Programming Languages

D. E. KNUTH, Editor

A Contribution to the Development of ALGOL

NIKLAUS WIRTH
*Stanford University**
Stanford, California

AND

C. A. R. HOARE
Elliott Automation Computers Ltd.,
Borehamwood, England

A programming language similar in many respects to ALGOL 60, but incorporating a large number of improvements based on six years' experience with that language, is described in detail. Part I consists of an introduction to the new language and a summary of the changes made to ALGOL 60, together with a discussion of the motives behind the revisions. Part II is a rigorous definition of the proposed language. Part III describes a set of proposed standard procedures to be used with the language, including facilities for input/output.

PART I. GENERAL INTRODUCTION

1. Historical Background

A preliminary version of this report was originally drafted by the first author on an invitation made by IFIP Working Group 2.1 at its meeting in May, 1965 at Princeton. It incorporated a number of opinions and suggestions made at that meeting and in its subcommittees, and it was distributed to members of the Working Group as "Proposal for a Report on a Successor of ALGOL 60" (MR75, Mathematical Centre, Amsterdam, August 1965).

However, at the following meeting of the Group at Grenoble in October, 1965 it was felt that the report did not represent a sufficient advance on ALGOL 60, either in its manner of language definition or in the content of the language itself. The draft therefore no longer had the status of an official Working Document of the Group and by kind permission of the Chairman it was released for wider publication.

At that time the authors agreed to collaborate on revising and supplementing the draft. The main changes were:

(1) verbal improvements and clarifications, many of which were kindly suggested by recipients of the original draft;

(2) additional or altered language features, in particular the replacement of tree structures by records as proposed by the second author;

(3) changes which appeared desirable in the course of designing a simple and efficient implementation of the language;

(4) addition of introductory and explanatory material, and further suggestions for standard procedures, in particular on input/output;

(5) use of a convenient notational facility to abbreviate the description of syntax, as suggested by van Wijngaarden in "Orthogonal Design and Description of a Formal Language" (MR76, Mathematical Centre, Amsterdam, Oct. 1965).

The incorporation of the revisions is not intended to reinstate the report as a candidate for consideration as a successor to ALGOL 60. However, it is believed that its publication will serve three purposes:

(1) To present to a wider public a view of the general direction in which the development of ALGOL is proceeding;

(2) To provide an opportunity for experimental implementation and use of the language, which may be of value in future discussions of language development;

(3) To describe some of the problems encountered in the attempt to extend the language further.

2. Aims of the Language

The design of the language is intended to reflect the outlook and intentions of IFIP Working Group 2.1, and in particular their belief in the value of a common programming language suitable for use by many people in many countries. It also recognizes that such a language should satisfy as far as possible the following criteria:

(1) The language must provide a suitable technique for the programming of digital computers. It must there-

This work was supported by the National Science Foundation (GP 4053 and GP 4298), and it is also published with due acknowledgment to Elliott-Automation Computers Ltd.

* Computer Science Department.

fore be closely oriented toward the capabilities of these machines, and must take into account their inherent limitations. As a result it should be possible to construct a fast, well-structured and reliable translator, translating programs into machine code which makes economic use of the power and capacity of a computer. In addition, the design of the language should act as an encouragement to the programmer to conceive the solution of his problems in terms which will produce effective programs on the computers he is likely to have at his disposal.

(2) The language must serve as a medium of communication between those engaged in problems capable of algorithmic solution. The notational structure of programs expressed in the language should correspond closely with the dynamic structure of the processes they describe. The programmer should be obliged to express himself explicitly clearly and fully, without confusing abbreviations or implicit presuppositions. The perspicuity of programs is believed to be a property of equal benefit to their readers and ultimately to their writers.

(3) The language must present a conceptual framework for teaching, reasoning and research in both theoretical and practical aspects of the science of computation. It must therefore be based on rigourous selection and abstraction of the most fundamental concepts of computational techniques. Its power and flexibility should derive from unifying simplicity, rather than from proliferation of poorly integrated features and facilities. As a consequence, for each purpose there will be exactly one obviously appropriate facility, so that there is minimal scope for erroneous choice and misapplication of facilities, whether due to misunderstanding, inadvertence or inexperience.

(4) The value of a language is increased in proportion to the range of applications in which it may effectively and conveniently be used. It is hoped that the language will find use throughout the field of algebraic and numeric applications, and that its use will begin to spread to non-numeric data processing in areas hitherto the preserve of special purpose languages, for example, the fields of simulation studies, design automation, information retrieval, graph theory, symbol manipulation and linguistic research.

To meet any of these four requirements, it is necessary that the language itself be defined with utmost clarity and rigor. The Report on ALGOL 60 has set a high standard in this respect, and in style and notation its example has been gratefully followed.

3. Summary of New Features

A large part of the language is, of course, taken directly from ALGOL 60. However, in some respects the language has been simplified, and in others extended. The following paragraphs summarize the major changes to ALGOL 60, and relate them to the declared aims of the language.

3.1. DATA TYPES

The range of primitive data types has been extended from three in ALGOL 60 to seven, or rather nine, if the long variants are included. In compensation, certain aspects of the concept of type have been simplified. In particular, the **own** concept has been abandoned as insufficiently useful to justify its position, and as leading to semantic ambiguities in many circumstances.

3.1.1. Numeric Data Types

The type **complex** has been introduced into the language to simplify the specification of algorithms involving complex numbers.

For the types **real** and **complex**, a long variant is provided to deal with calculations or sections of calculations in which the normal precision for floating-point number representation is not sufficient. It is expected that the significance of the representation will be approximately doubled.

No provision is made for specifying the exact required significance of floating-point representation in terms of the number of binary or decimal digits. It is considered most important that the values of primitive types should occupy a small integral number of computer words, so that their processing can be carried out with the maximum efficiency of the equipment available.

3.1.2. Sequences

The concept of a *sequence* occupies a position intermediate between that of an array and of other simple data types. Like single-dimensional arrays, they consist of ordered sequences of elements; however, unlike arrays, the most frequent operations performed on them are not the extraction or insertion of single elements, but rather the processing of whole sequences, or possibly subsequences of them.

Sequences are represented in the language by two new types, **bits** (sequence of binary digits), and **string** (sequence of characters). Operations defined for bit sequences include the logical operations $\neg$, $\wedge$ and $\vee$, and those of shifting left and right.

The most important feature of a bit sequence is that its elements are sufficiently small to occupy only a fraction of a "computer word," i.e. a unit of information which is in some sense natural to the computer. This means that space can be saved by "packing," and efficiency can be gained by operating on such natural units of information. In order that use of such natural units can be made by an implementation, the maximum number of elements in a sequence must be specified, when a variable of that type is declared. Operations defined for string sequences include the catenation operator **cat**.

3.1.3. Type Determination at Compile Time

The language has been designed in such a way that the type and length of the result of evaluating every expression and subexpression can be uniquely determined by a textual scan of the program, so that no type testing is required at run time, except possibly on procedure entry.

3.1.4. Type Conversions

The increase in the number of data types has caused an even greater number of possibilities for type conversion; some of these are intended to be inserted automatically by

the translator, and others have to be specified by the programmer by use of standard transfer functions provided for the purpose.

Automatic insertion of type conversion has been confined to cases where there could be no possible confusion about which conversion is intended: from **integer** to **real**, and **real** to **complex**, but not vice versa. Automatic conversions are also performed from shorter to longer variants of the data types; and in the case of numbers, from long to short as well.

For all other conversions explicit standard procedures must be used. This ensures that the complexity and possible inefficiency of the conversion process is not hidden from the programmer; furthermore, the existence of additional parameters of the procedure, or a choice of procedures, will draw his attention to the fact that there is more than one way of performing the conversion, and he is thereby encouraged to select the alternative which corresponds to his real requirements, rather than rely on a built-in "default" conversion, about which he may have only vague or even mistaken ideas.

3.2. Control of Sequencing

The only changes made to facilities associated with control of sequencing have been made in the direction of simplification and clarification, rather than extension.

3.2.1. Switches and the Case Construction

The switch declaration and the switch designator have been abolished. Their place has been taken by the case construction, applying to both expressions and statements. This construction permits the selection and execution (or evaluation) of one from a list of statements (or expressions); the selection is made in accordance with the value of an integer expression.

The case construction extends the facilities of the Algol conditional to circumstances where the choice is made from more than two alternatives. Like the conditional, it mirrors the dynamic structure of a program more clearly than go to statements and switches, and it eliminates the need for introducing a large number of labels in the program.

3.2.2. Labels

The concept of a label has been simplified so that it merely serves as a link between a goto statement and its destination; it has been stripped of all features suggesting that it is a manipulable object. In particular, designational expressions have been abolished, and labels can no longer be passed as parameters of procedures.

A further simplification is represented by the rule that a goto statement cannot lead from outside into a conditional statement or case statement, as well as iterative statement.

The Algol 60 integer labels have been eliminated.

3.2.3. Iterative Statements

The purpose of iterative statements is to enable the programmer to specify iterations in a simple and perspicuous manner, and to protect himself from the unexpected effects of some subtle or careless error. They also signalize to the translator that this is a special case, susceptible of simple optimization.

It is notorious that the Algol 60 for statement fails to satisfy any of these requirements, and therefore a drastic simplification has been made. The use of iterative statements has been confined to the really simple and common cases, rather than extended to cover more complex requirements, which can be more flexibly and perspicuously dealt with by explicit program instructions using labels.

The most general and powerful iterative statement, capable of covering all requirements, is that which indicates that a statement is to be executed repeatedly while a given condition remains true. The only alternative type of iterative statement allows a formal counter to take successive values in a finite arithmetic progression on each execution of the statement. No explicit assignments can be made to this counter, which is implicitly declared as local to the iterative statement.

3.3. Procedures and Parameters

A few minor changes have been made to the procedure concept of Algol 60, mainly in the interests of clarification and efficiency of implementation.

3.3.1. Value and Result Parameters

As in Algol 60, the meaning of parameters is explained in terms of the "copy rule," which prescribes the literal replacement of the formal parameter by the actual parameter. As a counterpart to the "value parameter," which is a convenient abbreviation for the frequent case where the formal parameter can be considered as a variable local to the procedure and initialized to the value of the actual parameter, a "result parameter" has been introduced. Again, the formal parameter is considered as a local variable, whose value is assigned to the corresponding actual parameter (which therefore always must be a variable) upon termination of the procedure.

The facility of calling an array parameter by value has been removed. It contributes no additional power to the language, and it contravenes the general policy that operations on entire arrays should be specified by means of explicit iterations, rather than concealed by an implicit notation.

3.3.2. Statement Parameters

A facility has been provided for writing a statement as an actual parameter corresponding to a formal specified as **procedure**. The statement can be considered as a proper procedure body without parameters. This represents a considerable notational convenience, since it enables the procedure to be specified actually in the place where it is to be used, rather than disjointly in the head of some embracing block.

The label parameter has been abolished; its function may be taken over by placing a goto statement in the corresponding actual parameter position.

3.3.3. Specifications

The specification of all formal parameters, and the correct matching of actuals to formals, has been made

obligatory. The purpose of specifications is to inform the user of the procedure of the correct conditions of its use, and to ensure that the translator can check that these conditions have been met.

One of the most important facts about a procedure which operates on array parameters is the dimensionality of the arrays it will accept as actual parameters. A means has therefore been provided for indicating this in the specification of the parameter.

