CMU-CS-84-133

The OPS83 Report*

May 1984

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

Abstract: OPS83 is a programming language for expert systems applications. It combines the rule-based programming paradigm of the earlier versions of OPS with the procedural programming paradigm of conventional programming languages. It is less restrictive than the earlier versions of OPS in several respects, including the data structures permitted in working memory and the kinds of expressions that can be used in the LHSs of rules. OPS83 is a compiler-based language, and it provides for separate compilation of modules with full type checking across modules.

*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 F33615-81-K-00450.

Table of Contents

1. Introduction

OPS83 is a programming language for expert systems applications. It supports not only the rule-based programming paradigm of the earlier versions OPS, but also the procedural programming paradigm of conventional programming languages such as PASCAL [6] or C [7]. Because of this, OPS83 permits more natural and more efficient solutions to many problems than the earlier versions of OPS did; while the rule-based paradigm has proven very powerful, it is not appropriate for all programming tasks.

This document contains a complete description of OPS83. This is not a tutorial introduction to the language; it is intended to be a concise reference for the language. In some cases it is necessary to refer to terms before they are defined. An attempt has been made to include references to the relevant sections whenever this presents a potential problem for the reader.

1.1 Background

The first version of OPS was developed in 1975 by Charles Forgy, John McDermott, Allen Newell, and Michael Rychener [1, 2]. OPS1 was influenced strongly by PSG [8] which was developed by Allen Newell and by PSNLST [9] which was developed by Michael Rychener. Since 1975 OPS has been revised several times [3, 4, 5, 10]. PSG, PSNLST, and the various versions of OPS have been used for a variety of tasks including research in cognitive psychology, artificial intelligence programming, research into learning systems, and expert systems development.

1.2 Notation

The language is described using BNF notation. In BNF, the terms used in the language description are divided into the two classes of *terminals* and *non-terminals*. Terminals are objects that actually occur in programs. Non-terminals are names that are used to describe constructs in the language. In the BNF, non-terminals are designated by words or phrases enclosed in angle brackets (the characters "<" and ">"). The meaning of each non-terminal is defined by production rules. A rule consists of the non-terminal being defined, the symbol ::=, and the sequence of terminals and non-terminals that constitute the definition. If there are alternate definitions for a non-terminal, the alternatives are separated by vertical bars (the character "|"). For example, in the following, the non-terminal <optional-sign> is defined to be either the terminal "+", the terminal "-", or the non-terminal <null>.

```
<optional-sign>          ::=      +  |  -  |  <null>
```

A sequence that is enclosed in braces ("{" and "}") may be repeated zero or more times. For example, in the following, a <digit-sequence> is defined to be a <digit> followed by zero or more repetitions of <digit>.

```
<digit-sequence>          ::=      <digit>  {  <digit>  }
```

When it is necessary to use one of the special characters such as "{" or "|" in a definition, the character is enclosed in quotes. For example, in the following, braces are used as terminals rather than special characters.

```
<compound>                ::=      "{"  <statement-list>  "}"
```

In order to enhance the readability of the BNF, the description has been simplified somewhat. Hence in some details the BNF is not a wholly accurate description of the language. However, an attempt has been made to ensure that the language described by the BNF is a legal subset of OPS83. That is, the OPS83 compiler will accept any program that corresponds to the BNF; in certain cases it will also accept programs that deviate slightly from the BNF.

1.3 Organization of the Report

This report describes the language in a generally bottom-up order. It begins in Chapter 2 by describing the kinds of values that OPS83 programs work with. In Chapters 3 through 10 it describes the language itself, starting with the lexical system and working up to modules and programs. Chapter 11 describes the interpreter's data structures. Finally, Chapter 12 describes the language's standard predefined routines.

2. Values and Types

This chapter describes the kinds of values that OPS83 programs can manipulate.

2.1 Scalar and Structured Values

The values can be divided into two classes, scalar values and structured values. Scalar values are values which are always operated on as a whole. Examples of scalar values include integers and real numbers. Structured values are aggregates of simpler values, and the language provides primitives for accessing the components of a structured value and for changing the components individually. The simplest example of a structured value is an array of integers. The values comprising a structured value can themselves be structured. Thus, for example, it possible to use an array of arrays.

2.2 Types

Every value has an attribute called its type. The type of a value indicates

- Whether the value is scalar or structured,
- If the value is structured, how the values comprising it are accessed,
- How the value is represented in the computer,
- Which operations may be performed on the value, and
- Whether the value can occur in working memory.

2.3 Type Compatibility

One of the reasons for having types in the language is to allow the compiler to distinguish certain semantic errors. In the following program constructs, the types of values are compared, and if the types are not the same, a compile-time error is generated:

- Assignment statements. In assignment statements, the type of the l-value (see Section 5.5) and the type of the value to be assigned to it must be the same.
- Expressions containing binary operators. The types of both operands of a binary operator must be the same.
- Calls to functions and procedures. The types of formal and actual parameters must be the same.

Two types are the same if

- Both types are named.
- The names of the two types are the same (i.e., the names are equal symbols).

2.4 Standard Scalar Types

OPS83 provides five standard scalar types: integer, real, symbol, char, and logical. The standard scalar types are described below. The conceptual meaning of each type and the machine representation of values of the type are given. The operations defined on the types are described in later sections of the report.

Integer values are values from the set of integers. Integers are represented on the computer as 32-bit binary fixed point numbers.

Real values are finite-precision approximations of values from the set of real numbers. Reals are represented on the computer as 64-bit floating point numbers.

Symbols are values that are represented in the program text as character strings. The character strings may contain no more than 127 characters. Symbols are represented on the computer as 16-bit quantities.

Logical values are elements from the set {True, False}. Logical values are represented on the computer as 8-bit quantities.

Char values are values from the computer's character set. Char values are represented on the computer as 8-bit quantities.

2.5 The Structured Types

OPS83 provides three kinds of structured types: arrays, records, and elements.

An array is a sequence of N values, where N is an integer greater than zero. The values in an array all have the same type. The positions in the sequence are called the fields of the array. The values in the fields are accessed by giving the ordinal number of the field; i.e., by giving a number between 1 and N. (See Section 5.5 for the syntax.)

A record is a sequence of values which do not necessarily have the same types. The positions in the sequence are called the fields of the record. Each field has a name, and the values are accessed by specifying the name of the field containing the value. (See Section 5.5.)

An element is very similar to a record. The only difference is that elements can occur in the program's working memory while records cannot. Elements are the only class of values that can occur in working memory.

2.6 Derived Types

OPS83 supports one further kind of type, the derived type. A derived type is a named type that is declared to be like another existing named type -- for example the type fix might be defined to be derived from the type integer. The reason for using derived types is to permit the type checking performed by the compiler to be more comprehensive. This makes it possible for the compiler to catch more logical errors. For example if one defined both price and dimension to be derived from the type real, then the compiler would be able to find errors such as comparing a price with a dimension.

A derived type has the same representation as the type it was derived from. It has the same constants, and the same operators are defined for values of the new type. For an example of this, suppose that some program has the type dimension which is derived from reals, and suppose that &D is declared to be a variable of type dimension and &R a real. Then both of the following statements would be legal.

```
&D = &D + 1.0;

&R = &R + 1.0;
```

Note that in the first assignment statement, the `1.0` is assumed by the compiler to have type dimension, and that the + in the first statement is assumed to be an operator for that type. This is what is meant by saying that a derived type has the same constants and operators as the type it is derived from.

From this example it can be seen that the type of a constant cannot be determined merely from the representation of the constant in the program text. The context in which it is used must also be considered. In OPS83, if a certain type is required in order for an expression to be legal, and if the constant can be that type, it is assumed to be that type. For example, in the following, if &X is type integer, the 1 is assumed to be type integer. If the &X is a type that was derived from integer then the 1 is assumed to be that type.

```
&X = &X + 1;
```

3. Lexemes

The source text of a program consists of a sequence of characters. The OPS83 compiler divides the sequence into meaningful units called lexemes. It recognizes nine classes of lexemes: operators, integers, real numbers, symbolic constants, logical constants, character constants, keywords, identifiers, and variables. In addition, it recognizes two classes of character sequences that are to be ignored: whitespace and comments.

