PARALLEL-SERIAL PRODUCTION SYSTEMS WITH MANY WORKING MEMORIES

by

Leonard Uhr

Computer Sciences Technical Report #313

January 1978

PARALLEL-SERIAL PRODUCTION SYSTEMS WITH MANY WORKING MEMORIES*

Leonard Uhr
University of Wisconsin

Abstract

This paper describes several extensions to standard Production System (PS) languages that appear to make them more conveniently usable for a wider variety of perceptual and cognitive systems. The extensions are described (to parallel productions, multiple memories, and productions that are implied by productions), and a programming language ($(PS)^2M^n$) is presented that incorporates them. Sketches are given indicating how several different kinds of cognitive and perceptual systems might be coded in this extended language, and suggestions are made as to additional extensions that might further augment the power and convenience of such a language.

*This research has been partially supported by National Science Foundation grant MCS76-07333.

Introduction

This paper describes a Parallel-Serial Multi-Memory Production System language, $(PS)^2M^n$, that handles 1) parallel sets of productions, 2) more than one working memory in which to look for the conditions of productions, and fire the consequences of successful productions, and 3) productions that are fired by productions. These extensions would appear to be convenient for a variety of systems, where parallel processes and dispersed memories are needed. The following such systems are briefly described:

A) Several "Knowledge Sources," all working in parallel with separate working memories, but communicating through a common single "Blackboard" memory.

B) Systems for parallel compatability relaxation, where a number of nodes are examined in parallel, for contextual information that might serve to disambiguate them.

C) Discrimination nets (e.g. for "concept formation" and "EPAM"), where a sorting tree is used to find the appropriate name.

D) Parallel-serial perceptual systems, where sensed information is successively transformed, by layers of parallel feature-detectors and characterizers, into successively more abstract memory representations.

E) Parallel search for solution paths in problem-solving.

The Appendix contains programs (in EASEy/Snobol*) for $(PS)^2M^n$ and for a traditional (serial, single-memory) Production System, along with a short Primer description of the language.

*See Uhr, 1974a; Griswold et al, 1968.

Parallel-Serial Production Systems With Many Working Memories

It is often necessary (as in systems for perception, memory search and heuristic problem-solving) to apply a whole set of transformations, or productions, to a Working Memory store in parallel. Systems with several knowledge sources use parallel processes and several memories. In multi-layered or hierarchical perceptual systems, the conditions of productions must be looked for in one WM_i and the consequent acts effected on another WM_j. Therefore a whole set of WM, WM_1,...WM_N are needed. Each production will look at a specified WM_i (or several WM) and, if fired (because all its conditions have been satisfied by a match with elements in that WM_i) effect its consequent acts (e.g. insertions, deletions, arithmetic operations, generations of productions, input-output) on the specified WM_j.

This paper describes extensions to current Production Systems (PS) that handle such processes, without the often cumbersome code otherwise needed, but also in a well-structured and simple way.

Traditional Production Systems

A Production System (PS) (Newell and McDermott, 1975; Rychener, 1976; Waterman, 1975) consists of an ordered set of Productions (P),P1,...PM, each with a set of Conditions (C) and a set of consequent Acts (A), plus one Working Memory (WM). E.g.:

P1: RED BLUE ⟹ GREEN DELETE (RED) DELETE (BLUE)

P2: BROWN YELLOW ⟹ DIRT

P3: GREEN-DIRT = POOR-GRASS (STOP)

WM: RED BLUE YELLOW BROWN

Each Condition specifies a set of elements. An element specifies a string (or list structure) of sub-elements to be matched with strings found in WM. The match usually demands that the first sub-elements match, and that each subsequent sub-element in C_i match a subsequent (but not necessarily the next) sub-element in WM. Variables are usually bound by whatever they find in the first match where they occur (rather than have the system try all possibilities). Acts specify insertions or deletions in Working Memory, the generation of new Productions, or any of several primitive Acts, like OUTPUT, or STOP. A Production "fires" its Acts if all its Conditions are satisfied. Usually the system moves from the top Production down, until one Production fires, when it jumps back to the top, to repeat the process. Some systems use different flows, e.g. continuing on to the next production whether the prior one fired or not (note that this does not mean that Productions are applied in parallel, for each fired Production immediately effects its Acts, and therefore modifies WM). Some systems find all Productions that succeed, and then use one of a variety of criteria for choosing and firing one Production, e.g. the one that matches the most recently added element in WM (Newell and McDermott, 1975).

Parallel-Serial Multi-Memory Production Systems

Living cognitive systems effect sets of parallel transformations, and use a number of temporary buffer Working Memories. Such parallel-serial multi-layered architectures seem similarly desirable for computer-programmed systems since (one they run on the parrallel-serial computers, see e.g. Duff, 1976, Lipovski, 1977, that current

technology is on the verge of making feasible) they offer great economy and power.

$(PS)^2M^n$ extends traditional Production Systems so that they can conveniently handle parallel processes applied to more than one Memory. The key extensions include the following:

A) Any number of Memories can be specified and used, with sets of Productions looking at and firing into each.

B) Each Production can look at several Memories, and fire into several Memories, if desired.

C) Sets of Productions can be applied in either a Parallel or a Serial mode, as specified, and the mode can be changed during a run.

D) A Production can fire another Production, or set of Productions, to be applied, as specified.

E) After a Production succeeds in firing, the system can either attempt to fire the next Production, or go back to attempt to fire the first Production, as specified.

A program (coded in EASEy/Snobol) for this extended Production System $(PS)^2M^n$ is given, and described in detail, in the Appendix, along with a brief Primer explaining how to program in it, and a program for a standard serial single-memory Production System.

These extensions allow for relatively convenient coding of a variety of perceptual and cognitive systems, some examples of which are described below.

Some of the Uses of Multiple Memories, and of Parallel and Dynamically Implied Productions

A single Memory can always be set up to do anything that multiple memories can do (since standard Production Systems are based

on Post Productions they are universal computers, and anything can be programmed in them) - but often at a heavy price. Essentially, the system can no longer use the Memories' names and attendant pointers to directly access sub-sets of information in the relevant Memories. Rather, elements in the single Memory must be given appropriate name-like tags, and the system must use the relatively slow Production-matching routines to search for the appropriately tagged elements. All the elements in a particular Memory could be grouped together, so that only one tag was needed. But that means the system must work with and try to match a large and cumbersome list of lists - and that can be a slow and costly process. Alternately, each element can be separate, but each with the same Memory tag. But that means the search for the appropriate tag must be turned into a loop, and will have to search through the entire Memory.

