Implanting FFP Trees in Binary Trees: An Architectural Proposal

Donald MacDavid Tolle*

Department of Computer Science
University of North Carolina at Chapel Hill

Introduction

The computer architecture described here was inspired by Magó's recently proposed cellular computer [1979] based on the Formal Functional Programming (FFP) languages introduced by Backus [1978]. Magó's machine is a binary tree of many simple processor/memory units, called cells. FFP programs are stored, one symbol per cell, in the leaf cells ("L" cells) of the machine. The machine fully exploits all the parallelism expressed in an FFP program, storage space permitting. It can, furthermore, execute many programs simultaneously, and it can perform in parallel many operations below the level expressed in the program, since a single "primitive" FFP operation may be composed of more basic machine operations that can be executed in parallel. Magó's work brings to light the interesting possibilities of a binary tree of processors operating on individual symbols of an FFP expression.

The machine described here is similar in many ways to Magó's, but differs substantially from it in others. During the execution of an FFP program in Magó's machine, the T cells (i.e., the non-leaf cells; the "tree" cells) are used only for the transmission of messages among L cells and for simple combining operations on the data in certain of those messages. The complexity and the power reside principally in the L cells. In the machine described here, by contrast, the T cells have rather sophisticated computational capabilities. In this machine, the syntactic structure of an innermost application of an FFP program is used to impose a structure on the T cells and L cells of the machine, by embedding a network of nodes corresponding to the subexpressions of the application. Because the network spans numerous cells, considerable computational power is dedicated to the reduction of each innermost application. Because the network's structure corresponds to that of the FFP text, the reduction is easily expressed in terms of operations performed by the nodes on data stored at and flowing among the nodes. Because the network also reflects the structure of the underlying binary tree, certain concurrent computational techniques (such as pipelining) that are not apparent at the FFP language level are made feasible. As new FFP expressions are produced during execution of the FFP program, new syntactic networks are automatically and dynamically embedded and used for further execution.

The machine described here appears to have some advantages over Magó's, in terms of conceptual simplicity, flexibility in defining the primitive FFP operators, and the potential for a higher degree of concurrency below the level expressed in the FFP language. Whether these advantages outweigh the disadvantages--in particular, the greater complexity and cost of the individual cells--remains to be seen. Henceforth, Magó's machine is called Machine I; the version described here is called Machine II. Machine II, like Machine I, is a binary tree of cells. For simplicity in this paper, it is assumed that the FFP text is stored, at most one symbol per cell, in left-to-right order in the L cells of the machine. (A more compact representation is described in [Tolle 1981]. It allows certain configurations of syntactic brackets to be stored with an FFP atom in

* Now at Shell Development Company, Houston, Texas.

a single L cell.)

Partitioning

The embedding of networks in Machine II has two phases: partitioning and parsing. The partitioning process divides the machine into disjoint areas corresponding to the innermost applications--also known as reducible applications or RAs--and thereby allocates to each RA a portion of the machine for that RA's exclusive use. The areas, though logically disjoint, may share T cells; that is, any given T cell may serve several RA areas at once. The parsing process then builds a syntactic structure in each RA's area of the machine. The partitioning and parsing processes are similar in many respects. Each consists essentially of a single sweep of information upward through the cells of the machine; each embeds in the machine a tree of several kinds of nodes, connected by several kinds of channels. The nodes act as processors and the channels carry information between the nodes. The structure embedded by the partitioning process is called the TA-Mediator network. Figure 1 shows an embedded TA-Mediator network; Figure 2 shows the same network, without the machine cells.

"TA" stands for "Top of Area of an RA." The partitioning process associates exactly one TA with each RA, embedding the TA in the lowest cell that "sees" the entire RA. The TA supervises execution of the RA. Each RA area (i.e., the TA and its subtree) is disjoint from the other RA areas. Corresponding to each pair of adjacent RAs (i.e., each pair of RAs not separated by another RA) is a Mediator node, embedded in the lowest machine cell that "sees" the entire text of both RAs. The Mediators supervise storage management.

