DIAGRAM PARSING - A NEW TECHNIQUE FOR ARTIFICIAL INTELLIGENCE

Rudi Lutz

1986

Cognitive Studies Research Papers

Serial No. CSRP.054

The University of Sussex,
Cognitive Studies Programme,
School of Social Sciences,

Introduction and Motivation

Many applications make use of diagrams to represent objects and/or knowledge about objects. Examples are electrical circuit diagrams and the control- and data- flow graphs used to represent programs and programming knowledge in the MIT Programmer's Apprentice project [10,11,15]. In many of these applications it is necessary to be able to systematically recognise how some diagram has been built up by piecing together other diagrams. This is analogous to the parsing problem for strings, and this paper will present a generalisation of traditional linguistic chart parsing techniques to cope with the case where the object being parsed is some kind of diagram (a flowgraph) and the grammar is an appropriate type of graph grammar (a flowgraph grammar).

Chart parsing offers many advantages over most other parsing techniques. Because the actions to be done are on an explicit agenda the parser is easily able to work from left to right, or from right to left, or top-down, or bottom-up. Furthermore it is efficient in that it avoids re-doing work which has already been done, since its basic data structure (the chart) is essentially a record of everything the parser has recognised or partially recognised so far. In addition the ability to parse bottom-up is important for applications in which one either may not have a complete "grammar", but wishes to recognise as much as possible, or in which there may be errors in the graph being parsed and one wishes to recognise as much as possible of what is correct. For these applications it is often also necessary to obtain information on all the "near-misses" and partially recognised structures so that advice can be offered on how to correct the graph. Since chart parsing quite explicitly builds structures representing this kind of information, the algorithm discussed here is suitable for these kinds of application.

Notation and Definitions

Plex languages and plex grammars were first studied by Feder [4]. Essentially a plex is a structure consisting of labelled nodes having an arbitrary number, n, of distinct attaching points which can be used to join nodes together. A node of this kind is called an n-attaching-point entity (NAPE), and the graph-like structures resulting from joining several such nodes by their attaching points is called a plex. Attaching points of NAPEs in a plex are not connected directly together, but are connected via intermediate points known as tie-points (see Figure 1). A single tie-point may be responsible for connecting together two or more attaching points of NAPEs. If the direction of the connections is important then the plex is known as a directed plex. We will only consider the special case where each NAPE's attaching points (from now on called ports)are subdivided into two mutually exclusive groups, known as the set of input ports and the set of output ports. Input ports are only allowed to have incoming connections, and output ports are only allowed outgoing connections. Although at first sight it would seem that these are a very special kind of plex, it can easily be seen that arbitrary plexes (directed and non-directed) can be rewritten as an equivalent plex of this restricted type so no generality has been lost (see Figure 2 for details). Webs are the special case in which each NAPE only has a single input port and a single output port. Directed graphs are a special case of webs in which every node has the same

label. Strings are also a special case of web, in which each tie-point only has one incoming and one outgoing connection. Figure 2 also shows examples of these various cases. From now on we will however restrict ourselves to the special case of all these kinds of plex in which each attaching point of a NAPE is only connected to a single tie-point. This type of plex is known as a flowgraph and is a generalisation of Brotsky's definition of the term to allow fan-in and fan-out at tie-points (Brotsky doesn't actually mention plexes or tie-points as each attaching-point of a NAPE in his graphs connects directly to one (and only one) attaching point of another NAPE). Note that in all diagrams of flowgraphs NAPEs will be drawn with the convention that input attaching points are on the left and output attaching points are on the right.

Just as a set of strings can be thought of as constituting a language, so a set of plexes can be thought of as constituting a plex language. Furthermore it is possible to define analogues of the notions of grammar and grammar rules, and to speak of the plex language generated by a plex grammar. Similar remarks apply to flowgraphs, webs, and graphs etc., and in the case of webs and graphs there is now a large body of theory associated with these kinds of grammars (see for example [3,9,12]).

A production in a string grammar essentially specifies how one string may be replaced by another either in producing strings or in recognising them. In plex grammars the same is true but we encounter a difficulty not apparent in the string case which is due to the 2-dimensional nature of the objects being parsed (or produced). In the string case a production like :

A => aXYb

applied to a string ....dAe.... (say) results in the string ....daXYbe.... , and the question of how the replacement string (in this case aXYb) is to be embedded in the host string in place of A never really arises because there is a single obvious (trivial) choice i.e. whatever is to the left of the A in the original string is to the left of the replacing string, and similarly on the right. In the graph case however we no longer have this simple left-right ordering on the NAPEs and this question of embedding becomes much more complicated. Most of the discussion of this topic is in the web and graph grammar literature [3,9,12], but most of it applies (with some slight modifications) to the flowgraph case as well. One possible way of specifying the embedding is to specify with each production that all NAPEs in the surrounding graph which are connected to tie-points in the graph being replaced and which bear a certain label are to be connected to certain tie-points in the replacing graph. This is the approach taken in most of the web grammar literature. An alternative approach (which leads to a nicer theoretical analysis, since it enables one to bring powerful mathematical tools such as category theory to bear on the problem [3]) is to specify with each production which tie-points on the left hand side correspond to which tie-points on the right and then connect everything connecting to one of these left hand tie-points (from the surrounding subgraph) to its corresponding right hand tie-point. This corresponds to a pushout in the category of flowgraphs and, as stated earlier, leads to some nice theoretical analyses of the languages in question. (See [3] for further details). This is the embedding technique used in the work reported in this paper.

More formally a flowgraph grammar is a 4-tuple G=(N,T,P,S) where

N is a finite non-empty set of NAPEs known as non-terminal NAPEs
T is a finite non-empty set of NAPEs known as terminal NAPEs
P is a finite set of productions
S is a special member of N known as the initial (or start) NAPE

The intersection of N and T must be empty.

If we arbitrarily order the input and output ports of a NAPE then each NAPE in a flowgraph can be represented in the form of a triple

(NAPE-label, input list, output list)

where NAPE-label is the label on the NAPE, and input list is a list in which the ith entry is the tiepoint to which the ith input port is connected. Similarly the output list specifies to which tie point each of the output ports is connected. Using this convention a complete flowgraph can be represented as a list (known as a component list) of such triples.

With the above conventions the productions in a flowgraph grammar have the general form

A Li Lo ==> C Ri Ro

where

A is known as the left-side structure,represented as a component list
C is known as the right-side structure, represented as a component list
Li is the left-side input tie-point list
Ri is the right-side output tie-point list.
Lo is the left-side output tie-point list
Ro is the right-side output tie-point list.

Li and Ri must be of the same length, as must Lo and Ro, and specify how an instance of the right-side structure is to be embedded into a structure W containing an instance of the left-side structure which is being rewritten according to the production. The rewriting and embedding is done as follows:

Remove the instance of the left-side structure from W and replace it by an instance of the right-side structure. Now, for each tiepoint X in Li replace any previous connections from NAPEs in W to X by connections from the same attaching points of the same NAPEs to the corresponding tie-point in Ri. Similarly for tie points in Lo and Ro. An example is shown in Figure 3, together with the component list representation for the production and the various flowgraphs. Note that in productions the tie points are labelled by variables, whereas in the actual flowgraphs being rewritten (or parsed) the tie-points are numbered. Also note that for flowgraphs one can eliminate the need for explicit storing of Ri and Ro by simply using the same variable names on the left and right hand sides of the production to denote corresponding tie-points. From now on flowgraphs and production rules will be represented by diagrams as these are easier to grasp, with the understanding that for a programs use they would be represented as just described above.