The original development of Production Systems was directed toward work with a very small "Working Memory" for scratch-pad like intermediate results, one that models the human "short-term memory" that is capable of handling some small number like 7 items. For such uses there is no need to add more Memories or other structures. But for a general-purpose programming language multiple Memories serve a number of purposes.

Without a parallel capability of the sort provided in $(PS)^{2}M^{n}$, a program must make several extra passes through a set of transforms to simulate parallel processes. The first pass would not actually change the Memory; rather, it would output its set of acts each specially tagged as a temporary output. Then the program would have to keep applying a set of Productions that looked for all these tags, use the output information from the tagged elements to modify Memory, and erase all these tagged temporary elements.

The dynamic implication of transforms gives the programmer powerful control capabilities for structuring the flow of processes. Without this feature a program must use a production that adds an element to Memory that only one other Production - the one to be fired next - will succeed in matching. This means the system must make an extra cycle through its productions to find that next production, in addition to adding and deleting the identifying element. Dynamic Productions make this process direct and simple.

The present system can use dynamically implied productions, but it cannot generate them. For it can insert newly generated productions only into its main list of Productions (as is done by standard Production Systems). But it also needs the ability to insert a Production's name into the list of Consequences of another Production, and to choose that Production correctly. This needs a capability of modifying a Production, and not simply creating one, and of accessing and conveniently storing histories of the names of Productions fired. Such an extension is discussed below, as part of a more general ability to create, modify, and erase both Productions and Memories - that is, to treat productions and memories in exactly the same way.

Parallel Sets of "Knowledge Sources" as Parallel Communicating Production Systems

An attractive structure for cognitive and perceptual systems is one that uses a set of independent "Knowledge Sources" (e.g. Reddy, 1973, Lowerre, 1976), or "demons" (e.g. Lenat, 1977). Each works at the same time, independently of the others, and therefore could be executed in parallel by one processor in a suitable network of minicomputers, or parallel processors. All communicate with one another by passing information into and out of a common "blackboard" memory. Such Systems cannot be conveniently coded in a standard Production System language like PSG or PAS-2, because the knowledge sources act in parallel, and often contain parallel processes.

Such a system can quite straightforwardly be handled by $(PS)^{2}M^{n}$, as follows: Each Knowledge Source is executed in turn, since each works with its own local memory. Then, after all have completed their processes, a set of parallel productions transfers their (present, intermediate) results to the common Blackboard Memory. Then a second set of parallel productions transfers selected information from the common Memory to the individual Knowledge Sources' memories.

The parallel capabilities of $(PS)^{2}M^{n}$ make the transfer of information into and out of the common Memory very convenient, since there is now no danger of unwanted interactions between Knowledge Sources. And now each Knowledge Source can itself contain a mixture of parallel and serial flows of productions. And each can contain one (or more) separate memories, whereas in a standard PS each subset of memories would have to be protected (in a very cumbersome way) from all Knowledge Sources that did not have access to it.

Most large Production-oriented programs of this sort appear to be Production Systems in spirit, but without being coded in an actual Production System language. The extensions handled by $(PS)^2_M{}^n$ would appear to move in the direction of a language that might be suitable for convenient coding and running of such large production Production Systems.

Compatibility Pairs

A production for relaxation using pairs of compatibility labels (e.g. Zucker, 1977; Rosenfeld and Davis, in Press; Barrow and Tenenbaum, 1976) need only be a 2-tuple, designating the label to be looked at and the context label. When both are found the label is fired into an output Memory. But each label must be search for in its designated memory, and many such productions must be applied in parallel. And to iterate to more distant contexts each must imply the next production to apply. The present system will handle deterministic constraints, if it raises a flag with each success, tests and lowers the flag after each iteration, and then stops iterating after one pass without raising the flag (meaning that nothing new has been accomplished). Extensions to be discussed below will handle probabilistic constraints and dynamic stopping rules, and also situations that mix compatibility pairs with other types of transforms.

A Production System that Uses Compatability Pairs to Choose Among Productions

Zucker, 1977, has suggested that compatibility pair relaxation techniques might fruitfully be used to choose the single judged-best Production to fire next, when a serial Production System finds that more than one Production might fire. (This is the case in those standard Production Systems that, instead of firing the first Production that succeeds, get all Productions that succeed, and then use some criterion to choose among them.)

$(PS)^2M^n$ allows such a system to be coded entirely as a parallel-serial Production System, as follows:

List all the Productions to be chosen among as a parallel set on the MASTER list (the main list of productions - see the Appendix). Each Production implies all of its compatibility labels as dynamically implied Productions on its set of ACTS. Thus all Productions that succeed fire their compatibility labels onto the set of (parallel) productions that will fire next, after this parallel set has been completed. Then this parallel set of compatibility labels will be applied. This procedure continues, as in the proceeding section, until either a single production remains or no more relaxation can be done. Thus the separate relaxation phase is absorbed into and handled by the $(PS)^2M^n$ Production System.

Discrimination Nets, Concept Formation, EPAM

The program uses a tree of Production, each making one test. The Master list starts with a Production that tests for the first node of the discrimination net (really a tree), followed by all Productions on its negative path. If it succeeds it implies the Production on its positive path to be immediately executed in the serial mode followed by all Productions on that node's negative path. Thus the fragment of a discrimination tree shown in Figure 1 needs the Productions shown in Table 1 (a number refers to the test for that node):

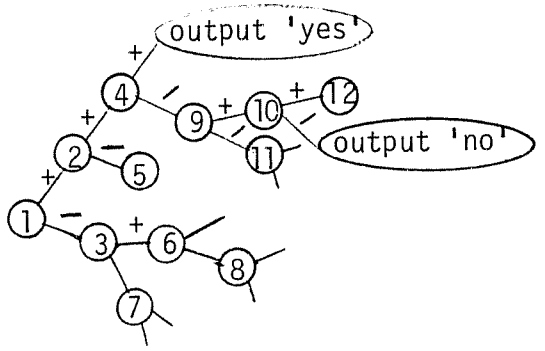

Figure 1

MASTER	= 1, 3, 7...
1	= 2, 5,...
3	= 6, 8,...
4	= output 'yes',9
9	= 10, 11...
10	= 12, output 'no'

Table 1

Note that the only Productions on the MASTER list are the first and the successive '-' (failure) branches from it. All other Productions are implied by Productions that fire.

This program uses a true discrimination tree, where each Production tests only the single next node. This is in sharp contrast to the EPAM program coded in PAS-2 (which uses a template for each entire path in the tree, and has no ability to dynamically imply transforms).

A program that generates nodes to build a true discrimination tree must wait for extended generation capabilities, as discussed below.

