AD-A019 702

AN OVERVIEW OF PRODUCTION SYSTEMS

Randall Davis, et al

Stanford University

Prepared for:

Advanced Research Projects Agency

October 1975

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

028140

Stanford Artificial Intelligence Laboratory
Memo AIM-271

October 1975

Computer Science Department
Report No. STAN-CS-75-524

An Overview of Production Systems

by

Randall Davis and Jonathan King

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494
and
National Institutes of Health

COMPUTER SCIENCE DEPARTMENT
Stanford University

AD A019702

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-75-524, AIM271	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* An Overview of Production Systems		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R. Davis and J. King		8. CONTRACT OR GRANT NUMBER(s) DAHC15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory Stanford University Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 2494
11. CONTROLLING OFFICE NAME AND ADDRESS Col.Dave Russell, Dep. Dir., ARPA, IPT, ARPA Headquarters, 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE October 1975 13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Philip Surra, ONR Representative Durand Aeronautics Building Room 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. *(of this report)* 15 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Releasable without limitations on dissemination

DISTRIBUTION STATEMENT A
Approved for public release
Distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT (

Since production systems were first proposed in 1943 as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. This paper is an attempt to provide an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other, and to other methodologies.

Accordingly, we use the term "production system" in a broad sense, and attempt to show how most systems which have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally-based techniqu but with reference also to some of the current developments in programming and t organization of data and knowledge bases.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Stanford Artificial Intelligence Laboratory
Memo AIM-271

October 1975

Computer Science Department
Report No. STAN-CS-75-524

An Overview of Production Systems

by

Randall Davis and Jonathan King

ABSTRACT

Since production systems were first proposed in 1943 as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. This paper is an attempt to provide an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other, and to other methodologies.

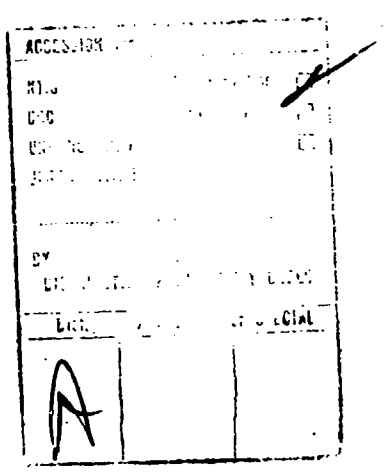
Accordingly, we use the term 'production system' in a broad sense, and attempt to show how most systems which have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally-based techniques, but with reference also to some of the current developments in programming and the organization of data and knowledge bases. This is a slightly revised version of a paper to appear in *Machine Representations of Knowledge*, Dordrecht, D. Reidel Publishing Company (1976).

The work reported here was funded in part by grants from the National Institutes of Health, NIH Grant HS01544, and the Advanced Research Projects Agency ARPA Contract DAHC15-73-C-0435

The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, NIH, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22151.

Contents



{1} INTRODUCTION AND BACKGROUND

Since production systems (PS) were first proposed by Post [1943] as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. This paper is an attempt to provide an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other, and to other methodologies [1].

Accordingly, we use the term 'production system' in a broad sense, and attempt to show how most systems which have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally-based techniques, but with reference also to some of the current developments in programming and the organization of data and knowledge bases.

We begin by offering a review of the essential structure and function of a PS, presenting a picture of a "pure" PS to provide a basis for subsequent elaborations. We then suggest that current views of PSs fall into two distinct classes, and demonstrate that this dichotomy may explain much of the existing variation in goals and methods. This is followed by some speculations on the nature of appropriate and inappropriate problem domains for PSs – i.e. what is it about a problem that makes the PS methodology appropriate, and how do these factors arise out of the system's basic structure and function? Next we review a dozen different characteristics which we found common to all systems, explaining how they contribute to the basic character, and noting their interrelationships. Finally, we present a taxonomy for PSs, selecting four dimensions of characterization, and indicating the range of possibilities as suggested by recent efforts.

Two points of methodology should be noted – first, we make frequent reference to what is "typically" found, and what is "in the spirit of things." Since there is really no one formal design for current PSs, and recent implementations have explored variations on virtually every aspect, their use becomes more an issue of a programming *style* than anything else. It is difficult, then, to exclude designs or methods on formal grounds, and we refer instead to an informal, but, we feel, well established style of approach.

A second, related point will be important to keep in mind as we compare the capabilities of PSs with those of other approaches. Since it is possible to imagine coding any given Turing machine in either procedural or PS terms (see [Anderson1976] for a formal proof of the latter), in the formal sense their computational power is equivalent. This suggests that, given sufficient effort, they are ultimately capable of solving the same problems. The issues we wish to examine are not, however, questions of absolute computational power, but the impact of a particular methodology on program structure, as well as the relative ease or difficulty with which certain capabilities can be achieved, and the extent to which they can be achieved "in the spirit of things."

{2} "PURE" PRODUCTION SYSTEMS

A production system may be viewed as consisting of three basic components: a set of rules, a data base, and an interpreter for the rules. In the simplest design, a rule is an ordered pair of symbol strings, with a left and right hand side (LHS and RHS); the rule set has a predetermined, total ordering; and the data base is simply a collection of symbols. The interpreter in this simple design operates by scanning the LHS of each rule until one is found which can be successfully matched against the data base. At that point the symbols matched in the data base are replaced with those found in the RHS of the rule, and scanning either continues with the next rule or begins again with the first. A rule can also be viewed as simple conditional statement, and the invocation of rules as a chained sequence of *modus ponens* actions.

{2.1} RULES

More generally, one side of a rule is *evaluated* with reference to the data base, and if this succeeds (i.e. evaluates to TRUE in some sense), the action specified by the other side is performed. Note that *evaluate* is typically taken to mean a passive operation of "perception", or, "an operation involving only matching and detection" [2], while the action is generally one or more conceptually primitive operations (although more complex constructs are also being examined; see {5.9}). As noted, the simplest evaluation is a matching of literals, and the simplest action a replacement.

Note that we do not specify which side is to be matched, since either is possible. For example, given a grammar written in production rule form [3],

$$S \rightarrow A\ B\ A$$

$$B \rightarrow B\ 0 \qquad A \rightarrow A\ 1$$

$$B \rightarrow 0 \qquad A \rightarrow 1$$

matching the LHS on a data base which consists of the start symbol S, gives a generator for strings in the language. Matching on the RHS of the same set of rules gives a recognizer for the language. We can also vary the methodology slightly to obtain a top down recognizer, by interpreting elements of the LHS as goals to be obtained by the successful matching of elements from the RHS. In this case the rules "unwind." Thus we can use the same set of rules in several ways. Note, however, that in doing so we obtain quite different systems, with characteristically different control structures and behavior.

The organization and accessing of the rule set is also an important issue. The simplest scheme is the fixed, total ordering mentioned, but elaborations quickly grow more complex. The term *conflict resolution* has been used to describe the process of selecting a rule. These issues of rule evaluation and organization are explored in more detail below.

{2.2} DATA BASE

In the simplest production system, the data base is simply a collection of symbols intended to reflect the state of the world, but the interpretation of those symbols depends in large part on the nature of the application. For those systems intended to explore symbol processing aspects of human cognition, the data base is interpreted as modelling the contents of some memory mechanism (typically short term memory, STM), with each symbol representing some "chunk" of knowledge, and hence its total length (typically around seven elements), and organization (linear, hierarchical, etc.), are important theoretical issues. Typical contents of STM for psychological models are those of PSG [Newell1973], where STM might contain purely content-free symbols like:

```
QQ
(EE FF)
TT
```

or VIS [Moran1973a], where the STM contains symbols representing directions on a visualized map:

```
(NEW C-1 CORNER WEST L-1 NORTH L-2)
(L-2 LINE EAST P-2 P-1)
(HEAR NORTH EAST % END)
```

For systems intended to be knowledge-based experts, the data base contains facts and assertions about the world, is typically of arbitrary size, and has no *a priori* constraints on the complexity of organization. For example, the MYCIN system [Davis1975] uses a collection of 4-tuples, consisting of an associative triple and a "certainty factor" (CF, see [Shortliffe1975b]), which indicates (on a scale from -1 to 1) how strongly the fact has been confirmed (CF > 0) or disconfirmed (CF < 0):

```
(IDENTITY ORGANISM-1 E.COLI .8)
(SITE CULTURE-2 BLOOD 1.0)
(SENSITIVE ORGANISM-1 PENICILLIN -1.0)
```

As another example, in the DENDRAL system [Feigenbaum1971, Smith1972] the data base contains complex graph structures which represent molecules and molecular fragments. The structures are built by assigning unique numbers to each atom of a molecule and by describing chemical bonds by a pair of numbers indicating the atoms they join.

A third style of organization for the data base is the "token stream" approach used, for example, in LISP70 [Tesler1973]. Here the data base is a linear stream of tokens, accessible only in sequence. Each production in turn is matched against the beginning of the stream (i.e. if the first character of a production and the first character of the stream differ, the whole match fails), and if the rule is invoked, it may act to add, delete, of modify characters in the matched segment. The anchoring of the match at the first token offers the possibility of great efficiency in rule selection, since the productions can be "compiled" into a decision tree which keys off sequential tokens from

the stream (a very simple example is shown below).