In the well-known case of string grammars one can identify different classes of languages by the the kinds of productions they have. In particular in a production of the form

X ==> Y

where X and Y represent strings of non-terminal and terminal symbols, then by considering various restrictions on the form of X and Y one can arrive at the notions of context-sensitive, context-free, and regular languages.In a similar fashion one can obtain a heirarchy of flowgraph languages, such as unrestricted, context sensitive, and context free. However in the graph case there is also an orthogonal classification that can be made depending on restrictions on the embedding mechanism. Further details can be obtained in [3]. For our purposes we will content ourselves with noting that restricting the productions to have a single NAPE in their left-side structure gives us the flowgraph equivalent of context-free string grammars.

This paper will from now on be concerned only with the case of context-free flowgraph grammars.

Chart Parsing of Context-Free String Languages

This paper will only give a brief description of chart parsing in the string case as complete accounts are available elsewhere (e.g. [14]). However a brief discussion of the string case is helpful in order to motivate the generalisations needed to extend the chart parsing technique to flowgraph grammars. Chart parsing is essentially a technique whereby assertions about what has been found by the parsing algorithm are kept in a database known as the chart. Such assertions are called arcs (for reasons which will become apparent later) and are of two kinds:

Complete (also known as inactive) arcs, and
Partial (also known as active) arcs.

A complete arc is essentially a statement that a complete grammatical entity (corresponding to some terminal or non-terminal symbol of the grammar) has been found at some point in the string. Partial arcs are assertions that part of some grammatical entity has been found at some point in the string, and these arcs also contain information about what would need to be found in order to complete the grammatical entity. Each arc contains the following information:

1) An arc label. This is the name of the grammatical entity corresponding to the arc, and will normally be one of the terminal or non-terminal symbols of the grammar.

2) A starting point for the arc.

3) A finishing point for the arc.

4) What other arcs are involved in making up this arc i.e. what other grammatical entities (and where they are) were used to recognise this arc.

5) A description of what else needs to be found to complete the arc. In the case of a complete arc this will be empty.

The essence of the chart parsing strategy can then be stated as follows:

Every time a complete arc is added to the chart a search is made to see

if there are any incomplete arcs needing an arc of the sort just added at the appropriate place. If so the incomplete arcs are extended and then added to an agenda to be added to the chart at some appropriate time. Similarly every time an incomplete arc is added to the chart a search is made to see if there are any complete arcs around which are needed by the incomplete arc just added, and if so the arcs are extended and again added to an agenda to be processed when appropriate. One of the main issues here is how to organise the chart so that it can be searched efficiently for appropriate arcs. In the string case it is sufficient to keep two separate databases, one for the complete arcs, and one for the incomplete arcs. If one is only going to parse the string from left to right then it is sufficient to represent each of these databases by an array of length equal to the number of tie-points (gaps between the words). Complete arcs are stored in the appropriate array at the position corresponding to their starting point. Incomplete arcs are stored in the appropriate array at the position corresponding to their finishing point. If one is going right to left as well then it becomes neccessary to index the arcs by their other end as well, and two more arrays can be used for this purpose. For greater efficiency each array entry can be a hash table and complete arcs are stored by hashing on their arc label. Incomplete arcs are stored by hashing on the label of the first arc needed to extend the incomplete arc further.

A simple example of bottom-up chart parsing in the string case is shown in Figure 4, which shows a simple grammar and string to be parsed, together with the state of the chart at various points after the first few arcs have been added. Each arc is represented in this figure by an arc spanning the appropriate part of the string. Each arc is annotated by its label, and two lists, the first of which represents existing sub-arcs of the arc, and the second of which denotes what remains to be found. The chart is initialised by adding a complete arc for each symbol in the string. When a complete arc is added the productions are scanned to see if there are any kinds of object which can start with the arc just added. As an A can start with a c in two ways we add two incomplete (empty) A arcs looking for [c] or [cA]. As there are c's at two points in the string these empty arcs are added at both places. Addition of these incomplete arcs results in addition of further arcs as these incomplete arcs "pair off" with the complete c arcs. Some of these new arcs are complete, while others are incomplete. Figure 4 shows the chart at this stage and finally at the stage after these new arcs have resulted in yet more arcs. This process continues until no further arcs can be added. Note that at no point are arcs thrown away, so the resulting chart at the end of the process contains information about all grammatically correct components of the string as well as about all partial components.

The Flowgraph Case

The main change to the algorithm for the flowgraph case is to the notion of an arc. Rather than an arc spanning some substring of words in a sentence an arc should now be thought of as a circle surrounding a subgraph of the graph being parsed. Such circles will be called segments to avoid confusing arcs with edges of the graph. Segments in the flowgraph case will consist of the same components as arcs in the string case, the main change being that the notion of starting and stopping point must be generalised to allow sets of left-hand (input) (tie-)points and sets of right-hand (output) (tie-)points. Furthermore the description of what is needed to complete a segment (in the case of

partial segments) needs to be generalised to an arbitrary flowgraph structure rather than a simple string. Notice however that this means that more than one component may be simultaneously required by an active segment rather than just one as in the string case. The structure needed will be represented by a component list. With these changes the algorithm is essentially the same as before but with some added complexity in the organisation of the chart to enable efficient retrieval of partial and complete segments. The operation of combining a partial and a complete segment to make a new segment is also much more complex in the flowgraph case due to the fact that there are many more distinctions between different kinds of edges that can be drawn. Furthermore, in the flowgraph case one has to build in the machinery to go right to left as well as left to right. This is due to the 2 dimensional nature of the structures being recognised which means that it is possible to bypass some structure when going left to right resulting in a need to look for things to the left of the structure so far recognised. Figure 5 shows an example of this.

The main change to the organisation of the chart is now that each segment has to be entered in several places since it now has sets of starting and finishing tie-points. So complete segments are entered into the chart indexed by each of its input and output tie-points, and hashed by its label. Partial segments are also entered into the chart indexed by their input and output tie-points, and hashed by the labels of the segments they need which connect immediately to what has been found so far. Note that for efficiency segments are entered into separate arrays corresponding to input ports and output ports, just as in the string case.

The main algorithm is now more or less the same as in the string case. Again there is a change due to the fact that more than one tie-point can be on the leading "edge" of a flowgraph so that when parsing top-down it is neccessary to add empty segments for each possible sub-segment connected to input tie-points of the segment required. When parsing bottom-up it again may be neccessary to add more than one segment for each complete segment added, even if only one production has the complete segment connected to its input tie-points, since the production may have more than one occurrence of this type of segment connected to its inputs.

Entry of segments into the chart has already been described. The only major part of the algorithm still to be described is now the process of joining a partial segment and a complete segment to form a new segment. Figure 6 shows a partial segment being joined to a complete segment to make a new segment (the enclosing box). Input and output tie-points (i.e. those by which the segment is allowed to connect to the surrounding flowgraph) of a partial segment will be described as active if the segment itself is still seeking other components attached to these tie-points. They are inactive otherwise. All input and output tie-points of a complete segment are inactive. On connecting the two segments all the inactive tie-points of the partial segment remain inactive. Some of its active tie-points will correspond to tie-points of the complete segment (this is where the two segments actually join). Other active tie-points may remain active since the segment is still looking for other segments to attach to them. Of the complete segment's (input and output) tie-points some have already been mentioned i.e. those connecting directly to the partial segment. Others will become new inactive tie-points of the resulting segment since it will not be looking for anything to attach to them. However other (input and output) tie-points of the complete segment may now become active (viewed

as belonging to the new segment, since it may now expect other segments to attach to them in order to complete itself. Provided all these distinctions are kept clear there is no great difficulty in the joining operation. Of course, in the case where the new segment is complete i.e. is seeking nothing further, all its (input and output) tie-points will be inactive. Note that this reduces to the normal string chart parsing algorithm if the input flowgraph is a string.

