September 1961

ALGOL BULLETIN SUPPLEMENT No. 16

The Structure of Formula - Translators

(Theoretical Investigation of Translators)

by P. LUCAS §

Introduction.

The research reported in this paper has been made possible through the support and sponsorship of the United States Government.

This Algol Bulletin Supplement is a reprint of a part of the Final Report DA - 91 - 591 - EUC - 1430 "An Extension of the Algorithmic Language ALGOL" and, therefore, has the headlines and the page - numbering of this report. The same text has appeared also in German in the August 1961 issue of "Elektronische Rechenanlagen" (Munich).

§ MAILÜFTERL, Vienna IV, Gusshausstrasse 25

1.1 Theoretical Investigation of Translators

1.10 Statement of the Problem

The following paper is intended to show a method how to derive the structure of a translator on the basis of the syntactic definitions of the programming language.

1.101 Introduction

This paper was written in the course of designing a translator for the algorithmic formula language ALGOL 60 [2]. Besides this it is based on the work done by F. L. BAUER and K. SAMELSON [3].

The method developed can be applied to any given programming language which satisfies certain conditions, and which is stated with the help of the types of definitions described in chapter 2.2. The advantage of this method is that in one single run it can deal on the one hand with the translation of the programming language and on the other hand with the check of the program for syntactical correctness.

1.102 General Features of the Translator

The starting point of the translation process is the program stated in the programming language the so called source program. The aim of the translation is the program in the machine language which corresponds to the source program, i.e. the generated program.

---

[2] Report on the Algorithmic Language ALGOL 60
Comm. ACM 2 (1960) 106 - 136.

[3] K. SAMELSON, F. L. BAUER, Sequential Formula Translation
Comm. ACM 3 (1960) 76 - 83.

The source program contains first the information to be dealt with in the form of data or in the form of variables denoting data. Secondly it contains instructions, the so called operators, which define how these variables are to be dealt with.

Furthermore, the source program contains additional information concerning the data or variables, required for an exact determination of the meaning of individual expressions. Data and variables correspond to addresses and contents of addresses in the generated program. Each operator corresponds to a set of instructions of the generated program. The translator contains a list of these sets of instructions in which the addresses of the arguments are left out. The translator calls for the set of instructions corresponding to an operator and inserts the addresses corresponding to the arguments. The resulting set is then added to the program already generated. Since the operators can not always be carried out in the same sequence in which they occured in the source program, they must in such cases be reordered.

The tasks enumerated just now are taken over by subroutines of the translator. Which subroutines are to be called by the translator at a certain moment of the translation process is determined by the last character which has been and by the history, i.e. by the state of the translator. The character which has been read last and the state represent the information which governs the translation process. For this reason this information will be called control information. The present paper deals with the control of translation processes. It will be shown, in which way the control of a translation process can on certain conditions be derived from the syntactic definitions of a programming language. The method will be demonstrated by means of examples, which belong to the language ALGOL 60.

1.11 Definition of Terms

1.111 Levels of Language

As for all artificial languages, it is necessary for programming languages to describe the language, i.e. to speak about the language. For this purpose one may use for example the English language. However, for the description of a formalized language one may use another formalized language, which is the only exact method. In order to avoid ambiguities and misunderstandings one may use different symbols for the language to be described and for the description itself.

The language that is to be described is termed object language, while the language for the description of the object language is called metalanguage.

In the current case the metalanguage itself becomes likewise object of investigation and thus becomes object language. For its description another, second metalanguage must be introduced. The first level metalanguage will be called for short metalanguage 1, the second metalanguage will be called metalanguage 2 or meta - metalanguage.

1.112 Syntax

If the description of the language relates only to possible groupings of the basic symbols of that language and does not concern itself with the meaning, this description is called syntax.

The syntax of a language is therefore understood to mean those definitions and rules which permit the formation of expressions from the symbols of a language without having regard to the meaning of those symbols. The syntax states which strings of basic symbols constitute a possible expression of the language and which do not. It does not concern itself with the question, which strings are meaningful.

1.12 The Formalism for the Description of Programming Languages

1.121 Means of Expression for the Three Levels of Language

The object language on which this paper is based is ALGOL 60. The metalanguage 1 is that used in the ALGOL 60 - Report [2]. The metalanguage 2 is newly defined in chapter 1.1213.

1.1211 The Object Language: ALGOL 60

ALGOL 60 is primarily intended for the description of algorithms of numerical mathematics. An effort was made to utilize as much as possible the customary mathematical formula language, which therefore represents the core of ALGOL 60. The customary symbolism had to be supplemented by means of expressions which enable an unambiguous statement of the process of numerical evaluation.

In the following a few examples will be given for expressions belonging to the object language; the corresponding expressions in metalanguage are given in brackets <...> (see chapter 2.12):

<integer>	506, 2 3228
<identifier>	a, P5, alpha, sin
<arithmetic expression>	a + b / c
<Boolean expression>	a < b, X1 ∧ X2
<assignment statement>	y := a + b / c
<conditional statement>	if a < b then y := a + b / c else y := a - b / c

1.1212 Metalanguage 1 : Metalanguage of the ALGOL 60 - Report

The metalanguage 1 serves to define the syntax of the object language. The designations given in brackets represent variables in the metalanguage for which specific strings of symbols belonging to the object language may be inserted.

---

[2] **Report on the Algorithmic Language** ALGOL 60
Comm. ACM 2 (1960) 106 - 136.

The connective : : = is used for definition symbol and the symbol | is used for or.
For example :

⟨ letter⟩ : : = a | b | c | d | e ... y | z

In the following the variable to be defined will be called definiendum and the defining expression will be called definiens. In the definitions the definiendum is positioned on the left side and the definiens on the right side.

The formulas of the metalanguage may also incorporate symbols of the object language. In this case however, these symbols designate themselves.
For example :

⟨if clause⟩ : : = if ⟨Boolean expression⟩then

Juxtaposition of metalinguistic variables means the juxtaposition of the sequences of symbols represented by the variables;
for example :

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

1.1213 Basic Symbols and Means of Expression of Metalanguage 2

Metalanguage 2 is used to describe the types of definitions which are formulated in metalanguage 1. To some extent, the metalanguage will be the English language. However, it is also necessary to define some formal expressions, which will be enumerated in the following:

1. The symbols : : = and | are taken from metalanguage 1 and designate themselves.

2. Greek letters enclosed in double brackets denote syntactic units belonging to metalanguage 2.
For example :

⟨⟨α⟩⟩, ⟨⟨β⟩⟩, ⟨⟨γ⟩⟩, ...