3.1 Character Set

```
<letter>                ::=   A | B | C | D | E | F | G
                        |     H | I | J | K | L | M | N
                        |     O | P | Q | R | S | T | U
                        |     V | W | X | Y | Z
                        |     a | b | c | d | e | f | g
                        |     h | i | j | k | l | m | n
                        |     o | p | q | r | s | t | u
                        |     v | w | x | y | z

<digit>                 ::=   0 | 1 | 2 | 3 | 4
                        |     5 | 6 | 7 | 8 | 9

<special-char>          ::=   . | [ | ] | ( | ) | &
                        |     = | < | > | { | } | '
                        |     ~ | + | - | * | / | \
                        |     , | : | ; | @ | "|"

<separator>             ::=   <space>  |  <end-of-line>

<letter-or-digit>       ::=   <letter>  |  <digit>

<space>                 is the character that prints as a single blank
                        space (i.e., character number 32 decimal in the
                        ASCII character set).

<end-of-line>           is the character that terminates lines in the
                        program.
```

The set of characters available to OPS83 programmers is machine-dependent. The implementation of OPS83 assumes that at least the characters listed above are available. If other printing characters are available in the computer's character set, they are assigned to the class <letter>. If other non-printing characters are available, they are assigned to the class <separator>.

3.2 Constants

Many of the lexemes in a program designate constant values. Like any other value, a constant value has a type. The type of a constant is determined partly by the way it is represented in the program and partly by context. The type cannot be determined entirely from the representation because derived types inherit the constants of the type they are derived from (see Section 2.6). The following sections describe how constants are designated in programs; for each class of constants they indicate which types the constants may be.

3.2.1 Integers

```
<integer>               ::=     <digit-sequence>

<digit-sequence>        ::=     <digit>  {  <digit>  }
```

An integer is a decimal representation of a constant value from the set of integers. The type of the value is integer or any type that is derived from integer.

3.2.2 Real Numbers

```
<real>                  ::=     <digit-sequence>  <exponent>
                        |       <digit-sequence>  .  <optional-exponent>
                        |       .  <digit-sequence>  <optional-exponent>
                        |       <digit-sequence>  .  <digit-sequence>
                                <optional-exponent>

<optional-exponent>     ::=     <exponent>  |  <null>

<exponent>              ::=     <e-or-E>  <optional-sign>  <digit-sequence>

<e-or-E>                ::=     e  |  E

<optional-sign>         ::=     +  |  -  |  <null>
```

A real constant is a decimal representation of a non-integral numeric value. The type of the value is either real or a type that is derived from real. As the BNF shows, quite a bit of flexibility is provided in the representation of real numbers. All the following, for example, are legal real constants.

```
    1.0          1.          .1          6e23          4.136E-15
```

Note that a real constant must contain either a decimal point (the operator .) or an exponent. A constant which contains neither is assumed to be an integer.

3.2.3 Character Constants

```
<char>              ::=    ' <any-char> '


<any-char>          ::=    <letter>  |  <digit>
                    |      <special-char>  |  <separator>
```

A character constant is a representation of a constant value from the machine's character set. The type of the value is either char or a type that is derived from char. The following are legal character constants.

```
    'X'     'x'     '&'     '''     ' '
```

3.2.4 Logical Constants

```
<logical>           ::=    0b  |  0B  |  1b  |  1B
```

Logical constants represent the logical values True (1b or 1B) and False (0b or 0B). The type of the value is either logical or a type that is derived from logical.

3.2.5 Symbolic Constants

```
<symbol>            ::=    <letter>  {  <letter-or-digit>  }
                    |      "|"  {  <quoted-char>  }  "|"


<quoted-char>       is any character in the character set except
                    for <end-of-line>,  "|",  or the null character
                    (i.e., character 0).
```

A symbol is any sequence of characters that is treated as a unit and that is not interpreted as a number, a variable, a logical or character constant, etc. The vertical bar can be used to force a sequence of characters that would not ordinarily comprise a legal symbol to be interpreted as a symbol. The following are all legal symbols.

```
    nil     NIL     |0B|     |1.0|     |<>|     |&X|     ||     |. . .|
```

The vertical bars are not part of the symbols; they are merely delimiters. Hence |nil| and nil are the same symbol. The distinction between upper and lower case is important. Thus nil is not the same symbol as Nil or NIL. The type of a symbolic constant is either symbol or a type that is derived from symbol.

3.3 Keywords and Identifiers

Not all the symbols in a program are constants. Some of them are used as identifiers or keywords. An identifier is a symbol that the user declares as the name of a routine, type, etc. A keyword is a symbol that is used to identify a construct (a statement, declaration, etc.) in the language. For example, in the statement

```
     for &X = (1 to 10) write() f(&X);
```

the symbols `for`, `to`, and `write` are keywords, and `f` is an identifier.

OPS83 does not have any reserved symbols. The keyword symbols are recognized as keywords only when they occur in the appropriate context. Consequently, these symbols can also be used as symbolic constants and identifiers without restriction. For example, the following is a legal OPS83 statement.

```
     return(return(return));
```

The first `return` is a keyword. The second is an identifier that designates a function. The third is a symbolic constant.

3.4 Variables

```
<variable>              ::=     &  {  <letter-or-digit>  }
```

In OPS83 variables are distinguished from other lexemes by a leading ampersand (the character &). OPS83 is case-sensitive. Thus the variable &X is not the same as the variable &x.

3.5 Operators

Operators are single characters or sequences of two or three characters that have a predefined meaning in the language. The operators are:

.	[	]	(	)
=	<	>	{	}
*	+	-	/	\
~	,	:	;	@
\/	/\	<>	<=	>=
-->				

3.6 Whitespace

Sequences of separator characters are called whitespace. Any number of separator characters may occur between lexemes without changing the meaning of the program. In general, though, separator characters may not occur within lexemes. In certain places separator characters are required: symbols, numbers, and variables must be separated either by separator characters or operators. Thus the following statement is ill-formed.

```
     if(&X>10)&Y=10else&Y=&X;
```

To make it legal, separator characters must be added on both sides of the `else`:

```
     if(&X>10)&Y=10 else &Y=&X;
```

3.7 Comments

A comment is everything from the operator -- up to the end of the line. Everything in the comment except the terminating end of line is ignored. Thus a comment is effectively another kind of whitespace.

4. Type Declarations

```
<type-declare>          ::=     type  <typename>  =  <newtype>

<typename>              ::=     <symbol>

<newtype>               ::=     <arraydef>
                         |      <recorddef>
                         |      <elementdef>
                         |      <derivedtype>
```

A type declaration is used to describe a new type and to assign a name to the type. A type declaration consists of the keyword **type**, the name to be assigned to the type, an equals sign, and the description of the new type.

4.1 Arrays

```
<arraydef>              ::=     array  (  <integer>  :  <typedef>  )


<typedef>               ::=     <arraydef>
                         |      <recorddef>
                         |      <elementdef>
                         |      <typename>
```

An array definition defines a new array type. The definition consists of the keyword **array**, an open parenthesis, an integer, a colon, a type description, and a closing parenthesis. The integer indicates how many values the array contains. The integer must be at least 1. The type description indicates the type of the values comprising the array. For example, the following declares two array types; the first is a vector of one hundred real numbers, and the second a square array of twenty by twenty integers.

```
    type vec = array(100 : real);

    type sqr = array(20 : array(20 : integer));
```

4.2 Records

```
<recorddef>             ::=     record  (  <fieldlist>  )


<fieldlist>             ::=     <fielddef>  {  ;  <fielddef>  }  <opt-;>


<opt-;>                 ::=     ;  |  <null>


<fielddef>              ::=     <symbol>  :  <typedef>
```

A record definition defines a new record type. The definition consists of the keyword **record** followed by a parenthesized list which defines the fields in the record. The field definitions are separated by semicolons. An extra semicolon may optionally follow the last field definition in the list. A field definition consists of a symbol, a colon, and a type description. The symbol is the name of the field. The type description indicates the type of the values stored in the field. For example, the following declares a simple record with two fields.

```
    type set = record
      (
        size : integer;
        values : array(100 : symbol);
      );
```

4.3 Elements

```
<elementdef>            ::=     element  (  <fieldlist> )
```

An element definition defines a new element type. An element definition is exactly like a record definition except the keyword **element** is used in place of **record**. The following is a simple legal element definition.

```
    type goal = element
      (
        name : symbol;
        status : symbol;
        priority : real;
        object : symbol;
      );
```

4.4 Derived Types

```
<derivedtype>           ::=     <typename>
```

A derived type is a declared by specifying that one type name is to be considered equivalent to another (existing) type name. For example,

```
    type fix = integer;

    type price = real;
```

It is legal to define non-scalar derived types. For example, if vec has been defined to be an array, then the following is legal

```
    type ary = vec;
```

Generally, however, scalar derived types are more useful than non-scalar ones.