Relation to other parsing techniques

In its top-down strictly left-to-right form chart parsing of context-free string languages corresponds to Earley' algorithm [2]. This was generalised by Brotsky [1] to parsing flowgraphs of the kind described here, except that his algorithm could not cope with fan-in or fan-out at tie-points. The algorithm described here is exactly equivalent to Brotsky's algorithm for the top-down case restricted to the same class of flowgraph languages i.e. those in which each tie-point is connected to exactly one input port of a NAPE and to exactly one output port of another NAPE. The equivalence arises because the segments built by the chart parsing algorithm can be put into a one-to-one correspondence with the (Earley-type) items built by Brotsky's algorithm. However the approach taken here can also easily run bottom-up (which is important for many applications as discussed earlier) but can also cope with languages in which there is fan-in and/or fan-out. Furthermore it is quite happy to have segments sharing structure, which can only be handled in Brotsky's algorithm by having some "meta-level" component with knowledge about when structure sharing might be appropriate (e.g. in programming when results of some computation are passed to more than one place) looking at items on the agenda and taking appropriate action.

Complexity Analysis

I have been unable to reach any firm conclusions on the complexity of the algorithm presented. It is clear that in the string case its complexity is the same as Earley's algorithm (when operating top-down) and its worst case is O(n**3). The way one arrives at this is essentially by looking at how many arcs can start (or end) at a given tiepoint of the string . This is O(n) at the first tiepoint, O(n-1) at the second and so on. This gives O(n**2) altogether. When any of these is added to the chart there are in the worst case O(n) arcs it may be able to pair with and hence have to look at, giving a total of O(n**3) operations. The obvious generalisation of this to the graph case arises from noticing that arcs in the string case are really connected subgraphs of the string regarded as a graph, and so its worst case complexity is dependent on the answer to the question "How many connected subgraphs does a graph have? ". In the case of a complete graph where every node is connected to every other node every subset of the nodes forms a connected subgraph, so in this case at least the algorithm could take exponential time. So are there any useful categories of graphs more complex than strings for which the worst case complexity is less than exponential? The answer to this does not appear terribly hopeful, for even quite simple graphs such as that in Figure 7 can have an exponential number of connected subgraphs since every subset of the set of nodes B can be made into a distinct connected subgraph by adding appropriate nodes from the set A, and hence this graph has at least 2**(n/2) connected subgraphs. However it may be that the generalisation from substrings of a string to arbitrary connected

subgraphs is not the right one. Furthermore the algorithm appears to work well enough in practice, and this may be another example of the well-known result that although Earley's algorithm is O(n**3) in theory no-one has managed to produce a grammar which actually takes more than O(n**2). Finally in many applications the graphs being parsed can be divided quite naturally into smaller pieces which can be parsed independently, and one can then parse the whole graph on the basis of the results of these sub-parses. This certainly seems to be the right approach for programmers apprentice applications where the program falls naturally into loops and procedures etc. and the graphs corresponding to these should be analysed first, as suggested by Waters [15].

Applications

A. Logic Circuit Analysis.

The algorithm described above has been used to recognise circuits such as that shown in Figure 8. The grammar used was that in Figure 9. Although this grammar is not recursive it should be noted that the algorithm can cope with recursive productions quite easily. The implementation of the algorithm allows one to specify that certain gates are commutative in their inputs and the algorithm has been modified slightly to allow diagrams which differ only in which of two (or more) commutative input ports tie-points are connected to. Other than this change the algorithm is essentially "pure". To extend this to deal with more general kinds of electrical circuit (analog as well as digital) we need to introduce something like Sussman's concept of slices [13]. The changes to the algorithm are similar to those needed for overlays in the program understanding case. This is discussed below.

B. Surface Plan Analysis and Program Debugging

Control- and data-flow graphs (known as plan diagrams) have been used to represent programs and programming knowledge in the MIT Programmer's Apprentice project [10,11,15], and subsequently in other work concerned with analysing and debugging programs [5,6,7,8]. In this approach the source program is translated into a control- and data flow graph called the surface plan. Then a large library of "programming cliches" also represented in this form is used to abstract up from the surface plan to a high level description of the program.

Although the basic algorithm as described above can cope with recognising cliches in the surface plan, the full plan diagram formalism as described in [12] also includes the notion of overlays. These provide multiple viewpoints of objects, and enable one to talk of a list say not only as a list, but also as a set if the list is being used to represent a set. In this case the operation of inserting a new element into a list can also be viewed as that of adding a new element to a set (or bag). Some of the relevant plans and overlays are shown in Figures 10 and 11, and part of a surface plan which can be recognised as being an implementation of a set addition operation by means of splicing a new element into a list is shown in Figure 12. Note that this plan quite explicitly alters the cdr function so that it has a new value on some elements of the list, and also alters the car function so that the new pair being spliced in has the appropriate value (the one added to the set) as its value. This paper will not explain in full the meaning of these diagrams, except to note that overlays are similar to productions in that they specify how some flowgraph may be replaced by another in

the parsing process. However they include annotations to describe h
some parts (usually tie-points) of the flowgraph are to be firs
rewritten according to various data overlays (which are not shown in t
figure) and this can often result in new tie-points not present in t
original graph being added to the flowgraph. For instance the use
the overlay in Figure 10 rewrites an iterator consisting of two ti
points (subject to various conditions) as a single tie-poi
(representing a thread which is essentially a linear digraph) n
equivalent to either of the two original tie-points. The parser mu
keep track of where these new tie-points came from so that two overl
segments, one coming from a plan having certain tie points as output
and the other coming from a plan having the same tie-points as inpu
can be connected up properly. Data overlays are applied when a comple
segment is added to the chart and essentially checks that vario
constraints on what constitutes a valid use of the overlay a
satisfied. Once these have been done other overlays such as the on
shown can be applied. Those interested in more details about th
representation of programming knowledge are referred to [10]. A full
description of this and the use of this mechanism for program debuggi
as proposed in [7] will be discussed in a subsequent paper.

Both the original "pure" algorithm and the modified version for t
programming understanding work have been implemented in POP-11 runni
under POPLOG on a VAX-750, and they can do both the examples describ
in this paper as well as all the examples described in Brotsky's pap
[1]. It is hoped that this technique will find many other applicatio
in AI.

Acknowledgements

I am grateful to Richard Lewis for discussion on the question of h
many connected subgraphs a graph has.

