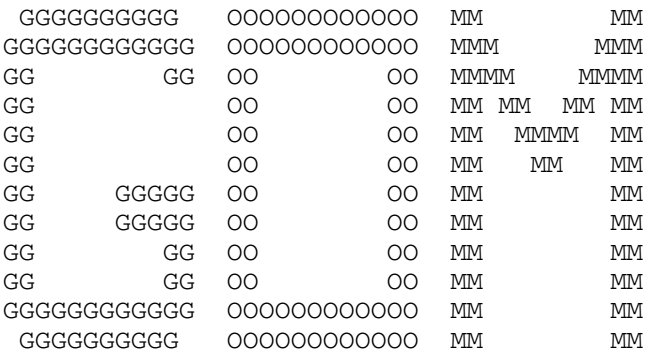


Last revision: 30 June 1989

TABLE OF CONTENTS

GOM == MAD/360

INTRODUCTION

This document provides a language description and usage information for a language which is essentially the 7090 MAD language, modified and extended for the 360/370 architecture.

This language is currently known as GOM (for Good Old MAD) to distinguish it from the ill-fated MAD/1 which was designed as an extensible language and which, although it provided many ideas which have been borrowed, was incredibly ugly and never extended itself into usability.

Why at this point in time resurrect the language? The main impetus was that a language was needed for writing Grungy Little Programs and pieces of the operating system. In both cases, the programs are often machine-dependent, and require matching previously established data structures and interfaces that are not "standard", in the sense of being used by most high-level languages. Fortran is not usually general enough, and PL1 suffers from its "environment" and in many cases is not general enough either.

Rather than invent a new language, MAD was chosen as a base because it provided a happy medium between the insufficiencies of Fortran and the excesses of PL1, and because the semantic implications of the various constructs were known. That did not, however, prevent judicious tailoring of the language to better fit the current programming practices and problems.

In the remainder of this document, Chapter I contains the mechanics of writing programs and running the compiler. Chapter II, the language description, is a revision of Chapter II of the MAD Manual (1966 edition). Appendix E contains all the sections (or parts thereof) that are currently not yet implemented. If and when these sections (or portions) become available, they will be moved to designated positions in these chapters.

Chapter I

Mechanics of Running GOM

Input Format

Input is free form. If the first column of a statement is blank, then the statement has no label and the first non-blank character begins the statement. If the first column is non-blank, there is a label beginning there and continuing to the first blank. End of input line is end of statement unless the next input line begins with a continuation character ("+") in column 1. Multiple statements may be put on one input line by separating them with semi-colons (";"). For example, a line which has the form

label stmt;label stmt;▪stmt (▪ represents a blank)

is a legal construct in GOM. (Note that the first label must appear in column 1 of the line.)

An asterisk ("*") in the first column indicates a comment. Lines beginning with the characters "1", "-", or "0" in column 1 are treated as spacing control. The column 1 character is used as carriage control and the remainder of the line (if any) is printed so that column 2 lines up with where column 1 normally prints for a regular source line. Note that this spacing control works only for the source listing when compiling a GOM program.

Input lines may be up to 255 characters in length. The whole of an input line is taken as input; no sequence id field is allowed.

| Input may be in upper or lower case. Except in character
| constants (where it is preserved), case is ignored.

Source Program Inclusion

Normally, the source cards for a GOM program need only be given with the SCARDS=sourcepgm parameter on the $RUN command. But GOM also allows additional source code inclusion via the standard

$CONTINUE WITH FDname RETURN

command. In addition to this, GOM has an INCLUDE statement, which not only allows for additional source code inclusion, but
| also provides source listing documentation of what's happening
| as well as a source library facility. It is in two forms:

| INCLUDE Member
| INCLUDE@NOLIST Member

| In both cases the INCLUDE statement itself is printed in the source data listing. In the second case the data listing coming from the member is suppressed. If an INCLUDE statement is used then unit 0 on the Run command for Gom must be assigned to a library or concatenation of libraries containing the members included. These libraries are in the same format as used for Plus and can be maintained using the same tools (for example, Plus:LibGen). The INCLUDE statement may not be continued, it must be complete on one line. They may be nested to a maximum depth of 10.

Running the Compiler

The compiler is in the file *GOM, and is run as follows:

```
$RUN *GOM SCARDS=input SPRINT=output SPUNCH=obj SERCOM=errors -
- 0=library PAR=parameters
```

Note that the MTS default continuation character, a minus sign, is used to break up the run command on two separate lines. Source modules are read from SCARDS and compiled until an end-of-file is encountered. A summary line for each module compiled is printed on the user's terminal. If SPUNCH is not assigned, the object program is put in the file -LOAD (which is emptied first). If SPRINT is not assigned in conversational mode, no listings are produced; otherwise, all listings are produced. If no listings are produced or if the listings are not printed on the user's terminal, error messages are produced on SERCOM (which defaults to the user's terminal). The PAR filed can be used to override things. Acceptable parameters are:

```
[NO]FLAGUNDEF - see below (Default is NOFLAGUNDEF)
[NO]DECK - object output (Default is DECK)
[NO]LIST - object program listing (Default is NOLIST)
[NO]MAP - variable storage layout (Default is NOMAP)
SM - same as SOURCE and MAP
SML - same as SOURCE and MAP and LIST
[NO]SOURCE - source program listing (Default is SOURCE)
[NO]TEST - produce SYM object records (Default is NOTEST)
[NEW]TEST - produce SYM object records in MDX format designed
       for GOM.  This is not supported by very  old  versions
       of SDS.
[NO]VWARN - see below (Default is VWARN)
[NO]XREF - cross reference listing (Default is XREF)
XREF=SHORT - see below
```

FLAGUNDEF: When this parameter is in effect, it causes all variables that have not been declared to be flagged with a warning message. If this option is requested, it replaces the list of variables that were referred to only once. The list of variables used in the program, but to which only one reference is made, is printed out for each compilation, right after the source statements listing. This list has proven to be an

invaluable debugging aid, since misspelled names almost invariably show up on this list. It should therefore be checked carefully whenever it appears.

VWARN: When this parameter is in effect, it prints a warning message about variables that are referenced only once.

XREF=SHORT: This parameter forces a cross reference listing for all symbols except those that have only one reference.

In its current state (unfinished), the compiler occasionally has indelicate moments. We would, however, like to know of any problems so we can fix them.

The GOM compiler batch compiles. That is, it reads from SCARDS until it encounters an end-of-file and compiles all modules found along the way. A summary line, similar to what FTN produces, is printed for each module, and the resultant return code is the maximum of the return codes encountered while compiling.

Running the Object Programs Produced

The compiler puts out a standard object deck that can be simply RUN (or DEBUGged). If the object deck is run under DEBUG, then the PAR=TEST parameter must be included on the run command since SDS requires that SYM records be present in the object deck.

Debugging Object Programs

Currently the SYM records produced by the PAR=TEST option are PL3 records instead of a type specifically for GOM. As a result, the following idiosyncracies exist:

1. Since all arrays start with subscript 0 (except multidimensional arrays where the user specifies a different lower bound), but SDS has never heard of an array starting except with subscript 1, array accesses via SDS will be off by 1 on the subscript.

2. Scalar statement label constants (e.g., "ALPHA" starting in first column of a source statement) are handled as normal -- they refer to the first instruction of that statement, and to set a breakpoint, one tells SDS "BREAK ALPHA". However, since "BETA(1)" as a statement label constant has no relationship to "BETA(2)" as a label on a different statement, the symbol table entry for SDS of BETA refers to an array of pointers to the statements involved, and thus to set a breapoint at BETA(2), one must tell SDS "BREAK $BETA(2)".

As with Fortran, internal statement numbers may be used to refer to a statement, using the same form as with Fortran, "IS#n", where "n" is the internal statement number minus any

leading zeros.

Diagnostics

During the process of translation, many kinds of errors in the formation of statements can be detected. To understand this error detection and the subsequent printing of diagnostic comments, some knowledge of the structure of the translator is helpful. The translation is accomplished in three major steps:

1. The decomposition of the executable statements into operand-operator-operand triples and the digestion of declarative statements into internal tables.

2. The analysis of the declarative information to do the storage allocation.

3. The combination of the information produced by the first two steps to translate the triples to object code.

When an error of severity greater than "warning" is encountered in one of these steps, translation proceeds only to the end of that step and then ceases. It should be understood, then, that not all dectectable errors may be found because some are detectable only in a later stage of translation. Also, an error in a statement may be such that it causes the translator to misinterpret later statements, thus giving error indications even though no error exists in the later statements.

