META II

A SYNTAX-ORIENTED COMPILER WRITING LANGUAGE

D. V. Schorre
UCLA Computing Facility

META II is a compiler writing language which consists of syntax equations resembling Backus normal form and into which instructions to output assembly language commands are inserted. Compilers have been written in this language for VALGOL I and VALGOL II. The former is a simple algebraic language designed for the purpose of illustrating META II. The latter contains a fairly large subset of ALGOL 60.

The method of writing compilers which is given in detail in the paper may be explained briefly as follows. Each syntax equation is translated into a recursive subroutine which tests the input string for a particular phrase structure, and deletes it if found. Backup is avoided by the extensive use of factoring in the syntax equations. For each source language, an interpreter is written and programs are compiled into that interpretive language.

META II is not intended as a standard language which everyone will use to write compilers. Rather, it is an example of a simple working language which can give one a good start in designing a compiler-writing compiler suited to his own needs. Indeed, the META II compiler is written in its own language, thus lending itself to modification.

History

The basic ideas behind META II were described in a series of three papers by Schmidt,[1] Metcalf,[2] and Schorre.[3] These papers were presented at the 1963 National A.C.M. Convention in Denver, and represented the activity of the Working Group on Syntax-Directed Compilers of the Los Angeles SIGPLAN. The methods used by that group are similar to those of Glennie and Conway, but differ in one important respect. Both of these researchers expressed syntax in the form of diagrams, which they subsequently coded for use on a computer. In the case of META II, the syntax is input to the computer in a notation resembling Backus normal form. The method of syntax analysis discussed in this paper is entirely different from the one used by Irons[6] and Bastian.[7] All of these methods can be traced back to the mathematical study of natural languages, as described by Chomsky.[8]

Syntax Notation

The notation used here is similar to the meta language of the ALGOL 60 report. Probably the main difference is that this notation can be keypunched. Symbols in the target language are represented as strings of characters, surrounded by quotes. Metalinguistic variables have the same form as identifiers in ALGOL, viz., a letter followed by a sequence of letters or digits. Items are written consecutively to indicate concatenation and separated by a slash to indicate alternation. Each equation ends with a semicolon which, due to keypunch limitations, is represented by a period followed by a comma. An example of a syntax equation is:

```
LOGICALVALUE = '.TRUE' / '.FALSE' .,
```

In the versions of ALGOL described in this paper the symbols which are usually printed in boldface type will begin with periods, for example:

```
.PROCEDURE .TRUE .IF
```

To indicate that a syntactic element is optional, it may be put in alternation with the word .EMPTY. For example:

```
SUBSECONDARY = '*' PRIMARY / .EMPTY .,
SECONDARY = PRIMARY SUBSECONDARY .,
```

By factoring, these two equations can be written as a single equation.

```
SECONDARY = PRIMARY('*' PRIMARY / .EMPTY) .,
```

Built into the META II language is the ability to recognize three basic symbols which are:

1. Identifiers -- represented by .ID,
2. Strings -- represented by .STRING,
3. Numbers -- represented by .NUMBER.

The definition of identifier is the same in META II as in ALGOL, viz., a letter followed by a sequence of letters or digits. The definition of a string is changed because of the limited character set available on the usual keypunch. In ALGOL, strings are surrounded by opening and closing quotation marks, making it possible to have quotes within a string. The single quotation mark on the keypunch is unique, imposing the restriction that a string in quotes can contain no other quotation marks.

The definition of number has been radically changed. The reason for this is to cut down on the space required by the machine subroutine which recognizes numbers. A number is considered to be a string of digits which may include imbedded periods, but may not begin or end with a period; moreover, periods may not be adjacent. The use of the subscript 10 has been eliminated.

Now we have enough of the syntax defining features of the META II language so that we can consider a simple example in some detail.

The example given here is a set of four syntax equations for defining a very limited class of algebraic expressions. The two operators, addition and multiplication, will be represented by + and * respectively. Multiplication takes precedence over addition; otherwise precedence is indicated by parentheses. Some examples are:

```
A
A + B
A + B * C
(A + B) * C
```

The syntax equations which define this class of expressions are as follows:

```
EX3 = .ID / '(' EX1 ')' .,
EX2 = EX3 ('*' EX2 / .EMPTY) .,
EX1 = EX2 ('+' EX1 / .EMPTY) .,
```

EX is an abbreviation for expression. The last equation, which defines an expression of order 1, is considered the main equation. The equations are read in this manner. An expression of order 3 is defined as an identifier or an open parenthesis followed by an expression of order 1 followed by a closed parenthesis. An expression of order 2 is defined as an expression of order 3, which may be followed by a star which is followed by an expression of order 2. An expression of order 1 is defined as an expression of order 2, which may be followed by a plus which is followed by an expression of order 1.

Although sequences can be defined recursively, it is more convenient and efficient to have a special operator for this purpose. For example, we can define a sequence of the letter A as follows:

```
SEQA = $ 'A' .,
```

The equations given previously are rewritten using the sequence operator as follows:

```
EX3 = .ID / '(' EX1 ')' .,
EX2 = EX3 $ ('*' EX3) .,
EX1 = EX2 $ ('+' EX2) .,
```

Output

Up to this point we have considered the notation in META II which describes object language syntax. To produce a compiler, output commands are inserted into the syntax equations. Output from a compiler written in META II is always in an assembly language, but not in the assembly language for the 1401. It is for an interpreter, such as the interpreter I call the META II machine, which is used for all compilers, or the interpreters I call the VALGOL I and VALGOL II machines, which obviously are used with their respective source languages. Each machine requires its own assembler, but the main difference between the assemblers is the operation code table. Constant codes and declarations may also be different. These assemblers all have the same format, which is shown below.

LABEL	CODE	ADDRESS
1- -6	8- -10	12- -70

An assembly language record contains either a label or an op code of up to 3 characters, but never both. A label begins in column 1 and may extend as far as column 70. If a record contains an op code, then column 1 must be blank. Thus labels may be any length and are not attached to instructions, but occur between instructions.