An IN node embedded in a cell corresponds to text that that cell has determined may be internal to some RA. The middle subtree of a TA is a binary tree of IN nodes. An XN node corresponds to text that the cell knows is not internal to any RA (hence is external). The (- and -) nodes correspond respectively to '(' and ')' in the program text. Each L cell embeds exactly one IN node (which may correspond to no text, if, for instance, the cell is empty). An L cell containing '(' embeds these three nodes:

XN (- IN

and an L cell containing ')' embeds these three nodes:

IN -) XN

Each type of node has a unique type of channel leading upward from it. The partitioning process merges certain combinations of adjacent channels to form the nodes. Figure 3 shows the allowable combinations, and Figure 4 shows all the possible configurations of a partitioned T cell. Notice that no T cell has to embed more than four nodes (but the parsing process may embed more, as explained below). Since the amount of information passed from a cell to its father cell during the partitioning process is limited, the partitioning process takes time proportional to the height of the machine, i.e., time that is logarithmic in the number of L cells.

Parsing

Once a TA has been formed, it takes charge of the reduction of its RA. Since each TA sees its RA through its own separate network of nodes and channels, the TAs can proceed independently of each other. Each TA initiates parsing immediately upon being formed, so that parsing occurs simultaneously with the continued building of the TA-Mediator network higher up in the machine.

The parsing process refines the middle subtree of the TA (recall that it is a binary tree of IN nodes) by embedding a more sophisticated tree reflecting the RA's syntax. Each IN node that lies inside an RA area and sees FFP text actively participates in the parsing, and thereby embeds within itself one or more new nodes and channels. The resulting tree under each TA is called the SN-CP network of the RA, in reference to two of its several kinds of nodes, the SN node and the CP node.

Since each machine cell has only a fixed amount of hardware, it is not possible to parse fully an arbitrarily deeply nested RA. Thus only the top few syntactic levels (i.e., the least deeply nested levels) of each RA are fully parsed. By the relative level number (RLN) of a symbol we mean the nesting depth of the symbol relative to the RA's application brackets. Most of the computation will be expressed in terms of operations on symbols with small RLNs--i.e., symbols at the top levels of the RA. (Of all the primitive operators mentioned in Backus's Turing Award Lecture [1978], only one requires knowledge of the structure of the operand to a depth greater than three.) It is essential, though, that all the text of the RA be "visible" to the TA for certain basic functions such as copying the text to another location or comparing two expressions for equality.

Thus the following strategy is used. A small positive integer D is chosen at the time the machine is built. In each RA, every sequence bracket with relative level number (RLN) greater than D is treated, for purposes of parsing, as if it were an FFP atom. In effect, this "flattens" all the syntactic structure below level D into a single "flat" level (with RLN = D + 1).

The TA, upon being formed, initiates a downsweep in its subtree to compute the RLN of every FFP symbol in the RA. This downsweep takes time proportional to the height of the cell in which the TA resides. The parsing upsweep then embeds new nodes within the IN nodes of the TA's subtree, thereby building the SN-CP network. At the completion of this upsweep, the IN nodes--as opposed to the nodes embedded within them--cease to be of use. The parsing rules are summarized in Figure 5, which shows (1) the combinations of channels that are merged to form the nodes, and (2) the channel that leads upward from each node. These rules, together with some straightforward rules controlling the embedding of leaf nodes in

the L cells (see [Tolle 1981]), define the parsing process. Parsing takes time proportional to the height of the machine cell in which the TA resides. The parsing process in Machine II corresponds in purpose to the combination in Machine I of building a directory and of marking; the purpose in each machine is to make the syntactic structure known to the cells that "see" the RA. The parsing in Machine II differs from that of Machine I in its greater uniformity, in the fact that it occurs in every RA (whereas marking occurs optionally, under microprogram control), and in its result: an embedded network rather than a directory (a set of values in the T cells) and a marking (a set of values in the L cells).

It can be shown that partitioning and parsing together embed no more than 3D+5 nodes in any T cell and no more than 3D+12 nodes in any L cell. A value of 5 for D yields the same parsing depth as in Machine I.

Figure 6 shows an SN-CP network for this RA:

(SP < < 3 2 8 > < 6 4 5 > >).

In Figure 6, the parsing depth, D, of the machine is assumed to be 5 or more, so that no "flat" nodes are involved. For visual clarity, the IN nodes are not shown. Figure 7 shows the same SN-CP network, without the machine cells.