When errors occur, the offending statement (if any) and the error message are produced on SERCOM (usually the user's terminal) if SPRINT has been assigned elsewhere-- this is in addition to the error message produced on SPRINT. If SPRINT defaults to the user's terminal, then the error message is printed on the terminal. The offending statement will also be printed if the SOURCE parameter has defaulted off. (If SOURCE is on, of course, the offending statement has already printed.)

Chapter II

DESCRIPTION OF THE LANGUAGE

"There is pleasure sure in being mad, which none but madmen know."

Dryden: The Spanish Friar

1. Modes: Constants, Variables, and Expressions

Operands can be divided into a number of categories called modes. From an abstract view, each mode is characterized by the range of values a variable of that mode may have and what operations are permitted. From a physical view, each mode may be also characterized by the number of bytes of storage a variable of that mode occupies, the internal representation used, and the hardware operations used.

The modes are:

Mode Number	Name	Size of Element	Default Alignment
0	FLOATING POINT	4	word
1	INTEGER	4	word
2	BOOLEAN	4	word
3	FUNCTION NAME	4	word
4	STATEMENT LABEL	4	word
5	CHARACTER	1	byte
6	SHORT INTEGER	2	halfword
7	BYTE INTEGER	1	byte
8	LONG INTEGER	8	word
11	POINTER	4	word
12	DYNAMIC RECORD	N/A	N/A

Important Note

Variables are not initialized by GOM. If you use a variable before giving it a value, it will have a more or less random value.

Two groups of modes will be discussed in what follows. They are:

(a) Arithmetic modes; those in which arithmetic-type operations can be done. These are Floating Point, Integer, Short Integer, Byte Integer, Long Integer, and Long Floating Point. The term expression or arithmetic expression will refer to expressions whose operands are from these modes.

(b) Integer modes; a subset of arithmetic modes is integer expressions. This refers to expressions whose operands are of Integer, Short Integer, Byte Integer, or Long

Integer mode.

NOTE: There are two modes that are not yet implemented. See Appendix E, Number 1 for the description of these modes (VARYING CHARACTER and LONG FLOATING POINT).

1.1 Constants

This section describes constants of all modes except statement label and function name, which are described in Sections 1.3 and 4.1 respectfully.

1.1.1 Integer Constants

An integer constant is composed of digits, optionally preceded by a "+" or "-" sign. The value of the constant must fall in the range of -2147483648 to +2147483647.

While the "+" sign may be omitted, the "-" sign must be present if the number is negative (e.g., 2, -2, 0, +0, -0, 100 are all integers). Note that commas are not allowed in integer constants (e.g.; 1,500 is illegal).

1.1.2 Floating Point Constants

Floating point constants may be written with or without exponents. If written without an exponent, the constant must contain a decimal point ".", which may appear anywhere in the number. Thus, 0., 1.5, -0.05, +100.0, .1 and -4. are all floating point constants.

If the number is written with an exponent, it may be written with or without a decimal point, followed by the letter "E", followed by the exponent of the power of 10 that multiplies the number. (If the decimal point is omitted, it is assumed to be immediately to the left of the letter "E".) The exponent m consists of one or two digits preceded by a sign (although a "+" sign may be omitted), and must satisify the condition $-78 \leq m \leq 76$. More specifically, the value of the number F must be 0 or else satisfy the condition

$$.5397605 \times 10^{-78} \leq |F| \leq .7237005 \times 10^{76}$$

Examples of floating point constants with exponent are: .05E-2 ($=.05 \times 10^{-2}$), -.05E2 ($= -.05 \times 10^{2}$), 5E02 ($= 5.0 \times 10^{2}$), 5.E2 ($= 5.0 \times 10^{2}$).

Negative floating point constants must be preceded by a "-" sign. Positive constants may be preceded by a "+" sign.

1.1.3 Character Constants

Constants of character mode consist of a string of characters preceded and followed by one of the two character delimiters: dollar sign ($) and double quote ("). Whichever delimiter begins a character constant must be the one that ends it. Note that blanks, while ignored elsewhere in the language, count as characters in character constants and that case is significant in them while it is not elsewhere.

Within any character constant, a pair of consecutive delimiters will be treated as a single occurance of that character which is to be placed in the resultant constant. Thus,

Both "A$B" and $A$$B$ represent the string A$B
$A"B$ and "A""B" represent the string A"B

Note that since the length of an element of character mode is 1, a character constant with more than one character in it is implicitly a one dimensional array, and as such it can be subscripted, just like a dimensional character variable.

For example: "1234567890"(I)

In this example, the result will be one character determined by the value of I. If I = 4, then the result of "1234567890"(I) will be the character 5.

Character constants may be padded with blanks by appending "@Ln" to the constant. For example, "GUSER"@L8 will pad the string "GUSER" with 3 blanks in order to make the length of the entire string equal to 8 characters. (See Section 1.1.10 for constant qualification.)

1.1.4 Boolean Constants

There are two Boolean constants -- "true", which is written 1B, and "false", which is written 0B.

1.1.5 Hexadecimal Constants

These constants are written as a string of hexadecimal digits followed by an "@X" qualifier. If not otherwise qualified, the quantity is right-justified in 4 bytes and assigned integer mode. (See Section 1.1.10 for constant qualification.)

Examples: 123@X 1AB7@X DEF@X

1.1.6 Short Integer Constants

These are written the same as integer constants, except

that they must have a qualifier appended to indicate that the constant is a 2-byte integer. The following are all short integer constants of value "1":

```
1@L2   1@IL2   1@SI   1@SR   1@H
```

The value of a short integer must lie in the range -32768 to +32767.

1.1.7 Byte Integer Constants

These are written the same as integer constants, except that they should not have a sign, and that they must have a qualifier appended to indicate that the constant is a 1-byte integer. The following are all byte integer constants of value "1":

```
1@L1   1@IL1   1@BI   1@BR
```

The value of a byte integer must lie in the range 0 to 255.

1.1.8 Long Integer Constants

A long integer constant is the same as an integer constant (it follows the same rules as an integer constant) except that the constant can fall in the range of -9223372036854775808 to +9223372036854775807. Any INTEGER constant which is too big for a fullword integer will automatically be converted to a LONG INTEGER constant. Constants can also be forced to LONG INTEGER mode by appending the modifiers @LR, @LI, or @L8 to the digits. Examples of LONG INTEGER constants are:

```
1234567898765@LI     -201473548714@LR     123456789ABC@XL8
```

In the third example, a LONG INTEGER in hex of length eight was specified.

1.1.9 Pointer Constants

A pointer constant is the same as an integer constant, except that it has an "@P" qualifier attached to indicate that it is of pointer mode. The only commonly-used pointer constant is the null pointer, written as 0@P .

1.1.10 Constant Qualification

Constants may have their default characteristics overridden by appending a qualifier consisting of an at-sign (@) followed by one or more characters. There are four categories:

(a) X says that the constant is expressed in hexadecimal

(b) Ln where n is a number, says that the resulting constant should (internally) be of length n. Note that this can also modify the mode, since many modes represent a specific combination of type and length. Thus an integer constant with "@L2" will set the mode to short integer, and is equivalent to @SI et al. given below.

(c) Mn says the resulting constant should have mode number n. (See the list of mode numbers in the table of Section 1.)

(d) Mode Specifications. These will also set the length, unless it is explicitly given (in form (b) above). The legal forms consist of several common forms, including the statement-type abbreviation (minus the prime) for the mode in question. They are:

@F	@FT		FLOATING POINT
@I	@IR		INTEGER
@BN			BOOLEAN
@C	@CH	@CR	CHARACTER
@SI	@SR	@H	SHORT INTEGER
@BI	@BR		BYTE INTEGER
@LI	@LR		LONG INTEGER
@VC	@VR		VARYING CHARACTER
@LT			LONG FLOATING POINT
@P			POINTER

Items from these categories may be combined in meaningful combinations . Examples of meaningful combinations are:

```
@XL8      @IL2
```

An example of a non-meaningful combination is:

```
@BIL8
```

(Here, we are trying to express a byte integer of length eight)

1.2 Variables (integer, floating-point or Boolean)

The name of an integer, floating-point or Boolean variable consists of one to 24 letters, underscores (_), or digits, the first of which must be a letter. If the variable is defined as an n-dimensional array variable (see Section 3.3) then the name of an element of the array consists of the variable name, (i.e., one to twentyfour letters, underscores, or digits, starting with a letter), followed by the appropriate subscripts separated by commas and enclosed in parentheses. Thus the following are "single variables": X51, ALPHA6, LAMBDA, GROSS, while the following are elements of arrays: BETA(C1, C2, 6), X15(Y,Z1), J(6), J(Z1 + 5*Z2, 5). (See Section 1.11 for the description of subscripts.) Parentheses enclosing subscripts may not be omitted.