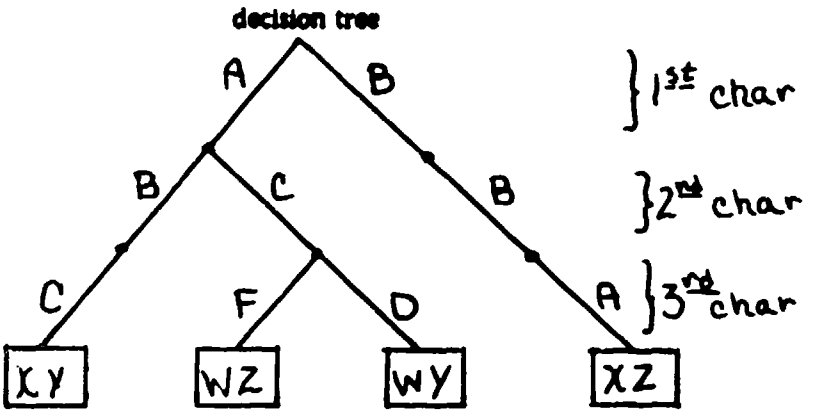
production set

A B C → X Y

A C F → W Z

B B A → X Z

A C D → W Y



Whatever the organization of the data base, one important characteristic should be noted: it is the sole storage medium for all state variables of the system. In particular, unlike procedurally-oriented languages there is no provision for separate storage of control state information – no separate program counter, pushdown stack, etc. There is nothing but the single data base, and all information to be recorded must go there. We refer to this as the unity of data and control store, and examine some of its implications below. This store is, moreover, universally accessible to every rule in the system, so that anything put there is potentially detectable by any rule. We will see that both of these have significant consequences for the use of the data base as a communication channel.

{2.3} INTERPRETER

The interpreter is the source of much of the variation found among different systems, but it may be seen in the simplest terms as a *select-execute* loop, in which one rule applicable to the current state of the data base is chosen, and then executed. Its action results in a modified data base, and the select phase begins again. Given that the selection is often a process of choosing the first rule which matches the current data base, it is clear why this cycle is often referred to as a 'recognize-act', or 'situation-action' loop. The range of variations on this theme is explored in the section on control cycle architecture.

This alternation of selection and execution is an essential element of PS architecture that is responsible for one of their most fundamental characteristics. By choosing each new rule for execution on the basis of the total contents of the data base, we are effectively performing a complete reevaluation of the control state of the system at every cycle (recall the unity of data and control store). This is distinctly different from procedurally-oriented approaches, in which control flow is typically the decision of the process currently executing, and is commonly dependent on only a small fraction of the total number of state variables. PSs are thus sensitive to any change in the entire environment, and potentially reactive to such changes within the scope of a single execution cycle. The price of such reactivity is, of course, the computation time required for the reevaluation.

An example of one execution of the recognize-act loop for a much simplified version of Newell's PSG system will illustrate some of the foregoing notions. The production system, called PS.ONE, is assumed for this example to contain two productions, PD1 and PD2. We indicate this as follows:

```
PS.ONE:(PD1 PD2)

PD1: (DD AND (EE) --> BB)
PD2: (XX --> CC DD)
```

PD1 says that if the symbol DD and some expression beginning with EE (that is, (EE ...)) is found in STM, then insert the symbol BB at the front of STM. PD2 says that if the symbol XX is found in STM, then first insert the symbol CC, then the symbol DD, at the front of the STM. The initial contents of STM are:

```
STM: (QQ (EE FF) RR XX SS)
```

This STM is assumed to have a fixed maximum capacity of five elements. As new elements are inserted at the beginning (left) of the STM, therefore, other elements will be lost (forgotten) off the right end. Second, as illustrated below, for this system elements accessed in matching the condition of a rule are 'refreshed' (pulled to the front of STM), rather than replaced.

The production system then scans the productions in order: PD1, then PD2. Only PD2 matches, so it is evoked. The contents of STM after this step are:

STM: (DD CC XX QQ (EE FF))

PD1 will match during the upcoming cycle to yield,

STM: (BB DD (EE FF) CC XX)

completing two cycles of the system.

{3} TWO VIEWS OF PRODUCTION SYSTEMS

Throughout much of the work reported, there appear to be two major views of PSs, as characterized on one hand by the psychological modelling efforts (PSG, PAS II, VIS, etc.), and on the other by the performance oriented, knowledge-based expert systems (e.g. MYCIN, DENDRAL). These appear to be two distinct efforts which have arrived at similar methodologies while pursuing differing goals.

The efforts to simulate human performance on simple tasks are aimed at creation of a program which embodies a theory of that behavior. From the performance record of experimental subjects the modeller attempts to formulate the minimally competent set of production rules, the smallest set still able to reproduce the behavior. Note that 'behavior' here is meant to include *all* aspects of human performance (including mistakes, the effects of forgetting, etc.), all shortcomings or successes which may arise out of (and hence may be clues to) the 'architecture' of human cognitive systems [4].

An example of this approach is the PSG system, from which we constructed the example above. This system has been used to test a number of theories to explain the results of the Sternberg memory scanning tasks [Newell1973], with each set of productions representing a different theory of how the human subject retains and recalls the information given to him during the psychological task. Here the subject first memorizes a small subset of a class of familiar symbols (e.g. digits), and then attempts to respond to a symbol flashed on a screen by indicating whether or not it was in the initial set. His response times are noted.

The task was first simulated with a simple production system that performed correctly, but did not account for timing variations (which were due to list length and other factors). Refinements were then developed to incorporate new hypotheses about how the symbols were brought into memory, and eventually a good simulation was built around a small number of productions.

Newell has reported [1973] that use of a production system methodology led in this case to a novel hypothesis that certain timing effects are caused by a decoding process rather than a search process, and that it also clearly illustrated the possible tradeoffs in speed and accuracy between differing processing strategies. As such it was an effective vehicle for the expression and evaluation of theories of behavior.

The performance-oriented expert systems, on the other hand, start with productions as a representation of knowledge about a task or domain, and attempt to build a program which displays competent behavior in that domain. These efforts are not concerned with similarities between the resulting systems and human performance (except insofar as the latter may provide a possible hint about ways to structure the domain or to approach the problem, or as a yardstick for success, since few AI programs approach human levels of competence). They are intended simply to perform the task without errors of any sort, human-like or otherwise.

The approach is characterized by the DENDRAL system, in which much of the development has involved embedding a chemist's knowledge about mass spectrometry into rules usable by the

program, without attempting to model the chemist's thinking. The program's knowledge is extended by adding rules that apply to new classes of chemical compounds. Similarly, much of the work on the MYCIN system has involved crystallizing informal knowledge of clinical medicine in a set of production rules.

Despite the difference in emphasis, both approaches have been drawn to PSs as a methodology. For the psychological modellers, production rules offer a clear, formal, and powerful way of expressing basic symbol processing acts, which form the primitives of information processing psychology [5]. For the designer of knowledge-based systems, production rules offer a representation of knowledge that is relatively easily accessed and modified, making it quite useful for systems designed for incremental approaches to competence. For example, much of the MYCIN system's capability for explaining its actions is based on the representation of knowledge as individual production rules [Shortliffe1975a]. This makes the knowledge far more accessable to the program itself than it might otherwise be if it were embodied in the form of ALGOL-like procedures. As in DENDRAL, the modification and upgrading of the system are by incremental modification of, or addition to the rule set.

Note that we are suggesting here, and through much of the rest of the paper, that it is possible to view a great deal of the work on PSs in terms of a unifying formalism. The intent is to offer a conceptual structure which can help organize what may appear to be a disparate collection of efforts. The presence of such a formalism should not, however, obscure the significant differences which arise out of the various perspectives. As one example, the decision to use RHS-driven rules in a goal-directed fashion implies a control structure which is simple and direct, but relatively inflexible. This offers a very different programming tool than the LHS-driven systems. The latter are capable of much more complex control structures, giving them capabilities much closer to those of a complete programming language. Recent efforts, especially, have begun to explore the issues of more complex, higher level control within the PS methodology (see {5.9}).

It should also be noted that production systems are seen by some [Newell1972] not as simply a convenient paradigm for approaching psychological modelling, but rather as a methodology whose power arises out of its close similarity to fundamental mechanisms of human cognition. In this view, human problem solving behavior can be modelled easily and successfully by a production system because it in fact is being generated by one:

> We confess to a strong premonition that the actual organization of human programs closely resembles the production system organization.... We cannot yet prove the correctness of this judgment, and we suspect that the ultimate verification may depend on this organization's proving relatively satisfactory in many different small ways, no one of them decisive.
>
> In summary, we do not think a conclusive case can be made yet for production systems as *the* appropriate form of [human] program organization. Many of the arguments ... raise difficulties. Nevertheless, our judgment stands that we should choose production systems as the preferred language for expressing programs and program organization.
>
> [Newell1972, p 809-4, 806]

This has led to speculation [6] that the interest in production systems on the part of those building high performance knowledge-based systems is more than a coincidence. It is suggested that this is a result of current research (re)discovering what has been learned by naturally intelligent systems through evolution – that structuring knowledge in a production system format is an effective approach to the organization, retrieval, and use of very large amounts of knowledge.

The success of some production rule based AI systems does give weight to this argument, and the PS methodology is clearly powerful. But whether this is a result of its equivalence to human cognitive processes, and whether this implies artificially intelligent systems ought to be similarly structured, are, we feel, still open questions.

{4} APPROPRIATE AND INAPPROPRIATE DOMAINS

Program designers have found that PSs easily model problems in some domains, but are awkward for others. Let us briefly investigate why this may be so, and relate it to the basic structure and function of a PS.

We can imagine two very different domains -- the first is best viewed and understood as consisting of many independent states, while the second seems best understood via a concise, unified theory, perhaps embodied in a single law. Examples of the former include some views of perceptual psychology or clinical medicine, in which there are a large number of states relative to the number of actions (this may be due either to our lack of a cohesive theory, or to the basic complexity of the system being modelled). Examples of the latter include well-established areas of physics and mathematics, in which a few basic tenets serve to embody much of the required knowledge, and in which the discovery of unifying principles has emphasized the similarlities in seemingly different states. This first distinction appears to be one important factor in distinguishing appropriate from inappropriate domains.

A second distinction concerns the complexity of control flow. As two extremes, we can imagine two processes, one of which is a set of independent actions, and the other a complex collection of multiple, parallel process involving several dependent sub-processes.

