FORMULA ALGOL

Carnegie-Mellon University

Department of Computer Science

Computation Center

TM	CCU- 73
Author	Jay Earley
Technical Editor	
Release Approval	
Distribution	FORMULA ALGOL mailing list
Date	June 26
Replaces	of
Supplements	
Addends Page	of
Page	of 125 pages

ABSTRACT

Formula Algol is an extension of Algol 60 [1] incorporating formula manipulation and list processing.

This manual describes the use of the version of Formula Algol which is presently running at Carnegie-Mellon University.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

GENERAL DESCRIPTION OF FORMULA ALGOL

Formula Algol is an extension of Algol 60 [1] incorporating formula manipulation and list processing. The extension is accomplished by adding two new types of data structures: formulae and list structures, with an appropriate set of processes to manipulate them. The control structure of Algol 60 is inherited and also extended.

Algorithms may construct formulae and list structures at run time. Operations are available which alter or combine formulae and list structures, and which access arbitrary subexpressions. Formulae may be evaluated, substituting numerical or logical values for occurrences of variables contained within. They may be subjected to substitution processes causing the replacement of occurrences of variables by designated formulae. They may be subjected to transformations defined by sets of rules akin to Markov algorithms. Predicates are available to determine precisely the structure and composition of any formula or list structure, and mechanisms are provided to extract subexpressions of a formula, or sublists of a list, provided its structure is known. Numerical, logical, and formula values may be stored as elements in list structures, and retrieval mechanisms exist to select them for use as constituents in other processes. Description lists composed of attributes with associated value-lists may be attached to list structures, and processes exist for retrieving value lists and for creating, altering, and deleting attribute-value list pairs. Push down stacks of arbitrary depth are available for the storage of all types of data structures and generators are provided in the form of new types of FOR statements which assign to control variables the elements of a single list structure, or alternatively, of several list structures in parallel, for use in an arbitrary process. Finally, both arrays and procedures may be defined having formulae or list structures as values.

HISTORY AND IMPLEMENTATION

The Formula Algol language has been designed by Dr. Alan J. Perlis, Renato Iturriaga, and Thomas A. Standish. It was initiated at Carnegie-Mellon in January, 1963, and has undergone continual evolution and expansion since that date. In August, 1963 an interpretive version was running and was reported at the Working Conference on Mechanical Language Structures in Princeton, New Jersey. [2].

The version reported in this manual has been implemented as a compiler on the CDC G-21 computer at Carnegie-Mellon University by Renato Iturriaga, Thomas A. Standish, Rudolph A. Krutar and Jay Earley. A discussion of the compiling techniques used was presented at AFIPS 1966 [6]. For those interested in the details of the compiler, a more complete document exists [4].

ACKNOWLEDGMENTS

A large part of Chapters III and IV is based on "A Definition of Formula Algol" [7], and much of Chapter II is based on the Algol-20 manual [3]. Special thanks goes to Gail Jaffre, Dr. David C. Cooper, and the implementers of the language for their help in preparing the manual.

INTRODUCTION TO THE MANUAL

This manual describes the use of the version of Formula Algol which is presently running at Carnegie-Mellon University. It is called by writing 'FORML' in the language field of a job card.

It is assumed in this manual that the reader is familiar with Algol 60. Since Algol 60 is not described, the Revised Report is included in the appendix. Below is an introduction to Formula Algol programming, which is intended for those who are familiar with programming, but not with this language. Chapters II, III, and IV describe the mechanisms available in Formula Algol and how they are to be used. All the mechanisms described in this part of the manual may not be working perfectly at a given time. They are, however, a short range goal at which the Formula Algol maintenance group will aim. A list of current system bugs and problems, which should be updated frequently, is included in the appendix. Operations which are illegal and therefore produce errors are not mentioned specifically in the manual except in the list of errors. It should be assumed that any operation or instance of an operation which is not mentioned as being legal in the manual will produce an error.

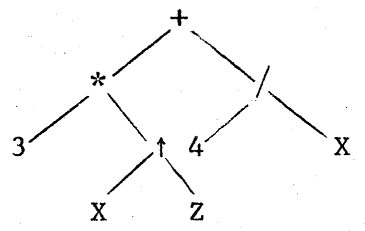
INTRODUCTION OF FORMULA ALGOL PROGRAMMING

This chapter is designed to introduce a programmer who is familiar with Algol to the mechanisms available in Formula Algol, and to give an idea how they may be used to do formula manipulation and list processing. No attempt has been made to be complete or rigorous. The individual mechanisms available are discussed more fully in Chapters III and IV.

Formula Manipulation

Suppose that we would like to write a procedure which takes as input a formula and differentiates it with respect to X. We first need some way of representing such a formula in our programming language. Algol is inconvenient for this because when an arithmetic expression is written in Algol, it is always to be evaluated, never to be kept around and examined. This forces the use of indirect representations.

For this purpose we have FORM variables. When a variable declared of type FORM is used in an expression, it indicates that a formula is to be constructed representing the expression. These formulae may be thought of as trees. Thus, 3*X↑Z + 4/X would cause the contruction of the following tree:



The normal Algol precedence of operators determines the form of the tree. If we assign the above expression to a FORM variable F. we can then access it later by referring to F.

We now have a way of inputting the expression to be differentiated. Next we need to be able to examine its structure.

For this, the language provides formula patterns. Thus we can write

F == ANY*ANY

The "==" is to be read "is an instance of." It tests whether a formula stored in F consists of any two subformulae connected by a multiplication sign. A formula pattern is a Boolean expression and can be used in an IF ... THEN statement.

Now that we can test for the form of a formula we want to be able to alter its form according to what we have found. To do this, we insert extractors into the pattern. An extractor is a formula variable followed by a colon. The pattern then looks like

```
F==LEFT: ANY*RIGHT: ANY
```

If the pattern matches, then the subformula which matched the left operand gets stored into LEFT and the subformula which matched the right operand gets stored into RIGHT. Thus if we executed this pattern on 3 * X, after it matched, LEFT would contain 3 and RIGHT would contain X.

We can now write one rule of our differentiation program

```
IF F==LEFT:  ANY * RIGHT: ANY THEN
DERV ← LEFT * DERV(RIGHT) + RIGHT * DERV(LEFT);
```

Assuming the DERV is the procedure we are writing to take the derivative, we are using it recursively here to find derivatives of expressions containing "*".

ANY is not the only word we can use in a pattern. We may use any declared type words to test for a subformula of certain type. An arithmetic or formula expression may also be used; these cause exact equality tests. Thus we may implement the "standard" derivative formula by

```
IF F== X↑N: REAL THEN
DERV ← N * X↑ (N-1)
```

However, suppose we want this transformation to apply only if N > 1. We can implement this by declaring a Boolean procedure to make this test.

```
BOOLEAN PROCEDURE  GR1 (I); VALUE I; FORM I;
    GR1 ← IF I == REAL THEN I > 1 ELSE FALSE;
```

Then we use the following pattern:

```
F==X↑N: OF (GR1)
```

and it will make the appropriate test for us.

Suppose in the derivative routine we would like to test whether the formula is a single unit (number, variable) or a binary combination (A + B). We may use the word ATOM, which yields true for number, FORM variables, etc.

```
        IF  F==ATOM  THEN DERV← IF  F==X THEN 1  ELSE 0.
```

We may search the formula to see if any of its subexpressions match a pattern instead of testing only the main expression. This is done by using ">>" in place of "==". The ">>" patterns are otherwise exactly the same.

Now, suppose that we have finished calculating the derivative of F and have stored it back into F. We may now want to substitute a number for X and evaluate the resulting expression. This is done by the EVAL operator:

```
        EVAL (X) F (3)
```

This substitutes 3 for all occurrences of X in F and calculates the result. If this substitution removes all formulae from F, then a number will result. However, if some are left, it will remain a formula, though it will probably be somewhat simplified. If we had wanted only to substitute 3 for X and not evaluate, we would have used "SUBS" in place of "EVAL". For a third possibility, we may want to replace X in the formula by whatever is the current value of X as a form variable. (Remember that the name X now appears in the formula, not its value.) This is done by REPLACE(F) which replaces all form variables in F by their current values, and then evaluates the resulting expression. Let's now suppose that instead of differentiating a formula we would like to make some simplifications in it. One thing we might like to do is apply the distributive law:

```
        IF  F==A: ANY * (B:  ANY + C:  ANY) THEN
        F ← A * B + A * C;
```

This works well, but this law is commutative, so we need a second rule for the case when A is to the right of B and C. We also need another law for subtraction. This expands our distributive law to four statements. We would like to contract them into one.

This is done by using operator classes. We will use one symbol to stand for plus or minus. For this we use a variable of type symbol, so that we can attach a description list to it (pg. 53). Let's call the symbol ADDOP. Then we execute

```
        ADDOP ← / [OPERATOR: +, -]
```

We can now write the pattern as

```
     F == A: ANY * (B: ANY |ADDOP| C: ANY)
```

and it will apply for both + and -. We can also use this mechanism to change F. If the above pattern matches, the operator which matched ADDOP will be stored as its value. Then we can write F ← A * B |<ADDOP>| A * C
to change F to the correct form.

Now we want to take care of the commutative instances of the distributive law. For this we declare an operator class for "*" and label it commutative:

```
        TIMES ← /[ OPERATOR: *] [COMM: TRUE]
```

Now, by using |TIMES| in place of "*", the test will also match an instance of

```
        (B: ANY  |ADDOP|  C: ANY)  *  A: ANY.
```

One final construction may be used to abbreviate some sequences of actions which might otherwise be quite long. Suppose we would like to write a routine to clear fractions. One transformation in it would be:

```
if  F==A: ANY - B: ANY / C: ANY THEN
 F← (A * C - B) / C;
```

We would need to write a sequence of these IF ... THEN statements plus proper circling back to the beginning to make sure that we have gotten all of the formula. This can be shortened by the use of productions. The production which corresponds to the above rule is:

```
     A:  any -B: any / C: any →(.A * .C - .B) / .C
```

The exact reason for the dots can be found by reading chapter 3 on formula manipulations. When this production is applied to a formula, it will have the same effect as the above IF ... THEN statement. However, we would like to apply a sequence of such productions in order to clear fractions, so we store a list of these productions by a list assignment statement (pg. 52). If the left formulae are L_i and the right are R_i, this will look like:

CLEAR ← [$L_1 \rightarrow R_1$, $L_2 \rightarrow R_2$, ..., $L_n \rightarrow R_n$];

This now gives CLEAR the semantics represented by these productions. Then if we apply this schema to a formula in F by the expression

F ↓ CLEAR,

F will be treated in the following way:

L_1 will be tested against F and then each of its subformulae, then L_2 will be, and so on. When a match is found, the corresponding transformation R_i is applied and control returns to L_1 again.

The complete schema for clearing fractions is on page 46.

List Processing

Suppose we want to write a program to play Solitaire. We can do this in the list processing part of Formula ALGOL. First we need to represent the cards of the deck. Let's make each card a variable of type SYMBOL, so the ace of spades is SPADEA and the 3 of clubs is CLUB3. We can represent the deck as a list which is the contents of the symbol DECK. So to initiate the deck we execute the assignment statement

DECK ← [SPADEA, SPADE2, SPADE3, ...];

where we string out all 52 cards.

Now we need to be able to deal out the cards into the seven solitaire piles. Let's make these a symbol array called PILE:

SYMBOL ARRAY PILE [1: 7].

In order to deal we need to be able to select cards from one list (DECK) and insert them into another. To select an element from a list we use a selector which refers to the position of the element in the list by number. Since we want the top element of the deck we use the expression

```
FIRST OF DECK
```

Since we will be putting cards on the top of the piles we use the statement

```
INSERT FIRST OF DECK BEFORE FIRST OF PILE[I];
```

We need to show that the card has been removed from the deck. This is done by

```
DELETE FIRST OF DECK.
```

Now this should do the dealing:

```
FOR J←1 STEP 1 UNTIL 7 DO
FOR I←J STEP 1 UNTIL 7 DO
BEGIN
INSERT FIRST OF DECK BEFORE FIRST OF PILE[I]:
DELETE FIRST OF DECK;
END;
```

We would like to be able to compare the suits and numbers of various cards to tell whether they can be placed on each other. For this our symbol names are inadequate. We need to be able to associate properties of the cards with them. This is done by using description lists. We should assign a description list to each card with a statement such as:

```
SPADE4←/[SUIT:  SPADES] [DENOM:  4];
```

In this statement, SUIT and DENOM are attributes, and SPADES and 4 are their respective values. However, we have to test mainly the color of the cards for solitaire, so let's add that attribute to our description list, too:

```
THE COLOR OF SPADE4 IS BLACK;
```

Note that COLOR, BLACK, SUIT, etc., are all symbol variables.

We may retrieve the value of an attribute by a statement such as:

```
THE SUIT OF SPADE4
```

or

```
SUIT(SPADE4)
```

Using this we could write a routine to add the color attribute to all the cards. For each card we would write

```
                IF SUIT(CARD) = SPADES ∨ SUIT(CARD) = CLUBS
                THEN THE COLOR OF CARD IS BLACK
                ELSE THE COLOR OF CARD IS RED;
```

To iterate through the deck we use a new type of for-statement which iterates on the elements of a list. Using this plus a standard Algol abbreviation for the IF ... THEN statement we have

```
    FOR CARD ← ELEMENTS OF DECK DO
    THE COLOR OF CARD IS
    IF SUIT (CARD) = SPADES ∨ SUIT (CARD) = CLUBS
    THEN BLACK ELSE RED;
```

There is an alternative to this course of action. Instead of storing the attribute color with each card, we can test each card to see if it is a spade or club each time in the program that we need to know its color. However, we don't want to have to write:

```
                IF SUIT (CARD) = SPADES ∨ SUIT (CARD) = CLUBS THEN
```

every time we want to test a card.

Therefore we use a class test:

```
LET (|BLACK|) = [X |  SUIT (X) = SPADES ∨ SUIT (X) = CLUBS];
```

This establishes a test for the class of black cards. We can now write

```
                IF CARD == (|BLACK|) THEN
```

and the test will be performed for us.

We can now write a routine to test whether one card can be placed on another or not.

Let's use color as an attribute and store JACK, QUEEN, and KING as 11, 12, 13. Since we can store numbers directly as values, or in fact as elements of a list, we can do just an arithmetic check on the value of DENOM in our routine. The following routine tests whether C1 can be placed on C2.

```
BOOLEAN PROCEDURE  PLACEON(C1, C2); VALUE C1, C2; SYMBOL C1, C2;
PLACEON ← COLOR(C1) ≠ COLOR(C2)
∧ DENOM(C1) + 1 = DENOM(C2);
```

Now let's switch from Solitaire to natural language processing. Assume we have the words of a paragraph stored in a list called PARA. We want to search it for the words "THERE ARE" followed by a number and then a plural noun, i.e., "THERE ARE 20 BUILDINGS." We then want to put the number as the value of NUMBER on the description list of the noun. We have a list of the plural nouns stored in NOUN.

To do this we need some new constructions:

(1) COUNT(L) produces an integer value corresponding to the number of elements in list L.

(2) AMONG(X, L) is TRUE if X is an element of list L.

(3) As with formula patterns, we may test to see if an element is of a particular type using "==".

The routine is

```
FOR I← 1 STEP 1 UNTIL COUNT(PARA) -3 DO
IF I TH OF PARA = THERE ∧
     (I + 1) TH OF PARA = ARE ∧
     (I + 2) TH OF PARA == INTEGER ∧
     AMONG ((I + 3) TH OF PARA, NOUN) THEN
THE NUMBER OF (I + 3) TH OF PARA IS (I + 2) TH OF PARA;
```

This is a lot of writing, so we would like to be able to use some of the mechanisms of COMIT [5] to make this test. Let's first construct a class name for nouns.

```
LET (|NOUN|) = [X| AMONG (X, NOUN)]
```

We can now use a list pattern to make the test

```
        IF PARA == [$, THERE, ARE, INTEGER, (|NOUN|), $] THEN
```

$ stands for an arbitrary number of elements. This pattern is tested against the list PARA for any match. After the match, however, we want to be able to perform the description list store. For this we need to be able to extract elements of PARA according to the part of the patterns they match. This is done by writing a symbol variable and a colon in front of an element of the pattern. Then if the pattern matches, the element that matched the pattern element is stored into the extractor variable.

The routine now becomes:

```
IF PARA == [$, THERE, ARE, N: INTEGER, OBJECTS: (|NOUN|), $]
     THEN THE NUMBER OF OBJECTS IS N;
```

CHAPTER II

NUMERIC PROCESSING

Although Formula Algol is an extension to Algol 60, there are certain restrictions on this reference language which have been made due to character set limitations and implementation. There are also some added features of Formula Algol over Algol 60 aside from the formula and list processing features. These are explained in this chapter.

SYMBOLS

Formula Algol accepts all of the special symbols of ALGOL-60 except for those shown in the following table:

ALGOL-60	
⊃ ("implies")	Use "→"
≡ ("is equivalent")	Use "="
× (multiplication)	Use "*"
÷	Not available, but entier may be used with "/" with the same effect.
≤	Use "¬>"
≧	Use "¬<"
:=	Use "←"
(string quotes)	Not available

DECIMAL CONSTANTS

A number, N, in a Formula Algol program must be zero (which may be punched with or without a decimal point) or else its absolute value N must satisfy:

$$1.275_{10}-57 \leqq N \leqq 3.450_{10}+69$$

Because of the nature of the G-21 computer, the distinction between real and integer numbers is unimportant. The programmer may write an integer-valued constant with or without a decimal point (e.g., "34", "34.", or "34.0") without changing the type of arithmetic performed on the constant.

Numbers are represented in the G-21 in "floating point" form with a maximum of 42 binary digits of mantissa, corresponding to approximately 12 decimal digits of precision. If more than 12 digits are written, the extra (least significant) digits will be ignored. (The number is rounded at the 14th octal digit.)

The last character of a real number <u>may</u> be a decimal point; thus, the number "6." is legal. Note: In Formula Algol "." is sometimes used as an operator. In these cases it should not be placed adjacent to a numerical constant so that these uses are not confused with its use as a decimal point.

OCTAL CONSTANTS

An octal (base 8) constant may be used in any context in Formula Algol where a decimal number is allowed; i.e., as a primary in any arithmetic or logic expression. Octal constants have the following syntax:

syntax:

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

<octalian> ::= <octal digit> | <octalian><octal digit>

<signed octalian> ::= <octalian> | +<octalian> | -<octalian>

<left-justified octal constant> ::= 8L<octalian>

<right-justified octal constant> ::= 8R<octalian>

<floctalian> ::= <octalian> | <octalian>.<octalian> | <octalian>. | .<octalian>

<power of 8> ::= $_{10}$<signed octalian>

<floating octal constant> ::= 8F<floctalian> | 8F<power of 8>
8F<floctalian><power of 8>

<logical octal constant> ::= <left-justified octal constant> |
<right-justified octal constant>

<octal constant> ::= <floating octal constant> | <logic octal constant>

Despite this syntax, the translator does not treat the digits 8 and 9 in octal constants as erroneous but will intepret them as $10)_8$ and $11)_8$, respectively. Thus 8R495 will be interpreted as 8R515.

