PAS-II REFERENCE MANUAL

D. A. Waterman

Version 29, June 17, 1973

TABLE OF CONTENTS

1. INTRODUCTION

1.1. AUTOMATIC PROTOCOL ANALYSIS

Automatic protocol analysis is a joint effort by man and machine to infer, from the verbalization of a subject solving a problem, the underlying information processes occurring during solution (1). An initial goal for automatic protocol analysis is to provide a description of the subject's behavior. An intermediate goal is to provide a model of the subject; one which can generate a description of his problem solving behavior. A long-term goal is to automate the analysis more completely, particularly the induction processes, such as induction of the problem space from the verbal protocol, or induction of the model from the descripion of the behavior of the subject.

1.2. GOAL OF PAS-II

The PAS-II system (2) is an interactive, task-free version of an earlier protocol analysis system, PAS-I (3, 4). At the current level of development, the goal of PAS-II is behavior description. The system is designed to analyze protocols and produce as output a problem behavior graph (PBG) describing the subject's search through a posited problem space. The input to PAS-II is a transcribed text of the subject's verbal protocol, plus a number of rules defining the problem space, grammar, and certain data transformation operations. The program performs a linguistic analysis of the text, producing a series of semantic elements. These elements are further processed to provide tentative groupings, each containing one operator plus knowledge representing the operator's inputs and outputs. The operator groups are represented as nodes in the PBG. Thus the output from PAS-II is a graph which defines the subject's knowledge state at each point in time and the operators he applied to change that knowledge.

1.3. RUNNING THE SYSTEM

PAS-II is currently running in LISP on the CMU PDP10A system. To load PAS-II type R PAS to the monitor. Then type either (BEGIN) or (PAS) to the LISP interpreter to actually enter PAS-II. To leave PAS-II either type EXIT to return to the LISP interpreter, or (control)C to return to the PDP-10 monitor.

PAS-II will run in Stanford LISP compiled, and either Stanford or Irvine LISP interpreted. To obtain your own copy of the system you need disk files PAS2.LAP(X320DW28) and PAS2P.LAP(X320DW28) (the compiled functions) and PAS2.LSP(X320DW28) (the uncompiled functions). You will need 65K of core and 43000 words of binary program space to assemble the compiled functions and load the uncompiled ones. The regular and special pushdown lists should be set to 3000 words each, with full words set to 1000. To run PAS-II interpreted simply load file PAS2(X320DW28) and run.

2. SYSTEM ORGANIZATION

2.1. MODULAR ORGANIZATION

2.1.1. Run and Rule Modes

PAS-II is organized as a modular data analysis system. The basic unit of organization is the mode: a processing state which has associated with it a buffer capable of holding rules or data. This buffer can be modified by the editing functions available in the command language. There are three types of modes: run modes, which hold the data being anallyzed, rule modes, which hold the processing rules, and auxiliary modes, which hold task-free system-oriented rules.

The difference between run and rule modes is that in a run mode you have two options: you can process the data in the previous run mode by applying rules from a rule mode, or you can edit the data in the current run mode. In a rule mode you have only one option: that of editing the rules in that mode.

Every run mode, except text, trace1, and trace3 has one or more rule modes associated with it. When a run mode is used to process data, the processing always consists of applying the rules from the associated rule modes to the data.

2.1.2. Auxiliary Modes

There are five auxiliary modes: save, control, association, scratch, and information. The save mode contains rules which specify which mode buffers are to be saved on (or read into from) a disk file when the WRITE (or READ) command is executed. The control mode contains rules which define the control cycle and control flow through the system. The association mode contains rules which specify which rule modes are associated with each run mode. The scratch mode is a general purpose, temporary buffer used as the destination mode for the apply function. The information mode, however, holds no rules and processes no data. It is unique in that it contains no buffer and recognizes none of the functions that constitute the command language. This mode provides the user with general information about PAS-II: its basic organization, purpose and techniques of operation. This is to be contrasted with the HELP function, which provides the user with specific, on-the-spot information about the mode he is in.

2.1.3. Mode Table

The PAS-II modes are listed below:

Run	Rule	Auxiliary
text		association
topic	segmentation	control
linguistic1	extraction	save
linguistic2	space, grammar	information
semantic1	integration	scratch
semantic2	normalization	
semantic3	grouping	
graphic1	unknowns	
graphic2	origin	
graphic3	conflict, pbg	
trace1		
trace2	ps, memory	
trace3		
trace4	match	

2.2. FUNCTIONAL ORGANIZATION

2.2.1. Types of Functions

The basic functions are: core, create, display, erase, exit, help, mode, move, next, prior, rule, run, and (mode name). They are applicable in all run and rule modes. Note that every mode name is a function, which when executed puts you into that mode. The most important function for the novice is help; it provides information about the use of all functions in every mode. Be sure to use help after you leave the information mode.

The edit functions are: break, connect, define, delete, ed, insert, read, renumber, and write. Most of them are applicable in all run and rule modes, and are used to edit the data in the mode buffers. They are applicable in the save mode but not in the information mode.

The flag functions are: automatic, batch, comment, fast, hush, numbers, print, search, suppress, time, version1, and version2. They are applicable in all but the information mode, and are used to set switches which control processing in the run modes.

The process functions are: again, apply, copy, go, recopy, restart, and start. They are applicable in all run modes, and are used to start the processing of data.

2.2.2. Executing Functions

Functions can be executed in all modes except the information mode: to execute a function type its name or, if it has arguments, its name and the arguments in parentheses. For example, typing 'DISPLAY' or '(DISPLAY)' or '(DISPLAY 3)' will execute the display function. It is not necessary to type the entire function name: any unambiguous initial substring of the function name will evoke execution, and if the name ends in a digit, any initial substring followed by that digit will evoke execution. For example, 'DISPLA', 'DISPL', and 'DISP' will all execute the display function. 'GRAPHIC3', 'GRAPH3' and 'G3' will all execute the graphic3 function. Note l: function names used as arguments to the help function cannot be abbreviated. Note 2: parentheses are needed only for disambiguation. Thus typing 'BREAK 1 2 RENUMBER DISPLAY' is equivalent to typing (BREAK 1 2) RENUMBER DISPLAY'.

2.2.3. Entering Data

A carriage return (CR) enters a line of data into the system. To continue from one line to the next without terminating the previous line use altmode instead of carriage return if you are on a tty. Use underline if you are on a Datel or 2741. A carriage return in response to a yes or no type question is interpreted as meaning no. The only exception to this is OK? question which interprets a carriage return as meaning yes.

The system indicates that it is ready to accept tty input by typing: ☆, (digit)., N(digit), or (string) = .

2.2.4. Numbers as Arguments

Most of the edit functions take numbers or number-groups as arguments. A number must be either an integer (1, 5, 56, etc.) or a real (1.5, 17.02, 0.7, etc.) without a plus or minus sign. A number group has the form: number-number or number+number. For example, 3-6 stands for 3, 4, 5, and 6. 12+3 stands for 12, 13, 14, and 15. An asterisk (☆) can be used to designate the last item in the buffer. Thus if the buffer contained ten lines (1-10) then typing '(DISPLAY 1 4 ☆)' would display lines 1, 4, and 10.

2.2.5. Function Table

The PAS-II functions are listed below:

Basic	Edit	Flag	Process
(mode name)	break	automatic	again
core	connect	batch	apply
create	define	comment	copy
display	delete	fast	go

```
erase       ed          hush        recopy
exit        insert      numbers     restart
help        read        print       start
mode        renumber    search
move        write       suppress
next                    time
prior                   version1
rule                    version2
run
```

2.3. PROCESSING DATA

2.3.1. Response to OK? Question

When the system finishes a processing step it will type a number or number range (indicating the lines processed from the previous mode) and lines of data (representing the lines of output to be added to the current mode). If the system types:

```
4. D IS 5
   T IS 0
```

this means that from line 4 of the previous mode the processing produced the above two lines to be added to the end of the current mode buffer. At this point (if batch = F) the system will ask 'OK?'. Type just a carriage return to signify that the processing was ok and you want to continue. Or you may respond as shown below:

? : means forget the last processing step, don't do it.
U: n : means consider that the input for the last processing step was the first n lines in the previous mode rather than what was used.
UR: n : means consider that both the input and the output for the last processing step were the first n lines in the previous mode's buffer.
R: string : means consider that the output for the last processing step is the string including everything between the R: and the next carriage return. The string may contain the symbol & to indicate that the output goes to different (but adjacent) lines in the current mode buffer.
(function call) : means execute the function in the normal way, but if it's a flag continue mode processing. If it's not a flag, don't continue mode processing.

Below are examples of combinations of responses and their effect:

OK? ? TIME : means don't do this last processing step (from ?), set time = T, and continue processing

```
          (since time is a flag). Thus the last
          processing  step will be repeated, this time
          with time = T.
OK? ? TEXT  : means don't do this last processing step,
          and furthermore don't do it over (since text
          isn't a flag), just go to text mode and return
          control to the user.
OK? R: L IS 1 & T IS 2  : means put 'L IS 1' into the
          next line of the current mode buffer, and
          'T IS 2' into the line after that, just as if
          they were the result of the processing step.
OK? UR: 4  : means consider that the processing
          consisted of taking the first 4 lines from the
          previous mode buffer and putting them into the
          end of the current buffer.
OK? U: 2 R: (EQ D 5)  : means consider that the last
          processing step used the first 2 lines from the
          previous mode as input and produced as output
          (EQ D 5). Thus the 2 lines are deleted and
          (EQ D 5) is added to the current mode buffer.
```

3. MODES

3.1. RUN MODES

3.1.1. Text Mode

The text mode is used simply as a storage buffer which provides data for the topic mode to process. No processing is done in the text mode.

In the text mode, the define function is used to type material into the text buffer. Typically this material is a string of text representing a subject's protocol.

Data in the text mode should consist of a string of text all stored in one line of the text buffer. For example: 'EACH D IS 5 ; THEREFORE T IS 0. I 'M CARRYING 1' would be typical for cryptarithmetic.

3.1.2. Topic Mode

The topic mode is used to process text data by applying the segmentation rules and placing the resulting segmented text into the topic buffer. These segments are called topic data.

In the topic mode, the define function is used to define topic segments representing a subject's segmented protocol.

Data in the topic mode should consist of segments of text, where each segment is in one line of the topic buffer and consists of basically a single item or assertion. For example:

```
          EACH D IS 5 ;
          THEREFORE T IS 0 .
```

are two typical segments for cryptarithmetic.

3.1.3. Linguistic1 Mode

The linguistic1 mode is used to process topic data by applying the extraction rules and placing the result, some subset of the topic data, into the linguistic1 buffer. This result is called linguistic1 data.

The extraction rules simply divide the contents of the topic buffer into smaller, more easily handled groups. For example, if the topic buffer contained 40 lines of data and the extraction rule were set to 10, then the data would be processed in 4 groups of 10 lines each, with each group being completely processed (linguistic2 through graphic3 modes) before processing starts for the next group.

In the linguistic1 mode, the define function is used to define a small set of topic segments representing an initial estimate of an operator group.

Data in the linguistic1 mode has exactly the same format as data in the topic mode.

3.1.4. Linguistic2 Mode

The linguistic2 mode is used to process linguistic1 data by applying the space and grammar rules and placing the result, a set of semantic elements, into the linguistic2 buffer. This result is called linguistic2 data.

In the linguistic2 mode, the define function is used to define semantic elements: operator, knowledge, and indicator elements.

Data in the linguistic2 mode should consist of sets of semantic elements, with each line of the buffer containing a set of one or more elements. For example:

```
            (EQ D 5)
            (THEREFORE) (EQ T 0)
```

are two sets for cryptarithmetic, the first containing one element, the second containing two elements.

3.1.5. Semantic1 Mode

The semantic1 mode is used to process linguistic2 data by applying the integration rules and placing the result, integrated semantic elements, into the semantic1 buffer. This result is called semantic1 data.

In the semantic1 mode, the define function is used to define semantic elements which have undergone integration processing.

Data in the semantic1 mode has essentially the same format as data in the linguistic2 mode.

3.1.6. Semantic2 Mode

The semantic2 mode is used to process semantic1 data by applying the normalization rules and placing the result, normalized semantic elements, into the semantic2 buffer. This result is called semantic2 data.