5. Variables

A variable is a name that is used to refer to a location the computer's memory in which a value can be stored.

5.1 Global Variables

```
<global>            ::=    global <globalvar> { , <globalvar> }

<globalvar>         ::=    <variable> : <typedef>
```

A global variable is characterized by two properties:

- The global variable can be accessed by any non-simple routine in the program.
- The memory location for the variable is allocated when the execution of the program begins, and it remains allocated to the variable throughout the execution of the program.

The names of the global variables in a program must all be distinct. Global variables are not automatically initialized before execution of a program begins.

Global variables must be declared before they can be used. The global declaration is used to declare one or more global variables. A declaration consists of the keyword **global** followed by a list of variable declarations separated by commas. Each variable declaration consists of the name of the variable, a colon, and a specification of the type of values that will be assigned to the variable. The following shows a declaration of two global variables.

```
global
    &HISTORY : array(100 : symbol),
    &HISTNDX : integer;
```

5.2 Local Variables

```
<local>             ::=    local <localvar> { , <localvar> }

<localvar>          ::=    <variable> : <typedef>
```

A local variable is characterized by two properties:

- A local variable may be accessed only in the routine in which it is declared.
- The memory location for the variable is allocated when a routine is entered, and it is

released when the routine is exited.[1]

The local variables in a routine must have distinct names, and the names must be distinct from the names of the formal parameters of the routine. The names need not be distinct from the names of variables used in other routines or from the global variables in the program. However, if a local variable has the same name as a global variable, the global variable cannot be referenced in the routine. Local variables are not automatically initialized when they are created.

Local variables must be declared before they can be used. The local declaration is used to declare one or more local variables. A declaration consists of the keyword **local** followed by a list of variable declarations separated by commas. Each variable declaration consists of the name of the variable, a colon, and a specification of the type of values that will be assigned to the variable. The following are legal declarations.

```
    local  &I:integer, &J:integer, &TMP:real;
```

5.3 Formals

A formal parameter is a variable that is used to pass values into a routine, and sometimes to pass values out of the routine. (For a description of the mechanism by which the values are passed, see Section 9.5.) A formal parameter may be used only in the routine in which it is declared. The formal parameters in a routine must have distinct names, and the names must be distinct from the names of the routine's local variables. The names need not be distinct from the names of variables used in other routines or from the global variables in the program. However, if a formal parameter has the same name as a global variable, the global variable cannot be referenced in the routine.

Formal parameters have an attribute called the w-attribute. The w-attribute indicates how the parameter will be used in the procedure. If the w-attribute is null, parameter is read-only; that is, it may occur in expressions in the routine, but it may not be assigned to. If the w-attribute is **ref** the parameter may be both used in expressions and assigned to. If the w-attribute is **wme** the parameter may be used in expressions and it may be used in modify or remove statements; it may not be assigned to outside of modify statements however.

[1]If the routine in which a local variable is declared is known not to be recursive, the compiler may allocate space for the variable permanently. However, programs should not take advantage of this fact because other versions of the compiler may not allocate the space permanently.

5.3.1 Formals in Procedures and Functions

```
<formals>          ::=     ( <formalparm> { , <formalparm> } )
                    |      ( )

<formalparm>       ::=     <w-attribute> <variable> : <typename>

<w-attribute>      ::=     ref
                    |      wme
                    |      <null>
```

The formal parameters in functions and procedures are declared in a parenthesized list that follows the name of the function or procedure. The items in the list are separated by commas. Each item consists of an indicator of the w-attribute of the parameter, the name of the parameter, a colon, and an indication of the type of the values to be associated with the formal. Procedures and functions may have a minimum of zero parameters and a maximum of six parameters.

5.3.2 Formals in Rules

The formal parameters in rules are declared by being used in the LHS of the rule (see Section 8.1). If a variable occurs in an LHS, it is a formal parameter whose type is the type of the condition element that follows it. The w-attribute of a variable in the LHS of a rule is null (read only). The formal parameters can also be used in the body of the rule; there, the w-attribute of the parameter is wme.

5.4 The @ Pseudo Variable

In condition elements and in make and modify statements, the @ pseudo variable is defined. The @ can be used exactly like a variable in these contexts. In a condition element it is used to refer to the working memory elements that the matcher attempts to use to satisfy the condition element. If the condition element is preceded by a variable, the @ may be used interchangeably with the variable. The type of the @ in a condition element is the type of the condition element. In make and modify statements it refers to the working memory elements that the statements operate on. The type of the @ in a make or modify is the type of the element being operated on. The @ is a kind of formal parameter. In condition elements the w-attribute of the parameter is null (read only). In make and modify statements, the w-attribute is ref.

5.5 L-values

```
<l-value>                 ::=     <variable>  <selector-list>
                          |       @  <selector-list>

<selector-list>           ::=     .  <field-name>  <selector-list>
                          |       [  <expr>  ]  <selector-list>
                          |       <null>

<field-name>              ::=     <symbol>
```

An l-value is a construct in the language that evaluates to the address of a location in the computer's memory. The simplest l-value is a variable by itself; such an l-value evaluates to the memory location of the variable. If a variable has a record or element type, the variable may be followed by a period and the name of a field in the record or element. Such an l-value evaluates to the memory location of the designated field within the variable. If a variable has an array type, the variable may be followed by an expression in square brackets. The expression must evaluate to an integer. If the value is K, the l-value evaluates to the memory location of the Kth field.

Since values comprising arrays, records, and elements may themselves be records or arrays, an l-value may contain a sequence of field selectors. The selectors are applied in order from left to right. For example, in

```
    &X.fld[3][7]
```

the `.fld` is applied first to get the address of a field within the record &X. The value stored in this field is an array. The `[3]` is then applied to get the address of a field within the array. The value stored in this field is another array. Finally the `[7]` is applied to get the address of a field within that array.

The fields that are designated in l-values must exist. If a specified field does not exist in a record or element, a compile-time error is generated. If a constant is used in an array field reference, and if the constant is too big or too small, a compile-time error is generated for that also. If a non-constant expression is used in an array field reference, the value of the expression is checked at run time. A value that is too big or too small in the LHS of a rule results in failure of the term containing the array reference. No error message is given in this case. In any other context, a value that is too big or too small results in a run-time error.

6. Expressions

An expression is a description of a method for computing a value from other values. An expression may contain variables, constants, operators, function calls, and parentheses. The variables and constants provide the initial values to be used in the computation. The operators and the function calls specify the operations that are to be performed. The parentheses along with a collection of rules of precedence determine the order in which the operations are performed.

When an expression is evaluated, the operators and functions are applied one at a time. Each operator takes one or two values, performs a calculation using the values, and produces one value as its result. A function takes some number (possibly zero) of values, performs a calculation using them, and produces one value as its result. The result values are operated on by other operators and functions. Finally a single result value is obtained for the expression. For an example of how this works, consider the following expression.

```
    &X * 3 - &Z * 2
```

Suppose the value stored in the the memory location of **&X** is the integer 5 and the value stored in the memory location of **&Z** is 7. Then the evaluation of the expression proceeds as follows: First the 5 stored in **&X** is multiplied by 3 to produce 15. Then the 7 stored in **&Z** is multiplied by 2 to produce 14. Finally the 14 is subtracted from the 15 to produce 1. This is the result value for the expression.

