CMU-CS-81-135

OPS5 User's Manual*

July 1981

Charles L. Forgy
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract: This is a combination introductory and reference manual for OPS5, a programming language for production systems. OPS5 is used primarily for applications in the areas of artificial intelligence, cognitive psychology, and expert systems. OPS5 interpreters have been implemented in LISP and BLISS.

*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

OPS5 is a member of the class of programming languages known as production systems. It is used primarily for applications in the areas of artificial intelligence, expert systems, and cognitive psychology. This manual is a combination introductory and reference manual for OPS5. The rest of Section 1 provides an overview of the language. Sections 2 through 8 describe the language and its interpreter in detail. To allow the new user to read the manual straight through, the material has been organized in a top-down fashion. To allow the experienced user to answer detailed questions quickly, the manual has been divided into short sections describing individual features of the language, and an index has been provided.

Three interpreters for OPS5 have been written, one in BLISS [1], one in MACLISP [9], and one in FRANZ LISP [3]. As could be expected, there are a few incompatibilities between the interpreters. The manual points out the differences between the three interpreters.

1.1. The Production System Architecture

A production system is a program composed entirely of conditional statements called *productions*. These productions operate on expressions stored in a global data base called *working memory*. The productions are stored in a separate memory called *production memory*. The production is similar to the If-Then statement of conventional programming languages: a production that contains n conditions C_1 through C_n and m actions A_1 through A_m means

> When working memory is such that C_1 through C_n are true simultaneously,
> then actions A_1 through A_m should be executed.

The condition part of a production is usually called its *LHS* (left hand side), and the action part is called its *RHS* (right hand side).

The production system interpreter executes a production system by performing a sequence of operations called the *recognize-act cycle*:

1. [Match] Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.

2. [Conflict resolution] Select one production with a satisfied LHS. If no productions have satisfied LHSs, halt the interpreter.

3. [Act] Perform the actions specified in the RHS of the selected production.

4. Go to step 1.

Production systems differ from conventional programs in two major respects. The first is that the

production system uses a different method for encoding the state of a computation. A conventional program encodes state by assigning values to local and global variables. A production system encodes state by putting expressions in the system's global working memory. The other difference between production systems and conventional programs is the way flow of control is managed. A conventional program uses sequential execution of statements plus a number of control constructs including subroutine calls, loops, and conditional branching. A production system uses LHS satisfaction. Each production's LHS is a description of the states in which the production is applicable; the LHS becomes true when there is some information in working memory that the production can process. When the interpreter performs the match process, it is in effect searching for a production that knows how to process the data that is in working memory. When it finds that production and executes its RHS, working memory is changed, and so on the next cycle, the interpreter performs the match again to find a production that can handle the new data.

1.2. OPS5's Working Memory

In OPS5, the most commonly used representation for information in working memory is the attribute-value representation. This representation is oriented towards describing objects and relations among objects; that is, even though it (like most representations) can be used for many other purposes, it is most naturally used to describe objects and relations. In this representation, every element in working memory consists of an object and a collection of associated attribute-value pairs. For example, in this representation, a single working memory element might indicate that block1 is a red block weighing 500 grams, measuring 100 mm on a side. The element would be

```
(block
    ↑name       block1
    ↑color      red
    ↑mass       500
    ↑length     100
    ↑width      100
    ↑height     100)
```

As this shows, an element consists of a class name (`block` in this case) followed by some number of attributes and values, with everything enclosed in parentheses. Attributes are distinguished by being preceded with the operator ↑.

1.3. OPS5's Production Memory

The LHS of a production consists of one or more patterns; i.e., one or more expressions that describe working memory elements. During the match part of the recognize-act cycle, the interpreter compares each pattern with the elements in working memory to determine if the pattern matches any of them. The pattern is considered satisfied if it matches at least one element. If all the patterns in a production's LHS are satisfied, the LHS is satisfied.

Patterns are abstract representations of working memory elements. One way a pattern can be an abstraction of a working memory element is to contain fewer attributes and values than the element. Such a pattern will match any working memory element that contains the information in the pattern. (It does not matter how much more information the working memory element contains.) Thus the pattern

```
(block    ↑color red)
```

would match the working memory element

```
(block
    ↑name       block1
    ↑color      red
    ↑mass       500
    ↑length     100
    ↑width      100
    ↑height     100)
```

Another way a pattern can be an abstraction of a working memory element is to contain incompletely specified values. OPS5 provides special pattern operators that can be used to specify values at various levels of detail. The most important operator is the variable. A variable is any symbol that begins with the character < and ends with the character > -- for example, **<x>** or **<status>**. A variable in a pattern may match anything, but if a variable occurs more than once in a production, it must match the same value everywhere. Thus if a cube is defined to be a block whose three sides are the same length, the following pattern will match only cubes.

```
(block    ↑length <x>    ↑width <x>    ↑height <x>)
```

The RHS of a production consists of an unconditional sequence of actions. OPS5's set of action types includes actions to manipulate working memory, actions to perform input and output, actions to add new productions to production memory, and others. The most important of the actions are the ones to manipulate working memory. The action **make** is used to create and add new elements. A **make** action consists of an open parenthesis, the symbol **make**, a description of the element to create, and a close parenthesis. The description of the element is similar in form to the patterns in the LHS. For example, the following would create the element for block1 shown above.

```
(make block
    ↑name       block1
    ↑color      red
    ↑mass       500
    ↑length     100
    ↑width      100
    ↑height     100)
```

The action **remove** is used to delete elements from working memory. A **remove** action consists of an open parenthesis, the symbol **remove**, a pointer to the element to delete, and a close parenthesis. The following for example would delete the element matching the third pattern of the production's LHS.

```
(remove 3)
```

The action **modify** is used to change one or more values of an existing element. A **modify** action consists of an open parenthesis, the symbol **modify**, a pointer to the element to change, a description of the changes to make, and a close parenthesis. The following for example would change the **status** of the element matching the first pattern in the LHS to **satisfied**.

```
(modify 1  ↑status satisfied)
```

A production consists of an open parenthesis, the symbol **p**, a name, the LHS of the production, the symbol **-->**, the RHS, and a close parenthesis. The following is a typical (though quite small) OPS5 production. The text after the semicolon on each line is a comment.

```
(p find-colored-block
    (goal                          ; If there is a goal
        ↑status active             ; which is active
        ↑type find                 ; to find
        ↑object block              ; a block
        ↑color <z>)                ; of a certain color
    (block                         ; And there is a block
        ↑color <z>                 ; of that color
        ↑name <block>)
    -->
    (make result                   ; Then make an element
        ↑pointer <block>)          ; to point to the block
    (modify 1                      ; And change the goal
        ↑status satisfied))        ; marking it satisfied
```

1.4. The OPS5 Lexical System

The input to OPS5 is completely free format. Spaces, tabs, and new lines may be used at will to improve the readability of productions and working memory elements; the interpreter uses the parentheses to determine where units begin and end. In addition, comments like those shown above may be used anywhere; when the interpreter reads a line containing a semicolon, it discards everything from the semicolon to the end of the line. The above production could also have been written

```
(p find-colored-block
    (goal    ↑status active    ↑type find    ↑object block
             ↑color <z>)
    (block    ↑color <z>    ↑name <block>)
    -->
    (make result    ↑pointer <block>)
    (modify 1    ↑status satisfied))
```

1.5. Acknowledgements

The first language in the OPS family [4, 5] was designed in 1975 at Carnegie-Mellon University by Charles Forgy, John McDermott, Allen Newell, and Michael Rychener. The design of the language was influenced by earlier production systems languages, including PSG [10] and PSNLST [11]. Since 1975 OPS has been

revised several times as better representations and more efficient interpreters have been developed [6, 7, 12]. Many people have contributed to the development of OPS, including the members of the CMU production systems, expert systems, and cognitive psychology groups, as well as the members of Digital Equipment Corporation's expert systems group.

2. Working Memory

Working memory is a set of ordered pairs

<Time tag, Working memory element>

A working memory element is a structure (usually a vector or record) of scalar values. The time tag is a unique numerical identifier that is supplied by the interpreter.

2.1. Organization of Working Memory

OPS5, like most programming languages, provides both scalar (sometimes called atomic) data types and structured data types. The elements in working memory may not be scalars. (However, it is legal to have a structure that contains only a single scalar value.)

The number of elements in working memory varies dynamically at run time. With the LISP-based interpreter, working memory may grow arbitrarily large. With the BLISS-based interpreter, a maximum size for the memory is established when the interpreter is installed; the current limit is 1023 elements.

2.2. Time Tags

Every element in working memory has an associated integer called the element's *time tag*. This integer indicates when the element was created or last modified; the elements with larger time tags were more recently created or modified. No two elements have the same time tag. Time tags are used in conflict resolution, and they are used to designate elements by many of the facilities that communicate with the user (see Section 8.1).

2.3. Scalar Values

OPS5 provides two scalar data types: numbers and symbolic atoms.