To produce output beginning in the op code field, we write .OUT and then surround the information to be reproduced with parentheses. A string is used for literal output and an asterisk to output the special symbol just found in the input. This is illustrated as follows:

```
EX3 = .ID .OUT('LD ' *) / '(' EX1 ')' .,
EX2 = EX3 $ ('*' EX3 .OUT('MLT')) .,
EX1 = EX2 $ ('+' EX2 .OUT('ADD')) .,
```

To cause output in the label field we write .LABEL followed by the item to be output. For example, if we want to test for an identifier and output it in the label field we write:

```
.ID .LABEL *
```

The META II compiler can generate labels of the form A01, A02, A03, ... A99, B01, ... To cause such a label to be generated, one uses *1 or *2. The first time *1 is referred to in any syntax equation, a label will be generated and assigned to it. This same label is output whenever *1 is referred to within that execution of the equation. The symbol *2 works in the same way. Thus a maximum of two different labels may be generated for each execution of any equation. Repeated executions, whether recursive or externally initiated, result in a continued sequence of generated labels. Thus all syntax equations contribute to the one sequence. A typical example in which labels are generated for branch commands is now given.

```
IFSTATEMENT = '.IF' EXP '.THEN' .OUT('BFP' *1)
     STATEMENT '.ELSE' .OUT('B ' *2) .LABEL *1
     STATEMENT .LABEL *2 .,
```

The op codes BFP and B are orders of the VALGOL I machine, and stand for "branch false and pop" and "branch" respectively. The equation also contains references to two other equations which are not explicitly given, viz., EXP and STATEMENT.

VALGOL I - A Simple Compiler Written in META II

Now we are ready for an example of a compiler written in META II. VALGOL I is an extremely simple language, based on ALGOL 60, which has been designed to illustrate the META II compiler.

The basic information about VALGOL I is given in figure 1 (the VALGOL I compiler written in META II) and figure 2 (order list of the VALGOL I machine). A sample program is given in figure 3. After each line of the program, the VALGOL I commands which the compiler produces from that line are shown, as well as the absolute interpretive language produced by the assembler. Figure 4 is output from the sample program. Let us study the compiler written in META II (figure 1) in more detail.

The identifier PROGRAM on the first line indicates that this is the main equation, and that control goes there first. The equation for PRIMARY is similar to that of EX3 in our previous example, but here numbers are recognized and reproduced with a "load literal" command. TERM is what was previously EX2; and EXP1 what was previously EX1 except for recognizing minus for subtraction. The equation EXP defines the relational operator "equal", which produces a value of 0

or 1 by making a comparison. Notice that this is handled just like the arithmetic operators but with a lower precedence. The conditional branch commands, "branch true and pop" and "branch false and pop", which are produced by the equations defining UNTILST and CONDITIONALST respectively, will test the top item in the stack and branch accordingly.

The "assignment statement" defined by the equation for ASSIGNST is reversed from the convention in ALGOL 60, i.e., the location into which the computed value is to be stored is on the right. Notice also that the equal sign is used for the assignment statement and that period equal (.=) is used for the relation discussed above. This is because assignment statements are more numerous in typical programs than equal compares, and so the simpler representation is chosen for the more frequently occurring.

The omission of statement labels from the VALGOL I and VALGOL II seems strange to most programmers. This was not done because of any difficulty in their implementation, but because of a dislike for statement labels on the part of the author. I have programmed for several years without using a single label, so I know that they are superfluous from a practical, as well as from a theoretical, standpoint. Nevertheless, it would be too much of a digression to try to justify this point here. The "until statement" has been added to facilitate writing loops without labels.

The "conditional" statement is similar to the one in ALGOL 60, but here the "else" clause is required.

The equation for "input/output", IOST, involves two commands, "edit" and "print". The words EDIT and PRINT do not begin with periods so that they will look like subroutines written in code. "EDIT" copies the given string into the print area, with the first character in the print position which is computed from the given expression. "PRINT" will print the current contents of the print area and then clear it to blanks. Giving a print command without previous edit commands results in writing a blank line.

IDSEQ1 and IDSEQ are given to simplify the syntax equation for DEC (declaration). Notice in the definition of DEC that a branch is given around the data.

From the definition of BLOCK it can be seen that what is considered a compound statement in ALGOL 60 is, in VALGOL I, a special case of a block which has no declaration.

In the definition of statement, the test for an IOST precedes that for an ASSIGNST. This is necessary, because if this were not done the words PRINT and EDIT would be mistaken as identifiers and the compiler would try to translate "input/output" statements as if they were "assignment" statements.

Notice that a PROGRAM is a block and that a standard set of commands is output after each program. The "halt" command causes the machine to stop on reaching the end of the outermost block, which is the program. The operation code SP is generated after the "halt" command. This is a completely 1401-oriented code, which serves to set a word mark at the end of the program. It would not be used if VALGOL I were implemented on a fixed word-length machine.

How the META II Compiler Was Written

Now we come to the most interesting part of this project, and consider how the META II compiler was written in its own language. The interpreter called the META II machine is not a much longer 1401 program than the VALGOL I machine. The syntax equations for META II (figure 5) are fewer in number than those for the VALGOL I machine (figure 1).

The META II compiler, which is an interpretive program for the META II machine, takes the syntax equations given in figure 5 and produces an assembly language version of this same interpretive program. Of course, to get this started, I had to write the first compiler-writing compiler by hand. After the program was running, it could produce the same program as written by hand. Someone always asks if the compiler really produced exactly the program I had written by hand and I have to say that it was "almost" the same program. I followed the syntax equations and tried to write just what the compiler was going to produce. Unfortunately I forgot one of the redundant instructions, so the results were not quite the same. Of course, when the first machine-produced compiler compiled itself the second time, it reproduced itself exactly.

The compiler originally written by hand was for a language called META I. This was used to implement the improved compiler for META II. Sometimes, when I wanted to change the metalanguage, I could not describe the new metalanguage directly in the current metalanguage. Then an intermediate language was created -- one which could be described in the current language and in which the new language could be described. I thought that it might sometimes be necessary to modify the assembly language output, but it seems that it is always possible to avoid this with the intermediate language.

The order list of the META II machine is given in figure 6.

All subroutines in META II programs are recursive. When the program enters a subroutine a stack is pushed down by three cells. One cell is for the exit address and the other two are for labels which may be generated during the execution of the subroutine. There is a switch which may be set or reset by the instructions which refer to the input string, and this is the switch referred to by the conditional branch commands.

The first thing in any META II machine program is the address of the first instruction. During the initialization for the interpreter, this address is placed into the instruction counter.

VALGOL II Written in META II

VALGOL II is an expansion of VALGOL I, and serves as an illustration of a fairly elaborate programming language implemented in the META II system. There are several features in the VALGOL II machine which were not present in the

VALGOL I machine, and which require some explanation. In the VALGOL II machine, addresses as well as numbers are put in the stack. They are marked appropriately so that they can be distinguished at execution time.

The main reason that addresses are allowed in the stack is that, in the case of a subscripted variable, an address is the result of a computation. In an assignment statement each left member is compiled into a sequence of code which leaves an address on top of the stack. This is done for simple variables as well as subscripted variables, because the philosophy of this compiler writing system has been to compile everything in the most general way. A variable, simple or subscripted, is always compiled into a sequence of instructions which leaves an address on top of the stack. The address is not replaced by its contents until the actual value of the variable is needed, as in an arithmetic expression.