The values of these variables are constants and variables of meta - language 1.

3. As in metalanguage 1, juxtaposition of variables designates juxtaposition of the syntactic units denoted by the variables.

4. The connectives $\neg$, $\wedge$ and $\vee$ are used in the customary sense.

5. To facilitate the description of the various types of definitions, a logical function "cont" is introduced to denote an important property of variables in meta - metalanguage:

> The logical function cont $(\langle\langle\alpha\rangle\rangle, \langle\langle\beta\rangle\rangle)$ is true if the variable $\langle\langle\alpha\rangle\rangle$ contains the variable $\langle\langle\beta\rangle\rangle$ in the tree of its definitions.

This means that the function is true if the definiens of $\langle\langle\alpha\rangle\rangle$ contains $\langle\langle\beta\rangle\rangle$ either directly or if it contains a variable whose definiens contains the variable $\langle\langle\beta\rangle\rangle$ or e.t.c. ...
For example:

> The general form of the definition of <integer> (see also the example 1.1212) would be
>
> $\langle\langle\varphi\rangle\rangle ::= \langle\langle\alpha\rangle\rangle \mid \langle\langle\alpha\rangle\rangle\langle\langle\varphi\rangle\rangle$
>
> where
>
> cont $(\langle\langle\varphi\rangle\rangle, \langle\langle\varphi\rangle\rangle)$

1.122 Types of Syntactical Definitions

One may divide the syntactical definitions used in this paper into

(1) enumerating definition,

(2) juxtaposing definition,

(3) combined definition (combined (1) and (2)),

and in each of these groups one may distinguish

(E) explicit or non recursive definitions and

(R) recursive definitions.

Each of the types of definition given in the above list corresponds to a certain program structure.

1,1221 The Enumerating Definition

The enumerating definition takes the form

$$\langle\langle \varkappa \rangle\rangle ::= \langle\langle \alpha_1 \rangle\rangle | \langle\langle \alpha_2 \rangle\rangle | \langle\langle \alpha_3 \rangle\rangle | \ldots \langle\langle \alpha_n \rangle\rangle \qquad (1)$$

This type defines a syntactic category. By this one understands a class of expressions which are interchangeable in certain positions without causing the entire expression to become syntactically wrong. Since according to the definition given above one of the variables $\langle\langle \alpha_1 \rangle\rangle$, $\langle\langle \alpha_2 \rangle\rangle$ ... $\langle\langle \alpha_n \rangle\rangle$ may be inserted in all positions which are occupied by $\langle\langle \varkappa \rangle\rangle$, the variables $\langle\langle \alpha_1 \rangle\rangle$, $\langle\langle \alpha_2 \rangle\rangle \ldots \langle\langle \alpha_n \rangle\rangle$ form a syntactic category.

For example:

⟨ multiplying operator ⟩ ::= ×|/|+ (1E)

⟨ declaration ⟩ ::= ⟨type declaration⟩|⟨ array declaration⟩|
|⟨switch declaration⟩|⟨ procedure de-
claration⟩ (1R)

1,1222 The Juxtaposing Definition

The juxtaposing definition has the form

$$\langle\langle \mathcal{S} \rangle\rangle ::= \langle\langle \alpha_1 \rangle\rangle\langle\langle \alpha_2 \rangle\rangle\langle\langle \alpha_3 \rangle\rangle \ldots \langle\langle \alpha_n \rangle\rangle \qquad (2)$$

The variable $\langle\langle \mathcal{S} \rangle\rangle$ may be replaced by $\langle\langle \alpha_1 \rangle\rangle\langle\langle \alpha_2 \rangle\rangle \ldots \langle\langle \alpha_n \rangle\rangle$.

For example:

⟨if clause⟩ ::= if ⟨Boolean expression⟩ then (2R)

1,1223 The Combined Definition

The combined definition is an enumerative definition the constituents of which may be chains.

For example :

⟨integer⟩ : : = ⟨ unsigned integer⟩ | + ⟨ unsigned integer⟩ |
| ⟨ unsigned integer ⟩ (1E)

⟨array declaration⟩ : : = array ⟨array list⟩ | ⟨local or own type⟩ array ⟨array list⟩ (3E)

1,1224 The Explicit Definition

Let be $\langle\langle\delta\rangle\rangle$ the definiendum and $\langle\langle\alpha_1\rangle\rangle, \langle\langle\alpha_2\rangle\rangle \ldots \langle\langle\alpha_n\rangle\rangle$ the variables appearing within the definiens. The definition is explicit if the following function is true :

$$\neg \text{ cont } (\langle\langle \delta \rangle\rangle, \langle\langle \delta \rangle\rangle)$$

For example :

⟨multiplying operator ⟩ : : = × | / | +

This definition is explicit because the definiens contains only basic symbols of the object language.

1,1225 The Recursive Definition

Let $\langle\langle \delta \rangle\rangle$ be the definiendum and $\langle\langle\alpha_1\rangle\rangle, \langle\langle\alpha_2\rangle\rangle \ldots \langle\langle\alpha_n\rangle\rangle$ the variables appearing within the definiens: The definition is recursive if the condition is valid:

$$\text{cont}(\langle\langle \delta \rangle\rangle, \langle\langle \delta \rangle\rangle)$$

In order to be able to break off the expression built by a recursive definition, there must exist for each variable $\langle\langle\alpha\rangle\rangle$ defined recursive, a variable $\langle\langle \beta \rangle\rangle$ which may be substituted for $\langle\langle\alpha\rangle\rangle$ and for which the condition $\neg$ cont $(\langle\langle\beta\rangle\rangle, \langle\langle\alpha\rangle\rangle)$ is valid.
For example :

⟨ if clause ⟩ : : = if ⟨ Boolean expression ⟩ then (2R)

The definition is recursive on account of the definition of the variable ⟨ Boolean expression ⟩:

⟨Boolean expression⟩ : : = ⟨simple Boolean⟩ | ⟨if clause⟩ ⟨simple Boolean ⟩ else ⟨Boolean expression⟩ (3R)

In the following two special forms of recursive definitions are shown for which a simpler programming structure may be given (see chapter 1,1334):