2.3.1. Numbers

The numeric type on the LISP-based interpreters for OPS5 includes both floating point and fixed point numbers. (The interpreters will make the appropriate conversions when mixed mode expressions are evaluated.) The BLISS-based interpreter allows only fixed point numbers to be used. Fixed point numbers consist of an optional sign, one or more decimal digits, and an optional decimal point. Valid fixed point numbers include

```
0
0.
-7
-7.
```

A floating point number consists of an optional sign, zero or more decimal digits, a decimal point, zero or more digits after the decimal point, and an optional exponent, consisting of the letter "e" followed by a signed or unsigned integer. The number must include either an exponent or a digit after the decimal point; if it contains neither the interpreter will take it to be an integer. Typical floating point numbers include

```
0.0
.05
6.02e-23
-1.e12
```

The computer on which OPS5 is run determines the legal range for fixed and floating point numbers and the number of digits of precision in floating point numbers.

2.3.2. Symbolic Atoms

A symbolic atom is any sequence of characters that does not constitute a number and that is treated as a single unit by the production system. Examples of symbolic atoms include

```
a
nil
---
4-7-76
```

Some non-printing characters such as escape (ASCII 33 octal) or control-C (ASCII 3 octal) cannot conveniently be used in atom names. In addition, on the BLISS-based interpreters, symbolic atoms must not contain the character '. But with this exception, all printing characters and many non-printing characters such as space and tab can be used.

Some characters will be incorporated into atoms only if they are quoted. If they are used unquoted they are taken to be operators or separators. The characters that need to be quoted include (but are not limited to) space, tab, period, comma, uparrow ("↑"), left and right braces ("{}"), and left and right parentheses ("()"). Different LISP interpreters provide different mechanisms for quoting characters. The best mechanism to use in OPS5 is probably the vertical bar (the character |) because it is understood by all the OPS5 interpreters. In all the interpreters, everything that occurs between two vertical bars constitutes an atom. Thus the atom))) would be entered |)))|.

2.3.3. Case

The MACLISP-based interpreter and the BLISS-based interpreter do case folding; that is, they convert lower case characters to upper case on input. The FRANZ LISP-based interpreter does not do case folding. Thus on that interpreter, p and P are distinct atoms. All commands to the FRANZ LISP-based interpreter must be given in lower case.

2.4. The Standard Structured Types

OPS5 provides two non-scalar data types, plus a mechanism which allows the user to implement other non-scalar types. The standard types are attribute-value elements and vectors.

2.4.1. Attribute-Value Elements

An attribute-value element consists of a class name and some number of attribute-value pairs, with everything enclosed in parentheses. Attributes are symbolic atoms, and values are either scalars or sequences of scalars. An attribute-value element may not contain more than 126 values. The following is a typical element.

```
(goal    ↑status active    ↑type find    ↑object block    ↑color red)
```

The class name of this element is **goal**. Its attributes are **status**, **type**, **object**, and **color**; the corresponding attributes are **active**, **find**, **object**, and **red**. The prefix operator ↑ is used to distinguish attributes from values.

The order in which attribute-value pairs are specified is not significant. Thus this element could also have been written say

```
(goal    ↑color red    ↑object block    ↑status active    ↑type find)
```

2.4.1.1. Declarations

Attribute names must be declared before they can be used. The usual way to declare names is with **literalize**. (Another method is described in Section 2.6.) A **literalize** declaration indicates which attributes will be used in elements of a given class. A declaration consists of the atom **literalize**, a class name, and the attributes for that class, all enclosed in parentheses. For the **goal** shown above, a declaration like the following would be given.

```
(literalize goal
        status
        type
        object
        color)
```

This indicates that elements of class **goal** can have the attributes **status**, **type**, **object**, and **color**.

An attribute may have only one scalar value at a time unless it has appeared in a **vector-attribute** declaration. A vector attribute may have one, two, three, or more values; the only restriction is that the total size of the working memory element may not exceed 126 values. The number of values assigned to a vector attribute may vary dynamically at run time. The declaration consists of the atom **vector-attribute** and one or more attribute names, all enclosed in parentheses. For example, if **contents** was to be made a vector attribute, it would be declared

```
(vector-attribute contents)
```

For an example of a vector attribute, consider a production system to solve the Towers of Hanoi problem. The vector attribute **contents** could be used to indicate which disks were on a given peg.

```
(peg
    ↑name peg2
    ↑contents disk1 disk3 disk4 disk5)
```

Two restrictions apply to vector attributes.

- An element class may not have more than one vector attribute.
- The vector attribute declaration is global. Each attribute is either a scalar attribute everywhere it is used or a vector attribute everywhere it is used. It is not possible for an attribute to be a scalar attribute in one element class and a vector attribute in another.

2.4.1.2. Error Checking

OPS5 does not perform extensive error checking of attribute-value elements. It will permit attributes to be used with element classes they were not declared for, and it will allow the user to treat scalar attributes as vector attributes. It cannot check for errors like these because attribute-value elements are implemented using a general mechanism that is also available to the user (see Section 2.6).

2.4.2. Vector Elements

The vector representation is used for data that needs to be represented as a sequence of symbols. An element in this representation consists of an open parenthesis, a sequence of atoms and numbers, and a close parenthesis. One common use for this representation is to hold input from the user. The element shown below for example might be a command given to a system for algebraic manipulation.

```
(differentiate expression 4 wrt x)
```

Vector working memory elements do not have to be declared. Vectors can vary in length at run time. A vector cannot contain more than 127 values.

2.5. Details of Implementation

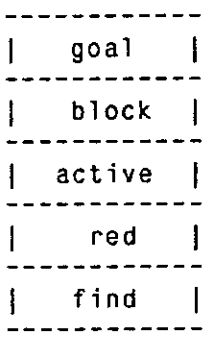
In the OPS5 interpreter, all working memory elements are stored as ordered lists or vectors of values. Attribute-value representations are implemented by mapping field names into indices. The lists shrink and grow as necessary when the elements are modified. An element may not grow to more than 127 values, however.

2.5.1. Attribute-Value Elements

In an attribute-value element, the class name is stored in the first field of the element, and the value of each attribute is stored in a field that is assigned to the attribute. For example, on one run of a production system **object** might be assigned 2, **status** assigned 3, **color** assigned 4, and **type** assigned 5. Then the working memory element

```
(goal    ↑status active    ↑type find    ↑object block    ↑color red)
```

would be stored internally



Each rectangle here represents one field in the working memory element.

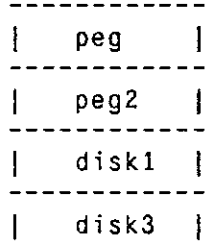
The assignment of field numbers to attributes is performed by the interpreter when the **literalize** declarations are processed. The number assigned to each attribute is global; if attribute A has number N in one element class, it will have number N in every class it occurs in.

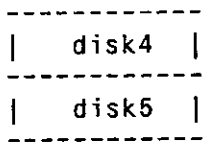
2.5.2. Vector Attributes

Vector attributes are implemented by assigning the vector attribute a higher number than any other attribute in the class. (If a vector attribute is used in more than one class, it is assigned a number that is higher than any other attribute in any of the classes.) This allows the tail of the element to be dedicated to the vector attribute. The values of the attribute consist of the value in its assigned field plus all the succeeding values to the end of the element. Thus if **name** was assigned 2 and **contents** was assigned 3, then the element

```
(peg
    ↑name peg2
    ↑contents disk1 disk3 disk4 disk5)
```

would be stored





Since OPS5 allows elements to grow and shrink dynamically at run time, the number of values assigned to a vector attribute can vary dynamically.

2.5.3. The Operator ↑

Since attributes are mapped by the interpreter into field numbers, the operator ↑ is essentially an index operator. To interpret

```
↑att
```

OPS5 converts `att` into an integer, and then uses that integer to index into the working memory element. The operator ↑ can also be used with numeric arguments. For example,

```
↑7
```

This designates the seventh value in an element.

Although it is common practice to write ↑ immediately adjacent to the attribute (or number) this is not required. Blanks, tabs, and other non-printing characters can be put between the ↑ and the attribute.

2.5.4. Default Values

In OPS5 it is legal to read the value in a field that has not received a value. (Sections 4 and 5 explain how productions read values from elements.) By default, every field in an element has the value `nil` until the production system changes it to something else. For consistency, the interpreter also returns `nil` if the production system reads beyond the end of the element (e.g., reading field 20 of an element that has values only in fields 1 through 10). It is not legal, however, to read non-existent fields; an attempt to read fields less than 1 or greater than 127 is an error.

2.6. User-Defined Representations

The declaration `literal` is provided to allow users to implement their own representations. The declaration is used to assign numbers to attributes. A `literal` declaration consists of an open parenthesis, the symbol `literal`, some number of triples of the form

attribute = *number*

followed by a close parenthesis. For example

```
(literal
    status = 2
    type = 3
    object = 4
    color = 5)
```

If a production system contains both `literal` and `literalize` declarations, the interpreter will process the `literal` declarations first (even if they are not written first). Then, if is possible, it will use the explicit `literal` assignments for the attributes that occur in both `literal` and `literalize` declarations. If it is not possible to accommodate the explicit assignments, an error message will be printed.

