OPS, A DOMAIN-INDEPENDENT PRODUCTION SYSTEM LANGUAGE

C. Forgy and J. McDermott
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract: It has been claimed that production systems have several advantages over other representational schemes. These include the potential for general self-augmentation (i.e., learning of new behavior) and the ability to function in complex environments. The production system language, OPS, was implemented to test these claims. In this paper we explore some of the issues that bear on the design of production system languages and try to show the adequacy of OPS for its intended purpose.

I. INTRODUCTION

Much of the work that has been done with production systems during the past few years has had as its primary goal the development of systems that are expert in some particular task. The tasks so far addressed include: chemical inference [Buchanan and Lederberg, J 971], medical diagnosis [Davis, Buchanan, and Shortliffe, 1975], discovery in mathematics [Lenat, 1976], speech recognition [Erman and Lesser, 1975; McCracken, 1977], and automatic programming [Barstow, 1977]. Although many of these systems have shown impressive power in the particular task for which they were designed, there remains a question of how suitable the production system representation is for large general problem solving programs.

The Instructable Production System (IPS) project at CMU [Rychener and Newell, 1977] is attempting to answer this question. It has been claimed that production systems are capable of learning in a non-trivial way. If this is true, a production system should be able to learn not only facts, but also new behaviors. It should be able to generalize easily; something learned in one context should be readily accessible in other, only remotely similar, contexts. The learning mechanisms should not be complex; in particular, they should not need to know much of the structure of the rest of the system. It has also been claimed that production systems are capable of functioning in complex environments. If this is true, a production system should be interruptable. It should be able to recognize and react immediately to important changes in its environment, and afterward, to return to its previous task with no loss. While these claims are not without support (some of the expert systems mentioned above are capable of learning, for example), they certainly have not yet been established. The IPS project is attempting to build a production system that displays both characteristics as fully as possible.

The first phase of this project involved designing a production system language whose features support generality. To our knowledge, no one has enumerated or systematically attempted to justify a set of characteristics that are appropriate for such a language, though Newell [1973] and Davis and King [1975] have suggested sets of dimensions that can be used to distinguish production system languages from one another, and Lenat and Harris [1977] have argued that a language designed to support tasks within a particular domain should have characteristics that exploit the features of that domain. This paper first discusses some of the more significant alternatives open to the designer of a production system language, and then describes OPS[2] and argues that it is a suitable language for the purposes of the IPS project.

II. THE ALTERNATIVES

The members of the class of production system languages share only a few common characteristics. All make use of conditional statements called productions, and their interpreters all have similar high level functions. The interpreters have access to two memories, production memory and data memory. Production memory is a place to store the productions and any static relations (e.g., a linear ordering) between productions. Data memory is a place to store the data processed by the productions and any static relations between the data. Most production system languages require the maintenance of some information in addition to that in these two memories (e.g., the name of the last production to fire). In this paper it will be assumed that all this information is stored in a third structure that we will call state memory. The interpreter functions by repeatedly matching the productions against a subset of the information held in the data and state memories, selecting one or more of the productions with true antecedent conditions, and then allowing the selected productions to execute and effect changes to data memory. If changes to state memory are necessary, the interpreter itself makes these. This sequence of operations constitutes what is called the recognize-act cycle.

Beyond this set of common characteristics, production system languages have diverged from one another in many ways. In some, for example, production

1 This work was supported in part by the Defense Advanced Research Projects Agency (F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research.

