A LISP Production System Facility

by

Ruven Brooks

Technical Report #98

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

A LISP Production System Facility

Ruven Brooks

This document describes a facility for programming in production systems. The facility is implemented in UCI LISP and the features available in it are based on a subset of the options available in the PSG system (Newell, 1974).

System Version 1.1

Documentation Updated February 12, 1977

Introduction

This manual is written with the assumption that users know LISP and are somewhat familiar with production systems. If this is not the case, then it is suggested that they learn LISP by first reading A LISP Primer by Carl Weissman and then looking through the Standford LISP 1.5 Manual and the UCI LISP Manual to become familiar with the details of the particular LISP system used here. Unfortunately, there is, as yet, no good place to learn about production systems.

Matching and the Pattern Language

A production system, in the conventional use in theory of automata, consists of a string chosen from an alphabet of symbols and a set of re-write rules consisting of a left half and a right half. When the symbols in the string match the left half of some rule, then they are replaced by the symbols in the right half. The system presented here differs from this usage in three major respects:

1. The symbols appearing in the string are list structures of arbitrary complexity rather than only atomic symbols. For this reason, the term, STM, (taken from psychology and standing for Short Term Memory) is used to refer to this string.

2. The symbols appearing on the left-hand, conditions side of a rule are not matched identically against the contents of STM, but are, instead, considered to be patterns stated in a pattern language described later in this document. This pattern language permits wild-carding with local variable assignment and one-level logical combination of disjunction and negation conditions.

3. The right hand side of each rule is considered to be a set of actions to be carried out, rather than just a set of replacements for items in the string. These actions may be of arbitrary complexity and may affect other structures than the STM.

Matching Procedure

In the system described here, the productions are assumed to be arranged in an ordered list. An attempt is made to match the conditions of each of the productions against the contents of the STM. If the match succeeds, then the associated actions are carried out. If failure occurs, then the matching procedure is applied to the next production in the list until either a production is found which does match or until all the productions have failed. After one production has been fired off, the search for the

next production to fire begins again at the beginning of the list.

Within the conditions for a given production, matching proceeds from left to right. Matches are made by taking each pattern and comparing it one by one with each of the items in the STM. As soon as a match is found, an attempt is made to match the next pattern. The matching is done without replacement so that if an item in STM is matched by one pattern, it cannot be used to match any of the other patterns. Note that the order in which patterns appear in the conditions and the order in which matching items occur in STM need not be the same. With the exception of special disjunction and negation conditions (to be described later), a production fires only if a match can be found for all of the patterns in its invoking conditions. Thus, if the STM contained

((A)(B)(C)(D))

a production with the invoking conditions,

((A)(D)(C))

would be fired off, while one with invoking conditions,

((A)(C) (F))

would not.

The Pattern Language

The matching procdure just described is that of a context-free grammer; however, because of the pattern language used in the matching procedure the system is effectively a scattered context grammer. This pattern language is interpreted by a function, MATCH, which takes a pattern and an item and indicates whether the pattern matches the item. Both patterns and items can be any legal LISP lists, not atoms. In addition, patterns can contain special expressions of the form,

(specialtoken arg1 .....argn)

Six special tokens are provided:

(*ATOM* arg1)

If this token appears in a special expression, the match succeeds at this point if the corresponding position in the item contains any atom. The first argument, if present, must evaluate to an atom; if the match succeeds,

the atom is set to whatever atom in the item matched the pattern. (Repeat: The argument must evaluate to an atom; it need not be an atom itself.) Additional arguments are ignored.

(*LIST* argl)

Identical to the preceeding, except that any list, but not an atom, is matched.

(*ANY* argl)

Identical to the preceeding, except that any s-expression is matched.

(*REST* argl)

Identical to the preceeding, except that it matches the entire tail (CDR) of the string at that point in the structure.

(*EVAL* argl)

The argument following this special token is presumed to evaluate to an atom (i.e., a name). If the value of this atom is equal to the s-expression at the corresponding position in the item, then the match succeeds. This is useful for matching against a variable that has been set

through the use of special tokens in some prior part of the matching process.

(*CLASS* arg1 arg2)

The argument following this special token is assumed to evaluate to a list. If the s-expression at the corresponding position in the item is a MEMBER of the list, then the match succeeds. Note that since the matching against the list is done by testing whether things are EQUAL, the items in the list cannot be patterns but can only be ordinary s-expression. If a second argument is present, it must evaluate to the name of a variable. Whichever member of the list matched is bound to that variable.

(*FN* function-name arg2 arg3)

This token is used to provide an extensibility feature to the matcher. The first argument is presumed to evaluate to the name of a function. When the special expression is encountered in matching, the function is called with two arguments: The first is the remainder of the pattern (after the special expression), and the second is the remainder of the item. Whatever value the function returns is presumed to be the value of the match for those portions of the pattern and item. Thus, if the function is intended to match only the next element of the target item, it must call

MATCH recursively to complete the match on the tail of the item. If the function returns a non-NIL value, the match succeeds; otherwise failure occurs.