1.3 Statement Labels

A statement may be labeled or unlabeled. Labels are used to refer to a statement by other statements. A statement label consists of from one to twentyfour letters, underscores, or digits, the first of which must be a letter, e.g., IN or BACK. A statement label may be an element of a statement label vector, in which case the vector name is followed by a constant integer subscript enclosed in parentheses, e.g., S(2) or LBL(3). A statement label appears in the first column of the statement it identifies. If a statement does not have a label, the first column must contain a blank. A statement label cannot have any embedded blanks; the first blank found after column 1 terminates the label, and the body of the statement begins with the first nonblank character following.

1.3.1 Statement Label Constants

A statement label starting in the first column of a statement will be called a constant of statement label mode. Statement label constants are assigned to the first CSECT. Previously (in MAD 7090), these labels were compiled as statement label variables that were initialized.

The only change visible to users is in the use of labels via SDS. In the past, all labels had SYM record entries that pointed to the label adcon, so in order to set a breakpoint, for example, one had to say "BRE $ALPHA". Now all scalar labels (that is, non-subscripted labels) have SYM record entries that actually point to the code for that statement, and thus can be used in the normal way (e.g., "BRE ALPHA"). However, a statement label constant array-- e.g., labels in a program that have the same name, but different subscripted values such as NAME(1), NAME(2), ..., NAME(N) --still had a SYM record entry pointing to the array of adcons, so you must go indirect via the "$". (This may be resolved later, to be more consistent.)

1.3.2 Statement Label Variables

A statement label which does not appear in the label field is a variable of statement label mode, provided it is so declared in a mode declaration (See Section 3.2).

1.4 Arithmetic Operations

The following arithmetic operations are available for operands of arithmetic modes, except where noted otherwise.

(a) Addition, written "+", e.g., Z5 + D.

(b) Subtraction, written as "-", e.g., Z5 - D.

(c) Multiplication, written as "*", e.g., Z5*D. (Note that the "*" may not be omitted. It is illegal to write Z5D, since it would be impossible to distinguish such a product from the variable Z5D.)

The operator .MPYLI. is available to obtain the double-word product produced by the hardware integer multiply operation. The operands of this operator must be of integer, short integer, or byte integer mode; the result is of long integer mode.

Multiplication is not allowed if either or both of the operands are long integers, currently.

(d) Division, written as "/"; e.g., Z5/D. If both Z5 and D are integers, the result is again an integer; e.g., the "fractional part" of the true quotient is truncated (not rounded). For example, if Z5 = 8, and D = 3, then Z5/D will have the value 2.

Long integers may not be divided by long integers; they may be divided only by integers, short integers or byte integers.

(e) Remainder, written as ".REM."; e.g., AB8.REM.BA7, and meaning the remainder from the integer division of AB8 from BA7. This is a binary operator whose operands must be integer, short integer, or byte integer. It has the same precedence as "/". It generates exactly the same code as "/", except that the value indicated as a result of the operation is what was in the other register. For example, if AB8=5 and BA7=3 then AB8.REM.BA7=2. Also note that if AB8=16 and BA7=4 then AB8.REM.BA7=0.

(f) Exponentiation, written as ".P."; e.g., Z5.P.D, and meaning Z5 raised to the power D.

Neither of the operands of the .P. operator can be long integers.

(g) Absolute value, written ".ABS."; e.g., .ABS.Z5, meaning |Z5|, the absolute value of Z5.
The operand of .ABS. may not be a long integer.

(h) Negation, written as "-"; e.g., -ALOHA, meaning the "negative of ALOHA". Thus -X.P.-5 means $-(X^{-5})$.
The operand may not be a long integer.

Important Note

When certain operations cannot be implemented due to operand conditions, the reason lies in the fact that the machine is unable to handle these special conditions, and not due to the fact that the GOM compiler is unable to handle it. For example, the "*" operator for multiplication cannot accept LONG INTEGERs as operands.

The following operations are available only for operands of one of the integer modes.

(i) Full word bitwise negation, written .N.i, where "i" is an integer expression, and meaning the operation of negating each binary digit in the binary representation of the value of "i". The result is again an integer.
Example: Let I = 6 which is represented internally as

000...000110 (32 bits)

Then .N.I would yield the value:

111...111001

(j) Full word bitwise logical operations and, or, and exclusive or written .A., .V., and .EV., respectively, meaning the bitwise and, or, and exclusive or of the full binary integer values of the operands. The result is again an integer.

Example: Let I = 17 which is represented internally as

0000...00010001

and let J = 9 which is represented internally as

0000...00001001

Then I.A.J would yield the value

0000...00000001

which is the integer 1,

I.V.J would yield the value

0000...00011001

which is the integer 25,

and I.EV.J would yield the value

0000...00011000

which is the integer 24.

(k) Full word integer shifts, written .LS. and .RS., respectively; e.g., i.LS.j and i.RS.j, where "i" and "j" are integer expressions (see Section 1.10).

"j" may not be a long integer. i.LS.j means the value of "i" shifted left "j" binary places. The shift is a logical shift, rather than an arithmetic shift, that is, "j" must be a positive quantity (or zero).

Similarly with .RS., digits shifted off either end of the computer word are lost. Created blank positions are filled with zeros. The result is an integer, unless "i" was a long integer, in which case the result is a long integer.

Example: Let I = 30 which is represented internally as

000...0011110

and let J = 4.

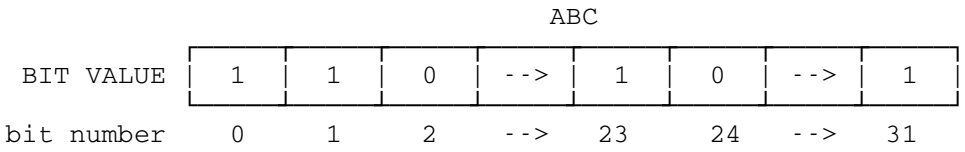
Then I.LS.J would yield a value which is represented as

000...111100000

and I.RS.J would yield a value which is represented as

00000000...0001

(1) Bit operations using the predefined bit operators .SETBIT. and .RESETBIT. These operations manipulate bit patterns of constants or variables. The constant or variable must be of integer mode. The figure below will be used to show how .SETBIT. and .RESETBIT. work. Note that the Boolean operator .BIT. is described in Section 1.7.



Here, the 23rd bit number of ABC has a BIT VALUE of 1

.SETBIT. is a binary operator which sets a bit to 1 regardless whether its current value is 0 or 1. Examples:

```
ABC .SETBIT. 24      sets bit #24 to 1
ABC .SETBIT. 23      keeps bit #23 at 1
```

.RESETBIT. is also a binary operator, but unlike .SETBIT., it sets a bit to 0 regardless whether its current value is 0 or 1. Examples:

```
ABC .RESETBIT. 1     sets bit #1 to 0
ABC .RESETBIT. 2     keeps bit #2 at 0
```

1.5 Arithmetic Expressions

Arithmetic expressions are defined inductively as follows:

(a) All integer and floating point constants, integer and floating point individual variables, subscripted integer and floating point array variables, and integer and floating point values of functions are arithmetic expressions. Note that a function value used in an expression must have an argument list, even if the function is to be called without any arguments (e.g., "FUNCT.()"). (See Sections 4.2 and 4.5 for more information on functions.) The word "integer" here means integer, short integer, byte integer and long integer.

(b) If E and F are any arithmetic expressions, and I and J integer expressions, then the following are also arithmetic expressions: +E, -E, .ABS.E, E + F, E - F, E*F, E/F, E .P. F, (E), .N. I, I .A. J, I .V. J, I .EV. J, I .LS. J, and I .RS. J (except for exclusions noted earlier).