The declaration `literal` should be used only when `literalize` cannot be used, because `literal` has two severe limitations. First, it is easy to make a mistake with `literal` and assign the same number to two attributes that were supposed to be distinct. This can cause obscure bugs in the production system. Second, `literal` does not provide enough information for the working memory element printer to work properly. When `literalize` is used, elements are printed in attribute-value format; when `literal` is used, elements must be printed as lists.

3. Production Memory

An OPS5 production memory consists of a set of productions.

3.1. Organization of the Memory

There is no structure imposed on production memory. In particular, the productions are not grouped into subroutines; any production can fire at any time. Furthermore, the order in which productions are entered into the system is not important.

Production memory can contain arbitrarily many productions. The only limit is the amount of memory available on the computer to store the productions.

3.2. Production Names

The name of a production must be a symbolic atom. The atom `nil` should not be used.

Two productions may not have the same name. If the user enters a production that has the same name as an existing production, the existing production is removed from production memory.

3.3. The Production

A production consists of (1) an open parenthesis, (2) the symbol `p`, (3) the name of the production, (4) the LHS of the production, (5) the symbol `-->`, (6) the RHS of the production, and (7) a close parenthesis. The production shown in Section 1.3 is typical.

```
(p find-colored-block
    (goal    ↑status active    ↑type find    ↑object block
             ↑color <z>)
    (block    ↑color <z>    ↑name <block>)
    -->
    (make result    ↑pointer <block>)
    (modify 1    ↑status satisfied))
```

4. The LHS

As Section 3.3 explained, the LHS of a production is everything between the production's name and the symbol `-->`. An LHS is a collection of patterns called *condition elements.*

4.1. The Condition Element

A condition element is a pattern to match a working memory element; it consists of an open parenthesis, some number of forms to specify attributes and values, and a close parenthesis. The forms are called condition element *terms.* A condition element is considered to match a working memory element if every term in the condition element matches the corresponding part of the working memory element.

4.1.1. Terms

A condition element term can be either

- The operator ↑ followed by an attribute and a specification of a value (OPS5 provides a variety of ways to specify values in condition elements -- see below)
- The operator ↑ followed by a number and a specification of a value, or
- Just a specification of a value.

4.1.2. The Operator ↑

The interpreter applies three rules to determine which value in a working memory element a term should be compared to.

1. If the term contains ↑ and an attribute name or a number, compare the term to the value in the indicated field in the working memory element.
2. If a term Ta that contains no ↑ is preceded by another term Tp, move to the position immediately after the position used for Tp, and compare Ta to the value there.
3. If a term that contains no ↑ is not preceded by another term, compare the term to the value in the first field in the working memory element.

To see how these rules work with vector and attribute-value representations, consider the following condition elements. In these elements, a1 and a2 are attributes, and v1 through v6 are values.

```
(v1 v2 v3)
(v4 ↑a1 v5 ↑a2 v6)
```

In the first condition element, by Rule 3, when v1 is processed it will be compared to the first value in the working memory element. By Rule 2, v2 will be compared to the second value, and by the same rule, v3 will be compared to the third value. Thus the rules cause vector style condition elements to be processed

correctly. In the second condition element, by Rule 3, v4 will be compared to the first value in the working memory element. By Rule 1, v5 will be compared to the value in the field for a1, and by the same rule, v6 will be compared to the value in the field for a2. Thus the rules also cause attribute-value style condition elements to be processed properly.

4.1.3. Values

The values in condition element terms can be specified as constants or by using the pattern operators provided by OPS5.

4.1.3.1. Constants

Symbolic atoms and numbers may occur in condition elements as well as in working memory elements. A symbolic atom in a condition element matches a symbolic atom in a working memory element if the sequences of characters composing the two elements are identical. A number in a condition element matches a number in a working memory element if the algebraic difference of the two is zero.

4.1.3.2. Variables

A variable in OPS5 is any symbolic atom whose first character is < and whose final character is >; for example, `<x>` or `<status>`. A variable will match any symbolic atom or number, but if a variable occurs more than once in an LHS, all occurrences must match the same value. A variable is said to be *bound* to the value it matches.

4.1.3.3. Disjunctions

The brackets << and >> specify that any of the contained values is acceptable as a match. Thus the following

```
<< nil 17 >>
```

will match either `nil` or `17`.

These brackets implicitly quote the symbols that they contain. Thus the following

```
<< <x> <y> >>
```

would match not the binding of `<x>` or `<y>`, but rather the symbols `<x>` or `<y>`. The brackets will also quote ↑ and all the pattern operators that are described below.

4.1.3.4. The Operator //

The prefix operator `//` is used to quote single symbols in condition elements. For example, to match the symbol `<x>` rather than the binding of the variable `<x>`, the following is used.

```
// <x>
```

For another example, to match the symbol // the following is used

```
// //
```

This operator can also be used to quote ↑, the brackets << and >>, and the other operators defined below.

4.1.3.5. Predicates

OPS5 has seven prefix operators called predicates which are used with constants and variables. The predicates are

```
=
<>
<=>
<
<=
>=
>
```

The first occurrence of a variable cannot be preceded by any predicate other than =. (This restriction is necessary because the first occurrence of the variable establishes the binding for the variable.)

The predicate <> is the not-equal predicate. If val1 is a variable or constant,

```
<> val1
```

will match any value except the values that are matched by

```
val1
```

The predicate = is provided only for completeness; if val2 is a constant or variable

```
= val2
```

is exactly equivalent to

```
val2
```

The predicate <=> is the same type predicate. If val3 is a number or a variable bound to a number,

```
<=> val3
```

will match any number. If val4 is a symbolic atom or a variable bound to a symbolic atom,

```
<=> val4
```

will match any symbolic atom.

The remaining predicates, <, <=, >=, and > are used only with numbers and with variables that are bound to numbers. They match, respectively, numbers that are less than, less than or equal to, greater than or equal to, or greater than the value in the condition element term. For example,

```
< 0
```

will match any negative number.

4.1.3.6. Conjunctions

The braces { and } are used to indicate that a value in a working memory element must match several things simultaneously. For example, to indicate that a value must be greater than zero, but less than ten, the following would be used.

```
{> 0  < 10}
```

Braces may contain constants, variables, either of these preceded by predicates, the operator //, and the brackets << and >>.

Braces are often used with variables. The braces allow specifying some restrictions on a value and binding a variable to the value that meets the restrictions. For example,

```
{<< a b c d >> <x>}
```

Will match a, b, c, or d and bind the value that is matched to <x>. As another example,

```
{<y> <> <x>}
```

will match anything that is not equal to the current binding of <x> and bind the value that is matched to <y>.

As a limiting case, empty braces place no restrictions on the value matched. Thus they can be used as place holders in a condition element. For example, the condition element

```
(<x> {} <x>)
```

will match any working memory element whose first and third values are equal, regardless of what the second value is.

4.2. The LHS as a Whole

The condition elements in an LHS may be negated or not, and the non-negated condition elements may have variables bound to them.

4.2.1. Negated and Non-negated Condition Elements

A condition element may be negated by preceding it with the operator -. An LHS consists of one non-negated condition element followed by zero or more negated or non-negated condition elements. An LHS is satisfied when

- There exist working memory elements that match all the non-negated condition elements, and
- There exist no working memory elements that match the negated condition elements.

Thus if P1, P2, and P3 are condition elements, the LHS

```
P1  P2  -P3
```

is satisfied only when working memory contains something matching P1, something matching P2, and

nothing matching P3.

4.2.2. Element Variables

A variable may be bound to the working memory element that matches a non-negated condition element through the use of the {} braces. The condition element and the variable are placed inside the braces; for example

```
{ <c2> (block ↑color <z>) }
```

or

```
{ (block ↑color <z>) <c2> }
```

These two lines are exactly equivalent.

These variables, which are called element variables, are not treated like the other variables. A given element variable can appear only once in an LHS. Thus element variables can only be bound on the LHS; they cannot be tested. An LHS may contain both an ordinary variable and an element variable with the same name; OPS5 will not confuse the two since the contexts they occur in are distinct.

4.2.3. Length of an LHS

On the LISP-based interpreters, LHSs can contain arbitrarily many negated and non-negated condition elements. On the BLISS-based interpreter, there is a limit of sixteen non-negated condition elements per LHS. There is no limit on the number of negated condition elements an LHS may contain, however.

5. The RHS

The RHS of a production is everything in the production after the **-->**. The RHS consists of an unconditional sequence of commands called *actions*. An action consists of an open parenthesis, the action type, the arguments to the action, and a close parenthesis. The actions in the production in Section 1.3 are

```
(make result    ↑pointer <block>)
(modify 1    ↑status satisfied)
```

The action types here are **make** and **modify**; everything else constitutes the arguments to the actions.

OPS5 provides twelve action types: **make**, **remove**, and **modify** to change working memory; **openfile**, **closefile**, and **default** to manipulate files; **write** to output information; **bind** and **cbind** to assign values to variables; **call** to call user-written subroutines; **halt** to cause the interpreter to stop firing productions; and **build** to add productions to production memory. Sections 5.1 and 5.2 explain how the arguments to these actions are evaluated. Section 5.3 describes the actions.