A formal parameter of a procedure is stored either as an address or as a value which is computed when the procedure is called. It is up to the load command to go through any number of indirect address in order to place the address of a number onto the stack. An argument of a procedure is always an algebraic expression. In case this expression is a variable, the value of the formal parameter will be an address computed upon entering the procedure; otherwise, the value of the formal parameter will be a number computed upon entering the procedure.

The operation of the load command is now described. It causes the given address to be put on top of the stack. If the content of this top item happens to be another address, then it is replaced by that other address. This continues until the top item on the stack is the address of something which is not an address. This allows for formal parameters to refer to other formal parameters to any depth.

No distinction is made between integer and real numbers. An integer is just a real number whose digits right of the decimal point are zero. Variables initially have a value called "undefined", and any attempt to use this value will be indicated as an error.

An assignment statement consists of any number of left parts followed by a right part. For each left part there is compiled a sequence of commands which puts an address on top of the stack. The right part is compiled into a sequence of instructions which leaves on top of the stack either a number or the address of a number. Following the instruction for the right part there is a sequence of store commands, one for each left part. The first command of this sequence is "save and store", and the rest are "plain" store commands. The "save and store" puts the number which is on top of the stack (or which is referred to by the address on top of the stack) into a register called SAVE. It then stores the contents of SAVE in the address which is held in the next to top position of the stack. Finally it pops the top two items, which it has used, out of the stack. The number, however, remains in SAVE for use by the following store commands. Most assignment statements have only one left part, so "plain" store commands are seldom produced, with the result that the number put in SAVE is seldom used again.

The method for calling a procedure can be explained by reference to illustrations 1 and 2. The arguments which are in the stack are moved to their place at the top of the procedure. If the

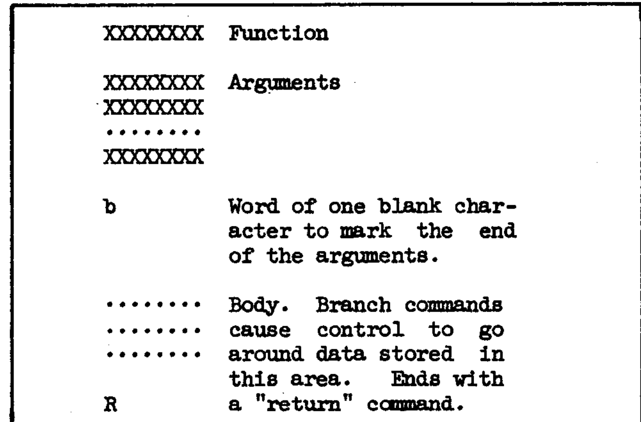

Illustration 1

Storage Map for VALGOL II Procedures

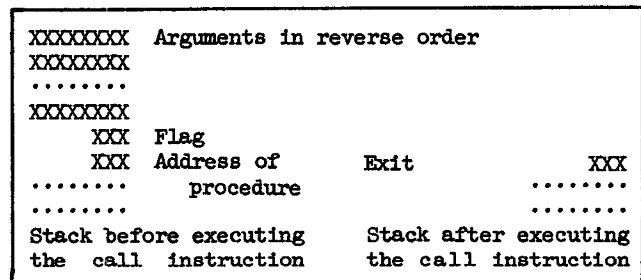

Illustration 2

Map of the Stack Relating to Procedure Calls

number of arguments in the stack does not correspond to the number of arguments in the procedure, an error is indicated. The "flag" in the stack works like this. In the VALGOL II machine there is a flag register. To set a flag in the stack, the contents of this register is put on top of the stack, then the address of the word above the top of the stack is put into the flag register. Initially, and whenever there are no flags in the stack, the flag register contains blanks. At other times it contains the address of the word in the stack which is just above the uppermost flag. Just before a call instruction is executed, the flag register contains the address of the word in the stack which is two above the word containing the address of the procedure to be executed. The call instruction picks up the arguments from the stack, beginning with the one stored just

above the flag, and continuing to the top of the stack. Arguments are moved into the appropriate places at the top of the procedure being called. An error message is given if the number of arguments in the stack does not correspond to the number of places in the procedure. Finally the old flag address, which is just below the procedure address in the stack, is put in the flag register. The exit address replaces the address of the procedure in the stack, and all the arguments, as well as the flag, are popped out. There are just two op codes which affect the flag register. The code "load flag" puts a flag into the stack, and the code "call" takes one out.

The library function "WHOLE" truncates a real number. It does not convert a real number to an integer, because no distinction is made between them. It is substituted for the recommended function "ENTIER" primarily because truncation takes fewer machine instructions to implement. Also, truncation seems to be used more frequently. The procedure ENTIER can be defined in VALGOL II as follows:

```
.PROCEDURE ENTIER(X) .,
   .IF 0 .L= X .THEN WHOLE (X) .ELSE
   .IF WHOLE(X) = X .THEN X .ELSE
   WHOLE(X) -1
```

The "for statement" in VALGOL II is not the same as it is in ALGOL. Exactly one list element is required. The "step .. until" portion of the element is mandatory, but the "while" portion may be added to terminate the loop immediately upon some condition. The iteration continues so long as the value of the variable is less than or equal to the maximum, irrespective of the sign of the increment. Illustration 3 is an example of a typical "for statement". A flow chart of this statement is given in illustration 4.

```
.FOR I = 0 .STEP 1 .UNTIL N .DO
      <statement>
      SET            Set switch to indicate first
                     time through.
A91
      LD    I
      FLP    ]       Test for first time through.
      BFP   A92]
      LDL   0  ]
      SST      ]     Initialize variable.
      B     A93]
A92
      LDL   1  ]     Increment variable.
      ADS      ]
A93
      RSR      ]     Compare variable to maximum.
      LD    N  ]
      LEQ      ]
      BFP   A94]
      <statement>
      RST            Reset switch to indicate not
                     first time through.
      B     A91
A94
```

