California University, Irvine, Department of Information and Computer Sciences, Technical Report

PRODUCTION SYSTEMS AS CONTROL STRUCTURES FOR PROGRAMMING LANGUAGES

Ruven Brooks

Technical Report #99

Department of Information and Computer Science

University of California, Irvine
Irvine, California, 92717

April 1977

Abstract

Production systems have recently found considerable favor as the control structure for systems in artificial intelligence. This work has lead other researchers to suggest the creation of languages which have production systems as a primary or alternative control structure. This paper explores some of the possibilities available in these languages; an experimental language with a production system control structure is presented, along with an example program written in it. Appropriate application domains for languages of this type are discussed, and the potential for using them with future hardware architectures is explored.

I. Introduction

A production system is a control structure for performing computations. It consists of two basic components, a collection of data structures referred to as the Working Memory (WM) (also called string, database, buffer, or context) and a set of condition-action rules. To perform computations, the condition or left-hand part of each rule is checked against the contents of the string. When the match succeeds, the action, or right-hand, side, consisting of one or more operations, is executed. The operations of one rule change the contents of the string, causing successive, different rules to be fired. This cycle continues until the desired result is achieved in the data structures of the string.

The use of production systems (re-write rules) as a notation for expressing computations is one of the most enduring ideas in computer science. The work of Post (1943) using re-write rules to formally define classes of computations predates by nearly a decade the invention of the first programming languages. Recently, a number of systems in the domain of artificial intelligence have made use of this idea. Among other uses, they have been adopted as the control structure for an adaptive game-playing system (Waterman, 1970), in an aide to mass spectroscopy work in chemistry (Feigenbaum, 1971), and in a medical consultation program (Shortliffe, 1976). They have also served as the basis for a number of psychological models; these include models of puzzle-solving behavior (Newell & Simon, 1972), visual imagery tasks (Moran, 1973; Farley, 1974), and programmer behavior (Brooks, 1975).

As the range of the work mentioned above indicates, production systems as control structures are currently of considerable interest for work in artificial intelligence. In all of these systems, however, the production system control structure was created on an ad hoc basis for the particular task. Recently, however, several workers have constructed languages which use production systems as replacements for or adjuncts to conventional control structures (Galkowski, 1976; Rychener, 1976; Bobrow and Winograd,1976). The purpose of this work is to enquire into some of the characteristics of these production system languages, particularly in regard to defining kinds of computations for which they are well-suited.

II. An Experimental Production System Language: EPS

As part of this effort, a language with a production system control flow was been constructed and used to solve a selected set of problems. For speed and ease of implementation, the Experimental Production System language (EPS) was constructed on a LISP base. (Further description of the language and a user´s manual are available in Brooks(1977)). In general form and flavor, it is similar to Newell´s PSG system (1975), a production system facility for constructing psychological models. The language presented here differs from PSG primarily in respects intended to make it suitable for a wider range of programming tasks. In particular, a richer basic data structure is provided, and some of the control structure options in PSG that are primarily of psychological importance have been eliminated.

Description of the Production Language.

In this language, a program consists of productions which are considered to be ordered. The search for left-hand sides that match the WM is done in this order; and the first production found whose left-hand side matches is executed. Search for the next production to fire off begins at the beginning of the ordering.

The nature of this production system language can be summarized by describing 1) the structures of the data elements in the WM, 2) the structure of the condition

- - - - - -

I am indebted to David Kiersey for work done in programming the system.

elements, 3) how conditions are matched to data, and 4) the range of permissible actions to be taken when the conditions are met.