Local octal constants (8L and 8R) are considered to be of type <u>LOGIC</u> and so are always accessed in logic mode. Floating octal constants (8F) are considered to be of arithmetic type, and are always accessed in arithmetic mode.

The character-pairs 8L, 8R and 8F are treated by the translator as single entities and must be punched in adjacent columns of the same card, without intervening blanks.

The value of a floating octal constant is determined by concatenating the octalian as an octal number and multiplying it by the appropriate power of 8, treating the number which follows the $_{10}$ as an octal integer. For example:

$8F_{10}10 = 8\uparrow8$

$8F11_{10}\text{-}5 = 9*8\uparrow\text{-}5$

The value of a left (right) justified octal constant is determined by prefixing (suffixing) to the octalian enough zeros to give eleven octal digits. This number is then concatenated and stored as a 32-bit logic word. Since eleven octal digits require thirty-three bits for representation, the leftmost bit of the leftmost octal digit is lost. Thus, 8L4=0 and 8L7=8L3.

IDENTIFIERS

Only upper case (capital) letters are available in Formula Algol. Neither spaces nor any operator may appear within an identifier (including "."). All identifiers must be separated from adjacent identifiers by at least one space to prevent the two from being interpreted as a single identifier.

Certain identifiers have special meanings in Formula Algol and are therefore reserved. The programmer may not use these identifiers for any purpose other than that of their reserved meanings. The reserved words in Formula Algol are

ABS	CONT	EXP	LAST	SIGN
AFTER	COPY	FALSE	LET	SIN
ALL	COS	FIRST	LIM	SQRT
ALSO	COUNT	FOR	LN	ST
ALTER	CREATE	FORM	LOGIC	STEP
AMONG	DELETE	GC	ND	STRING
AND	DERV	GO	NIL	SUBLIST
ANY	DL	GOTO	NOT	SUBS
ARCTAN	DO	HALF	OF	SWITCH
ARRAY	ELEMENTS	HAS	OPERATOR	SYMBOL
ATOM	ELSE	IF	OWN	TEXT
ATTRIBUTES	EMPTY	IN	PARALLEL	TH
BEFORE	END	INDEX	PRINT	THE
BEGIN	ENTIER	INFI	PROCEDURE	THEN
BETWEEN	ERADL	INSERT	RD	TO
BOOLEAN	EVAL	INTEGER	REAL	TRUE
CELLS	EX3	IS	RECU	UNTIL
COMM	EX4	JUMP	REDUCE	VALUE
COMMENT	EX5	LABEL	REPLACE	WHILE

VARIABLES

Formula Algol allows both simple and subscripted variables of type HALF, LOGIC, FORM and SYMBOL as well as REAL, INTEGER, and BOOLEAN.

REAL variables are stored in the G-21 with a precision of 42 binary digits, requiring two successive memory cells per variable. HALF variables are stored with a precision of only 21 binary digits (about 6 significant decimal digits) and occupy only a single location, but otherwise act as REAL variables. Therefore, the programmer may use HALF variables to gain memory space at the expense of precision.

The value of a REAL or HALF variable must either be zero or lie within the range given below:

REAL:	$1.275_{10}-57 \leqq \text{abs}(R) \leqq 3.450_{10}+69$
HALF:	$1.275_{10}-57 \leqq \text{abs}(H) \leqq 1.645_{10}+63$

INTEGER variables will always take on integer values in the range

$$-2097152 < I < 2097152 \ (=2^{21}).$$

LOGIC variables are always positive. If used as strings, they are four or less characters in length, ,and if used as numeric quantitites they are restricted to

$$0 \leqq L < 42949\ 67296 \quad (=2^{32}).$$

The values of BOOLEAN variables must be either TRUE or FALSE.

The G-21 replaces by zero any non-zero arithmetic result which is smaller than $1.275_{10}-57$ in magnitude; this situation is called an underflow. An intermediate arithmetic result which is greater than $3.450_{10}+69$, the largest number representable in the G-21, is called an overflow, and causes an error to be recorded. Executing an assignment to a half variable of intermediate results which exceed the bound of the variable causes an overflow. On the other hand, assignments to integer variables are truncated modulo their upper bound, and assignments to logic variables are truncated modulo their upper bound and made positive. In these two cases, no error occurs.

LOGIC EXPRESSIONS

In addition to arithmetic, Boolean, and designational expressions, Formula Algol syntax includes "logic expressions" which perform bit-by-bit logic operations on 32-bit G-21 logic words. A logic expression may include any of the following operands:

1. Logic constant: octal constant or string constant
2. Variable, simple or subscripted, of type <u>LOGIC</u>
3. Function designator of type <u>LOGIC</u>
4. Boolean primary (and, therefore, any Boolean expression in parentheses)
5. Arithmetic primary (and, therefore, any arithmetic expression in parentheses)

A Boolean primary used as a logic operand is interpreted as one of the two 32-bit logic words:

8R 37777777777 = 32 one bits for <u>TRUE</u>, or

8R' 0 = 32 zero bits for <u>FALSE</u>.

Each kind of logical operand (except number 5 above, arithmetic primary) will always be fetched from memory with a "logic access", rather than a "numeric access"; for example, a CAL command will be used to fetch a logic variable into the accumulator. When a logic variable or function designator forms the left-part of an assignment statement, then an STL command will perform the assignment. Therefore, an assignment statement of the form

<logic variable> ←<arithmetic expression>

will truncate the absolute value of the expression modulo 2^{32}. An STL command is also used for any temporary store of a logical subexpression (except an arithmetic primary) within a complete logical expression.

Any of the following three logical operators may appear in a logic expression:

$\neg$	(<u>complement</u> <u>logic</u>:	unary)
$\wedge$	(<u>extract</u> <u>logic</u>:	binary)
$\vee$	(<u>unite</u> <u>logic</u>:	binary)

Each of these operators performs the same operation simultaneously and independently in each of the 32-bit positions of its operand(s). If a bit = 1 represents the Boolean value <u>true</u> and a bit = 0 represents <u>false</u>, then the logic operators ¬, ∧, and ∨ can be considered to perform the Boolean operations ¬, ∧, and ∨ respectively, in each bit position.

The operators +, -, *, and / may also appear in a logic expression. Each of these operates in the usual way, considering its logical operands (except for arithmetic primaries) as 32-bit integers.

syntax:

```
<logic constant> ::= <string constant> | <logic octal constant>
<logic primary> ::= <logic constant> | <logic variable> | <logic function> |
                    <Boolean primary> | (<logic expression>) |
                    <arithmetic primary>
<logic factor> ::= <logic primary> | ¬ <logic primary>
<logic term> ::= <logic factor> | <logic term> ∧ <logic factor>
<simple logic expression> ::= <logic term> | <simple logic expression> ∨
                                              <logic term>
<logic expression> ::= <simple logic expression> | <if clause>
                    <simple logic expression> ELSE <logic expression>
```

THE PRECEDENCE OF OPERATORS AND RELATIONS IN FORMULA ALGOL

↑	(done first)
- +	(unary operators)
/ *	
- +	(binary operators)
¬	
∧	
∨	
→	(done last)

In cases of equal precedence, association to the left is used.

STANDARD FUNCTIONS

Formula Algol contains all the recommended standard functions of Algol 60. These are

<u>ABS</u>	<u>SIN</u>	<u>LN</u>
<u>SIGN</u>	<u>COS</u>	<u>EXP</u>
<u>SQRT</u>	<u>ARCTAN</u>	<u>ENTIER</u>

ASSIGNMENT STATEMENTS

In Formula Algol, "←" must be used instead of ":=". It has the same meaning except when storing a non-integer into an integer variable. In this case, the non-integer is truncated, not rounded.

In multiple assignment statements, the "left-part" variables need not all be of the same type. In fact, an assignment statement in Formula Algol may be treated as an expression whose value is the value which is assigned in the assignment statement. Thus

```
I ← 3 * K + (J ← 7-K) / 2;
```

is a legal statement. To insure that "←" is given the proper precedence, the assignment statement should be enclosed in parentheses.

CONDITIONAL STATEMENTS

In Formula Algol, unlike Algol 60, the construction

```
IF ... THEN
FOR ... DO <unconditioned statement>
ELSE <statement>
```

is legal and will be recognized correctly.

LABELS AND GO TO STATEMENTS

Only identifiers may be used as labels in Formula Algol;integer labels are not permitted.

In Formula Algol,

```
GO TO Label
GOTO Label
```

are equivalent and permissible.

FOR STATEMENTS

The value of the controlled variable is not undefined upon normal exit from a Formula Algol FOR statement. It is, in general, just what would be obtained if the equivalent basic programs (section 4.6.4 of the Algol 60 report) were substituted for the FOR statement. Thus, upon exit from an UNTIL or WHILE form of FOR list element, the FOR variable has the first value for which the final test failed.

Another form of FOR list element is permitted in Formula Algol,

```
FOR V ← E1 STEP E2 WHILE B DO S;
```

where E_1 and E_2 are arithmetic expressions, B is a Boolean expression, and S is any statement. This is equivalent to the simple program:

```
       V ← E1 ;
LOOP:  IF B THEN
       BEGIN
         S   ;
         V ← V + E2;  GO TO LOOP
       END  ;
```

ARRAYS

Formula Algol arrays may be of type INTEGER, REAL, BOOLEAN, HALF, LOGIC, FORM, or SYMBOL.

A non-integer value of a subscript expression in Formula Algol is not rounded, only truncated. This may lead to hard-to-detect errors. For example, suppose that the result computed for a subscript expression is 3.9999... instead of 4, because of round-off error, this value will be truncated to 3, referring to the wrong element of the array.

Run-time tests are made with each array access so that an access which is out-of-bounds will produce an error. OWN arrays may not be used in Formula Algol.

PROCEDURES AND BLOCK STRUCTURE

All formal parameters in a Formula Algol procedure declaration must be specified. The following is a list of current restrictions on the use of procedures and blocks.

1. Switches and strings may not be passed as parameters.
2. Arrays may not be called by value.

CHAPTER III

FORMULA MANIPULATION

FORM VARIABLES

Variables may be declared of type FORM indicating that their values are to be formulae. With each FORM variable there is associated a data item called an atomic formula, which may form part of a formula expression. When a FORM variable F is declared, its value is initialized to be the atomic formula of F. Also, a description list is associated with F, into which attributes and values may be entered and retrieved in exactly the same way as with SYMBOL variables (pg. 53) except that a description list may be associated only with a FORM variable, not with a sub formula.

FORM ARRAYS

Arrays may be declared of type FORM in which case their elements may be formulae. These are accessed in the same way as other arrays. Unlike simple FORM variables, array elements are not initialized, and therefore should not be accessed before they have been stored into.

FORMULA EXPRESSIONS

Syntax:

```
        <formula expression> ::= <arithmetic expression> |
        <Boolean expression> | <an arithmetic expression (Boolean expression)
                in which some of the primaries (Boolean primaries) have been
                replaced by formula primaries and in which some operators have
                been prefixed with a dot> † | <assignment formula> |
        <formula expression> "|" [<identifier>]
        "|" <formula expression>
<formula primary> ::= <array formula> | <procedure formula>|
        <transformed formula> | <evaluated formula> | .<identifier> |
        <conditional formula> | ( <formula expression> )
<array formula> ::= <array identifier> . [ <subscript list> ]
<procedure formula> ::= <procedure identifier> . <actual parameter part>
<conditional formula ::= . IF <formula expression> THEN
        <formula expression> ELSE <formula expression>
<assignment formula> ::= <variable> . ← <formula expression>
```

Semantics:

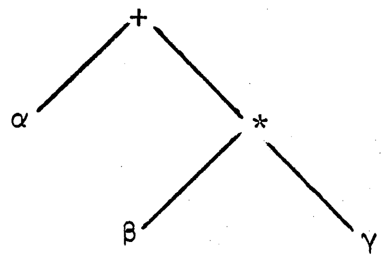
A formula is a piece of Algol text which is to be stored for testing, manipulation, and possibly execution later on. An Algol expression is to be treated as a formula when either of its operands is a form variable or is already a formula. A dot preceding a variable is used to indicate the atomic formula of that variable.

The process by which the value of a formula expression is obtained will be explained by means of a recursively defined function called VAL. This function

† This is a short description of what could be a formal syntactic statement.

does not appear explicitly in the syntax of the source language; rather, it is executed implicitly at run time whenever the value of an expression is needed. In subsequent definitions quoted strings represent formulae. Such formulae are represented within the machine as trees, with operators at their nodes, atomic formulae at their leaves, and each branch representing a subformula. Thus

' $\alpha + \beta * \gamma$ ' is represented



The normal precedence of Algol operators is used to determine how the tree will be constructed. In addition, the "|[ ]|" construction has precedence just above that of "→" (pg. 47).

These formulae may be assigned to <u>FORM</u> variables, which may then be evaluated or used in other formula expressions. In fact, any type of Formula Algol expression may be assigned to a <u>FORM</u> variable except one of type <u>SYMBOL</u>.

Formula Algol is a strict extension of Algol 60 with regard to values and types. Exactly as in Algol 60 each value has an associated type. In the explanation of the function VAL below, the association of a type with a value is given explicitly by an ordered pair of the form (TYPE, VALUE).

FORMAL DEFINITION OF VAL (E):

1. E is a constant which is either a <number> or a <logical value>.

   TYPE (E) = <u>INTEGER</u> if VALUE(E) is an integer, <u>REAL</u> if VALUE(E) is a floating point number, and <u>BOOLEAN</u> if E is a <logical value>.

   VALUE (E) = the conventional value of a number or a logical value (identical to that given by the Algol Report).

2. E is $.\alpha$, where α is an <identifier> declared of type FORM.

   TYPE (E) = FORM

   VALUE (E) = the atomic formula of α.

3. E is a variable - simple or subscripted.

   TYPE (E) = the type of the most recently assigned value of E, taken as a constant.

   VALUE (E) = the most recently assigned value of E.

4. E is a function designator, say $P(X_1,\dots,X_n)$

   TYPE (E) = the declared type of P.

   VALUE (E) = the value produced by executing the procedure P as defined in the Algol report.

5. E is a binary expression A ω B where A and B are expressions and

   $\omega ::= + \mid - \mid * \mid / \mid \uparrow \mid < \mid \neg < \mid > \mid \neg > \mid = \mid \neq \mid \vee \mid \wedge \mid \rightarrow$

   TYPE (E) is defined by the following table

TYPE (A) \ TYPE (B)	REAL	INTEGER	BOOLEAN	FORM
REAL	T1	T1	error	T4
INTEGER	T1	T2	error	T4
BOOLEAN	error	error	T3	T5
FORM	T4	T4	T5	FORM

where:

$$T1 = \begin{cases} \underline{\text{REAL}} & \text{if } \omega \text{ is a numeric operator} \\ \underline{\text{BOOLEAN}} & \text{if } \omega \text{ is a numeric operator} \\ \text{error} & \text{otherwise} \end{cases}$$

$$T2 = \begin{cases} \underline{\text{INTEGER}} & \text{if } \omega \text{ is a numeric operator other than } / \\ \underline{\text{REAL}} & \text{if } \omega \text{ is } / \\ \underline{\text{BOOLEAN}} & \text{if } \omega \text{ is a relational operator} \\ \text{error} & \text{otherwise} \end{cases}$$

$$T3 = \begin{cases} \underline{\text{BOOLEAN}} & \text{if } \omega \text{ is a logical connective} \\ \text{error} & \text{otherwise} \end{cases}$$

$$T4 = \begin{cases} \underline{\text{FORM}} & \text{if } \omega \text{ is either a numeric or relational operator} \\ \text{error} & \text{otherwise} \end{cases}$$

$$T5 = \begin{cases} \underline{\text{FORM}} & \text{if } \omega \text{ is a logical connective} \\ \text{error} & \text{otherwise} \end{cases}$$

if TYPE (E) = REAL, INTEGER or BOOLEAN then

VALUE (E) = the number or logical value obtained by carrying out the operation ω with arguments VALUE (A) and VALUE (B).

If TYPE (E) = FORM then VALUE (E) = '$\alpha\ \omega\ \beta$' where α is VALUE (A) and β is VALUE (B).

6. E is A| [T]|B where T is an operator class name.

   TYPE E = FORM

   VALUE (E) = '$\alpha\ \omega\ \beta$' where ω = the operator most recently assigned to T by a pattern or assignment statement (pg. 37), and α = VALUE (A) and β = VALUE (B).

7. E is a unary expression ω A where A is an expression and ω::=¬|+|- or E is of the form ω (A) where ω::=

   SIN|COS|EXP|LN|SQRT|ARCTAN|SIGN|ENTIER|ABS

   TYPE (E) is defined by the following table:

TYPE (A) \ ω	SIN,COS,EXP LN,SQRT	SIGN ENTIER	ABS ±	¬
REAL	REAL	INTEGER	REAL	error
INTEGER	REAL	INTEGER	INTEGER	error
BOOLEAN	error	error	error	BOOLEAN
FORM	FORM	FORM	FORM	FORM

If TYPE (E) = REAL, INTEGER or BOOLEAN then VALUE (E) = the number or logical value obtained by carrying out the operation ω with argument VALUE (A). If TYPE (E) = FORM then VALUE (E) = the expression 'ω α' where α = VALUE (A).

Examples

Suppose that at a certain point in some program R and G have been declared of type FORM, X and Y have been declared of type REAL, X has been assigned the value 3.2, Y has been assigned the value 2, F has been assigned the value 'G/5', and G has as its value the atomic formula of G. Consider the following sequence of assignment statements:

(a) X ← (X + Y) ↑ 2;

(b) F ← 3 * SIN(G) + (F + X) ↑ Y;

(c) F ← SQRT(F) ;

In statement (a) all variables are numeric. Thus the arithmetic expression (X + Y) ↑ 2 is evaluated numerically using the current values of X and Y and the result (27.04) is stored as the value of X. In statement (b) the value of F becomes the formula expression '3 * SIN(G) + (G/5 + 27.04) ↑ 2'. Finally, statement (c) replaces the value of F by the formula

'SQRT (3 * SIN(G) + (G/5 + 27.04) ↑ 2)'.