Layered Perceptual Systems (E.G. Cones, Pyramids, Hierarchical Relaxation

The use of a sequence of several memories between pairs of which are sandwiched sets of parallel productions can be quite straightforwardly coded as a parallel-serial production system. Productions need only be stored as parallel sets, their INput MEMory and OUTput MEMory specified for each such set, with the OUTput MEMory for one set the INput MEMory for the corresponding subsequent set.

Dynamically implied transforms can be handled as productions that are implied as consequent acts of successful productions. Iterations through relaxation can be handled by using the same INput MEMory and OUTput MEMory for several cycles. These would be especially attractive for parallel-serial layered perceptual systems such as the cones and pyramids being developed by Hanson and Riseman, 1974; Tanimoto, 1976; Uhr, 1972, 1974b, and others.

Such systems would benefit greatly from several extensions (that are projected for future productions systems). These include a) weights, thresholds, and probabilistic productions, b) a convenient way to iterate through a whole array of memory locations (as in the 2-dimensional retinal mosaic), applying the same set of parallel productions in parallel to each cell in the array, and c) techniques for choosing among alternative implications (as in naming and describing), and for deciding when to choose (as in deciding when to stop relaxing).

Heuristic Problem-Solving

A heuristic problem-solver could take advantages of several levels of parallel processes, given a suitable language. Typically, a whole set of heuristic evaluation functions is to be applied in parallel to a node that might be a candidate for further search. And a whole set of such candidate nodes might conveniently be evaluated in this way, in parallel.

A $(PS)^{2}{}_{M}{}^{n}$ program with the following structure can handle such a system:

The set of heuristics is coded as a parallel set of Productions.

For a heuristically guided search for a path from a set of Givens to a Goal (or from Goal to Givens), the Givens (or the Goal) are input as elements in the first WM. The Rules of Inference (sometimes called legal moves, transformations, etc.) are coded as a parallel set of productions and put on the MASTER list. They are applied, giving the buds (which are put into a second WM) from the presently achieved expressions, delete all elements in the first WM, and fire the set of heuristics (coded as a second parallel set of productions). The heuristics now look at these budded expressions, and accumulate the evaluation of each. They are followed by a serial set of productions that chooses and puts into the first WM the most highly valued expression(s), deleting all expressions in the second WM. The Rules of Inference are again applied, and this process continues.

I have ignored a number of important details, but almost certainly several more Working Memories will be preferable - e.g. to store the pointers back from buds to their fathers for later use in getting the solution-path, and to get that path.

Some Additional Extensions That Would Seem to be Useful

There are several further extensions that would appear to make for substantial further improvements in the ease of coding of cognitive and perceptual systems, and the resulting transparancy and clarity of structure of the resulting systems. These include the following:

A) Probabilistic transforms would allow for the merging of multiple implications, from perceptual characterizers or problem-solving heuristics. This entails rather simple extensions (that will be reported on in a subsequent paper): 1) Associated with each element of a condition will be a number (that can be interpreted as a weight, probability, fuzzy value, or whatever the programmer desires), associated with each production will be a threshold, and associated with each implied act will be a weight. When the combined weights of successful conditions exceeds the Production's threshold, the Production will be considered to succeed, and fire its implications with their assigned weights. 2) Several implications of the same thing will have their weights merged. 3) When this is specified in the program, then the system will choose among alternative implications from among some mutually exclusive set that the programmer has specified.

B) A set of Productions should be able to iterate conveniently through an array of memories. This would allow for very convenient and straightforward coding of perception systems, e.g. pyramids and cones, and relaxation labelling, where sets of transforms (coded as Productions) could be applied to the different cells, or regions, of the retinal image, or internal abstracted images.

C) A system should have a more general set of capabilities for creating and erasing lists (whether Productions or Memories), and adding and deleting information to and from these lists. This would allow for the convenient building, and modifying, of complex list structures and graphs of information. If done with as much generality as possible, it should also make Productions and Memories, and their manipulations, as similar as possible.

Discussion

Today's production systems have a heritage much older than Post's productions, and much broader than problem-selving. For Post productions are probably the single Turing machine-equivalent formulation that most directly embodies that elusive (and possibly imaginary) entity, "intuitive thinking." The rules of inference of Euclid's geometry, and of Aristotle's syllogism, are very similar to productions. Thus modus ponens (If A then B; A is true, therefore B is true), and the more modern mathematical function or mapping, which transforms from sets of things to new sets of things, as specified, are closely related.

And this type of thinking has repeatedly been re-invented, as the basis of diverse systems of science and of thought. The linguist's rewrite rules and transformations, the physicist's Feynam diagrams, the psychologist's S-R mappings, the lines of reasoning of detective, diagnostician or trouble-shooter - these are only a few of the examples of the kinds of thinking people, whether technically trained or not, frequently if not almost always fall into.

So an embodiment of thinking in terms of productions would seem especially desirable. We could use it conveniently; we could understand it; and it might well be appropriate, and powerful. But it is therefore especially important that the particular basis we choose be sufficiently flexible and powerful so that it does not limit the person who programs, or formulates, a system.

The tree major extensions to Production Systems incorporated into $(PS)^2M^n$ - 1) the ability to apply productions in parallel, when desired, 2) the partitioning of information into several working memories, and 3) the firing of productions to be applied next by productions that are fired - often appear to go together, and to

complement and enhance one another. But one might prefer to use only one, or two, of these extensions, for the following reasons:

1) If it is known that all programmers who will use the production system language will code all their programs using only serial productions, then the parallel capability is not needed. Or one might want to force all programs into the serial discipline, to try to develop as powerful as possible a set of serial algorithms (e.g. so that they will execute efficiently on serial computers). This might be of more general value if we had reason to think that thinking is best handled serially. But many cognitive processes, especially in perception and associative memory search but also in heuristic deductive probelm-solving, appear to have important parallel components. And although yesterday's computers are serial, today's are beginning to be parallel, and tomorrow's will become increasingly parallel.

2) From a certain esthetic perspective as to the meaning of "simplicity" the single Working Memory seems simpler than several Working Memories. But if we define "simple" as the most direct mapping of the flow of processes effected by the program into that program's architecture, then if the program uses several memory stores it seems less simple to throw them all together into one, and then do a lot of otherwise unnecessary searching and matching to find the one that is needed.

3) Productions that fire productions to be applied may well violate the simple modular structure of standard production languages, with their top-down flow through the stack of productions.