1. Structure of the data elements. The contents of the WM are arbitrary list structures, rather than atomic symbols as in the Post productions. Permissible, individual elements of the WM thus might be:

```
(A B)
(A ( B C) D)
```

The entire contents of WM might appear as:

```
(A),(B),(C)
(A (B C)),(D (E) F),(G)
```

2. Structure of the Condition Elements. The left-hand side of production rules do not have to be exact specifications of symbols in the WM to match them, a number of special pattern symbols may be used which permit a given left-hand side to match whole sets of symbols. For example, the pattern symbol, (*ATOM*), matches any single atom. Thus, the construction, (A (*ATOM*)), can be used to match any symbol which has a single letter in the second position, so that this pattern will match the symbols, (A C) or (A E), and it will not match (A (C E)). Similarly, the pattern symbol, (*REST*), matches the tail of any list; as an

example, the construction, (A (*REST*)), will match (A B C) or (A (B C)).

Agreement among parts of successive symbols can also be specified, through a local variable feature. A left-hand side of a rule using this feature would appear as (A (*ATOM* VAR1)),(B (*EVAL* VAR1)). VAR1 is the local variable, and the effect of this construction is to match only those WMs containing symbols in which A and B are followed by the same letter.

These constructs may be used together in various ways in the same left-hand side, and a variety of other special symbols are also available. These include *ANY* which matches either an atom or a list, *LIST* which matches only a list, and *CLASS* which causes the match to be made by testing for equality against a list of specified alternatives. The absence of a symbol in the WM or the presence of one of a list of alternatives in the WM can also be specified.

3. <u>How the Conditions are Matched to Data.</u> The operation of matching symbols in the left-hand side of a rule against the target WM has been altered so that each symbol is matched individually, rather than as a contiguous group. If a match can be found somewhere in the WM for each symbol in the left-hand side, then the rule is fired off.

Essentially, this is equivalent to using the conjuction of a set of context-free rules as the left-hand side.

Matching proceeds by taking each element of the left-hand side in left to right order and trying to find a match for it, also in left to right order, in the WM. Matching is done without replacement; once a symbol in the WM has been used to match one symbol in the left-hand side of a rule, it cannot be used to match a second symbol.

The overall workings of the system can be appeciated from the following example:

```
WM:               (A B)(D E)(A C)
Left side:        (A (*ATOM* VAR1))(A (*ATOM* VAR2))
```

After matching, VAR1 will be bound to B and VAR2 will be bound to C.

4. <u>The rule actions.</u> The range of actions available on the right-hand side of the rule include replacement of a symbol in the WM by a new symbol composed of parts of existing ones. For example, a rule might be:

```
(GOAL (*ATOM* VAR1)) => Replace (GOAL (*ATOM* VAR1))
                        by (OLD-GOAL $VAR1)
```

The "$" symbol is used to mean the "value of" the variable name following. The whole operation is interpreted as "replace what matched on the left side with a new symbol composed of OLD-GOAL and the value of VAR1." Note that composition of the new symbol may use local variables from the match of the left side but that arbitrary calculation is not possible.

In addition, several other new actions have been added. The REMOVE action removes a symbol from the WM and is equivalent to replacement by the null element. The PUSHON and SHOVE actions add new symbols to the left-hand end of the WM. The difference between the two is that the SHOVE operation removes the right-most symbol in the WM to keep the total length of the WM constant, while the PUSHON operation permits the length to increase by one. The REHEARSE operator moves a symbol already in the WM to the left-most position in the WM. Finally, a feature for user-defined operations permits the inclusion of printing and other input-output functions.

Examples of the Production Language in Use

A pre-order tree traversal provides an interesting example of what programs in this language look like. For the tree shown below, a total of 14 productions are used.

-- insert Figure 1 about here --

The first 9 productions are used to encode the tree itself. Production 2 and 5 in Figure 2 are examples of this group and the others in the group have a similar structure. The last five productions, numbers 10-14 in Figure 2, act to control the order in which the nodes are traversed. Note that since these productions are at the end of the list, they will be executed only if none of the productions that describe the tree can be executed.

- insert Figure 2 about here -

If the WM initially just contained the single symbol, (A), the sequence of production firings would be:

1,13,2,13,3,13,10,12,10,11, ...

