PSG MANUAL

A. Newell and J. McDermott
September, 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213

CONTENTS PAGE

I. INTRODUCTION

> PRODUCTION SYSTEM ARCHITECTURES

> A production system architecture is a particular kind of information processing system; it is a control structure which supports 1) a set of productions, each production having a condition side containing zero or more condition elements and an action side containing one or more action elements; and 2) a working memory (a collection of data structures) from which condition elements can be determined to be true or false and on which action elements operate. If all of the condition elements in a production match elements in working memory, then that production is satisfied and is executed; when it is executed some of its action elements may modify working memory; this modification may result in some other production becoming satisfied and being executed, and so on.

> PSG

> PSG (for production system version G) is an architecture which was designed to faciliate the building and testing of psychological theories and for carrying out other explorations with production systems. In PSG, a production is satisfied if its condition elements all match the elements in the part of working memory called STM (short term memory). When a true production is executed, those elements in STM which matched the condition elements of the production are rehearsed (eg, brought to the front of STM); then each action element either modifies STM, modifies some other part of working memory, evokes the execution of other productions or production systems, or carries out operations external to the working memory (such as printing data at the terminal).

> OPTIONS

> There are, of course, many alternative hypotheses, purporting to help account for different aspects of human cognitive behavior. These can be combined in a variety of ways to produce a theory of the control structure of human information processing. PSG offers the user a set of options which make it relatively easy to test different combinations of these hypotheses.

> For example, one of these options is the CONFLICT.1 option which offers the user several different ways of specifying which production is to be executed whenever two or more productions are true at the same time. Two of the option settings for the CONFLICT.1 option are PD.ORDER and STM.ORDER. Thus the CONFLICT option allows the user to decide whether to let the order of the productions in the production memory or the order of the elements in STM determine which production is to be executed. For example, if three productions (call them PD1 PD2 and PD3) are true at the

same time, and if no condition element in PD1 matches the first element in STM, and if PD2 and PD3 do each have a condition element that matches the first element, then the behavior of PSG will depend upon which CONFLICT.1 setting was specified. If PD.ORDER was specified, then PD1 will be executed. If STM.ORDER was specified, then either PD2 or PD3 will be executed; which one is executed will depend on how the CONFLICT.2 and CONFLICT.3 options were set

> Each of the options is described in the body of this manual; the name of the option settings are always given with brackets (eg. [PD.ORDER]); after an option has been discussed, the format for setting the option is indicated and the default setting is given.

> See OPTIONS in Section IV

> USING PSG

> To initiate a run of PSG, give the Monitor the command R PSG. PSG will return a ">"

> The ">" is the PSG prompt character and indicates that PSG is awaiting a line of input from the user

> PSG will not attend to the line of input until you do a carriage return

> Interaction with PSG takes place by the user typing in a series of expressions and commands

> Expressions define productions, condition elements, action elements, memory elements, etc.

> All expressions are enclosed in parentheses to indicate their scope

> Any expression can be given a name by writing the name, followed by a colon, followed by the expression; the name can then be used in place of the expression

> Examples:

STM: (AA BB (CC DD))

PD1: (AA --> (EE FF))

> See NAMES AND NUMBERS in Section II

> After you have defined or redefined an expression, PSG acknowledges that it has stored it by coming back with its prompt character

> Commands tell PSG to do things; there are two different ways in which commands can be used

> The interpreter can be directed to execute a command immediately. To do this, the user types in the arguments and then the command followed by an exclamation point and a carriage return

> Examples:

PS1 START!

(PD1 PD6) PS!

> See CONTROL in Section II and CONTROL-POINTS in Section IV
> A command can be put in a command expression on the action side of a production; this command expression will be executed whenever the production is fired
> Examples:
PD2: ((AA) --> (NTC BB) (AA ===>))
PD3: (BB --> (PS (PD1 PD6)) (SAY BB) STOP)
> See ACTION in Section III

> You may create a file containing definitions of productions and various other expressions, and then load the file on top of PSG
> To read a file you have created, do <filnam>.<ext> RDF!; the response will be
##>
Each # indicates that a line has been read, and the ">" indicates that the file is loaded
> The file may contain commands which you want executed immediately; for example, if you include
S"FILE LOADED" PRSTRING!
as the last line in your file, then FILE LOADED will be typed out at your terminal when the file has been loaded
> To read someone else's file do <filnam>.<ext> <ppn> RDFPPN!
> Example:
SAMPLE.PSG A110PS00 RDFPPN!

> To quit PSG do ↑C if PSG is waiting for input; do ↑C↑C if PSG is running; in both cases you return to the Monitor
> If you then do CONT (or CONTINUE) you will be right back where you were
> If you do REE (or REENTER) the prior command will be terminated, but all the rest of your structures will be as they were before you did ↑C↑C

> To save a core image of your job, do <filnam> SAVE! (do not indicate an extension); the response will be
VXX.XX CONTINUING
>
To run from a saved core image do RUN <filnam>; the response will be
VXX.XX RESTARTING
>

> HELP
> The HELP command lists the current PSG documentation files and the current L* documentation files

> TEACHER
> TEACHER calls an interactive introductory SCRIPT which will

lead you through PSG; if you are going to want the teacher, do R PSGLRN instead of R PSG and then type TEACHER

> A SAMPLE PRODUCTION SYSTEM

```
00100   STM.ORDER S.CONFLICT.1!
00200   (CS/CTRL PR.CS PR.ACT) ON.ALL!
00300
00400   DIGIT: (CLASS 0 1 2 3 4 5 6 7 8 9)
00500   ANY: (VAR)
00600
00700   RESPOND: (ACTION (NTC (RESPONSE ANY)) (SAY ANY) (OLD **))
00800
00900   STMI: (READY (ELM 1) (ELM 6) (ELM 3) NIL NIL)
01000
01100   PM.ST1: (PD1 PD2 PD3 PD4)
01200   PD1: ((PROBE) (OLD (RESPONSE)) --> (OLD **))
01300   PD2: ((PROBE DIGIT) (ELM DIGIT) --> (RESPONSE YES) RESPOND)
01400   PD3: ((PROBE) (ELM) --> (RESPONSE NO) RESPOND)
01500   PD4: (READY --> ATTEND (NTC (ELM)))
01600
01700   S"PS.ST1 LOADED" PRSTRING!
```