If a second argument is present, it is assumed to evaluate to an atom, and whatever value the function returned is assigned to it. If a third argument is present, it is passed unaltered to the function as a third argument. If this argument is to be used, the function must be defined to accept three arguments. (This may seem weird, but it's useful for passing parameters to the function.)

Some examples of the beast at work are:

Pattern:
(A B (*ATOM*))

Item:
(A B ORANGE)

Succeeds.

Pattern:
(A B (*ATOM*))

Item:
(A B (ORANGE))

Fails.

Pattern:
(A B C (D (*LIST* @DUMMY)))

Item:
(A B C (D (E F)))

Succeeds and sets DUMMY to (E F).

Pattern:

```
(A B  C (*REST* @DUMMY))
```

Item:
```
(A B C D E F)
```

Succeeds and sets DUMMY to (D E F)

Pattern:
```
(A B (*EVAL* @DUMMY) G)
```

Item:
```
(A B (D E F) G)
```

If DUMMY was set to (D E F) by the previous match, then this succeeds.

Pattern:
```
(A (B (*REST* @DUMMY)))
```
Item:
```
(A B C D)
```

Fails.

Pattern:
```
(A B (*FN* @IS-A-K @WHATKWAS))
```

Item:
```
(A B  K )
```

If IS-A-K is a function which checks whether its argument is the letter, K, then this function will be called with (K) as its argument. Whatever it returns will be assigned to WHATKWAS.

Pattern:
```
(A (*ATOM* @V1) B (*LIST*) C (D (*REST*)))
```

Item:
```
(A D B (THIS IS IT) C (D E (F G)))
```

Succeeds and sets V1 to D.

Item:
```
(A F B G C (D R))
```

Fails.

In addition to the special expressions within patterns, the way in which the patterns are to be combined can also be specified. Normally, the combination is done conjuctively; if all the patterns match, the production fires off. Sometimes, it's nice to be able to specify that a production should fire if a pattern cannot be matched in the STM. This can be done by preceeding the pattern with the atom, *ABSENT*. For example, if the invoking conditions of the production are:

1. (A)
2. (B)
3. *ABSENT* (C)

Then the associated actions will be executed only if (C) is not present in the STM.

Another situation which occurs frequently is a production is to be fired if any one of a set of patterns can be matched. This capability is provided by preceeding a list of productions with the atom, *OR*. Thus, a production with the invoking conditions,

1. (A)
2. *OR* ((C)(D))

will fire if (A) and (C) or (D) is present in the STM.

Production Actions

In addition to the pattern language and matcher, this system provides built-in functions for performing the most common actions. (In fact, any LISP function can appear as the action side of a production, but the user should be aware of the internals of the system before writing functions which alter the contents of the STM or which modify the productions.) These functions are given below.

REPLACE

This EXPR takes two arguments: an item in the STM and its replacement. It replaces the first by the second in the same position. If the first argument is not found in STM, an error occurs.

PUSHON

This function evaluates its argument and pushes it onto the front of STM, increasing the length of STM by one.

SHOVE

This function evaluates its argument and shoves it on to the front of STM, losing an element off the end of STM, so that STM stays the same size.

REHEARSE

This EXPR function takes as its argument an item which is present in the STM. It moves that item to the head of the list and pushes all the other items down. If the item is not in STM, an error break occurs.

PDREMOVE

This EXPR function evaluates its argument and attempts to remove it from STM. If the argument is not present in STM, an error occurs.

MATCH-ITEM

Sometimes it is desirable on the action side of a production to refer to one of the items that invoked the production. Evaluating (MATCH-ITEM n) returns the thing which matched the nth invoking condition. Thus, (REHEARSE (MATCH-ITEM 4)) will rehearse the thing which matched the 4th invoking condition.

STMSWTCH

One of the capabilities of this system is to use mulitple, different STMs, though only one is STM is used at a given time to fire productions. This function takes one argument, the name of an STM and makes that STM be the

current one for firing productions. This function can appear anywhere on the action side of a production, but, after it is executed, any further actions which affect STM will refer to this new STM, not the old one. Thus, if the actions are:

```
((PUSHON @(A)) (STMSWTCH @STM2) (PUSHON @(A)))
```

the first PUSHON will act on the STM used to fire this production, while the second one will act on the new STM, STM2.

PDSWTCH

Another capability of the system is to switch which particular production system is being used. While it can be shown that a single, large production system can be constructed which combines two smaller ones, keeping the smaller systems separate is often convenient. This function makes it possible to do so by allowing the user to dynamically switch to a new production system as an action of a prior one. It takes one argument, the name of the new system. After it is executed, all future productions will be fired from the new system.

XACUTE

Here it is backward-chaining freaks! This EXPR takes one argument which should evaluate to the name of a production. The production must be defined, but it need not be inserted in the production list. As its name hints, XACUTE tries to fire the named production if its conditions are met. If the attempt is successful, XACUTE returns the name of the production, NIL otherwise. To backward chain, simply maintain a list of productions in some useful place, say, as part of an element in STM, and then call XACUTE on them one by one.

NEWPD

The intent of this function is to permit production systems to dynamically add to themselves. It is an EXPR which takes four arguments:

1. Conditions of a new production in the form of a single list.
2. Actions of a new production in a single list.
3. Name of the new production.
4. Position in the existing list of productions.