If the 5 control productions are presumed to correspond to the instructions in a conventional program, the production system implementation is about as long as a "pure" LISP (3 lines) function to perform the same task and considerably shorter than a lower-level language implementation (15 lines - Knuth, 1968). In terms of the execution steps required, a total of 49 productions are executed. Of these, 9 are productions encoding linkages in the tree; the remainder of the executions come from the 5 productions responsible for control functions. While exact comparision is difficult, this is probably the same as the combined total in LISP of recursive calls to the main traversing function and calls to functions to traverse the lists that are used to represent the tree.

III. Selection of Application Domains

Since production systems have been shown to be formally adequate to represent any computation that can be performed on exisitng computers, a question of interest is whether there are classes of problems or application domains for which production systems are a particularly suitable (or unsuitable) representation. The example presented in the previous section were intended only to convey the general flavor of programs written in a production system language. To make more general statements about appropriate domains for production system programs, an analysis of the particular properties of production systems as languages for stating computations will be presented. (For an alternative treatment of some of these same issues, see Davis and King (1975)).

Control of Sequencing

One of the more immediately visible characteristics of production system languages is that processes are stated as collections of independent actions. Sequences of actions occur because each step in the sequence leaves behind in the WM the precise set of symbols necessary to invoke the next step. One determinant of the suitability of a production system representation is the extent to which constructions of special, unique symbols is necessary to insure proper control flow; the more such symbols are needed, the less suitable the production system representation. As an illustration, compare the operations to be performed in computing payroll deductions with those required in a hypothetical system to automobile repair diagnosis. In the payroll case, the working balance alone is insufficient to determine which deductions have been made so far; the amount, $748, gives no indication whether the health plan deduction has been made or not. To step through all the deductions, each deduction must leave behind some kind of marker indicating that it has been completed. Thus, arbitrary symbols would be needed for each deduction, such as:

(DEDUCTION-PERFORMED HEALTH-PLAN)

Compare this with a payroll program in COBOL in which the statement sequence keeps track of what has been done; no such markers are necessary. This implies that production system representations are not particularly suitable for this kind of probelm.

In the repair diagnosis problem, on the other hand, such markers are unnecessary. The intermediate states in the solution can automatically serve as markers to guide the computation. For example, suppose that an intermediate states was represented as follows:

```
WM: (GOAL TEST FUEL-SYSTEM)(LOCATION CARBURETOR)(FUEL-LINE.....
```

This information alone could be sufficient to indicate what operationsor tests are to be performed next, and no special symbols or markers are needed.

While these examples adequately convey informally the differences between the two types of problems, a formal distinction is somewhat more difficult to state. One approach is to consider the flow diagram associated with the computation. Assume that the operations associated with each vertex are available either as simple operations in a conventional programming language or can be peformed as the right-hand size of a single production. (Footnote 2) The case in which the number of vertices is one less than the

------

(2) This is intended to include systems, such as EPS, in which the right-hand size can contain a sequence of actions.

number of edges corresponds to a program in which the statements are always executed in the same sequence and to a production system in which the rules always fire in the same sequence, regardless of the input data. Any differences in the invoking conditions of the rules must, therefore, be for the sole purpose of insuring the set sequence of rule firing. In contrast, consider the computation whose diagram is maximally connected; depending on the input data, the operations can be executed in any order! The differences in the invoking conditions of the rules capture the conditions under which different operations will be performed. In the first case, use of a production system representation is clearly superfluous. In the second it may result in an expression of the computation that is shorter and simpler to read than one written in a language with a more conventional control structure. Hence, the relative amount of interconnection in the flow diagram may serve as a useful guide to the appropriateness of production system representations.

Uniform Program Structure

A second characteristic of production systems which can be expected to strongly impact how they are used is that production system programs are composed of a single statement type, the production. Additionally, there are no subroutines, subprograms, functions, or other program modules. This completely uniform program structure is a source of both advantages and drawbacks.