In the semantic2 mode, the define function is used to define semantic elements which have undergone integration and normalization processing.

Data in the semantic2 mode has essentially the same format as data in the linguistic2 mode, except that the semantic elements 'BECAUSEOF', 'COND', and 'OPIO' may also be present.

3.1.7. Semantic3 Mode

The semantic3 mode is used to process semantic2 data by applying the grouping rules and placing the result, a protogroup, into the semantic3 buffer. This result is called semantic3 data.

In the semantic3 mode, the define function is used to define a small number of semantic elements which represent a reasonable estimate of an operator group.

Data in the semantic3 mode has exactly the same format as data in the semantic2 mode.

3.1.8. Graphic1 Mode

The graphic1 mode is used to process semantic3 data by applying the unknowns rules and placing the result, elements without unknowns, into the graphic1 buffer. This result is called graphic1 data.

In the graphic1 mode, the define function is used to define semantic elements which, in general, do not contain unknowns.

Data in the graphic1 mode has exactly the same format as data in the semantic2 mode.

3.1.9. Graphic2 Mode

The graphic2 mode is used to process graphic1 data by applying the origin rules and placing the result, an operator group, into the graphic2 buffer. This result is called graphic2 data.

In graphic2 the system will query the user about the origins of the elements in graphic1. For a knowledge element, the system will ask the user for the operator that produced it, the operator's inputs, the operators that produced the inputs, etc. For each operator element it will ask for the operator (in case the user wants to change notation) its inputs, outputs, etc. If there are multiple operators in graphic1 the system will ask if each is an occurance of the others. For example, in cryptarithmetic for DONALD+GERALD=ROBERT (PLUS D D) can be considered an occurance of (PC 1), process column

1.

In the graphic2 mode, the define function is used to define operator groups such that each group occupies one line of the graphic2 buffer.

Data in the graphic2 mode must have the following format:

```
     (operator) (input list) (outputs)
```

For example: (PC 2) ((EQ D 5) (EQ C1 0)) (EQ T 0) (EQ C2 1) represents a cryptarithmetic operator group with operator (PC 2), inputs (EQ D 5) and (EQ C1 0), and outputs (EQ T 0) and (EQ C2 1).

3.1.10. Graphic3 Mode

The graphic3 mode is used to process graphic2 data by applying the conflict and pbg rules and placing the result, a pbg node, into the graphic3 buffer. This result is called the graphic3 data or problem behavior graph.

In the graphic3 mode, the define function is used to define PBG nodes. The system prompts with a node number, even though define takes line numbers as arguments.

Data for the graphic3 mode must have the following format:

```
   (number) OP (operator) IN (knowledge) OUT (knowledge)
```

For example: 3 OP (PC 1) IN (EQ D 5) (EQ C1 0) OUT (EQ T 0) defines a node which points back to node 3 and consists of the operator (PC 1) with inputs (EQ D 5) (EQ C1 0) and an output of (EQ T 0). The minimal legal format is: OP (operator).

3.1.11. Trace1 Mode

The trace1 mode is used to linearize the pbg stored in the graphic3 buffer. Executing 'GO' in the trace1 mode puts a linear version of the pbg into the trace1 buffer. In the linear pbg, backups are represented as nodes and the pbg itself is just an ordered set of nodes.

In the trace1 mode, the define function is used to define a linear representation of the pbg that is stored in the graphic3 mode.

In the trace1 mode each buffer line either represents the contents of one pbg node or indicates a backup. For example:

```
 1. (PC 1) ((EQ D 5)(EQ C1 0)) ((EQ C2 1))
 2. (AV R) () ((AEQ R 6))
 3. (PC 2) ((EQ C2 1)) ((ODD R))
 4. (BACKUP (PC 1))
```

would be typical for cryptarithmetic.

3.1.12. Trace2 Mode

The trace2 mode is used to generate a trace of the production system defined in the ps rule mode. The initial contents of the various memories are defined in the memory rule mode.

In the trace2 mode, the define function is used to define a list of problem space or production system operators. This list represents the trace produced by the production system that was defined in the ps rule mode.

Elements in the trace2 mode must have a format where each buffer line contains just one problem space or production system operator. This list represents the trace produced by a production system. Thus,

```
     1. (PC 1)
     2. (DEPOSIT (EQ T 0))
     3. (DEPOSIT (EQ C2 1))
     4. (PC 2)
     5. (DEPOSIT (ODD R))
     6. (REMOVE (EQ R 6))
```

would be a typical trace for cryptarithmetic.

3.1.13. Trace3 Mode

The trace3 mode is used to standardize the trace currently in the trace2 buffer by giving it the same general form as the pbg which is stored in the trace1 buffer.

In the trace3 mode the define function is used to define a list of problem space or production system operators. This list represents a standardized version of the trace currently in the trace2 mode buffer.

In the trace3 mode each buffer line represents either a problem space operator with inputs and outputs or a production system operator. For example:

```
 1. (PC 1) ((EQ D 5)(EQ C1 0)) ((EQ C2 1))
 2. (AV R) () ((AEQ R 6))
 3. (PC 2) ((EQ C2 1)) ((ODD R))
 4. (REMOVE (AEQ R 6))
```

would be typical for cryptarithmetic.

3.1.14. Trace4 Mode

The trace4 mode is used to find the best match between the pbg in trace1 and the trace in trace3. The problem is similar to that of comparing two pbg's and trying to fine the best way to pair similar nodes so a measure of how well one pbg matches the other can be obtained.

In the trace4 mode the define function is used to define a representation of the correspondence or best match between the pbg in trace1 and the trace in trace3.

Each buffer line in the trace4 mode must contain some representation of either a pbg node, a trace node, or both. The pbg nodes are denoted g1, g2, ..., and the trace nodes t1, t2, ... For example:

```
 1. G1 ((PC 1)((EQ D 5))((EQ T 0)))
 2. G2 ((AV R)()((AEQ R 7))) T1 ((AV R)()((AEQ R 7)))
 3. T2 ((PC 2)()())
```

indicates that node G2 from the pbg in trace1 and node T1 from the trace in trace3 matched each other while none of the other nodes matched.

3.2. RULE MODES

3.2.1. Segmentation Mode

In the segmentation mode, the define function is used to define segmentation rules: rules for breaking the text into topic segments. These rules are used by the topic mode to process text data.

Rules in the segmentation mode must have the following format: (string) / (string), where a string is any sequence of words, punctuation marks, or word classes, including the null sequence. For example, . / means break the text after the occurrence of every period, / AND SO means break the text just before the occurrence of every 'AND SO' string, and <D> , / <D> means break the text after every occurrence of a digit comma string which is followed by a digit. (Assuming that <D> stands for a class of digits which is already defined in the grammar mode before the class is mentioned in the segmentation mode.)

3.2.2. Extraction Mode

In the extraction mode, the define function is used to define an extraction rule: a rule specifying the minimum number of segments which are likely to contain one entire operator group. These rules are used by the linguistic1 mode to process topic data.

Rules in the extraction mode must have the format: (number), and only the rule in the first line of the buffer is used. For example, 6 in line one means to extract 6 segments from the topic mode and use these segments as an initial estimate of an operator group. If the first line of the extraction buffer does not consist of a number, then segments are extracted 10 at a time.

3.2.3. Space Mode

In the space mode, the define function is used to define the problem space in terms of semantic elements. These problem space rules are used by the linguistic2 mode to help process linguistic1 data.

Most rules in the space mode have the following format:

```
            (semantic-element) (type)
```

where the semantic element has either word classes or other semantic elements for arguments, and the type is either KN, OP, or IND, depending on whether the element is a knowledge, operator, or indicator element. For example,

```
            (EQ <V> <D>) KN
            (EQC (PLUS <U> <U>) <U>) OP
            (NEG) IND
```

are space rules for cryptarithmetic. Note that all classes (<V>, <D>, and <U> above) must be defined in the grammar mode before the space rules are applied to data.

In general, rules in the space mode must have the format:

```
        (semantic-template) (type) .
```

The semantic-template is used to match the tree resulting from the application of grammar rules, in order to construct a semantic element containing specific entities from the particular linguistic segment at hand. The left-most item in the template should be the name of the grammar class to which the semantic element corresponds, which will be the label of the entire tree being matched except that here the name occurs without angle-brackets. The other items in the template are names of grammar classes (this time with the angle-brackets), names of spaset's, names of splabl's, or semantic sub-templates in parentheses which are then used to match particular subtrees of the main tree which have labels (according to grammar classes) which correspond to the left-most item in the sub-template in much the same way as the label of the entire tree corresponds to the left-most name in the semantic-template. Elements from the tree are in general matched to elements in the template in left-to-right order, with the exception that ordering of subtrees relative to the other elements is insignificant, that is the subtrees are matched independently of the other elements and in fact in some cases it will be desirable to pick out a particular subtree (with a particular label) no matter in which position it occurs relative to the other elements since in some cases order can be reversed with the same semantic meaning. For instance the semantic element

```
   (EQC (PLUS B C) A)
```

can be used to represent the meaning of: 'A IS B PLUS C' and 'B PLUS C IS A'.

The type part of the space rule is one of the words KN, OP, IND, SPASET, or SPLABL. The first three denote respectively knowledge, operator, and indicator elements. These three are used to specify which grammar rules are to be applied to the linguistic segments, and the order in which those rules are taken. SPASET and SPLABL have nothing to do with the grammar rules, but deal with ways of using the space rule templates for building semantic elements. Examples will follow as each feature of the templates is discussed in more detail. The most common element used in a template is a grammar class, usually a class which defines terminal elements. If the class doesn't define a terminal, a subtree of the parse tree is put into the semantic element. For example, given grammar rules:

```
     <EQC> = (<SUM> IS <LETTER>) (<LETTER> IS <SUM>)
     <EQ> = (<LETTER> IS <DIGIT>)
     <SUM> = (<LETTER> PLUS <LETTER>)
     <LETTER> = A B C
     <DIGIT> = 3 5
```

and space rules:

```
     (EQ <LETTER> <DIGIT>) KN
     (EQC (SUM <LETTER> <LETTER>) <LETTER>) OP
```

we would get semantic elements as follows:

```
     (EQ B 3) from 'B IS 3'
     (EQC (SUM A B) C) from 'A PLUS B IS C'
and   (EQC (SUM B C) A) from 'A IS B PLUS C'.
```

A space rule template can contain as little as the name. This is common for indicator elements, for example '(AND) IND', '(THEREFORE) IND'.

An alternative to the use of grammatical class names in the template is to use the name of a spaset, which stands for the set of class names which are listed after the name in the space rule. The semantic element will then contain the terminal associated with whichever class name in the spaset occurs in the parse tree. An important restriction to note is that within any particular template (excluding sub-templates) the spaset's must be disjoint, because the matcher cannot decide which spaset a particular class belongs to in case of conflict, and in fact will associate it with both spasets. As an example of use of a spaset, we can generalize <EQC> as defined in the above example to be:

```
     <EQC> = (<SUM> IS <LETDIG>) (<LETDIG> IS <SUM>)
     <LETDIG> = (<LETTER>) (<DIGIT>).
```

Then in space mode if we have:

```
     (EQC (SUM LD LD) LD) OP
     (LD <LETTER> <DIGIT>) SPASET
```

we get the semantic element

```
     (EQC (SUM A B) 3) from 'A PLUS B IS 3'
```

and the results in the above example would be unaltered.

In some cases it is desirable to use a label of a subtree as an item in a semantic element, instead of including more specific contents of the subtree. In this case we use the splabl type of space rule to give a name to that subtree from which we wish to extract the label of a subordinate subtree. For instance, suppose we want to know the type of an exclamation in a linguistic segment. If <EXCL> = (<HAPPY>) (<NEUTRAL>) (<SAD>), where <HAPPY>, <NEUTRAL>, or <SAD> are further defined to be specific sorts of utterances, then we could use

```
      (EX <EXCL>) SPLABL
      (EXCL EX) IND
```

to signal to the semantic matcher that it should return the name of the subtree of <EXCL>. For instance we would get (EXCL <HAPPY>) from 'VERY GOOD !' and (EXCL <SAD>) from 'UH OH !'.