Notice that there is an SN ("syntax node") corresponding to each atom and to < 3 2 8 > and to < 6 4 5 > and to < <3 2 8 > < 6 4 5 > >. The TA node corresponds to the entire RA. The CP ("combining place") nodes connect together pairs of channels from SN or other CP nodes. There is, between each SN node that corresponds to a sequence and the SN nodes that correspond to the top level subexpressions of that sequence, a binary tree of CP nodes. Such a binary tree of CP nodes is quite useful in a wide variety of computations: it can be used to broadcast or selectively distribute operators and data downward from the "father" SN node, and it can be used to combine together streams of data that flow back upward from the "son" SN nodes.

Reducing the RA

To continue the example of Figure 7, assume that SP is an FFP-primitive operator that takes a "sum of products" of the components of its operand. In the present example, then, the result should be 168, since 168 = (3)(2)(8) + (6)(4)(5). Given the appropriate instructions, the nodes of the SN-CP network can easily and concurrently compute this result. In the SN-CP network of Figure 7, for instance, the 3 and 2 can be multiplied in the CP node just above the SN nodes corresponding to 3 and 2. Then in the next CP above, the 6 and the 8 can be multiplied together. While this is going on, the product of 6, 4, and 5 can be found in a similar way. Then the sum can be computed in the CP node that joins the < 3 2 8 > and the < 6 4 5 >. The degree of concurrency is, of course, much greater if the operand has many subsequences, each with many elements. This trivial computation gives but a glimpse of the potential of the machine for sophisticated transformations on streams of data flowing through the embedded network. Many complex and interesting computations are rather naturally expressible in terms of function calls by SN nodes upon their SN syntactic sons, using the intervening CP nodes for the distribution of operators and data, and for combining the streams of data returned upward by the sons. A "Syntax Tree Language" (STL) for specifying such computations in an SN-CP network is described in [Tolle 1981], where examples are given for sorting, matrix multiplication, and solving simultaneous linear equations.

Here, we merely describe in general terms the conventions adopted in the STL. In Machine II, each FFP-primitive operator is defined by an STL program; each cell of the machine has a copy of these definitions. During execution of an STL program, control passes from the top nodes of the SN-CP network downward, via function calls. Control and data are eventually returned upward. Execution of the RA begins in the TA node, which calls upon its left syntactic son (i.e., the SN node corresponding to the RA's operator expression), to find out what the RA's operator is, and then upon the right son to execute that operator.

There is only one way for a node to communicate with its syntactic sons, and that is to call them via the STL operator CALL_SONS. The effect of a node's execution of CALL_SONS is to send both an operator name and some input data (a single FFP expression of arbitrary length and depth) to each son. Different sons may get different operators to execute and different data upon which to operate. Each SN node waits until it is called--i.e., until it receives an operator name (and perhaps a few short modifying parameters) and the first symbol of its input data. It then executes the operator (which may involve calling on its own sons), thereby consumes the data, and returns a single FFP expression toward the father. The CP nodes have the duty of distributing the operator names and the data downward to the sons being called, and of combining the FFP expressions that the sons return upward. These duties are performed under control of operators supplied to CALL_SONS by the calling node. Each CP combines the two FFP expressions coming up into a single FFP expression, so that the calling node eventually receives a single FFP expression as a result of its call.

Let us consider a simple example. Figure 8 shows part of an RA's SN-CP network. The topmost SN node shown has been called with the operator NN ("Nearest Neighbor") and with the input data < 5 2 4 >. This SN is to compute the square of the distance from the input vector to its nearest neighbor among all the vectors stored below the SN. The SN does so by calling upon its syntactic sons (the SN nodes that see the individual vectors) with the operator VDIST ("Vector Distance") and passing each of them < 5 2 4 > as input data; the CP nodes in between distribute VDIST and the input vector to the sons, and save for later use the MIN operator. Each of the SN sons executes VDIST; this consists in calling (via CALL_SONS) its own sons, passing them the operator DIFF2 ("difference squared"). The CPs in between get the + operator to use in combining together the expressions that come back

upward from the leaf SNs; the CPs distribute DIFF2 to all the SN sons, and they "deal" the elements of < 5 2 4 > out to the sons, so that the i-th son gets the i-th element of the input vector. The leaf SNs, having received the operator DIFF2, execute it and thereby compute the square of the difference between their stored value and the incoming value. These squared differences are sent upward and added by the CPs. Thus the SNs above the leaf SNs each receive a single value: the square of the distance from the input vector to the stored vector. These numbers are sent upward, and the CPs execute MIN, so that the topmost SN finally receives the minimum of the squares of the distances; it can then send that value upward toward its father. (The operators can easily be modified to return not only the minimum value, but also the index of the corresponding vector.)

