell in question. The fields of the data structure hold signed integers with the following interpretations:

P is the sum of the positive flags in the segment ($P \geq 0$),

N is the sum of the negative flags in the segment ($N \leq 0$),

BL is "boundary condition on the left" -- the number of cells to be moved into the segment on the left, (BL may be either positive or negative),

BR is "boundary condition on the right" -- the number of cells to be moved into the segment on the right (BR may be either positive or negative).

To describe these algorithms the following notation is used: if an arbitrary node of T is associated with (BL,P,N,BR), then its left and right sons (cells of T or L) are associated with (BL1,P1,N1,BR1) and (BL2,P2,N2,BR2), respectively.

The description of the primary storage management algorithm is as follows:

(1) Starting with the P and N fields of all cells of L, information flows upward and the P and N fields of the tree nodes are computed: for each node of T, P = P1 + P2 and N = N1 + N2.

(2) At the root node of T, BL = BR = 0 (no cells are moved across the endpoints of the memory, since it is not circular).

(3) Starting at the root node of T, information flows downward and the BL and BR fields of all cells of T and L are computed, according to the following rules:

BL1 = BL, BR2 = BR (these are boundary conditions already given)

BL2 = -BR1 (follows from the conventions for signs)

BR1 is specified in the table of Fig. 15 with S1 = BL1 + P1 + N1 and S2 = P2 + N2 + BR2.

The table shows that cells are moved across segment boundaries only if absolutely necessary: if sign (S1) = sign (S2), no movement takes place between the two segments (either both of them have enough empty cells, or both have too many insertion requests), whereas if sign (S1) $\neq$ sign (S2), min (|S1|, |S2|) cells are moved across the boundary (the sign of the table entry properly specifying the direction of the movement).

(4) From (BL,P,N,BR) obtained for all cells of L the results are computed according to Table IV.

The secondary storage management algorithm differs from the primary one only in its third part, BR1 being specified by a different table (see Fig. 16).

This table has the same entries as the previous one, unless the two segments together have more empty cells than insertion requests, in which case $\lfloor 0.5 \times (S2 - S1) \rfloor$ cells are moved across the boundary (the sign again correctly indicates the direction of that movement). It can be shown that after storage management, the difference between the number of empty cells in any two segments associated with cells at the same level of the tree is at most one.

4.6.5 DATA MOVEMENT

Data items to be inserted, and their destinations (cells of L) both have to be marked somehow so that each data item can be copied into

the proper cell; we shall use positive integers as markers. In a particular reducible application the markers of data items must be unique, but the markers of cells need not be, since we may want to create several copies of a data item. The destinations do not have to be empty cells: two data items can be made to change place by properly assigning their markers, and the markers of the cells presently holding them.

In the storage management phase (discussed in the previous section), the movement of data items only involves shifting among the cells of L. At the time the storage management phase begins, it is known which data items are to move and where to. Because the destination of every data item had been determined by a single agent, in a consistent manner, the data items can be shifted without any reference to each other and cannot get into each other's way while moving. In contrast, at the time the data movement phase begins, it will have been determined which data items are to move (those marked by positive integers), and where to (all cells marked by the same integer); however, carrying out the move is much more complicated because the data items must be duplicated, interchanged, and rearranged. In order to make this possible, the data items are moved not along the linear array of cells L, but through the tree network.

We will use the following simple method of restricting the movement of data through the tree network: all data items that have to be moved will rise to the top of the active area, pass through the top node in some order, then descend to their destinations. The order in which data items are moved can depend on the value of the marker, such as with the following algorithm (which we will adopt): on their way up at every node of the tree the data items with larger markers yield to those with smaller

ones. But the movement can be quite complex; data items can get into each other's way, they may have to queue up at certain nodes, etc.

To avoid overwriting and thereby destroying information, reply signals (a technique well known to logic designers[6]), indicating if a register is full or empty, will be used to provide local control for the orderly upward movement of data in the tree network. To guide the downward movement of data, every tree node will hold two integers: the smallest and largest cell-markers in the segment of L under the tree node in question. When a data item on its way down arrived at an arbitrary tree node, the following algorithm will be used to decide whether it should be saved and sent down further. Let M be the marker of the data item and L and U be the lower and upper bounds of the cell-markers in the node in question. Then if $M \in [L, U]$, send the data item to the two son nodes. Note that two copies of the data item will have been created, although neither may be used. (There must be some mechanism to prevent data items from going outside the area of the reducible application; this is the role of the "walls" mentioned in section 4.6.1.)

Owing to the simplicity of the operation, we can easily express the exact amount of time needed to complete it as $t = 2h + n$ time units, where h is the height of the active area, n is the number of data items marked originally, and a time unit could be as small as two clock-cycles. (Explanation of the formula: the first item to pass through the top takes h time units to rise to it, n items pass through the top, and the last item takes h time units to descend to its destination.)

Input and output can use the same mechanism (the second or the first half of it, respectively) with the time requirement $t = H + n$, where

H is the height of T.

4.7 OVERALL ORGANIZATION OF THE PROCESSOR

In section 4.3 we have observed that the processing requirements of a single reducible application naturally divide into three phases. During the first and third phases the reducible application does not interfere with the program text around it, whereas during the second one it most likely will.

The overall organization of the processor is determined by the way the three phases of different reducible applications are coordinated with each other. All possible organizations are between the following two extremes:

(1) the most synchronous organization (simplest to control), where all reducible applications go through the same phase at the same time, and all wait for the slowest one to finish before proceeding to the next phase; if execution times vary a great deal, this might waste a lot of time

(2) the most asynchronous organization (hardest to control), where all reducible applications proceed independently of each other.

Although one might expect the most asynchronous organization to be the most efficient one from the point of view of execution times, we can easily show that this is not necessarily the case. To that end we note that there is a very important difference between phase 2 and the other two phases: during phase 2 parts of the reducible application are shifted in L without reference to the tree network above, while during phases 1 and 3 the tree network above the reducible application contains information necessary for the correct execution of these phases, hence movement of the reducible application during these phases cannot occur. As a result, in the most asynchronous organization it can easily happen that a reducible application

enters its phase 2 while being surrounded by others in their "unmovable" phases, and as a consequence, it might be held up a long time waiting for enough empty cells to become available around it.

For this reason, the most promising organization appears to be a compromise, initiating and terminating storage management for all reducible applications at the same time, and then letting the different applications proceed through their phases 3 and 1 at their own pace. With the help of Fig. 17 we mention three definite advantages of this simple organization:

(1) The storage management algorithm can be used to best advantage, since it has the whole "memory" (L) available. One can use either the primary or the secondary storage management algorithm, or the following combination of the two depicted in Fig. 17: determine what the largest primary shift amount is, and carry on with the secondary storage management algorithm for that amount of time (i.e. do everything that must be done, and in the time needed for that do everything that can be done).

(2) Since most primitives do not need any data movement, and the ones needing data movement need widely differing amounts of time for it, this solution gives a good overlap between them.

(3) Storage management can be initiated so as to interfere with the rest of execution as little as possible. This will be done according to some rule we have not specified, e.g. as a function of the number of reducible applications waiting for storage management.

The overall organization just described can be implemented by endowing every cell of T and L with the same finite-state control, and letting the state of any cell change when the clock pulses that move up and down the levels of T reach the cell. The state diagram of this finite-state control is shown in Fig. 18, and in what follows we shall tie the whole operation of

the processor to the states of this diagram.

Processing is taking place in states 1 through 18, while the others -- for one reason or another -- are just marking time. The three phases of processing (see section 4.3) correspond to the following group of states:

phase 1 1 through 6

phase 2 7 through 10

phase 3 11 through 18.

The more important processing activities correspond to individual states as follows (the arrows indicate if the state in question is entered during an upward or a downward cycle):

1 (↑) partition nodes of T (4.6.1)
locate left ends (roots) of reducible applications, determine their absolute level numbers (4.2) and their operators (3, 4.6.1)

2 (↓) store relative level numbers (4.5) into L
recognize right ends of reducible applications (4.6.1)
set up directories (4.6.2)
start bringing in microprograms

3 (↑) recognize "⊥" in reducible applications
find left ends (roots) of internal lambda expressions (4.6.3)
continue bringing microprograms in

4 (↓) complete "sealing off" internal lambda expressions (4.6.3)
complete the process of bringing microprograms in

5 (↑) detect syntax errors (4.5)
carry out arithmetic and Boolean processing
determine length of segments to be moved

6	(↓)	store insertion requests into L (4.6.4) modify and erase level numbers and symbols (only if there are no insertion requests, hence processing ends here)
7	(↑) and 8 (↓)	compute shift amounts (4.6.4) adjust insertion requests (4.6.4)
9 10	~~(↑) and 10~~ (↑)	find largest primary shift amount to know when to ~~stop (4.6.4)~~ ~~complete storage management~~
11	(↓)	since storage management is complete for this reducible application, mark all cells just inserted with consecutive integers to guide data movement (4.6.5)
12 through 17		move data through T to L, either from the outside (e.g. I/O, defined operators) or from L (e.g. DISTL)
14	(↑)	detect the end of data movement (4.6.5)
18	(↓)	adjust list of free variables (4.2) change formal applications into applications modify and erase level numbers and symbols, thereby ending processing
19 through 23		cells in these states are just marking time (they are either in the inactive area (4.6.1), or in a reducible application which is held up because storage management could not satisfy all insertion requests).

We conclude this section with two comments on the state diagram of Fig. 18:

(1) the rather rigidly synchronized operation of the whole processor will be implemented by making sure that if we look at all cells of T and L at the end of any cycle, their states will be in the same column of the state diagram (e.g. 17, 11, 2, and 19); this rule guarantees, for example, that

storage management will not begin unless all data movement through T had been completed, because there is no transition from 15 to 7

(2) the content of L is always a well-formed expression, with the exception of states 12 through 17, during which data is moved through T and L is in disarray. (In states 6 and 18 we can assume that changing one expression into another is instantaneous.)

5. PERFORMANCE

Having outlined the structure and operation of the processor, we are in a good position to analyze some performance characteristics of it, chiefly related to execution times. We begin by recalling some of our original design objectives: the execution of a single reducible application should be at least competitive with that on a von Neumann computer, and all reducible applications should be able to proceed independently.

Assuming that L contains a single reducible application only, we shall try to estimate the time requirements of the three phases of its execution. (In all expressions that follow, the time unit is going to be a clock cycle.)

Phase 1. The time required to complete this phase is $6(\ell g_2 N)T_1$ (see Fig. 18) where N is the total number of cells in L (for N a power of 2), and T_1 is the time taken on any level of the tree network to do the required processing and pass the result to the next - higher or lower - level of T. Since the processing in all these cycles involves only comparing integers for magnitude, comparing symbols for equality, and routing, T_1 can be as small as two clock cycles.

Phase 2. The preparation for storage management (states 7 and 8 of Fig. 18) takes $2(\ell g_2 N)T_2$ time, where T_2 -- the time required to execute the algorithm of section 4.6.4 on any level of the tree network -- is

approximately the time to add two integers.

The duration of the actual moving of information in L is determined by the largest ~~primary~~ shift amount (LPS), but cannot be less than $\ell g_2 N$, which is the time needed to determine LPS by propagating this information up to the root of T (state ~~9~~ 10 in Fig. 18). Combining these two results we obtain the formula

$$2(\ell g_2 N)T_2 + \max(\ell g_2 N, LPS) \rightarrow (\ell g_2 N)T_2 + LS$$

for the time needed to complete storage management.

Relating LPS to the number of symbols to be moved by the reducible application can be quite complicated. We give only a simple example of such an analysis by considering the case of the primitive DISTL (section 4.3), with two different initial distributions of the cells of the reducible application in L. If the reducible application initially contains no empty cells, LPS is going to be proportional to n(length of x), while if the reducible application contains all the empty cells needed, they are equally distributed in the reducible application, and the length of all the y's is the same, then LPS is proportional to the length of x - an n-fold improvement. This brief example was included to draw attention to two important prerequisites for efficient storage management: having enough empty cells inside and around the reducible application (empty cells should be considered as a resource in this processor, since their availability will generally speed up processing), and having the program text equally distributed.

Phase 3. As mentioned already in section 4.6.5, the time needed to complete data movement is $(2\ell g_2 N + n)T_1$, where n is the number of symbols passing through the top of the active area (for DISTL this would be the length of x). Owing to the synchronized nature of the processor, this time

is made available in units of $6(\ell g_2 N)T_1$, and taking into account cycles 1 and 11 that are used to rebuild the active areas (partition the nodes of T again, etc.) the expression becomes $6(\ell g_2 N)\lceil(4\ell g_2 N + n)/(6\ell g_2 N)\rceil T_1$, which can be replaced by the much simpler $(10\ell g_2 N + n)T_1$ serving as a good upper bound. (Fortunately, T_1 is enough time to do the processing here, which is basically the comparison of two integers for magnitude.)

Collecting our results we arrive at the following expression for the time required to execute a single, stand-alone reducible application:

if there is no data movement, $6(\ell g_2 N)T_1$, otherwise

~~$\ell g_2 N(16T_1 + 2T_2) + \max(\ell g_2 N, LPS) + nT_1$.~~ $\ell g_2 N(17T_1 + 2T_2) + L\ldots + nT_1$

The formula shows that all reductions without the need to move data are executed in the same, constant amount of time. If there is a need to move data, the execution time has three components, the first one of which is a constant, determined by the size of the processor(N), and by T_1 and T_2 whose values will be determined by design details of the processor which we have not treated here. The second component, for a given number of insertion requests, may vary considerably depending on the number and distribution of empty cells in L, while the third component depends on the number of symbols to be moved through T.

The following comments should lead to the conclusion that we have fulfilled the first design objective:

(1) The processor is able to implement many "big" primitives without any data movement; examples are "associative referencing" and taking the minimum, maximum, sum, and product of an arbitrary number of elements.

(2) Because of the synchronous operation of the processor, the "smallest" primitives (e.g. TAIL, AND) will use up the same amount of time, which, however, is still not too bad compared with at least two full memory cycles

needed on a von Neumann computer.

(3) If there are enough empty cells in and around the reducible application, the time needed to execute a reducible application with data movement is a linear function of the number of symbols to be moved, and the factor of proportionality for this linear bound is two clock cycles, as opposed to two full memory cycles in a von Neumann computer.

The much more important second design objective has obviously been reached, and the ability of this processor to reduce simultaneously an unbounded number of applications (limited only by N) is the major source of its power.

From the many alternatives we mention four different ways of taking advantage of the parallel processing capabilities of this processor:

(1) Often a single algorithm lends itself easily to parallel processing. For example, in a backtracking program all the alternatives can be simultaneously pursued, and we can say that such an algorithm literally allows us to trade storage for time.

(2) Many frequently occurring problems related to matrices, strings, etc. can be solved very efficiently on this processor (e.g. matrix multiply with effective space-time product of $O(n^3)$ [7]). The performance of such algorithms requires mathematical analysis, and results of this nature will be reported elsewhere.

(3) When writing any program for this processor, the programmer should try to introduce as much parallelism into his algorithm as he can, because such programs are immediately and automatically rewarded by the processor with rapid execution.

(4) Since reducible applications are independent of each other, this

processor is eminently capable of executing an arbitrary number of independent user programs without needing any extra machinery (e.g. we can make them elements of a "top-level" sequence).

Although all the reducible applications can proceed simultaneously, they do interact during storage management by competing for the same resource (the empty cells of L), and the more time is spent on storage management (possibly unnecessarily because of incorrect organization) the more performance deteriorates.

This is obviously a problem needing quantitative analysis, and most likely simulation. Assuming that our storage management algorithm (4.6.4) keeps the program text more or less equally distributed, the most important parameter influencing performance is likely to be p = (number of occupied cells in L/total number of cells in L). There is likely to be a minimum value of p, below which performance is not influenced by it, guaranteeing that most of the time, most of the reducible applications have enough empty cells around them. Once such a value of p is determined some way, the processor can very easily monitor p, and make sure that it does not exceed a certain bound by possibly delaying the execution of certain reducible applications. The description of such "operating-system" type functions is, however, beyond the scope of this paper.

Although throughout the paper we stated several times that the processor communicates with the outside world via the root node of T, such a scheme cannot be implemented without impairing either the cellularity of T or the performance of the processor. The problem of efficient I/O can be solved in such a way that T can remain cellular, but its description requires the specification of many low-level details, and consequently it could not be included in this paper.

6. COST-EFFECTIVENESS

In this section we take a brief look at the price we have to pay for the increased performance of our processor in terms of hardware. We shall try to answer the question if the processor is worth the price, and when it is worth it, by comparing a processor as described here with a von Neumann computer under some simplifying assumptions.

Fig. 19 shows the amount of hardware needed by the two machines as the function of memory size N. In the diagram m represents the amount of hardware associated with one word of memory of the von Neumann computer (a single-processor, one-address machine), in particular the hardware actually to hold say a 32 bit word plus the proportionate amount of the selection circuitry. The other curve is based on the observation made in section 4.1, that the amount of hardware in our processor is a linear function of N, where N is taken to be a power of two, and ℓ and t are the amount of hardware in a single cell of L and T, respectively.

To be able to make a comparison we assume that ℓ, t, and m either refer to hardware built with the same technology, or they express production costs. The most important factor affecting the outcome of the comparison is the ratio $k = (\ell+t)/m$, especially for larger values of N.

Although in this paper we have not carried out a detailed logic design for the cells of L and T, nor even given exact encodings for symbols, micro-instructions and the like, nevertheless, we did give sufficiently detailed descriptions of the most important mechanisms in the processor to convince the reader that the cells of L and T are relatively simple.

We found that the cells of both L and T must have 23 states; the cells of L must be able to hold a single symbol and related structural information,

and must be able to decode and execute microinstructions; the cells of T must contain a handful of registers, must be able to perform certain routing functions and some simple processing (including arithmetic). The exact amount of hardware will strongly depend on the width of data paths between cells and on many other details of the logic design (which in turn will determine the values of T_1 and T_2), but it is quite safe to assume that k will be somewhere between say 20 and 200.

