1965

WHAT

James A. Moore

Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making
of photocopies or other reproductions of copyrighted material. Any copying of this
document without permission of its author may be prohibited by law.
ACKNOWLEDGEMENTS

The publication of this manual represents the combined efforts of several people. The manual was written under the direction of Professor Robert Braden; the author gratefully acknowledges his advice and encouragement.

The WHAT language was added to ALGOL-20 by Ronald Bushyager with assistance from David Blocher.

The author is deeply indebted to Professor Braden, Arthur Evans Jr., Jan Fierst, Ron Bushyager and Dave Blocher for invaluable advice concerning form and technical content of the manual.

Finally, as a result of the similarities between WHAT and THAT, the author borrowed heavily from portions of the THAT manual.
# TABLE OF CONTENTS

## I. ELEMENTS OF WHAT

1.1 Symbols

1.2 WHAT Symbols
   1.2.1 Label Symbols
   1.2.2 Region Symbols
   1.2.3 The "A" Symbols
   1.2.4 Scope of Definition
   1.2.5 Precedence of Definition

1.3 Expressions

## II. SOURCE PROGRAM FORMAT

2.1 Language Field
2.2 Location Field
2.3 Operation Field
2.4 Mode Field
2.5 Address Field
2.6 Index Field
2.7 Comment Field

## III. WHAT/ALGOL INTERACTION

3.1 Direct Access of ALGOL Symbols
3.2 Indirect Access of ALGOL Symbols
   3.2.1 Forward and Cross-Block Transfers
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2 Subscripted Variables</td>
</tr>
<tr>
<td>3.2.3 Formal Parameter Called by Name</td>
</tr>
<tr>
<td>3.3 Cross-Block Transfers</td>
</tr>
<tr>
<td>3.4 Index Allocation</td>
</tr>
<tr>
<td>3.5 Statement Termination</td>
</tr>
<tr>
<td>IV. SUDO INSTRUCTIONS IN WHAT.</td>
</tr>
<tr>
<td>V. ERROR MESSAGES</td>
</tr>
<tr>
<td>5.1 Errors Detected During Compilation</td>
</tr>
<tr>
<td>5.1.1 Errors in G-20 Instructions</td>
</tr>
<tr>
<td>5.1.2 Errors in Sudo Instructions</td>
</tr>
<tr>
<td>5.2 Errors Detected During Running</td>
</tr>
<tr>
<td>APPENDICES</td>
</tr>
<tr>
<td>A. G-20 ALPHABET</td>
</tr>
<tr>
<td>B. G-20 'WHAT' OPCODES</td>
</tr>
<tr>
<td>C. SUDOS IN 'WHAT'</td>
</tr>
<tr>
<td>D. G-20 SHIFT MULTIPLIERS</td>
</tr>
<tr>
<td>OCTAL - DECIMAL TABLE</td>
</tr>
<tr>
<td>E. SAMPLE WHAT/ALGOL PROGRAM</td>
</tr>
</tbody>
</table>
CHAPTER 1 - ELEMENTS OF WHAT

WHAT is a symbolic assembly program designed to permit the use of G-20 machine code within an ALGOL program, in order to achieve efficiencies and/or capabilities unavailable in ALGOL alone. This manual describes the WHAT language and the associated assembly program developed at the Carnegie Institute of Technology Computation Center. The reader is referred to the G-20 reference manual ("Central Processor/Machine Language Manual", CDC G-20 Publication #611) for information on the logical organization, word formats, arithmetic rules, addressing schemes, and operations of the Central Processor. SECTION 2 of the Computation Center User's Manual describes the hardware modifications which have been made to the Carnegie Tech system, converting it from a G-20 to a G-21. Finally, it is assumed that the user is familiar with ALGOL-20, the C.I.T. implementation of ALGOL-60. (See "ALGOL-20, A Language Manual", C.I.T. C.C., Jan Fierst, Editor.)

If during compilation the ALGOL compiler reads a card with "WH" in the language field, control is transferred to the WHAT assembler. The assembler reads the source cards containing code in the WHAT language and translates ("assembles") them into binary machine language in core memory. This translation process is generally one-for-one; thus, each WHAT statement, occupying a separate line or "card image" of the source program, is generally translated into a single binary instruction or data word. When the WHAT assembler reads a card with "AL" in the language field, the ALGOL compiler resumes operation. See Appendix E for an example of a WHAT/ALGOL program.
WHAT code may appear only where a statement or a declaration is permitted. In any other context, the use of WHAT code will be treated as an error condition, but it will be assembled anyway. With this exception, WHAT cards have no effect on the ALGOL compiler. (See Section 3.5, p.27.)

Like the rest of the ALGOL compiler, the WHAT Assembler performs the translation with only one pass over the source deck, assembling the absolute instructions directly into core memory without the use of an intermediate "scratch tape". Instructions from the source program are (normally) assembled into the core locations from which they will subsequently be executed; at present there is no provision for automatic relocation. As each card image of the source program is processed, its image is listed on the printer along with the address in octal of the core location into which the corresponding binary instruction is assembled. Printing may be suppressed with the appropriate "SY" card. (See ALGOL Manual, Chapter 4.)

The WHAT language is "symbolic", meaning that symbols may be used for machine addresses and mnemonic names may be used for operation codes. Since it operates in a single pass, the WHAT Assembler may encounter address fields which contain WHAT symbols which have not yet been defined. The Assembler keeps lists of all such occurrences of undefined address symbols, and when the symbol is subsequently defined all references to it are properly "fixed up" in the assembled instructions in core memory. There are some important restrictions on the use of such undefined symbols; see Section 1.3, p. 11.

The index field of a WHAT statement is further restricted; all symbols must be defined before being encountered in an index field. There is no
provision for "fixing up" undefined symbols used in the index field.

In general, each line of WHAT code includes an operation code -- usually as a three-letter mnemonic but possibly in absolute octal form. These mnemonics must be one of the following:

(1) A standard G-20 machine language opcode mnemonic, as listed in Appendix B of this manual; or

(2) A "sudo" (pseudo-instruction) mnemonic. A sudo does not stand for an actual machine command but is rather an instruction to the WHAT Assembler, to be executed when the sudo is encountered during the assembly process. All WHAT sudos are listed in alphabetical order with an explanation, in Chapter 4 of this manual, page 28.
1.1 SYMBOLS

There are two kinds of symbols available for use within WHAT:

(1) WHAT symbols.

These are declared and defined by WHAT coding and may be used only within WHAT.

(2) ALGOL symbols.

These are declared and defined by ALGOL declarations and may be used in either ALGOL or WHAT.

For the remainder of this chapter, all references to "symbol" will mean a WHAT symbol.
1.2 WHAT SYMBOLS

The purpose of symbols is three-fold: (1) The programmer may refer symbolically to an address which will not be known until the entire program has been written and assembled. (2) The programmer may parameterize his program and assign values to the parameters at assembly time, so that sizes of buffers, data storage blocks, program segments, etc., may subsequently be changed by simple reassembly runs. (3) The symbols may give some mnemonic value to the program, aiding the programmer in the task of writing, debugging, and changing the program.

Each WHAT symbol has the form of a class name followed by an integer; the integer is referred to as the "subscript" part of the symbol. Class names are one character, and may be any of the 26 letters or one of the four special characters: →, ←, →, or |. These rules are summarized by the following syntax: (See Users Manual, Section 2.146 for details on B.N.F.)

```plaintext
<class name> ::= <letter> | ← | → | ← | <the mark "|">  
<subscript> ::= <integer> | <empty>  
<symbol> ::= <class name> <subscript>
```

Notice that the subscript may be omitted; in this case a subscript of zero is assumed except in the case of the "LBL", "CHK", "PRT" and "REL" symbols where the maximum declared subscript is assumed.

Some symbols, with class name "-" and "|", are predefined at the beginning of each translation. These symbols give the programmer access to routines and information in both the monitor and the running ALGOL program environment. The symbols |0 to |39 are defined in the THEM THINGS write up, and provide reference to the monitor. |40 to |99 (the "upper bars") are available to Computation Center staff members and certain others for monitor references needed by system programmers. For information, see the User Consultant. The
symbols from 200 up are used for certain quantities connected with the ALGOL input/output system, and are described in p. 0.15 of the ALGOL Manual. The "L" symbols are defined to let the user refer to many of the routines and switches of the ALGOL run-time environment. They are listed in the ALGOL Manual, p. 0.14, although in many cases an understanding of some of these symbols requires more information about ALGOL-20 than appears in any document.