2 A. Newell and M. Rychener, together with the authors, designed OPS. OPS contains many of the features found in PSG [Newell and McDermott, 1975], PSNLST [Rychener, 1976], and RETE, a system designed by Forgy.

memory can be partitioned to give an effect something like subroutines in conventional languages [e.g., Newell and McDermott, 1975]. Data memory can be partitioned in some others and different access mechanisms provided for each partition [Lenat and Harris, 1977]. Some languages allow every satisfied production (that is, *eVery* production whose antecedent condition is true) to fire on each cycle [McCracken, 1977], others use a few simple decision procedures to choose one or a few productions to fire [Anderson, 1976; Newell and McDermott, 1975], and others use complex decision procedures to choose with some care which productions to fire [Erman and Lesser, 1975]. An automatic backtracking search mechanism is included in some interpreters (these are the "deductive" production systems like Rita [Anderson and Gillogly, 1976] and Mycin [Davis, Buchanan, and Shortliffe, 1975]), while the others make it necessary to program all searches explicitly. Differences such as these can be viewed as creating a space of possible production system languages.

The remainder of this section presents arguments for one region in this space. Because the goals of the IPS project differ from the goals of most projects that use production systems, the arguments differ from those that might be made for other production system languages. We will consider each of the four components of a production system language (the interpreter and the three memories).

The Interpreter

Perhaps the most fundamental consideration in the design of a production system language is the amount of processing that will occur during a recognize-act cycle. Existing production systems vary widely in this. Productions in the HSII system [Erman and Lesser, 1975], for example, accomplish far more in one firing than do the productions in Mycin [Davis, Buchanan, and Shortliffe, 1975]. The production system language should not force a discipline on the user (i.e., it should not force him to perform a fixed amount of processing on each cycle), but it should provide features that make it easy for the user to adopt whatever discipline he finds appropriate.

The amount of processing that will be appropriate on any one cycle is dependent on the current task of the system. If some production is very knowledgeable about a particular situation, it is appropriate that the production take powerful actions. In chess, for example, if a production recognized a book position in the opening, the production should be able to make the move and avoid the interference of other productions that are less able in this situation. If there is no production that understands the situation, however, the system should be more cautious in its processing. If it takes a big step and that step is wrong, it will progress far down the incorrect path before it has a chance to recognize its mistake. Moreover, a big step will make the system less able to use whatever store of information it has about similar tasks because it will be skipping over so many of the points where that information could be appropriate. It will be less likely to notice events such as the sudden feasibility of a new approach to the current task, the arrival of a more important task, or the unexpected satisfaction of a pending task. It can be expected that a large task will be diverse; it will have some sub tasks that are well understood by the production system and others that are difficult or poorly understood. The appropriate amount of processing, then, can vary greatly from minute to minute even in the performance of a single task.

Thus the production system language should neither make it impossible to perform complex actions nor make it economically infeasible to perform simple actions. Unfortunately, these requirements tend to conflict. If the monitor supports complex actions and powerful patterns, there is likely to be significant overhead to the recognize-act cycle. To perform only a few simple actions when the overhead is high may not always be economically feasible. The production system language must therefore incorporate some compromise between unlimited power and potential for short cycle times.

The recognize part of the cycle is potentially more of a problem, but current technology provides a reasonable solution. The problem with using powerful recognition criteria is that even though the productions that contain the more powerful patterns may fire only infrequently, the patterns must be tested on every cycle. Powerful actions, in contrast, have a cost only when they are executed. A solution to this problem is provided in the methods now available that allow quite powerful patterns while making the time to perform recognition almost independent of the number of productions in the system [Forgy, 1977; McDermott, Newell, and Moore, 1977]. These methods do have one major limitation: they do not allow the use of variables in Data Memory.[3]

In summary, the production system language should allow antecedent conditions that are as powerful as possible given the constraint that the time required to perform recognition must be independent of the sizes of data and production memories. The power that should be allowed in the actions is less easy to characterize. While there is not the same efficiency issue here as there is in the case of antecedent conditions (actions have a cost only when they are executed), actions whose power is incommensurate with the power of the match would appear to be unusable. If there are limits to the power of antecedent conditions, it will not be possible to describe the character of a situation fully enough to insure that an ultra-powerful (and hence ultra-specific action) is applied only at appropriate times.