6.1 Logical Operators

```
<expr>                  ::=     ~  <disjunction>
                        |       <disjunction>

<disjunction>           ::=     <conjunction>  {  \/  <conjunction>  }

<conjunction>           ::=     <relation>  {  /\  <relation>  }
```

The logical operators are ~, \/, and /\. These operators may be used only with values of type logical or types derived from logical. The first operator, ~, computes the logical not of its operand. When ~ is applied to 0b, it returns 1b; when it is applied to 1b, it returns 0b. The second operator, \/ computes the logical or of its operands. The result of an \/ expression is 1b if either of its operands is 1b, and 0b if both of the operands are 0b. The third operator, /\ computes the logical and of its operands. The result of an /\ expression is 0b if either of its operands is 0b, and 1b if both of the operands are 1b.

Expressions containing /\ and \/ are not always evaluated fully. An expression containing /\ or \/ is evaluated as follows:

- The evaluation proceeds from left to right in the expression.
- In a sequence of /\ operators, as soon as one of the operands evaluates to **0b**, evaluation of the sequence terminates.
- In a sequence of \/ operators, as soon as one of the operands evaluates to **1b**, evaluation of the sequence terminates.

For an example of this, consider the following expression in which *E1*, *E2*, and *E3* are complex logical-valued expressions.

E1 /\ *E2* /\ *E3*

E1 is evaluated first. If it evaluates to 0b, *E2* and *E3* will not be evaluated. It is not necessary to evaluate them because regardless of the values they return, the /\ expression will evaluate to 0b.

6.2 Relational Operators

```
<relation>              ::=     <simplevalue>  <relop>  <simplevalue>
                        |       <simplevalue>


<relop>                 ::=     =  |  <>  |  <  |  >  |  <=  |  >=
```

The relational operators are =, <>, <, >, <=, and >=. All of these operators return logical values; they compare their operands and return 1b if they are related in a certain way, and 0b if they are not. The operators = and <> may be used with operands of any scalar type, though both operands must have the same type. The first returns 1b if its operands are equal. The second returns 1b if its operands are different. The operators <, >, <=, and >= may be used only with integers, reals, and types derived from integer or real. The two operands must have the same type. The operator < returns 1b if the first operand is less than the second; > returns 1b if the first operand is greater than the second; <= returns 1b if the first operand is less than or equal to the second; and >= returns 1b if the first operand is greater than or equal to the second.

6.3 Arithmetic Operators

```
<simplevalue>           ::=     -  <uvalue>
                        |       <uvalue>


<uvalue>                ::=     <term>  {  <addop>  <term>  }


<addop>                 ::=     +  |  -


<term>                  ::=     <factor>  {  <mulop>  <factor>  }
```

```
<mulop>                  ::=     *  |  /  |  \
```

The arithmetic operators are *, /, \, +, and -. The last of these has both a binary and a unary form; that is, it can be used either with two operands or with a single operand. All the operators may be used with operands of type integer and any type derived from integer. All the operators except \ may also be used with operands of type real and any type derived from real. In all cases, both operands must have the same type. The type of the result is the same as the type of the operands. The operator * multiplies its operands. The operator / divides the first operand by the second and produces the quotient as its result. The operator \ divides the first operand by the second and produces the remainder as its result. The operator + adds its operands together. The binary operator - subtracts the second operand from the first. The unary operator - multiplies its operand by -1.

6.4 Constants and L-values

```
<factor>                 ::=     <l-value>
                         |       <integer>
                         |       <real>
                         |       <logical>
                         |       <char>
                         |       <symbol>
                         |       <funcall>
                         |       <parenthesized-expr>
```

Constants and l-values provide the initial values used in an expression. An l-value in an expression is evaluated to produce the address of a memory location, and the value stored in that location is retrieved and used in the computation. The type of the value is the type of the l-value. Where a constant occurs in an expression, the value of the constant is used in the computation.

6.5 Function Calls

```
<funcall>                ::=     <symbol>  (  <arglist>  )
```

When a function call occurs in an expression, the arguments are evaluated and the routine is called as usual. (See Section 9.5). The value returned by the function is then used in computing the result of the expression. The type of the value is the type that was declared for the function.

6.6 Parenthesized Subexpressions

```
<parenthesized-expr>     ::=     (  <expr>  )
```

Parentheses are used in expressions to indicate the order in which operators are to be applied. (See the following section.)

6.7 Rules of Precedence

The rules of precedence determine the relative order in which the operators in an expression should be applied. The rules impose a partial ordering on the operators in the expression; when no relative ordering can be determined for two operators, the operators may be applied in any order.

Before the rules of precedence can be described, two terms must be defined. The first is the term "competing." Two operators are said to be competing for a value if the application of either operator would consume the value and make it unavailable for the other operator. Consider for example the following expression.

```
    &X + 2 * &Y
```

If either operator in this expression was applied, the 2 would be consumed. Thus the operators are competing for the 2. The other term is "binding strength." The binding strengths of operators are numeric values that are used to determine the relative order in which competing operators should be applied. The binding strengths of the operators, from most strongly binding to least strongly binding are:

1. *   /   \
2. +   - (binary)
3. - (unary)
4. =   <>   <   >   <=   >=
5. /\
6. \/
7. ~

The rules of precedence in OPS83 are

- Operators inside a set of parentheses are performed before any operator which uses the result of the parenthesized subexpression.
- If alpha and beta are competing operators, and alpha binds more tightly than beta, then alpha is applied before beta. Thus, for example, because * binds more tightly than +, and because application of either * or + would consume the 2, in the following expression, * is evaluated first.

  ```
      &X + 2 * &Y
  ```
- If alpha and beta are competing operators that have equal binding strengths, and if alpha precedes beta in the expression, then alpha is applied before beta. Thus, since + and - bind equally tightly, the operators in the following expression are applied from left to right.

```
&A - &B + &C
```

6.8 Reordering Expressions

The compiler may take advantage of algebraic properties such as associativity and commutativity to reorder the operations in an expression. The compiler may reorder expressions even in the presence of parentheses. The reordered calculation will always be algebraically equivalent to a calculation performed in strict compliance with the rules of precedence. However, because of round-off errors in finite precision arithmetic, the values that result from the two calculations may differ.

The compiler will not reorder \/ or /\ expressions. That is, if an expression contains a sequence of \/ or /\ operators, the expression is evaluated exactly as specified in Section 6.1. If the arguments of the \/ and /\ operators are expressions containing other operators, the operators in the arguments may be reordered. Thus in the following if *E1* and *E2* are expressions that do not involve /\ or \/, then it is guaranteed that *E1* will be evaluated before *E2* (and *E2* may never be evaluated), but the operators in *E1* and *E2* may be reordered.

```
E1 \/ E2
```

7. Statements

```
<statement-list>        ::=     <stmt>  {  ;  <stmt>  }  <opt-;>

<stmt>                  ::=     <assignment>
                        |       <cast>
                        |       <if>
                        |       <while>
                        |       <for>
                        |       <return>
                        |       <call>
                        |       <read>
                        |       <write>
                        |       <wm>
                        |       <on>
                        |       <compound>
```

A statement list consists of a sequence of statements separated by semicolons. An optional semicolon may be placed after the last statement in the list. When a statement list is executed, the statements are executed in the order in which they are written.

7.1 Assignment statements

```
<assignment>            ::=     <l-value>  =  <expr>
```

The assignment statement is used to change the value stored in a variable's memory location. The l-value is evaluated to produce the address of a location in the computer's memory. The expression is evaluated to produce a value. The value is then stored in the l-value's location. The type of the expression must be the same as the type of the l-value. The type must be a scalar type.

7.2 Cast

```
<cast>                  ::=     cast  <l-value>  =  <expr>
```

The cast statement is a variant of the assignment statement. The l-value and the expression are evaluated as in the ordinary assignment statement. If the type of the expression is the same as the type of the l-value, the value is stored into the l-value's location, just as it is in the ordinary assignment statement. When the types are not equal, the cast statement attempts to change the type of the value to the type of the l-value. There are two cases the cast statement has to be able to handle:

- If the type of the l-value and the type of the value are derived from a common type, the type of the value can be changed without affecting the value itself.

- If the two types are not derived from a common type, the value has to be converted. Cast will convert from integer to real, from integer to char, from real to integer, from real to char, from char to integer, and from char to real. In addition, it will convert between types that are derived from these types.

