WHAT

JAMES A. MOORE

FIRST PRINTING JUNE 1965

THIS WORK WAS SUPPORTED BY THE
ADVANCED RESEARCH PROJECTS AGENCY OF THE
OFFICE OF THE SECRETARY OF DEFENSE:
CONTRACT SD-146

CARNEGIE INSTITUTE OF TECHNOLOGY

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

TABLE OF CONTENTS

(continued)

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 <u>only</u> within WHAT.

(2) ALGOL symbols.

These are declared and defined by ALGOL declarations and may be used in <u>either</u> 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 assemlby 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 2.346 for details on B.N.F.)

```
<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" sudos 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 "¬" 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", "PRT" and "CHK" sudos)

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     R0  =  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 "A0" 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 "A0" 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:

```
      L1            TRA    L1 + 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> ::= 8L<integer> |  8R<integer> |  8F<number> -- (See ALGOL Manual
                                                           <octal constant>,
                                                           p. 6e.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 similiar definitions for ALGOL.)

Examples:

```
418

/77 * $12

L1 -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 "└┐" or "|" symbol.

An expression defined by these rules may be used in the address or index fields of a line of WHAT code. The meaning of an expression is obtained by performing the indicated operations from left to right with no hierarchy and truncating to 32 bits after the entire expression has been evaluated. Thus, 2 + 3*4 = 20. An empty expression has the value zero.

Expressions are generally used to represent G-20 (or G-21) addresses, so their values will usually be positive integers less than 2↑15.

The term "$n", where n is an integer less than 32_{10}, has the value 2↑n; i.e. "$n" stands for a one in bit position n of a logic word.

The value of a floating octal constant (8F<number>) is determined by concatenating the <number> as an octal number and multiplying it by the appropriate power of 8, treating the number which follows the $_{10}$ as an octal integer. For example:

8F$_{10}$10 = 8↑8

8F11$_{10}$-5 = 9*8↑-5

The value of a left (right) justified octal constant (8L<integer>, 8R<integer>) is determined by prefixing (suffixing) to the <integer> enough zeros to give eleven octal digits. This number is then concatenated and stored as a 32-bit logic word. Since eleven octal digits require thirty-three bits for representation, the leftmost bit of the leftmost octal digit is lost. Thus, 8L4 = 0 and 8L7 = 8L3. "/<integer>" is equivalent to "8R<integer>".

The character-pairs 8L, 8R and 8F are treated by the translator as single entities and <u>must</u> 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 <u>only</u> 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_{10}, 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:

Contents	Cols.
1. Language	1 - 2
2. Location	4 - 12
3. Operation	15 - 17
4. Mode	20
5. Address, Index; Comments	24 - RIGHT MARGIN

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.)
                  1        2        2
|1      |4      |5      |0      |4
/|WH     |E4     |CLA    |0      |/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:

Tab	Column	Field
1	4	Label
2	15	Opcode
3	20	Mode
4	24	Address, Index; Comments
5	40	Comments (See Section 2.7, p. 20.)

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        PO, I ; ZERO A LOCATION
     SXT     0  1, I  ; DECREMENT AND TEST
     TRA        E2    ; LOOP
```

16.

```
          (case 3)

L2        TRA    A7                ; transfer around

          LWD    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 (<u>without</u> 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.

STI	STL	TRA	REP
STS	STZ	TRM	
STD			

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

defined, its value will be placed in the address when it is defined.

3. If it is an expression, the value of the expression will be entered as the address in the assembled instruction. It is a detectable error if any symbol in the expression has not been defined previously. See 5.1.1, p. 39.

The value of the expression in the address should be less than 2↑15, but no assembly error will result from a larger value.

2.6 INDEX FIELD (Columns 24 - RIGHT MARGIN)

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.

CHAPTER 3 - WHAT/ALGOL INTERACTION

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 ALGCL label are at different block levels, the "TRA" _must_ be replaced by an ALGOL "_go to_" statement.

EXAMPLES:

```
              ...                  )
      WH      TRA      ALPHA;      |
              ...                  |      Repeat:
      AL      begin real X;        }      This will
              ALPHA:               |      not work
              ...                  )

      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 boundries, 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;
             ...
```

<u>will</u> <u>not</u> <u>work</u> 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    ...
```

<u>will</u> <u>not</u> <u>work</u>.