5.1. Element Designators

Some of the actions and functions in OPS5 refer to working memory elements. Working memory elements may be designated either by number or by use of element variables (see Section 4.2.2). If an element variable is used, it refers to the working memory element that it was bound to in the LHS. (Element variables can be bound explicitly in the RHS -- see Section 5.3.11. If the variable has been given an explicit binding, that binding is used.) If a number K is used, it refers to the element matching the Kth non-negated condition element in the LHS. It is important to note that the interpreter does not count negated condition elements when it is evaluating a numeric element designator. Thus in the RHS of the following production

```
(p ex1
        (...)
      - (...)
        {(...) <c>}
    --> ...)
```

The element variable **<c>** and the numeric element designator **2** both refer to the same working memory element -- the one matching the last condition element in the LHS.

5.2. Patterns

Many of the RHS actions take patterns like condition elements as arguments. The **make** action, which is described in Section 5.3.1, is typical; its only argument is a pattern. For instance,

```
(make block    ↑name block1    ↑color red    ↑mass 500    ↑length 100
       ↑width 100    ↑height 100)
```

When the interpreter evaluates a pattern in the RHS, it instantiates the pattern into an element by replacing variables with the values they are bound to, supplying default values for unspecified parts of the element, etc.

The element that results does not necessarily get put into working memory. Some of the actions put the element in working memory; some use it for other purposes and then delete it. The element that is built is called the *result element.*

5.2.1. Terms

An RHS pattern, like a condition element, consists of a sequence of terms. An RHS term can be

- The operator ↑ followed by an attribute and a specification of a value,
- The operator ↑ followed by a number and a specification of a value,
- The operator ↑ followed by a variable and a specification of a value (this is not allowed in the LHS), or
- Just a specification of a value.

5.2.2. Evaluating Terms

In outline, the process of instantiating a pattern is

1. Fill the result element entirely with `nil`.
2. Evaluate each term in the pattern in order from left to right, changing the result element as the term indicates.

5.2.3. The Operator ↑

The interpreter uses three rules to determine which position in the result element a term refers to.[1]

- If a term contains ↑ and an attribute name or a number, move to the indicated field and change its value as the term specifies.
- If a term Ta that does not contain ↑ is preceded by another term Tp, move to the position immediately after the position used for Tp and change its value as Ta specifies.
- If a term that does not contain ↑ is not preceded by any other term, change the first field in the result element as the term specifies.

[1] These rules are like the ones used in processing patterns in the LHS. See section 4.1.2.

5.2.4. Constants

Symbolic atoms and numbers are copied into the result element without change. Thus if

```
(make ... ↑4 nil    ↑5 0 ... )
```

is evaluated, position 4 of the element is set to `nil`, and position 5 to `0`.

5.2.5. Variables

When a variable in an RHS pattern is evaluated, the binding of the variable is copied into the result element. Thus if `<x>` is bound to `nil`, when the following is evaluated

```
(make ... ↑6 <x> ... )
```

position 6 of the element is given the value `nil`.

5.2.6. The Operator //

The symbol `//` is used to keep symbols from being evaluated. If sym is any symbol,

```
// sym
```

causes sym to be placed directly into the result element. Thus if

```
(make ... ↑7 // ↑    ↑8 // <z>    ↑9 // // ... )
```

is evaluated, position 7 is given the value `↑`, position 8 is given the value `<z>`, and position 9 is given the value `//`.

5.2.7. RHS Functions

An RHS function is a subroutine that puts one or more values into the result element. The syntax of an RHS function call is like the syntax of an action: an open parenthesis, the name of the function, the arguments to the function if any, and a close parenthesis.

5.2.7.1. substr

The function `substr` extracts a sequence of values from an existing working memory element and puts the values in the result element. The function takes three arguments. The first argument is an element designator. (See Section 5.1.) This argument indicates which working memory element is to be examined to get the values. The second argument should be an integer, an attribute name, or a variable that is bound to an integer or attribute name. This argument indicates the first value that is to be extracted. The third argument should be an integer, an attribute name, a variable that is bound to an integer or attribute name, or the symbol `inf`. This argument indicates the final value to extract. For example, if `<w>` is bound to `(a b c d e)`, then evaluating

```
(make ... ↑10 (substr <w> 3 3) ... )
```

will cause the atom `c` to be copied into position 10 of the result element. When more than one value is

extracted, the values are placed in contiguous fields in the element; thus

```
(make ... ↑11 (substr <w> 2 4) ... )
```

will cause **b** to be copied into position 11, **c** to be copied into position 12, and **d** to be copied into position 13. The special symbol **inf** indicates that **substr** is to continue taking values until it reaches the end of the element it is extracting them from. Thus

```
(make ... ↑14 (substr <w> 4 inf) ... )
```

will copy **d** into position 14 and **e** into position 15.

The function **substr** can be used to extract information from attribute-value elements, but it should be used carefully. It is legal to call **substr** to copy all the values in a certain range -- for example, to use

```
(substr 3 status object)
```

to copy all the values from the value of **status** to the value of **object** -- but this is a questionable practice. If the interpreter assigns numbers to attributes, the positions of **status** and **object** may vary from run to run; in fact, on some runs **status** may come after **object**. There are two safe uses of **substr** with attribute-value elements however. The first is to extract the value of a particular attribute. If the same attribute name is used for the second and third arguments, **substr** will return just the value of that attribute. For example, the following would be used to copy the **from** value of one element into the **to** field of another.

```
(make ... ↑to (substr <x> from from) ... )
```

The other safe use with attribute-value elements is copying an entire element. For example, executing

```
(make ... ↑1 (substr <z> 1 inf) ... )
```

copies all the values of element **<z>** into the corresponding fields of the result element.

5.2.7.2. genatom

The function **genatom** creates a new symbolic atom and puts it in the result element. This function takes no arguments, so a call on it always has the form **(genatom)**.

5.2.7.3. compute

The function **compute** evaluates arithmetic expressions. The expressions can contain five operators, **+**, **-**, *****, **//**, and ****, which denote respectively addition, subtraction, multiplication, division, and modulus. Standard infix notation is used, but operator precedence is not used; **compute** evaluates the operators from right to left. Parentheses can be used to override the right to left evaluation. Only numbers and variables that are bound to numbers can be used in the expressions. Typical calls on **compute** include

```
(compute <x> + 1)
(compute (<b> * <b>) - 4 * <a> * <c>)
```

5.2.7.4. litval

The function **litval** puts into the result element the number which has been assigned to an attribute name. That is, if **a** is an attribute name, then **(litval a)** determines the number of the field that is used for attribute **a** and puts the number into the result element. The function takes one argument, which normally is an attribute name or a variable which is bound to an attribute name. The function will also accept numbers or variables bound to numbers; when it is called with such an argument, it returns the number.

5.2.7.5. accept

The function **accept** takes input from the user and puts it into the result element. The function takes either one or zero arguments. If it has an argument, the argument must be a symbolic atom or a variable that is bound to a symbolic atom. The following are legal calls on **accept**

```
(accept)
(accept infile)
(accept <x>)
```

If **accept** is called with no arguments, it takes its input from the current default input stream. (See Section 5.3.6.) If it is called with an argument, **accept** takes its input from the file that has been associated with the atom. (See Section 5.3.4.)

The function will read either a single atom or a list. When it reads a list, it strips the parentheses from the list and puts the atoms of the list into the result element. The interpreter determines whether it is to read a list or a single atom by inspecting the first printing character in the input. If the interpreter encounters **(**, it expects to read a list, so it does not stop reading until it reaches **)**. If it encounters any other printing character, it reads only one atom.

If **accept** is asked to read beyond the end of a file, it puts the atom **end-of-file** in the result element. In the LISP-based interpreters, if the end of the file is reached while a list is being read, a LISP error will occur.

5.2.7.6. acceptline

The function **acceptline** is also used to read input. The difference between **accept** and **acceptline** is that the latter always reads exactly one line of input. The function reads everything on the line, removes any parentheses that are there, and puts the atoms into the result element.

This function takes any number of arguments. If the first argument is associated with an input file (see Section 5.3.4) **acceptline** takes the input from that file; otherwise, it takes the input from the current default input stream (see Section 5.3.6). The rest of the arguments are used when a null line is read or when **acceptline** tries to read beyond the end of a file. A null line is a line that contains no characters other than

spaces and tabs. When **acceptline** encounters a null line or the end of a file, it puts its arguments into the result element. (If the first argument is not the name of a file, it is put in the result element along with the other arguments.) Thus when the function

```
(acceptline nothing read)
```

is evaluated, the interpreter will read the default input (assuming that **nothing** is not associated to a file) and then put into the result element either one line of input or the two atoms **nothing** and **read**.

5.3. Actions

The actions in OPS5 are **make**, **remove**, **modify**, **openfile**, **closefile**, **default**, **write**, **call**, **halt**, **bind**, **cbind**, and **build**.