Another choice facing the designer of a production system language is whether to include the backtracking * search feature of the deductive production systems. Certainly search is important in the types of programs most often written in production systems, and for many environments an exhaustive depth first search is appropriate. It seems not to be appropriate for systems that must search through large spaces or that must learn to modify old behavior. If the system is to modify old behavior, including old search methods, the old methods must be accessible to the system, not hidden inside the interpreter.

3 Perhaps the methods could be extended to allow variables in Data Memory, but since the necessary studies have not yet been made, we are unsure.

Data Memory

Complex environments are best handled by systems that use a single uniform data memory of limited size. Any system that is able to function intelligently in many different situations will have available a great quantity of Knowledge, much of which will be useful in more than one situation. If the system is not to be impossibly slow in its response to changing situations, it must have means whereby the information relevant to a new situation can be located immediately. The interpreter provides one such means in the mappings performed during the recognition phase of the cycle. All long term information can be stored in productions whose antecedent conditions express the character of the situations in which the information is potentially relevant. When a new situation arises or an old situation is transformed, the relevant information is found and possibly (that is, if the interpreter so chooses) brought into data memory. Since this approach to storing long term information makes data memory essentially an attention focusing device, there is little advantage to having more information in data memory than can be usefully attended to all at once,

Partitioned data memories are often useful, but the same purposes can be served by the more general mechanism of tagging data. Consider, for example, how a partitioned memory might be used in a deductive production system. In these systems, before execution of each production a new partition can be created, and any assertions made by the production affect only the new partition. If the interpreter later needs to back up and undo the effects of the production, it can do so simply by deleting the appropriate partition and its contents. A production system that performs its own searches can achieve similar results with an unpartitioned data memory by tagging the data to indicate that it is contingent, to indicate what it is contingent upon, to indicate why it is contingent (e.g., that it is a goal or that it is hypothesized but not yet accepted), or any other purpose that might arise. In contrast, since they are supplied with a fixed set of mechanisms for the manipulation of the partitions and their contents, partitions on data memory are often difficult to use for purposes other than those that appeared useful at the time the language was designed.

Production Memory

Production memory should, like data memory, be a single uniform structure; unlike data memory, it should have no size limit. The system would be unable to learn if both data and production memories were limited. It should be a single uniform structure to insure that all potentially relevant knowledge is accessible to the system at all times. Since a system functioning in a complex environment can never know what the state of the world will be from cycle to cycle, it makes no sense to exclude some set of productions from consideration during a particular cycle. To exclude productions Is to limit the amount of knowledge that the system can bring to bear.

State Memory

If production memory is large and if the environment is complex, considerable intelligence will be required to select the most appropriate productions to fire on each cycle. While most of the information on which the selection should be made is found in data memory, the information that is available in state memory (e.g., the set of all currently satisfied productions and information about the past actions of the production system) should not be ignored. Because this information is hidden from the production system in most existing production system languages, the interpreter has to make the final decision of which productions to fire. But certainly the interpreter is less well-suited for such decision making than the production system itself; production systems are, after all, better suited than conventional programs for making quick decisions based on large amounts of data and involving many complex criteria. Thus productions should be allowed to read state memory as well as data memory so that they can have as large a role as possible in the selection process[4].

III. A DESCRIPTION OF OPS

In the previous section we indicated a set of characteristics that seem appropriate for a production system language that is to be used for building production systems capable of generality. In this section a particular language, OPS, is described. With one exception, OPS has these characteristics; the exception is that OPS does not give the production system access to state memory. In the description that follows, we attempt to justify our lower level design choices.

The Data

The data processed by OPS are autonomous, constant list structures. The elements are constant because, as explained above, we do not yet know an economical way to handle variables in data memory. The data elements are autonomous structures for reasons of simplicity. Because variables are allowed in productions, it is possible to implicitly link two assertions simply by including the same unique constant in both. Thus, to allow explicit links between elements would add no expressive power to the language.