1.6 Boolean Operations

The following Boolean (or logical) operations are available in the language (where M and P are Boolean expressions):

(a) .NOT.M has the value 1B if and only if M has the value 0B.
(b) (M) has the same value as M.
(c) M.OR.P has the value 0B if and only if both M and P have the value 0B.
(d) M.AND.P has the value 1B if and only if both M and P have the value 1B.
(e) M.EXOR.P has the value 1B if and only if either M or P has the value 1B, but not both.
(f) M.EQV.P has the value 1B if and only if M and P have the same values.

Thus .NOT., .OR., .AND., .EXOR., and .EQV. correspond to the usual logical operations.

1.7 Boolean Expressions

Boolean expressions are defined inductively as follows:

(a) Boolean constants, individual Boolean variables, subscripted Boolean array variables and Boolean-valued functions are Boolean expressions. (See Sections 1.1.4 and 3.2)

(b) If H and F are arithmetic expressions, then the following comparatives are Boolean expressions:

H<F	H.L.F	H.LT.F	meaning	h<f
H<=F	H.LE.F	H.LE.F	meaning	h≤f
H=F	H.E.F	H.EQ.F	meaning	h=f
H¬=F	H.NE.F	H.NE.F	meaning	h≠f
H>F	H.G.F	H.GT.F	meaning	h>f
H>=F	H.GE.F	H.GE.F	meaning	h≥f

(The forms in the first column are the "extended" forms, the second column are the old 7090 MAD forms, and the third column are the FORTRAN forms. All three forms are legal to use in GOM).

(c) If M and P are Boolean expressions, then the following are also Boolean expressions: .NOT.M, (M), M.OR.P, M.AND.P, M.EXOR.P., and M.EQV.P.

(d) A constant or variable operated on by the .BIT. operator is a Boolean expression. The .BIT. operator examines a specified bit number of a constant or variable, and returns 1B (true) if the value at the bit number is 1, or it will return 0B (false) if the value is 0. An example .BIT. operation is

```
IF ABC .BIT. 23
  X=Y    ;* X=Y if ABC .BIT. 23 returned 1B
ENDIF
```

Note: see Section 2.3 for the description of the IF statement.

Examples of Boolean expressions are:

```
(X.G.3 .AND. Y.LE.2) .OR. (GAMMA.L.EPSILON)
(.ABS.(X1-X2)/X1 .LE. EPSILON) .AND. (F.(X1) .L. EPSILON)
(P.AND.Q) .EQV. (P .OR. .NOT.P)
```

where P and Q are Boolean variables.

1.8 Parentheses Conventions (Precedence of Operators)

Parentheses are used in the same way as in ordinary algebra and logic to specify the order of the computation. Also, certain conventions are used to allow deletion of parentheses. The conventions used here are the same as in ordinary algebra and logic, i.e.; parentheses may be omitted, subject to the rules (A) and (B) below, but redundant parentheses are allowed.

Appendix B contains a list of all operators. In it, references are made as to where various operators are defined in this document.

(A) The sequence of computation within any expression, unless otherwise indicated by parentheses, is shown below. Note that

higher operations on this list are done before lower operations. The PREC=n settings are only for use by Computing Center staff for diagnostic purposes.

```
PREC=42   .IND.
     40   function_call  subscription  :  .SIZE.
     39   -> (as return code operator)
     38   .LOC.  .DIMVEC.
     37   .AS.
     36   .N. .LS. .RS. .ABS. .NBRARG. .ARG.
     34   .A.
     32   .V. .EV.
     30   .P.
     28   - (unary)
     26   * / .MPYLI. .REM.
     24   + -
     22   .BIT. .L. .LT.  < .E. .EQ. =(comparative) .G. .GT.  >
          .LE. <= .NE. ¬= .GE. >=
     20   .NOT.
     18   .AND.
     16   .OR.  .EXOR.
     12   .EQV.
     10   ...  =(in arg for R-Call)  ->(out arg for R-Call)
      6   .SETBIT. .RESETBIT. =(substitution)
```

Examples:

(1) .ABS.(B-C) means |B-C|, while .ABS.B-C means |B|-C.
(2) -B + C means (-B) + (C), while -(B + C) means the negation of the sum.
(3) B.P. - 2 + 3 means $B^{-2} + 3$, while B.P.(-2 + 3) means B^{-2+3}.
(4) K2/Z - 3 means (K2/Z) - 3, while K2/(Z - 3) implies that Z - 3 is the denominator.
(5) A * B + C means (A * B) + C.
(6) A.P.3/J means $(A^3)/J$.
(7) X.L. Y + 3 means (X) .L. (Y + 3).
(8) P.AND..NOT.P.OR.Q means (P.AND.(.NOT.P)) .OR.Q.
(9) Z = X + Y/QA means Z = (X + (Y/QA))
(10) A = -B.P.2 means A = $-(B^2)$.

(B) Within an expression, operations appearing on the same line of the list in (A) are to be performed from left to right, unless otherwise indicated by parentheses.

Examples:

(1) A + B - C + D - E means (((A + B) - C) + D) - E.
(2) X/Z * Y/R * S means (((X/Z) * Y)/R) * S.

1.9 Mode of Expressions

The kind of arithmetic performed on a constant, variable or

function value is determined by its mode. There are many modes; they were discussed in Section 1. Section 3.2 describes how the modes of variables and functions are specified.

If an expression consists entirely of one constant, one variable, or one functional value, the mode is that of the constant, variable, or functional value itself. If the expression is a compound expression, i.e., it consists of two or more subexpressions joined by logical or arithmetic operations, the following rules apply:

> If an expression is a Boolean expression as defined in Section 1.8, then its mode is Boolean. An arithmetic expression is considered to be in the floating point mode if any operand of any arithmetic operation in the expression is in the floating point mode. If all operands are integer, then the expression is considered to be in the integer mode. In this determination arguments, though not values, of functions are ignored. For other modes, see Sections 1.12 through 1.14.
>
> Operands that are short integers and byte integers are automatically converted into integers before the operation. Integer results that are to be stored in short integers have the right half of the expression stored (i.e., the hardware operation STH is used). Integer results that are to be stored in byte integers have the right-most byte stored (i.e., the hardware operation STC is used). Users not familiar with these hardware operations should read the I.B.M. 370 Principles of Operations Manual.
>
> If one of the operands is a long integer and the other is an integer, short integer, or byte integer, then the shorter operand is converted to long integer before the operation, except for the right operand of division, shifts, and the subscript in a subscription operation. Long integer results that are to be stored in integers have the right-most word stored.

Thus, if Y, Z, and W are floating point variables, while the function GCD. and the variables I and J are in the integer mode, then the expressions

```
Y + GCD.(I,J)
Y + Z - I
I + 1.
GCD.(I, J)/Z
```

are all floating point expressions while the expressions

```
I + GCD.(I, J)
(I + J)/3
I + 1
GCD.(I,J)/I
```

are all integer expressions.

If an expression has subexpressions of different modes, a conversion may be necessary before some of the operations can be performed. Thus, in the expression

```
Y + 3
```

if Y is in the floating point mode it cannot be added directly to the integer 3. The user need not be concerned with this since the instructions necessary for the conversion of the integer to floating point form before adding are automatically inserted by the translator during the translation process.

In some cases, however, the user must understand the sequence in which the conversions will be made. Consider the expression

```
(Y + 7/3) + (I * J/K)
```

where Y is in the floating point mode, and I, J, and K are in the integer mode. According to the parenthesizing conventions, a computation will proceed in the following order (where the T's are temporary locations):

$T^1 = I \times J$
$T^2 = T^1/K$
$T^3 = 7/3$
$T^4 = Y + T^3$
$T^5 = T^4 + T^2$

and T^5 will be the value of the expression.

Now, since both I and J are integers, the first multiplication will be integer multiplication, and T^1 will be an integer. Since the next step involves two integers, it will be integer division, and, if K happens to have a larger value than T^1 the quotient is 0. Similarly, T^3 will have the value 2 because of the division of two integers. In the computation of T^4, however, we have "mixed modes," since Y is floating point and T^3 is integer. Here T^3 will be automatically converted to floating point before adding. Likewise, in the next step, the integer T^2 will be converted to floating point before adding to the floating point number T^4.

In other words, although the mode of the expression is floating point because of the presence of the floating point variable Y, some of the computation (until Y is involved) is performed in integer arithmetic, and this may occasionally cause the final value to be different from the value one might expect from a different analysis.