A third distinction concerns the extent to which the knowledge to be embedded in a system can be separated from the manner in which it is to be used (also known as the controversy between declarative and procedural representations; see [Winograd1975] for a extensive discussion). As one example, we can imagine simply stating facts, perhaps in a language like predicate calculus. This makes no assumptions about the way those facts will be employed. Alternatively, we could write procedural descriptions of how to accomplish a stated goal. Here the use of the knowledge is for the most part predetermined during the process of embodying it in this representation.

In all three of these distinctions, a PS is well suited to the first description and ill suited to the latter. The existence of multiple, non-trivially different, independent states is an indication of the feasibility of writing multiple, nontrivial, modular rules. A process composed of a set of independent actions requires only limited communication between the actions, and, as we shall see, this is an important characteristic of PSs. The ability to state what knowledge ought to be in the system without also describing its use makes an important difference in the ease with which a PS can be written (see {5.9}).

For the latter description (unified theory, complex control flow, predetermined use for the knowledge), the economy of the basic theory makes for either trivial rules, or multiple, almost redundant rules. In addition, a complex looping and branching process requires explicit communication between actions, in which one explicitly invokes the next, while interacting subgoals require a similary advanced communication process to avoid conflict. Such communication is not easily supplied in a PS-based system. The same difficulty also makes it hard to specify in advance exactly how a given fact should be used.

It seems also to be the nature of production systems to focus upon the variations within a domain rather than upon the common threads that link different facts or operations. Thus for example, (as we shall describe in more detail below) the process of addition is naturally expressed via productions as n^2 rewrite operations involving two symbols (the digits being added). The fact that addition is commutative, or rather, that there is a property of "commutativity" shared by all operations that we consider to be addition, is a rather awkward one to express in production system terms.

This same characteristic may, conversely, be viewed as a capability for focussing on and handling significant amounts of detail. Thus, where the emphasis of a task is on recognition of large numbers of distinct states, PSs provide a significant advantage. In a procedurally-oriented approach, it is both difficult to organize and troublesome to update the repeated checking of large numbers of state variables and the corresponding transfers of control. The task is far easier in PS terms, where each rule can be viewed as a demon awaiting the occurrance of a specific state. [In the case of one current PS (DENDRAL), the initial, procedural approach proved sufficiently inflexible that the entire system was rewritten in production rule terms.]

In addition, we noted above the potential sensitivity and reactivity of PSs which arises from their continual reevaluation of the control state. This has also been referred to as the 'openness' of production rule based systems. It is characterized by the principle that 'any rule can fire at any time', which emphasizes the fact that at any point in the computation, any rule could possibly be the next to be selected, depending only on the state of the data base at the end of the current cycle. Compare this to the normal situation in a procedurally-oriented language, where such a principle is manifestly untrue: it is simply not typically the case that, depending on the contents of that data base, any procedure in the entire program could potentially be the next one invoked.

PSs therefore appear to be useful where it is important to detect and deal with a large number of independent states, in a system which requires a broad scope of attention, and the capability of reacting quickly to small changes. In addition, where knowledge of the problem domain falls naturally into a sequence of independent 'recognize-act' pairs, PSs offer a convenient formalism for structuring and expressing that knowledge.

Finally, note that the implication is not that both approaches *couldn't* perform in both domains, but that there are tasks for which one of them would prove awkward, and the resulting system unenlightening. Such tasks are far more elegantly accomplished in only one of the two methodologies. The main point is that we can, to some extent, formalize our intuitive notion of which approach seems more appropriate by considering two essential characteristics of any PS – its set of multiple, independent rules, and its limited, indirect channel of interaction via the data base.

{5} PRODUCTION SYSTEM CHARACTERISTICS

Despite the range of variations in methodologies, there appear to be many characteristics common to almost all PSs. It is the presence of these, and their interactions, that contribute to the 'nature' of a PS, its capabilities, deficiencies, and characteristic behavior. This section is an attempt to isolate and analyze each of these factors, discovering the primitive elements which compose them, and providing an overview of their interactions.

The network of Figure 1 is a summary of features and relationships. Each box represents some feature, capability, or parameter of interest, with arrows labelled with +'s and -'s suggesting the interactions between them. This rough scale of facilitation and inhibition is naturally very crude, but does indicate the interactions as we saw them.

Figure 1 contains at least three conceptually different sorts of factors: (a) those fundamental characteristics of the basic PS scheme (e.g. indirect/limited channel, constrained format); (b) secondary effects of these (e.g. automated modification of behavior); and (c) performance parameters of implementation which are helpful in characterising PS strengths and weaknesses (e.g. visibility of behavior flow, extensibility). This division of factors is suggested by the three levels illustrated in the figure.

Below we briefly describe each feature and suggest the nature of its interaction with the others.

{5.1} INDIRECT/LIMITED CHANNEL OF INTERACTION

Perhaps the most fundamental and significant characteristic of PSs is their restriction on the interactions between rules. In the simplest model, a pure PS, we have a completely ordered set of rules, with no interaction channel other than the data base. The total effect of any rule is determined by its modifications to the data base, and hence subsequent rules must 'read' there any traces it may leave behind. Winograd [1975] characterizes this in discussing global modularity in programming:

> We can view production systems as a programming language in which all interaction is forced through a very narrow channel....The temporal interactions [of individual productions] is completely determined by the data in this STM, and a uniform ordering regime for deciding which productions will be activated in cases where more than one might apply.... Of course it is possible to use the STM to pass arbitrarily complex messages which embody any degree of interaction we want. But the spirit of the venture is very much opposed to this, and the formalism is interesting to the degree that complex processes can be described without resort to such kludgery, maintaining the clear modularity between the pieces of knowledge and the global process which uses them.

While this characterization is clearly true for a pure PS, with its limitations on the size of STM, we can generalize on it slightly to deal with a broader class of systems. First, in the more

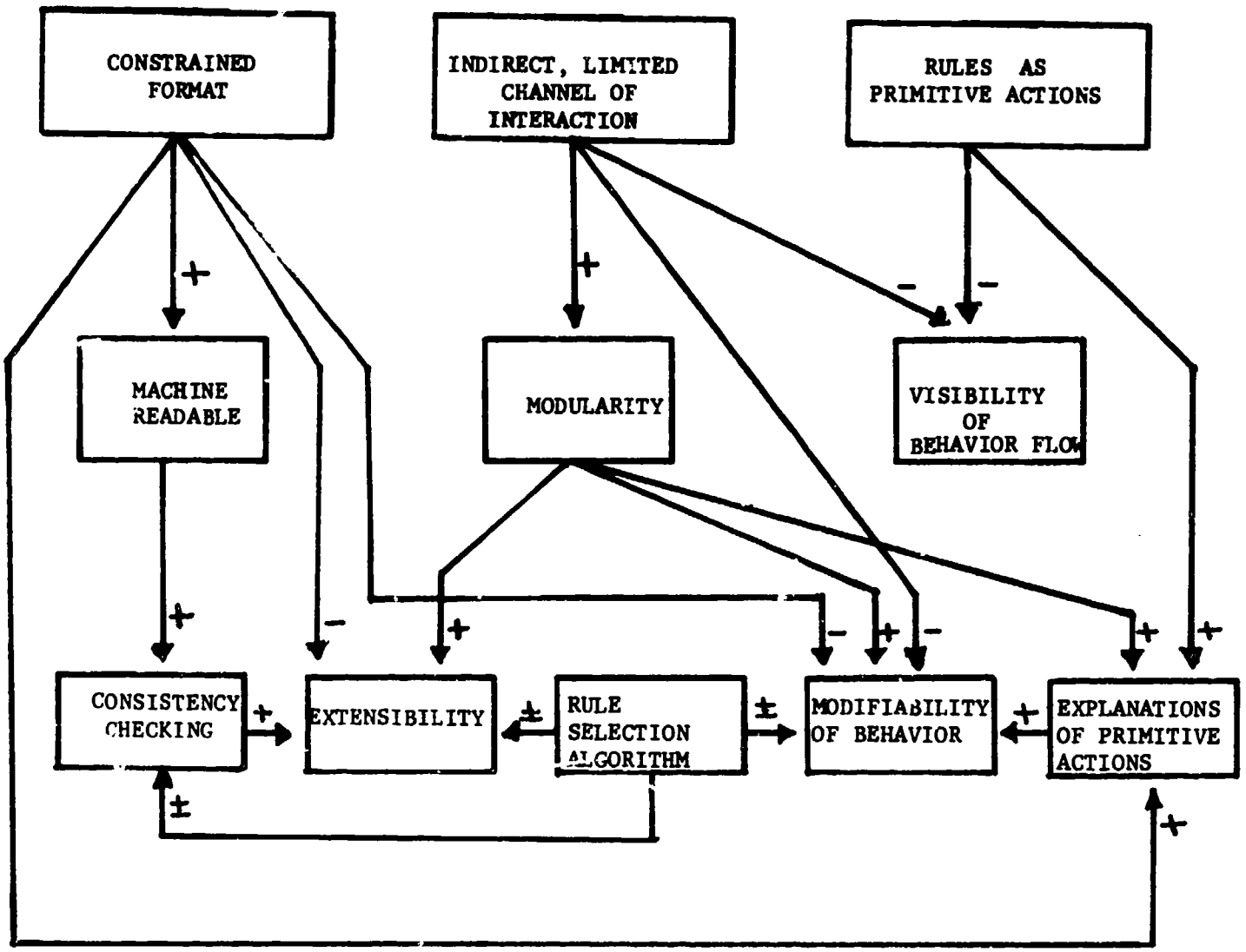

CONSTRAINED FORMAT
INDIRECT, LIMITED CHANNEL OF INTERACTION
RULES AS PRIMITIVE ACTIONS
MACHINE READABLE
MODULARITY
VISIBILITY OF BEHAVIOR FLOW
CONSISTENCY CHECKING
EXTENSIBILITY
RULE SELECTION ALGORITHM
MODIFIABILITY OF BEHAVIOR
EXPLANATIONS OF PRIMITIVE ACTIONS

FIGURE 1

general case, the channel is not so much narrow as *indirect* and *unique*. Second, the kludgery arises not from arbitrarily complex messages, but from *specially crafted* messages which force highly specific, carefully chosen interactions.