(I) $\langle\langle \varrho \rangle\rangle ::= \langle\langle \alpha \rangle\rangle | \langle\langle \beta \rangle\rangle \langle\langle \varrho \rangle\rangle$ (3R)

(II) $\langle\langle \varrho \rangle\rangle ::= \langle\langle \alpha \rangle\rangle | \langle\langle \varrho \rangle\rangle \langle\langle \beta \rangle\rangle$ (3R)

Definition (I) yields a chain of the form

$$\langle\langle \beta \rangle\rangle \langle\langle \beta \rangle\rangle \ldots \langle\langle \beta \rangle\rangle \langle\langle \alpha \rangle\rangle$$

Definition (II) yields a chain of the form

$$\langle\langle \alpha \rangle\rangle \langle\langle \beta \rangle\rangle \ldots \langle\langle \beta \rangle\rangle \langle\langle \beta \rangle\rangle$$

For example:

<unsigned integer> ::= <digit> | <unsigned integer> <digit> (3R)

1,123 Syntactic check

The translator must be able to process in the right way all expressions which may be built by means of the syntactic definitions of the object language. All expressions which do not satisfy the syntactic definitions are forbidden because they do not belong to the object language. It is desirable that the translator detects and indicates expressions which are forbidden. This is possible with a programming structure which is essentially the same as necessary for the translation process because for translation and syntactical checking the same questions have to be posed and the same decisions have to be made. Therefore it is economical to do the translation process and the syntactic check in one single run.

Out of the elementary programming structure which will be described in the following a tree structure may be built up which on the one hand permits the checking of the syntax and on the other hand controls the translation process.

1.13 The Derivation of the Programming Structures from the Types of Definition

1.131 Basic Principle and Method of Representation

1.1311 Basic Principle

Each metalinguistic variable represents certain strings of symbols. According to their meaning these strings of symbols must be transformed into the corresponding information for the generated program.

The translator will be designed in such a way, that each meaning of a metalinguistic variable corresponds to a subroutine which carries out the transformation of the string of symbols. As the meaning of a metalinguistic variable may depend on the context, the subroutine corresponding to a certain variable also contains the transformation of the information, the meaning of which is not defined till the variable occurs. This part of the information of a variable which cannot be utilized when the variable occurs, is stored in auxiliary storage cells. Those storage cells which lie within the scope of a recursive subroutine must be arranged in a cellar (see chapter 1.133412). If a variable $\langle\langle \alpha \rangle\rangle$ has different meaning in different positions, different subroutines must be provided according to the meanings. In order to facilitate the representation, it will be assumed in the following that to each variable is assigned only a single meaning, so that only a single subroutine has to be provided for each variable.

If a variable occurs only in a single position in a definiens of the syntactic definitions, a simple program element can be inserted in the structure instead of the corresponding subroutine.

Syntactic checking and translator may be built up from subroutines in the described manner, because of the relation existing between the structure of subroutines and the syntax. A variable can or cannot contain another variable; this relation exists also between

subroutines. If a variable contains itself, then in the translator a subroutine must contain itself. This requirement cannot be met with the usual technique of subroutines; the solution is possible with the use of the cellar principle (see chapter 1.1334).

Consideration of the meaning of the symbols is not needed for syntactic check. The meaning becomes essential, however, for the translation process which takes the meaning of the symbols partially from the source program and partially from the translator itself. Furthermore, in opposition to the syntax, interlacing structures arise when the meaning is taken into consideration.

The case that the variable $\langle\langle\alpha\rangle\rangle$ contains the variable $\langle\langle\beta\rangle\rangle$ means for the corresponding expression belonging to the object language that the string represented by $\langle\langle\beta\rangle\rangle$ is positioned within the string represented by $\langle\langle\alpha\rangle\rangle$ That is, the string $\langle\langle\beta\rangle\rangle$ is embraced by the other symbols of $\langle\langle\alpha\rangle\rangle$, for that reason interlacing structures may not arise. However, if the meaning of symbols is considered, interlacing structures may occur; for example of the form A1 B1 A2 E2 where first A1 and A2 and second B1 and B2 are pairs of correlated elements. These interlacing structures may simply be handled by storing the information of A1 and B1 in two different storage cells or in two block cells (see 1.133412) and utilizing this information when A2 and B2 occur.

1.1312 Means of Description

Principally it would be possible to describe the following programming structures in ALGOL 60; but in order to make the description more obvious in this paper flow charts are preferred.

The subroutine corresponding to the variable $\langle\langle\alpha_i\rangle\rangle$ is denoted by $\langle\langle\alpha_i\rangle\rangle$, enclosed in a rectangle (see Fig. 1.1-1). The question of the program whether the variable $\langle\langle\alpha_i\rangle\rangle$ is present or not is represented by $\langle\langle\alpha_i\rangle\rangle$? in the customary manner (see Fig 1.1-2)

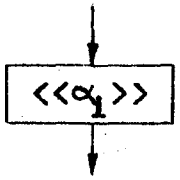


Fig. 1.1 - 1 : Subroutine for the translation of <<α_i>>

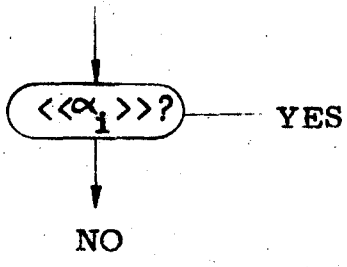


Fig. 1.1 - 2 : Identification of <<α_i>>

The returnjump out of a subroutine is denoted by RJ

1.132 Assumptions

1.1321 Assumptions Concerning the Syntactic Definitions

1.13211 Each metalinguistic variable must be distinguishable from the other constituents of the syntactic category to which it belongs by the very first symbol of the string represented by the variable.

On account of this assumption each metalinguistic variable may be recognized as soon as the first symbol of this variable is known to the translator. The question for a certain metalinguistic variable is therefore identical with the question for the first symbol of the string represented by this variable. This assumption serves to facilitate the descriptions in this paper; moreover it simplifies the structure of the translator if it is valid in the syntax for a language. For some definitions of ALGOL 60 this assumption is not satisfied. In such

cases the translator has to ask for further symbols of the string in order to get an unambiguous answer; but no new problem arises thereby.

1,13212 The end of a string represented by a variable must be recognizable either with the last symbol of the string or with the first symbol of the following variable.

