ZASTOSOWANIA MATEMATYKI
APPLICATIONES MATHEMATICAE
XII, 4 (1972)

KRYSTYNA JERZYKIEWICZ (Wrocław)

# SEMANTIC ANALYSIS OF AN ALGOL TEXT

TABLE OF CONTENTS

## 1. INTRODUCTION

**1.1. Summary.** The paper contains a description of a method of semantic analysis of a syntactically correct ALGOL text and an algorithm which realizes the method under assumption that during processing the algorithm there are available in the machine storage all the informations concerning the identifiers and constants appearing in the text being analysed. The algorithm has been applied as the third pass of the ODRA-ALGOL compiler, a hardware representation of ALGOL 60 for the ODRA 1204 digital computer. The grammar of the ODRA-ALGOL language and the two first passes of the compiler had been described by Szczepkowicz [13]. The input text for the algorithm is identical with the output text of the second pass of the compiler. The algorithm produces an object program in the ODRA 1204 machine code.

The algorithm is a modification of the known methods of compiling the ALGOL 60 programs in the case of a single-address, one-accumulator digital computer. The generation of the object instructions is performed during a single left-to-right scanning of the source text. The algorithm is controlled by means of a procedure that compares the operator precedences, which are equal to the machine representations of the operators. The algorithm does not contain recursive subroutines and uses a single stack.

The dependence of the algorithm upon the ODRA 1204 computer arises solely from the actual implementation of syntax elements in the above-mentioned hardware representation. The implementation was worked out in collaboration with the author of the first two passes of the compiler and will be given in a separate publication. In the present paper only the meaning of operational codes of the instructions generated by the compiler is explained. The contents of subroutines forming a constant part of object programs is not explained. In order to obtain an object program for a different implementation of syntax elements it is sufficient to change values of the variables which correspond to the operational codes, and — if the realization requires it — to omit or add the generation of some instructions.

**1.2. A Concise Review of Methods of Sem antic Analysis Used in Compilers.** The semantic analysis of a program written in a certain language is a part of the compiler of the language. Therefore, every complete description of a compiler contains a description of the method of semantic analysis used. The method depends to a large extend upon the language being the object of the analysis, and also upon the organization of the whole compiler, as well as upon the form of the input and output texts. There are, however, elements common to all languages or, at last, to large groups of languages. As examples of such elements one can give the arithmetic and Boolean expressions which appear in ALGOL 60, FORTRAN etc. It is therefore possible to speak of compiling the arithmetic and Boolean expressions without reference to the language in which these expressions appear. In the literature one can find a number of papers dealing with methods of compiling the expressions, which does not mean, however, that the methods are so numerous. In fact, there are three basic methods, viz.,

a. Compiling expressions into Reverse Polish notation [3].

b. Recursive compiling of expressions [6].

c. Compiling from a syntax tree [1], [7].

The last of the above-listed methods was never, as far as we know, applied in a compiler. The method requires the generation of a syntax tree during performing the syntactical analysis of a program. Theoretical remarks concerning the semantic analysis of arithmetic expressions stored in the computer memory in the form of a syntax tree were given by Ingerman [7]. He suggested to generate the syntax tree when performing the syntactical analysis. In order to store the syntax tree as a sequence, a table containing informations about the elements of the sequence would be formed. The informations are needed to make it possible to reproduce all branches of the syntax tree. However, such a form of the input text, on which the semantic analysis is to be performed, is inconvenient because during the analysis the text is not scanned, but the elements are taken in the order depending on the table of informations. Therefore, the whole input text must be stored until the semantic analysis is completed.

Several modifications of the method of compiling expressions into Reverse Polish notation and also a generalization of the method for the case of compiling a complete ALGOL 60 program are described in the literature. Dijkstra [3] was first to describe such a generalization. He gives examples how the interpretation of some symbols (e. g., the colon and parentheses) depends upon the context in which the symbols appear (e. g., a colon separating subscripts, bound pairs, actual parameters, or for list elements) and provides methods distinguishing the meaning of symbols in a text, he introduces the table of operator precedence. The generations of object program instructions is performed during a single, left-to-right scanning of the text, in such a way that instructions corresponding to the elements of the source program other that operators are generated immediately after reading in the element, but instructions corresponding to operators are generated while unstacking an operator. At run-time of the program, a stack of accumulators is used, in which values or addresses of arguments are stored, and the operations are performed on the topmost elements of the stack.

An ALGOL 60 compiler based on the same principles is also described in reference [12]. As the result of compiling a program a set of symbols is formed, which corresponds to the program written out in the generalized Reverse Polish notation. The compiler is a single-pass one, so that during compiling a source program into Reverse Polish notation the syntax of the program is also checked. The execution of the object program consists in interpreting the successive symbols of the text and, at the same time, checking whether operations are performable.

Another modification of the Dijkstra method has been applied in the ALGOL 60 compiler to the GIER digital computer [9]. In the case of this compiler, an object program is obtained as a set of the machine code instructions. The generation of the object instructions take place during two (of several) passes of the compiler. The first of them compiles a syntactically correct text of a program into the Reverse Polish notation in such a way that the output text still contains some additional informations, e. g., operator **if** has two representations in the Reverse Polish notation, depending upon whether the operator appears within an expression or not. The second of them generates the object instructions and performs an optimisation. The optimisation consists in substituting for the expressions that are all formed of arithmetic constants, another arithmetic constants, which have the value of the corresponding expressions. Computing a value of an expression actually from left to right is performed only in the case where in the expression there appears a non-standard procedure call or a formal parameter called by name.

The recursive method of compiling expressions has been applied in the two-pass ELLIOTT-ALGOL compiler [6]. The first pass of this compiler consists of a set of recursive procedures that generate an object program as a sequence of instructions in the machine code. In its second pass the compiler completes the addresses of jump instructions.

From the point of view of the user the most important features of a compiler are the compiling-time and the execution-time of the program. It follows from experience that a compiler designed as a set of recursive subroutines compiles a program very slowly. A compiler working according to the principle of comparing operator precedences can be much faster and is able to generate efficient object programs. For these reasons in the third pass of ODRA-ALGOL compiler a method was employed which is a modification of the method given by Dijkstra. A suitable choice of the machine representation of operators made it possible to avoid the table of precedences; the introduction of certain auxiliary operators (Table III) removed the ambiguity of some operators appearing in the source programs. An insignificant departure from the rule of computing the values of expressions from left to right (see 2.3) gave as a result a more compact object programs. The third pass of ODRA--ALGOL compiler produces a sequence of instructions in the ODRA 1204 machine code, equivalent to the source program.

**1.3. Table of Symbols.**

| Symbol | Meaning of the symbol |
|---|---|
| *AccC* | Accumulator contents at the run-time of the program. |
| *ALOC* | Address of the computer location. |
| *LOC*(*A*) | Contents of the location whose address is equal to *A*. |
| *SP* | Topmost element of the working stack of the compiler. |
| '*operator*' | Subroutine corresponding to the operator. |
| *Rj* | Anonymous variables of the object program. |
| *RI* | Address of the reservation indicator. |

**1.4. A Short Description of the ODRA 1204 Digital Computer.** A detailed description of the ODRA 1204 digital computer and its machine code can be found elsewhere [4]. In what follows we shall give — in order to facilitate the understanding of the principles along which third pass of the compiler is working and of the algorithm — some essential features of the computer.

The ODRA 1204 digital computer is a single-address parallel machine. The standard machine configuration for which the ODRA-ALGOL compiler was written consists of the central processing unit (16384 24-bit locations) and the peripherals: the paper tape reader, paper tape punch, and the control typewriter. The ODRA-ALGOL compiler works under the control of a single-program operating system.

The registers of the computer, which will be referred to later on, are the following: a 24-bit accumulator, a 24-bit auxiliary register, and a 16-bit sequence control register. Every location can contain a machine instruction or a fixed-point number. A floating-point number occupies two successive locations. During the execution of operations on the floating-point numbers the accumulator and the auxiliary register work as a single, double-length register.

The locations with the addresses equal to 1, 2, and 3 differ from the others for they can be used as modification registers of the address part of an instruction. If an instruction is modified by the register $n$ ($n = 1, 2, 3$), then to address part of the instruction there is added the contents of the register $n$.

A machine instruction consists of the parts

| *P* | *B* | *OR* | *AR* |
|---|---|---|---|

where *AR* (14 bits) denotes the address part, *OR* (7 bits) is the operational code, *B* (2 bits) denotes modification by the corresponding register (if

$B = 0$, then the instruction is not modified), $P$ (1 bit) is still another kind of modifying the $AR$ part: if $P \neq 0$, then the instruction is executed with its address part equal to $LOC(AR)$, after first modifying $AR$ with the register indicated by $B$, otherwise ($P = 0$) there is no such modification.

**1.5. Informations about the Source Program and their Storage at the Start of the Semantic Analysis.** The semantic analysis of the program is performed only when the program is syntactically correct, i.e., after the second pass of the compiler had been completed. In the machine storage there are then stored these informations about the program which form the output text produced by the second pass. The method of obtaining the informations and their form have been described by Szczepkowicz [13]; the below given summary of the problem is intended to assist the reader in understanding the method of semantic analysis given in the present paper.

The machine store can be divided into the following blocks, containing the informations about the source program and the working space for the third pass of the compiler:

*L0*: Fixed part, common to all source programs (subroutines of the standard functions and of the storage administration).

*L1*: Simple variables, space for the array addresses and the reservation indicators (see 2.5).

*L2*: Strings appearing in the program.

*L3*: Arithmetic constants appearing in the program.

*L4*: Space for the object program (the output text).

*L5*: Text of the program at the completion of the second pass of the compiler (the input text).

*L6*: Working stack of the third pass of the compiler.

*L7*: Description of the standard identifiers and the identifiers declared in the program.

*L8*: The compiler.

During the semantic analysis the space of the three following blocks is changed:

*L4* — The space of this block is increasing because of adding at its end the successive object instructions.

*L5* — Text of the program is only once scanned from left to right, so that during the analysis successive elements of the text (starting with the elements which appear at the beginning of the block) are removed from the block and, therefore, the space of the block is decreasing.

*L6* — The working stack is used in such a way that elements which are to be stacked are added to the end of the block *L6*, and the unstacking an element causes removing the last element from the block. Therefore the space of the block can increase or decrease, depending upon the program; at the start of the analysis block *L6* is empty.

The text of the program (block *L5*) is a sequence of syllables. Every syllable is equal to $IN \times 4 + CN$ (where $0 \leqslant CN \leqslant 3$, $0 \leqslant IN \leqslant 1023$, see Table I and II) and is stored in 12 bits of machine location — each location of the block *L5* contains two successive syllables of the program. The syllables for which there is $CN \neq 0$ will henceforth be referred to as operands. Operands are divided into the following categories: a string, an arithmetic or Boolean constant, and an identifier. The category to which an operand belongs is recognized on the basis of the corresponding value of $CN$, and the value of $IN$ is equal to the address or linear function of the address of the operand (or of its description) in the corresponding block (see Table II). The Boolean constants are an exception: they are stored in block *L0*.

The syllables for which $CN = 0$ are henceforth referred to as operators. The numerical representation of all operators (the value of $IN$) is listed in Table III. One can distinguish the following three categories of operators:

a. Operators appearing in the source programs (e. g., **begin** **if** $+$).

b. Operators introduced during the syntactical analysis instead of the other operators or instead of the standard procedure identifiers (operators numbered $2, 7, 10, 17, 35, 41, 50, 51, \ldots, 84, 128$). To this group should also be added operator number 20 (comma) which appears instead of colon in the array declarations.

c. Auxiliary operators introduced during the semantic analysis (operators numbered $6, 12, 14, 16, 18$).

In a sequence of syllables can also appear operators with the numerical representation equal to zero. They correspond to dummy statements and are ignored by the third pass of the compiler.

The input text of the program does not contain the simple variable descriptions, the formal parameter lists, the value parts, the specification parts, and the declarators defining types of arrays and procedures. These informations are removed from the source program by the first pass of the compiler, and stored separately as the identifier descriptions that form block *L7*. To every identifier declared in the program there is assigned its description. The description occupies four machine locations for a non-standard procedure and two locations for all other identifiers. Quantities appearing in the identifier descriptions are listed in Table IV.