The OPS data memory is a set of limited size. The set nature of memory is maintained by automatically deleting elements when identical new elements are asserted. The limited size is maintained by automatically deleting elements when they have been in data memory for some fixed amount of time. Because data are deleted after only a short stay, data memory is a temporary workspace that focusses the system's attention on knowledge that is currently relevant. There is no great significance to the

4 MYCIN, with its meta-rules, is one of the few systems to allow this [see Davis, 1976].

5 A full description of OPS is given in Forgy and McDermott [1976].

6 The amount of time, measured in actions executed, Is specified by the user; the default is 300 actions.

decision that data memory should be a set; there are styles of programming that can be adopted when data memory allows multiple occurrences of elements and other equally good styles that can be adopted when it does not.

The Match

The search performed by the OPS interpreter is complete; *every* legal instantiation of every production is found. This makes OPS quite different from its predecessor, PSG [Newell and McDermott, 1975]. Because it does not perform exhaustive searches, the PSG interpreter may fail to find an instantiation of a production even though one exists. PSG's match algorithm is dependent on the order in which a production's condition elements occur. It was decided that this dependence on order is unacceptable for a production system that is to grow through the acquisition of new productions; to add a suitable production is difficult enough without having to contend with the problem of specifying an order on its condition elements that would be appropriate for any situation that the system might encounter.

The antecedent of an OPS production is composed of one or more condition elements, each of which is a form to be instantiated by one element from data memory. The expressive power of OPS condition elements is greater than that of condition elements in languages like PSG. It is possible to give either exact or (somewhat) inexact specifications for both the shape and content of the data to be matched. Condition elements are, like data elements, list structures. Generally, the shape of a condition element must correspond exactly to that of a data element for the two to match. Two means are provided, however, to allow the matching of the head of a list without fully describing its tail. The symbol "..." is used to specify that the tail of a list is to be ignored in the match. For example, the following condition element will match any data element that begins with "a b"

(a b ...)

The symbol "." makes it possible to specify that a tail is to be matched and then to give information about the content of the tail by following the "." by a pattern.

Several elementary pattern types, called match functions, are allowed in OPS. The most basic match function is the constant, which will match only itself. This is seen in.the example above where the condition element will match only those data elements whose first subelement is an "a" and whose second subelement is a "b" It is also possible to specify that a data subelement must be equal to one of a group of constants, be not equal to a constant, or be not equal to any one of a group of constants. The ANY function provides the first of these capabilities; the NOTANY function, the other two. For example, the following condition element will match any data element composed of two subelements where the first subelement is one of "a", "b", and V and where the second subelement is not a "d"

((ANY a b c) (NOTANY d))

NOTANY with no arguments is particularly useful; it will match any sublement. Thus, the condition element

((NOTANY) (NOTANY))

will match any data element with two subelements.

Because data elements can be arbitrarily complex list structures, the need was felt to be able to specify something about the contents of data elements. The CONTAINS and NOTCONTA1NS functions were included to fill this need. As their names imply, the former allows the specification that the subelement contain (at any level) at least one occurrence of one of some set of specified constants, and the latter that the subelement not contain any occurrences of any of the specified constants. The following condition element, which uses "." as described above, will match any data element of two or more subelements provided the constant "a" occurs somewhere after the first subelement

((NOTANY). (CONTAINS a))

Two means are provided for specifying the relationship among condition elements. Simply writing an antecedent without putting separating marks between the condition elements indicates that all condition elements must be satisfied simultaneously in order for the antecedent to be satisfied. For example, the antecedent

(a ...) (b ...)

will be satisfied if there is an element in data memory beginning with "a" and another beginning with "b". The NOT match function allows another kind of grouping, negated conjunction. An antecedent composed of one condition element and one negated condition element is satisfied when the non-negated condition element is satisfied and the negated condition element is not. Thus, the following antecedent,

(a ...) (NOT (b ...)),

ir which "(b ...)" has been negated, is satisfied only if data memory contains a data element beginning with "a" and no element beginning with "b". Negated conjunctions may of course involve more than just two condition elements. Any group of condition elements (including groups containing other negated condition elements) may be negated and conjoined to any other group. The antecedent condition

(a ...) (NOT (b ...) (c ...))

is satisfied when there is a data element in working memory beginning with "a" but not both a data element beginning with "b" and a data element beginning with "c". When NOTs are nested, evaluation proceeds from the innermost level outward.