To contrast the performance of the two machines, we refer to Fig. 20, which is not a representation of exactly known quantitative relationships, but rather an informal, visual aid to illustrate the arguments that follow. The diagram shows the well-known fact that the performance of a von Neumann computer cannot be increased beyond a certain point by adding more memory, in sharp contrast to our processor whose performance grows linearly with N (doubling N allows us to run two copies of the same program in the same amount of time, etc.). The relative position of the two curves is affected by many factors, but we can argue that the break-even point is at a memory size where the reduction machine can - on the average - keep at least r reducible applications going simultaneously. (Assuming that the two machines work with the same clock rate, the conclusions of section 5 indicate that r is likely to be very small.)

And now we are in the position to state the main conclusion of this section: there is a critical memory size beyond which the reduction machine not only outperforms the von Neumann computer, but does so cost-effectively. Moreover, this critical memory size is such that it allows the reduction machine to execute - on the average - $k \cdot r$ reducible applications simultaneously.

Carrying out the full logic design, performance analysis, and simulation

of the processor will supply us with this critical size, but whatever it turns out to be, one thing is certain: the processor is destined to be a large-scale processor, since the farther its size exceeds the critical size, the more dominant its advantages will become.

It should be mentioned here that commercially available microprocessors could be programmed to act as cells of T and L (partitioning the cells of T can be replaced by time multiplexing, etc.), providing a somewhat slow, but relatively low-cost realization of the processor for experimental purposes.

7. SUMMARY

In this paper we have demonstrated that

(1) reduction languages can be executed efficiently on suitably organized hardware, and that

(2) it is possible to devise a computer architecture - in the form of a regular network of identical microprocessors - whose performance is exactly proportional to the amount of hardware built into it.

The first statement implies that to the long list of attractive properties of reduction languages an important new one has been added, leading to the conclusion that reduction languages should be seriously considered as user languages. Research is already going on in this area[2,3,8], but obviously there is room for a great deal more.

The second statement has to be qualified by saying that the processor executes a high-level programming language, and it responds to nothing else. The role of the reduction language is absolutely vital to the high performance achieved by the processor: the programmer, by expressing his algorithm in this language, organizes his computation to facilitate efficient execution.

Since the processor was designed above all to be able to move data

around efficiently, its satisfactory operation is not restricted to any particular data representation, or to any special class of problems, and we shall take this fact to imply that the architecture is a general purpose one[9].

High performance can be attributed to the computation being successfully decomposed both in time and in space: a large number of subcomputations (not necessarily identical) can proceed any time, and since they need to communicate with at most two others according to a simple pattern, they can be localized in space and implemented in a cellular network of processors.

There is massive parallelism both on and below the language level. On the language level there is unbounded parellelism[10] -- memory space permitting, all reducible applications can proceed simultaneously. This is made possible by the assignment of cells to individual symbols, and by the simple, linear representation of the program text, which in turn is maintained by constantly moving data around.

Efficient reduction of individual applications is made possible by the simultaneous execution of microprograms, and by all the internal mechanisms, which involve a great deal of low-level parallelism too.

All internal mechanisms have been defined in terms of the tree network, and they realize our original desire to exercise efficient global control over L. We say we have global control over all N cells of L, because we can influence all of them simultaneously, and each one differently. Thanks to T, we can do this efficiently, meaning in $O(\ell g_2 N)$ steps, and by using properly organized local activities only (no cell in the processor has access to anything but its immediate neighbours).

Besides being able to pass information between the leaves (L) and the root-node in $O(\ell g_2 N)$ time, the tree network has another unique ability:

partial results can be stored into all its nodes, the position of any node expressing many things about the partial result in it, such as which segment of L it applies to, what its relation to other partial results is, and so on.

Finally, we conclude that the last design objective, simplicity, can be considered to have been achieved. The many concurrent processing activities have been organized and synchronized to a sufficient degree so that

(1) it is relatively easy to explain and understand how the processor works,

(2) the major factors affecting efficient execution can be clearly identified (so that when it comes to simulation, we know exactly what the parameters should be), and

(3) bounds on the execution time and storage requirement of an algorithm can often be obtained analytically.

ACKNOWLEDGMENT

The author is indebted to Frederick P. Brooks, Jr. for several helpful discussions on the material presented here. Special thanks are due to Donald F. Stanat, whose careful reading of the manuscript led to many improvements in the presentation.

REFERENCES

(1) J. W. Backus, "Reduction languages and variable-free programming," IBM Research Report RJ1010, Yorktown Heights, NY, April 1972

(2) J. W. Backus, "Programming Language Semantics and Closed Applicative Languages," in Conference Record of ACM Symposium on Principles of Programming Languages, Boston, Mass., October 1973, pp. 71-86

(3) K. J. Berkling, "Reduction Languages for Reduction Machines," in Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston, Texas, January 1975, pp. 133-140

(4) K. J. Thurber, "Interconnection Networks - A Survey and Assessment," in Conference Proceedings of the AFIPS National Computer Conference, May 1974, pp. 909-919

(5) W. J. Meyers, "Linear Representation of Tree Structure: A Mathematical Theory of Parenthesis-Free Notation," in Proceedings of Third Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, May 1971, pp. 50-62

(6) S. H. Unger, Asynchronous Sequential Switching Circuits, New York: Wiley-Interscience, 1969, Chapter 6

(7) A. Koster, PhD Dissertation (in preparation), Department of Computer Science, University of North Carolina, Chapel Hill, N. C.

(8) M. Pozefsky, PhD Dissertation (in preparation), Department of Computer Science, University of North Carolina, Chapel Hill, N. C.

(9) K. J. Thurber and L. D. Wald, "Associative and Parallel Processors," ACM Computing Surveys, vol. 7, December 1975, pp. 215-255

(10) A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, New York:American Elsevier, 1974, p. 7

TABLE I.

Microprogram for the primitive TAIL.

Constituent of program text (symbol/relative level number)	Destination expression	Microinstructions
</0	(S/0):	erase;
TAIL/1	(S/1):	erase;
(/1	(E/1):	if S='(' then L:= L-1
		else S:= '⊥';
	(overlap S/1):	if valency = 1 then S:= 'ϕ';
x1/2	(E/2):	erase;
x2/2	(to last E/2):	;

TABLE II.

Microprogram for the primitive AND.

Constituent of program text (symbol/relative level number)	Destination expression	Microinstructions
</0	(S/0):	S:= result;
AND/1	(S/1):	erase;
(/1	(E/1):	if S='(' then erase else S:= '⊥';
x/2	(overlap S/2):	operand1:= S;
y/2	(S/2):	operand2:= S;

TABLE III.

Microprogram for the primitive DISTL.

Constituent of program text (symbol/relative level number)	Destination expression	Microinstructions
</0	(S/0):	erase;
DISTL/1	(S/1):	erase;
(/1	(S/1):	if S='(' then begin becomes string1; erase; end else S:= '⊥';
x/2	(E/2):	becomes string2; erase;
(/2	(S/2):	if S='(' then L:= L-2; else S:= '⊥';
y1/3	(to last E/3):	before insert (string1, string2); L:= L-1;

Table IV. (revised)

The results of the storage management algorithm

BL	P	N	BR	P'	comment
$\pm$ n	0	0	$\mp$ n	—	cell remains full
$\pm$ n	0	-1	$\mp$ n	—	cell remains empty
n+1	0	-1	-n	—	cell becomes full
-n	0	-1	n+1	—	cell becomes full
-n-1	0	0	n	—	cell becomes empty
n	0	0	-n-1	—	cell becomes empty
q	p	0	r	p+q+r	

where $n, p \geq 0, \quad p \geq p+q+r$

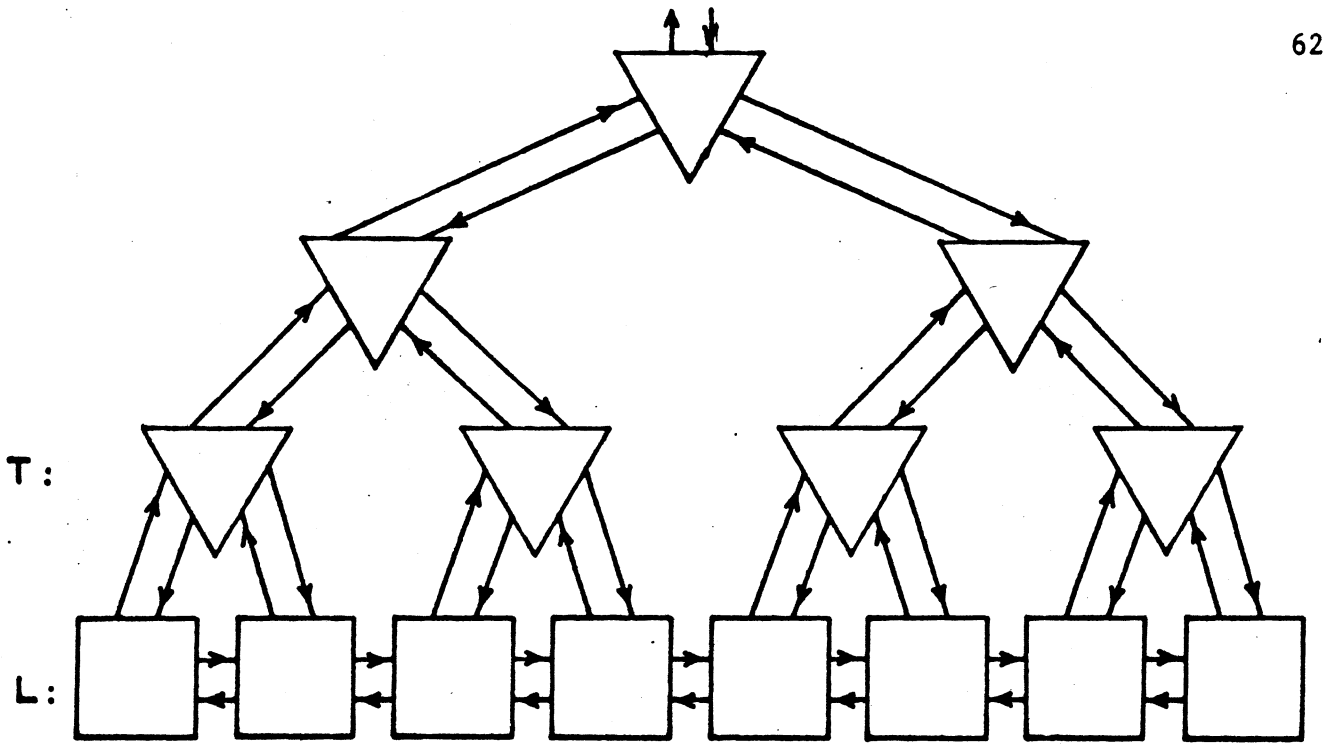
incorrect

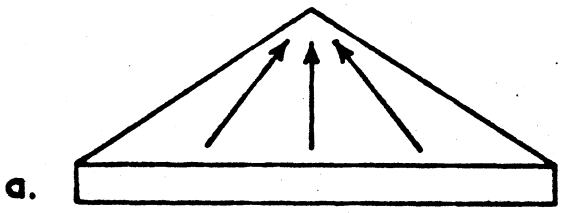
TABLE IV.

Computing the results of the storage management algorithm.

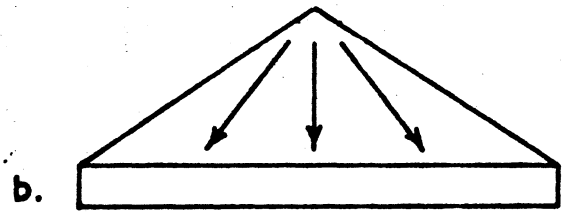
BL	P	N	BR	S	P'
0	0	0	0	0	-
n	0	0	-n	-n	-
-n	0	0	n	n	-
n+1	0	-1	-n	-	-
-n	0	-1	n+1	-	-
-q	p	0	r	r	p-q+r
r	p	0	-q	-r	p-q+r

where n, p, q, r $\geq$ 0, and p $\geq$ p - q + r $\geq$ 0

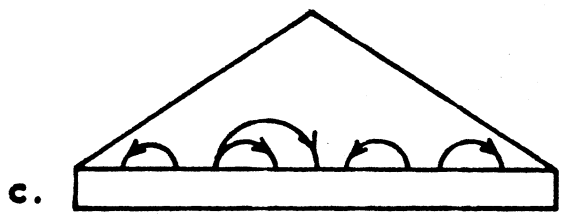




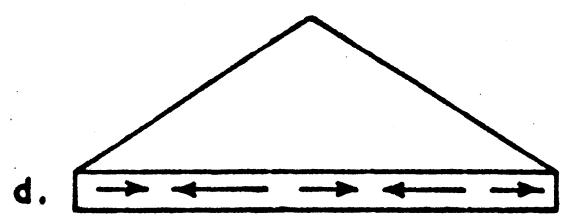
a. gathering information from L output



b. distributing information to L input



c. moving data through T

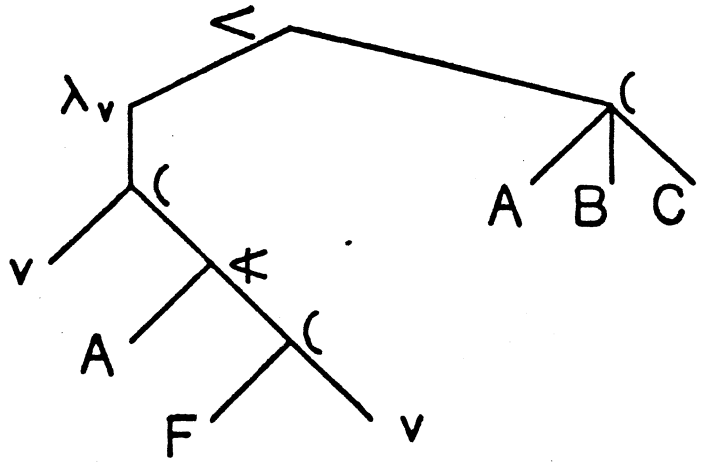


d. moving data in L

Original representation

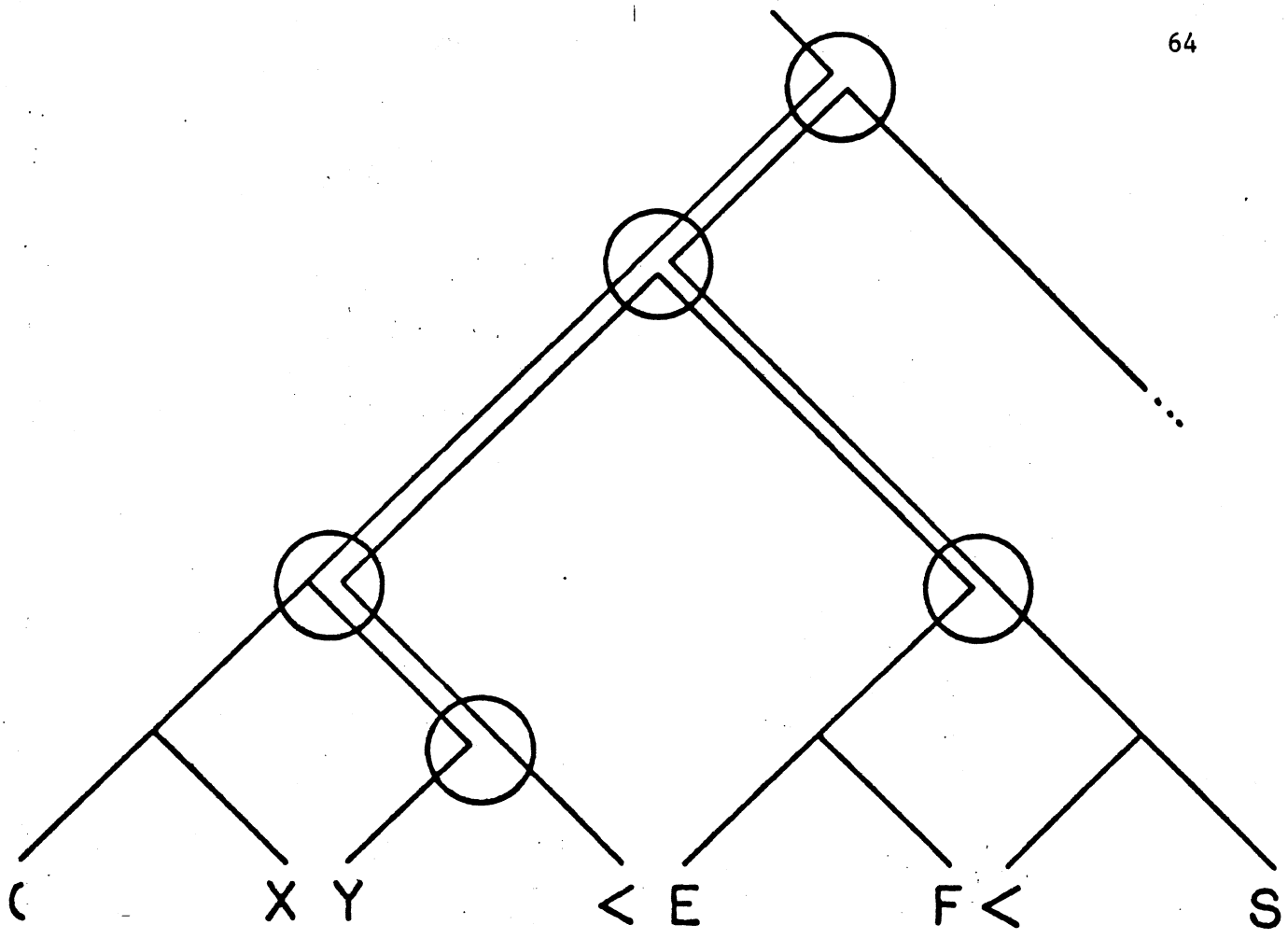
$$<(\lambda v(v\not<A(Fv)\not>))(ABC)>$$