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 <u>begin</u>'s and <u>end</u>'s are encountered must not be altered by the addition of WHAT coding.

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

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         ...    }
                     ...    }  scope of
          WH         ...    }  for-statement
          AL    <statement>; }
      - - - - - - - - - - - - - - - -
          AL    if   ...  then
          WH         ...    }
                     ...    }  scope of then
          WH         ...    }
          AL    else  <statement>;
      - - - - - - - - - - - - - - - -
          AL    begin
          WH         ...    }  this is treated
                     ...    }  as a single statement
          WH         ...    }
          AL    end
```

CHAPTER 4 - SUDO INSTRUCTIONS IN WHAT

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 thι particular sudo takes place.

A sudo may be <u>listable</u> ιr <u>non-listable</u>. 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
     'ALPHANUMERIC'
```

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       4ERROR NUMBER ONE
```

THIS LINE WILL CAUSE THE LOADING OF:

```
     ERRO     INTO    W1
     R NU     INTO    W1+1
     MBER     INTO    W1+2
      ONE     INTO    W1+3
```

AND IS EQUIVALENT TO THE FOUR LINES:

```
W1   ALF       1ERRO
     ALF       1R NU
     ALF       1MBER
     ALF       1 ONE
```

```
CHK        <WHAT SYMBOL>
                                          LISTABLE
```

'CHECK'

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 ENCOUNTED, AN ERROR PRINT OUT TAKES PLACE WITH THE FOLLOWING FORM:

```
UND       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       D5;
LBL       W10;
LBL       R90;
( PROGRAM )
CHK       D,W5,R;
```

D0 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>
'COPY'               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'

THE VALUE OF THE EXPRESSION WILL BE CALCULATED AND TAKEN MODULO 2+21, AND THE SYMBOL WILL BE GIVEN THIS VALUE. IF THE LETTER OF THE SYMBOL HAS BEEN DECLARED AS A LABEL, THE PARTICULAR LABEL GIVEN IS THEREBY DEFINED. IF THE LETTER IS NOT A LABEL, THE CORRESPONDING REGIONAL BASE IS DEFINED AS

<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 UNDEFINED.

```
DEF       C10=/7020;
```

C0 IS GIVEN THE VALUE /7006, AND ALL C'S ARE DEFINED.

```
DMP       <EXPRESSION>,<EXPRESSION>
                                 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-1;
```

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

```
ENT       <IMMATERIAL>
```

'ENTRY'

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

EXAMPLES:

```
P1    ENT       SUBROUTINE
```

THIS DESIGNATES THE ENTRY INTO A SUBROUTINE THAT IS REFERRED TO BY THE LABEL P1. ZERO IS LOADED INTO THE LOCATION P1.

FPC <TERM>

LISTABLE

'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 $3.450873173389_{10}69$ 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⏨5,3.44463⏨-5
```

W10 AND W10+1 WILL BE LOADED WITH 10, W10+2 AND W10+3 WILL BE LOADED WITH 4.000159*10+15, W11 AND W11+1 WILL BE LOADED WITH -2*10+5, AND W11+2 AND W11+3 WILL BE LOADED WITH 3.44463*10+-5, ALL IN STANDARD G-20 FULL PRECISION FORM. W10 AND W11 MUST BE LABELS, SINCE THEY ARE NOT IN ADJACENT LOCATIONS.

HPC <TERM>

LISTABLE

'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 $3.450873173389_{10}69$ 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' SUDO, THEN THE SUBSCRIPT IS THE MAXIMUM SUBSCRIPT WHICH MAY BE USED FOR THAT LABEL.