Variables make it possible to specify that the content of a condition element is dependent on the content of other condition elements in the same antecedent. There are four types of variables in OPS. The most important type, denoted by preceding the variable name with "=", is the "simple" variable. All occurrences of such a variable must have EQUAL bindings. Thus the antecedent

(a . -x) (b . =x)

will match two elements, the first of which begins with "a", the second of which begins with "b", and both of which have identical tails. The second form of variable, denoted by preceding the variable name with "≠", may not be used unless there is a simple variable with the same name elsewhere in the same antecedent. Any data matching one of these not-variables must differ from the data matching the simple variable. For example, the antecedent

=x (NOT ≠x)

will be satisfied only if there is exactly one element in data memory. The final two forms of variables, denoted by preceding the variable names with ">" and "<", are used in comparing numbers. Like the variables preceded by 'V", use of these is legal only if there is a simple variable of the same name elsewhere in the antecedent. A variable preceded by ">" will match numbers greater than the number matched by the simple variable; a variable preceded by "<", numbers less than the number matched by the simple variable. Thus, the antecedent

=x (NOT >x)

will bind x to the largest number in data memory.

It is often useful to be able to give multiple specifications for a data element. For example, one may want to specify that a data element be one of a set of constants and then bind a variable to the element so that the exact value may be determined; this can be denoted using the symbol *"$"*. For example,

(ANY a b c d e) $ =x.

The symbol "$" indicates that the two condition elements that it separates are both to match the same data element. As other examples, the antecedent

=x -y $ >x (NOT <y S >x)

will bind x to a number and y to the next larger number; the antecedent

=x =y *$ ≠x* (NOT ≠x 8 *≠y)*

will succeed if there are exactly two elements in data memory.

Finally, OPS provides a QUOTE function so that the match functions can be used as constants. For example, to use "(NOT x)" as a constant, one would write

(QUOTE (NOT x))

Conflict Resolution

On each cycle OPS selects a single production instantiation to execute. The selection is a two step process in which first the set of all legal instantiations of all productions are found and then one instantiation chosen from that set. The second step, called conflict resolution. must be performed solely by the interpreter because state memory, which holds the information on which the selection is made, is closed to the productions. The decision not to allow the productions to access state memory wa[r]. based on our (since reformed) belief that sufficiently general selection rules could be built into the interpreter to allow it to function adequately in all situations. Conflict resolution is performed in five steps using a different rule on each step. The first rule is absolute in its effect; if there is no instantiation that meets its condition, the system halts. If a set of instantiations do meet its condition, then the remaining rules are applied, in the order given below, until all but one have been rejected.

1. No instantiation may be executed more than once.

2. The instantiations containing the most recently asserted data are given preference. In comparing two instantiations, the rule first compares the most recent data element of each. If these elements differ in recency, the rule selects the instantiation containing the more recent element. If both elements are of equal recency, the next most recent elements are compared, and so on. If the data contained in one instantiation is exhausted before that contained in the other, the instantiation containing more data is chosen. Only if both instantiations are exhausted simultaneously and no elements of differing recency are found are the two instantiations considered equal under this rule.

3. The instantiations of productions with the greatest number of condition elements are given preference. Negated condition elements, including nested negated condition elements, are taken into account.