To compensate for the obligatory nature of specifications, their notation has been simplified by including them in the formal parameter list, rather than placing them in a separate specification part, as in ALGOL 60.

3.4 DATA STRUCTURES

The concept of an array has been taken from ALGOL 60 virtually unchanged, with the exception of a slight notational simplification.

To supplement the array concept, the language has been extended by the addition of a new type of structure (the *record*) consisting, like the array, of one or more elements (or *fields*). With each record there is associated a unique value of type **reference** which is said to refer to that record. This reference may be assigned as the value of a suitable field in another record, with which the given record has some meaningful relationship. In this way, groups of records may be linked in structural networks of any desired complexity.

The concept of records has been pioneered in the AED–I language by D. T. Ross.

3.4.1. Records and Fields

Like the array, a record is intended to occupy a given fixed number of locations in the store of a computer. It differs from the array in that the types of the fields are not required to be identical, so that in general each field of a record may occupy a different amount of storage. This, of course, makes it unattractive to select an element from a record by means of a computed ordinal number, or index; instead, each field position is given a unique invented name (identifier), which is written in the program whenever that field is referred to.

A record may be used to represent inside the computer some discrete physical or conceptual object to be examined or manipulated by the program, for example, a person, a town, a geometric figure, a node of a graph, etc. The fields of the record then represent properties of that object, for example, the name of a person, the distance of a town from some starting point, the length of a line, the time of joining a queue, etc. Normally, the name of the field suggests the property represented by that field.

In contrast to arrays, records are not created by declarations; rather, they are created dynamically by statements of the program. Thus their lifetimes do not have to be nested, and stack methods of storage control must be supplemented by more sophisticated techniques. It is intended that automatic "garbage collection" will be applicable to records, so that records which have become inaccessible may be detected, and the space they occupy released for other purposes.

3.4.2. References

The normal data types (**string, real, integer,** etc.) are sufficient to represent the properties of the objects represented by records; but a new type of data is required to represent relationships holding between these objects. Provided that the relationship is a functional relationship (i.e. many-one or one-one), it can be represented by placing as a field of one record a reference to the other record to which it is related. For example, if a record which represents a person has a field named *father*, then this is likely to be used to contain a reference to the record which represents that person's father. A similar treatment is possible to deal with the relationship between a town and the next town visited on some journey, between a customer and the person following him in some queue, between a directed line and its starting point, etc.

References are also used to provide the means by which the program gains access to records; for this purpose, variables of type **reference** should be declared in the head of the block which uses them. Such variables will at any given time refer to some subset of the currently existing records. Fields of records can be referred to directly by associating the name of the field with the value of the variable holding a reference to the relevant record. If that record itself has fields containing references to yet further records outside the initial subset, then fields of these other records may be accessed indirectly by further associating their names with the construction which identified the reference to the relevant record. By assignment of references, records previously accessible only indirectly can be made directly accessible, and records previously directly accessible can lose this status, or even become totally inaccessible, in which case they are considered as deleted.

Thus, for example, if B is a variable of type **reference** declared in the head of some enclosing block, and if *age* and *father* are field identifiers and if B contains a reference to a certain person, then

$$age\ (B)$$

(called a field designator) gives that person's age;

$$father(B)$$

is a reference to that person's father, and

$$age\ (father(B))$$

gives his father's age.

3.4.3. Record Classes

Two records may be defined as similar if they have the same number of fields, and if corresponding fields in the two records have the same names and the same types. Similarity in this sense is an equivalence relationship and may be used to split all records into mutually exclusive and exhaustive equivalence classes, called *record classes*. These classes tend to correspond to the natural classification of objects under some generic term, for example:

person, *town* or *quadrilateral*. Each record class must be introduced in a program by means of a record class declaration, which associates a name with the class and specifies the names and types of the fields which characterize the members of the class.

One of the major pitfalls in the use of references is the mistaken assumption that the value of a reference variable, –field or –parameter refers to a record of some given class, whereas on execution of the program it turns out that the reference value is associated with some record of quite a different class. If the programmer attempts to access a field inappropriate to the actual class referred to, he will get a meaningless result; but if he attempts to make an assignment to such a field, the consequences could be disastrous to the whole scheme of storage control. To avoid this pitfall, it is specified that the programmer can associate with the definition of every reference variable, –field or –parameter the name of the record class to which any record referred to by it will belong. The *translator* is then able to verify that the mistake described can never occur.

3.4.4. Efficiency of Implementation

Many applications for which record handling will be found useful are severely limited by the speed and capacity of the computers available. It has therefore been a major aim in the design of the record-handling facilities that in implementation the accessing of records and fields should be accomplished with the utmost efficiency, and that the layout of storage be subjected only to a minimum administrative overhead.

4. Possibilities for Language Extension

In the design of the language a number of inviting possibilities for extensions were considered. In many cases the investigation of these extensions seemed to reveal inconsistencies, indecisions and difficulties which could not readily be solved. In other cases it seemed undesirable to make the extension into a standard feature of the language, in view of the extra complexity involved.

In this section, suggested extensions are outlined for the consideration of implementors, users and other language designers.

4.1. Further String Operations

For some applications it seems desirable to provide facilities for referring to subsequences of bits and strings. The position of the subsequence could be indicated by a notation similar to subscript bounds, viz.

$S[i:j]$ the subsequence of S consisting of the ith to jth elements inclusive.

This notation is more compact than the use of a standard procedure, and it represents the fact that extraction is more likely to be performed by an open subroutine than a closed one. However, the notational similarity suggests that the construction might also appear in the left part of an assignment, in which case it denotes insertion rather than extraction, i.e. assignment to a part of the quantity. Apart from the undesirability of the same construction denoting two different operations, this would require that strings be classified as structured values along with arrays.

4.2. Further Data Types

Suggestions have been made for facilities to specify the precision of numbers in a more "flexible" way, e.g. by indicating the number of required decimal places. This solution has been rejected because it ignores the fundamental distinction between the number itself and one of its possible denotations, and as a consequence is utterly inappropriate for calculators not using the decimal number representation. As an alternative, the notion of a precision hierarchy could be introduced by prefixing declarations with a sequence of symbols **long**, where the number of **long**s determines the precision class. For reasons of simplicity, and in order that an implementation may closely reflect the properties of a real machine (single vs. double precision real arithmetic), allowing for only one **long** was considered as appropriate. Whether an implementation actually distinguishes between **real** and **long real** can be determined by an environment enquiry (cf. Part III, 2).

4.3. Initial Values and Local Constants

It is a minor notational convenience to be able to assign an initial value to a variable as part of the declaration which introduces that variable. A more important advantage is that the notation enables the programmer to express a very important feature of his calculations, namely, that this is an unique initial assignment made once only on the first entry to the block; furthermore it completely rules out the possibility of the elementary but all too common error of failing to make an assignment before the use of a variable.

However, such a facility rests on the notions of "compile time" and "run time" action, which, if at all, should be introduced at a conceptually much more fundamental level.

In some cases it is known that a variable only ever takes one value throughout its lifetime, and a means may be provided to make these cases notationally distinct from those of initial assignment. This means that the intention of the programmer can be made explicit for the benefit of the reader, and the translator is capable of checking that the assumption of constancy is in fact justified. Furthermore, the translator can sometimes take advantage of the declaration of constancy to optimize a program.

4.4. Array Constructors

To provide the same technique for the initialization of arrays as for other variables, some method should be provided for enumerating the values of an array as a sequence of expressions. This would require the definition of a reference denotation for array values, which, if available, would consequently suggest the introduction of operations on values of type array. The reasons for not extending the language in this direction have already been explained.

4.5. Record Class Discrimination

In general, the rule that the values of a particular reference variable or field must be confined to a single record class will be found to present little hardship; however, there are circumstances in which it is useful to relax this rule, and to permit the values of a reference variable to range over more than one record class. A facility is then desirable to determine the record class to which a referred record actually belongs.

Two possibilities for record class discriminations are outlined as follows.

1. A record union declaration is introduced with the form

union ⟨record union identifier⟩ (⟨record class identifier list⟩)

The record class identifier accompanying a reference variable declaration could then be replaced by a record union identifier, indicating that the values of that reference variable may range over all record classes included in that union. An integer primary of the form

⟨record union identifier⟩ (⟨reference expression⟩)

would then yield the ordinal number of the record class in that union to which the record referred to by the reference expression belongs.

2. Record class specifications in reference variable declarations are omitted, and a logical primary of the form

⟨reference primary⟩ **is** ⟨record class identifier⟩

could be introduced with the value **true**, if and only if the reference primary refers to a record of the specified record class.

While the introduction of a new kind of declaration (1) may seem undesirable, solution (2) reintroduces the dangerous pitfalls described in 3.4.3.

4.6. Procedure Parameters

It has been realized that in most implementations an actual parameter being an expression constitutes a function procedure declaration, and that one being a statement constitutes a proper procedure declaration. These quasi-procedure declarations, however, are confined to being parameterless. Samelson has suggested a notation for functionals which essentially does nothing more than remove this restriction: an actual parameter may include in its heading formal parameter specifications (cf. *ALGOL Bulletin 20.3.3.*). In a paper by Wirth and Weber, the notational distinction between procedure declarations and actual parameters has been entirely removed [cf. *Comm. ACM 9*, 2 (Feb. 1966), 89 ff.]. This was done along with the introduction of a new kind of actual parameters similar in nature to the references introduced here in connection with records.

However, neither ad hoc solutions nor a radical change from the parameter mechanism and notation of Algol 60 seemed desirable.

PART II. DEFINITION OF THE LANGUAGE

CONTENTS

1. Terminology, notation, and basic definitions
   - 1.1 Notation
   - 1.2 Definitions
2. Sets of basic symbols and syntactic entities
   - 2.1 Basic symbols
   - 2.2 Syntactic entities
3. Identifiers
4. Values and types
   - 4.1. Numbers
   - 4.2. Logical values
   - 4.3. Bit sequences
   - 4.4. Strings
   - 4.5. References
5. Declarations
   - 5.1. Simple variable declarations
   - 5.2. Array declarations
   - 5.3. Procedure declarations
   - 5.4. Record class declarations
6. Expressions
   - 6.1. Variables
   - 6.2. Function designators
   - 6.3. Arithmetic expressions
   - 6.4. Logical expressions
   - 6.5. Bit expressions
   - 6.6. String expressions
   - 6.7. Reference expressions
7. Statements
   - 7.1. Blocks
   - 7.2. Assignment statements
   - 7.3. Procedure statements
   - 7.4. Goto statements
   - 7.5. If statements
   - 7.6. Case statements
   - 7.7. Iterative statements

1. Terminology, Notation and Basic Definitions

The Reference Language is a phrase structure language, defined by a formal system. This formal system makes use of the notation and the definitions explained below. The structure of the language Algol is determined by the three quantities:

(1) $\mathfrak{V}$, the set of basic constituents of the language,
(2) $\mathfrak{U}$, the set of syntactic entities, and
(3) $\mathfrak{P}$, the set of syntactic rules, or productions.