With reference to the first point, one of the most fundamental characteristics of the pure PS organization is that rules must interact indirectly through a single channel. Indirection implies that all interaction must occur by the effect of modifications written in the data base; uniqueness of the channel implies that these modifications are accessible to every one of the rules. Thus, to produce a system with a specified behavior, one must think not in the usual terms of having one section of code call another explicitly, but rather use an indirect approach in which each piece of code (i.e. each rule) leaves behind the proper traces to trigger the next relevant piece. The uniform access to the channel, along with the openness of PSs, implies that those traces must be constructed in the light of a potential response from any rule in the system. It is in some sense like a difficult case of programming purely by side effects.

With reference to the second point, in many systems the action of a single rule may, quite legitimately, result in the addition of very complex structures to the data base (e.g. DENDRAL, see Taxonomy, {6.1}). Yet another rule in the same system may deposit just one carefully selected symbol, chosen solely because it will serve as an unmistakable symbol to precisely one other (carefully preselected) rule. Choosing the symbol carefully provides a way of sending what becomes a private message through a public channel; the continual reevaluation of the control state assures that the message can take immediate effect. The result is that one rule has effectively called another, procedure style, and this is the variety of kludgery which is contrary to the style of knowledge organization typically associated with a PS. We argue below (see {5.5}) that it is, in particuar, the premeditated nature of such message passing (typically in an attempt to 'produce a system with specified behavior') which is the primary violation of the spirit of PS methodology.

The primary effect of this indirect, limited interaction is to produce a system which is strongly modular, since no rule is ever called directly. It is also, however, perhaps the most significant factor in making the behavior flow of a PS more difficult to analyze. This is due to the fact that, even for very simple tasks, overall behavior of a PS may not be at all evident from a simple review of its rules.

To illustrate many of these issues, consider the algorithm for addition of positive, single digit integers used by Waterman [1974] with his PAS II production system interpreter. First, the procedural version of the algorithm, in which transfer of control is direct and simple:

```
        acid(m,n) ::=
A]          count ← 0; nn ← n;
B]      L1: if count = m then return(nn);
C]          count ← successor(count);
D]          nn ← successor(nn);
E]          go(L1);
```

Compare this with the set of productions for the same task, in Figure 2 (the notation may be somewhat unfamiliar; we have provided a brief explanation). The two are not precisely analogous, since the procedural version does simple addition, while the production set both adds and "learns" (see note [7] and the comments in Figure 2). The example is still quite illustrative, however, and Waterman points out some direct correspondences between productions and statements in the procedure. For example, productions 1 and 2 accomplish the initialization of line A, rule 3 corresponds to line B, and rule 4 to lines C and D. There is no production equivalent to the goto of line E because the production system execution cycle takes care of that implicitly. On the other hand, note that in the procedure, there is no question whatsoever that the initialization step nn ← n is the second statement of "add", and that it is to be executed just once, at the beginning of the procedure. In the productions, the same action is predicated on an unintuitive condition of the STM (essentially it says if the value of n is known, but nn has never been referenced or incremented, then initialize nn to the value that n has at that time). This degree of explicitness is necessary because the production system has no notion that the initialization step has already been performed in the given ordering of statements, so the system must check the conditions each time it goes through a new cycle.

Thus, procedural languages are oriented toward the explicit handling of control flow and stress the importance of its influence on the fundamental organization of the program (as, for example, in recent developments in structured programming). PSs, on the other hand, emphasize the statement of independent chunks of knowledge from a domain, and make control flow a secondary issue. Given, moreover, the limited form of communication available, it is more difficult to express concepts which require structures larger than a single rule. Thus, where the emphasis is on global behavior of a system rather than the expression of small chunks of knowledge, PSs are, in general, going to be less transparent than formulations of equivalent routines in procedural terms.

CONDITION		ACTION
1] (READY) (ORDER X1)	=>	(REP (READY) (COUNT X1)) (ATTEND)
2] (N X1) - (NN) - (S NN)	=>	(DEP (NN X1))
3] (COUNT X1) (M X1) (NN X2) (N X3)	=>	(SAY X2 IS THE ANSWER) (COND (M X1) (N X3)) (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER)) (PROC) (STOP)
4] (COUNT) (NN)	=>	(REP (COUNT) (S COUNT)) (REP (NN) (S NN))
5] (ORDER X1 X2)	=>	(REP (X1 X2) (X2)) (COND (S X3 X1)) (ACTION (REP (S X3 X1) (X3 X2))) (PROD)

initial STM: (READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

notation:

The Xi's in the CONDITION are variables in the pattern match, all other symbols are literals. An Xi appearing *only* in the ACTION is also taken as a literal. Thus if rule 5 is matched with X1 = 4 and X2 = 5, as its second action it would deposit (COND (S X3 4)) in STM.

All elements of the LHS must be matched for a match to succeed, except "–" indicates the ANDNOT operation.

An expression enclosed in parentheses and starting with a literal [e.g. (COUNT) in production 4] will match any expression in STM which starts with the same literal [e.g. (COUNT 2)].

Figure 2a
(after [Waterman1974], simplified slightly)

notation (continued)

REP	REPlace, so that, e.g. the RHS of production 1 will replace the expression (READY) in the data base with the expression (COUNT X1) [where the variable X1 stands for the element matched by the X1 in (ORDER X1)]
DEP	DEPosit symbols at front of STM
ATTEND	wait for input from tty. For this example, typing (M 4)(N 2) will have the system add 4 and 2
SAY	output to tty
(COND ...)	shorthand for (DEP (COND ...))
(ACTION ...)	shorthand for (DEP (ACTION ...))
PROD	gather all items in the STM of the form (COND ...) and put them together into a LHS; gather all items of the form (ACTION ...) and put them together into a RHS; removing all these expressions from the STM. Form a production from the resulting LHS and RHS, and add it to the front of the set of productions (i.e. before rule 1). [see note [11] for a comment on the importance of rule order]

comments on system behavior

The "S" (rules 2, 3 and 5) is intended to indicate the successor function.

After initialization (rules 1 and 2), the system loops around rules 4 and 5, incrementing N by 1 for M interations. In this loop, intermediate calculations (the results of successor function computations) are saved via the (PROD) in rule 5, and the final answer is saved by the (PROD) in rule 3. Thus, after computing 4 + 2, the rule set will contain the additional rules:

```
(S X3 0)      =>   (REP (S X3 0) (X3 1))
(S X3 1)      =>   (REP (S X3 1) (X3 2))
(M 4) (N 2)   =>   (SAY 6 IS THE ANSWER) (STOP)
```

The system is thus recording its intermediate and final results by writing new productions, and in the future will have these answers available a single step. Note that the set of productions therefore *is* memory (and in fact long term memory, LTM, since productions are never lost from the set).

Figure 2b

{5.2} CONSTRAINED FORMAT

While there are wide variations in the format permitted by various PSs, in any given system the syntax is traditionally quite restrictive, and generally follows the conventions accepted for PSs [8]. Most commonly this means, first, that the side of the rule to be matched should be a simple predicate built out of a Boolean combination of computationally primitive operations, which involve (as noted above) only matching and detection. Second, it means the side of the rule to be executed should perform conceptually simple operations on the data base. In many of the systems oriented toward psychological modelling, the side to be matched consists of a set of literals o. simple patterns, with the understanding that the set is to be taken as a conjunction, so that the predicate is an implicit one regarding the success or failure of matching all of the elements. Similiarly, the side to be executed performs a simple symbol replacement or rearrangement.

Whatever the format, though, the conventions noted lead to clear restrictions for a pure production system: First, as a predicate, the matching side of a rule should return only some indication of the success or failure of the match (while binding individual variables or segments in the process of pattern matching is quite often used, it would be considered inappropriate to have the matching process produce a complex data structure intended for processing by another part of the system). Second, as a simple expression, the matching operation is precluded from using more complex control structures like iteration or recursion within the expression itself (such operations can be constructed from multiple rules, however). Finally, as a matching and detection operation, it must only 'observe' the state of the data base, and not change it in the operation of testing it.

We can characterize a continuum of possibilities for the side of the rule to be executed. There might be a single primitive action, a simple collection of independent actions, a carefully ordered sequence of actions, or yet more complex control structures. We suggest that there are two related forms of simplicity which are important here. First, each action to be performed should be one which is a conceptual primtive for the domain. In the DENDRAL system, for example, it is appropriate to use chemical bond breaking as the primitive, rather than describing the process at some lower level. Second, the complexity of control flow for the execution of these primitives should be limited – in a 'pure' production system, for example, we might be wary of a complex set of actions that is, in effect, a small program of its own. Again, it should be noted that the system designer may of course follow or disregard these restrictions. The result, however, will conform to the traditional PS architecture to the extent that they are met.

These constraints on form make the dissection and 'understanding' of productions by other parts of the program a more straightforward task, strongly enhancing the possibility of having the program itself read, and/or modify (rewrite) its own productions [9]. Expressability suffers, however, since the limited syntax may not be sufficiently powerful to make expressing each piece of knowledge an easy task. This in turn both restricts extensibility (adding something is difficult if it's hard to express it), and makes modification of the system's behavior more difficult (e.g. it might not be particularly attractive to implement a desired iteration if it requires several rules rather than a line or two of code).

{5.3} RULES AS PRIMITIVE ACTIONS

In a 'pure' PS, the smallest unit of behavior is a rule invocation, which, at its simplest, involves the matching of literals on the LHS, followed by replacement of those symbols in the data base with the ones found on the RHS. While the variations can be more complex, it is in some sense a violation of the spirit of things to have e.g. a sequence of actions in the RHS.