4. The instantiations of the most recently created production are given preference.

5. An instantiation is selected at random.

These rules, because they make use of a variety of state information that is not available to the productions, provide considerable additional discriminative power. Briefly, the first rule helps insure that the system will consider information that has not yet been taken into account. The second rule, by giving preference to more recent information, encourages the system to continue to attend to whatever task it is currently engaged in; in addition, if given a choice among several productions which are relevant to the same situation, it prefers the most discriminating. The third rule simply extends the definition of "discriminating" implicit in the second rule. The fourth rule provides a way in which the system can mask older productions with newer, more adequate ones.[8]

7 Recall that reasserting a data element will result in the deletion of the existing element. The instantiations containing the new element will have no inhibitions even though they may be identical to instantiations that have already fired.

8 For a more complete discussion of how these rules provide support for domain-independent systems, see McDermott and Forgy [1977].

The Actions

After a production is chosen, it is executed by individually executing each of its actions in order. The actions are simple functions that can modify the contents of data memory, modify the contents of production memory, or interact with the system's environment.

There are only two methods by which the contents of data memory may be modified; an action may assert a new data element or delete an existing element. This is a complete, if primitive, set of actions. The means by which the elements to be asserted or deleted are constructed are equally primitive. On the action side of a production there are a number of elements similar in form to condition elements. The execution of an action involves instantiating the element, performing whatever operations *are* specified by the element, and then if a value results, asserting that value.

If variables occur in an action element, they are replaced by the values to which they were bound during the match. The ability to bind variables in the antecedent provides a quite general extractor mechanism. The ability to recover the bindings while instantiating a form provides a general constructor mechanism. The list manipulation abilities of OPS are comparable to those of LISP. For example, to extract the CAR and CDR of a list, one writes the condition element

```
(=car . =cdr)
```

To extract other subelements, one can write condition elements like

```
(=car (=caadr ...) ...)
```

Variables and "." may also be used in action elements in order to build lists. In an action element, "." has the effect of stripping away the top level of the following list structure and leaving the etements of the list. Thus, the production

```
(a . =x) — > (a . -x)
```

will do nothing but reassert the matched element. More complex processing is of course possible. For example, the transformation of data performed by the production

```
Ox . =y) —> ( . =y =x)
```

is similar to that of

```
(APPEND (CDR Z) (LIST (CAR Z))).
```

Self Modification

Since extensive use may be made of OPS's list processing capabilities in manipulating productions, only three functions are provided expressly for this purpose. One of these, READP, takes as its argument the name of a production and causes a copy of that production to be deposited in data memory. Once in data memory the production can be processed as any other data element. A second function, BUIl D, takes a data element having the form of a production as its argument and adds it to production memory. The third function, EXCISE, takes the name of a production as its argument and deletes that production from production memory. This is, again, a primitive but complete set of functions. These three functions give the system the capabilities of creating new productions and of modifying existing productions. To modify an existing production, the system would bring the production into data memory with READP, delete the existing production with EXCISE, modify the copy using the general processing capabilities of OPS, and then place the modified production in production memory using BUILD.

Input and Output

As with the other capabilities, only a minimal set of functions are provided for interaction with the outside world. There are two functions, READ and WRITE. WRITE instantiates one or more forms and writes them on user's terminal. READ .accepts one or more list structures from the user and deposits them in data memory.