Tree representation

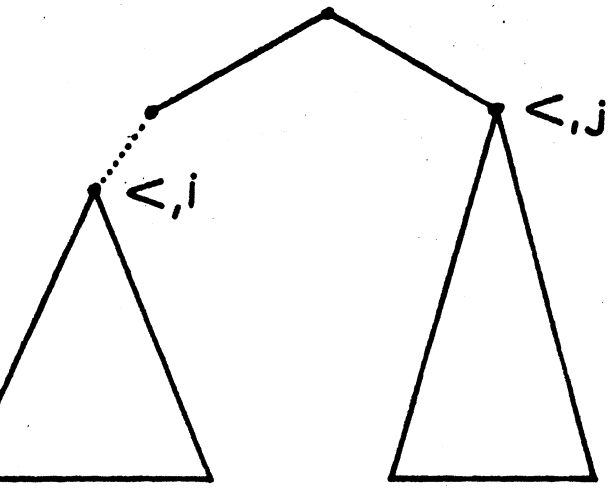


Internal representation

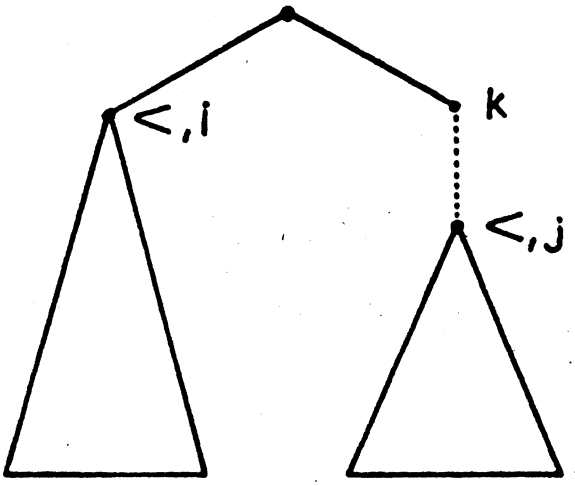
Symbol	$<$	λ_v	(	v	$\not<$	A	(	F	v	(	A	B	C
Nesting level	3	4	5	6	6	7	7	8	8	4	5	5	5



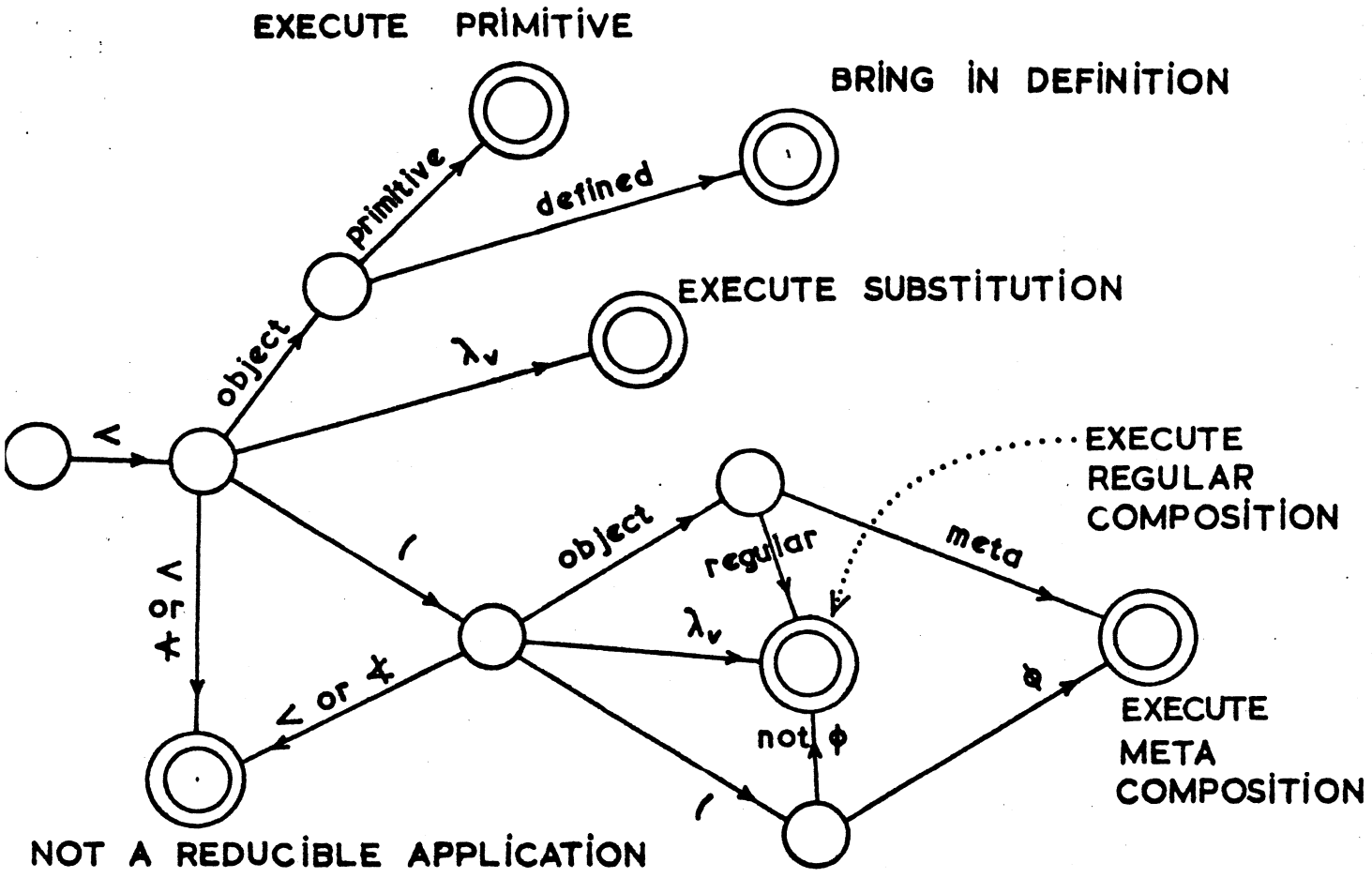
(4)



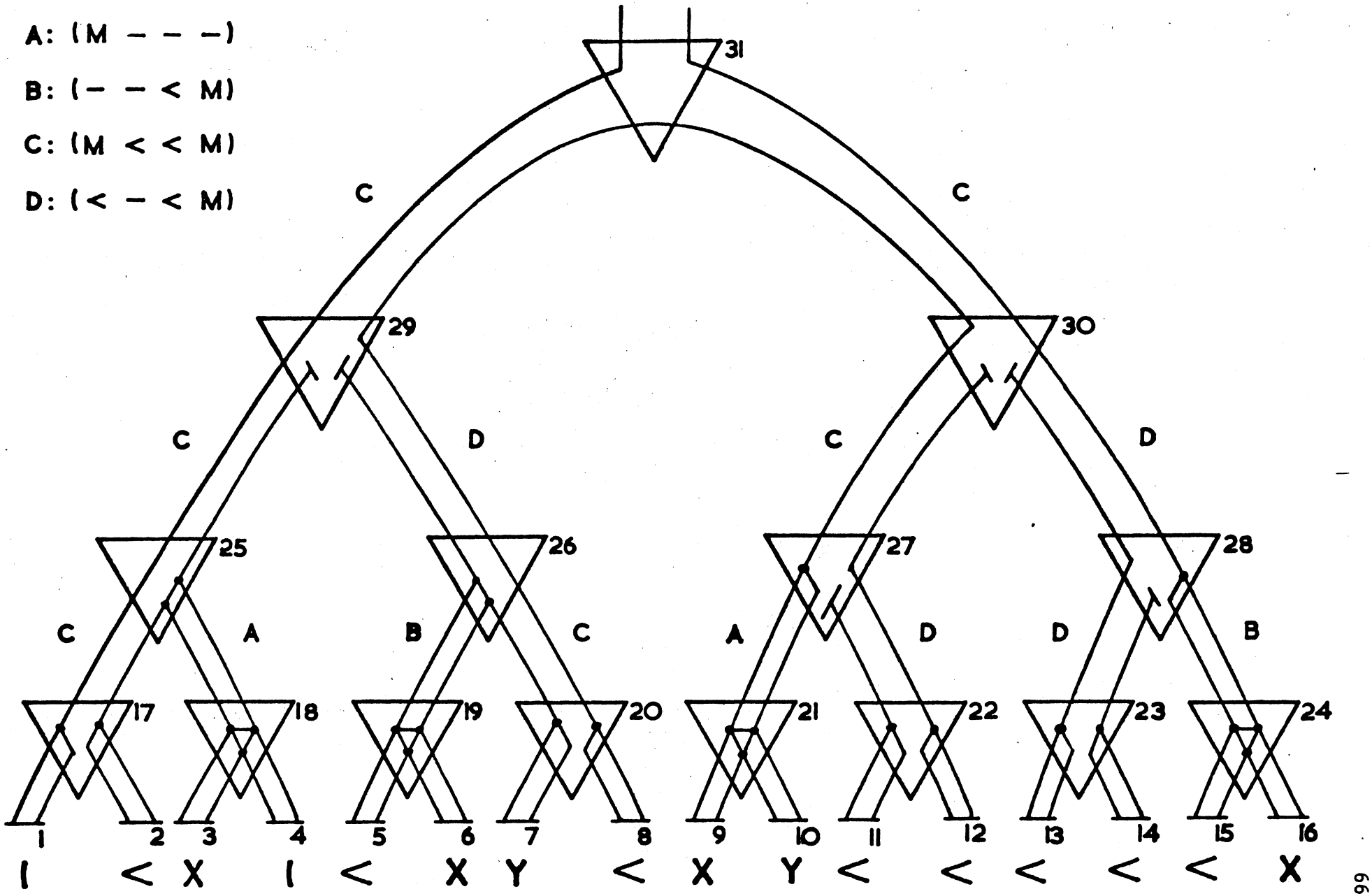
A. $i \geq j$

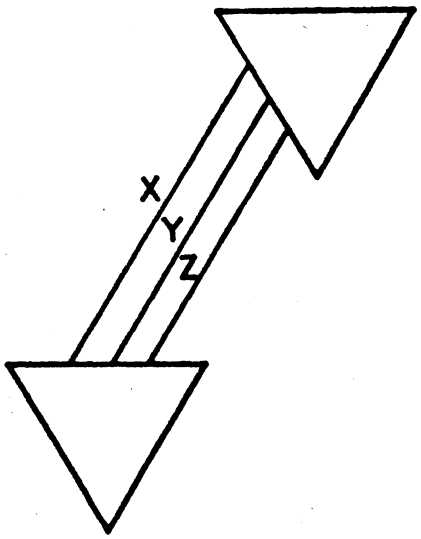


B. $i<j$ and $k=i$


EXECUTE PRIMITIVE
BRING IN DEFINITION
primitive
defined
EXECUTE SUBSTITUTION
object
λv
<
EXECUTE
REGULAR
COMPOSITION
object
regular
meta
λv
< or ↓
< or ↓
not ϕ
ϕ
EXECUTE
META
COMPOSITION
NOT A REDUCIBLE APPLICATION

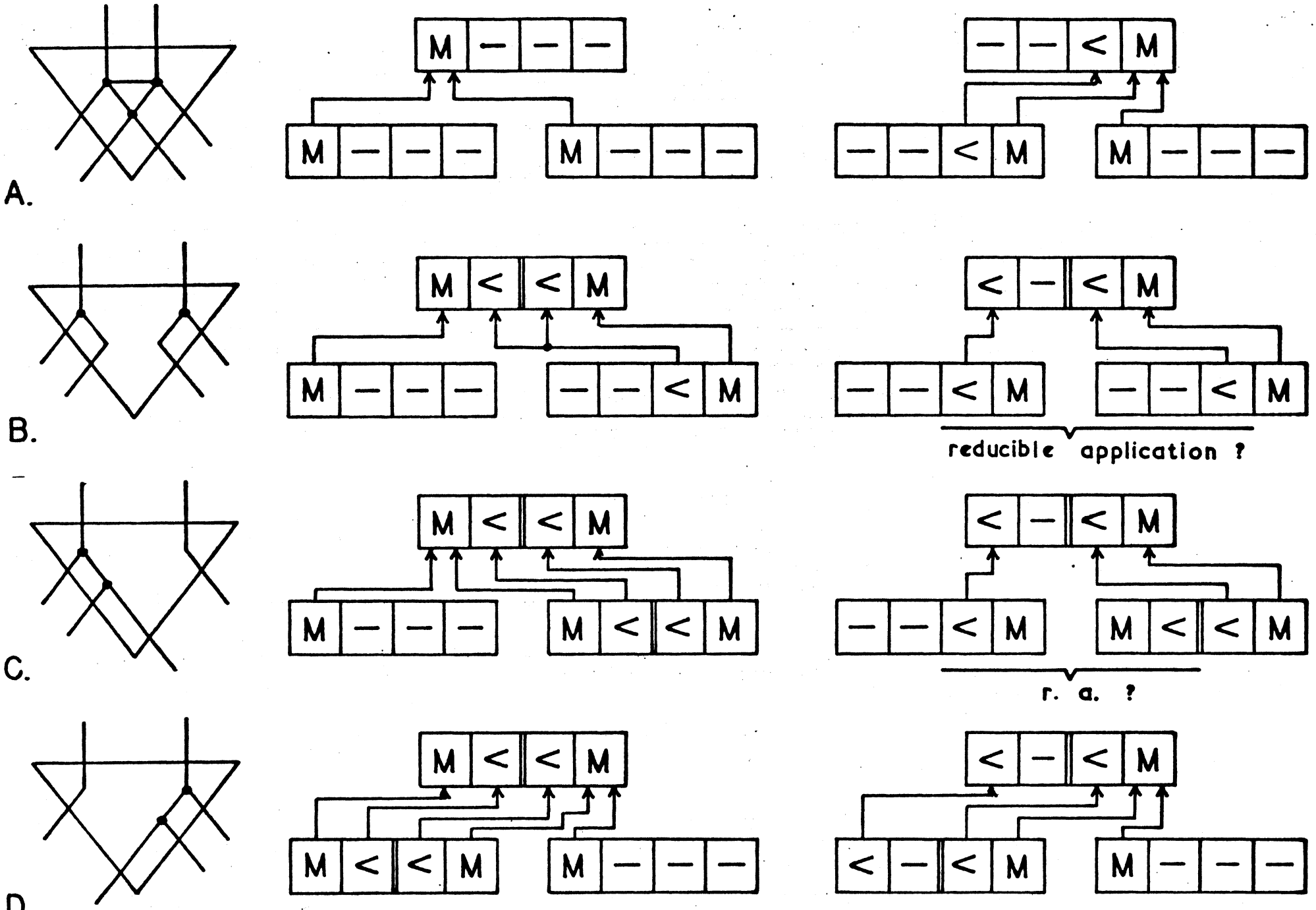
6


A: (M – – –)
B: (– – < M)
C: (M < < M)
D: (< – < M)
31
29
30
25
26
27
28
17
18
19
20
21
22
23
24
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
I < X I < X Y < X Y < < < < < X

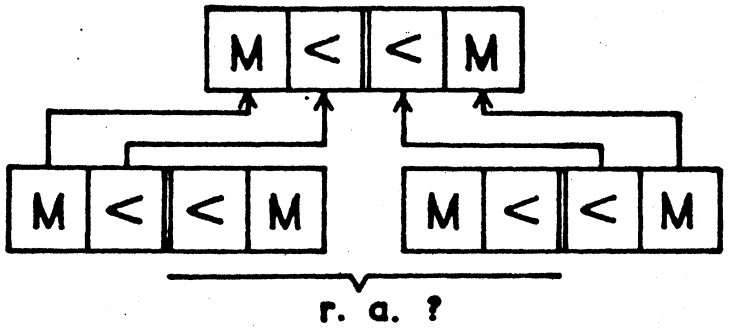


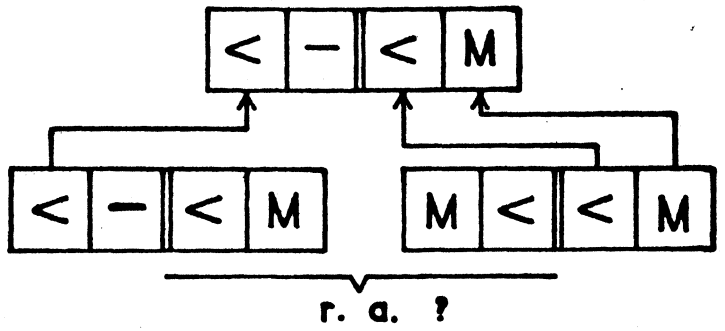
	X	Y	Z
1	—	—	—
2	—	—	<
3	—	<	—
4	—	<	<
5	<	—	—
6	<	—	<
7	<	<	—
8	<	<	<

(8)

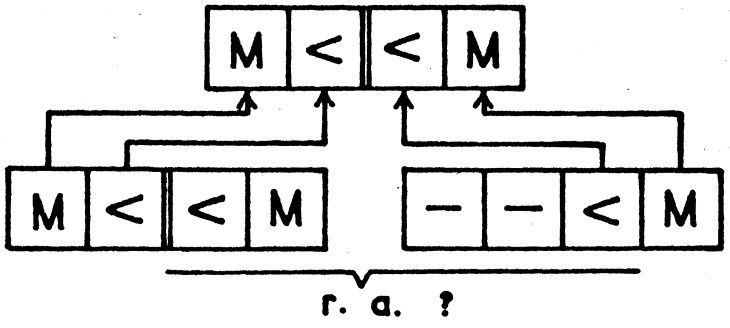

A.
M – – –
M – – –
M – – –
– – < M
– – < M
M – – –
B.
M < < M
M – – –
– – < M
< – < M
– – < M
– – < M
reducible application ?
C.
M < < M
M – – –
M < < M
< – < M
– – < M
M < < M
r. a. ?
D.
M < < M
M < < M
M – – –
< – < M
< – < M
M – – –

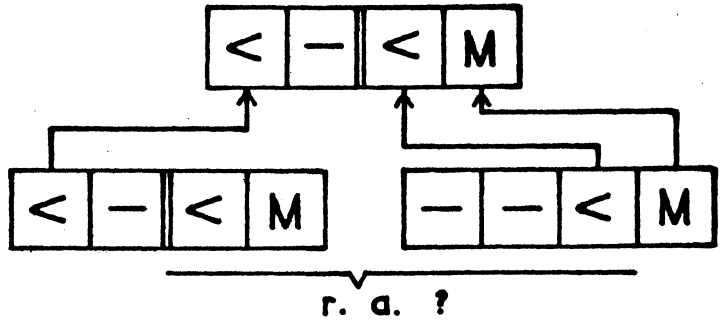
E.