Moran [1973b], for example, acknowledges a deviation from the spirit of production systems in his VIS, when he groups rules in "procedures" within which the rules are totally ordered for the purpose of conflict resolution. He sees several advantages in this departure: it is "natural" for the user (a builder of psychological models) to write rules as a group working toward a single goal. This grouping restricts the context of the rules. It also helps minimize the problem of implicit context: when rules are ordered, a rule which occurs later in the list may really be applicable only if some of the conditions checked by earlier rules are untrue. This dependency, referred to as implicit context, is often not made explicit in the rule, but may be critical to system performance. The price paid for these advantages is twofold: first, extra rules, less directly attributable to psychological processes, are needed to switch among procedures; second, it violates the basic production system tenet that any rule should (in principle) be able to fire at any time -- here only those in the currently active procedure can fire.

To the extent that the pure production system restrictions are met, we can consider rules as the quanta of intelligent behavior in the system. Otherwise, as in the VIS, system, we must look at larger aggregations of rules to trace behavior. In doing so we lose some of the ability to quantify and measure behavior, as is done, for example, with the PSG system simulation of the Sternberg task, where response times are attributed to individual production rules, and then compared against actual psychological data.

A different sort of deviation is found in the DENDRAL system, and in a few MYCIN rules. In both, the RHS is effectively a small program, carrying out complex sequences of actions. In this case, the quanta of behavior are the individual actions of these programs, and understanding the system thus requires familiarity with them.

By embodying these bits of behavior in a stylized format, it becomes possible for the system to 'read' them to its users (as is done in MYCIN, see [Shortliffe1975a] and note [9]), and hence provide some explanation of its behavior, at least at this level. This prohibition against complex behaviors within a rule, however, may force us to implement what are (conceptually) simple control structures by using the combined effects of several rules. This of course may make overall behavior of the system much more opaque [see Visibility, {5.5} below].

{5.4} MODULARITY

We can regard the *modularity* of a program as the degree of separation of its functional units into isolatable pieces. A program is *highly modular* if any functional unit can be changed (added, [illegible] or replaced) with no unanticipated change to other functional units. Thus program modularity is inversely related to the strength of coupling between its functional units.

The modularity of programs written as pure production systems arises from the important fact that the next rule to be invoked is determined solely by the contents of the data base, and no rule is ever called directly. Thus the addition (or deletion) of a rule does not require the modification of any other rule to provide for (delete) a call to it. We might demonstrate this by repeatedly removing rules from a PS: many systems will continue to display some sort of "reasonable" behavior, up to a point [10]. By contrast, adding a procedure to an ALGOL-like program requires modification of other parts of the code to insure that it is invoked, while removing an arbitrary procedure from such a program will generally cripple it.

Note that the issue here is more than simply the 'undefined function' error message which would result from a missing procedure. The problem persists even if the compiler or interpreter were altered to treat undefined functions as no-ops. The issue is a much more fundamental one concerning organization of knowledge: programs written in procedure-oriented languages stress the kind of explicit passing of control from one section of code to another that is characterized by the calling of procedures. This is typically done at a selected time and in a particular context, both carefully chosen by the programmer. If a no-op is substituted for a missing procedure, the context upon returning will not be what the programmer expected, and subsequent procedure calls will be executed in increasingly incorrect environments. Similarly, procedures which have been added must be called from *somewhere* in the program, but the location of the call must be chosen carefully if the effect is to be meaningful.

Production systems, on the other hand, especially in their pure form, emphasize the decoupling of control flow from the writing of rules. Each rule is designed to be, ideally, an independent chunk of knowledge with its own statement of relevance (either the conditions of the LHS, as in a data-driven system, or the action of the RHS, as in a goal-directed system). Thus where the ALGOL programmer carefully chooses the order of procedure calls to create a selected sequence of environments, in a production system it is the environment which chooses the next rule for execution. And since a rule can only be chosen if its criteria of relevance have been met, the choice will continue to be a plausible one, and system behavior remain "reasonable", even as rules are successively deleted.

This inherent modularity of pure production systems eases the task of programming in them. Given some primitive action that the system fails to perform, it becomes a matter of writing a rule whose LHS matches the relevant indicators in the data base, and whose RHS performs the action. Where the task is then complete for a pure PS, systems which vary from this design have the additional task of assuring proper invocation of the rule (not unlike assuring the proper call of a new procedure). The difficulty of this varies from trivial in the case of systems with goal oriented behavior (like MYCIN), to substantial in systems that use more complex LHS scans and conflict

resolution strategies.

For systems using the goal-oriented approach, rule order is usually unimportant. Insertion of a new rule is thus simple, and can often be totally automated. This is of course a distinct advantage where the rule set is large, and the problems of system complexity are significant.

For others (like PSG and PASII) rule order can be critical to performance and hence requires careful attention. This can, however, be viewed as an advantage, and indeed, Newell [1973] tests different theories of behavior by the simple expedient of changing the order of rules. The family of Sternberg task simulators there includes a number of production systems which differ only by the interchange of two rules, yet display very different behavior. Waterman's system [1974] accomplishes 'adaptation' by the simple heuristic of placing a new rule immediately before a rule that causes an an error [11].

{5.5} VISIBILITY OF BEHAVIOR FLOW

Visibility of behavior flow is the ease with which the overall behavior of a PS can be understood, either by observing the system, or by reviewing its rule base. Even for conceptually simple tasks, the stepwise behavior of a PS is often rather opaque. The poor visibility of PS behavior compared to that of the procedural formalism is illustrated by the Waterman arithmetic example. The procedural version of the iterative loop there is reasonably clear (lines B,C and E), and an ALGOL-type FOR I:=1 UNTIL N DO... would be completely obvious. Yet the PS formalism for the same thing requires non-intuitive productions (like 1 and 2), and symbols like NN whose only purpose is to "mask" the condition portion of a rule so it will not be invoked later (such symbols are termed *control elements* [Anderson1976]).

The requirement for control elements and much of the opacity of PS behavior is a direct result of two factors noted above: the unity of control and data store, and the reevaluation of the data base at every cycle. Any attempt to 'read' a PS requires keeping in mind the entire contents of the data base, and scanning the entire rule set at every cycle. Control is much more explicit and localized in procedural languages, so that reading ALGOL code is a far easier task (Indeed, one of the motivations for the the current interest in structured programming is the attempt to emphasize still further the degree of explicitness and localization of control).

The perspective on knowledge representation suggested by PSs also contributes to this opacity. As suggested above, PSs are appropriate when it is possible to specify the content of required knowledge, without also specifying the way in which it is to be used. Thus, reading a PS does not generally make clear how it works, so much as what it may know, and the behavior is consequently obscured. The situation is often reversed in procedural languages – program behavior may be reasonably clear, but the domain knowledge used is often opaquely embedded in the procedures. The two methodologies thus emphasize different aspects of knowledge and program organization.

{5.6} MACHINE READABLE

Several interesting capabilities arise from making it possible for the system to examine its own rules. As one example, it becomes possible to implement automatic consistency checking. This can proceed at several levels: in the simplest approach we can search for straightforward 'syntactic' problems such as contradictions (e.g. 2 rules of the form $A \wedge B \rightarrow C$ and $A \wedge B \rightarrow \neg C$), or subsumption (e.g. $D \wedge E \wedge F \rightarrow G$, $D \wedge F \rightarrow G$). A more sophisticated approach, which would require entensive domain-specific knowledge, might be able to detect 'semantic' problems, as for example a rule of the form $A \wedge B \rightarrow C$, when it is known from the meaning of A and B that $A \rightarrow \neg B$. Many other (domain specific) tests may also be possible. The point is that by automating the process, extensive (perhaps exhaustive) checks of newly added productions are possible (and could perhaps be run in background mode when the system is otherwise idle).

A second sort of capability is described in detail in note [9], and deals with the MYCIN system's approach to examining its rules. This is used in several ways [Davis1976], and produces both a more efficient control structure, and precise explanations of system behavior.

{5.7} EXPLANATION OF PRIMITIVE ACTIONS

Production system rules are intended to be modular chunks of knowledge and to represent primitive actions. Thus, explaining primitive acts should be as simple as stating the corresponding rule -- all necessary contextual information should be included in the rule itself. Achieving such clear explanations, however, evidently strongly depends upon the extent to which the assumptions of modularity and explicit context are met. In the case where stating a rule does provide a clear explanation, the task of modification of program behavior becomes easier.

As an example, the MYCIN system often successfully uses rules to explain its behavior (see [Davis1975] and [Shortliffe1975a] for examples). This form of explanation fails, however, when considerations of system performance or human engineering lead to rules whose context is obscure. One rule, for example, says in effect, "If A seems to be true, and B seems to be true, then that's (more) evidence in favor of A". It is phrased this way rather than simply "if B seems true, that's evidence in favor of A", because B is a very rare condition, and it appears counterintuitive to ask about it unless you suspect A to begin with. The first clause of the rule is thus acting as a strategic filter, to insure that the rule is not even tried unless it has a reasonable chance of succeeding. System performance has been improved (especially as regards human engineering considerations), at the cost of a somewhat more opaque rule.

{5.8} MODIFIABILITY, CONSISTENCY, RULE SELECTION MECHANISM

As noted above, the tightly constrained format of rules makes it possible for the system to examine its own rule base, with the possibility of modifying it in response to requests from the user, or to insure consistency with respect to newly added rules. While all these are conceivable in a system using a standard procedural approach, it is the heavily stylized format of rules and the typically simple control structure of the interpreters that makes them all realizable prospects in a PS.

Finally, the relative complexity of the rule selection mechanism will have varying effects on

the ability to automate consistency checks, or behavior modification and extension. A RHS scan with backward chaining seems to be the easiest to follow since it mimics part of human reasoning behavior, while a LHS scan with a complex conflict resolution strategy makes the system generally more difficult to understand. As a result, predicting and controlling the effects of changes in, or additions to, the rule base are directly influenced in either direction by the choice of rule selection method.