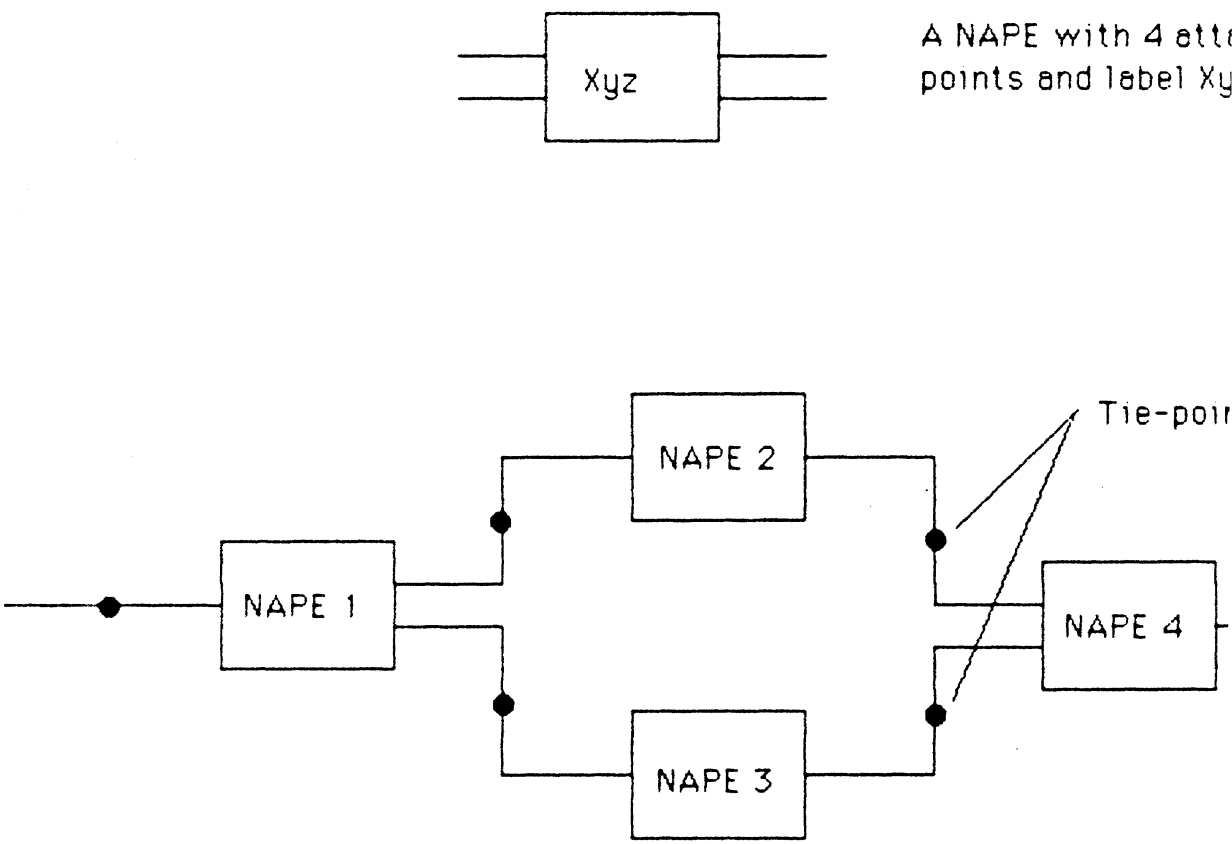
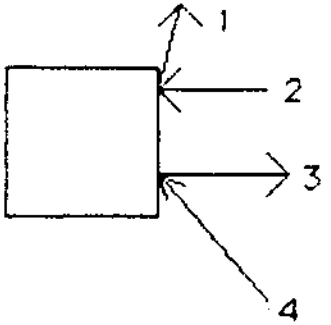


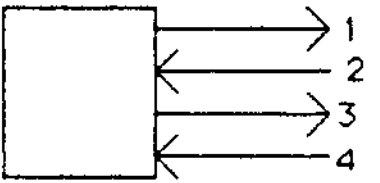
Figure 1 A NAPE and a simple plex illustrating connection of NAPEs via tie-points

Undirected plex

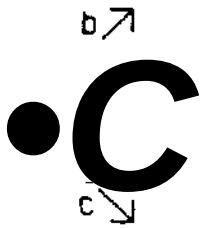
Directed Equivalent



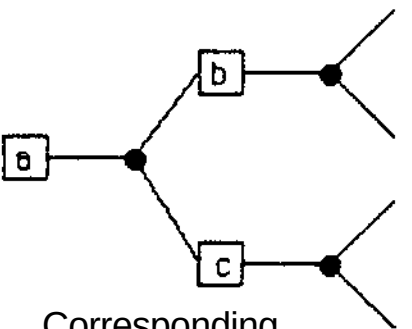
Plex with incoming and outgoing edges at each attaching point.



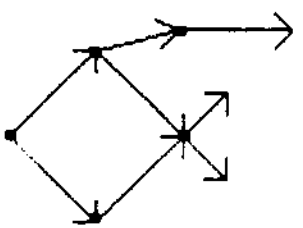
Equivalent plex with twice as many attaching points separated into those with incoming edges and those with outgoing



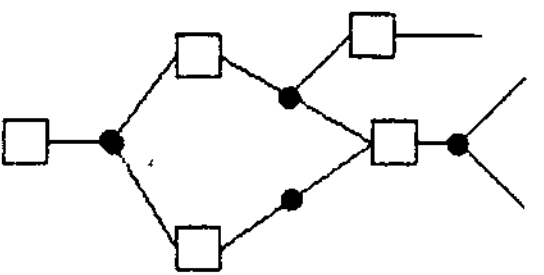
Web



Corresponding Plex



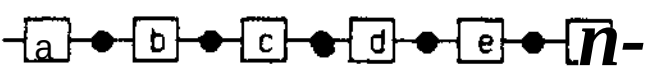
Directed Graph



Corresponding Plex

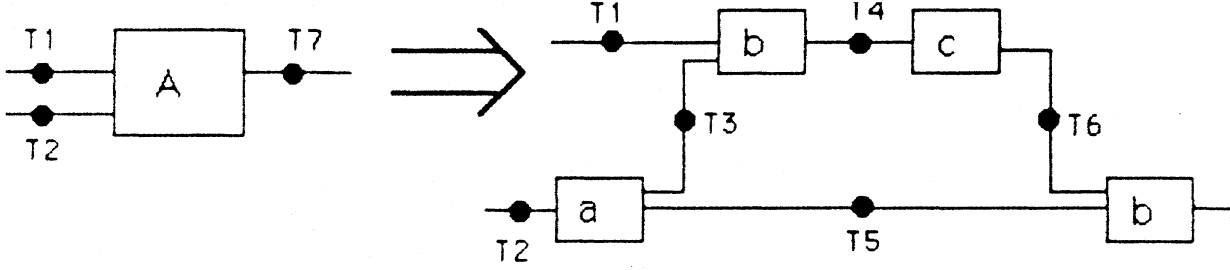
abcdef

String

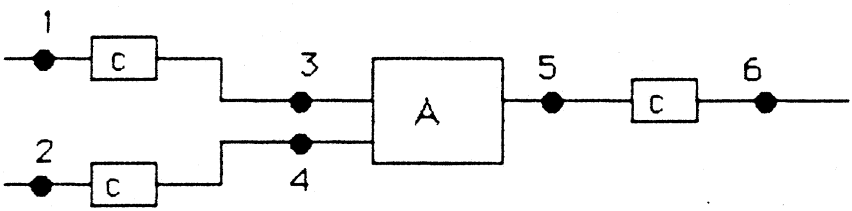


Corresponding Flowgraph

Figure 2

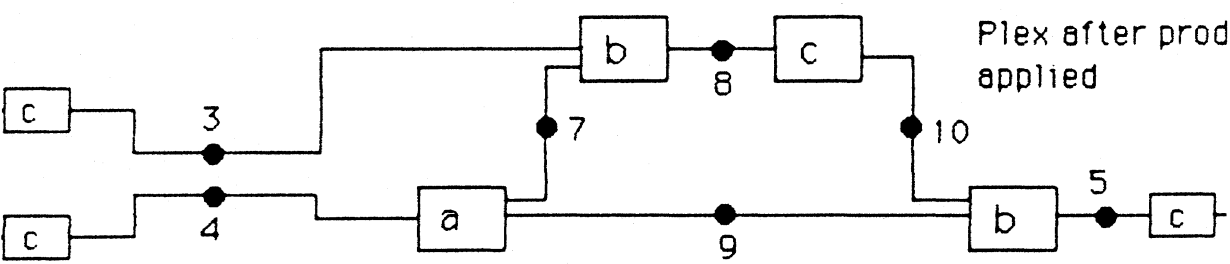


[A [T1 T2][T7]] ==> [[a[T2][T3 T5]] [b [T1 T3][T4]] [c [T4][T6]] [b [T



Plex before production applied

[[c [1][3]] [c [2][4]] [A [3 4][5]] [c [5][6]]]