The position at which the new production is to be inserted can be specified in one of two ways. If a digit is given, the new production will be inserted before that ordinal

position in the list, i.e. if the argument is 4, then the new production will be inserted after the 3rd one in the existing list. An argument of Ø will cause the new production to be placed before any of the old ones. Alternately, the position can be specified by giving the name of an existing production. In this case, the new production is inserted before it.

KILLPD

This EXPR takes an argument which must evaluate to the name of an existing production in the list. It makes the production go away entirely. All kinds of obscure errors occur if the named production is not in the list.

A few examples of productions are given below:

```
Conditions:

1. (SYMBOL X)
2. (GOAL (FIND X))

Actions:

1. (REPLACE (MATCH-ITEM 1) (LIST @OLD-SYMBOL @X))
2. (PDREMOVE (MATCH-ITEM 2))


Conditions:

1. (SENTENCE (*ANY* @NOUN) ((*CLASS* @VERBS
 @VERB1))
2. (CONTEXT (SPEAKER1 JOHN)(SPEAKER2 BILL))

Actions:
```

```
1. (PUSHON (LIST @ACTION VERB1))
2. (PDREMOVE (MATCH-ITEM 2))
```

For a more extended example of a production system, see Appendix 1.

The Production System Environment

The top level for the production facility is called PDTOP. It replaces the normal top level of LISP and can be used to define, edit, run, and trace productions. It has it's own prompt character, ">". Everything typed to the > prompt is interpreted by PDTOP.

(To invoke it, load in all the functions. Then do:

```
(INITFN @PDTOP)
(PDTOP)
```

You are now typing to PDTOP.)

Using PDTOP

PDTOP differs from the normal EVAL loop in one major respect: Typing an atom name does not cause that atom to be evaluated; instead, they are assumed to be commands relevant to the production system. List expressions are evaluated in the usual manner.

PDTOP allows the user to string several command together on a line, but if a command takes arguements, they must follow it on the same line. Errors are treated the same way as they are by the LISP editor; a message gets printed and the rest of the line is ignored. Some of the

more frequently used commands can be abbreviated to only two letters.

For PDTOP, every production system has a name and a definition. The definition is a set of productions. In turn, each production also has a name and a definition. STMs are considered to have names and contents.

At any given time, several different production systems may be defined. There may also be several STMs, each with different contents. The system is set up to keep track of which production system and which STM is to be used when productions are run. The user may use the INIT command, described below, to switch among productions systems and STMs. (Alternatively, it is possible to write production systems which call other production systems or switch STMs. See the System Architecture section for details.)

In the following sections, underlining is used to indicate information typed in by the user.

Commands That Run Productions

FINDPD

FINDPD, which invokes the function, FINDPD, finds the first production in the current production system which is "true" for the current contents of the STM. It prints the

name of this production or NIL if no production is true. It also sets a global variable, PDFOUND, to the production itself (i.e., to the value, not the name) and two other variables, CONDITIONS and ACTIONS, to the conditions and actions of the production respectively. Note that it just finds the production; it doesn't run it.

Example:

```
>FINDPD
(TREE-17)
>
```

APPLYPD

This command uses the function, APPLYPD, to execute the production last found by the FINDPD function. To do this, it looks at the contents of PDFOUND.

Example:

```
>FINDPD
(PD-1)
>APPLYPD
>
```

RUNPD

This command uses the RUNPD function to find and run productions. The number of productions that will be run is determined by the CYCLES command described below. If the CYCLES command is not given, just one prodction is run. To run productions until some condition is met, see the definition of the RUNPD function. Note that RUNPD produces

no printed indication that a production has been run; to do that, the TRACE command must be used.

Example:

```
>RUNPD
>
```

CYCLES

This command takes one argument, the number of productions to be run the next time a RUNPD command is given. It must be given again for each RUNPD command; otherwise, only a single production will be run. The value of CYCLES is the number of productions that will be run on next RUNPD command.

Example:

```
>CYCLES 10
>RUNPD
>
```

This would cause 10 productions to be found and run.

TRACE , !UNTRACE

TRACE can take as its argument either an atom or an expression. If it is an atom, then it is presumed to be the name of something to be printed out each time a production is run. Currently, the names that are recognized are

NAME NUMBER STM SIZE

NAME and NUMBER stand, respectively, for the name and number in the list of the production that was just run. STM causes

the contents of STM, in numbered format, to be printed out each time a production is run. SIZE causes the size of STM to be printed out each time.

Giving an expression as the argument to TRACE causes the expression to be evaluated after each production is run. This allows the user to create arbitrary traces of his or her own choosing. For example, to keep a count of the number of productions run since a certain point, do TRACE (SETQ NCOUNT (ADD1 NCOUNT)), where NCOUNT is the name of the counter. TRACE commands are cumulative; to trace several things at once, simply give several trace commands.

To stop all tracing, type !UNTRACE. (In the future, the capability will be added to selectively stop tracing.)