In the example above, the divisions would be performed in the floating point mode if the expressions were written:

```
(Y + 7./3) + (I * J)/(K + 0.)
```

Of course, many times the expression will be written as originally stated just to achieve the "truncation" effect.

.AS. Operator

The mode of a variable may be overridden for the span of one operation by using the .AS. operator. This operator takes a variable as the left operand and a mode name as the right operand. Note that unlike constant qualification (using the "@" notation), which often causes the internal form of the constant to be changed, this operator never causes any conversion by itself, and in fact, its purpose is to prevent unwanted conversions.

Note that the .AS. operator has no effect on variables occurring in a statement that is essentially an interface to another module, since modes are determined by other methods in those cases. This includes: arguments to subroutines, and both formatted and Simple I/O lists.

Note also that the .AS. operator tells the compiler to treat a variable as if it were of another mode, and the compiler will therefore generate machine instructions suitable for that other mode. It is the user's responsibility to ensure that this will work for a given case, especially with regard to alignment of the operands when the resultant object program is to run on a 360.

```
Example:        INTEGER I
                FLOATING POINT F
                I = F.AS.INTEGER
                I.AS.FLOATING POINT = F
```

The last two statements of this example both have the effect of picking up the fullword from F and putting it in I without conversion.

```
Example:        INTEGER IARR(1)
                LONG INTEGER L
                IARR.AS.LONG INTEGER = L
```

The previous statement, "IARR.AS.LONG INTEGER = L", does the same as the following two:

```
                IARR(0) = L.RS.32
                IARR(1) = L
```

1.10 Subscript Expressions

Any arithmetic expression (except long integers) may be used as a subscript expression for an element of a linear or two-dimensional array. If the value of the expression is in the

floating point mode (see Section 1.9), it is truncated to integer form before being used as a subscript.

The expressions for subscript elements of an array whose dimension is three or greater must be of integer mode. Moreover, for arrays of dimension three or greater, the use of subscripts having other than integer mode will not be caught as an error by the compiler. Subscript expressions may contain variables with subscripts, etc.

Examples of subscripted variables: J(3), K10(Z + 5 * XY/T), MP(A,B + 6 * J, I * 6/TDX), T(I(J)), MA(K(Z + 5) + T(1) + 6).

1.11 Block Notation

Input/Output lists (see Sections 2.8 and 2.9), VECTOR VALUES statements (see Section 3.6), and character substrings (see Section 1.13) allow the use of block notation. This has the form

A...B

which is interpreted as the entire region from A to B, inclusive. The most common use is in terms of a single array; e.g., A(1)...A(N), and B(I,J)...B(M,N). These would be interpreted as the regions: A(1), A(2), A(3), ..., A(N) and B(I,J), B(I,J+1), B(I,J+2), ..., B(I+1,J), B(I+1,J+1), ..., B(M,N).

1.12 Statement Label Expressions

A single constant, variable, or function value of statement label mode is an expression of statement label mode. The only operations legal for operands of statement label mode, besides transferring to the label specified, are substitution and the .E. and .NE. comparatives (or their syntactic equivalents). There are no automatic conversions from statement label mode to any other mode.

1.13 Character Expressions

A single constant, variable, or function value of character mode is an expression of character mode. The only operations legal for operands of character mode are substitution and comparatives. There are no automatic conversions from character mode to any other mode.

Since most character strings are longer than one character, which would require a loop to process, a subscript range notation may be applied to character variables or constants in

comparatives and in substitution statements. This takes the forms:

1. Name(b...e)
2. Name(b|l)

where in form 1, b is the first subscript to be used and e is the last. In form 2, b again represents the first subscript and l represents the length of the substring starting at subscript b. In both forms, b, e, and l must be integers. In form 1, either or both of the b and/or e may be omitted; they default to the lower and upper bounds of the array or constant. Thus, if "NAME" is a character array:

```
CHARACTER NAME(3)
```

then the following four forms are equivalent and all represent the entire array:

```
NAME(0...3)   NAME(...3)   NAME(0...)   NAME(...)
```

If only name is given with no subscript range at all, then it is the same as name(0) if name is a variable name. If name is a character constant, then when name is given with no subscript range in a comparative or substitution where subscript ranges are involved, it is the same as name(...) and refers to the entire constant as written. Thus, assuming "NAME" is as defined in the preceding paragraph, the following are equivalent:

```
NAME(...) = "ABCD"
NAME(...) = "ABCD"(...)
```

It is important to note how padding and truncation works for strings in a GOM program. The only time when this may be a problem is when two strings of un-equal length are being compared, or when one is assigned the value of the other. When two string of unequal length are being compared, first the shorter string gets padded with blanks (only for the duration of the comparison) so that it has the same length as the "longer" string. Now the comparison can be made. String assignments are a bit more complicated for strings of un-equal length because two situations can arise:

1. The longer string can be assigned the value of the shorter string.
2. The shorter string can be assigned the value of the longer string.

In the first case, the longer string gets the entire value of the shorter string, and then the longer string is padded on the right with blanks up to its full length. In the second case, the shorter string only gets loaded with characters starting from the left most side of the longer string. In other words, the rightmost characters that will not fit in the shorter string are truncated. Note that these two cases apply only when entire

strings are involved. If substrings are being used, then only that portion of the string which has been specified in the substring notation will be compared or assigned a value.

The code produced by the GOM compiler for substring notation is made shorter for some "common" cases:

1. When both subcript ranges can be determined during compilation (due to the fact that they are constants or omitted entirely).

2. When the lengths are the same or else the the target string is shorter (ie, no padding is required), and the number of characters to move is 256 or less.

1.14 Pointer Operations and Expressions

A single constant, variable, or function value of pointer mode is an expression of pointer mode. Integer expressions may be added to or subtracted from pointer expressions; the result is still an expression of pointer mode. A pointer may be subtracted from another pointer; the result is of integer mode. Pointer mode expressions are required as the left operand of the ":" operator. Other operations legal for pointer mode operands are substitution and the comparatives.

The location and indirection operators are also used with pointer variables. The location operator returns a location as a value; the indirection operator returns a value as a location. The location operator is the unary operator ".LOC.". Its value is of pointer mode and is the location of the operand. The indirection operator is the binary operator ".IND.". Its left operand must be of pointer mode and specifies the pointer through which we are going indirect. The right operand must be a mode name specifying the mode of what the pointer expression is pointing at.

For example, consider a hard way to set the value of the integer variable Q to 2:

```
POINTER P
P = .LOC.Q
P.IND.INTEGER = 2
```

Note that if the last statement had been "P.IND.FLOATING POINT = 2", then it would have converted "2" into a floating point value before storing it in the integer variable Q. It is up to the user to keep track of what he is pointing at when using these operators.

2. Statements (Executable)

See Appendix A for admissible abbreviations.

2.1 Assignment Statement

This statement has the form illustrated by

```
ALPHA = Y + Z + F.(X, Y, Z)
```

That is, the left side of the "=" sign consists of the name of a variable (either an individual variable or a subscripted array variable), and the right side consists of any expression of the same mode. The only exceptions to this mode requirement are the cases: (1) If the variable on the left is of integer mode then the value of a floating point expression on the right will be converted to integer mode. (2) If the variable on the left is of floating point mode then the value of an integer expression on the right will be converted to floating point mode. (3) The different integer modes will be converted among themselves as described in Section 1.9.

The assignment statement (sometimes referred to as the substitution statement) is to be interpreted as: "(1) Compute the value of the expression on the right, (2) convert it, if necessary, to the mode of the variable on the left of the "=" sign, and (3) give the variable on the left the value which results from Steps (1) and (2)." (See Section 1.9 for mode of expressions.)

Thus, if Y is a floating point variable, then the statement

```
Y = 1
```

will cause the integer 1 to be converted to floating point and then stored in the location called "Y"; i.e., Y will now have the value 1. (as a floating point number). If the statement were written

```
Y = 1.
```

then the floating point number 1. would be stored in the location "Y"; i.e., Y would again have the floating point value 1., but in this case the conversion of the integer is unnecessary, thus speeding up the computation.