Examples of WHAT symbols:

L4
-27
3

T (same as: TO, except in "LBL", "REL", "FRT" and "CHK" suds)

The possible symbols are divided into 30 classes by the class names. All symbols of a particular class will be either:

(1) **Label symbols**, whose values may be defined independently and in any order; or

(2) **Regional symbols**, which bear a fixed relationship to each other, and so are all defined when any member of the class is defined.

These two kinds are discussed in Sections 1.2.1 and 1.2.2, below. The one class name "A" has special significance, and is discussed in Section 1.2.3.

Symbols are most frequently used to represent addresses with values between 0 and $2^{15}-1$. However, a symbol may be defined (by a "DEF" sudo) to have any value between 0 and $2^{21}-1$. 

1.2.1 LABEL SYMBOLS

All symbols with a particular class name may be declared to be label symbols with an "LBL" ("LaBeL") sudo instruction. The "LBL" sudo contains the class name character followed by the maximum subscript integer which labels of the class will be allowed. For example:

```
LBL K20
```

declares a set of 21 label symbols: K0, K1, K2, ..., K20. These symbols are free and arbitrary and may be defined in any order with any set of values. Label symbols are defined in one of two ways:

(1) explicitly, with a "DEF" sudo, or

(2) implicitly, by appearing in the location field of an instruction.

In this case the symbol is defined to be equal to the current value of the Assembler's location counter for that instruction.

The symbols of the class are related only in that at assembly time they occupy adjacent positions in the symbol table created by the Assembler. This fact may be of importance to the programmer who needs to reuse symbols or reclaim label table space during assembly of very large programs; see the sudo instructions "CHK", "LBL" and "REL" in Chapter 4 for more information. The maximum subscript given in the "LBL" declaration is used by the Assembler to allocate label table space.
1.2.2 REGION SYMBOLS

A class name denotes a region if:

1. that class has not previously been declared as consisting of label symbols (by a "LBL" sudo instruction), and if
2. any symbol in that class is given a value (by a "DEF" sudo instruction).

All symbols with the same regional class name refer to the same area of memory, and their values are related in a fixed way: the symbol whose subscript part is the integer n stands for the nth memory address of the region. Thus, defining any one symbol of the class defines them all. For example, assume that R has not appeared in a "LBL" declaration. Then the line:

```
DEF RO = 4000
```

will make R a region whose first cell is address 4000. Then all R symbols will be defined; e.g. R9 = 4009 and in general Rn = 4000 + n where n is any integer. The following "DEF" operation would have the same effect:

```
DEF R9 = 4009
```

The expressions R0 + 23 and R + 23 are equivalent to the symbol R23, if R is a region.

A class of symbols which has been used as a region may later be declared in a "LBL" sudo instruction and thereafter be used as independent label symbols. Conversely, a class name which has been used for labels may be changed into a region by releasing it from its role as a label class with an instruction of the form:

```
REL <class name>
```

and subsequently defining a member of the class with a "DEF" sudo. Regional symbols need not (and may not) be "RELeased" for redefinition.
1.2.3 THE "A" SYMBOLS

The symbols in class "A" have special significance in the WHAT language and may not be used as label symbols. The symbol "A" or "AO" always has as value the current value of the Assembler's location counter; i.e. the memory location into which the current instruction is to be assembled. After processing each line of the source program, the Assembler increments the value of "A" by the number of words it has loaded into memory. The value of "A" at the beginning of processing each line is printed if the assembly of that line changes that value.

The "A" symbols other than "AO" behave as if "A" were a region name; that is, "An" has the value: "A" + n.

Example:

```
TRA A + 3   (or: TRA A3)
```

has the same effect as:

```
LI TRA LI + 3
```

where L is a label class name.

1.2.4 SCOPE OF DEFINITION

Once a symbol has been defined, its definition is available for use, within WHAT, until it is explicitly redefined, or, if it is a label, until its class is released. The scope of definition of a symbol is not affected by any number of intervening ALGOL cards, regardless of their content.

Consequently, WHAT symbols are not subject to ALGOL block structure. (See Section 3.3, p. 26.)
1.2.5 PRECEDENCE OF DEFINITION

Some symbols are legal in both WHAT and ALGOL. If a symbol is defined differently in the two languages, the definition which takes precedence depends on the language being used. In WHAT, the WHAT definition is used; in ALGOL, the ALGOL definition takes precedence. In ALGOL, WHAT definitions are never available so the conflict does not arise. When WHAT encounters a symbol of the form: <letter> or <letter> <integer> it first looks for the symbol in the WHAT symbol table. If this search succeeds, the WHAT definition is used; if not, WHAT uses the ALGOL definition.

Example:

```
AL real A, B, C, D ;
   index X, Y, Z ;
   ...
WH DEF B7 = /1007 ;
WH CLA B, X ;
   ...
```

The "B" used in WHAT has the value of /1000 and is not related to the real variable declared in ALGOL. However, the "X" refers to the ALGOL index variable if "X0" is not defined in WHAT.
1.3 EXPRESSIONS

Symbols may be used to build expressions, whose syntax may be defined as follows:

<octal digit> ::= 0|1|2|3|4|5|6|7
<digit> ::= <octal digit> | 8|9
<integer> ::= <digit> | <integer> <digit>
<octal integer> ::= /<octal digit> | <octal integer> <octal digit>
<number> ::= (as defined in ALGOL-60 report)
<8 octal> ::= 8I<integer> | 8R<integer> | 8F<number> -- (See ALGOL Manual octal constant>, p. 5e.1)
<power of two> ::= $<integer>
<operator> ::= +|−|*|/
<primary> ::= <defined symbol> | <integer> | <octal integer>
 | <power of two> | <8 octal>
<term> ::= <primary> | <term> <operator> <primary>
<expression> ::= <term> | WHAT symbol | ALGOL symbol | <empty>

(Note that these definitions are for the purpose of this manual only, and are not necessarily related to similar definitions for ALGOL.)

Examples:

418
/77 * $12
LI -6-L0*/3

Here <defined symbol> means a WHAT symbol whose value has been defined previously in the assembly. The symbol must have been defined in one of the following ways:
(1) It may be a regional or label symbol which has received a value from a "DEF" sudo.

(2) It may be a label symbol which has appeared in the location field of a previous instruction.

(3) It may be a pre-defined "$\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\..
is lost. Thus, $8L4 = 0$ and $8L7 = 8L3$. "$<\text{integer}>"$ is equivalent to "$8<\text{integer}>"$.

The character-pairs $8L$, $8R$ and $8F$ are treated by the translator as single entities and must be punched in adjacent columns without intervening blanks. The translator does not treat the digits 8 and 9 in octal constants as erroneous but will interpret them as $10_8$ and $11_8$, respectively. Thus $8R495 = 8R515$.

All WHAT symbols which are not yet defined may appear in an expression only if the expression consists of that symbol alone.

Likewise, ALGOL symbols should not be used in expressions except by themselves. Violations of this last restriction will not be error-flagged but will generally give undesired results.

The value of an expression is computed in double-precision arithmetic format. Address, index, command and mode fields are evaluated, shifted to the appropriate position, united, and the resulting 32-bit logic word is stored in the program being assembled. It is the programmer's responsibility to see that the value of the address expression does not exceed $2^{15} - 1$ and the index expression does not exceed $63_n$, since the Assembler does not check for this condition.
CHAPTER 2 - SOURCE PROGRAM FORMAT

A line of WHAT language source code contains information in some or all of the following fixed fields:

<table>
<thead>
<tr>
<th>Contents</th>
<th>Cols.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Language</td>
<td>1 - 2</td>
</tr>
<tr>
<td>2. Location</td>
<td>4 - 12</td>
</tr>
<tr>
<td>3. Operation</td>
<td>15 - 17</td>
</tr>
<tr>
<td>4. Mode</td>
<td>20</td>
</tr>
<tr>
<td>5. Address, Index; Comments</td>
<td>24 - RIGHT MARGIN</td>
</tr>
</tbody>
</table>

The RIGHT MARGIN is initially set to column 72 but may be changed with the appropriate "SY" card. (See ALGOL-20 Manual, Chapter 4.)