All arithmetic operators are treated as binary operators (even those which are associative and commutative) with association to the left. This is illustrated by the following examples:

(d) F + (X + Y) produces 'G/5 + 5.2'

but (e) F + X + Y is equivalent to 'G/5 + 3.2 + 2'

8. E is a conditional formula

.IF B THEN A ELSE C, where A, B, and C are expressions and B is of type FORM or BOOLEAN.

TYPE (E) = FORM

VALUE (E) = 'IF β THEN α ELSE γ'

where B = VALUE (B), α = VALUE (A) and γ = VALUE (C)

9. E is a procedure formula

E = $\alpha.(X_1, X_2, \ldots, X_n)$ where α is the name of a declared procedure, and $X_1, X_2, \ldots, X_n$ are expressions.

TYPE (E) = FORM

VALUE (E) = '$\alpha(N_1, N_2, \ldots N_n)$' where N_i = VALUE (X_i).

Note: The formal parameters of any procedure which is used as a procedure formula must all be of TYPE FORM.

10. E is an array formula

A.$[X_1, X_2, \ldots, X_n]$ where A is the name of a declared array, and $X_1, X_2, \ldots, X_n$ are formula expressions.

TYPE (E) = FORM

VALUE (E) = 'A$[N_1, N_2, \ldots, N_n]$' where N_i = VALUE (X_i)

An important application of array formulae is the generation of names dynamically at run-time. Upon entrance to a block containing the declaration FORM ARRAY A[1:N], N array elements are created whose names may be used in the construction of formulae even without any values having been stored into them. Thus the name of the fifth of these is "A.[5]". Later, values may be assigned to these elements and the formulae may then be evaluated, if desired.

1. E is an assignment formula

   α .← B where α is a variable and B is an expression

   TYPE (E) = FORM

   VALUE (E) = '$\alpha \leftarrow \beta$' where β = VALUE (B)

   Evaluated and transformed formulae will be explained in succeeding sections.

EVALUATION OF FORMULAE

Syntax:

<evaluated formula> ::= EVAL <variable> |

EVAL (<substitution list>) <formula expression> (<substitution list>) |

SUBS (<substitution list>) <formula expression> (<substitution list>) |

REPLACE (<formula expression>)

<substitution list> ::= <formula expression list> | [<variable>]

<formula expression list> ::= <formula expression> | <formula expression list>, <formula expression>

Semantics:

At some point in the execution of a program, we may wish to carry out completely or partially the computation represented by a formula. To do this, we could substitute values for all occurrences of some of the variables appearing in a formula, and combine these values according to the computation expressed by the formula. In order to accomplish the above we have the EVAL operator. This is in some sense the inverse of the "." operator. The dot postpones the action of certain Algol expressions by making them formulae, while EVAL causes the evaluation and/or execution of formulae.

If we have a formula consisting of names of formula variables joined by arithmetic operators, then if we assign each of the formula variables a numerical value, the result of the evaluation of the formula will be a number. Analogously, substitution of Boolean values for formula variables in a Boolean formula produces a Boolean value.

On the other hand, we need not substitute arithmetic or Boolean values for formula variables, but rather, we can substitute other formulae. Thus, in this case, evaluation of the formula, instead of producing a single value, creates a new formula. Hence, EVAL may be used to construct formulae.

A third use of EVAL is that of producing trivial simplifications in a formula without altering its value and without substitution. This is done according to the following table:

Simplifications of EVAL

$A \uparrow 0 \rightarrow 1$	$A * 0 \rightarrow 0$	
$A \uparrow 1 \rightarrow A$	$A * 1 \rightarrow A$	commutative
$A \uparrow -1 \rightarrow 1/A$	$A * -1 \rightarrow -A$	
$A \uparrow -n \rightarrow 1/A \uparrow n$	$A * -n \rightarrow -(A * n)$	
$A / 1 \rightarrow A$	$A + 0 \rightarrow A$	
$A /(-1) \rightarrow -A$	$A + (-n) \rightarrow A - n$	
$A /(-n) \rightarrow -(A/n)$	$0 + A \rightarrow A$	
$0 / A \rightarrow 0$	$(-n) + A \rightarrow A - n$	
$(-n) / A \rightarrow -(n/A)$	$A - 0 \rightarrow A$	
	$A - (-n) \rightarrow A + n$	
	$0 - A \rightarrow -A$	
	$(-n) - A \rightarrow -(n + A)$	

$$\left.\begin{array}{l} X \vee \text{true} \rightarrow \text{true} \\ X \wedge \text{true} \rightarrow X \\ X \vee \text{false} \rightarrow X \\ X \wedge \text{false} \rightarrow \text{false} \end{array}\right\} \text{commutative}$$

Whenever an expression contains two numeric (Boolean) arguments joined by an arithmetic (Boolean) operator, it is replaced by its value. Similarly, the truth values of relations are obtained if both arguments are numeric.

A final use of EVAL is to execute the Algol code which is represented by an array, procedure, conditional, or assignment formula.

These uses of EVAL are usually combined; thus evaluation of a formula may produce partial expansion and some trivial simplification.

In order to define the EVAL operator we will first define the operator SUBS, which performs part of the operation of EVAL and may also be evoked in the source language.

Consider a statement of the form

$D \leftarrow \underline{\text{SUBS}}\ (X_1, X_2, \ldots, X_m)\ F\ (Y_1, Y_2, \ldots, Y_n)$ (1)

where $N \geq 1$ and $m \geq 1$ (normally $n = m$).

If F is a formula expression then

(a) If TYPE (F) is numeric or BOOLEAN or if VALUE (F) is a number or Boolean constant then the effect of (1) is precisely that of $D \leftarrow F$.

(b) If TYPE (F) = FORM and VALUE (F) is a formula, then D will have the value obtained by substituting VALUE (Y_i) for each occurrence of VALUE (X_i) in a copy of VALUE (F) for all $i \leq \min(m,n)$ for which VALUE (X_i) is an atomic formula. If $m \neq n$, any extras on either side are ignored.

Now we define the EVAL operator:

Consider a statement of the following form:

$D \leftarrow \underline{\text{EVAL}}\ (X_1, X_2, \ldots, X_m)\ F\ (Y_1, Y_2, \ldots, Y_n)$

First the rules for SUBS are applied. Then the formula is evaluated by a recursive process which starts at the top of the tree and is applied successively to each subformula as follows:

(1) If the formula is a constant or atomic formula, it is left unchanged.

(2) If the formula is a binary formula, its operands are evaluated from right to left. If they reduce to numbers or logical values, then the operation indicated by the operator is carried out and the result replaces the formula. Also, if any of the simplifications listed previously applies, it is carried out. A similar process is carried out for unary formulae.

(3) If it is a procedure formula, the parameters are evaluated from left to right and then the procedure call is executed and its value replaces the formula. Note: Since the procedure call is made regardless of collapsing of formulae, all its arguments must be of the right type to correspond to their actual parameters (e.g., a partially collapsed formula can't be passed as a real).

(4) If it is an assignment formula the expression to be assigned is evaluated, the assignment statement is executed, and the formula is replaced by the assigned value.

(5) If it is an array formula, the subscript expressions are evaluated from left to right and if all reduce to numbers, the array access is carried out and its value replaces the formula.

(6) If it is a conditional formula, the IF formula is evaluated and if it reduces to a logical value, then the corresponding THEN or ELSE formula is evaluated and replaces the conditional formula.

In the above cases if the operands of the formula do not reduce properly, the formula is left as simplified as the above transformations provide.

EVAL and SUBS may also use [T] in place of either list of formulae where T must be a symbol which has been previously assigned a list of formula. This list is then used as has been explained in the operation of EVAL.

The function REPLACE:

The function designator REPLACE (F) where F is a formula expression produces a formula which is obtained from F by replacing every atomic variable by the current value of its associated FORM variable and evaluating the result as in EVAL. The atomic variables used in the formula F must be declared either locally or globally to the block in which REPLACE (F) is executed.

Examples: All variables are of type FORM.

```
Initially          F ← X + Y * Z;
                   Y ← 1 ; Z ← 2;
Executing          SUBS (Y, Z) F (3, 4)
however, will produce 'X + 12'
and                REPLACE (F)
will produce 'X + 2'
Let F be 'IF B THEN P(X) ELSE A[Y+Z]'
Executing          EVAL (B) F (TRUE)
```

will yield 'R' where R is the result of calling procedure P with the Formula X as a parameter

EVAL (B, Z) F (FALSE, 2)

will yield 'A[Y + 2]'. Since the subscript did not reduce to an integer, the access was not carried out.

FORMULA PATTERNS

Syntax:

```
<formula pattern> ::= <formula expression> == <formula pattern structure> |
     <formula expression> >> <formula pattern structure> |
     <extractor> <formula expression> >> <extractor> <formula pattern structure>
<extractor> ::= <variable> :
<formula pattern structure> ::= <a formula expression in which some of the
     primaries may have been replaced by pattern primaries and some of the
     operators may have been replaced by operator classes> †
<formula pattern primary> ::= <type> | ATOM | ANY | OF (<variable>) |
     OF (<procedure identifier>) | (<formula pattern structure>) |
     <extractor> <formula pattern primary>
<operator class> ::= "¦" <operator class name> "|"
<operator class name> ::= <variable>
<operator class assignment> ::= <operator class name> ←
     /[operator: <operator list>] <comm segment> <index segment>
<operator list> ::= <operator> | <operator list>, <operator>
<comm segment> ::= <empty> | [COMM: <logical value list>]
<index segment> ::= <empty> | [INDEX: <variable>]
<logical value list> ::= TRUE | FALSE | <logical value list>, TRUE |
     <logical value list>, FALSE
```

Semantics:

A mechanism is needed to determine precisely the structure of any formula. Formula patterns are used for this purpose; they constitute a set of predicates over the class of formula data structures. These formula patterns are sufficient in the sense that whatever constructions are used to create a formula, the pro-

† This is a short description of what could be a formal syntactic statement.

cess may be reversed by the choice of a sequence of predicates. Furthermore, a given formula pattern may be used to represent a class of possible formulae, and any formula may be tested for membership in this class.

In the definition of a formula, a formula expression F is compared with a formula pattern structure P to determine one of two things: (1) corresponding to the construction F==P, whether the expression F is an exact instance of the formula pattern structure P or, (2) corresponding to the construction F>>P, whether the formula expression F. contains as a subexpression an instance of the formula pattern structure P. Both consturctions F==P and F>>P are Boolean expressions yielding values TRUE or FALSE.

The Construction F==P. The formula expression F is defined recursively to be an exact instance of the formula pattern structure P as follows:

1. If P is an atomic formula then F==P is true if and only if F is the same atomic formula.

2. If P is a type name REAL, INTEGER, BOOLEAN, or FORM, then F==P is TRUE if and only if the value of F is a real number, an integer, a logical value, or a formula, respectively. (Note that numbers and logical values are not of type FORM.)

3. If P is the reserved word ATOM then F==P is TRUE if and only if the value of F is either a number, a logical value, or an atomic formula.

4. If P is the reserved word ANY then F==P is always TRUE.

5. If P is the construction OF (S), where S is a symbol which has been assigned a list of formula pattern structures, say $[P_1, P_2, \ldots, P_n]$, then F==P is TRUE if and only if $F{==}P \vee F{==}P_2 \vee \ldots \vee F{==}P_n$ is TRUE. S may optionally be given the special attribute INDEX; see Operator Classes.

6. If P is the construction OF (<procedure identifier>) where the procedure identifier names a Boolean procedure with one formal parameter specified of type FORM, (for example, BOOLEAN PROCEDURE B(X); FORM X; <procedure body>) then F==P is TRUE if and only if the procedure call B(F) yields the value TRUE.

7. If P is $A_1\ \omega_1\ B_1$, then F==P is TRUE if and only if (a) F is $A_2\ \omega_2\ B_2$, (b) A_2==A_1, (c) B_2==B_1, and (d) ω_1 is ω_2, where ω_1 and ω_2 are binary operators. Similarly, for unary operators, if P is $\omega_1\ B_1$ then F==P is TRUE if and only if (a) F is $\omega_2\ B_2$ and conditions (c) and (d) above are true. For the case where ω_i is an operator class, see the next section.

8. If P is

(a) A. $[S_1, S_2,\ldots S_n]$ where A is an array identifier

(b) A. $(S_1, S_2,\ldots, S_n)$ where A is a procedure identifier

(c) V. $\leftarrow S_1$ where V is a variable

or (d) .IF S_1 THEN S_2 ELSE S_3

where $S_1, S_2,\ldots S_n$ are formula pattern structures, then F==P if and only if, respectively:

(a) F = $'A[T_1, T_2,\ldots, T_n]'$

(b) F = $'A(T_1, T_2,\ldots, T_n)$

(c) F = $'V \leftarrow T_1'$

or (d) F = 'IF T_1 THEN T_2 ELSE T_3' respectively

where T_i==S_i $1 \leq i \leq n$.

Operator Classes. Before an operator class is used in a formula pattern, it must be defined. The definition is accomplished by an operator class assignment, which assigns to a variable of type SYMBOL an operator description list.

Suppose R is a variable declared of type SYMBOL for which the following operator class assignment has been executed:

R ← / [OPERATOR: +, -, /] [COMM: TRUE, FALSE, FALSE] [INDEX: J]

where J must be a variable declared of type INTEGER and where OPERATOR, COMM, and INDEX are reserved words used for special attributes. Let P be a formula structure having the form

$$A_1 \mid R \mid B_1$$

Then F==P is true if and only if (a) F is of the form 'A_2 ω B_2' and (b) one of the two following conditions holds:

(i) A_2==A_1, B_2==B_1, and ω is a member of the operator value list found on the description list of R. In the specific case above, this list is [+,-,/].

(ii) B_2==A_1, A_2==B_1, and ω is a member of the list of operators whose corresponding member of the COMM list is TRUE. (In this specific case, this must be +). (Note that [COMM: TRUE, FALSE, FALSE] need not appear on the description list of R at all in which case no commutative instances of any operator will be considered.)

If F==P is true the integer variable used as a value of the attribute INDEX will be set to an integer denoting the position of ω in the operator value list. (In the specific case above, J is set to 1, 2, or 3 according to whether ω was +, -, or / respectively). The operator ω is stored as the value of R. Later the construction | <R> | can be used in an expression in place of an operator, and the operator ω extracted during the previous matching will be used in the construction of the formula data structure that the expression represents. Alternatively, R may be assigned any operator by the assignment statement R ←<operator> and | <R> | may be used in the same fashion.

Extractors. Wherever an extractor is used in a formula pattern preceding a formula pattern primary the subexpression in F which matches that formula pattern primary is assigned as the value of the variable found to the left of the colon in the extractor. This variable must be of type FORM. This assignment is made as soon as the pattern primary is matched. Therefore, even though a pattern may fail as a whole, some of its extractors may have been assigned values. When ":" is used in this context it binds more closely than any other formula operator.

The Construction F>>P. The formula pattern F>>P is TRUE if F contains a subexpression, say S (which may be equal to F itself) such that S==P is TRUE. A recursive process is used to sequence through the set of subexpressions of F for successive testing against the formula pattern structure P. The sequencing has the properties that if two subexpressions S_1 and S_2 are both instances of P, then if S_2 is nested inside S_1, S_1 will match P first, and if neither is nested inside the other, then the one on the right in a linearized written form of S, is recognized first.

The formula pattern A:F>>B:P, in which extractors precede the right and left hand sides of the formula pattern, has the following meaning: First F>>P is tested. If the result is true then (a) the subexpression of F which matches P is stored as the value of B, and (b) a formula is constructed consisting of F with the subexpression matching P replaced by the previous value of B (the value B had before the assignment described in (a) took place). This formula is stored as the value of A.

Examples

Example 1. Let A,B,X,Y, and Z be declared of type FORM, let R be declared of type REAL, and let all form variables have their atomic formulae as values. Suppose that the statement

X ← 3 * SIN(Y) + (Y - Z) / R + 2 * R ;

has been executed. Consider the statement:

IF X >> A: INTEGER * B: SIN (FORM) THEN Z ← 2 * B + A

Since the pattern X>>A: INTEGER * B: SIN (FORM) is TRUE, the assignment Z ← 2 * B + A will be executed assigning as the value of Z the formula 2 * SIN (Y) + 3 because A has the value 3 and B has the value SIN (Y).

Example 2. Let X be of type SYMBOL, A, B, Y, M, T, G, and P be of type FORM, and D be of type BOOLEAN. Then executing the statements: X ← [REAL, INTEGER, BOOLEAN] ; G ← Y + 8 * (M - T) ; P ← FORM + A: OF (X) * B: FORM; D ← G==P; causes D to be set to TRUE because the pattern G==P is TRUE, and causes A to be set to 8 and B to be set to M - T.

Example 3. Suppose we execute the statements F ← 2 * (SIN(X ↑ 2 + Y ↑ 2) + COS (X ↑ 2 - Y ↑ 2)) / 5; T ← .T; G ← SIN (FORM) + COS (FORM) ; where all variables used are of type FORM. Then A: F>>T: G is a pattern with value TRUE. T gets assigned SIN (X ↑ 2 + T ↑ 2) + COS (X ↑ 2 - Y ↑ 2) the subpattern of F which matched G. A gets assigned 2 * T/5, a copy of F with the matched subpattern replaced by the previous value of F.

Example 4. Assume all variables in the following sequence of declarations and statements are of type FORM.

BOOLEAN PROCEDURE HASX(F) ; VALUE F ; FORM F ; HASX ← F>>X ;

G ← (X ↑ 2 + 3) ↑ 2 * (Y -1) : F ← A: OF (HASX) * B: (ANY -1) : T ← G==F ;

Then T is set to TRUE, A is set to (X ↑ 2 + 3) ↑ 2 and B is set to Y - 1. Here we use HASX to find any formula which is a function of X.

TRANSFORMED FORMULAE

Syntax:

<transformed formula> ::= <formula expression> .↓ <schema variable>

<schema variable> ::= <variable>

<schema assignment> ::= <schema variable> ← [<schema>]

<schema> ::= <schema element> | <schema>, <schema element>

<schema element> ::= <variable> | <single production> |
    <parallel production>

<single production> ::= <formula pattern structure> →<formula expression> |
    <formula pattern structure> . →<formula expression>

<parallel production> ::= [<parallel elements>]

<parallel elements> ::= <variable> | <single production> |
    <parallel elements>, <variable> |
    <parallel elements>, <single production>

The following is an additional restriction on the Syntax:

If any schema element has an extractor as its left-most member, then the whole element must be enclosed in parentheses.