The drawbacks involved are mainly those that occur because, in a production system program, locality in the program text does not usually coincide with execution locality. The reasons for this is that the same production may take part in a variety of different computational sequences. One problem that this creates is in debugging; it´s often very difficult to localize a bug by narrowing down the particular section of program listing in which the bug occurs. Similarly, it´s difficult to have different parts of the program written and tested by different individuals, a characteristic which runs counter to modern trends in modular programming (Yourdan, 1975).

The advantage of the uniform program structure comes in situations in which ease of program modification is a critical issue in selecting a programming environment. One situation in which this property is frequently of considerable importance is in work in artificial intelligence, since the precise structure of a system is rarly known before construction begins. The particular benefits of production system organization come in adding new pieces of behavior to a program. If the productions are constructed in such a way that each individual production captures some complete unit of behavior, then adding a new piece of behavior, such as a new inference, can be done by simply adding a new production. Several workers in the field (Newell and Simon, 1972; Shortliffe, 1976) have commented on how advantageous this property is for incrementally shaping behavior of a system.

A second situation in which this uniform program structure is desirable is in those situations in which a program is intended to be self-modifying. This ability may be required both in programs which are explicitly intended to "learn," as in pattern recognition, and in programs which must adapt to changing circumstances, such as self-optimizing programs. To have a program modify itself, two, related pieces of information are necessary. First, the particular, logical part of the algorithm that must be altered to produce the overall change in program behavior must be identified. Second, the place where that part of the algorithm is implemented must be located in the program code. How easily the first piece of information can be found will be almost entirely a function of the particular algorithm and task domain. Ease of finding the second piece of information, on the other hand, will depend to a significant extent on the programming language and control structure used to implement the algorithm.

One common approach to building self-modifying programs is to formulate them as interpreters driven by the contents of data structures. Two examples of this approach are a table-driven BASIC interpreter and an array-based FORTRAN pattern recognition program. These systems have the advantage that modifying the behavior of a program is simply a matter of changing the proper entry in the data structure. The correspondence between the entry in the data structure and the program behavior is usually easy to find. On the other hand, the range of possible alteration in program behavior is usually quite constrained.

A way of overcoming this difficulty is to use a single, uniform construction for the entire program and then to provide primitives in the language for manipulating this construction as data. If the primitives are suitably chosen, this guarantees that the program can re-write itself in any way that a human programmer can, while still keeping the actual modification a simple and straight-forward process. An example of such a system is the LISP s-expression and evaluation mechanism.

A similar capability can be added to production system languages by providing right-side primitives for adding and deleting productions. The addition primitive would take new productions composed in the WM and add them to the list of productions to be searched. Similarly, the delete primitive would find and remove a production from the list. In comparision with LISP, making program modifications in this manner will probably be much easier, since there is no need to search through deeply nested pieces of programs to find the site at which the modification is to be made. Waterman (1975) has used just such a mechanism to build a system which acquires serial patterns. Production system languages are, thus, particularly suited to situations in which self-modifying programs are desirable and in which the programmer desires the minimum constraints on which aspects of program behavior can be altered.

Response to External Events

A final, important property of productions systems which is relevant to their usefulness for certain domains of tasks is their ability to respond to external events. At the hardware level, external events are signaled by means of interrupts which start execution of routines to save the status of the interrupted computation and to respond to the external event. In higher level languages, two approaches have been taken. One, typified by some FORTRAN and BASIC laboratory systems, is to discard the notion of interruption and replace it by constructions which cause the program to explicitly wait for the event to occur. The other alternative is to provide constructs which cause specified procedures in the program to be executed when the event occurs.

