CMU-CS-79-132

OPS4 User's Manual

Charles L. Forgy

July, 1979

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

OPS4 is a member of the class of programming languages known as production systems.[1] The peculiar properties of production systems make these languages well suited to a few task domains. Because they have a particularly flexible mechanism for determining the flow of control through the program, these languages are used for Artificial Intelligence applications and for complex tasks such as command and control. Because this flexible control mechanism allows the parts of the program to be loosely coupled, production systems are used for tasks that require the program to be modified extensively or require the program to modify itself as it is used. And because production systems are believed by some to be a reasonable model of the cognitive facilities of the human brain, these languages are used in the construction of psychological models.

The interpreter on which the production system programs[2] execute can be viewed as a machine comprising two memories and a processing engine. The first of these memories, production memory, holds the program; the other memory, working memory holds the data operated on by the program. Production memory contains an unordered set of condition-action pairs called productions. A production that contains k conditions, C1, C2, ..., Ck, and j actions, A1, A2, ..., Aj is read

> When working memory is such that conditions
> C1, C2, ..., and Ck are true simultaneously,
> actions A1, A2, ..., and Aj should be performed.

The conditions constitute what is called the LHS (left hand side) of the production; the actions constitute the RHS (right hand side). Since the action types that can occur in the RHS include both I/O and data transforming operations, these actions form the basis of a complete programming language. All that is needed in addition to the actions is a means of controlling the order in which the productions are fired.

The ordering mechanism is found in the LHSs and in the treatment accorded the LHSs by the interpreter. To select the actions to perform, the interpreter repeatedly executes the

---

[1] The original version of the OPS language [1, 2] was designed in 1975 at Carnegie-Mellon University by Charles Forgy, John McDermott, Allen Newell, and Michael Rychener. Many of the features of OPS had occurred earlier in PSG [7], PSNLST [8], and RETE, a system designed by Forgy. A revised version of OPS, called OPS2 [3], was developed about a year later and has seen considerable use since then. Rychener wrote a preprocessor for OPS2 which allowed the language to process data that was structured as sets of name-attribute-value triples; OPS2 together with the preprocessor was called OPS3FX. The language described in this manual is another revision of OPS. For general background on production systems, see [10].

[2] Current usage dictates that the term production system should be used both for the language and for the programs written in the language. Hereinafter we will follow that usage.

following steps.

1. Determine which productions have satisfied LHSs.
2. If no productions have satisfied LHSs, halt; otherwise, select one production to be executed.
3. Perform the actions in the selected production.
4. Go to step 1.

This sequence of operations is called the recognize-act cycle. Step 1 is called the match. Step 2 is called conflict resolution. Step 3 is called act. As its name suggests, step 1 involves syntactic pattern matching. The conditions in the productions' LHSs are patterns. During the match, the interpreter tries to find instances of the class defined by each pattern among the objects in working memory -- a process called instantiating the pattern. A production is ready to be executed when all its conditions are instantiated. The selection performed during conflict resolution is guided by a set of rules built into the interpreter. The OPS4 conflict resolution rules were chosen so as to make it easy to simulate common control constructs like iteration and subroutine calls. In OPS4 there is no restriction on the amount or kind of processing that can occur during the act. Typically, however, a production will perform only a few simple actions -- perhaps adding one or two new objects to working memory and deleting one or two existing objects.

2. Basic Features of the Language

OPS4 is an extensible language. It consists of a fairly simple, though complete, base language plus a collection of mechanisms for extending the language. This section describes the base language. Section 5 describes the extension mechanisms. Appendix I contains a complete example production system, to which the reader may want to refer as he reads this section.

2.1 LISP

OPS4 is implemented in MACLISP, a dialect of LISP developed at MIT (see appendix IV), and one must know a small amount about LISP to use OPS4. This section explains enough about LISP to allow the user to run production systems which use only the basic LHS and RHS primitives. To learn more about the language, the reader should consult one of the introductory LISP manuals (for example, [9, 11]) and the MACLISP reference manual [6].

2.1.1 LISP Data Types

LISP has many data types, but only two of them are used in OPS4 productions: the atom and list types. There are three kinds of atoms, integers (fixnums in the MACLISP jargon), floating point numbers (flonums), and a class of non-numeric objects called symbols.[1] The fixnums and flonums are exactly like the numeric data types occurring in other languages. Valid fixnums include

```
0
8
8.
-999999999
```

Valid flonums include

```
0.0
0.8e1
80.e-1
-1.e12
.05
```

Note that a flonum is distinguished from a fixnum by having either a leading or embedded decimal or by having a trailing exponent. A number with a trailing decimal point and no exponent is a fixnum. A symbol is any string of characters not recognized by LISP as a

[1] Actually, MACLISP has more than three kinds of atoms, but since the other kinds are not ordinarily used in production systems, they will not be considered here.

number.

```
A
=X
---
LISP
```

Any character that is recognized by the computer can be used in a symbol; some characters, such as space or line feed, however, will be incorporated in the symbol only if the user specifically requests that they be. Ordinarily, such symbols are used to delimit successive atoms. Thus if one wrote

```
A B C
```

on a line, LISP would take that to be three separate atoms.

The remaining LISP data type, the list, can be defined recursively as a sequence of zero or more atoms or other lists enclosed in parentheses.[1]

```
(A B C)
()
(1 (2) ((3)))
```

2.1.2 NIL

LISP has one atom, **NIL**, which has a number of peculiar properties. For example, **NIL** is considered to be both an atom and a list; it is identical to the list **()**. Because of its peculiar properties, it should not occur in any of the objects stored in working memory. The result of using **NIL** in working memory objects is undefined.

2.1.3 Equality

Since tests for equality of objects form the basis of syntactic pattern matching, it is necessary to consider what it means for two LISP objects to be equal. The concept of equality is handled very poorly by LISP; it is impossible to state what it means for two objects to be equal without bringing in many details of the implementation of LISP being used. However, if the OPS4 programmer uses only the base language, or the language with prudent additions, the following set of rules will capture the meaning of equality.

1. Objects of different types are never equal. A list cannot be equal to an atom (except **NIL**, which should not be used in an OPS4 production system). A fixnum

[1] This definition also is simplified to include only the features of the term that are necessary to the understanding of OPS4 production systems.

cannot be equal to a flonum.

2. Two numeric atoms are equal if their algebraic difference is zero.

3. Two symbols are equal if they are spelled identically. Note that OPS4 automatically converts lower case letters to upper case.

4. Two lists are equal if they have the same number of subelements and the corresponding subelements are equal.

For example, `.01e2` is equal to `1.0` and to `10.e-1`, but not equal to `1.` or to `1.00001`. If automatic conversion of lower to upper case is not disabled (see the MACLISP manual to determine how to do that), then the atom `Start` would be equal to the atom `START`. The list `(a b (c))` would be equal to `(A B (C))`, but it would not be equal to `(AB (C))` or to `(B A (C))` or to `(A B C)`.

2.2 Working Memory

The OPS4 working memory is a set; that is, it is an unordered collection of objects without repetitions. The objects in working memory are called working memory elements. A (very small) working memory might be

```
{ 17 17.0 (GOAL (MCCARTHY AT AIRPORT)) }
```

Note that the number seventeen occurs twice in this working memory. The reason that it does is that since one of the numbers is fixed and the other floating, they are not considered equal (see section 2.1.3). The interpreter itself insures that the set nature of working memory is never violated by not allowing an element to be added twice (attempting to add an element a second time is not an error -- see section 2.5.9).

All working memory elements are constants. OPS4 does not evaluate them in any fashion.

2.3 Production Memory

The OPS4 production memory is an unordered collection of productions. A production is a list containing a right pointing arrow (the symbol `-->`). The LHS is the part of the list before the arrow, and the RHS is the part after the arrow. The following two sections describe the kinds of objects that can occur in the LHS and RHS, and they explain the meaning ascribed to these objects by the interpreter.

2.4 The LHS

As explained in section 1, the LHS of a production contains patterns to be evaluated during

the match. These patterns are called condition elements. The first section below describes the kinds of condition elements that can be constructed in OPS4. The three sections after that explain how these condition elements can be combined to form LHSs.

2.4.1 Condition Elements

This section describes the primitives from which the condition elements are constructed and explains how the primitives are combined to yield meaningful patterns. Since this section is not concerned with how multiple condition elements are combined in a single LHS, all the productions shown in this section have only a single condition element.