Plex after prod applied

[3]] [c [2][4]] [a [4][7 9]] [b [3 7][8]] [c [8][10]] [b [9 10][5]] [c [5][6]]]

Figure 3

S ⇒ Aa

A ⇒ c

A ⇒ cA

cca

Initial Chart

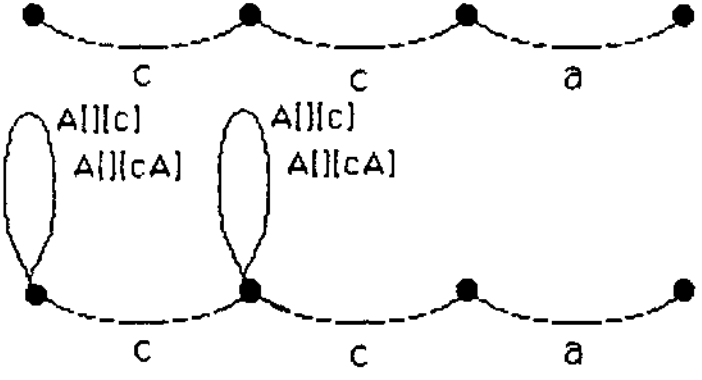
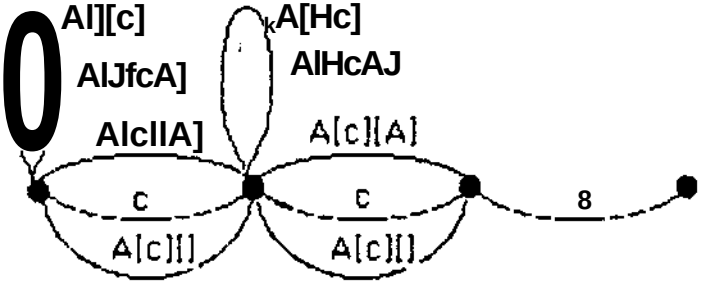


Chart after adding first few partial arcs



Next few stages of the parsing process

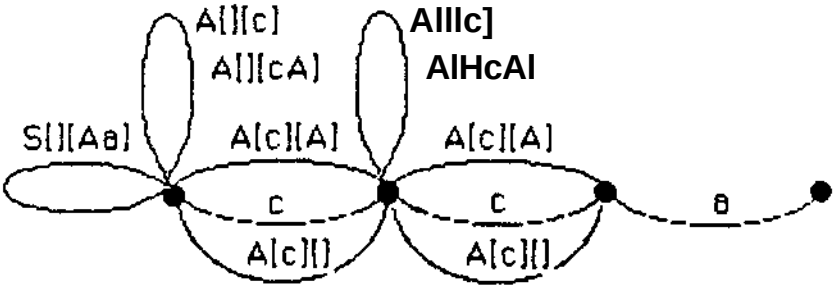
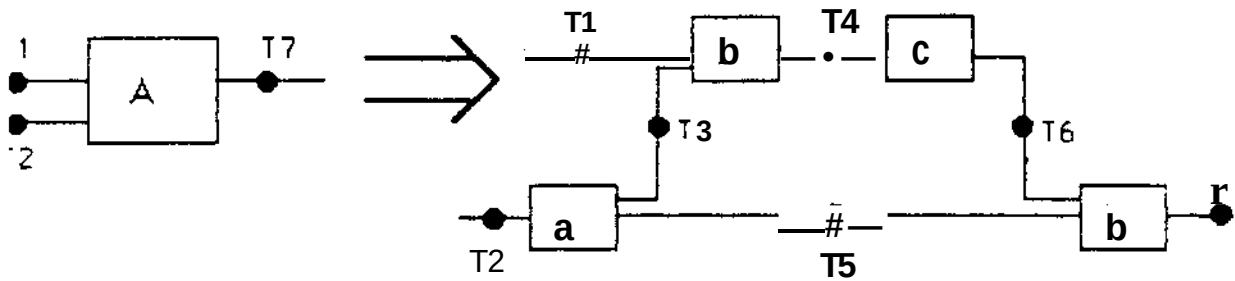
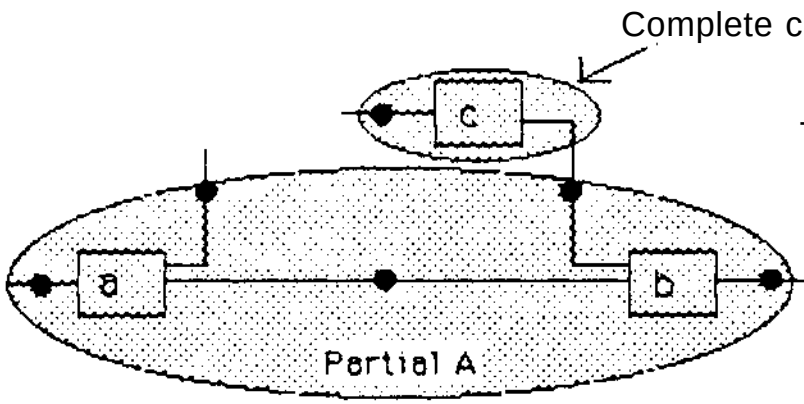