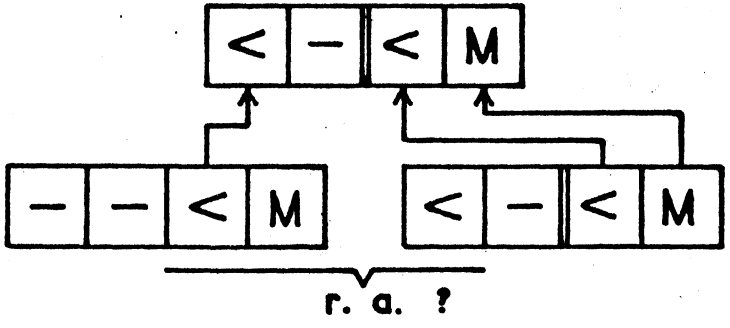


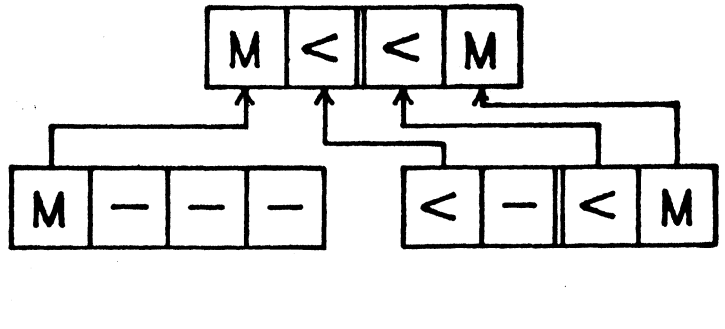
F.



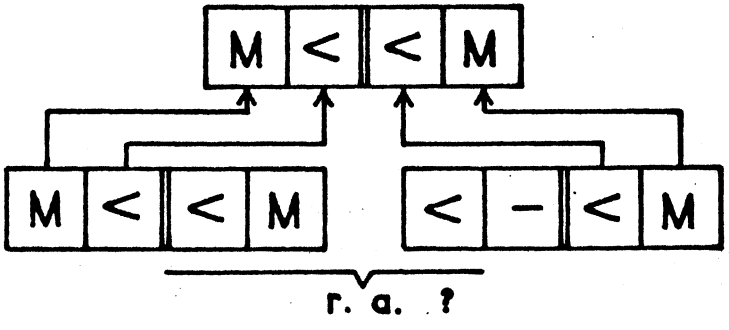


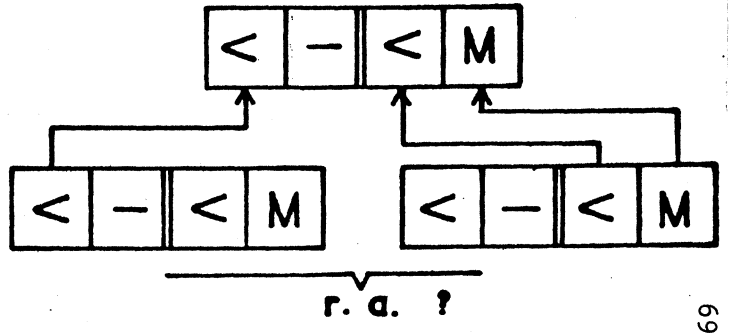
G.

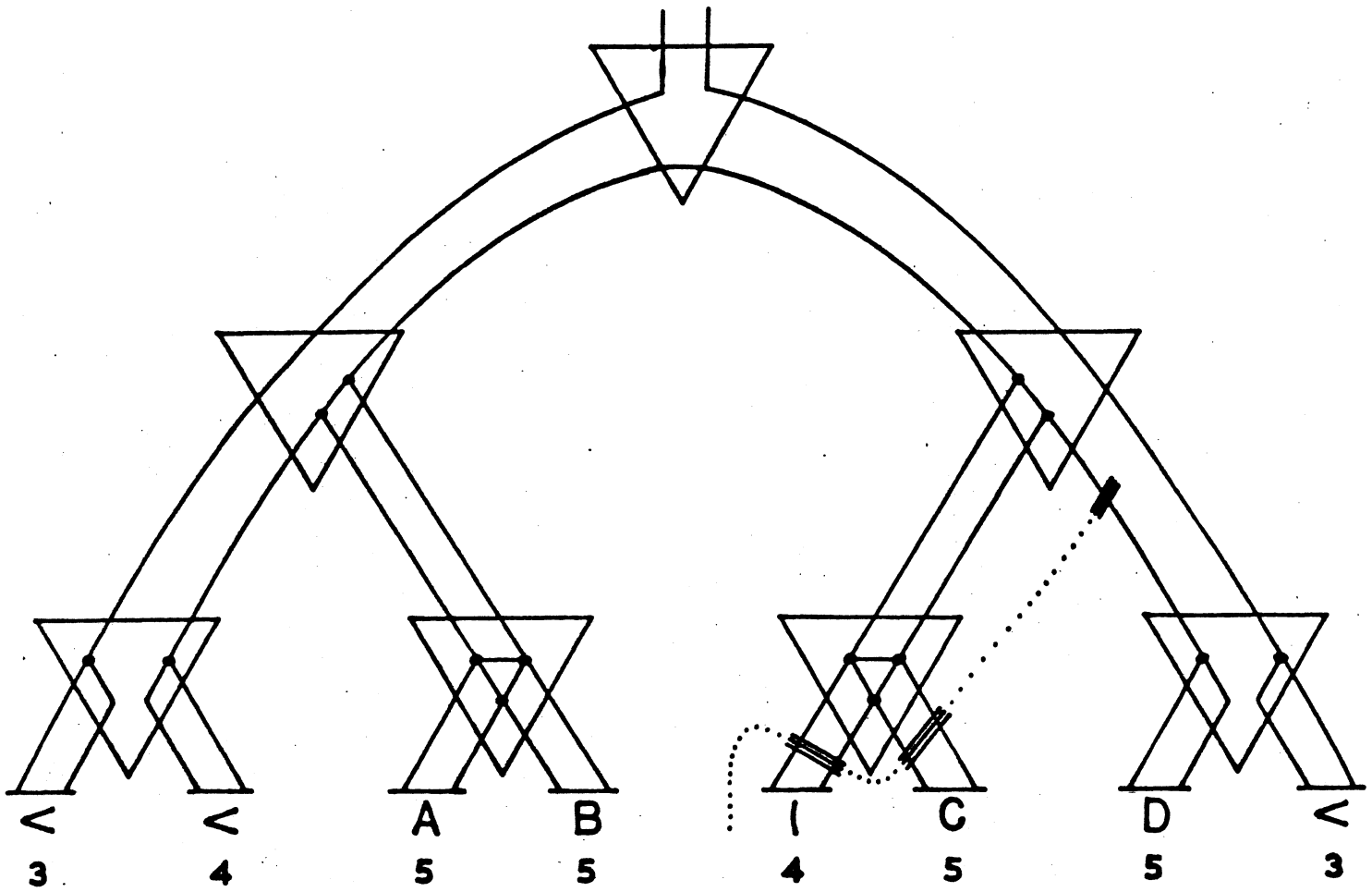




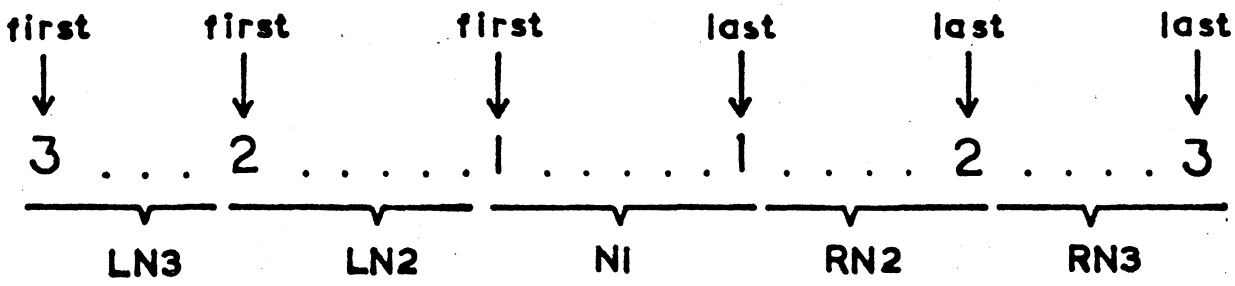
H.



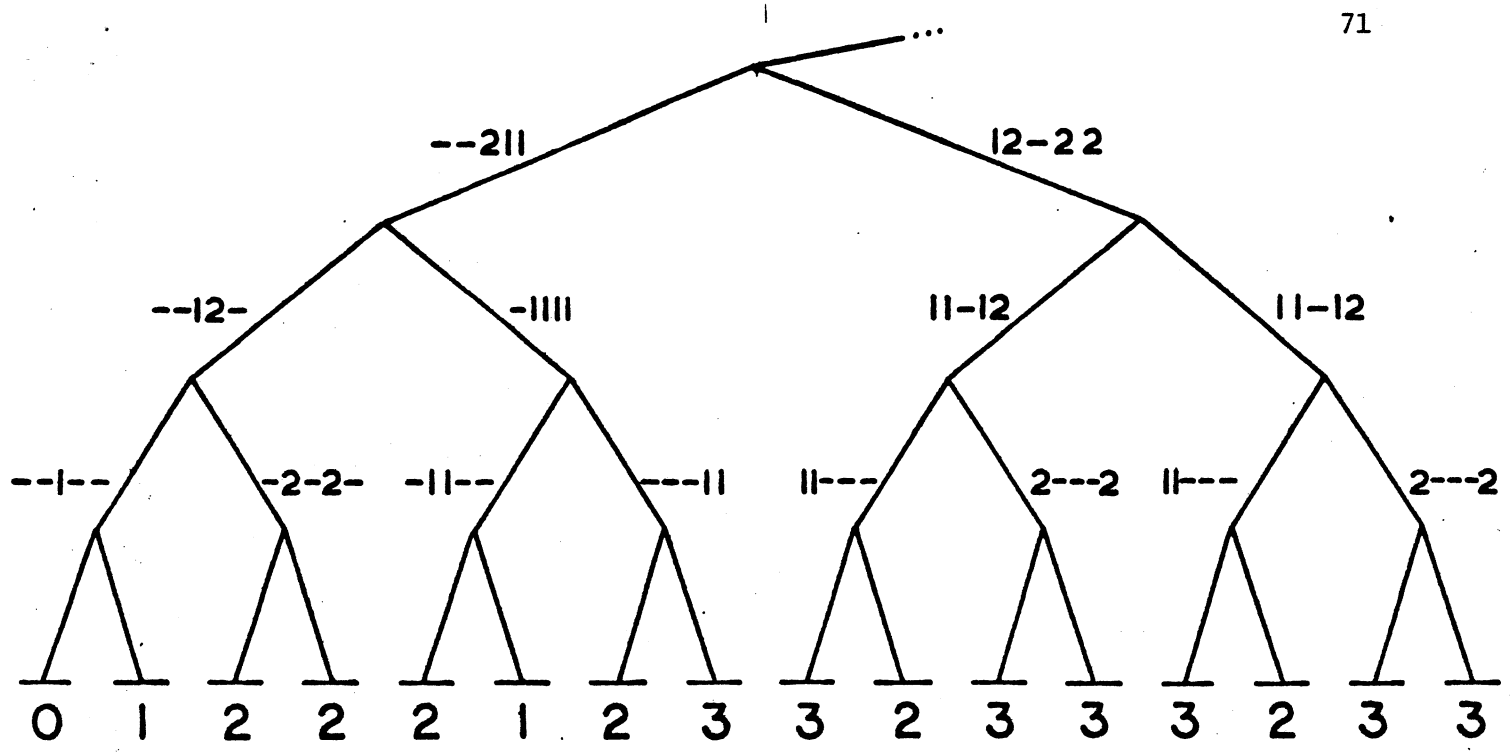



< < A B I C D <
3 4 5 5 4 5 5 3

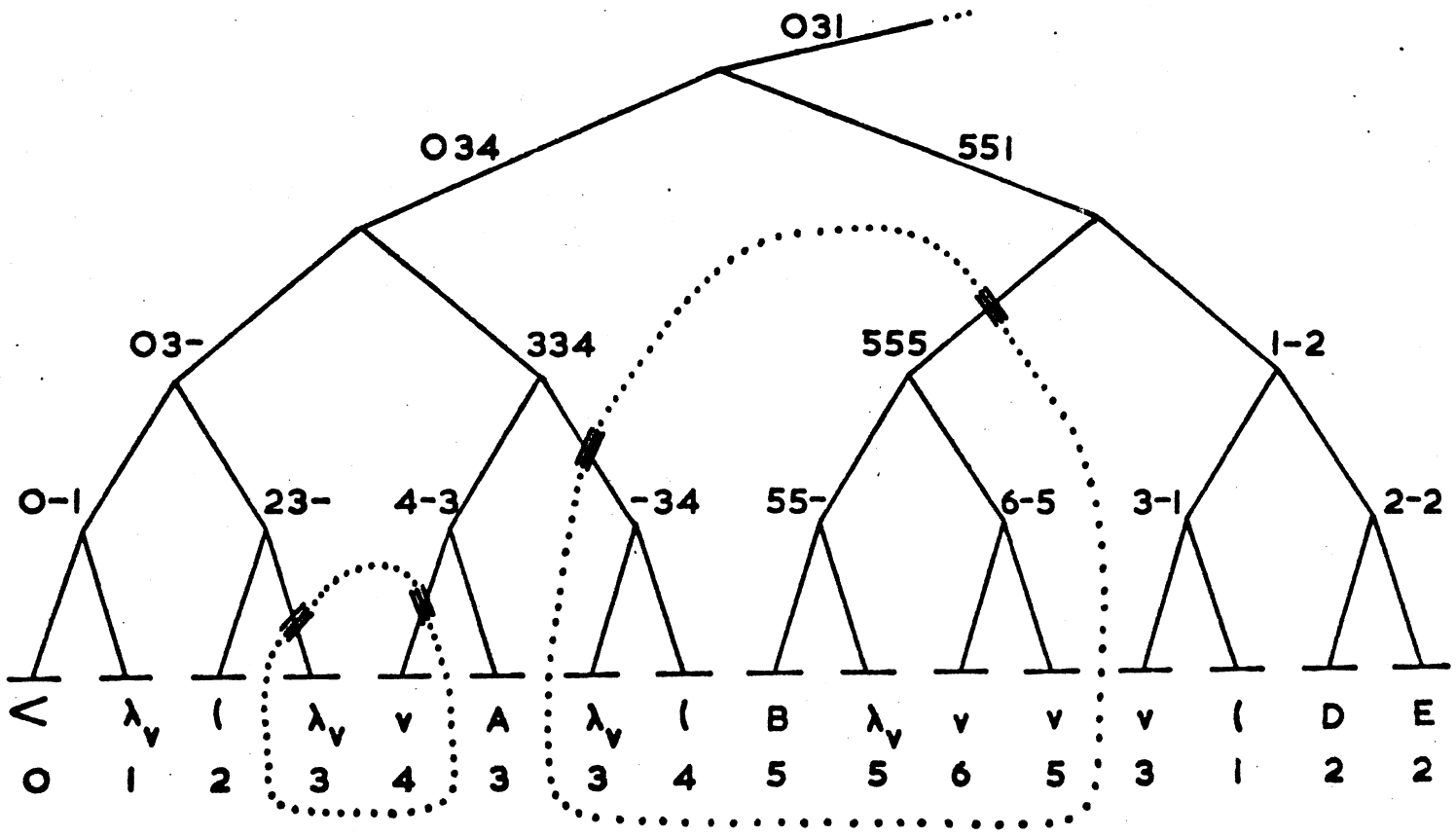
(10)


first first first last last last
3 2 1 1 2 3
LN3 LN2 NI RN2 RN3

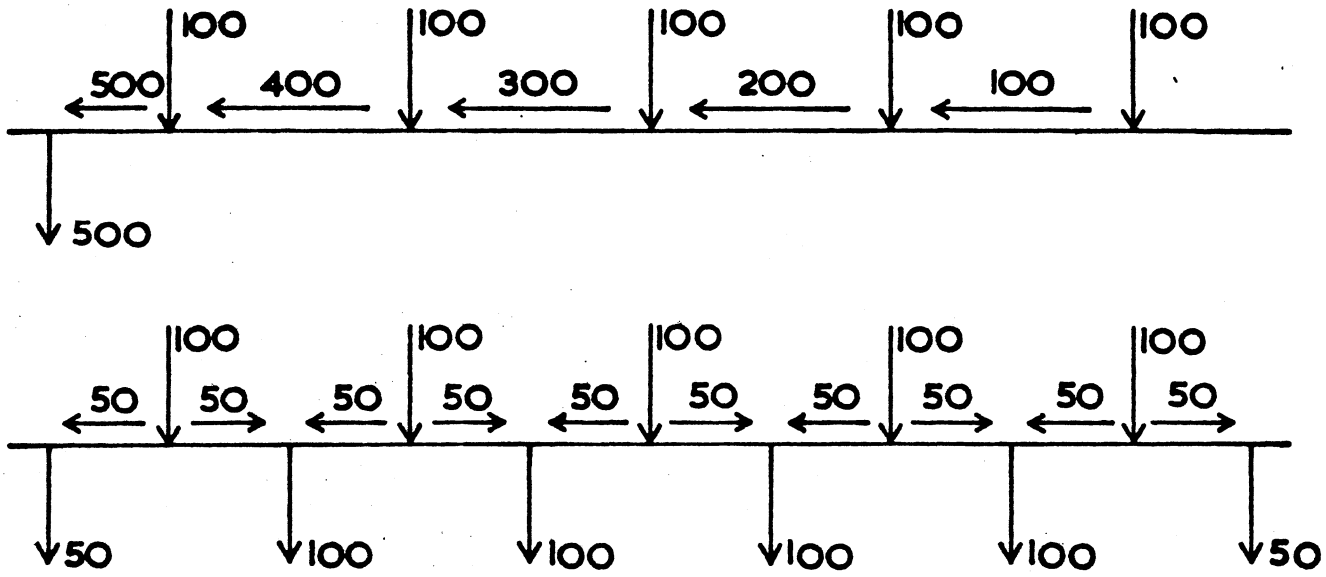
(11)