> A SAMPLE RUN

```
>PS.ST1 START!
0.  STM: (READY (ELM 1) (ELM 6) (ELM 3) NIL NIL)
  TRUE: PD4: (READY --> ATTEND (NTC (ELM)))
  VARM: (NIL NIL $1 READY)
0.  ACTION- ATTEND
  ATTENDING - INPUT NEXT STIMULUS = >(PROBE 6) ↑Z
1.  ACTION- (PROBE 6)
1.  ACTION- (NTC (ELM))
2.  STM: ((ELM 1) (PROBE 6) READY (ELM 6) (ELM 3) NIL)
    CONFLICT.SET: (PD2 PD3 PD4)
    CONFLICT.SET: (PD2 PD3)
  TRUE: PD2: ((PROBE DIGIT) (ELM DIGIT) --> (RESPONSE YES) RESPOND)
  VARM: (NIL NIL $2 (ELM 6) $1 (PROBE 6) DIGIT 6)
2.  ACTION- (RESPONSE YES)
3.  ACTION- RESPOND
4.  ACTION- (NTC (RESPONSE ANY))
5.  ACTION- (SAY ANY)

********** YES

6.  ACTION- (OLD **)
7.  STM: ((OLD (RESPONSE YES)) (PROBE 6) (ELM 6) (ELM 1) READY
          (ELM 3))
```

```
    CONFLICT.SET: (PD1 PD2 PD3 PD4)
  TRUE: PD1: ((PROBE) (OLD (RESPONSE)) --> (OLD **))
  VARM: (NIL NIL $2 (OLD (RESPONSE YES)) $1 (PROBE 6))
7.  ACTION- (OLD **)
8.  STM: ((OLD (PROBE 6)) (OLD (RESPONSE YES)) (ELM 6) (ELM 1)
         READY (ELM 3))
  TRUE: PD4: (READY --> ATTEND (NTC (ELM)))
  VARM: (NIL NIL $1 READY)
8.  ACTION- ATTEND
 ATTENDING - INPUT NEXT STIMULUS = >(PROBE 0) ↑Z
9.  ACTION- (PROBE 0)
9.  ACTION- (NTC (ELM))
10.  STM: ((ELM 6) (PROBE 0) READY (OLD (PROBE 6))
          (OLD (RESPONSE YES)) (ELM 1))
    CONFLICT.SET: (PD1 PD3 PD4)
  TRUE: PD3: ((PROBE) (ELM) --> (RESPONSE NO) RESPOND)
  VARM: (NIL NIL $2 (ELM 6) $1 (PROBE 0))
10.  ACTION- (RESPONSE NO)
11.  ACTION- RESPOND
12.  ACTION- (NTC (RESPONSE ANY))
13.  ACTION- (SAY ANY)

********** NO

14.  ACTION- (OLD **)
15.  STM: ((OLD (RESPONSE NO)) (PROBE 0) (ELM 6) READY
          (OLD (PROBE 6)) (OLD (RESPONSE YES)))
    CONFLICT.SET: (PD1 PD3 PD4)
  TRUE: PD1: ((PROBE) (OLD (RESPONSE)) --> (OLD **))
  VARM: (NIL NIL $2 (OLD (RESPONSE NO)) $1 (PROBE 0))
15.  ACTION- (OLD **)
16.  STM: ((OLD (PROBE 0)) (OLD (RESPONSE NO)) (ELM 6) READY
          (OLD (PROBE 6)) (OLD (RESPONSE YES)))
  TRUE: PD4: (READY --> ATTEND (NTC (ELM)))
  VARM: (NIL NIL $1 READY)
16.  ACTION- ATTEND
  ATTENDING - INPUT NEXT STIMULUS = >STOP ↑Z
17.  ACTION- STOP
17.  ACTION- (NTC (ELM))
END: PS PS.ST1 STOPPED
```

II. THE ARCHITECTURE

> PSG is implemented in L✲(I) and is run on a PDP-10
  > See Section V

> NAMES AND NUMBERS
  > A name may be any string of characters except that it may not begin with a number, and non-printing characters (eg, space) may not be used in names
    > A string of characters becomes a name simply by using it
    > Many punctuation marks and special symbols are recognized even when adjacent to names. This prohibits their occurrence in names defined by mention since L✲ treats them as boundaries and breaks the name; for example, attempting to define AB(C to be a name runs afoul of the use of (as a boundary character
      > Boundary characters can occur in names if on first use the name is prefixed by " and terminated with a space; for example, "AB(C would define AB(C as a name; on subsequent occasions it would be recognized without the use of "
      > The marks used as boundary characters are
) (: ;] [- % ✲ " ' ! @ ↑ ← \
  > Any expression that has been given a name may be denamed or renamed
    > Format: X1 DENAME!
    > Format: X1 RENAME!

  > Digit and digit sequences may be used in one of two ways
    > They may designate numbers [NUMBERS]
      > This is the normal usage; the digit "6" denotes the number 6 and the digit sequence "606" denotes the number 606
    > They may be names [NAMES]
      > Then "6" and "606" are used in the same way as "A" and "ALPHA"
        > Example: 606: (AA BB)
      > If you want to write an actual number (in order, say, to have PSG do some computation), the digit or digit sequence must be prefixed by #
        > Examples: #2 #-666 #+18
        > This means that the character "#" cannot be used in a name
    > Setting the option
      > Example: NAMES S.DIGITS!
      > Default: NUMBERS

> THE DATA STRUCTURES
  > PRODUCTION MEMORY
    > A production memory is an ordered list of productions. Each production pairs a condition with an action. Thus a production memory is a set of contingent instructions

> A production memory may be viewed as a long term memory or as that part of a long term memory in which knowledge is represented procedurally
> A production is activated when a set of elements in working memory match the condition elements on the condition side of the production

> WORKING MEMORY
> The central part of working memory is short term memory (STM); it is a list of elements that are initially the internal representation of some specific situation. This memory is highly volatile; as productions become satisfied and are fired, this representation is modified until some overt behavior such as modifying the external environment or stating the solution to a problem becomes appropriate
> Working memory may also include a task environment (TE); this memory, which can be used to simulate an external task environment, is a list of elements, any of which can be read into STM; elements in STM can also be written into TE
> A third memory, long term memory (LTM), may be included as a part of working memory; this memory can be used to explore long term memory structures which are different from production memories

> CONTROL
> The basic unit of behavior is the recognition-act cycle. A cycle has two parts: 1) discovering the conflict set, the subset of true productions, and selecting one of the productions in the conflict set as the production to be fired, and then 2) firing (executing the action elements of) the production selected. Processing continues, cycle follows cycle, until either no production is true or the command STOP (an action element) stops the processing

> CONFLICT RESOLUTION
> Four different resolution procedures exist for settling conflicts when more than one production is satisfied at the beginning of a cycle
> The productions may be ordered according to their position in the production memory; then all productions except the first are deleted from the conflict set [PO.ORDER]
> The productions may be partially ordered by the working memory elements they match; a production with a supporting memory element that occurs earlier in working memory than those of another production takes precedence over it. Thus, those true productions with a condition element matched by the first supporting memory element become the conflict set [STM.ORDER]
> Example: Given the following expressions

STM: (AA (BB) (CC DD) EE)
PD1: ((CC DD) --> ...)
PD2: (EE (BB) --> ...)
PD2 would be selected to be fired

> The productions may be partially ordered by the relation of being a special case; a production which is a special case of another takes precedence over it. Thus, those true productions which have no true productions which are special cases of them become the conflict set [SPECIAL.CASE.ORDER]
  > PDX is a special case of PDY (during a particular cycle) if and only if each working memory element which supports a condition element in PDY also supports a condition element in PDX and PDX has at least one more condition element than PDY
  > Example: Given the following expressions
    STM: ((RR SS) TT ((UU) VV))
    PD1: (TT (RR SS) --> ...)
    PD2: ((RR) TT ((UU)) --> ...)
    PD2 would be selected to be fired
> A production may be selected at random from among the true productions; then all productions except that one are deleted from the conflict set [RANDOM]

> To provide the possibility of using a combination of resolution procedures and to insure that a single production is selected to be fired, three different procedures may be applied successively to the conflict set; the procedures are specifed by setting CONFLICT.1, CONFLICT.2, and CONFLICT.3. Both CONFLICT.1 and CONFLICT.2 may be set to any of the four settings settings above; CONFLICT.3 may be set to either PD.ORDER or RANDOM
  > As soon as the conflict set contains only a single production, that production is fired
> Setting the option
  > Example: STM.ORDER S.CONFLICT.1! SPECIAL.CASE.ORDER S.CONFLICT.2! PD.ORDER S.CONFLICT.3!
  > Example: PD.ORDER S.CONFLICT.1!
  > Example: STM.ORDER S.CONFLICT.1! RANDOM S.CONFLICT.2!
  > Default: The default setting for all three options is PD.ORDER
  > Note that the order of specification is relevant; for example, setting CONFLICT.1 to STM.ORDER and CONFLICT.2 to SPECIAL.CASE.ORDER is not the same as setting CONFLICT.1 to SPECIAL.CASE.ORDER and CONFLICT.2 to STM.ORDER

> COMMANDS
  > The following commands may be used to immediately execute action elements, productions, or production systems
    > DO
    > PD

> PD.TE
> PS
> PS.1
> PS.TE
> PS.1.TE
> START

> For a description of these commands, see Section VII

> EFFICIENCY

> In order to decrease the number of productions which have to be tested at the beginning of each cycle, PSG has been implemented with a partial filter which keeps track of the elements entering and leaving working memory and which is therefore able to exclude some productions from consideration. The filter can be set to be more or less discriminating; the most discriminating setting, of course, has the highest overhead.