1,1322 Conventions Concerning the Subroutines

1,13221 The input routine reads when activated the next basic symbol of the source program. This basic symbol is then stored by the input routine in the storage location C. That means that in the storage location C all basic symbols of the source program appear successively in the course of the translation process. The symbols read in successively by the translator are processed as early as possible (principle of sequential translation).

1,13222 The subroutine corresponding to a certain variable is called as soon as the input routine has stored the first symbol of the string represented by the variable in C.

1,13223 After the return jump from the subroutine for the processing of a variable, the first symbol of the succeeding syntactic unit is located in C.

On account of the assumptions 1,13211 and 1,13212, these three conventions can always be observed.

1,133 Types of Structures

1,1331 The Structure Corresponding to the Enumerating Definition

The program structure corresponding to the enumerating definition (1)

$$\langle\langle x \rangle\rangle ::= \langle\langle \alpha_1 \rangle\rangle \mid \langle\langle \alpha_2 \rangle\rangle \mid \ldots \mid \langle\langle \alpha_n \rangle\rangle \qquad (1)$$

has the very simple form shown in Fig. 1,1 - 3. In Fig. 1,1 - 4 the

flow chart for the variable < multiplying operator > is shown as one example.

If for<< α >> the function cont (<< α >>,<< α >>) is true, the program has to be made a recursive subroutine (see chapter 1,1334).

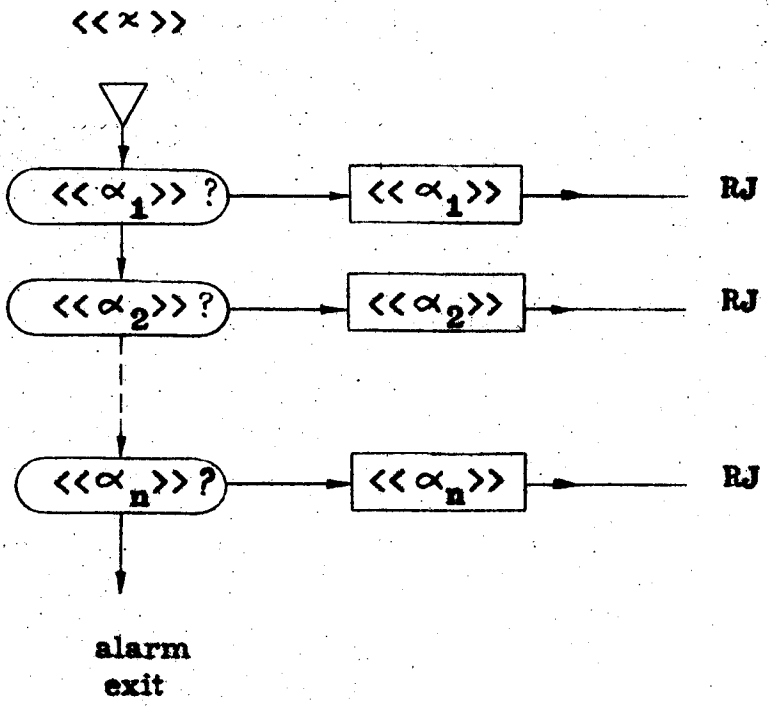


Fig. 1,1 - 3 Programming structure for the enumerating definition

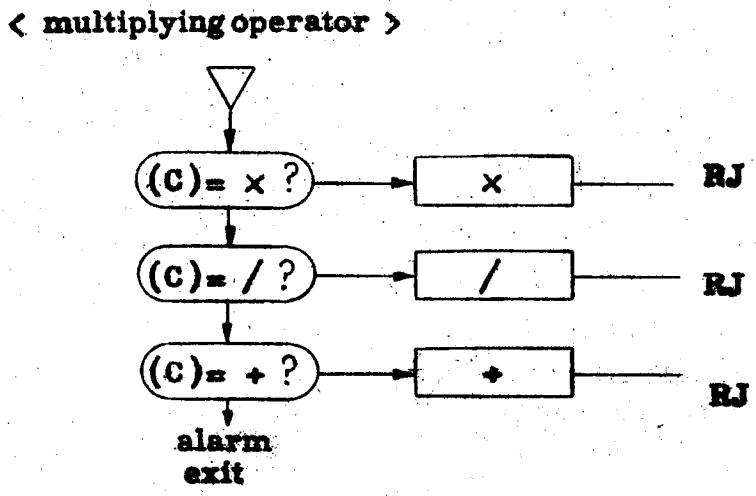


Fig. 1,1 - 4 Example for the enumerating definition

1.1332 The Structure Corresponding to the Juxtaposing Definition

The program structure corresponding to the juxtaposing definition (2)

$$\langle\langle \mathcal{S} \rangle\rangle ::= \langle\langle \alpha_1 \rangle\rangle \langle\langle \alpha_2 \rangle\rangle \cdots \langle\langle \alpha_n \rangle\rangle \qquad (2)$$

is shown in Fig. 1.1 - 5

If the definition of $\langle\langle \mathcal{S} \rangle\rangle$ is recursive, the program has to be made a recursive subroutine (see chapter 1.1334). Chapters 1.1523 and 1.1524 may serve as examples.