The simplest primitive is a constant atom or a list of constant atoms.

```
STOP
(CANNIBAL1 ON LEFT BANK)
(GOAL (CANNIBAL1 ON LEFT BANK))
```

A constant will match[1] only an object equal to itself. Thus the following production would fire and stop the interpreter when STOP entered working memory. (The RHS action <HALT> will be described below.)

```
( STOP --> (<HALT>) )
```

A second important primitive is the variable. A variable is an atomic symbol whose name begins with =.

```
=X
=PLACE
```

A variable may match any single datum, but if a variable occurs multiple times within a single LHS, all occurrences must match equal objects. The object which the variable matches is said to be bound to the variable. The following production, then, would match any element in working memory.

```
( =Z --> . . . )
```

and this production would match any element having two identical subelements.

```
( (=Y =Y) --> . . . )
```

---

[1] When "match" is used as a verb in this paper, it has a particular technical meaning. Object A matches object B if the OPS4 interpreter can create a correspondence between the two (treating A as a form to be evaluated, and B as a constant).

A condition element may be constructed containing both variables and constants; for example,

```
( (GOAL =G) --> . . . )
```

Such a condition will match any working memory element that has the same number of subelements, and has the same constants in the same positions as the condition. Thus the above condition would match

```
(GOAL (MCCARTHY AT AIRPORT))
(GOAL (CANNIBAL1 ON LEFT BANK))
(GOAL 17)
```

or infinitely many other elements. If a variable occurs more than once within a list, the restriction on equality of binding applies. Thus the condition element in the following production

```
( (A =Y =Y) --> . . . )
```

would match

```
(A 1 1)
(A A A)
(A 1.0 1.0)
```

but it would not match

```
(A 1 B)
(A 1 1.0)
```

One variable, = by itself, has been defined to be a "don't care". This variable will match any single subelement, and there is no restriction that multiple occurrences must match equal subelements. Thus this condition element

```
( (= =) --> . . . )
```

will match any element with two subelements, including all the following.

```
(A A)
(1 1.0)
(B C)
(1 (1 2 3))
```

In OPS4 the concept of variable has been extended in two directions: allowing a variable to match more than one subelement and allowing different kinds of restrictions to be placed on the subelements bound to the variable. Let us look first at how different restrictions are

specified. The variables beginning with =, already described, allow specifying that two subelements must be equal, as in

```
( (=X A =X) --> . . . )
```

The one other variable that is defined in the base language, the variable that begins with #, allows specifying that two subelements must be unequal. A variable beginning with # may occur in an LHS only if a variable beginning with =, but otherwise spelled identically, occurs there also. The subelements matched by the two variables are required to be unequal. For example,

```
( (=Z #Z) --> . . . )
```

would match

```
(A B)
(1 1.0)
(1 (1))
```

but not

```
(A A)
(1 1)
```

There is no restriction that two variables beginning with # must match equal or unequal subelements. Thus

```
( (=X #X #X) --> . . . )
```

would match both of the following.

```
(A B C)
(A B B)
```

Although the examples here have shown the = variable occuring before the # variable, this is not required; the = variable may be placed anywhere in the LHS.

To indicate that a variable may match more than one subelement, the symbol ! is placed before the variable. A variable which is preceded by ! matches everything up to the end of the corresponding list in the data element. Thus the condition element

```
( (GOAL ! =X) --> . . . )
```

would match both of the following

```
(GOAL CANNIBAL1 ON LEFT BANK)
(GOAL (CANNIBAL1 ON LEFT BANK))
```

In the case of the first data element, =X would match (and be bound to)

```
(CANNIBAL1 ON LEFT BANK)
```

In the case of the second data element, =X would be bound to

```
((CANNIBAL1 ON LEFT BANK))
```

As another example, the condition element

```
( (=Y ! =Y) --> . . . )
```

would match

```
((A) A)
```

but it would not match

```
(A A)
```

The variable following the ! may match any number of elements, including zero. Thus the condition element

```
( (GOAL ! =X) --> . . . )
```

would also match the following working memory element.

```
(GOAL)
```

Only one ! may occur in any list, and the ! must occur just before the last element in the list. The following condition elements are all legal.

```
( (P Q R ! =) --> . . . )
( (GOAL ! =X) --> . . . )
( (A ! #X) --> . . . )
```

The following is not legal, however,

```
( (A ! =X C) --> . . . )
```

because C occurs after the variable. Each list may contain one !, so the following is also legal.

```
( (A (B ! =X) ! =Y) --> . . . )
```

A final primitive, &, allows specifying that several condition subelements must match the same data subelement. This symbol groups the condition subelements on either side of it so that they will match the same datum. To see how this might be used, consider the condition element from above that matched any data element containing two dissimilar subelements.

```
( (=X #X) --> . . . )
```

Suppose that one wanted to write a condition element that matched data elements containing three dissimilar subelements. The condition element

```
( (=X #X #X) --> . . . )
```

will not work because the second and third variables may match equal subelements; for example

```
(1 2 2)
```

However, by using & and binding a variable to the subelement in the second position, this can be made to work. The correct condition subelement is

```
( (=X #X & =Y #X & #Y) --> . . . )
```

This would not match

```
(1 2 2)
```

but it would match

```
(1 2 3)
```

In this case, =X would match 1; the sequence #X & =Y would match 2, binding =Y to 2; and the sequence #X & #Y would match 3.

Finally, since LISP allows lists to be nested inside other lists to arbitrary depths, OPS4 allows condition elements to be created with arbitrary nesting as well.

```
( =X --> . . . )
( (=Y) --> . . . )
( (=W (=Z)) --> . . . )
( (=A ((((((A)))))) =A) --> . . . )
```

2.4.2 Multiple Condition Elements

An LHS may contain more than one condition element. A multiple condition element LHS is satisfied when all the condition elements match working memory elements simultaneously. The interpreter takes into account the bindings of the variables when it determines whether the LHS is satisfied. Thus the LHS in this production

```
( (A =X ! =) (B =X ! =) --> . . .)
```

would be satisfied if working memory contained

```
{ . . . (A 1) . . . (B 1) . . . }
```

but it would not be satisfied by

{ (A 1) (B 2) }

Since working memory is a set, the positions of the data elements do not matter; the above production would be satisfied if working memory contained the following.

{ . . . (B 1) . . . (A 1) . . . }

2.4.3 Negated Condition Elements

Condition elements and groups of condition elements may have their senses inverted; that is, they may be marked so as to make the interpreter try to achieve their dissatisfaction. A simple example of this is seen in the following production.

((A =X) - (B =X) --> . . .)

The second condition element has had its sense inverted by the symbol -. If working memory contained

{ . . . (A 1) . . . }

but nothing beginning with B, then this production would be satisfied -- because the first condition element would match something and the second would not. Furthermore, if working memory contained

{ . . . (A 1) . . . (B 2) . . . }

but nothing else beginning with B, then the LHS would still be satisfied -- because =X cannot be bound to both 1 and 2. And finally, if working memory contained

{ . . . (A 1) . . . (B 2) . . . (A 2) . . . }

but nothing else beginning with B, the LHS would still be satisfied. It would only be satisfied in one way, however. The first condition element cannot match (A 2) because that would then allow the second to match (B 2). In general, when the interpreter processes an LHS containing both negated and non-negated condition elements, it tries to find matches for the non-negated condition elements that do not allow matches for the negated ones. Only if it is successful is the LHS satisfied.

A condition element may also be negated by embedding it in a list beginning with <NOT>. The following two LHSs are equivalent.

```
( (A =X) - (B =X) --> . . . )
( (A =X) (<NOT> (B =X)) --> . . . )
```

The reason for having the longer form is that it allows negating several condition elements simultaneously.

```
( (A =X) (<NOT> (B =X =Y) (C =X =Y)) --> . . . )
```

The interpreter evaluates such LHSs from the inside out, treating each <NOT>'s condition elements as if they constituted an LHS. To see how this works, consider the following working memory.

```
{ (A 1) (A 2) (B 1 7) (B 2 8) (C 2 8) }
```

First the interpreter determines that the two condition elements in the <NOT> can match the last two working memory elements if everything outside the <NOT> is ignored. Then it determines that the first condition element can match the first two working memory elements if the <NOT> is ignored. It then determines that (A 2) must be disallowed because of the matches found for the <NOT>. Thus it has finally determined that the LHS is satisfied only if the first condition element matches (A 1).

It should be noted that the production

```
( (A =X) (<NOT> (B =X =Y) (C =X =Y)) --> . . . )
```

is not equivalent to

```
( (A =X) (<NOT> (B =X =Y)) (<NOT> (C =X =Y)) --> . . . )
```

In the case of the second LHS, the negated condition elements are evaluated independently. The second LHS is not satisfied by the working memory elements shown above.