But in my opinion they more closely reflect the actual operative flow of the program, in the sense of the productions that actually will fire. Without this capability the programmer must resort to cumbersome devices that make use of the addition and deletion of special data symbols to direct the program's flow. These devices clutter up the program, waste time in execution, and make it far more difficult to understand what the program is doing. Thus (in my opinion) they superficially introduce better, simpler, structure when in actuality they introduce far worse problems of structuring.

In any case, firing productions serves chiefly to facilitate serial processes, and can most easily be dispensed with for the parallel perceptual and associative memory systems that have chiefly motivated these extensions. The more serial the process, the greater the saving in computer time. For example, a typical serial "discrimination net" (really a tree) program for concept formation (or Waterman's version of EPAM) will need to try to fire one half of its total set of productions for each discrimination-step (on the average, assuming all branches of the tree are equally likely). Thus as the tree grows larger (as it will, unless the system is only asked to handle toy problems) the traditional PS program takes longer. But a $(PS)^2M^n$ program, since each node directly fires the appropriate next production to try to fire, remains as fast no matter how large the discrimination net might become.

The standard and the extended Production Systems presented in this paper are relatively simple programs because they are coded in Snobol/EASEy, which itself is a powerful pattern-matching production

language (that also has many of the features, in particular multiple memories, incorporated into $(PS)^{2}{}_{M}{}^{n}$).

In fact Snobol/EASEy might be used directly as a vehicle for coding a Production System. Its built-in left-to-right match is in ways more powerful (with much more back-tracking) than the matches used in most Production Systems. And its multiple memories, powerful control structure, arithmetic and inequalities, and recursive functions can be very convenient.

But a language like Snobol, and a typical Snobol-encoded program, do not have the clean, simple, modular structure of a Production System. A very interesting and fruitful line of future development would seem to lie in the combining of these two approaches to Productions.

Appendices

A) Format Rules for Coding and Inputting Productions

Building Productions and Memories from Elements

Both Productions and Memories are linear strings of elements. They are exactly the same, except that a Production contains an equal sign ('=') that separates the IF-Conditions (those that must match for the Production to succeed and fire) on its left-hand-side from the THEN-Consequences (the acts that will be effected if the production succeeds) on its right-hand-side.

An element is enclosed in element brackets ('<' ... '>'), and made up of a string of sub-elements, each followed by one (or more) space(s). A sub-element is a string of symbols (not including spaces). If the first symbol is an asterick ('*') the sub-element is a variable that is to be bound and given a value during the matching process (see below). Elements can be grouped in sub-strings (for purposes of specifying the Memories to which they are to be applied, as described below), by ending each sub-string with a semi-colon(';'). (Note that many Production Systems handle elements with embedded parentheses. The $(PS)^2M^n$ program given below needs a more powerful MATCH routine to handle such constructs.)

Matching Productions with Memories

A Production is considered to have succeeded in matching its elements with memory elements only if all its elements are matched, according to the following procedure: Each Element is looked for in a specified Memory. The program takes the first Memory input to it during its initializing phase as the dominant Memory (called 'INMEM')

into which Productions are to look, and the second Memory (or if there is no second Memory, the first) given it as the dominant Memory (called 'OUTMEM') for firing out its acts (if it succeeds). But these dominant memories can be changed, as desired by the programmer, by specifying new dominant INMEM and/or OUTMEM as part of a card that specifies the MODE of the subsequent set of Productions (described below). (The format used is :(new dominant INMEM): and/or ;(new dominant OUTMEM); without any separating spaces.) A temporary different Memory can be specified for any sub-string of elements that follows it by :(new temporary INMEM): and ;(new temporary OUTMEM);. These temporary memories hold only for the sub-string of elements that they start.

The program looks for each Element in the (either the current dominant or the just-specified temporary) Memory. The first sub-element of the Element must exactly match the first sub-element of some element in the specified Memory, and all subsequent sub-elements must match some sub-element of that same Element, in order. (Thus the match insists that the Production Element be a left-anchored ordered sub-string of the Memory element that it is considered to match.) Any variables are assigned as their values the first sub-element that they match, and the match from that binding on-ward insists upon finding that assigned sub-element value when that variable is to be matched again. (Thus there is no attempt to re-bind variables.) The first element that matches in the Memory is taken as the matching element (which therefore binds any variables).

This procedure is fairly typical of Production Systems, although there has been a great deal of variety in the details of the match.

But virtually all Production Systems insist that the first sub-element of the Production match the first sub-element of the Memory element, and bind variables permanently - as does this system. I should note that this contrasts with a language like Snobol, that will try all possible bindings, in an attempt to make a much more general (and, potentially, explosive in terms of time taken) match.

Acting, When the Production Succeeds

Each Element of the THEN-Consequences specifies an act to be effected, consequent upon the perfect match of all the Elements of that Production. As with IF-Conditions the Act is effected into the current Dominant Memory unless a temporary output Memory has just been specified for its sub-string.

The first sub-element of each Act element specifies its Operation (e.g. 'DELETE' or 'ADD' or 'OUTPUT', which immediately outputs to the external world) or 'STOP', which immediately ends this run), and the rest of the elements specify its arguments. Currently coded Acts include:

ADD (the specified new element to the output memory specified);

DELETE (the specified element from the memory specified);

SAY (outputs the specified string);

STOP (ends the run, outputting "STOPPED");

GEN (generates the new specified production, inserting it where specified on the MASTER list);

PROD (puts the name of the specified production on the list of productions to be fired next).

Coding a Standard Production System, with a Single Working Memory

If only a standard Production System is needed, coding is quite simple and straightforward. Each Production is input with a P in column 1 (followed by cards with a + in column 1, if needed because the production does not fit on one card). The Production's elements then follow, ended by a right-bracket. The Memory is input in the same format, but with an M in column 1.

That is all that is needed for a Parallel production system, since the MAINMODE is initialized in the program to PARallel. But if a serial system is desired (and that is the standard mode for today's Production Systems) one additional input card is needed:

M MAINMODE SER]

The program is initialized to go DOWN to the next Production after a Production succeeds. That is a common alternative used in today's systems. But if the alternative that the program return to the first Production after success is preferred one additional input card is needed:

M FLOW TOP]

Note that the programmer need not name the Productions. The program names them P1,P2,...Pn. If the program has generate acts that refer to Productions, the programmer can figure out what these assigned names will be, and use them accordingly. But it may be more convenient for the programmer to assign names, either to those productions that he refers to in his program, or to all productions. In that case the name is put right after the right-bracket on the Production's input card. (For the programmer's convenience these named Productions are also numbered.)