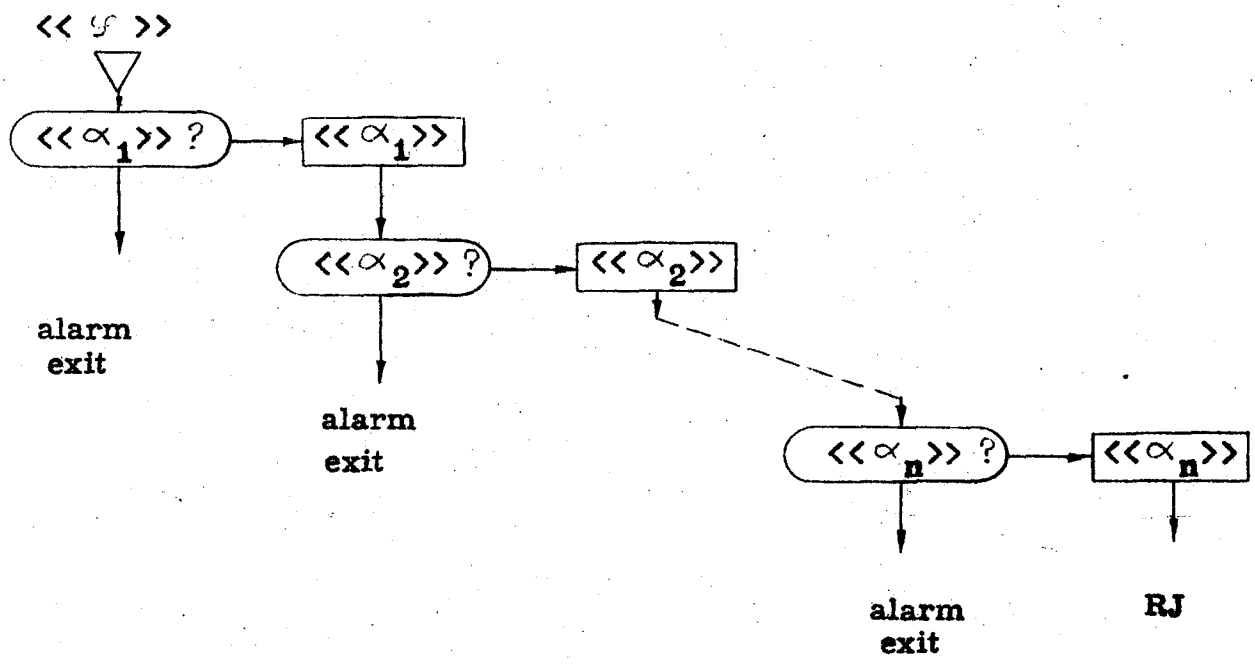


Fig. 1.1 - 5 Programming structure for the juxtaposing definition

1.1333 Explicit Definitions

For the processing of explicitly defined variables the usual subroutine technique is applicable.

1.1334 The Structure Corresponding to the Recursive Definition

1.13341 General Case

The usual subroutine techique is not applicable to the translation of recursively defined variables. As these variables contain themselves,

the subroutine for the processing of the variables also had to contain itself and during the course of the program the return jump as well as the contents of certain auxiliary storage cells would be destroyed.

1.133411 Handling of the Return Jump

It has been shown in chapter 1.1311 that subroutines may embrace themselves but that they cannot have interlacing structures. Therefore the return jumps occur exactly in the reverse succession of their appearence. For this reason it is sufficient to have access to the last return jump. This means that the return jumps of all subroutines can be stored in one single cellar. The cellar principle used for symbols by BAUER and SAMELSON is now applied to the storage of return jumps. The subroutine corresponding to the recursive definition will be called recursive subroutine. The course of recursive subroutines is shown in Fig. 1.1 - 6. R_i denotes the linear storage array for the return jump callar.

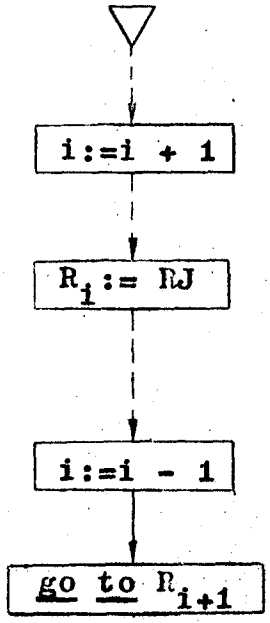


Fig. 1.1 - 6 Course of a recursive subroutine with return jump cellar

1.133412 Handling of Auxiliary Storage Cells : Block Cellar.

Those jumps which lead from a recursive subroutine into a recursive subroutine will be called recursive jumps. An auxiliary cell whose contents have been stored before the occurrence of a recursive jump and still are needed after the return, must be treated like the return jumps in chapter 1.133411, this means that a linear storage array H_k must be organized instead of a single storage cell H. In order to avoid the manipulation with numerous cellars - which would require more storage locations - a single cellar can be used for the return jumps and the auxiliary cells. The cellar principle must be extended for this purpose.

The principle is extended in that way, that not only single cells but blocks of cells are built up and removed in the cellar. The blocks may have different length. The individual cells of such a block can be used as auxiliary cells, and one of these cells can be used for the storage of the return jump.

Each recursive subroutine is assigned one block. The block which is assigned to a certain subroutine is built up at the beginning of this subroutine and removed at the end of the subroutine. This process is shown in Fig. 1.1 - 7 ; B is the storage array which is provided for the block cellar and k is the length of the block.

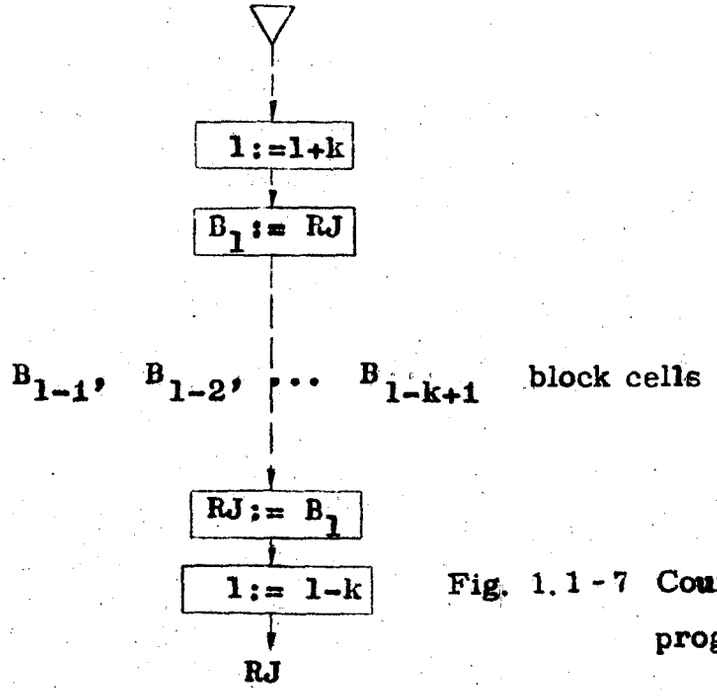


Fig. 1.1 - 7 Course of a recursive program with block cellar

The block cellar cannot be used for those auxiliary storage arrays of a subroutine the length of which depends on the source program. In this case for each of these arrays an auxiliary cellar must be provided and the start address of its blocks be stored in one block cell of the first cellar. This technique may be applied in all cases where lists are required. For example it may be used for the addressbooks in ALGOL, in order to be able to handle the different levels of nomenclature of the "block concept" of ALGOL 60.