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

FIRST, THE OPERATION OF A 'CHK' SUDO 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
     'LINE'
          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 SIMILIAR 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 P. 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.
```

REL   <SYMBOL>

LISTABLE

'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.

WRD   <SIGNED EXPRESSION>

LISTABLE

'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:

```
    W8    WRD       -/735+8
```

W8 WILL BE LOADED WITH THE NEGATIVE INTEGER /725

```
    W10   WRD       /77777777777
```

W10 WILL BE LOADED WITH THE LOGIC WORD /77777777777.

NOTE: AS WITH 'LWD', BIT 31 MAY NOT BE USED.

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 PRECEEDED 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	
LABL	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 ALLOWED
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 6B.)

APPENDIX A

G-20 ALPHABET

SYMBOL	INTERNAL	CARD CODE	SYMBOL	INTERNAL	CARD CODE
SPACE	00	NO PUNCH	0	40	0
A	01	+ 1	1	41	1
B	02	+ 2	2	42	2
C	03	+ 3	3	43	3
D	04	+ 4	4	44	4
E	05	+ 5	5	45	5
F	06	+ 6	6	46	6
G	07	+ 7	7	47	7
H	10	+ 8	8	50	8
I	11	+ 9	9	51	9
J	12	- 1	$_{10}$	52	0 7 8 NOTE 1
K	13	- 2	.	53	+ 3 8
L	14	- 3	+	54	+
M	15	- 4	-	55	-
N	16	- 5	*	56	- 4 8
O	17	- 6	/	57	0 1
P	20	- 7	=	60	3 8
Q	21	- 8	∨	61	+ 7 8 NOTE 1
R	22	- 9	≠	62	- 2 8 NOTE 1
S	23	0 2	∧	63	+ 6 8 NOTE 1
T	24	0 3	<	64	- 5 8 NOTE 1
U	25	0 4	$	65	- 3 8
V	26	0 5	>	66	- 6 8 NOTE 1
W	27	0 6	;	67	4 8 NOTE 2
X	30	0 7	(	70	0 4 8
Y	31	0 8	[	71	0 5 8 NOTE 1
Z	32	0 9	]	72	0 6 8 NOTE 1
|	33	2 8 NOTE 1	)	73	+ 4 8
←	34	6 8 NOTE 1	↓	74	7 8 NOTE 1
→	35	- 7 8 NOTE 1	↑	75	+ 2 8 NOTE 1
¬	36	+ 5 8 NOTE 1	:	76	0 2 8 NOTE 1
,	37	0 3 8	'	77	5 8 NOTE 2

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
 OCA 000 X → (OA)
 OCS 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) + 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)
 SUN 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~~ X
 FGO 061 (ACC) > ~~0~~ X
 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 → (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 *
 IOZ 011 X = 0
 ICZ 031 ¬X = 0
 ISN 051 (ACC) + X ≠ 0
 IUO 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, 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
```

```
      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.



'WHAT' ASSEMBLES ALL COMMANDS
IN MODE 2 EXCEPT THE FOLLOWING
WHICH ARE ASSEMBLED IN MODE 0:
STI            TRA
STS            TRM
STD            REP
STL            STZ
```