{5.9} PROGRAMMABILITY

It is hard to imagine any factor in this section which does not interact with programmability (and it has therefore been omitted from Figure 1). In our experience, the answer to "how easy is it to program in this formalism?" is "it's reasonably difficult." This experience is apparently not unique:

> Any structure which is added to the system diminishes the explicitness of rule conditions....Thus rules acquire implicit conditions. This makes them (superficially) more concise, but at the price of clarity and precision....Another questionable device in most present production systems (including mine) is the use of tags, markers, and other cute conventions for communicating between rules. Again, this makes for conciseness, but it obscures the meaning of what is intended. The consequence of this in my program is that it is very delicate: one little slip with a tag and it goes off the track. Also, it is very difficult to alter the program; it takes a lot of time to readjust the signals.
>
> [Moran1979a]

One source of the difficulties in programming production systems is the necessity referred to above, of programming "by side effect." Another is the difficulty of using the PS methodology on a problem that cannot be broken down into the solution of independent subproblems, or into the synthesis of a behavior which is neatly decomposable.

Several techniques have been investigated to deal with this difficulty. One of them is the use of 'tags and markers' (control elements) referred to above. These can be used in various ways, and we have come to believe that the manner in which they are used, particularly in the psychological modelling systems, can be an indication of how successfully the problem has been put into PS terms.

To demonstrate this, consider two very different (and somewhat idealized) approaches to writing a PS. In the first, the programmer writes each rule independently of all the others, simply attempting to capture in each some chunk of required knowledge. The creation of each rule is thus a separate task. Only when all of them have been written are they assembled, the data base is initialized, and the behavior produced by the entire set of rules is noted.

As a second approach, the programmer starts out with a specific behavior which he wants to recreate. The entire rule set is written as a group with this in mind, and, where necessary, one rule might deposit a symbol like A00124 in STM solely to trigger a second specific rule on the next cycle.

In the first case, the control elements would correspond to recognizable states of the system. As such, they function as indicators of those states and serve to trigger what is generally a large class of potentially applicable rules [12]. In the second case there is no such correspondence, and often only a single rule recognizes a given control element. The idea here is to insure the execution of a specific sequence of rules, often because a desired effect could not be accomplished in a single rule invocation.

Such idiosyncratic use of control elements is formally equivalent to allowing one rule to call a second, specific rule, and hence is very much out of character for a PS. To the extent that it is used, it appears to us to be suggestive of a failure of the methodology – perhaps because a PS was ill-suited to the task to begin with, or because the particular decomposition used for the task was not well chosen. [The possibility remains, of course, that a 'natural' interpretation of a control element will be forthcoming as the model develops, and additional rules which refer to it will be added. In that case the ease of adding the new rules arises out of the fact that the technique of allowing one rule to call another was not used.] Since one fundamental assumption of the PS methodology as a psychological modelling tool is that states of the system correspond to what are at least plausible (if not immediately recognizable) individual "states of mind", the relative abundance of the two uses of control elements mentioned above can conceivably be taken as an indication of how successfully the methodology has been applied.

A second approach to dealing with the difficulty of programming in PSs is the use of increasingly complex forms within a single rule. Where a 'pure' PS might have a single action in its RHS, many of the current psychological modelling systems (PAS II, VIS) have explored the use of more complex sequences of actions, including the use of conditional exits from the sequence.

Finally, one recent effort [Rychener1975] has investigated the use of PSs which are unconstrained by prior restrictions on rule format, use of tags, etc. The aim here is to employ the methodology as a formalism for explicating knowledge sources, understanding control structures, and examining the effectiveness of PSs for attacking the large problems typical of artificial intelligence. The productions in this system often turn out to have a relatively simple format, but complex control structures are built via carefully orchestrated interaction of rules. This is done with several techniques, including explicit reliance on both control elements and certain characteristics of the data base architecture. For example, iterative loops are manufactured via explicit use of control elements, and data is (redundantly) re-asserted in order to make use of the 'recency' ordering on rules [the rule which mentions the most recently asserted data item is chosen first; see {6.3}]. These techniques have supported the reincarnation into PSs of a number of sizable AI programs (e.g. Bobrow's STUDENT [Bobrow1968]), but, as the author notes, "control tends to be rather inflexible, failing to take advantage of the openness that seems to be inherent in PSs."

This reflects something of a new perspective on the use of PSs. Previous efforts have used them as tools for analyzing both the core of knowledge essential to a given task, and the manner in which such knowledge is used. Such efforts relied in part on the austerity of the available

control structure to keep all the knowledge explicit (recall Moran's comment above). The expectation is that each production will embody a single chunk of knowledge. Even in the work of [Newell1973], which used PSs as a medium for expressing different theories [= different control structures] in the Sternberg task, an important emphasis is placed on productions as a model of the detailed control structure of humans. In fact, "every aspect of the system" is assumed to have a psychological correlate [Newell1973, pg 472].

The work reported in [Rychener1975], however, after explicitly detailing the chunks of knowledge required in the word problem domain of STUDENT, notes a many-to-many mapping between its knowledge chunks and productions. It also focusses on complex control regimes which can be built using PSs. While still concerned with knowledge extraction and explication, it views PSs more as an abstract programming language and uses them as a vehicle for exploring control structures. While this approach does offer an interesting perspective on such issues, it should also be noted that as productions and their interactions grow more complex, many of the advantages associated with 'traditional' PS architecture may be lost (as for example, the loss of openness noted above). The benefits to be gained are roughly analogous to those of using a higher level programming language: while the finer grain of the process being examined may become less obvious, the power of the language permits far larger scale tasks to be undertaken, and makes it easier to examine larger scale phenomena like the interaction of entire categories of knowledge.

The use of PS has thus grown to encompass many different forms, many of which are far more complex than the 'pure' PS model described initially.

{6} TAXONOMY FOR PRODUCTION SYSTEMS

In this section we suggest four dimensions along which to characterize PSs, examine related issues for each, and indicate the range of each dimension as evidenced by systems currently (or recently) in operation.

{6.1} Form - how primitive or complex should the syntax of each side be?

There is a wide variation in syntax used by various systems, and corresponding differences in both the matching & detection process, and the subsequent action caused by rule invocation. For matching, in the simplest case only literals are allowed, and it is a conceptually trivial process (although the rule and data base may be so large that efficiency becomes a consideration). Successively more complex approaches allow free variables (Waterman's poker player [Waterman1970]), syntactic classes (as in some parsing systems), and increasingly sophisticated capabilities of variable and segment binding, and pattern specification (PAS II, VIS, LISP70; for an especially thorough discussion of pattern matching methods in production systems as used in VIS, see [Moran1973a], pp. 42-5).

The content of the data base also influences the question of form. One interesting example is Anderson's ACT system [Anderson1976], whose rules have node networks in their LHS. The appearance of an additional piece of network as input results in a "spread of activation" occurring in parallel through the LHS of each production. The rule that is chosen is the one whose LHS most closely matches the input and which has the largest subpiece of network already in its working memory.

As another example, the DENDRAL system uses a literal pattern match, but its patterns are graphs representing chemical classes, and can be quite complex. Each class is defined by a basic chemical structure, referred to as a *skeleton*. As in the data base, atoms composing the skeleton are given unique numbers, and chemical bonds are described by the numbers of the atoms they join [e.g. (5 6)]. The LHS of a rule is the name of one of these skeletons, and a side effect of a successful match is the recording of the correspondence between atoms in the skeleton and those in the molecule.

The action parts of these rules describe a sequence of actions to perform: *break* one or more bonds, saving a molecular fragment, and *transfer* one or more hydrogen atoms from one fragment to another. An example of a simple rule is:

```
ESTROGEN  ==>  (BREAK (14 15) (13 17) )
               (HTRANS +1 +2)
```

The LHS here is the name of the graph structure which describes the estrogen class of molecules, while the RHS indicates the likely locations for bond breakages and hydrogen transfers when such molecules are subjected to mass spectral bombardment. Note that while both sides of the rule are

relatively complex, they are written in terms which are conceptual primitives in the domain.

A related issue is illustrated by the rules used by MYCIN, where the LHS consists of a Boolean combination of standardized predicate functions. Here the testing of a rule for relevance consists of having the standard LISP evaluator evaluate the LHS, and all matching and detection is controlled by the functions themselves. While there is power available in using functions that is missing from a simple pattern match, there is also the temptation of writing one function to do what should have been expressed by several rules.

For example, one small task in MYCIN is to deduce that certain organisms are present, even though they have not been recovered from any culture. This is a conceptually complex, multi-step operation, which is currently handled by invocation of a single function. (Work is underway in MYCIN to provide a much cleaner, rule based solution, which will allow easier access and modification of the knowledge required for the task).

If one succumbs often to the temptation to write one function rather than several rules, the result can be a system that may perform the initial task, but which loses a great many of the other advantages of the PS approach. The problem is that the knowledge embodied in these functions is unavailable to anything else in the system. Where rules can be accessed and their knowledge examined (because of their constrained format), chunks of ALGOL-like code are not nearly as informative. The availability of a standardized, well structured set of operational primitives can help to avoid the temptation to create new functions unnecessarily.

{6.2} Content - How 'far' conceptually is it from the LHS to the RHS? Which conceptual levels of knowledge belong in rules?

The question here is how large a reasoning step is to be embodied in a single rule, and there seem to be two distinct approaches. Systems designed for psychological modelling (PAS II, PSG, etc.), try to measure and compare tasks and determine required knowledge and skills. As a result, they try to dissect cognition into its most primitive terms. While there is, of course, a range of possibilities, from the simple literal replacement of PSG to the more sophisticated abilities of PAS II to construct new productions, rules in these systems tend to embody only the most basic conceptual steps

Grouped at the other end of this spectrum are the 'task oriented' systems like DENDRAL and MYCIN, which are designed to be competent at selected real world problems. Here the conceptual primitives are at a much higher level, encompassing in a single rule a piece of reasoning which may be based both on experience and a highly complex model of the domain. For example, the statement that "a gram negative rod in the blood is likely to be an E.coli" is based in part on a knowledge of physiological systems, and in part on clinical experience. Often the reasoning step is sufficiently large that the rule becomes a significant statement of a fact or principle in the domain, and, especially where reasoning is not yet highly formalized, a comprehensive collection of such rules may represent a substantial portion of the knowledge in the field.