1.1 Notation

A syntactic entity is denoted by its name (a sequence of letters) enclosed in the brackets ⟨ and ⟩. A syntactic rule has the form

$$\langle A \rangle ::= x$$

where $\langle A \rangle$ is a member of $\mathfrak{U}$, x is any possible sequence of basic constituents and syntactic entities, simply to be called a "sequence". The form

$$\langle A \rangle ::= x \mid y \mid \cdots \mid z$$

is used as an abbreviation for the set of syntactic rules

$$\langle A \rangle ::= x$$
$$\langle A \rangle ::= y$$
$$\cdots\cdots\cdots$$
$$\langle A \rangle ::= z$$

1.2 Definitions

1. A sequence x is said to *directly produce* a sequence y if and only if there exist (possibly empty) sequences u and w, so that either (*i*) for some $\langle A \rangle$ in $\mathfrak{U}$, $x = u\langle A \rangle w$, $y = uvw$, and $\langle A \rangle ::= v$ is a rule in $\mathfrak{P}$; or (*ii*) $x = uw$, $y = uvw$ and v is a "comment" (see below).

2. A sequence x is said to *produce* a sequence y if and only if there exists an ordered set of sequences $s[0]$, $s[1]$, $\cdots$, $s[n]$, so that $x = s[0]$, $s[n] = y$, and $s[i-1]$ directly produces $s[i]$ for all $i = 1, \cdots, n$.

3. A sequence x is said to be an Algol program if and only if its constituents are members of the set $\mathfrak{C}$, and x can be produced from the syntactic entity ⟨program⟩.

The sets $\mathfrak{C}$ and $\mathfrak{U}$ are defined through enumeration of their members in Section 2 of this Report (cf. also 4.4). The members of the set of syntactic rules are given throughout the sequel of the Report. To provide explanations for the meaning of Algol programs, the letter sequences denoting syntactic entities have been chosen to be English words describing approximately the nature of that syntactic entity or construct. Where words which have appeared in this manner are used elsewhere in the text, they refer to the corresponding syntactic definition. Along with these letter sequences the symbol ŝ may occur. It is understood that this symbol must be replaced by any one of a finite set of English words (or word pairs). Unless otherwise specified in the particular section, all occurrences of the symbol ŝ within one syntactic rule must be replaced consistently, and the replacing words are

integer	logical
real	bit
long real	string
complex	reference
long complex	

It is recognized that typographical entities of lower order than basic symbols (cf. 2.1), called characters, exist. Some basic symbols may be identical with characters; others, so-called word-delimiters, are generally represented as a sequence of two or more characters. Neither the set of available characters nor the decomposition of basic symbols into them is defined here. It is understood that basic symbols are not the same as characters and that there may exist characters which are neither basic symbols nor constituents of them; these characters may, however, enter the program as constitutents of strings, i.e. character sequences delimited by so-called string quotes.

The symbol **comment** followed by any sequence of characters not containing semicolons, followed by a semicolon (;), is called a comment. A comment has no effect on the meaning of a program, and is ignored during execution of the program. An identifier immediately following the basic symbol **end** is also regarded as a comment.

The basic constituents of the language are the basic symbols (cf. 2.1), strings (cf. 4.4), and comments.

All quantities referred to in a program must be defined. Their definition is achieved either within the Algol program by so-called declarations and label definitions, or is thought to be done in a text, possibly written in another language, in which the Algol program is embedded. A program containing references to quantities defined in the latter way can only be executed in an environment where these quantities are known, and this environment is considered to be a block containing that program.

The execution of a program can be considered as a sequence of units of action. The sequence of these units of action is defined as the evaluation of expressions and the execution of statements as denoted by the program. In the definition of the language the evaluation or execution of certain constructions is (1) either not precisely defined, e.g. real arithmetic, or (2) is left undefined, e.g. the order of evaluation of primaries in expressions, or (3) is even said to be undefined or not valid. This is to be interpreted in the sense that a program which uses constructions of the first two categories fully defines a computational process only if accompanying information specifies what is not given in the definition of the language. If in case (2) this information is not supplied, then a unique result of such a process is defined only if all possible alternatives lead to the same result. No meaning can be attributed to a program using constructions of the third category.

2. Sets of Basic Symbols and Syntactic Entities

2.1. Basic Symbols

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* |

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

i | **b** | **true** | **false** | " | **null** |

integer | **real** | **complex** | **logical** | **bits** | **string** | **reference** | **long** | **array** | **procedure** | **record** |

, | ; | : | . | (|) | [|] | **begin** | **end** | **if** | **then** | **else** | **case** | **of** |

+ | − | × | / | **div** | **rem** | ↑ | **abs** | ∨ | ∧ | ¬ | ↓ | **cat** | = | ≠ | < | ≤ | ≥ | > |

:= | **goto** | **for** | **step** | **until** | **do** | **while** | **comment** | **value** | **result**

2.2. Syntactic Entities

(with corresponding section numbers)

⟨actual parameter list⟩	7.3	⟨declaration⟩	5
⟨actual parameter⟩	7.3	⟨digit⟩	3.1
⟨array declaration⟩	5.2	⟨equality operator⟩	6.4
⟨bit factor⟩	6.5	⟨expression list⟩	6.7
⟨bit primary⟩	6.5	⟨field list⟩	5.4
⟨bit secondary⟩	6.5	⟨for clause⟩	7.7
⟨bit sequence⟩	4.3	⟨formal parameter list⟩	5.3
⟨bit term⟩	6.5	⟨formal parameter	
⟨bit⟩	4.3	segment⟩	5.3
⟨block body⟩	7.1	⟨formal type⟩	5.3
⟨block head⟩	7.1	⟨go to statement⟩	7.4
⟨block⟩	7.1	⟨identifier list⟩	3.1
⟨bound pair list⟩	5.2	⟨identifier⟩	3.1
⟨bound pair⟩	5.2	⟨if clause⟩	6
⟨case clause⟩	6	⟨if statement⟩	7.5
⟨case statement⟩	7.6	⟨imaginary part⟩	4.1
⟨control identifier⟩	3.1	⟨increment⟩	7.7

⟨initial value⟩	7.7	⟨simple 5 expression⟩	6.3
⟨iterative statement⟩	7.7	⟨simple type⟩	5.1
⟨label definition⟩	7.1	⟨simple variable declaration⟩	5.1
⟨label identifier⟩	3.1	⟨statement list⟩	7.6
⟨letter⟩	3.1	⟨statement⟩	7
⟨limit⟩	7.7	⟨string primary⟩	6.6
⟨logical factor⟩	6.4	⟨string⟩	4.4
⟨logical primary⟩	6.4	⟨subscript⟩	6.1
⟨logical secondary⟩	6.4	⟨5 array designator⟩	6.1
⟨logical term⟩	6.4	⟨5 array identifier⟩	3.1
⟨logical value⟩	4.2	⟨5 assignment statement⟩	7.2
⟨lower bound⟩	5.2	⟨5 expression list⟩	6
⟨null reference⟩	4.5	⟨5 expression⟩	6
⟨procedure declaration⟩	5.3	⟨5 factor⟩	6.3
⟨procedure heading⟩	5.3	⟨5 field designator⟩	6.1
⟨procedure identifier⟩	3.1	⟨5 field identifier⟩	3.1
⟨procedure statement⟩	7.3	⟨5 function designator⟩	6.2
⟨program⟩	7	⟨5 function identifier⟩	3.1
⟨proper procedure body⟩	5.3	⟨5 function procedure body⟩	5.3
⟨proper procedure declaration⟩	5.3	⟨5 function procedure declaration⟩	5.3
⟨real part⟩	4.1	⟨5 left part⟩	7.2
⟨record class declaration⟩	5.4	⟨5 number⟩	4.1
⟨record class identifier⟩	3.1	⟨5 primary⟩	6.3
⟨record designator⟩	6.7	⟨5 secondary⟩	6.3
⟨relation⟩	6.4	⟨5 term⟩	6.3
⟨relational operator⟩	6.4	⟨5 variable identifier⟩	3.1
⟨scale factor⟩	4.1	⟨5 variable⟩	6.1
⟨sign⟩	4.1	⟨type⟩	5.3
⟨simple bit expression⟩	6.5	⟨unscaled real⟩	4.1
⟨simple logical expression⟩	6.4	⟨unsigned 5 number⟩	4.1
⟨simple reference expression⟩	6.7	⟨upper bound⟩	5.2
⟨simple statement⟩	7	⟨while clause⟩	7.7
⟨simple string expression⟩	6.6		

3. Identifiers

3.1. Syntax

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩
⟨5 variable identifier⟩ ::= ⟨identifier⟩
⟨5 array identifier⟩ ::= ⟨identifier⟩
⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨5 function identifier⟩ ::= ⟨identifier⟩
⟨record class identifier⟩ ::= ⟨identifier⟩
⟨5 field identifier⟩ ::= ⟨identifier⟩
⟨label identifier⟩ ::= ⟨identifier⟩
⟨control identifier⟩ ::= ⟨identifier⟩
⟨letter⟩ ::= *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z*
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩, ⟨identifier⟩

3.2. Semantics

Variables, arrays, procedures, record classes and record fields are said to be *quantities*. Identifiers serve to identify quantities, or they stand as labels, formal parameters or control identifiers. Identifiers have no inherent meaning, and can be chosen freely.

Every identifier used in a program must be defined. This is achieved through

(a) a declaration (cf. Section 5), if the identifier identifies a quantity. It is then said to denote that quantity and to be a 5 variable–, 5 array–, procedure–, 5 function–, record class–, or 5 field identifier, where the symbol 5 stands for the appropriate word reflecting the type of the declared quantity;

(b) a label definition (cf. 7.1), if the identifier stands as a label. It is then said to be a label identifier;

(c) its occurrence in a formal parameter list (cf. 5.3). It is then said to be a formal parameter;

(d) its occurrence in a for clause following the symbol **for** (cf. 7.7). It is then said to be a control identifier.

The identification of the definition of a given identifier is determined by the following rules:

Step 1. If the identifier is defined within the smallest block embracing the given occurrence of that identifier by a declaration of a quantity or by its standing as a label, then it denotes that quantity or that label. A statement following a procedure heading or a for clause is considered to be a block.

Step 2. Otherwise, if that block is a procedure body and if the given identifier is identical with a formal parameter in the associated procedure heading, then it stands as that formal parameter.

Step 3. Otherwise, if that block is preceded by a for clause and the identifier is identical to the control identifier of that for clause, then it stands as that control identifier.

Otherwise, these rules are applied considering the smallest block embracing the block which has previously been considered.

If either step 1 or step 2 could lead to more than one definition, then the identification is undefined.

The scope of a quantity, a label, a formal parameter, or a control identifier is the set of statements in which occurrences of an identifier may refer by the above rules to the definition of that quantity, label, formal parameter or control identifier.

3.3. Examples

i
Person
elder sibling
x15

4. Values and Types

Constants and variables are said to possess a *value*. The value of a constant is determined by the denotation of the constant. In the language, every constant (except references) has a reference denotation (cf. 4.1–4.4). The value of a variable is the one most recently assigned to that variable. A value is (recursively) defined as either being a simple value, or a structured value, i.e. an ordered set of one or more values. Every value is said to be of a certain *type*. The following types of simple values are distinguished:

integer: the value is an integer,
real or **long real**: the value is a real number,
complex or **long complex**: the value is a complex number,
logical: the value is a logical value,
bits: the value is a linear sequence of bits,

string: the value is a linear sequence of characters.

reference: the value is a reference to a record.

The following types of structured values are distinguished:

array: the value is an ordered set of values, all of identical type and subscript bounds,

record: the value is a set of simple values.