5.3.1. make

The action **make** creates new elements and adds them to working memory. The argument to **make** is an RHS pattern; it is evaluated as described in Section 5.2. A typical example of a **make** action is

```
(make result ↑pointer <block>)
```

If **<block>** was bound to **block1**, this action would add to working memory the element

```
(result ↑pointer block1)
```

A bigger example of **make** was shown before:

```
(make block
    ↑name       block1
    ↑color      red
    ↑mass       500
    ↑length     100
    ↑width      100
    ↑height     100)
```

which puts into working memory the element

```
(block
    ↑name       block1
    ↑color      red
    ↑mass       500
    ↑length     100
    ↑width      100
    ↑height     100)
```

5.3.2. remove

The action **remove** is used to delete elements from working memory. Any number of arguments may be given to **remove**; the arguments must be element designators. When the action is executed, the indicated working memory elements are deleted from working memory. A typical call on **remove** is

```
(remove 1 <c3>)
```

5.3.2.1. Element Designators and remove

Deleting working memory elements does not change the bindings of element variables or of numeric element designators. Thus in the following RHS, the two calls on **substr** return the same value, even though element **<c>** is deleted between the two calls.

```
(... -->
        (make ... (substr <c> 5 10))
        (remove <c>)
        (make ... (substr <c> 5 10)))
```

5.3.2.2. Multiple remove's of an Element

It is legal to call **remove** with the same argument more than once in an RHS. When the interpreter encounters this situation, it executes the first **remove** and then ignores the rest.

5.3.3. modify

The action **modify** is used to change one or more values in an existing working memory element. It takes as arguments a condition element designator and an RHS pattern. It removes the old form of the designated element from working memory, changes it as the pattern specifies, and then puts it back into working memory. For example, when the **modify** in the following production executes

```
(p find-colored-block
    (goal    ↑status active    ↑type find    ↑object block
             ↑color <z>)
    (block    ↑color <z>    ↑name <block>)
    -->
    (make result    ↑pointer <block>)
    (modify 1    ↑status satisfied))
```

it deletes the element that matched the first condition element -- say

```
(goal    ↑status active    ↑type find    ↑object block    ↑color red)
```

and replaces it with a similar element

```
(goal    ↑status satisfied    ↑type find    ↑object block    ↑color red)
```

It is possible to change more than one value in a **modify** action. The following, for example, is a legal action

```
(modify 3    ↑status followed    ↑value <response>    ↑id <newid>)
```

The action **modify** is defined to be equivalent to a **remove** followed by a **make**. The action

(modify *designator pattern*)

does precisely what the two actions

```
(remove designator)
(make (substr designator 1 inf) pattern)
```

would do.[2] Thus the action

```
(modify 3    ↑status followed    ↑value <response>    ↑id <newid>)
```

is equivalent to

```
(remove 3)
(make (substr 3 1 inf)    ↑status followed    ↑value <response>
    ↑id <newid>)
```

5.3.3.1. Element Designators and modify

Modifying elements does not change the bindings of element variables or of numeric element designators. Thus in the following RHS, the two calls on **substr** both return the same result.

```
(... -->
        (make ... (substr <c> 5 10))
        (modify <c> ↑7 nil)
        (make ... (substr <c> 5 10)))
```

5.3.3.2. Multiple modify's of an Element

It is legal to modify an element more than once in an RHS. That is, an RHS like the following is legal.

```
(... -->
        (modify <x> ↑2 0)
        (modify <x> ↑2 1))
```

To understand what happens in this case, recall that **modify** is defined to be equivalent to a **remove** followed by a **make**. Thus this RHS is equivalent to

```
(... -->
        (remove <x>)
        (make (substr <x> 1 inf) ↑2 0)
        (remove <x>)
        (make (substr <x> 1 inf) ↑2 1))
```

As explained in the previous section, the binding of <x> remains unchanged while the RHS executes. Thus the two calls on **make** produce two elements that are identical except for their second subelements. As explained in Section 5.3.2, if **remove** is called more than once with the same argument, the second and later calls have no effect. Thus the second **remove** here is a no op. In short then, the two calls on **modify** result in the original element being deleted from working memory and replaced by two slightly different copies.

5.3.4. openfile

The action **openfile** is used to open files and associate names with the files. The action takes an RHS pattern as its argument. After the pattern is evaluated, the first three fields in the result element should contain values. The first value should be a symbolic atom; this is the name that the production system will use to refer to the file. The second value should be a valid file name for the system on which OPS5 is being

---

[2] If the pattern does not begin with the operator ↑, then it is necessary to put ↑1 between the **substr** and the pattern in **make**.

run. The third value should be either **in** or **out**; this value indicates whether the file is to be opened for input or output. A typical use of **outfile** is

```
(openfile tracefile |trace.r11| out)
```

This opens the file **trace.r11** for output and associates the name **tracefile** with the open file.

The atom **nil** cannot be used as the first argument to **openfile**. This atom is used to refer to the user's terminal (see Section 5.3.6).

5.3.5. closefile

The action **closefile** is used to close files that have been opened with **openfile**. This action takes an RHS pattern as its argument. The pattern should evaluate to one or more symbolic atoms. These atoms should be names which have been associated with files by **openfile**. When **closefile** is executed, the operating system is called to close the files and the associations between the names and the files are removed. Thus to close the file that was opened in the example above, the following would be executed

```
(closefile tracefile)
```

It is important that output files be closed before the OPS5 interpreter is exited. On some systems, the files will be lost if they are not closed.

5.3.6. default

The action **default** is used to control where **write** and the trace routines print their information and where **accept** and **acceptline** read their information. This action takes an RHS pattern as its argument. After the pattern is evaluated, the first two positions in the result element should contain values. The first position should contain either **nil** or a symbolic atom that has been associated with a file by **openfile**. The second position should contain either **trace**, **write**, or **accept**; the value in this position determines which default is being set. (The atom **acceptline** is not a valid value for the second position; **acceptline** reads from the same default file as **accept**.) As an example of its use, to make the file that was opened in the example in Section 5.3.4 be the default for trace information, the following would be executed.

```
(default tracefile trace)
```

If the second argument to **default** is **nil**, then the default is set to the user's terminal. Thus to undo the effects of the previous call to **default**, the following would be used

```
(default nil trace)
```

5.3.7. write

The action **write** is used to output information from the production system. The action takes an RHS pattern as its argument. It instantiates the pattern and then prints the values in the result element on the user's terminal or a file. (Thus the pattern should be in vector format; if it is in attribute-value format, the information will come out in a jumbled order that depends on the assignment of numbers to attribute names.)

If the value in the first field of the result element has been associated with an output file by **openfile**, the information will be written to that file. If the value has not been associated to an output file, the information will be written to the current default stream for **write**. The value in the first position is not printed if it is a file specifier.

As explained in the following sections, the user can specify printer control information in **write**. When information is not supplied, **write** prints its values on the current output line, putting one space between values.

5.3.7.1. Special Functions for write

Three functions, **crlf**, **tabto**, and **rjust** are provided for use with **write**. It is possible to call these functions within **make**, **modify**, or other action, but this is not recommended.

In some implementations of the OPS5 interpreter these functions place only a single value into the result element; in other implementations they place two. Nonetheless, production systems will always give the same results provided the operator ↑ is not used in **write**.

5.3.7.2. crlf

The function **crlf** puts into the result element a value that will cause **write** to begin a new line when it encounters the atom. The function takes no arguments, so a call on it has the form **(crlf)**. As an example of its use, the following action

```
(write (crlf) a b c (crlf) (crlf) d e f)
```

will cause the interpreter to begin a new line, print **a b c**, skip a line (by executing the operation to begin a new line twice), and then print **d e f**. Thus the output is

```
a b c

d e f
```

5.3.7.3. tabto

The function **tabto** places values into the result element that cause the **write** action to move to a specified column. The function takes one argument, the column number. The argument must be a numeric atom or a variable that is bound to a numeric atom. Typical calls on **tabto** are

```
(tabto 30)
(tabto <x>)
```

If the specified column is to the left of the last column printed, a new line is begun. Thus the action

```
(write (crlf) (tabto 5) * (tabto 3) * (tabto 1) *)
```

would print

```
    *
  *
*
```

The action

```
(write (crlf) (tabto 1) * (tabto 3) * (tabto 5) *)
```

would print

```
* * *
```

5.3.7.4. rjust

The function **rjust** is used to print values flush-right in fields of specified widths. The function takes one argument, an indication of the width of the field. The argument must be a numeric atom or a variable that evaluates to a numeric atom. When the action is evaluated it places print-control information in the result element. When **write** processes the information, it allocates a field of the indicated width beginning at the next available position on the output line. Then **write** determines the number of characters that the next value to be printed will need and prints enough blanks to cause the value to be right justified in the field. Thus the action

```
(write (crlf) (tabto 10) (rjust 10) abc)
```

will cause **a** to be printed in column 18, **b** in column 19, and **c** in column 20. This action is equivalent to

```
(write (crlf) (tabto 18) abc)
```

If the value to be printed is wider than the field, **write** reverts to the normal mode of printing. That is, it prints a single space and then the value.

The action must immediately precede a printable value. That is, it must not precede a call on **crlf**, **tabto**, or **rjust**. However, it is legal for **rjust** to follow **crlf** or **tabto**.

5.3.8. call

The action **call** is used to call subroutines written by the user. The action takes as arguments the name of a subroutine and an RHS pattern. It instantiates the pattern and then calls the subroutine. The subroutine can interrogate the OPS5 interpreter to determine what information is in the result element. (See Section 7 for more information about the interaction between OPS5 and the subroutine.)