Overview of the Operation of Machine II

What is stored in the L array is one or more FFP expressions, each comprising a single user-program of the form (u d), where u is an operator expression of the form < QUIT jobinfo userinfo >, holding some identifying information about the job and the user; and where d is the FFP program the user wants executed. The operator QUIT is executed, of course, only after d has been reduced to a constant.

Assume that some user-programs have been brought into the machine and stored in the L array, and that the machine has been running for a while. The activities of the machine are cyclic. Figure 9 shows how the steps of the cycle are related. The description begins with the most easily identified point in the cycle:

1. [Polling] The topmost MED node in the machine sends a polling signal down its M channels (see Figure 1). This signal spreads to all the TAs in the machine.

2. [Finding molten zones] Every TA responds immediately to the polling signal by sending one of three status indications to its father MED: (a) the RA has been reduced; (b) the RA needs storage now; (c) the RA is still executing, and storage is not needed at this time. On the basis of the status information, the MEDs locate the so-called molten zones of the machine: those regions outside the RAs with status (c). The RAs with status (c) continue their execution without interference. Any RA with status (a) or (b) becomes part of a molten zone, as does some of the non-RA text in the machine. The molten zones are the regions of the machine that are ready to have their text moved to provide storage where it is needed. Corresponding to each molten zone is a unique MED node that sees that entire molten zone; that MED node supervises steps 3 through 8 of the cycle for that zone.

3. [User-program removal] In each molten zone, each completed user-program is detected and removed from the machine--i.e., the result (the final expression) is returned to the user and erased from the L cells.

4. [Compaction] (This is a step that is desirable if the FFP text representation described in [Tolle 1981] is used. It is included here for compatibility of step numbers.)

5. [Insertion decisions and denial decisions] Each molten zone decides whether it has too little space to honor all its storage requests, exactly the required amount, or an abundance. If there is too little, it denies the requests of some RAs; they wait until next time. If there is exactly the required amount, then all the requests are satisfied. If there is an abundance, and if the zone is not entirely inside a user-program, then the number of excess cells is reported. If there are user-programs of the appropriate sizes waiting to be brought in, then a decision is made concerning which of them to bring in, and their sizes are reported back to the top MED node of the molten zone.

6. [Destination computation] Each molten zone--on the basis of the sizes and positions of its undenied storage requests, the sizes of the new user-programs to be inserted, and the locations of the empty L cells--computes where each text symbol presently in the zone should move in order to create empty L cells where they are needed to satisfy all the storage requests of the (undenied) RAs and to provide space for the new user-programs.

7. [Text movement] In each molten zone, the FFP symbols move from their present locations in L, up through the machine tree, and back down to their L cell destinations.

8. [New user-program insertion] The new user-programs are brought in and stored in the space allocated for them.

9. [Partitioning] In each molten zone, when the text has finished moving and the new programs (if any) have been stored, partitioning begins. This process builds a new TA-Mediator network reflecting the new alignment of text in the L array. The partitioning upsweep, in a sense, synchronizes the whole machine, terminating in the root cell only when all the molten zones have finished their text movement and been re-integrated into the global network (i.e., the new TA-Mediator network). The topmost MED in the machine can then initiate another polling signal to start the next cycle.

10. [Parsing] As each TA is found on the partitioning upsweep, it notices the status of its RA. There are three possibilities: (a) new RA; (b) old RA, storage having been allocated; (c) old RA, storage having been denied. In case (c), the TA simply has to wait for the next polling signal to come down, and then ask again for storage. In case (a) or (b), though, the TA immediately initiates the parsing process in its area.

11. [RA execution] The RAs are reduced independently of each other, each beginning as soon as its parsing is completed. Within each RA's area, all the nodes are initially quiescent. The TA initiates the execution. It

can activate computational processes in the other nodes of its network. Although the computation can involve the concurrent actions of a large number of machine cells, it is always the case that control eventually returns to the TA, all the other nodes of the RA's network becoming quiescent. At this point, the computation may be complete, or it may merely have reached a point at which additional storage is needed in order to continue the reduction. In either case, the TA waits until the next polling signal and then announces its status.