OPS4 allows <NOT>s to be nested inside of other <NOT>s to arbitrary depths. The meaning is determined by evaluating the expressions from the innermost out. The only restriction on the nesting is that the first element in each <NOT> and the first element in the LHS cannot be negated.

2.4.4 Restrictions on the LHS

The interpreter processes LHSs (and lists of condition elements within <NOT>s) from left to right, performing every possible test at each step. It will not test more than six variables when processing a condition element. Because of this limit on the number of variable tests, the following production would not be legal. Since the third condition element has seven variables in common with the first two condition elements, processing the last condition element would entail making seven variable tests.

```
( (=X1 =X2 =X3 =X4)
  (=X5 =X6 =X7)
  (=X1 =X2 =X3 =X4 =X5 =X6 =X7)
  --> . . .)
```

The production can be made legal by rewriting it to take advantage of the left to right order of processing condition elements. If it were reordered as follows it would be legal. Processing the second condition element in this LHS entails four variable tests (for =X1, =X2, =X3, and =X4) and processing the third condition element entails three variable tests (for =X5, =X6, and =X7).

```
( (=X1 =X2 =X3 =X4)
  (=X1 =X2 =X3 =X4 =X5 =X6 =X7)
  (=X5 =X6 =X7)
  --> . . .)
```

The LHS compiler will generate an error message if it compiles a condition element requiring more than six variable tests.

2.5 The RHS

The RHS of a production consists of zero or more lists or atoms called actions. During the act phase of the recognize-act cycle, all the actions of one production are executed in order from left to right. This section describes the primitive action types of the OPS4 base language.

2.5.1 Explicit and Default Effects

Most actions are executed for the purpose of changing working memory -- either adding new elements to the memory or deleting old elements. Before explaining how the affected working memory elements are described in an action, it is worthwhile explaining how the desired operation (adding or deleting) is specified.

OPS4 has two mechanisms for specifying this. The first mechanism is stating explicitly the operation to be performed. To specify that an element is to be deleted from working memory, for example, one uses the <DELETE> operation. If the RHS contains the following action (description1 is a description of a data element; the format of these descriptions will be given in the following sections)

```
( . . . --> (<DELETE> description1) )
```

the description will be evaluated, and the resulting working memory element will be deleted. Similarly, the <ADD> operation can be used to add an element to working memory. The action

```
( . . . --> (<ADD> description2) )
```

evaluates *description2* and adds the resulting element to working memory. Since adding an

element to working memory is the most frequently executed action, another mechanism is provided to allow easier specification of adds; if the description of an element occurs in the RHS without any specification of the action to be taken, the element is added to working memory by default. Thus

(. . . --> *description2*)

is equivalent to

(. . . --> (<ADD> *description2*))

2.5.2 LHS Primitives in the RHS

Several of the LHS primitives are also legal RHS primitives. For the sake of uniformity, the effects of the primitives are such that any pattern which is legal in both the LHS and the RHS will describe the same object in both cases. That is, the effects of the primitives are such that executing a production like

(*description3* --> *description3*)

results simply in reasserting the element matched by the LHS. The primitives that can occur in both LHSs and RHSs are constants, = variables, and the character !.

Descriptions containing these primitives are evaluated by copying the description, replacing variables by the values to which they were bound in the LHS, and stripping away the parentheses from the lists preceded by !. Consider the following production.

((A ! =X) --> (A ! =X))

Suppose the condition element matches the working memory element

(A 1 2 3)

Then when the action is evaluated, =X will evaluate to the list to which it was bound (1 2 3), the ! will strip away the parentheses from this list, and A will evaluate to itself. Thus the action evaluates to the matched element, as desired.

Variables in the RHS are recognized only if they received a binding in the LHS. Unbound variables in the RHS are treated as constants.

The character ! can occur in more places in the RHS than it can in the LHS. In the RHS, the character can occur at the beginning or middle of a list as well as at the end, and the character can occur more than one time in a list. Thus the following would be a legal RHS action, though it would not be a legal condition element.

(! =X ! =Y ! =Z)

Even in the RHS, however, it is not legal to have two consecutive segment characters:

```
(A ! ! =W)
```

If the subelement following a ! evaluates to an atom, the ! obviously cannot strip away any parentheses. So when this situation occurs, the interpreter simply ignores the !.

2.5.3 RHS Functions for Describing Elements

A second method for describing structure in the RHS, to be used in conjunction with the LHS forms, is the RHS function. An RHS function is a list whose first element is a function name and whose remaining elements are arguments to the function. The following, which will be explained below, are typical calls on RHS functions.

```
(<QUOTE> =X =Y)
(<READ>)
(<EVAL> (<READ>))
```

These functions can occur anywhere in an RHS action. A function can be part of a description just as a constant or a variable can. A function can even occur as an argument to another function (this is seen in the third example above).

When the function is executed, it returns some number of values which are then used in the expression being built. For example, when the following description is evaluated,

```
(A (<QUOTE> =X =Y) B)
```

it results in the following constant expression.

```
(A =X =Y B)
```

Note that the <QUOTE> function returned two values in this case. It is also possible for an RHS function to return zero values, in which case no trace of the function occurs in the final expression.

2.5.4 <QUOTE>

<QUOTE> takes any number of arguments; it causes all symbols contained within its arguments to be treated as constants. For example, in the production described above, the action

```
(A ! =X)
```

evaluated to

```
(A 1 2 3)
```

If the action had been

```
(A (<QUOTE> !) =X)
```

it would have evaluated to

```
(A ! (1 2 3))
```

If it had been

```
(A (<QUOTE> ! =X))
```

it would have evaluated to

```
(A ! =X)
```

2.5.5 <EVAL>

<EVAL>, is essentially the inverse of <QUOTE>. <EVAL> also takes any number of arguments. It causes the interpreter to evaluate the arguments twice. To see how this works, suppose the variable =X was bound to =Y and the variable =Y was bound to 17. Then the form

```
=X
```

would evaluate to

```
=Y
```

The form

```
(<EVAL>  =X)
```

would evaluate to

```
17
```

The form

```
(<EVAL> (<QUOTE> =X))
```

would evaluate to

```
=Y
```

The form

```
(<QUOTE>  (<EVAL> =X))
```

would evaluate to

```
(<EVAL> =X)
```

2.5.6 <BIND>

The function <BIND> is used to generate new names and to force variable bindings on the RHS. <BIND> may take zero, one, or two arguments. When it is called with zero arguments, <BIND> returns an integer which is different from all integers returned on previous calls to the function. <BIND> with one argument does the same thing, but before returning the value, it forces the variable occurring as its argument to be bound to that value. Its argument, which should begin with =, may be used in the remainder of the RHS as if it were bound on the LHS. <BIND> with two arguments evaluates its second argument, forces the first argument to be bound to the resulting value, and then returns that value. <BIND> evaluates both its arguments. Note that if the variable given to <BIND> received a binding on the LHS, the binding from the LHS will be lost when the variable is rebound.

2.5.7 <NULL>

<NULL> takes any number of arguments. It evaluates the arguments and then returns nothing. <NULL> is generally used mask the returned value of a function that both returns a value and produces a side effect. The following is a typical use of <NULL>.

```
(<NULL> (<BIND> =X))
```

2.5.8 <ADD> and <DELETE>

The functions <ADD> and <DELETE> were described briefly above. This section describes the functions in full detail, considering aspects of the functions that could not reasonably be discussed until the element-describing functions had been introduced.

<ADD> and <DELETE>, like all the RHS functions, can be used in describing elements, though this is not recommended. These two functions return nothing (like <NULL>) so whether they occur at the top level of the RHS or embedded in some RHS action, they will have no effects other than the ones described below.

<ADD> and <DELETE> take any number of arguments. <ADD> evaluates its arguments and causes the resulting elements to be added to working memory. <DELETE> evaluates its arguments and causes the resulting elements to be removed from working memory. If any of the elements are not in working memory, <DELETE> does nothing with them.

2.5.9 <REASSERT>

The function <REASSERT> is closely related to the function <ADD>. <REASSERT> evaluates

its arguments and adds them to working memory as does the other function. This new function, however, will delete any existing elements that are equal to the added elements before the addition takes place. Since working memory is a set and therefore cannot contain duplicate elements, these two functions leave working memory in almost the same state. The only difference is that <REASSERT> will leave pristine elements in working memory while <ADD> will simply leave the old elements. Since conflict resolution is sensitive to the history of the working memory elements, it will sometimes make different decisions depending on which action is performed. (See section 3.2.2).

It should be emphasized that the default action (see section 2.5.1) is <ADD>, not <REASSERT>.