Example:

```
>TRACE NAME
>RUNPD
(PD-1)
>
```

Commands for Creating and Editing Production Systems

This command is used to initialize a new production system. PDTOP responds to it by prompting for the name of the production system and the name of the STM that is to be used with it. If the productionsystem and the STM both already exist, then they become the ones which are

used. If the STM does not already exist, then the user is prompted for an initial list of items to be used as the initial contents for the STM. If the production system does not already exist, it is defined to have a single production called DUMMY in it. (Be sure to remove this dummy production before actually attempting to run the system.)

PDTOP also has a set of commands which allow definition and editing of productions. In many of these commands, it is assumed that PDTOPs attention is "focused" on a given production. When a new production system is initialized, this attention is directed to the first production, and other commands may be used to move this attention.

Example:

```
>INIT
PRODUCTION SYSTEM NAME>DEMO-PD
STM NAME>S1
INITIAL VALUE FOR STM>((A)(B)(C))
>
```

NEW

This command defines a new production, but does not place it into the production system. It prompts for the name of the new production, the invoking conditions, and the actions. Conditions and actions are are prompted for, one at a time. Each condition or action must be a list, and an error message will be given if it is not. Reading is done by the LINEREAD function, so that each condition or action

must be typed in its entirety before giving a carriage return. To terminate prompting for the next item, type NIL. If the name of an existing production is given, an error message will be given.

Example:

```
>NEW
NAME?TEST-1
ENTER CONDITIONS:
1. (A)
2. (B)
3. NIL
ENTER ACTIONS:
1. (PDREMOVE (MATCH-ITEM 1))
2. (PUSHON @(FIRST LETTER))
3. NIL
>
```

INSERT

The INSERT command takes a defined production and inserts it into the production system. The command takes 2 arguments which must follow it on the command line. The first of these is the name of the production to be inserted; the second is the position at which it is to be added. The position may be specified in one of two ways. If it is given by a number, then the new production is inserted after the nth production. (i.e. if 5 is the number, then the new production is stuck in at position 6, pushing everything after it down one position) A zero is used to indicate that the new production is to be added at the beginning of the list. Alternately, the position for the new production can be specified by giving the name of a production already in

the system. The new production will then be inserted after it. In either case, the inserted production is made the current attention focus.

Example:

```
>INSERT PD-7  3
>INSERT PD-7 PD-3
```

REMOVE

This command removes the production that is the current focus of the editor's attention from the production system. The production is still "defined" however, so that it may still be inserted again later on.

```
>REMOVE PD-1
>
```

MOVE

This command takes two arguments, a production name and a position. The production is removed from its current location and is moved to the new position. The position can be specified in the same way as for the INSERT command.

Example:

```
>MOVE PD-1 8
>
```

KILL

This command takes one argument, the name of a production to be killed. Once a production has been killed,

the name is available for redefinition and will not cause an error message from the NEW command. Only productions that are not in the production list can be killed; if a production to be killed has already been inserted in the list, it is necessary to REMOVE it before KILLing it.

Example:

```
>KILL PD-3
>
```

PP

This command pretty-prints the editor's attention focus.

Example:

```
>PP
CONDITIONS:
 (A)
 (B)
ACTIONS:
 (PDREMOVE (MATCH-ITEM 1))
 (PUSHON @(FIRST LETTER))
```

?

Prints the current focus without indenting, wrapping it around to successive lines as necessary.

Example:

```
>?
(((A)(B))((PDREMOVE (MATCH-ITEM 1))(PUSHON @(FIRST
 LETTER))))
>
```

P

Prints the current focus but only to a depth of 3. "Deeper" expressions are indicated by &.

```
>P
((A)(B(C(& &))))(PUSHON @(A)))
>
```

NEXT

Moves the focus of the editor's attention to the next production in sequence.

Example:

```
>NEXT
>
```

BACK

Moves the focus of the editor's attention back to the previous production.

Example:

```
>BACK
>
```

TO

This command moves the focus of the editor's attention to the location specified by the next thing on the command line. The location may be specified by either a number or the name of a production. Thus, TO 5 will change the focus to the 5th production in the set.

Example:

```
>TO 5
>
```

FIND

FIND takes a single argument which must follow it on the command line. The argument is a pattern, and the production system is searched until a production is found which contains the pattern. This production then becomes the editor's attention focus. Patterns can contain any of the wildcard constructions that the LISP edtior F commands use. They can match any part or all of the whole production or of the conditions or actions. If the pattern cannot be found, an error message will be printed, and the old focus will not be changed.

Example:

```
>FIND   (PDREMOVE (MATCH-ITEM 1))
>
```

EDA,EDC

The LISP editor is called on the conditions or actions respectively of the production which is at the editor's focus. (Warning: Don't call the LISP editor with (EDITF BLAH), where BLAH is the name of a production; it messes up the quick matching routines.)

Example:

```
>EDA
EDIT
#
>
```

EDITA,EDITC

These commands take as an argument the name of a production and call the LISP editor on the conditions or actions respectively.

Example:

```
>EDITA PD-1
EDIT
#
```

NEWSTM

This command takes one argument which must follow it on the command line. The argument is evaluated, and the current STM is set to the value.