When a floating point number is to be converted to an integer, it is first expressed as a number with both an integer part and a fractional part, and then the fractional part is truncated. Thus, the following floating point numbers:

```
3E5     .3E0     .34568127E2     -.345681E10
```

would convert to the following integers, respectively:

```
          300000     0     34     -3456810000
```

Examples of assignment statements in other modes are:

(1) Assuming B and C have been declared Boolean

```
C = B .AND. D .L. 10.
```

(2) Assuming BILL(3) is a statement label constant and HARRY is a statement label variable

```
HARRY = BILL(3)
```

(3) Assuming FUN is a function name variable

```
FUN = COS.
```

2.2 Transfer Statement (GO TO Statement)

This statement has the form:

```
GO TO s
```

Here s may be any statement label or any expression in statement label mode which labels an executable statement. Execution of this statement causes the computation to continue from the statement whose label is the value of s. Examples:

```
GO TO SUMX

GO TO SWITCH(K+2)
```

Note: if K = 4 then the value of SWITCH(K + 2) is SWITCH(6). This is useful when labels defined in the program run from SWITCH(0)...SWITCH(6)...SWITCH(10) for example. Thus, one can write a program that calls the appropriate driver routine (a SWITCH(N) label) based on some variable K.

NOTE: Medial blanks can be omitted in the GO TO statement if desired (e.g. GOTO). Also, there is another verb available for a transfer statement which is TRANSFER TO s. It follows the same rules as the GO TO verb. When using the GO TO transfer statement verb, make sure to leave at least one blank between GO TO and any following text. The reason for this is that any identifiers of length six or less must be kept distinguished from reserved words that are also of length six or less. It should be remembered that GOM, like FORTRAN, is a blankless language in that all blanks not in character strings are ignored (and in fact, the first thing the translator does is to remove them all). In order to accomodate the above options and still allow one to use variable names of six or fewer characters without difficulty from reserved words, there must now be the requirement that at least one blank be put between the above new "verbs" and any following text. There will be problems only for

users who put blanks in their identifiers (e.g., IF AND ONLY IF = 32), which most people are not aware they can do.

2.3 Conditional Statements

There are two types of conditional statements.

(a) Simple Conditional

```
IF B,Q
```

Here B is a Boolean expression and Q any executable statement except the following: END OF PROGRAM, another conditional, iteration and function entry. If at the time of execution of this statement, the expression B has the value 1B, i.e., true, the statement Q is executed. If, however, B has the value 0B, i.e., false, then Q is skipped and computation continues from the next statement in sequence. The comma in this statement, as in other statements containing punctuation, must be included.

Examples:
```
IF XM.LE.1, GO TO END
IF I.GE.N .AND. J.NE.I-1, I = 0
```

(b) Compound Conditional

```
S1    IF B1

        Z1

S2    ELSE IF B2

        Z2

        .
        .

Sk    ELSE IF Bk

        Zk

Sl    END IF
```

NOTE: The above code was indented for easier readability. In GOM, it is not necessary to indent, but it is nevertheless a good practice. All the programs (or partial programs) in this manual are indented.

Often the last condition Bk expressed is one for which the condition is always true. This may be expressed by the statement

ELSE IF 1B

or alternately the statement

ELSE

The "S"s are statement labels which need not be used unless desired. "k" may be equal one (if no "ELSE IF ..." statements are used). B^1, ..., Bk are Boolean expressions. "Zi" is a sequence (possibly empty) of statements. These may include conditional statements, of either form. Each B is tested in turn, starting with B^1. If B^1 has the value 0B, then B^2 is tested, etc. As soon as some expression, say Bj, has the value 1B, then the statements Zj are executed. At this point the execution of the entire block is considered ended, and computation continues from the first statement after the END IF statement which, in this illustration, we have chosen to label S1. In other words, no more than one of the alternative computations is performed. If none of the B's has the value 1B, none of the computation in the scope of these statements is performed.

Example: The evaluation of the function whose graph is shown in Figure 2-1

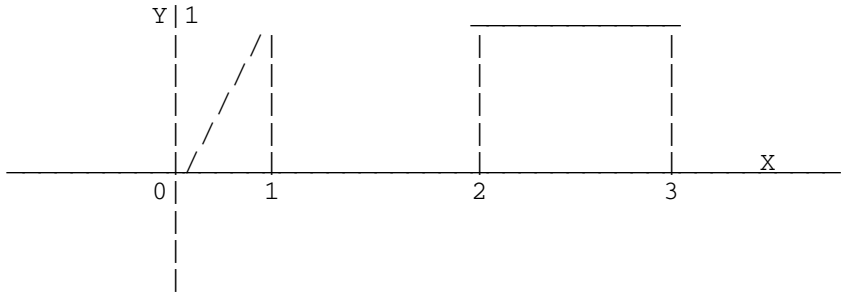


Figure 2-1

might be done by this partial program:

```
IF X<=0. .OR. 1.<=X.AND.X<2 .OR. X>=3.
  Y = 0.
ELSE IF 0.<=X .AND. X<1.
  Y = X.
ELSE IF 2.<=X .AND. X<3.
  Y = 1.
END IF
```

This section of program could be rewritten in another way.

```
IF 0.<=X .AND. X<1.
  Y = X
ELSE IF 2.<=X .AND. X<3.
  Y = 1.
ELSE
```

```
        Y = 0.
    END IF
```

NOTE: Medial blanks may be omitted if desired (e.g. ELSEIF, ENDIF). Also, there are alternate names available for conditional statements. WHENEVER can be used in place of IF, OTHERWISE can be used in place of ELSE, OR WHENEVER can be used in place of ELSE IF, and END OF CONDITIONAL can be used in place of END IF. These alternate names follow the same rules as the previous described names. Make sure to put at least one blank between the short name conditional statements and any following text. The reason for this was described in the previous section (Section 2.2).

2.4 CONTINUE Statement

This statement has the simple form:

```
CONTINUE
```

When labeled, it serves as a junction point in the program, but causes no computation to be performed by its presence. It is merely a dummy or "do-nothing" statement. Its chief use is to indicate the scope of an iteration statement. A statement whose body is blank, but which may have a statement label is treated as a CONTINUE statement.

2.5 Iteration Statement (LOOP Statement)

The iteration statement takes on two forms in GOM. These are defined separately in the following two sub-sections.

2.5.1 Count Controlled Iteration Statement (LOOP FOR)

Figure 2-2 is a program segment which illustrates the use of this statement. A LOOP statement causes the block of statements which follows immediately afterwards to be repeatedly executed, each time varying the value of some variable, until the specified list of values for that variable is exhausted, or until some specified condition is satisfied.

```
        .
        .
        .
        K = 1
        L = 1
        A = D(1,1)
        LOOP FOR I = 1,1,I > 10
            LOOP FOR J = 1,1,J > 10
               IF A >= D(I,J), GO TO ST1
                 K = I
                 L = J
  ST1       END LOOP
        END LOOP
        .
        .
        .
```

Figure 2-2

A program segment to determine the largest element (A) in a 10 row by 10 column array called D and to record its location (K,L). The search proceeds left to right, row by row. If the largest value appears more than once in the array, only the location of the first such element is recorded.

The LOOP statement may take the following form:

LOOP FOR V = E^1, E^2, B

END LOOP marks the statement which defines the "scope" of the LOOP statement, i.e., the block of statements following and not including the LOOP statement, up to the END LOOP statement. END LOOP marks the last executable statement in the block to be repeated. Following the word FOR is the name of the iteration variable (in the illustration V), which may be an individual variable or subscripted array variable of any mode, e.g., V may be an integer or a floating point variable and E^1 and E^2 may be integer or floating point expressions. In fact V, E^1, and E^2 may be of any modes such that V = E^1 and V = V + E^2 are defined. B is a Boolean expression.

The execution of the statement proceeds as follows: The variable V is set equal to the value of E^1. If the value of B = 1B, the scope of the LOOP statement is not executed. If the value of B = 0B, the scope is executed. V is then incremented by the value of E^2, and B is tested again. In general, as soon as B = 1B, the scope is not executed, and the computation proceeds from the statement immediately following the END LOOP statement. Each time B = 0B, the statements in the scope are executed, then V is incremented by E^2, and B is tested again.

Thus, when the iteration is finished and B = 1B, V has the value used during the last computation of the scope, incremented by E^2. The scope will not have been executed for this value of V. (The value of V will be E^1, of course, if B = 1B before the scope is executed at all.) If, at any time, the computation transfers out of the iteration to another part of the program, the value of V will be the current value at the time the transfer was made.