Semantic templates can be recursive and refer to themselves as illustrated by the following example. If we have in the grammar:

```
 <SUM> = (<LETTER> PLUS <SUM>) (<LETTER> PLUS <LETTER>)
```

then we could have space rules

```
 (SUM <LETTER> (SUM)) KN
 (SUM <LETTER>) SPASET.
```

This will indicate that as long as there is a subtree with label <SUM>, it will match (SUM <LETTER> (SUM)), and so on recursively until a <SUM> is <LETTER> PLUS <LETTER>, in which case the template used for matching will be (SUM <LETTER> <LETTER>). The (SUM) part of the template is replaced by the element of the spaset for SUM. The element of this particular kind of spaset can be a name of another spaset, for instance:

```
 (SUM U (SUM)) KN
 (SUM U) SPASET
 (U <LETTER> <DIGIT>) SPASET.
```

Spaset names in spaset's are otherwise invalid. For example we would get the semantic element (SUM D (SUM G (SUM R B))) from 'D PLUS G PLUS R PLUS B'.

Sub-templates need not be specified in full if elsewhere there is a space rule defined with the same name as used in the sub-template. For instance if we have (SUM <LETTER> <LETTER>) KN then we can shorten (EQC (SUM <LETTER> <LETTER>) <LETTER>) OP to (EQC (SUM) <LETTER>). Each would return (EQC (SUM A B) C) from 'A PLUS B IS C'.

The left-most name in a sub-template may be a spaset name in which case the rest of the sub-template is used to match against whichever element in the spaset occurs in the actual parse tree. For instance, if the grammar contains:

```
      <EQC> = (<PLUSSUM> EQUAL <LETTER>)
      <PLUSSUM> = (<PLUSTWO>) (<SUM>)
      <PLUSTWO> = (TWO <LETTER> 'S)
      <SUM> = (<LETTER> PLUS <LETTER>)
```

and the space rules contain:

```
      (EQC (PLSM) <LETTER>) OP
      (PLSM <PLUSTWO> <SUM>) SPASET
      (PLUSTWO <LETTER>) KN
      (SUM <LETTER> <LETTER>) KN
```

then we would get (EQC (PLUSTWO B) T) from 'TWO B 'S EQUAL T' and (EQC (SUM B C) T) from 'B PLUS C EQUAL T'.

As an example illustrating a slightly different use consider a grammar containing:

```
      <FIND> = (FIND * <LETTER>)
      <NEED> = (NEED * <FIND>) (NEED * <IN>)
      <IN> = (HAVE * <LETTER>)
```

and space rules containing:

```
      (NEED (NEEDS <LETTER>)) OP
      (NEEDS <FIND> <IN>) SPASET .
```

Then we get (NEED (NEEDS A)) from 'I NEED TO FIND AN A' and from 'I NEED TO HAVE AN A'. Note that in this case the spaset name 'NEEDS' appears in the semantic element because the sub-template of the space rule had <LETTER> explicitly, while in the previous example, we only had (PLSM) in the space rule, and 'PLSM' did not appear in the semantic element, but either 'PLUSTWO' or 'SUM' did.

Any name, undefined elsewhere, used in any position in a space rule, other than the leftmost position of the main template, will be carried over literally to the resulting semantic element. If an otherwise-undefined name is used as a space rule name, this space rule is disregarded entirely in the processing.

This section gives the complete definition of the SPACE MODE, although not all of the features described from this point on are currently implemented. The rules in the SPACE MODE determine what grammatical analyses will be made of the text segments in an input buffer (eg, the TOPIC MODE) to produce semantic elements in an output buffer (eg, the LINGUISTIC2 MODE). All SPACE RULES are of the form:

```
   RULE-BODY  TYPE
```

The TYPE determines the interpretation of the RULE-BODY. For each TYPE we exhibit below the form of its RULE-BODY. After all the forms are presented, we give the interpretation.

Notation: In presenting the forms we use the following grammar-like conventions:

Terms that contain - stand for classes of forms.
The definition for the class of forms named X-Y is given by:

```
      X-Y :=  (here is the def)
```

Alternative forms for X-Y are indicated by:

```
      X-Y := alternative1 ! alternative2 ! ...
```

```
   Form classes ending in NAME (eg, X-NAME) may be any symbol
   if not defined further.

   Form classes ending in LIST (eg, X-LIST) may be any list
   of zero or more elements of form X.

 SEMANTIC-TEMPLATE  KN
 SEMANTIC-TEMPLATE  OP
 SEMANTIC-TEMPLATE  IND

    SEMANTIC-TEMPLATE := (TEMPLATE-NAME TEMPLATE-ELM-LIST)
    TEMPLATE-NAME := CLASS-NAME ! SPASET-NAME ! SPLABL-NAME
    TEMPLATE-ELM := <CLASS-NAME> ! SPASET-NAME ! SPLABL-NAME !
       PATH-NAME ! SYMBOL ! SUB-TEMPLATE ! (NOTE TEMPLATE-ELM)
    SUB-TEMPLATE := (SUB-TEMPLATE-NAME TEMPLATE-ELM-LIST)
    SUB-TEMPLATE-NAME := TEMPLATE-NAME ! PATH-NAME ! SYMBOL
    NOTE := !V  !  !O  !  !D  !  !U  !  !F
       Note: (1) CLASS-NAME is the name of a grammar class with
                 the <>-brackets stripped off, eg, XX for <XX>.
             (2) Temporarily TEMPLATE-NAMEs in SEMANTIC-
                 TEMPLATEs can only be CLASS-NAMEs.

 (SPASET-NAME SPASET-ELM-LIST)  SPASET

    SPASET-ELM := <CLASS-NAME> ! SPASET-NAME ! SPLABL-NAME !
                  PATH-NAME

 (SPLABL-NAME <CLASS-NAME>-LIST)  SPLABL

 (PATH-NAME STEP-LIST FINAL-EXP)  PATH

    STEP := <CLASS-NAME> ! SPASET-NAME ! S ! T !
            POSITIVE-INTEGER ! NEGATIVE-INTEGER
    FINAL-EXP :=  (null) ! C ! P ! L

 (TERM-ELM-LIST)  TERM

    TERM-ELM := <CLASS-NAME> ! SPASET-NAME

 (STAR-NAME TERM-ELM-LIST)  STAR

    STAR-NAME := *SYMBOL

 (TEMPLATE-NAME <CLASS-NAME> REPLACEMENT-NAME)  RETAIN
```

Space rules specify the semantic elements that are to be built from linguistic segments. The parser processes the Space rules of types OP, IND, and KN in the order of occurrence in the SPACE buffer. For each it searches for the corresponding grammar classes in the text segment under consideration (the current line of the input buffer). Space rules of types RETAIN and STAR are used to modify parsing actions. After the parse tree is formed for a given rule, its SEMANTIC-TEMPLATE specifies how to retrieve elements from the parse tree to make up the actual semantic elements. Space rules of types SPASET, SPLABL, PATH, and TERM are used in this construction process, and their order in the buffer is irrelevant.

SEMANTIC-TEMPLATE: Each template produces a single semantic element in the output buffer. All templates generated by a single line of text in the input buffer produce semantic elements in the same output line, and in the order in the SPACE buffer (ie, the order in which they are considered). The TEMPLATE-NAME determines a grammar class. This is <CLASS-NAME> if the TEMPLATE-NAME is CLASS-NAME; otherwise it is the grammar class determined by SPASET or SPLABL as the case may be (cf their definition). The specified grammar class is used to search the text as it exists when input to the space rule (it may have been modified by prior space rules, cf RETAIN). If the grammar class is not defined in the GRAMMAR then the template is ignored. The parse with <CLASS-NAME> determines a parse tree (a null one if the parse is unsuccessful). All the TEMPLATE-ELMs in the template refer to this parse tree, called the current parse tree, in determing their own behavior. They select subtrees and classes from it, but are limited to the analysis present in this orignal tree (ie, no further parsings occur).

TEMPLATE-ELM: Each element determines a single subelement of the output sementic element, positioned in the same place as in the

template. The subelement produced depends on the type of the template element (ie., whether <CLASS-NAME>, SPASET-NAME, ... etc.). The type

of the element is determined by its definition in a semantic rule elsewhere in the SPACE buffer. In general each type of semantic rule provides a means for obtaining some particular type of information from the parse tree made available by the <CLASS-NAME> of the SEMANTIC-TEMPLATE. SPASET, SPALABL and PATH are described below under their own types; the rest are covered here.

<CLASS-NAME> as TEMPLATE-ELM: This produces the subtree of the parse tree directly below <CLASS-NAME>. (This could be a single word if a terminal, a sequence of words, or an entire parse tree.)

SYMBOL as TEMPLATE-ELM: This produces a literal copy of itself in the semantic element. (By a SYMBOL is meant a not-otherwise defined in the GRAMMAR or the SPACE RULES.)

SUB-TEMPLATE: This produces a subelement in the semantic element that is the image of the subtemplate. The SUB-TEMPLATE-NAME determines a new place in the parse tree, and the TEMPLATE-ELMs in the subtemplate are taken relative to this sub-parse-tree, ie, it becomes the current parse tree. What is written in the semantic element for the SUB-TEMPLATE-NAME is determined by its type (eg, whether a SPASET, etc). A CLASS-NAME or SYMBOL is reproduced literally. If the SUB-TEMPLATE-NAME is a SYMBOL, then in essence it selects the null parse tree. Hence the entire SUB-TEMPLATE is copied over literally (since none of its SUB-TEMPLATE-ELMs will be found it the null tree).

NOTES: Various options about the semantic element to be produced are given by prefixing NOTES. The NOTE and the element it refers to are enclosed in parenthesis, though these parentheses do not show up in the final semantic element. The NOTES are:

!O OPTIONAL: If the element is not found in the parse tree, then no element is to appear in the semantic element.

!V VARIABLE: If the element is not found in the parse tree, then it is to appear in the semantic element as a literal (to be interpreted as a variable).

!D DUPLICATE: If the element is not found in the parse tree, then a duplicate of the last semantic element that was used for a similar element will occur in semantic element.
[DO NOT UNDERSTAND]

!U UNBRACKET: The <CLASS-NAME> following is to be produced in the semantic element without brackets, eg, as CLASS-NAME.
(If the subelement following is not of form <CLASS-NAME>, !U has no effect.)

!F FLATTEN: The parentheses are to be taken as indicating structure (ie, a SUB-TEMPLATE) for the purpose of forming the subelement, but the parentheses are to be removed in the semantic element (thus producing a set of sementic elements)

SPASET (SPACE ALTERNATIVE-SET): The occurrence of a SPASET-NAME in a TEMPLATE or SUB-TEMPLATE is to be considered replaced (at that occurrence) by the member of the SPASET (as defined in the SPASET rule-body) that is found in the parse-tree (as defined at the point of occurrence). Whatever other interpretation is to be given to the SPASET-NAME applies to the symbol that replaces it.

SPLABL (SPACE LABEL): A SPLABL is a special case of a SPASET, which has a specific convention for what expression is to be written into the semantic element. As in a SPASET, the first SPLABL-ELM which is found in the in the current parse tree is used. (This ultimately is a <CLASS-NAME>.) The expression produced in the semantic element is the level in the parse tree immediately under this <CLASS-NAME>.(Makes sense in a context in which the <CLASS-NAME> names a set of alternative subclasses, so that it functions as a variable. What is produced in the semantic element is then the value of the variable holding for the particular parse, ie, the the sub grammar class.)

PATH (PATH EXPRESSION): The PATH provides a general scheme for designating portions of the current parse tree to be produced in the semantic element. A PATH specifies a search starting at the top of the current parse tree. The PATH consists of a sequence of STEPs, each being taken from the result of the prior STEP and making a search in the parse tree relative to it. At the end of the path a specific node in the parse tree has been determined. FINAL-EXP determines what shall be produced in the semantic element: the label of the node (C, for class name), the entire parse tree (P) or the level one-below (L, as in the SPLABL). If the FINAL-EXP is not present the default convention determines it. The STEPS that can be taken are:

<CLASS-NAME>: Find the first occurrence of the <CLASS-NAME> in the current subtree.

SPASET-NAME: Find the first occurrence of whichever element of the SPASET is defined in the current subtree.