If the compiler is compiling the following program:

**begin**
  **integer** $n$;
  **integer procedure** $S(i)$;
    **value** $i$;
    **integer** $i$;
    $S$: = **if** $i \leqslant 0$ **then** $1$ **else** $i \times S(i-1)$;
  *read* $(n)$;
  *print* $(\text{'}n = \text{'}, n, \text{'}S(n) = \text{'}, S(n))$
**end**

then at the start of the execution of the third pass the blocks *L1*, *L2*, *L3*, *L5*, *L7* will be containing the following informations (the labels are given here in order to make readable the sequence of syllables which forms the input text for the third pass of the compiler):

Block *L1*
  *an*: space for the variable $n$
    space for reservation indicators

Block *L2*
  *s1*: '$n$ = '
  *s2*: '$S(n)$ = '

Block *L3*
  *c1*: 0
  *c2*: 1

Block *L5*

| Number of syllable | *CN* | *IN* | Number of syllable | *CN* | *IN* |
|---|---|---|---|---|---|
| 1 | 0 | 41 | 2 | 0 | 49 |
| 3 | 1 | *d2* | 4 | 0 | 19 |
| 5 | 1 | *d2* | 6 | 0 | 21 |
| 7 | 0 | 42 | 8 | 1 | *d3* |
| 9 | 0 | 31 | 10 | 2 | *c1* |
| 11 | 0 | 3 | 12 | 2 | *c2* |
| 13 | 0 | 15 | 14 | 1 | *d3* |
| 15 | 0 | 37 | 16 | 1 | *d2* |
| 17 | 0 | 43 | 18 | 1 | *d3* |
| 19 | 0 | 33 | 20 | 2 | *c2* |
| 21 | 0 | 4 | 22 | 0 | 19 |
| 23 | 0 | 40 | 24 | 0 | 51 |
| 25 | 0 | 43 | 26 | 1 | *d1* |
| 27 | 0 | 17 | 28 | 0 | 1 |
| 29 | 0 | 19 | 30 | 0 | 40 |

| | | | | | |
|---|---|---|---|---|---|
| 31 | 0 | 57 | 32 | 0 | 43 |
| 33 | 3 | *s1* | 34 | 0 | 4 |
| 35 | 0 | 19 | 36 | 0 | 55 |
| 37 | 0 | 43 | 38 | 1 | *d1* |
| 39 | 0 | 4 | 40 | 0 | 19 |
| 41 | 0 | 57 | 42 | 0 | 43 |
| 43 | 3 | *s2* | 44 | 0 | 4 |
| 45 | 0 | 19 | 46 | 0 | 55 |
| 47 | 0 | 43 | 48 | 1 | *d2* |
| 49 | 0 | 43 | 50 | 1 | *d1* |
| 51 | 0 | 4 | 52 | 0 | 4 |
| 53 | 0 | 1 | 54 | 0 | 2 |
| 55 | 0 | 128 | | | |

This sequence of syllables is equivalent to the following transformed program:

**beginb**
  **procedure** $S$;
    $S:=$ **if** $i \leqslant 0$ **then** $1$ **else** $i \times S(i-1)$;
  **begin readinteger** $(n\}$ **end**;
  **begin printstring** ('$n=$'); **printreal** $(n)$;
      **printstring** ('$S(n)=$'); **printreal** $(S(n))$
  **end**
**endb endprogram**

Block *L7*

*d1*: description of identifier $n$ with the following quantities defined:
  $Type = 0$, $Addr = an$, $IL = 0$, $Decl = 8$

*d2*: description of identifier $S$
  $Type = 0$, $NFP = 1$, $FPLP = d3$, $Addr = 0$, $IL = 1$, $MaxA = 6$,
  $MaxR = 1$, $AInt = 0$, $Decl = 10$

*d3*: description of identifier $i$
  $Type = 1$, $FPN = 1$, $VP = 1$, $Addr = 4$, $IL = 1$, $Decl = 8$

*d4*: descriptions of the standard identifiers

## 2. GENERATION OF THE OBJECT PROGRAM

**2.1. Introductory Remarks.** The main objective of the third pass of the ODRA-ALGOL compiler consists in generating a sequence of machine code instructions equivalent to the syntactically correct ALGOL text (the form of text has been defined in 1.5). This is achieved during a single, left-to-right scanning of the text, with no back-up and no look-ahead.

During compiling it is assumed that there are available all the informations about the identifiers appearing in the text. Henceforth, for the sake of clarity, in the program texts (input and output texts of the programs) we shall use symbolic notation, rather than the machine representation.

During the generation of instructions there is used a single working stack on which we define the following operations:

1. Stacking, i.e., storing a single element on the top of the stack, which causes the increase of the stack space by one element.

2. Unstacking, i.e., removing the topmost element from the stack with the decrease of the stack space by one element.

In the stack there are stored fragments of the compiled text (e.g., still not compiled fragments of expressions) and parameters (e.g., the number of the compiled subscript of a subscripted variable, the address of an instruction, the address part of which is to be completed).

**2.2. An Outline of the Compiling Algorithm.** Let us divide the input text into pairs $\langle$*operand, operator*$\rangle$; the first element of the pair can be empty. The number of pairs formed is equal to the number of operators in the text. If the sequence of syllables after a successive step of analysis has the form $s_l, s_{l+1}, \ldots, s_n$, then the successive pair $\langle$*operand, operator*$\rangle$ is equal to:

$\langle s_l, s_{l+1}\rangle$ if $s_l$ is an operand (in the input text two operands cannot appear in the immediate succession),

$\langle empty,\ s_l\rangle$ if $s_l$ is an operator.

In the pair defined in this way the operand is placed in front of the operator, and the operator follows the operand.

A similar division into pairs had been introduced by Randell and Russell [12], since however the output text in the translator described there is expressed in the generalized Reverse Polish notation, the operand of the pair is the successive element of the output text and is not futrher analysed.

Let us denote any two successive pairs $\langle$*operand, operator*$\rangle$ by $\langle PO, Op\rangle$, $\langle CO, NO\rangle$, respectively; the pair $\langle PO, Op\rangle$ was formed prior to the pair $\langle CO, NO\rangle$ (in the sequence of syllables operator $Op$ appears in front of operand $CO$, and operand $CO$ appears after operator $Op$).

If a fragment of the analysed text has the form

$$a+(b\times c),$$

then the successive pairs (in order in which they are formed) are as follows:

$$\langle a+\rangle \quad \langle empty\ (\rangle\ \langle b\times\rangle\ \langle c\ )\rangle.$$

Let us denote by *C1*, *C2*, and *C3* the classes into which operators are divided in the following way:

*C1.* Operators with their machine representation equal to $39, 40, \ldots, 49$ (see Table III).The operators commence: a go to statement, the compound statement, block, if clause, an expression in parantheses, the subscript list, switch declaration, for statement, labelled statement, the array declaration, the procedure declaration.

*C2.* Operators with their machine representation equal to $1, 2, \ldots, 20$. The operators end some syntax elements of the program, viz., those the compilation of which requires the generation of additional object instructions or performing additional operations in the compiler (e. g., completing a jump instruction, reproducing the parameters stored in the stack).

*C3.* Other operators. Compiling each of the operators causes the generation of instructions which correspond to the operation defined by the given operator (i.e., there is no generation of additional instructions).

During the process of compiling a program there is analysed the pair $\langle PO, Op\rangle$ or there are analysed the two successive pairs $\langle PO, Op\rangle$ and $\langle CO, NO\rangle$.

The third pass of the compiler can be divided into three essential parts, according to the above-mentioned classification of operators, viz.,

*c1*: Set of subroutines which assign values to parameters (and generate instructions, if necessary), if operator *Op* belongs to class *C1*. The subroutines perform also stacking the pair ⟨*operand, operator*⟩, where the operand is generally a parameter, and the operator is defined in the following way:

| Subroutine | The operator at the top of the stack on exit from the subroutine |
|---|---|
| **'go to'** | as before entering (operator **go to** is skipped) |
| **'begin'** | **end** |
| **'beginb'** | **endb** |
| **'if'** | **then** |
| '(' | ) |
| '[' | , |
| **'switch'** | **endswitch** |
| **'for'** | **for-assign** |
| ':' | as before entering (the subroutine fixes only the address of the label appearing in front of the colon) |
| **'array'** | , (the subroutine reads and stores in the stack the array list up to the operator [ which commences the bound pair list) |
| **'procedure'** | **endprocedure** |

*c2*: Set of subroutines executed in the case where operators *Op* and *NO* belong to class *C2*. In the subroutines the instructions are generated, the parameters stored in the stack are unstacked, and sometime new parameters are stacked. In the sequel we give the operators which are on the top of the stack on the exit from a given subroutine. The subroutines are chosen according to operator *Op*.

| 'Subroutine | Topmost operator in the stack |
|---|---|
| '**end**' | if *NO* is **end** then no change, otherwise **end** |
| ,**endb**' | if *NO* is **endb** then no change, otherwise **endb** |
| '**then**' | **else** |
| ')' | *SP* is transferred to ⟨*PO*, *Op*⟩ and the number of elements in the stack is decreased by one |
| '**endswitch**' | if *NO* is comma then no change, otherwise as before execution of '**switch**' |
| '**for-assign**' | **step** |
| '**step**' | *SP* depends on *NO* in the following maner:<br>*NO* — *SP*<br>**step** — **until**<br>**end-for-list-element** — **step**<br>**while** — **while**<br>**do** — **endfor** |
| '**until**' | **end-for-list-element** |
| '**end-for-list-element**' | **step** |
| '**endfor**' | as before execution of '**for**' |
| '**while**' | **step** |
| '**endprocedure**' | as before execution of '**procedure**' |
| '**else**' | if *NO* is **else** then **endelse**, otherwise no change |
| '**endparameter**' | (see 2.4 — compiling the procedure statement) |
| ',' | if *NO* is a comma then no change, if *NO* ends a subscript list then as after executing ')', otherwise (end of a segment of arrays) the subroutine reads in the next pair ⟨*operand*, *operator*⟩ and if the read in operator is a comma then a jump into the body of subroutine '**array**' is executed, otherwise *SP* is such as before the execution of '**array**' |

In the set there are no subroutines corresponding to operators ] **do**}; because the situation that operator *Op* is equal to one of these operators and operator *NO* belongs to class *C2* cannot arise. The above-listed subroutines (with the exception of '**for-assign**') generate also the instructions of loading the machine accumulator with the operand *CO*, if the operand is not *empty* and parameter *Res* (see 2.3) is equal to zero.

*c3*: The part that compiles expressions and the assignment statements (the generation of instructions which correspond to operators of class *C3*) and is controlled by a procedure comparing precedences of operators *Op* and *NO* according to their machine representation (the procedure *Compare* which takes on Boolean values — see Appendix). This part is discussed in detail in paragraph 2.3.

The main features of the scheme of compiling the programs which do not contain procedure identifiers and the formal parameters called by name (handling of such cases will be discussed in paragraph 2.4) can now be presented in the following manner:

```
E1: read (PO, Op);
   comment By reading a pair it is understood here to form the successive
     pair from the input text with, at the same time, shortening the
     input sequence of syllables;
   if Op ≥ 39 ∧ Op ≤ 49 then go to c1 [Op-38];
   comment Operator Op belongs to class C1. The value of the switch
     designator in the go to statement is the label of the respective sub-
     routine from part c1. Return from the subroutine takes place to the
     label E1;
   if Op ≤ 20 then
    begin
     CO: = PO; NO: = Op; go to E3
    end Op ≤ 20;
E2: read (CO, NO);
   if Compare then
    begin
     . . . . . .
     comment Dots denote statements which generate the object instruc-
       tions corresponding to the operator Op;
  E3: Unstack (PO, Op);
     comment Assigning the top of the stack to the pair ⟨PO, Op⟩ with
       decreasing the stack at the same time;
     . . . . . .
    end Compare;
   if Op ≤ 20 ∧ NO ≤ 20 then go to c2[Op];
   comment Operators Op and NO belong to class C2. The go to sta-
     tement chooses, according to the switch designator, the respective
     subroutine of the part c2. After executing subroutines ')' or ',' there
     is a jump to the statement labelled with E2, otherwise to the sta-
     tement labelled with E1;
   Stack (PO, Op);
```

**comment** Increase of the stack and stacking the pair $\langle PO, Op\rangle$;
$PO := CO$; $Op := NO$;
**go to** *E2*;

We give the topmost element, pairs $\langle PO, Op\rangle$, $\langle CO, NO\rangle$, and the subroutine executed during compiling the following statement:

**begin**
*E*: **if** *BE1* **then** $x := AE1$;
  **go to** *E1*
**end**

| topmost element of the stack | | $\langle PO, Op\rangle$ | | $\langle CO, NO\rangle$ | | subroutine |
|---|---|---|---|---|---|---|
| undefined | | *empty* | **begin** | undefined | | **'begin'** |
| *empty* | **end** | *E* | : | undefined | | ':' |
| *empty* | **end** | *empty* | **if** | undefined | | **'if'** |
| par. | **then** | *BE1* | **then** | undefined | | |
| *empty* | **end** | par. | **then** | *BE1* | **then** | **'then'** |
| par. | **else** | *x* | := | undefined | | |
| par. | **else** | *x* | := | *AE1* | ; | the generation of instructions corresponding to the assignment statement |
| *empty* | **end** | par. | **else** | *AE1* | ; | **'else'** |
| *empty* | **end** | *empty* | **go to** | undefined | | **'go to'** |
| *empty* | **end** | *E1* | **end** | undefined | | |
| undefined | | *empty* | **end** | *E1* | **end** | **'end'** |

When this analysis is performed, the following sequence of instructions is generated:

take Boolean *BE1*;
jump if false to *L1*;
take arithmetic *AE1*;
store in *x*;
*L1*: jump to *E1*;

**2.3. Expressions and Assignment Statements.** In the present paragraph we shall not consider the case where the operand is the procedure identifier or the formal parameter called by name (this case is discussed in 2.4).

In what follows we shall not distinguish the operator := from the other operators occurring in expressions, so that the assignment statement will be treated as a particular case of an expression. The described method is a method of generating instructions for a single-accumulator machine. By *accumulator contents* (*AccC*) we shall understand here the accumulator contents during execution of the generated sequence of instructions. Let us introduce some further symbols:

*Rj* — The anonymous variable introduced by the third pass of the compiler in order to store a given *AccC* (if the accumulator is needed for some other purpose) or to store an address (e. g., the address of a subscripted variable). The method of assigning addresses to the variables is presented in 2.5.

*Res* — Parameter different from zero if *AccC* is defined, i.e., if in the accumulator there is value of one of the operands of compiled expression, otherwise $Res = 0$. The value of *Res* is zero before commencing the compiling, it changes to a non zero after the instruction of taking to the accumulator an operand value is generated, and is zeroed after the instruction of storing *AccC* in *Rj* is generated and in subroutines of the part *c2* (with the exception of ')').

*LDA*(*A*) — The generation of the instruction of taking to the accumulator the value of operand *A* (if *A* is a label, then instead of the instruction of taking, there is generated a jump instruction) with assigning a value to the parameter *Res* at the same time.

*GSI* — The generation of the instruction of storing *AccC* in the successive *Rj* with, at the same time, zeroing *Res* and assigning to operand *PO* the operand which denotes the anonymous variable (see Table V).

*Code1*[*n*] — The operational code of the instruction that corresponds to operator *n*, if *AccC* is the operand occurring in front of operator *n*.

*Code2*[*n*] — The operational code of the instruction that corresponds to the operator *n*, when *AccC* is the operand occurring after the operator *n* (e. g., the implementation of unary operators).

*Compile* (*OR*, *AR*) — The generation of the instruction with its operational code is equal to *OR* and its address part equal to *AR* (the way of computing an operand value depends upon the category of the operand — see procedure *Addr1* in the Appendix).