IV. CONCLUDING REMARKS

OPS has been in use for more than a year. During that time it has been the primary tool of a research group that has as its goal the construction of an instructable production system whose production memory will eventually contain several thousand productions. OPS has proven to be basically satisfactory, but it has not been without its problems. Three problems have been particularly irksome; OPS is slow, it is somewhat inflexible, and the information in state memory is hidden from the system. On a KL version of the PDP-10, OPS executes about 5 actions per second (this figure is almost independent of the number of productions in the system). In the successor to OPS, which is scheduled to be completed in the summer of 1977, we expect a speed increase of at least one order of magnitude. The inflexibility of OPS has caused trouble in several ways. No mechanism was provided to make it easy for a user to modify the set of match functions or the set of conflict resolution rules. Yet it became apparent rather quickly that neither the set of match functions nor the set of conflict resolution rules was completely satisfactory. Some of the match functions (e.g., CONTAINS and NOTCONTAINS) received almost no use, while other functions that appear to have promise have not yet been implemented. In the first version of OPS, a mechanism was provided that allowed the user to add his own action functions. In the second version of OPS this mechanism is being expanded to allow adding new match functions as well. Unfortunately, no solution to the problems with conflict resolution strategies has been found. A number of sets of conflict resolution rules have been tried, but no universally appropriate set has emerged. Simply allowing the user to modify the conflict resolution rules at will is not a solution; it now appears that the problems with conflict resolution have resulted, at least in part, from the decision not to allow productions to read the information in state memory.

ACKNOWLEDGEMENTS

The development of many of the ideas discussed above owes much to the members of the instructable production system project at Carnegie-Mellon University. The members of this project, in addition to the authors, are J. Laird, P. Langley, A. Newell, K. Ramakrishna, P. Rosenbloom, and M. Rychener.

REFERENCES

Anderson, J. Language, Memory, and Thought. Lawrence Erlbaum Associates, 1976.

Anderson, R. H. and Gillogly, J. J. Rand intelligent terminal agent (RITA): design philosophy. Report R-1809-ARPA. The Rand Corporation, Santa Monica, CA, 1976.

Barstow, D. Automatic construction of algorithms and data structures using a knowledge base of programming rules. Artificial Intelligence Laboratory, Stanford University, 1977.

Buchanan, B, and Lederberg, J. The heuristic DENDRAL program for explaining empirical data. IF IP, 1971, pp. 179-188.

Davis, R. Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases. SAIL AIM-271. Artificial Intelligence Laboratory, Stanford University, 1976.

Davis, R., Buchanan, B., and Shortliffe, E. Production rules as a representation for a knowledge-based consultation program. Report STAN-CS-75-519, Memo AIM-266. Computer Science Department, Stanford University, 1975.

Davis, R. and King, J. An overview of production systems. Report STAN-CS-75-524, Memo A1M-271. Department of Computer Science, Stanford University, 1975.

Erman, L. and Lesser, V. A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge. IJCAI 4, 1975, pp 483-490.

Forgy, C. A production system monitor for parallel . computers. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1977.

Forgy, C. and McDermott, J. The OPS reference manual. Department of Computer Science, Carnegie-Mellon University, 1976.

Lenat, D. AM: an artificial intelligence approach to discovery in mathematics as heuristic search. Report STAN-CS-76-570, Memo AIM-286. Computer Science Department, Stanford University, 1976.

Lenat, D. and Harris G. Designing a rule system that searches for scientific discoveries. In D. A. Waterman and F. Hayes-Roth (eds), Pattern-Directed Inference Systems. Academic Press, 1977 (forthcoming).

McCracken, D. A parallel production system architecture for speech understanding. Technical Report. Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, 1977.

McDermott, J. and Forgy, C. Production system conflict resolution strategies. In D. A. Waterman and F. Hayes-Roth (eds), Pattern-Directed Inference Systems. Academic Press, 1977 (forthcoming).

McDermott, J., Newell, A., and Moore, J. The efficiency of certain production system implementations. In D. A. Waterman and F. Hayes-Roth (eds), Pattern-Directed Inference Systems. Academic Press, 1977 (forthcoming).

Newell, A. Production systems: models of control structures. In Chase, W. (ed.), Visual Information Processing. Academic. Press, 1973, pp. 463-526.

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

Rychener, M. D. Production systems as a programming language for artificial intelligence applications. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1976.

Rychener, M. and Newell, A. An instructable production system: initial design issues. In D. A. Waterman and F. Hayes-Roth (eds), Pattern-Directed Inference Systems. Academic Press, 1977 (forthcoming).