> Use setting 1, the least discriminating setting, unless a large proportion of the condition expressions in the production memory have the same initial subelement [FILTER1]

> Use setting 2 if a large proportion of condition expressions have the same initial subelement and only a small proportion of condition expressions contain subexpressions [FILTER2]

> Use setting 3 if a large proportion of condition expressions have the same initial subelement and a significant proportion of condition expressions contain subexpressions [FILTER3]

> Setting the option

> Example: FILTER2 S.FILTER!

> Default: FILTER1

III. MEMORIES

> STM: SHORT TERM MEMORY
  > STM STRUCTURE
    > STM is a list of expressions or elements
      > This list is considered to be ordered, so that it is meaningful to give specifications in terms of front, end, etc.
    > STM has an initial size, determined by the number of expressions it contains
      > The size may be fixed [FIXED]
        > That is, it will continue to be whatever size is set initially
          > See INITIALIZATION in Section IV
      > The size may expand indefinitely [EXPAND]
        > That is, with each addition to STM it grows by one
      > The size may be determined by the interplay between processes of growth and decay [DYNAMIC]
      > Setting the option
        > Example: EXPAND S.STM.SIZE!
        > Default: FIXED

  > STM MODIFICATION
    > New expressions are added to STM relative to some location
      > The location must be specified
        > It may be at the front [FRONT]
        > It may be at the end [END]
        > It may be determined randomly [RANDOM]
        > It may be given by the user at the terminal [USER]
        > Setting the option
          > Example: END S.STM.ADD.LOC!
          > Default: FRONT
      > How expressions are to be added relative to the location must also be specified
        > They may be inserted before the element in the location [INSERT.BEFORE]
        > They may be inserted after the element in the location [INSERT.AFTER]
        > They may replace the element in the location [REPLACE]
        > Setting the option
          > Example: REPLACE S.STM.ADD.HOW!
          > Default: INSERT.BEFORE
    > Expressions are deleted from a location in STM
      > It may be the front [FRONT]
      > It may be the end [END]
      > It may be determined randomly [RANDOM]
      > It may be given by the user at the terminal [USER]
      > Setting the option

> Example: USER S.STM.DEL.LOC!
> Default: FRONT

> STM EXPRESSIONS
> An STM expression is a symbol; it may or may not have an associated name
> Example:
STM: (AA (BB CC (DD)) LBL)
> NIL is the null element
> Though it holds a location in STM, NIL may not be matched against; that is, it cannot be part of a template expression

> AUTOMATIC REHEARSAL
> Automatic rehearsal moves (reorders) the expressions in STM after a production has been selected to be fired, but before the first action element is executed
> There are two options
> All of the expressions in STM recognized by the evoked production may be rehearsed [RECOGNITION]
> There may be no automatic rehearsal [NO]
> Setting the option
> Example: NO S.REHEARSE!
> Default: RECOGNITION
> If automatic rehearsal is specified, then how the rehearsal is to be realized must also be specified
> The expressions may be moved to the front [MOVE.FRONT]
> Copies may be moved to the front [COPY.FRONT]
> The expressions may be moved to the end [MOVE.END]
> Copies may be moved to the end [COPY.END]
> The expressions may be treated as if they were new elements being added to STM [MOVE.NEW]
> Copies may be treated as if they were new elements being added to STM [COPY.NEW]
> Setting the option
> Example: COPY.FRONT S.REHEARSE.HOW!
> Default: MOVE.FRONT
> Expressions in STM can be deliberately rehearsed while the action side of a production is being executed by using NTC in a command expression
> See ACTIONS in this Section

> PRODUCTION MEMORY
> A production memory is a set of productions. Each production consists of a condition paired with an action; productions may be named
> Format:
<name>: (<condition> --> <action>)

> CONDITION
  > A condition consists of an ordered sequence of condition expressions or elements (CEs) each of which must match an element in STM in order for the condition to be true
    > Format:
      (CE1 CE2 CE3 ... --> ...)
    > Example:
      ((ELM AA) (ELM) BB --> ...)

  > CONDITION EXPRESSIONS
    > Except for connectives, all condition elements are template expressions. A template expression is a form which may include variables
      > Variable elements
        > Types
          > Free variables
            > If X is to be a free variable then it is declared by X: (VAR); X will match any expression
              > Example:
                X4: (VAR)

          > Variables with specified domains
            > If X is to be the variable and D is to be the domain, then X is declared by X: (VAR D); X will match any expression that matches D
              > Example:
                X5: (VAR (ELM (A1 A2 A3)))
            > Local variables
              > If in a template expression X == ELM occurs in place of a component, then X is being declared as a variable with domain ELM within the scope of the production; for example, if X: (VAR) then (AA X == (BB) CC) matches (AA (BB) CC), but (X == BB X) does not match (BB CC)
              > This is simply a naming convenience, so that the same names can be used over again; it implies the same processing capabilities as varibles with domains

          > Class Variables
            > If X is to be a class variable and C1, C2, ... the members of the class, then it is declared by X: (CLASS C1, C2 ...); a class variable can contain other class variables
            > Example:
              X6: (CLASS AA BB X2)
        > Scope

> The variables in the production fired retain their bindings only for the remainder of the cycle
> Variables may be set on the action side of a production as well as on the condition side
> See ACTION in this section
> A class variable matches any expression that matches one of the members of the class
> It may be bound to one value and have that value for the remainder of the cycle [VAR]
> In this case, class variables have a disjunctive domain
> It may not be bound at all in which case a single variable can match different elements in the class within a single production [TEST]
> Setting the option
> Example: TEST S.CLASS!
> Default: VAR
> VARM is the memory that holds the current variable bindings; it has the form
VARM: (Xn <value n> ... NIL NIL ... X1 <value 1>)
> The NIL NIL separates the bindings that have been established (for X1 ...) from those that are still tentative

> Connectives
> The connective ABS is used to stipulate that there must be no element in STM which matches the CE preceding it
> Example:
(AA BB ABS (ELM (AA)) ABS (ELM (CC)) --> ...)
> (BB BB ABS --> ...) stipulates that there must be only one element in STM which matches BB
> The connective OR can be used to collapse several productions with the same action elements into one production; for example
(AA BB OR CC OR DD EE ABS --> FF)
is equivalent to
(AA BB --> FF) (CC --> FF) (DD EE ABS --> FF)
> The connective AND was used in place of juxtaposition in earlier versions; it is still permitted, but is unnecessary

> RULES FOR MATCHING
> Distinctness Rule:
> Every template expression on the condition side of a production must match a working memory element
> Each working memory element must be distinct [YES]
> A working memory element may be matched by more than one condition element [NO]
>Setting the option

>Example: NO S.DISTINCT!
>Default: YES

> Order Rule:
  > The template expressions on the condition side of productions are considered in order; a match is attempted with each working memory expression, considered in order, until a match is found; the first match sticks
> Comparison Rule:
  > A working memory expression and a template expression match only if they are the same symbol or the condition element is a matching variable or the subelements match recursively as expressions
  > Variable match rule:
    > A variable bound to an working memory element matches another working memory element if and only if the two working memory elements are the same symbol
  > Penetration Rule:
    > Ordinarily if a symbol has a name, it will not match another symbol which does not have that name even if the two symbols have identical structures; in other words, named symbols are closed or impenetrable. A named symbol can, however, be opened so that PSG's match procedure will treat it as if it were not named
      > See OPEN and CLOSE in Section VII
    > Two terminal symbols with different names do not match even if both symbols have been opened; for example, AA: () and BB: () do not match recursively
> Expression Rule:
  > A template expression and a working memory expression match only if the subelements of the template expression and the subelements of the working memory expression are in the same order; for example, (AA BB) does not match (BB AA)
    > The subelements must all match [ALL]
    > The subelements of the template expression must match an initial consecutive sequence of subelements of the working memory expression [FRONT]
    > Each subelement of the template expression must match a subelement of the working memory expression and the first subelement of the template expression must match the first subelement of the working memory expression [SEQ.ANCHOR]
    > Each subelement of the template expression must match a subelement of the working memory expression [SEQ.ANY]
    > Setting the option
      > Example: ALL S.MATCH.EXPR!
      > Default: SEQ.ANCHOR