A procedure may yield a value, in which case it is said to be a function procedure, or it may not yield a value, in which case it is called a proper procedure. The value of a function procedure is defined as the value which results from the execution of the procedure body (cf. 6.2.2).

Subsequently, the reference denotation of constants is defined. The reference denotation of any constant consists of a sequence of characters. This, however, does not imply that the value of the denoted constant is a sequence of characters, nor that it has the properties of a sequence of characters, except, of course, in the case of strings.

4.1. Numbers

4.1.1 Syntax

In the first rule below, every occurrence of the symbol 3 must be systematically replaced by one of the following words (or word pairs):

integer
real
long real
complex
long complex

⟨3 number⟩ ::= ⟨unsigned 3 number⟩ | ⟨sign⟩⟨unsigned 3 number⟩
⟨unsigned long complex number⟩ ::=
  long ⟨unsigned complex number⟩
⟨unsigned complex number⟩ ::= ⟨real part⟩**i**⟨imaginary part⟩
⟨real part⟩ ::= ⟨unsigned real number⟩ | ⟨unsigned integer number⟩
⟨imaginary part⟩ ::= ⟨real number⟩ | ⟨integer number⟩
⟨unsigned long real number⟩ ::= *long* ⟨unsigned real number⟩ |
  long ⟨unsigned integer number⟩
⟨unsigned real number⟩ ::= ⟨unscaled real⟩ | ⟨unscaled real⟩
  ⟨scale factor⟩ | ⟨unsigned integer number⟩⟨scale factor⟩
⟨unscaled real⟩ ::= ⟨unsigned integer number⟩.
  ⟨unsigned integer number⟩ | .⟨unsigned integer number⟩
⟨scale factor⟩ ::= ₁₀⟨integer number⟩
⟨unsigned integer number⟩ ::= ⟨digit⟩ |
  ⟨unsigned integer number⟩ ⟨digit⟩
⟨sign⟩ ::= + | −

4.1.2. Semantics

Numbers are interpreted according to the conventional decimal notation. A scale factor denotes an integral power of 10 which is multiplied by the unscaled real or integer number preceding it. Each number has a uniquely defined type.

4.1.3 Examples

1	.5	1i−1
−0100	1₁₀3	−0.33i0.67
3.1416	6.02486₁₀+23	**long** 0i1
+**long** 2.71828182845904523536028		

Note that −0.33i0.67 denotes −(0.33i0.67).

4.2 Logical Values

4.2.1 Syntax

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

4.3. Bit Sequences

4.3.1. Syntax

⟨bit sequence⟩ ::= **b**⟨bit⟩ | ⟨bit sequence⟩⟨bit⟩
⟨bit⟩ ::= 0 | 1

4.3.2. Semantics

The number of bits in a bit sequence is said to be the length of the bit sequence.

4.3.3. Examples

b10011
b001

4.4. Strings

4.4.1. Syntax

⟨string⟩ ::= "⟨sequence of characters⟩"

4.4.2. Semantics

Strings consist of any sequence of characters enclosed by but not containing the character ", called string quote. They are considered to be basic constituents of the language (cf. Section 1). The number of characters in a string excluding the quotes is said to be the length of the string.

4.5. References

4.5.1. Syntax

⟨null reference⟩ ::= **null**

4.5.2. Semantics

The reference value **null** fails to designate a record; if a reference expression occurring in a field designator has this value, then the field designator is undefined.

5. Declarations

Declarations serve to associate identifiers with the quantities used in the program, to attribute certain permanent properties to these quantities (e.g. type, structure), and to determine their scope. The quantities declared by declarations are simple variables, arrays, procedures and record classes.

Upon exit from a block, all quantities declared within that block lose their value and significance (cf. 7.1.2 and 7.4.2).

Syntax:

⟨declaration⟩ ::= ⟨simple variable declaration⟩ |
  ⟨array declaration⟩ | ⟨procedure declaration⟩ |
  ⟨record class declaration⟩

5.1. Simple Variable Declarations

5.1.1. Syntax

⟨simple variable declaration⟩ ::= ⟨simple type⟩ ⟨identifier list⟩
⟨simple type⟩ ::= **integer** | **real** | **long real** | **complex** |
  long complex | **logical** | **bits** (⟨unsigned integer number⟩) |
  bits | **string** | **reference** (⟨record class identifier⟩)

5.1.2. Semantics

Each identifier of the identifier list is associated with a

variable which is declared to be of the indicated type. A variable is called a simple variable, if its value is simple (cf. Section 4). If a variable is declared to be of a certain type, then this implies that only values which are assignment compatible with this type (cf. 7.2.2) can be assigned to it.

It is understood that the value of a variable of type **integer** is only equal to the value of the expression most recently assigned to it, if this value lies within certain unspecified limits. It is also understood that the value of a variable of type **real** is available only with a possible, unspecified deviation from the value of the expression most recently assigned to it. If in a declaration the symbol **real** is preceded by the symbol **long**, then this deviation is expected to be not greater than when the symbol **long** is missing. In the case of a variable of type **long complex** this holds separately for the real and imaginary parts of the complex number.

In the case of a variable of type **bits** the integer enclosed in parentheses indicates the actual length of the sequence which constitutes the value of this variable. If this specification is missing, then the length is assumed to be equal to the value of the environment enquiry function *bits in word* (cf. III.2).

In the case of a variable of type **reference**, the record class identifier enclosed within parentheses indicates the record class to whose records that reference variable may refer.

5.1.3. Examples

integer *i, j, k, m, n*
real *x, y, z*
long complex *c*
logical *p, q*
bits *g, h*
string *r, s, t*
reference (*Person*) *Jack, Jill*

5.2. Array Declarations

5.2.1. Syntax

⟨array declaration⟩ ::= ⟨simple type⟩ **array** ⟨bound pair list⟩ ⟨identifier list⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩ | ⟨bound pair⟩⟨bound pair list⟩
⟨bound pair⟩ ::= [⟨lower bound⟩:⟨upper bound⟩]
⟨lower bound⟩ ::= ⟨integer expression⟩
⟨upper bound⟩ ::= ⟨integer expression⟩

5.2.2. Semantics

Each identifier of the identifier list of an array declaration is associated with a variable which is declared to be of type **array**. A variable of type **array** is an ordered set of variables. Their number is determined by the leftmost element of the bound pair list. If the bound pair list consists of one element only, then their type is the simple type preceding the symbol **array**. Otherwise their type is **array**, and the number of elements and the type of these arrays are in turn defined by the given rules when applied to the remaining bound pair list.

Every element of an array is identified by an index. The indices are the integers between and including the values of the lower bound and the upper bound. Every expression in the bound pair list is evaluated exactly once upon entry to the block in which the declaration occurs. In order to be valid, for every bound pair, the value of the upper bound must not be less than the value of the lower bound.

5.2.3. Examples

integer array [1:100] *H*
real array [1:*m*] [1:*n*] *A,B*
string array [*j*:*k*+1] *street, town, city*

5.3. Procedure Declarations

5.3.1. Syntax

⟨procedure declaration⟩ ::= ⟨proper procedure declaration⟩ |
  ⟨ℑ function procedure declaration⟩
⟨proper procedure declaration⟩ ::= **procedure**
  ⟨procedure heading⟩; ⟨proper procedure body⟩
⟨ℑ function procedure declaration⟩ ::= ⟨simple type⟩ **procedure**
  ⟨procedure heading⟩; ⟨ℑ function procedure body⟩
⟨proper procedure body⟩ ::= ⟨statement⟩
⟨ℑ function procedure body⟩ ::= ⟨ℑ expression⟩ |
  ⟨block body⟩⟨ℑ expression⟩ **end**
⟨procedure heading⟩ ::= ⟨identifier⟩ |
  ⟨identifier⟩ (⟨formal parameter list⟩)
⟨formal parameter list⟩ ::= ⟨formal parameter segment⟩ |
  ⟨formal parameter list⟩; ⟨formal parameter segment⟩
⟨formal parameter segment⟩ ::= ⟨formal type⟩ ⟨identifier list⟩
⟨formal type⟩ ::= ⟨type⟩ | ⟨simple type⟩ **value** |
  ⟨simple type⟩ **result** | ⟨simple type⟩ **value result** |
  ⟨simple type⟩ **procedure** | **procedure** | **reference**
⟨type⟩ ::= ⟨simple type⟩ | ⟨type⟩ **array**

5.3.2. Semantics

A procedure declaration associates the procedure body with the identifier immediately following the symbol **procedure**. The principal part of the procedure declaration is the procedure body. Other parts of the block in whose heading the procedure is declared can then cause this procedure body to be executed or evaluated. A proper procedure is activated by a procedure statement (cf. 7.3), a function procedure by a function designator (cf. 6.2). Associated with the procedure body is a heading, containing the procedure identifier and possibly a list of formal parameters.

5.3.2.1. Specifications of formal parameters. All formal parameters of a formal parameter segment are of the same indicated type. It must be such that the substitution of the formal by an actual parameter of this specified type leads to correct Algol expressions and statements (cf. 7.3.2). The word **array** should be repeated as many times as appropriate.

5.3.2.2. The effect of the symbols **value** and **result** appearing in a formal type is explained by the following rewriting rule which is applied to the procedure body before the procedure is invoked:

(1) The procedure body is enclosed by the symbols **begin** and **end** if it is not already enclosed by these symbols;

(2) For every formal parameter whose formal type contains the symbol **value** or **result** (or both),

(a) a declaration followed by a semicolon is inserted in the heading of the procedure body, with a simple type as indicated in the formal type, and with an identifier different from any identifier valid at the place of the declaration.

(b) throughout the procedure body, every occurrence of the formal parameter identifier is replaced by the identifier defined in step 2a;

(c) if the formal type contains the symbol **value**, an assignment statement followed by a semicolon is inserted after the declarations of the procedure body. Its left part contains the identifier defined in step 2a, and its expression consists of the formal parameter identifier. The symbol **value** is then deleted;

(d) if the formal type contains the symbol **result**, an assignment statement preceeded by a semicolon is inserted before the symbol **end** which terminates a proper procedure body. In the case of a function procedure, an assignment statement followed by a semicolon is inserted before the final expression of the function procedure body. Its left part contains the formal parameter identifier, and its expression consists of the identifier defined in step 2a. The symbol **result** is then deleted.

5.3.3. Examples

```
procedure Increment; x := x+1
real procedure max (real value x, y); if x < y then y else x
procedure Copy (real array array U, V; integer value a, b);
  for i := 1 step 1 until a do
  for j := 1 step 1 until b do U[i][j] := V[i][j]
real procedure Horner (real array a; integer value n;
  real value x);
begin real s; s := 0;
  for i := 0 step 1 until n do s := s × x + a[i]; s
end
long real procedure sum (integer k, n; long real x);
begin long real y; y := 0; k := n;
  while k ≥ 1 do begin y := y + x; k := k − 1
  end; y
end
reference (Person) procedure youngest uncle
  (reference (Person) R);
begin reference (Person) p, m;
  p := youngest offspring (father(father(R)));
  while (p ≠ null) ∧ (¬male(p)) ∨ (p = father(R)) do
    p := elder sibling (p);
  m := youngest offspring (mother(mother(R)));
  while (m ≠ null) ∧ (¬ male(m)) do m := elder sibling (m);
  if p = null then m else
  if m = null then p else
  if age (p) < age (m) then p else m
end
```

5.4. Record Class Declarations

5.4.1. Syntax

```
⟨record class declaration⟩ ::= record ⟨record class identifier⟩
  (⟨field list⟩)
⟨field list⟩ ::= ⟨simple variable declaration⟩ |
  ⟨field list⟩; ⟨simple variable declaration⟩
```

5.4.2. Semantics

A record class declaration serves to define the structural properties of records belonging to the class. The principal constituent of a record class declaration is a sequence of simple variable declarations which define the fields and their types of the records of this class and associate identifiers with the individual fields. A record class identifier can be used in a record designator to construct a new record of the given class.