5.3.9. halt

The action **halt** sets an internal flag in the interpreter that causes the interpreter to stop firing productions after completing the recognize-act cycle in progress. The action takes no arguments; a call on **halt** always takes the following form.

```
(halt)
```

5.3.10. bind

The action **bind** is used to assign values to variables. There are two forms of calls on **bind**. In the more general form **bind** is given two arguments: a variable and an RHS pattern. It evaluates the pattern and then assigns to the variable the value that is in position 1 of the result element. For example, to add 1 to the binding of **<x>**, the following would be executed.

```
(bind <x> (compute <x> + 1))
```

In the other form of **bind**, the action is given only one argument -- the variable to be bound. When this action is executed, a new symbolic atom is created and assigned to the variable. Thus the action

```
(bind <z>)
```

is equivalent to

```
(bind <z> (genatom))
```

5.3.11. cbind

The action cbind is used to assign values to element variables. The action takes only one argument, the variable. A typical call is

```
(cbind <c>)
```

The variable is bound to the last element that was added to working memory (by **make**, **modify**, or infrequently **call**). The result of executing **cbind** before the RHS has added an element is undefined.

5.3.12. build

The action **build** is supported only by the LISP-based interpreters for OPS5. This action is used to add a new production to production memory while the system is executing. Because some of the variables, actions, and functions in the argument to **build** are meant to be evaluated when the action is performed, while

others are meant to be incorporated as they are in the new production, **build** cannot use the ordinary OPS5 argument evaluation mechanism. Instead, when **build** is evaluated, all its arguments are treated as constants unless they are preceded by the special unquote operator, \\. The arguments to **build** should evaluate to a symbolic atom (the production's name), a sequence of condition elements, the atom `-->`, and a sequence of actions.

6. The Recognize-Act Cycle

By convention, the steps in the recognize-act cycle are usually said to occur in the following order:

1. [Match] Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.

2. [Conflict Resolution] Select one production with a satisfied LHS. If no productions have satisfied LHSs, return control to the user.

3. [Act] Perform the actions specified in the RHS of the selected production.

4. If a **halt** action was performed, return control to the user; otherwise go to step 1.

In the OPS5 interpreter, the cycle has been changed to:

1. [Conflict Resolution] Select one production with a satisfied LHS. If no productions have satisfied LHSs, return control to the user.

2. [Act] Perform the actions specified in the RHS of the selected production.

3. [Match] Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.

4. If a **halt** action was performed, return control to the user; otherwise go to step 1.

The OPS5 cycle is more convenient for the user because when the cycle ends, the conflict set is consistent with the current contents of working memory.

6.1. Conflict Resolution

The output of the match process, and the input to conflict resolution, is a set called the *conflict set.* The objects in the conflict set are called *instantiations.* An instantiation is an ordered pair of a production name and a list of working memory elements satisfying the production's LHS. During conflict resolution the interpreter examines the conflict set to find an instantiation which dominates all the others under the ordering rules listed below. The dominant instantiation will be executed in the act phase of the cycle.

A set of ordering rules for instantiations is called a conflict resolution strategy. OPS5 provides two strategies called *LEX* and *MEA.* Although these strategies are rather complex to describe, what they achieve is simple:

- Both strategies prevent instantiations from executing more than once. Early production systems were subject to trivial loops in which the interpreter fired a production on the same data indefinitely. The OPS5 strategies contain a mechanism to prevent these loops.

- They make production systems attend to the most recent data in working memory. This makes

production systems 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 earlier tasks. The difference between LEX and MEA is that MEA makes the system more sensitive to recent tasks. With the MEA strategy, the system cannot be distracted from its current task.

- They give 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.

These three things are important because they make it easy to add productions to an existing set and have the new productions fire at the right time, and because they make it easy to simulate common control constructs such as loops and subroutine calls. See [8] for a defense of these assertions.

6.1.1. The LEX Strategy

The LEX conflict resolution strategy contains four rules which are applied in order to find the instantiation that dominates under them.

1. Discard from the conflict set the instantiations that have already fired. If there are no instantiations that have not fired, conflict resolution fails and no instantiation is selected.

2. Order the instantiations on the basis of the recency of the working memory elements, using the following algorithm to compare pairs of instantiations: First compare the most recent elements from the two instantiations. If one element is more recent than the other, the instantiation containing that element dominates. If the two elements are equally recent, compare the second most recent elements from the instantiations. Continue in this manner either until one element of one instantiation is found to be more recent than the corresponding element in the other instantiation, or until no elements remain for one instantiation. If one instantiation is exhausted before the other, the instantiation not exhausted dominates; if the two instantiations are exhausted at the same time, neither dominates.

3. If no one instantiation dominates all the others under the previous rule, compare the dominant instantiations on the basis of the specificity of the LHSs of the productions. Count the number of tests (for constants and variables) that have to be made in finding an instantiation for the LHS. The LHSs that require more tests dominate.

4. If no single instantiation dominates after the previous rule, make an arbitrary selection of the dominant instantiation.

6.1.2. The MEA Strategy

The MEA strategy differs from LEX in that another rule has been added after the first. The rule that was second had to be modified slightly to accommodate the new rule. The rules for MEA are:

1. Discard from the conflict set the instantiations that have already fired. If there are no instantiations that have not fired, conflict resolution fails and no instantiation is selected.

2. Compare the recencies of the working memory elements matching the first condition elements of the instantiations. The instantiations using the most recent working memory elements dominate.

3. Order the instantiations on the basis of the recencies of the remaining working memory elements, using the following algorithm to compare pairs of instantiations: First compare the most recent elements from the two instantiations. If one element is more recent than the other, the instantiation containing that element dominates. If the two elements are equally recent, compare the second most recent elements from the instantiations. Continue in this manner either until one element of one instantiation is found to be more recent than the corresponding element in the other instantiation, or until no elements remain for one instantiation. If one instantiation is exhausted before the other, the instantiation not exhausted dominates; if the two instantiations are exhausted at the same time, neither dominates.

4. If no one instantiation dominates all the others under the previous rule, compare the dominant instantiations on the basis of the specificity of the LHSs of the productions. Count the number of tests (for constants and variables) that have to be made in finding an instantiation for the LHS. The LHSs that require more tests dominate.

5. If no single instantiation dominates after the previous rule, make an arbitrary selection of the dominant instantiation.

6.1.3. Which Instantiations to Discard

The first rule in both strategies specifies that instantiations that have already fired are to be discarded. Implementing this rule requires that a precise definition of equality for instantiations be chosen; and this in turn requires that a precise definition of equality for working memory elements be chosen. In OPS5 the latter is simple: Working memory elements X and Y are equal if they have equal time tags. The former is somewhat more complex; the definition of equality for instantiations that is used in OPS5 is: Instantiations A and B are equal if

- A and B are instantiations of the same production,
- A and B contain the same list of working memory elements, and
- If A was in the conflict set at time Ta and B was in the conflict set at time Tb, there is no time Tc between Ta and Tb such that A and B were not in the conflict set at time Tc.

The last item here probably requires an explanation. It is needed for productions that contain negated condition elements. It is possible for such a production to be satisfied by some list of working memory elements (instantiation A), become unsatisfied because something enters working memory that matches the negated condition element, and then become satisfied again on the original list of elements when the new element is deleted (instantiation B). The third rule is included so that the production will be able to respond to these changes by firing a second time.

6.2. Act

In the act phase of the cycle, the actions in the chosen production are executed one at a time, in the order they are written. Actions take effect immediately. Hence if an RHS contains several **make** or **modify** actions, the element added by the last action in the RHS is more recent than the elements added by the rest.

6.3. Match

During the match, the interpreter determines every instantiation of every production. That is, it finds every production that is instantiated, and if any of the productions can be instantiated by more than one list of working memory elements, it finds every list of elements. It puts the instantiations into the conflict set.

7. User-Defined Actions and Functions

The OPS5 interpreters allow users to write their own actions and functions. The BLISS-based interpreter will call routines written in BLISS (or any other language that uses the BLISS subroutine calling conventions); the LISP-based interpreter will call routines written in LISP.

7.1. Declarations

The user's routines must be declared to the interpreter before they are used in an RHS. The syntax of the declaration is: an open parenthesis, the atom **external**, one or more routine names, and a close parenthesis. Any number of routines may be declared external in one declaration, and any number of declarations may be made in a production system. Thus to declare **min** and **max**, either of the following could be used:

```
(external
    min
    max)
```

or

```
(external min)
(external max)
```

7.2. Actions

User-defined actions are called, using **call**, from the RHS of a production or from the top level (see Sections 5.3.8 and 8.1.6). The routine should take no arguments, and it should return no values (if values are returned they are ignored). All communication between the interpreter and the routine is accomplished through use of the functions described below.

7.2.1. $parameter

The second argument to the **call** action is an RHS pattern, which is instantiated into the result element before the user's routine is called. The function **$parameter** allows the routine to read values out of the element. The function takes one argument, an integer; when it is called with the argument K, it returns the value in the Kth field in the element. Thus to get the first value in the element, a routine written in LISP would execute

```
($parameter 1)
```

and an action written in BLISS would execute

```
$parameter(1)
```

Following the usual OPS5 convention, when **$parameter** is called to access a field that was not explicitly given a value, it returns **nil**. It is considered an error, however, to access a non-existent field (i.e., to use an

index less than 1 or greater than 127).