Cast can work with only scalar types.

7.3 If

```
<if>                ::=     if ( <expr> ) <statement-1> <endif>

<statement-1>       ::=     <statement>

<endif>             ::=     else <statement-2>
                    |       <null>

<statement-2>       ::=     <statement>
```

The if statement is used to execute statements conditionally. The parenthesized expression must return a value of type logical or a type derived from logical. When the if statement is executed, the expression is evaluated. If it evaluates to 1b, the statement called statement-1 above is executed. If it evaluates to 0b, the statement called statement-2 is executed (if that statement is present).

7.4 While

```
<while>             ::=     while ( <expr> ) <statement>
```

The while statement is used to execute statements repeatedly. The parenthesized expression must evaluate to a value of type logical or a type derived from logical. A while statement is executed by performing the following loop

1. Evaluate the parenthesized expression; if it evaluates to 0b, exit the while statement.

2. Execute the statement.

3. Goto step 1.

7.5 For

```
<for>               ::=     for <variable> = <range> <statement>

<range>             ::=     ( <expr> <forincr> <expr> )

<forincr>           ::=     to | downto
```

The for statement is used to execute statements repeatedly while a variable is stepped through the values in a range. If *V* is a variable, *E1* and *E2* expressions, and *S* a statement, a for statement of the form

```
        for V = (E1 to E2) S;
```

is equivalent to the following, where *T* is a variable that does not occur elsewhere in the routine.

```
        V = E1;
        T = E2;
        while (V <= T)
          {
            S;
            V = V + 1;
          };
```

And a for statement of the form

```
        for V = (E1 downto E2) S;
```

is equivalent to the following.

```
        V = E1;
        T = E2;
        while (V >= T)
          {
            S;
            V = V - 1;
          };
```

Note that in both cases, the expressions designating the initial and final values are evaluated only once. The two expressions may be evaluated in any order. The variable and the the two expressions must have the same type; the type must be either integer or a type derived from integer.

7.6 Return

```
<return>                ::=     return
                        |       return  (  <expr>  )
```

The return statement is used to exit from a function, procedure, or rule. When the return statement is executed, control returns to the point from which the function, procedure, or rule was called. The first form of return (without a parenthesized expression) must be used in procedures. The second form (with a parenthesized expression) must be used in rules and functions. The expression specifies the value that the function or rule is to return. The type of the expression must be the same as the type of the rule or function.

7.7 Call

```
<call>                  ::=     call  <symbol>  (  <arglist>  )
```

The call statement is used to invoke a procedure. The symbol is the name of the procedure. When the call statement is executed, the argument list is evaluated (see section 9.5) and control is passed to the routine.

7.8 Input and Output

Two statements, read and write, are used to perform input and output.

7.8.1 Read

```
<read>              ::=     read  (  <file>  )
                            <l-value>  {  ,  <l-value>  }

<file>              :=      <expr>  |  <null>
```

The read statement is used to input information from a file. If the parentheses contain an expression, the type of the expression must be integer. The expression must evaluate to the number of a file that is open for input. If the parentheses do not contain an expression, the value 0 is used by default. The information that is read is taken from the specified file.

Each l-value is processed to produce the address of a memory location, and the next value read from the file is stored into that location. The l-values must have scalar types. The way the values are read depends on the type of the l-value. If the l-value has type char or a type derived from char the next character, whatever it is, is removed from the input stream and stored into the l-value's location. If the l-value has type integer, real, logical, symbol, or a type derived from one of these the following sequence of operations are performed.

1. Read and discard any initial separator characters.
2. Read and accumulate the characters in the file up to the next separator character. Do not read the separator character from the file.
3. Try to interpret the accumulated characters as a constant of the required type.

How the characters are interpreted in the last step depends on the type of the l-value. If the type is logical or a type derived from logical, the characters must be either **0b**, **0B**, **1b**, or **1B**. If the type is symbol or a type derived from symbol, the characters are interpreted as a symbolic value, regardless of what they are. The only restriction is that the number of characters accumulated must not exceed the number of characters permitted in a symbol. If the type is integer, real, or a type derived from one of these, quite a bit of flexibility is allowed. The characters should generally conform to the descriptions of integer and real constants given in Sections 3.2.1 and 3.2.2. However, read will tolerate some deviation from the standard descriptions. In particular, a leading sign of + or - is permitted.

Attempting to read past the end of a file causes a run time error.

7.8.2 Write

```
<write>                 ::=     write  (  <file>  )  <expr>  {  ,  <expr>  }
```

The write statement is used to output information to a file. If the parentheses contain an expression, the type of the expression must be integer. The expression must evaluate to the number of a file that is open for output. If the parentheses do not contain an expression, the value 1 is used by default. The information that is written is sent to the specified file.

The expressions are evaluated, and the values that they return are written onto the specified file. The values must have scalar types. If the value has type char or a type derived from char, the character is written directly to the file. If the value has type integer, real, logical, or a type derived from one of these, the value is converted to a text form that would allow it to be read back in with a read statement. If the value has type symbol or a type derived from symbol, the characters comprising the symbol are written directly to the file.

7.8.3 Standard Files

Three standard files are always defined in programs: file 0 which is the standard input file, file 1 which is the standard output file for most output, and file 2 which is the standard output file for error messages. Ordinarily these are not true files but rather connections to the terminal from which the program is being run.[2]

7.9 Working Memory Modification

```
<wm>                    ::=     <make>  |  <modify>  |  <remove>
```

Three statements are included in the language to manipulate working memory elements: make, modify, and remove statements.

7.9.1 Make

```
<make>                  ::=     make  (  <symbol>  <rhstermlist>  )
```

The make statement is used to create a new element and add it to working memory. The symbol just after the open parenthesis indicates the type of element to be created. The type must be an element type. The list of rhs terms indicates the values that are to be put into the fields of the element. The rhs terms are evaluated as described in Section 7.9.4. Any fields that do not receive explicit values are given the following default values:

[2]It is possible to change the connections for the standard files by use of the operating system facilities. The standard files correspond to STDIN, STDOUT, and STDERR in C programs, and they are changed the same way as STDIN, STDOUT, and STDERR.

Type of Field	Default value
integer	0
real	0.0
logical	0b
symbol	||
char	ASCII 0

A field of type T, where T is derived from type B, is given the default for type B. When a make is executed, the following sequence of actions occurs:

1. An element is created and all its fields are set to the defaults.
2. The rhs terms are evaluated.
3. The element is put into working memory.

7.9.2 Modify

```
<modify>                ::=     modify  <variable>  (  <rhstermlist>  )
```

The modify statement is used to change one or more fields of an existing element. The variable in the statement must have been declared to be an element-class structure. The variable must be a formal parameter of the routine in which the remove statement occurs, and it must have a w-attribute of wme. When the modify statement is executed, the following sequence of actions occurs:

1. The element is removed from working memory.
2. The variable's w-attribute is changed from wme to ref.
3. The rhs terms are evaluated.
4. The variable's attribute is changed back to wme.
5. The element is put back into working memory.

7.9.3 Remove

```
<remove>                ::=     remove  <variable>  {  ,  <variable>  }
```

The remove statement is used to delete elements from working memory. The variables in the statement must have element types. That is, the types must have been declared to be element-class structures. The variables must be formal parameters of the routine in which the remove statement occurs, and they must have w-attributes of wme. When the statement is executed, the elements that the variables refer to are immediately removed from working memory. However, the elements are not deleted as long as any formal parameters are bound to them.