Steps 2 through 8 are collectively referred to as storage management. In any particular molten zone, steps 2 through 9 are mutually exclusive: each step is completed before the next one begins. However, each molten zone goes through these steps independently of the other molten zones. Step 9 (partitioning) eventually causes the molten zones to go out of existence, as the new TA-Mediator network is built.

An RA, once it has started execution, proceeds without pause through as many cycles as needed, until it either finishes or decides to ask for storage. Its only continuing responsibility to the rest of the system is the trivial one of responding to each polling signal that comes down to the TA.

Comparison of the Two Machines

The fundamental difference between the two machines lies in the way they represent and manipulate the syntactic structure of an RA. Machine II's embedding of a syntactic network above every RA requires considerably more hardware resources than does Machine I's directory building and marking.

Machine II's STL is more flexible than Machine I's microprogramming language, and more complex, and its library of operator definitions needs to reside in every cell of the machine, whereas the microprograms of Machine I reside outside the machine, to be brought in on demand. These facts imply a need for more storage space in every cell of Machine II.

Storage management is global in Machine I, whereas it is globally coordinated but locally managed in Machine II. Thus Machine I has a better chance of satisfying all storage requests in a given machine cycle. Machine II, on the other hand, never interrupts an RA that is busy executing, and--more importantly--Machine II's more powerful STL lets it avoid asking for storage in many situations that require storage in Machine I.

Machine II's movement of text during storage management takes place through the tree rather than directly from L cell to L cell as in Machine I, thus obviating the need for connections between neighboring L cells and simplifying the layout problem.

Summary

Magó has proposed a computer architecture [1979] that is a radical departure from that of the conventional von Neumann machine. This paper describes an architecture much like Magó's in some fundamental ways, yet differing substantially from it in others. The central problem dealt with in both architectures is that of coordination of computation in a very large fixed network of essentially identical processors. Both machines are driven by FFP programs, which can specify a very large number of non-identical, independent concurrent computations. In each machine, the fixed binary tree of machine cells adapts itself dynamically to the changing requirements of the programs being executed, automatically allocating processors and storage space to the computational tasks.

In Magó's machine, most of the complexity and computational power reside in the L cells--the leaf cells of the binary tree. The research described here investigates the possibility of putting more sophisticated computational power in the T cells of the machine, specifically by imposing on the binary tree of machine cells a syntactic structure corresponding closely to that of the FFP program to be executed. That syntactic structure--the embedded SN-CP network--is, in effect, a computer designed and built dynamically and specifically to execute the underlying FFP program. Many highly concurrent and sophisticated computations are rather naturally expressible in terms of operations performed on streams of data flowing through the embedded network. The question of whether the advantages of this approach outweigh the disadvantages (such as the increased complexity of the individual cells) awaits further work.

References

Backus, John W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." Communications of the ACM 21, 8 (August 1978), 613-641.

Magó, Gyula A. "A network of microprocessors to execute reduction languages." Two parts. International Journal of Computer and Information Sciences 8, 5 (October 1979) and 8, 6 (December 1979).

Tolle, Donald MacDavid. "Coordination of computation in a binary tree of processors: an architectural proposal." Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1981.