Using the above-introduced symbols, one can write the scheme of compiling the program given in the previous paragraph in the following form:

```
E1: read (PO, Op);
   if Op ⩾ 39 ∧ Op ⩽ 49 then go to c1[Op−38];
   if Op ⩽ 20 then
    begin
     CO: = PO; NO: = Op;  go to E3
    end Op ⩽ 20;
E2: read (CO, NO);
```

```
if Compare then
 begin
  if Res = 0 then
   begin
    if Op = 21 ∨ Op = 26 ∨ Op = 35 then
     begin
      LDA (CO);    go to EX3
     end Op = 21 ∨ Op = 26 ∨ Op = 35
     else LDA(PO)
   end Res = 0;
  Compile (Code1[Op], CO);
E3: Unstack (PO, Op);
  if Compare then
   begin
 EX3: Compile (Code2[Op], PO);    go to E3
   end Compare
 end Compare;
if Op ⩽ 20 ∧ NO ⩽ 20 then go to c2[Op];
if Res ≠ 0 then GSI;
Stack (PO, OP);
PO: = CO;    Op: = NO;
go to E2;
```

And now, as an example, consider compiling the statement

$$x\colon = a+b\times(\textbf{if}\ a>b\ \textbf{then}\ a\ \textbf{else}\ a\times b+a/b);$$

according to the given scheme, it runs as follows:

| topmost element of the stack | | ⟨*PO*, *Op*⟩ | | ⟨*CO*, *NO*⟩ | | *Res* (after performing the analysis) |
|---|---|---|---|---|---|---|
| undefined | | *x* | : = | undefined | | 0 |
| undefined | | *x* | : = | *a* | + | 0 |
| *x* | : = | *a* | + | *b* | × | 0 |
| *a* | + | *b* | × | *empty* | ( | 0 |
| *b* | × | *empty* | ( | undefined | | 0 |
| *empty* | ) | *empty* | **if** | undefined | | 0 |
| par. | **then** | *a* | > | undefined | | 0 |
| par. | **then** | *a* | > | *b* | **then** | 1 |
| *empty* | ) | par. | **then** | *b* | **then** | 0 |
| par. | **else** | *a* | **else** | undefined | | 0 |
| *empty* | ) | par. | **else** | *a* | **else** | 0 |
| par. | **endelse** | *a* | × | undefined | | 0 |
| par. | **endelse** | *a* | × | *b* | + | 1 |
| par. | **endelse** | *b* | + | *a* | / | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *R1* | + | *a* | / | *b* | ) | 1 |
| par. | **endelse** | *R1* | + | *b* | ) | 1 |
| *empty* | ) | par. | **endelse** | *b* | ) | 1 |
| *b* | × | *empty* | ) | *b* | ) | 1 |
| *a* | + | *b* | × | *empty* | ; | 1 |
| *x* | := | *a* | + | *empty* | ; | 1 |
| undefined | | *x* | := | *empty* | ; | 1 |

During compiling, the following instructions will be generated:

take $a$;
compute value of relation $AccC > b$;
jump if false to $L1$;
take $a$;
jump to $L2$;
$L1$: take $a$;
multiply by $b$;
store in $R1$;
take $a$;
divide by $b$;
add $R1$;
$L2$: multiply by $b$;
add $a$;
store in $x$;

It should be pointed out that compiling an expression does not form a separate part of the process of semantic analysis of an ALGOL text (e. g., if clause is compiled in the same manner for the conditional statement, as well as for the arithmetic, Boolean, and designational expressions). An essential role is played here by the procedure *Compare*, which has the value **true** only when the operator *Op* precedence is not smaller than the operator *NO* precedence (values of the procedure *Compare* are given in the Appendix), i.e., when the instructions that implement operator *Op* are to be generated. After generating these instructions the length of the compiled program (stored partly in the stack and partly in the form of a sequence of syllables) is decreased by one pair (the running pair $\langle PO, Op\rangle$). The machine representation of operators (see Table III) is chosen in such a way that the precedence of operators does not decrease with the number assigned to them. Because of that, no separate table of the operator precedences during compiling a text is necessary.

When generating the object instructions for expressions the compiler violates in some instances the rule of computing the expression values from left to right (see the above-mentioned example). The violation serves to decrease the length of the object program and to decrease the number of introduced anonymous variables. The desired order of computing the expression values can always be achieved by introducing additional parentheses. So that the two expressions given below, which differ only from each other for the additional parentheses (and it could be thought that the parentheses change nothing), are compiled in a quite different way, viz.

$a+b\times c$
take $b$;
multiply by $c$;
add $a$;

$(a)+b\times c$
take $a$;
store in $R1$;
take $b$;
multiply by $c$;
add $R1$;

Treating the expressions in such a way is in accordance with the semantics of expressions as given in the Revised Report on ALGOL 60 [11]. The differences between these two expressions are negligible when generating instructions from the Reverse Polish notation and when compiling a program into the Reverse Polish notation. In such situations the authors of different compilers use diverse methods. E. g., Randell and Russell [12] propose to compute the values of operands strictly from left to right, and Naur [10] suggests to compute the operand values from left to right only when during the computing an expression value there can take place a change of the operand value that occurred earlier.

During compiling Boolean expressions there is performed no optimisation which could have been designed in such cases where the value of an expression, e. g., a logical sum, is known already after computing the first component value. The method and examples of such optimisation of compiling the Boolean expressions have been given by Bottenbruch and Grau [2]. The performing such an optimisation by an ALGOL compiler can be regarded as aimless for the following two reasons:

1. The same effect as optimisation by the compiler can always be obtained by a suitable conditional expression inserted in the program.

2. It is impossible to define statistically the effects of computing the optimised expression if there appear in it a procedure identifier or the formal parameter called by name.

**2.4. Procedure Call and Formal Parameters.** In the decsribed here method of the semantic analysis no difference is made between compiling the function designator and the procedure statement, so henceforth we shall discuss only compiling the procedure call, which contains both cases. By a *formal parameter call* we shall understand computing the address of the actual parameter that corresponds to the formal parameter.

The compiling an $n$-parameter procedure call ($n > 0$) runs according to the following scheme:

fix *ALOC* which begins the dynamical reservation in the procedure body and store address *BCP*;
jump to the first object instruction of the procedure declaration;
*BCP*: jump to *EP*;
jump to *AP1*;
jump to *AP2*;
. . . . . .
jump to *APn*;
*AP1*: subroutine of the actual parameter number 1;
*AP2*: subroutine of the actual parameter number 2;
. . . . . .
*APn*: subroutine of the actual parameter number $n$;
*EP*: next object instruction;

where *EP* denotes the label in front of the instruction to which there is transferred after executing the procedure body (for type-procedures, *AccC* is equal to the address of the procedure value). From the $j$-th actual parameter subroutine the return is made to the respective object instruction of the procedure declaration with *AccC* equal to this parameter address. The exceptions are subroutines of parameters with specification **label** and **switch**, from which there is no return to object instructions of the procedure declaration.

The implementation of procedure calls in ODRA-ALGOL compiler is similar to the implementation described by Ingerman [8]. An exception is the part of an object program corresponding to the actual parameter being a switch identifier. In the implementation described by Ingerman from subroutine of such an actual parameter the return is made to the object instructions corresponding to the procedure declaration.

The formal parameter call and procedure call consist of several instructions, after executing of which *AccC* is changed. Therefore, if during the generation of instructions it turns out after reading in the successive pair $\langle$*operand, operator*$\rangle$, that the operand is a procedure identifier or a formal parameter different from the switch identifier, then regardless to operators *Op* and *NO* there are the following operations related with the operand (after performing the operations the transfer is made to the further analysis in such a manner, as if the operand has been neither a formal parameter, nor procedure identifier):

1. If $Res \neq 0$, then in the successive *Rj* the *AccC* is stored.

2. If the operand is the formal parameter, then there are generated the relevant call instructions, and depending upon the value of parameter

*Left* (see Table IV) there are generated the following instructions: if *Left* $\neq 0$ (in the procedure body there occurs the assignment of a value to the parameter), the storing *AccC* in the successive *Rj* as the operand address, otherwise that of assignment to *AccC* the value of *LOC*(*AccC*) and to the parameter *Res* there is assigned a value different from zero.

3. If the operand is a procedure identifier and the operator appearing after it is : =, then there are generated the instructions of assignment to the successive *Rj* the address of the location intended for the procedure value, and to the operand there is assigned a parameter denoting the anonymous variable which contains the address (see Table V), and the transfer is made to perform further analysis.

4. If the operand is a procedure identifier without parameters, then there are generated instructions to call the procedure:

fix *ALOC* from which the dynamical reservation begins in the procedure body and store address *BCP*;
jump to the first object instruction of the procedure declaration;
*BCP*: next object instruction;

and for the procedure with type there are generated the instructions of taking to the accumulator the value of *LOC*(*AccC*) and the parameter *Res* is suitably defined.

5. If the operand is the identifier of the procedure with parameters (the operator that makes pair with the operand is the opening parenthese), then there are generated the instructions to call the procedure as for a procedure without parameters and additionally $n+1$ jump instructions with their address parts empty. On the top of the stack there are stored the procedure identifier and address *BCP*, and for the pair $\langle PO, Op\rangle$ there is assigned $\langle BCP+1$, **endparameter**$\rangle$, afterwards a transfer is made to execute operations related with the beginning of compiling the actual parameter (see *s2* below).

With compiling the procedure call there is connected a subroutine of part *c2*, viz., '**endparameter**', which consists of the two following parts:

*s1*. The generation of instructions connected with the end of the compiling an actual parameter (after executing these instructions *AccC* is equal to the parameter address and there takes place a return to object instructions of the procedure declaration for parameters with their specification different from **label** and **switch**, and for the actual parameter which is a designational expression or switch identifier no instructions are generated).

*s2*. Operations connected with the beginning of compiling an actual parameter and the end of compiling the procedure call. Let us denote by *j* the number of the last compiled actual parameter of a procedure with $n$ parameters (if no parameters were compiled, then $j = 0$). If $j < n$, then there is completed a jump instruction to the subroutine of the $(j+1)$-th actual parameter (according to the value of $PO$), and in the stack there is being stored the pair $\langle PO+1$, **endparameter**$\rangle$. If $j = n$, then according to the parameter stored in the stack there is completed a jump to the end of compiling the procedure call, and for procedures with type there are generated the instructions of assigning the value $LOC(AccC)$ to $AccC$.

For a parameter with the specification **switch** the instructions of the parameter call are generated in the subroutine ',' after compiling the subscript expression (see 2.7).

As an example we give a scheme of compiling the following statement:

$$a: = a \times f(a, \text{'}m\text{'}) + x + a;$$

where $f$ is a non-standard procedure identifier, and $x$ is a formal parameter called by name.

| Topmost element of the stack | $\langle PO, Op\rangle$ | | $\langle CO, NO\rangle$ | | *Res* |
|---|---|---|---|---|---|
| undefined | $a$ | $:=$ | undefined | | 0 |
| undefined | $a$ | $:=$ | $a$ | $\times$ | 0 |
| $a$ $:=$ | $a$ | $\times$ | $f$ | ( | 0 |
| par. **endparameter** | $a$ | , | undefined | | 0 |
| parameter | par. | **endparameter** | $a$ | , | 0 |
| par. **endparameter** | '$m$' | ) | undefined | | 0 |
| parameter | par. | **endparameter** | '$m$' | ) | 1 |
| $a$ $:=$ | $a$ | $\times$ | *empty* | + | 1 |
| undefined | $a$ | $:=$ | *empty* | + | 1 |
| $a$ $:=$ | *empty* | + | $x$ | + | 1 |
| $a$ $:=$ | *R1* | + | *empty* | + | 1 |
| undefined | $a$ | $:=$ | *empty* | + | 1 |
| $a$ $:=$ | *empty* | + | $a$ | ; | 1 |
| undefined | $a$ | $:=$ | $a$ | ; | 1 |

During compiling there are generated the following instructions:

entrance to first instruction of the procedure $f$ declaration;
jump to $ECf$;
jump to $AP1$;
jump to $AP2$;
$AP1$: take address of $a$;
return to the object instructions of $f$ declaration;
$AP2$: take address of '$m$';
return to the object instructions of $f$ declaration;

*ECf*: take *LOC*(*AccC*);
multiply by *a*;
store in *R1*;
call the parameter *x*;
take *LOC*(*AccC*);
add *R1*;
add *a*;
store in *a*;

From the example it follows that the side effects are different in ODRA-ALGOL that in ALGOL 60 (if in the function *f* body the value of variable *a* changes, then to compute the value of entire expression the new value of a will be taken). This is caused by the discussed already violation of the rule of computing expressions from left to right (see 2.3). In order to obtain the side effects in accordance with the Report on ALGOL 60, all the variables whose values can change during computing the value of entire expression because of the side effects should be enclosed in parentheses.

**2.5. Addresses Calculation.** All identifiers occurring in a program (with the exception of the formal parameters with specifications **label**, **switch** and of the formal parameters with specifications **real**, **integer**, **Boolean**, called by name) have the addresses assigned to them (the parameter *Addr* in an identifier description — see Table IV). For the identifier of the simple variables, arrays and formal parameters, the value of *Addr* is fixed during the execution of the first pass of the compiler, where for the identifiers declared outside the procedure bodies *Addr* is equal to the address of some location from the block *L1* (see 1.5), while for the identifiers declared in a procedure body and for the formal parameters it is a relative address (relative to the beginning of the storage space, which is the working space of the procedure during the execution of its body). The address of the beginning of the working space of a procedure is computed dynamically at the start of execution of the procedure body.

The first pass of the compiler introduces also some additional variables, referred to as reservation indicators, which in the run-time of a program contain informations as to the storage space dynamically reserved by the program. Reservation indicators were introduced by Gries [5]. Assigning addresses to the reservation indicators is performed by the first pass of the compiler in the manner which is to be described now.

The main program is the procedure of level zero. If a procedure is declared on level $n$ (i.e., in the body of procedure of level $n$), then its level is equal to $n+1$ (in the ODRA-ALGOL compiler the level of a procedure is limited — it cannot be greater than 3). For each level there are defined the block orders. Every procedure is a block of order zero. A block is of order $n$ ($n > 0$), if it is contained in a block of order $n-1$. The reservation indicators are assigned to blocks separately for each level and in such a way that, for blocks the orders of which are the successive numbers, the reservation indicator addresses of the blocks are also successive numbers, and for the blocks of the same order on the same level the values of *RI* are all the same. For the level zero each value of *RI* is equal to the address of a location from the block *L1*, and for a level greater than zero the *RI* are relative (relatively to the beginning of the working space of a procedure). At run-time of an ODRA-ALGOL object program the reservation indicators of the executed blocks contain the address of the last dynamically reserved location.