7.9.4 Rhs Terms

```
<rhstermlist>         ::=     <rhsterm>  {  ;  <rhsterm>  }  <opt-;>

<rhsterm>             ::=     <field-name>  <selector-list>  =  <expr>
                      |       <call>
```

The rhs terms in a make or modify specify the fields that are to be set or changed when the statement is executed. Semicolons (;) are used to separate the terms in the list. An optional semicolon may be used after the last term in the statement. There are two forms of rhs terms. The first form specifies a field in the element and an expression to be evaluated and put into that field. The field must have a scalar type. The type of the expression must be the same as the type of the field. The second form of rhs term is an ordinary procedure call. This form is provided in order to permit the element (or a part of the element) to be passed to a routine which will fill in some of the fields of the element. The following statement illustrates the use of both kinds of rhs terms.

```
    modify &3
        (on = table;                    -- assigning to a field
         call set(@.loc, &4.loc));      -- calling a procedure
```

7.10 On Statements

```
<on>                  ::=     on wmchange call <onroutine>

<onroutine>           ::=     <symbol>
```

The on statement is used to specify a procedure to be called every time an element is added to or deleted from working memory. The statement

```
    on wmchange call P;
```

specifies that *P* is to be called after every change. The procedure in an on statement must have no parameters. It need not be a simple procedure. The on statement may be executed any number of times during the execution of a program. At any given time, the routine that is called is the routine that was specified in the last on statement executed. The procedure is called immediately after elements are deleted from working memory by remove or modify actions, and immediately after elements are added by make or modify actions. (Thus a modify action results in the procedure being called twice.)

7.11 Compound Statements

```
<compound>            ::=     "{"  <statement-list>  "}"
```

The compound statement permits a sequence of statements to be used where a single statement would otherwise occur. In particular, the statements in if, for, and while statements can be compound statements. When a compound statement is executed, the individual statements in the list are executed in sequence.

8. Rule LHSs

```
<lhs>                   ::=     <posce>  {  ;  <anyce>  }  <opt-;>


<anyce>                 ::=     <posce>
                        |       <negce>


<negce>                 ::=     ~  <pattern>
```

The part of a rule which preceeds the arrow (the -->) is called the rule's LHS (left hand side). Each LHS consists of a positive condition element followed by zero or more positive or negative condition elements. A negative condition element is a condition element that is preceded by the operator ~. The condition elements are separated by semicolons. An optional semicolon may follow the last condition element.

The LHS is a test of the contents of working memory. When the LHS is satisfied, the rule is placed into the conflict set. The LHS is considered satisfied when

- All the positive conditions are satisfied, and
- None of the negative conditions are satisfied.

8.1 Variables in the LHS

```
<posce>                 ::=     <variable>  <pattern>
                        |       <pattern>
```

Variables may be placed before non-negated condition elements. When an attempt is made to satisfy a condition element with a working memory element, the variable is bound to the working memory element. The variable can be used to refer to the working memory element in expressions in that condition element as well as in expressions in condition elements that occur later in the LHS. (See also Section 5.3.2).

8.2 Condition Elements

```
<pattern>               ::=     (  <symbol>  <lhstermlist>  )
```

A condition element is a pattern to be compared to the elements in working memory. A condition element consists of an open parenthesis, a symbolic constant, a list of LHS terms, and a close parenthesis. The symbol indicates the type of the pattern. The pattern is considered satisfied by a given working memory element if the following two conditions are met.

- The type of the pattern is the same as the type of the working memory element.

- All the terms of the pattern are satisfied.

8.3 LHS Terms

```
<lhstermlist>            ::=      <lhsterm>  {  ;  <lhsterm>  }  <opt-;>
                          |       <null>


<lhsterm>                ::=      (  <expr>  )
                          |       <field-name> <selector-list> <relop> <expr>
                          |       <funcall>
```

The terms in a condition element are separated by semicolons. An optional semicolon may be placed after the last term. There are three kinds of terms. The most general kind of LHS term is a parenthesized expression. The type of the expression must be logical or a type derived from logical. If the expression evaluates to **1b** it is considered satisfied; if it evaluates to **0b** it is considered not satisfied. The other two kinds of terms are simply more compact ways to write common cases of the first kind of term. The second kind of LHS term has the form of a field selector followed by a relational operator followed by an expression. If *F* is a field designator, *r* a relational operator, and *E* an expression, then a term of the form

F r E;

is equivalent to the following general term.

(@.*F r E*);

The third form of LHS terms is a function call. The type of the function must be logical or a type derived from logical. If *f(a)* is a function and its argument list, then a term of the form

f(a);

is equivalent to the following general term.

(*f(a)*);

As an example, the following condition element contains one term of each kind.

```
  (tank
      (@.name = T1 \/ @.name = T2);        -- general term
      status = lost;                       -- field selector term
      significant(@.change))               -- function call
```

9. Functions, Procedures, and Rules

A routine is a block of executable statements which can be invoked (i.e., caused to be executed) from other places in the program. There are three kinds of routines: procedures, functions, and rules. They differ in how they are invoked and in whether they return values.

9.1 Procedures and Functions

```
<procedure>              ::=     <ordinary-procedure>
                         |       <simple-procedure>
                         |       <forward-procedure>
                         |       <external-procedure>


<function>               ::=     <ordinary-function>
                         |       <simple-function>
                         |       <forward-function>
                         |       <external-function>
```

A procedure is a routine that is called in a call statement. A function is a routine that is called by evaluating a reference to the function in an expression. Because they are used in expressions, functions must return values. Procedures do not return values. There are several kinds of functions and procedures; the differences between them are explained below.

9.1.1 Ordinary Procedures and Functions

```
<ordinary-procedure>     ::=     <procedure-header>   "{"   <body>   "}"

<procedure-header>       ::=     procedure   <symbol>   <formals>

<ordinary-function>      ::=     <function-header>   "{"   <body>   "}"

<function-header>        ::=     function   <symbol>   <formals>
                                     :   <typename>
```

The "ordinary" function and procedure definitions are the most commonly used definitions. The other forms of function and procedure definitions are used only in special circumstances (which are described in the following sections). The ordinary function definition consists of the keyword **function**, the name of the function, a parenthesized list of formals, a colon, a specification of the type returned by the function, and finally the body of the function. The type of the function must be a scalar type. The ordinary procedure definition consists of the keyword **procedure**, the name of the procedure, a parenthesized list of formals, and the body of the procedure.

9.1.2 Simple Procedures and Functions

```
<simple-procedure>        ::=      simple  <procedure-header>
                                       "{"  <body>  "}"


<simple-function>         ::=      simple <function-header>
                                       "{"  <body>  "}"
```

The declaration of a simple procedure or function is identical to the declaration of an ordinary procedure or function except that the keyword **simple** is placed before the declaration. A simple routine is eligible to be called from simple contexts. The body of a simple function or procedure must conform to the rules for simple contexts. See Section 9.4 for a discussion of simple contexts.

9.1.3 External Procedures and Functions

```
<external-procedure>      ::=      external  <procedure-header>


<external-function>       ::=      external <function-header>
```

External routine definitions are used for routines that are written in other programming languages, and that will be linked with the OPS83 routines. The external definition indicates the parameters that the routine expects, and if the routine is a function, the type of the value that it returns. In external routine definitions, the routine body must not be present. It is required that all the formal parameters of an external routine be declared to have a w-attribute of ref.

9.1.4 Forward Procedures and Functions

```
<forward-procedure>       ::=      <procedure-header>  forward
                          |        simple  <procedure-header>  forward


<forward-function>        ::=      <function-header>  forward
                          |        simple  <function-header>  forward


<forwardbody>             ::=      body  <symbol>  "{"  <body>  "}"
```

In OPS83, a routine must be declared before any calls on the routine occur in the program. Generally this is not a problem; it merely means that the routine declarations must be ordered properly -- with the lower level routines being declared earlier in the program. In one case, however, this is a problem: when the program contains mutually recursive routines. Forward routine declarations are used in this case; the forward declaration permits the header of a routine to be separated from the body of the routine. Thus even when mutually recursive routines are compiled, it is possible for all the routine headers to be processed before any of the bodies are compiled. In a forward declaration, the keyword **forward** is written instead of the routine body. Later in the same module the keyword `body` must occur followed by the name of the routine and the routine body in braces. Both simple and non-simple routines may be declared forward.

9.2 Rules

```
<rule>                  ::=     rule  <symbol>  "{"  <lhs>  -->  <body>  "}"
```

A rule is a routine that is called by context rather than by specifying the name of the routine explicitly. A rule definition consists of the keyword `rule`, the name of the rule, an open brace, the LHS of the rule, a right pointing arrow, the body of the rule, and a closing brace. The LHS of a rule is described above in Section 8. The body of a rule is precisely the same as the body of a function. The LHS of a rule is a simple context; the body of a rule is not. Rules are implicitly defined to return values of type symbol.