2.5.10 <HALT>

One action, <HALT>, is provided for affecting the processing of the production system. Executing <HALT> sets a flag in the interpreter which causes it to terminate processing at the end of the current recognize-act cycle. Although <HALT> usually is not given arguments, it will accept arguments; it returns them unchanged (like <QUOTE>).

2.5.11 Input-Output

Three RHS actions provide the input-output capability of the OPS4 language.

<WRITE> is the basic output function. It accepts any number of arguments which it evaluates and then prints. The printed expressions appear on one line, and a carriage return is supplied before the first expression. <WRITE> returns nothing.

<WRITE&> is similar to <WRITE> except that it does not supply a carriage return before the first expression.

<READ> accepts any number of expressions typed by the user, and returns those expressions as a result. This function cannot be given arguments. Since <READ> will accept more than one input expression, a special convention must be used to indicate the end of the input; <READ> terminates input on the first unbalanced right parenthesis. Appendix II contains more information about <READ>; in particular, it explains how various special characters are handled. Another special convention allows the user to execute LISP or OPS4 functions while typing information to <READ>. If the user types a line of input and terminates the line with an escape, OPS4 will enclose the line in parentheses and then pass it to LISP to be evaluated. For example, to reset the current tracing mode one might type the following line to <READ>. (See section 4.5.2 for a description of the tracing facilities; see section 4.5

for a description of the function SWITCHES.)

SWITCHES TRACE LEVEL1 *esc*

To terminate processing, the user could type the following.

<HALT> *esc*

Executing a function like this has no effect on the value returned by <READ>; the result of the function call is not returned to <READ>.

<READ> does not evaluate the expressions it returns, nor does it mark them as having come from the outside. If the user wishes either of these things done, he can use the other RHS features of OPS4 to effect them. To cause evaluation of the things that are read in, the following action would be used.

(<EVAL> (<READ>))

Note that since <EVAL> can accept any number of arguments, this will work correctly whether <READ> returns a single expression or many expressions. To mark input so that the other productions can recognize what it is, the following condition element might be used.

(INPUT (<READ>))

This will gather the expressions returned by <READ> into a list and put the constant INPUT at the head of the list. Finally, it might be noted that these abilities (and other abilities) can be combined by nesting function calls. Since INPUT is a constant, the following two actions are equivalent.

```
(INPUT (<EVAL> (<READ>)))
(<EVAL> (INPUT (<READ>)))
```

2.5.12 Manipulating Production Memory

OPS4 provides two functions through which a production system can modify itself.

The function <BUILD> is used to add productions to production memory. <BUILD> can take either one or two arguments. If it has only one argument, that argument should evaluate to a list having the form of a production (i.e., a list which contains the atom -->). This list will be compiled as a production and added to production memory. If two arguments are given, the first argument will be used as the name of the production, and the second argument will be used as just described. A name is created for the production if one is not specified. If NIL is given as the first argument, the production will have no name. <BUILD> returns the name of the production if it has one, and nothing if has no name.

<EXCISE> is used to remove productions from production memory. This function takes any number of arguments, all of which should evaluate to the names of productions. It removes each of the named productions from production memory. It returns nothing.

3. Operation of the Interpreter

This section explains how the OPS4 interpreter executes production systems. It treats separately the three phases of the recognize-act cycle: match, conflict resolution, and act.

3.1 Match

During the match phase of the cycle, the interpreter finds every satisfied LHS, and if some of the LHSs can be satisfied in more than one way, it finds all the alternative ways.

3.2 Conflict Resolution

The set of satisfied LHSs found during the match phase is called the conflict set. The objects in the conflict set are called instantiations. An instantiation is an ordered pair of a production and the list of working memory elements that satisfy its LHS. During conflict resolution the interpreter discards some of the instantiations and then selects one of the remaining instantiations to execute. If the conflict set is empty or if the interpreter discards all the instantiations, execution halts and control returns to the user.

Before describing the conflict resolution process, it is necessary to introduce a concept that might be called the age of a working memory element. Associated with each working memory element is a time tag. The OPS4 interpreter maintains a count of the number of changes that are made to working memory and copies this count into the time tag of an element when the element is <ADD>ed or <REASSERT>ed. The age of a working memory element is computed by subtracting the element's time tag from the current count of the number of working memory changes. Since each <ADD> or <REASSERT> causes the count to change, distinct elements will always have distinct time tags.

The algorithm used by conflict resolution to select an instantiation to fire is described in the next section. Although this algorithm is complex, what it achieves can be characterized fairly briefly:

- First, conflict resolution prevents instantiations from firing more than once. Early production systems were prone to falling into a kind of infinite loop in which the same instantiations fired again and again, never changing working memory enough to cause another instantiation to fire. The OPS4 conflict resolution eliminates this problem.

- Second, conflict resolution makes the production system attend to the most recent data in working memory. This makes the production system easier to program because direction is given to the system's processing; once the system begins a subtask, it is unlikely to be distracted by anything left over from the supertask.

- Third, conflict resolution gives preference to productions with more specific LHSs. Since productions with more specific LHSs are satisfied in fewer cases, they are more likely to be appropriate for those cases in which they are satisfied. More specific productions are therefore chosen when they are available.

3.2.1 The Conflict Resolution Algorithm

When an instantiation is chosen for execution, the instantiation is removed from the conflict set, and it will not take part in conflict resolution again. The process of discarding elements from the conflict set is called refraction. (See section 3.2.2 for a mechanism for circumventing refraction.)

On each cycle the interpreter chooses which of the remaining instantiations to execute by applying in order up to five rules.

1. Order the instantiations on the basis of the ages of the working memory elements in the instantiations. In performing this ordering the interpreter considers all the working memory elements of each instantiation. To order two instantiations, the interpreter first compares their most recent working memory elements (that is, the elements that have the greatest time tag). If one element is more recent than the other, the instantiation containing the more recent element dominates. It the two elements are equally recent, the interpreter compares their second most recent elements, and so on. If the data elements of one instantiation are exhausted before the elements of the other instantiation, the instantiation not exhausted dominates. Only if the two instantiations are exhausted simultaneously does this rule judge them equal.

2. If more than one instantiation dominates under the first rule, order the dominant instantiations based on the number of condition elements contained in the productions' LHSs. Productions with greater numbers of condition elements dominate. Both negated and non-negated condition elements are counted.

3. If more than one instantiation dominates under the second rule, order the dominant instantiations based on the number of constant symbols contained in the productions' LHSs. Productions with greater numbers of constants dominate.

4. If more than one instantiation dominates under the third rule, order the dominant instantiations based on the ages of the productions. The production most recently added to production memory dominates.

5. Finally, if there is still not a single dominant instantiation, make an arbitrary selection of the instantiation to execute.

This set of rules was chosen because (1) the rules make it easy to add productions to an existing set and have the new productions fire at the right time and (2) the rules make it easy to simulate common control constructs such as loops, subroutine calls, and gotos. See

[5] for a defense of these claims.

3.2.2 Adding Elements a Second Time

When <ADD> and <REASSERT> were described above, it was not possible to explain why it was significant that <REASSERT> deleted old copies of elements while <ADD> simply reused them. Now that conflict resolution has been described, the explanation can be given. <ADD>ing an element may affect the ordering of instantiations under rules 1 through 5 above but it will not affect refraction. Since <REASSERT> causes a fresh element to be added to working memory, <REASSERT>ing an element can affect both the ordering and refraction.

3.3 Act

During the act phase of the recognize-act cycle, the interpreter executes the actions of the chosen production. There are two steps to this execution. First, the actions are evaluated one at a time from left to right. Then after the side effects (such as writing information to the user's terminal) have occurred and all the descriptions of working memory elements have been evaluated, the elements to be added to working memory are added one at a time *from right to left*. The reason for this order is to encourage the other productions to process the added elements *from left to right*. Recall that the conflict resolution rules described above cause the production system to attend first to the most recently added working memory elements. Hence, if the elements A, B, and C in this production

```
( . . . --> A B C)
```

were added in the order in which they are written, C would tend to dominate B and A. But since the interpreter adds them in the opposite order, A tends to dominate.

3.3.1 Elements Acted Upon Multiple Times

Since the OPS4 working memory is a set, a production cannot perform more than one operation on a given working memory element. Obviously if it tries to add an element twice, only one addition will take place; and also obviously, if it tries to delete an element twice, only one deletion can take place. But what happens when a production tries both to add and to delete an element? The answer is that OPS4 will execute only the leftmost action.

This convention was adopted because it allows the user to determine on a case by case basis whether he wants the adds or the deletes to take precedence. If the user writes

```
( . . . --> (<ADD> des1) (<DELETE> des2) )
```

then when the two descriptions evaluate to equal elements, the <ADD> will take precedence.