5.4.3. Examples

```
record Node (reference (Node) left, right)
record Person (string name; integer age; logical male;
  reference (Person) father, mother, youngest offspring,
    elder sibling)
```

6. Expressions

Expressions are rules which specify how new values are computed from existing ones. These new values are obtained by performing the operations indicated by the operators on the values of the operands. According to the type of their value, several types of expressions are distinguished. Their structure is defined by the following rules, in which the symbol 3 has to be replaced consistently as described in Section 1, and where the triplets 3_0, 3_1, 3_2 have to be either consistently replaced by the words

logical
bit
string
reference

or by any combination of words as indicated by the following table, which yields 3_0 given 3_1 and 3_2:

3_1 \ 3_2	integer	real	complex
integer	integer	real	complex
real	real	real	complex
complex	complex	complex	complex

3_0 has the quality "long" if either both 3_1 and 3_2 have that quality, or if one has the quality and the other is "integer".

Syntax:

```
⟨3 expression⟩ ::= ⟨simple 3 expression⟩ |
    ⟨case clause⟩ (⟨3 expression list⟩)
⟨3₀ expression⟩ ::= ⟨if clause⟩⟨simple 3₁ expression⟩ else
    ⟨3₂ expression⟩
⟨3 expression list⟩ ::= ⟨3 expression⟩
⟨3₀ expression list⟩ ::= ⟨3₁ expression list⟩, ⟨3₂ expression⟩
⟨if clause⟩ ::= if ⟨logical expression⟩ then
⟨case clause⟩ ::= case ⟨integer expression⟩ of
```

The operands are either constants, variables or function designators or other expressions between parentheses. The evaluation of the latter three may involve smaller units of action such as the evaluation of other expressions or the execution of statements. The value of an expression between parentheses is obtained by evaluating that expression. If an operator operates on two operands, then these operands may be evaluated in any order, or even in

parallel, with the exception of the case mentioned in 6.4.2.2. The construction

⟨if clause⟩⟨simple $\mathfrak{Z}_1$ expression⟩ **else** ⟨$\mathfrak{Z}_2$ expression⟩

causes the selection and evaluation of an expression on the basis of the current value of the logical expression contained in the if clause. If this value is **true**, the simple expression following the if clause is selected, if the value is **false**, the expression following **else** is selected. The construction

⟨case clause⟩ (⟨$\mathfrak{Z}$ expression list⟩)

causes the selection of the expression whose ordinal number in the expression list is equal to the current value of the integer expression contained in the case clause. In order that the case expression is defined, the current value of this expression must be the ordinal number of some expression in the expression list.

6.1. Variables

6.1.1. Syntax

⟨$\mathfrak{Z}$ variable⟩ ::= ⟨$\mathfrak{Z}$ variable identifier⟩|⟨$\mathfrak{Z}$ field designator⟩|
⟨$\mathfrak{Z}$ array designator⟩ ⟨subscript⟩
⟨$\mathfrak{Z}$ field designator⟩ ::= ⟨$\mathfrak{Z}$ field identifier⟩ (⟨reference expression⟩)
⟨$\mathfrak{Z}$ array designator⟩ ::= ⟨$\mathfrak{Z}$ array identifier⟩|
⟨$\mathfrak{Z}$ array designator⟩⟨subscript⟩
⟨subscript⟩ ::= [⟨integer expression⟩|

6.1.2. Semantics

A subscripted array designator denotes the variable whose index, in the ordered set of variables denoted by the array designator, is the current value of the expression in the subscript. This value must lie within the declared bounds.

The value of a variable may be used in expressions for forming other values, and may be changed by assignments to that variable.

A field designator designates a field in the record referred to by its reference expression. The type of the field designator is defined by the declaration of that field identifier in the record class designated by the reference expression of the field designator (cf. 5.4).

6.1.3. Examples

x
A[*i*]
M[*i*+*j*][*i*−*j*]
father(*Jack*)
mother(*father*(*Jill*))

6.2. Function Designators

6.2.1. Syntax

⟨$\mathfrak{Z}$ function designator⟩ ::= ⟨$\mathfrak{Z}$ function identifier⟩|
⟨$\mathfrak{Z}$ function identifier⟩ (⟨actual parameter list⟩)

6.2.2. Semantics

A function designator defines a value which can be obtained by a process performed in the following steps:

Step 1. A copy is taken of the body of the function procedure whose procedure identifier is given by the function designator and of the actual parameters of the latter.

Steps 2, 3, 4. As specified in 7.3.2.

Step 5. The copy of the function procedure body, modified as indicated in steps 2–4, is executed. The value of the function designator is the value of the expression which constitutes or is part of the modified function procedure body. The type of the function designator is the type preceding **procedure** preceding the heading of the corresponding function procedure declaration.

6.2.3. Examples

max (*x* ↑ 2, *y*×2)
sum (*i*, 100, *H*[*i*])
sum (*i*, *m*, *sum*(*j*, *n*, *A*[*i*][*j*]))
youngest uncle (*Jill*)
sum (*i*, 10, *X*[*i*]×*Y*[*i*])
Horner (*X*, 10, 2.7)

6.3. Arithmetic Expressions

6.3.1. Syntax

In any of the following rules, every occurrence of the symbol $\mathfrak{Z}$ must be systematically replaced by one of the following words (or word pairs):

integer
real
long real
complex
long complex

The rules governing the replacement of the symbols $\mathfrak{Z}_0$ $\mathfrak{Z}_1$ and $\mathfrak{Z}_2$ are given in 6.3.2.

⟨simple $\mathfrak{Z}$ expression⟩ ::= ⟨$\mathfrak{Z}$ term⟩| + ⟨$\mathfrak{Z}$ term⟩| − ⟨$\mathfrak{Z}$ term⟩
⟨simple $\mathfrak{Z}_0$ expression⟩ ::= ⟨simple $\mathfrak{Z}_1$ expression⟩ + ⟨$\mathfrak{Z}_2$ term⟩
⟨simple $\mathfrak{Z}_1$ expression⟩ − ⟨$\mathfrak{Z}_2$ term⟩
⟨$\mathfrak{Z}$ term⟩ ::= ⟨$\mathfrak{Z}$ factor⟩
⟨$\mathfrak{Z}_0$ term⟩ ::= ⟨$\mathfrak{Z}_1$ term⟩ × ⟨$\mathfrak{Z}_2$ factor⟩
⟨$\mathfrak{Z}_0$ term⟩ ::= ⟨$\mathfrak{Z}_1$ term⟩/⟨$\mathfrak{Z}_2$ factor⟩
⟨integer term⟩ ::= ⟨integer term⟩ **div** ⟨integer factor⟩|
⟨integer term⟩ **rem** ⟨integer factor⟩
⟨$\mathfrak{Z}_1$ factor⟩ ::= ⟨$\mathfrak{Z}_0$ secondary⟩|⟨$\mathfrak{Z}_1$ factor⟩ ↑ ⟨integer secondary⟩
⟨$\mathfrak{Z}_0$ secondary⟩ ::= ⟨$\mathfrak{Z}$ primary⟩|⟨unsigned $\mathfrak{Z}$ number⟩
⟨$\mathfrak{Z}_0$ secondary⟩ ::= **abs** ⟨$\mathfrak{Z}_1$ primary⟩|**abs** ⟨unsigned $\mathfrak{Z}_1$ number⟩
⟨long $\mathfrak{Z}_0$ primary⟩ ::= **long** ⟨$\mathfrak{Z}_1$ primary⟩
⟨$\mathfrak{Z}$ primary⟩ ::= ⟨$\mathfrak{Z}$ variable⟩|⟨$\mathfrak{Z}$ function designator⟩|
(⟨$\mathfrak{Z}$ expression⟩)

6.3.2. Semantics

An arithmetic expression is a rule for computing a number.

According to its type it is either called an integer–, real–, long real–, complex–, or long complex expression.

6.3.2.1. The operators +, −, × and / have the conventional meaning of addition, subtraction, multiplication and division. In the relevant syntactic rules of 6.3.1 the symbols $\mathfrak{Z}_0$, $\mathfrak{Z}_1$ and $\mathfrak{Z}_2$ have to be replaced by any combination of words according to the following table which indicates $\mathfrak{Z}_0$ for any combination of given $\mathfrak{Z}_1$ and $\mathfrak{Z}_2$.

Operators + | −

$\mathfrak{Z}_1$ \ $\mathfrak{Z}_2$	integer	real	complex
integer	integer	real	complex
real	real	real	complex
complex	complex	complex	complex

$\mathfrak{Z}_0$ has the quality "long" if both $\mathfrak{Z}_1$ and $\mathfrak{Z}_2$ have the quality "long", or if one has the quality "long" and the other is "integer".

Operator ×	$\mathfrak{Z}_2$ / $\mathfrak{Z}_1$	integer	real	complex
	integer	integer	long real	long complex
	real	long real	long real	long complex
	complex	long complex	long complex	long complex

$\mathfrak{Z}_1$ or $\mathfrak{Z}_2$ having the quality "long" does not affect the type of the result.

Operator /	$\mathfrak{Z}_2$ / $\mathfrak{Z}_1$	integer	real	complex
	integer	real	real	complex
	real	real	real	complex
	complex	complex	complex	complex

The specifications for the quality "long" are those given for + and −.

6.3.2.2. The operator − standing as the first symbol of a simple expression denotes the monadic operation of sign inversion. The type of the result is the type of the operand. The operator + standing as the first symbol of a simple expression denotes the monadic operation of identity.

6.3.2.3. The operator **div** is mathematically defined as

$$a \textbf{ div } b = sgn(a \times b) \times d(\textbf{abs } a, \textbf{ abs } b)$$

where the function procedures *sgn* and *d* are declared as

```
integer procedure sgn(integer value a);
  if a < 0 then −1 else 1;
integer procedure d(integer value a, b);
  if a < b then 0 else d(a−b, b) + 1
```

6.3.2.4. The operator **rem** (remainder) is mathematically defined as

$$a \textbf{ rem } b = a - (a \textbf{ div } b) \times b$$

6.3.2.5. The operator ↑ denotes exponentiation of the first operand to the power of the second operand. In the relevant syntactic rule of 6.3.1 the symbols $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ have to be replaced by any of the following combinations of words:

$\mathfrak{Z}_0$	$\mathfrak{Z}_1$
real	integer
real	real
complex	complex

$\mathfrak{Z}_0$ has the quality "long" if and only if $\mathfrak{Z}_1$ does.

6.3.2.6. The monadic operator **abs** yields the absolute value of the operand. In the relevant syntactic rule of 6.3.1 the symbols $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ have to be replaced by any of the following combinations of words:

$\mathfrak{Z}_0$	$\mathfrak{Z}_1$
integer	integer
real	real
real	complex

If $\mathfrak{Z}_1$ has the quality "long", then so does $\mathfrak{Z}_0$.