> ACTION
  > An action consists of an ordered sequence of action expressions or elements (AEs)
    > The action expressions are interpreted in order
    > The effects of automatic rehearsal on STM take place before any action expressions are interpreted
    > Format:
      (... --> AE1 AE2 ...)
    > Example:
      (... --> (NTC CC) STOP)

  > ACTION EXPRESSIONS
    > All action elements are either template expressions or command expressions
    > When a template expression is encountered in the action side of a production being fired, it is added to STM
      > If the template expression is or contains a variable, then the variable is replaced by its value before the new expression is added to STM; for example, if
        STM: ((AA BB) (EE FF) BB NIL NIL NIL)
        X1: (VAR) X2: (VAR)
        PD24: ((AA X1) X1 X2 --> (X1 CC X2) X1)
        then when PD24 is fired, X1 will be bound to BB and X2 will be bound to (EE FF); two new expressions will be added to STM; it will become
        STM: (BB (BB CC (EE FF)) (AA BB) (EE FF) BB NIL)
      > If the template expression is unnamed, then it is copied before it is added to STM; if it is named, then it will not be copied unless it has been opened
        > See OPEN and CLOSE in Section VII
    > When a command expression is encountered in the action side of a production, the command is executed; the remaining subelements in the expression (if there are any) are the arguments for the command; some commands take template expressions as their arguments
      > The following commands may be used on the action side of productions
        > ↤ (ASSIGN NUMBER)
        > == (ASSIGN VARIABLE)
        > $i (DOLLAR i for i from 1 to 10)
        > ☆☆ (ENCODE)
        > && (FLATTEN)
        > ==> (REPLACE)
        > ===> (REPLACE)
        > ====> (REPLACE)
        > ACCEPT/PD
        > ACTION
        > ATTEND

> BEGIN
> COPY
> D$i (DELETE $i for i from 1 to 10)
> DEMO
> EMBED
> FAIL
> HOLD
> IGNORE
> NTC
> OFF
> OFF.ALL
> ON
> ON.ALL
> PD
> PR
> PR$i (PRINT $i for i from 1 to 10)
> PRNAME
> PRPSPD
> PRSTRING
> PRVL
> PS
> PS.1
> RELEASE
> RPL
> SAY
> STOP
> STM+1
> STM-1
> WHERE
> WHIZ

> For a description of these commands, see Section VII

> TE: TASK ENVIRONMENT

> TE, like STM, is an ordered list of expressions or elements; it is indefinite in size. It uses the internal encoding which STM uses so that it bypasses the need for perceptual mechanisms to encode the external environment into internal form

> TE AND STM

> TE is used in conjunction with STM to create a more complete simulation. The expressions in TE must be accessed explicitly; expressions may be read from TE into STM; they may also be written from STM into TE

> The following commands may be used in command expressions to operate on TE

> =TE=> (REPLACE)
> DTE
> ITE

> LOCTE
> NTCTE
> For a description of these commands, see Section VII
> The read operations, LOCTE and NTCTE, involve matching a template expression in a command expression with an expression in TE; a command expression whose command is NTCTE may be a condition element as well as an action element. The matching rules are given above (see CONDITION); however, the distinctness rule need not be the same for TE and STM
> Every template expression on the condition side of a production which is part of a NTCTE command expression must match a TE element
> Each TE element must be distinct [YES]
> A TE element may be matched by more than one template expression [NO]
> Setting the option
> Example: YES S.TEDISTINCT!
> Default: NO

> OPERATING ON TE IN ISOLATION
> TEWM rather than STM may be used as the central part of working memory when a simulation of the external environment is desired
> Though this memory is structurally identical to STM, it need not have the same size
> The size may be fixed [FIXED]
> The size may expand indefinitely [EXPAND]
> Setting the option
> Example: FIXED S.TEWM.SIZE!
> Default: EXPAND
> Commands
> The following commands may be used to immediately execute productions on TEWM; they may also be used in command expressions
> PD.TE
> PS.TE
> PS.1.TE
> For a description of these commands, see Section VII

> LTM: LONG TERM MEMORY
> The following version of long term memory is just one of many possible memory structures; it has been included because of its simplicity, not because it has any psychological virtues

> LTM is a simple deliberate content-addressed element long term memory; like STM and TE, it is an ordered list of expressions; it is indefinite in size

> LTM AND STM

> LTM, like TE, is used in conjunction with STM. The elements in LTM must be accessed explicitly
  > The following commands may be used in command expressions to operate on LTM
    > DLTM
    > ILTM
    > NTCLTM
  > For a description of these commands, see Section VII

IV. OPERATING FACILITIES

> INITIALIZATION
  > Several structures need to be initialized to values specified by the user. This section describes each of these structures and the means available for initialization

  > WORKING MEMORY
    > STM, TE, TEWM, and LTM can all be initialized
      > INIT.STM creates a new version of STM which is a copy of STMI
      > INIT.TE creates a new version of TE which is a copy of TEI
      > INIT.TEWM creates a new version of TEWM which is a copy of TEWMI
      > INIT.LTM creates a new version of LTM which is a copy of LTMI

  > TIME and COUNTS
    > INIT.TC sets TIME to 0 and all counts to 0
      > See CONTROL-POINTS in this Section

  > The command RESET calls INIT.STM, INIT.TE, INIT.TEWM, INIT.LTM and INIT.TC
    > See RESET in Section VII

> CONTROL-POINTS
  > Control and monitoring of the running system is achieved through a set of CONTROL-POINTS embedded at stategic places in the program
    > Every time the program passes through that point it is prepared to print characteristic messages, to keep statistics or to engage in arbitrary monitoring and controlling behavior

  > For each control-point two quantities and five processes are defined
    > COUNT.<ctrl> contains the count of the times the system has passed through this control-point since it was last reset to 0
    > TIME.<ctrl> is the increment of time that is to be accumulated to TIME for passing through this point
      > The units are arbitrary
      > See ← (ASSIGN NUMBER) in Section VII
    > <ctrl>/CTRL is the control-point set-up process; in essence it names the control-point
      > Turning a set-up process off (OFF!) disables all the automatic processing that occurs at that control-point
      > Turning a set-up process on (ON!) enables the other processes to be turned on and off individually
      > PSG will run more efficiently if all <ctrl>/CTRL processes not in use are turned off
    > PR.<ctrl> prints the message associated with the control point <ctrl>

```
      > This happens automatically on each passage
      > The message can be rewritten by the user to suit his needs
   > <ctrl>/C is the control program for <ctrl>
      > This is an arbitrary program that gets executed on each
        passage
      > It can engage in any sort of controlling or monitoring
        behavior
      > The normal behavior for <ctrl>/C is to come to the terminal
        and give control to the user so he can do anything he
        desires.  To return control to PSG, the user must type
        control Z (↑Z)
   > ADVC.<ctrl> adds 1 to COUNT.<ctrl>
      > This happens automatically on each passage
      > PRCOUNTS prints all of the counts that are on
   > ADVT.<ctrl> adds TIME.<ctrl> to TIME
      > This happens automatically on each passage
      > PRTIME prints TIME

> The CONTROL-POINTS are:
     ACT/CTRL      PR.ACT      ACT/C      ADVC.ACT      ADVT.ACT
     CALL/CTRL     PR.CALL     CALL/C     ADVC.CALL     ADVT.CALL
     CE/CTRL       PR.CE       CE/C       ADVC.CE       ADVT.CE
     CS/CTRL       PR.CS       CS/C       ADVC.CS       ADVT.CS
     LTM/CTRL      PR.LTM      LTM/C      ADVC.LTM      ADVT.LTM
     MCH/CTRL      PR.MCH      MCH/C      ADVC.MCH      ADVT.MCH
     PD/CTRL       PR.PD       PD/C       ADVC.PD       ADVT.PD
     PS/CTRL       PR.PS       PS/C       ADVC.PS       ADVT.PS
     START/CTRL    PR.START    START/C    ADVC.START    ADVT.START
     TE/CTRL       PR.TE       TE/C       ADVC.TE       ADVT.TE
     VARSET/CTRL   PR.VARSET   VARSET/C   ADVC.VARSET   ADVT.VARSET

> Control is maintained over all these processes by turning them on
  or off
   > X1 ON! turns X1 on
   > X1 OFF! turns X1 off
   > (X1 X2 ...) ON.ALL! turns X1, X2, ... all on
   > (X1 X2 ...) OFF.ALL! turns X1, X2, ... all off

> Several lists and general processes are available to help the user
  select the CONTROL-POINTS he wants
   > LIST/CTRL is the list of all <ctrl>/CTRL points
   > LIST/C is the list of all control programs
   > LIST.COUNTS is the list of all COUNT.<ctrl> points
   > LIST.TIMES is the list of all TIME.<ctrl> points
   > LIST.ADVC is the list of all ADVC.<ctrl> points
   > LIST.ADVT is the list of all ADVT.<ctrl> points
   > LIST.PR is the list of all PR.<ctrl> points
   > DEMO.LIST is a list of a full set of things to turn on to give
```

a detailed demonstration
- > WHIZ.LIST is a minimal list of things to turn on for standard operation
- > Any of these lists can be printed to see their actual contents
- > ?CTRL.STATE prints the names of all of the control-points that are on