T: Find the terminal elements of the current subtree.

S: Find the lowest point at which a single <CLASS-NAME> (or terminal) comprehends the entire current subtree.

POSITIVE-INTEGER (N): Find the point N nodes further down the current subtree (towards the terminal). This cannot go below a terminal and a number greater than that is taken to specify the terminal.

NEGATIVE-INTEGER (-N): Find the point N nodes further back up the parse tree toward the top. This cannot go above the top of the current parse tree and a number greater than that is taken to specify the top.

TERM (TERMINAL DECLARATION): A TERM space rule declares that one or more <CLASS-NAME>s will be treated as terminal elements in determination of the semantic element from the TEMPLATE (and SUB-TEMPLATEs) once the parse tree has been determined from the text segment. This holds for all operations, eg, use of T or searches in PATHs. If a <CLASS-NAME> that is declared to be a terminal appears explicitly in a PATH, then the search continues below it (ie, it is not treated as a terminal for that search).

STAR (STAR-SYMBOL DECLARATION): A STAR space rule defines a new star-symbol for use in the grammar. Its name must begin with *. It operates like *, skipping over words in the text, except that it will not skip over any word that occurs in the definitions of the <CLASS-NAME>s in its definition (ie, directly or indirectly in the <CLASS-NAME>s, SPASETs or SPLABLs in TERM-LIST)

Note: Temporarily only terminals, <CLASS-NAME>s that define terminal sets and SPASETS whose elements are terminals or terminal classes can be used.

RETAIN: A RETAIN rule determines what happens to the text involved in a specific parse. In the context of the parse for the SEMANTIC-TEMPLATE with TEMPLATE-NAME, the terminals (words) for <CLASS-NAME> will be replaced by the symbol specified by REPLACEMENT-NAME. It occurs in the place of the first terminal of <CLASS-NAME>. If no REPLACEMENT-NAME is given, then all the terminals are left in the text.

[DOESN'T MAKE IT EASY TO KEEP ALL THE WORDS NOT DEALT WITH AT THEN END OF THE PARSE -- THE RESIDUAL]

3.2.4. Grammar Mode

In the grammar mode, the define function is used to define a grammar which includes grammatical classes for every element defined by the space rules. Both the space and grammar rules are used by the linguistic2 mode to process linguistic1 data.

Rules in the grammar mode must have the following format:

```
   <class> = (item item ...) (item item ...) ...
```

where an item is either a class (denoted by angle brackets) or a literal (such as a word, letter, or character). A # can be used at the beginning or end of the string of items to indicate a segment boundary, and a * can be used between two items to indicate a match with any string of text.

For example, <EQ> = (<LETTER> * IS * <DIGIT>) defines the class EQ as matching any string containing a member of the class LETTER, followed (not necessarily immediately) by the literal 'IS', followed (not necessarily immediately) by a member of the class DIGIT. <DIGIT> = (0) (1) (2) defines the class DIGIT as matching any string containing 0 or 1 or 2. Any grammar rule which is a disjunction of single literals can be written without parentheses, and defines what is called a terminal class. Thus <DIGIT> = 0 1 2 is a terminal class and is equivalent (even preferred) to the above form.

Classes may need to have extensions: thus if you have the rule <PLUS> = (<LETTER.L2> AND <LETTER.L1>) and in the space buffer you have the rule (PLUS <L> <L>) KN , then when 'D and G' is parsed D will be substituted for the second occurrence of <L> in the space rule and G for the first occurrence, giving (PLUS G D) as the resulting semantic element. If the rule had instead been <PLUS> = (<LETTER.L2:L> and <LETTER.L1>) then the result would have been (PLUS <L> D). the class name following the colon is used in place of the literal that matches the class. NOTE: Extensions are only needed if you are using the old grammar, i.e., the flag VERSION1 = T. Extensions are not needed if the flag VERSION2 = T when the grammar and space rules are defined and used.

Restrictions on grammar rules: The grammar rules cannot be 'left recursive', that is in the lists of items defining a grammar class, the leftmost item cannot be the same as the grammar class being defined. Any other item can be the same as the defined class, however, giving in effect full recursive capabilities. For example,

```
     <SUM> = (<LETTER> + <SUM>) (<LETTER>)
```

is allowed, but

```
     <SUM> = (<SUM> + <LETTER>) (<LETTER>)
```

is not allowed.

Also it is worthwhile to note that if there are several lists of items defining a grammar class, these are examined in left-to-right order to see if they exist in the current linguistic segment. It is generally necessary to put short alternatives which are contained in other longer ones after the others, so that the longest possible match is attempted first, before a shorter match which would also succeed.

The order in which grammar classes are matched against the linguistic segments to check their presence is determined by the order of the space rules defined in space mode.

3.2.5. Integration Mode

In the integration mode, the define function is used to define replacement rules which integrate information within a set of elements or between adjacent sets. These rules are used by the semantic1 mode to process linguistic2 data.

Rules in the integration mode must have the following format:

```
 e ... e / e ... e / ... => e ... e / e ... e / ...
```

where each e is a semantic element and the / indicates that the next elements occur on the next line of the buffer. The elements may contain variables, denoted by X1, X2, ... or by class names which refer to terminal classes. A class name in the left side of an integration rule will match any word in that class, however, class names in the data are not treated as variables. Consecutive strings of elements in adjacent lines can be referred to by using the variables S1, S2, ... Thus (DIGIT S1) refers to the list (DIGIT X1) / (DIGIT X2) / ... , a sequence of indefinite length.

These rules are replacement rules which mean that an occurrence of everything to the left of the arrow is replaced by everything to the right of the arrow. For example,

```
      (NEG) (EQ X1 X2) => (NEQ X1 X2)
```

means that if both (NEG) and the EQ element are found in the same set of elements (i.e. on the same buffer line) in any order then the EQ is changed to NEQ, and the (NEG) is deleted. Also,

```
    (EQ X1 X2) / (DIGIT X3) => (MEQ X1 X2 X3)
```

means that if EQ is found in one set of elements and DIGIT is found in the next adjacent set then they are transformed into the MEQ element. Furthermore,

```
    (EQ X1 X2) / (DIGIT S1) => (MEQ X1 X2 S1+)
```

means that if EQ is in one set and there are adjacent sets with DIGIT in them then the EQ element and all the DIGIT elements are replaced with (MEQ X1 X2 S1+), where S1+ stands for the list of all arguments of the DIGIT element. This rule would change:

```
   (IF) (EQ R 1)
   (DIGIT 3)     into    (IF) (MEQ R 1 3 7)
   (DIGIT 7)             (ODD R)
   (ODD R)
```

A rule with (DIGIT S1) as the left side applied to the data shown below:

```
      1. (DIGIT 1)(EQ L 4)
      2. (NEG)(DIGIT 3)
      3. (EQ R 7)
```

would match the first two lines above. The (DIGIT S1) matches (DIGIT 1) (EQ L 3) / (NEG)(DIGIT 3), that is, everything on both lines. The S1 matches 1 and 3, the arguments of DIGIT.

3.2.6. Normalization Mode

In the normalization mode, the define function is used to define replacement rules that create elements like 'BECAUSEOF', 'COND', and 'OPIO' which specify group relations existing between problem space elements. These rules are used by the semantic2 mode to process semantic1 data.

Rules in the normalization mode must have the following format: e ... e / e ... e / ... => e ... e / e ... e / ... as in the integration mode. But here the variables A1, A2, ... can be used to stand for any consecutive knowledge element sets connected by 'AND', i.e., A1 would match the consecutive sets (EQ D 5) / (AND) (EQ T 0) / (AND) (ODD R).

Also, K1, K2, ... can be used to stand for individual knowledge elements. These K's will match the first non-indicator element in a line, going from left to right. For example,

```
   A1 / (THEREFORE) A2 => (BECAUSEOF A1 A2)
```

would transform

```
 (EQ D 5)
 (AND)(EQ C1 0)        into (BECAUSEOF ((EQ D 5)(EQ C1 0))
 (THEREFORE)(EQ T 0)                   ((EQ T 0)(EQ C2 1)) )
 (AND)(EQ C2 1)
```

Also, the rule

```
 (IF) K1 / (THEN) K2 => (COND K1 K2)
```

would transform

```
 (IF)(OR)(EQ R 7)
 (YES)(THEN)(EQ C2 1)   into   (COND ((EQ R 7))((EQ C2 1)))
```

See the preliminary results paper(Waterman and Newell, 1972) pages 37-38 for examples of the use of A and K in normalization rules.

3.2.7. Grouping Mode

The grouping mode is currently non-operational. That is, the grouping rules are built into PAS-II at a level which is not accessible to the user. These rules operate on the data in the semantic2 mode, and result in transferring from semantic2 to semantic3 the largest consecutive sequence of element sets (starting from the element set in the first line of the semantic2 buffer) containing no more than one operator element.

3.2.8. Unknowns Mode

The unknowns mode is currently non-operational. That is, the unknowns rules are built into the system and are not accessible to the user. These rules essentially fill in the values of variables in the semantic elements located in the semantic3 mode. This is done by comparing the element containing variables with all the elements currently active in the PBG, i.e., the current context. Then, when a match is found the appropriate values are filled in. A variable is just a grammar class taken without its domain.

3.2.9. Origin Mode

The origin mode is currently non-operational. That is, the origin rules are built into the system; they are not accessible to the user. These rules define processing in graphic2 to be a joint man-machine effort. The goal is to hypothesize for each knowledge element in graphic1 its origin, i.e., the operator and its inputs (and the operators that produced those inputs, etc.) that produced that knowledge element as output. The system queries the user asking for all operators and inputs that could have produced the element whose origin we want. From this information the system grows an origin tree, and hypothesizes which path through the tree represents the actual origin of the element. The path is picked on the basis of the agreement between the hypothesized inputs and the actual context defined by the current PBG.

3.2.10. Conflict Mode

In the conflict mode, the define function is used to define a state vector and situation-action rules which are used to determine whether or not two particular knowledge elements conflict. The first element is always the output of a node in the pbg, the second is always an output of the group that is currently being grown onto the pbg. These rules are used by the graphic3 mode to help process graphic2 data.

Rules in the conflict mode use a state vector format.

The state vector variables currently available are:

(SAME N) : is T if the n'th arguments of both knowledge elements of the conflict pair are identical, else is F.

(ITEM n m) : is the n'th argument of the m'th element of the conflict pair.

SIMILAR : is T if the group being grown onto the pbg is similar to the node that produces the first element of the conflict pair, else is F. By similar we mean that the group inputs are a subset of the node inputs and the operators are identical.

The actions currently available are:

SIM : the two knowledge elements are considered similar.

CON : the two knowledge elements are considered conflicting.

NO-CON : the two knowledge elements are considered unconflicting.

ASK-IF-CON : the user is asked if the elements are conflicting.

For example, if the conflict rules have the form:

```
        SV = (SIMILAR (SAME 2) (ITEM 1 2))
        (T * *) => SIM
        (* F *) => NO-CON
        (* * EQ) => ASK-IF-CON
```

then if the two knowledge elements are outputs of different operators and are (EQ D 5) and (EQ R 7), the evaluated SV is (F F T) and the action taken is NO-CON. For (EQ D 5) and (EQ D 7) the SV is (F T T) and the action taken is ASK-IF-CON. For (EQ D 5) and (EQ D 5) the SV is (T T T) and the action taken is SIM. In general, if the conflict rules produce SIM or CON this leads to a restructuring of the PBG as the new node is grown.

3.2.11. PBG Mode

In the pbg mode, the define function is used to define a state vector and situation-action rules which are used to determine how a particular group should be incorporated into the pbg. These rules are used by the graphic3 mode to help process graphic2 data.

Rules in the pbg mode use a state vector format.

The state vector variables currently available are:

```
   TYPE : is SIM if a similarity is found, or CON if a
       conflict is found.
   (ITEM n m) : is the n'th item of the m'th element of
       the similarity or conflict pair.
   INCLUDE : is T if all output elements of cn1 are
       included in those of cn2.
```

Also, the prefix 'NOT-' on a state vector variable is equivalent to asking for the complement of the value of the variable.