In production system languages, external events can be responded to by having them place symbols into the WM. Productions sensitive to these symbols then perform the needed computation. This mechanism requires no special actions to save the state of the ongoing computation. In comparision to the use of explicit waits, it preserves the asynchrony of the ongoing computation and the interrupted event. By appropriately ordering productions, a priority of the external event can be specified relative to other computations; constructions such as the ON <event> statement in PL/1 do not usually offer this capability. The only major drawback to using production systems in this way is one mentioned in the section on the drawbacks of having only a single statement type, the inability to control the way in which programmers pass information between sections of program. In this context, it means that the language provides no safeguards against the writing of responses to external events which damage or destroy an on-going computation. If this problem can be dealt with in other ways and if the simplicity and power of the way in which production systems can handle external events is attractive, then production system languages should prove useful for situations in which response to external events is required.

IV. Production Systems and Data-flow Architecture

So far, this discussion has dealt with the characteristics of production system languages only from the viewpoint of their power as a programming tool; tool; nothing has been said about program efficiency. Leaving aside for a moment the comparative efficiency of implementations in production systesm as versus conventional languages, the speed with which a production system program can be executed will depend on two components: the time spent searching through left-hand sides to find the next production to fire and the time spent carrying out the actions on the right-hand sides. The time spent in searching through the left-hand sides will, like other search processes, depend on the implementation. If hashing or indexing can be used, the time will be a constant regardless of the number of rules. In the worst case, it will, of course, be a linear function of the number. Whether or not the constant time schemes can be used will depend on the design of the particular production system language, and those language designs which are desirable in other respects are not guarenteed to lend themselves to these schemes. Since the search is an inherent part of the control structure, production systems may possess the unpleasant property that execution time is a function of static program size.

Just because a search is an inherent part of a production system control structure does not, however, eternally damn production system programs to be slower than those in conventional languages. Part of the problem with search has to do with the single-processor nature of conventional hardware architecture; appropriate architectures using many processors may reduce the search for the next rule to use to a constant-time process.

A framework for selecting such architectures may lie in the concept of data-flow languages (Dennis, 1974; Kosinski, 1975). In such languages, "sequencing of operations is determined by the availability of data for them, rather than by a separate and explicit locus of control" (Kosinski, p.89). Since sequencing among productions in a production language is accomplished by placing data into the WM, they clearly fall into this class.

Several different architectures have been proposed for executing different data flow languages (Arvind & Gostelow, 1977; Rumbaugh, 1975; Dennis, 1974). To illustrate how these architectures might execute production system programs, the architecture proposed by Arvind and Gostelow will be used as an example. In their design, a large number of processors are connected to what is effectively a large ring bus. Tokens, which are pointers into a large memory, move along the bus to be input to the processors, and the output of the processors is also in the form of tokens. Processors are dynamically allocated to computations by means of distinguished tokens on the same bus structure.

Using their interpreter, a production system language such as EPS might be executed in the following manner: The WM is implemented as a sequence of tokens circulating on the bus. For each production, a set of processors is allocated to check the invoking conditions of each rule against the WM. If any rule matches, tokens are placed onto the bus which allocate processors to carry out the actions of the rule.

Two points need further explication in this description. First, it makes no mention of the ordering of the rules. This could be handled by replacing ordered rules with unordered ones with additional elements to their invoking conditions. (This equivalence can be demonstrated formally.) Second, there may be more rules than there are processors available. If this is the case, then, after a group of processors have completed checking the conditions for one production, they can be reallocated to check those for another production.

If the number of rules is not very much greater than the number of processors, then search for the next rule to be applied should take roughly a constant amount of time.

Hence, when executed on a dataflow machine of this type, the penalty for search that production systems pay on conventional architectures is eliminated. Additionally, this type of architecture raises another intriguing possibility. If the productions are unordered, any rule can executed as soon as its conditions are matched and processors are allocated to it. While this parallel execution would create the need for mechanisms to guard against deadlock conditions, it may open the way for more rapid execution of production system programs than of those written in conventional languages.