> control-point actions
- > ACT/C is executed just before each action expression is executed
- > CALL/C is executed whenever a command calling for terminal input is executed
- > CE/C is executed just before a condition expression is matched with the expressions in the central part of working memory
- > CS/C is executed whenever there is at least one production in the conflict set; CS/C takes a subset of the set of true productions (followed by ↑Z) as input; it then replaces the current conflict set with this subset
- > LTM/C is executed just before a template expression is matched with the expressions in LTM
- > MCH/C is executed just before a match is attempted between a template expression and an expression in the central part of working memory
- > PD/C is executed just before the production selected to be fired is fired; PD/C takes a true production (followed by ↑Z) as input; this production, rather than the production selected by the conflict resolution procedures, is then fired
- > PS/C is executed just before a production system is processed
- > START/C is executed when a run is about to start
- > TE/C is executed just before a template expression is matched with the expressions in TE
- > VARSET/C is executed whenever a variable is about to be assigned a value

> INTERACTIVE OPERATION
- > PSG is intended to be used in an interactive mode; this intent should be apparent from the nature of the command language
  - > The interactive printing, definition, editing, etc
  - > The use of CONTROL-POINTS to permit intervention and exploration while running

> Interaction is a means of simulating the task environment
- > Instead of creating programs to provide for the TE, the production system can come to the terminal when it requires input from the environment

> Interaction is a means of simulating aspects of the subject model
- > Instead of creating a complete set of productions, the production system can come to the terminal when it needs to

evoke unmodeled actions such as perceptual operations, basic operations such as addition and multiplication, or LTM acquisition processes

> ATTEND is the mechanism for interactive operation
  > ATTEND evokes CALL/CTRL which does the following things:
    > It comes to the terminal
    > It prints PR.CALL, which can be modified by the user to create the display he wants
    > It gives control to the user who can then do anything he wants
    > It takes back control when user does ↑Z (Control-Z) and accepts all of the action elements input by the user as an action sequence to be executed
      > It executes the actions in the order in which they were input

> AUTOMATIC OPERATION
  > Interactive computing is good while large amounts of uncertainty exist about the operation, but it does require a fallible human to attend continuously

  > A facility exists for replacing the user at the terminal with a fixed sequence of inputs
    > Define the sequence of inputs to be the list STIMULUS
      > Example: STIMULUS: (READY (ELM 1) (ELM 6) (PROBE 6))
    > Turn on AUTO.ATTEND
      > Format: AUTO.ATTEND ON!
    > Then use AUTO.ATTEND in place of ATTEND (or do ATTEND: (OPR AUTO.ATTEND); AUTO.ATTEND will take the next stimulus from STIMULUS; when STIMULUS is empty, the processing of the production system will stop
    > To stop the automatic responding, turn off AUTO.ATTEND
      > Format: AUTO.ATTEND OFF!

> FILES AND RECORDING
  > The standard practice is to keep disk files which contain production systems; these files can be read in when wanted
    > See RDF and RDFPPN in Section VII
    > The external disk files can be edited with external editors such as SOS or TECO
      > PSG does not read in the line numbers used in SOS, so you need not worry about them

  > It is possible to make a complete recording of all that happens at a terminal
    > See RECORD and RECORD. in Section VII

> ACCOUNTING
  > PSG has three types of accounting numbers it can generate
    > It can count the number of times the system has passed through specified CONTROL-POINTS
    > It can calculate pseudo time (called TIME)
      > This is a weighted sum of fixed contributions from each of the CONTROL-POINTS; the weights can be set by setting TIME.<ctrl> for each control-point
        > Example: (--> (TIME.CND+ ← 100))
      > The weights are integers
    > It can calculate real time (Pc cycles)
      > See TIME.PROCESS and UNTIME.PROCESS in Section VII

> EDITING
  > There is an interactive editor for modifying any of the structures in PSG; it is the L☆ editor

  > The editor is evoked on list structure X1 by doing X1 EDIT!
    > All of the commands within the editor are active; that is, they do not have to be executed by !; they all start with "!" to indicate this
      > EDIT, the process that initiates an edit, is not active; it is not within the editor, but outside it

  > The editor always locates you somewhere in the list structure
    > It prints out a location; this is the name of a symbol if it has one; otherwise it is the address (eg, 36166%)
    > With every edit command you give PSG, it does something (gets to a new place, makes a modification such as replace) and then puts you back in control
    > The commands of the editor allow you to move around in the structure and to make modifications; you remain in the structure after a modification so that you can see the modification, correct it, or go on to another one

  > The edit commands
    > <LF> goes to the next symbol at the current level
    > <ALT> goes backwards to the immediately preceding location
    > !S goes down a level in the structure
      > Note that !S (and all of the other commands below) are active only after you do a carriage return (so that the machine can regain control)
    > ?S prints the current item
    > !U goes up a level in the structure
    > !F searches the current level (and up) for the first occurrence of the symbol X1
      > Format: X1 !F
    > !FX searches all unnamed levels for the first occurrence of the

symbol X1
> Format: X1 !FX
> !R replaces the current item with X1
> Format: X1 !R
> !I inserts X1 in front of the current item
> Format: X1 !I
> !IN inserts X1 after the current item
> Format: X1 !IN
> !D deletes the current item
> Format: !D
> !. exits from the editor
> Example:
PDX: ((ELM X1 X1) (ELM (X2)) (X2) --> (NTC $3) (OLD **) (SAY $4))
then to insert EMBED after (OLD **) and replace $4 with (X1 X2), do
PDX EDIT!
(OLD **) !F
EMBED !IN
$4 !FX
(X1 X2) !R !.

> OPTIONS
> S.<option> is the name of the process that represents <option>
> Example: S.CONFLICT.1 represents CONFLICT.1

> <option-setting> S.<option>! will set the option to the option setting specified by the user

> (<option option-setting> ...) S.SETUP! takes a list as input and sets the options on that list to the option settings specified; options which are not input to S.SETUP will be assigned their default option settings
> Example: (S.STM.SIZE FIXED S.CONFLICT STM.ORDER) S.SETUP!