Illustration 3

Compilation of a typical "for statement" in VALGOL II

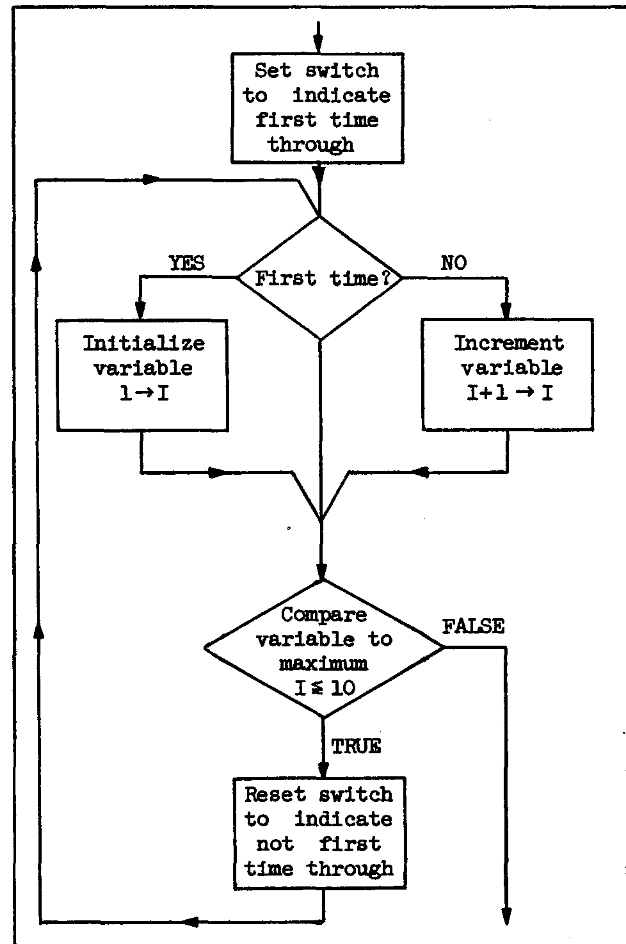

Illustration 4

Flow chart of the "for statement" given in figure 12

Figure 7 is a listing of the VALGOL II compiler written in META II. Figure 8 gives the or-order list of the VALGOL II machine. A sample program to take a determinant is given in figure 9.

Backup vs. No Backup

Suppose that, upon entry to a recursive subroutine, which represents some syntax equation, the position of the input and output are saved. When some non-first term of a component is not found, the compiler does not have to stop with an indication of a syntax error. It can back-up the input and output and return false. The advantages of backup are as follows:

1. It is possible to describe languages, using backup, which cannot be described without backup.

2. Even for a language which can be described without backup, the syntax equations can often be simplified when backup is allowed.

The advantages claimed for non-backup are as follows:

1. Syntax analysis is faster.

2. It is possible to tell whether syntax equations will work just by examining them, without following through numerous examples.

The fact that rather sophisticated languages such as ALGOL and COBOL can be implemented without backup is pointed out by various people, including Conway,[5] and they are aware of the speed advantages of so doing. I have seen no mention of the second advantage of no-backup, so I will explain this in more detail.

Basically one writes alternations in which each term begins with a different symbol. Then it is not possible for the compiler to go down the wrong path. This is made more complicated because of the use of ".EMPTY". An optional item can never be followed by something that begins with the same symbol it begins with.

The method described above is not the only way in which backup can be handled. Variations are worth considering, as a way may be found to have the advantages of both backup and no-backup.

Further Development of META Languages

As mentioned earlier, META II is not presented as a standard language, but as a point of departure from which a user may develop his own META language. The term "META Language," with "META" in capital letters, is used to denote any compiler-writing language so developed.

The language which Schmidt[1] implemented on the PDP-1 was based on META I. He has now implemented an improved version of this language for a Beckman machine.

Rutman[9] has implemented LOGIK, a compiler for bit-time simulation, on the 7090. He uses a META language to compile Boolean expressions into efficient machine code. Schneider and Johnson[10] have implemented META 3 on the IBM 7094, with the goal of producing an ALGOL compiler which generates efficient machine code. They are planning a META language which will be suitable for any block structured language. To this compiler-writing language they give the name META 4 (pronounced metaphor).

References

1. Schmidt, L., "Implementation of a Symbol Manipulator for Heuristic Translation," 1963 ACM Natl. Conf., Denver, Colo.

2. Metcalfe, Howard, "A Parameterized Compiler Based on Mechanical Linguistics," 1963 ACM Natl. Conf., Denver, Colo.

3. Schorre, Val, "A Syntax - Directed SMALGOL for the 1401," 1963 ACM Natl. Conf., Denver, Colo.

4. Glennie, A., "On the Syntax Machine and the Construction of a Universal Compiler," Tech. Report No. 2, Contract NR 049-141, Carnegie Inst. of Tech., July, 1960.

5. Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler," Comm. ACM, July 1963.

6. Irons, E. T., The Structure and Use of the Syntax - Directed Compiler," Annual Review in Automatic Programming, The Macmillan Co., New York.

7. Bastian, Lewis, "A Phrase-Structure Language Translator," AFCRL-Rept-62-549, Aug. 1962.

8. Chomsky, Noam, "Syntax Structures," Mouton and Co., Publishers, The Hague, Netherlands.

9. Rutman, Roger, "LOGIK, A Syntax Directed Compiler for Computer Bit-Time Simulation," Master Thesis, UCLA, August 1964.

10. Schneider, F. W., and G. D. Johnson, "A Syntax-Directed Compiler-Writing Compiler to Generate Efficient Code," 1964 ACM Natl. Conf., Philadelphia.

THE VALGOL I COMPILER WRITTEN IN META II LANGUAGE
FIGURE 1

```
.SYNTAX PROGRAM

PRIMARY = .ID .OUT('LD ' *) /
     .NUMBER .OUT('LDL' *) /
     '(' EXP ')' .,

TERM = PRIMARY $('*' PRIMARY .OUT('MLT') ) .,

EXP1 = TERM $('+' TERM .OUT('ADD') /
     '-' TERM .OUT('SUB') ) .,

EXP = EXP1 ( '.=' EXP1 .OUT('EQU') / .EMPTY) .,

ASSIGNST = EXP '=' .ID .OUT('ST ' *) .,

UNTILST = '.UNTIL' .LABEL *1 EXP '.DO' .OUT('BTP' *2)
     ST .OUT('B  ' *1) .LABEL *2 .,

CONDITIONALST = '.IF' EXP '.THEN' .OUT('BFP' *1)
     ST '.ELSE' .OUT('B  ' *2) .LABEL *1
     ST .LABEL *2 .,

IOST = 'EDIT' '(' EXP ',' .STRING
     .OUT('EDT' *) ')' /
     'PRINT' .OUT('PNT') .,

IDSEQ1 = .ID .LABEL * .OUT('BLK 1') .,

IDSEQ = IDSEQ1 $(',' IDSEQ1) .,

DEC = '.REAL' .OUT('B  ' *1) IDSEQ .LABEL *1 .,

BLOCK = '.BEGIN' (DEC '.,' / .EMPTY)
     ST $('.,' ST) '.END' .,

ST = IOST / ASSIGNST / UNTILST /
     CONDITIONALST / BLOCK .,

PROGRAM = BLOCK .OUT('HLT')
     .OUT('SP  1') .OUT('END') .,

.END
```