COMMANDS IN NUMERICAL ORDER

000	OCA	OPERAND CLEAR ADD	X → (OA)	
001	FOP	IF OPERAND PLUS	TEST X > 0	
002	ADX	ADD TO INDEX	(I) + X → (I)	
005	CLA	CLEAR ADD	X → (ACC)	
006	AXT	ADD TO INDEX AND TEST	(I) + X → (I)	(TEST(I)≠0)
010	XEQ	EXECUTE OPERAND	EXECUTE X	
011	IOZ	IF OPERAND ZERO	TEST X = 0	
012	LXP	LOAD INDEX PLUS	X → (I)	
013	REP	REPEAT	REPEAT	
015	CAL	CLEAR ADD LOGIC	X → (ACC)	
016	XPT	LOAD INDEX PLUS AND TEST	X → (I)	(TEST(I)≠0)
017	TRA	TRANSFER	X → (NC)	
020	OCS	OPERAND CLEAR SUBTRACT	- X → (OA)	
021	FOM	IF OPERAND MINUS	TEST X < 0	
022	SUX	SUBTRACT FROM INDEX	(I) - X → (I)	
025	CLS	CLEAR SUBTRACT	- X → (ACC)	
026	SXT	SUBTRACT FROM INDEX AND TEST	(I) - X → (I)	(TEST(I)≠0)
031	ICZ	IF COMPLEMENT ZERO	TEST ¬X = 0	
032	LXM	LOAD INDEX MINUS	- X → (I)	
035	CCL	CLEAR ADD COMPLEMENT LOGIC	¬X → (ACC)	
036	XMT	LOAD INDEX MINUS AND TEST	- X → (I)	(TEST(I)≠0)
040	OAD	OPERAND ADD	(ACC) + X → (OA)	
041	FSP	IF SUM PLUS	TEST (ACC) + X > 0	
045	ADD	ADD	(ACC) + X → (ACC)	
051	ISN	IF SUM NON-ZERO	TEST (ACC) + X ≠ 0	
053	DIV	DIVIDE	(ACC) / X → (ACC)	
055	ADL	ADD LOGIC	(ACC) + X → (ACC)	
057	RDV	REVERSE DIVIDE	X / (ACC) → (ACC)	
060	OSU	OPERAND SUBTRACT	(ACC) - X → (OA)	
061	FGO	IF GREATER THAN OPERAND	TEST (ACC) > X	
065	SUB	SUBTRACT	(ACC) - X → (ACC)	
071	IUO	IF UNEQUAL OPERAND	TEST (ACC) ≠ X	
073	STZ	STORE ZERO	0 → X	
075	SUL	SUBTRACT LOGIC	(ACC) - X → (ACC)	
077	MPY	MULTIPLY	(ACC) * X → (ACC)	

100	OAN	OPERAND ADD AND NEGATE	- (ACC) - X → (OA)
101	FSM	IF SUM MUNUS	TEST (ACC) + X < 0
105	ADN	ADD AND NEGATE	- (ACC) - X → (ACC)
111	IEZ	IF EXTRACT ZERO	TEST (ACC) ∧ X = 0
113	STS	STORE SINGLE	(ACC) → X
115	EXL	EXTRACT LOGIC	(ACC) ∧ X → (ACC)
120	OSN	OPERAND SUBTRACT AND NEGATE	- (ACC) + X → (OA)
121	FLO	IF LESS THAN OPERAND	TEST (ACC) < X
125	SUN	SUBTRACT AND NEGATE	- (ACC) + X → (ACC)
131	IEC	IF EXTRACT COMPLEMENT ZERO	TEST (ACC) ∧ ¬ X = 0
133	STI	STORE INTEGER	(ACC) → X
135	ECL	EXTRACT COMPLEMENT LOGIC	(ACC) ∧ ¬X → (ACC)
137	SKP	SKIP	(NC) + X → (NC)
140	OAA	OPERAND ADD AND ABSOLUTE	|(ACC) + X| → (OA)
141	FSN	IF SUM NON-ZERO	TEST (ACC) + X ≠ 0
145	ADA	ADD AND ABSOLUTE	|(ACC) + X| → (ACC)
151	IUZ	IF UNION ZERO	TEST (ACC) ∨ X = 0
153	STD	STORE DOUBLE	(ACC) → X, X + 1
155	UNL	UNITE LOGIC	(ACC) ∨ X → (ACC)
160	OSA	OPERAND SUBTRACT AND ABSOLUTE	|(ACC) - X| → (OA)
161	FUO	IF UNEQUAL OPERAND	TEST (ACC) ≠ X
165	SUA	SUBTRACT AND ABSOLUTE	|(ACC) - X| → (ACC)
171	IUC	IF UNION COMPLEMENT ZERO	TEST (ACC) ∨ ¬ X = 0
173	STL	STORE LOGIC	(ACC) → X
175	UCL	UNITE COMPLEMENT LOGIC	(ACC) ∨ ¬X → (ACC)
177	TRM	TRANSFER AND MARK	(NC) → (X) ; X + 1 → (NC)