Example:

(cols.)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

WH | E4 | CLA | J4 |

/77, R2; GET NEXT VALUE.

2.1 LANGUAGE FIELD (Columns 1-2)

When card images are typed-in from a remote, the language field is used to set the meaning of the TAB key for the language. The mnemonic "WH" will set the TAB columns for WHAT card images as follows:

<table>
<thead>
<tr>
<th>Tab</th>
<th>Column</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>Label</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>Opcode</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>Mode</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>Address, Index; Comments</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>Comments (See Section 2.7, p. 20.)</td>
</tr>
</tbody>
</table>

For more details, see SECTION 2 of the User's Manual.
2.2 LOCATION FIELD (Columns 4-12)

In general the location field will be blank unless a reference is made to that line of code. The location field may contain any of the following:

1. Blank
2. A label which is currently undefined. The effect is to define that label by giving it the current value of the location counter ("A").
3. An expression which equals the current value of the location counter. This may be used for explanatory or documentary purposes.
4. A <string of operators, letters or numbers>. This may be used as a comment.

Examples:

(case 2)
MPY M5 ; shift rt 5 octals
...
M5 105 1 ; shift constant

(case 2)
LXP 0 20, I ; SET UP TO
E2 STZ P0, I ; ZERO A LOCATION
SXT 0 1, I ; DECREMENT AND TEST
TRA E2 ; LOOP
16.

(case 3)

L2   TRA   A7   ; transfer around
LND  1,2,3,4,5,6 ; table of integers
L2+7 TRM   Q5

(case 4)

A-3  CLA   0   X,I   ; get ith Y
SUX   0   1,I   ; step i
FGO   E0   ; compare w/E0
TRA   A-3 ; loop if greater
2.3 OPERATION FIELD (Columns 15-17)

The operation field may contain one of the following:

1. Blanks. The line will be processed as a "COM" sudo, i.e., a comment card.

2. An octal integer (without the preceding slash). In this case, it will be interpreted as the operation part of a G-20 instruction and the octal integer will occur right-justified in bits 29 to 21 of the assembled instruction.

3. The three-letter mnemonic for a G-20 operation. The corresponding octal code will be loaded into bits 29 to 21 of the assembled instruction. G-20 mnemonics are listed in Appendix B.

4. The mnemonic for a WHAT sudo. The action taken for the possible sudos is described in Chapter 4.

The operation field must be either 3 letters, 3 digits or 3 blanks. Any mixture of these will generate garbage and may not be error flagged.
2.4 MODE FIELD (Column 20)

Each G-20 mnemonic has associated with it a "normal" mode for that operation as described below. If the normal mode is desired, the mode field should be left empty; otherwise, the desired mode: 0, 1, 2, or 3, must be punched. A mode punch always supercedes the normal mode. The mode field of a sudo is checked for error but is otherwise ignored.

All G-20 mnemonics have a normal mode of 2 except the following, which have a normal mode of 0.

<table>
<thead>
<tr>
<th>STI</th>
<th>STL</th>
<th>TRA</th>
<th>REP</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS</td>
<td>STZ</td>
<td>TRM</td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.5 ADDRESS FIELD (Columns 24 - RIGHT MARGIN)

The address field contains the operand or the address of the operand. Blanks in the address field are ignored (except in "ALF" and "NAM" sudos). The address is terminated by a comma, a semi-colon, or RIGHT MARGIN + 1 (which is not scanned), whichever occurs first. If it is terminated by a comma, an index is then expected.

If the operation field of a line contains a G-20 mnemonic or an octal integer, the following applies to the address field:

1. If it is blank, the address (bits 14 - 0) of the assembled instruction will be zero.
2. If it is a single symbol which is already defined, the value of the symbol will be placed in the address (bits 14 - 0) of the assembled instruction. If the symbol is a label which is not yet
If any index register is to be specified, the address field must be terminated by a comma, followed by a symbol (or expression) whose value is the address of an index register. Blanks in the index field are ignored, and the field is terminated by a semi-colon or the RIGHT MARGIN +1 (which is not scanned), whichever occurs first.

The value of the expression in the index field is loaded right-justified into bits 20 – 15 of the assembled instruction. If the value is not defined, an error message will be printed. No error message will be printed if the value of the index field is greater than 63.

Since the monitor and ALGOL are both doing things behind the user's back, it is unsafe for a user to choose his own index registers. It is strongly recommended that only ALGOL variables of type index be used in the index field. (See Sections 3.1 and 3.4, p 21 and 26.)
2.7 COMMENT FIELD (Columns 24-80)

All columns to the right of the first semi-colon in the address-index field are ignored by the Assembler, and may therefore be used for comments, which may extend to Column 80. All columns of the input line including the AND sequence number are printed. A tab to column 40 is included in the tab table to allow the user to align his comments. However, columns 40 - RIGHT MARGIN are part of the address-index field unless a ";" has appeared previously.
3.1 DIRECT ACCESS OF ALGOL SYMBOLS

In four cases ALGOL symbols may be referenced directly in WHAT code:
1) Simple variables
2) Index variables
3) Backwards transfers within the same block
4) Formal parameters called by value.

EXAMPLE:

```
AL begin
  real ALPHA;
  index BETA;
  GAMMA:
  ...
WH LXP 0 1, BETA;
  CLA 3 ALPHA, BETA;
  TRA GAMMA;
  ...
AL end
```

The only precaution necessary in these cases is that the type of access (single, double, logic, index, transfer address) matches the type of the ALGOL symbol referenced.
3.2 INDIRECT ACCESS ALGOL SYMBOLS

Three cases require special treatment:
1) Forward and cross-block transfers
2) Subscripted variables
3) Formal parameters called by name

3.2.1 FORWARD AND CROSS-BLOCK TRANSFERS

A forward or cross-block transfer in WHAT to an ALGOL label will not assemble properly and may not be detected as an error. This problem may be skirted in one of two ways:

If the "TRA" and the ALGOL label are at different block levels, the "TRA" must be replaced by an ALGOL "go to" statement.

EXAMPLES:

```
WH TRA ALPHA;
...
AL begin real X;
ALPHA:
...
```

must be replaced by:

```
WH ...
AL go to ALPHA;
WH ...
...
AL begin real X;
ALPHA:
...
```

The "go to" compiles into two locations and must not follow a test instruction.
If the transfer does not cross block boundaries, it may be effected by a "TRA" in WHAT to a WHAT label which is defined to have the desired value. The following two forms are equivalent:

EXAMPLES:

...  
WH   TRA   L7;        
...  
WH L7   COM   DEFINE L7
AL     ALPHA;          
...  
...  or  ---  ---  
...  
WH   TRA   L7;        
...  
AL     ALPHA;          
WH   DEF   L7 = ALPHA;  
AL     ...
3.2.2 SUBSCRIPTED VARIABLES

To access a subscripted variable, the best method is to use the available ALGOL machinery. To place the value of a subscripted variable into the accumulator, use the reserved identifier, "ACC":

```
WH ...
AL ACC <-<subscripted variable>;
WH ... ; accumulator = desired value.
```

To store the value of the accumulator into a subscripted variable, use:

```
WH ...
AL TEMP <-ACC;
<subscripted variable> <-TEMP;
WH ...
```

where TEMP is a simple variable having the same type as the subscripted variable.

Note that the construction:

```
...
AL <subscripted variable> <-ACC;
...
```

will not work since ALGOL uses the accumulator to evaluate the address of the subscripted variable.
3.2.3. FORMAL PARAMETERS CALLED BY NAME

Within a procedure, formal parameters called by name may not be accessed directly. Again, the easiest method of referencing these is via the "ACC" symbol. To load the accumulator with the value of a formal parameter, use:

```
WH ... AL ACC <-<formal parameter>; WH ...
```

To store the accumulator in a formal parameter, use:

```
WH ... AL TEMP <- ACC; <formal parameter> <- TEMP; WH ...
```

where TEMP and the formal parameter have the same type.