1.133413 The Input Parameters for Recursive Subroutines

As input parameters fixed auxiliary cells can be provided. These auxiliary cells are filled before the corresponding recursive jump. If the input parameters shall be preserved beyond a further recursive jump, they must be stored in the corresponding block cells, at the beginning of the subroutine.

1.133414 The Output Parameters of Recursive Subroutines

As for the input parameters, fixed auxiliary cells can be provided for the output parameters. The contents of the cells are defined before the return jump is activated and are taken over by the master program. The output parameters may also be taken from the block cells of the subroutine (immediately after the return jump), since the position of the block cells is known immediately after the return jump.

As the arrangement of the block cells is not necessarily known when a program is designed, it is often more comfortable to process input and output parameters by means of fixed cells.

1.13342 The Structure Corresponding to the Definition of a Chain

The two forms of the definition of a chain were (see chapter 1.12242)

(I) $\langle\langle \varphi \rangle\rangle ::= \langle\langle \alpha \rangle\rangle \mid \langle\langle \beta \rangle\rangle \langle\langle \varphi \rangle\rangle$ (3R)

(II) $\langle\langle \varphi \rangle\rangle ::= \langle\langle \alpha \rangle\rangle \mid \langle\langle \varphi \rangle\rangle \langle\langle \beta \rangle\rangle$ (3R)

This special case of the recursive definition can be realized with the help of two simple program loops (Fig. 1.1 - 8).

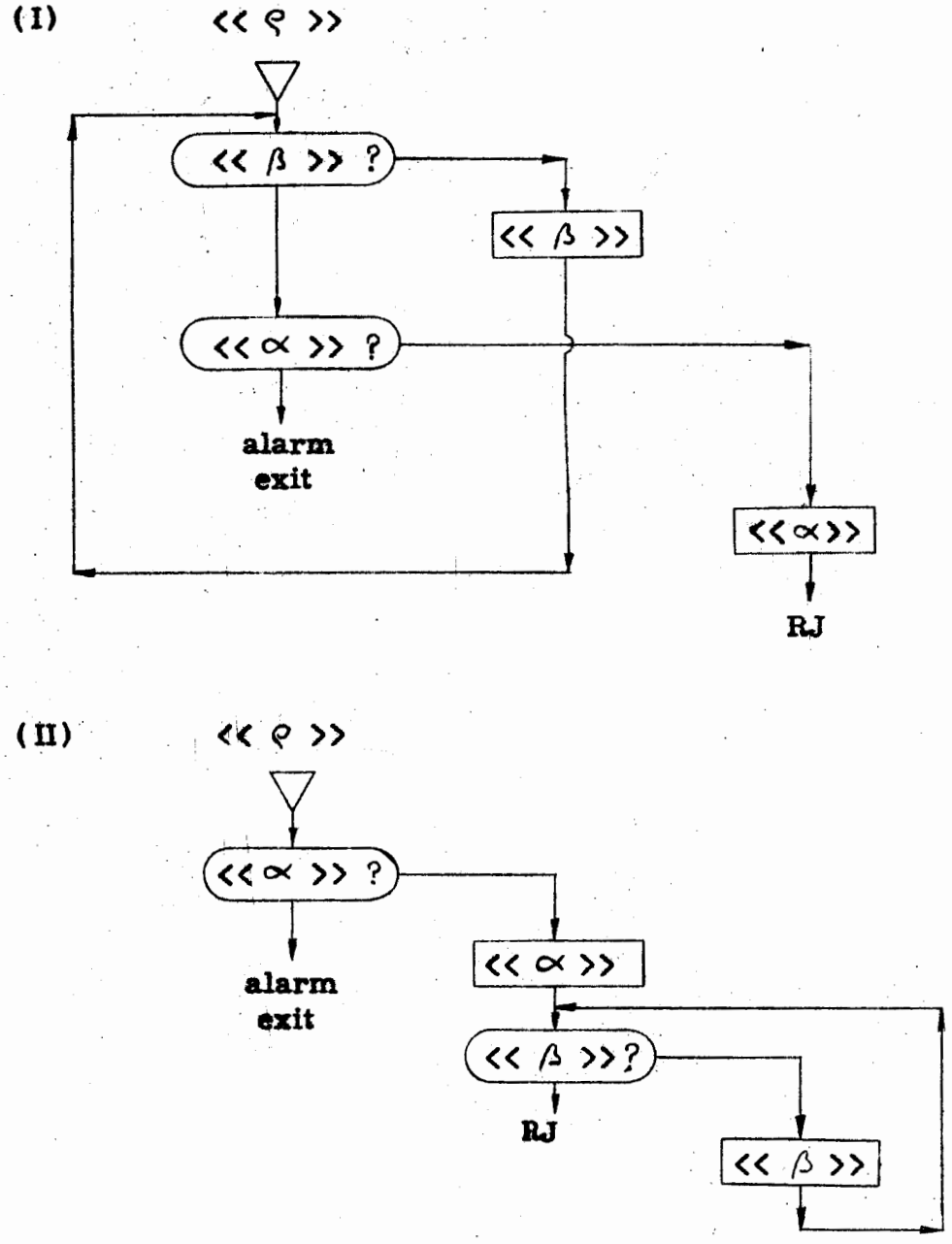


Fig. 1.1 - 8 Program loops corresponding to the recursive definition of a chain

1.134 Simplification by Omitting Redundant Decisions

1.1341 The question for a variable is identical with the question for the first symbol of that variable (see 1.13211). If the answer to the question is an alarm and if the same question, combined with an alarm exit in the subroutine for the variable is posed again at the beginning of this subroutine, one of the questions can be left out.

1.1342 If a subroutine has been called on account of a question for a certain symbol, then is no need to check this symbol within the subroutine.

1.135 Realization of the Syntactic Checking

A syntactic error in the source program leads to one of the alarm exits of the translation program. In order to facilitate localizing of the error, it is useful to mark the alarm exits. One may use for this purpose the name or the number of the corresponding subroutine and the designation of the class of symbols which would be syntactically correct at this point.

The alarm exit can be linked either to a simple stop instruction or to the call of a corresponding printing program. The printing program supplies the information about the syntactic error, for instance the mark of the alarm output, the symbol which was read last (from the storage cell C), the number of statements which have been translated already, and so on.