Example:

```
>NEWSTM @((A)(B)(C))
>
```

FIXSTM

This command calls the LISP editor on the current STM, permitting the user to alter its contents as he or she likes.

Example:

```
>FIXSTM
EDIT
#
```

STM

Prints the current contents of STM, stringing it out across lines if necessary.

Example:

```
>STM
 ((A)(B)(C)(C)(E))
>
```

NSTM

Prints the contents of STM, numbering each element and pretty-printing it.

Example:

```
>NSTM
1. (A)
2. (B)
3. (C)
4. (C)
5. (E)
```

PDNAMES

Prints the names of all the productions in the production system. To print just a section of the names, use the function call from of this command described below.

Example:

```
1. PD-1
2. FIRST-PD
3. HALT-7
4. TRY-AGAIN
```

PDPRINT

Prints out all the productions, showing name, condtions, and actions for each one. To print juct a section of the productions, use the function call from of the command described below.

Example:

```
>PDPRINT
1.
NAME: PD-1
CONDITIONS:
  (A)
  (B)
  (C)
ACTIONS:
(REHEARSE (MATCH-ITEM 1))
(PDREMOVE (MATCH-ITEM 2))

2.
NAME: FUN-PD
CONDITIONS:
   (HAVE A (*ATOM* @V1) TIME)
   (WHAT KIND OF TIME?)
ACTIONS:
    (REPLACE (MATCH-ITEM 1)(LIST @HAVE @A @FUN @TIME))
 (PUSHON (LIST @BIG (CAR (MATCH-ITEM 1)))
```

DISPLAY

The one argument to this command must be the name of a production. The command causes the conditions and actions of the production to be pretty-printed.

Example:

```
>DISPLAY PD-1
CONDITIONS:
 1. (A)
 2. (B)
ACTIONS:
 1. (PUSHON @(THIS IS IT))
 2. (PDREMOVE (MATCH-ITEM 2))
```

READ-IN, WRITE-OUT

These commands read in and write out a production system and STM onto a disk file. They both take one argument, the name of the file to be used. If the file name

has an extention, the name and extension must be enclosed in parentheses, as in (SAVEPD.LSP).

The file that is written will contain the contents of the current STM and the definitions of the individual productions. If the user desires to have additional things saved, they can be included by putting them on a list called PDSAVELIST; for example, to cause the value of the class name, FLOOR-CLASS, to be saved, do (SETQ PDSAVELIST (CONS @(FLOOR-CLASS) PDSAVELIST)). PDSAVELIST may also be edited using the normal LISP editor.

Example:

>READ-IN (OLD.PD)

FORMATFILE

This commands takes as its argument the name of a file. As usual, files with extentions must be enclosed in parentheses. It causes the current contents of STM and the current productions to be printed out in a readeable format. Alternately, this command creates the production system equivalent of a compiler listing file.

EXample:

>FORMATFILE (OLDPD.LST)
^C
.PRINT OLDPD.LST

E

The next item on the command line after this command is evaluated and the value printed. This provides a simple way to evaluate atoms instead of having them treated as commands.

Example:

```
>E PEOPLE
(JOE JILL JOHN JACK)   This list was the value of PEOPLE.

OK
```

This command causes an exit from PDTOP and a return to the normal LISP top level.

```
>OK
LEAVING PDTOP
*
```

The following two items are not commands, but must be called as functions.

PDPRINT, PDNAMES, NSTM

Another nice thing to have is functions which display a production system in a reasonable display format. Evaluating (PDPRINT n m) causes the nth through mth productions to be printed in numbered order with conditions and actions separately labeled, numbered and GRINDEFed. Omitting m causes the printing to run from the nth production through the end, while omitting both m and n causes all the productions to be printed. PDNAMES works the same way, but prints only the names of productions. NSTM

does the same thing for the contents of the STM, useful if it contains a large number of items.

Example:

```
>(PDPRINT 1 2)
1.
NAME: PD-1
CONDITIONS:
  (A)
  (B)
  (C)
ACTIONS:
(REHEARSE (MATCH-ITEM 1))
(PDREMOVE (MATCH-ITEM 2))

2.
NAME: FUN-PD
CONDITIONS:
   (HAVE A (*ATOM* @V1) TIME)
   (WHAT KIND OF TIME?)
ACTIONS:
    (REPLACE (MATCH-ITEM 1)(LIST @HAVE @A @FUN @TIME))
 (PUSHON (LIST @BIG (CAR (MATCH-ITEM 1)))
```

SAVE,RESET,UNSAVE

Very often in debugging a production system, it is desirable to return to some previous point, and try again. These three commands provide a simple backtracking capability for this purpose. Typing SAVE pushes a copy of the current contents of STM onto a stack called PDCONTEXTSTACK. Typing RESET followed by a number causes the contents of STM to be set to the nth contents to be saved; i.e., RESET 1 always causes things to be set to the oldest context. UNSAVE causes the contents on the top of the stack, the newest, to be deleted from the stack.

The function which do the context saving "know" the STM name. Thus, if the user switches STMs, the proper restoration will take place. As promised earlier, the user can cause other things besides the STM to be saved. To do this, make a list of the things whose VALUEs are to be saved be the value of the atom, CNTXTI (for CoNTeXT Items).