As with subscripted variables, the construction:

```
WH ... AL <formal parameter> <- ACC; WH ...
```

will not work.
3.3. CROSS-BLOCK TRANSFERS

Since intervening ALGOL cards have no effect on the scope of definition of WHAT symbols, WHAT is entirely independent of ALGOL block structure. (However, WHAT code may only reference those ALGOL symbols defined in the block containing the WHAT code.) As a result, a programmer may have tables and machine-code subroutines which may be accessed by any part of his WHAT/ALGOL program.

Great care must be exercised by the programmer who uses cross-block transfers via WHAT code. Two rules must be strictly adhered to:

(1) When transferring control to a subroutine in any block different from the current block, the subroutine may only access those ALGOL symbols which are defined identically in both blocks.

(2) The order in which begin's and end's are encountered must not be altered by the addition of WHAT coding.

Cross-block transfers are completely contrary to the philosophy of ALGOL and have implications which are beyond the scope of this manual.

3.4. INDEX ALLOCATION

Whenever an ALGOL index variable loses definition due to a block exit, the index register to which it was assigned is also released for later index variables. This feature may be utilized in a large WHAT program as an aid to the programmer in assigning/releasing his index registers. No more than 28 index variables may be defined at any time.
3.5. STATEMENT TERMINATION

When the ALGOL compiler is prepared to accept a statement and encounters WHAT code, the "expected" statement is not terminated until a ";" or other statement terminator is encountered in ALGOL. This is most likely to create difficulty when WHAT code occurs in the scope of some ALGOL construction.

Examples:

```
AL for ... do
WH ...
  ...
WH ...
AL <statement>;
```

```
AL if ... then
WH ...
  ...
WH ...
AL else <statement>;
```

```
AL begin
WH ...
  ...
WH ...
AL end
```

---

This is treated as a single statement.
A sudo (pseudo-instruction) is an instruction to WHAT rather than a G-20 command to be assembled for later execution. The mnemonic name of the sudo is punched in the operation field of the source program card. For all sudos the following holds:

1. The location field is first treated as described in Section 2.2 for machine commands.
2. The mode field is checked for error but is otherwise ignored.
3. The action of the particular sudo takes place.

A sudo may be listable or non-listable. For a listable sudo the parameter set may be repeated, separated by commas, as many times as desired in the space provided on the card, up to the RIGHT MARGIN. For a non-listable sudo only one parameter set is allowed in the address field. The effect of a listable sudo is the same as if the sudo were repeated on successive lines with one parameter set per line; the parameter sets are processed in the left-to-right order.

The remainder of Chapter consists of an alphabetical listing of the sudos, with an explanation and examples of the use of each. The format used in explaining the sudos is as follows:

XXX PARAMETER SYNTAX

LISTABLE

"EXECUTE EXTRA EXEC"

The first line gives the three letter sudo name and the type and format of the parameter set(s). The second line states whether the sudo is listable or non-listable (for sudos for which the concept is meaningful). The third line contains a summary of the action of the sudo. Note that the above sudo is only a hypothetical example.
ALF (BLANK) (CHARACTER STRING) (DIGIT) (CHARACTER STRING)
NON-LISTABLE

THE EFFECT IS TO LOAD THE G-20 INTERNAL REPRESENTATION OF THE STRING OF CHARACTERS INTO SUCCESSIVE MACHINE LOCATIONS, 4 CHARACTERS PER WORD. THE DIGIT GIVES THE NUMBER OF WORDS TO BE LOADED, WITH A BLANK BEING TREATED AS 1, AND 0 BEING TREATED AS 10. THE BLANK OR DIGIT MUST APPEAR IN THE FIRST POSITION OF THE ADDRESS FIELD, COLUMN 24. THE STRING TO BE LOADED EXTENDS FROM COLUMN 25 TO COLUMN (25+4K), WHERE K IS THE NUMBER OF WORDS SPECIFIED.

EXAMPLES:

W1  ALF  ERROR  NUMBER  ONE

THIS LINE WILL CAUSE THE LOADING OF:
ERROR INTO W1
R NU INTO W1+1
MBER INTO W1+2
ONE INTO W1+3

AND IS EQUIVALENT TO THE FOUR LINES:
W1  ALF  1ERROR
ALF  JER  NU
ALF  BMBER
ALF  1 ONE
The function is to check whether or not labels which have been used are defined. The symbol must be a label, if its subscript is zero or blank, then the subscript is considered to be the maximum allowed subscript. The labels from (class name)0 to (class name)subscript are then checked to see if all those which have been used are defined. In case an undefined label is encountered, an error print out takes place with the following form:

UNDEF T5 26347

This means that the label T5 is undefined, and that it has last been used in location 26347.

The checking will continue until the list of parameters has been exhausted.

Examples:

LBL 05;
LBL W10;
LBL R90;
{ PROGRAM }
CHK 0.5R;

Do to D5, W0 to W5, and R0 to R90 are checked.
**COM** (IMMATERIAL)

'COMMENT'

The location field is treated as usual and the mode field will get an error message if it is illegal, otherwise the line is ignored.

Examples:

```
LBL L1
COM THIS IS A COMMENT
L1 COM GEE... ANOTHER COMMENT
```

These lines will be printed. Two L's will be declared as labels and L1 will be defined as the current value of 'A'. However, no code will be compiled.

**CPY** (EXPRESSION), (EXPRESSION)

'CPY' NON-LISTABLE

'COPY'

Let the value of the first and the second expressions be N1 and N2, respectively. Then the next N1 words will be filled by copying from the last N2 words assembled. That is, the words in A-N2, A-N2+1, ..., A-1 will be copied repeatedly until N1 have been copied. N1 need not be a multiple of N2; N1 must not equal zero. After 'CPY' has been executed, the location counter 'A' has been increased by N1.

Warning: If the last N2 words contain any undefined labels, these will not later be defined in the copies.

Examples:

```
W8 LWD 737
LWD W53;
CPY 500.2
```

(W8) and (W8+1) will be copied into the next 500 locations.

```
E1 LWD 0;
CPY 499.1;
```

The effect is to store zero into 500 locations starting at E1.
DEF *<SYMBOL>*=<EXPRESSION> LISTABLE

*DEFINE*


<EXPRESSION> - <SUBSCRIPT>

IN THE USUAL CASE, THE SUBSCRIPT EQUALS ZERO.

EXAMPLES:

LBL B30
DEF B0=/22750

THIRTY-ONE B'S ARE DESIGNATED AS LABELS, AND B0 IS GIVEN THE VALUE /22750. B1, B2,..., B30 ARE STILL UNEDEFINED.

DEF C10=/7020;

CO IS GIVEN THE VALUE /7006, AND ALL C'S ARE DEFINED.
DMP \langle EXPRESSION\rangle, \langle EXPRESSION\rangle

LISTABLE
PRINTING AFTER EXECUTION

*DUMP*

THE EFFECT IS TO GIVE AN ASSEMBLY-TIME OCTAL DUMP ON THE PRINTER OF THE LOCATIONS FROM THE VALUE OF THE FIRST EXPRESSION UP TO AND INCLUDING THE VALUE OF THE SECOND EXPRESSION.

WARNING: THERE IS NO CHECK THAT THE VALUES ARE PROPER MACHINE LOCATIONS.

EXAMPLES:

DMP /21000./22000

AN OCTAL DUMP WILL BE GIVEN OF /1001 WORDS FROM LOCATION /21000 UP TO AND INCLUDING THE LOCATION /22000.

DMP A-100.A-11;

AN OCTAL DUMP OF THE LAST 100 LOCATIONS WILL BE GIVEN.

ENT \langle IMMATERIAL\rangle

*ENTRY*

THE EFFECT IS TO ASSEMBLE AN ALL ZERO WORD. THIS SUCCEEDS MAY BE USED FOR ENTRY INTO A SUBROUTINE. A LABEL APPEARING IN THE LOCATION FIELD WILL BE DEFINED AS USUAL.