The actions currently available are:

```
   BLOCKREJ : restructure using block rejection.
   GROW : add the group to end of pbg, no restructuring.
```

The action BLOCKREJ can have the following arguments:

```
   COPY : copy cn2 during restructuring.
   CHAIN-N : redefine cn1 to be the earliest node which
       produces an 'n' element used as input to the
       original cn1 node.
```

For example, if the pbg rules have the form:

```
          SV = (TYPE (NOT-ITEM 1 2))
          (CON EQ) => (BLOCKREJ COPY CHAINAEQ)
          (CON *) => BLOCKREJ
```

then if the two elements are (EQ D 5) and (EQ D 7), the evaluated SV is (CON (ODD EVEN AEQ)), assuming that EQ, AEQ, ODD, and EVEN are all knowledge elements defined in the space rules. The action taken would be BLOCKREJ. If the elements are (ODD R) and (EVEN R) the SV is (CON (EQ AEQ ODD)) and the action taken is (BLOCKREJ COPY CHAINAEQ).

CN1 refers to the operator which produced the first knowledge element of the similarity or conflict pair. CN2 refers to the operator which produced the second knowledge element of the similarity or conflict pair.

3.2.12. PS Mode

In the ps mode, the define function is used to define an ordered set of production rules which, taken together, form a production system. Both the ps and memory rules are used by the trace2 mode to generate a trace of the production system defined in the ps mode.

Rules in the ps mode must have the following format:

```
       s s ... s => a a ... a
 where s can be: k, -k, (k ... k m ... m), or
       -(k ... k m ... m)
       a can be: o or (o m ... m)
 and k is any problem space element,
     o is any operator
     m is any memory.
```

The left side of a rule is true (and leads to the evocation of the right side) when all k's in the left side are found to currently be in at least one of the specified memories. When the right side of a rule is evoked, the operators are executed and the result put into each of the specified memories.

The production system operators are

```
    (DEPOSIT k) : puts k into the front of memory
    (REMOVE k) : removes k from memory
    (FIRE p) : fires production system p
    (ASSIGN x v) : variable or class x is assigned value v
    (DEASSIGN x) : variable or class x is made undefined
    (REPLACE (b c d) (b1 c1 d1) n) : b c d in n'th element
      of memory is replaced with b1 c1 d1. If n is missing,
      it is assumed to be 1.
    (NOTICE k) : if found, k is moved to front of memory
    (FAIL) : the production rule execution is halted
    (STOP) : the production system execution is halted.
```

Below are examples of legal rules: (where memory isn't specified the default memory is assumed):

```
 1. (EQ D 5) (EQ T 0) => (PC 1)
 2. (EQ <L> <D>) => (FC <L>) (PC <COL>)
 3. (EQ D 5 NEW) ((EQ T 0) LTM) => (PC 1)
 4. ((EQ D 5) (EQ T 0) LTM) => ((PC 1) LTM)
 5. ((EQ D 5) STM LTM) => ((PC 1) STM LTM)
 6. (EQ D 5 (NEW UNCLEAR)) => (PC 1)
 7. (EQ X1 X2) - (ODD <L>) => ((DEPOSIT (PC 1)) STM)
 8. (* 5) => (NOTICE (DIGIT 5))
 9. (ODD X1) => (REPLACE (X1) (R))
```

Note that variables are denoted X1, X2, ... and classes are denoted by angle brackets (<>). Consider rule 5 above. if (EQ D 5) is in either STM or LTM then (PC 1) is put into both STM and LTM. The * in rule 8 above matches any string of words. Thus (* 5) matches (EQ D 5), (DIGIT 5), and (NEQ R 5).

3.2.13. Memory Mode

In the memory mode, the define function is used to define and initialize the various memories to be accessed by the productions in the production system modes (such as the ps mode). Both the memory and ps rules are used by the trace2 mode to create a trace of the production system defined in the ps mode.

Rules in the memory mode must have the following format:

```
        name = element element ... element
```

where 'name' is the name of the memory and 'element' is any problem space element. The element list defines the contents of the memory, and the memory defined by the first rule in the buffer is always made the default memory. Any number of memories may be defined and initialized, and they may have arbitrary names. For example, if the memory buffer held:

```
      1. KS = (EQ D 5)(EQ C1 0)(EQ C7 0)
      2. STM = ()
      3. LTM = (EQ D 5)
```

then the system would assume that 3 memories were available: KS with three elements in it, STM with no elements (empty), and LTM with one element in it. The default memory would be KS.

3.3. AUXILIARY MODES

3.3.1. Association Mode

The association mode contains rules which define the associations between run and rule modes, i.e., which rule modes are associated with each run mode. These associations, together with the control rules, define both the control cycle, (i.e., the order in which modes are accessed using the next function or automatic flag) and the location of the data for the rules in each rule mode to process.

In the association mode, the define function is used to define rules indicating which rule modes are associated with which run modes.

Rules in the association mode have the following format: M : M ... M
where the M's stand for run or rule mode names. The M to the left of the colon must be a run mode name, those to the right, rule mode names. For example, the rule LINGUISTIC2 : SPACE GRAMMAR indicates that the linguistic2 mode has two rule modes associated with it, space and grammar. Thus processing in linguistic2 consists of applying the space and grammar rules to the data in the previous run mode (or the run mode indicated in the control rules). The default association rules for PAS-II are shown below.

1. TEXT :
2. TOPIC : SEGMENTATION
3. LINGUISTIC1 : EXTRACTION
4. LINGUISTIC2 : SPACE GRAMMAR
5. SEMANTIC1 : INTEGRATION
6. SEMANTIC2 : NORMALIZATION
7. SEMANTIC3 : GROUPING
8. GRAPHIC1 : UNKNOWNS
9. GRAPHIC2 : ORIGIN
10. GRAPHIC3 : CONFLICT PBG
11. TRACE1 :
12. TRACE2 : PS MEMORY
13. TRACE3 :
14. TRACE4 : MATCH

Note that some run modes have no associated rule modes.

3.3.2. Control Mode

The control mode contains rules which together define the control cycle for the system.

In the control mode, the define function is used to define rules specifying where a run mode obtains its input data and what the next run mode is relative to the control cycle.

Rules in the control mode have the following format:

```
      M (M ... M) => M M
```

where M stands for any mode name. The modes in parentheses indicate where the mode at the left obtains its input for processing. The modes to the right of the arrow indicate where control passes after processing is finished in the mode at the left. For example, the rule

```
  LINGUISTIC1 (TOPIC) => LINGUISTIC2 TRACE1
```

indicates that in the linguistic1 mode data from the topic mode is processed and then the next mode in the control cycle is either linguistic2 or trace1. When processing is started in linguistic1 if topic is empty then processing cannot occur (no data to process) and the next mode entered is trace1, the second mode name to the right of the arrow in the rule above. If topic is not empty processing occurs in linguistic1, and the next mode entered is linguistic2, the first mode name to the right of the arrow.

The default control rules for PAS-II are shown below:

1. TEXT () => TOPIC TOPIC
2. TOPIC (TEXT) => LINGUISTIC1 LINGUISTIC1
3. LINGUISTIC1 (TOPIC) => LINGUISTIC2 TRACE1
4. LINGUISTIC2 (LINGUISTIC1) => SEMANTIC1 SEMANTIC1
5. SEMANTIC1 (LINGUISTIC2) => SEMANTIC2 SEMANTIC2

```
 6. SEMANTIC2 (SEMANTIC1) => SEMANTIC3 SEMANTIC3
 7. SEMANTIC3 (SEMANTIC2) => GRAPHIC1 LINGUISTIC1
 8. GRAPHIC1 (SEMANTIC3) => GRAPHIC2 GRAPHIC2
 9. GRAPHIC2 (GRAPHIC1) => GRAPHIC3 SEMANTIC3
10. GRAPHIC3 (GRAPHIC2) => GRAPHIC2 GRAPHIC2
11. TRACE1 (GRAPHIC3) => TRACE2 TRACE2
12. TRACE2 () => TRACE3 TRACE3
13. TRACE3 (TRACE2) => TRACE4 TRACE4
14. TRACE4 (TRACE3 TRACE1) => TRACE4 TRACE4
```

3.3.3. Information Mode

The information mode is totally unlike all other modes in the system. It recognizes only index words or key words (like 'INDEX' and 'PROMPT'), and cannot execute functions. In all the other modes, however, there is no knowledge of index or key words; these modes expect all tty input, except for data, to consist of functions: non-functions will elicit error messages. Currently 'HELP' is the only keyword available.

The index words and their descriptions are listed below:

DEF :	definition of a mode.
DIF :	difference between run and rule modes.
INF :	information about the information mode.
ASSOC :	run modes and associated rule modes.
BASIC :	description of basic functions.
EDIT :	description of edit functions.
FLAG :	description of flag functions.
PROCESS :	description of process functions.
EXECUTE :	how functions are executed (and abbreviated).
PROMPT :	ways the system may prompt you for input.
CR :	use of the carriage return.

3.3.4. Save Mode

In the save mode, the define function is used to define rules for specifying which mode buffers are to be saved when 'WRITE' is executed.

Rules in the save mode are simply sets of mode names. For example, line 1 in the save buffer contains (as a default condition) all the run mode names. Line 2 contains (as a default condition) all the rule mode names.

In the save mode, the read function lets you read a disk file into the run and rule mode buffers, thus establishing a context that was previously saved by using 'WRITE' in the save mode. The file being read should not have line numbers (sos line numbers are ok) unless the flag NUMBERS = T, but it may have an extension. If you type '(READ SAVE1)' then file SAVE1 is read, a different part of it going into each run and rule mode buffer. The file being read must consist of sequences of the form:

```
            mode name
            rules or data
            blank line
```

Thus a file of the form:

```
        TEXT
    L IS 3 , AND D
    IS 5 .

        TOPIC
    L IS 3 ,
    AND D IS 5 .

        SEGMENTATION
    ,/
```

would be read into the text, topic, and segmentation buffers, replacing the old information in these buffers. The information in the other buffers (all except text, topic, and segmentation) is left unchanged. The mode ordering in the file (text before topic, etc.) can be arbitrarily chosen.

In the save mode, the write function lets you write the contents of all run and rule mode buffers onto a disk file when given one argument, the name of the file to be read. This file may have an extension, if desired. If you type '(WRITE SAVE2)' then the current context (all buffers) are saved on a disk file named SAVE2.

To save certain selected buffers, give as the 2nd, 3rd, etc. arguments to the write function either mode names or numbers referring to save mode buffer lines which contain mode names. Then only the buffers named will be saved. Initially, line 1 of the save buffer contains all the run mode names, line 2 all the rule mode names. Thus to save the contents of all rule mode buffers on file SAVE3 type '(WRITE SAVE3 2)'. To save just the contents of the space and grammar mode buffers on file SAVE3 type '(WRITE SAVE3 SPACE GRAMMAR)'.

3.3.5. Scratch Mode

In the scratch mode, the define function is used to type material into the scratch buffer. This material may consist of any rules or data.

The scratch mode is used as a temporary storage buffer for any type of rules or data. In particular, the apply function puts the result of its processing into the end of this buffer. Thus the user may apply a rule to a line of data and see the result without changing the current status of the run modes.

Rules or data stored in the scratch buffer need not have any particular format.

4. FUNCTIONS

4.1. BASIC FUNCTIONS

4.1.1. Mode Name

The name of each mode in PAS-II is interpreted as a function which puts the user into that particular mode. For example, to go to the topic mode the user simply types 'TOPIC', and the system responds by typing back 'TOPIC MODE' to indicate that the user is now in that mode.

4.1.2. Core Function

The core function tells you how many words of free storage you have left in core. This free storage is used by PAS-II as temporary working storage during processing. If the number of words in free storage is small the system spends too much time garbage collecting (returning unneeded words to free storage) and the response at your terminal will be quite slow. You can increase free storage by expanding core (see PANIC: emergency procedures). When core is expanded, roughly three-fourths of all additional core is allocated as free storage.

4.1.3. Create Function

The create function lets you create a new rule mode. This mode will be of the same type as the mode you are currently in. It takes one argument, the name you wish the new mode to have. Thus, if you type '(CREATE PS5)' while in the PS mode, then a PS5 buffer is created, you are automatically put into PS5 mode, and the rules defined in the newly created mode must have the format described for rules in the PS mode. Currently create is not defined in the save, association, control, memory, text, trace2, and segmentation modes.