Example:

```
>SAVE
1
-other commands-
>SAVE
2
-more commands-
>RESET 1
CONTEXT RESTORED TO 1
>
```

Production Running and Tracing Functions

These are primarily of interest to the systems builder and hack.

RUNPD

RUNPD is a global production-running function. It takes two arguments; either the second argument or both arguments may be omitted - i.e. if only one argument is given it must be the first one. The first argument is a list of forms to be evaluated each time a production is run. The second argument is a list of conditions which are evaluated before a production is run; if any of the

conditions are true, then the production isnt run. If no arguments are given, then an attempt is made to run just one production; if no productions are true, an error break occurs. The list of forms will usually include things like forms to print out various things; if it is desired to run more than one production at a time, a function can be included to decrement a counter. The list of conditions to be evaluated will include various reasonable terminating conditions, such as the counter having reached zero. (In fact, the "no arguments" situation is handled by adding a form to the list of forms which CONSes a T onto the list of conditions so that no further productions are run after the first one.)

Example:

```
(RUNPD @((SETQ N (PLUS N 1))) @((EQUAL N 5)))
```

If N is Ø, then this call to RUNPD will cause 5 productions to be run.

EVERY

EVERY is a function which evaluates a list of s-expressions on every nth count of some counter; it's handy for doing things on every nth cycle of a production system

or on every nth time a certain production fires. It is an EXPR with 3 arguments. The first is the current value of the counter; the second is the interval to be used, and the third is the list of forms to be evaluated. If PDCOUNT were being used to count the cycles of a production system, then including (EVERY PDCOUNT 5 @((PRINC STM))) in the list of actions to be performed for each production, then STM would be printed out on every 5th cycle of the production system.

Extension Functions for the Pattern Language

The following functions are handy extensions to the pattern language when used in the *FN* construction.

EITHER

The optional third argument is presumed to evaluate to a list of patterns or pieces of patterns. If the corresponding part of the item matches any one of the patterns, then the function CONSes that part of the item onto whatever MATCH returns for the rest of the pattern and the item. This function is intended to allow matching to a number of alternatives at any point in the pattern.

Example:

If S1 has the value, (A B C D), then (W X Y A Z) will be matched by (W X Y (*FN* @EITHER @DUMMY @S1) Z) and DUMMY will have the value, (A).

NOTMATCH

This function provides a negation condition within a pattern. The optional third argument is presumed to evaluate to a list of pieces of patterns. If the part of the item occuring in the same position as the *FN* construct in the pattern is not a member of the list, the match succeeds.

Example:

The pattern is (A (*FN* @NOTMATCH @DUMMY @(F) (G)(H))). It will succeed when matched against (A E) but will fail when matched against (A J). In either case, DUMMY will be set to NIL.

System Architecture

The following section is intended to give an understand of system architecture to users who would like to either modify it or push it to extremes.

The Matching Procedure

From the point of view of the user, the system appears to go through the list of productions and to attempt, one at a time, to match the invoking conditions of each production against the contents of the STM until one of the matches succeeds. In fact, the internals of the system proceed in substantially the same way, but two techniques are used to insure that the match takes place in the minimum amount of time.

The first of these involves discovering whether the condtions of a production and the STM contain elements with the same atoms in them, whether or not the list structures are the same. If the atoms used in the invoking conditions are A,B, and C and the items in STM are built up out of G, H, and I, the match can never succeed. If this can be discovered before attempting the match, the system will run faster. To perform this check efficiently, each time a production is defined or altered, a "flattened" version of it is also created by the function, PDSETFAST. This flattened version is simply a list of all the atoms that appear anywhere in the invoking conditions; thus, (GOAL LEFT-LINK (SUBTREE (A))) and (PERFORM (SEARCH (TREE))) would be flattened to (GOAL LEFT-LINK SUBTREE A PERFOMR SEARCH TREE).

For this technique to be effective, some rapid way to check these atoms against those in STM must be available. The mehtod used here is to add a property, %STM, to each atom used in an STM element. The value of this property is the number of times that atom appears in STM. These reference counts are created initially by PDUNREF and PDIREF. Functions which alter the contents of STM, such as PUSHON or PDREMOVE, update these reference counts. Checking whether an atom is in STM is done simply by finding if (GET TOKEN @%STM) returns a non-NIL value. This check is done by the QCKMATCH function.

The second speed-up technique is applied if the first technique shows that an atom correspondence does, indeed, exist. It consists of converting the patterns that form the invoking conditions from a passive into an active form; this active form is then executed to determine whether a match occurs. For example, (OBJECT RED (*ATOM*)) is converted to:

```
(AND (EQ (LENGTH STRING) 3)
     (EQ (CAR STRING) @OBJECT)
     (EQ (CAR (SETQ STRING (CDR STRING))) @RED)
     (ATOM (CAR (SETQ STRING (CDR STRING)))))
```

When this expression is evaluated, it returns T is the original pattern would have matched. This compilation is

carried only one level deep; for more deeply nested structures, a recursive matcher, MATCH, operates on the passive pattern.