Conclusion

As is perhaps the case with other new disciplines, computer science suffers from a slight tendency to view each new advance as useful to a much wider range of problems then it does, in fact, attack. An already classic example is the use of the term, "automatic programming," to describe the first compilers. Condition-action control structures are a concept that is currently receiving considerable attention from workers in the artificial intelligence area. Consequently, there is a slight tendency, perhaps mainly on the part of students and the less sophisticated, to view them as applicable to a wider or different range of situations than will be the case five years hence. This tendency may be enhanced in the case of production system languages by the seductive possibility of hardware architectures that can execute production system programs as particularly high speeds.

While this tendency is, in the main, harmless, it does have one undersirable consequence. When a concept is waxing, it is view with enthusiasm; when it is waning, most of the attention it receives is negatively critical. This paper has attempted to explore a new concept, that of languages with production system control structures. To avoid the cycle of critical boom and bust, this exposition has focused on identifying computations for which a production system control structure is particularly advantageous; these include computations in which there is a high level of interconnection between nodes in the flow diagram, situations requiring flexible response to external events, and situations in which a uniform program structure is desirable. Further work with these languages should lead to experimental validation and refinement of these guidelines as their suitability, or unsuitability, for different classes of problems is demonstrated.

Bibliography

Arvind & Gostelow, K.P. A computer capable of exchanging processors for time. to appear in Proceedings of IFIP Congress 77.

Bobrow, D. & Winograd, T. An overview of KRL. Technical Report AIM-293. Computer Science Department. Stanford University, 1976.

Brooks, R. A model of human cognitive processes in writing code for computer programs, Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University, 1975.

Brooks, R. A LISP production system facility. Technical report, Department of Information and Computer Science, University of California - Irvine, 1977.

Davis, R. & King, J. An overview of production systems. Computer Science Dept., Stanford University, 1975.

Dennis, J.B., First version of a data flow procedure language. MAC Technical Memorandum 61, Project MAC, Massachusetts Institute of Technology, May, 1975.

Farley, A.M. VIPS: A visual imagery and perception system; the result of a protocol analysis. Dept. of Computer Science, Carnegie-Mellon University, 1974.

Feigenbaum, E. A., Buchanan, B.G., & Lederberg, J., On generality and problem solving - a case study involving the DENDRAL program. in Meltzer, B. & Michie, D. (Eds.), Machine Intelligence 6. pp. 165-190, Edinburgh University Press, 1971.

Galkowski, J.T. Prlisp. SIGART Newsletter, No. 57, April,1976.

Kosinski, P.R., A data flow language for operating systems programming. Proceedings of the ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices, Vol. 8, No. 9, Sept. 1973.

Knuth, D. Fundamental Algorithms. Addison-Wesley Publishing Co. 1968.

Moran, T.P., The symbolic imagery hypothesis: a production system model. Computer Science Dept., Carnegie-Mellon University, 1973.

Newell, A. & Simon, H.A. Human Problem Solving, Prentice-Hall, 1972.

Newell, A. & McDermott, J. PSG Manual. Department of Computer Science, Carnegie-Mellon University, 1975.

Post, E. Formal reductions of the general combinatorial problem. American Journal of Mathematics, 65:197-268, cited in Minsky, Marvin Computation: Finite and Infinite Prentice-Hall, 1967.

Rumbaugh, J. A parallel asynchronous architecture for data flow programs. MAC Technical Report 150, Project MAC, Department of Electrical Engineering, Massachusetts Insitute of Technology, 1975.

Rychener, M. D. Production systems as a programming language for artifical intelligence applications. Technical reprot, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa. 15213.

Shortliffe, E. H. Computer-based Medical Consultations: MYCIN. American Elesvier Publishing Company, New York, 1976.

Waterman, D. A. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence, 1:121-170, 1970.

Waterman, D.A. Adaptive production systems. Proceedings 4th International Joint Conference on Artificial Intelligence, 1975.