Inputting and Formatting a Set of Productions and Memories, and Initiating a Run

The program is initialized so that the Dominant Mode of processing (called MAINMODE) is Parallel (called PAR) and the Dominant Flow of Productions (called FLOW) is to the next Production (called DOWN) after a production fires. The system outputs the string: 'STARTING A NEW SET OF PRODUCTIONS.' and then waits for a set of inputs, that will initialize its memory with Productions, Memories and other information, and initiate a run.

These inputs must be of the following form:

In Column 1 the TYPE is specified, followed by a space, followed by a string of INFOrmation, followed by an end-bracket (']'), followed by a NAME (the NAME is optional in the case of a Production).

The program will handle any number of runs through the same Production System, or modified Production Systems, or completely different Production Systems. A run begins with the input of the needed Productions, Memories or Memory, and parameters.

Productions must be given TYPE = P, and input in order. They will be numbered i = 1,2,...n, and given the NAME Pi unless a NAME has been specified by the programmer.

Memories must be given TYPE = M. The first two Memories must be input in order if there are more than one (so that the program can assign them as the dominant INput MEMory and OUTput MEMory) but all other memories can be input in any order desired.

Other parameters and lists, including Productions that are implied by other Productions but are not on the MASTER list of

Productions are input with TYPE = X and must have a NAME. This is the way to change MAINMODE (to either SER - for SERial - or PAR - for PARallel) and FLOW (to either TOP - to return to the first Production when a Production succeeds - or DOWN - to apply the next Production when a Production succeeds).

When any of the above does not fit on one card it can be followed by continuation cards with TYPE = + (that is, a + in the first column), with the NAME assumed to be the same.

The MODE can be assigned and re-assigned with a TYPE = MODE card, where the MODE is either PAR or SER. These must be input interspersed among the ordered Production cards, at the actual points where they are to change the MODE. Note that the format is:
MODE (PAR or SER)]

An element is a bracketed ordered list of sub-elements, each followed by one space. A sub-element is an ordered list of symbols (not including the space symbol).

A Memory is an ordered list of elements.

A Production is an ordered list of elements, with the following additions: An equal sign (' = ') (surrounded by spaces) separates the Production's IF-CONDITIONS from its THEN-CONSEQUENCES. Elements must be grouped into a sub-set (indicated by a trailing percent ('%') sign) if they are to be looked for in, or fired into, some Memory other than the current Dominant Memory, with the temporary memory specified at the start of that sub-set (see below). This means that first comes the sub-set of elements that refer to the Dominant Memory, followed by %, then the temporary Memory, then the sub-set that will refer to it, then a % (then other sub-sets referring to still other memories, using the same format).

Changes in Memories to be looked at and to be fired into must be handled in the following ways:

When the Dominant Memory is to be changed (that is, from then on until the next change) the change(s) must be indicated as part of a MODE card. The INput MEMory must be surrounded by colons (':(new input memory):') and the OUTput MEMory must be surrounded by simicolons (';(new output memory);'). If both are changed, input must precede output, and the new MODE (either 'PAR' or 'SER') must always be given,placed last. No spaces can be used between memories and mode. E.g.: MODE :MEM7:;BLACKBOARD;PAR (sets INMEM as MEM7, OUTMEM as BLACKBOARD and PARallel MODE.)

When some other Memory is to be used by some of the elements in a Production that Memory must be specified at the start of a sub-set of elements (to which that memory refers). That sub-set of elements is ended with a percent sign ('%'). A temporary Input MEMory is surrounded by colons (':(temporary input memory):') and a temporary output memory is surrounded by semi-colons (';(temporary output memory);') with no spaces allowed.

After a run, the system can be re-initialized, so that all its Productions and Memories (that is, all lists input with TYPE = P or M) are erased, by inputting one card with TYPE = INIT - that is, INIT]

(Note that this does not erase any of the parameters or dynamic transforms that were input with TYPE = X cards. If their erasure is desired - as may be the case if a lot of dynamically implied productions might otherwise waste too much memory space (but note that they will not have any effect on subsequent runs) - they can simply be input with TYPE = M cards, rather than TYPE = X cards.

That is possible because only the first two memories will be noticed by the program - but that means that when only one Memory is specified it must be specified twice as the first two memories (its contents need be given only the second time).)

If only a few things need to be changed for the next run, the programmer should not use the INIT card, but instead simply input cards (of TYPE = M) with the NAMEs or the Productions, Memories, and other lists to be changed.

A run is initiated with a card whose TYPE = GO - that is, GO]

(NOTE that a second run can be initiated through the same Production System by having two GO cards immediately following the input of that system. But the second run will use the system that results from the first run. The only ways the initial system can be used twice is to input it again, or restore it to its original form (which will usually be too complex a process to attempt). It would be quite simple and straightforward to have the system store a copy of the original Production System for subsequent runs. But it did not seem worth bothering with such an option in the present demonstration system, since it might easily waste excessive amounts of memory.)

A Detailed Statement-by-Statement Description of the $(PS)^2M^n$ Programs

Initializing the Program and Setting up the Productions and Memories

Statements I1 through I22 Initialize the system, Inputting its Productions, Memories and parameters, and re-initializing for subsequent runs.

I1 sets the maximum number of Productions to be TRIED in a run (after which the run will be terminated). (TRIED = 9999 in the sample program.)

I2 sets the MAIN MODE for processing to be PARallel (the other option is SERial).

I3 sets the main FLOW to be DOWN (to the next Production after a Production fires; the other option is TOP, which goes to the first Production after a Production succeeds).

I4 outputs that a new run is starting.

I5 inputs INFOrmation of a specified TYPE and with an optional NAME, and goes to the statement for that TYPE.

I6-I12 handle Productions: I6 assigns the next integer to this Production. I7-I8 give it an integer NAME if no name was input, I9 stores this Production's NAME on the MASTER list, I10 stores the INFOrmation on this Production's contents under its NAME, I11 outputs this NAME and its contents (from the present card), and I12 copies this NAME (in case there are continuation cards).

I10-I12 handle inputs that are parameters and dynamically implied Productions. (No integer number is assigned, and the NAME is not listed on MASTER; otherwise the same processes are effected.)

I13, I10-I12 input Memories.

I14-I15 input continuation cards for lists too long for a single card.

I16 inputs a change MODE (PARallel or SERial).

I17-I18 re-INITialize for a new run, erasing all MEMORIES and MASTER Productions.

I19-I22 put the MAINMODE at the start of the MASTER list and get the dominant INput MEMory and OUTput MEMory.

Applying the Productions

Statements P1 through P15 form the main program, applying Productions, looking for their IF-CONDITIONS and firing the THEN-CONSEQUENCES of those that succeed, until some stopping rule is invoked:

P1-P2 initialize, making a Copy of the MASTER list of Productions and setting the number of FIRED Productions to zero.