Figure 4 Showing first few stages of chart parsing for string *race*





The partial segmt
requires a compis
segment c to Us

Figure 5

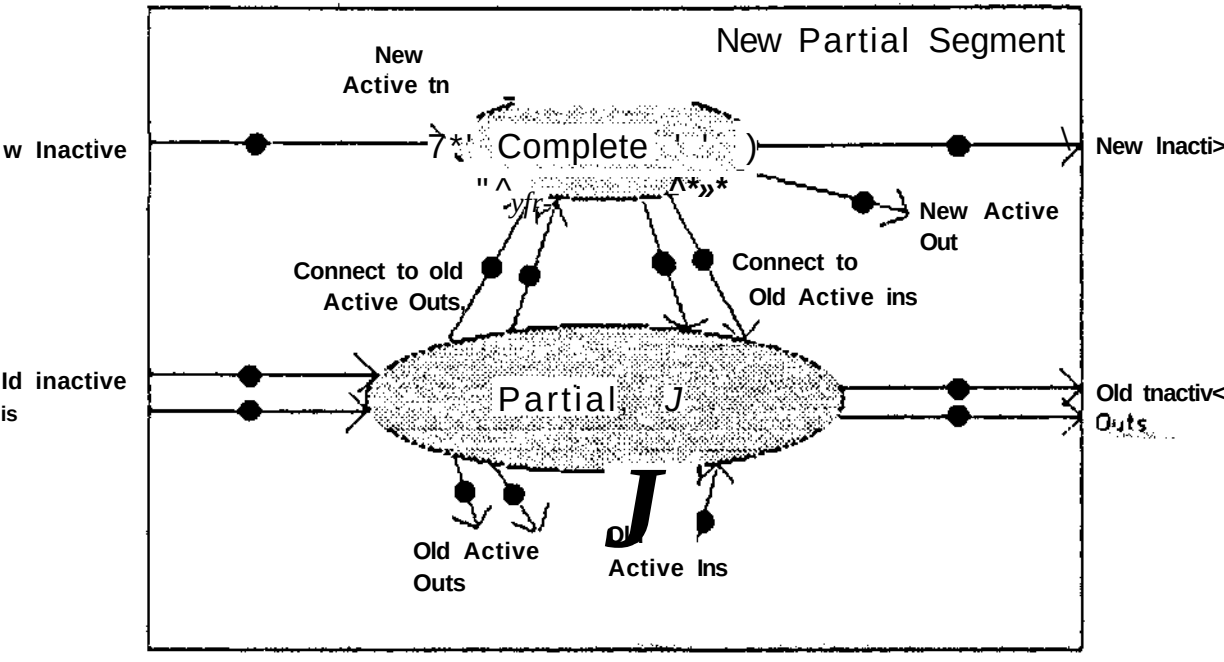
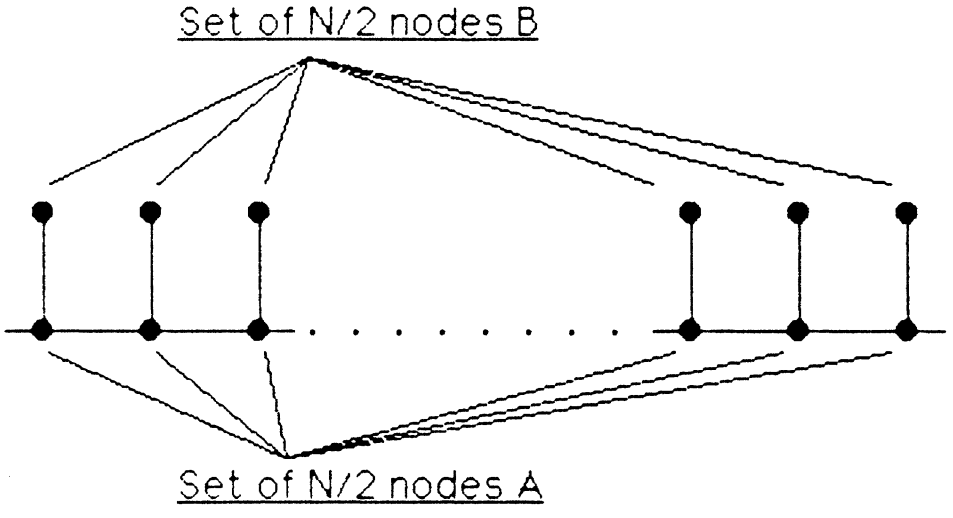
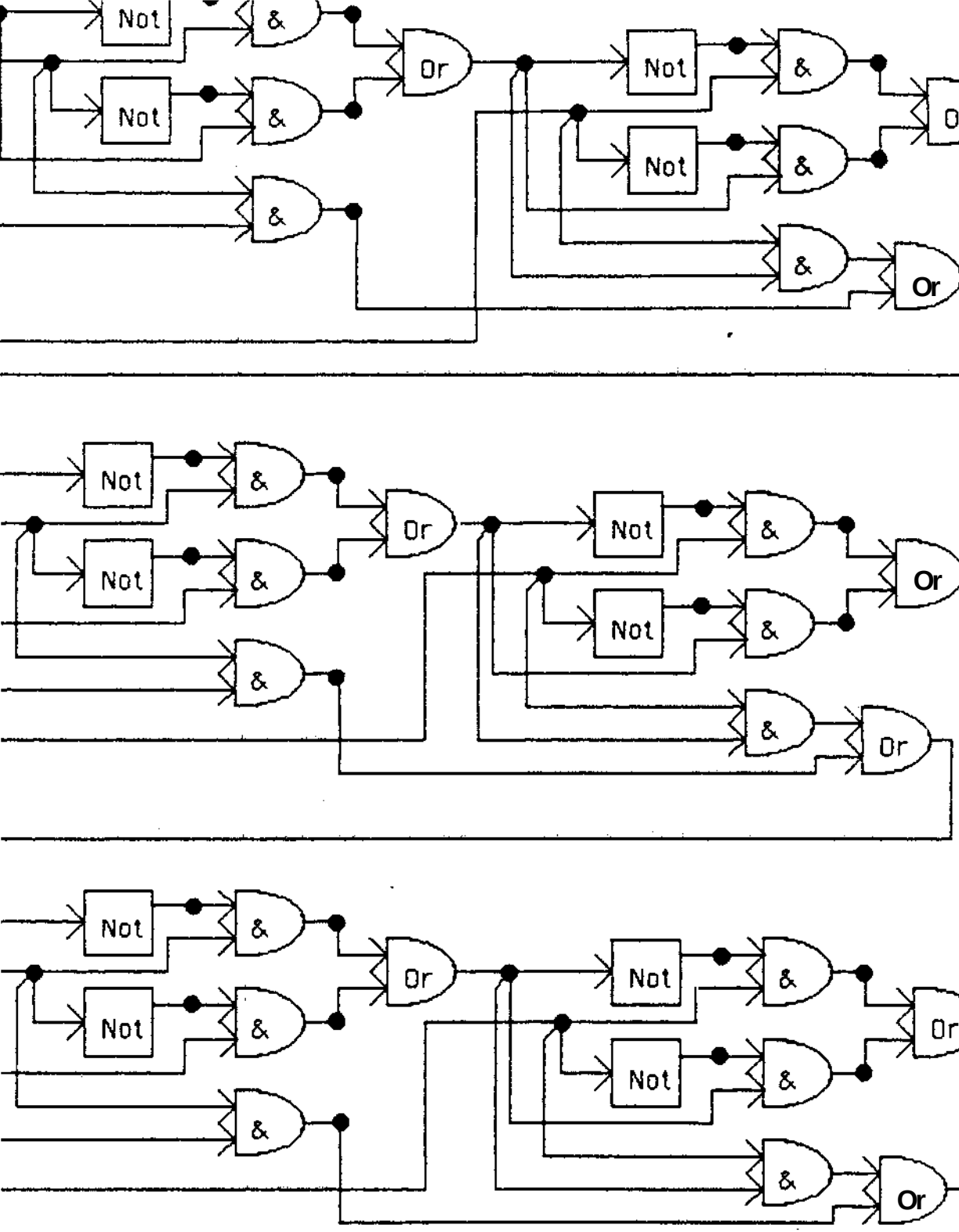


Figure 6
Joining A Partial Segment and a Complete Segment to Make a New Partial Segment.