EXAMPLES:

PI ENT SUBROUTINE

THIS DESIGNATES THE ENTRY INTO A SUBROUTINE THAT IS REFERRED TO BY THE LABEL PI. ZERO IS LOADED INTO THE LOCATION PI.
FPC < TERM >

FULL PRECISION CONSTANT

THE FUNCTION IS TO LOAD THE OCTAL REPRESENTATION OF THE DECIMAL NUMBER INTO THE NEXT TWO LOCATIONS.

WARNING: THE ABSOLUTE VALUE OF THE NUMBER MUST BE LESS THAN 3.450873173369*69 AND THE EXPONENT LESS THAN 70, OR AN EXPONENT OVERFLOW WILL OCCUR AT ASSEMBLY TIME.

EXAMPLES:

W10  FPC  10.4.000159*16
W11  FPC  -2*5.3.44563*5


HPC < TERM >

HALF PRECISION CONSTANT

THE FUNCTION IS TO LOAD THE OCTAL REPRESENTATION OF THE DECIMAL NUMBER INTO THE NEXT LOCATION. THE MANTISSA OF THE NUMBER IS ROUNDED TO SEVEN (OCTAL) DIGITS BEFORE STORING.

EXAMPLES:

W12  HPC  0,1,2,3;
      HPC  -4.15*6;

0, 1, 2, 3, AND -4.15*10+-6 WILL BE LOADED INTO FIVE CONSECUTIVE LOCATIONS STARTING AT W12.

WARNING: THE ABSOLUTE VALUE OF THE NUMBER MUST BE LESS THAN 3.450873173369*69 AND THE EXPONENT LESS THAN 70, OR AN EXPONENT OVERFLOW WILL OCCUR AT ASSEMBLY TIME.
LBL (WHAT SYMBOL) LISTABLE

"LABEL"

THE CLASS IS DECLARED TO BE A LABEL CLASS. IF THE CLASS NAME HAS NOT PREVIOUSLY APPEARED IN A "LBL" SYMBOL, THEN THE SUBSCRIPT IS THE MAXIMUM SUBSCRIPT WHICH MAY BE USED FOR THAT LABEL.

IF THE CLASS NAME HAS PREVIOUSLY APPEARED IN AN "LBL" SYMBOL, THE FOLLOWING ACTIONS TAKE PLACE:

FIRST, THE OPERATION OF A "CHK" SYMBOL IS DONE ON THE SYMBOL. THEN THE LABELS FROM <LETTER>0 TO <LETTER> <SUBSCRIPT> ARE CLEARED TO USE AGAIN, WHILE ANY LABELS GREATER THAN THE SUBSCRIPT APPEARING IN <LETTER> <SUBSCRIPT> ARE LEFT UNTOUCHED.

IF THE SUBSCRIPT IS ABSENT, THE MAXIMUM SUBSCRIPT FOR THAT CLASS IS ASSUMED.

IN CASE "CHK" FINDS ONE OR MORE UNDEFINED LABELS AN ERROR MESSAGE WILL BE PRINTED (SEE "CHK") AND THE VALUE OF THE LABEL WILL BE CLEARED SO THAT IT MAY BE REDEFINED.

EXAMPLES:

LBL D10

DO THROUGH D10 WILL BE PERMITTED FOR USE AS LABELS.

( PROGRAM )
LBL D7
( PROGRAM )

THE LABELS DO THROUGH D7 WILL BE CLEARED FOR REDEFINITION AS NEW LABELS, WITH AN ALARM MESSAGE PRINTED IF ANY ARE UNDEFINED.
LIN <EXPRESSION>  NON-LISTABLE
CARD IMAGE NOT PRINTED
'LINES'
If N is the value of the expression, N blank lines are printed, if printing is on.
If N = 0 or the address field is blank, 1 _LINE UPSPACE will occur. (The effect is similar to '*SY LINE*'.)

EXAMPLES:
CLA P9;
LIN 2;
EXL K21;

Above are the cards as they were punched. Below is the compilation of the cards.
CLA P9;

EXL K21;

Notice that 2 lines were skipped and the 'LIN' SUDO was not printed.

LWD <EXPRESSION>  NON-LISTABLE
'LOGIC WORD'
The effect is to load the value of the expression into the next machine location as a _LOGIC word
(i.e. with an '*'STL' command). Any punching in the mode field will be checked for error but will otherwise be ignored.
No checks are made to see if the values of the expressions are within the limit of the fields.

EXAMPLES:
LBL E2;
E0 LWD /350 + $4;
E1 LWD /7777+$1;
E2 LWD /7777777777;

Note: Do not assemble any LOGIC word with bit 30 = 1. This will confuse the ALGOL relocator. If this bit is needed, it must be added at run-time. (See writeup of -1, -2, -3, ALGOL MANUAL 2. 0.14.)
NAM (STRING) NON-LISTABLE

'NAME'
The effect is to pack the six bit representation of the 5 characters in columns 24 to 28 into the rightmost 30 bits of the next machine location. Any punching in the mode field will be checked for error but will otherwise be ignored.

EXAMPLES:
NAM PN3.$
The 6-bit representations of the characters P, N, 3, and $ will be loaded into the next machine location. This is the same as
LWD /20 16 43 53 65;

PAG (IMMATERIAL) PRINTING AFTER EXECUTION

'PAGE'
If printing is turned on, the paper in the printer will be moved to the next page.

PRT (SYMBOL) LISTABLE PRINTING BEFORE EXECUTION

'PRINT'
The function is similar to 'CHK', but in addition, if the printing is on, the values of all used labels will be listed on the printer.

EXAMPLES:
PRT W, P, D, Q10;
All the used labels of the symbols W, P, D and Q0 to Q10 and the locations to which they have been assigned are listed on the printer.
**WRD**

*SIGNED EXPRESSION*

"**WORD**"

The effect is to store the value of the expression into the core location specified by the location counter A. If the value of the expression is negative, **WRD** will store it into memory as an integer, provided that it is <2^21 in value. (i.e., using an 'STI' command); if positive, it will be stored as a logic word with an 'STL' command.

**Examples:**

WB WRD /735+8

WB will be loaded with the negative integer /725

W10 WRD /777777777

W10 will be loaded with the logic word /7777777777.

Note: As with 'LWD', bit 31 may not be used.

**REL**

*SYMBOL*

"**RELEASE**"

The function is to release labels; i.e., to clear the definition of a letter as a label so that it can be used thereafter as a region (or a new label).

First 'CHK' is performed. If no undefined label is encountered, the letter is then marked as unused. Under certain circumstances the space used for the label table will also be released. This will occur if the letter being released is the last letter declared as a label, or if all letters declared since have been released and their space reclaimed.

If an undefined label is encountered by 'CHK', an error message will be printed, as with 'CHK', and the 'RELEASE' will be performed anyway.

**Examples:**

LBL R10
(PROGRAM)
REL R
LBL R11

The set of labels R0 through R10 is released and then a new set of labels R0 through R11 is defined.
CHAPTER 5 - ERROR MESSAGES

5.1 ERRORS DETECTED DURING COMPILATION

Any error detected by 'WHAT' during the processing of a line will cause a print out of the line of code preceded by an error message, as follows.