6.3.2.7. Precedence of operators. The syntax of 6.3.1 implies the following hierarchy of operator precedences:

long
abs
↑
× / **div rem**
+ −

Sequences of operations of equal precedence shall be executed in order from left to right.

6.3.2.8. Precision of arithmetic. If the result of an arithmetic operation is of type **real** or **complex**, then it is the mathematically understood result of the operation performed on operands which may deviate from the actual operands. In case of the operands being of a type with the quality "long", this deviation, as described in 5.1.2, is intended to be smaller, and is expected to be not greater than if that quality is missing.

In the relevant syntactic rule of 6.3.1 the symbols $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ must be replaced by any of the following combinations of words (or word pairs):

Operator **long**	$\mathfrak{Z}_0$	$\mathfrak{Z}_1$
	long real	real
	long real	integer
	long complex	complex

6.3.3. Examples

$x + c/H[j-1]$
$c + A[i] \times B[i]$
$exp\ (-x/(2 \times sigma))/sqrt\ (2 \times sigma)$

6.4. Logical Expressions

6.4.1. Syntax

In the following rules for ⟨relation⟩ the symbols $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ must either be identically replaced by any one of the following words:

bit
string
reference

or by any of the words from:

complex
long complex
real
long real
integer

and the symbols $\mathfrak{Z}_2$ and $\mathfrak{Z}_3$ must be replaced by any of the last three: real, long real, integer.

⟨simple logical expression⟩ ::= ⟨logical term⟩|⟨relation⟩
⟨logical term⟩ ::= ⟨logical factor⟩|⟨logical term⟩ ∨ ⟨logical factor⟩
⟨logical factor⟩ ::= ⟨logical secondary⟩|
  ⟨logical factor⟩ ∧ ⟨logical secondary⟩
⟨logical secondary⟩ ::= ⟨logical primary⟩|¬⟨logical primary⟩
⟨logical primary⟩ ::= ⟨logical value⟩|⟨logical variable⟩|
  ⟨logical function designator⟩|(⟨logical expression⟩)
⟨relation⟩ ::=
  ⟨simple $\mathfrak{Z}_0$ expression⟩⟨equality operator⟩⟨simple $\mathfrak{Z}_1$ expression⟩|
  ⟨logical term⟩⟨equality operator⟩⟨logical term⟩|
  ⟨simple $\mathfrak{Z}_2$ expression⟩⟨relational operator⟩⟨simple $\mathfrak{Z}_3$ expression⟩
⟨relational operator⟩ ::= <|≤|≥|>
⟨equality operator⟩ ::= =|≠

6.4.2. Semantics

A logical expression is a rule for computing a logical value.

6.4.2.1. The relational operators have their conventional meanings, and yield the logical value **true** if the relation is satisfied for the values of the two operands; **false**, otherwise. Two references are equal if and only if they are both **null** or both refer to the same record. Two strings are equal if and only if they have the same length and the same ordered sequence of characters.

A comparison of two bit sequences of different lengths is preceded by insertion of an appropriate number of 0's after the symbol **b** of the shorter operand.

6.4.2.2. The operators $\neg$ (not), $\wedge$ (and), and $\vee$ (or), operating on logical values, are defined by the following equivalences:

$\neg x$ **if** x **then false else true**
$x \wedge y$ **if** x **then** y **else false**
$x \vee y$ **if** x **then true else** y

6.4.2.3. Precedence of operators. The syntax of 6.4.1 implies the following hierarchy of operator precedences:

$$\begin{array}{c} \neg \\ \wedge \\ \vee \\ < \leq = \neq \geq > \end{array}$$

6.4.3. Examples

$p \vee \neg q$
$(x<y) \wedge (y<z)$
$(i=j) = (m=n)$
youngest offspring (*Jack*) $\neq$ **null**

6.5. Bit Expressions

6.5.1. Syntax

⟨simple bit expression⟩ ::= ⟨bit term⟩|
⟨simple bit expression⟩ $\vee$ ⟨bit term⟩
⟨bit term⟩ ::= ⟨bit factor⟩|⟨bit term⟩ $\wedge$ ⟨bit factor⟩
⟨bit factor⟩ ::= ⟨bit secondary⟩|$\neg$⟨bit secondary⟩
⟨bit secondary⟩ ::= ⟨bit primary⟩|
⟨bit secondary⟩ $\uparrow$ ⟨integer secondary⟩|
⟨bit secondary⟩ $\downarrow$ ⟨integer secondary⟩
⟨bit primary⟩ ::= ⟨bit sequence⟩|⟨bit variable⟩|
⟨bit function designator⟩|(⟨bit expression⟩)

6.5.2. Semantics

A bit expression is a rule for computing a bit sequence.

The operators $\vee$, $\wedge$ and $\neg$ produce a result of type **bits**, every bit being dependent on the corresponding bit(s) in the operand(s) as follows:

x	y	$\neg x$	$x \wedge y$	$x \vee y$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

The operators $\uparrow$ and $\downarrow$ denote the shifting operation to the left and to the right respectively by the number of bit positions indicated by the absolute value of the integer secondary. Vacated bit positions to the right or left respectively are assigned the bit sequence value **b**0. If in the case of the $\wedge$ and $\vee$ operators the two operands are not of equal length, then the shorter operand is extended by insertion of an appropriate number of 0's after the symbol **b**. The length of the result of a bit operator is equal to the length of the operand(s).

6.5.3. Examples

$g \wedge h \vee$ **b**111000
$g \wedge \neg (h \vee g) \downarrow 8$

6.6. String Expressions

6.6.1. Syntax

⟨simple string expression⟩ ::= ⟨string primary⟩|
⟨simple string expression⟩ **cat** ⟨string primary⟩
⟨string primary⟩ ::= ⟨string⟩|⟨string variable⟩|
⟨string function designator⟩|(⟨string expression⟩)

6.6.2. Semantics

A string expression is a rule for computing a string (sequence of characters).

6.6.2.1. The operator **cat** (catenate) yields the string consisting of the sequence of characters resulting from evaluation of the first operand, immediately followed by the sequence of characters resulting from evaluation of the second operand, mathematically defined as

"⟨sequence-1⟩" **cat** "⟨sequence-2⟩" = "⟨sequence-1⟩⟨sequence-2⟩"

The length of the result is the sum of the lengths of the operands.

6.6.3. Example

s **cat** "␣+␣" **cat** t

6.7. Reference Expressions

6.7.1. Syntax

⟨simple reference expression⟩ ::= ⟨null reference⟩|
⟨reference variable⟩|⟨reference function designator⟩|
⟨record designator⟩|(⟨reference expression⟩)
⟨record designator⟩ ::= ⟨record class identifier⟩|
⟨record class identifier⟩ (⟨expression list⟩)
⟨expression list⟩ ::= ⟨3 expression⟩|
⟨expression list⟩, ⟨3 expression⟩

6.7.2. Semantics

A reference expression is a rule for computing a reference to a record. All simple reference expressions in a reference expression must be of the same record class.

The value of a record designator is the reference to a newly created record belonging to the designated record class. If the record designator contains an expression list, then the values of the expressions are assigned to the fields of the new record. The entries in the expression list are taken in the same order as the fields in the record class declaration, and the types of the fields must be assignment compatible with the types of the expressions (cf. 7.2.2).

6.7.3. Example

Person ("Carol", 0, **false**, *Jack*, *Jill*, **null**,
youngest offspring(*Jack*))

7. Statements

A statement is said to denote a unit of action. By the execution of a statement is meant the performance of this unit of action which may consist of smaller units of action such as the evaluation of expressions or the execution of other statements.

A statement containing no symbols denotes no action.

Syntax:

⟨program⟩ ::= ⟨block⟩
⟨statement⟩ ::= ⟨simple statement⟩|⟨iterative statement⟩|
⟨if statement⟩|⟨case statement⟩
⟨simple statement⟩ ::= ⟨block⟩|⟨𝔍 assignment statement⟩|
⟨procedure statement⟩|⟨goto statement⟩

7.1. Blocks

7.1.1. Syntax

⟨block⟩ ::= ⟨block body⟩ ⟨statement⟩ **end**
⟨block body⟩ ::= ⟨block head⟩|⟨block body⟩⟨statement⟩; |
⟨block body⟩⟨label definition⟩
⟨block head⟩ ::= **begin**|⟨block head⟩⟨declaration⟩;
⟨label definition⟩ ::= ⟨identifier⟩:

7.1.2. Semantics

Every block introduces a new level of nomenclature. This is realized by execution of the block in the following steps:

Step 1. If an identifier defined in the block head or in a label definition of the block body is already defined at the place from where the block is entered, then every occurrence of that identifier within the block is systematically replaced by another identifier, which is defined neither within the block nor at the place from where the block is entered.

Step 2. If the declarations of the block contain array bound expressions, then these expressions are evaluated.

Step 3. Execution of the statements contained in the block body begins with the execution of the first statement following the block head.

After execution of the last statement of the block body (unless it is a goto statement) a block exit occurs, and the statement following the entire block is executed.

7.1.3. Example

begin real u;
$u := x$; $x := y$; $y := z$; $z := u$
end

7.2. Assignment Statements

7.2.1. Syntax

In the following rules the symbols $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ must be replaced by words as indicated in Section 1, subject to the restriction that the type $\mathfrak{Z}_0$ is assignment compatible with the type $\mathfrak{Z}_1$ as defined in 7.2.2.

⟨$\mathfrak{Z}_0$ assignment statement⟩ ::= ⟨$\mathfrak{Z}_0$ left part⟩⟨$\mathfrak{Z}_1$ expression⟩|
⟨$\mathfrak{Z}_0$ left part⟩⟨$\mathfrak{Z}_1$ assignment statement⟩
⟨𝔍 left part⟩ ::= ⟨𝔍 variable⟩ :=

7.2.2. Semantics

The execution of assignment statements causes the assignment of the value of the expression to one or several variables. The assignment is performed after the evaluation of the expression. The types of all left part variables must be assignment compatible with the type of the expression.

A type $\mathfrak{Z}_0$ is said to be assignment compatible with a type $\mathfrak{Z}_1$, if either

(1) the two types are identical (except possibly for length specifications), or

(2) $\mathfrak{Z}_0$ is **real** or **long real**, and $\mathfrak{Z}_1$ is **integer**, **real**, or **long real**, or

(3) $\mathfrak{Z}_0$ is **complex** or **long complex**, and $\mathfrak{Z}_1$ is **integer**, **real**, **long real**, **complex** or **long complex**.

In the case of the type **bits**, the length specified for $\mathfrak{Z}_0$ must be not less than the length specified for $\mathfrak{Z}_1$.

If the length of a bit sequence to be assigned is smaller than the length specified for $\mathfrak{Z}_0$, then a suitable number of 0's are inserted after the symbol **b**.

In the case of a reference, the reference to be assigned must refer to a record of the class specified by the record class identifier associated with the reference variable in its declaration.

7.2.3. Examples

$z := age(Jack) := 28$
$x := y + \mathbf{abs}\ z$
$c := i + x + c$
$p := x \neq y$

7.3. Procedure Statements

7.3.1. Syntax

⟨procedure statement⟩ ::= ⟨procedure identifier⟩|
⟨procedure identifier⟩ (⟨actual parameter list⟩)
⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
⟨actual parameter list⟩, ⟨actual parameter⟩
⟨actual parameter⟩ ::= ⟨expression⟩|⟨statement⟩|
⟨𝔍 array designator⟩|⟨procedure identifier⟩|⟨𝔍 function identifier⟩

7.3.2. Semantics

The execution of a procedure statement is equivalent to a process performed in the following steps:

Step 1. A copy is taken of the body of the proper procedure whose procedure identifier is given by the procedure statement, and of the actual parameters of the latter.