COMMANDS IN ALPHABETICAL ORDER

145	ADA	ADD AND ABSOLUTE	|(ACC) + X| → (ACC)
045	ADD	ADD	(ACC) + X → (ACC)
055	ADL	ADD LOGIC	(ACC) + X → (ACC)
105	ADN	ADD AND NEGATE	- (ACC) - X → (ACC)
002	ADX	ADD TO INDEX	(I) + X → (I)
006	AXT	ADD TO INDEX AND TEST	(I) + X → (I) (TEST(I)≠0)
015	CAL	CLEAR ADD LOGIC	X → (ACC)
005	CLA	CLEAR ADD	X → (ACC)
035	CCL	CLEAR ADD COMPLEMENT LOGIC	¬X → (ACC)
025	CLS	CLEAR SUBTRACT	- X → (ACC)
053	DIV	DIVIDE	(ACC) / X → (ACC)
135	ECL	EXTRACT COMPLEMENT LOGIC	(ACC) ^ ¬X → (ACC)
115	EXL	EXTRACT LOGIC	(ACC) ^ X → (ACC)
061	FGO	IF GREATER THAN OPERAND	TEST (ACC) > X
121	FLO	IF LESS THAN OPERAND	TEST (ACC) < X
021	FOM	IF OPERAND MINUS	TEST X < 0
001	FOP	IF OPERAND PLUS	TEST X > 0
101	FSM	IF SUM MINUS	TEST (ACC) + X < 0
141	FSN	IF SUM NON-ZERO	TEST (ACC) + X ≠ 0
041	FSP	IF SUM PLUS	TEST (ACC) + X > 0
161	FUO	IF UNEQUAL OPERAND	TEST (ACC) ≠ X
031	ICZ	IF COMPLEMENT ZERO	TEST ¬X = 0
131	IEC	IF EXTRACT COMPLEMENT ZERO	TEST (ACC) ^ ¬ X = 0
111	IEZ	IF EXTRACT ZERO	TEST (ACC) ^ X = 0
011	IOZ	IF OPERAND ZERO	TEST X = 0
051	ISN	IF SUM NON-ZERO	TEST (ACC) + X ≠ 0
171	IUC	IF UNION COMPLEMENT ZERO	TEST (ACC) v ¬ X = 0
071	IUO	IF UNEQUAL OPERAND	TEST (ACC) ≠ X
151	IUZ	IF UNION ZERO	TEST (ACC) v X = 0
032	LXM	LOAD INDEX MINUS	- X → (I)
012	LXP	LOAD INDEX PLUS	X → (I)
077	MPY	MULTIPLY	(ACC) * X → (ACC)
140	OAA	OPERAND ADD AND ABSOLUTE	|(ACC) + X| → (OA)
040	OAD	OPERAND ADD	(ACC) + X → (OA)
100	OAN	OPERAND ADD AND NEGATE	- (ACC) - X → (OA)

000	OCA	OPERAND CLEAR ADD	X → (OA)
020	OCS	OPERAND CLEAR SUBTRACT	- X → (OA)
160	OSA	OPERAND SUBTRACT AND ABSOLUTE	|(ACC) - X| → (OA)
120	OSN	OPERAND SUBTRACT AND NEGATE	- (ACC) + X → (OA)
060	OSU	OPERAND SUBTRACT	(ACC) - X → (OA)
057	RDV	REVERSE DIVIDE	X / (ACC) → (ACC)
013	REP	REPEAT	REPEAT
137	SKP	SKIP	(NC) + X → (NC)
153	STD	STORE DOUBLE	(ACC) → X, X + 1
133	STI	STORE INTEGER	(ACC) → X
173	STL	STORE LOGIC	(ACC) → X
113	STS	STORE SINGLE	(ACC) → X
073	STZ	STORE ZERO	0 → X
065	SUA	SUBTRACT AND ABSOLUTE	|(ACC) - X| → (ACC)
165	SUB	SUBTRACT	(ACC) - X → (ACC)
075	SUL	SUBTRACT LOGIC	(ACC) - X → (ACC)
125	SUN	SUBTRACT AND NEGATE	- (ACC) + X → (ACC)
022	SUX	SUBTRACT FROM INDEX	(I) - X → (I)
026	SXT	SUBTRACT FROM INDEX AND TEST	(I) - X → (I) (TEST(I)≠0)
017	TRA	TRANSFER	X → (NC)
177	TRM	TRANSFER AND MARK	(NC) → (X) ; X + 1 → (NC)
175	UCL	UNITE COMPLEMENT LOGIC	(ACC) ∨ ¬X → (ACC)
155	UNL	UNITE LOGIC	(ACC) ∨ X → (ACC)
010	XEQ	EXECUTE OPERAND	EXECUTE X AS COMMAND
016	XPT	LOAD INDEX PLUS AND TEST	X → (I) (TEST(I)≠0)
036	XMT	LOAD INDEX MINUS AND TEST	- X → (I) (TEST(I)≠0)