Yourdan, E. Techniques of Program Structure and Design. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

<u>Productions from the Tree Traversal System</u>

```
2.   Conditions:  (GOAL LEFT-LINK B)
                  (B)
     Actions:     Pushon (D)
                  Replace (GOAL LEFT-LINK B) by
                  (OLD-GOAL LEFT-LINK B).
```

"If the goal is for the left link of B, then push on D and mark the goal for the left link as old."

```
5.   Conditions:  (GOAL RIGHT-LINK B)
                  (B)
     Actions:     Pushon (E)
                  Replace (GOAL RIGHT-LINK B) by
                      (OLD-GOAL RIGHT-LINK B)
```

"If the goal is for the right link of B, then push on E and mark the goal for the right link as old."

```
10.  Conditions:  (GOAL (*REST* VAR1))
     Actions:     Pushon (FAIL)
                  Replace (GOAL (*REST* VAR1)) by (OLD-GOAL
 $VAR1).
```

"If there is a goal for either a right or left link which cannot be satisfied, push on a failure marker."

```
11.  Conditions:  (FAIL)
                  (OLD-GOAL RIGHT-LINK (*ATOM* VAR1))
                  ((*EVAL* VAR1))
                  (OLD-GOAL LEFT-LINK (*EVAL* VAR1))
     Actions:     Remove (FAIL).
                  Remove (OLD-GOAL RIGHT-LINK (*ATOM* VAR1))
                  Remove ((*EVAL* VAR1))
                  Remove (OLD-GOAL LEFT-LINK (*EVAL* VAR1)).
```

"If both the right and left descendents of a given node have been visited or if it has no descendents, remove the node from further consideration."

```
12.  Conditions:  (FAIL)
                  (OLD-GOAL LEFT-LINK (*ATOM* VAR1))
                  ((*EVAL* VAR1))

     Actions:     Remove (FAIL)
                  Rehearse ((*EVAL* VAR1)).
                  Pushon (GOAL RIGHT-LINK $VAR1).
```

"If a node has no left descendent, push on a goal for its right descendent."

```
13.  Conditions:   *ABSENT* (FAIL)
                   ((*ATOM* VAR1))
                   *ABSENT* (OLD-GOAL LEFT-LINK (*EVAL*
 VAR1))
14.  Actions:      Pushon (GOAL LEFT-LINK $VAR1)
                   Print $VAR1 "visited".
```

"If the left descendent of a node has not yet been visited, push on a goal to visit it.

```
14.  Conditions:   *ABSENT* (GOAL (*REST*))
                   *ABSENT* (FAIL)
                   (OLD-GOAL LEFT-LINK (*ATOM* VAR1))
                   ((*EVAL* VAR1))
     Actions:      Pushon (GOAL RIGHT-LINK $VAR1)
```

"If there are no goals active or failed and if a node is present whose left descendent has been visited, push on a goal to visit its descendent."

Figure 2.
Selected Productions from the Tree Traversal Program

(Preceeding a symbol by *ABSENT* indicates that the symbol must be absent from the WM if the match is to succeed. Preceeding a variable name by $, as in $VAR1, indicates that the value of that variable, rather than the name itself, is to be used.)

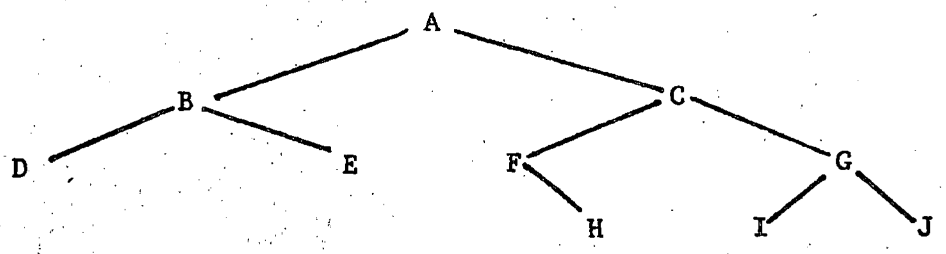

Figure 1.