ORDER LIST OF THE VALGOL I MACHINE
FIGURE 2

MACHINE CODES

LD AAA	LOAD	PUT THE CONTENTS OF THE ADDRESS AAA ON TOP OF THE STACK.
LDL NUMBER	LOAD LITERAL	PUT THE GIVEN NUMBER ON TOP OF THE STACK.
ST AAA	STORE	STORE THE NUMBER WHICH IS ON TOP OF THE STACK INTO THE ADDRESS AAA AND POP UP THE STACK.
ADD	ADD	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR SUM.
SUB	SUBTRACT	SUBTRACT THE NUMBER WHICH IS ON TOP OF THE STACK FROM THE NUMBER WHICH IS NEXT TO THE TOP, THEN REPLACE THEM BY THIS DIFFERENCE.
MLT	MULTIPLY	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR PRODUCT.
EQU	EQUAL	COMPARE THE TWO NUMBERS ON TOP OF THE STACK. REPLACE THEM BY THE INTEGER 1, IF THEY ARE EQUAL, OR BY THE INTEGER 0, IF THEY ARE UNEQUAL.
B AAA	BRANCH	BRANCH TO THE ADDRESS AAA.
BFP AAA	BRANCH FALSE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
BTP AAA	BRANCH TRUE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
EDT STRING	EDIT	ROUND THE NUMBER WHICH IS ON TOP OF THE STACK TO THE NEAREST INTEGER N. MOVE THE GIVEN STRING INTO THE PRINT AREA SO THAT ITS FIRST CHARACTER FALLS ON PRINT POSITION N. IN CASE THIS WOULD CAUSE CHARACTERS TO FALL OUTSIDE THE PRINT AREA, NO MOVEMENT TAKES PLACE.
PNT	PRINT	PRINT A LINE, THEN SPACE AND CLEAR THE PRINT AREA.
HLT	HALT	HALT.

CONSTANT AND CONTROL CODES

SP N	SPACE	N = 1--9. CONSTANT CODE PRODUCING N BLANK SPACES.
BLK NNN	BLOCK	PRODUCES A BLOCK OF NNN EIGHT CHARACTER WORDS.
END	END	DENOTES THE END OF THE PROGRAM.

A PROGRAM AS COMPILED FOR THE VALGOL I MACHINE
FIGURE 3

```
.BEGIN
.REAL X ., 0 = X .,
              B   A01                                     0000 G 0012
       X                                                  0004
              BLK 001                                     0004
       A01                                                0012
              LDL 0                                       0012 A
              ST  X                                       0021 B 0004
.UNTIL X .= 3 .DO .BEGIN
       A02                                                0025
              LD  X                                       0025 0 0004
              LDL 3                                       0029 A
              EQU                                         0038 F
              BTP A03                                     0039 K 0097
     EDIT( X*X * 10 + 1, '*') ., PRINT ., X + 0.1 = X
              LD  X                                       0043 0 0004
              LD  X                                       0047 0 0004
              MLT                                         0051 E
              LDL 10                                      0052 A
              MLT                                         0061 E
              LDL 1                                       0062 A
              ADD                                         0071 C
              EDT 01'*'                                   0072 I
              PNT                                         0074 0
              LD  X                                       0075 0 0004
              LDL 0.1                                     0079 A
              ADD                                         0088 C
              ST  X                                       0089 B 0004
     .END
              B   A02                                     0093 G 0025
       A03                                                0097
.END
              HLT                                         0097 J
              SP  1                                       0098
              END                                         0099
```

OUTPUT FROM THE VALGOL I PROGRAM GIVEN IN FIGURE 3
FIGURE 4

```
*
*
*
 *
  *
   *
    *
     *
      *
        *
          *
            *
              *
                 *
                    *
                       *
                          *
                             *
                                *
                                    *
                                        *
                                            *
                                                *
                                                     *
                                                          *
                                                               *
                                                                    *
```

THE META II COMPILER WRITTEN IN ITS OWN LANGUAGE
FIGURE 5

```
.SYNTAX PROGRAM

OUT1 = '*1' .OUT('GN1') / '*2' .OUT('GN2') /
'*' .OUT('CI') / .STRING .OUT('CL ' *).,

OUTPUT = ('.OUT' '('
$ OUT1 ')' / '.LABEL' .OUT('LB') OUT1) .OUT('OUT') .,

EX3 = .ID .OUT ('CLL' *) / .STRING
.OUT('TST' *) / '.ID' .OUT('ID') /
'.NUMBER' .OUT('NUM') /
'.STRING' .OUT('SR') / '(' EX1 ')' /
'.EMPTY' .OUT('SET') /
'$' .LABEL *1 EX3
.OUT ('BT ' *1) .OUT('SET').,

EX2 = (EX3 .OUT('BF ' *1) / OUTPUT)
$(EX3 .OUT('BE') / OUTPUT)
.LABEL *1 .,

EX1 = EX2 $('/' .OUT('BT ' *1) EX2 )
.LABEL *1 .,

ST = .ID .LABEL * '=' EX1
'.,' .OUT('R').,

PROGRAM = '.SYNTAX' .ID .OUT('ADR' *)
$ ST '.END' .OUT('END').,

.END
```

ORDER LIST OF THE META II MACHINE
FIGURE 6

MACHINE CODES

TST STRING	TEST	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, COMPARE IT TO THE STRING GIVEN AS ARGUMENT. IF THE COMPARISON IS MET, DELETE THE MATCHED PORTION FROM THE INPUT AND SET SWITCH. IF NOT MET, RESET SWITCH.
ID	IDENTIFIER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH AN IDENTIFIER, IE., A LETTER FOLLOWED BY A SEQUENCE OF LETTERS AND/OR DIGITS. IF SO, DELETE THE IDENTIFIER AND SET SWITCH. IF NOT, RESET SWITCH.
NUM	NUMBER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A NUMBER. A NUMBER IS A STRING OF DIGITS WHICH MAY CONTAIN IMBEDED PERIODS, BUT MAY NOT BEGIN OR END WITH A PERIOD. MOREOVER, NO TWO PERIODS MAY BE NEXT TO ONE ANOTHER. IF A NUMBER IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
SR	STRING	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A STRING, IE., A SINGLE QUOTE FOLLOWED BY A SEQUENCE OF ANY CHARACTERS OTHER THAN SINGLE QUOTE FOLLOWED BY ANOTHER SINGLE QUOTE. IF A STRING IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
CLL AAA	CALL	ENTER THE SUBROUTINE BEGINNING IN LOCATION AAA. IF THE TOP TWO TERMS OF THE STACK ARE BLANK, PUSH THE STACK DOWN BY ONE CELL. OTHERWISE, PUSH IT DOWN BY THREE CELLS. SET A FLAG IN THE STACK TO INDICATE WHETHER IT HAS BEEN PUSHED BY ONE OR THREE CELLS. THIS FLAG AND THE EXIT ADDRESS GO INTO THE THIRD CELL. CLEAR THE TOP TWO CELLS TO BLANKS TO INDICATE THAT THEY CAN ACCEPT ADDRESSES WHICH MAY BE GENERATED WITHIN THE SUBROUTINE.