In all cases, every reference to an expression E will involve its current value at the time of reference. Moreover, the variable V may have its value changed at any time during the execution of the scope. Note that the Boolean (B) can be an expression not involving the counter variable (V) in any way. For example, below is a legal LOOP statement in GOM:

```
LOOP FOR ABC=1,1,BD > 5
    SUM = ABC + SUM
    BD = BD + 1
END LOOP
```

This routine will execute properly, but in general a LOOP statement should have as its Boolean (B), an expression dealing with its counter variable (V). The reason for this is that one is more apt to construct an infinite loop by either failing to increment the Boolean (B), or misinterpreting the semantics of such a statement.

More LOOP statement examples:

(1) To evaluate the polynominal

$$c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$$

using the formula

$$(\ldots((c_n x + c_{n-1})x + c_{n-2})x + \ldots + c_1)x + c_0$$

(nested multiplication), we may write the program:

```
INTEGER J,N
Y = 0.
LOOP FOR J = N, -1, J < 0
    Y = X * Y + C(J)
END LOOP
```

(For the meaning of the statement INTEGER J,N, see Section 3.2)

(2) A Newton's Method solution ($x_{k+1} = x_k - f(x_k)/f'(x_k)$)

of the equation f(x) = COS x - x = 0 could be written as a

single statement, using the criterion

$$|f(x)| < e \text{ and } |x_{k} - x_{k+1}| = |f(x_{k})/f'(x_{k})| < e$$

to stop the iteration:

```
 LOOP FOR X=X0,(COS.(X)-X)/(SIN.(X)+1.),
+    .ABS.(COS.(X)-X) .L. EPSILON .AND.
+    .ABS.(COS.(X)-X)/(SIN.(X)+1.)) .L. EPSILON
 END LOOP
```

where X0 is the initial guess. Note: the computation in this case is actually done by the loop increment.

(3) If the value of the iteration variable is to be altered within the scope of the iteration, one may use a zero increment. For example, suppose J is an integer variable, and the scope of the iteration is to be performed for those values of J which are multiples of 2, but not multiples of 5, and at the same time are less than the value of the integer K. One might write the iteration as follows:

```
LOOP FOR J = 2, 0, J.GE.K
     ...
     ...
    J = J + 2
    IF J.E.(J/5)*5, J=J+2
END LOOP
```

(4) A table-look-up procedure using an iteration statement. Suppose that a string of alphabetic (or numeric) characters (i.e., a "sentence") has been stored in C(1), C(2), ..., C(K), where K is the length of the string. Then the first occurrence of a comma could be found as follows:

```
LOOP FOR I=1,1,C(I)=$,$ .OR. I>K
END LOOP
IF I > K, GO TO NOCOMA
```

NOTE: Medial blanks may be omitted if desired (e.g. ENDLOOP). Also, there are other forms available for iteration statements. They are: THROUGH s and ITERATE statements.

The THROUGH s form defines the "scope" to be inclusive of the s statement label-- the last executable statement in the block to be repeated. Note that when using the THROUGH s form, no "end of THROUGH" statement marker is used since the statement label s has already become the last executable line in the repeated block of statements. The THROUGH form follows the same rules as the LOOP form does.

The ITERATE form is much more like the LOOP form because all one has to do is replace LOOP with ITERATE, and replace END LOOP with END OF ITERATION to obtain the ITERATE form of an iteration statement. The ITERATE form follows the same rules as

the LOOP form does.

ALSO NOTE: Make sure to leave at least one blank between LOOP and any other text so that identifiers can be distinguished from the reserved word LOOP.

LAST NOTE: There is another form of the LOOP FOR statement which is not implemented. See Appendix E, Number 2 for details.

2.5.2 Conditional Iteration Statements (LOOP UNTIL, LOOP WHILE)

Only the LOOP and ITERATE forms can be used in extended iteration type statements. There are two forms (only the LOOP verb will be used here):

```
(a)  LOOP UNTIL b
      ...
     END LOOP

(b)  LOOP WHILE b
      ...
     END LOOP
```

where form (a) is the same as

```
          L2   IF b, GO TO L
               ...
               GO TO L2
          L    ...
```

And form (b) is the same as

```
          L2   IF .NOT.b, GO TO L
               ...
               GO TO L2
          L    ...
```

2.6 Nested Iteration Statement

As indicated in Section 2.5, the "scope" of an iteration statement is the block of statements designated for repeated execution:

```
                LOOP FOR V = E¹, E², B
scope ─┌         ...
       └         ...
                END LOOP
```

Some of the statements within the scope of an iteration may themselves be iteration statements. However, if iteration statement b is in the scope of iteration statement a, then b must be entirely within the scope of a. (The same holds true for an iteration statement c in the scope of b in Figure 2-3.)

When iteration statements occur in the scope of other iteration statements (such as with c inside b which is in turn inside a), they are said to be "nested." Likewise, compound conditionals in the scope of other compound conditionals are nested. The "nesting depth" of a statement is the number of iterations and compound conditionals in whose scope it appears. Figure 2-3 shows a valid configuration:

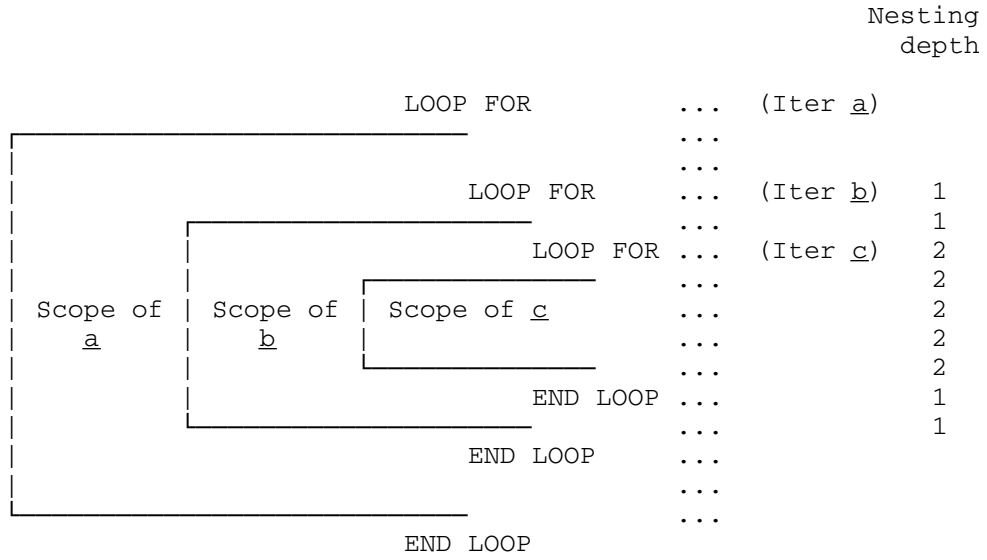


Figure 2-3

In Figure 2-3 iteration a has a nesting depth 0, iteration b has nesting depth 1, and iteration c has nesting depth 2.

A form of nesting which often leads to confusion, although the compiler will accept it, is shown in Figure 2-4. This is the case of a partial overlap in the scopes of an iteration and of a compound conditional. Such overlap should be avoided.

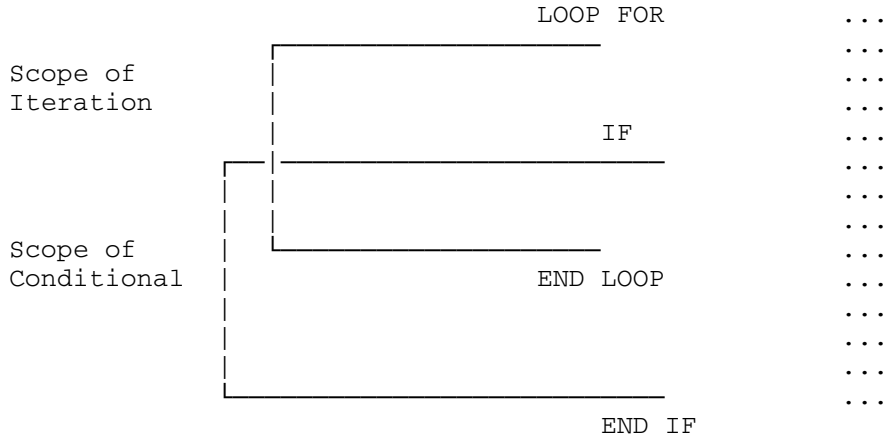


Figure 2-4

There are no restrictions on jumping into or out of the statements in the scope of an iteration.