5.1.1 ERRORS IN G-20 INSTRUCTIONS

AD U Undefined construction in address field of G-20 instruction
IR U Undefined construction in index field of a G-20 instruction
LABEL Error in location field
MODE Error in the mode field of a G-20 instruction
OPER Error in operation field
902 Illegal use of | or END of CARD ) error message follows
900 Catch-all for several other errors ) error line

5.1.2 ERRORS IN SUDO INSTRUCTIONS

AD U Undefined construction where an expression is needed in the address field of a SUDO.
A U 'A' is not within bounds of user's memory. (Upon storing a word)
LBL A subscript on a label symbol is greater than ALONGED
TERM Undefined construction where a symbol is wanted in the address field of a SUDO.
WHAT A letter which has not been declared as a label appears in a symbol in the address field of a SUDO where a label symbol is required.
TBL Space in label table is exhausted

5.2 ERRORS DETECTED DURING RUNNING

All run-time errors occurring in 'WHAT' are handled exactly the same as in ALGOL. (See ALGOL-20 Manual, Chapter 63.)
APPENDIX A

G-20 ALPHABET

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>INTERNAL CARD CODE</th>
<th>SYMBOL</th>
<th>INTERNAL CARD CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPACE</td>
<td>00</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>A</td>
<td>01</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>02</td>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>03</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>04</td>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>E</td>
<td>05</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>F</td>
<td>06</td>
<td>6</td>
<td>46</td>
</tr>
<tr>
<td>G</td>
<td>07</td>
<td>7</td>
<td>47</td>
</tr>
<tr>
<td>H</td>
<td>10</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>I</td>
<td>11</td>
<td>9</td>
<td>51</td>
</tr>
<tr>
<td>J</td>
<td>12</td>
<td>#</td>
<td>52</td>
</tr>
<tr>
<td>K</td>
<td>13</td>
<td>*</td>
<td>53</td>
</tr>
<tr>
<td>L</td>
<td>14</td>
<td>+</td>
<td>54</td>
</tr>
<tr>
<td>M</td>
<td>15</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>N</td>
<td>16</td>
<td>-</td>
<td>56</td>
</tr>
<tr>
<td>O</td>
<td>17</td>
<td>-</td>
<td>57</td>
</tr>
<tr>
<td>P</td>
<td>20</td>
<td>-</td>
<td>50</td>
</tr>
<tr>
<td>Q</td>
<td>21</td>
<td>-</td>
<td>51</td>
</tr>
<tr>
<td>R</td>
<td>22</td>
<td>#</td>
<td>52</td>
</tr>
<tr>
<td>S</td>
<td>23</td>
<td>+</td>
<td>53</td>
</tr>
<tr>
<td>T</td>
<td>24</td>
<td>+</td>
<td>54</td>
</tr>
<tr>
<td>U</td>
<td>25</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>V</td>
<td>26</td>
<td>-</td>
<td>56</td>
</tr>
<tr>
<td>W</td>
<td>27</td>
<td>-</td>
<td>57</td>
</tr>
<tr>
<td>X</td>
<td>30</td>
<td>-</td>
<td>50</td>
</tr>
<tr>
<td>Y</td>
<td>31</td>
<td>-</td>
<td>51</td>
</tr>
<tr>
<td>Z</td>
<td>32</td>
<td>-</td>
<td>52</td>
</tr>
<tr>
<td>I</td>
<td>33</td>
<td>-</td>
<td>53</td>
</tr>
<tr>
<td>@</td>
<td>34</td>
<td>-</td>
<td>54</td>
</tr>
<tr>
<td>*</td>
<td>35</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>#</td>
<td>36</td>
<td>-</td>
<td>56</td>
</tr>
<tr>
<td>+</td>
<td>37</td>
<td>-</td>
<td>57</td>
</tr>
</tbody>
</table>

THE INTERNAL REPRESENTATIONS ABOVE ARE OCTAL INTEGERS.

**NOTE 1:** MUST BE PUNCHED USING THE MULTIPLE PUNCH BUTTON

**NOTE 2:** THE KEY MARKED QUOTE ON THE KEYPUNCH ACTUALLY PUNCHES THE SEMI-COLON - THE 4-8 COMBINATION. THE G-20 CHARACTER QUOTE MUST BE MULTI-PUNCHED AS 5-8.
APPENDIX B
G-20 'WHAT' OPCodes

ADDRESS PREPARATION
DCA 000 X * (OA)
DCS 020 -X * (OA)
OAD 040 (ACC) + X * (OA)
OSU 060 (ACC) - X + (OA)
OSN 120 -(ACC) + X + (OA)
OAN 100 -(ACC) - X • (OA)
OAA 140 |(ACC) I X| ^ (OA)
OSA 160 |(ACC) - X| •> (OA)

ADD AND SUBTRACT
CLA 005 X • (ACC)
CLS 025 - X * (ACC)
ADD 045 (ACC) + X • (ACC)
SUB 065 (ACC) - X • (ACC)
ADN 105 - (ACC) - X * (ACC)
SIN 125 -(ACC) + X • (ACC)
ADA 145 |(ACC) + X| • (ACC)
SUA 165 |(ACC) - X| • (ACC)

ARITHMETIC TESTS *
FOM 021 X < 0
FOP 001 X > 0
FLO 121 (ACC) < 0
FGO 061 (ACC) > 0
FUO 161 (ACC) # X
FSM 101 (ACC) + X < 0
FSN 141 (ACC) + X # 0
FSP 041 (ACC) + X > 0

MULTIPLY AND DIVIDE
MPY 077 (ACC) * X V (ACC)
RDV 057 X / (ACC) * (ACC)
DIV 053 (ACC) / X -> (ACC)

LOGIC OPERATIONS
CAL 015 X * (ACC)
CCL 035 ^X * (ACC)
ADL 055 (ACC) + X + (ACC)
SUL 075 (ACC) - X + (ACC)
EXL 115 (ACC) ^ X + (ACC)
ECL 135 (ACC) ^-X + (ACC)
UNL 155 (ACC) v X + (ACC)
UCL 175 (ACC) v^X + (ACC)

LOGIC TESTS *
IDZ 011 X = 0
ICZ 031 ^X = 0
ISN 051 (ACC) + X # 0
IUW 071 (ACC) - X # 0
IEZ 111 (ACC) ^X = 0
IEC 131 (ACC) ^-X = 0
IUC 171 (ACC) v^X = 0
IUZ 151 (ACC).v X = 0

STORE
STL 173 (ACC) • X
STD 153 (ACC) • X + 1
STS 113 (ACC) • X
STI 133 (ACC) • X
STZ 073 0 * X

INDEX REGISTER CODES
LXP 012 X * (I)
LXM 032 -X * (I)
ADX 002 (I) + X * (I)
SUX 022 (I) - X * (I)
XPT 016 X + (I) • (TEST(I) # 0)*
XMT 036 -X + (I) • (TEST(I) # 0)*
AXT 006 (I) + X • (I) • (TEST(I) # 0)*
SXT 026 (I) - X • (I) • (TEST(I) # 0)*

TRANSFER OF CONTROL
TRA 017 X • (NC)
SKP 137 (NC) + X • (NC)
TRM 177 (NC) • X; X+1 • (NC)

SPECIAL
REP 013 REPEAT
XEQ 010 EXECUTE X

X + 1

NOTE: (Z) = CONTENTS OF Z
*FOR ALL TESTS, DO NEXT IF CONDITION INDICATED IS TRUE,
SKIP IF FALSE.

'MIX' ASSEMBLES ALL COMMANDS
IN MODE 2 EXCEPT THE FOLLOWING WHICH ARE ASSEMBLED IN MODE 0:

MODE INTERPRETATION
0 X=(I) + (OA) + ADDRESS
1 X=(I) + (OA) + (ADDRESS)
2 X=(I) + (OA) + ADDRESS
3 X=(I) + (OA) + (ADDRESS)