In the program given below (on the right-hand side there are denoted the procedure levels) the zero level procedure (i.e. the main program) is assigned to three reservation indicators with the addresses $RI_0$, $RI_1$, $RI_2$ (where $RI_1 = RI_0+1$, $RI_2 = RI_1+1$), because the procedure is a zero order block and contains block *E1* (of order 1) and block *E2* (of order 2); the procedure $f$ of level 1 is assigned to two reservation indicators (a zero order block and block *E3* of order 1); the procedure *f1* of the level 2 is also assigned to two reservation indicators (a zero order block and block *E4* of order 1); and the procedure *f2* of level 2 is assigned to one reservation indicator (a zero order block).

```
E1 : begin
       real x;
       . . . . . .
  E2 : begin
         procedure f;
        E3 : begin
               procedure f1;
              E4 : begin
                     real x;              2
                     . . . . . .                     0
                   end f1;                      1
               procedure f2;
                   . . . . . .            2
               . . . . . .
             end f;
         . . . . . .
       end E2
     end E1
```

The instructions which perform operations on the reservation indicators are generated in the subroutines '**beginb**' and '**procedure**' (the execution of the instructions causes the assignment of an initial value to respective reservation indicator) and in the subroutine ',' after generating instructions of the storage reservation for arrays (the execution of them causes assigning a new value to the reservation indicator, because the storage space reserved dynamically by the program has increased).

The value of *Addr* for the procedure identifiers, labels and switch identifiers is the address of this location in which there occurs the first object instruction corresponding to the procedure declaration, the labelled statement or the switch declaration, respectively; therefore, *Addr* can be defined only during the generation of object instructions. Since generation of an object instruction can require the *Addr* of a still undefined value (a forward reference), the Dijkstra method of changing the address parts [3] has been applied. We shall now give a short description of the method.

In the identifier descriptions, the *Addr* for which are fixed by the third pass of the compiler there appears as additional parameter *IntA* with the initial value equal to zero. The parameter value is set to one at the moment when *Addr* is being defined (in subroutines ':', '**procedure**', '**switch**'). For $IntA \neq 0$ the *Addr* value is the address assigned to the identifier. If in order to generate an instruction a value of *Addr* is needed and $IntA = 0$, then initially the current value of *Addr* will be taken as the address part of the generated instruction, and to *Addr* there is assigned the address of the instruction. Hence, for $IntA = 0$, *Addr* is equal to the address of the last generated instruction, in which $AR$ is to be equal to the address assigned to the identifier, and $AR$ is equal to the address of the preceding such instruction; for the first instruction $AR = 0$.

The just described situation is illustrated in the following example: During compiling the fragment of a program

**go to** $E$;
. . . . . .
**go to** $E$;
. . . . . .
**go to** $E$;

there are generated the instructions

*L1*: jump to 0;
. . . . . .
*L2*: jump to *L1*;
. . . . . .
*L3*: jump to *L2*;

and in the description of identifier $E$ were fixed the values: $Addr = L3$. $IntA = 0$.

In such a manner there arises a sequence of the addresses of locations (the first element of the sequence is zero, and the last one is equal to the running value of *Addr*) in which the address parts are to be changed at the moment, when to a given identifier the address is assigned.

In the object program there appear also the anonymous variables $R_j$ (see 2.3). The address computation for these variables is performed by third pass of the compiler separately for each level (*Rj* is local on a given level). The anonymous variables are used to store the intermediate results when computing the values of expressions, the subscripted variable address, the formal parameters, and the location reserved for the function values, and also to store the return address of the subroutine computing the address of a controlled variable and of the subroutine of a controlled statement. The variables *Rj* play on each level the role of a working stack, with the maximum depth of the stack determined during compiling, while in the run-time of a program the space for the stack is fixed when the levels are changed. Introducing a new *Rj* is therefore connected with increasing the working stack of the compiled program by one or two elements (the latter only in the case of storing an arithmetic value in *Rj*). At the moment the value of *Rj* is used up, the current depth of the stack is decreased by as many locations as was the number of locations occupied by the last *Rj*. Since the maximum depth of the working stack is known only after the completion of compiling the level, during compiling *Rj* is assigned a relative address. In the main program assigning the address to *Rj* starts with one. Since on the zero level the *B* part of the generated instruction word is zero, to the relative address *Rj* a parameter is added, which changes *B*. After compiling whole of the program, *B* is zeroed in these instructions, and to the *AR* part there is added the address of the last instruction of the object program. Hence, the output text of the third pass of the compiler consists of the following two parts:

the generated object instructions,

the space for the working stack of level zero.

In the procedure bodies *Rj* is a local variable, and therefore its address remains relative. The address of the first *Rj* is computed on the basis of a procedure identifier descritpion (in '**procedure**'); it is the working space volume of the procedure as computed by the first pass of the compiler increased by one (the space for values and addresses of the formal parameters — see 2.8, the local variables declared in the procedure body and reservation indicators of a given level). After the completion of compiling the procedure declaration (in '**endprocedure**') its working space is increased by the maximum depth of the working stack.

**2.6. For Statements.** Compiling the for statement in its general form

$$\textbf{for } cv := FLE1, FLE2, \ldots, FLEn \textbf{ do } S$$

where *FLEi* is any for list element, causes the generation of the following sequence of instructions:

jump to *FLE1*;

*CV*: subroutine of computing the address of variable *cv*;

*FLE1*: realisation of the first for list element containing a jump to the subroutine of instruction *S* and a jump to *FLE2* obeyed in the case when the element is exhausted;

. . . . . .

*FLEn*: realisation of the *n*-th for list element containing a jump to the subroutine of instruction *S* and a jump to *EF* obeyed in the case when the element is exhaused;

*S*: subroutine of the controlled statement *S*;

*EF*: next object instruction;

The subroutine *CV* and the jump preceding it are generated only when *cv* is a subscripted variable or a formal parameter called by name (this variable address can change during the execution of for statement).

With compiling a for statement there are connected the following subroutines:

'**for**' and '**for-assign**' which generate the subroutine *CV*;

'**step**' which generates the instructions of assigning the first value to the controlled variable and assigns values to parameters according to the kind of the compiled for list element or prepares compilation of subroutine *S*;

'**until**' which generates instructions of the value *cv* change (for the element of the form $AE_1$ **step** $AE_2$ **until** $AE_3$);

'**end-for-list-element**' and '**while**' which end compiling the successive *FLEi*;

'**endfor**' which ends compiling the subroutine *S*.

The only optimisation performed by the compiler during compiling a for statement consists in generating or not the subroutine *CV*. The decision depends solely upon the form of *cv*.

**2.7. Subscripted Variables and the Array Declaration.** Since in the input text of the third pass of the compiler the colons in the bound pair lists had been substituted with commas, a bound pair list has the form of a subscript list and so compiling an array declaration is connected with compiling a subscripted variable.

To a subscript list there corresponds a sequence of instructions of computing the successive subscript expressions and of storing their values in the successive *Rj*. The instructions of storing are generated in the subroutine ',', which determines also the number of subscript expressions. The number is stored in the stack as a parameter, together with the operator number 20 (comma) and is the address part of the one of the generated instructions of computing the address of a subscripted variable. The instructions are generated also in the subroutine ',' in the moment when *NO* is ] and *NO* actually ends the subscript list. In order to recognize whether the compiled subscript list is a bound pair list there is used the parameter *AD*, which has a negative value after compiling a bound pair list. At the start of compiling a program parameter *AD* is equal to zero, in the subroutine '[' is increased by one, and after compiling a subscript list is decreased by one. Additionally, *AD* is decreased by one in the subroutine '**array**', in which there are read in and stored in the stack all the pairs ⟨*operand, operator*⟩ for which the operator is a comma, and subsequently the transfer is made to execute the subroutine '['.

After compiling a bound pair list ($AD < 0$) to the parameter *AD*, zero is assigned and there are generated instructions to form the array segment vector (informations about the values of bound pairs and the array dimension, equal to one-half of the determined number of subscripts) and to reserve the necessary store for arrays, whose identifiers are stored in the stack. Next, in order to find out if the array declaration is already compiled, the next pair ⟨*operand, operator*⟩ is read. If the operator turns out to be a comma, then compiling the next segment begins (transfer is made to the subroutine '**array**'), otherwise the declaration is already compiled.

In the subroutine '[' there is stored in the stack the operand appearing in front of [ and, additionally, as the topmost element, the pair ⟨1 ,⟩. If the operand stored in the stack is a switch identifier, then after compiling the subscript list (in the subroutine ',') there is generated the instruction of jump according to the switch designator, and for the formal parameter with the specification **switch** there are generated instructions of the formal parameter call (see 2.4) with *AccC* equal to the subscript expression value.

**2.8. Procedure Declaration.** Before starting to compile a procedure body there are generated the object instructions corresponding to its heading, i.e., the storage administration instructions and instructions connected with the formal parameter list. The generation of the instructions and some additional operations are performed in the subroutine '**procedure**':

1. There is generated the jump instruction that skips the object instructions corresponding to the procedure declaration.

2. The next pair ⟨*operand, operator*⟩, which has the form ⟨*procedure identifier* ;⟩ is read in (see 1.5), and from the procedure identifier description there are obtained all the informations about this procedure heading.

3. There are generated the storage administration instructions.

4. There are analysed the formal parameters of the procedure (on the basis of their descriptions, in the order they appear in the formal parameter list), and for the parameters listed in the value part, as well as for strings and arrays called by name there are generated call instructions (during execution a procedure body in the implementation addopted for the ODRA-ALGOL compiler the address of the beginning of a string and of an array cannot change). The difference between the parameter $A$ call and the parameter $B$ call, where $A$ is the formal array called by value and $B$ is the formal array called by name, consists in the following: elements of the array $A$ are transferred to the working space of the procedure, while for $B$ only the array address is stored. In the descriptions of these formal parameters, for which the calls were generated, the parameter *Type* is set to zero (see Table IV). Hence, if one of these parameters will play the role of the operand during compiling the procedure body, then it will not be recognized as the formal parameter.

5. In the stack there are stored the informations which are reproduced in the subroutine '**endprocedure**' when compiling the procedure declaration is completed (e. g., the address of the running $Rj$, the address of the jump which skips the procedure declaration). After compiling a type-procedure, in subroutine '**endprocedure**' there is generated the instruction to load the accumulator with such a value $ALOC$ that $LOC(ALOC)$ is the procedure value. The last object instruction corresponding to the procedure declaration is the exit instruction.

As an example consider the following procedure declaration:

**real procedure** $G(A, i, j)$;
  **value** $i$;
  **integer** $i, j$;
  **array** $A$;
    $G := A[i \times i, j]$;

The corresponding object instructions are the following:

jump to $EG$;
instructions of the storage administration;
call the parameter $A$;
call the parameter $i$;
assign $AG$ to $R1$;
take $i$;
multiply by $i$;
store in $R2$;
call the parameter $j$;
take $LOC(AccC)$;
store in $R3$;
compute and store in $R2$ the address $A[R2, R3]$;

take $LOC(R2)$;
store in $LOC(R1)$;
take $AG$;
exit from the procedure;
$EG$: next object instruction;

By $AG$ there is denoted the address of the location reserved for the procedure $G$ value.

The system of storage administration for procedures has been described by Watt in [14]. In what follows we give some essential features of the system in connection with the ODRA-ALGOL object programs.

All the variables local in a procedure body and formal parameters whose calls appear before the execution of the procedure body (e. g., parameters called by value, parameters with the specification **string**) are assigned to relative addresses (see 2.5). The generated instruction, in which the address part $AR$ is relative address, has a non zero part $B$ (see 1.4). More accurately, $B$ is equal to the level $n$ ($n \leqslant 3$) on which an identifier (or variable) has been declared (or introduced by compiler), and in the run-time of the program $LOC(n)$ is equal to the address of the location that is the first of those reserved for the working space of the procedure. Storing the current value (it is restored at the exit from the procedure) and assigning a new one into the modification register $n$ is performed by means of the instructions connected with the procedure call (see 2.4); it is increased by one value of the reservation indicator of the block, in which the procedure call has occurred. After the procedure is entered, the instructions of the storage administration assign a value to the reservation indicator assigned to the procedure. If, therefore, a procedure is a recursive one, then a repeated call causes a change of the corresponding modification register value, i.e., the reservation of the storage locations for a new working space of the procedure, and exit from the procedure causes the assignment of the previous value to this reservation indicator (a return to the previous working space of the procedure) and the transfer to execute instructions appearing after the procedure call. During compiling a program the compiler does not distinguish the recursive procedures from the other procedures, for the recursivity is already guaranted by the implementation adopted in the ODRA-ALGOL compiler.

**2.9. Switch Declaration.** The set of the object instructions corresponding to the switch declaration is similar to the set of object instructions corresponding to the procedure with parameters (see 2.4), in which instructions of storage administration are skipped. Therefore the method of compiling the switch declaration is the same as the method of compiling the procedure call, so that subroutines **'endswitch'** and **'endparameter'** have common parts.

To the switch declaration

**switch** $S := DE1, DE2, \ldots, DEn$

there correspond the following object instructions:

jump to *ES*;
jump to *TDE1*;
jump to *TDE2*;
. . . . . .
jump to *TDEn*;
*TDE1*: object instructions corresponding to the expression *DE1*;
*TDE2*: object instructions corresponding to the expression *DE2*;
. . . . . .
*TDEn*: object instructions corresponding to the expression *DEn*;
*ES*: the next object instruction of the program;

Subroutine '**endswitch**' performs the following optimisation: if the currently compiled expression *DEi* is a label *E* which is not a formal parameter (object instructions corresponding to *DEi* occupy in this case one jump instruction), then the instruction — jump to *TDEi* — is changed into the instruction — jump to *E* — and there are separately generated no object instructions corresponding to the expression *DEi*.