Step 2. If the procedure body is a block, then a systematic change of identifiers in its copy is performed as specified by step 1 of 7.1.2.

Step 3. The copies of the actual parameters are treated in an undefined order as follows: If the copy is an expression different from a variable, then it is enclosed by a pair of parentheses, or if it is a statement it is enclosed by the symbols **begin** and **end**.

Step 4. In the copy of the procedure body every occurrence of an identifier identifying a formal parameter is replaced by the copy of the corresponding actual parameter (cf. 7.3.2.1). In order for the process to be defined, these replacements must lead to correct Algol expressions and statements.

Step 5. The copy of the procedure body, modified as indicated in steps 2–4, is executed.

7.3.2.1. Actual formal correspondence

The correspondence between the actual parameters and the formal parameters is established as follows: The actual parameter list of the procedure statement (or of the function designator) 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.

7.3.2.2. Formal specifications

If a formal parameter is specified by **value,** then the formal type must be assignment compatible with the type of the actual parameter. If it is specified as **result,** then the type of the actual variable must be assignment compatible with the formal type. In all other cases, the types must be identical. If an actual parameter is a statement, then the specification of its corresponding formal parameter must be **procedure.**

7.3.3. Examples

Increment
Copy (*A*, *B*, *m*, *n*)

7.4. Goto Statements

7.4.1. Syntax

⟨goto statement⟩ ::= **goto** ⟨label identifier⟩

7.4.2. Semantics

An identifier is called a label identifier if it stands as a label.

A goto statement determines that execution of the text be continued after the label definition of the label identifier. The identification of that label definition is accomplished in the following steps:

Step 1. If some label definition within the most recently activated but not yet terminated block contains the label identifier, then this is the designated label definition. Otherwise,

Step 2. The execution of that block is considered as terminated and Step 1 is taken as specified above.

7.5. If Statements

7.5.1. Syntax

⟨if statement⟩ ::= ⟨if clause⟩⟨statement⟩|
⟨if clause⟩⟨simple statement⟩ **else** ⟨statement⟩
⟨if clause⟩ ::= **if** ⟨logical expression⟩ **then**

7.5.2. Semantics

The execution of if statements causes certain statements to be executed or skipped depending on the values of specified logical expressions. An if statement of the form

⟨if clause⟩⟨statement⟩

is executed in the following steps:

Step 1. The logical expression in the if clause is evaluated.

Step 2. If the result of Step 1 is **true**, then the statement following the if clause is executed. Otherwise step 2 causes no action to be taken at all.

An if statement of the form

⟨if clause⟩⟨simple statement⟩ **else** ⟨statement⟩

is executed in the following steps:

Step 1. The logical expression in the if clause is evaluated.

Step 2. If the result of Step 1 is **true**, then the simple statement following the if clause is executed. Otherwise the statement following **else** is executed.

7.5.3. Examples

if $x = y$ **then goto** L
if $x < y$ **then** $u := x$ **else if** $y < z$ **then** $u := y$ **else** $v := z$

7.6. Case Statements

7.6.1. Syntax

⟨case statement⟩ ::= ⟨case clause⟩ **begin** ⟨statement list⟩ **end**
⟨statement list⟩ ::= ⟨statement⟩|⟨statement list⟩; ⟨statement⟩
⟨case clause⟩ ::= **case** ⟨integer expression⟩ **of**

7.6.2. Semantics

The execution of a case statement proceeds in the following steps:

Step 1. The expression of the case clause is evaluated.

Step 2. The statement whose ordinal number in the statement list is equal to the value obtained in Step 1 is executed. In order that the case statement is defined, the current value of the expression in the case clause must be the ordinal number of some statement of the statement list.

7.6.3. Examples

case i **of**
begin $x := x + y$;
  $y := y + z$;
  $z := z + x$
end

case j **of**
begin $H[i] := -H[i]$;
  begin $H[i-1] := H[i-1] + H[i]$; $i := i - 1$ **end**;
  begin $H[i-1] := H[i-1] \times H[i]$; $i := i - 1$ **end**;
  begin $H[H[i-1]] := H[i]$; $i := i - 2$ **end**
end

7.7. Iterative Statements

7.7.1. Syntax

⟨iterative statement⟩ ::= ⟨for clause⟩⟨statement⟩|
⟨while clause⟩⟨statement⟩
⟨for clause⟩ ::= **for** ⟨control identifier⟩ :=
⟨initial value⟩ **step** ⟨increment⟩ **until** ⟨limit⟩ **do**
⟨initial value⟩ ::= ⟨integer expression⟩
⟨increment⟩ ::= ⟨integer expression⟩
⟨limit⟩ ::= ⟨integer expression⟩
⟨while clause⟩ ::= **while** ⟨logical expression⟩ **do**

7.7.2. Semantics

The iterative statement serves to express that a statement be executed repeatedly depending on certain conditions specified by a for clause or a while clause. The state-

ment following the for clause or the while clause always acts as a block, whether it has the form of a block or not.

(a) An iterative statement of the form

for ⟨control identifier⟩ := *e*1 **step** *e*2 **until** *e*3 **do** ⟨statement⟩

is exactly equivalent to the block

begin ⟨statement-0⟩; ⟨statement-1⟩; ...; ⟨statement-*i*⟩; ...; ⟨statement-*n*⟩ **end**

when in the *i*th statement every occurrence of the control identifier is replaced by the reference denotation of the value of the expression *e*1 + *i* × *e*2, enclosed in parentheses.

The index *n* of the last statement is determined by $n \leq (e3-e1)/e2 < n+1$. If $n < 0$, then it is understood that the sequence is empty. The expressions *e*1, *e*2, and *e*3 are evaluated exactly once, namely before execution of ⟨statement-0⟩.

(b) An iterative statement of the form

while *e* **do** ⟨statement⟩

is exactly equivalent to

```
if e then
begin ⟨statement⟩;
  while e do ⟨statement⟩
end
```

7.7.3. Examples

```
for v := 1 step 1 until n − 1 do s := s + A[v][v]
for k := m step − 1 until 1 do
  if H[k−1] > H[k] then
  begin m := H[k−1]; H[k−1] := H[k]; H[k] := m end
while (j>0) ∧ (city [j]≠s) do j := j − 1
```

PART III. PROPOSED SET OF STANDARD PROCEDURES

The principal language features described in previous sections should be supplemented by additional facilities supplied in the form of procedures, which are assumed to be declared in the environment in which an ALGOL program is executed. It is recommended that some or all of the procedures listed in this section be so treated. They are classified into the following groups:

(1) Input/output procedures
(2) Environment enquiries
(3) Functions of analysis
(4) Transfer functions

1. Standard Input/Output Procedures

1.1. Introduction

This proposal is based on suggestions of Jan V. Garwick [*ALGOL Bull. 19*, 39–40].

1.2. Design Criteria

1.2.1. The input/output proposal is essentially simple, and the various facilities provided are relatively independent of one another. No attempt is made to provide discrimination, looping and sequencing facilities within the input/output proposal, since this merely duplicates features which are already provided in the general purpose language which the proposal supplements.

1.2.2. It is plainly recognized that different input/output media have radically different properties, and no attempt is made to introduce an artificial similarity into their use, nor to mislead a programmer by such an apparent similarity.

1.2.3. Advantage is taken of the essential differences between input and output, in particular of the fact that input of numbers does not require the same variety of format specifications as output.

1.2.4. Facilities are provided such that the specification of all matters associated with input and output can be written explicitly in a single sequence of instructions; errors due to incorrect mating of a format string and the sequence of input/output data which it is intended to control therefore cannot occur.

1.2.5. The number of digits of a number to be output can be specified by means of an integer expression, which can readily be calculated by the program itself.

1.2.6. The proposal is not intended to satisfy every requirement, but only to provide facilities adequate for most circumstances and capable of being used to build more complex input/output algorithms for more unusual requirements. Furthermore, there is no embargo on the provision of yet further standard procedures to perform additional, more complex functions.

1.3. Summary

Input and output channels of a computer are classified into three essentially different categories:

(1) *Legible input channels*, on which the information is presented in a form closely mapping its legible transcription. The main representatives of this class are card readers and paper tape readers.

(2) *Legible output channels*, in which the form of the information output either is, or closely maps, its legible transcription. The main representatives of this class are line printers, card punches, paper tape punches, and CRT character displays.

(3) *Input/output channels*, in which the information is stored in a form not suitable for human inspection, and can be read only by a computer. Input/output channels are divided into two classes, those with random access (e.g., drums, disks, or bulk core memories) and those with which access is essentially serial (e.g., magnetic tapes).

Legible output is achieved in two stages; first an "output line" of characters is assembled, and then it is transmitted on a specified channel. Since these operations are clearly distinct, they are performed by distinct procedures.

Facilities provided for legible input are the simplest, since in general no specifications of format are required.

Operations on (nonlegible) input/output channels are defined only for arrays, which are transferred in their entirety to and from the input/output medium.

On serial input/output channels, the positioning of the information is determined by the current position of the medium. On random access channels, the output instruction provides the programmer with an integer position identification, which he may use for specifying reinput of the same information.

1.4. Legible Output Channels

procedure *scaled* (**string value result** *line*; **integer value** *position, length*; **long real value** *expression*);
comment This procedure is used when the order of magnitude of a number is unknown. The value of *expression* is converted to decimal form, and placed in the *length* character positions of the string *line* starting at position *position*. The character position *position* is occupied by a minus sign if the number is negative or a space otherwise. The next position is occupied by a digit, the following position by a decimal point. The fourth last character position is occupied by $_{10}$, the next position by a plus or minus sign, and the remaining two positions by digits.

Examples: $1.234_{10}+01$
$-1.234_{10}-70$
$1.234_{10}+00$
$0.000_{10}+00$;

procedure *aligned* (**string value result** *line*; **integer value** *position, length, decimals*; **long real value** *expression*);
comment This procedure is used when the order of magnitude of a number is known. The value of *expression* is converted to decimal form, and placed in the *length* character positions of the string *line*, starting at position *position*.

The last *decimals* character positions of the field are occupied by digits and preceded by a decimal point, which itself is preceded by digits. Leading zeros are suppressed, up to but not including the last position before the point, and a minus sign (if any) precedes the leftmost digit.

Examples: 1.234
−123.456
−0.123
0.000

If the absolute value of the number is too great for it to be expressed in this way, the result is undefined;

procedure *decimal* (**string value result** *line*; **integer value** *position, length, expression*);
comment The value of *expression* is converted to decimal form, and placed in the *length* character positions of the string *line*, beginning at position *position*.

Leading zeros are suppressed up to, but not including the last digit. The first digit is preceded by either a space or a minus sign.