Semantics:

Let F and G be formulae, and let P be a formula pattern. The application of the production P→ G to the formula F is defined as follows:

1. If F==P is FALSE then the application is said to fail.
2. If F==P is TRUE then the application is said to succeed, and F is changed according to G as follows: If P contains extractors, subexpressions of F matching corresponding parts of P are assigned as values of the extractors. Now in order to rearrange F according to the structure of the extractor variables in G, we change the

subformula of F which matched P into <u>REPLACE</u>(G). This substitutes the extracted subexpressions for their extractor variables in G causing the desired rearrangement.

For example, the distributive law of multiplication over addition may be executed as a transformation by applying the production

$$A:\ \underline{ANY} * (B:\ \underline{ANY} + C:\ \underline{ANY}) \rightarrow .A * .B + .A * .C \qquad (1)$$

to a given formula. Suppose that F contains $X \uparrow 2 * (Y + \underline{SIN}\ (Z))$. Then applying the production (1) to F will result in the extraction of the subexpressions $X \uparrow 2$, Y, and $\underline{SIN}\ (Z)$ into the variables A, B, and C respectively, and will cause the replacement of the atomic formulae A, B, and C occurring on the right hand side of (1) with these subexpressions, resulting in the transformation of the value of F into the formula $X \uparrow 2 * Y + X \uparrow 2 * \sin\ (Z)$.

A schema is a list of transformation rules. Each rule is either a single production or a list of single productions defining a parallel production. Variables occurring in a schema must have single productions as values. Expressions of the form $F. \downarrow S$, where F is a formula and S a list, are formula primaries, and thus may be used as constituents in the construction of formulae. The value of such a formula primary is a formula which results from applying the productions of the schema S to to the formula F according to one of the two possible sequencing modes explained as follows: Sequencing modes give the order in which productions of a given schema S are applied to a given formula F and to its subexpressions. The two sequencing modes differ in the order in which a given production will be applied to different subexpressions of F, and in the conditions defining when to stop.

One-by-one Sequencing:

One by one sequencing corresponds to a syntactic construction of the form $S \leftarrow [P_1, P_2, \ldots, P_n]$. For $j \leftarrow 1$ step 1 until n, production P_j is applied to F. If the application of P_j succeeds, P_j's transformation is applied to F and the whole process (starting at P_1) is reapplied to the result. If P_j fails to apply to F, it is applied recursively to each subexpression of F. Therefore, production P_k is applied to F if and only if production P_{k-1} is not applicable either to F itself or to any sub-expression of F. This sequencing will stop either when no production can be applied to F or any of its subexpressions or when a production containing $.\rightarrow$ has been executed.

Parallel Sequencing

Parallel sequencing corresponds to a syntactic construction of the form $\leftarrow [[P_1, P_2, \ldots, P_n]]$ or any form in which the brackets are nested at a depth of two. Here j is initially set to 1. when a production P_j is applied to F, if it succeeds, we apply its transformation and return to the beginning as with one-by-one sequencing. If the application P_j fails, production P_{j+1} is applied to F, and so on up to P_n. If all single productions of a parallel production fail at the topmost level of F, then the whole sequence is applied recursively to the next lowest subexpressions of F. Thus in parallel sequencing each one of the productions is applied at level k of the formula F only if all productions have failed at level k-1. The termination condition is reached when all productions fail at the bottom level of F or when a production containing $.\rightarrow$ has been executed.

In general a schema may have a combination of both sequencing modes, such as $S \leftarrow [P_1, P_2, [P_3, P_4, P_5], P_6]$. In this case P_1, P_2, the parallel sequence, and P_6 are treated one-by-one. When the sequence $[P_3, P_4, P_5]$ is reached in this schema, it is treated in parallel. Any number of these parallel schema may be used at the same level, but none may be nested at a depth greater than two.

The schema varaible S has to be declared of type SYMBOL. Optionally, a description list may be associated with S. If the special attribute INDEX occurs in the description list of S then, when the transformation has been completed, the value of an INTEGER variable used as the value of the attribute INDEX is set to 0 if no transformation took place, i.e., no production was applicable to F. The variable is set to 1 if at least one transformation took place and exit occurred because no further production of S was applicable. Finally, the variable is set to 2 if a production containing .→ was applicable. The following complete example of a schema clears fractions in arithmetic expressions.

```
BEGIN FORM F,X,A,B,C; SYMBOL S,P,T;
    A ← A: ANY: B← B: ANY; C← C: ANY;
    P ← / [OPERATOR: +] [COMM: TRUE] ; T ← / [OPERATOR: * ] [COMM: TRUE] ;
S ← [A ↑ (-B) → 1/ .A ↑ . B,
    A |P| (B/C) → (.A * .C + .B) /.C,
    A |T| (B/C) → (.A * .B) / .C,
    A -B/C → (.A * .C -.B) / .C,
    B/C - A → (.B -.A * .C) / .C,
    A/ (B/C) → (.A * ..C) / .B,
    (B/C) /A → .B/ (.C * .A),
    (B/A) ↑ C → .B ↑ .C / .A ↑ .C] ;
F ← (X + 3/X) ↑ 2 / (X -1/X);
```

PRINT (F. ↓ S) END

The above program will print X * (X ↑ 2 + 3) ↑ 2/ (X ↑2 * (X ↑ 2 - 1)).

PRECEDENCE OF FORMULA OPERATORS

Now that all the formula expressions have been explained, we present the precedence of formula operators in both expressions and patterns:

:   (done first)

↑

- +   (unary)

/ *

- +   (binary)

= ≠ > <   -▷ -<

¬

∧

∨

| | or |< >|

→

. ↓   (done last)

In cases of equal precedence, association to the left is used.

SPECIAL FUNCTIONS

The following functions are built into Formula Algol:

DERV (F,X)	A FORM function designator whose value is the derivative of F with respect to X.
CELLS	An INTEGER function designator whose value is the number of cells remaining on the available space list [see 4].

FORM AL-3-48

CHAPTER IV

LIST PROCESSING

SYMBOL VARIABLES

Variables may be declared of type SYMBOL, indicating that their values are to be list structures. In addition to this function, they may also serve as data to be manipulated and stored in list structures. In this context they are called atomic symbols. When a symbol S is declared, as with a form variable, its value is initialized to the atomic symbol S and a description list is associated with it.

SYMBOL ARRAYS

Arrays may be declared of type SYMBOL whose elements may be list structures. Again like form arrays, they are accessed in the normal manner and they are not initialized.

SYMBOLIC EXPRESSIONS

Syntax:

```
<symbolic expression> ::= .<identifier>|
        <variable>|<function designator>|
        <value retrieval expression>|<selection expression>|
        "<" <symbolic expression>">"| NIL
```

Semantics:

A symbolic expression has as its value either an atomic symbol or a list according to the following rules:

1. If it is a symbol variable preceded by a dot, its value is the atomic

symbol represented by the variable.

2. If it is a symbol variable S, its value is the contents of S. The contents of a symbol may be modified by assignment statements (pg. 52), push and pop statements (pg. 61), and extractors (pg. 58).
3. If it is a function designator resulting from the declaration of a symbol procedure, its value is that assigned to the procedure identifier by executing the body of the procedure using actual parameters given in the function designator call.
4. Selection expressions are explained on page 55.
5. Value retrieval expressions are explained on page 53.
6. If it is of the form<T>, where T is a symbolic expression, the value of T is first computed and must result in an atomic symbol. The value of the symbolic expression is then the contents of that atomic symbol. The angular brackets may be nested arbitrarily many times to provide many levels of indirect access.
7. NIL is a special symbol with no contents or description list which may be treated as an atomic symbol. It acts as an identity element under concatenation of list elements (pg. 51).

LISTS

Syntax:

```
<list> ::= <list element>|<list>,<list element>
<list element> ::= <expression>|<list expression><description list>|
          <symbolic expression><description list>|<list pattern primary>
<list expression> ::= [<list>]
<expression> ::= <arithmentic expression>|<Boolean expression>|
          <formula expression>|<formula pattern structure>|
          <symbolic expression>|<list expression>|<list expression>
```

Semantics:

Symbols may be concatenated into a list by writing them one after another, separating them with commas, and enclosing them in brackets. In addition to symbol variables, any expression except a designational expression may be written as an element of a list and its value will be entered. For example, let X. Y, and Z be formula variables, let A, B, and C be Boolean variables, let U, V, and W be real variables, and let R, S, and T be symbol variables. Then the value of

[X* SIN(Y), 3 + 2 * U, IF B THEN R ELSE T, [R,T,R], -36]

is obtained by causing each expression on the right to be evaluated, and their results concatenated. If one of the results is NIL, the element disappears completely from the list. Automatic data term conversion results from using non-symbolic values in lists. The second from the last item in the above list is the quantity [R,T,R], which becomes a sublist of the list. Hence, the expression, in reality, is a list structure. It is further possible for certain of the elements of a list to bear local description lists (pg.53).

It should be noted that one-element lists and single values are treated identically when appearing as the contents of a symbol. Thus S ← 3 and S ← [3] are the same when S is a symbol variable. If we wished to make the contents of S a list with one number, 3, we would execute S ← [[3]].

List pattern primaries may be stored in lists so that the list may later be used in a list pattern (pg. 58).

ASSIGNMENT STATEMENTS

Syntax:

We may extend the Algol 60 syntax as follows:

```
<assignment statement> ::= ... |
        <symbolic expression> ← <expression>|
        <symbolic expression> ← <description list>|
        <variable> ← <description list>
```

Semantics:

When a symbolic expression (other than a variable) appears on the left hand side of an assignment statement, it is first evaluated and must result in an atomic symbol. The value of the expression on the right then becomes its contents, or the description list on the right replaces its description list. Thus any symbolic expression, unlike those of other variables, is allowed on the left side of an assignment. In the case that a symbol variable appears on the left by itself, the right side expression replaces the contents of the variable mentioned, instead of the contents of its value. Description lists may also be assigned to variables of type FORM.

DESCRIPTION LISTS

Syntax:

```
    <description list> ::= /<attribute-value list>
    <attribute value list> ::= <attribute value segment>|
         <attribute value list><attribute value segment>
    <attribute value segment> ::= [<attribute>:<list>]|
         [<attribute> : <empty>]
    <value retrieval expression> ::= <identifier> ( <form or symb>)|
         THE <attribute> OF <form or symb>
    <form or symb> ::= <symbolic expression>|<formula expression>
    <attribute> ::= <symbolic expression>
```

Semantics:

A description list is a sequence of associated attributes and value-lists. An attribute must be a symbolic expression which results in an atomic symbol. Each attribute is followed by its value-list which is of the same form as an ordinary list. It may contain more than one member, it may contain only one member, or it may be empty. A description list may be attached to one of three types of objects:

1. A variable declared of type SYMBOL for which there are two cases (a) global attachment, and (b) local attachment.

2. A variable declared of type FORM.

3. A sublist of a list.

To describe these uses, consider these examples: Assuming that all variables involved have been declared of type SYMBOL, the statements

```
S ←/[TYPES: MU,PI,RHO][ANCESTORS: ORTHOL,PARA5][COLOR: GREEN];        (1)
T ← [F,A/[NUM:1],B,C,A/[NUM: 2],D,E];                                (2)
```

assign respectively a description list to S and a list as the contents of T. The description list attached to S is globally attached, meaning that it is permanently bound to S for the lifetime of the variable S. In the list assigned as the value of T, the symbol A occurs twice - in the second and fourth positions. The description lists attached to these two separate occurrences of A are attached locally, meaning that the separate occurrences of a given atomic symbol within a list have been given descriptions which interfere neither with each other nor with the global description list attached to A if such should occur. The attributes and values of a given local description list are accessible only by means of symbolic expressions accessing that particular occurrence of the symbol to which the given local description list attached.

Thus, if one desired to access the global description list of that copy of A, he would remove it from the list T, destroying its local description list and then perform the value retrieval. E.g., T1 ← 2ND OF T; then use NUM OF T1.

In the following examples suppose F is a variable declared of type FORM and that all other variables involved are variables declared of type SYMBOL.

```
F ← /[PROPERTIES: CONTINUOUS, DIFFERENTIABLE];                              (3)
V ← [A, [B,C]/[PROCESSED: TRUE],A, [B,C]/[PROCESSED: FALSE], A];            (4)
```

In example (3) a description list is attached to a formula. In example (4) the list assigned to be the contents of V has two identical sublists [B,C] in the second and fourth positions having different local description lists.

Value lists stored in description lists are retrieved by means of value retrieval expressions. To accomplish retrieval, two arguments must be supplied: (1) an attribute consisting of an atomic symbol and (2) the atomic symbol or formula having the description list. The attribute is then located on the description list and its associated value list becomes the value of the retrieval expression. If there is no description list, or if there is a description list but the attribute does not appear on it, or if the attribute does appear on it but has an empty value list, then the value of the retrieval expression is the symbol NIL. Thus in examples (1) and (2) above, the value retrieval expressions COLOR(.S), NUM(2ND OF T), and NUM(3RD OF T) have the values GREEN, 1, and NIL respectively. If in a value retrieval expression either the description list or the attribute is missing, it is added with a value of NIL. The construction, THE COLOR OF .S, accomplishes the same function as COLOR(.S) but is slightly more versatile in that any symbolic or formula expression may be used to calculate the attribute whereas only identifiers may be used for the attribute in the form <identifier> (<symbolic expression>).

SELECTION EXPRESSIONS

Syntax:

```
<selection expression>::= <selector> OF < symbolic expression >
<ordinal suffix>::= ST | ND | RD | TH
<ordinal selector>::= <arithmetic primary><ordinal suffix>|LAST|FIRST
<elementary position>::= <ordinal selector>|
        <ordinal selector> <kind> |
        <ordinal selector> INTEGER <arithmetic primary>
<kind>::= <augmented  type> | <expression> | <class name>
<position>::= <elementary position> | <arithmetic primary>
          <ordinal suffix> BEFORE <elementary position> |
          <arithmetic primary ><ordinal suffix> AFTER <elementary position>
<selector>::= BETWEEN<position>AND<position>|ALL AFTER<position>|
          ALL BEFORE<position>|FIRST<arithmetic primary>|
          LAST<arithmetic primary>|<position>|ALL<kind>|
<augmented type>::= REAL|INTEGER|BOOLEAN|FORM|SYMBOL|SUBLIST|ATOM|ANY
```

Semantics:

Selection expressions are formed by composing selector operators with symbolic expressions. A symbolic expression is first evaluated producing a symbolic data structure as a value. A selector operator is then applied to the resulting symbolic data structure to gain access to a part of it. Assume first that the symbolic data structure S on which a selector operates is a simple list. Then

1. An ordinal selector refers to an element of this list either by numerical position, or by designating the last element. E.g. 3 RD OF S, LAST OF S.

2. An elementary position refers to an element of this list by designating it (a) as the N TH or LAST instance of an augmented type, e.g. N TH REAL, LAST SUBLIST, where N is an expression whose value is an integer, (b) as the N TH or LAST instance of the value of an expression, e.g. N TH (F+G), LAST [A,B,C], (c) as the N TH or LAST instance of a member of a class (pg.61), e.g. 5TH (|TRIGFUNCTION|), LAST (|VOWEL|), (d) or by ordinal selection.

3. A position refers to an element of this list either by designating its elementary position or by designating it as the N TH BEFORE or in the N TH AFTER some elementary position.

4. A selector refers to an element by its position or else designates one of the following sublists of the list
   (a) The sublist between two positions not including either position named, e.g. BETWEEN 3 RD and 7TH OF S produces a list consisting of the 4th, 5th, and 6th.

   (b) The sublist consisting of all elements before or after a given position, e.g. ALL AFTER 3 RD SYMBOL OF S, ALL BEFORE LAST REAL OF S.

(c) The sublists consisting of the first n elements or the last n elements, e.g. FIRST 3 OF S, LAST K OF S.

(d) The sublists composed by selecting and then concatenating (i) all instances of a given expression, e.g. ALL F OF S, (ii) all instances of a given augmented type, e.g. ALL REAL OF S, (iii) all instances of elements which are members of a given class, e.g. ALL (|TRIGFUNCTION|) OF S. These elements are concatenated in the same order that they occur in the list from which they are selected.

Selectors may be compounded to access sublists and their elements. Suppose the statement S ←[A,[X,X,[A,A[,X],A] has been executed. Then the expression 2 ND OF S is a list valued symbolic expression with the list [X,X, [A,A], X] as value, whereas the expression 3RD OF 2 ND OF S has the list [A,A] as value, and the expression LAST OF 3 RD OF 2 ND OF S has the single atomic symbol A as value.

If a selector refers to an element of a list which doesn't exist because the list is of insufficient length (e.g. the 5th of a 3-element list), then the value of the expression is NIL, and the extra NIL's are added to the structure to make it the right length.

Note that there could be an ambiguity with the statement FIRST 3 OF S. It could mean the first 3 elements of S or the first integer '3' in S. We have chosen to use the former interpretation and to require one to write FIRST INTEGER 3 OF S if he desires the latter.

LIST PATTERNS

Syntax:

```
<list pattern>::= <symb or list> == <symb or list>|
                  <symbolic expression> == <kind>|
                  <symb or list> = <symb or list>
<symb or list>::= <symbolic expression>|<list expression>
<list pattern primary>::= $ | $<arithmetic primary>|
                  <kind>|<extractor><list element>
<extractor>::= <variable>:
```

Semantics:

List patterns are predictates for determining the structure of lists. They use mechanisms like those found in COMIT [5] to test whether a list is an instance of a certain linear pattern. The construction to the left of the == is the list structure being tested according to the pattern on the right. This pattern will consist of a sequence of list pattern primaries (possibly one), some of which may be ordinary list elements. In order for the list to match the pattern, the entire list must match the pattern, not just a subpart of it as in COMIT.

The elements of the list pattern evoke tests as follows:

The normal list elements are evaluated as in ordinary lists. If they result in atomic constructions, these are used in direct equality tests. If they result in lists, then each element of the list is treated as another list pattern primary. The one exception to this is if the element is actually a sublist (is enclosed in brackets). This will only match the list pattern primaries of the pattern sublist. This feature allows patterns to test whole list structures.

The other list pattern primaries are matched in the following ways:

(1) An augmented type will match an element which is of that type as defined for formula patterns. (Page 37). In addition SYMBOL will match only atomic symbols and SUBLIST naturally matches sublists.

(2) A class name will match an element which satisfied its class test (pg. 61).

(3) $n will match any n consecutive elements, where n is an expression whose value is a positive integer.

(4) $ will match an arbitrary number of elements, including 0. However, there is a limitation on this which can be explained by giving a brief idea of the scanning algorithm for $.

When a $ is encountered in the pattern, we first pair it with no elements and then try to match the rest of the pattern. This failing, we pair it with one element and try again. We keep increasing the scope of the $ until a match is found or we run over the end of the list. However, once we have matched the pattern primaries to the right of a $ up to the next $, we consider the first $ fixed and we do not try to enlarge its scope any more. If we meet failure in matching the second dollar sign, the pattern fails. We do not back up to the first. (E.g. [1,A,2,B,A,2,B,C] == [$,A,$,B,$1] is false since after matching the B after the second $, we will not back up to find new matches for the $'s.)