If he writes

(. . . --> (<DELETE> *des2*) (<ADD> *des1*))

then the <DELETE> will take precedence.

4. Building, Manipulating, and Executing Systems

This chapter explains how to interact with the OPS4 interpreter. Appendix I contains a sample run of a production system which makes use of many of the features described in this section.

4.1 Evoking the OPS4 Interpreter

OPS4 runs on a PDP-10 computer under the TOPS-10 operating system. To evoke the OPS4 interpreter, the user gives to the operating system the command

```
.R OPS4
```

Anything typed after that is interpreted by the OPS4 user interface. The user can give commands to build, manipulate, or execute production systems (these commands are described below) or he can give commands to LISP to do things such as read in files (see appendix IV). If files are read in, the contents of the files are treated just as if they had been typed in from the user's terminal. Thus files can also contain commands to do things like defining production systems or even commands to read in other files.

4.2 Adding Productions to Production Memory

Productions are added to production memory using the command **SYSTEM**. The user types a left parenthesis, the symbol **SYSTEM**, some number of productions, and then a right parenthesis. For example, the following defines one production.

```
(SYSTEM    ((GOAL =X) =X --> (<DELETE> (GOAL =X))) )
```

SYSTEM compiles the LHSs of the productions and adds the productions to production memory.[1] While it is possible to define an entire production system using only one call on **SYSTEM**, the defining will proceed slightly faster if many calls are used, each containing only a few productions.

If the user wishes to name a production, he may do so by preceding the production with an atomic symbol which will become the name. The following defines three named productions.

[1] For an explanation of how the productions are compiled, see [4].

```
(SYSTEM

PLUS0X
  ((SIMPLIFY =N (+ 0 =X)) & =C  --> (RESULT =N =X) (<DELETE> =C))

PLUSX0
  ((SIMPLIFY =N (+ =X 0)) & =C  --> (RESULT =N =X) (<DELETE> =C))

PLUSXX
  ((SIMPLIFY =N (+ =X =X)) & =C
     --> (RESULT =N (* 2 =X)) (<DELETE> =C)) )
```

If the symbol NIL appears in a name position, the interpreter ignores the symbol, and the production is unnamed. Named and unnamed productions can both be defined in a single call of SYSTEM. The following a legal use of SYSTEM; it defines one named and two unnamed productions.

```
(SYSTEM

NIL
 ((SIMPLIFY =N (* =X 1)) & =C --> (RESULT =N =X) (<DELETE> =C))

 ((SIMPLIFY =N (* =X 0)) & =C --> (RESULT =N 0) (<DELETE> =C))

TIMEXX
 ((SIMPLIFY =N (* =X =X)) & =C
   --> (RESULT =N (^ =X 2)) (<DELETE> =C)) )
```

4.3 Executing a Production System

Execution of a production system is initiated with the function START. The user types a left parenthesis, the symbol START, an indication of what the initial contents of working memory should be, and a right parenthesis. START will delete the elements currently in working memory, if there are are any, and then add the indicated elements and allow the production system to begin execution. There are several ways to indicate the initial contents of working memory. If a list is given to START, it will use the contents of that list as the initial contents.

```
(START ((GOAL (PEG3 HOLDS 1 2 3 4 5))
        (PEG1 HOLDS 1 2 3 4 5)
        (PEG2 HOLDS)
        (PEG3 HOLDS)))
```

If more than one argument is given to START, it will encapsulate the arguments in a list and

then treat the list as above. Thus, the following call on START has the same effect as the previous one.

```
(START  (GOAL (PEG3 HOLDS 1 2 3 4 5))
        (PEG1 HOLDS 1 2 3 4 5)
        (PEG2 HOLDS)
        (PEG3 HOLDS))
```

Note that as a special case of this, the empty list can be given to START, which will cause it to delete the current contents of working memory and then add nothing.

```
(START)
```

If an atomic symbol is given to START, it will retrieve the binding of the symbol and use it; the binding must be a list. (The documentation for MACLISP [6] explains how to bind a value to an atom.)

```
(START TOWER-OF-HANOI)
```

The function CONTINUE provides an alternate way to initiate execution of a production system. This function is used exactly like START. The only difference between this function and START is that CONTINUE does not delete the existing working memory elements. The elements added by CONTINUE will be more recent than the existing elements.

4.4 Interrupting Processing

The user can interrupt the processing of a production system at any time. If he simply starts typing on the terminal, the interpreter will pause at the beginning of the next recognize-act cycle and execute the following action.

```
(<REASSERT> (ATTEND (<READ>)))
```

The <READ> will pick up the information already typed, plus everything up to the) that terminates the <READ>. This action will be treated as if it were the first action in the RHS of the production that fires. Hence the element that is added to working memory will be the most recent element in the memory, and it will dominate the rest of the elements under the OPS4 conflict resolution rules.

4.5 Setting Switches

There are a number of user setable switches that determine exactly how the OPS4 interpreter handles the productions. These switches are set with the function SWITCHES. The user types a left parenthesis, the symbol SWITCHES, and then one or more switch names,

following each name with the new setting for it. He might, for example, type the following. (The meanings of these switches are given below.)

```
(SWITCHES
        KEEP-LHS OFF
        COMPILE-PRIORITY ORDER
        RETIRE-AT 1000
        TRACE NIL)
```

Setting the switches and then resetting them later is allowed.

4.5.1 Directing the LHS Compiler

The OPS4 interpreter compiles LHSs, so it is not necessary to keep the uncompiled LHSs just to run a production system. However, if the user plans to display or edit productions, he will of course want the uncompiled LHSs. The switch KEEP-LHS indicates to the compiler whether the LHSs are to be kept. Productions which are compiled while the value of the switch is ON have their LHSs stored with them; productions which are compiled while the value is OFF do not have their LHSs stored. Since it is often convenient to keep the LHSs of some productions while discarding the LHSs of the rest, sequences like the following are common.

```
(SWITCHES KEEP-LHS ON)
(SYSTEM . . .)
(SWITCHES KEEP-LHS OFF)
(SYSTEM . . .)
```

Since there is no way for the user to access unnamed productions, the LHSs of unnamed productions are always discarded, regardless of the setting of KEEP-LHS. The default setting of this switch is OFF.

The switch COMPILE-PRIORITY is used to give information to the LHS compiler which will help it to compile more efficient code. Executing

```
(SWITCHES COMPILE-PRIORITY SYMBOLS)
```

declares to the compiler that in this production system, tests for symbols are more discriminating than tests for lengths of lists, and that the compiler should therefore arrange for symbol tests to be performed first when possible. Executing

```
(SWITCHES COMPILE-PRIORITY LENGTH)
```

declares that tests for list lengths are more discriminating and that they should therefore be given this priority. Executing

```
(SWITCHES COMPILE-PRIORITY ORDER)
```

declares that neither is significantly more discriminating and that the tests should reflect the order of subelements in the condition elements. To use this switch effectively may require experimentation; the user might compile the system three times to determine which setting gives the smallest or fastest compiled program. Although it is legal to compile different parts of the production system with different settings, doing so generally results in larger and slower programs. The default setting is ORDER.

4.5.2 Tracing Execution

The switch TRACE is used to tell the interpreter how much trace information to print while the production system is executing. Setting the switch to NIL turns off all tracing. Setting the switch to LEVEL1 turns on a minimal trace; with this setting, the names of the fired productions and a count of the number of firings is printed. Setting the switch to LEVEL2 turns on a more complete trace; with this setting, the count of the number of firings, the name of the production, the data matched, and the resulting working memory changes are printed.

Sometimes the user will want to trace only certain productions. The functions TRACER and UNTRACER are provided to allow him to declare which ones to trace. Executing

```
(TRACER P1 P2 P3 . . . Pk)
```

declares that productions P1, P2, P3, ... Pk are to be traced. The effect of multiple calls on TRACER is cumulative, so

```
(TRACER A B C)
(TRACER D)
```

has the same result as

```
(TRACER A B C D)
```

The function UNTRACER undoes the effect of TRACER. Executing

```
(UNTRACER P Q R)
```

marks P, Q, and R so that they will no longer be traced. Executing the function without arguments undoes the effect of all previous calls of TRACER:

```
(UNTRACER)
```

The interpreter goes into a special tracing mode when there are no productions marked for tracing. In this mode, it prints trace information about all productions, including the unnamed productions.

The defaults are that no productions are marked for tracing and that the setting of TRACE is LEVEL1.

4.5.3 Refreshing Matched Working Memory Elements

The OPS4 interpreter has a facility for automatically keeping recent the information which is in use. If the switch LHS-REFRESH is set to ON, the interpreter will <ADD> each element matched by the LHS of each fired production. The <ADD>s will be treated as if they occurred at the right end of the RHS, so the actions in the RHS will take precedence over these automatic ones. Setting the switch to OFF will inhibit these automatic actions. The default setting is OFF.