Examples: −12
1234
0
123

If the absolute value of the number is too great for it to be expressed in this way, the result is undefined;

procedure *insert* (**string value result** *line*; **integer value** *position*; **string value** *message*);
comment The string *message* is inserted in the string *line*, beginning at position *position*;

string procedure *substring* (**string value** *line*; **integer value** *postion, length*);
comment The substring consists of the *length* characters beginning at position *position* of the string *line*;

procedure *output* (**integer value** *channel, n*; **string value** *line*);
comment The first *n* characters of the string *line* are output on the specified legible output channel. If the channel has a natural unit of information and is incapable of accommodating in this unit (e.g. print line) the number of characters transmitted, the result is undefined. If it can accommodate more characters, then the remaining character positions are filled with spaces;

integer procedure *lastcol* (**integer value** *channel*);
comment This is an environment enquiry, and enables the programmer to find the number of characters in the natural unit of information on the specified legible channel, if there is such a unit. This procedure also applies to legible input channels;

1.5. Legible Input Channels

procedure *inreal* (**integer value** *channel*; **real result** *x*);
comment The next real or integer number (defined in accordance with II. 4.1.1) is read in from the specified channel, and its value is assigned to the variable *x*.

In each case, the characters read consist of an initial sequence of nonnumeric characters, followed by a sequence of numeric characters, terminated by, but not including, a nonnumeric character. The decimal digits and the delimiters . $_{10}$ + and − are numeric characters, and all other characters (including space, tab, and change to a new line) are nonnumeric. If the sequence of numeric characters does not conform to the definition of a real or integer number, the consequences are undefined;

procedure *ininteger* (**integer value** *channel*; **integer result** *i*);
comment This procedure is identical to *inreal*, except that the numeric sequence must conform to the definition of an integer number, and the result is assigned to the integer variable *i*;

procedure *input* (**integer value** *channel, n*; **string result** *line*);
comment *n* characters are read on the specified legible input channel and assigned to the string variable *line*. If the channel has a natural unit of information (e.g., card record) and the number of characters in that unit is greater than *n*, then the remaining characters are ignored, and if it is smaller than *n* then the result is undefined;

1.6. Serial Input/Output Channels

procedure *outserial* (**integer value** *channel*; **array** *information*);
comment The channel is a serial input/output channel. The entire array is output to the next available position of the medium in such a way that it can be read in by *inserial*. If there is insufficient room on the medium to write the information, the result is undefined. This procedure may be used for arrays of any type, order, or size;

procedure *rewind* (**integer value** *channel*);
comment On a serial channel, the medium is rewound to the position of the first information output;

procedure *inserial* (**integer value** *channel*; **array** *information*);
comment On a serial channel, the next array stored on the medium is input. This array must be of the same type and order, and have identical subscript bounds to the array output in this position; otherwise the result is undefined. Furthermore, output

instructions must be separated by a rewind from any input instruction. An attempt to read information which has not been written leads to undefined results. The procedure may be used for arrays of any type, order or size;

1.7. Random Input/Output Channels

procedure *outrandom* (**integer value** *channel*; **integer result** *identification*; **array** *information*);
comment The entire array is output on the specified random access channel, and the variable corresponding to the formal parameter *identification* is assigned a value which identifies the position of the information on the channel. If there is insufficient room on the medium, the result is undefined;

procedure *inrandom* (**integer value** *channel, identification*; **array** *information*);
comment The array which was output with the identification specified is reinput. The type, order and dimensions of the array must be the same as that which was output;

procedure *overwrite* (**integer value** *channel, identification*; **array** *information*);
comment The array is output to the specified random access channel, overwriting the information which originally was given the identification specified by the second parameter. The type, order and dimensions of the array must be the same as those which were originally written;

procedure *resetrandom* (**integer value** *channel, identification*);
comment All information on the channel written at the position specified by the identification is deleted, and the space which it occupied becomes free for further use;

1.8. Operating Procedures

procedure *open input* (**integer result** *channel*; **string value** *device*);
comment The variable *channel* is assigned the number of the legible input channel identified by the string parameter;

procedure *open output* (**integer result** *channel*; **string value** *device*);
comment The variable *channel* is assigned the number of the legible output channel identified by the string parameter;

procedure *open serial input* (**integer result** *channel*; **string value** *file label*);
comment Similar to *open input*, for a serial input/output channel;

procedure *open serial output* (**integer result** *channel*; **string value** *file label*);
comment The variable *channel* is assigned the number of some available serial input/output channel, and that channel is made unavailable. The implementation ensures that if the output medium is later removed, it has the identification specified by the string parameter;

procedure *open random input* (**integer result** *channel*; **string value** *file label*);
comment Similar to *open input*, for a random input/output channel;

procedure *open random output* (**integer result** *channel*; **string value** *file label*);
comment Similar to *open output*, for a random input/output channel;

procedure *open serial* (**integer result** *channel*);
comment The variable *channel* is assigned the number of some available serial input/output channel, and that channel is made nonavailable. This procedure is recommended for claiming "scratch" tapes;

procedure *open random* (**integer result** *channel*);
comment The variable *channel* is assigned the number of some available random input/output channel, and that channel is made unavailable. This procedure is recommended for claiming "scratch" files;

procedure *close* (**integer value** *channel*);
comment The specified channel is made available for reuse;

2. Standard Environment Enquiries

2.1 Introduction

It is recognized that different implementations of the language must adopt different techniques for dealing with certain language features. The programmer may wish to obtain information on these points, so that he may adapt his algorithmic methods accordingly, or even indicate that the algorithm is inappropriate.

The concept of an environment enquiry was originated by Peter Naur [*ALGOL Bull. 18.3.9.1*].

2.2 Functions Provided

real procedure *epsilon*;
comment The smallest possible number such that both $1 + epsilon \neq 1$ and $1 - epsilon \neq 1$;

long real procedure *epsilon squared*;

integer procedure *intmax*;
comment The largest positive integer provided by the implementation;

real procedure *realmax*;
comment The largest positive real number provided by the implementation;

integer procedure *bits in word*;
comment The number of elements of a bit sequence which is accommodated in a single word;

integer procedure *lowerbound* (**array** *A*);
comment The value of the lower subscript bound of the array *A*, which may be of any type or order;

integer procedure *upperbound* (**array** *A*);
comment The value of the upper subscript bound of the array *A*, which may be of any type or order;

integer procedure *string length* (**string** *s*);
comment The number of characters in the string *s*;

3. Standard Functions of Analysis

real procedure *sin* (**real value** *x*);

real procedure *cos* (**real value** *x*);

real procedure *arctan* (**real value** *x*);
comment $-\pi/2 < arctan(x) < \pi/2$;

real procedure *ln* (**real value** *x*);

real procedure *exp* (**real value** *x*);

real procedure *sqrt* (**real value** *x*);

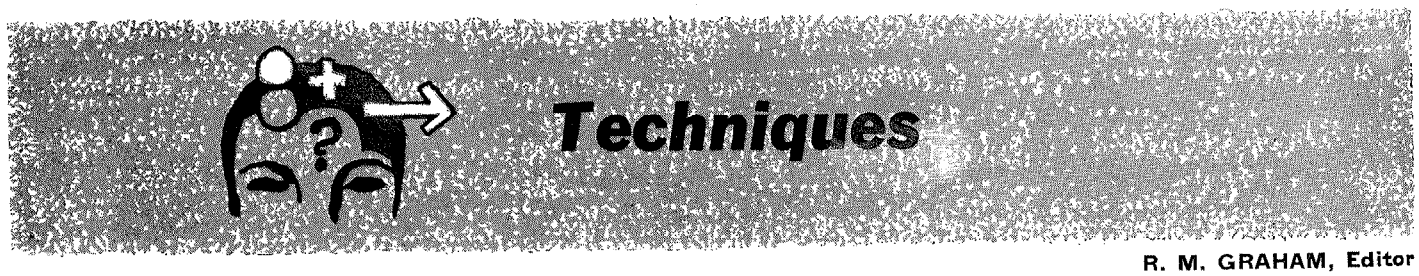
real procedure *arcsin* (**real value** *x*);
comment $-\pi/2 \leq arcsin(x) \leq \pi/2$;

real procedure *arccos* (**real value** *x*);
comment $-\pi/2 \leq arccos(x) \leq \pi/2$;

real procedure *tan* (**real value** *x*);

real procedure *pi*;
comment π with the accuracy available for real numbers;

(*Continued on page 432*)



R. M. GRAHAM, Editor

A New Uniform Pseudorandom Number Generator

DAVID W. HUTCHINSON
University of California, Berkeley*

A new multiplicative congruential pseudorandom number generator is discussed, in which the modulus is the largest prime within accumulator capacity and the multiplier is a primitive root of that prime. This generator passes the usual statistical tests and in addition the least significant bits appear to be as random as the most significant bits—a property which generators having modulus 2^k do not possess.

1. Introduction

In the past five or six years several papers have appeared on pseudorandom number generators for binary machines using the congruential method. These generators produce pseudorandom integers which then can be transformed to fixed-point fractions or floating-point numbers. The method which has come to be known as "multiplicative congruential" generates the ith pseudorandom integer by the recursion relation:

$$X_{i+1} = AX_i \pmod{M}$$

where A is the multiplier and M, the modulus, is usually chosen to be 2^k for a machine with a k-bit accumulator. See [1] and [2] for a description of how to choose A, M and X_0 to achieve a sufficiently long period and for results of some tests of randomness.

The "mixed congruential" method is:

$$X_{i+1} = AX_i + C \pmod{M}$$

where C is an odd integer [1, 2].

In the mixed congruential method, A is usually chosen to be 2^q+1 so that the multiplication can be effected by a shift and add. This saves time but leads to generators which can have serious defects, depending on the choice of q (see [1, 3]). On an IBM 7094 the multiplicative congruential method is faster than the mixed congruential. However, even on computers where this is not true it is doubtful that the gain in time is worth the risk of the poorer statistical behavior of the mixed congruential method.

2. The Lehmer Method

It was the poor behavior of a mixed congruential generator which caused us to have a talk with D. H. Lehmer who first proposed the congruential method for generating pseudorandom integers [4]. Lehmer said we were being too miserly with time in trying to do a shift and add rather than a full multiplication. He suggested the generator:

$$X_{i+1} = AX_i \bmod (2^{35} - 31)$$

(for a 7094), which involves doing yet an additional division. Here $2^{35} - 31$ is the largest prime less than 2^{35} and A is a primitive root of $2^{35} - 31$, say, $A = 5^5$ or 5^{13}. $A = 5$ is also a primitive root, but has only two bits and testing has proved it to be unsatisfactory. If A is a primitive root

* Statistics Department. This work was supported by the United States Public Health Service Grant GM-10525.

Continued from page 431

It is understood that also long variants of these procedures exist, e.g.,

long real procedure *longsin* (**long real value** x);

4. Standard Transfer Functions

integer procedure *round* (**real value** x);
integer procedure *truncate* (**real value** x);
integer procedure *entier* (**real value** x);
real procedure *realpart* (**complex value** x);
real procedure *imagpart* (**complex value** x);
long real procedure *longrealpart* (**long complex value** x);
long real procedure *longimagpart* (**long complex value** x);
complex procedure *complex* (**real value** x, y);
long complex procedure *longcomplex* (**long real value** x, y);
logical procedure *odd* (**integer value** x);
bits procedure *bitstring* (**integer value** i);
integer procedure *number* (**bits value** b);
comment the number with binary representation b;
integer procedure *decode* (**string value** *char*);
comment The numeric code of the character in the single-element string *char*;
string procedure *code* (**integer value** n);

Acknowledgment. The authors wish to thank the referee for his most exacting and valuable suggestions.

RECEIVED JANUARY, 1966; REVISED FEBRUARY, 1966