A. It should be noted that testing for the type or class of a single element is nothing more than a list pattern in which the right side is a single list pattern primary. Thus we may write:

3 rd OF S == INTEGER

THE A OF B == (| NOUN |)

Like formula patterns, list patterns are boolean primaries and thus may be combined with other booleans using logical connectives or may be used in IF - THEN statements.

As an example, consider the list

```
S ← [ A, 1, B, C, A, A, C ];
S == [ A, INTEGER, $, A, $2 ] is TRUE.
```

As with the formula pattern structures, list patterns may function not only as predicates but also as selectors. The same mechanism is used to accomplish this. If any list pattern primary in a list pattern structure is preceded by a variable declared of type SYMBOL followed by a colon, then in the event that there is a match, the element which matches the list pattern primary becomes the value of the symbol variable. It may then be accessed at any later point in the program. In the case that there is only a partial match, however, some of the extractors may be assigned values anyway.

Suppose the statement S ← [A,B,C,D] has been executed where all variables are symbols and where A, B, C, and D have as values their atomic symbols. Then, executing the statement

```
IF S == [T:$2, V:$2] THEN S ← [V,T];
```

changes the contents of S to be the list [C,D,A,B]. This is because the contents of T is the list [A,B], and V has as its value the list [C,D].

Two list structures may be tested for exact equality by means of a single =. This is necessary above the == predicate only in that it permits testing of stored list patterns. Thus we may store a pattern containing ':', REAL, '$', etc., and then later test it for exact form using those symbols in the patterns. For example, "== REAL" will match any real number; while "= REAL" will match only the element "REAL".

CLASS TESTS

Syntax:

<class name> ::= ("|"<symbolic expression>"|")

<class definition> ::= Let <class name = [<formal parameter> "|" <Boolean expression>]

Semantics:

Sets may be defined by means of class definition. For example, suppose the statement V ← [A,E,I,O,U] has been executed. Then the statement LET (|VOWEL|) = [X | AMONG(X,V)]; defines the set of all vowels where AMONG(P,Q) is a Boolean procedure which is TRUE if P is an element of the list contained in Q, and FALSE otherwise. Suppose that having previously executed the statement S ← [A,B,C], we execute the statement

IF 1 ST OF S == (|VOWEL|) THEN <statement>

The list pattern 1 ST OF S == (|VOWEL|) will be evaluated by first computing the value of the expression 1ST OF S, which is the symbol A, and second by substituting A for the formal parameter X in the class definition of (|VOWEL|). This results in the execution of procedure AMONG(A,V) which produces the value TRUE. Thus, A is a member of the class (|VOWEL|), and the list pattern 1ST OF S == (|VOWEL|) is TRUE, causing the <statement> to be executed.

Any arbitrary Boolean expression, including a Boolean procedure call, may be used to define a class. Thus the full generality of Boolean procedures is obtained.

PUSH DOWN AND POP UP STATEMENTS

Syntax:

<push down operator> ::= ↓|<push down operator> ↓

<pop up operator> ::= ↑|<pop up operator> ↑

<push down statement> ::= <push down operator> <symbolic expression>

<pop up statement> ::= <pop up operator> <symbolic expression>

Semantics:

The contents of any variable declared of type SYMBOL is a push down stack. The contents of the variable consists of the current topmost level of the push down stack. Applying a single push down operator ↓ to such a variable pushes down each level of the stack making the topmost level (level 0) empty and replacing the contents stored at level k with the contents stored previously at level k-1. The empty topmost level may then acquire a value as its contents by means of the execution of an appropriate assignment statement. A lower level of the push down stack is not accessible to the operation of extracting contents until the execution of a pop up statement restores it to the topmost level. Applying a single pop up operator ↑ to the name of a variable destroys the contents of the topmost level and replaces the contents stored at level k with the contents previously at level k + 1. A push down operator (pop up operator) consisting of n consecutive occurrences of a single push down operator (pop up operator) has the same effect as n consecutive applications of a single push down operator (pop up operator). A push down operator (pop up operator) is applied to a symbolic expression by evaluating the symbolic expression and, if it results in an atomic symbol, the operator is applied to the push down stack which is the contents of the atomic symbol as described above. Any structure which occupies the contents of a symbol variable S may become the contents of a lower level of the push down stack in S by application to the push down operator S. In particular, list structures may be stored in the push down stack in S.

ADDITIONAL FOR STATEMENTS

Syntax:

<for list> ::= ... |

ELEMENTS OF <symbolic expression> |

ATTRIBUTES OF <symbolic expression>

<for clause> ::= ... | FOR <symbolic expression> ← <for list> DO |

PARALLEL FOR <symb or list> ←

ELEMENTS OF <symb or list> DO

Semantics:

We may wish to generate the element of a list or the attributes of a description list one by one in order to assign them to the controlled variable in a FOR statement. Attributes on the description list of the value of S, which must be atomic symbols, are generated in the order that they occur by "ATTRIBUTES OF S", and "ELEMENTS OF S" generates the successive elements of the list which is of the value of S. In the former case S must be any symbolic expression with an atomic symbol as value because the attributes from its description list will be generate. In the latter case S may be any list valued symbolic expression. Successive elements generated are assigned to the control variable given in the FOR clause. In either case, the lists of values to be assigned to the control variable are fixed upon initial entry to the FOR statement, and any changes to them in the body of the FOR statement will not be reflected.

Parallel generation is also permissible. Here the expression to the left of the "←" is a list of n atomic symbols and the expression on its right is a list of n lists or n symbols containing lists. For example: if S ← [A,B,C], T ← [D,E], and U ← [F, G, H, M] have been executed where the variables A through I have as values their atomic symbols then executing the statement

PARALLEL FOR [I,J,K] ← ELEMENTS OF [[S], [T], [U]] DO L ← [L,I,J,K];

causes the following to happen. First, all first elements of the lists contained in S, T, U, respectively are generated and placed in the contents of the controlled variables I, J, and K, respectively. Control then passes to the body of the parallel FOR statement and returns when finished with its execution. On the second cycle, all second elements of S, T, and U are generated and placed in the controlled variables I, J and K, respectively. Control then passes the statement following the DO and returns. On the third cycle, all third elements are generated, on the fourth cycle all fourth elements are generated, and so on. If any list runs out of elements before any of its neighbors, the symbol NIL continues to be generated. The parallel generation stops just before the symbol NIL would have been generated from all lists.

List valued symbolic expressions may be used to supply lists of control variables and lists of lists to generate in parallel, as, for example, in the construction

PARALLEL FOR V ← ELEMENTS OF W DO L ← [L, I, J, K];

where the statements V ← [I,J,K] and W ← [[S], [T], [U]] have been executed previously. At the end L should contain [L,A,D,F,B,E,G,C,H,I].

EDITING STATEMENTS

Syntax:

<editing statement> ::= INSERT <symb or list> <insertion locator list> <symbolic expression> | < DELETE <selector list> <symbolic expression> | DELETE <symbolic expression> | ALTER <selector list> <symbolic> TO <expression> | <description list editing statement>

<insertion locator> ::= BEFORE <position> OF | AFTER <position> OF

<insertion locator list> ::= <insertion locator> | <insertion locator list>, <insertion locator>

<selector list> ::= <selector> OF | <selector list>, <selector> OF

<description list editing statement> ::= THE <symbolic expression> OF <symbolic expression> <is phrase> <expression>

<is phrase> ::= IS | IS NOT | IS ALSO

Semantics:

Editing statements are used to transform, permute, alter, and delete elements of lists. The INSERT construction causes a list structure to be inserted at each of the places given by an insertion locator list. The list on which insertion is to be performed is obtained by evaluating the symbolic expression which occurs last in the statement. The expression to be inserted is then evaluated, and if it produces a list, each element of the list is inserted as an element of the list being altered. To insert a sublist in a list it must be surrounded by two sets of brackets. Thus, if S ← [A,B,C,1]; INSERT [X,Y] BEFORE 2ND OF, AFTER LAST OF S causes S to be [A,X,Y,C,1,X,Y] but INSERT [[X,Y]] BEFORE 2ND OF, AFTER LAST OF S causes S to be [A,[X,Y],C,1,[X,Y]]. All the insertions take place simultaneously.

The first DELETE construction above performs simultaneous deletions of parts of a list. The list of parts to be deleted is specified by the selector list in accord with the semantics of selectors. Thus, DELETE 2ND BEFORE FIRST INTEGER OF S will cause our original list S to be [A,C,1]. The second delete construction removes the value of the symbolic expression from the list structure in which it resides according to the form of the symbolic expression. Thus, DELETE THE COLOR OF APPLE removes the value-list of this attribute. DELETE . S is meaningless.

The ALTER construction is equivalent to a series of deletions followed by insertions at each point where something was deleted.

ALTER ALL SYMBOL OF S TO [3,4] changes S to [3,4,3,4,3,4,1].

Whenever an assignment is made of a list structure, the entire structure is copied and the copy becomes the contents at the left-side variables. Thus editing statements should be used instead of assignment statements if a copy is not needed when altering a list. For example:

INSERT A AFTER LAST OF S

is more efficient than

S ← [S,A]

Description List Editing Statements. Description list editing statements add or delete values on description lists. They supplement the role performed by assignment statements in this regard. Suppose that S ← / [THPE: MU, PI, RHO] [COLOR: RED] has been executed. Then, if the statement THE COLOR OF S IS GREEN; is executed, the value of the attribute COLOR on the description list of S is replaced with the new value GREEN. This yields the altered description list / [TYPE: MU, PI, RHO] [COLOR: GREEN] as a result. On the other hand, the statement: THE COLOR OF S IS ALSO GREEN; could be executed. Instead of replacing the color RED with the value GREEN the latter statement appends the value GREEN to the value list following the attribute COLOR. This yields the description list / [TYPE: MU, PI, RHO] [COLOR: RED, GREEN] as a result. Finally, description list editing statements may be used to delete values from value lists of a specific attribute. Executing the statement: THE TYPE OF S IS NOT PI; alters the above description list to / [TYPE: MU, RHO] [COLOR: GREEN].

SPECIAL FUNCTIONS

CREATE(N)	A SYMBOL function designator whose value is a list of N created atomic symbols. CREATE ≡ CREATE (1).
ERADL(S)	A statement which erases the description list attached to the symbol S.
COUNT(L)	An INTEGER function designator having as value the number of elements in the list which is the value of L.
EMPTY(S)	A BOOLEAN function designator which is true if S contains

	no elements. It is false if the structure contains anything including NIL.
AMONG(S,L)	A BOOLEAN function designator which is TRUE if S is a member of the list L and FALSE otherwise.

APPENDIX I

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, C. KATZ, H. RUTISHAUSER, J. H. WEGSTEIN
F. L. BAUER, J. McCARTHY, K. SAMELSON, A. VAN WIJNGAARDEN
J. GREEN, A. J. PERLIS, B. VAUQUOIS, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

CONTENTS

This report was published simultaneously in the Communications of the ACM, 6, No. 1 (1963), 1-17, the Numerische Mathematik, and the Computer Journal.

REVISED ALGOL 60

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,[1,2] as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an *ALGOL Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the *Communications of the ACM*, where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives,[3] from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference [Edited by M. Woodger]

A meeting of some of the authors of ALGOL 60 was held on April 2–3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman, IFIP TC 2.1 Working Group ALGOL)
J. Green	R. Franciotti	
C. Katz	P. Z. Ingerman	
R. Kogon (representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in *ALGOL Bulletin* No. 14 were used as a guide.

This report* constitutes a supplement to the ALGOL 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced pro-

* [EDITOR'S NOTE. The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 conference modified according to the agreements reached during the April 1962 conference. Thus the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote; 2.3; 2.7; 3.3.3; 3.3.4.2; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.3; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.]

[1] Preliminary report—International Algebraic Language. *Comm. ACM 1*, 12 (1958), 8.

[2] Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson. *Num. Math. 1* (1959), 41-60.

[3] William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

gramming languages will lead to better resolution:

1. Side effects of functions
2. The call by name concept
3. **own:** static or dynamic
4. For statement: static or dynamic
5. Conflict between specification and declaration

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference Language	*Publication Language*
Subscript bracket [ ]	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	Raising of the exponent
Parentheses ( )	Any form of parentheses, brackets, braces
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign

DESCRIPTION OF THE REFERENCE LANGUAGE

Was sich überhaupt sagen lässt, lässt sich klar sagen; und wovon man nicht reden kann, darüber muss man schweigen.
LUDWIG WITTGENSTEIN.

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given.[4]

1.1. Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae.[5] Their interpretation is best explained by an example

⟨ab⟩ ::= (| [| ⟨ab⟩ (| ⟨ab⟩⟨d⟩

Sequences of characters enclosed in the brackets ⟨ ⟩ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

```
[(((1(37(
(12345(
(((
[86
```

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨ ⟩ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

⟨empty⟩ ::=
(i.e. the null string of symbols).

[4] Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

[5] Cf. J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris, June 1959.

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts.

The reference language is built up from the following basic symbols:

⟨basic symbol⟩ ::= ⟨letter⟩|⟨digit⟩|⟨logical value⟩|⟨delimiter⟩

2.1. Letters

⟨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*

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings[6] (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.1. Digits

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical Values

⟨logical value⟩ ::= **true**|**false**

The logical values have a fixed obvious meaning.

2.3. Delimiters

⟨delimiter⟩ ::= ⟨operator⟩|⟨separator⟩|⟨bracket⟩|⟨declarator⟩|
⟨specificator⟩
⟨operator⟩ ::= ⟨arithmetic operator⟩|⟨relational operator⟩|
⟨logical operator⟩|⟨sequential operator⟩
⟨arithmetic operator⟩ ::= +|−|×|/|÷|↑
⟨relational operator⟩ ::= <|≦|=|≧|>|≠
⟨logical operator⟩ ::= ≡|⊃|∨|∧|¬
⟨sequential operator⟩ ::= **go to**|**if**|**then**|**else**|**for**|**do**[7]
⟨separator⟩ ::= ,|.|$_{10}$|:|;|:=|␣|**step**|**until**|**while**|**comment**
⟨bracket⟩ ::= (|)|[|]|‘|’|**begin**|**end**
⟨declarator⟩ ::= **own**|**Boolean**|**integer**|**real**|**array**|**switch**|
procedure
⟨specificator⟩ ::= **string**|**label**|**value**

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of

[6] It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface type in printed copy—Ed.] is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report [not including headings—Ed.], boldface will be used for no other purpose.

[7] **do** is used in for statements. It has no relation whatsoever to the *do* of the preliminary report, which is not included in ALGOL 60.

a program the following "comment" conventions hold:

The sequence of basic symbols:	*is equivalent to*
; **comment** ⟨any sequence not containing ;⟩;	;
begin comment ⟨any sequence not containing ;⟩;	**begin**
end ⟨any sequence not containing **end** or ; or **else**⟩	**end**

By equivalence is here meant that any of the three structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4. Identifiers

2.4.1. Syntax

⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩

2.4.2. Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax

⟨unsigned integer⟩ ::= ⟨digit⟩|⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩|+⟨unsigned integer⟩|
 −⟨unsigned integer⟩
⟨decimal fraction⟩ ::= .⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩|⟨decimal fraction⟩|
 ⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩|⟨exponent part⟩|
 ⟨decimal number⟩⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩|+⟨unsigned number⟩|
 −⟨unsigned number⟩

2.5.2. Examples

0	-200.084	$-.083_{10}-02$
177	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$2-_{10}4$	$+_{10}+5$

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types

Integers are of type **integer.** All other numbers are of type **real** (cf. section 5.1. Type Declarations).

2.6. Strings

2.6.1. Syntax

⟨proper string⟩ ::= ⟨any sequence of basic symbols not containing
 ‘ or ’⟩|⟨empty⟩
⟨open string⟩ ::= ⟨proper string⟩|‘⟨open string⟩’|
 ⟨open string⟩⟨open string⟩
⟨string⟩ ::= ‘⟨open string⟩’

2.6.2. Examples

‘5k,,−‘[[[‘∧=/:’Tt’’
‘.. This ␣ is ␣ a ␣ ‘string’’

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ‘ and ’ are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.7. Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8. Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩ ::= ⟨arithmetic expression⟩|⟨Boolean expression⟩|
 ⟨designational expression⟩

3.1. Variables

3.1.1. Syntax

⟨variable identifier⟩ ::= ⟨identifier⟩
⟨simple variable⟩ ::= ⟨variable identifier⟩
⟨subscript expression⟩ ::= ⟨arithmetic expression⟩
⟨subscript list⟩ ::= ⟨subscript expression⟩|⟨subscript list⟩, ⟨subscript expression⟩
⟨array identifier⟩ ::= ⟨identifier⟩
⟨subscripted variable⟩ ::= ⟨array identifier⟩[⟨subscript list⟩]
⟨variable⟩ ::= ⟨simple variable⟩|⟨subscripted variable⟩

3.1.2. Examples

epsilon
detA
*a*17
Q[7,2]
x[*sin*(*n*×*pi*/2),*Q*[3,*n*,4]]

3.1.3. Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2. Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2. Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. Arithmetic Expressions).

3.1.4.2. Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. Array Declarations).