## 3. TABLES

Table I. Values of the Parameter *CN* when it is a Part of a Syllable

| Value of *CN* | Element of the source program |
|---|---|
| 0 | an operator |
| 1 | an identifier |
| 2 | an arithmetic or Boolean constant |
| 3 | a string |

Table II. Values and Interpretation of the Parameter *IN* of a Syllable

| Value of *CN* | Interpretation and value of *IN* |
|---|---|
| 0 | the number of an operator (see Table III) |
| 1 | one half of the relative address of the identifier description |
| 2 | *IN* = the Boolean value **false**<br>*IN* = the Boolean value **true**<br>*IN* > $IN \times 2 - 3$ is the relative address of the arithmetic constant |
| 3 | the relative address of the string |

Table III. Machine Representation of Operators

| Operator number | Operator | Remarks |
|---|---|---|
| 1 | end | end which ends a compound statement |
| 2 | endb | end which ends a block |
| 3 | then | |
| 4 | ) | |
| 5 | ] | |
| 6 | endswitch | an auxiliary operator while compiling a switch declaration |
| 7 | for-assign | stands for the assignment symbol in a for clause |
| 8 | step | |
| 9 | until | |
| 10 | end-for-list-element | stands for the comma which separates for list elements; also appears before do |
| 11 | do | |
| 12 | endfor | an auxiliary operator while compiling a for statement |
| 13 | while | |
| 14 | endprocedure | an auxiliary operator while compiling a procedure declaration |
| 15 | else | |
| 16 | endelse | an auxiliary operator while compiling statements or expressions containing an if clause |
| 17 | } | stands for the closing round bracket in the read a number statement |
| 18 | endparameter | an auxiliary operator while compiling a procedure call |
| 19 | ; | |
| 20 | , | |
| 21 | := | |

| | | |
|---|---|---|
| 22 | ≡ | |
| 23 | ⊃ | |
| 24 | ∨ | |
| 25 | ∧ | |
| 26 | ¬ | |
| 27 | > | |
| 28 | ≥ | |
| 29 | ≠ | |
| 30 | = | |
| 31 | ≤ | |
| 32 | < | |
| 33 | − | |
| 34 | + | |
| 35 | neg | stands for unary minus |
| 36 | / | |
| 37 | × | |
| 38 | ↑ | |
| 39 | go to | |
| 40 | begin | begin which commences a compound statement |
| 41 | beginb | begin which commences a block |
| 42 | if | |
| 43 | ( | |
| 44 | [ | |
| 45 | switch | |
| 46 | for | |
| 47 | : | |
| 48 | array | |
| 49 | procedure | |
| 50 | readreal | the read-real-number operator |
| 51 | readinteger | the read-integer-number operator |
| 52 | readarray | the read-real-array operator |
| 53 | readintegerarray | the read-integer-array operator |
| 54 | readBooleanarray | the read-Boolean-array operator |
| 55 | printreal | the print-arithmetic-value operator |
| 56 | printarray | the print-arithmetic-array operator |

| | | |
|---|---|---|
| 57 | printstring | the print-string-or Boolean-array operator |
| 58 | line | the standard procedure line |
| 59 | space | the standard procedure space |
| 60 | format | the standard procedure format |
| 61 | affix | the standard procedure affix |
| 62 | outdev | the standard procedure outdev |
| 63 | outchar | the standard procedure outchar |
| 64 | wait | the standard procedure wait |
| 65 | indev | the standard procedure indev |
| 66 | number | the standard function number |
| 67 | char | the standard function char |
| 68 | button | the standard function button having a Boolean value |
| 69 | abs | the standard function abs |
| 70 | sign | the standard function sign |
| 71 | entier | the standard function entier |
| 72 | sqrt | the standard function sqrt |
| 73 | sin | the standard function sin |
| 74 | cos | the standard function cos |
| 75 | tan | the standard function tan |
| 76 | arcsin | the standard function arcsin |
| 77 | arccos | the standard function arccos |
| 78 | arctan | the standard function arctan |
| 79 | ln | the standard function ln |
| 80 | log10 | the standard function log10 |
| 81 | exp | the standard function exp |
| 82 | exp10 | the standard function exp10 |
| 83 | max | the standard function max |
| 84 | min | the standard function min |
| 128 | endp | the operator placed after the last end of the source program |

Table IV. Quantities Occurring in the Identifier Descriptions

| Number | Quantity | Element of the source program for which a value is defined | Values |
|---|---|---|---|
| 1 | Type | all identifiers | 1 - for the formal parameters<br>0 - for other identifiers |
| 2 | FPN | formal parameters | number of the parameter on formal parameter list |
| 3 | VP | formal parameters | 1 - for parameters called by value<br>0 - otherwise |
| 4 | AEAP | formal parameters of the specification _real_, _integer_, or _Boolean_, called by name | 1 - if at last one of the actual parameters is not a variable<br>0 - otherwise |
| 5 | Left | as above | 1 - if to the formal parameter there is assigned a value in the procedure body<br>0 - otherwise |
| 6 | NFP | procedure identifiers | the number of formal parameters of a procedure |
| 7 | FPLP | procedure identifiers | one half of relative address of the first formal parameter description (the address of the description of the next formal parameter is smaller by two from the previous) |
| 8 | Addr | all identifier except the formal parameters of the specification _real_, _integer_, _Boolean_, _label_, or _switch_ called by name | the address assigned to an identifier; for the formal parameters, variables, and arrays declared in a procedure body the addresses are relative |
| 9 | IL | all identifiers except labels and switches | number of the level on which the identifier is declared |

| | | | |
|---|---|---|---|
| 10 | MaxA | procedure identifiers | the number of storage locations reserved for: the variables declared in a procedure body, formal parameters called by value, formal parameters of the specification <u>array</u>, <u>integer array</u>, <u>Boolean array</u>, or <u>string</u> called by name, and a function value |
| 11 | MaxR | procedure identifiers | the number of reservation indicators of a procedure |
| 12 | AInt | procedur, label, and switch identifiers | 0 - if the address of an identifier is not yet set<br>1 - otherwise |
| 13 | SLL | switch identifiers | the length of a switch list in a switch declaration |
| 14 | Decl | all identifiers | 0 - for labels<br>1 - for switches<br>2 - for procedures<br>3 - for strings<br>4 - for the Boolean variables<br>5 - for the Boolean arrays<br>6 - for the Boolean functions<br>8 - for the integer variables<br>9 - for the integer arrays<br>10- for the integer functions<br>12- for the real variables<br>13- for the real arrays<br>14- for the real functions |

Table V. The Operand Values during the Analysis

| Operand value | Element of the source program represented by the operand | Parameter $A$ stands for |
|---|---|---|
| $(16 \times Type(A) + Decl(A)) \times 2048 + A$ | identifier declared in the program | relative address of the identifier description |
| 0 | operand is empty | |
| $4 \times 2048 + 2$ | Boolean value **true** | |
| $4 \times 2048$ | Boolean value **false** | |
| $7 \times 2048 + A$ | string | relative address of the string on the string list |
| $15 \times 2048 + A$ | arithmetic constant | relative address of the constant on the constant list |
| $(8+3) \times 8192 + A$ | anonymous variable containing a real value | relative address of the anonymous variable |
| $(8+2) \times 8192 + A$ | anonymous variable containing an integer value | as above |
| $(8+1) \times 8192 + A$ | anonymous variable containing a Boolean value | as above |
| $(12+3) \times 8192 + A$ | anonymous variable containing the address of a real value | as above |
| $(12+2) \times 8192 + A$ | anonymous variable containing the address of an integer value | as above |
| $(12+1) \times 8192 + A$ | anonymous variable containing the address of a Boolean value | as above |

**Note.** From the above-mentioned operand values it follows that if an operand is written in the form

$$((((((a \times 2 + b) \times 2 + c) \times 2 + d) \times 2 + e) \times 2 + f) \times 2 + g) \times 2048 + A,$$

where $a, b, c, d, e, f, g$ take on the values 0 or 1 (denote the successive bits), and $0 \leqslant A < 2048$, then from the parameter values the program element represented by the operand can uniquely be determined.

**Acknowledgments**

The author is greatly indebted to Dr. S. Paszkowski and Dr. W. M. Turski for the valuable discussion of the subject-matter of the paper. Especial thanks are due to Dr. W. M. Turski for many helpful suggestions on reading an earlier version of the manuscript.

The author would also like to thank Dr. J. Szczepkowicz who has helped to overcome same difficulties encountered in the course of preparing the third pass of the ODRA-ALGOL compiler.

Appendix. The Procedure which Performs The third Pass of the ODRA-ALGOL Compiler

The identifiers in round brackets at the right hand side of a page do not belong to the algorithm; they are used for reference purpose. If before the semicolon which terminates a comment there appears an identifier in round brackets, the comment pertains to the lines succeding it, down to the line indicated by the identifier.

<u>procedure</u> Pass3;

<u>comment</u> The global quantities in the body of the procedure Pass3.

a. The addresses of Boolean values and of working locations used by the object program.

FALSE The Boolean value <u>false</u>.

TRUE The Boolean value <u>true</u>.

ADS,C,R,S1 Working locations.

b. The addresses of the fixed parts of the object program.

FIN The exit from the object program due to normal completion.

BFNS The part which causes the message that the value of a function is undefined.

KPP The exit from the actual parameter subroutine.

KP[i] (i=1,2,3) The exit from a procedure of the level i.

c. The addresses of subroutines used by the object program.

Variable Function of the associated subroutine

START Setting the initial parameter values before executing an object program.

CCZP Rounding off the accumulator value.

INDP The computation of the switch designator address and the jump to a corresponding place of the program.

PRPI The reservation of storage locations, copying and rounding off elements of the integer array listed in the value part.

PRPT The reservation of storage locations and copying the real or Boolean array listed in the value part.

RSA Forming a vector of informations about the bound pair list.

AST The reservation of storage locations for the array according to the information stored by the subroutine RSA.

WEN Setting parameters before leaving a level j to execute the body of a procedure which has the level i ($j \leq i$).

PP[i] ($i=1,2,3$) Resetting parameters before the execution of a procedure body on the level i.

WE[i,j] ($j>i$, $j=2,3$, $i=1,2$) Setting parameters before leaving level j to execute the procedure body on the level i.

WY[i,j] ($j>i$, $j=2,3$, $i=1,2$) Resetting parameters after the exit from a procedure of the level i to the level j.

PF[i] ($i=1,2,3$) Setting parameters before calling a formal parameter with the specification other than label or switch of a procedure of the level i.

PFM[i] ($i=1,2,3$) Setting parameters before calling a formal parameter with the specification label or switch of a procedure of the level i.

d. The variables which have values equal to the operational codes of the ODRA 1204 digital computer (the detailed information concerning the codes is given in the reference [4]).

ROPO The zero programmed operation. The operation body is a fixed part of the for statement having its controlled variable of the real type and its for list element of the form: $AE_1$ step $AE_2$ until $AE_3$, where $AE_i$ (i=1,2,3) denotes an arithmetic expression.

ROP2 The second programmed operation. The operation body is equivalent to that of the operation ROPO, but it is executed when the controlled variable is of the integer type.

ROP7 The seventh programmed operation. The computing of the subscripted variable address.

ZEAW The clearing of the accumulator contents and of the auxiliary register contents.

SKB Unconditional jump.

USAN Loading the accumulator with a value of an address.

USAK Loading a fixed-point number.

UZAK Loading a floating-point number.

PSKA Storing a fixed-poin number.

PZKA Storing a floating-point number.

SKP Subroutine jump and storing the return address.

MOK1 Modification of the address part of the next instruction by a location contents.

MOA1 Modification of the address part of the next instruction by the accumulator contents.

SSAN Adding an address to the accumulator contents.

ODKJ Decreasing by one a location contents.

SKZ Jump if the result of the previous operation was zero.

SKD Jump if the result of the previous operation was positive.

PLR Storing the sequence control register contents increased by two.

USWK Loading auxiliary register.

PAKW Storing an auxiliary register contents.

UBK[i] (i=1,2,3) Storing a location contents in the i-th modification register.

Code1[i] (i=22,23,...,38) The operational code that is an implementation of the i-th operator (see Table III) if in the accumulator there is the value of an operand appearing in front of the operator (for the unary operators the accumulator value is zero).

Code2[i] (i=22,23,...,38,50,51,...,84) The operational code that is an implementation of i-th operator (see Table III) if in the accumulator there is a value of the operand which appears after the operator.

e. The labels.

AssEr The error mesage. It results if in the source program the variables and the procedure identifiers of a left part list of an assignment statement have diverse types, or if an actual parameter corresponding to the formal parameter for which Left=1 (see Table IV) is not a variable.

MEr The mesage of an erroneous operation of the computer. The label is equal to such a value of the switch designator which cannot occur if the execution of the body of the procedure Pass3 is correct.

f. Other global variables and arrays.

RI Address of the reservation indicator (see 2.5). Initial value of the variable is defined by the second pass of the compiler.

NLN The current value of this variable is printed together with the mesage of an error in the source program. It is being increased at the start of compiling a block, for statement, and a procedure declaration.

ASL Address of the beginning of the string list.

ACL Address of the beginning of the constant list.

OPA Address of the currently generated instruction. Before the execution of the Pass3 procedure body the value of this variable is equal to the address of the last location occupied by block L3 (see 1.5).

Store The machine core storage, considered as an one-dimensional array.