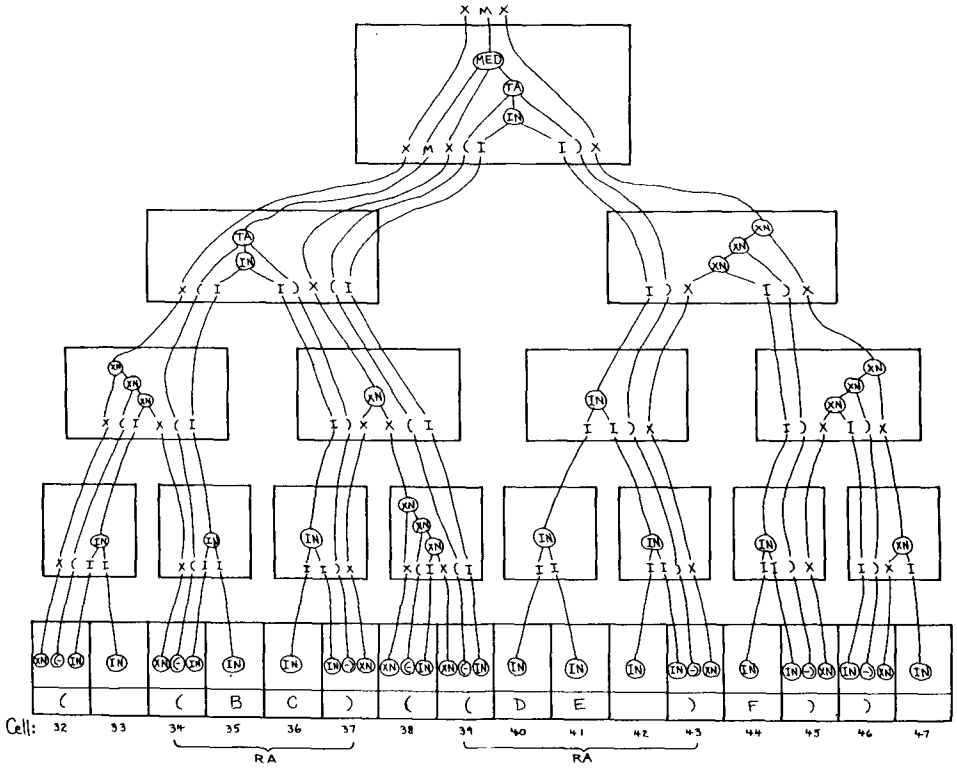


Figure 1: An embedded TA-Mediator network

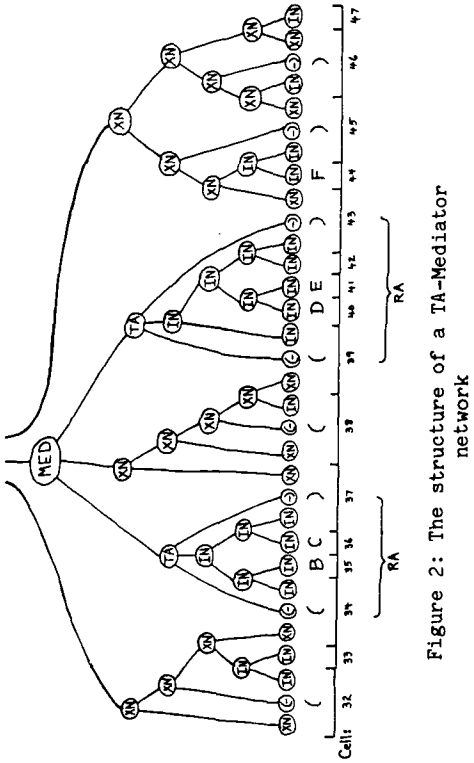


Figure 2: The structure of a TA-Mediator network

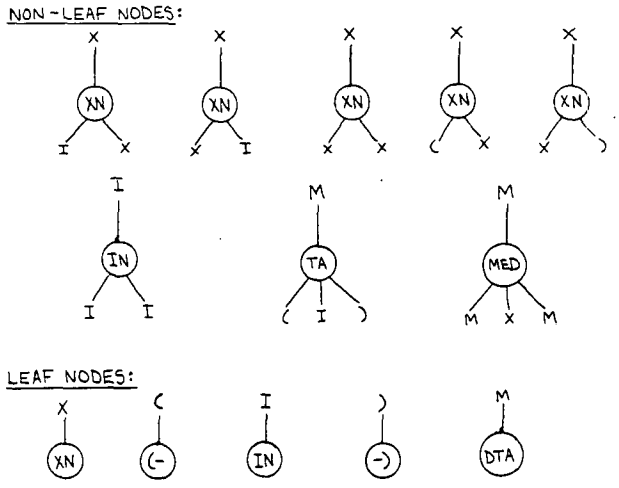


Figure 3: Summary: channels into and out of the nodes