4.1.4. Display Function

The display function displays the data present in the mode buffer, including buffer line numbers. It takes any number of arguments, which should be either numbers or number-groups. In the grammar mode, DISPLAY can also take class names as arguments. If you type 'DISPLAY', all non-empty lines of the buffer will be displayed. If you type '(DISPLAY 6)', then only line 6 will be displayed. '(DISPLAY 1 3 10-12)' gives you lines 1, 3, 10, 11, and 12.

4.1.5. Erase Function

The erase function lets you erase (uncreate) any mode you have created using the create function. It takes one argument, the name of the mode to be erased. If you erase the mode you are currently in, you are put into the text mode.

4.1.6. Exit Function

The exit function takes you out of PAS-II and leaves you talking to the Lisp interpreter. If you're not familiar with Lisp, don't use EXIT. To return to PAS-II after using EXIT type '(BEGIN)'.

4.1.7. Help Function

The legal arguments for the help function are:

(function name) : tells how to use the function in the current mode.
(mode name) : gives information about the mode.
(index word) : gives general information about PAS-II.
INDEX : describes and lists the index words.
HELP : describes the legal arguments for the help function.
ARGS : lists the legal arguments for the help function.
FORMAT : gives data or rule formats for the current mode.
FUNCTIONS : lists all functions.
MODES : lists all run modes and associated rule modes.
NUMBER : describes numbers and number-groups.
STATE-VECTOR : describes state vector rules and their operation.
CHANGES : describes all the recent changes to the PAS-II system.
COMMENTS : describes the use of comments in buffer lines.
PROTOCOL : gives general info about automatic protocol analysis.
PANIC : describes emergency procedures.
LOAD : tells how to load and run PAS-II.
OLD : tells how to run earlier versions of PAS-II.
AUX : describes the auxiliary modes.
REF : lists protocol analysis referenes.
PAS : gives a general description of PAS-II operation.
DEM : tells how to obtain a PAS-II demonstration listing.
CN : defines the terms cn1 and cn2.
OK? : describes the responses that should be used to 'ok?'.

4.1.8. Mode Function

The mode function tells you what mode you are currently in.

4.1.9. Move Function

The move function moves lines of data or rules from one buffer to another. It has the general form:

```
    (MOVE lines1 mode1 TO lines2 mode2)
```

where lines1 and mode1 are the source lines and mode, and lines2 and mode2 are the destination. The move function must be given at least one argument. It fills in missing arguments according to the default values shown below.

ARGUMENT	DEFAULT VALUE
lines1	all lines in mode1 buffer
mode1	current mode
lines2	empty lines at end of mode2 buffer
mode2	current mode

The number of lines specified by the argument lines1 must be equal to the number specified by lines2. Move deletes the source lines from the buffer and redefines the destination lines.

Examples of the use of the move function are shown below. We will assume that the current mode is TEXT.

```
 (MOVE 1-3 SCRATCH TO 9 10 15 TOPIC) : 1, 2, and 3 in scratch
            are moved to 9, 10, and 15 in topic.
 (MOVE 6 10 TO 8-9 SCRATCH) : 6 and 10 in text are moved to
            8 and 9 in scratch.
 (MOVE 4 TO SCRATCH) : 4 in text is moved to end of scratch.
 (MOVE 9 11 SPACE) : 9,11 in space are moved to end of text.
 (MOVE TO SCRATCH) : all text is moved to end of scratch.
 (MOVE 4 TO 12) : 4 in text is moved to 12 in text.
 (MOVE 6) : 6 in text is moved to end of text.
 (MOVE SCRATCH) : all scratch moved to end of text.
```

Note that (MOVE) or (MOVE TO) are not legal function calls.

4.1.10. Next Function

The next function changes the current mode to the next appropriate run mode as defined by the control cycle. Thus you can go to the next appropriate run mode and process data there by typing 'NEXT GO'. 'NEXT' changes the mode, 'GO' starts the processing.

4.1.11. Prior Function

The prior function changes the current run mode to the previous run mode as defined by the control cycle. This previous or prior run mode is the one that supplies data for the current mode to process.

4.1.12. Rule Function

The rule function is used to go from a run mode to the rule mode associated with that run mode. If you type 'RULE' while in the topic mode, you will be put into the segmentation mode.

4.1.13. Run Function

The run function is used to go from a rule mode to the run mode associated with that rule mode. If you type 'RUN' while in the segmentation mode, you will be put into the topic mode.

4.2. EDIT FUNCTIONS

4.2.1. Break Function

The break function breaks a line in the mode buffer into two or more shorter lines. If you type '(BREAK 4 . IS (6 ,))', line 4 will be broken after the first occurrence of '.', after the next occurrence of 'IS', and after the next occurrence of the string '6 ,'. the old line 4 will be replaced by lines 4.1, 4.2, 4.3, and 4.4.

If you are using a tty, you can type '(BREAK 4)'. The system will prompt you with '4.' and will then expect to read either altmode (to print the next word in the line), line feed (to break the line after the last word printed), space (to back up in the list of words printed), or carriage return (to break the line after the last word printed and leave the break function).

4.2.2. Connect Function

The connect function connects together two or more lines in the mode buffer. It takes two or more arguments, which should be numbers or number-groups. If you type '(CONNECT 2 3 4)', lines 2, 3, and 4 will be joined together and redefined as line 2; lines 3 and 4 will be deleted.

In the graphic3 mode, the connect function is used to connect two pbg nodes together, thus effecting a restructuring. It takes two numbers as arguments, each of which should represent an active node in the pbg. If you type '(CONNECT 1 5)' then node 5 will be made to point back to node 1 in the pbg.

4.2.3. Define Function

The define function permits you to enter data into the mode buffer. It takes any number of arguments, which should be either numbers or number-groups. If you type 'DEFINE', you will enter data into the buffer starting with the line after the last nonempty line. '(DEFINE 6 9)' will enter data into line 6, destroying old data, if any, and then after the first carriage return will enter data into line 9. Enter data after the system prompts you with the line number. To stop entering data, type a carriage return immediately after the system prompts you with the line number.

4.2.4. Delete Function

The delete function deletes data present in the mode buffer. It takes one or more arguments, which should be either numbers or number-groups. For example, if you type '(DELETE 6 8 10-12)', then buffer lines 6, 8, 10, 11, and 12 will be deleted.

4.2.5. ED Function

To edit the current buffer type 'ED', to edit a particular line (say line 4) type '(ED 4)'. Then type 'GET' to load the buffer data into Alvine, the Lisp editor. Do the editing, and then type 'PUT' to put the revised data back into the buffer. Then type ↑ to leave the editor. Warning: do not attempt to use ED unless you are familiar with the Lisp editor!!! (see the Stanford Lisp 1.6 manual for a description of Alvine).

4.2.6. Insert Function

The insert function inserts data after a line in the mode buffer. It takes any number of arguments, which should be either numbers or number-groups. For example, if you type '(INSERT 6)' and both lines 6 and 7 are empty then the command is exactly like typing '(DEFINE 6)'. If line 6 is full but 7 is empty it is like typing '(DEFINE 7). If both 6 and 7 are full it is like typing '(DEFINE 6.5)'.

4.2.7. Read Function

The read function lets you read a disk file into the mode buffer. It takes any number of arguments, which should be names of current disk files. The file being read should not have line numbers (although sos line numbers are OK) unless the flag NUMBERS = T, and the first word of the first line must be the name of the mode buffer the file is being read into. The rules or data should start on the second line of the file. If you type '(READ FILE1)' then the contents of FILE1 on the disk will be read into the buffer, starting with the line immediately after the last non-empty buffer line. Note that read in a run or rule mode does not delete or write over the original contents of the buffers, while read in the save mode zeros each mode buffer being read into before actually reading from the disk file.

In the save mode, the read function lets you read a disk file into the run and rule mode buffers, thus establishing a context that was previously saved by using 'WRITE' in the save mode. The file being read should not have line numbers (sos line numbers are ok) unless the flag NUMBERS = T, but it may have an extension. If you type '(READ SAVE1)' then file SAVE1 is read, a different part of it going into each run and rule mode buffer. The file being read must consist of sequences of the form:

```
          mode name
          rules or data
          blank line
```

Thus a file of the form:

```
       TEXT
    L IS 3 , AND D
    IS 5 .

       TOPIC
    L IS 3 ,
    AND D IS 5 .

       SEGMENTATION
    ,/
```

would be read into the text, topic, and segmentation buffers, replacing the old information in these buffers. The information in the other buffers (all except text, topic, and segmentation) is left unchanged. The mode ordering in the file (text before topic, etc.) can be arbitrarily chosen.

4.2.8. Renumber Function

The renumber function lets you renumber all lines in the mode buffer, starting with 1 in increments of 1. It takes no arguments. If the buffer contains lines 4, 7, 8.6, and 12 and you type 'RENUMBER', the buffer will then contain lines 1, 2, 3, and 4.

4.2.9. Write Function

The write function lets you write everything in the mode buffer onto a disk file. It takes one argument, the name of the file to be read. If you type '(WRITE FILE2)' everything in the buffer will be written on a disk file named FILE2.

In the save mode, the write function lets you write the contents of all run and rule mode buffers onto a disk file when given one argument, the name of the file to be read. This file may have an extension, if desired. If you type '(WRITE SAVE2)' then the current context (all buffers) are saved on a disk file named SAVE2.

To save certain selected buffers, give as the 2nd, 3rd, etc. arguments to the write function either mode names or numbers referring to save mode buffer lines which contain mode names. Then only the buffers named will be saved. Initially, line 1 of the save buffer contains all the run mode names, line 2 all the rule mode names. Thus to save the contents of all rule mode buffers on file SAVE3 type '(WRITE SAVE3 2)'. To save just the contents of the space and grammar mode buffers on file SAVE3 type '(WRITE SAVE3 SPACE GRAMMAR)'.

4.3. FLAG FUNCTIONS

A flag is a function which sets the value of the flag variable to either T (true) or F (false). The flags currently available are: automatic, batch, comment, fast, hush, numbers, print, search, suppress, time, version1, and version2. Thus the flag hush can be set to T by typing 'HUSH', or '(HUSH T)'. It can be set to F by typing '(HUSH F)'. To determine the current value of any flag give it the argument '?'. Thus (HUSH ?) will give you the current value of the hush flag.

4.3.1. Automatic Flag

If automatic = T then during mode processing the system will automatically change to the next appropriate run mode and execute 'GO' whenever the current processing is completed. If automatic = F (the default condition) this will not happen.

4.3.2. Batch Flag

If batch = T then during mode processing the system will not ask the user to confirm each step of the processing. If batch = F (the default condition) the user will be asked to confirm each step.

4.3.3. Comment Flag

If comment = T (the default condition) then when buffers are displayed all comments will be printed. If comment = F the comments will be suppressed.

Comments may be added to the end of any buffer line without affecting the operation of the system. A comment is defined to be everything between the comment marker (initially a double colon, ::) and the end of the buffer line. Lines containing nothing but comments are allowed.

The comment flag may be used to merely insert comments into your console display, since the flag isn't changed unless the first argument is either T or F. Thus typing

```
        (COMMENT : THIS IS A COMMENT)
```

records this line on your terminal without changing the value of the comment flag.

To change the comment marker type 'EXIT', followed by the string '(SETQ CHAR! @x)' where x is the new comment marker. Then type '(BEGIN)' to return to PAS-II. The marker can be any string that starts with a letter and is followed by letters or digits; where the following are considered letters: * = > < ; : % ? , . - + / @ Note: the last six characters above must be preceded by a slash (/) when setting CHAR!

4.3.4. Fast Flag

If FAST = T then when files are read from the disk no format checking takes place, speeding up the reading process. If FAST = F (the default condition) format checking takes place and reading is rather slow.

4.3.5. Hush Flag

If hush = T then error messages are abbreviated, if hush = F (the default condition) they are not abbreviated.

4.3.6. Numbers Flag

If numbers = T, then when a file is written on the disk the buffer line numbers will be included as part of the file. When a file is read from the disk, the system assumes that all lines begin with a buffer line number. If numbers = F (the default condition), files are written without line numbers, and are read under the assumption that no lines begin with line numbers.