1.14 Comparison to the Control by Means of a Transition Matrix

1.141 The Transition Matrix by BAUER and SAMELSON

In this chapter the control of a translation process by means of a transition matrix described by F. L. BAUER and K. SAMELSON, will be briefly discussed.

The two main elements of the control are the symbol cellar and the transition matrix. In the symbol cellar symbols for the state of the translator are stored. Each state symbol corresponds to a column and each basic symbol S_i of the object language corresponds to a row of the transition matrix. The functioning will be best described by a citation from the paper of BAUER and SAMELSON[3] :

"The contents of the symbols cellar determines a state which, in effect, at any given time depends on the highest element only, and the new symbol of source information plus state determine output (of machine instructions) and successor state."

1.142 Equivalence of Transition Matrix and Subroutine Structure

The symbol cellar corresponds exactly to the return jump cellar described in chapter 1.133411, because the return jump cellar also determines the state of the translator unequivocally.

Each column of the matrix corresponds to a chain of questions for all basic symbols S_i (see Fig. 1.1 - 9). Therefore for each chain of questions, which result from the program structures described above, a column of the matrix, i.e. a certain state, has to be arranged (also in the case that the chain consists of only one single question).

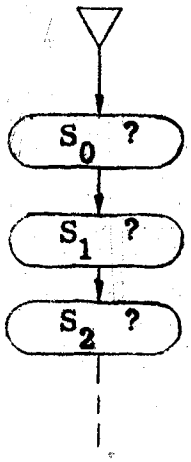


Fig. 1.1 - 9 The chain of questions which is equivalent to a column of the matrix

1,143 Comparison of the Methods

Since in most states only a limited number of basic symbols can occur (and often only a single one), the transition matrix contains many redundant questions. The technique which has been described avoids this disadvantage by means of subroutines. The rules, given in chapter 1,1322, for simplification of the program structure by omitting redundant questions, cannot be applied to the transition matrix; for this reason the transition matrix is always redundaut. Furthermore, in the case of the autonomous subroutines for the translation of the individual units of the language the hierarchy of the units is stressed distinctly, whereas the rows of the matrix somehow assign the same level to all units.

One advantage of the matrix, compared to the subroutine technique, must be mentioned. If a large core store is available, so that the whole matrix can be arranged in the core store, the translation process is accelerated, as the computation of the address of one matrix point within a column can be carried out with one single addition.

1,15 Examples from the Application to ALGOL 60

1,151 Introductory Remark

The assumptions which were made in chapter 1,1321 concerning the syntactic definitions are not satisfied for ALGOL 60 in all items. For example the labels (< label >), labelled statements (<statement>) and certain lists (e.g. < left part list >) must be handled separately.

The following examples were selected, because they contain all types of definitions which have been treated (partially in a combined form). Furthermore, these examples represent the core of the ALGOL - translator. The separate handling of some cases, which has just been mentioned, is not contained in the examples.

1.152 Examples

1.1521 List of Symbols

rSR	recursive subroutine
RJ	return jump
(C)	read character
B1	storage array for the block cellar
l	level of the block cellar
PR	store for the generated program
p	level of the program store
N	number cellar
k	level of the number cellar
(X)	contents of address X
OP	auxiliary cell for the output parameter of the subroutine

Jump instructions, the addresspart of which is not known, are called open jumps.

Subroutines have the names of the metalinguistic variables they translate.

1.1522 Subroutines Which Will Not Be Described in Detail

read (C)

This subroutine transfers the next ALGOL - symbol to C; all delimiters, identifiers and unsigned numbers are treated as one symbol.

< Boolean expression >

The subroutine generates the program for the evaluation of a logical expression. The output parameter of this subroutine is the address of the result of the logical expression in OP.

< simple arithmetic expression >

The subroutine generates the program for the evaluation of an arithmetic expression. The output parameter is the address of the result of the arithmetic expression in OP.

1.1523 Translation of the < if clause >

The if clause is defined in the ALGOL 60 - report in item 3.3.1. With the help of the <if clause > conditional statements and conditional expressions can be formulated.

1.15231 Syntactic Definition

< if clause > :: = if <Boolean expression> then (2R)

1.15232 Analysis

In this case we have a recursive, juxtaposing definition. The definition is recursive, because the logical expression may be a conditional expression (see the example in chapter 1.1225). According to chapter 1.1332 (structure of the juxtaposing definition) three questions had to be posed corresponding to the three constituents of the definiens. However, according to the rules for simplification, the first two questions can be omitted and then remains only the question for the symbol then.

1.15233 The Meaning of the < if clause >

The if clause is always followed by a statement or an expression. The statement or the expression are carried out if the logical expression of the if clause is true. If the logical expression is false, the statement or the expression is overjumped. As the target address of this jump is unknown during the translation of the if clause (open jump), the subroutine delivers the address of the open jump as output parameter.

Fig. 1.1-10 shows the flow of the subroutine.