--211
12-22
--12-
-1111
11-12
11-12
--1--
-2-2-
-11--
---11
11---
2---2
11---
2---2
0 1 2 2 2 1 2 3 3 2 3 3 3 2 3 3

(12)


031
034
551
03-
334
555
1-2
0-1
23-
4-3
-34
55-
6-5
3-1
2-2
< λv (λv v A λv (B λv v v v (D E
0 1 2 3 4 3 3 4 5 5 6 5 3 1 2 2

(3)



4

$S1 \geq 0$

$|S1| \geq |S2|$

S2	—	S2	—
-S1	—	-S1	—

$S2 \geq 0$

5

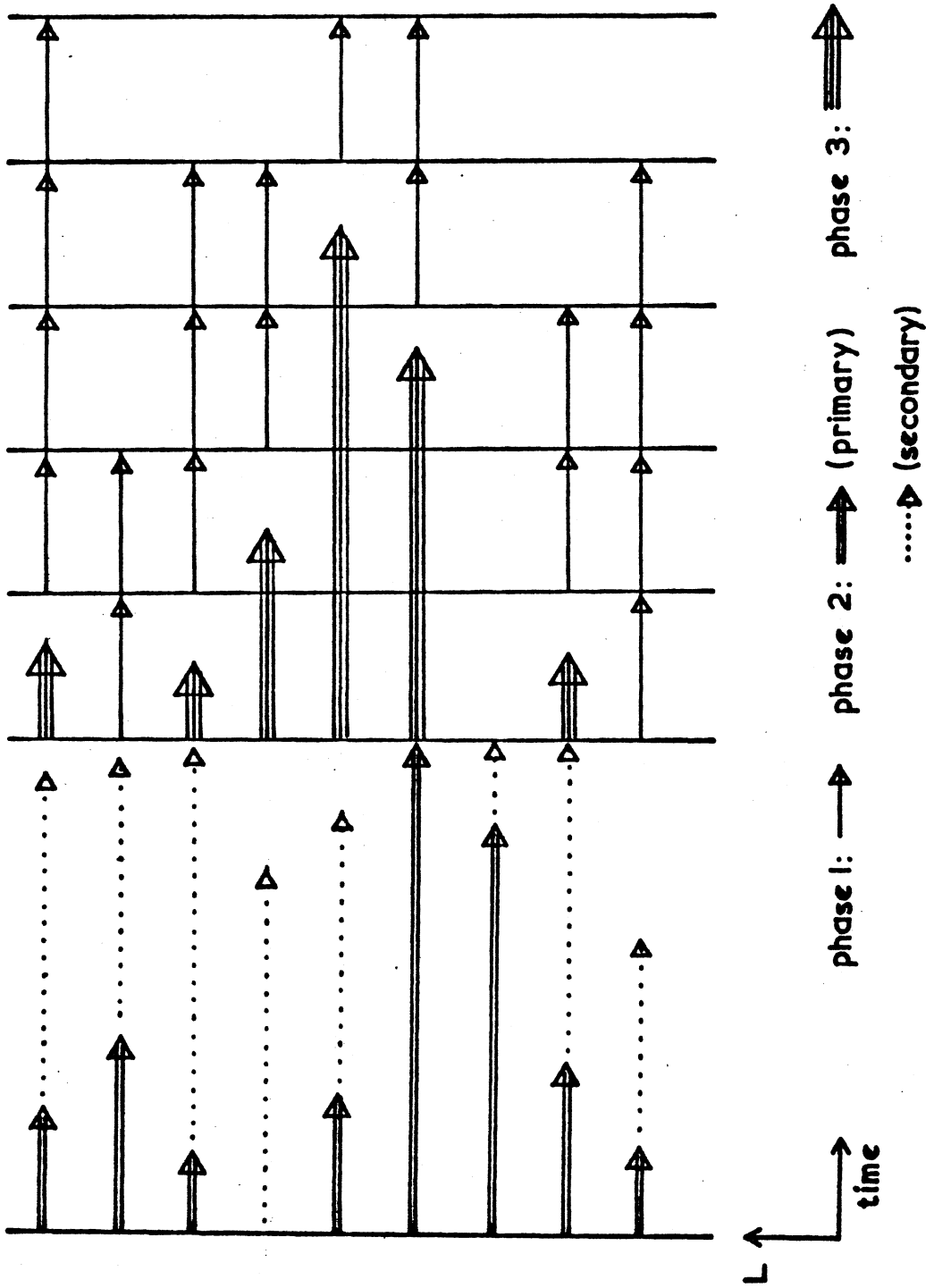
$S1 \geq 0$

$|S1| \geq |S2|$

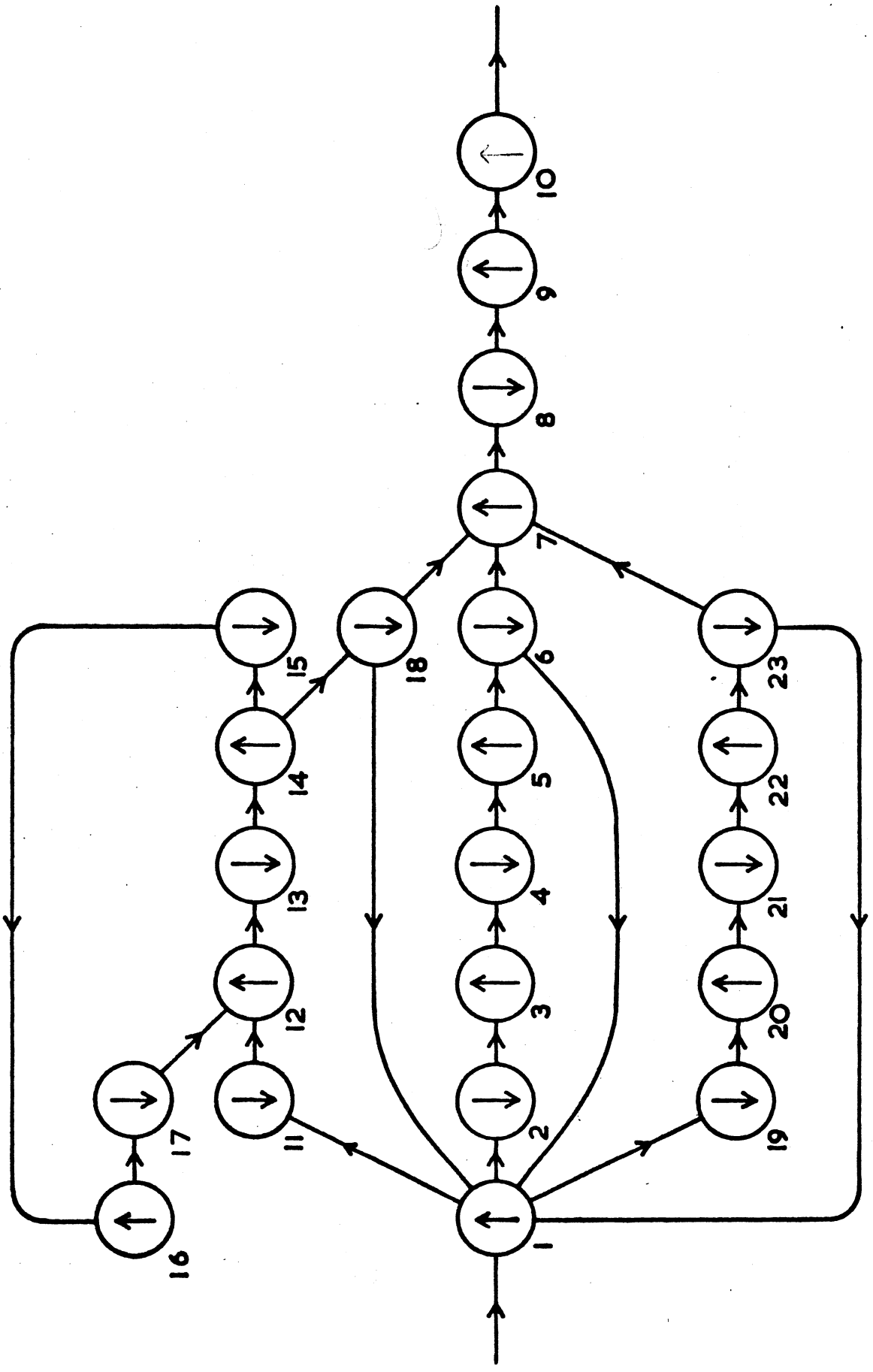
S2	—	$\left\lfloor \frac{S2-S1}{2} \right\rfloor$	$\left\lfloor \frac{S2-S1}{2} \right\rfloor$
$\left\lfloor \frac{S2-S1}{2} \right\rfloor$	—	-S1	$\left\lfloor \frac{S2-S1}{2} \right\rfloor$

$S2 \geq 0$

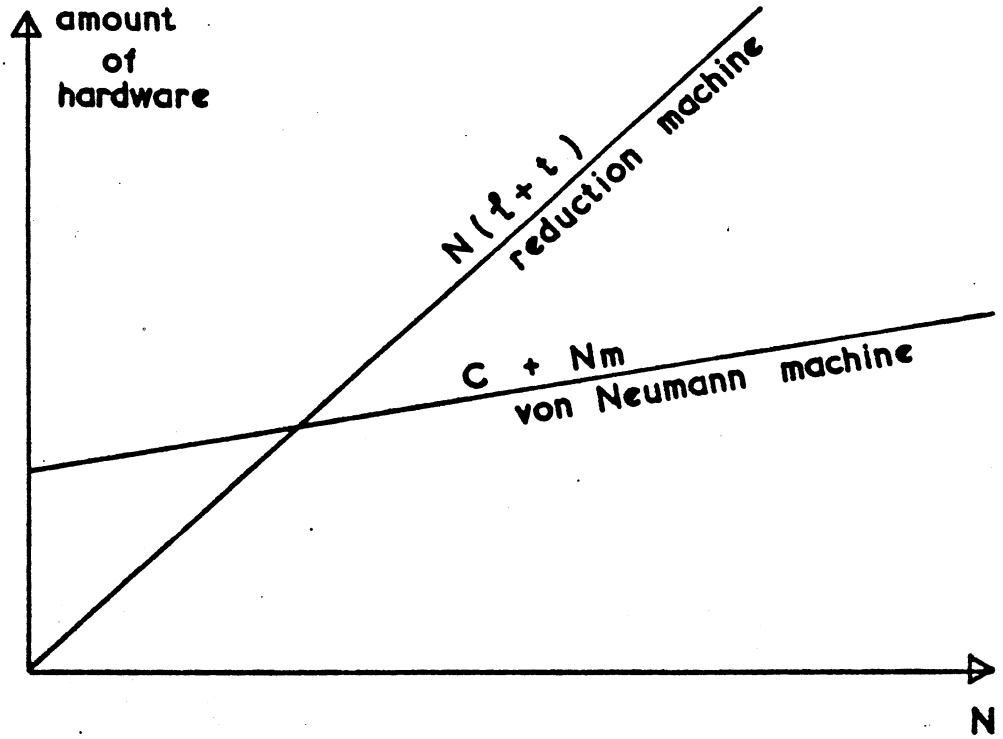
16



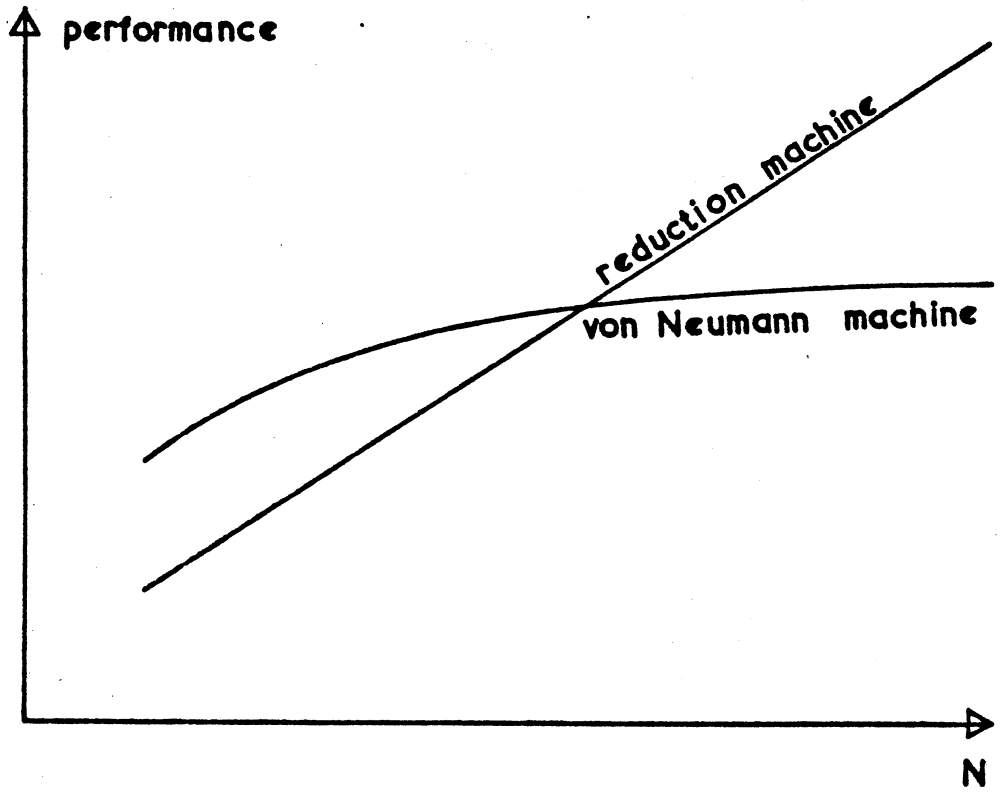
(17)



(18)


amount
of
hardware
N(t+t)
reduction machine
C + Nm
von Neumann machine
N

19


performance
reduction machine
von Neumann machine
N

20

Captions for illustrations:

Fig. 1 : The structure of the processor.

Fig. 2 : Patterns of moving information in the processor.

Fig. 3 : Three representations of the program text.

Fig. 4 : Example to show that nodes of T may have to be shared by different reducible applications. (The nodes circled have to be shared by two or three reducible applications.)

Fig. 5 : Illustrating the algorithm to determine if an application is reducible. (In both cases it is assumed that the subtree on the left contains no internal "<" symbols, hence it is a reducible application.)

Fig. 6 : Algorithm to determine what a reducible application is supposed to do.

Fig. 7 : An example for partitioning the nodes of T. (All cells are numbered for ease of reference. Also, the formats of the data structures associated with some of the double links are shown.)

Fig. 8 : Three links are never needed between two nodes of T.

Fig. 9 : The algorithm to partition the nodes of T. The sixteen different combinations of the input data structures produce eight patterns of partitioning the nodes of T (r.a. is short for reducible application).

Fig. 10: Example to show finding the right end of a reducible application.

Fig. 11: Interpretation of the directory used to distribute microinstructions.

Fig. 12: An example showing the directories in an active area. (In L the program text is omitted, only the relative level numbers are shown.)