4.3.7. Print Flag

If print = T, then all the interaction at your terminal will be written on a disk file. When print is first set to T the system will ask for the file name. If the file already exists, its old contents will be lost. If print = F (the default condition) the interaction will not be saved. Note 1: the file is not saved on the disk until print is set back to F. Do not forget to set print back to F before you leave PAS-II! Note 2: do not execute the write function while print = T!

4.3.8. Search Flag

The value of the search flag affects the processing in semantic1 and semantic 2 as follows: the integration (or normalization) rules will be applied to the data starting with line 1 until no further applications are possible. Then the rules are applied starting with line 2 until no further applications are possible. This continues until the rules have all been applied to the last line of data. Now if search = F (the default condition) processing stops. However, if search = T the processing starts over again from line 1 and this cycle continues until no rules are applicable to any lines of data.

4.3.9. Suppress Flag

In the linguistic2 mode, if suppress = T then during processing the parse tree is not printed. If suppress = F (the default condition) the tree is printed.

In the semantic1, semantic2, and graphic3 modes, if suppress = T then during processing the state vector values and the s-a rules matched are not printed. If suppress = F (the default condition) they are printed.

4.3.10. Time Flag

If time = T then during mode processing timing information is printed. If time = F (the default condition) it is not printed.

4.3.11. Version1 and Version2 Flags

If version1 = T (the default condition) then the system expects to find the old type of formats used in the grammar and space rules, and the old parser is used for processing data during parsing in the linguistic2 mode. If version2 = T, the system expects the new grammar and space formats and uses the new parsing system when parsing in the linguistic2 mode. Note: when version1 is set to T, version2 is automatically set to F and vice versa. Thus the flags version1 and version2 never have the same value at the same time.

4.4. PROCESS FUNCTIONS

4.4.1. Again Function

The again function takes the data out of the current run mode buffer, puts it into the previous run mode buffer (deleting any data left there), and executes go.

4.4.2. Apply Function

The apply function applies rules from the current rule mode to lines of data in the appropriate run mode and places the result at the end of the scratch buffer. The 'appropriate' run mode is the one defined by the control cycle to provide input to the run mode associated with the current rule mode. Thus calling apply in the space mode applies space rules to the data in linguistic1 (see control cycle layout). The function has the general form:

```
        (APPLY lines1 TO lines2)
```

where lines1 are the lines of rules from the current mode, and lines2 are the lines of data from the appropriate run mode. Missing arguments are given the default values shown below:

ARGUMENT	DEFAULT VALUE
lines1	all lines in the current rule mode
lines2	all lines in appropriate run mode

Any number of lines may be specified as arguments to the apply function.

Examples of the use of apply are shown below. It is assumed that the current mode is SPACE.

```
 (APPLY 1-8 TO 6) : applies rules 1 through 8 in space to
        line 6 in linguistic1 and puts result in scratch.
 (APPLY TO 4 10) : applies all rules in space to lines 4
        and 10 in linguistic1 and puts result in scratch.
 (APPLY 8) : applies rule 8 in space to all lines in
        linguistic1 and puts result in scratch.
 (APPLY) : applies all rules in space to all lines in
        linguistic1 and puts result in scratch.
```

Note that apply does not affect the contents of any run or rule modes. It only modifies scratch, an auxiliary mode.

4.4.3. Copy, Go, Recopy, Restart, and Start Functions

The go function takes data from the previous run mode, processes it, and adds the result to the end of the current mode buffer. The data is deleted from the previous mode buffer, but a copy is made.

The start function deletes all lines in the current mode buffer and then executes 'GO'.

The restart function puts the copy of the data deleted from the previous mode (when GO was executed) back into the front of the previous mode's buffer and then executes 'START'.

The copy function simply prints at your terminal the lines which comprise the copy of the data in the current mode.

The recopy function puts the current copy of the data back into the front of the mode buffer, but it does not delete the data in the following mode. Thus executing 'RECOPY NEXT GO' is not equivalent to executing 'NEXT RESTART' since in the latter case the data in the following mode is deleted. For example, if you are in the topic mode and execute 'GO' the data in the text mode is automatically deleted as it is processed. To restore that data to the text mode, enter the text mode and execute 'RECOPY'.

5. EVOLUTION

5.1. CHANGES TO PAS-II

Changes to PAS-II are listed below:

Version 29 June 17, 1973

1. Apply is a new process function for applying a subset of the rules in a mode to some subset of the associated data.
2. Scratch is a new auxiliary mode, used as the destination mode for the apply function.
3. Move is a new basic function for moving rules or data from one mode to another (or within a mode).
4. Prior is a new basic function that puts the user into the prior mode as defined in the control cycle.
5. Core is a new basic function that indicates how much free storage is left.
6. It is now possible to write when PRINT = T.
7. Semantic1 and semantic2 now have an expanded trace when SUPPRESS = F.
8. The status of flags can be obtained by using '?' as the argument to the flag function.
9. The comment flag can now be used to record comments on your console output.

Version 28 April 30, 1973

1. Trace modes were renamed: trace1, trace2, trace3, trace4.
2. Match is now the rule mode for trace4.
3. Search is now flag. If search=T processing using integration and normalization rules become more exhaustive.
4. Again is a new control function which reprocesses the data just processed.
5. Association is a new mode that defines the associations between run and rule modes.
6. Control is a new mode that defines the control cycle.
7. The create function has been generalized to permit creation of different types of modes.
8. The erase function has been generalized.

Version 27 December 22, 1972

1. Buffer lines containing nothing but comments are now allowed.
2. Numbers is a new flag. if numbers=T, disk files with bufferline numbers can be created.
3. Version1 and version2 are new flags. if version1=T, the system uses the old parser (and space and grammar

formats). If version2=T the new, improved parser (and formats) are used.
4. Graphic4, trace1, trace2, and match are new run modes.

Version 26 September 10, 1972

1. '(HELP DEM)' tells you how to get a PAS-II demonstration listing.
2. Print is a new flag. If print=T all tty interactions are written on a disk file.
3. Create and erase are new functions which create a new production system mode or erase it.
4. PS is a new rule mode which holds production rules which together define a production system.
5. Memory is a new rule mode which holds rules defining the memories used by the production system modes and their current contents.
6. Trace1 is a new run mode associated with the production system and memory modes. (not yet operational)

Version 25 August 15, 1972

1. PAS-II starts from the text rather than the information mode.
2. Comments may now be added to buffer lines.
3. '(HELP PAS)' tells how to run old versions of PAS-II.
4. Comment is a new flag. If comment=F, comments are

Version 24 August 1, 1972

1. Parentheses are no longer needed in function calls, except for disambiguation.
2. Two new control functions: copy and recopy have been added.
3. All index words are now legal arguments for the help function.
4. The editing functions now interpret an asterisk (*) in the argument list as refering to the last item in the buffer.
5. ED is a new function which lets you use the Lisp editor (Alvine) to modify a buffer.

Version 23 July 21, 1972

1. Save mode completely revised: will now accept rules in save buffers; can read or write partial contexts.
2. Read function now expects a different format on disk files.
3. File names with extensions are now allowed.
4. Fast is a new flag; if fast=T, reading disk files is faster.

5.2. OLD VERSIONS OF PAS-II

Version n of PAS-II can be found on dec-tape DW28-PASn. Thus, version 27 is on dec-tape DW28-PAS27, etc. Below is a script showing how to load old versions of PAS-II.

```
   .R LISP 65
   ALLOC? Y
   FULL WDS=
   BIN.PROG.SP=
   SPEC.PDL=3000
   REG.PDL=3000
   HASH=
   AUXILIARY FILES?
   DECIMAL?Y
   *(INC (INPUT DTA1: PAS27))
   *(PAS-I)
   INITIALIZED
   *(BEGIN)
```

To obtain help information for version 27 of PAS-II (or older versions) the following steps are needed:

```
   *(INC (INPUT DSK: INFO7))
   *(INFO-I)
   INITIALIZED
```

before executing (BEGIN). Also, Lisp should be loaded with 80K of core.

6. EMERGENCY PROCEDURES

When using PAS-II it can be useful to know the proper action to take in case of system trouble, whether it be a PDP-10, LISP, or PAS problem. Listed below are emergency procedure recommendations:

1. If you get a LISP error message such as X UNBOUND VARIABLE, or X UNDEFINED FUNCTION then do:
   ```
           *(PAS)
   ```
   to return to the PAS-II system.

2. If you get the LISP error message NO FREE STORAGE LEFT, or NO FW STORAGE LEFT then do:
   ```
           *(control)C
           .CORE n
           .REE
           *(PAS)
   ```
   This will change your core allocation from 65K to nK, where n should be a number such as 75 or 80.

3. If PAS does not come back with a prompt(*) after a reasonable length of time then do:
   ```
           (control)C
           .REE
           *(PAS)
   ```
   This will return you to PAS.

4. If you have a large investment (time or effort) in rules, data, or processing then do:
   ```
           *(control)C
           .SAVE DSK: filename
           .REE
           *(PAS)
   ```
   This saves your core image on the disk as file 'filename'.

5. If the system crashes and you have previously done a SAVE DSK: filename, then do:
   ```
           .RUN filename
           *(PAS)
   ```
   This restores your old core image and returns you to PAS.

6. If PAS continually gives LISP error messages (or I FEEL SICK messages) then do:
   ```
           *(PAS)
           *SAVE WRITE filename
           *(control)C
           .R PAS
           *(PAS)
           *SAVE READ filename
   ```

This will save the contents of the PAS buffers, read in a fresh version of PAS, and restore the buffers.

7. REFERENCES

1. Newell, A., and Simon, H. A., Human Problem Solving, Prentice-Hall, Englewood Cliffs, N. J., 1971.
2. Waterman, D. A., and Newell, A., An interactive, task-free version of an automatic protocol analysis system, CMU Computer Science Report, 1973.
3. Waterman, D. A., and Newell, A., Protocol analysis as a task for artificial intelligence. Artificial Intelligence, vol. 2, nos. 2 and 3, 1971, pp. 285-318.
4. Waterman, D. A., and Newell, A., Preliminary results with a system for automatic protocol analysis. CIP Paper no. 211, May, 1972.

ACKNOWLEDGMENTS

The contribution of Mike Rychener, both in developing the improved parser and describing it in this manual is gratefully acknowledged. Also, the help of Al Newell and Dick Hayes in revising this manual is much appreciated.

8. APPENDIX

8.1. STATE VECTORS

Rules using a state vector format must have a state vector definition in the first line of the buffer and situation-action rules (s-a rules) in the rest of the buffer. The state vector definition must have the form:

```
        SV = (var var ... var)
```

where var refers to state vector variables which have a value defined by the current situation, i.e., the data stored in the run mode associated with this particular rule mode.

The s-a rules must have the form:

```
        (val val ... val) => (action)
```

where val is a legal value of the corresponding sv variable in the current situation. To decide what action to take, the sv variables in the first line of the buffer are evaluated and these values matched against the values in the situation part of the s-a rules below. The first match determines the action to be taken. A ☆ can be used in the situation part of the s-a rules to indicate A match with any sv value.