As with the list of atoms in the first technique, a compiled version of a pattern is created when a production is defined or edited. The function responsible for doing this is PDCMPL. It calls MKMTCHP, MKMTCH1, MKMTCH2, and PATP. The combined effect of this compilation technique and the signiture technique described previously is to produce a two-fold increase in speed over direct matching.

Data Structures

Internally, a production is the value of the production name. It is organized as a 5-tuple. The elements of the 5-tuple are:

1 list of atoms used in the condition
2 compiled version of the conditions
3 uncompiled conditions
4 actions
5 production name

Note that since the production name is also part of its value, it is possible to go either from the name to the value or in the opposite direction. Because the user is not

required to know about the fast matching techniques, the display functions show only the last 3 elements of the 5-tuple.

A production system is the value of a production system name. It is organized as a dotted pair. The CAR of the pair is a list of production names. The CDR of the pair is a list of the values of the production names (hopefully) in the same order as the names themselves.

STM is a simple list of items. Each item must be a list. Items cannot be atoms.

In addition to the production system and the STM, the system makes use of 6 global variables. PDNAME has as its value the name, not the value, of the current production system. Most of the functions concerned with finding and running productions look at the value of this variable to know which production system to use. The value of this name can be changed dynamically by the action of productions, so that one production system can "call" another. Similarly, STMNAME has as its value the name of the current STM. Interesting effects can be achieved by changing the value of STMNAME so that the STM that the productions are sensitive to also changes.

The remaining 4 globals are used to store various information about the most recent production found by the

matching process. When a production is found to be "true," CONDITIONS is set to the invoking conditions of that production; ACTIONS is set to the ACTIONS, and PDFOUND is set to the entire condition-action pair. ACTIONS is used by the function which executes the action side of the production, while CONDITIONS and PDFOUND are used by various tracing routines. WHATMATCHED is set to the actual items used to satisfy the conditions; it is looked at by the MATCHiITEM function.

Flow of Control

FINDPD, a function of no arguments, is the basic function for finding the next production to fire. It evaluates STMNAME and PDNAME to find the current STM and production system and then calls CMATCH to find a production whose invoking conditions are met. FINDPD sets the CONDITIONS, ACTIONS, and PDFOUND global variables to their appropriate values, or NIL if no production has been found. It returns the value of PDFOUND.

CMATCH

CMATCH takes two arguments, an STM and a production system, and searches the production system to find one whose conditions are met. It sets WHATMATCHED to the actual items which were used to match the invoking conditions, and it

returns the production which it found "true." CMATCH calls two internal functions, CMATCH*1 and CMATCH*2, and the general matching function, MATCH.

APPLYPD

APPLYPD is a function of no arguments which is responsible for carrying out the action side of a production. Its definition is simply (MAPCAR (FUNCTION EVAL) ACTIONS), so that any legitimate LISP S-expression may appear on the action side of a production. Note that the production finder, FINDPD, and the production applier, APPLYPD, communicate only through the global ACTIONS variable. This means that, for debugging, it is possible to do repeated production searches without firing a production, or to fire a production repeatedly without searching for it.

RUNPD

RUNPD is a top level function to run and trace productions. To actually run productions, it simply calls FINDPD followed by APPLYPD. The other capabilities described in the previous section are obtained simply by evaluating lists of expressions before FINDPD and APPLYPD are called.

System Cross Reference

The following section gives a list of the functions used in the system and the calling heirarchy. Using the LISP BREAK package, a backtrace of function calls can be obtained if an error occurs. This section and the prior one on system architecture will, hopefully, enable the user to interpret this backtrace and find the cause of the problem. Note that some of the functions can be called directly by the user, as well as by the function listed in the table.

APPLYPD	RUNPD,PDTOP
CMATCH	FINDPD
CMATCH*1	CMATCH*2
CMATCH*2	CMATCH
FINDPD	RUNPD,PDTOP
KSUBST	REPLACE
MATCH	
MATCH-ITEM	PDREMOVE,REPLACE,SHOVE,PUSHON,REHEARSE,
user-written functions	
MKMTCH1	MKMTCHP
MKMTCH2	MKMTCH1
PATP	MKMTCH1
PDCMPL	PDTOP
PDFSVCNTX	PDTOP
PDIREF	PDTOP
PDREMOVE	APPLYPD
PDRSET	PDTOP

PDSETFAST	PDTOP
PDSWTCH	APPYPD
PDTOP	-
PDUNREF	PDTOP
PDUNSVCNTX	PDTOP
PUSHON	APPLYPD
QCKMATCH	CMATCH
REHEARSE	APPLYPD
REPLACE	APPLYPD
RUNPD	PDTOP
SETFASTCK	PDIREF,PDREMOVE,REPLACE, SHOVE, PUSHON
SETFASTLIST	PDSETFAST
SHOVE	APPLYPD
STMSWTCH	APPLYPD
UNCHECK	PDREMOVE, REPLACE, SHOVE
UNZAP	PDUNREF

The following functions are general utility functions that get called by a variety of the functions in the system.

ALPH

APPENDVALUE

CARNTH

DELASSOC

DO-UNTIL

DSET

GREAT

LESS

KWOTE

LPRINT

MAPO

MAPOL

MAPS

MAPT

MAPTL

MASSOC

MKASSOC1

NEWNAME

NEWONE

NUMPRINT

PRINCS

Appendix 1.