9.3 Routine Bodies

```
<body>                  ::=     {  <local>  ;  }  <statement-list>
```

The body of a routine consists of optional local variable declarations followed by a list of executable statements. When the routine is executed, the statements in the list are executed in order.

9.4 Simple Contexts

The simple contexts in a program are the LHS's of the rules plus the bodies of any functions or procedures that are defined to be simple. In a simple context the following things are prohibited:

- L-values involving global variables.
- Read, write, on, make, modify, and remove statements.
- Calling non-simple routines.

9.5 Actual-Formal Parameter Correspondence

Two kinds of parameters are involved in routine calls:

- Formal parameters. These are the variables that are defined in the header in procedures and functions. They are used in the routine bodies to refer to the values passed to the routines.
- Actual parameters. These are the values passed to the routines. They are produced by evaluating a list of expressions at the call site.

OPS83 uses call-by-reference. That means that each actual parameter defines a location in memory. During the evaluation of the body of the routine, when a formal parameter is used in an expression, the value stored in the memory location of the corresponding actual parameter is retrieved; and when the formal parameter is assigned to, the new value is stored into that location. To determine the address to pass for each actual parameter, the compiler does the following:

- If the actual parameter is an l-value, the address of the l-value is passed.
- If the actual parameter is not an l-value, a temporary location is allocated, the value of the actual parameter is copied into the temporary, and the address of the temporary is passed to the routine.

9.5.1 Restrictions on Actual Parameters

The actual parameters in a function or procedure call must meet the following requirements:

- The number of actual parameters provided must be equal to the number of formal parameters of the routine.
- The type of each actual parameter must be the same as the type of the corresponding formal parameter.
- The w-attributes of each actual parameter must be compatible with the w-attributes of the corresponding formal parameter.

To determine whether the w-attributes of formal and actual parameters are compatible, the following procedure is used: First the w-attributes of the actual parameters are determined. L-values involving local and global variables and expressions that are not l-values are defined to have w-attributes of ref. L-values involving formal parameters of the routine being executed are defined to have the w-attributes of the formal parameters. Second, the w-attribute of the actual parameter is compared to the w-attribute of the formal parameter. If the w-attribute of an actual parameter is ref, the w-attribute of the formal must be either ref or null. If the w-attribute of an actual parameter is wme, the w-attribute of the formal must be either wme or null. If the w-attribute of an actual parameter is null, the w-attribute of the formal must be also be null.

9.6 Default Values

If the end of the list of statements in a routine body is reached, control returns from the routine to the site where the routine was called. If control is returned from a function or rule this way rather than through the execution of a return statement, a return value is supplied by default. The value returned depends on the type of the function or rule; the following table shows the values used.

Type	Value
integer	0
real	0.0
logical	0b
symbol	||
char	ASCII 0

If a function has a type T that was derived from primitive type B, the default value for type B is returned.

10. Programs and Modules

A program consists of a collection of type declarations, global variable declarations, and routine definitions that can be linked together and executed. A program may be divided into units called modules. A module is a piece of a program that can be compiled separately.

10.1 Modules

```
<module>              ::=    <progunit>  {  ;  <progunit>  }  <opt-;>

<progunit>            ::=    <use>
                      |      <global>
                      |      <type>
                      |      <rule>
                      |      <procedure>
                      |      <function>
                      |      <forwardbody>
```

A module is a file containing a sequence of use statements, global variable declarations, type declarations, and routine declarations. They are separated by semicolons. An optional semicolon may follow the last unit in the module. The order in which the units are presented is not important except that things must be defined before they are used. Forward routines are supported (see Section 9.1.4) so that mutually recursive routines may be used.

10.2 Use Statement

```
<use>                 ::=    use  <symbol>
```

The use statement is provided to allow the code in one module to make use of types, variables, and routines defined in another module. The symbol should be the name of the file that contains the other module, without any extension. All the type, variable, and routine definitions are available after the use statement is processed.

10.3 The Names of Routines and Global Variables

The names of the routines in the compiled code are the same as the names of the routines in the source code. It is the programmer's responsibility to insure that the names used for the routines are legal names for the operating system being used. In addition, if the operating system does not distinguish between upper and lower case characters, it is the responsibility of the programmer to insure that the names of the routines are distinct after case folding. The names of global variables in the compiled code are the same as the names of the global variables in the source code except that if the & is not a legal character for the system, the compiler will change it to a legal character. The compiler that runs on VAX/VMS systems changes the & to a period (".").

10.4 Main Procedure

Every program must contain one procedure named `main`. This procedure must not have any formal parameters. The procedure main is called automatically when the program is executed.

10.5 Guidelines for Modular Compilation

Some care must be taken with programs that are divided into modules. If the following guidelines are not followed, in some cases programs will compile without error messages but then fail to execute correctly:

- Every module in the program should be mentioned in a use statement in the module containing the main procedure. The use statements are necessary in order for the compiler to generate proper initialization code for the program.
- When a module is changed and recompiled, all the modules that use information defined in that module should also be recompiled.

11. The Interpreter's Data Structures

The interpreter maintains two data structures which the programmer should be aware of: working memory and the conflict set.

11.1 Working Memory

The working memory of a production system is the set of all the working memory elements that are in existence at any given time. When the execution of a production system begins, the working memory is empty. When make statements are executed, elements are added to working memory. When remove statements are executed, elements are deleted from working memory.

11.2 Conflict Set

An instantiation is an ordered pair of a rule name and a list of working memory elements that match its LHS. The conflict set is the set of all the current instantiations in the system. The OPS83 interpreter keeps the conflict set current. Every time a working memory element is added to or deleted from working memory, the interpreter evaluates the LHS's in the system to determine what effect that change has on the conflict set and updates the conflict set accordingly.

The instantiations in the conflict set are numbered from 1 through K, where K is the number of current instantiations. These numbers are used by several of the standard routines to designate instantiations (see Section 12.1). The numbers have no significance, and the number that is assigned to an instantiation may change during the course of a run. However, it is guaranteed that the numbers will change only when the contents of working memory change. Thus as long as no make, modify, or remove statements are executed, the numbers will remain unchanged.

12. Standard Routines

The OPS83 system includes a number of predefined routines. The following sections describe the predefined routines, giving a synopsis of their definitions and an explanation of what they do.

12.1 Using the Conflict Set

12.1.1 fire

DEFINITION:

```
    function fire(&INST:integer):symbol
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. The instantiation designated by &INST is executed. If &INST is not the number of an instantiation in the conflict set, no instantiation is executed. It returns the value returned by the rule if an instantiation is executed, || if not.

12.1.2 cssize

DEFINITION:

```
    function cssize():integer
```

DESCRIPTION: The function returns the number of instantiations in the conflict set.

12.1.3 irule

DEFINITION:

```
    function irule(&INST:integer):symbol
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. The function returns the name of the rule in instantiation number &INST. If &INST is not the number of an instantiation, it returns ||.

12.1.4 iuse

DEFINITION:

```
    function iuse(&INST:integer):integer
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. The function returns the number of times that instantiation &INST has been fired. If &INST is not the number of an instantiation, it returns -1.

12.1.5 iccnt

DEFINITION:

```
    function iccnt(&INST:integer):integer
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. The function returns the number of condition elements that the rule in instantiation &INST contains. If &INST is not the number of an instantiation, it returns -1.