S The working stack, considered as an one-dimensional array;

```
begin
 integer procedure RSyl;
  comment The function is equal to the successive syllable
  of the text of the source program, e. g., IN×4+CN (see
  Table I and Table II);
  code;
 procedure Compile(i,i1);
  value i,i1;
  integer i,i1;
```

```
 comment The generation of an object instruction, for which
 operational code is equal to i, and the address part is
 equal to i1. The execution of the procedure body causes
 also an increase of value of the variable OPA. E. g. The
 execution of the statement Compile(SKP,OPA+3) causes the
 generation of the instruction of the unconditional jump
 to the location having its address equal to OPA+3, and
 next the increase by 1 the value of the variable OPA;
 code;
procedure UID(i,i1,i2);
 value i,i1,i2;
 integer i,i1,i2;
 comment The assignment of the value i1 to the i-th quan-
 tity in the description of the identifier determined by
 the parameter i2 (see Table IV), e. g., the execution of
 the statement UID(12,1,PO) causes the assignment of value
 1 to the quantity AInt from the description of the identi-
 fier corresponding to the operand PO;
 code;
comment The value of each of the fourteen given below
functions is equal to the value of the quantity from the
identifier description indicated by the procedure parame-
ter. Name of quantity is identical with the procedure
identifier. E. g. the value Type(PO) is equal to the value
of quantity Type from the identifier description indicated
by operand PO;
integer procedure Type(i);
 value i; integer i;
 code;
integer procedure FPN(i);
```

```
 value i;  integer i;
 code;
integer procedure VP(i);
 value i;  integer i;
 code;
integer procedure AEAP(i);
 value i;  integer i;
 code;
integer procedure Left(i);
 value i;  integer i;
 code;
integer procedure NFP(i);
 value i;  integer i;
 code;
integer procedure FPLP(i);
 value i;  integer i;
 code;
integer procedure Addr(i);
 value i;  integer i;
 code;
integer procedure IL(i);
 value i;  integer i;
 code;
integer procedure MaxA(i);
 value i;  integer i;
 code;
integer procedure MaxR(i);
 value i;  integer i;
 code;
integer procedure AInt(i);
```

```
   value i;  integer i;
   code;
  integer procedure SLL(i);
   value i;  ineger i;
   code;
  integer procedure Decl(i);
   value i;  integer i;
   code;
  integer Op,NO,PO,CO,AD,PH,AssT,J,MaxJ,ModJ,CV,CVA,
          CPS,CPDI,NAP,Res,AccT,BP,p,i,q,k,k1,k2;
  comment The local variables in the body of the procedure
          Pass3.
```

| The variable identifier | Values of the variable |
|---|---|
| Op | Operator. The number or the first analysed operator (see Table III). |
| NO | Next Operator. The number of the operator appearing after the Op operator. |
| PO | Preceding Operand. The operand (i. e., the element of the program different from an operator) appearing in front of the Op operator (see Table V). |
| CO | Current Operand. The operand appearing in front of the NO operator. |
| AD | Array Declaration. The parameter used when compiling subscript expressions and having negative values only at the beginning and at the end of the compiling an array declaration. |
| PH | Procedure Hierarchy. The level of the proce- |

dure whose body is being currently compiled.

AssT — Assignment Type. The parameter defining the type of the variable, together with an assignment symbol forms the left part list of an assignment statement. It takes on the following values:
1 for a Boolean variable,
2 for an integer variable,
3 for a real variable.

J — The number of the currently used anonymous variables. During compiling a procedure body it is equal to the relative address of the last occupied anonymous variable.

MaxJ — The maximum value of the variable J.

ModJ — Parameter, which has to be added to the value of J in order to obtain the relative address of the anonymous variable of number J.

CV,CVA — Controlled Variable, Controlled Variable Address. These parameters are used in for clause compilation.

CPS — Current Procedure Statement. The value of this variable is different from zero during the compilation of the actual parameter list of a procedure.

CPDI — Compiled Procedure Identifier. The operand corresponding to a procedure identifier when compiling the actual parameter list of the procedure.

NAP — Number of Actual Parameter. The address of the formal parameter description when compiling the

| | |
|---|---|
| | actual parameter of the procedure. |
| Res | Result. This parameter value is different from zero if AccC is defined. |
| AccT | Accumulator Type. This parametr value is different from zero if in the accumulator there is an arithmetic value, otherwise is equal to zero. |
| BP | Beginning of Program. The address of the first generated instruction. |
| p | The stack indicator. |
| i,q,k,k1,k2 | The working variables; |

```
procedure Read(PPO,POp);
 integer PPO,POp;
 comment If a successive syllable is an operand, the
 suitably transformed value of the syllable is assigned
 to the parameter PPO (see Table V), and the operator
 number defined by the next syllable is assigned to the
 parameter POp. Otherwise, zero and the operator with is
 number defined by the syllable are assigned to the para-
 meters PPO and POp, respectively. If the operator ends
 the program (operator number 128), a jump to the label
 EPass3 is executed;
 begin
  integer i,q;
  i:=RSyl;
  q:=i÷4×2;
  i:=i-q×2;
  if i≠0
   then
   begin
```

```
    i:=if i=1 then 16×Type(q)+Decl(q) else if i=3 then 7
       else if q<3 then 8 else 15;
    i:=i×2048+(if i=7 then q÷2 else q);
    q:=RSyl÷4
   end i≠0;
  if q=128
   then go to EPass3;
  PPO:=i;
  POp:=q
 end Read;
integer procedure Type1(FP);
 value FP;
 integer FP;
 comment The function is equal to 1 if the operand FP
 is the formal parameter called by name, otherwise the
 function value is equal to zero;
  Type1:=if FP>65536∨FP<32768 then 0 else 1;
integer procedure Decl1(FP);
 value FP;
 integer FP;
 comment If the operand FP is an identifier, the func-
 tion is equal to the value of Decl from the identifier
 description. Other cases are the following:
 If the operand is                        the function
                                          is equal to

 a Boolean value, or an anonymous var-
 iable which contains a Boolean value
 or a Boolean value address                   4
 a constant string                            7
 an anonymous variable containing an
```

```
integer value or an integer value
address                                        8
an anonymous variable containing a
real value or real value address              12
an arithmetic constant                        15;
 if FP>65536
 then
  begin
   FP:=FP÷8192;
  Decl1:=4×(if FP>4 then FP-4 else FP)
 end FP>65536
 else
 begin
  FP:=FP÷2048;
  Decl1:=if FP>16 then FP-16 else FP
 end FP≤65536;
integer procedure Addr1(FP);
 value FP;
 integer FP;
 comment The function is equal to the absolute or
 relative (the latter for anonymous variables and
 for the identifiers declared in a procedure body)
 address of the operand FP. If FP is an identifier
 whose address is not yet set, the function is equal
 to the address of the last instruction, the address
 part of which is to be also the address of FP;
 begin
  integer i,q;
  i:=Decl1(FP);
  q:=FP-FP÷2048×2048;
```

```
  Addr1:=if FP>65536 then ModJ+(if FP>98304 then 8388608
        else 0)+FP-FP÷8192×8192 else if i<3∨i=6∨i=10∨i=14
        then ForwJ(FP) else if i=7 then ASL+q else if i=15
        then ACL+q else if i=4∧q<3 then (if q=0 then FALSE
        else TRUE) else 2097152×IL(FP)+Addr(FP)
 end Addr1;
procedure LDA(FP);
 value FP;
 integer FP;
 comment The generation of the instruction of loading
 the operand FP value to the accumulator. If FP is
 a label, a jump instruction is generated. The proce-
 dure assigns also a value to the parameter AccT;
 if FP=0
  then
  begin
   AccT:=0;
   Compile(ZEAW,0)
  end FP=0
  else
  begin
   integer i,q;
   i:=Decl1(FP);
   i:=if i=0 then SKB else if i=7 then USAN else
      if i=8∨i=12 then UZAK else USAK;
   AccT:=if i=UZAK then 1 else 0;
   Compile(i,Addr1(FP))
  end FP≠0,LDA;
Boolean procedure Compare;
 comment The function has the value true if before gene-
```