Corrections to

A NETWORK OF MICROPROCESSORS TO EXECUTE REDUCTION LANGUAGES

page 37 item (1), which now extends from line 15 to line 17, is to be replaced by:

"(1) two integers are stored into every cell of L specifying the number of symbols entering or leaving the given cell of L on the left and on the right during storage management (these integers implicitly determine the so-called shift amount for each symbol in L, specifying in which direction and how far the given symbol should be shifted in L)"

page 38 line 11
delete "stored into any cell of L"

page 39 line 1 change "computes" to "results in"
line 2 change "to" to "that"

page 44 delete sentence extending from line 10 to line 15.

page 46 change lines 6, 7, and 8 to:

"9 carry out storage management

10 (↑) detect end of storage management"

page 48 line 3 change "largest primary shift amount (LPS)"
to "largest shift amount (LS)"

line 4 change "to determine LPS"

to "to detect that every cell of L had been entered and left by the required number of symbols"

line 5 change "state 9"
to "state 10"

line 7 change "max(lg_2N, LPS)"

to "$(lg_2N)T_1$ + LS"

line 9 change "LPS" to "LS"
line 14 change "LPS" to "LS"
line 17 change "LPS" to "LS"

page 49 replace line 10 by:
"$lg_2N(17T_1 + 2T_2)$ + LS + nT_1"

Table IV replace it with new version

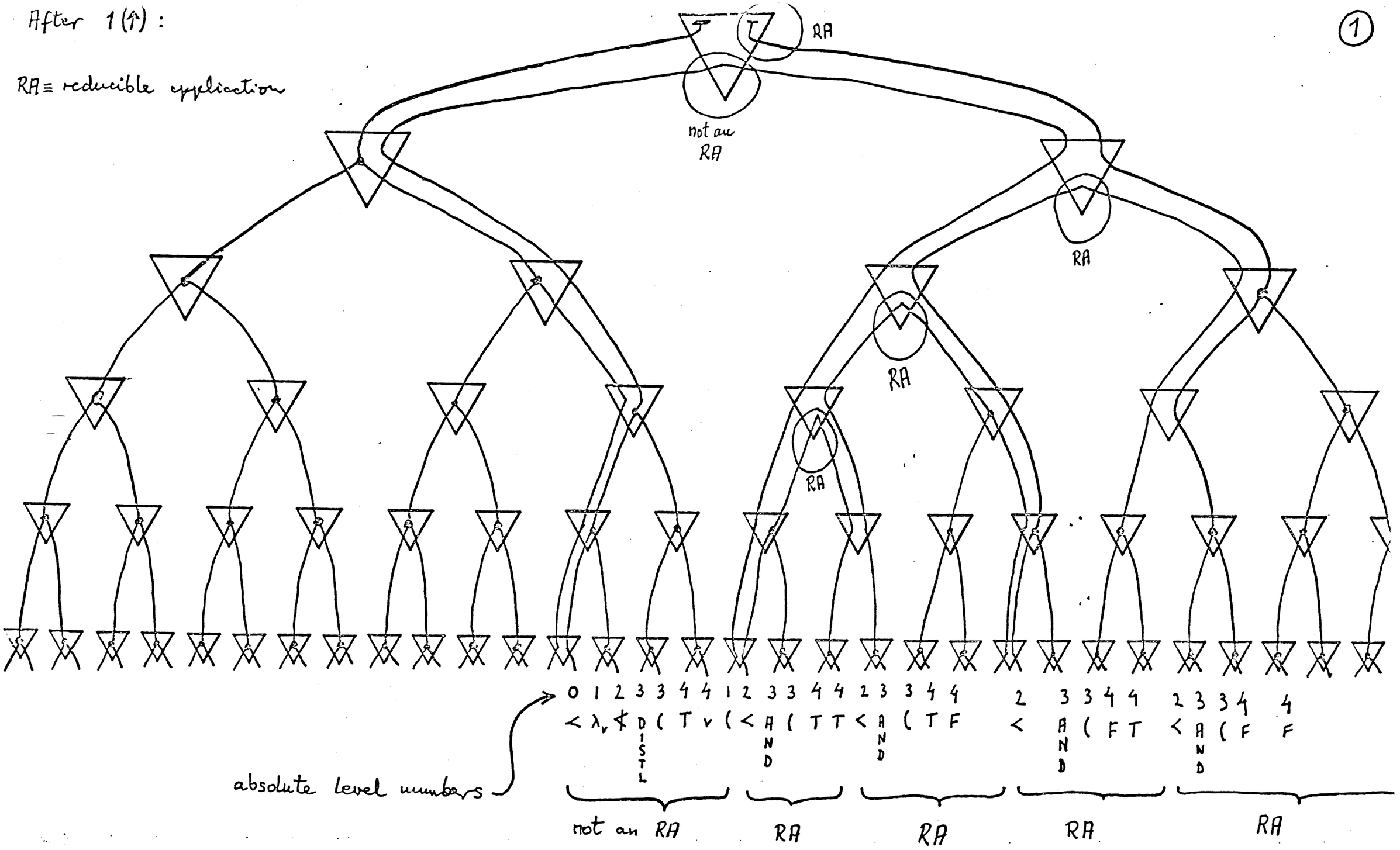
Fig. 17 omit references to secondary storage management algorithm

Fig. 18 in state 9 delete "↑"

in state 10 add "↑"

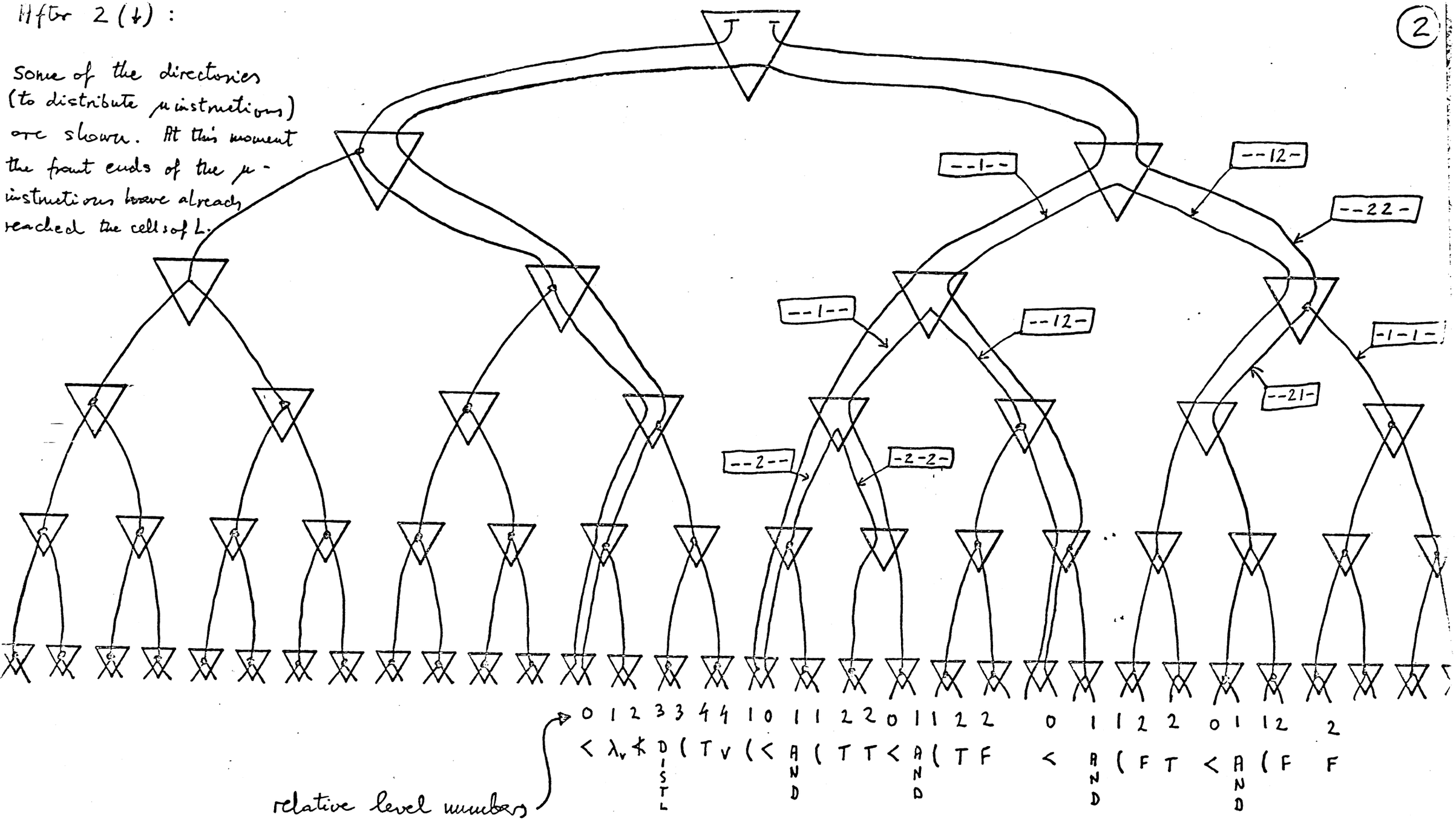
After 1(↑):

RA ≡ reducible application



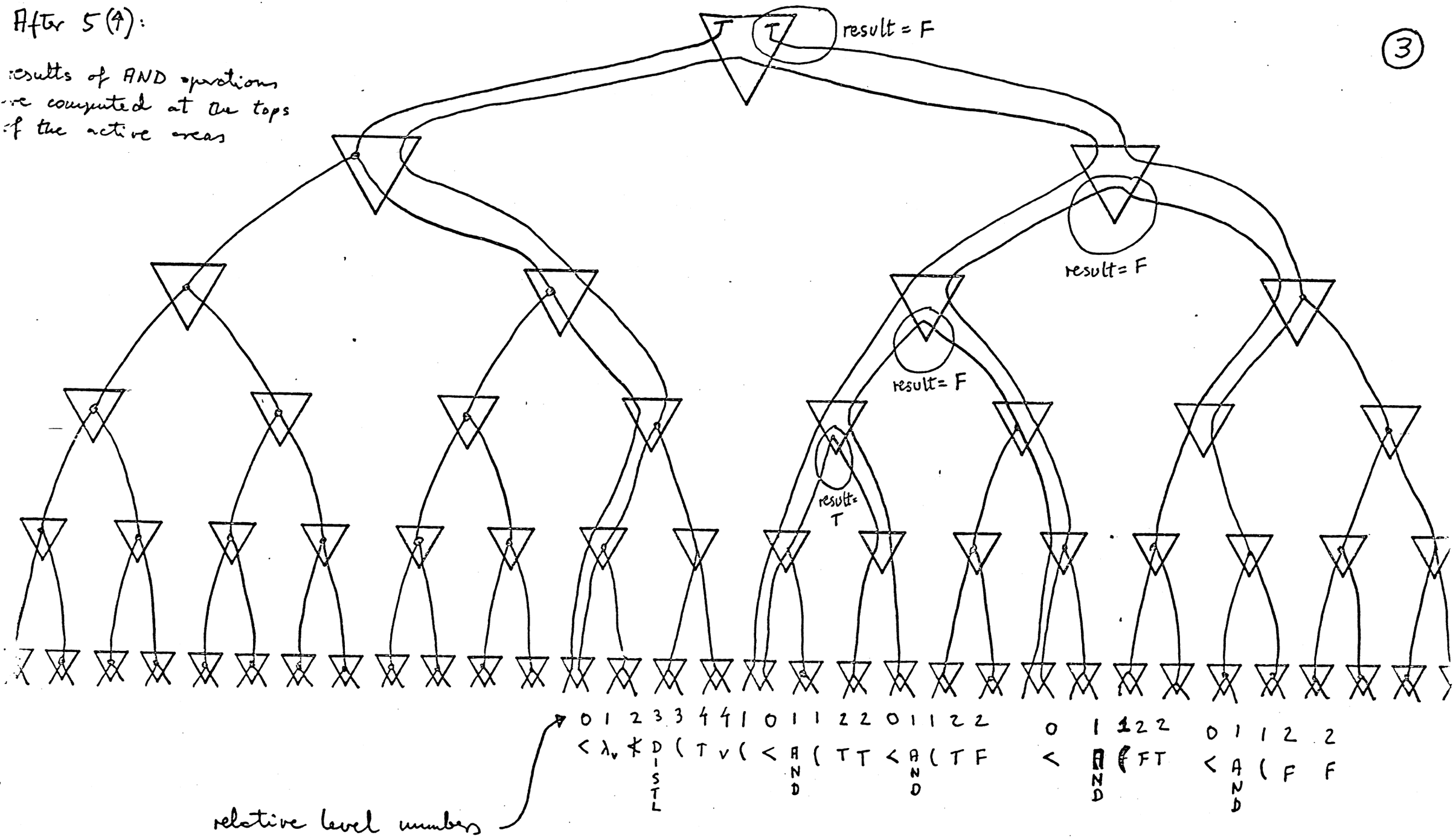
After 2(+):

Some of the directories (to distribute μ instructions) are shown. At this moment the front ends of the μ-instructions have already reached the cells of L.



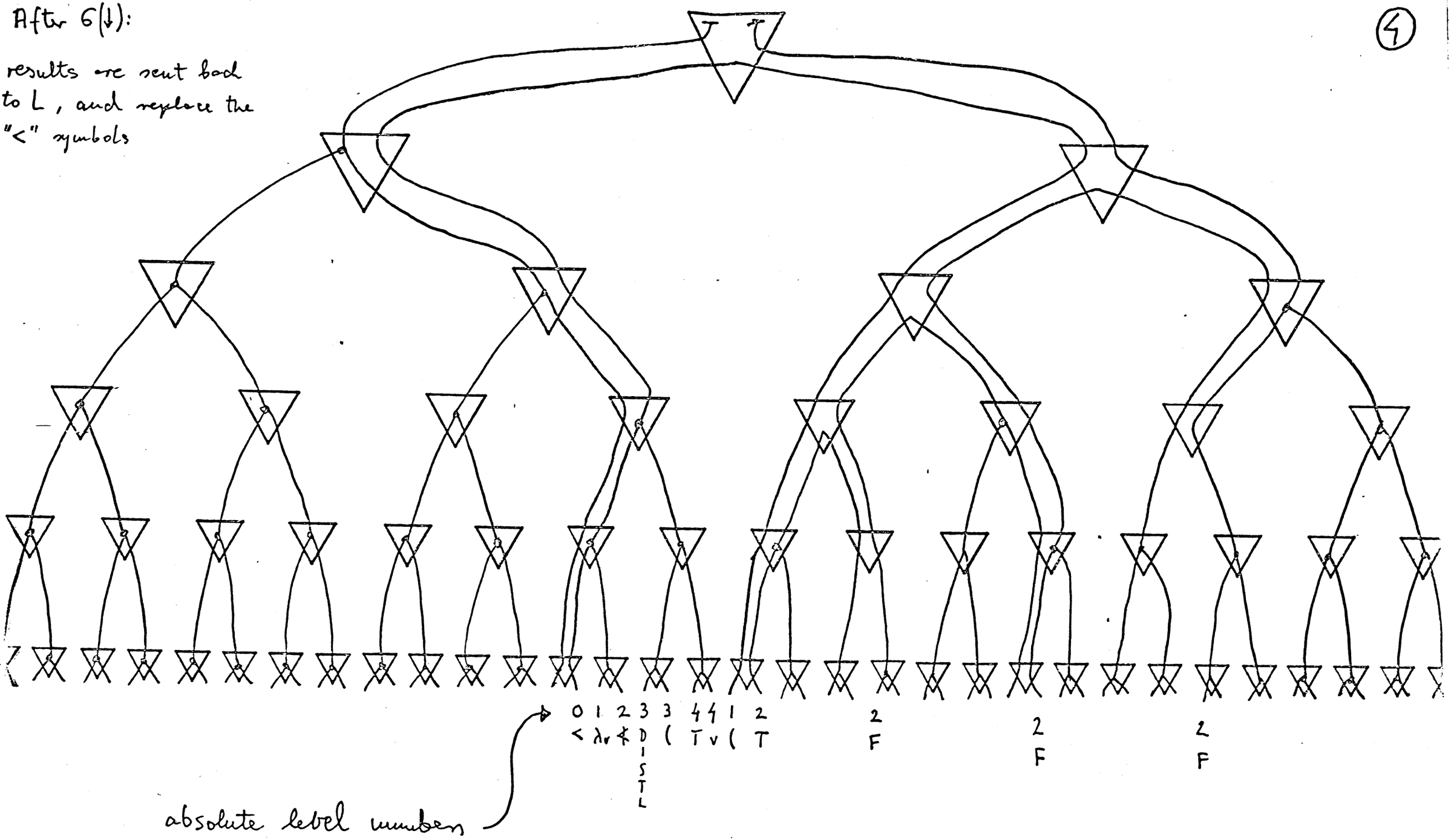
After 5(4):

results of AND operations are computed at the tops of the active areas

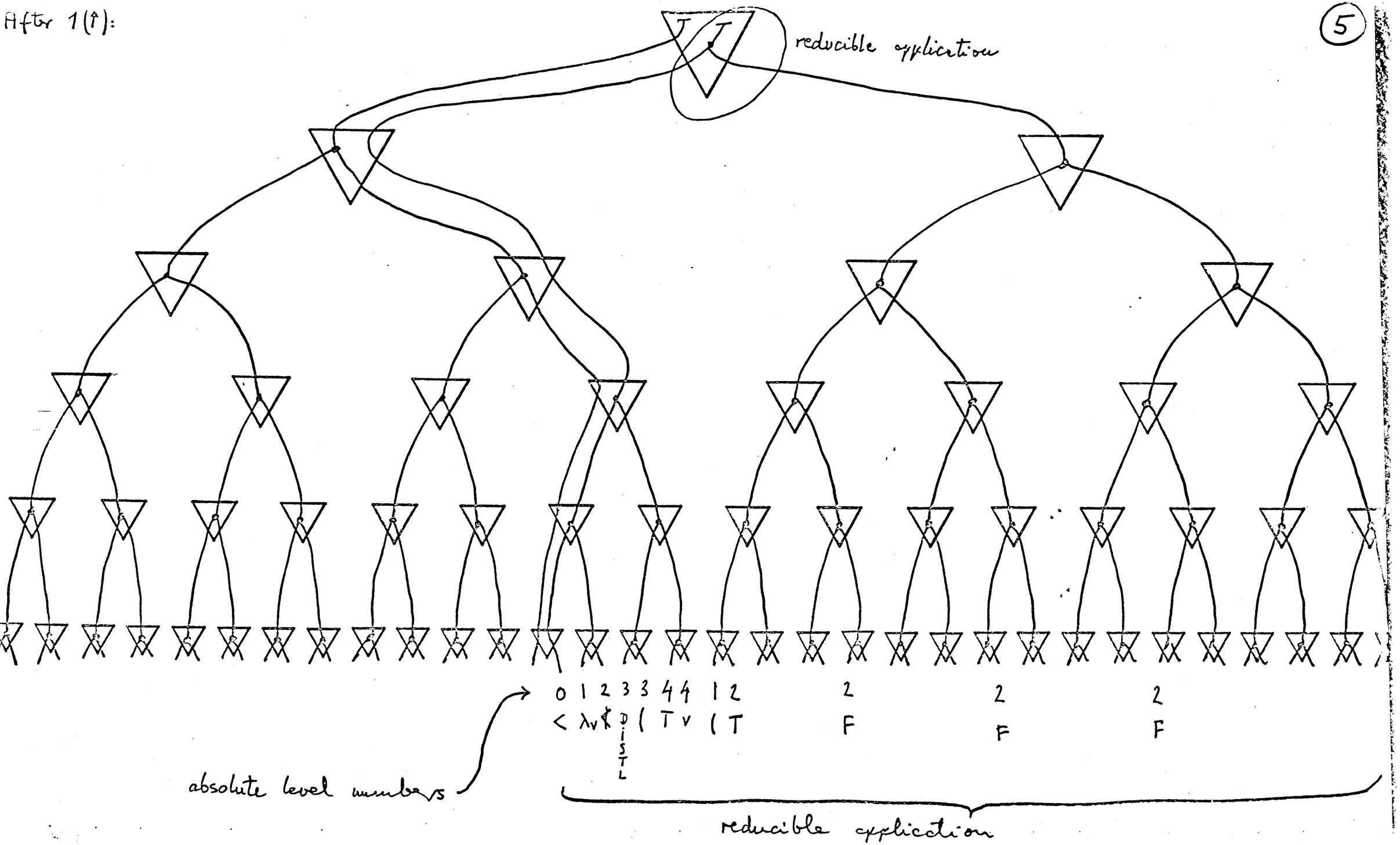


After 6(↓):