> S.<option> ?STATE will print the current option setting for the option
> Example: S.CONFLICT.2 ?STATE

> S.<option> ?OPTION will print the settings possible for the option
> Example: S.STM.SIZE ?OPTIONS

> ?OPTION.STATE will print a list of all the option states with their current settings

> The option states and their default settings are:
STM.ADD.LOC FRONT
STM.ADD.HOW INSERT.BEFORE
STM.DEL.LOC END

STM.SIZE	FIXED
TEWM.SIZE	EXPAND
DISTINCT	YES
TEDISTINCT	NO
MATCH.EXPR	SEQ.ANCHOR
VAR.ACTION	YES
REHEARSE	RECOGNITION
REHEARSE.HOW	MOVE.FRONT
CONFLICT.1.	PD.ORDER
CONFLICT.2	PD.ORDER
CONFLICT.3	PD.ORDER
CLASS	VAR
DIGITS	NUMBERS

V. L*(I)

> TEACHER
  > Do R LSISCP and then type TEACHER for an interactive script describing L*(I)

  > The following files are available on disk [A110LI00] CMU-10A
    > SCPTXT.LSI is the text of the SCRIPT
    > LSI.DOC is an introduction to L* and a guide to the M-file
    > LSIM18 is the M-file (common source)
    > FINDEX.LSI is a facility index to the M-file
    > AINDEX.LSI is an alphabetic index to the M-file
    > INDEX.LSI is an index to the source files

> Although PSG is implemented in L*(I), when working normally in PSG the user is in a protected name context such that no names defined within L* are accessible; thus it is safe to use any name not otherwise used in this manual
  > The name of the protected name context is PSGU

  > The command UNPROTECT moves into a name context that includes PSGU and all inner contexts; PROTECT moves back into PSGU context

  > The command FIXIT moves into a name context that excludes PSGU, but includes all inner contexts; FIXIT. moves back to whatever context was on top of the context stack when the command FIXIT was given

> PSG is designed to be used by individuals with no knowledge of L*. However, the user who knows L* can extend PSG so that it meets his particular needs
  > Specifically, the user who knows L* can define additional commands to be used in command expressions on the action side of productions
    > To create a command which takes no arguments, make OPR the first symbol in the L* program
      > Example:
        INCR: (OPR W\NUM 1 +W U)
    > To create a command which can be used in a command expression of the form (<command> X1 X2 ... Xn), define the command and then do <command> SETACXN!
      > The one argument to the command will be a list (X1 X2 ... Xn)
      > Example:
        DEL: (S D) DEL SETACXN!

> The version of PSG implemented in L*(I) is upward compatible with the version implemented in L*(H) with the following exceptions

> The option setting SEQ.ANY for the option MATCH.EXPR is not available in the current L*(I) version

> The options CONFLICT and CONFLICT.SELECT have been replaced by CONFLICT.1, CONFLICT.2, and CONFLICT.3

> Neither the option OPAQUE nor the option ELM.ACT is available in the current version; but see OPEN and CLOSE in Section VII

> The control-points CND+/CTRL, CND-/CTRL, and PD+/CTRL have been eliminated from the current version

> The commands ACCEPT/PD and IGNORE have been eliminated from the current version

VI. REFERENCES

> Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In Melton, A. W. and Martin, E. (eds.). Coding Processes in Human Memory. Winston and Sons, Washington, D.C., 1972, 373-434.
  > This paper explores a particular task of stimulus encoding to show how production systems handle it. It uses PSF, an earlier version of PSG

> Newell, A. Production systems: models of control structures. In Chase, W. C. (ed.). Visual Information Processing. Academic Press, New York, N. Y., 1973, 463-526.
  > This is a basic introductory paper on productions systems as models of the human immediate processor; it contains detailed treatment of the Sternberg paradigm. It uses PSG

> Newell, A. and Simon, H. A. Human Problem Solving, Prentice-Hall, 1972.
  > This book contains an extensive general treatment of production systems in the context of problem solving; it includes detailed treatment of cryptarithmetic task, and much general background on information processing theories. It does not use PSG

> For a more complete bibliography, see PS.BIB[A110PS00] CMU-10A

VII. AVAILABLE COMANDS

> This section is a reference section for the available PSG commands; it has two parts: in the first part, the available commands are grouped according to their function; in the second part, all of the commands are listed alphabetically and each command is described. The descriptions include an statement of the format or formats of each command; if the command is followed by an exclamation point, then it is an immediately executable command; if it is enclosed in parentheses, then it may be used in a command expression; X1, X2, ... Xn are used to indicate the number of arguments each command takes and where they are to be placed in relation to the command

> LISTING BY FUNCTION

- > Control
  - > ACTION
  - > ATTEND
  - > BEGIN
  - > CLOSE
  - > DO
  - > FAIL
  - > OPEN
  - > PD
  - > PS
  - > PS.1
  - > S.<option>
  - > START
  - > STOP

- > Printing
  - > PR
  - > PR$i (PRINT DOLLAR VARIABLE)
  - > PRNAME
  - > PRPSPD
  - > PRSTRING
  - > PRVL
  - > SAY

- > STM processes
  - > ←← (ASSIGN NUMBER)
  - > == (ASSIGN VARIABLE)
  - > $i (DOLLAR VARIABLE)
  - > ☆☆ (ENCODE)
  - > && (FLATTEN)
  - > ==> (REPLACE)
  - > ===> (REPLACE)
  - > ====> (REPLACE)
  - > COPY

> D$i (DELETE DOLLAR VARIABLE)
> EMBED
> NTC
> RPL

> TE processes
> =TE=> (REPLACE)
> DTE
> ITE
> LOCTE
> NTCTE
> PD.TE
> PS.TE
> PS.1.TE

> LTM processes
> DLTM
> ILTM
> NTCbTM

> Initialization
> INIT.ACT
> INIT.LTM
> INIT.STM
> INIT.TE
> INIT.TEWM
> RESET
> STM+1
> STM-1
> STMI+1
> STMI-1

> Control-point Processes
> <ctrl>/C
> ADVC.<ctrl>
> ADVT.<ctrl>
> DEMO
> OFF
> OFF.ALL
> ON
> ON.ALL
> PR.<ctrl>
> WHERE
> WHIZ

> Accounting
> HOLD
> RELEASE

- > TIME.PROCESS
- > UNTIME.PROCESS

> Editing
- > EDIT

> File operations
- > RDF
- > RDFPPN
- > RECORD
- > RECORD.
- > SAVE
- > USRSAV

> L* access
- > PROTECT
- > UNPROTECT

> ALPHABETICAL LISTING

> ←← (ASSIGN NUMBER) assigns a number to a numerical variable

> Numerical variables must be declared to be of type word; this is done by prefixing the characters "\W" to the variable name the first time it is used

> Format:

(... --> (X1 ←← X2))

> Examples:

(←--> (TIME ←← 0))
(--> (W\NUM ←← 100))
(--> (NUM ←← 200))

> == (ASSIGN VARIABLE) assigns an expression to be the value of a variable

> The variable has to be declared a variable for this operation

> Format:

(... --> (X1 == X2))

> Examples:

(--> (X == (AA BB)))
(--> (X == BB))

> <ctrl>/C is a subroutine which is executed whenever the program passes through its parent control-point, <ctrl>/CTRL, provided that it has been turned on and its parent has been turned on

> Whenever the program passes through the parent control point, control is passed to the user who after inputting whatever he wishes can return control to PSG by typing ↑Z

> Any control-point action may be redefined so that instead of passing control to the user, the control point action executes a command expression

> Format:

<ctrl>/CTRL ON! <ctrl>/C ON!
<ctrl>/CTRL OFF!
<ctrl>/CTRL ON! <ctrl>/C OFF!

> Examples:

CS/CTRL ON! CS/C ON!

> See CONTROL-POINTS in Section IV

> $i (DOLLAR i for i from 1 to 9) is a free variable whose value is the expression in STM which matched the ith condition element

> Format:

(CE1 CE2 --> $1 $2)

> Example:

(ELM (DD)) --> (COPY $1) $2)

> ** (ENCODE) stands for the first STM element and is used to replace that element with an expression which contains one or more occurrences of it
  > Format:
    (CE1 --> (**))
  > Example:
    ((AA BB) --> (OLD ** (AA **)))

> && (FLATTEN) removes the outer level of parentheses from an expression
  > Format:
    (... --> (&& ((X1 (&& (X2)))))
  > Example: (AA && (BB CC)) produces (AA BB CC)

> ==> (REPLACE) replaces X1 with X2 if the first element in STM is an expression containing X1; otherwise it does nothing and processing continues with the next action element
  > Format:
    ((X1) --> (X1 ==> X2))
  > Example: If STM: ((DD) (ELM AA BB) CC and PD18: ((ELM) --> (AA ==> CC (FF))), then after PD18 is fired, STM: ((ELM CC (FF) BB) CC (DD))
  > This command will only replace elements in an expression; it will not replace subelements
    > If the expression (ELM (AA BB)) is the first element in STM, the command ((AA) ==> (CC AA) will change the expression to (ELM (CC AA)). To change (ELM (AA BB)) to (ELM (CC AA BB)), fire a production of the form ((ELM X) --> (X ==> (CC && X)))