7.2.2. $parametercount

The function **$parametercount** returns an integer; the integer is the number of the last field in the result element that received a value. Thus if the **call** did not contain the operator ↑, this function indicates how many values were put into the result element. (Generally, ↑ is not used with **call**.) The function takes no arguments.

7.2.3. $assert

Some of the actions written by users add elements to working memory. The actions put an element in working memory by clearing the result element (see Section 7.2.6), putting the new values in the result element (see Sections 7.2.5 and 7.2.4), and then executing the function **$assert**. The function **$assert** copies the result element into working memory. After it is copied into working memory, the result element can be cleared again and another collection of values assembled there. The function **$assert** takes no arguments.

7.2.4. $tab

The function **$tab** controls where the next value will be placed in the result element. This function takes one argument, which should be either an integer or a symbolic atom which has been assigned an integer in a **literalize** or **literal** declaration. When **$tab** is executed it informs the interpreter that the next value put into the result element should go into the indicated field.

7.2.5. $value

The function **$value** is used to put one symbolic atom or number into the result element. It is called with one argument, the value to put in. If no **$tab** has been executed since the last call on **$value**, it puts the value in the field just after the one used on the previous call. If **$tab** has been executed since the last call on **$value**, it puts the value in the field that **$tab** designated. If no calls on either **$tab** or **$value** have been made since the result element was cleared, the value is placed in the first field. (These rules for deciding where to put values are equivalent to the rules used for terms in the RHS -- see Section 5.2.3.)

7.2.6. $reset

The function **$reset** is used to remove the information currently in the result element. This function takes no arguments. It should be noted that **$assert** does not automatically perform a **$reset**.

7.2.7. $ifile and $ofile

The functions **$ifile** and **$ofile** are used to access files that were opened with **openfile**. The function **$ifile** takes a single argument, which should be a symbolic atom that is associated with an open file. That is, the atom should have occurred as the first argument to **openfile**. If the atom is associated with a file that is currently open for input, the file is returned. (More precisely, in FRANZ LISP, a port is returned; in MACLISP, a file object is returned; and in BLISS, the address of an XPORT IOB is returned [2].) If the atom is not associated with a file that is open for input, a failure signal is returned: in LISP, the atom **nil** is returned, and in BLISS, the XPORT value **xpo$k_failure** is returned. The function **$ofile** is identical except that it returns files that are open for output.

7.3. Functions

The syntax of a call on a user-written function is identical to the syntax of a call on a standard function: The call consists of an open parenthesis, the name of the function, the arguments to the function (if any), and a close parenthesis.

The conventions for passing arguments to functions are not the same in the LISP- and BLISS-based interpreters. In the BLISS-based interpreter, the arguments are evaluated (i.e., OPS5 variables are replaced by their bindings) and then they are passed using the ordinary BLISS parameter passing mechanism. Thus if the function in the RHS has three parameters, the BLISS routine is called with three arguments. In the LISP-based interpreter, the arguments are passed unevaluated. The LISP routine must be a **fexpr**. If the LISP routine needs the arguments to be evaluated, it calls routines in the interpreter to perform the evaluation. (See the two sections immediately following.)

RHS functions do not return values using the normal value return mechanism of LISP or BLISS. (If values are returned with the normal mechanism, OPS5 discards them.) Instead, values are returned using the function **$value** described in Section 7.2.5.

7.3.1. $varbind

The function **$varbind** is provided in the LISP-based interpreter to allow RHS functions to evaluate their arguments. This function takes one argument. If the argument is a bound variable, the binding of the variable is returned. If the argument is not a bound variable, the argument is returned unchanged.

7.3.2. $litbind

The function **$litbind** is provided in both the LISP- and BLISS-based interpreters. This function takes one argument. If the argument has been assigned a number in a **literal** or **literalize** declaration, the number is returned. If the argument has not been assigned a number, the argument is returned unchanged.

7.4. Atoms

The scalar values in the LISP-based interpreters are ordinary LISP atoms, so user-supplied routines can process them using the usual LISP functions. The scalar values in the BLISS-based interpreters are data types that are implemented in the OPS5 interpreter, so user-supplied routines must call routines in the interpreter to process them. The following are the necessary routines.

7.4.1. $eql

An atom in the BLISS-based interpreter is a one word value (32 or 36 bits, depending on the computer being used). To compare two atoms for equality, the routine **$eql** is used. The routine takes two parameters, the atoms to compare. It returns a true value if the atoms are the same type and

- They are symbolic atoms that consist of the same string of characters, or
- They are numeric atoms whose algebraic difference is zero.

7.4.2. $symbol

The routine **$symbol** is used to test the type of atoms. It takes a single parameter, the atom to test. The routine returns a true value if the atom is a symbolic atom, and a false value if it is a numeric atom.

7.4.3. $intern

The routine **$intern** is used to convert a string of characters into a symbolic atom. It takes two parameters, a BLISS character string pointer and a count of the number of characters in the string. It returns the symbolic atom that represents the string.

7.4.4. $cvan and $cvna

The routines **$cvan** and **$cvna** are used to convert between numeric atoms and ordinary BLISS integers. Both routines take a single parameter. The routine **$cvan** takes an atom as its parameter and returns an ordinary integer. The routine **$cvna** takes an ordinary number and returns a numeric atom.

8. Using the OPS5 Interpreter

This section explains how to load a production system into the interpreter and how to run the production system after it is loaded.

8.1. The Top Level

After OPS5 is installed on a system, it is invoked as any other program on the system is. When the interpreter starts, it begins executing the top level routine. When the production system stops executing for any reason, the interpreter returns to the top level routine. This routine allows the user to add productions to production memory (in LISP only), to put elements into working memory, to inspect the state of the production system, to start the production system executing, etc. The top level routine is

1. Read a command from the user.

2. Execute the command.

3. Goto 1.

The following sections describe the commands that the OPS5 interpreter supports.[3]

The syntax of all commands is the same: A command consists of an open parenthesis, the name of the command, the arguments to the command if any, and a close parenthesis. On the BLISS-based interpreter, if the command does not have arguments, the parentheses may be omitted. The commands are free format; end of line is treated like a space.

8.1.1. make

The action **make** can be executed at the top level as well as in a production's RHS. If the user types

```
(make start)
```

the element

```
(start)
```

will be created and placed into working memory. At the top level, **make** will not accept variables, the operator //, or functions as arguments. Constant symbols and numbers, ↑, and literalized atoms are acceptable as arguments.

When **make** is executed, the match process is performed, and the conflict set is updated.

---

[3]The OPS5 interpreters that are written in LISP use the normal LISP top level. Thus in these interpreters the user can execute any LISP command. However, the interpreter written in BLISS accepts only the commands listed here.

8.1.2. remove

The action **remove** may also be executed at the top level. However, since variables cannot be used at the top level, **remove** uses a different method to designate the elements to delete. If the user types

```
(remove *)
```

the interpreter deletes everything from working memory. If the user gives one or more numbers as arguments, the elements having those time tags are deleted. Thus typing

```
(remove 117 118)
```

will cause elements with time tags 117 and 118 to be deleted.

When **remove** is executed, the match process is performed, and the conflict set is updated.

8.1.3. openfile

The action **openfile** may be executed at the top level as well as in the RHS of a production. It has the same effect as **openfile** in the RHS. When called at the top level, its argument should not contain variables, the operator **//**, or function calls.

8.1.4. closefile

The action **closefile** may be executed at the top level as well as in the RHS of a production. It has the same effect as **closefile** in the RHS. When called at the top level, its argument should not contain variables, the operator **//**, or function calls.

8.1.5. default

The action **default** may be executed at the top level as well as in the RHS of a production. It has the same effect as **default** in the RHS. When called at the top level, its argument should not contain variables, the operator **//**, or function calls.

8.1.6. call

The action **call** can also be used at the top level. Like the RHS command **call** (see Section 5.3.8) this command is used to invoke user-defined subroutines. Its arguments should be a routine name and an optional pattern like the patterns given to **make** at the top level. The pattern should not contain variables, the operator **//**, or function calls. The interpreter instantiates the pattern and invokes the routine. The routine must have been declared **external**.

8.1.7. run

The command **run** causes the interpreter to execute a production system. If the user types

```
(run)
```

the production system is allowed to execute until it halts or a breakpoint is reached (see Section 8.1.15). If the user gives a numeric argument to **run** the interpreter will automatically halt after that many cycles. Thus entering

```
(run 100)
```

will cause the interpreter to run 100 cycles and halt. (Of course, the system may not execute the full 100 cycles, because the conflict set may become empty, a production may execute the **halt** action, etc.)

8.1.8. ppwm

The command **ppwm** is one of two commands to print working memory elements. (See also **wm**, below.) This command takes a pattern like a condition element; it prints all the elements matching the pattern. For example

```
(ppwm goal ↑status active)
```

will print all the active goals. When **ppwm** is called with a null pattern, as in

```
(ppwm)
```

it prints every element in working memory. The pattern can contain constant symbols and numbers, the operator **↑**, and literalized atoms. It should not contain variables, predicates, the operator **//**, or the two kinds of brackets ({ } and << >>).