An interesting, related point of methodology is the question of what kinds of knowledge ought to go into rules. Rules expressing knowledge about the domain are the necessary initial step, but interest has been generated lately in the question of embodying strategies in rules. One of us has been actively pursuing this in the implementation of *meta-rules* in the MYCIN system [Davis1975]. These are rules about rules, and contain strategies and heuristics. Thus while the ordinary rules contain standard object-level knowledge about the medical domain, meta-rules contain information about rules, and embody strategies for selecting potentially useful paths of reasoning. For example, a meta-rule might suggest that

> if the patient has had an ulcer, then in concluding about organism identity, rules which mention the gastro-intestinal tract are more likely to be useful

There is clearly no reason to stop at one level, however -- third order rules could be used to select from or order the meta-rules, by using information about how to select a strategy (and hence represent a search through "strategy space"); fourth order rules would suggest how to select criteria for choosing a strategy, etc.

This approach appears to be promising for several reasons. First, the expression of any new level of knowledge in the system can mean an increase in competence. This sort of strategy information, moreover, may translate rather directly into increased speed (since fewer rules need be tried), or equivalently, no degradation in speed even with large increases in the number of rules. Second, since meta-rules refer to rule content rather than rule names, they automatically take care of new object level rules that may be added to the system. Third, the possibility of expressing this information in a format that is essentially the same as the standard one means a uniform expression of many levels of knowledge. This uniformity in turn means that the advantages which arise out of

the embodiment of any knowledge in a production rule (accessibility, and the possibility of automated explanations, modifications, and acquisition of rules) should be available for the higher order rules as well.

{6.3} Control cycle architecture [13]

The basic control cycle can be broken down into two phases called *recognition* and *action*. The recognition phase involves selecting a single rule for execution, and can be further subdivided into *selection* and *conflict resolution*. In the selection process, one or more potentially applicable rules are chosen from the set, and passed to the conflict resolution algorithm, which choses one of them.

There are several approaches to *selection*, which can be categorized by their rule scan method. Most systems (e.g. PSG, PASII) use some variation of a LHS scan, in which each LHS is evaluated in turn. Many stop scanning at the first successful evaluation (e.g. PSG), and hence conflict resolution becomes a trivial step (although the question then remains of where to start the scan on the next cycle: start over at the first rule, or continue from the current rule).

Some systems, however, collect all rules whose LHS's evaluate successfully. Conflict resolution then requires some criterion for choosing a single rule from this set (called the conflict set). Several have been suggested, and include:

i) rule order – there is a complete ordering of all rules in the system, and the rule in the conflict set with the highest priority is chosen

ii) data order – elements of the data base are ordered, and that rule chosen which matches element(s) in the data base with highest priority

iii) generality order – the most specific rule is chosen

iv) rule precedence – a precedence network (perhaps containing cycles) determines the hierarchy

v) recency order – choosing either the most recently executed rule, or the rule containing the most recently updated element of the data base

For example, the LISP70 interpreter uses (iii), while DENDRAL uses (iv).

A different approach to the selection process is used in the MYCIN system. It is goal-oriented, and uses a RHS scan. The process is quite similar to the unwinding of consequent theorems in PLANNER [Hewitt1972] – given a required subgoal, the system retrieves the (unordered) set of rules whose actions conclude something about that subgoal. The evaluation of the first LHS is begun, and if any clause in it refers to a fact not yet in the data base, a generalized version of this fact becomes the new subgoal, and the process recurses. However, because MYCIN is designed to work with judgmental knowledge in a domain (clinical medicine) where collecting all relevant data and considering all possibilities are very important, it in general executes *all* rules from the conflict set, rather than stopping after the first success.

The meta-rules mentioned above may also be seen as a way of selecting a subset of the conflict set for execution. There are several interesting advantages to this. First, the conflict resolution algorithm is stated explicitly in the meta-rules (rather than implicitly in the system's interpreter), and in the same representation as the rest of the rule-based knowledge.

Second, since there can be a set of meta-rules for each subgoal type, MYCIN can specify

distinct, and hence potentially more customized conflict resolution strategies for each individual subgoal. Since the backward chaining of rules may also be viewed as a depth first search of an AND/OR goal tree, we have the appearance of doing a tree search through a tree with an interesting property – a collection of specific heuristics about which path to take are stored at every branch point in the tree.

In addition, rules in the system are inexact, judgemental rules with a model of "approximate implication" in which the user may specify a measure of how firmly he believes that a given LHS implies its RHS [Shortliffe1975b]. This admits the possibility of writing numerous, perhaps conflicting heuristics, whose *combined* judgement forms the conflict resolution algorithm.

Control cycle architecture affects the rest of the production system in various ways. Overall efficiency, for example, can be strongly influenced. The RHS scan in a goal-oriented system insures that only relevant rules are considered in the conflict set. Since this is often a small subset of the total, and one which can be computed once and stored for reference, there is no search necessary at execution time, so the approach can be quite efficient. (In addition, since this approach seems natural to humans, the system's behavior becomes easier to follow).

Among the conflict resolution algorithms mentioned, rule order and recency order require a minimal amount of checking to determine the rule with highest priority. The generality order can be efficiently implemented, and in fact the LISP70 compiler uses it quite effectively. Data order and rule precedence require a significant amount of bookkeeping and processing, and hence may be slower (PSH, a recent development along the lines of PSG, attacks precisely this problem.)

The relative difficulty of adding a new rule to the system is also determined to a significant degree by the choice of control cycle architecture. Like PLANNER with its consequent theorems, the goal oriented approach makes it possible to simply "throw the rule in the pot" and still be assured that it will be retrieved properly. The generality ordering technique also admits of a simple, automatic method for placing the new rule, as do the data ordering and recency strategy. In the latter two cases, however, the primary factor in ordering is external to the rule, and hence while they may be added to the rule set easily, it is somewhat harder to predict and control their subsequent selection. For both complete rule order and rule precedence networks, rule addition may be a substantially more difficult problem, and depends primarily on the complexity of the criteria used to determine the hierarchy.

{6.4} System augmentability, extensibility

Learning, viewed as augmentation of the system's rule base, is of concern to both the information processing psychologists, who view it as an essential aspect of human cognition, and designers of knowledge-based systems, who acknowledge that building truly expert systems requires an incremental approach to competence. As yet we have no range or even points of comparison to offer, because of the scarcity of examples. Instead we suggest some standards by which the ease of augmentation may be judged. {It should be noted, however, that this discussion is oriented primarily toward an interactive, mixed initiative view of learning, in which the human expert teaches the system, and answers questions it may generate. It has also been influenced by the experience of one of us in attacking this problem for the MYCIN system [Davis1976]. Many other models of the process (e.g. teaching by selected examples) are of course possible, and would most likely require differenting sets of criteria}.

Perhaps the most basic question is "How automatic is it?" The ability to learn is clearly an area of competence by itself, and thus we are really asking how much of that competence has been captured in the system, and how much the user has to supply. Some aspects of this competence include --

> if the current system displays evidence of a bug caused by a missing or incorrect rule, how much of the diagnosing of the bug is handled by the system, and how much tracing must be done by the user?
>
> once the bug is uncovered, who fixes it? Must the user modify the code by hand; tell the system in some command language what to do; indicate the generic type of the error; can he simply point out the offending rule, or can the system locate and fix the bug itself?
>
> can the system indicate if the new rule will in fact fix the bug, or whether it will have side effects or undesired interactions?
>
> how much must the user know about rule format conventions when expressing a new (or modified) rule? Must he know how to code it explicitly; know precisely the vocabulary to use; know generally how to phrase it; or can he indicate in some general way the desired rule and allow the system to make the transformation? Who has to know the semantics of the domain? For example, can the system detect impossible conjunctions [A $\wedge$ B, where A $\equiv$ not-B, but 'equivalent' in the semantic sense], or trivial disjunctions [A $\vee$ B, where A $\equiv$ not-B]? Who knows enough about the system's idiosyncracies to suggest optimally fast or powerful ways of expressing rules?
>
> how hard is it to enter strategies?
>
> how hard is it to enter control structure information? Where is the control structure information stored: in aggregations of rules, or in higher order rules? The former makes augmentation or modification a difficult problem, the latter makes it somewhat easier, since the information is explicit, and concentrated in one place.
>
> can you assure continued consistency of the rule base? Who has to do the

checking?

We believe these are questions that will be important and useful to confront in designing almost any system intended to do knowledge acquisition, and especially for those built around production rules as knowledge representation.

{7} CONCLUSIONS

In artificial intelligence research, production systems were first used as a means of embodying primitive chunks of information processing behavior in simulation programs. Their adaptation to other uses, and increased experience with them has focussed attention on their possible utility as a general programming mechanism. Production systems permit the representation of knowledge in a highly uniform and modular way. This may pay off handsomely in two areas of investigation – development of programs that can manipulate their own representations, and development of a theory of loosely coupled systems, both computational and psychological. Production systems are potentially useful as a flexible modeiling tool for many types of systems; current research efforts are sufficiently diverse to discover the extent to which this potential may be realized.

Information processing psychologists continue to be interested in production systems for many reasons. PSs can be used to study a wide range of tasks [2]; they constitute a general programming system with the full power of a Turing Machine, but use a homogeneous encoding of knowledge; to the extent that the methodology is that of a pure production system, the knowledge embedded is completely explicit, and thus aids experimental verification or falsifiability of theories which use PSs as a medium of expression; productions may correspond to verifiable bits of psychological behavior [Moran1973a], reflecting the role of postulated human information processing structures such as short term memory; they are flexible enough to permit a wide range of variation based on reaction times, adaptation, or other commonly tested psychological variables: the space of PS can be fit to various alternative human information processing strategies; and finally, they provide a method for studying learning and adaptive behavior [Waterman1974].