4.5.4 Removing Old Working Memory Elements

Since it is sometimes inconvenient in a production system to handle deleting of old information from working memory, the OPS4 interpreter has a facility for automatically deleting old information. If the switch RETIRE-AT is set to an integer, k, the interpreter will examine working memory at intervals and delete all elements that were <ADD>ed (or <REASSERT>ed) more than k action times in the past. The value of this switch must be an integer. The default is a large enough value that this automatic deletion will never occur.

4.5.5 Restarting the Production System

The switch RESTART allows the user to specify that the production system is to be restarted automatically when the conflict set becomes empty. If the switch is set to ON, when the conflict set empties, the interpreter will add the atom RESTART to working memory and perform another recognize-act cycle. The production system should contain a production which is sensitive to this atom and which will deposit the necessary data elements to cause processing to resume. To reduce the possibility of an infinite loop, the interpreter will not add RESTART if it is already present in working memory. The default setting of this switch is OFF.

4.6 Examining Working Memory

Two functions are provided which allow the user to examine the contents of working memory. The function WM, which takes no arguments, builds and returns a list of the elements in the memory. To use this function, the following is typed.

```
(WM)
```

The function PP-WM, which also takes no arguments, prints the contents of working memory in a readable format.

```
(PP-WM)
```

4.7 Examining Production Memory

The function PR allows the user to examine productions. This function takes any number of arguments; the arguments should be the names of productions that were defined while the switch KEEP-LHS was on.

```
(PR P3 P7 P2)
```

PR will print each production in a readable format.

4.8 Modifying Existing Production Systems

The function EXCISER is used to remove productions from production memory. Executing

```
(EXCISER PA PB ... PZ)
```

will remove productions PA, PB, ... PZ. Only named productions can be excised.

The function EDITR is used to edit productions. The user calls the function with a list of production names as its argument.

```
(EDITR P5)
(EDITR P1 P3)
```

The function then calls the LISP editor (see appendix IV) so the user can edit the productions. If the LHS of a production is changed, it will be recompiled automatically when the editor is exited. (Since the RHSs of productions are not compiled by OPS4, no special action is taken if only the RHS is changed.) Only named productions that were defined while KEEP-LHS was set to ON can be edited.

5. Extending the Base Language

The user can extend OPS4 in several ways. He can define new kinds of pattern primitives (e.g., a pattern that will match any one of a list of constants), he can define new kinds of variables (i.e., variables in addition to = variables and # variables), and he can define new structure building functions for the RHS. In addition, some of the interpreter's built-in functions (such as the trace functions) can be replaced. Unfortunately, all these require that the user write a small amount of LISP code, and consequently this section of the manual must assume a knowledge of MACLISP. Appendix III contains a set of functions for those users who do not care to write their own.

5.1 New LHS Functions

This section explains how to define pattern primitives that describe individual subelements. That is, it explains how to define primitives such as one that will match any of a list of constants, or one that will match anything but a given constant, or one that will match only numbers.

Since no LHS functions like these are provided in the base language, it is worthwhile to show how they are used before explaining how they are defined. In the first version of OPS there was a function ANY which would match any one of a list of constants given as an argument to it. The condition

```
( (A (ANY B C D) E) --> . . . )
```

for example, would match any of the following data elements.

```
(A B E)
(A C E)
(A D E)
```

A call on an LHS function always takes this form: a list whose first element is the function name and whose remaining elements (if any) are the arguments of the function. Functions of this kind are very useful; the only reason they are left out of the OPS4 base language is that since they are easy to define, it is better to let the user define a set that does exactly what he wants rather than give him a set that does what the implementor guessed he might want.

To add a new primitive, the user writes a LISP function to perform the appropriate tests and declares to the interpreter that the function is a match function. The function must be an EXPR or SUBR of two arguments.[1] The first argument is the CDR of the list in which the

[1] Throughout this section it will be assumed that the reader is familiar with LISP and LISP jargon like EXPR and SUBR.

function name appears. In the above example, when the function ANY is called, the first argument is bound to

```
(B C D)
```

The second argument is bound to the subelement being tested on this particular call of the function. For example, when the working memory element

```
(A C E)
```

is tested, the second argument is bound to C. The function must be a normal LISP predicate; it must return NIL when the test fails and something other than NIL when the test succeeds. Thus the following function defines ANY.

```
(DEFUN ANY (OPTIONS SUBELEMENT) (MEMQ SUBELEMENT OPTIONS))
```

In addition to writing the code, the user must declare to the LHS compiler that the function's name is no longer to be treated as a constant. The declaration is made using the function PREDICATE. This function accepts any number of names and marks them as being LHS functions. If only the above function was to be declared, the call of PREDICATE would be

```
(PREDICATE ANY)
```

Because OPS4 is a compiler based system, the call on PREDICATE must be made before any productions containing the LHS function are defined.

It should be noted that even if the first argument to the LHS function is not needed, it must be supplied since the LHS compiler assumes it will be there. For example, if one wanted to define an LHS function that would match only numbers, the function would have to be as follows.

```
(DEFUN <NUM> (DUMMY SUBELEMENT) (NUMBERP SUBELEMENT))
```

In addition, when the function is used in the LHS, it must be enclosed in parentheses even though there are no arguments to enclose.

```
(A (<NUM>) E)
```

5.2 New Variable Types

OPS4 allows the user to define variable types in addition to the = and # types. The new types are added through a mechanism much like the mechanism used to add new LHS functions: The user writes one or two LISP functions to perform the tests required for the

variable type and declares to the compiler how the new variables are to be recognized.

In testing bindings for the new variables, as in testing bindings for the # variables, the interpreter compares the proposed binding for the new variable with the proposed binding for the = variable of the same name. Thus the = variable must be present in the LHS or the new variable cannot be used.

The functions the user writes are the ones that compare the proposed binding of the = variable to the proposed binding of the new variable. Hence these functions have two arguments. Whether the user needs to write one or two functions depends on whether the predicate they implement is commutative; if P(x,y)=P(y,x) for all x and y, one function is enough.

The "not equal" predicate which defines the # variable is typical of the commutative predicates. The function for this variable type is:

```
(DEFUN NOT-EQUAL (X Y) (NOT (EQUAL X Y)))
```

An example of a variable type that requires two functions is the < type from OPS2. A variable of this type would match only numbers that were strictly less than the numbers matching the = variable. The two functions that are needed include one which has the binding of the = variable as its first argument and one which has the binding of the = variable as its second argument.[1] They could be defined as follows.

```
(DEFUN EQ1ST (EQVAR LTVAR) (LTVARTEST EQVAR LTVAR))

(DEFUN EQ2ND (LTVAR EQVAR) (LTVARTEST EQVAR LTVAR))

(DEFUN LTVARTEST (E L) (AND (NUMBERP E)(NUMBERP L)(LESSP L E)))
```

In addition to writing the LISP functions, the user must declare that the functions are to be associated with variables of a particular type. The OPS4 function VARIABLE is used for this declaration. The function takes either two or three arguments. The first argument is the character that will signal variables of this new type. The second and third (if there is a third) arguments are the names of the functions to test the predicates. The following would declare the above two variable types.

```
(VARIABLE # NOT-EQUAL)
(VARIABLE < EQ1ST EQ2ND)
```

---

[1] One function will be used for the LHSs in which the = variable occurs before the < variable, and the other function will be used for the LHSs in which the < variable occurs before the = variable.

Note that when two function names are given, the first is the one that has the binding of the = variable as its first argument. The third argument to VARIABLE may be given even when it is not required; the two following calls are both legal and give identical results.

```
(VARIABLE # NOT-EQUAL)
(VARIABLE # NOT-EQUAL NOT-EQUAL)
```

VARIABLE can also be used to redefine existing variable types. For example, if the user wanted a more liberal definition of equality to be used for = variables, he could define a "nearly equal" predicate and then give the following declaration.

```
(VARIABLE = NEARLY-EQUAL)
```

If the compiler is to know which atoms are constants and which are variables, all calls to VARIABLE must be made before the productions containing the variables are compiled. Calls to VARIABLE which just redefine existing variable types can be made at any time, however.

5.3 New RHS Functions

New RHS structure-building functions (like <QUOTE>, etc.) can be added using a mechanism similar to the one used to add new LHS functions and variables. A LISP function is written to perform the structure building and then a declaration is made to the interpreter that the function is a RHS function.