8.2. PAS-II RULES

The following rules are designed to handle the very simple cryptarithmetic protocol shown in the text mode below. These rules were used in PAS-II to generate a script of a PAS-II demonstration.

```
             TEXT MODE
   1. D IS 5 ; THEREFORE T IS 0 . ASSUME R EQUALS 4 . SINCE YOU
      CARRY 1 , R IS ODD . ASSUME R IS 7 , NOT 5 .

             SPACE RULES
   1. (NEG) IND
   2. (ODD <V>) KN
   3. (EQ <V> <DIGIT>) KN
   4. (THEREFORE) IND
   5. (BECAUSE) IND
   6. (ASSUME) IND
   7. (DIGIT <DIGIT>) KN
   8. (<V> <LETTER> <CARRY>) SPASET

             GRAMMAR RULES
   1. <EQ> = (<CARRYEQ>) (<LETTER> ☆ <EQUAL> ☆ <DIGIT>)
   2. <CARRYEQ> = (<CARRY> ☆ <DIGIT>) (<CARRY>)
   3. <ODD> = (<LETTER> ☆ <EQUAL> ☆ ODD)
   4. <EQUAL> = IS EQUAL EQUALS BE WAS ARE
```

```
     5. <NEG> = CANNOT NOT NO N'T
     6. <THEREFORE> = THEREFORE IMPLIES
     7. <ASSUME> = ASSUME ASSUMING
     8. <BECAUSE> = BECAUSE SINCE
     9. <CARRY> = CARRY CARRYING CARRIED
    10. <LETTER> = A B D E G L N O R T
    11. <DIGIT> = 0 1 2 3 4 5 6 7 8 9

                SEGMENTATION RULES
     1. . /
     2. ; /
     3. <DIGIT> , /
     4. <LETTER> , /

                EXTRACTION RULES
     1. 12

                INTEGRATION RULES
     1. (X1 CARRY X2)  =>  (X1 <C> X2)
     2. (EQ X1 X2) / (DIGIT X3)  =>  (EQ X1 X2) / (EQ X1 X3)
     3. (NEG) (EQ <LETTER> <DIGIT>)  =>  (NEQ <LETTER> <DIGIT>)
     4. (ASSUME) (EQ <LETTER> <DIGIT>)  =>  (AEQ <LETTER> <DIGIT>)

                NORMALIZATION RULES
     1. A1 / (THEREFORE) A2  =>  (BECAUSEOF A1 A2)
     2. (BECAUSE) A1 / A2  =>  (BECAUSEOF A1 A2)

                CONFLICT RULES
     1. SV = ((SAME 2) (ITEM 1 1) (ITEM 1 2))
     2. (F * *) => NO-CON
     3. (* ODD NEQ) => NO-CON
     4. (* * *) => ASK-IF-CON

                PBG RULES
     1. SV = (TYPE (ITEM 1 2))
     2. (CON NEQ) => BLOCKREJ
     3. (CON *) => (BLOCKREJ COPY)
     4. (* *) => BLOCKREJ
```

8.3. PAS-II SCRIPT

A listing of the PAS-II demonstration given at the cognitive workshop on June 21, 1972 is on disk file PAS2.DEM[X320DW28]. This demonstration consists of using PAS-II to analyze a short crypt-arithmetic protocol, using the rules in disk file RULES.CA[X320DW28].

Below is a relatively short script of a PAS-II demonstration using the rules and cryptarithmetic protocol listed in section 7.1. of the

PAS-II reference manual. Items in upper case were typed by the PAS-II system, those in lower case by the user.

```
*text display
TEXT MODE
     1. D IS 5 ; THEREFORE T IS 0 . ASSUME R EQUALS 4 . SINCE YOU
        CARRY 1 , R IS ODD . ASSUME R IS 7 , NOT 5 .
*next go
TOPIC MODE
1. D IS 5 ;
   THEREFORE T IS 0 .
   ASSUME R EQUALS 4 .
   SINCE YOU CARRY 1 ,
   R IS ODD .
   ASSUME R IS 7 ,
   NOT 5 .
OK? yes
TOPIC MODE FINISHED
*next go
LINGUISTIC1 MODE
   1. D IS 5 ;
   2. THEREFORE T IS 0 .
   3. ASSUME R EQUALS 4 .
   4. SINCE YOU CARRY 1 ,
   5. R IS ODD .
   6. ASSUME R IS 7 ,
   7. NOT 5 .
OK? yes
*next go
LINGUISTIC2 MODE
<EQ>  <LETTER>  D
      <EQUAL>  IS
      <DIGIT>  5
   1. (EQ D 5)
      FROM :   D IS 5 ;
OK? yes batch suppress
BATCH=T
SUPPRESS=T
   2. (EQ T 0) (THEREFORE)
      FROM :   THEREFORE T IS 0 .
   3. (EQ R 4) (ASSUME)
      FROM :   ASSUME R EQUALS 4 .
   4. (EQ CARRY 1) (BECAUSE)
      FROM :   SINCE YOU CARRY 1 ,
   5. (ODD R)
      FROM :   R IS ODD .
   6. (EQ R 7) (ASSUME)
      FROM :   ASSUME R IS 7 ,
   7. (NEG) (DIGIT 5)
      FROM :   NOT 5 .
```

```
LINGUISTIC2 MODE FINISHED
*(batch f)(suppress f) automatic
BATCH=F
SUPPRESS=F
AUTOMATIC=T
*next go
SEMANTIC1 MODE
RULES APPLIED :   4  1  2  4  3
   1. (EQ D 5)
   2. (EQ T 0) (THEREFORE)
   3. (AEQ R 4)
   4. (BECAUSE) (EQ <C> 1)
   5. (ODD R)
   6. (AEQ R 7)
   7. (NEQ R 5)
OK? yes
SEMANTIC1 MODE FINISHED
SEMANTIC2 MODE
RULES APPLIED :   1  2
1-7. (BECAUSEOF ((EQ D 5)) ((EQ T 0)))
     (AEQ R 4)
     (BECAUSEOF ((EQ <C> 1)) ((ODD R)))
     (AEQ R 7)
     (NEQ R 5)
OK? yes
SEMANTIC2 MODE FINISHED
SEMANTIC3 MODE
   1. (BECAUSEOF ((EQ D 5)) ((EQ T 0)))
   2. (AEQ R 4)
   3. (BECAUSEOF ((EQ <C> 1)) ((ODD R)))
   4. (AEQ R 7)
   5. (NEQ R 5)
OK? yes
GRAPHIC1 MODE
   1. (BECAUSEOF ((EQ D 5)) ((EQ T 0)))
      FROM :   (BECAUSEOF ((EQ D 5)) ((EQ T 0)))
OK? yes
   2. (AEQ R 4)
      FROM :   (AEQ R 4)
OK? yes
   3. (BECAUSEOF ((EQ <C> 1)) ((ODD R)))
      FROM :   (BECAUSEOF ((EQ <C> 1)) ((ODD R)))
OK? yes batch suppress r: (becauseof ((eq c2 1))((odd r)))
BATCH=T
DO YOU REALLY WANT BOTH AUTOMATIC=T AND BATCH=T ? yes
SUPPRESS=T
   4. (AEQ R 7)
      FROM :   (AEQ R 7)
   5. (NEQ R 5)
      FROM :   (NEQ R 5)
```

```
GRAPHIC1 MODE FINISHED
GRAPHIC2 MODE
FOR (BECAUSEOF ((EQ D 5)) ((EQ T 0))) :
OP = (pc 1)
OUTPUTS = (eq t 0)
INPUTS = (eq d 5)(eq c1 0)
FOR (EQ D 5) :
OP = (recall d)
INPUTS =
OTHER ORIGINS FOR (EQ D 5) ? no
FOR (EQ C1 0) :
OP = (recall c1)
INPUTS =
OTHER ORIGINS FOR (EQ C1 0) ? no
ORIGIN TREE :
(EQ T 0)  (PC 1)  (EQ D 5)  (RECALL D)
                  (EQ C1 0 *)  (RECALL C1)
1. (RECALL D) NIL (EQ D 5)
   (RECALL C1) NIL (EQ C1 0)
   (PC 1) ((EQ D 5) (EQ C1 0)) (EQ T 0)
      FROM :   (BECAUSEOF ((EQ D 5)) ((EQ T 0)))
GRAPHIC3 MODE
   1. GROW (EQ D 5)
      FROM :   (RECALL D) NIL (EQ D 5)
   2. GROW (EQ C1 0)
      FROM :   (RECALL C1) NIL (EQ C1 0)
   3. GROW (EQ T 0)
      FROM :   (PC 1) ((EQ D 5) (EQ C1 0)) (EQ T 0)
GRAPHIC3 MODE FINISHED
GRAPHIC2 MODE
FOR (AEQ R 4) :
OP = (av r)
INPUTS =
OTHER ORIGINS FOR (AEQ R 4) ? no
ORIGIN TREE :
(AEQ R 4)  (AV R)
   2. (AV R) NIL (AEQ R 4)
      FROM :   (AEQ R 4)
GRAPHIC3 MODE
   1. GROW (AEQ R 4)
      FROM :   (AV R) NIL (AEQ R 4)
GRAPHIC3 MODE FINISHED
GRAPHIC2 MODE
FOR (BECAUSEOF ((EQ C2 1)) ((ODD R))) :
OP = (pc 2)
OUTPUTS = (odd r)
INPUTS = (eq c2 1)
FOR (EQ C2 1) :
OP = (av c2)
INPUTS =
OTHER ORIGINS FOR (EQ C2 1) ? yes
FOR (EQ C2 1) :
OP = (pc 1)
INPUTS = (eq d 5)(eq c1 0)
(EQ D 5) FOUND IN PBG
(EQ C1 0) FOUND IN PBG
OTHER ORIGINS FOR (EQ C2 1) ? no
ORIGIN TREE :
(ODD R)  (PC 2)  (EQ C2 1)  (AV C2)
                            (PC 1)  (EQ D 5)
                                    (EQ C1 0)
3. (PC 1) ((EQ D 5) (EQ C1 0)) (EQ C2 1)
   (PC 2) ((EQ C2 1)) (ODD R)
      FROM :   (BECAUSEOF ((EQ C2 1)) ((ODD R)))
GRAPHIC3 MODE
   1. GROW (EQ C2 1)
      FROM :   (PC 1) ((EQ D 5) (EQ C1 0)) (EQ C2 1)
DO (AEQ R 4) AND (ODD R) CONFLICT ? yes
   2. CONFLICT: N4 (AEQ R 4) AND (ODD R) WITH
      (BLOCKREJ COPY CHAINAEQ)
      FROM :   (PC 2) ((EQ C2 1)) (ODD R)
GRAPHIC3 MODE FINISHED
GRAPHIC2 MODE
FOR (AEQ R 7) :
OP = (av r)
INPUTS =
OTHER ORIGINS FOR (AEQ R 7) ? no
ORIGIN TREE :
(AEQ R 7)  (AV R)
   4. (AV R) NIL (AEQ R 7)
      FROM :   (AEQ R 7)
GRAPHIC3 MODE
DO (ODD R) AND (AEQ R 7) CONFLICT ? no
   1. GROW (AEQ R 7)
      FROM :   (AV R) NIL (AEQ R 7)
GRAPHIC3 MODE FINISHED
GRAPHIC2 MODE
FOR (NEQ R 5) :
OP = (td r 5)
INPUTS = (eq d 5)
(EQ D 5) FOUND IN PBG
OTHER ORIGINS FOR (NEQ R 5) ? no
ORIGIN TREE :
(NEQ R 5)  (TD R 5)  (EQ D 5)
   5. (TD R 5) ((EQ D 5)) (NEQ R 5)
      FROM :   (NEQ R 5)
GRAPHIC3 MODE
DO (AEQ R 7) AND (NEQ R 5) CONFLICT ? no
   1. GROW (NEQ R 5)
      FROM :   (TD R 5) ((EQ D 5)) (NEQ R 5)
```

```
GRAPHIC3 MODE FINISHED
GRAPHIC2 MODE
GRAPHIC2 MODE FINISHED
SEMANTIC3 MODE
SEMANTIC3 MODE FINISHED
LINGUISTIC1 MODE
LINGUISTIC1 MODE FINISHED
PAS-2 FINISHED
*graphic3 display
GRAPHIC3 MODE
     N1       0   OP  (RECALL D)   OUT  (EQ D 5)
     N2           OP  (RECALL C1)   OUT  (EQ C1 0)
     N3           OP  (PC 1)   IN  (EQ D 5)  (EQ C1 0)   OUT  (EQ T 0)
     N4           OP  (AV R)   OUT  (AEQ R 4)
     N5           OP  (PC 1)   IN  (EQ D 5)  (EQ C1 0)   OUT  (EQ C2 1)
     N6           OP  (PC 2)   IN  (EQ C2 1)   OUT  (ODD R)     0
     N7       3   OP  (PC 1)   IN  (EQ D 5)  (EQ C1 0)   OUT  (EQ C2 1)
     N8           OP  (PC 2)   IN  (EQ C2 1)   OUT  (ODD R)
     N9           OP  (AV R)   OUT  (AEQ R 7)
    N10           OP  (TD R 5)   IN  (EQ D 5)   OUT  (NEQ R 5)
```