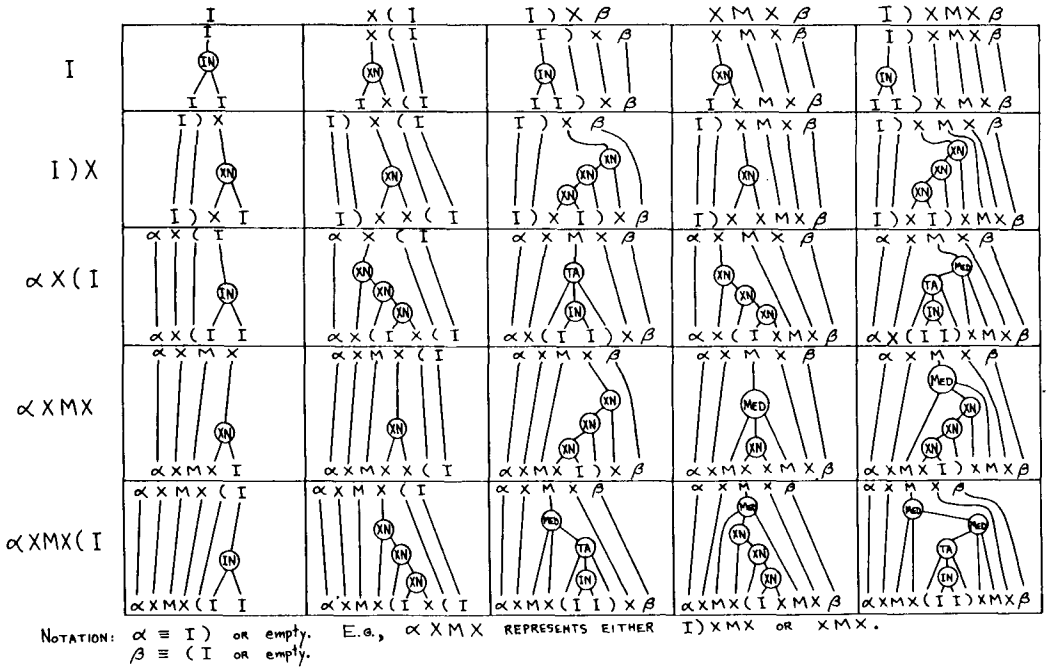


Figure 4: Partitioning configurations for a T cell

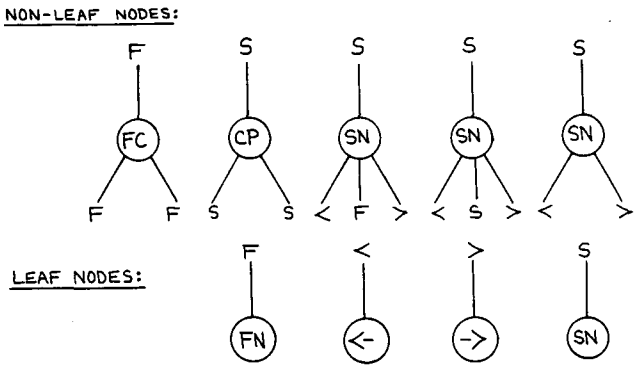


Figure 5: Summary: channels into and out of the parse nodes

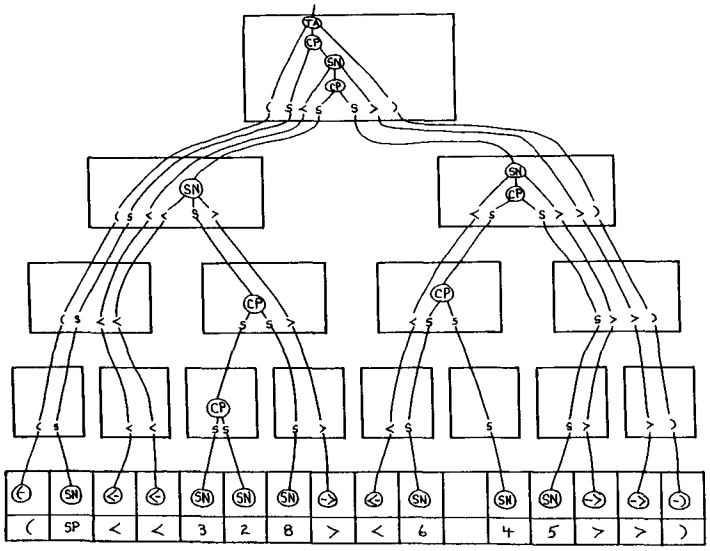


Figure 6: An embedded SN-CP network