3.2. Function Designators

3.2.1. Syntax

⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩| ⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩
⟨parameter delimiter⟩ ::= ,|)⟨letter string⟩ :(
⟨actual parameter list⟩ ::= ⟨actual parameter⟩| ⟨actual parameter list⟩⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨actual parameter part⟩ ::= ⟨empty⟩|(⟨actual parameter list⟩)
⟨function designator⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩

3.2.2. Examples

sin(*a*−*b*)
J(*v*+*s*,*n*)
R
S(*s*−5)Temperature:(*T*)Pressure:(*P*)
Compile(' := ')Stack:(*Q*)

3.2.3. Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E)	for the modulus (absolute value) of the value of the expression E
sign(E)	for the sign of the value of E(+1 for E>0, 0 for E=0, −1 for E<0)
sqrt(E)	for the square root of the value of E
sin(E)	for the sine of the value of E
cos(E)	for the cosine of the value of E
arctan(E)	for the principal value of the arctangent of the value of E
ln(E)	for the natural logarithm of the value of E
exp(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type **real** and **integer**. They will all yield values of type **real**, except for *sign*(E) which will have values of type **integer**. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.5. Transfer functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely,

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. Arithmetic Expressions

3.3.1. Syntax

⟨adding operator⟩ ::= +|−
⟨multiplying operator⟩ ::= ×|/|÷
⟨primary⟩ ::= ⟨unsigned number⟩|⟨variable⟩| ⟨function designator⟩|(⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩|⟨factor⟩↑⟨primary⟩
⟨term⟩ ::= ⟨factor⟩|⟨term⟩⟨multiplying operator⟩⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩| ⟨adding operator⟩⟨term⟩|⟨simple arithmetic expression⟩ ⟨adding operator⟩⟨term⟩
⟨if clause⟩ ::= **if** ⟨Boolean expression⟩**then**
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩| ⟨if clause⟩⟨simple arithmetic expression⟩**else** ⟨arithmetic expression⟩

3.3.2. Examples

Primaries:

$7.394_{10}{-8}$
sum
$w[i+2,8]$
$\mathit{cos}(y+z\times 3)$
$(a-3/y+vu\uparrow 8)$

Factors:

omega
$\mathit{sum}\uparrow \mathit{cos}(y+z\times 3)$
$7.394_{10}{-8}\uparrow w[i+2,8]\uparrow(a-3/y+vu\uparrow 8)$

Terms:

U
$\mathit{omega}\times \mathit{sum}\uparrow \mathit{cos}(y+z\times 3)/7.394_{10}{-8}\uparrow w[i+2,8]\uparrow$
$\quad(a-3/y+vu\uparrow 8)$

Simple arithmetic expression:

$U-Yu+\mathit{omega}\times \mathit{sum}\uparrow \mathit{cos}(y+z\times 3)/7.394\ \ {-8}\uparrow w[i+2,8]\uparrow$
$\quad(a-3/y+vu\uparrow 8)$

Arithmetic expressions:

$w\times u-Q(S+Cu)\uparrow 2$
if $q>0$ **then** $S+3\times Q/A$ **else** $2\times S+3\times q$
if $a<0$ **then** $U+V$ **else if** $a\times b>17$ **then** U/V **else if**
  $k\neq y$ **then** V/U **else** 0
$a\times \mathit{sin}(\mathit{omega}\times t)$
$0.57_{10}12\times a[N\times(N-1)/2,\ 0]$
$(A\times \mathit{arctan}(y)+Z)\uparrow(7+Q)$
if q **then** $n-1$ **else** n
if $a<0$ **then** A/B **else if** $b=0$ **then** B/A **else** z

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4. Operators and types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators $+$, $-$, and $\times$ have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations ⟨term⟩/⟨factor⟩ and ⟨term⟩ ÷ ⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b\times 7/(p-q)\times v/s$$

means

$$((((a\times(b^{-1}))\times 7)\times((p-q)^{-1}))\times v)\times(s^{-1})$$

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator ÷ is defined only for two operands both of type **integer** and will yield a result of type **integer**, mathematically defined as follows:

$$a\div b=\mathit{sign}\ (a/b)\times \mathit{entier}(\mathit{abs}(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨factor⟩↑⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

$$2\uparrow n\uparrow k \qquad \text{means} \qquad (2^n)^k$$

while

$$2\uparrow(n\uparrow m) \qquad \text{means} \qquad 2^{(n^m)}$$

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

$a\uparrow i$	If $i>0$, $a\times a\times \ldots \times a$ (i times), of the same type as a.
	If $i=0$, if $a\neq 0$, 1, of the same type as a.
	if $a=0$, undefined.
	If $i<0$, if $a\neq 0$, $1/(a\times a\times \ldots \times a)$ (the denominator has $-i$ factors), of type **real**.
	if $a=0$, undefined.
$a\uparrow r$	If $a>0$, $\mathit{exp}(r\times \mathit{ln}(a))$, of type **real**.
	If $a=0$, if $r>0$, 0.0, of type **real**.
	if $r\leqq 0$, undefined.
	If $a<0$, always undefined.

3.3.5. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first:	$\uparrow$
second:	$\times / \div$
third:	$+ -$

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of **real** quantities

Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean Expressions

3.4.1. Syntax

⟨relational operator⟩ ::= $<|\leqq|=|\geqq|>|\neq$
⟨relation⟩ ::= ⟨simple arithmetic expression⟩
⟨relational operator⟩⟨simple arithmetic expression⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩|⟨variable⟩|
⟨function designator⟩|⟨relation⟩|(⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩|$\neg$⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|
⟨Boolean factor⟩$\wedge$⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩|⟨Boolean term⟩
$\vee$⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩|⟨implication⟩$\supset$⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩|
⟨simple Boolean⟩$\equiv$⟨implication⟩
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|
⟨if clause⟩⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

3.4.2. Examples

$x= -2$
$Y>V \vee z<q$
$a+b > -5 \wedge z-d > q\uparrow 2$
$p\wedge q \vee x\neq y$
$g\equiv \neg a\wedge b\wedge \neg c\vee d\vee e\supset \neg f$
if $k<1$ **then** $s>w$ **else** $h\leqq c$
if if if a **then** b **else** c **then** d **else** f **then** g **else** $h<k$

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean primaries must be declared **Boolean** (cf. section 5.1. Type Declarations and section 5.4.4. Values of Function Designators).

3.4.5. The operators

Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**.

The meaning of the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table.

b1	**false**	**false**	**true**	**true**
b2	**false**	**true**	**false**	**true**
$\neg$b1	**true**	**true**	**false**	**false**
b1$\wedge$b2	**false**	**false**	**false**	**true**
b1$\vee$b2	**false**	**true**	**true**	**true**
b1$\supset$b2	**true**	**true**	**false**	**true**
b1$\equiv$b2	**true**	**false**	**false**	**true**

3.4.6. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first:	arithmetic expressions according to section 3.3.5.
second:	$<\leqq=\geqq>\neq$
third:	$\neg$
fourth:	$\wedge$
fifth:	$\vee$
sixth:	$\supset$
seventh:	$\equiv$

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. Designational Expressions

3.5.1. Syntax

⟨label⟩ ::= ⟨identifier⟩|⟨unsigned integer⟩
⟨switch identifier⟩ ::= ⟨identifier⟩
⟨switch designator⟩ ::= ⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩ ::= ⟨label⟩|⟨switch designator⟩|
(⟨designational expression⟩)
⟨designational expression⟩ ::= ⟨simple designational expression⟩|
⟨if clause⟩⟨simple designational expression⟩ **else**
⟨designational expression⟩

3.5.2. Examples

17
$p9$
$Choose[n-1]$
$Town$[**if** $y<0$ **then** N **else** $N+1$]
if $Ab<c$ **then** 17 **else** q[**if** $w\leqq 0$ **then** 2 **else** n]

3.5.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3.

Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, . . . , n, where n is the number of entries in the switch list.

3.5.5. Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound Statements and Blocks

4.1.1. Syntax

```
⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩|
    ⟨go to statement⟩|⟨dummy statement⟩|⟨procedure statement⟩
⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩|⟨label⟩:
    ⟨basic statement⟩
⟨unconditional statement⟩ ::= ⟨basic statement⟩|
    ⟨compound statement⟩|⟨block⟩
⟨statement⟩ ::= ⟨unconditional statement⟩|
    ⟨conditional statement⟩|⟨for statement⟩
⟨compound tail⟩ ::= ⟨statement⟩ end |⟨statement⟩ ;
    ⟨compound tail⟩
⟨block head⟩ ::= begin⟨declaration⟩|⟨block head⟩ ;
    ⟨declaration⟩
⟨unlabelled compound⟩ ::= begin ⟨compound tail⟩
⟨unlabelled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩
⟨compound statement⟩ ::= ⟨unlabelled compound⟩|
    ⟨label⟩:⟨compound statement⟩
⟨block⟩ ::= ⟨unlabelled block⟩|⟨label⟩:⟨block⟩
⟨program⟩ ::= ⟨block⟩|⟨compound statement⟩
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

```
L: L: ... begin S ; S ; ... S ; S end
```

Block:

```
L: L: ... begin D ; D ; .. D ; S ; S ; ...S ;
   S end
```

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p+q
go to Naples
START: CONTINUE: W := 7.993
```

Compound statement:

```
begin x := 0 ; for y := 1 step 1 until n do
        x := x+A[y] ;
      if x>q then go to STOP else if x>w−2 then
        go to S ;
      Aw: St: W := x+bob end
```

Block:

```
Q: begin integer i, k ; real w ;
     for i := 1 step 1 until m do
     for k := i+1 step 1 until m do
     begin w := A[i, k] ;
           A[i, k] := A[k, i] ;
           A[k, i] := w end for i and k
   end block Q
```

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

```
⟨left part⟩ ::= ⟨variable⟩ := |⟨procedure identifier⟩ := .
⟨left part list⟩ ::= ⟨left part⟩|⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩|
    ⟨left part list⟩⟨Boolean expression⟩
```

4.2.2. Examples

```
s := p[0] := n := n+1+s
n := n+1
A := B/C−v−q×S
S[v,k+2] := 3−arctan(s×zeta)
V := Q>Y∧Z
```

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean**, the expression must likewise be **Boolean.** If the type is **real** or **integer**, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type, the transfer function is understood to yield a result equivalent to

entier(E+0 5)

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3. Go To Statements

4.3.1. Syntax

⟨go to statement⟩ ::= **go to** ⟨designational expression⟩

4.3.2. Examples

```
go to 8
go to exit [n+1]
go to Town[if y<0 then N else N+1]
go to if Ab<c then 17 else q[if w<0 then 2 else n]
```

4.3.3. Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5. Go to an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy Statements

4.4.1. Syntax

⟨dummy statement⟩ ::= ⟨empty⟩

4.4.2. Examples

```
L:
begin ... ; John: end
```

4.4.3. Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5. Conditional Statements

4.5.1. Syntax

⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨unconditional statement⟩ ::= ⟨basic statement⟩|
  ⟨compound statement⟩|⟨block⟩
⟨if statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩ ::= ⟨if statement⟩|⟨if statement⟩ **else**
  ⟨statement⟩|⟨if clause⟩⟨for statement⟩|
  ⟨label⟩ : ⟨conditional statement⟩

4.5.2. Examples

```
if x>0 then n := n+1
if v>u then V: q := n+m else go to R
if s<0∨P≦Q then AA: begin if q<v then a := v/s
                else y := 2×a end
            else if v>s then a := v−q else if v>s−1
              then go to S
```

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 **then** S1 **else if** B2 **then** S2 **else** S3 ; S4

and

if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3 ; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e. the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

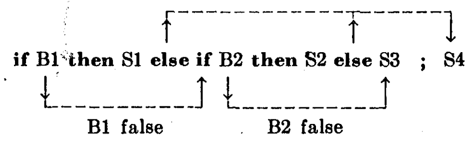
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. For Statements

4.6.1. Syntax

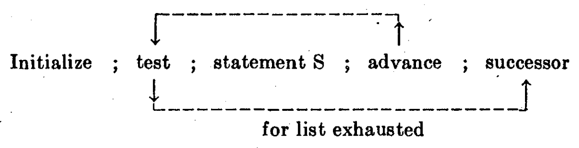
⟨for list element⟩ ::= ⟨arithmetic expression⟩|
  ⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until**
  ⟨arithmetic expression⟩|⟨arithmetic expression⟩ **while**
  ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩|⟨for list⟩ , ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩|
  ⟨label⟩: ⟨for statement⟩

4.6.2. Examples

```
for q := 1 step s until n do A[q] := B[q]
for k := 1, V1×2 while V1<N do
  for j := I+G, L, 1 step 1 until N, C+D do
      A[k,j] := B[k,j]
```

4.6.3. Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element of the form A **step** B **until** C, where A, B, and C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional Algol statements as follows:

```
    V := A  ;
L1: if (V−C)× sign(B)>0 then go to element exhausted;
    statement S  ;
    V := V+B  ;
    go to L1  ;
```

where V is the controlled variable of the for clause and *element exhausted* points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional Algol statements as follows:

```
L3: V := E  ;
    if ¬F then go to element exhausted  ;
    Statement S  ;
    go to L3  ;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. Procedure Statements

4.7.1. Syntax

⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩|
  ⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩

⟨parameter delimiter⟩ ::= ,|)⟨letter string⟩:(
⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
    ⟨actual parameter list⟩⟨parameter delimiter⟩
    ⟨actual parameter⟩
⟨actual parameter part⟩ ::= ⟨empty⟩|
    (⟨actual parameter list⟩)
⟨procedure statement⟩ ::= ⟨procedure identifier⟩
    ⟨actual parameter part⟩

4.7.2. Examples

Spur (*A*)Order: (7)Result to: (*V*)
Transpose (*W*,*v*+1)
Absmax(*A*,*N*,*M*,*Yy*,*I*,*K*)
Innerproduct(*A*[*t*,*P*,*u*],*B*[*P*],10,*P*,*Y*)

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator **own**. This has the following effect: upon a re-entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

⟨declaration⟩ ::= ⟨type declaration⟩|⟨array declaration⟩|
  ⟨switch declaration⟩|⟨procedure declaration⟩

5.1. Type Declarations

5.1.1. Syntax

⟨type list⟩ ::= ⟨simple variable⟩|
  ⟨simple variable⟩ , ⟨type list⟩
⟨type⟩ ::= **real** | **integer** | **Boolean**
⟨local or own type⟩ ::= ⟨type⟩|**own** ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩

5.1.2. Examples

integer *p,q,s*
own Boolean *Acryl,n*

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. Array Declarations

5.2.1. Syntax

⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩|⟨bound pair list⟩,⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩]|
  ⟨array identifier⟩,⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩|⟨array list⟩,⟨array segment⟩
⟨array declaration⟩ ::= **array** ⟨array list⟩|⟨local or own type⟩
  array ⟨array list⟩

5.2.2. Examples

array *a, b, c*[7:*n*,2:*m*], *s*[−2:10]
own integer array *A*[**if** *c*<0 **then** 2 **else** 1:20]
real array *q*[−7:−1]

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch Declarations

5.3.1. Syntax

```
⟨switch list⟩ ::= ⟨designational expression⟩|
    ⟨switch list⟩,⟨designational expression⟩
⟨switch declaration⟩ ::= switch ⟨switch identifier⟩:= ⟨switch list⟩
```

5.3.2. Examples

```
switch S := S1,S2,Q[m], if v>−5 then S3 else S4
switch Q := p1,w
```

5.3.3. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ... , obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. Procedure Declarations

5.4.1. Syntax

```
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩|
        ⟨formal parameter list⟩⟨parameter delimiter⟩
        ⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩ ::= ⟨identifier⟩|⟨identifier list⟩,⟨identifier⟩
⟨value part⟩ ::= value⟨identifier list⟩ ; |⟨empty⟩
⟨specifier⟩ ::= string|⟨type⟩|array|⟨type⟩array|label|switch|
    procedure|⟨type⟩procedure
⟨specification part⟩ ::= ⟨empty⟩|⟨specifier⟩⟨identifier list⟩ ; |
    ⟨specification part⟩⟨specifier⟩⟨identifier list⟩ ;
⟨procedure heading⟩ ::= ⟨procedure identifier⟩
    ⟨formal parameter part⟩ ; ⟨value part⟩⟨specification part⟩
⟨procedure body⟩ ::= ⟨statement⟩|⟨code⟩
⟨procedure declaration⟩ ::=
    procedure ⟨procedure heading⟩⟨procedure body⟩|
    ⟨type⟩ procedure ⟨procedure heading⟩⟨procedure body⟩
```

5.4.2. Examples (see also the examples at the end of the report)

```
procedure Spur(a)Order:(n)Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s+a[k,k]
end

procedure Transpose(a)Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
     for k := 1+i step 1 until n do
     begin w := a[i,k] ;
           a[i,k] := a[k,i] ;
           a[k,i] := w
     end
end Transpose

integer procedure Step (u) ; real u ;
Step := if 0≦u∧u≦1 then 1 else 0

procedure Absmax(a)size:(n,m)Result:(y)Subscripts:(i,k);
comment The absolute greatest element of the matrix a,
     of size n by m is transferred to y, and the subscripts of this
     element to i and k ;
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q])>y then begin y := abs(a[p,q]) ; i := p ;
  k := q
end end Absmax

procedure Innerproduct(a,b)Order:(k,p)Result:(y) ; value k ;
integer k,p ; real y,a,b ;
begin real s ;
s := 0 ;
for p := 1 step 1 until k do s := s+a×b ;
y := s
end Innerproduct
```

5.4.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language

Examples of Procedure Declarations:

EXAMPLE 1.