The following production system does a preorder traversal of a binary tree. The links in the tree are themselves encoded as productions, that is, the fact that C is the right descendent of A is encoded by a production which stores that linkage.

Productions:

```
1.
NAME:START-1
CONDITIONS:
  ((AT (*ANY* (QUOTE V1))) *ABSENT* (VISITED (*ATOM*))
 *ABSENT* (GO (*ATOM*)))
```

```
ACTIONS:
  ((PUSHON (QUOTE (GO LEFT))))
2.
NAME:LINK-1
CONDITIONS:
  ((AT A) (GO LEFT) *ABSENT* (STACK B))
ACTIONS:
  ((PUSHON (QUOTE (STACK A))) (PUSHON (QUOTE (AT B)))
 (PDREMOVE (MATCH-ITEM 1)) (PRINT (QUOTE A)) (TERPRI))
3.
NAME:LINK-2
CONDITIONS:
  ((AT B) (GO LEFT) *ABSENT* (VISITED D))
ACTIONS:
  ((PUSHON (QUOTE (STACK B))) (PUSHON (QUOTE (AT D)))
 (PDREMOVE (MATCH-ITEM 1)) (PRINT (QUOTE B)) (TERPRI))
4.
NAME:LINK-3
CONDITIONS:
  ((AT B) (STACK B) (GO RIGHT) *ABSENT* (STACK E))
ACTIONS:
  ((PDREMOVE (MATCH-ITEM 1))
   (PDREMOVE (MATCH-ITEM 2))
   (REPLACE (MATCH-ITEM 3) (QUOTE (GO LEFT)))
   (PUSHON (QUOTE (AT E))))
5.
NAME:LINK-4
CONDITIONS:
  ((AT A) (STACK A) (GO RIGHT) *ABSENT* (STACK C))
ACTIONS:
  ((PDREMOVE (MATCH-ITEM 1))
   (PDREMOVE (MATCH-ITEM 2))
   (REPLACE (MATCH-ITEM 3) (QUOTE (GO LEFT)))
   (PUSHON (QUOTE (AT C))))
6.
NAME:LINK-5
CONDITIONS:
  ((AT C) (GO LEFT) *ABSENT* (STACK F))
ACTIONS:
  ((PUSHON (QUOTE (STACK C))) (PUSHON (QUOTE (AT F)))
 (PDREMOVE (MATCH-ITEM 1)) (PRINT (QUOTE C)) (TERPRI))
7.
NAME:TRY-RIGHT
CONDITIONS:
  ((FAIL) (GO LEFT) (AT (*ATOM*)))
ACTIONS:
  ((PDREMOVE (MATCH-ITEM 2))
   (PDREMOVE (MATCH-ITEM 1))
   (PUSHON (QUOTE (GO RIGHT)))
   (PRINT (CADR (MATCH-ITEM 3)))
   (TERPRI))
8.
```

```
NAME:FAIL-LINK
CONDITIONS:
  ((AT (*ATOM*)) *ABSENT* (FAIL) (GO (*ATOM*)))
ACTIONS:
  ((PUSHON (QUOTE (FAIL))))
9.
NAME:BACK-UP
CONDITIONS:
  ((FAIL) (GO RIGHT) (AT (*ATOM* (QUOTE V1))) *ABSENT*
 (STACK (*EVAL* (QUOTE V1))) (STACK (*ATOM* (QUOTE V2))))
ACTIONS:
  ((PDREMOVE (MATCH-ITEM 1)) (PDREMOVE (MATCH-ITEM 3))
 (PUSHON (LIST (QUOTE AT) V2)))
10.
NAME:QUIT
CONDITIONS:
  ((FAIL) (GO RIGHT) (AT (*ATOM* (QUOTE V1))) (STACK (*EVAL*
 (QUOTE V1))))
ACTIONS:
  ((PRINT (QUOTE " TREE TRAVERSED "))
   (TERPRI)
   (PDREMOVE (MATCH-ITEM 1))
   (PDREMOVE (MATCH-ITEM 2))
   (PDREMOVE (MATCH-ITEM 3))
   (PDREMOVE (MATCH-ITEM 4)))
```

Trace of the production system.

The following is a trace of part of the production system operation, consisting of the 6 through 10th productions that were fired.

```
Production fired: (FAIL-LINK)

Stm:
1. (FAIL)
2. (GO RIGHT)
3. (AT D)
4. (STACK B)
5. (STACK A)

Production fired:(BACK-UP)

Stm:
1. (AT B)
2. (GO RIGHT)
```

```
3. (STACK B)
4. (STACK A)

Production fired:(LINK-3)

Stm:
1. (AT E)
2. (GO LEFT)
3. (STACK A)

Production fired: (FAIL-LINK)

Stm:
1. (FAIL)
2. (AT E)
3. (GO LEFT)
4. (STACK A)

Node printed out:E

Production fired: (TRY-RIGHT)

Stm:
1. (GO RIGHT)
2. (AT E)
3. (STACK A)
```