Figure 6.1

R	RETURN	RETURN TO THE EXIT ADDRESS, POPPING UP THE STACK BY ONE OR THREE CELLS ACCORDING TO THE FLAG. IF THE STACK IS POPPED BY ONLY ONE CELL, THEN CLEAR THE TOP TWO CELLS TO BLANKS, BECAUSE THEY WERE BLANK WHEN THE SUBROUTINE WAS ENTERED.
SET	SET	SET BRANCH SWITCH ON.
B AAA	BRANCH	BRANCH UNCONDITIONALLY TO LOCATION AAA.
BT AAA	BRANCH IF TRUE	BRANCH TO LOCATION AAA IF SWITCH IS ON. OTHERWISE, CONTINUE IN SEQUENCE.
BF AAA	BRANCH IF FALSE	BRANCH TO LOCATION AAA IF SWITCH IS OFF. OTHERWISE, CONTINUE IN SEQUENCE.
BE	BRANCH TO ERROR IF FALSE	HALT IF SWITCH IS OFF. OTHERWISE, CONTINUE IN SEQUENCE.
CL STRING	COPY LITERAL	OUTPUT THE VARIABLE LENGTH STRING GIVEN AS THE ARGUMENT. A BLANK CHARACTER WILL BE INSERTED IN THE OUTPUT FOLLOWING THE STRING.
CI	COPY INPUT	OUTPUT THE LAST SEQUENCE OF CHARACTERS DELETED FROM THE INPUT STRING. THIS COMMAND MAY NOT FUNCTION PROPERLY IF THE LAST COMMAND WHICH COULD CAUSE DELETION FAILED TO DO SO.
GN1	GENERATE 1	THIS CONCERNS THE CURRENT LABEL 1 CELL, IE., THE NEXT TO TOP CELL IN THE STACK, WHICH IS EITHER CLEAR OR CONTAINS A GENERATED LABEL. IF CLEAR, GENERATE A LABEL AND PUT IT INTO THAT CELL. WHETHER THE LABEL HAS JUST BEEN PUT INTO THE CELL OR WAS ALREADY THERE, OUTPUT IT. FINALLY, INSERT A BLANK CHARACTER IN THE OUTPUT FOLLOWING THE LABEL.
GN2	GENERATE 2	SAME AS GN1, EXCEPT THAT IT CONCERNS THE CURRENT LABEL 2 CELL, IE., THE TOP CELL IN THE STACK.
LB	LABEL	SET THE OUTPUT COUNTER TO CARD COLUMN 1.
OUT	OUTPUT	PUNCH CARD AND RESET OUTPUT COUNTER TO CARD COLUMN 8.

Figure 6.2

CONSTANT AND CONTROL CODES

ADR IDENT	ADDRESS	PRODUCES THE ADDRESS WHICH IS ASSIGNED TO THE GIVEN IDENTIFIER AS A CONSTANT.
END	END	DENOTES THE END OF THE PROGRAM.

Figure 6.3

VALGOL II COMPILER WRITTEN IN META II
FIGURE 7

```
.SYNTAX PROGRAM

ARRAYPART = '(.' EXP '.)' .OUT('AIA') .,

CALLPART = '(' .OUT('LDF') (EXP $(',' EXP) /
      .EMPTY) ')' .OUT('CLL') .,

VARIABLE = .ID .OUT('LD ' *) (ARRAYPART / .EMPTY) .,

PRIMARY = 'WHOLE' '(' EXP ')' .OUT('WHL') /
      .ID .OUT('LD ' *) (ARRAYPART / CALLPART / .EMPTY) /
      '.TRUE' .OUT('SET') / '.FALSE' .OUT('RST') /
      '0 ' .OUT('RST') / '1 ' .OUT('SET') /
      .NUMBER .OUT('LDL' *) /
      '(' EXP ')' .,

TERM = PRIMARY $ ('*' PRIMARY .OUT('MLT') /
      '/' PRIMARY .OUT('DIV') /
      './.' PRIMARY .OUT('DIV') .OUT('WHL') ) .,

EXP2 = '-' TERM .OUT('NEG') /
      '+' TERM / TERM .,

EXP1 = EXP2 $('+' TERM .OUT('ADD') /
      '-' TERM .OUT('SUB')) .,

RELATION = EXP1 (
      '.L=' EXP1 .OUT('LEQ') /
      '.L' EXP1 .OUT('LES') /
      '.=' EXP1 .OUT('EQU') /
      '.-=' EXP1 .OUT('EQU') .OUT('NOT') /
      '.G=' EXP1 .OUT('LES') .OUT('NOT') /
      '.G' EXP1 .OUT('LEQ') .OUT('NOT') /
      .EMPTY) .,

BPRIMARY = '.-' RELATION .OUT('NOT') /
      RELATION .,

BTERM = BPRIMARY $ ('.∧' .OUT('BF ' *1)
      .OUT('POP') BPRIMARY)
      .LABEL *1 .,

BEXP1 = BTERM $( '.V' .OUT('BT ' *1)
      .OUT('POP') BTERM)
      .LABEL *1 .,

IMPLICATION1 = '.IMP' .OUT('NOT')
      .OUT('BT ' *1) .OUT('POP')
      BEXP1 .LABEL *1 .,

IMPLICATION = BEXP1 $ IMPLICATION1 .,
```