```
procedure euler (fct, sum, eps, tim)  ; value eps, tim  ;
integer tim  ; real procedure fct  ; real sum, eps  ;
comment euler computes the sum of fct(i) for i from zero up to
infinity by means of a suitabley refined euler transformation. The
summation is stopped as soon as tim times in succession the abso-
lute value of the terms of the transformed series are found to be
less than eps. Hence, one should provide a function fct with one
integer argument, an upper bound eps, and an integer tim. The
output is the sum sum. euler is particularly efficient in the case
of a slowly convergent or divergent alternating series  ;
begin integer i, k, n, t  ; array m[0:15]  ; real mn, mp, ds  ;
i := n := t := 0  ; m[0] := fct(0)  ; sum := m[0]/2  ;
nextterm: i := i+1  ; mn := fct(i)  ;
        for k := 0 step 1 until n do
            begin mp := (mn+m[k])/2  ; m[k] := mn  ;
              mn := mp end means  ;
        if (abs(mn)<abs(m[n]))∧(n<15) then
            begin ds := mn/2  ; n := n+1  ; m[n] :=
              mn end accept
        else ds := mn  ;
        sum := sum + ds  ;
        if abs(ds)<eps then t := t+1 else t := 0  ;
        if t<tim then go to nextterm
end euler
```

EXAMPLE 2.[8]

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi)  ; value x,y  ;
integer n  ; Boolean fi  ; real x,eps,eta,xE  ; array
y,yE  ; procedure FKT  ;
comment: RK integrates the system yk'=fk(x,y1 ,y2 , ... , yn)
(k=1,2, ... ,n) of differential equations with the method of Runge-
Kutta with automatic search for appropriate length of integration
step. Parameters are: The initial values x and y[k] for x and the un-
known functions yk(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e.
the set of functions fk . The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre-
sents the solution at x=xE. The Boolean variable fi, which must
always be given the value true for an isolated or first entry into
RK. If however the functions y must be available at several mesh-
points x0 , x1 , ... , xn , then the procedure must be called repeat-
edly (with x=xk , xE=xk+1, for k=0, 1, ... , n-1) and then the
later calls may occur with fi=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=fk(x,y[1], y[2], ... , y[n])
for x and the actual y's. A procedure comp enters as a nonlocal
identifier  ;
begin
  array z,y1,y2,y3[1:n]  ; real x1,x2,x3,H  ; Boolean out  ;
  integer k,j  ; own real s,Hs  ;
  procedure RK1ST(x,y,h,xe,ye)  ; real x,h,xe  ; array
            y,ye  ;
    comment: RK1ST integrates one single RUNGE-KUTTA
            with initial values x,y[k] which yields the output
            parameters xe=x+h and ye[k], the latter being the
            solution at xe. Important: the parameters n, FKT, z
            enter RK1ST as nonlocal entities  ;
    begin
      array w[1:n], a[1:5]  ; integer k,j  ;
      a[1] := a[2] := a[5] := h/2  ; a[3] := a[4] := h  ;
      xe := x  ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k]  ;
      for j := 1 step 1 until 4 do
      begin
        FKT(xe,w,n,z)  ;
        xe := x+a[j]  ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k]+a[j]×z[k]  ;
          ye[k] := ye[k] + a[j+1]×z[k]/3
```

[8] This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step-by-step integration of differential equations in an automatic computing machine, [*Proc. Camb. Phil. Soc.* *47* (1951), 96]; and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, [*Fysiograf. Sällsk. Lund, Förhd.* *20*, 11 (1950), 136–152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

```
        end k
      end ;
    end RK1ST   ;
Begin of program:
    if fi then begin H := xE−x   ;  s := 0 end else H := Hs   ;
    out := false   ;
AA: if (x+2.01×H−xE>0)≡(H>0) then
    begin Hs := H   ; out := true   ; H := (xE−x)/2
      end if   ;
    RK1ST (x,y,2×H,x1,y1)   ;
BB: RK1ST (x,y,H,x2,y2)   ; RK1ST(x2,y2,H,x3,y3)   ;
    for k := 1 step 1 until n do
         if comp(y1[k],y3[k],eta)>eps then go to CC   ;
    comment: comp(a,bc,) is a function designator, the value
      of which is the absolute value of the difference of the
      mantissae of a and b, after the exponents of these quan-
      tities have been made equal to the largest of the exponents
      of the originally given parameters a,b,c   ;
    x := x3   ; if out then go to DD   ;
    for k := 1 step 1 until n do y[k] := y3[k]   ;
    if s=5 then begin s := 0   ; H := 2×H end if   ;
    s := s+1   ; go to AA   ;
CC: H := 0.5×H   ; out := false   ; x1 := x2   ;
    for k := 1 step 1 until n do y1[k] := y2[k]   ;
    go to BB   ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words [in typewritten copy; boldface in printed copy—Ed.] have been collected at the beginning.

The examples have been ignored in compiling the index.

END OF THE REPORT

APPENDIX 2

CURRENT SYSTEM LIMITS

May 1, 1967

The following are a list of limits on the numbers of objects available in the system:

(a) The maximum number of distinct identifiers and labels allowable is 100 where print names of 6 characters or less count one and print names of 7 or more characters count 1 for the first six and 1 for each 4 or fraction of 4 characters. Note that any 2 identifiers which have the same first six characters may be treated as the same name (including reserved words). This restriction does not affect the internal working of the program. It means only that when an identifier overflows and the table is printed, what is printed is unpredictable.

*(b) The maximum number of declared objects (variables, arrays, etc.) plus block entries is 300.

(c) The maximum number of nested dynamic blocks is 180.

(d) The maximum number of dynamically defined (e.g., by recursion) FORM and SYMBOL variables is 832.

(e) The maximum number of words of code produced by the compiler is /21000.

(f) The maximum number of words for variables and array storage is /11600.

(g) Available space is constructed from the unused part of (e) and (f). This gives roughly 6800 cells for small programs.

*(h) The maximum number of procedure declarations and labels at one

level is 24.

For a rough estimate, each element of a list and operand or operator of a formula takes up two words of available space.

* It is possible to extend the maximums in these cases. See the user consultant.

APENDIX 3

DEBUG SNAPSHOTS

The following is a list of snapshots which may be inserted between lines of a Formula Algol program. They provide special commands to the compiler for printing, corrections, and debugging. The Format is "SN" in columns 1 and 2, the name of the snapshot starting in column 10, and two optional parameters in columns 15 and 25. Teletype tabs will give the correct columns. Most of them have effects at compile time; the ones which don't are so indicated.

In the following explanations whenever a snapshot may have a parameter of either 0 or 1, it will be denoted "0,1". It is to be understood that for all these snapshots, the 1 turns on a certain action and the 0 turns it off. Only the action will be described.

Some of these snapshots require a more detailed knowledge of the system. In these cases see [4] or the user consultant.

SN ␣AND		The And system is entered at compile time.
SN AND		The And system is entered at run time.
SN BKPT	0,1	At the end of each line a transfer to a closed subroutine is compiled. At routine, this subroutine prints the location of the line of code to which control has arrived. (It is, in effect, a logical trace of the program's execution.)
SN CDLC	0,1	At the end of each line a command is compiled to load the current location of compiled code into an index register. This feature is normally on.
SN $ CMPL	<VAL>	<VAL> is compiled as a machine command

APPENDIX 3 (continued)

		directly into the current location for compiled code.
SN CODE	0,1	Code is printed as it is compiled.
SN COR	<LOC> <VAL>	This can be used to change the contents of locations at compile time. First <LOC>, its contents, and <VAL> are printed. The VAL replaces the contents of <LOC>.
SN DEES		This prints out a series of critical entry points of the compiler.
SN DUMP		This causes the compiled code and the generated abcons to be printed after the compilation of the program and before it is run.
SN ENTR	0,1	A trace of all table entries is printed.
SN EXEC	0,1	This prints a trace of the calls on the semantic routines with parameters.
SN IXRS		This prints the index registers /30-/77 at compile time.
SN LINE	<NUM>	This upspaces <NUM> lines at compile time.
SN LOOK	0,1	A trace of all table look-ups is printed.
SN PAGE		The printer is upspaced to the next page.
SN Q1		This allows the action of SN DUMP to be printed on TTY.
SN REMO	0,1	The program prints on the teletype.

APPENDIX 3 (continued)

			Program output will print if REMO is 1 at the end of compilation.
SN	RCOR	<LOC> <VAL>	At run time <VAL> replaces the contents of <LOC>.
SN	RTRC	<NUM> <LOC>	This has the same effect as SN TRAC, except at run-time.
SN	-RUN		The program will be terminated after compilation.
SN	SCAN	0,1	Characters of the input string are printed as they are read by subscan.
SN	STAC		At compile time, the semantic stack is printed.
SN	STOP		Halts compilation immediately.
SN	TRAC	<NUM> <LOC>	At compile time, commands flags are put on <NUM> words starting at location <LOC>. When these words are executed Monitor trace routines will print them.
SN	==	0,1	A trace of the syntax analyzer is printed. When an attempt is made to match a production, the top of the stack and the production are printed.

APPENDIX 4

ERROR MESSAGES

There are three kinds of errors in Formula Algol: Syntax errors, semantic errors, and run errors. The first two kinds of errors occur at compile time, and the third at run time. Some of these messages require a more detailed knowledge of the system. In these cases see [4] or the User Consultant.

SYNTAX ERRORS

These are of the form

ERROR XXX

0	Program does not start with 'BEGIN'
1	Statement does not begin with legal character
2	Statement starts with identifier not followed by legal character
3	First character of an expression expected but not found
4	Expression formed but not followed by legal character
5	']' is not preceded by a legal construct
6	Array element not found in legal context
7	':' not preceded by a legal construct
8	'←' not preceded by a legal construct
9	')' not preceded by a legal construct
10	',' not preceded by a legal construct
11	'THEN' not preceded by a legal construct
12	'ELSE' not preceded by a legal construct
13	Illegal statement construction
14	Impossible error, system error
17	'STEP' not preceded by a legal construct

18	'UNTIL' not preceded by a legal construction
19	'WHILE' not preceded by a legal construction
20	'DO' not preceded by a legal construction
21	'GO' not followed by a legal construction
22	'GO TO IF...THEN...' not followed by 'ELSE'
24	Obscure error in GO TO statement
25	'|→' not in stack after scanning 'BEGIN'
28	Too many 'END's within a procedure
38	Illegal construction within an IF...THEN...statement
39	More than one subscript in a switch call
42	Array declaration does not contain bounds expression
44	System error in GO TO statement
62	Attempt to 'ALTER' a non-symbol
75	'PRINT' not followed by '('
76	Function designator not followed by legal character
77	'.' not followed by legal character
78	Class operator not formed correctly
80	A value of 'OPERATOR' was not an operator
81	Improper description list construction
85	Operator expected and not found
98	'|→' not in stack at beginning of statement
99	System error
100	Illegal operator or control character scanned
101	ABCON table full
102	Number incorrectly formed (while scanning '.')
103	Number incorrectly formed (while scanning '=')
104	Impossible error, system error

105	Illegal bar variable
106	Illegal SY card
108	Impossible error, system error
109	An insertion locator was expected but not found
110	An expression has been found in an illegal context
111	A selector was expected but not found
112	A selector is not followed by 'PF'
113	/ not followed by [
115	Improper 'INDEX' construction
116	Improper 'PARALLEL FOR' construction
117	DOT not followed by identifier in text
118	Class Name improperly formed
144	Variable declaration does not terminate properly or '[' missing
144	In array declaration
145	Array declaration does not terminate properly
163	Procedure head is incorrectly formed
164	Value or specifier part is incorrectly formed
171	Specifier list not initiated properly
174	Declaration does not begin with a legal construction
190	Identifier not found in identifier list
194	')' missing in formal parameter list
195	Value list not terminated properly
196	Specifier list not terminated properly
200	Formal parameter list for EVAL does not contain all identifiers
201	EVAL statement not formed correctly
250	Switch declaration improperly initiated
251	Missing delimiter in switch declaration
999	Impossible error, system error

SEMANTIC ERRORS

These are of the form

FAULT XXX

2	Procedure not declared as such
5	An identifier in a value list is not a formal parameter
6	An identifier in a specifier list is not a formal parameter
7	An identifier is not declared or
7	A procedure is used where a function is expected or
7	An array identifier is used where a simple variable is expected or
7	A switch identifier is used where a simple variable is expected
12	An identifier as an actual parameter has not been declared
15	In 'GO TO S[...]', S is not a switch
16	In an array access the identifier is not an array
20	Function has not yet been declared
21	Function designator not declared
22	Identifier of a class operator is not a variable
27	Boolean expression expected in 'WHILE' clause, and not found
30	In 'IF B THEN....' B is not of type Boolean
44	Switch identifier is used without parameter
47	Expression in ordinal selector is not of type integer
59	Improper editing statement construction
61	System error
63	Attempt to apply selector to non-symbol
69	A value of 'OPERATOR' is not an operator
70	In 'EVAL F', F is not a formula or symbol
72	In 'EVAL(...)F(...)', F is not formula or symbol

75 A class operator is not a symbol

76 System error processing extractor which is array element

77 System error in class operators

78 Attempt to erase description list of non-symbol

83 System error in pattern construction with types as primaries

85 In'F = = P' or in 'F>>P' F is not a formula

87 A label in a pattern is not of type form

88 In 'IF B THEN...' B is not Boolean or formula

91 A label is used twice in the same block

94 In a DOT array the identifier is not an array

97 Expression in < > is not a symbol

98 The second parameter of 'DERV' is not a formula

99 System error in print routine

100 In a binary arithmetic expression one of the operands is of illegal type

103 Attempt to add local description list to non-symbol

105 In a binary Boolean expression one of the operands is of illegal type

106 Attempt to access non-symbolic attribute

107 Parameter of a function designator is not numeric or formula

108 Attempt to access description list of non-symbol or non-formula

109 Improper value entry construction

112 Attempt to store into illegal entity or legal entity of wrong type

116 '¬' is not followed by Boolean or logic expression

155 Boolean procedure or pattern list expected and not found

175 Attempt to construct non-symbolic attribute

176 Attempt to store list or do value entry with non-symbol

179	Value of index is not declared integer
183	Attempt to test non-symbol against symbolic pattern
184	Expression following $ is not of type integer
186	Non-symbolic label in list pattern
189	Identifier in description list expression is not formula
190	Impossible error, system error
191	An identifier is not declared
192	Form or symbol variable expected and not found
198	Designational expression is used as actual parameter
203	Attempt to count non-symbol
213	Non-symbol in symbolic 'FOR' statement
214	Argument of 'ATTRIBUTES OF' other than symbol
229	Expression preceding ordinal selector is not of type integer
230	Argument of ERADL other than symbol
235	Second parameter to AMONG is not of type symbol
239	Parameter of 'EMPTY' is not a symbol
315	Switch not declared
391	Obscure error in procedure calls
512	Attempt to store non-numeric expression into a numeric variable
612	Attempt to store non-Boolean expression into Boolean variable
712	Attempt to store into a constant
912	System error
990	Impossible error, system error
998	System error
999	System error in 'STEP' statement
4L01	Improper left side of DOT assignment

RUN ERRORS

Run Errors in Formula Manipulation

These are of the form:

```
RUN ERROR  NNN  AFTER LOCATION  XXXXX
LLLLL
```

where NNN refers to the list below, XXXXX indicates the line in which the error occurred and LLLLL is the location of the error routine.

* 1 Attempt to eval an expression containing $\rightarrow$, $\cdot\!\rightarrow$, :, or $|\;|$.

6 Attempt to eval an expression containing +, -, x, /, or $\uparrow$ in which one of the operands is neither a formula nor a number.

10 In eval $\sim x$, x is not logic, boolean, or formula.

* 20 Error when printing, a formula.

21 In eval $x \wedge y$ or $x \vee y$ one of the operands is not logic, boolean, or formula.

22 In eval $x \wedge y$ or $x \vee y$, there is a mixture of types logic and boolean.

25 Recursion stack overflow.

27 Run-time symbol table overflow.

* 30 Error when printing a chain.

* 31 Attempt to find an attribute on an ill-formed chain.

37 Too many subscripts in an array element.

38 Subscript in an array element is too small.

39 Subscript in an array element is too large.

40 Not enough subscripts in an array element.

46 Too many block entries.

48 Subscript in a switch designator is out of bounds.

50 Available space is empty.

56 In derv(f,x), f is not a number or a formula.

57 A boolean data term was expected and not found.

63 Obscure error when storing a chain into a symbol.

* 79 In f . ↓ s, s has no contents attribute.

82 In f . ↓ s, s has a parallel production within a parallel production.

83 In f . ↓ s, s has a formula which is not a production.

85 A malformed formula (system error), or

A class operator encountered within a formula to which a production is to be applied, or

In a dot array (production), a subscript (parameter) is not of type form.

100 Attempt to eval ln(-infi) or sqrt(-infi).

141 In EVAL (x <r> y) where r is >,<,¬<, or ¬>, one of the operands is either undefined, a symbol chain, or of type Boolean.

* 182 Variable of interpretive store has undefined type.

183 Interpretive store of undefined mixture of variable types, or interpretive store into a symbol is not implemented.

271 In eval of A.[s1, ..., sn], some subscript si is not a formula or a number.

325 In eval . if B then ..., B is neither a boolean or a formula.

600 In F = = P, F is the pattern of( ).

601 In F = = P, F is a symbol.

602 Obscure error in F = = P, probably an attempt to test a pattern against another pattern.

603 In F = = P, P has a class operator which has no attribute 'operator'.

701	Attempt to compute 0 ↑ -number.
702	Attempt to compute X ↑ A where x<] and A is not an integer.
703	Attempt to compute X ↑ A where A*ln(x) > 160.117.
711	Attempt to compute ln(X) where X < 0.
721	Attempt to compute EψP(x) where ψ is out of range.
731	Attempt to compute sin(X) where X is out of range.
751	Attempt to compute sqrt(X) where X < 0.
5501	Attempt to eval the pattern 'of(B)'.
5502	Attempt to compute replace(F) where F>> A:atom.
6702	A class operator or extractor encountered in a formula to which a production is to be applied.
*7701	Attempt to create A |[T]| B where T has no contents attribute.
7702	Attempt to create A |[T]| B where [T] is empty.
7703	Attempt to create A |[T]| B where [T] is unary.
7704	Attempt to create |[T]| B where [T] is binary.
9009	Attempt to EVAL (0↑0).
9011	Attempt to EVAL (ANY/0).

* denotes a system error.

Run Errors in Symbol Manipulation

The following messages are printed:

Recoverable Errors

Not enough chain operands cf. p. 1

Unless store into unused chain

Attempt to store into open chain

Attempt to get interior of empty clsd ch

Attempt to discard nil

First element of plural list uncarried

Attempt to select non-existant referent

Class name undefined

Non-Recoverable Errors

Parity of chainacc destroyed

Negative chainacc

Attempt to store in non-symbol

Malformed chain

Chainacc exceeded

Plural list used where symbol needed

Attempt at VR from non-symbol

Attempt at VR without attribute

Empty list used where symbol needed

Attempt to generate ATRS. of non-chain

System Error

Illegal selector

Non-primitive for ID. routine

For attempts to generate non-list

P-for control variable non-symbol

No.contrl.var ¬ = no. of lists in P-for

Malformed pattern

Non-numeric data term used as number

Available space exhausted

Improper symbol array access

Value of symbol array el.no exst.

Illegal transfer function

Run Errors in Recursion

These are of the form:

```
XXXX   MM:SS:ss
<octal dump of index registers /50,...,/77 >
```

XXXX	is the name of the error.
MM:SS:ss	is the running time in sixtieths of a second.
<octal dump>	indicates the state of the program.

*	CLOB	System error indicating that the historian has been clobbered.
	STOR	Variable stack overflow.
	HIST	Historian stack overflow.
*	PROC	Obscure error related to procedure names as actual parameters.
*	LINK	Premature or illegal attempt to leave a codepiece.
	LABL	Attempt to goto an undefined label or to call an undefined procedure or switch.
	~64K	Request for extra memory was refused

* indicates a system error.

APPENDIX 5

INPUT - OUTPUT

Formual Algol has no read statements.

At the present time, Formula Algol contains a primitive print statement of the form PRINT(X), where X is a list of any of the following possible objects:

(a) The name of any declared variable, in which case the value of that variable will be printed.

(b) Any arithmetic, Boolean or Formula expression, in which case the value of the expression will be printed.

(c) Any symbolic expression provided a switch is set as indicated below.

For example:

```
FORM F, G; REAL A,B; BOOLEAN C; SYMBOL S;
LOGIC L; HALF H;
    F ← F + G; A ← 3.5; B ← 2 × A; C ← B < A ;
    S ← [F, A]; L ← 10; H ← 2.8;
    PRINT(F,G,A,B,C,S,L,H, 111, G+A);
```

This causes the following to be printed:

```
F + G
G
.35000000000₁₀ +01
.70000000000₁₀ +01
FALS
```

```
/[CONT: F+G,(.35000000000₁₀ +01)][NAME:S]
00000000012
.28000000000₁₀ +01
111
G+ (.35000000000₁₀ +01)
```

Lists may be printed in three styles: style 0, style 1, and style 2. Style 0 is in the system to begin with and causes description lists to be printed. Style 1 prints lists and sublists with square brackets [,] and commas separating the elements, each sublist being delimited by a pair of square brackets. Style 2 prints lists without square brackets and commas by concatenating the elements directly into the print line.

For example:

```
 SYMBOL S, ADJ, EC, TIVE, A,B,C, COLOR, APPLE,RED;
 APPLE ←/[COLOR: RED];
S ←[A,A,[B,B, [C,C,C],B],A];
A ←[ADJ,EC,TIVE];
```

In Style 0 the statement PRINT(APPLE, S, A) gives:

```
/[CONT: APPLE][COLOR:RED][NAME:APPLE];
/[CONT: A,A, /[CONT:B,B, /[CONT:C,C,C][NAME:],B][NAME:],
    A][NAME:S]
/[CONT: ADJ, EC, TIVE][NAME:A]
```

In Style 1 the same print statement gives:

```
[APPLE]
[A,A,[B,B,[C,C,C],B],A]
[ADJ, EC, TIVE]
```

In Style 2 the same print statement gives:

```
APPLE

AABBCCCBA

ADJECTIVE
```

Thus, Style 0 is used to print description lists, Style 1 is used to print lists and sublists, and Style 2 is used to print compacted lists. Executing the following snapshot correction changes the style switch.