results are sent back to L, and replace the "<" symbols

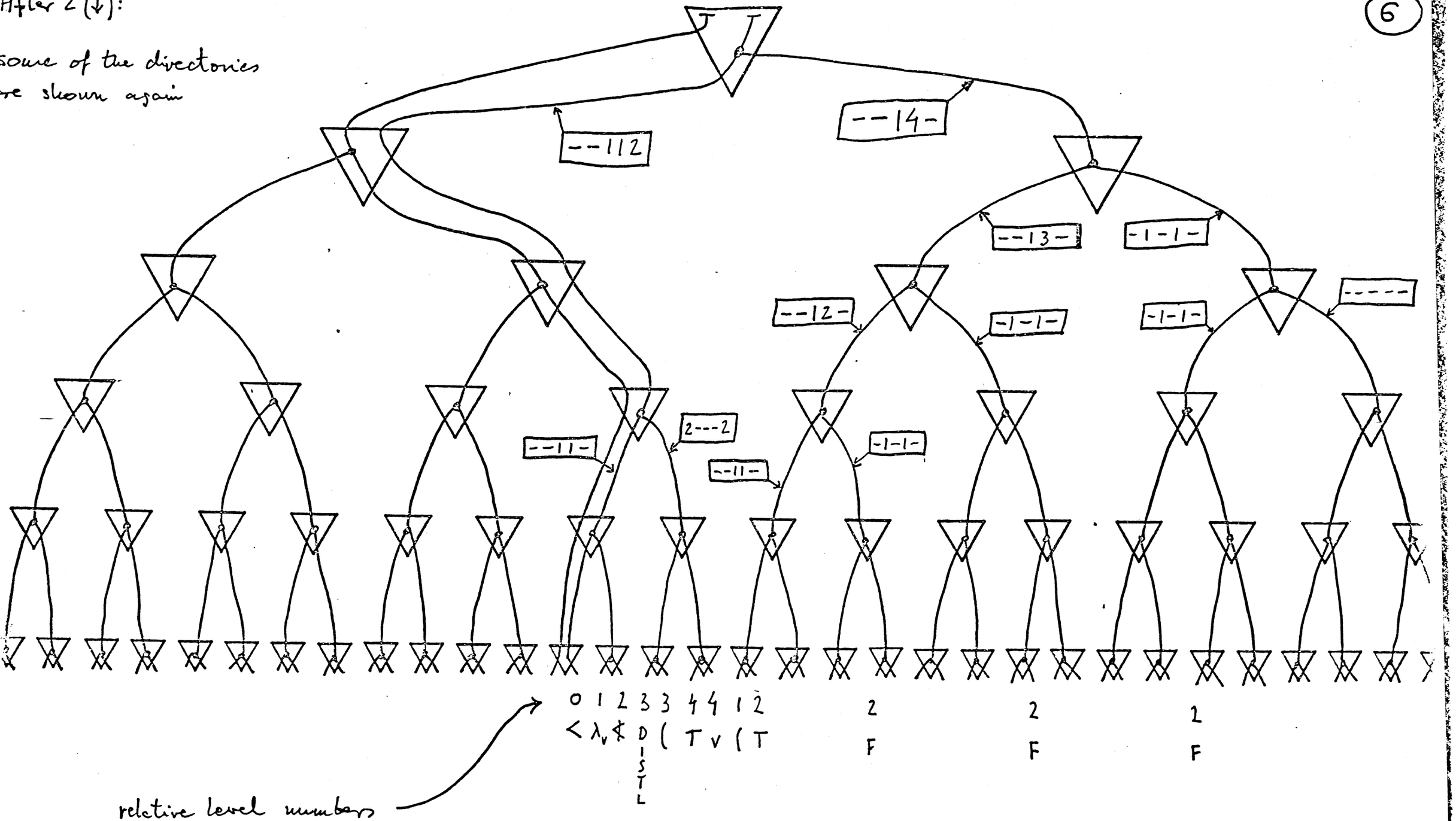


After 1(f):


T T
reducible application
0 1 2 3 3 4 4 1 2 2 2 2
< λv & DISTL (T v (T F F F
absolute level numbers
reducible application

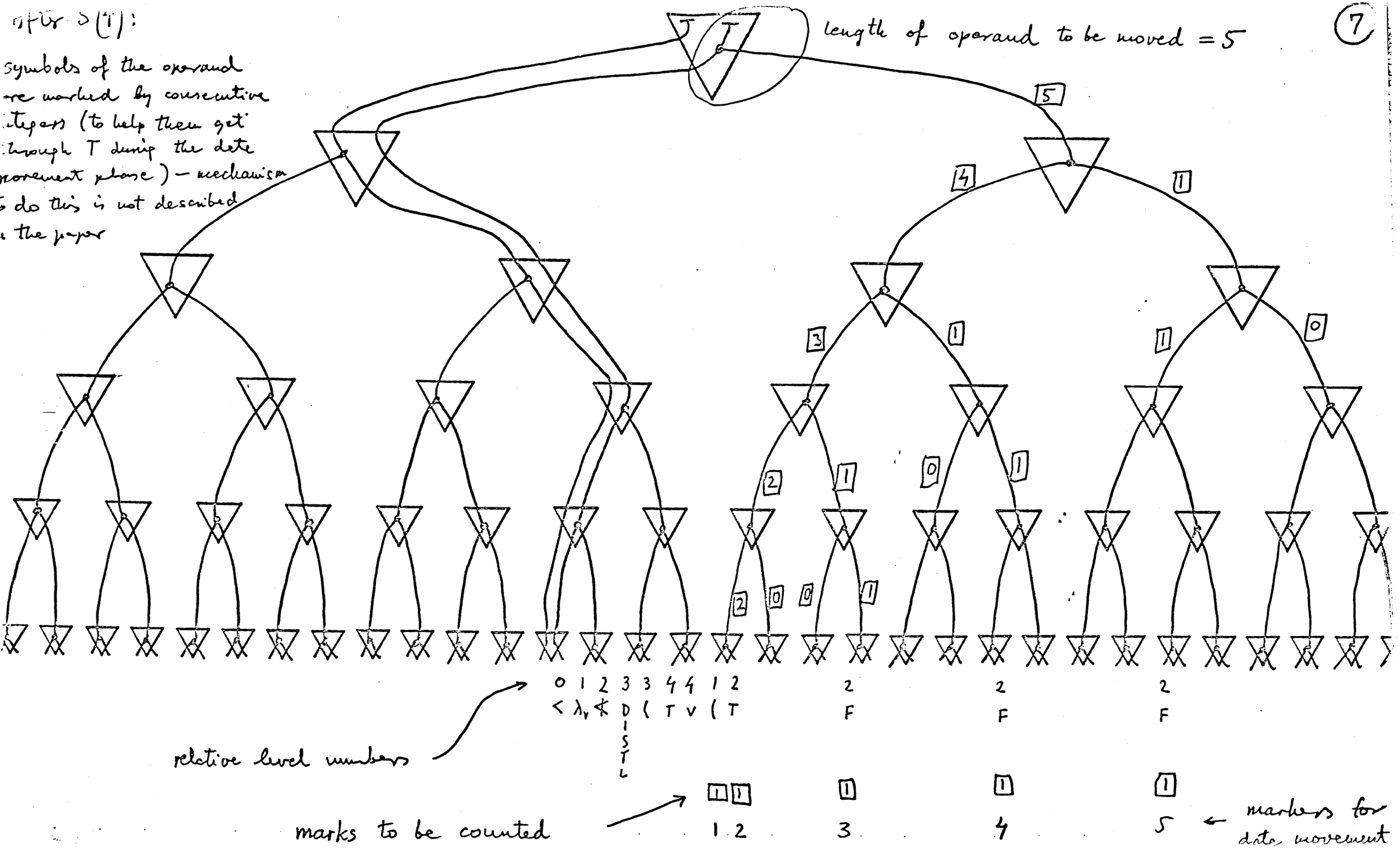
After 2 (↓):

source of the directories
re shown again



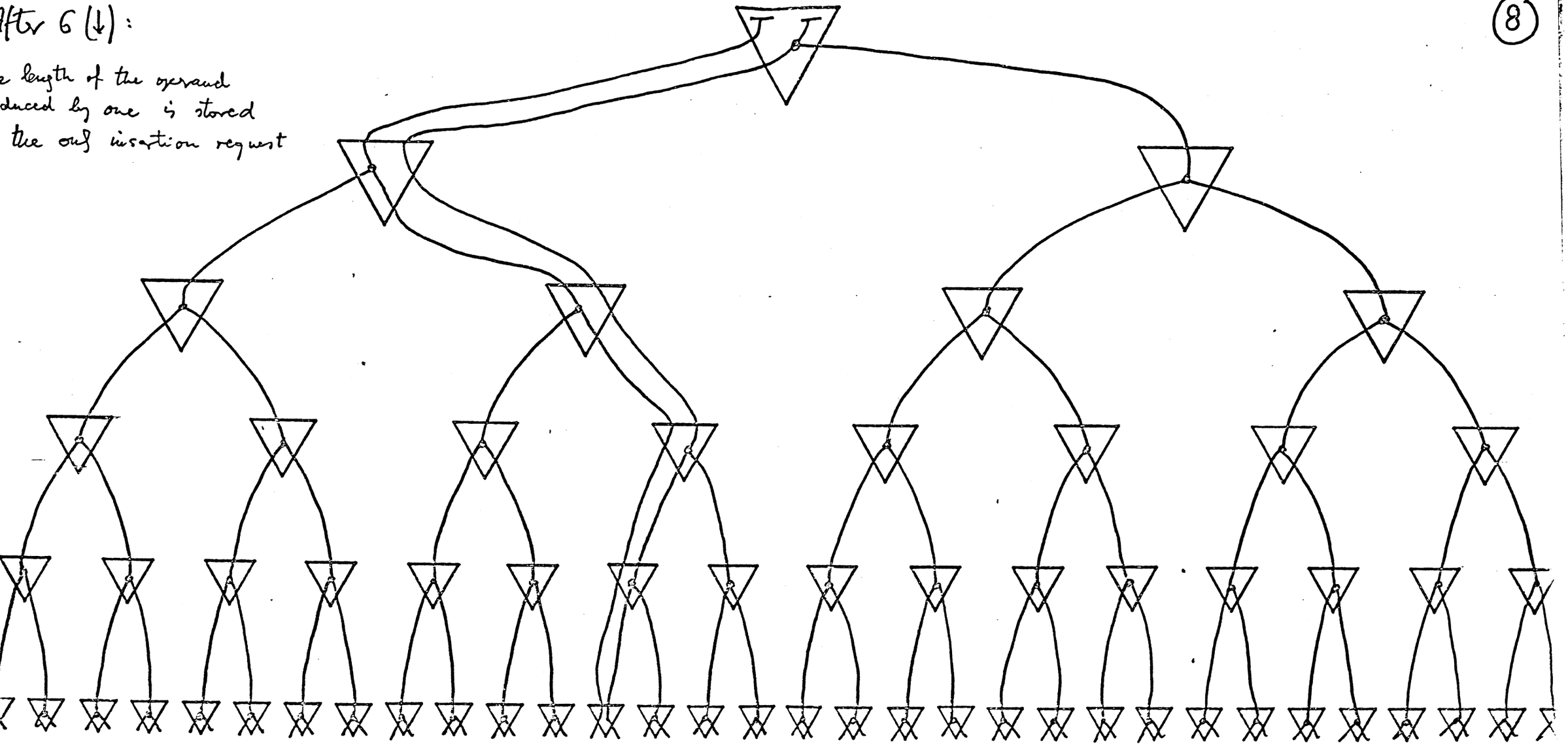
for $\sigma(1)$:

symbols of the operand are marked by consecutive integers (to help them get through T during the data movement phase) – mechanism to do this is not described in the paper



ftr 6 (↓):

e length of the operand
duced by one is stored
the ons insertion request



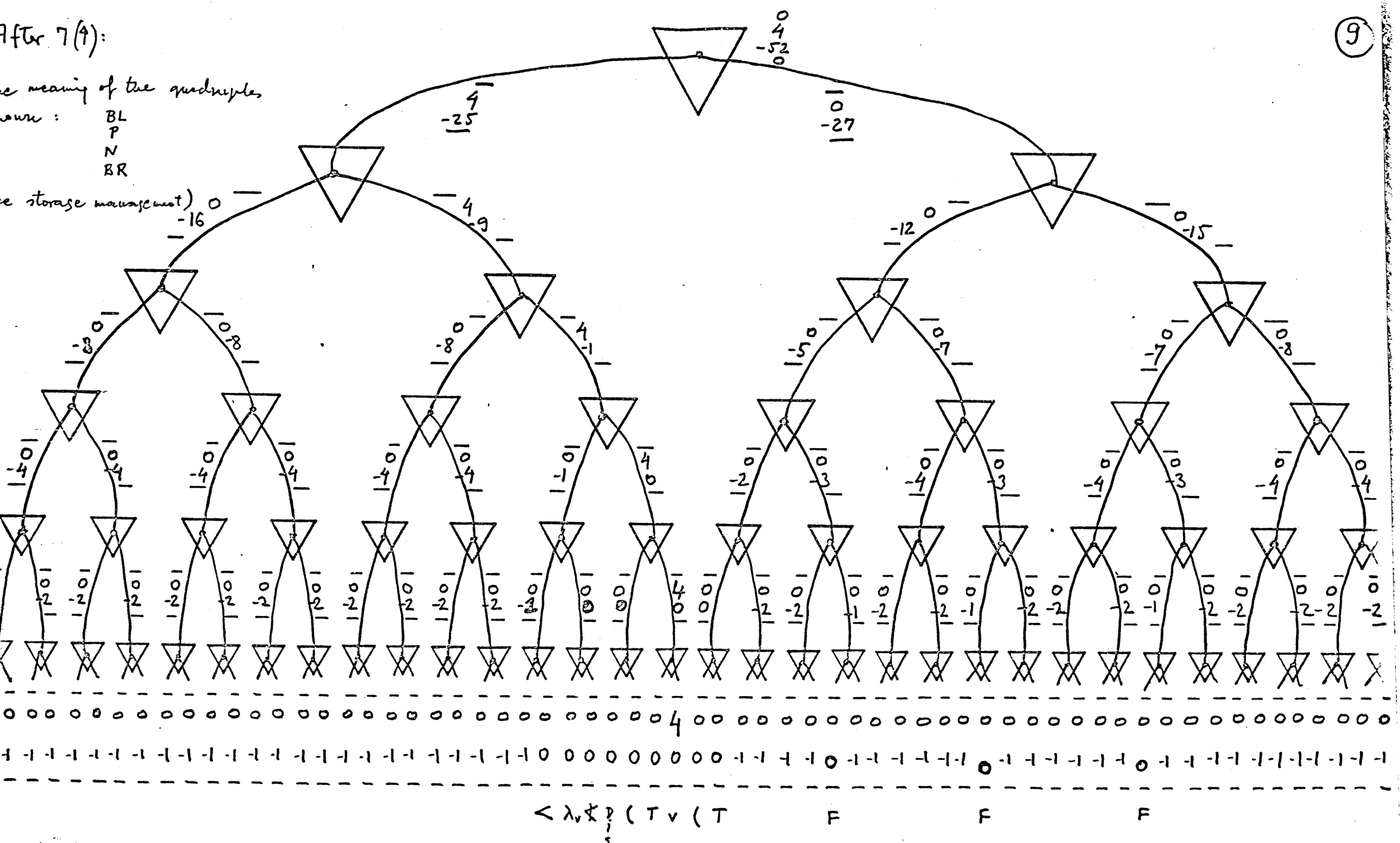
0 1 2 3 3 4 4 1 2 2 2 2
< λv ⊀ D (T v (T F F F
I
S
T
L

insertion requests (P) → 0 0 0 0 0 0 4 0 0 0 0 0

After 7(4):