For those wishing to build knowledge-based expert systems, the homogeneous encoding of knowledge offers the possibility of automating parts of the task of dealing with the growing complexity of such systems. Knowledge in production rules is both accessible and relatively easy to modify. It can be executed by one part of the system as procedural code, and examined by another part as if it were a declarative expression. Despite the difficulties of programming PSs, and their occasionally restrictive syntax, the fundamental methodology at times suggests a convenient and appropriate framework for the task of structuring and specifiying large amounts of knowledge. It may thus prove to be of great utility in dealing with the problems of complexity encountered in the construction of large knowledge bases.

Acknowledgements

The creation and development of this paper has extended for several months, and has profited greatly from advice from many quarters. We are indebted to Bruce Buchanan for several careful readings and quite a few extended discussions. Many helpful suggestions were offered by Allen Newell, Ed Feigenbaum, Don Waterman, Tom Moran, Terry Winograd, and Stu Card, who were patient enough to help debug early drafts.

NOTES

[1] Some of the observations reported here are insights gained by listening to the speakers in a seminar held at Stanford in late 1974, and we have tried to acknowledge those sources as accurately as possible. Apologies are offered in advance for any omissions; unattributed opinions are those of the authors.

Despite out best efforts, inevitable biases appear, so we may as well be explicit: our experience with production systems as high performance application programs is more extensive than that with the psychological modelling applications, and this is no doubt apparent at times. We have done our best to minimize such occurrances.

[2] Newell A, Simon H *Human Problem Solving* p 44, Prentice-Hall, 1972.

[3] One class of production systems we will not attempt to treat at any length here is their use as grammars for formal languages. While the intellectual roots are similar [Floyd1961, Evans1964], their use has evolved a distinctly different flavor. In particular, their use of non-determinism is an important factor which provides a very different perspective on issues of control structure and effectively renders the question of rule selection a moot point.

[4] For example:

> The critical evaluation of EPAM must ultimately depend not upon the interest which it may have as a learning machine, but upon its ability to explain and predict phenomena of verbal learning.
>
> [Feigenbaum1963]

These phenomena included stimulus and response generalization, oscillation, retroactive inhibition, and forgetting, all of which are 'mistakes' for a system intended for high performance, but are important in a system meant to model human learning behavior.

[5] Newell A, Remarks on the relationship between artificial intelligence and cognitive psychology, in *Theoretical Approaches to Non-Numerical Problem Solving*, Part IV, pp 363-400, Springer-Verlag, 1970.

[6] E. Feigenbaum, private communication.

[7] As noted in [Waterman1974], the production rule version does not assume the existence of a successor function, instead rule 5 writes new productions that give the successor for specific integers. Rule 3 builds what amounts to an addition table, writing a new production for each example the system is given. Placing these new rules at the 'front' of the rule set (i.e. before rule 1) means that the addition table and successor function table will always be consulted before a computation is attempted, and the answers obtained in one step if possible. Without these extra steps, and with a successor function, the production rule set could be smaller and hence slightly less complex.

[8] Note, however, that the tradition arises out of a commonly followed convention rather than any essential characteristic of a PS.

[9] The current MYCIN system makes strong use of the concept of allowing one part to the system to 'read' the rules being executed by another part. As one example, the system does a partial evaluation of rule premises. Since a premise is a Boolean combination of predicate functions like

```
($AND(SAME CNTXT SITE BLOOD)          {the site of the culture is blood and
      (SAME CNTXT GRAM GRAMPOS)        the gramstain is grampositive and
      (DEFIS CNTXT AIR AEROBIC))       the aerobicity is definitely aerobic}
```

and since clauses which are unknown cause subproblems to be set up which may involve long computations, it makes sense to check to see if, based on what is currently known, the entire premise is sure to fail (e.g. if any clause of a conjunction is known to be false). We cannot simple EVAL each clause, since this will trigger a search if the value is still unknown. But if the clause can be 'unpacked' into its proper constituents, it is possible to determine whether or not the value is known as yet, and if so, what it is. This is done via a TEMPLATE associated with each predicate function. For example, the TEMPLATE for SAME is

```
(SAME CNTXT PARM VALU)
```

and it gives the generic type and order of arguments to the function (much like a simplified procedure declaration). By using this as a guide to unpack and extract the needed items, we can safely do a partial evaluation of the rule premise. A similar technique is used to separate the known and unknown clauses of a rule for the user's benefit when the system is explaining itself (see [Shortliffe1975a] for several examples).

Note that, first, part of the system is 'reading' the code being executed by the other part, and second, that this reading is guided by information carried in the rule components themselves. This latter characteristic assures that the capability is unaffected by the addition of new rules or predicate functions to the system.

[10] How many rules could be removed without performance degradation (short of redundancies) is an interesting characteristic, which would appear to be correlated with the issue of which of the two common approaches to PSs is taken. The psychological modelling systems would apparently degenerate fastest, since they are designed to be minimally competent sets of rules. Knowledge-based expert systems, on the other hand, tend to embody numerous independent subproblems in rules, and often contain overlapping or even purposefully redundant representations of knowledge. Hence while losing their competence on selected problems, it appears they would still function reasonably well even with several rules removed.

[11] One specific example of the importance of rule order can be seen in our earlier example of addition (Figure 2a). Here Rule 5 assumes that an ordering of the digits exists in STM in the form

```
(ORDER 0 1 2 ...)
```

and from this can be created the successor function for each digit. If Rule 5 were placed before Rule 1, the system wouldn't add at all. In addition, acquiring the notion of successor in subsequent runs depends entirely on the placement of the new succesor productions *before* Rule 3, or the effect of this new knowledge would be masked.

[12] This basic technique of 'broadcasting' of information and allowing individual segments of the system to determine their own relevance has been extended and generalized in systems like HEARSAYII [Lesser1974], and the BEINGS system of Lenat [Lenat1975].

[13] The range of conflict resolution algorithms in this section was suggested in a talk at the seminar by Don Waterman.

REFERENCES

[Anderson1976]
Anderson J, *Language, Memory and Thought*, L Erlbaum Associates, 1976 (in preparation)

[Bobrow1968]
Bobrow, D G. Natural language input for a computer problem-solving system, in *Semantic Information Processing*, Minsky [ed], pp 146-226, 1968.

[Davis1975]
Davis R, Buchanan B G, Shortliffe E H. Production rules as a representation for a knowledge-based consultation program, AI Memo 266, (October 1975), Computer Science Department, Stanford University.

[Davis1976]
Davis R, Knowledge representation and management in expert systems, Forthcoming PhD thesis, Stanford University.

[Evans1964]
Evans A, An ALGOL 60 Compiler, in *Annual Review of Automatic Programming*, Goodman (ed), 4:87-124, 1964.

[Feigenbaum1963]
Feigenbaum E A, Simulation of Verbal Learning Behavior, in *Computers and Thought*, Feigenbaum and Feldman (eds.) pp 297-309, McGraw Hill, 1963.

[Feigenbaum1971]
Feigenbaum E A, et. al., On Generality and Problem Solving – a case study involving the DENDRAL program, in *Machine Intelligence 6*, Meltzer and Michie (eds.), pp 165-190, Edinburgh University Press 1971.

[Floyd1961]
Floyd R, A Descriptive Language for Symbol Manipulation, *J ACM*, 8:579-584, 1961.

[Hewitt1972]
Hewitt C, Description and theoretical analysis of PLANNER, PhD thesis, Department of Mathematics, 1972, M I T, Cambridge, Mass.

[Lenat1975]
Lenat D L, BEINGS: Knowledge as Interacting Experts, *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, September 1975.

[Lesser1974]
Lesser R L, et. al., Organization of the HEARSAYII Speech Understanding System, *IEEE Symmposium on Speech Recognition*, p 11-21, April 1974.

[Moran1973a]
Moran T P, *The Symbolic Imagery Hypothesis: A Production System Model*, Computer Science Department, Carnegie-Mellon University, December, 1973.

[Moran1973b]
Moran T P, The Symbolic Nature of Visual Imagery, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, August 1973

[Newell1972]
Newell A, Simon H, *Human Problem Solving*, Prentice-Hall, 1972.

[Newell1973]
Newell A, Production Systems: Models of Control Structures, in *Visual Information Processing*, William G. Chase (ed.), pp 463-526, Academic Press, 1973.

[Post1943]
Post E, Formal Reductions of the General Combinatorial Problem, *Am Jnl Math*, 65:197-268
For an interesting introduction to the general principles involved, see Minksy, *Computation: Finite and Infinite Machines*, Prentice Hall, 1967, Chapter 12.

[Rychener1975]
Rychener M D, The studnt [sic] production system: a study of encoding knowledge in production systems, Technical Report (in press), Computer Science Department, Carnegie-Mellon University.

[Shortliffe1975a]
Shortliffe E, et. al., Computer-Based Consultations in Clinical Therapeutics – explanation and rule acquisition capabilities of the MYCIN system, *Computers and Biomedical Research* Aug 1975.

[Shortliffe1975b]
Shortliffe E, Buchanan B, A Model of Inexact Reasoning in Medicine, *Mathematical Biosciences*, 23, pp 351-379, 1975.

[Smith1972]
Smith D H, et.al., Applications of Artificial Intelligence for Chemical Inference VIII *Journal of the American Chemical Society*, 94, p 5962 ff, August, 1972.

[Tesler 1973]
Tesler L G, Enea H J, Smith D C, The LISP70 Pattern Matching System, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, p 671-676, Stanford, California, 1973.

[Waterman1970]
Waterman D A, Generalization Learning Techniques for Automating the Learning of Heuristics *Artificial Intelligence*,1:121-170,1970.

[Waterman1974]
Waterman D A, *Adaptive Production Systems* Complex Information Processing Working Paper No. 285, Psychology Department, Carnegie-Mellon University, December, 1974.

[Winograd1975]
Winograd T, Frame representations and the Procedural/Declarative Controversy, in *Representation and Understanding*, Bobrow and Collins (eds.), Academic Press, 1975