NOTE: (Z) = CONTENTS OF Z

*FOR ALL TESTS, DO NEXT IF CONDITION INDICATED IS TRUE,
SKIP IF FALSE.
COMMANDS IN NUMERICAL ORDER

000 OCA OPERAND CLEAR ADD
001 FDP IF OPERAND PLUS
002 ADX ADD TO INDEX
005 CLA CLEAR ADD
006 AXT ADD TO INDEX AND TEST
010 XEO EXECUTE OPERAND
011 IOZ IF OPERAND ZERO
012 LXP LOAD INDEX PLUS
013 REP REPEAT
015 CAL CLEAR ADD LOGIC
016 XPT LOAD INDEX PLUS AND TEST
017 TRA TRANSFER
020 DCs OPERAND CLEAR SUBTRACT
021 FOM IF OPERAND MINUS
022 SUX SUBTRACT FROM INDEX
025 CLS CLEAR SUBTRACT
026 SXT SUBTRACT FROM INDEX AND TEST
031 ICZ IF COMPLEMENT ZERO
032 LXM LOAD INDEX MINUS
035 CCL CLEAR ADD COMPLEMENT LOGIC
036 XMT LOAD INDEX MINUS AND TEST
040 OAD OPERAND ADD
041 FSP IF SUM PLUS
045 ADD ADD
051 ISN IF SUM NON-ZERO
053 DIV DIVIDE
055 AD ADD LOGIC
057 Rov REVERSE DIVIDE
060 DSU OPERAND SUBTRACT
061 FGO IF GREATER THAN OPERAND
065 SUB SUBTRACT
071 IUO IF UNEQUAL OPERAND
073 STZ STORE ZERO
075 SU SUBTRACT LOGIC
077 MPY MULtiPLY
100 DSN OPERAND ADD AND NEGATE
101 FSM IF SUM MUNUS
105 ADD ADD AND NEGATE
111 PEZ IF EXTRACT ZERO
113 STS STORE SINGLE
115 EXL EXTRACT LOGIC
120 DSM OPERAND SUBTRACT AND NEGATE
121 FLO IF LESS THAN OPERAND
125 SUM SUBTRACT AND NEGATE
131 IEC IF EXTRACT COMPLEMENT ZERO
133 STI STORE INTEGER
135 ECL EXTRACT COMPLEMENT LOGIC
137 SKP SKIP
140 OAA OPERAND ADD AND ABSOLUTE
141 FSM IF SUM NON-ZERO
145 ADA ADD AND ABSOLUTE
151 IUZ IF UNION ZERO
153 STD STORE DOUBLE
155 ULI UNITE LOGIC
160 DSA OPERAND SUBTRACT AND ABSOLUTE
161 FUD IF UNEQUAL OPERAND
165 SUA SUBTRACT AND ABSOLUTE
171 IUC IF UNION COMPLEMENT ZERO
173 STL STORE LOGIC
175 UCL UNITE COMPLEMENT LOGIC
177 TRM TRANSFER AND MARK

- (ACC) - X + (OA)
TEST (ACC) + X < 0
- (ACC) - X + (ACC)
TEST (ACC) ^ X = 0
(ACC) + X
(ACC) ^ X + (ACC)
- (ACC) + X + (OA)
TEST (ACC) < X
- (ACC) + X + (ACC)
TEST (ACC) ^ - X = 0
(ACC) + X
(ACC) ^ ~X + (ACC)
(NC) + X * (NC)
([ACC] + X] + (OA)
TEST (ACC) + X # 0
([ACC] + X] + (ACC)
TEST (ACC) v X = 0
(ACC) + X, X + 1
(ACC) v X + (ACC)
([ACC] - X] + (OA)
TEST (ACC) # X
([ACC] - X] + (ACC)
TEST (ACC) v - X = 0
(ACC) + X
(ACC) v ~X + (ACC)
(NC) + (X) ; X + 1 * (NC)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0A</td>
<td>ADD AND ABSOLUTE</td>
</tr>
<tr>
<td>045</td>
<td>ADD ADD</td>
</tr>
<tr>
<td>055</td>
<td>ADD LOGIC</td>
</tr>
<tr>
<td>105</td>
<td>ADD AND NEGATE</td>
</tr>
<tr>
<td>002</td>
<td>ADD TO INDEX</td>
</tr>
<tr>
<td>006</td>
<td>ADD TO INDEX AND TEST</td>
</tr>
<tr>
<td>015</td>
<td>CLEAR ADD LOGIC</td>
</tr>
<tr>
<td>005</td>
<td>CLEAR ADD</td>
</tr>
<tr>
<td>035</td>
<td>CLEAR ADD COMPLEMENT LOGIC</td>
</tr>
<tr>
<td>025</td>
<td>CLEAR SUBTRACT</td>
</tr>
<tr>
<td>053</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>135</td>
<td>EXTRACT COMPLEMENT LOGIC</td>
</tr>
<tr>
<td>115</td>
<td>EXTRACT LOGIC</td>
</tr>
<tr>
<td>051</td>
<td>FG0 IF GREATER THAN OPERAND</td>
</tr>
<tr>
<td>121</td>
<td>FJ0 IF LESS THAN OPERAND</td>
</tr>
<tr>
<td>021</td>
<td>FDM IF OPERAND MINUS</td>
</tr>
<tr>
<td>001</td>
<td>FDP IF OPERAND PLUS</td>
</tr>
<tr>
<td>101</td>
<td>FSM IF SUM MINUS</td>
</tr>
<tr>
<td>141</td>
<td>FSN IF SUM NON-ZERO</td>
</tr>
<tr>
<td>041</td>
<td>FSP IF SUM PLUS</td>
</tr>
<tr>
<td>161</td>
<td>FUD IF UNEQUAL OPERAND</td>
</tr>
<tr>
<td>031</td>
<td>ICZ IF COMPLEMENT ZERO</td>
</tr>
<tr>
<td>131</td>
<td>IEC IF EXTRACT COMPLEMENT ZERO</td>
</tr>
<tr>
<td>111</td>
<td>IEZ IF EXTRACT ZERO</td>
</tr>
<tr>
<td>011</td>
<td>I0Z IF OPERAND ZERO</td>
</tr>
<tr>
<td>051</td>
<td>ISV IF SUM NON-ZERO</td>
</tr>
<tr>
<td>171</td>
<td>IUC IF UNION COMPLEMENT ZERO</td>
</tr>
<tr>
<td>071</td>
<td>IUD IF UNEQUAL OPERAND</td>
</tr>
<tr>
<td>151</td>
<td>IUZ IF UNION ZERO</td>
</tr>
<tr>
<td>032</td>
<td>LXU LOAD INDEX MINUS</td>
</tr>
<tr>
<td>012</td>
<td>LXP LOAD INDEX PLUS</td>
</tr>
<tr>
<td>077</td>
<td>MPY MULTIPLY</td>
</tr>
<tr>
<td>140</td>
<td>OAA OPERAND ADD AND ABSOLUTE</td>
</tr>
<tr>
<td>040</td>
<td>OAD OPERAND ADD</td>
</tr>
<tr>
<td>100</td>
<td>DAN OPERAND ADD AND NEGATE</td>
</tr>
</tbody>
</table>

- \( (\text{ACC}) + x \) if \( x \) is positive
- \( (\text{ACC}) - x \) if \( x \) is negative
- \( (\text{ACC}) \times x \) if \( x \) is positive
- \( (\text{ACC}) / x \) if \( x \) is positive
- \( (\text{ACC}) \div x \) if \( x \) is positive
000 OCA OPERAND CLEAR ADD
020 OCS OPERAND CLEAR SUBTRACT
160 OSA OPERAND SUBTRACT AND ABSOLUTE
120 OSN OPERAND SUBTRACT AND NEGATE
060 OSU OPERAND SUBTRACT
057 ROV REVERSE DIVIDE
013 REP REPEAT
137 SKP SKIP
153 STD STORE DOUBLE
133 STI STORE INTEGER
173 STL STORE LOGIC
113 STS STORE SINGLE
073 STZ STORE ZERO
065 SUA SUBTRACT AND ABSOLUTE
165 SUB SUBTRACT
075 SUL SUBTRACT LOGIC
125 SUV SUBTRACT AND NEGATE
022 SUX SUBTRACT FROM INDEX
026 SUX SUBTRACT FROM INDEX AND TEST
017 TRA TRANSFER
177 TRM TRANSFER AND MARK
175 UCL UNITE COMPLEMENT LOGIC
155 UNL UNITE LOGIC
010 XEQ EXECUTE OPERAND
016 XPT LOAD INDEX PLUS AND TEST
036 XMT LOAD INDEX MINUS AND TEST

X + (OA)
-(ACC) + X + (OA)
| (ACC) - X| + (OA)
-(ACC) + X + (OA)
(ACC) - X + (OA)
(ACC) + (ACC)
(ACC) * X
(ACC) * X
0 * X
((ACC) - X| + (ACC)
(ACC) - X + (ACC)
(ACC) - X + (ACC)
(ACC) + X + (ACC)
(ACC) - X + (ACC)
(ACC) - X + (ACC)
(ACC) + X + (ACC)
X + (NC)
(NC) + (X) + X + 1 + (NC)
(ACC) + (ACC)
(ACC) * X + (ACC)
EXECUTE X AS COMMAND
X + (I) (TEST(I)#0)
-(X) (TEST(I)#0)
### APPENDIX C