Like other RHS structure-building functions, these new functions can accept any number of arguments and can return any number of results (including zero). To allow this, three conventions must be adhered to:

1. The RHS function must be an FEXPR (or FSUBR). That is, it must accept a list of unevaluated expressions as its argument.

2. The function must return its values encapsulated in a list. Even if there is only one result, the result must be returned in a list.

3. If the RHS function wants its arguments evaluated (as most do) it must call the built in function EVAL-LIST.

Since the function EVAL-LIST is an FSUBR, it is ordinarily APPLYed rather than being called directly. If ARGS is the formal parameter of the RHS function, the following expression is used to evaluate the list.

```
(APPLY 'EVAL-LIST ARGS)
```

The following three examples are taked from the OPS4 source.

```
(DEFUN <QUOTE> FEXPR (L)  L)

(DEFUN <NULL> FEXPR (L) (APPLY 'EVAL-LIST L) NIL)

(DEFUN <EVAL> FEXPR (L) (APPLY 'EVAL-LIST (APPLY 'EVAL-LIST L)))
```

As another example, suppose one needed a function which would accept two numbers and add them together. It might be written as follows.

```
(DEFUN <+> FEXPR (L)
  (PROG (R SUM)
        (SETQ R (APPLY 'EVAL-LIST L))
        (SETQ SUM (PLUS (CAR R) (CADR R)))
        (RETURN (LIST SUM))))
```

The RHS function must be declared to the interpreter using **RHS-FUNCTION**. This function takes any number of arguments; the arguments are the names of the new RHS functions.

```
(RHS-FUNCTION <+>)
```

Since OPS4 does not compile RHSs, this declaration may be given either before or after the productions are compiled.

5.4 Examining Working Memory

The user may on occasion need to examine working memory -- perhaps to implement some RHS function or to implement something like **PP-WM**. The function **MAPWM** is provided as an efficient way to accomplish this. **MAPWM** is a SUBR of one argument. When it is called, this argument should be bound to a function of one argument; **MAPWM** will call this argument function once for each element in working memory, passing to the function a dotted pair:

(*element* . *time-of-adding-this-element*)

5.5 New Trace Functions

If the user is unhappy with the supplied trace functions (**LEVEL1** and **LEVEL2**) he can write his own functions and activate them with **SWITCHES** just as the supplied functions are activated. The activated function will be called on each cycle to print the information deemed important by the user. A trace function must be an EXPR or SUBR of four arguments. The first argument will be bound to the name of the production that fires (or a list containing the RHS and some information about variable bindings if the production has no name). The second argument will be bound to the list of working memory elements that the production

matched. The third will be bound to a list of the elements added by the production, and the fourth to a list of the elements deleted from working memory by the production.

The function CYCLE-COUNT, which takes no arguments can be called to get the count of the number of production firings that have occurred. The function ACTION-COUNT, which also takes no arguments, can be called to get the count of the number of actions that have occurred.

5.6 Tracing <BUILD>

Often the user will want to monitor the adding of productions by <BUILD>. The interpreter provides a mechanism for doing so which is similar to the mechanism for tracing execution of the system. The user writes a function to monitor <BUILD> and then activates it with SWITCHES; if the function is named F, he will execute

```
(SWITCHES BUILD-TRACE F)
```

The function to trace <BUILD> will be called each time <BUILD> is executed, just after the new production is compiled. The tracing function should be an EXPR or SUBR of one argument. That argument will be bound to the name of the new production when the function is called. The OPS4 function MATRIX can be used to retrieve the body of the production. MATRIX is a SUBR of one argument. If that argument is bound to the name of a production which was defined while the switch KEEP-LHS was on, MATRIX will return the production; otherwise it will return NIL.

I. A Sample Production System

The following is the transcription of a session with OPS4. Commentary has been added in italics. Perhaps it should be noted that although this shows the production system being entered directly by the user, this is not the normal mode of operation in OPS4; normally the production system is contained in one or more files which the user loads using SLURP (see appendix IV). It should also be noted that OPS4 productions are usually more complex than the productions shown here; these are simple because they are examples.

```
.R OPS4
OPS4
(CREATED 5 - 5 - 79)
```

The user starts OPS4, which prints an acknowledgment and then waits for input. The user then begins defining the system.

```
(SWITCHES
        LHS-REFRESH OFF
        RETIRE-AT 100
        TRACE NIL
        BUILD-TRACE NIL
        KEEP-LHS OFF
        COMPILE-PRIORITY ORDER)
SET                          The interpreter responds after each input.

 (DEFUN <+1> FEXPR (L)
  (PROG (X)
        (COMMENT THIS FUNCTION ADDS 1 TO ITS ARGUMENT)
        (SETQ X (APPLY 'EVAL-LIST L))
        (RETURN (LIST (ADD1 (CAR X))))))
<+1>

 (DEFUN <-1> FEXPR (L)
  (PROG (X)
        (COMMENT THIS FUNCTION SUBTRACTS 1 FROM ITS ARGUMENT)
        (SETQ X (APPLY 'EVAL-LIST L))
        (RETURN (LIST (SUB1 (CAR X))))))
<-1>

(RHS-FUNCTION <+1> <-1>)
(<+1> <-1>)

(SYSTEM
START   ((=A + =B) --> (COUNT =A =B))
COUNT   ((COUNT =A =B) --> (COUNT (<+1> =A) (<-1> =B)))
```

```
HALT     ((=A + =B) (COUNT =R 0) & =Z
           -->
           (<DELETE> =Z)
           (<BUILD> ((=A + =B)
                     -->
                     (<QUOTE> (<WRITE>))
                     ((<QUOTE> <WRITE>) =A + =B = =R)
                     (<QUOTE> (<WRITE>) (<WRITE>))
                     ((<QUOTE> <DELETE>) (=A + =B)))))
)
###NIL                    A '#' is printed as each production is compiled.
```

The user resets the switch KEEP-LHS because he does not want the LHSs of the built productions to be discarded.

```
(SWITCHES KEEP-LHS ON)
SET
```

The following input asks the production system to add two numbers. Note that the system has four productions when it halts. This is because it learns the answer to this problem.

```
(START ((9 + 9)))


 9 + 9 = 18


END -- NO PRODUCTION TRUE
 4 productions (14 / 20 nodes) (7 / 13 features)
 12 firings (14 RHS actions)
 6.4166667 mean working memory size (11 maximum)
 1.08333333 mean conflict set size (2 maximum)
 1.08333333 mean token memory size (2 maximum)
 0.08 seconds (6.67 msec per firing) (5.71 msec per action)
NIL
```

The system is able to handle more than one problem at a time.

```
(START (8 + 3) (6 + 4))


 8 + 3 = 11



 6 + 4 = 10
```

```
END -- NO PRODUCTION TRUE
 6 productions (22 / 30 nodes) (13 / 21 features)
 13 firings (27 RHS actions)
 6.2307692 mean working memory size (10 maximum)
 1.6923077 mean conflict set size (4 maximum)
 1.61538461 mean token memory size (3 maximum)
 0.107 seconds (8.23 msec per firing) (3.96 msec per action)
NIL

(START ((9 + 9)))


 9 + 9 = 18


END -- NO PRODUCTION TRUE
 6 productions (22 / 30 nodes) (13 / 21 features)
 1 firings (11 RHS actions)
 1.0 mean working memory size (1 maximum)
 2.0 mean conflict set size (2 maximum)
 1.0 mean token memory size (1 maximum)
 0.034 seconds (34.0 msec per firing) (3.09 msec per action)
NIL
```

Resetting the trace function will show how the system handles the problems. One new problem and two previously given problems are posed below.

```
(SWITCHES TRACE LEVEL1)
SET

(START (6 + 4) (3 + 2) (9 + 9))


 6 + 4 = 10

        1. G0004        2. START        3. COUNT        4. COUNT
 5. HALT

 3 + 2 = 5

        6. G0005

 9 + 9 = 18
```

```
        7. G0002
END -- NO PRODUCTION TRUE
 7 productions (26 / 35 nodes) (16 / 25 features)
 7 firings (11 RHS actions)
 3.7142857 mean working memory size (5 maximum)
 3.42857143 mean conflict set size (5 maximum)
 2.14285713 mean token memory size (3 maximum)
 0.103 seconds (14.71 msec per firing) (9.36 msec per action)
NIL
```

Four productions have been learned.

```
(PR G0002 G0003 G0004 G0005)


G0002
((9 + 9) --> (<WRITE>) (<WRITE> 9 + 9 = 18) (<WRITE>) (<WRITE>)
(<DELETE> (9 + 9)))

G0003
((8 + 3) --> (<WRITE>) (<WRITE> 8 + 3 = 11) (<WRITE>) (<WRITE>)
(<DELETE> (8 + 3)))

G0004
((6 + 4) --> (<WRITE>) (<WRITE> 6 + 4 = 10) (<WRITE>) (<WRITE>)
(<DELETE> (6 + 4)))

G0005
((3 + 2) --> (<WRITE>) (<WRITE> 3 + 2 = 5) (<WRITE>) (<WRITE>)
(<DELETE> (3 + 2)))
```

II. The OPS4 Read Routine

The read routine used by <READ>, which is not the usual LISP read routine, processes characters one at a time, building atoms and lists, and executing functions as indicated by the characters. The characters are divided into several classes, and each class receives different treatment. The classes and what the routine does:

1. Escape (ASCII 27). Make a list of the preceding objects on the line and execute the list. Discard the line and the result of its execution.