Figure 7.1

```
EQUIV = IMPLICATION $('.EQ' .OUT('EQU') ) .,

EXP = '.IF' EXP '.THEN' .OUT('BFP' *1)
      EXP .OUT('B  ' *2) .LABEL *1
      '.ELSE' EXP .LABEL *2 /
      EQUIV .,

ASSIGNPART = '=' EXP ( ASSIGNPART .OUT('ST') /
      .EMPTY .OUT('SST') ) .,

ASSIGNCALLST = .ID .OUT('LD ' *) (ARRAYPART ASSIGNPART /
      ASSIGNPART / (CALLPART / .EMPTY
      .OUT('LDF') .OUT('CLL') )
      .OUT('POP') ) .,

UNTILST = '.UNTIL' .LABEL *1 EXP
      '.DO' .OUT('BTP' *2) ST
      .OUT('B  ' *1) .LABEL *2 .,

WHILECLAUSE = '.WHILE' .OUT('BF ' *1)
      .OUT('POP') EXP .LABEL *1 / .EMPTY .,

FORCLAUSE = VARIABLE '=' .OUT('FLP')
      .OUT('BFP' *1) EXP '.STEP'
      .OUT('SST') .OUT('B  ' *2)
      .LABEL *1 EXP '.UNTIL' .OUT('ADS')
      .LABEL *2 .OUT('RSR') EXP
      .OUT('LEQ') WHILECLAUSE '.DO' .,

FORST = '.FOR' .OUT('SET') .LABEL *1
      FORCLAUSE .OUT('BFP' *2) ST
      .OUT('RST') .OUT('B  ' *1)
      .LABEL *2 .,

IOCALL = 'READ' '(' VARIABLE ',' EXP ')' .OUT('RED') /
      'WRITE' '(' VARIABLE ',' EXP ')' .OUT('WRT') /
      'EDIT' '(' EXP ',' .STRING
      .OUT('EDT' *) ')' /
      'PRINT' .OUT('PNT') /
      'EJECT' .OUT('EJT') .,

IDSEQ1 = .ID .LABEL* .OUT('BLK 1') .,

IDSEQ = IDSEQ1 $(',' IDSEQ1) .,

TYPEDEC = '.REAL' IDSEQ .,

ARRAY1 = .ID .LABEL * '(.' '0' '..' .NUMBER
      .OUT('BLK 1') .OUT('BLK' *) '.)' .,

ARRAYDEC = '.ARRAY' ARRAY1 $( ',' ARRAY1) .,

PROCEDURE = '.PROCEDURE' .ID .LABEL *
      .LABEL *1 .OUT('BLK 1') '('
      (IDSEQ / .EMPTY) ')' .OUT('SP  1') '.,'
      ST .OUT('R  ' *1) .,
```

Figure 7.2

```
DEC = TYPEDEC / ARRAYDEC / PROCEDURE .,

BLOCK = '.BEGIN' .OUT('B  ' *1) $(DEC '.,')
      .LABEL *1 ST $('.,' ST) '.END'
      (.ID / .EMPTY) .,

UNCONDITIONALST = IOCALL / ASSIGNCALLST /
      BLOCK .,

CONDST = '.IF' EXP '.THEN' .OUT('BFP' *1)
      (UNCONDITIONALST ('.ELSE' .OUT('B  ' *2)
      .LABEL *1 ST .LABEL *2 / .EMPTY
      .LABEL *1) / (FORST / UNTILST)
      .LABEL *1) .,

ST = CONDST / UNCONDITIONALST / FORST /
      UNTILST / .EMPTY .,

PROGRAM = BLOCK
      .OUT('HLT') .OUT('SP  1') .OUT('END') .,

.END
```

Figure 7.3

ORDER LIST OF THE VALGOL II MACHINE
FIGURE 8

MACHINE CODES

Code	Name	Description
LD AAA	LOAD	PUT THE ADDRESS AAA ON TOP OF THE STACK.
LDL NUMBER	LOAD LITERAL	PUT THE GIVEN NUMBER ON TOP OF THE STACK.
SET	SET	PUT THE INTEGER 1 ON TOP OF THE STACK.
RST	RESET	PUT THE INTEGER 0 ON TOP OF THE STACK.
ST	STORE	STORE THE CONTENTS OF THE REGISTER, STACK1, IN THE ADDRESS WHICH IS ON TOP OF THE STACK, THEN POP UP THE STACK.
ADS	ADD TO STORAGE NOTE 1	ADD THE NUMBER ON TOP OF THE STACK TO THE NUMBER WHOSE ADDRESS IS NEXT TO THE TOP, AND PLACE THE SUM IN THE REGISTER, STACK1. THEN STORE THE CONTENTS OF THAT REGISTER IN THAT ADDRESS, AND POP THE TOP TWO ITEMS OUT OF THE STACK.
SST	SAVE AND STORE NOTE 1	PUT THE NUMBER ON TOP OF THE STACK INTO THE REGISTER, STACK1. THEN STORE THE CONTENTS OF THAT REGISTER IN THE ADDRESS WHICH IS THE NEXT TO TOP TERM OF THE STACK. THE TOP TWO ITEMS ARE POPPED OUT OF THE STACK.
RSR	RESTORE	PUT THE CONTENTS OF THE REGISTER, STACK1, ON TOP OF THE STACK.
ADD	ADD NOTE 2	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR SUM.
SUB	SUBTRACT NOTE 2	SUBTRACT THE NUMBER WHICH IS ON TOP OF THE STACK FROM THE NUMBER WHICH IS NEXT TO THE TOP, THEN REPLACE THEM BY THIS DIFFERENCE.
MLT	MULTIPLY NOTE 2	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR PRODUCT.
DIV	DIVIDE NOTE 2	DIVIDE THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK BY THE NUMBER WHICH IS ON TOP OF THE STACK, THEN REPLACE THEM BY THIS QUOTIENT.

Figure 8.1

Code	Name	Description
NEG	NEGATE NOTE 1	CHANGE THE SIGN OF THE NUMBER ON TOP OF THE STACK.
WHL	WHOLE	TRUNCATE THE NUMBER WHICH IS ON TOP OF THE STACK.
NOT	NOT	IF THE TOP TERM IN THE STACK IS THE INTEGER 0, THEN REPLACE IT WITH THE INTEGER 1, OTHERWISE, REPLACE IT WITH THE INTEGER 0.
LEQ	LESS THAN OR EQUAL NOTE 2	IF THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK IS LESS THAN OR EQUAL TO THE NUMBER ON TOP OF THE STACK, THEN REPLACE THEM WITH THE INTEGER 1. OTHERWISE, REPLACE THEM WITH THE INTEGER 0.
LES	LESS THAN NOTE 2	IF THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK IS LESS THAN THE NUMBER ON TOP OF THE STACK, THEN REPLACE THEM WITH THE INTEGER 1. OTHERWISE, REPLACE THEM WITH THE INTEGER 0.
EQU	EQUAL NOTE 2	COMPARE THE TWO NUMBERS ON TOP OF THE STACK. REPLACE THEM BY THE INTEGER 1, IF THEY ARE EQUAL, OR BY THE INTEGER 0, IF THEY ARE UNEQUAL.
B AAA	BRANCH	BRANCH TO THE ADDRESS AAA.
BT AAA	BRANCH TRUE	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. DO NOT POP UP THE STACK.
BF AAA	BRANCH FALSE	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. DO NOT POP UP THE STACK.
BTP AAA	BRANCH TRUE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
BFP AAA	BRANCH FALSE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.