**SODS IN "WHAT"**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALF</td>
<td>ALPHANUMERIC INFORMATION</td>
</tr>
<tr>
<td>CHK</td>
<td>CHECK</td>
</tr>
<tr>
<td>COM</td>
<td>COMMENT</td>
</tr>
<tr>
<td>CPY</td>
<td>COPY</td>
</tr>
<tr>
<td>DEF</td>
<td>DEFINE</td>
</tr>
<tr>
<td>DMP</td>
<td>DUMP</td>
</tr>
<tr>
<td>ENT</td>
<td>ENTRY</td>
</tr>
<tr>
<td>FPC</td>
<td>FULL PRECISION CONSTANT</td>
</tr>
<tr>
<td>HPC</td>
<td>HALF PRECISION CONSTANT</td>
</tr>
<tr>
<td>LBL</td>
<td>LABEL</td>
</tr>
<tr>
<td>LIN</td>
<td>LINE</td>
</tr>
<tr>
<td>LWD</td>
<td>LOGIC WORD</td>
</tr>
<tr>
<td>NAM</td>
<td>NAME</td>
</tr>
<tr>
<td>PAG</td>
<td>PAGE</td>
</tr>
<tr>
<td>PRT</td>
<td>PRINT</td>
</tr>
<tr>
<td>REL</td>
<td>RELEASE</td>
</tr>
<tr>
<td>WRD</td>
<td>WORD</td>
</tr>
</tbody>
</table>
## APPENDIX D

### G-20 SHIFT MULTIPLIERS

<table>
<thead>
<tr>
<th>LEFT SHIFT</th>
<th>NUMBER</th>
<th>RIGHT SHIFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>000 00 0001</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>101 00 0004</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>101 00 0002</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>101 00 0001</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>102 00 0004</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>102 00 0002</td>
</tr>
<tr>
<td>100</td>
<td>6</td>
<td>102 00 0001</td>
</tr>
<tr>
<td>200</td>
<td>7</td>
<td>103 00 0004</td>
</tr>
<tr>
<td>400</td>
<td>8</td>
<td>103 00 0002</td>
</tr>
<tr>
<td>1000</td>
<td>9</td>
<td>103 00 0001</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>104 00 0004</td>
</tr>
<tr>
<td>4000</td>
<td>11</td>
<td>104 00 0002</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>104 00 0001</td>
</tr>
<tr>
<td>20000</td>
<td>13</td>
<td>105 00 0004</td>
</tr>
<tr>
<td>40000</td>
<td>14</td>
<td>105 00 0002</td>
</tr>
<tr>
<td>05 00 00001</td>
<td>15</td>
<td>105 00 0001</td>
</tr>
<tr>
<td>05 00 00002</td>
<td>16</td>
<td>106 00 0004</td>
</tr>
<tr>
<td>05 00 00004</td>
<td>17</td>
<td>106 00 0002</td>
</tr>
<tr>
<td>06 00 00001</td>
<td>18</td>
<td>106 00 0001</td>
</tr>
<tr>
<td>06 00 00002</td>
<td>19</td>
<td>107 00 0004</td>
</tr>
<tr>
<td>06 00 00004</td>
<td>20</td>
<td>107 00 0002</td>
</tr>
<tr>
<td>07 00 00001</td>
<td>21</td>
<td>107 00 0001</td>
</tr>
<tr>
<td>07 00 00002</td>
<td>22</td>
<td>110 00 0004</td>
</tr>
<tr>
<td>07 00 00004</td>
<td>23</td>
<td>110 00 0002</td>
</tr>
<tr>
<td>10 00 00001</td>
<td>24</td>
<td>110 00 0001</td>
</tr>
<tr>
<td>10 00 00002</td>
<td>25</td>
<td>111 00 0004</td>
</tr>
<tr>
<td>10 00 00004</td>
<td>26</td>
<td>111 00 0002</td>
</tr>
<tr>
<td>11 00 00001</td>
<td>27</td>
<td>111 00 0001</td>
</tr>
<tr>
<td>11 00 00002</td>
<td>28</td>
<td>112 00 0004</td>
</tr>
<tr>
<td>11 00 00004</td>
<td>29</td>
<td>112 00 0002</td>
</tr>
<tr>
<td>12 00 00001</td>
<td>30</td>
<td>112 00 0001</td>
</tr>
<tr>
<td>12 00 00002</td>
<td>31</td>
<td>113 00 0004</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>OCTAL</td>
<td>OCTAL</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>36</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>62</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>74</td>
<td>50</td>
</tr>
<tr>
<td>70</td>
<td>106</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>132</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>144</td>
<td>200</td>
</tr>
<tr>
<td>200</td>
<td>310</td>
<td>300</td>
</tr>
<tr>
<td>300</td>
<td>454</td>
<td>400</td>
</tr>
<tr>
<td>400</td>
<td>620</td>
<td>500</td>
</tr>
<tr>
<td>500</td>
<td>764</td>
<td>600</td>
</tr>
<tr>
<td>600</td>
<td>1 130</td>
<td></td>
</tr>
<tr>
<td>700</td>
<td>1 274</td>
<td>1 000</td>
</tr>
<tr>
<td>800</td>
<td>1 440</td>
<td>2 000</td>
</tr>
<tr>
<td>900</td>
<td>1 604</td>
<td>3 000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 000</td>
</tr>
<tr>
<td>1 000</td>
<td>1 750</td>
<td>5 000</td>
</tr>
<tr>
<td>2 000</td>
<td>3 720</td>
<td>6 000</td>
</tr>
<tr>
<td>3 000</td>
<td>5 670</td>
<td>7 000</td>
</tr>
<tr>
<td>4 000</td>
<td>7 640</td>
<td></td>
</tr>
<tr>
<td>5 000</td>
<td>11 610</td>
<td>10 000</td>
</tr>
<tr>
<td>6 000</td>
<td>13 560</td>
<td>20 000</td>
</tr>
<tr>
<td>7 000</td>
<td>15 530</td>
<td>30 000</td>
</tr>
<tr>
<td>8 000</td>
<td>17 500</td>
<td>40 000</td>
</tr>
<tr>
<td>9 000</td>
<td>21 450</td>
<td>50 000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60 000</td>
</tr>
<tr>
<td>10 000</td>
<td>23 420</td>
<td>70 000</td>
</tr>
<tr>
<td>20 000</td>
<td>47 040</td>
<td></td>
</tr>
<tr>
<td>30 000</td>
<td>72 460</td>
<td></td>
</tr>
<tr>
<td>40 000</td>
<td>116 100</td>
<td></td>
</tr>
<tr>
<td>50 000</td>
<td>141 520</td>
<td></td>
</tr>
<tr>
<td>60 000</td>
<td>165 140</td>
<td></td>
</tr>
<tr>
<td>70 000</td>
<td>210 560</td>
<td></td>
</tr>
<tr>
<td>80 000</td>
<td>234 200</td>
<td></td>
</tr>
<tr>
<td>90 000</td>
<td>257 620</td>
<td></td>
</tr>
<tr>
<td>100 000</td>
<td>303 240</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX E

SAMPLE WHAT/ALGOL PROCEDURE

logic procedure NEXTCHAR (B,C);

boolean B;          || if B = T, initialize, else continue.
logic array C;       || C[0] = 1st word of text.

begin

comment

If B is true, return first character of text buffer. If B is false, return
next character;

logic L;             || temp storage.
own integer I;       || word pointer.
own index J;         || character pointer.

WH LBL M1            ; will need two labels.

AL if B then begin   || if B = true

I ← 0;              || reset word and
J ← 4;              || character

end                  || pointers.

ACC ← C[I];         || fetch appropriate word.

WH MPY MO, J         ; shift to right-justify.
EXL 0 /377           ; mask out garbage.
STL L                ; save for ALGOL store.

AL NEXTCHAR ← L;     || value of procedure = L.

WH SXT 0 1, J        ; step to next character.
TRA A2               ; test for shift to new word.
TRA M1 ; no shift, exit.
XPT 0 4, J ; shift req'd, reset
CLA I ; character pointer
ADD 0 1 ; and word pointer
STI I ; then
TRA M1 ; exit.

MO shift constants
LWD $24, $16, $8, $0;
M1 exit from procedure
AL end of NEXTCHAR