rating the instruction which corresponds to the operator NO it is necessary to generate instruction corresponding to the operator Op (the operator Op precedence is greater than the precedence of the operator NO);

```
 Compare:=if Op>37 then Op≥NO else if Op≥36 then NO≤37
          else if Op≥33 then NO≤35 else if Op≥27 then
          NO≤32 else Op≥21∧Op≥NO∧(Op≠21∨NO≠21);
procedure IncJ(FP);
 value FP;
 integer FP;
 comment Increasing by FP the number J of the currently
 used anonymous variables and determining the maximum
 value of the hitherto existing values of the variable J;
 begin
  J:=J+FP;
  if MaxJ<J
   then MaxJ:=J
 end IncJ;
procedure DecJ(FP);
 value FP;
 integer FP;
 comment Decreasing the number of the currently used
 anonymous variables by one or two according to that
 how many locations are occupied by the operand FP;
 begin
  FP:=FP÷8192-8;
  J:=J-(if FP>4∨FP=1 then 1 else 2)
 end DecJ;
integer procedure ForwJ(FP);
 value FP;
```

```
integer FP;
comment The function is equal to the address of the
identifier indicated by FP (see procedure Addr1 above);
begin
 ForwJ:=Addr(FP);
 if AInt(FP)=0
  then UID(8,OPA+1,FP)
end ForwJ;
procedure DetA(FP);
value FP;
integer FP;
comment Assigning a value OPA+1 to the address of the
identifier indicated by FP;
begin
 integer i,q;
 i:=Addr(FP);
 UID(8,OPA+1,FP);
 UID(12,1,FP);
 comment A change of address parts of the instructions
 which refer to the identifier indicated by FP (A1);
 for FP:=i while i≠0 do
  begin
   q:=Store[FP];
   i:=q-q÷16384×16384;
   Store[FP]:=q-i+OPA+1
  end FP                                        (A1)
end DetA;
procedure GSI;
comment The procedure body is executed only when AccC
is defined. An instruction of storing the value in
```

```
 an anonymous variable is then generated and the value
 of the operand PO is changed;
 if Res≠0
  then
  begin
   IncJ(AccT+1);
   Compile(if AccT=0 then PSKA else PZKA,ModJ+J);
   Res:=0;
   PO:=8192×(9+AccT)+J
  end GSI;
procedure FRP1(FP);
 value FP;
 integer FP;
 comment The generation of instructions to call the
 formal parameter of the specification label or switch
 indicated by FP;
 begin
  Compile(SKP,PFM[IL(FP)]);
  Compile(MOK1,S1);
  Compile(MOK1,1);
  Compile(SKB,FPN(FP))
 end FRP1;
procedure FRP2(FP);
 value FP;
 integer FP;
 comment The generation of instructions to call the
 formal parameter of any specification but label and
 switch;
 begin
  integer i;
```

```
  i:=IL(FP);
  if CPS=0
   then Compile(USAK,RI)
   else
   begin
    Compile(USAK,R);
    Compile(SSAN,3)
   end CPS≠0;
  Compile(SKP,PF[i]);
  Compile(MOK1,S1);
  Compile(MOK1,1);
  Compile(SKB,FPN(FP));
  Compile(MOK1,R);
  Compile(UBK[i],2);
  if CPS≠0
   then
   for i:=1,2,3 do
    Compile(ODKJ,R)
 end FRP2;
procedure FRP(FP);
 integer FP;
 comment The generation of instructions to call the
 formal parameter indicated by FP and a change of the
 operand FP value;
 begin
  integer ap,i;
  ap:=FP;
  i:=Decl(ap);
  comment The formal parameter has the specification
  label (A2);
```

```
if i=0
 then FRP1(ap)                                                   (A2)
 else
 begin
  FRP2(ap);
  if AEAP(ap)=0
   then
   begin
    comment All the corresponding actual parameters are
    variables. The address of the corresponding actual
    parameter is stored in an anonymous variable (A3);
    IncJ(1);
    Compile(PSKA,ModJ+J);
    FP:=8192×(12+i÷4)+J
   end AEAP(ap)=0                                                (A3)
   else
   begin
    comment The accumulator is loaded with a value of
    the corresponding actual parameter (A4);
    Res:=1;
    Compile(MOA1,0);
    Compile(if i<8 then USAK else UZAK,0);
    AccT:=i÷8;
    FP:=0
   end AEAP(ap)≠0                                                (A4)
 end i≠0
end FRP;
procedure AssS(FP);
 value FP;
 integer FP;
```

```
  comment The generation of the instruction assigning
  AccC to the operand FP;
  begin
   if AssT=2
    then
    begin
     comment The generation of the instruction of a jump
     to the round off AccC subroutine before the first
     assignment of a value to an integer variable (A5);
     Compile(SKP,CCZP);
     AssT:=3
    end AssT=2;                                        (A5)
   Compile(if AssT=0 then PSKA else PZKA,Addr1(FP))
  end AssS;
 switch c1:=Re,DecOp,Op41,DecOp,DecOp,LB,Op45,Op46,
           Op47,Op48,Op49;
 switch c2:=if NO=1 then EB else RNS,if NO=2 then Op2
           else RNS,Op3,RRP,MEr,ESD,MEr,Op8,Op9,Op10,
           MEr,Op12,Op13,Op14,Op15,GAFR,MEr,EEAP,
           MEr,Op20;
 comment Assigning initial values to the parameters (A6);
 PH:=CPS:=AD:=NLN:=J:=MaxJ:=0;
 Compile(SKP,START);
 BP:=OPA;
 ModJ:=4194304;                                        (A6)
 comment The beginning of cpmpiling a new statement;
RNS:
 AssT:=0;
 comment The beginning of compiling a new expression;
RNS1:
```

```
  Res:=0;
  comment Reading the pair ⟨PO,Op⟩;
Re:
  Read(PO,Op);
  comment The analysis of the operand appearing in front
  of operator Op (A13);
Re1:
  if PO>65536
   then go to Open;
  comment The operand PO is not an anonymous variable (A13);
  q:=Decl1(PO);
  if Type1(PO)=0
   then
   begin
    comment The operand PO is not a formal parameter (A11);
    if q-q÷4×4=2
     then
     begin
      comment The operand PO is a procedure identifier
      (A10);
      i:=IL(PO);
      if Op=21
       then
       begin
        comment The pair ⟨PO,Op⟩ is a left part of an
        assignment statement (A7);
        Compile(USAN,2097152×i+MaxA(PO));
        Compile(PSKA,8388608+i);
        IncJ(1);
        Compile(PSKA,ModJ+J);
```

```
  PO:=8192×(12+q÷4)+J;
  go to SAS
 end OP=21;                                        (A7)
comment The generation of instructions of a proce-
dure call (A8);
if CPS=0
 then Compile(USAK,RI)
 else
 begin
  Compile(USAK,R);
  if Decl(NAP)≠0
   then Compile(SSAN,3)
 end CPS≠0;
CPS:=CPS+1;
Compile(SKP,if PH≤i then WEN else WE[PH,i]);
Compile(SKB,ForwJ(PO));
q:=NFP(PO);
if q≠0
 then
 begin
  comment The operand PO is a procedure with
  parameters (A8);
  p:=p+5;
  S[p-4]:=CPDI;
  S[p-3]:=NAP;
  S[p-2]:=OPA+1;
  NAP:=FPLP(PO)×2;
  CPDI:=PO;
  S[p-1]:=OPA+2;
  S[p]:=18;
```

```
          for i:=0 step 1 until q do
           Compile(SKB,0);
          go to ESD1
         end q≠0;                                                    (A8)
        comment The generation of instructions to be execu-
        ted after the exit from a procedure body (A10);
EPS:    i:=IL(PO);
        if PH>i
         then Compile(SKP,WY[PH,i]);
        CPS:=CPS-1;
        if CPS≠0
         then
         for i:=1,2,3 do
          Compile(ODKJ,R);
        i:=Decl1(PO);
        if i≠2
         then
         begin
          comment The procedure is a function. Loading the
          accumulator with a value of a function (A9);
          Compile(PSKA,C);
          Compile(SKZ,BFNS);
          Compile(if i<8 then USAK else UZAK,8388608+C);
          AccT:=i÷8
         end i≠2;                                                    (A9)
        Res:=1;
        if Op=4
         then go to RRP1;
        NO:=Op;
        PO:=0;
```

```
     go to Unstack
    end q-q÷4×4=2                                          (A10)
  end Type1(PO)=0                                          (A11)
  else
  begin
   comment The operand PO is a formal parameter (A13);
   if q=1
    then
    begin
     comment The formal parameter is a switch identi-
     fier (A12);
     if Op=44
      then go to LB;
     FRP1(PO);
     Res:=1;
     go to Stack2
    end q=1;                                               (A12)
   FRP(PO)
  end Type1(PO)≠0;                                         (A13)
 comment The analysis of the operator Op (A27);
 comment A choice of a subroutine from the part c1
 (see 2.1) (A14);
Open:
 if Op>38∧Op<50
  then go to c1[Op-38];                                    (A14)
Close:
 if Op=7∨Op=21
  then
SAS:
  begin
```

```
    comment Assigning a value to the parameter AssT (A15);
    i:=Decl1(PO)+4;
    if AssT≠0∧AssT≠i
     then go to AssEr;
    AssT:=i
   end Op=7∨Op=21;                                                  (A15)
  if Op<21
   then
Stack2:
   begin
    comment The operator Op belongs to the class C2 (A16);
    CO:=PO;
    NO:=Op
   end Op<21                                                        (A16)
   else
   begin
    comment Reading the pair ⟨CO,NO⟩ (A17);
    Read(CO,NO);                                                    (A17)
    if NO>38∧NO<50
     then
Stack1:
     begin
      comment The operator NO belongs to the class C1 (A18);
      GSI;
      p:=p+2;
      S[p-1]:=PO;
      S[p]:=Op;
Stack:PO:=CO;
      Op:=NO;
      go to Re1
```

```
 end NO>38∧NO<50;                                                  (A18)
 comment The analysis of the contex of operators Op
 and NO (A29);
Dec1:
 if ¬Compare
  then go to Stack1;
 comment The generation of instructions corresponding
 to the operator Op and the operands PO and CO (A29);
 if CO<65536
  then
  begin
   comment The operand CO is not a formal parameter
   (A19);
   if Type1(CO)=0
    then
    begin
     i:=Decl1(CO);
     go to if i-i÷4×4=2 then Stack1 else Dec4
    end Type1(CO)=0;                                               (A19)
   comment The operand CO is a formal parameter (A20);
   GSI;
   FRP(CO);
   if Res=0
    then go to Dec3
  end CO<65536                                                     (A20)
  else
Dec4:if Res≠0
   then go to Dec5;
 comment AccC is undefined (A22);
 Res:=1;
```

```
    if Op=21
     then
     begin
      comment The generation of instructions to compile
      an assignment statement if to the right of the
      assignment symbol there occurs a single quantity
      (A21);
      LDA(CO);
      AssS(PO)
     end Op=21                                          (A21)
     else
     begin
      comment The generation of the instruction to load
      the accumulator with the operand PO value and of
      instructions which realise the operator Op if a
      value of the operand preceding it is in the accu-
      mulator (A24);
      LDA(PO);                                          (A22)
      comment A change of the parameter AccT value, if
      the execution of the instruction which realises
      the operator Op results in a change of a real or
      integer accumulator value into a Boolean one (A23);
Dec5: if Op<33VOp=68
       then AccT:=0;                                    (A23)
      Compile(CODE1[Op],Addr1(CO))
     end Op≠21;                                         (A24)
    comment Decreasing the number of currently used ano-
    nymous variables (A25);
    if CO>65536
     then DecJ(CO);
```

```
    CO:=0;
    if PO>65536
     then DecJ(PO)                                          (A25)
   end Op≥21;                                               (A26)
  comment Picking up an ⟨operand, operator⟩ pair from
  the stack (A27);
Unstack:
  Op:=S[p];
  PO:=S[p-1];                                               (A27)
  comment The analysis of the context of operators Op
  and NO in the case when there have been already gene-
  rated the instructions which realise at last one ope-
  rator occurring in the source program between Op and
  NO, and AccC is defined (A29);
Dec2:
  if Compare
   then
   begin
    comment Removing from the stack the previously
    picked up ⟨operand, operator⟩ pair and the genera-
    tion of the instructions which realise the opera-
    tor Op when in the accumulator there is the value
    of the operand occurring after the operator (A29);
    p:=p-2;
Dec3:
    if Op=21
     then AssS(PO)
     else
     begin
      if Op<33∨Op=68
```

```
      then AccT:=0;
     Compile(CODE2[Op],Addr1(PO))
    end Op≠21;
   comment Decreasing the number of currently used ano-
   nymous variables (A28);
   if PO>65536
    then DecJ(PO);                                        (A28)
   go to Unstack
  end Compare;                                            (A29)
 if NO≥21
  then go to Stack;
 comment The operator NO belongs to the class C2 (A47);
 if NO≠7
  then
  begin
   comment The operator NO is not the for-assign
   operator (A47);
   if Res≠0VCO=0
    then go to Close1;
   comment Subroutine '}' (A30);
   if NO=17
    then Compile(USAN,Addr1(CO))                          (A30)
    else
    begin
     if Op=18
      then
EEAP: begin
       comment Subroutine 'endparameter' (A40);
       i:=Decl(NAP);
       if i=0
```

```
then
begin
 comment The corresponding formal parameter has
 the specification label (A31);
 if Res=0
  then Compile(SKB,ForwJ(CO))
end i=0                                                (A31)
else
if i=1
 then
 begin
  comment The corresponding formal parameter
  has the specification switch (A32);
  Compile(SKP,INDP);
  Compile(0,SLL(CO)×16384+ForwJ(CO))
 end i=1                                               (A32)
 else
 begin
  comment The corresponding formal parameter
  is an array of any type or a string (A33);
  if i-i÷2×2=0
   then LDA(CO)                                        (A33)
   else
   if Res=0
    then
    begin
     comment AccC is not defined (A36);
     i:=Decl1(CO);
     comment The actual parameter is a constant
     and in the description of the corresponding
```

```
      formal parameter the value of Left is equal
      to 1 (see Table IV) (A34);
      if (i=15∨i=8∨CO-i×2048<3)∧Left(CO)=1
       then go to AssEr;                              (A34)
      Compile(USAN,Addr1(CO));
      comment Decreasing the number of currently
      used anonymous variables (A35);
      if CO>65536
       then J:=J-1                                    (A35)
     end Res=0                                        (A36)
     else
     begin
      comment AccC is defined (A37);
      if Left(CO)=1
       then go to AssEr;
      IncJ(AccT+1);
      i:=ModJ+J;
      Compile(if AccT=0 then PSKA else PZKA,i);
      Compile(USAN,i);
      J:=J-AccT-1
     end Res≠0;                                       (A37)
   comment The generation of the instruction to
   exit from the actual parameter subroutine (A38);
   Compile(SKB,KPP)                                   (A38)
  end i>1;
NAP:=NAP-2;
if NO=20
 then go to ESD1;
comment The end of compiling a procedure call (A39);
p:=p-5;
```

```
        PO:=S[p+3];
        Store[PO]:=Store[PO]+OPA+1;
        PO:=CPDI;
        NAP:=S[p+2];
        CPDI:=S[p+1];
        go to EPS                                               (A39)
       end Op=18;                                               (A40)
      if Op=6
       then
       begin
        comment Subroutine 'endswitch' (A45);
        comment The change of the address part in the
        instruction of a jump to compute a designational
        expression value in the switch list, if the expre-
        ssion is a label (A41);
        q:=OPA;
        OPA:=PO-2;
        Store[PO-1]:=Store[PO-1]-16384×16384+ForwJ(CO);
        OPA:=q;                                                 (A41)
ESD:    if NO=20
         then
ESD1:    begin
          comment A change of parameters in the stack
          after compiling the successive element of the
          switch list or the successive actual parame-
          ter (A42);
          S[p-1]:=PO+1;
          S[p]:=Op
         end NO=20                                              (A42)
         else
```

```
ESD2:    begin
          comment The end of compiling the switch decla-
          ration (A43);
          p:=p-3;
          PO:=S[p+1]
         end NO≠20;                                        (A43)
        comment Completing the address part in the instruc-
        tion of jump to compute a designational expression
        in the switch list or to an actual parameter sub-
        routine (A44);
        Store[PO]:=Store[PO]+OPA+1;                        (A44)
        go to RNS1
       end Op=6;                                           (A45)
      LDA(CO)
     end NO≠17;
    Res:=1;
    if CO>65536
     then DecJ(CO);
    comment A choice of a subroutine of the part c2 (A46);
Close1:
    go to c2[Op]                                           (A46)
   end NO≠7;                                               (A47)
  comment Subroutine 'for-assign' (A50);
  p:=p-2;
  if CO<65536
   then
   begin
    comment The controlled variable is a simple one,
    and the value of CVA is equal to its address, and
    the CV value is equal to zero (A48);
```

```
    IncJ(1);
    CVA:=Addr(CO);
    OPA:=OPA-1;
    CV:=0
   end CO<65536                                          (A48)
   else
   begin
    comment The controlled variable is a subscripted one
    or is a formal parameter. The value of CVA is equal
    to the address of the subroutine computing address
    of the controlled variable and CV value is different
    from zero (A49);
    Store[CVA]:=Store[CVA]+OPA+2;
    CVA:=CVA+1;
    CV:=1;
    Compile(SKB,8388607+ModJ+J)
   end CO≥65536;                                         (A49)
  go to EFLE;                                            (A50)
  comment Subroutine 'beginb' (A51);
Op41:
  NLN:=NLN+1;
  Compile(USAK,RI);
  RI:=RI+1;
  Compile(PSKA,RI);
  go to DecOp;                                           (A51)
  comment Subroutine 'switch' (A53);
Op45:
  Read(PO,Op);
  DetA(PO);
  p:=p+3;
```

```
  S[p-2]:=OPA+1;
  PO:=OPA+2;
  Op:=6;
  comment The generation of jump instructions to compute
  the switch designators in the switch list (A52);
  for q:=SLL(PO) step -1 until 0 do
   Compile(SKB,0);                                    (A52)
  go to ESD1;                                         (A53)
  comment Subroutine 'for' (A55);
Op46:
  Compile(SKB,0);
  PO:=CVA:=OPA;
  IncJ(2);
  comment Subroutines 'begin', 'if' and '(' (A55);
  comment The change of an operator which belongs to the
  class c1 (A54);
DecOp:
  Op:=Op-39;                                          (A54)
  comment Stacking a pair ⟨PO,Op⟩, transfer to assigning
  new values to ⟨PO,Op⟩ (A55);
Open1:
  p:=p+2;
  S[p-1]:=PO;
  S[p]:=Op;
  go to Re;                                           (A55)
  comment Subroutine ':' (A56);
Op47:
  DetA(PO);
  go to Re;                                           (A56)
  comment Subroutine 'array' (A58);
```

```
Op48:
  Read(PO,Op);
  Compile(USAN,if Decl(PO)<8 then 1 else 2);
  IncJ(1);
  Compile(PSKA,ModJ+J);
  AD:=AD-1;
  comment Stacking array identifiers separated by commas
  (arrays share the bound pair list) (A57);
Arr1:
  if Op=20
   then
   begin
    p:=p+2;
    S[p-1]:=PO;
    S[p]:=Op;
Arr:Read(PO,Op);
    go to Arr1
   end Op=20;                                                  (A57)
  comment Subroutine '[' (A58);
LB:
  AD:=AD+1;
  p:=p+4;
  S[p-3]:=PO;
  S[p-2]:=5;
  S[p-1]:=0;
  S[p]:=20;
  go to Re;                                                    (A58)
  comment Subroutine 'procedure' (A65);
Op49:
  NLN:=NLN+1;
```

```
Compile(SKB,0);
p:=p+4;
S[p-3]:=4096×MaxJ+J;
S[p-2]:=RI;
S[p-1]:=OPA;
S[p]:=14;
Read(PO,Op);
DetA(PO);
i:=MaxA(PO);
J:=MaxJ:=MaxR(PO)+i;
PH:=PH+1;
ModJ:=2097152×PH;
RI:=ModJ+i;
UID(10,i-1,PO);
Compile(USAN,0);
Compile(SKP,PP[PH]);
Compile(PSKA,RI);
CO:=FPLP(PO)×2;
i:=NFP(PO);
comment The generation of instructions to call the for-
mal parameters called by value or those which are arrays
or strings (A63);
for q:=1 step 1 until i do
 begin
  k:=Decl(CO);
  k1:=VP(CO);
  k2:=k-k÷4×4;
  if k1≠0∨k≠1∧k2≠0
   then
   begin
```

```
NO:=ModJ+Addr(CO);
FRP2(CO);
if k2=0
 then
 begin
  comment The parameter has one of the following
  specifications: real, integer or Boolean (A59);
  Compile(MOA1,0);
  Compile(if k=4 then USAK else UZAK,0);
  if k=8
   then Compile(SKP,CCZP);
  Compile(if k=4 then PSKA else PZKA,NO)
 end k2=0                                          (A59)
 else
 begin
  comment The parameter is an array or a string
  (A61);
  if k1≠0
   then
   begin
    comment The parameter is an array in the value
    part (A60);
    Compile(USWK,RI);
    Compile(SKP,if k=9 then PRPI else PRPT):
    Compile(PAKW,RI)
   end k1≠0;                                       (A60)
  Compile(PSKA,NO)
 end k2≠0;                                         (A61)
comment A change of the value of Type (see Table IV)
in the formal parameter description (A62);
```

```
      UID(1,0,CO)                                        (A62)
     end k1≠0∨k≠1∧k2≠0;
    CO:=CO-2
   end q;                                                (A63)
  comment Clearing the variable which contains the function
  value address after executing the function body (A64);
  if Decl(PO)≠2
   then Compile(ZERK,8388608+PH);                        (A64)
  go to Re;                                              (A65)
  comment Subroutine 'endb' (A66);
Op2:
  RI:=RI-1;
  comment Subroutine 'end' (A66);
EB:
  p:=p-2;
  go to RNS;                                             (A66)
  comment Subroutine 'then' (A67);
Op3:
  Compile(SKD,0);
  S[p-1]:=OPA;
  S[p]:=15;
  go to RNS1;                                            (A67)
  comment Subroutine 'step' (A76);
Op8:
  if NO=11
   then
   begin
    comment The end of a for clause (A69);
    NLN:=NLN+1;
    OPA:=OPA-2×CV;
```

```
   Compile(SKB,0);
   comment Completing jump instructions to the subroutine
   executing a controll statement (A68);
   for Op:=S[p-2] while Op=7 do
    begin
     p:=p-2;
     PO:=S[p+1];
     Store[PO]:=Store[PO]+OPA+1
    end Op;                                                  (A68)
   S[p-1]:=OPA;
   S[p]:=12;
   J:=J-2;
   go to RNS
  end NO=11;                                                 (A69)
comment Assigning the accumulator value to controlled
variable (A70);
if AssT=2
 then Compile(SKP,CCZP);
Compile(PZKA,if CV=0 then CVA else 8388608+ModJ+J);         (A70)
comment Setting the parameter values when NO is the
operator while (A71);
if NO=13
 then S[p]:=13                                               (A71)
 else
 if NO=8
  then
  begin
   comment The operator NO is step (A72);
   Compile(ZERK,ModJ+J-2);
   S[p-1]:=OPA+1;
```

```
     S[p]:=9;
     if CV≠0
      then go to EFLE2;
     Compile(USAN,CVA);
     Compile(PSKA,ModJ+J)
    end NO=8                                              (A72)
    else
    begin
     comment The generation of instructions to enter the
     subroutine executing the controlled statement if the
     for list element is an arithmetic expression (A73);
     Compile(PLR,ModJ+J-2);
     Compile(SKB,0);
     S[p-1]:=OPA;                                         (A73)
     comment Setting values of parameters in the stack
     after compiling an element of the for list (A74);
EFLE1:
     S[p]:=7;
EFLE:p:=p+2;
     S[p-1]:=OPA+1;
     S[p]:=8;                                             (A74)
     if CV≠0
      then
EFLE2:begin
       comment The generation of instructions to call the
       subroutine which computes the address of the con-
       trolled variable (A75);
       Compile(PLR,ModJ+J-1);
       Compile(SKB,CVA)
      end CV≠0                                            (A75)
```

```
    end NO≠8,NO≠13;
  go to RNS1;                                                    (A76)
  comment Subroutine 'until' (A77);
Op9:
  IncJ(2);
  Compile(PZKA,ModJ+J);
  S[p]:=10;
  go to RNS1;                                                    (A77)
  comment Subroutine 'end-for-list-element' (A79);
Op10:
  Compile(if AssT=2 then ROP2 else ROP0,ModJ+J);
  J:=J-2;
  comment The generation of instruction to enter the
  subroutine which executes a controlled statement and
  the instruction to return for assigning a new value
  to the controlled variable (A78);
EFLE3:
  Compile(PLR,ModJ+J-2);
  Compile(SKB,0);
  Compile(SKB,P0);                                               (A78)
  S[p-1]:=OPA-1;
  go to EFLE1;                                                   (A79)
  comment Subroutine 'endfor' (A80);
Op12:
  Compile(SKB,8388608+ModJ+J);
  J:=J-1;
  go to GAFR;                                                    (A80)
  comment Subroutine 'while' (A81);
Op13:
  Compile(SKD,OPA+5);
```

```
  go to EFLE3;                                                  (A81)
  comment Subroutine 'endprocedure' (A82);
Op14:
  Compile(USAK,8388608+PH);
  Compile(SKB,KP[PH]);
  Store[PO]:=Store[PO]+OPA+1;
  PO:=PO+1;
  Store[PO]:=Store[PO]+MaxJ+1;
  PH:=PH-1;
  ModJ:=2097152×(if PH=0 then 2 else PH);
  p:=p-4;
  RI:=S[p+2];
  J:=S[p+1];
  MaxJ:=J÷4096;
  J:=J-MaxJ×4096;
  go to Unstack;                                                (A82)
  comment Subroutine 'else' (A84);
Op15:
  if NO=15
   then
   begin
    comment Setting parameters when the operator NO is
    the operator else (A83);
    i:=Store[OPA]÷16384;
    if i-i÷128×128=PZKA
     then AssT:=0;
    Compile(SKB,0);
    S[p-1]:=OPA;
    S[p]:=16;
    Store[PO]:=Store[PO]+OPA+1;
```

```
    go to RNS1
   end NO=15;                                                  (A83)
  comment Completing the address part of the instruction
  to skip the execution of a statement or computing the
  value of an expression appearing after an if clause,
  or after else, or to skip the subroutine executing a
  controlled statement (A84);
GAFR:
  p:=p-2;
  Store[PO]:=Store[PO]+OPA+1;
  go to Unstack;                                               (A84)
  comment Subroutine ',' (A93);
Op20:
  IncJ(2);
  Compile(PZKA,ModJ+J);
  PO:=PO+1;
  if NO=5
   then
   begin
    comment Setting parameters before compiling the next
    subscript expression (A85);
    S[p-1]:=S[p-1]+1;
    go to RNS1
   end NO=5;                                                   (A85)
  comment Setting parameters when the operator NO is a
  closing square bracket (A86);
  p:=p-2;
  J:=J-2×PO;
  NO:=S[p];
  CO:=S[p-1];
```

```
AD:=AD-1;                                                    (A86)
if AD<0
 then
 begin
  comment The end of the bound pair list in an array
  declaration (A89);
  Compile(USAN,Addr(CO));
  Compile(PSKA,ADS);
  Compile(USWK,RI);
  Compile(USAN,ModJ+J);
  Compile(PSKA,4);
  Compile(USAN,PO);
  Compile(SKP,RSA);
  comment The generation of the instruction of a jump
  to the subroutine of reservation the storage locations
  for the arrays, the bound pair list of which has just
  been compiled (A87);
  for p:=p,p-2 while S[p]=20 do
   Compile(SKP,AST);                                         (A87)
  Compile(PSKA,RI);
  Res:=0;
  Read(PO,Op);
  if Op=20
   then go to Arr;
  comment The end of an array declaration (A88);
  AD:=AD+1;
  J:=J-1;
  go to Stack2                                               (A88)
 end AD<0;                                                   (A89)
i:=Decl(CO);
```

```
 if i=1
  then
  begin
   comment The compiled subscript expression is the
   subscript in a switch designator (A92);
   OPA:=OPA-1;
   if Type(CO)=0
    then
    begin
     Compile(SKP,INDP);
     Compile(0,16384×SLL(CO)+ForwJ(CO))
    end Type(CO)=0
    else FRP1(CO);
   comment Subroutine ')' (A91);
   comment The removing an <operand, operator> pair
   from the stack (A90);
RRP:p:=p-2;                                                    (A90)
   comment The assignment of a new value  to operators
   and operands after the instructions which realise
   the operators Op and NO had been generated (A91);
RRP1:
   Op:=S[p];
   PO:=S[p-1];
   if Op<21
    then go to Re;
   Read(CO,NO);
   go to Dec2                                                  (A91)
  end i=1;                                                     (A92)
 comment The end of the subscript list in a subscripted
 variable (A93);
```

```
  InoJ(1);
  Compile(USWK,Addr(CO));
  Compile(USAN,PO);
  Compile(ROP7,ModJ+J);
  p:=p-2;
  Res:=0;
  Read(CO,NO);
  CO:=8192×(12+i+4)+J;
  Op:=S[p];
  PO:=S[p-1];
  if Op<21
   then go to Stack;
  p:=p-2;
  go to Dec1;                                               (A93)
  comment The operations which end compiling the program
  (A97);
EPass3:
  Compile(SKB,FIN);
  PO:=SKP+PP[1];
  CO:=SKB+KP[1];
  comment Completing address parts of the instructions
  referring to the anonymous variables and occurring
  outside procedure bodies (A96);
  for BP:= BP+1 step 1 until OPA do
   begin
    i:=Store[BP];
    if i=PO
     then
     begin
      comment Skipping the object program instructions
```

```
 corresponding to a procedure declaration (A94);
 for i:=Store[BP] while i≠CO do
  BP:=BP+1
end i=PO                                            (A94)
else
begin
 q:=i÷4194304;
 if q=1∨q=3
  then Store[BP]:=i-4194304+OPA                     (A95)
end i≠PO
end BP;                                             (A96)
comment The assignment of a value to the reservation
indicator corresponding to the main program (A97);
Store[RI]:=OPA+MaxJ                                 (A97)
end Pass3;
```