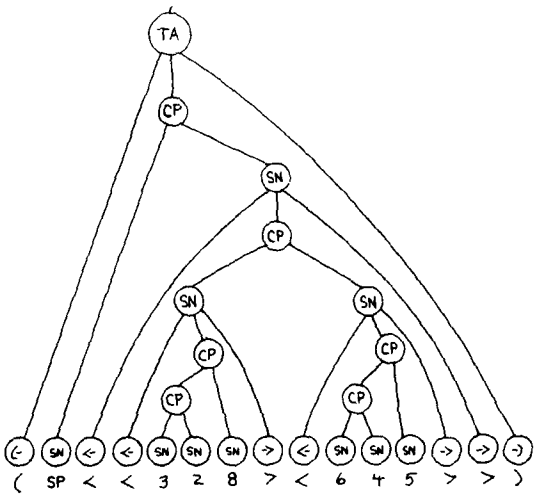
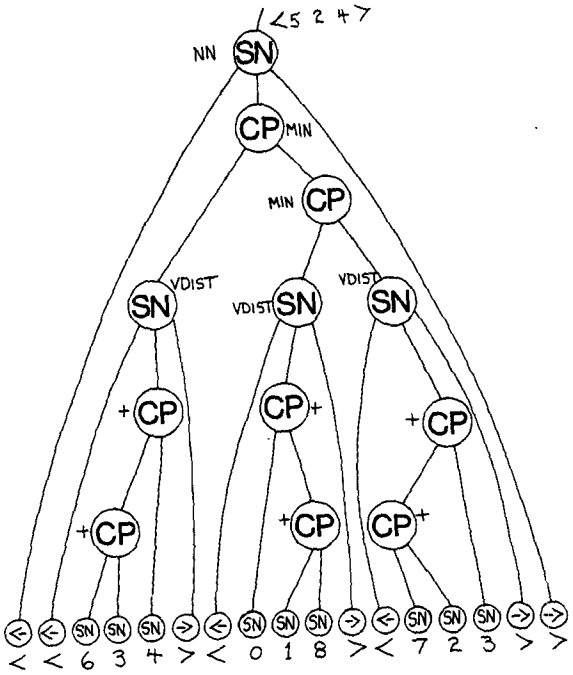


Figure 7: The structure of an SN-CP network



(Each leaf SN node executes DIFF2.)

Figure 8: Nearest Neighbor example

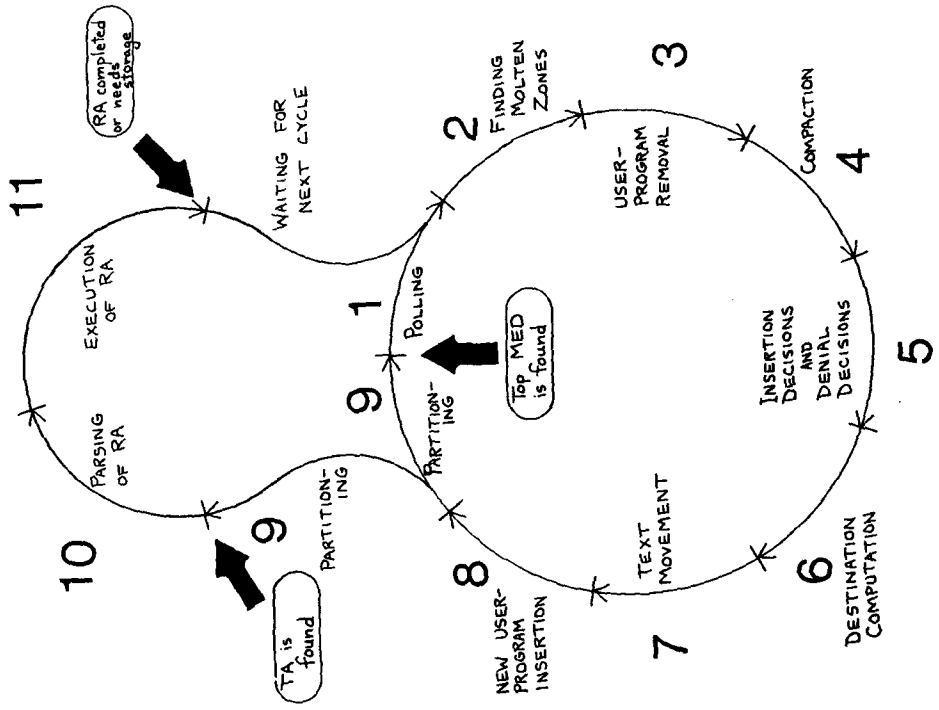


Figure 9: The execution cycle of Machine II