P3 gets the next MODE and set of Productions TODO from CMASTER, P5-P6 get (if they are specified) the new dominant INMEMory and/or OUTMEMory from the start of MODE, and the system goes to the indicated MODE (either PARallel or SERial).

If CMASTER has been emptied, P4 checks whether any Productions have been fired in this pass: If yes, it goes to TOP to make another pass through MASTER; if no, the run has ended, and it goes to BEGIN a subsequent run.

P7-P9 handle the SERial mode, by getting just one Production TODO and putting any LATER ones back onto CMASTER.

P10 calls the APPly function for this set of (one or more) Productions TODO.

P11 FIREs any successful Productions' ACTs.

P12 checks whether any Productions have been fired (they will have been put onto ACT). If no, the program goes to the statement indicated by the FLOW (either DOWN or back to the TOP). If yes, these implied Productions are put at the start of CMASTER, with the specified mode (or the dominant MAINMODE if no mode is specified).

The Functions that APPLY and MATCH Productions, and ACT

Statement A0 defines the APPLY function. A1 gets the next PRODuction from TODO, A2 adds 1 to the number of Productions TRIED (the program ending this run if it reaches 9999), and A3 calls the MATCH function.

M0 defines the MATCH function, which tries to match one PRODuction. M1 gets the PRODuction's CONDitionS and ACTS. M2 gets the next sub-set of elements (IFS) from the CONDitionS and M3-M5 get any temporary INput MEMory indicated.

M6 gets the next element from IFS (its FIRST sub-element and the REST of its sub-elements). (If no more M7 erases this MEMory and its Copy (CMEM).) M8 makes a Copy (CMEM of this MEMory) for this set of elements. M9 gets the first occurrent as a first sub-element of the FIRST sub-element in the CMEMory and the REST of that element in Memory (MREST). M10 lists this as a possible ELement for match. M11-M13 look for each PART of the REST of this element, but first M17 checks whether this PART is a variable (indicated by an asterisk ('*') at its start). If it is, M18 checks whether it is already bound and if not M19 gets the next sub-element in the REST of this Memory element as the VALue that binds it, and M20 lists it as a possible binding on MAYBIND. But if M19 can't get a binding (because no more

sub-elements remain), or M21 can't match the already-bound variable, this element is abandoned and M9 tries the next element in memory that matches the FIRST part of this element (if no more match the whole match fails).

After M13 has matched all PARTs of an element M14 erases that ELement from MEMory and M15-M16 add its MAYBINDings to the chosen BINDINGS. (NOTE WELL how only the first bindings of variables are used, rather than having the program try all possible bindings.)

After M2 has matched all IFS in this Production's CONDitionS, M21-M26 binds all this Production's ACTS' variables (indicated by asterisks ('*')), listing them on APPLY, and adds 1 to the count of Productions FIRED.

After all the Productions TODO have been APPLYed, any ACTS implied by successfully fired Productions are effected by the ACT function.

ACT0 defines ACT. ACT1-ACT3 gets the next sub-group of acts (THEN) and their MEMory. ACT4 gets the next element (an OPeration followed by its ARGument) and goes to that OPeration.

Operations presently coded include the following:

ADD adds the ARGument as a new element of the indicated MEMory (ACT5).

DELETE erases the ARGument if it is a complete element of the indicated MEMory (ACT6).

SAY outputs the ARGument (e.g. the concept, name, or other answer) (ACT7).

STOP ends the run, outputs STOPPED, and will return the program to BEGIN any subsequent run (ACT8).

GEN will GENerate a new Production, listing it right after WHERE on the MASTER list (ACT9-ACT13). (Its variables were bound on BINDINGS; ACT11 surrounds its Elements with <...>).

PROD indicates a PRODuction to be fired, which is put onto ACT (ACT14).