APPENDIX C
SUDOS IN 'WHAT'

ALF	ALPHANUMERIC INFORMATION
CHK	CHECK
COM	COMMENT
CPY	COPY
DEF	DEFINE
DMP	DUMP
ENT	ENTRY
FPC	FULL PRECISION CONSTANT
HPC	HALF PRECISION CONSTANT
LBL	LABEL
LIN	LINE
LWD	LOGIC WORD
NAM	NAME
PAG	PAGE
PRT	PRINT
REL	RELEASE
WRD	WORD

APPENDIX D
G-20 SHIFT MULTIPLIERS

LEFT SHIFT	NUMBER	RIGHT SHIFT
1	0	000 00 00001
2	1	101 00 00004
4	2	101 00 00002
10	3	101 00 00001
20	4	102 00 00004
40	5	102 00 00002
100	6	102 00 00001
200	7	103 00 00004
400	8	103 00 00002
1000	9	103 00 00001
2000	10	104 00 00004
4000	11	104 00 00002
10000	12	104 00 00001
20000	13	105 00 00004
40000	14	105 00 00002
05 00 00001	15	105 00 00001
05 00 00002	16	106 00 00004
05 00 00004	17	106 00 00002
06 00 00001	18	106 00 00001
06 00 00002	19	107 00 00004
06 00 00004	20	107 00 00002
07 00 00001	21	107 00 00001
07 00 00002	22	110 00 00004
07 00 00004	23	110 00 00002
10 00 00001	24	110 00 00001
10 00 00002	25	111 00 00004
10 00 00004	26	111 00 00002
11 00 00001	27	111 00 00001
11 00 00002	28	112 00 00004
11 00 00004	29	112 00 00002
12 00 00001	30	112 00 00001
12 00 00002	31	113 00 00004

BRIEF DECIMAL-OCTAL CONVERSION TABLE

DECIMAL	OCTAL
10	12
20	24
30	36
40	50
50	62
60	74
70	106
80	120
90	132
100	144
200	310
300	454
400	620
500	764
600	1 130
700	1 274
800	1 440
900	1 604
1 000	1 750
2 000	3 720
3 000	5 670
4 000	7 640
5 000	11 610
6 000	13 560
7 000	15 530
8 000	17 500
9 000	21 450
10 000	23 420
20 000	47 040
30 000	72 460
40 000	116 100
50 000	141 520
60 000	165 140
70 000	210 560
80 000	234 200
90 000	257 620
100 000	303 240

OCTAL	DECIMAL
10	8
20	16
30	24
40	32
50	40
50	48
70	56
100	64
200	128
300	192
400	256
500	320
600	384
700	448
1 000	512
2 000	1 024
3 000	1 536
4 000	2 048
5 000	2 560
6 000	3 072
7 000	3 584
10 000	4 096
20 000	8 192
30 000	12 288
40 000	16 384
50 000	20 480
60 000	24 576
70 000	28 672
100 000	32 768
200 000	65 536
300 000	98 304
400 000	131 072
500 000	163 840
600 000	196 608
700 000	229 376
1 000 000	262 144

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         M0, 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.
     M0          shift constants
          LWD        $24, $16, $8, $0;
     M1          exit from procedure
AL   end  of NEXTCHAR
```