gure 7 A Graph with N nodes and at least 2^N Connected Subgraphs


Not
Not
&
&
&
Or
Not
Not
&
&
&
Or

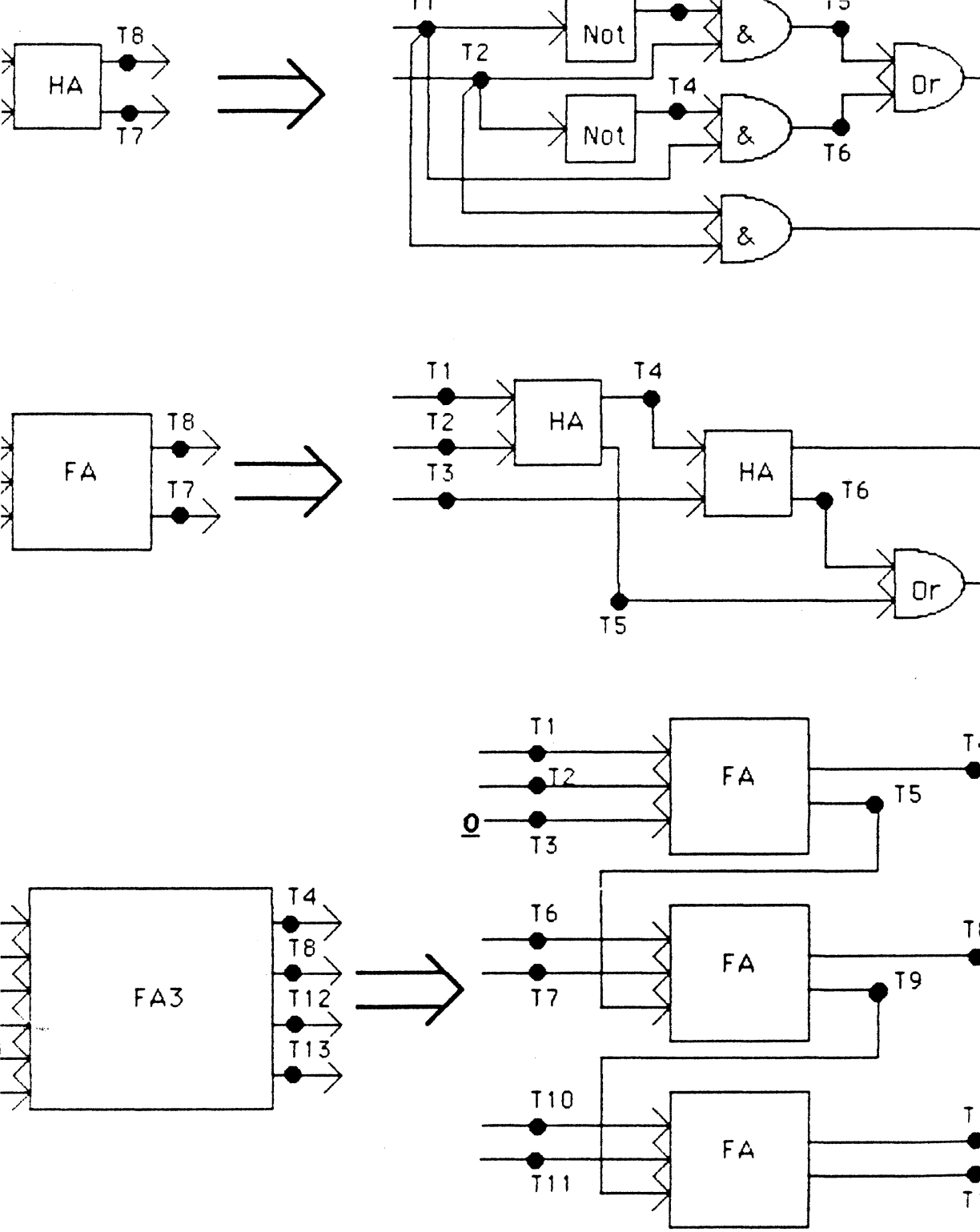
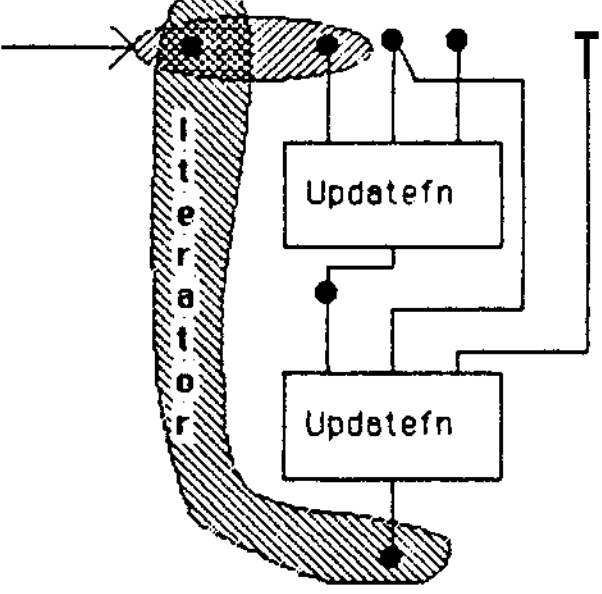


Figure 9 A Plex Grammar for 3-Bit Addition Circuits



Plan SDiicein

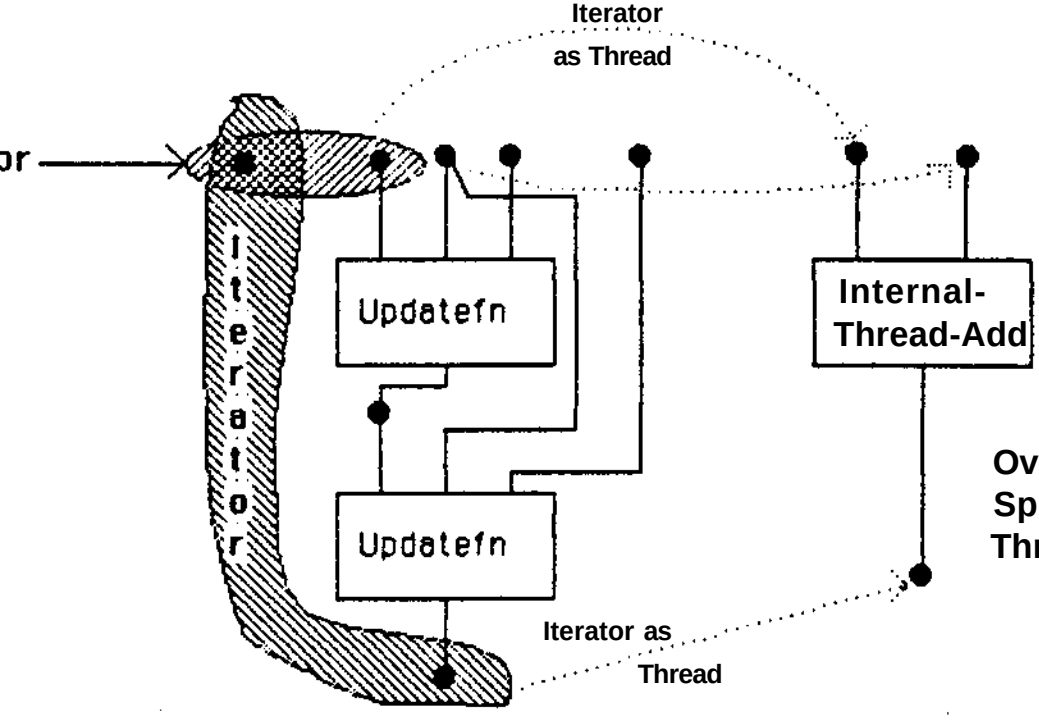
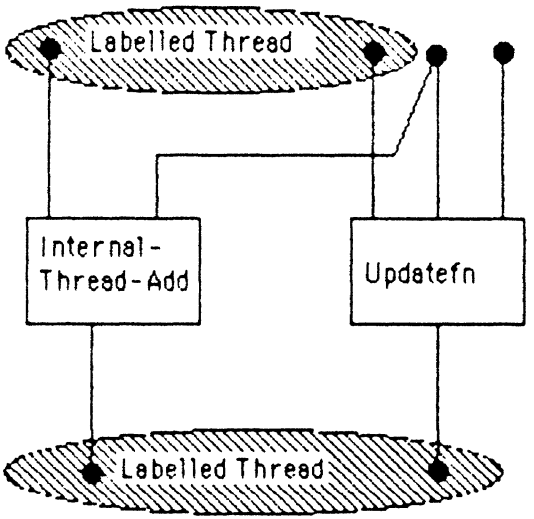