## References

[1] J. P. Anderson, *A note on some compiling algorithms*, CACM 7 (1964), p. 149-150.

[2] H. Bottenbruch and A. A. Grau, *On translation of Boolean expressions*, CACM 5 (1962), p. 384-386.

[3] E. W. Dijkstra, *Making a translator for ALGOL 60*, Annual Review in Automatic Programming, Pergamon Press, London 1963, p. 347-356.

[4] *Dokumantacja techniczno ruchowa maszyny ODRA 1204*, Opis funkcjonalny, WZE Elwro, 1968.

[5] D. Gries, *The object program produced by the ALCOR ILLINOIS 7090 compiler*, Rep. no. 6412, Rechenzentrum der Techn. Hochsch., München 1964.

[6] C. A. R. Hoare, *The ELLIOTT ALGOL programming system*, Introduction to system programming, Academic Press, London 1964, p. 156-165.

[7] P. Z. Ingerman, *A syntax-oriented translator*, Academic Press, London 1966.

[8] — *Thunks*, CACM 4 (1961), p. 55-58.

[9] J. Jensen, *Generation of machine code in ALGOL compilers*, BIT 5 (1965), p. 235-245.

[10] P. Naur, *The design of the GIER ALGOL compiler*, BIT 3 (1963), p. 123-166.

[11] S. Paszkowski, *Język ALGOL 60*, PWN, Warszawa 1968.

[12] B. Randell and L. J. Russel, *ALGOL 60 implementation*, Academic Press, London 1964.

[13] J. Szczepkowicz, *On table-driven syntax-checking within an ALGOL compiler* Zastosow. Matem., 11 (1969), p. 3-89.

[14] J. M. Watt, *The realization of ALGOL procedures and designational expressions*, Computer Journal 6 (1963), p. 332-337.

MATHEMATICAL INSTITUTE
UNIVERSITY OF WROCŁAW

*Received on 20. 3. 1970;*
*revised version on 23. 9. 1971*

---

KRYSTYNA JERZYKIEWICZ (Wrocław)

## ANALIZA SEMANTYCZNA TEKSTU ALGOLOWSKIEGO

### STRESZCZENIE

Praca zawiera opis metody analizy semantycznej poprawnego pod względem syntaktycznym tekstu algolowskiego i algorytm realizujący tę metodę przy założeniu, że w czasie wykonywania algorytm są dostępne w pamięci maszyny wszystkie informacje o nazwach i stałych używanych w analizowanym programie. Podany w pracy algorytm stanowi trzeci przebieg translatora ODRA-ALGOLu; język ODRA-ALGOL jest konkretną realizacją ALGOLu 60 dla maszyny cyfrowej ODRA 1204. Gramatyka języka ODRA-ALGOL i dwa pierwsze przebiegi translatora tego języka są opisane w pracy Szczepkowicza [13]. W algorytmie realizującym trzeci przebieg translatora zakłada się, że tekstem początkowym dla algorytmu jest tekst końcowy drugiego przebiegu translatora, a w wyniku działania algorytmu otrzymuje się w pamięci maszyny równoważny program w języku węwnetrznym maszyny ODRA 1204.

Metoda działania algorytmu jest pewną modyfikacją znanych metod rozwiązania zadania tłumaczenia programu napisanego w ALGOLu 60 na język maszyny jednoadresowej z jednym akumulatorem. Generowanie rozkazów przekładu wykonuje się w czasie jednego przeglądania tekstu programu z lewej do prawej. Algorytm jest sterowany procedurą porównującą pierwszeństwa operatorów, które są równe wewnętrznej reprezentacji tych operatorów. Algorytm nie zawiera podprogramów rekursywnych i używa jednego stosu roboczego.

Zależność opisywanego algorytmu od maszyny ODRA 1204 jest związana tylko z realizacją maszynową poszczególnych elementów składniowych języka ODRA-ALGOL. Realizacja ta została opracowana wspólnie z autorem dwóch pierwszych przebiegów translatora i będzie przedmiotem oddzielnej publikacji. W pracy wyjaśnia się jedynie znaczenie kodów operacyjnych rozkazów generowanych przez translator. Nie wyjaśnia się natomiast treści podprogramów, które są częścią stałą programów przetłumaczonych. Aby otrzymać inną realizację elementów składniowych, wystarczy nadać odpowiednie wartości zmiennym oznaczającym kody operacyjne i (jeśli wymaga tego realizacja) usunąć lub dołączyć generowanie niektórych rozkazów.