Additional acts can easily be added, each as a separate set of code labelled with the act's OPeration, the act's ARGument formatted appropriately.

```
(A Parallel-Serial Production System with Many                Statement No.
(Working Memories - (PS²)Mⁿ

BEGIN     TRIED   =   9999                                          I1
(Assume PARallel mode and DOWNward FLOW till input specifies otherwise
          MAINMODE  =   'PAR'                                       I2
          FLOW      =   'DOWN'                                      I3
(INputs Productions, Memories, MODE, and other parameters
(Subsequent runs can modify any Productions and Memories, or
(INITialize, erasing all, then building anew.
          OUTPUT   'STARTING A NEW SET OF PRODUCTIONS.'             I4
IN        INPUT    TYPE INFO  ]  NAME  [+ $('I'TYPE) - end]         I5
(Production can have programmer given NAME (else uses next integer)
IP        P  =  P  +  1                                             I6
          IDENT (NAME, NULL)  [ - IP2 ]                             I7
             NAME   =   'P'  P                                      I8
IP2       On MASTER list NAME                                       I9
(Initializes dynamically implied Productions and other things not
(listed on Memories
IX        $NAME  =  INFO                                            I10
I2  output P'  ' NAME ' = ' INFO [IN]                               I11
(Initializes Memories
          CNAME  =  NAME                                            I12
IM        on MEMORIES list      NAME  [IX]                          I13
(Adds information to Memory list and long Production (comes
(right after, or must give its name)
I+        IDENT(NAME, NULL) yes - NAME = CNAME                      I14
          on $CNAME  list  INFO   [I2]                              I15
(Input change of MODE (and :inmem: and/or ;outmem; - no spaces allowed
IMODE     on MASTER set  >  INFO  [ I2 ]                            I16
(INITialize card erases all Productions and Memories
IINIT     from $(MASTER MEMORIES) get THING = [-IN]                  I17
          erase  $THING  [IINIT]                                    I18
(Starts processing the just-input Productions and Memories
IGO       MASTER = > MAINMODE MASTER '>END'                         I19
          output 'MASTER LIST = ' MASTER                            I20
```

```
(Assumes first Memory(s) are INMEM and OUTMEM (till told otherwise
(by Productions)
          from MEMORIES get INMEM  OUTMEM   [+  TOP]            I21
          from MEMORIES get INMEM                               I22
          OUTMEM  =  INMEM                                      I23
TOP       CMASTER  =  MASTER                                    P1
          erase   FIRED                                         P2
DOWN  at start of CMASTER get >MODE TODO >MODE2 = >MODE2 [+ MEM] P3
(If no Productions fired in complete pass through MASTER the run
(has ended.
          GT(FIRED, 0  )  [ + TOP  -  BEGIN]                    P4
(Optionally INMEMory and OUTMEMory can be changed at each MODE
(indication
MEM       at start of MODE get  : INMEM : =                     P5
          at start of MODE get  ; OUTMEM ; = [$MODE]            P6
(In SERial mode applies only 1 production at a time
SER       from TODO get TODO LATER till end                     P7
          INDENT( LATER , NULL )  [  +  PAR  ]                  P8
          at start of CMASTER list > MODE LATER                 P9
PAR       APPLY( )        [ - BEGIN]                            P10
FIRE      ACT( )          [ - BEGIN]                            P11
(If any new Productions have been implied (onto ACT) will APPLY
(them right now
          IDENT( ACT NULL )     [ + $FLOW ]                     P12
(Uses specified mode of MAINMODE
          at start of ACT get > [ + FI2 ]                       P13
          at start of ACT list MAINMODE                         P14
(Puts implied Productions at start of CMASTER, to be pulled off
(and applied by DOWN
FI2       at start of CMASTER list MAINMODE ACT  [DOWN]         P15
(Will APPLY a Parallel set (so if serial there's only 1 PRODuction
in TODO)
APPLY     DEFINE:  APPLY(  )                                    A0
AP1       from TODO get  PROD =  [ - return ]                   A1
          LT(TRIED , 0 ) TRIED = TRIED - 1 [ freturn ]          A2
          MATCH($PROD) [AP1]                                    A3
```

```
MATCH      DEFINE: MATCH(PROD) erase BINDINGS                          M0
MA1        from  PROD  get  CONDS  '='    ACTS till end               M1
(Get the MEMory and IFS from CONDITIONS
MA2        from CONDS get     IFS % + [-BINDACTS]                      M2
           start IFS get : MEM : = [MA8]                               M3
(Get the FIRST subelement and the REST of the next element
           MEM = INMEM                                                 M4
MA8        MEM = $MEM                                                  M5
MA3        from IFS get < FIRST REST > = [+ MA5]                       M6
           erase MEM  CMEM  [MA2]                                      M7
MA5        CMEM = MEM                                                  M8
(Get the next element whose FIRST subelement matches
MA9        from CMEM get LEFT < that FIRST MREST > = [- freturn]       M9
           EL = FIRST MREST                                            M10
           CREST = REST                                                M11
           erase MAYBIND                                               M12
MA7        from CREST get PART = [+ MA4]                               M13
(erase this successfully matched Element from MEMory and add its
(BINDINGS (from MAYBIND)
           from  MEM  get < that EL > =                                M14
           on BINDINGS set MAYBIND                                     M15
           erase MAYBIND  [MA3]                                        M16
MA4        at start of PART get '*' =  [ - MA6]                        M17
           from $(BINDINGS MAYBIND) get that PART PART [ + MA6]        M18
(Bind this variable PART with VALue got from MEMORY
           from MREST get VAL =  [ - MA9 ]                             M19
           on MAYBIND list PART VAL  [ MA7 ]                           M20
MA6        from MREST get LEFT that PART = [ + MA7 - MA9]              M21
(Use BINDINGS got in MATCH to assign values to VARiables in ACTS
BINDACTS   from ACTS get LEFT '*' VAR = [ + BI2 ]                      M22
           FIRED = FIRED + 1                                           M23
           on APPLY list ACTS [return]                                 M24
BI2        from BINDINGS get that VAR VAL                              M25
           on APPLY list LEFT VAL [BINDACTS]                           M26
```

```
ACT         DEFINE: ACT( )                                          ACT0
ACT1        from APPLY get    THENS  % = [ - return]                ACT1
            at start THENS get ; MEM ; = [+ACT2]                    ACT2
            MEM = OUTMEM                                            ACT3
ACT2        from THENS get < OP  ARG > = [ + $OP - ACT1 ]           ACT4
ADD         start $MEM set < ARG > [ACT2]                           ACT5
DELETE      from $MEM get < that ARG > = [ACT2]                     ACT6
SAY         output  ARG  [ACT1]                                     ACT7
STOP        output 'STOPPED' [freturn]                              ACT8
(The ARGument will be in Production form with variables that
(have been assigned values
GEN         from ARG get  WHERE  =                                  ACT9
            P  =  P  +  1                                           ACT10
            GEN1 from ARG get '(' EL ')' = < EL > [+GEN1]           ACT11
            on $('P'P) list ARG                                     ACT12
GEN2        from MASTER get that WHERE = WHERE 'P' P [ACT2]         ACT13
(Will imply a PRODuction (put on ACT) to fire next
PROD        on ACT list ARG  [ACT1]                                 ACT14
end
```

A Standard Serial Production System with a Single Memory

The following is a Snobol/EASEy program that handles a standard Production System, one that applies a sequence of Productions in series, all inputting from and outputting to a single working Memory. To keep the program as simple as possible (and to reflect the workings of any particular Productions System, like PSG or PAS-2), only one type of flow is built in - a return to the first Production (i.e., as done by FLOW = TOP in $(PS)^2M^n$). The system will end when either a STOP act is fired or it has TRIED 9999 Productions; it will not re-initialize itself and handle a sequence of runs. (Nor, of course, does it handle more than one Memory, or parallel sets of Productions, or dynamically implied Productions - since these are not handled by standard Production Systems.)

The statements are numbered at the right to indicate resemblances to statements in $(PS)^2M^n$, with .A indicating that some (usually minor) Alterations have been made. This program might be made even more similar to the (sub-set of) code in $(PS)^2M^n$, if functions were used, as there, to MATCH and to ACT. But for variety, and to indicate more clearly the program's flow, it is coded without functions.

This program might best be examined by comparing its statements to the similar statements in $(PS)^2M^n$. But a brief description is given now:

Statements I1-I5 input the Productions (which the program will number from P1 through Pn) and the single Memory (which must be input first, and whose name is P1).

P1-P2 loop through the Productions (if the last has been tried, unsuccessfully, as indicated by CMASTER being empty, the system has

made an entire pass through all Productions without firing any, and therefore the entire run ends).

M1-M17 try to MATCH this Production. If M2 finds no more CONDitionS ACT1-ACT12 fire the successful Production's ACTS:

ACT1-ACT4 get any VARiables' VALues from BINDINGS.

ACT5-ACT13 effect the ACTS (almost exactly as done in $(PS)^2M^n$), except for the Single Memory and the absence of implied Productions).