Figure 10



Plan Internal-Labelled-Thread-Add

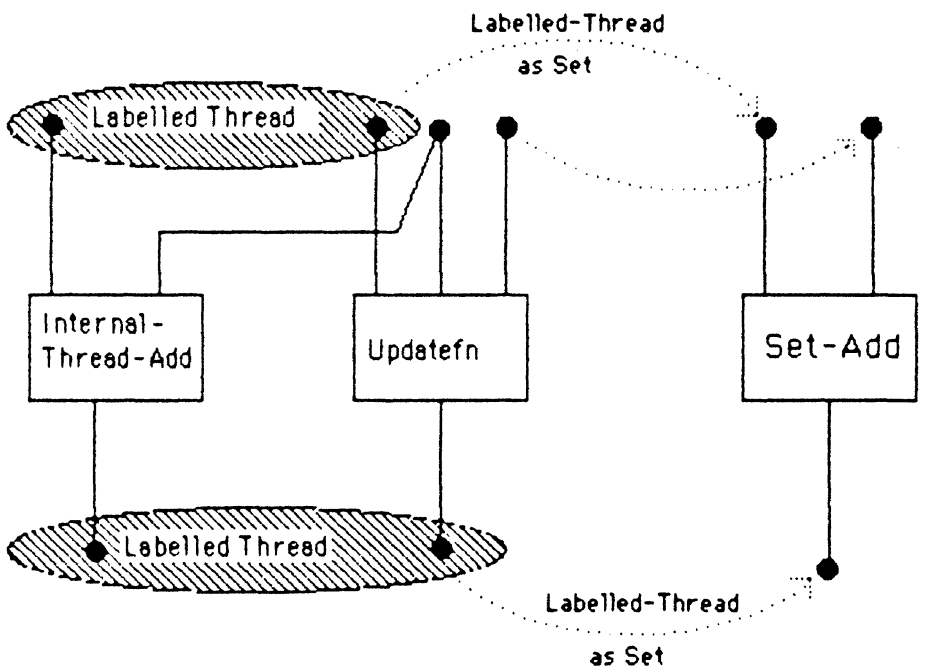


Figure 11

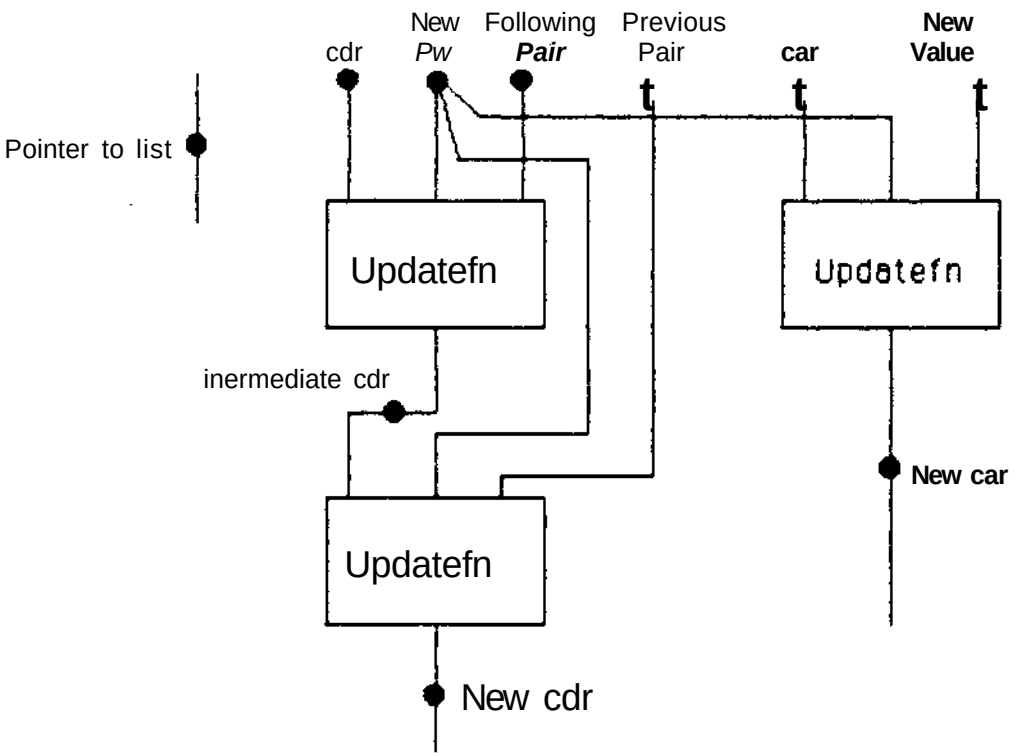

cdr
New
Pw
Following
Pair
Previous
Pair
car
New
Value
t
t
t
Pointer to list
Updatefn
Updatefn
inermediate cdr
New car
Updatefn
New cdr

Figure 12

References

1. Brotsky, D.C. An Algorithm for Parsing Flow Graphs. Technical Report AI-TR-704 MIT Artificial Intelligence Laboratory 1984.

2. Earley, J. An Efficient Context-Free Parsing Algorithm. CACM 13(2) pp. 94-102 (1970).

3. Ehrig H. Introduction to the Algebraic Theory of Graph Grammars (A Survey Graph Grammars and their Application to Computer Science and Biology. (eds. Claus, V., Ehrig, H., and Rozenberg, G.) Lecture Notes in Computer Science Springer-Verlag (1979).

4. Feder, J. Plex Languages. Information Sciences, Vol. 3 (1971) pp. 225-241

5. Lutz, R.K. Towards an Intelligent Debugging System for Pascal Programs: A Research Proposal. Open University Human Cognition Research Laboratory Technical Report No. 8 April 1984.

6. Lutz, R.K. Program Debugging by Near-Miss Recognition and Evaluation. Pro ECAI 1984.

7. Lutz, R.K. Program Debugging by Near-miss Recognition and Symbolic Evaluation. Cognitive Science Research Paper CSRP 44, University of Sussex 1985.

8. Laubsch J. , Eisenstadt M.
Towards an Automated Debugging Assistant for Novice Programmers.
Proc. Artificial Intelligence and Simulated Behavior Conference Amsterdam 1980

9. Pfaltz, J.L., and Rosenfeld, A. Web Grammars. Proc. IJCAI 1 (1969) pp. 609-619

10. Rich C. Inspection Methods in Programming. MIT Artificial Intelligence Laboratory AI-TR-604 June 1981.

11. Rich, C. and Shrobe, H. Initial Report on a LISP Programmers' Apprentice IEEE transactions on Software Engineering. SE-4(6) pp. 450-467 (1978).

12. Rosenfeld, A. and Milgram, D.L. Web Automata and Web Grammars. Machine Intelligence 7 pp. 307-324 (eds. Meltzer, B. and Michie, D.) Edinburgh University Press 1972.

13. Sussman, G.J. Slices at the Boundary Between Analysis and Synthesis. Artificial Intelligence and Pattern Recognition in Computer Aided Design (ed Latombe) North-Holland (1978).

14. Thompson, H. and Ritchie, G. Implementing Natural Language Parsers. Artificial Intelligence: Tools, Techniques, and Applications pp.245-300 (eds O'Shea, T. and Eisenstadt M.) Harper and Row, 1984.

15. Waters, R.C. Automatic Analysis of the Logical Structure of Programs. MI Artificial Intelligence Laboratory AI-TR-492 December 1978.