2. Left parenthesis ("("). If the characters of a symbolic or numeric atom are being accumulated, terminate the accumulation and make an atom of the accumulated characters. Begin the construction of a list.

3. Right parenthesis (")"). If a list is being constructed, terminate the list; if not, return from <READ>.

4. Period ("."). If a numeric atom is being accumulated, add this character; otherwise treat this as a one-character identifier.

5. One-character identifiers ("!" "," "." ":" ";" "?" "[" "]"). If an atom is being accumulated, terminate the accumulation and make an atom of the accumulated characters. Make another atom of this character.

6. Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). If a literal or numeric atom is being accumulated, add this character to the atom; otherwise begin a numeric atom.

7. Break characters (space, line feed, carriage return, horizontal tab, vertical tab, and form feed). If a literal or numeric atom is being accumulated, terminate the accumulation and make an atom of the accumulated characters. Discard this character.

8. Ignored characters (ASCII 0-8, 14-26, and 127). Discard this character, but take no other action. Note that while OPS4 ignores all of these characters, the monitor does not. Control-C, rubout, etc. produce their usual effects.

9. Identifier characters (all other characters). If an atom is being accumulated, add this to the atom; otherwise begin accumulation of an atom.

III. A Basic Set of Extensions

The following is a copy of the file USER.LSP from area A110PS99 on the CMU-10A.

```
; This is USER.LSP[A110PS99], a basic set of extensions to OPS4
; provided for users who do not wish to provide their own
; extensions.  It contains the necessary calls to define seven
; LHS functions, two new variable types, and five RHS functions.

; The two new variable types are the < and > types.  The <
; variable will match numeric atoms that are equal to or less
; than the atoms matching the corresponding = variable.  The >
; variables will match numeric atoms that are equal to or
; greater than the atoms matching the corresponding = variable.
; In order not to slow the execution, the functions that define
; the variable types are compiled.  (This file contains a
; command to read in the compiled code.)  The uncompiled
; functions are:
;
;       (DEFUN L1ST (L G)
;         (AND (NUMBERP G) (NUMBERP L) (NOT (LESSP G L))))
;
;       (DEFUN L2ND (G L)
;         (AND (NUMBERP G) (NUMBERP L) (NOT (LESSP G L))))

; The seven new LHS functions are <<, >>, <=, >=, <ANY>,
; <NOTANY>, and <TYPE>.  The first four of these are numeric
; functions which take exactly one argument.  << matches numbers
; that are strictly smaller than its argument.  >> matches
; numbers that are strictly greater than its argument.  <=
; matches numbers that are equal to or smaller than its
; argument.  >= matches numbers that are equal to or greater
; than its argument.  The other three functions will accept more
; than one argument.  <ANY> will match any atom which is equal
; to one of its arguments.  <NOTANY> will match any atom except
; one which is equal to one of its arguments.  <TYPE> will match
; any element whose type is listed in its argument.  This
; function accepts for arguments one or more of ATOM, LIST,
; NUMBER, and SYMBOL.  Thus (<TYPE> SYMBOL) will match symbolic
; atoms only, and (<TYPE> LIST NUMBER) will match anything but
; symbolic atoms.  Note that <ANY> can be used as a LHS quote:
; in order to get an atom like & or = in the LHS, one puts it as
; an argument to <ANY>.  For example, the condition element
; (<ANY> =X) will match the atom =X.  The definitions of these
; functions:
```

```
;
;       (DEFUN << (CONST ELEM)
;         (AND (NUMBERP ELEM) (LESSP ELEM (CAR CONST))))
;
;       (DEFUN >> (CONST ELEM)
;         (AND (NUMBERP ELEM) (GREATERP ELEM (CAR CONST))))
;
;       (DEFUN >= (CONST ELEM)
;         (AND (NUMBERP ELEM) (NOT (LESSP ELEM (CAR CONST)))))
;
;       (DEFUN <= (CONST ELEM)
;         (AND (NUMBERP ELEM) (NOT (GREATERP ELEM (CAR CONST)))))
;
;       (DEFUN <ANY> (ARG ELM) (MEMBER ELM ARG))
;
;       (DEFUN <NOTANY> (ARG ELM) (NOT (MEMBER ELM ARG)))
;
;       (DEFUN <TYPE> (TYPES E)
;         (OR (AND (MEMQ 'ATOM TYPES) (ATOM E))
;             (AND (MEMQ 'LIST TYPES) (NOT (ATOM E)))
;             (AND (MEMQ 'NUMBER TYPES) (NUMBERP E))
;             (AND (MEMQ 'SYMBOL TYPES)
;                  (ATOM E) (NOT (NUMBERP E)))))

; The five RHS functions, <+>, <->, <*>, <//>, and <^>, are
; arithmetic functions. <+>, <->, <*>, and <//> take any number
; of arguments. <^> takes exactly two arguments. <+> adds its
; arguments together. <*> multiplies its arguments. <->
; subtracts the second through last arguments (if there is more
; than one argument) from the first. <//> divides the first
; argument by the second through last arguments. <^> raises the
; first argument to the power of the second argument. All the
; functions are capable of mixed mode arithmetic. They are
; defined as follows.
;
;       (DEFUN <+> FEXPR (X)
;         (LIST (APPLY 'PLUS (APPLY 'EVAL-LIST X))))
;
;       (DEFUN <-> FEXPR (X)
;         (LIST (APPLY 'DIFFERENCE (APPLY 'EVAL-LIST X))))
;
;       (DEFUN <*> FEXPR (X)
;         (LIST (APPLY 'TIMES (APPLY 'EVAL-LIST X))))
;
```

```
;       (DEFUN <//> FEXPR (X)
;         (LIST (APPLY 'QUOTIENT (APPLY 'EVAL-LIST X))))
;
;       (DEFUN <^> FEXPR (X)
;         (LIST (APPLY 'EXPT (APPLY 'EVAL-LIST X))))

(FASLOAD (DSK A110PS99) USER FAS)     ; Load the compiled code

(VARIABLE < L2ND L1ST)              ; Make the declarations
(VARIABLE > L1ST L2ND)

(PREDICATE << >> <= >= <ANY> <NOTANY> <TYPE>)

(RHS-FUNCTION <+> <-> <*> <//>)
```

IV. MACLISP at CMU

The documentation for the MACLISP language is [6]. Since the CMU Computer Science Department is in the process of importing MACLISP from MIT, the documentation for the support packages is necessarily somewhat unstable at the moment. Pointers to the current documentation can be found in the file MACLSP.DIR[C380ML5P]. The OPS4 user will probably be interested in the EDIT and SLURP packages. The EDIT package contains a set of functions for editing LISP objects (such as productions). The SLURP package contains functions for performing input and output.

References

1. C. Forgy and J. McDermott. The OPS Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1976.

2. C. Forgy and J. McDermott. OPS, A Domain-independent Production System. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 933-939. Cambridge, MA.

3. C. Forgy and J. McDermott. The OPS2 Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1978.

4. C. Forgy. *On the Efficient Implementation of Production Systems.* Ph.D. Th., Carnegie-Mellon University, February 1979.

5. J. McDermott and C. Forgy. Production System Conflict Resolution Strategies. In D. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 177-199.

6. MIT AI Lab and Project MAC. MACLISP Manual. Massachusetts Institute of Technology, 1978.

7. A. Newell and J. McDermott. PSG Manual. Department of Computer Science, Carnegie-Mellon University, 1975.

8. M. Rychener. *Production Systems as a Programming Language for Artificial Intelligence* Applications. Ph.D. Th., Carnegie-Mellon University, December 1976.

9. L. Siklossy. *Let's Talk Lisp.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

10. D. A. Waterman and F. Hayes-Roth (eds). *Pattern-Directed Inference Systems.* Academic Press, New York, 1978.

11. C. Weissman. *Lisp 1.5 Primer.* Dickenson, Belmont, CA, 1967.

Index