12.1.6 iwcnt

DEFINITION:

```
    function iwcnt(&INST:integer):integer
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. The function returns the number of working memory elements that instantiation &INST contains. If &INST is not the number of an instantiation, it returns -1.

12.1.7 itag

DEFINITION:

```
    function itag(&INST:integer, &WME:integer):integer
```

DESCRIPTION: &INST should be the number of an instantiation in the conflict set. &WME should be the number of a working memory element in the instantiation. The function returns the time tag of working memory element &WME in instantiation &INST. If &INST is not the number of an instantiation, or &WME is not the number of a working memory element in the instantiation, it returns -1.

12.2 Examining the Partial Matches of Rules

12.2.1 cmatches

DEFINITION:

```
    function cmatches(&R:symbol, &C:integer):integer
```

DESCRIPTION: &R should be the name of a rule. &C should be the number of a condition element in the rule's LHS. The function returns a count of the number of working memory elements that match condition element number &C of rule &R when the condition element is considered in isolation. That is, the number of working memory elements that match the condition element if the terms involving variables bound by other condition elements are ignored. If &R is not the name of a rule, or &C not the number of a condition element in the rule, it returns -1.

12.2.2 pmatches

DEFINITION:
```
    function pmatches(&R:symbol, &C:integer):integer
```

DESCRIPTION: &R should be the name of a rule. &C should be the number of a condition element in the rule's LHS. The function returns a count of the number of partial instantiations there are for condition elements number 1 through &C of rule &R. That is, the number of instantiations the rule would have if its LHS contained only the first &C of the condition elements. If &R is not the name of a rule, or &C not the number of a condition element in the rule, it returns -1.

12.3 Tracing Working Memory Changes

The following routines are used to determine what kind of change was last made to working memory. These routines are used in constructing routines to trace working memory activity.

12.3.1 wmctype

DEFINITION:
```
    function wmctype():symbol
```

DESCRIPTION: The function returns the type of the working memory element that was processed in the last make, modify, or remove statement executed.

12.3.2 wmctag

DEFINITION:
```
    function wmctag():integer
```

DESCRIPTION: The function returns the time tag of the working memory element that was processed in the last make, modify, or remove statement executed.

12.3.3 wmcadd

DEFINITION:
```
    function wmcadd():logical
```

DESCRIPTION: The function returns 1b if the last change to working memory involved adding an element. It returns 0b if the last change involved deleting an element.

12.3.4 wmcact

DEFINITION:

```
    function wmcact():symbol
```

DESCRIPTION: The function returns **make** if the last working memory change was caused by a make statement, **modify** if the last change was caused by a modify statement, and **remove** if the last change was caused by a remove statement.

12.4 Manipulating Files

The following three functions are used to open files for input and output and to close files.

12.4.1 open

DEFINITION:

```
    function open(&F:symbol):integer
```

DESCRIPTION: &F should be the name of a file that exists on the system. The function opens the file for reading, and returns an integer which can be used to reference the file in read statements. If the file cannot be opened for any reason, -1 is returned.

12.4.2 create

DEFINITION:

```
    function create(&F:symbol):integer
```

DESCRIPTION: &F should be a legal name for a file on the system. The function creates an empty file by that name, opens it for writing, and returns an integer which can be used to reference the file in write statements. What happens if there is already a file by that name is system dependent. If the file cannot be created and opened for any reason, -1 is returned.

12.4.3 close

DEFINITION:

```
    function close(&FNUM:integer):logical
```

DESCRIPTION: &FNUM should be a number which was returned by open or create. This function closes the file. It returns 1b if the file was closed successfully, and 0b if it could not be closed.

12.4.4 status

DEFINITION:

```
function status(&FNUM:integer):integer
```

DESCRIPTION: &FNUM should be a number which was returned by open or create. This function returns the status of the indicated file. The status is encoded in a bit vector. The low order bit of the status is 1 if the file is open for input. The next higher bit is 1 if the file is open for output. The next higher bit is 1 if the end of file has been reached. Some common statuses are: 0 if the file is not open at all, 1 if it is open for input and the end of file has not been reached, 5 (binary 101) if it is open for input and the end of file has been reached, and 2 (binary 010) it it is open for output. The end of file bit is never 1 if &FNUM refers to the standard input file; that is because the standard input file is assumed to be a terminal.

12.5 Manipulating Symbols

The following standard routines are provided to manipulate the character strings that comprise symbols. The first function, nthchar, is used to read characters from symbols. The other three routines are used in creating new symbols. New symbols are created by assembling a string of characters in an internal buffer and then adding the string to the symbol table. First the internal buffer is cleared by calling clearbuffer. Then the characters in the symbol are added to the buffer one at a time with addbuffer. Finally the symbol is put into the symbol table with installbuffer.

12.5.1 nthchar

DEFINITION:

```
function nthchar(&S:symbol, &C:integer):char
```

DESCRIPTION: &S can be any symbol. &C should be the number of a character in the string of characters comprising the symbol &S. The function returns character number &C from symbol &S. If &C is not the number of a character in &S, the null character (character 0 in the character set) is returned.

12.5.2 clearbuffer

DEFINITION:

```
procedure clearbuffer()
```

DESCRIPTION: The buffer in which symbols are accumulated is cleared.

12.5.3 addbuffer

DEFINITION:

```
    function addbuffer(&C:char):logical
```

DESCRIPTION: &C can be any character except the null character (character 0 in the character set). The character is added to the buffer if that is possible. It cannot be added to the buffer if the buffer is full or if &C is the null character. The function returns 1b if the character was added to the buffer, 0b if the character could not be added.

12.5.4 installbuffer

DEFINITION:

```
    function installbuffer():symbol
```

DESCRIPTION: The characters in the buffer are converted to a symbol and the symbol is returned.

12.6 Manipulating Structured Values

The following two functions are used to extract values from and store values into records and elements. They are used when the names of one or more of the fields are non-constants.

12.6.1 extract

DEFINITION: The argument list to this function has a non-standard form. The rest of the definition of the function is:

```
    simple function extract( . . . ):logical
```

This function takes three or more parameters. The first parameter can be any scalar type. The second parameter can be any non-scalar type. The rest of the parameters can be integers or symbols. The w-attribute of the first parameter is ref; the rest of the parameters have null w-attributes.

DESCRIPTION: The third and later arguments to extract are interpreted as array indexes (if they are integers) or field selectors (if they are symbols). They are applied to the value designated by the second argument to produce an l-value. After all the array indexes and field designators are applied, the value in the location that is finally designated is assigned to the first parameter. If the value is extracted and the assignment made successfully, extract returns 1b; if the value cannot be extracted for any reason, or if the assignment cannot be accomplished, 0b is returned. For example, if **&F** evaluates to `att`, then the assignment made by

```
    extract(&R, &OBJ, fld, &F, 4)
```

is similar to the assignment made by the following ordinary assignment statement.

```
&R = &OBJ.fld.att[4];
```

12.6.2 store

DEFINITION: The argument list to this function has a non-standard form. The rest of the definition of the function is:

```
     simple function store( . . . ):logical
```

This function takes three or more parameters. The first parameter can be any scalar type. The second parameter can be any non-scalar type. The rest of the parameters can be integers or symbols. The w-attribute of the second parameter is ref; the rest of the parameters have null w-attributes.

DESCRIPTION: The third and later arguments to this function are interpreted as array indexes (if they are integers) or field selectors (if they are symbols). They are applied to the value designated by the second argument to produce an l-value. After all the array indexes and field designators are applied, the value that the first argument produced is stored into the location that the l-value designates. If the l-value is computed and the assignment made successfully, store returns 1b; if the l-value cannot be computed for any reason, or if the assignment cannot be accomplished, it returns 0b. For example, if &F evaluates to `att`, then the assignment made by

```
     store(&R, &OBJ, fld, &F, 4)
```

is similar to the assignment made by the following ordinary assignment statement.

```
     &OBJ.fld.att[4] = &R;
```

12.7 Examining Working Memory

DEFINITION: The argument list to this function has a non-standard form. The rest of the definition of the function is:

```
     simple function timetag( . . . ):integer
```

This function takes one parameter. The parameter can be any element type. The w-attribute is null.

DESCRIPTION: This function returns the time tag of its argument.

References

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

2. Forgy, C. L. and McDermott, J. OPS, A Domain-Independent Production System. IJCAI-77, International Joint Conferences on Artificial Intelligence, 1977.

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

4. Forgy, C. L. OPS4 Users Manual. Department of Computer Science, Carnegie-Mellon University, 1979.

5. Forgy, C. L. OPS5 Users Manual. Department of Computer Science, Carnegie-Mellon University, 1981.

6. Jensen, K. and Wirth, N.. *Pascal User Manual and Report.* Springer-Verlag, 1974.

7. Kernighan, B. W. and Ritchie, D. M.. *The C Programming Language.* Prentice-Hall, 1978.

8. Newell, A. PSG Manual. Department of Computer Science, Carnegie-Mellon University, 1973.

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

10. Rychener, M. D. OPS3 Production System Language Tutorial and Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1980.