8.1.9. wm

The command **wm**, like **ppwm**, is a command to print working memory elements. It differs from **ppwm** in the kind of arguments it takes. This command takes a list of time tags and prints the elements with those time tags. It is useful because some of the other OPS5 commands print time tags rather than working memory elements to save space; **wm** is used to expand the time tags into the elements they represent. Thus

```
(wm 5 6 7)
```

causes the interpreter to print the three elements whose time tags are 5, 6, and 7. When **wm** is given with no arguments, as in

```
(wm)
```

the interpreter prints the entire contents of working memory, as **ppwm** with no arguments does.

8.1.10. pm

The command **pm** displays productions on the user's terminal. It is called with one or more production names, and it prints the productions in a readable format. This command is not supported in the BLISS-based interpreter.

8.1.11. cs

The command **cs** prints the current contents of the conflict set. The command does not accept arguments.

8.1.12. matches

The command **matches** prints the partial matches for productions. It is called with one or more production names as its argument; for example

```
(matches find-colored-block)
```

It prints the time tags of the elements matching each condition element of each production; it prints the pairs of working memory elements matching the first two condition elements; it prints the triples matching the first three condition elements; and so forth.

8.1.13. strategy

The command **strategy** prints or sets the conflict resolution strategy being used. If the command is given with no arguments, as in

```
(strategy)
```

it prints the current strategy (it will be either **mea** or **lex**). If the command

```
(strategy mea)
```

is given, it sets the current strategy to **mea**. If the command

```
(strategy lex)
```

is given, it sets the current strategy to **lex**. The only legal arguments to **strategy** are **lex** and **mea**.

The default strategy -- that is, the one in effect when the interpreter starts -- is **lex**.

8.1.14. watch

The command **watch** controls how much trace information the interpreter prints while it executes a production system. If the user executes

```
(watch 0)
```

the system will print no trace information. If the user executes

```
(watch 1)
```

the system will print the name of each production that fires along with a list of the time tags of the elements

instantiating the production. If the user executes

```
(watch 2)
```

the interpreter will print the information of level 1, and it will print the elements that are added to or deleted from working memory. If the user executes

```
(watch 3)
```

the interpreter will print the information of level 2, and it will print every change to the conflict set when it happens. Level 3 of tracing is not supported in the LISP-based interpreters. If **watch** is called with no arguments, it reports the current trace level.

8.1.15. pbreak

The command **pbreak** sets and removes breakpoints on the productions. If a breakpoint is set on a production, the interpreter will halt and return to top level whenever that production fires. The production is allowed to execute, but then the recognize-act cycle is exited. Giving the command **pbreak** with no arguments causes the interpreter to print the names of the productions that have breakpoints set. Giving the command with productions as arguments, as in

```
(pbreak r16 r17)
```

toggles the state of the listed productions: The productions that had breakpoints set have them removed; the productions that did not have breakpoints have them set.

8.1.16. exit

The command **exit** causes the interpreter to cease operation and returns the user to the monitor. The command does not take arguments.

In the BLISS-based interpreter, a control-Z character (ASCII 32 octal) is treated like the **exit** command.

8.1.17. excise

The command **excise** is used to delete productions from production memory. When **excise** is called, its argument list should contain one or more production names.

8.1.18. back

The command **back** is supported only in the LISP-based interpreters. This command causes the interpreter to restore the production system to an earlier state. The command takes one argument, a number indicating how many recognize-act cycles to back up. Thus

```
(back 1)
```

causes the system to back up 1 cycle. To save space, the interpreter maintains only enough information to

back up 32 cycles.

The commands **back** and **run** can be intermixed without confusing the interpreter. The following sequence, for example, is legal.

```
(run 100)
(back 10)
(run 5)
(back 15)
```

If no productions have fired before, this will cause the interpreter to perform cycles 1 to 100, back up to the state that existed after cycle 90, run for another 5 cycles, and then back up to the state that existed after cycle 80.

8.2. Loading a Production System

When the BLISS-based OPS5 interpreter is used, productions are compiled and linked with the interpreter before the interpreter is started. Thus with this interpreter the system is always ready to run as soon as the interpreter is started.

With the LISP-based OPS5 interpreter, productions are usually defined after the interpreter is started. (In fact, unless the user has saved his own core image, production memory will contain no productions when the interpreter is started.) Productions are defined by typing in the declarations and the productions, by loading files that contain the declarations and the productions, or both.

Appendix I
Syntax of OPS5

The following is a simplified BNF description of the syntax of OPS5. Terminals are printed in a Roman type face, and non-terminals are printed in italics. The only nonstandard meta symbol used is the star ("*"). The star indicates that the preceding item is to be repeated zero or more times.

production : : = (p *constant-symbolic-atom lhs* --> *rhs*)

lhs : : = *positive-ce ce**

ce : : = *positive-ce*
: : = *negative-ce*

positive-ce : : = *form*
: : = { *element-variable form* }
: : = { *form element-variable* }

negative-ce : : = - *form*

form : : = (*lhs-term**)

lhs-term : : = ↑ *constant-symbolic-atom lhs-value*
: : = ↑ *number lhs-value*
: : = *lhs-value*

lhs-value : : = { *restriction** }
: : = *restriction*

restriction : : = << *any-atom** >>
: : = *predicate atomic-value*
: : = *atomic-value*

atomic-value : : = // *any-atom*
: : = *var-or-constant*

var-or-constant : : = *constant-symbolic-atom*
: : = *number*
: : = *variable*

predicate : : = =
: : = <>
: : = <
: : = <=
: : = >=
: : = >
: : = <=>

rhs : : = *action**

action : : = (make *rhs-term**)
: : = (remove *element-designator**)
: : = (modify *element-designator rhs-term**)

```
                    ::=   ( halt )
                    ::=   ( bind variable )
                    ::=   ( bind variable rhs-term* )
                    ::=   ( cbind element-variable )
                    ::=   ( call constant-symbolic-atom rhs-term* )
                    ::=   ( write rhs-term* )
                    ::=   ( openfile rhs-term* )
                    ::=   ( closefile rhs-term* )
                    ::=   ( default rhs-term* )
                    ::=   ( build quoted-form* )

element-designator  ::=   number
                    ::=   element-variable

rhs-term            ::=   ↑ var-or-constant rhs-value
                    ::=   rhs-value

rhs-value           ::=   atomic-value
                    ::=   function

function            ::=   ( litval var-or-constant )
                    ::=   ( substr element-designator var-or-constant var-or-constant )
                    ::=   ( genatom )
                    ::=   ( crlf )
                    ::=   ( rjust var-or-constant )
                    ::=   ( tabto var-or-constant )
                    ::=   ( accept )
                    ::=   ( accept var-or-constant )
                    ::=   ( acceptline var-or-constant* )
                    ::=   ( compute expression )
                    ::=   user-defined-function

user-defined-function ::=  ( constant-symbolic-atom var-or-constant* )

expression          ::=   number
                    ::=   variable
                    ::=   expression operator expression
                    ::=   ( expression )

operator            ::=   +
                    ::=   -
                    ::=   *
                    ::=   //
                    ::=   \\

quoted-form         ::=   \\ rhs-value
                    ::=   any-atom
                    ::=   ( quoted-form* )
```

Several terms have been left undefined: *variable*, *element-variable*, *constant-symbolic-atom*, *any-atom*, and *number*. Symbolic atoms and numbers are described in Section 2. The two kinds of variables are described in Sections 4 and 5. The only thing that needs to be explained here is the difference between *any-atom* and

constant-symbolic-atom. The former is an atom that is treated as a constant because it is quoted (with **//** or **<<** **>>** usually). The latter is an atom that is treated as a constant because it does not have the form of a variable or operator.

References

1. Digital Equipment Corporation. *BLISS language guide.* 1980.

2. Digital Equipment Corporation. *XPORT programmer's guide.* 1980.

3. Foderaro, J. K. *The FRANZ LISP manual.* University of California at Berkeley, 1980.

4. Forgy, C. L. and McDermott, J. The OPS reference manual. Department of Computer Science, Carnegie-Mellon University, 1976.

5. Forgy, C. L. and McDermott, J. OPS, a domain-independent production system. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 933-939.

6. Forgy, C. L. and McDermott, J. The OPS2 reference manual. Department of Computer Science, Carnegie-Mellon University, 1978.

7. Forgy, C. L. OPS4 user's manual. Department of Computer Science, Carnegie-Mellon University, 1979.

8. McDermott, J. and Forgy, C. L. Production system conflict resolution strategies. In Waterman, D. A. and Hayes-Roth, F., Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 177-199.

9. MIT AI Lab and Project MAC. *MACLISP manual.* Massachusetts Institute of Technology, 1978.

10. Newell, A. PSG manual. Department of Computer Science, Carnegie-Mellon University, 1973.

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

12. Rychener, M. D. OPS3 production system language tutorial and reference manual. Department of Computer Science, Carnegie-Mellon University, 1980.

Index