Figure 8.2

Code	Name	Description
CLL	CALL	ENTER A PROCEDURE AT THE ADDRESS WHICH IS BELOW THE FLAG.
LDF	LOAD FLAG	PUT THE ADDRESS WHICH IS IN THE FLAG REGISTER ON TOP OF THE STACK, AND PUT THE ADDRESS OF THE TOP OF THE STACK INTO THE FLAG REGISTER.
R AAA	RETURN	RETURN FROM PROCEDURE.
AIA	ARRAY INCREMENT ADDRESS	INCREMENT THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK BY THE INTEGER WHICH IS ON TOP OF THE STACK, AND REPLACE THESE BY THE RESULTING ADDRESS.
FLP	FLIP	INTERCHANGE THE TOP TWO TERMS OF THE STACK.
POP	POP	POP UP THE STACK.
EDT STRING	EDIT NOTE 1	ROUND THE NUMBER WHICH IS ON TOP OF THE STACK TO THE NEAREST INTEGER N. MOVE THE GIVEN STRING INTO THE PRINT AREA SO THAT ITS FIRST CHARACTER FALLS ON PRINT POSITION N. IN CASE THIS WOULD CAUSE CHARACTERS TO FALL OUTSIDE THE PRINT AREA, NO MOVEMENT TAKES PLACE.
PNT	PRINT	PRINT A LINE, THEN SPACE AND CLEAR THE PRINT AREA.
EJT	EJECT	POSITION THE PAPER IN THE PRINTER TO THE TOP LINE OF THE NEXT PAGE.
RED	READ	READ THE FIRST N NUMBERS FROM A CARD AND STORE THEM BEGINNING IN THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK. THE INTEGER N IS THE TOP TERM OF THE STACK. POP OUT BOTH THE ADDRESS AND THE INTEGER. CARDS ARE PUNCHED WITH UP TO 10 EIGHT DIGIT NUMBERS. DECIMAL POINT IS ASSUMED TO BE IN THE MIDDLE. AN 11-PUNCH OVER THE RIGHTMOST DIGIT INDICATES A NEGATIVE NUMBER.

Figure 8.3

Code	Name	Description
WRT	WRITE	PRINT A LINE OF N NUMBERS BEGINNING IN THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK. THE INTEGER N IS THE TOP TERM OF THE STACK. POP OUT BOTH THE ADDRESS AND THE INTEGER. TWELVE CHARACTER POSITIONS ARE ALLOWED FOR EACH NUMBER. THERE ARE FOUR DIGITS BEFORE AND FOUR DIGITS AFTER THE DECIMAL. LEADING ZEROES IN FRONT OF THE DECIMAL ARE CHANGED TO BLANKS. IF THE NUMBER IS NEGATIVE, A MINUS SIGN IS PRINTED IN THE POSITION BEFORE THE FIRST NON-BLANK CHARACTER.
HLT	HALT	HALT.

CONSTANT AND CONTROL CODES

Code	Name	Description
SP N	SPACE	N = 1--9. CONSTANT CODE PRODUCING N BLANK SPACES.
BLK NNN	BLOCK	PRODUCES A BLOCK OF NNN EIGHT CHARACTER WORDS.
END	END	DENOTES THE END OF THE PROGRAM.

NOTE 1. IF THE TOP ITEM IN THE STACK IS AN ADDRESS, IT IS REPLACED BY ITS CONTENTS BEFORE BEGINNING THIS OPERATION.

NOTE 2. SAME AS NOTE 1, BUT APPLIES TO THE TOP TWO ITEMS.

Figure 8.4

EXAMPLE PROGRAM IN VALGOL II
FIGURE 9

```
.BEGIN
.PROCEDURE DETERMINANT(A, N) .,
.BEGIN
.PROCEDURE DUMP() .,
.BEGIN
.REAL D .,
.FOR D = 0 .STEP 1 .UNTIL N-1 .DO
     WRITE(MATRIX(. N*D .), N) .,
PRINT
.END DUMP .,
.PROCEDURE ABS(X) .,
     ABS = .IF 0 .L= X .THEN X .ELSE -X .,
.REAL PRODUCT, FACTOR, TEMP, R, I, J .,
PRODUCT = 1 .,
.FOR R = 0 .STEP 1 .UNTIL N-2
.WHILE PRODUCT .-= 0 .DO .BEGIN
     I = R .,
     .FOR J = R+1 .STEP 1 .UNTIL N-1 .DO
          .IF ABS( A(. N*I + R .) ) .L
          ABS( A(. N*J + R .) ) .THEN
               I = J .,
     .IF A(. N*I + R .) .= 0 .THEN
          PRODUCT = 0
     .ELSE
          .IF I .-= R .THEN .BEGIN
               PRODUCT = -PRODUCT .,
               .FOR J = R .STEP 1 .UNTIL N-1 .DO
               .BEGIN
                    TEMP = A(. N*R + J .) .,
                    A(. N*R + J .) = A(. N* I + J .) .,
                    A(. N*I + J .) = TEMP .END .END .,
          TEMP = A(. N*R + R .) .,
          .FOR I = R+1 .STEP 1 .UNTIL N-1 .DO
          .BEGIN
               FACTOR = A(. N*I + R .) / TEMP .,
               .FOR J = R .STEP 1 .UNTIL N-1 .DO
                    A(. N*I + J .) = A(. N*I + J .)
                    -FACTOR * A(. N*R + J .) .,
          DUMP
          .END .END .,
.FOR I = 0 .STEP 1 .UNTIL N-1 .DO
     PRODUCT = PRODUCT * A(. N*I + I .) .,
DETERMINANT = PRODUCT
.END DETERMINANT .,
.REAL M, W, T .. .ARRAY MATRIX (. 0 .. 24 .) .,
.UNTIL .FALSE .DO .BEGIN
     EDIT(1, 'FIND DETERMINANT OF' ) ., PRINT., PRINT.,
     READ(M, 1) .,
     .FOR W = 0 .STEP 1 .UNTIL M-1 .DO .BEGIN
          READ(MATRIX (. M*W .), M) .,
          WRITE(MATRIX (. M*W .), M) .END .,
     PRINT .,  T = DETERMINANT (MATRIX, M) .,
     WRITE(T, 1) ., PRINT., PRINT .END
.END PROGRAM
```