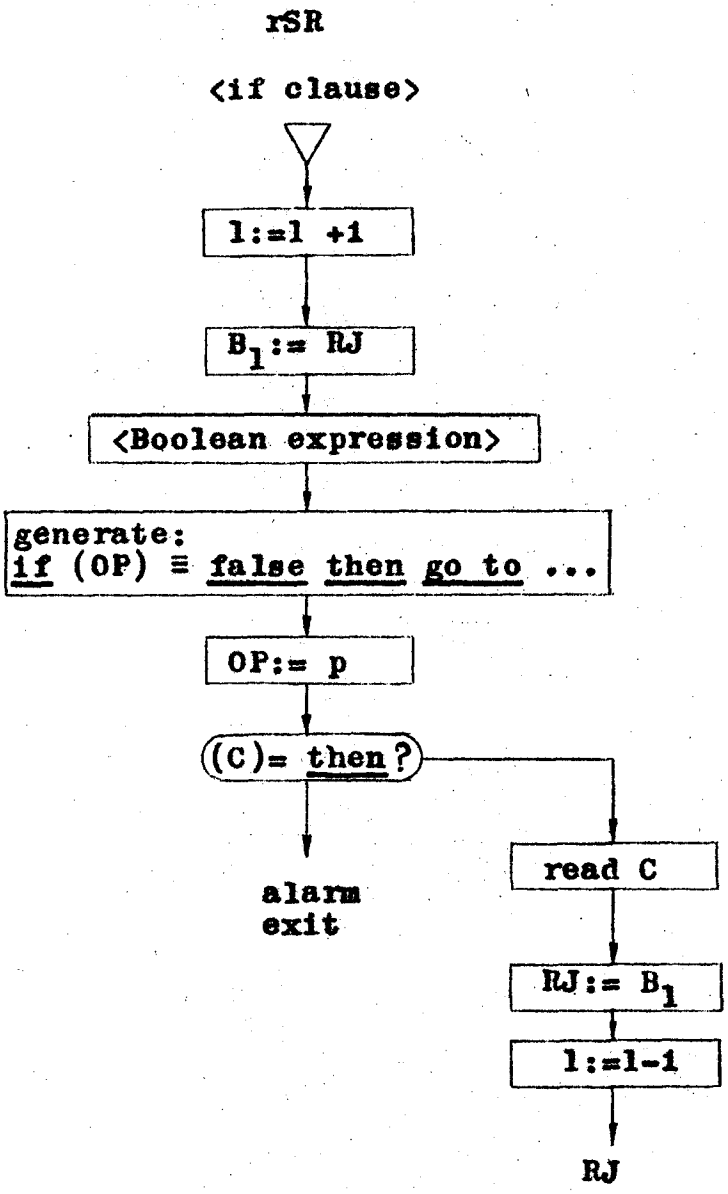


Fig. 1.1 - 10 Subroutine for the translation of the <if clause >

1.1524 Translation of Arithmetic Expressions

The arithmetic expression is defined in the ALGOL 60 - report in item 3.3.1.

1.15241 Syntactic Definition

<arithmetic expression> ::= < simple arithmetic expression> |
| < if clause><simple arithmetic expression >
else <arithmetic expression> (3R)

1.15242 Analysis

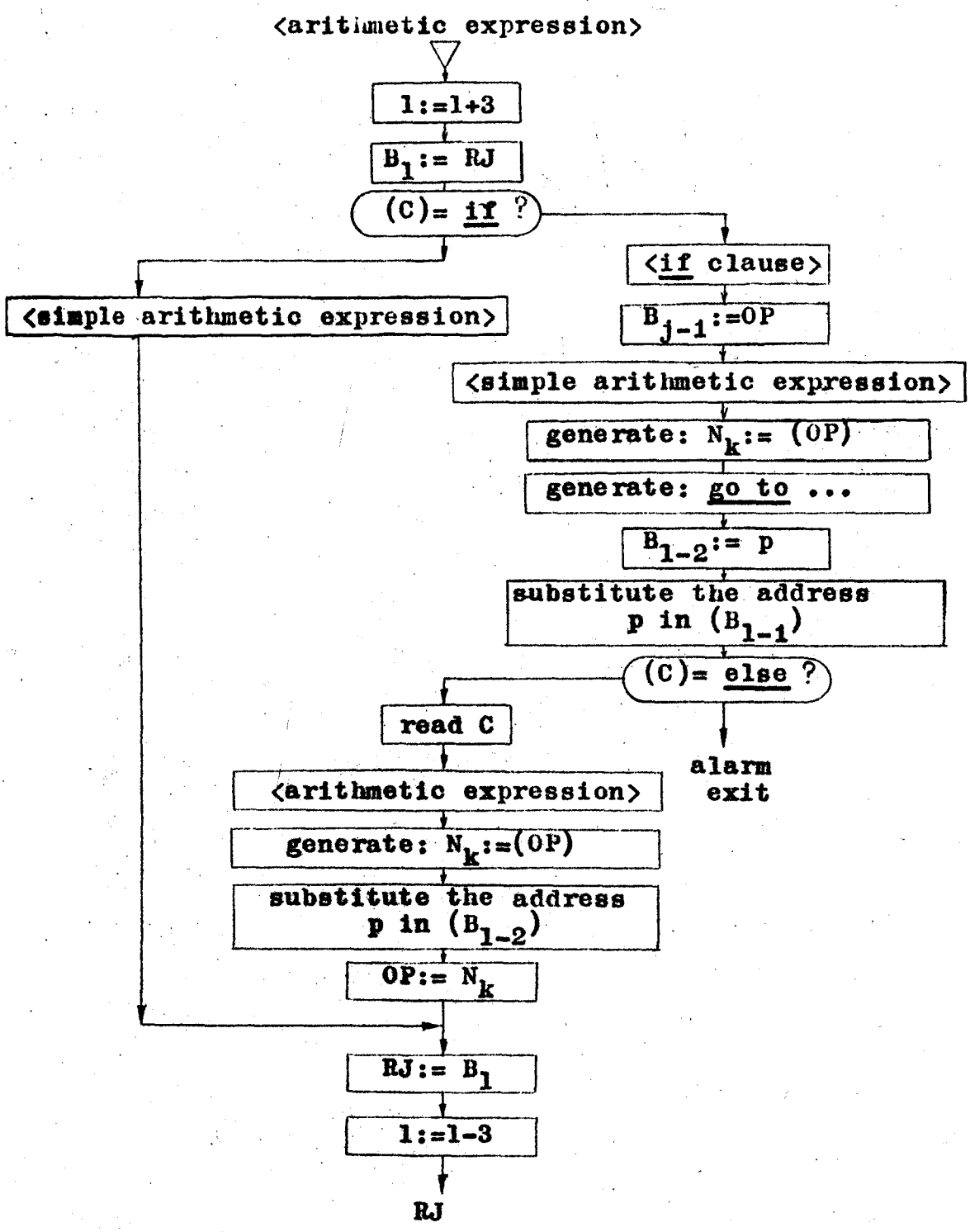
The definition is of the enumerating and juxtaposing type, moreover it is recursive. After the rules for simplification have been applied, there remain only the questions for the symbols if and else.

1.15243 The Meaning of the < arithmetic expression >

An arithmetic expression may be conditional or unconditional. Conditional arithmetic expressions contain two different arithmetic expressions, a first one, which is computed if the condition is satisfied and a second one, which is computed if the condition is not satisfied.

Fig. 1.1 - 11 shows the flow of the subroutine.

rSR



Contents of the block cells:

B_1 return jump

B_{l-1} address of the conditioned jump

B_{l-2} address of the unconditioned jump

Fig. 1.1 - 11 Subroutine for the translation of < arithmetic expression >