```
SN     RCOR     55212     1     sets Style to 1

SN     RCOR     55212     2     sets Style to 2

SN     RCOR     55212     0     sets Style to 0
```

This snapshot follows the same conventions as other debug snapshots (see Appendix 3).

APPENDIX 6

SYNTAX INDEX

SYNTAX CLASSES

RESERVED WORDS

APPENDIX 7

COMPLETE EXAMPLES

The attached photocopies of computer output present three ways that Formula Algol can be used to solve an algebraic equation for the single occurrence of the variable X. These three solutions are by Markov Algorithms, by recursion, and by iteration. Formula Algol is well suited to programming this problem because its data structures and source language instructions were chosen to be well adapted to problems in formal algebraic manipulation. It can be seen from the attached programs that the Formula Algol programmer has detailed control over the specification of formula manipulation algorithms and that,at the same time, abbreviation devices, such as the Markov Algorithm, make it convenient to write them. Brief explanations of the three solutions are as follows.

I. MARKOV ALGORITHM SOLUTION

Lines 12 to 29 define a Markov Algorithm which gives the rules of transformation by which equations are to be solved for X. The equation to be solved for X is stored as the value of the variable E in line 30, and line 31 prints both E and E. ↓S the result of applying the Markov Algorithm S to E, which result is the solved equation. In lines 10 and 11, <u>plus</u> and <u>times</u> are defined to be operators with commutative properties so that in lines 14 and 15 commutative instances of A*B and A+B will be considered. Lines 7,8, and 9 define A to be a formula pattern which will match any subexpression of a formula containing an occurrence of X, and B and C to be formula patterns which will match any arbitrary subexpression of a formula. The A's, B's, and C's are used in the construction of the left hand sides of the transformations in the Markov Algorithm and stand for patterns with these properties. On the right hand sides of the transformations the .A's, .B's, and .C's are objects which are replaced by the subexpressions which match the A's, B's, and C's when given transformation applies to an input equation.

II. RECURSIVE SOLUTION

Lines 4, 5, and 9 define patterns A, B, and C with the same properties as in the Markov Algorithm solution. The recursive procedure SOLVE(LHS,RHS) given in lines 8 to 28 analyzes the form of the left hand side of the equation, LHS, which is assumed to contain X, and recursively calls SOLVE with that subexpression of LHS containing X as its new first parameter, and an appropriate inverse expression composed of an appropriate inverse operator applied to RHS and a subexpression of LHS not containing X as its new second parameter. The procedure Answer(E) given in lines 30 to 34 analyzes the input equation E to see which side contains X and passes the side containing X as the left hand side and the side not containing X as the right hand side to SOLVE which delivers the answer to the problem. An equation is assigned to E in line 36 and both E and Answer(E) are printed in line 37. The printed solution is the same as that given in the first and third solutions.

III. ITERATIVE SOLUTION

Lines 6 and 7 define two operator classes OP1 and OP2 consisting respectively of the binary operators to be used in input equations and the unary operators to be used in input equations. An integer variable I is attached to the definition of each operator class as an "Index". In lines 12 and 13 the input equation G is compared with two patterns. The first pattern matches if the left hand side of G contains a binary operator in the class OP1 and the index vari le I is set to contain an integer denoting the ordinal position of this operator in the list of operators given on line 6. Similarly, the second pattern matches if G's left hand side is of the form <unary operator>(<expression>)

and the index I is set to the ordinal position of the unary operator in the list of unary operators in line 7. The integer value of this index I is used in a designational expression containing a switch to transfer control to an appropriate statement to perform the required transformation of the equation. These transformations are given in lines 15 to 27. The iteration is under the control of a FOR-WHILE statement and halts when the equation G has X as its left hand side. The printed solution is the same as that for solutions I and II.

IV. COMPARISON OF THE THREE SOLUTIONS

	Markov Algorithm	Recursion	Iteration
seconds required	5 ± 1	4 ± 1	3 ± 1
cells required	232	471	183
code required	771	826	595

The times given here are not measured as precisely as they should be for a truly useful comparison.

```
A OPER. 1202 24 OCT 66  22:23:34  AND    PAGES:  50  TIME:  3
993    4C052062    C0003011403*           TS01

STATUS MAR. 25,1966: EXPERIMENTAL SYSTEM.
    002:          AL BEGIN
    003:  11002     FORM E,K,M,H,N,P;
    004:  11020     FORM A,B,C,X; SYMBOL PLUS, TIMES, S;
    005:  11037     BOOLEAN PROCEDURE HASX(F); VALUE F; FORM F;
    006:  11051     HASX ← F >> X;
    007:  11060     A←A:OF(HASX);
    008:  11072     B←B:ANY;
    009:  11103     C←C:ANY;
    010:  11114     PLUS←/[OPERATOR:+][COMM: TRUE];
    011:  11144     TIMES←/[OPERATOR:*][COMM: TRUE];
    012:  11174     S ← [
    013:            [
    014:  11177     (A|TIMES|B) = C   →   .A = .C / .B,
    015:  11243     (A|PLUS |B) = C   →   .A = .C - .B,
    016:  11310     A    -    B = C   →   .A = .C + .B,
    017:  11353     B    -    A = C   →   .A = .B - .C,
    018:  11416     A    /    B = C   →   .A = .C * .B,
    019:  11461     B    /    A = C   →   .A = .B / .C,
    020:  11524     A    ↑    B = C   →   .A = .C ↑ (1/.B),
    021:  11574     B    ↑    A = C   →   .A = LN(.C)/LN(.B),
    022:  11643          -    A = C   →   .A = -.C,
    023:  11677       EXP(A)  = C   →   .A = LN(.C),
    024:  11733        LN(A)  = C   →   .A = EXP(.C),
    025:  11767      SQRT(A)  = C   →   .A = .C ↑ 2,
    026:  12026    ARCTAN(A)  = C   →   .A = SIN(.C)/COS(.C),
    027:  12073       SIN(A)  = C   →   .A = ARCTAN(.C/SQRT(1-.C↑2)),
    028:  12152       COS(A)  = C   →   .A = ARCTAN(SQRT(1-.C↑2)/.C),
    029:  12231            X  = C   →   .X = .C ] ];
    030:  12263     E ← K↑2 + LN(M + SIN( (X↑3-K)/(H+4)*M↑5 )↑N - K)*M  =  P;
    031:  12370     PRINT( E, E.↓S );
    032:  12400      PRINT(CELLS);
    033:  12403      END;
0  ERRORS

BEGIN EXECUTION 22:28:53;  ₀06423 AVAILABLE CELLS
K↑2 + LN(M + SIN((X↑3 - K)/(H + 4)*M↑5)↑N - K)*M=P
X=(ARCT((EXP((P - K↑2)/M) + K - M)↑(1/N)/SQRT(1 - (EXP((P
- K↑2)/M) + K - M)↑(1/N)↑2 ))/M↑5*(H + 4) + K)↑(
.333333333333₀+00)
6191

  TIME USED: 00:00:36 PAGES: 3    12404 22:28:59  0 284 3 0 0 0 0 0 46 0
   22:33:28 END
```

```
                                                   ·IIME:  3
993    20050002     00003011403*           TS01                                        16:0
   FORM AL-3-118


STATUS MAR. 25,1966: EXPERIMENTAL SYSTEM.
   11002       BEGIN FORM E,K,M,N,H,P,F,G,X;
   11026       SYMBOL PLUS,TIMES;
   11033       BOOLEAN PROCEDURE HASX(F); VALUE F; FORM F; HASX←F>>X;
   11053       PLUS←/[OPERATOR:+][COMM: TRUE]; TIMES←/[OPERATOR:*][COMM: TRUE];

                 BEGIN
   11132         FORM PROCEDURE SOLVE(LHS,RHS); FORM LHS,RHS;
   11137           BEGIN FORM A,B,C;A←A:OF(HASX);B←B:ANY;C←C:ANY;
   11201           IF LHS == (A|PLUS|B) THEN SOLVE←SOLVE(A,RHS-B);
   11243           IF LHS == (A|TIMES|B) THEN SOLVE←SOLVE(A,RHS/B);
   11305           IF LHS == A-B  THEN SOLVE ← SOLVE(A,RHS+B);
   11345           IF LHS == B-A  THEN SOLVE ← SOLVE(A,B-RHS);
   11405           IF LHS == A/B  THEN SOLVE ← SOLVE(A,RHS*B);
   11445           IF LHS == B/A  THEN SOLVE ← SOLVE(A,B/RHS);
   11505           IF LHS == A↑B  THEN SOLVE ← SOLVE(A,RHS↑(1/B));
   11554           IF LHS == B↑A  THEN SOLVE ← SOLVE(A,LN(RHS)/LN(B));
   11622           IF LHS == -A THEN SOLVE  ← SOLVE(A,-RHS);
   11656           IF LHS == EXP(A) THEN SOLVE ← SOLVE(A,LN(RHS));
   11712           IF LHS == LN(A) THEN SOLVE ← SOLVE(A,EXP(RHS));
   11746           IF LHS == SQRT(A) THEN SOLVE ← SOLVE(A,RHS↑2);
   12005           IF LHS == ARCTAN(A) THEN SOLVE ← SOLVE(A,SIN(RHS)/COS(RHS));
   12053           IF LHS == SIN(A) THEN
   12071                     SOLVE ← SOLVE(A,ARCTAN(RHS/SQRT(1-RHS↑2)));
   12134           IF LHS == COS(A) THEN
   12152                     SOLVE ← SOLVE(A,ARCTAN(SQRT(1-RHS↑2)/RHS));
   12215           IF LHS == X THEN SOLVE ← X = RHS;
   12241           END;

   12244         FORM PROCEDURE ANSWER(E); FORM E;
   12247           BEGIN FORM F,G;
   12256           IF E == G:ANY=F:ANY THEN BEGIN IF F>>X THEN
   12323           ANSWER←SOLVE(F,G) ELSE ANSWER←SOLVE(G,F) END ELSE
   12344           ANSWER←.NOEQUATION; END;

   12352       E ← K↑2 + LN(M + SIN((X↑3-K)/(H+4)*M↑5)↑N-K)*M =P;
   12457       PRINT(E,ANSWER(E)); PRINT(CELLS);
   12472       END; END;
0  ERRORS

BEGIN EXECUTION 16:20:24;  ₀06418 AVAILABLE CELLS
K↑2 + LN(M + SIN((X↑3 - K)/(H + 4)*M↑5)↑N - K)*M=P
X=(ARCT((EXP((P - K↑2)/M) + K - M)↑(1/N)/SQRT(1 - (EXP((P
- K↑2)/M) + K - M)↑(1/N)↑2))/M↑5*(H + 4) + K)↑(1/3)
5947

  TIME USED: 00:00:32 PAGES: 2    12474 16:20:28  0 0 0 0 0 0 0 0 50 0
```

FORM AL-3-119

```
 $=DONE
OUTPUT
  TIME USED: 00:00:42 PAGES: 3     12124 00:41:17  0 0 0 0 0 0 0 0 45 0          \\\\
A OPER. SM02 25 OCT 66  00:40:36  AND    PAGES:  50  TIME:  3
993    40052062    00003011403             TS01                                 00:3

CR09 TS01     80 /  4          32     9456        3

STATUS MAR. 25,1966:EXPERIMENTAL SYSTEM.
                         BEGIN
          11002          FORM G,K,M,H,N,P, A, B,C,X;SYMBOL OP1,OP2;
          11034          INTEGER I; SWITCH L← L1,L2,L3,L4,L5;
          11052          SWITCH Q ← Q1,Q2,Q3,Q4,Q5,Q6,Q7;
          11073          OP1←/[OPERATOR:*,+,-,/,↑][INDEX:I];
          11146          OP2←/[OPERATOR:-,EXP,LN,SQRT,ARCTAN,SIN,COS][INDEX:I];
          11233          G←K↑2 +LN(M+SIN( (X↑3-K)/(H+4)*M↑5)↑N-K)*M=P;

          11340          FOR G ← G WHILE ¬(G == X=ANY ) DO
                           BEGIN
          11363            IF G == (A:ANY|OP1|B:ANY)=C:ANY THEN GO TO L[I];
          11435           IF G == (+|OP2| A:ANY)=C:ANY THEN GO TO Q[I] ;
          11476            PRINT(.NOEQUATION); GO TO CONTINUE;
          11503            L1:G←IF A>>X THEN A=C/B ELSE B=C/A; GO TO CONTINUE;
          11537            L2:G←IF A>>X THEN A=C-B ELSE B=C-A; GO TO CONTINUE;
          11573            L3:G←IF A>>X THEN A=C+B ELSE B=A-C; GO TO CONTINUE;
          11627            L4:G←IF A>>X THEN A=C*B ELSE B=A/C; GO TO CONTINUE;
          11663            L5:G←IF A>>X THEN A=C↑(1/B) ELSE B=LN(C)/LN(A);
          11731                GO TO CONTINUE;
          11733            Q1:G←A=-C; GO TO CONTINUE;
          11745            Q2:G←A=LN(C); GO TO CONTINUE;
          11757            Q3:G←A=EXP(C); GO TO CONTINUE;
          11771            Q4:G←A=C↑2; GO TO CONTINUE;
          12006            Q5:G←A=SIN(C)/COS(C); GO TO CONTINUE;
          12030            Q6:G←A=ARCTAN(C/SQRT(1-C↑2)); GO TO CONTINUE;
          12062            Q7:G←A=ARCTAN(SQRT(1-C↑2)/C); GO TO CONTINUE;
                           CONTINUE:  ;
          12114            END;

          12116          PRINT(G); PRINT(CELLS);
          12123          END;
0  ERRORS

BEGIN EXECUTION 00:41:15;  06529 AVAILABLE CELLS
X=(ARCT((EXP((P - K↑2)/M) + K - M)↑(1/N)/SQRT(1 - (EXP((P
- K↑2)/M) + K - M)↑(1/N)↑2))/M↑5*(H + 4) + K)↑(1/3)
6346

  TIME USED: 00:00:42 PAGES: 3     12124 00:41:17  0 0 0 0 0 0 0 0 45 0
   00:46:21 END
```

APPENDIX 8

CURRENT SYSTEM BUGS

May 1, 1967

The following is a list of constructions which are currently not functioning in Formula Algol:

1. Attempting to access a switch with an index which is out of bounds. Gives a run error instead of returning as defined in Algol 60.
2. Recursive class names.
3. A selector using itself within itself through a class name (i.e., 3RD (|VOWEL|) where the code for VOWEL uses the "nTH" selector).
4. "Own" variables.
5. The "»" predicate will not test for subformulae of subscripts to an array formula or parameters to a procedure formula; schema will, however.
6. "SUBS" in either array, procedure, assignment or conditional formula.
7. A construction of the form:

   F»...OF(B)...

   where B is of the form:

   <u>BOOLEAN</u> <u>PROCEDURE</u> B: <u>FORM</u> X:

   G»... OF(B)....
8. Cannot pass switches as parameters.
9. Real arrays are not stored into properly if the right hand side is only a variable, not an expression.

   e.g. A[I] ← X; does not work (stores logic)

   A[I] ← X+0; works

10. Logic Arrays are always accessed arithmetically.

11. A procedure which has the form of a compound statement is treated as a block in the declaration of labels.

12. Switches may neither be forward referenced nor recursively referenced.

13. Print routine will not print incomplete chain.

    PROCEDURE P(...,L), SYMBOL L;   'L' is called by name

        PRINT(...,[.,L]); ← incomplete chain

    Inasmuch as this is now a nonrecoverable error, caution should be exercised to avoid using this construction.

14. In the EVAL operation, the formulae which are substituted are not evaluated in themselves, but only in combination with the rest of the formula. Thus, if '3+4' is one of the substituted values, it will not be reduced.

15. The identity of atomic formulae does not follow the outlines of block structure. They act as though they were all declared globally.

16. A multiple assignment statement for a description list is not allowed.

17. In a procedure, A ↑ B does not work unless A and B are either not local to any procedures or local to the same procedure.

18. SYMBOL and FORM variables which are formal parameters of a procedure cannot be dotted.

19. The construction

        S ← (if B then C else D) + E

    will not work if C is an arithmetic expression, but D is a number which is to be extracted from a list or formula structure. Reversing C and D fails also.

The construction

```
     FOR I ← 1 STEP 1 UNTIL EVAL F DO S;
```

fails for the same reasons.

REFERENCES

[1] Naur, P. et.al., "Revised Report on Algorithmic Language ALGOL 60," Communications of the ACM, Vol. , p. 1-17, (January 1963).

[2] Perlis, A. J. and Iturriaga, R., "An Extension to ALGOL for Manipulating Formulae," Communications of the ACM, Vol. 7, p. 127, (February 1964).

[3] Fierst, J. W., Ed., Algol-20, A Language Manual, Carnegie Institute of Technology, 1965.

[4] Iturriaga, R., Standish, T. A., Krutar, R. A., Earley, J. C., The Implementation of Formula Algol in FSL, Carnegie Institute of Technology, 1965.

[5] Yngve, V. H., COMIT Programmers Reference Manual, The M.I.T. Press, (September 1961).

[6] Iturriaga, R., Standish, T. A., Krutar, R. A., and Earley, J. C., "Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula Algol Compiler," Proceedings Spring Joint Computer Conference 1966, Spartan Books.

[7] Perlis, A. J., Iturriaga, R., and Standish, T. A., A Definition of Formula Algol, Carnegie Institute of Technology, 1966.