> ===> (REPLACE) replaces X1 with X2 if the second element in STM is an expression containing X1
  > Otherwise it is like ==>

> ====> (REPLACE) replaces X1 with X2 if the third element in STM is an expression containing X1
  > Otherwise it is like ==>

> =TE=> (REPLACE) replaces the X1 located by the pointer TEL with X2; it requires NTCTE or LOCTE to establish the pointer
  > Format:
    (... --> (NTCTE X1) (X1 =TE=> X2))
  > Example:
    (--> (LOCTE (ELM AA)) (AA =TE=> BB))

> ACTION interprets the rest of the expression as a list of action elements

> Format:
(... --> (ACTION X1 X2 ... Xn))
> Example:
(--> (ACTION (SAY LOOK) (NTC (ELM X X)))

> ADVC.<ctrl> advances the count associated with the specified control-point
> Format:
ADVC.<ctrl> ON!
ADVC.<ctrl> OFF!
> Example:
ADVC.MCH ON!
> See CONTROL-POINTS in Section IV

> ADVT.<ctrl> advances TIME by the amount associated with the specified control-point
> Format:
ADVT.<ctrl> ON!
ADVT.<ctrl> OFF!
> Example:
ADVT.PD ON!
> See CONTROL-POINTS in Section IV

> ATTEND passes control to the user who after inputting whatever he wishes can return control to PSG by typing ↑Z
> Format:
(... --> ATTEND)
> Example:
(--> (SAY S"INPUT (ELM AA)") ATTEND)

> BEGIN begins a new recognition-act cycle
> Format:
BEGIN ↑Z
> Example: Assume ACT/CTRL and ACT/C are on, then when control passes to the user (ELM BB) (ELM CC) BEGIN ↑Z will cause PSG to begin a new cycle

> CLOSE is used to restore the impenetrability of an expression; see OPEN
> Format:
X1 CLOSE!
> Example:
AA CLOSE!

> COPY is used to add to STM an unnamed expression which is a copy of some named or unnamed template expression
> COPY assigns a new dollar variable, $i, where $i is the first unassigned dollar variable between $1 and $9

> Format:
(... --> (COPY X1))
> Example: IF SUE: (AA BB), then (COPY SUE) puts a new unnamed expression (AA BB) into STM

> D$i (DELETE $i for i from 1 to 9) deletes from STM the expression which matched the ith condition element
> Format:
(CE1 CE2 CE3 --> D$1 D$3)
> Example:
(AA BB --> (NTC CC) D$3)

> DEMO turns on the standard demonstration control-point actions given on DEMO.LIST
> Initially DEMO.LIST contains many PR.<ctrl>s as well as CS/C which calls to the terminal whenever there is more than one production in the conflict set so that the user can select the production to be fired
> DEMO.LIST can be modified by the user
> Format:
DEMO!
> Example:
DEMO! MCH/CTRL OFF!
> See CONTROL-POINTS in section IV

> DLTM deletes an expression from LTM
> Only the first instance of the expression is deleted
> Formats:
(... --> (DLTM X1))
X1 DLTM!
> Examples:
(--> (DLTM MOTHER))
MOTHER DLTM!

> DO executes an action expression (as if it were an action expression in the production being fired)
> Format:
X1 DO!
> Example:
(ELM AA BB) DO!

> DTE deletes an expression from TE
> Otherwise it is like DLTM

> EDIT is used to edit expressions
> Format:
X1 EDIT!
> Example:

PD3 EDIT!
> See EDITING in Section IV

> EMBED embeds the first expression in STM in parentheses
> Format:
(... --> EMBED)
> Example: If STM: (AA BB CC), then EMBED produces
STM: ((AA) BB CC)

> FAIL terminates the action sequence
> Format:
(... --> FAIL)
> Example: Assume ACT/CTRL is on, then
ACT/C: (OPR 10 W\NUM <W .+ FAIL) will FAIL if NUM is
greater than 9 when the program passes through ACT/CTRL

> HOLD holds (ceases to increment) TIME and the COUNT.<ctrl>s
> Format:
(... --> HOLD)
> Example:
(--> HOLD (SAY S"INPUT AN ELM") RELEASE ATTEND)

> ILTM inserts an expression at the front of LTM
> If ILTM is used in a command expression, then if the
instantiated template expression is unnamed or has been
opened, it is copied before it is inserted into LTM
> Formats:
(... --> (ILTM X1))
X1 ILTM!
> Examples:
(--> (ILTM (AA X)))
(AA BB) ILTM!

> INIT.ACT is a list of actions which are executed by START after STM
has been set equal to STMI and before the production
system begins its first cycle
> Default: INIT.ACT: ()
> Format:
INIT.ACT: (X1 X2 ... Xn)
> Example:
INIT.ACT: ((SAY L"INPUT AA") ATTEND)

> INIT.STM creates a new version of STM which is a copy of STMI
> It is called by RESET (and thus by START)
> Format:
INIT.STMI!
> For INIT.TE, INIT.TEWM and INIT.LTM, see INITIALIZATION
in Section IV

> ITE inserts an expression at the front of TE
> > Otherwise it is like ILTM

> LOCTE locates an expression in TE
> > It does not modify TE or STM; it simply sets the pointer TEL for the operation of =TE=>
> > Format:
(... --> (LOCTE X1))
> > Example:
(--> (LOCTE (BB)) (BB =TE=> CC))

> NTC notices an expression in STM
> > It searches STM for an element matching its argument which is a template expression; if it does not find one, the production fails at that point; if it does, it moves the element in the same way it would have been moved by automatic rehearsal had it matched a condition expression
> > NTC assigns a new dollar variable, $i, where $i is the first unassigned dollar variable between $1 and $9
> > Format:
(... --> (NTC X1))
> > Example: If REHEARSE is set to RECOGNITION and REHEARSE.HOW to MOVE.FRONT, then firing PD6: (BB --> (NTC CC)) on STM: (AA BB CC) would result in STM: (CC BB AA)

> NTCLTM notices an expression in LTM
> > It searches LTM for an element matching its argument which is a template expression; if it does not find one, the production fails at that point; it it does, it inserts the element found into STM according to the option settings of STM.ADD.LOC and STM.ADD.HOW; it does not modify LTM
> > Format:
(... --> (NTCLTM X1))
> > Example:
(--> (NTCLTM (LAND)))

> NTCTE notices an expression in TE
> > It may be used in both condition expressions and action expressions; in both cases it searches TE for an element matching its argument which is a template expression, and in both cases it sets the pointer TEL for the operation of =TE=>
> > > As a condition expression, it is like a condition template expression except that its argument is

matched with the elements in TE rather than with the elements in STM and if a match is found a copy of the TE element is added to STM
> The automatic rehearsal mechanism skips it
> As an action expression it is like NTCLTM
> Format:
((NTCTE X1) --> (NTCTE X2))
> Example: If STM: (AA (BB CC) (DD) NIL NIL), TE: ((MM PP) NN), and PD16: ((NTCTE NN) (BB CC) --> (NTCTE (MM))), then after PD16 is fired, STM: ((MM PP) (BB CC) NN AA (DD)) and TE: ((MM PP) NN)

> OFF — turns a process off
> It is used on processes with no inputs (primarily on control-point processes); it has no effect if the process is already turned off
> Format:
X1 OFF!
> Example:
PR.ACT OFF!

> OFF.ALL — turns off all processes in a list by doing OFF on each element of the list
> Format:
(X1 X2 ... Xn) OFF.ALL!
> Example:
(PR.ACT PS/CTRL PS/C) OFF.ALL!

> ON — turns a process on
> It is used on processes that have been turned off; it has no effect if the process is already on
> Format:
X1 ON!
> Example:
PR.ACT ON!

> ON.ALL — turns on all processes in a list by doing ON on each element of the list
> Format:
(X1 X2 ... Xn) ON.ALL!
> Example:
(PR.ACT PS/CTRL PS/C) ON.ALL!