M4-M17 are almost identical to $(PS)^2M^n$ in the way they attempt to MATCH the Production (except for the absence of specifications of different Memories for groups of elements). Each element of the Production is matched with the first matching element of the (single) MEMory, their FIRST sub-elements both coming first, and the first BINDING of a VARiable used throughout (or else the match fails).

```
(A Standard Production System (Serially Applies Productions to
(a Single Memory)
(First inputs the Memory, then the Productions
(Program assigns all names, P1, P2,...Pn (P1 is the Memory)
IN        input  INFO  ] TYPE  [ + $('I' TYPE) - end]    I5.A     I1
I         P = P + 1                                      I6       I2
          on MASTER list  'P' P                          I9       I3
          $('P' P)  =  INFO  [IN]                        I10.A    I4
(Continuation cards  Must follow immediately
I+        on $CNAME list INFO [IN]                       I15      I5
IGO       from MASTER get MEM CMASTER till end           P1.A     P1
NEXT      from CMASTER get PROD = [- end]                A1.A     P2
N2        from $PROD get CONDS  '='  ACTS till end       M1.A     M1
MATCH     from CONDS get < FIRST REST > = [+ M1]         M6.A     M2
(After successful MATCH, Production's ACTS output, GO back
(to top Production
ACT1      from ACTS get < OP ARG > = [ - IGO ]           ACT4.A ACT1
ACT3      from ARG get '*' VAR = '*%'  [ - $OP ]         M22.A  ACT2
          from BINDINGS get that VAR  VAL                M24.A  ACT3
          from ARG get '*%' = VAL ' ' [ACT3]             M25.A  ACT4
ADD       start $('P' P) set < ARG > [ACT1]              ACT5.A ACT5
DELETE    from $('P' P) get < that ARG > [ACT1]          ACT6.A ACT6
SAY       output  ARG  [ACT1]                            ACT7   ACT7
STOP      output  'STOPPED'  [ end ]                     ACT8.A ACT8
GEN       from  ARC  get  WHERE  =                       ACT9   ACT9
          P  -  P  +  1                                  ACT10  ACT10
GEN1      from ARG get '(' EL ')' = < EL > [ + GEN1 ]    ACT11  ACT11
          on  $('P' P)  list  ARG                        ACT12  ACT12
          from MASTER get that WHERE = WHERE 'P' P [ACT1] ACT13 ACT13
M1        CMEM  =  MEM                                   M8.A   M3
MA9       from CMEM get LEFT < that FIRST MREST > = [+ M3]M9.A  M4
          LT(TRIED, 9999)  TRIED = TRIED + 1 [ - end ]   A2.A   M5
M3        EL  =  FIRST MREST                             M10.A  M6
          CREST  =  REST                                 M11    M7
          erase  MAYBIND                                 M12    M8
MA7       from CREST get PART  =  [ + MA4 ]              M13    M9
```

```
        from  MEM  get < that  EL > =                          M14    M10
        on BINDINGS set  MAYBIND                               M15    M11
        erase  MAYBIND  [MATCH ]                               M16.A  M12
MA4     at start of PART get  '*'  = [ + MA6]                  M17    M13
        from $(BINDINGS MAYBIND) get that PART PART [+ MA6] M18 M14
        from MREST get VAL = [ - MA9]                          M19    M15
        on MAYBIND list PART VAL [MA7]                         M20    M16
MA6     from MREST get LEFT that PART = [ + MA7 - MA9]         M21    M17
end
```

References

Davis, R. and King, J. An overview of production systems, Comp. Sci. Dept. Memo AIM-271, Stanford Univ., 1975.

Duff, M. J. B. CLIP 4: a large scale integrated circuit array parallel processor, Proc. IJCPR-3, 1976, 4, 728-733.

Erman, L. D. and Lesser, V. R. A multi-level ortanization for problem-solving using many, diverse, cooperating sources of knowledge, Proc. 4th IJCAI, 1975, 483-490.

Feigenbaum, E., Buchanan, B. and Lederberg, J. On generality and problem-solving: a case study using the DENDRAL program. In B. Melzer and D. Michie (Eds.), Machine Intelligence 6, Edinburgh U. Press, 1971, pp. 165-190.

Griswold, R. E., Poage, J. F. and Polonsky, I. P., The SNOBOL4 Programming Language, Englewood-Cliffs: Prentice-Hall, 1968.

Hanson, A. R. and Riseman, E. M. Pre-processing cones: a computational structure for scene analysis, Tech. Rept. 74C-7, Univ. of Mass., 1974.

Lenat, D. B. Automated theory formation in mathematics, Proc. 5th IJCAI, 1977, 833-842.

Lipovski, J. On a varistructured array of microprocessors, IEEE Trans. Computers, 1977, 26, 125-138.

Lowerre, B. T. The Harpy Speech Recognition System, Unpubl. Ph. D. Diss., Carnegie-Mellon Univ., 1976.

McDermott, J., Newell, A. and Moore, J. The efficiency of certain production system implementations, Comp. Sci. Dept. Tech. Rept., Carnegie-Mellon Univ., 1976.

Newell, A. and Simon, H. A. Human Problem Solving, Englewood-Cliffs: Prentice-Hall, 1972.

Newell, A. and McDermott, J. PSG manual, Comp. Sci. Dept. Tech. Rept., Carnegie-Mellon Univ., 1975.

Reddy, D. R., et al. The HEARSAY-1 speech understanding system, Proc. IJCAI-3, 1973, 185-193.

Rosenfeld, A. and Davis, L. S. Hierarchical relaxation, In: Machine Vision (Edited by A. R. Hanson and E. M. Riseman), New York: Academic, in press.

Rychener, M. D., Production systems as a programming language for Artificial Intelligence applications, Unpubl. Ph.D. Diss., Carnegie-Mellon Univ., 1976.

Tanimoto, S. L. Pictorial feature distortion in a pyramid, Comp. Graphics Image Proc., 1976, 5, 333-352.

Tennenbaum, J. M. and Barrow, H. G. IGS: A paradigm for integrated image segmentation and interpretation, Proc. 3rd IJCPR, 1976, 504-513.

Uhr, L. Layered "recognition cone" networks that pre-process, classify and describe, IEEE Trans. Computers, 1972, 21, 758-768.

Uhr, L., EASEy: An English-like programming language for artificial intelligence and complex information processing, Comp. Sci. Dept. Tech. Rept. 233, Univ. of Wisconsin, 1974. (a)

Uhr, L. A model of form perception and scene description. Computer Sciences Dept. Tech. Rept. 231, Univ. of Wisconsin, 1974. (b)

Waterman, D. A., Adaptive production systems, Proc. 4th IJCAI, 1975, 296-303.

Zucker, S. W. Relaxation labeling and the reduction of local ambiguities, Proc. IJCPR-3, 1976, 4, 852-861.