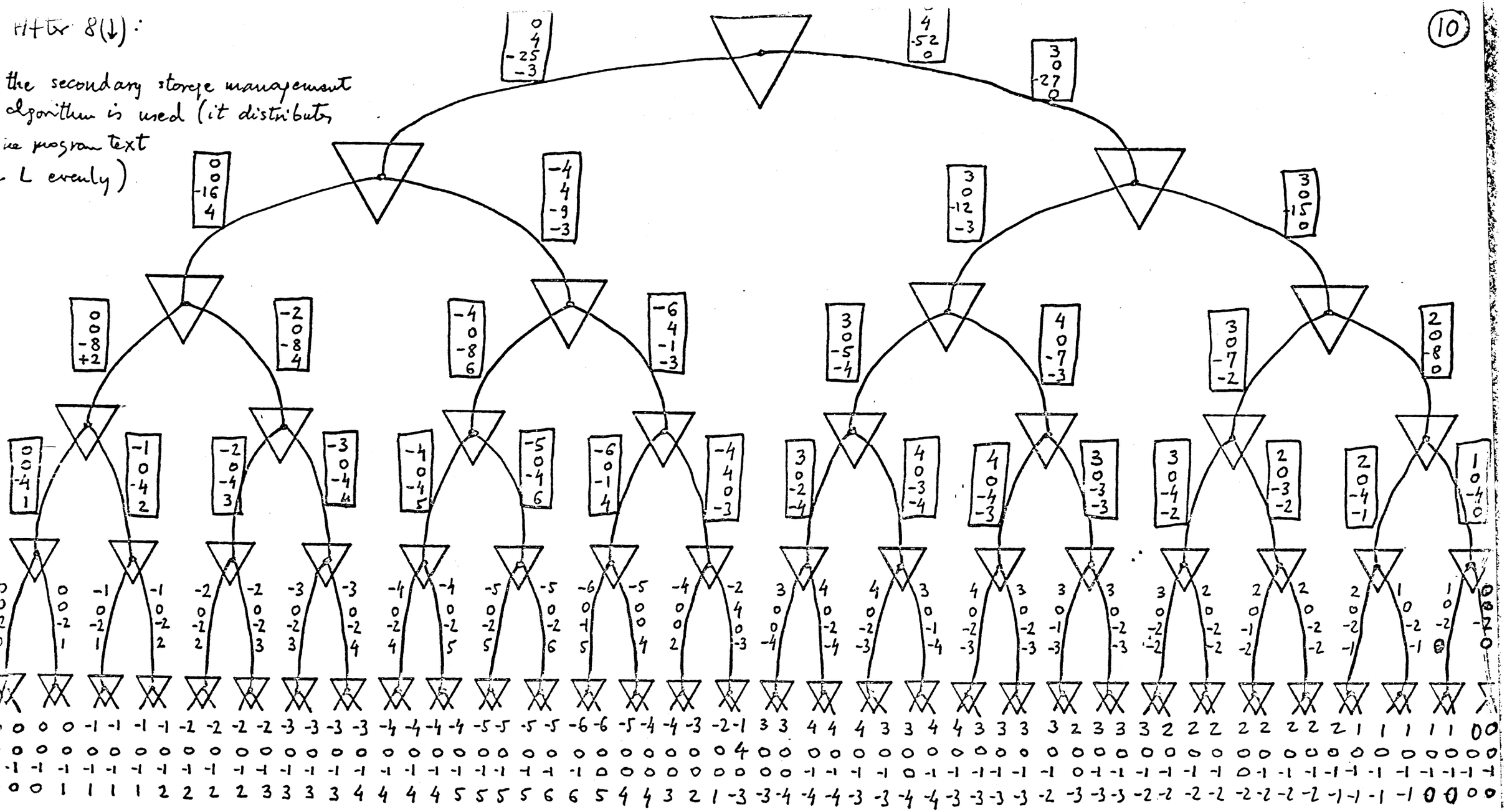
e meaning of the quadruples
ours : BL
P
N
BR

e storage management)



Htr 8(↓):

the secondary storage management algorithm is used (it distributes the program text L evenly)

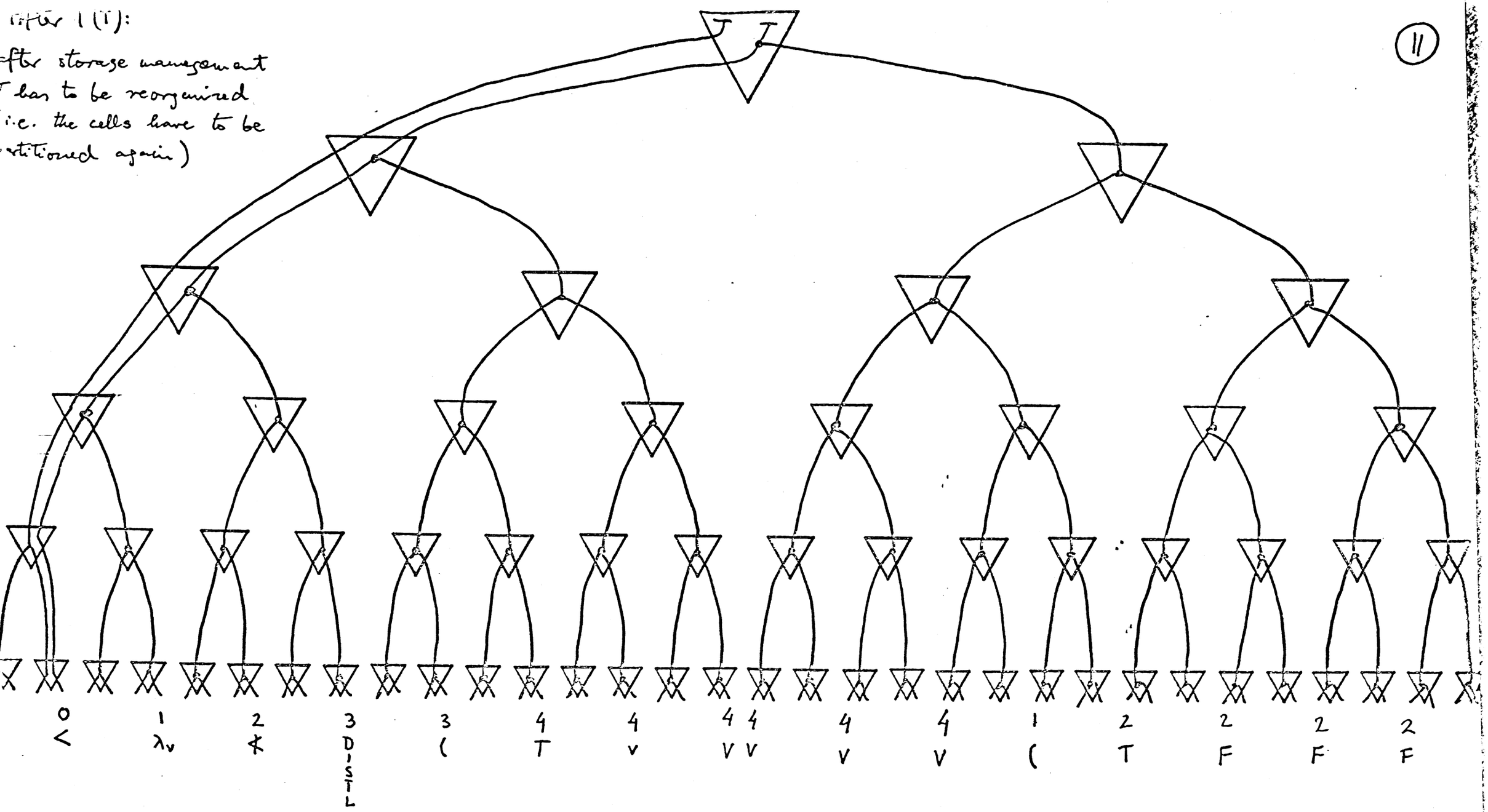


before storage management : < λv ∢ DIST (T v (T F F F

after: < λv ∢ DIST (T v v v v (T F F F

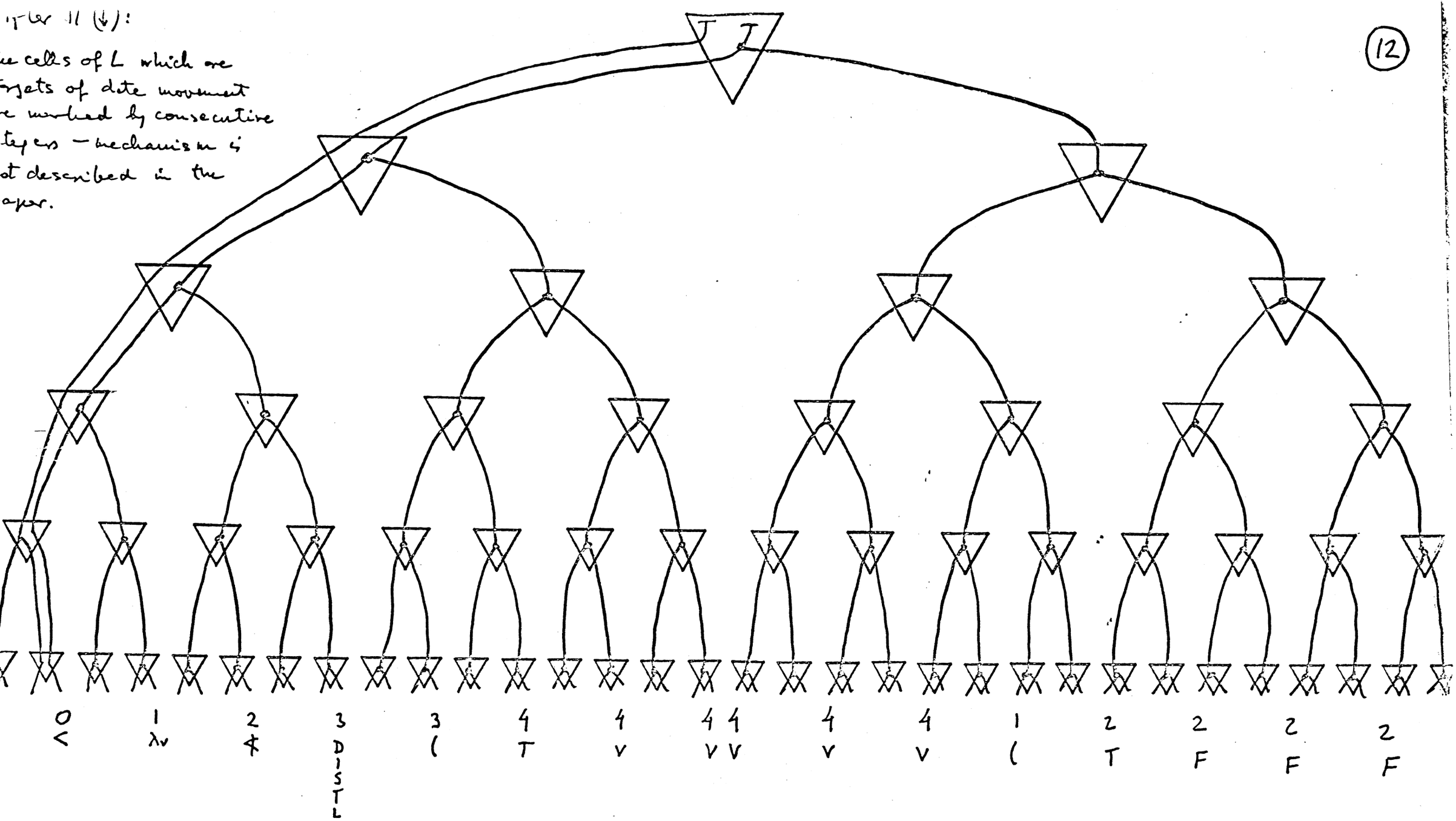
after I (T):

after storage management
T has to be reorganized
(i.e. the cells have to be
partitioned again)



...ter II (↓):

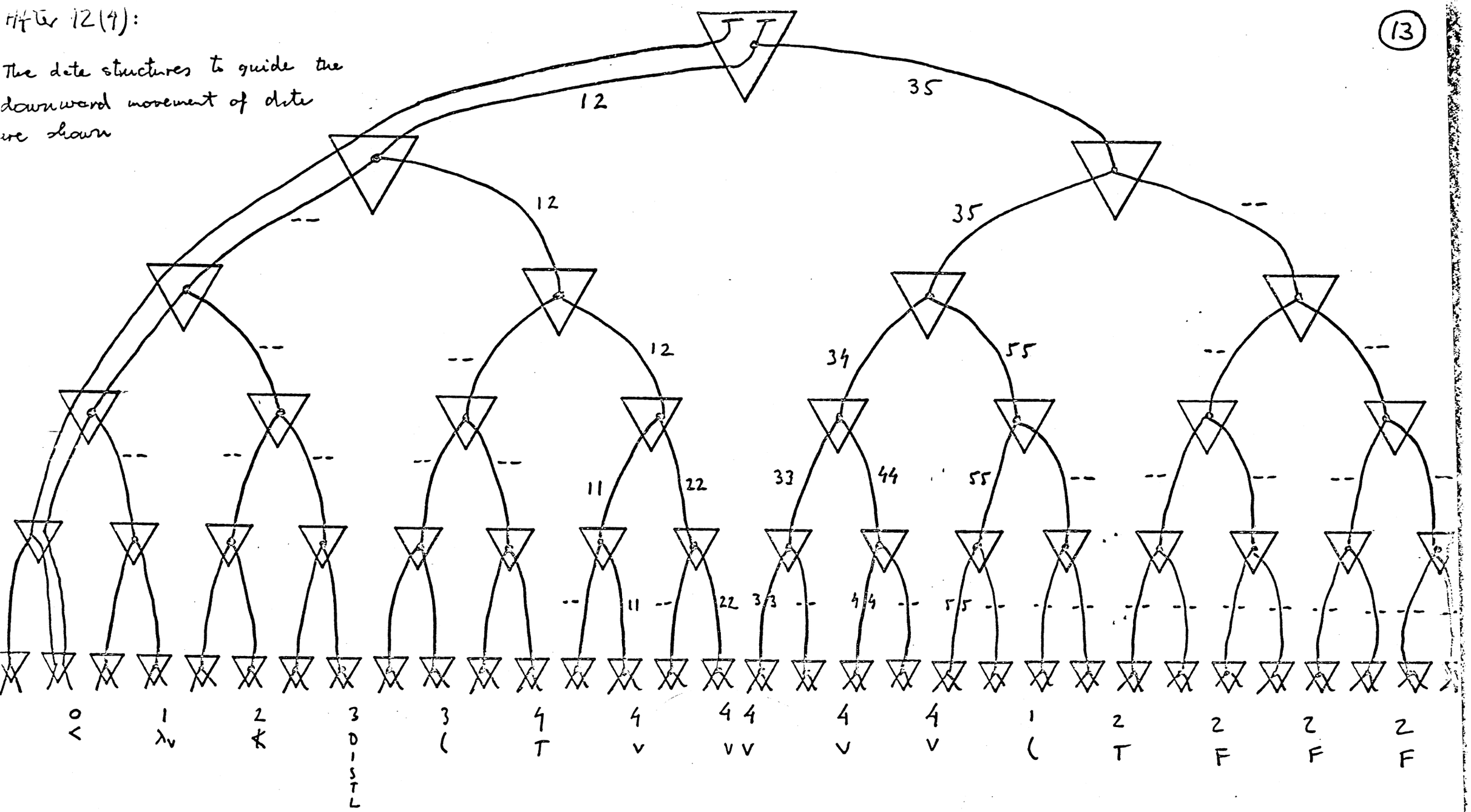
...e cells of L which are
...rgets of data movement
...e marked by consecutive
...layers — mechanism is
...ot described in the
...aper.



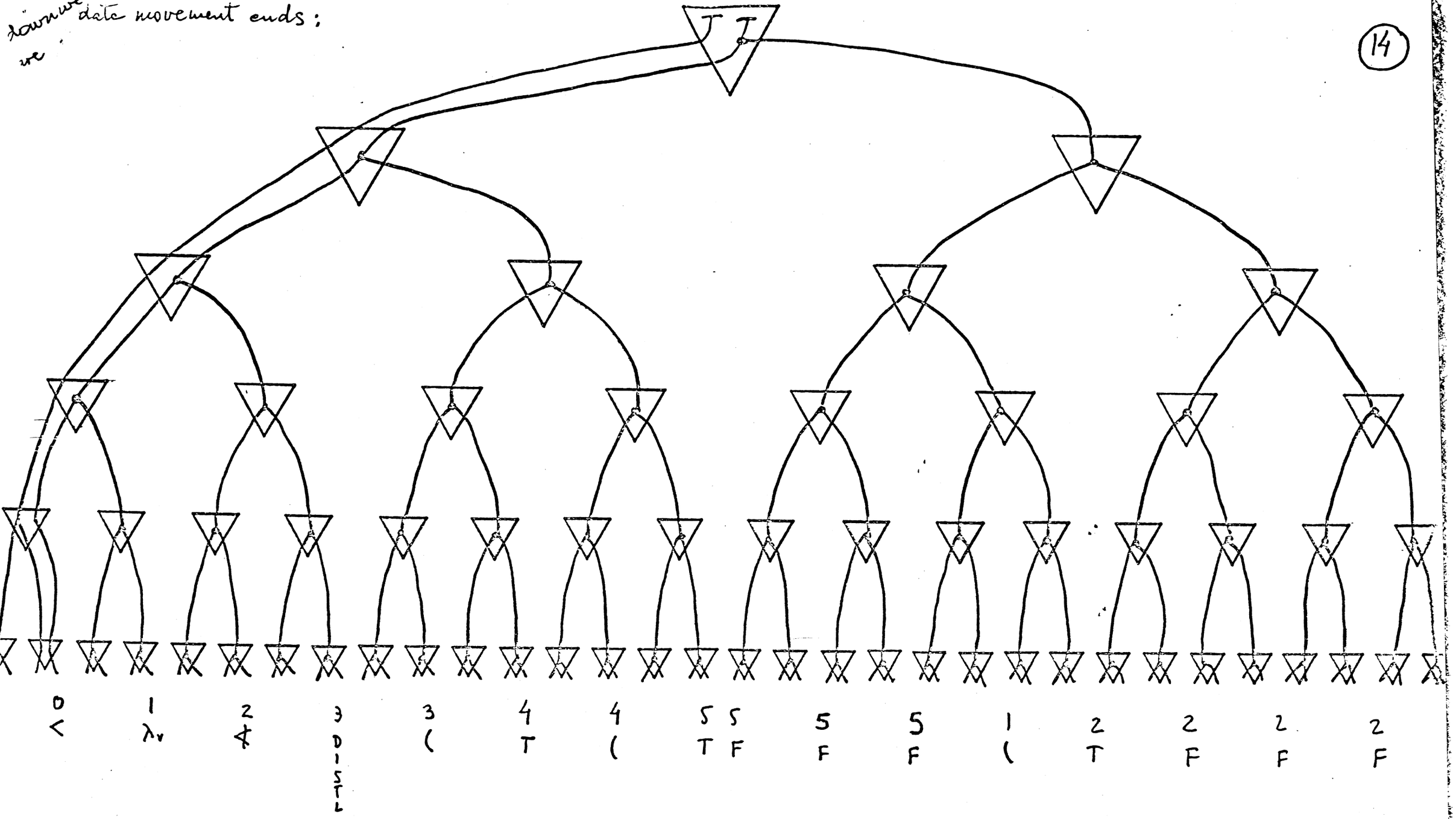
Markers to guide data through T: 1 2 3 4 5

After 12(4):

The data structures to guide the downward movement of data are shown



downward data movement ends;
we



After 18(↓):

The μ instructions to erase, and to rewrite level numbers have been stored into cells of L previously, and are executed now.