> OPEN — makes a named expression penetrable; that is, after executing OPEN on a named symbol, that symbol will be treated as if it were not named
> Format:

X1 OPEN!
> Example: If you want LBL: (AA BB) to match (AA BB), then do LBL OPEN!

> PD executes a production on STM
> If PD is used in a command expression and the production it is to execute is not satisfied, then processing continues with the next action element
> Formats:
X1 PD!
(... --> (PD X1))
> Examples:
PD2 PD!
(--> (PD PD2))

> PD.TE executes a production on TEWM
> Otherwise it is like PD

> PR prints an expression; both the name, if it exists, and the structure are printed
> ? and PR! are equivalent
> Formats:
X1 PR!
(... --> (PR X1))
> Examples:
PD2 PR!
(--> (PR (ELM CC))

> PR$i (PRINT $i for i from 1 to 9) prints the element in STM that matched the ith condition element
> Format:
(CE1 CE2 --> PR$1 PR$2)
> Exampl
((ELM (DD)) --> PR$1)

> PR.<crtl> prints the expression associated with the control point <crtl>
> A specific print routine is defined for each control point; it may be redefined by the user
> Example: You may wish to redefine PR.ACT so that it prints STM before it prints the action element; do UNPROTECT! PR.ACT: (PRTIME STM PR PRTIME S"ACTION- " WRS ZPDAE S PRXS) PROTECT!

> PRNAME prints the name of an expression; if the expression has no name, then its internal address is printed
> Format:
(... --> (PRNAME X1))

> Example:
(X --> (PRNAME X))

> PROTECT protects the user from inadvertently using (or redefining) a routine in the underlying programming language (L*)
> PSG is protected unless the user does UNPROTECT!

> PRPSPD prints a production memory and then all of its productions
> ?PS and PRPSPD! are equivalent
> Format:
PSX1 PRPSPD!
> Example:
PS1 ?PS

> PRSTRING prints a string
> A literal string is written S"SOME STRING"
> Formats:
S"X1" PRSTRING!
(... --> (PRSTRING S"X1"))
> Examples:
S"FILE LOADED" PRSTRING!
(--> (PRSTRING S"INPUT (ELM BB)"))

> PRVL prints the value of a variable element; the value will be the expression that was assigned during matching
> Format:
(X1 --> (PRVL X1))
> Example: If X = (AA BB), where the "=" is used to indicate that the expression (AA BB) does not have the name X, but is only a temporarily assigned value (over the scope of a single production), then (PRVL X) would print X = (AA BB)

> PS executes a set of productions on the present STM; that is, RESET is not called; see START
> If PS is used in a command expression and no production in the set of productions it is to execute is satisfied, then processing continues with the next action element
> Formats:
X1 PS!
(... --> (PS X1))
> Examples:
PS2 PS!
(PD1 PD6 PD2) PS!
(--> (PS PS2))
(--> (PS (PD1 PD6 PD2)))

> PS.TE executes a production memory on the present TEWM

> Otherwise it is like PS

> PS.1 executes a production memory on STM for one cycle
> It is like PS except that it quits after a single selection; thus it operates like a case statement

> PS.1.TE executes a production memory on TEWM for one cycle
> Otherwise it is like PS.1

> RDF reads a file from the user's disk area; the file is read into the job just as if it were being typed in at a terminal
> All commands followed by an exclamation point are executed during the read as they are encountered
> All printing goes to the terminal
> Format:
<filnam>.<ext> RDF!
> Example:
TST1.PSG RDF!

> RDFPPN reads a file from another individual's disk area
> It is identical to RDF except that it takes a Project-programmer-number (PPN) without brackets as its second argument
> Format:
<filnam>.<ext> <cmuppn> RDFPPN!
> Example:
SAMPLE.PSG A110PS00 RDFPPN!

> RECORD records the entire terminal interaction on a file
> ↑O, which turns off display at the terminal while leaving the output generating, will not turn off the output stream to the file
> RECORD does not record what happens if you go out to the Monitor with ↑C and then return
> Format:
<filnam>.<ext> RECORD!
> Example:
PS3RG.1 RECORD!

> RECORD. terminates recording, closes the record file, and prints a message to let you know the file has been closed
> Format:
RECORD.!
> Emergency procedure: If you are recording and you end up in the Monitor with an unrecoverable error (so that you cannot go back and RECORD.!), then do FINISH (or FIN); this is a Monitor command to close files

> RELEASE undoes HOLD
> > Format:
> > (... --> RELEASE)
> > Example:
> > (--> HOLD (SAY S"INPUT AN ELM") RELEASE ATTEND)

> RESET is called by START; it sets STM, TE, TEWM, LTM, Counts, and TIME to their initial states
> > Formats:
> > RESET!
> > (... --> RESET)
> > Examples:
> > RESET!
> > (--> RESET)
> > See INITIALIZATION in Section IV

> RPL replaces X1 by X2 throughout X3 if X3 is an element in working memory and X1 is that element or a subexpression in that element
> > Format:
> > (... --> (RPL X1 X2 X3))
> > Examples:
> > ((AA X (BB X Y)) (CC X) --> (RPL X Y $1))
> > ((AA X (BB X Y)) (CC X) --> (RPL $2 (DD X) $2))

> S.<option> is a process that is executed to select an option setting for <option>
> > See OPTIONS in Section IV

> SAVE saves a core image of your job
> > After doing a SAVE you will still be in the job and able to continue; when you run the job later, you will be back in exactly the same state as you were when you did the SAVE
> > Format:
> > <filnam> SAVE!
> > Example:
> > PSGEXP SAVE!

> SAY prints the rest of the expression in a noticeable format
> > If Xi is a list, then it prints its name, if it has one, and the structure; if Xi is (or contains) a variable, the value of the variable is printed
> > If Xi is T/W, then it prints its structure
> > If Xi is a literal string (eg, S"A STRING"), then it prints that string
> > Format:

(... --> (SAY X1 X2 ... Xn))
> Example:
((AGE X) --> (SAY S"THE GOAT IS " X S" YEARS OLD"))

> START executes a production memory on STMI
> It is equivalent to (RESET PS)
> Format:
X1 START!
> Examples:
PS3 START!
(PD3 PD6) START!

> STOP stops executing the production system at the end of the current production
> Format:
(... --> STOP)
> Example:
(--> (SAY S"FINISHED") STOP)

> STM+1 adds 1 symbol to STM by inserting a NIL at the end; this is a permanent lengthening
> Format:
(... --> STM+1)
> Example: If STM.SIZE is FIXED, with STM of length 6, then STM+1 will increase the number of elements in STM to 7

> STM-1 subtracts one symbol from STM by removing the last symbol
> Otherwise it is like STM+1

> STMI+1 adds 1 symbol to STMI
> Otherwise it is like STM+1

> STMI-1 subtracts 1 symbol from STMI
> Otherwise it is like STM+1

> TIME.PROCESS sets a process to be timed; the time is in Pc seconds accounted against the actual running time of the program, measured with a 10 Us clock
> The time is printed at the end of each execution
> Format:
X1 TIME.PROCESS!
> Example:
PS TIME.PROCESS!

> UNPROTECT gives the user access to the routines written in the underlying programming language (L*) by changing ZCXCRL and ZCXRGL

> This command should not be used by anyone unfamiliar with L*

> UNTIME.PROCESS sets a process so that it will no longer be timed; it has no effect if the process was not set to be timed
> Format:
X1 UNTIME.PROCESS!
> Example:
PS UNTIME.PROCESS!

> USRSAV does a SAVE on a file that is set-up for a user; when run it will be in START/C
> Format:
<filnam> USRSAV!
> Example:
PSGU USRSAV!

> WHERE prints the control-point the system is currently at
> Format:
WHERE!
> Example: If conrol has just been passed to you, type WHERE! to find out where you are

> WHIZ turns on the standard control-point actions for rapid operation given on WHIZ.LIST
> Initially WHIZ.LIST contains no calls to the terminal except CALL/C and no PR.<ctrl>s except PR.PS, PR.PD, and PR.CALL
> WHIZ.LIST can be modified by the user
> Format:
WHIZ!
> See CONTROL-POINTS in Section IV