Automatic indication of nesting depth

At the left side of the listing of the source program, when a GOM program is compiled, between the MTS line number and the internal statement number, there appears a "character" on occasion. This character indicates the current nesting depth of compound conditionals and iterations. If it is zero, it is not printed. The number is actually a single character which goes from 1 to 9 and then A to Z, representing nesting depths of 1 to 35. This is especially useful in cases where either an END OF CONDITIONAL statement or the statement ending a THROUGH loop is omitted.

2.7 End of Program Statement

This executable statement has the form:

END OF PROGRAM

This statement must be physically the last statement in the program (i.e., the last card of the program being compiled). It may also be the last step in the sequence of computation. Execution of this statement will transfer control to the operating system in which the translated program is embedded. An alternate way to terminating a program-- i.e., returning to the operating system --is to attempt to execute an input statement which has no "RC->v" construct when the data has been exhausted. (See Section 2.9 for the return code construct.)

2.8 Input/Output Statements

This section discusses input and output statements that use a format specification to control conversions and formatting. The "Simple I/O" statements are described in Section 2.9. There are no statements for unformatted I/O since it is assumed that users will call the I/O subroutines (such as SCARDS, READ, etc.) directly if they wish to do raw I/O.

These formatted I/O statements call on IOH360 to do the formatting and conversions; consequently, the formats specified are IOH formats, not FORTRAN formats. See MTS Volume 5 for a description of the IOH format language.

The statements are:

```
PRINT FORMAT   f [,RC->v][,list]
PUNCH FORMAT   f [,RC->v][,list]
READ FORMAT    f [,RC->v][,list]
LOOK AT FORMAT f [,RC->v][,list]
```

```
READ FROM   i, f [,RC->v][,list]
WRITE ON    o, f [,RC->v][,list]
```

where

f is a character constant or a character variable, that contains the format.

v is an integer non-subscripted variable in which the return code will be returned. If at any time during processing the I/O list the return code becomes nonzero, the remainder of the I/O processing will be skipped. On an input statement, if the return code is not requested via the "RC->v" construct, a nonzero return code (such as due to end-of-file) will cause termination of the program.

i is an input specification and consists of one of the following:

```
SCARDS
GUSER
UNIT u
FDUB p
```

where u is an integer-valued expression giving a unit number and p specifies a FDUB pointer.

o is an output specification and consists of one of the following:

```
SPRINT
SPUNCH
SERCOM
UNIT u
FDUB p
```

where u and p are as for input specifications (above).

list is an input/output list of elements separated by commas. Elements may be:

(1) Single variable names, or array names with subscripts
(2) Blocks of the form A(i)...A(j) (See Section 1.12)

In addition to these, when list designates an output list, the elements can be:

(3) constants
(4) expressions

NOTE: There is another option in the list parameter that allows elements to be "iteration elements." However, this form is not implemented yet. It is described in Appendix E, Number 3.

Example of an output list: AB, D, 2.5, MTX(1)...MTX(N), P(14),

J(I,K). Example of a list which may be used either for input or output: A, B, K(3), J(24*I-L), A(K+1)...A(L*2).

Here are some examples of Input/Output statements:

PRINT FORMAT: produces its output on logical I/O unit SPRINT

PUNCH FORMAT: produces its output on logical I/O unit SPUNCH

READ FORMAT: reads its input from logical I/O unit SCARDS

LOOK AT FORMAT: also reads from SCARDS, but does so "without going past the record". Hence the next time a READ FORMAT or LOOK AT FORMAT statement is processed, the same input record will again be transmitted.

Warning: If more than one record is specified (via one or more slashes in the format), each instruction to get a new record merely causes the same record to be rescanned.

2.9 Simplified Input/Output Statements

One type of simple input statement is the READ DATA statement which may have the following forms:

```
READ DATA
READ DATA FROM i
```

where

i is an input specification as given in Section 2.8.

If a return code specification is given, it must follow the input specification, e.g.:

```
READ DATA FROM UNIT 4, RC->BFX
```

The values to be read and the variables which are to receive those values must be on the input records in a sequence of fields of the form:

$$V^1 = n^1, \quad V^2 = n^2, \quad V^3 = n^3, \ldots, \quad Vk = nk*$$

The V^1, ..., Vk are the variable names and n^1, ..., nk are the corresponding values. Reading is continued from record to record until the terminating mark (*) is encountered. Fields cannot be divided between records, so the last character in a record not terminated by asterisk would normally be a comma. However, as a convenience, the end of the record is treated as an implied comma and hence this final comma may be omitted. The variable names may designate single variables or elements of linear and two dimensional arrays. The subscripts on the array variables must be integer constants. The values may be floating

point, integer, hex, Boolean, or character with the forms described in constants of corresponding mode (see Section 1.1), except that the only qualifier that may be used is "@X" for hexadecimal numbers.

For convenience in entering values of array elements it is possible to designate only one variable name and have successive numbers, written without names, interpreted as the consecutive values of the array, i.e.,

V(j) = n^1, n^2, n^3, ..., nk

would be the same as

V(j) = n^1, V(j+1) = n^2, ..., V(j+k-1) = nk

For 2-dimensional arrays successive numbers will be entered in succeeding columns of the designated row until the row - as determined from the current value of the dimension vector - is filled, and then the next row will be started. (The dimension vector is discussed in Section 3.3)

Adjacent commas (,,) are skipped. Blanks are ignored throughout except between character delimiters.

As an example illustrating many of the features described herein, consider the input data:

```
X1=1.2, Y1=-6.8, INDEX=4, A(4) = 3.1, -10.93,
12.6, MATRIX (2,1) = 25E-2, 1.8E-10, 3.14E-8,
STRING(1) = " END OF PROBLEM " *
```

Character strings may extend over more than one data record; column 1 of the next record is considered to immediately follow the last column of the previous record. As with character constants, the delimiter character may be represented within the input string by writing two of them with no space between them. However, this pair of delimiters must appear together on one record; they cannot be split between two records.

A second type of simple input statement is READ AND PRINT DATA, which may be written:

```
READ AND PRINT DATA
READ AND PRINT DATA FROM i
READ AND PRINT DATA ON o
READ AND PRINT DATA FROM i, ON o
```

where

i is an input specification as given Section 2.8.

o is an output specification as given in Section 2.8.

This has the same effect as READ DATA, except that after a

card is read it is also printed. The exact image of the input card is printed. The default input specification is SCARDS and the default output specification is SPRINT. If a return code specification is given, it must follow all I/O specifications.

Important Note

There is a way to intercept an EOF (end-of-file) condition for Simple I/O (READ DATA and READ AND PRINT DATA) input. If, for example, the statement is of the form:

```
READ DATA RC->intvar
```

where intvar is an integer variable, then when an EOF is sensed on SCARDS, a return code of 4 is returned and stored in intvar, and execution proceeds with the next statement. If the READ DATA terminated without an EOF occurring, then a return code of 0 (zero) is returned. If the "RC->var" construct is not provided, then an EOF will cause termination of the program. READ AND PRINT DATA behaves similarly.

One type of simple output statement is PRINT COMMENT, which may have the forms:

```
PRINT COMMENT string
PRINT COMMENT ON o, string
```

where

string is either a character string constant or the name of a character variable or array;

o is an output specification as given in Section 2.8.

Two string delimiters are allowed for PRINT COMMENT statements. They are the double-quote (") and the dollar-sign ($). The delimiter that begins a string must be the one that ends it, and must be doubled if it is to appear inside it. Notice that the same rules apply for Character Constants (described in Section 1.1.3) and PRINT COMMENT statements. Examples:

```
PRINT COMMENT $Hello there$
PRINT COMMENT "Hello there again"
PRINT COMMENT $"$
PRINT COMMENT "$"
PRINT COMMENT $"$$"$
PRINT COMMENT "$""$"
PRINT COMMENT "Geoff says, ""That's $10 please."""
```

Another type of simple output statement is PRINT RESULTS, which may be written:

```
PRINT RESULTS list
PRINT RESULTS ON o, list
```

where

list is the list of variables or blocks to be printed (also see below)

o is an output specification as given in Section 2.8.

Since the default destination is SPRINT, if o is given as SPRINT, both forms given above do the same thing.

The list designates a list of variable names, block designations, or expressions, but not iteration elements. The printed output is analogous to the input for READ DATA in that values of variables are preceded by the appropriate variable name and an equal sign; e.g., "X = -12.4". Blocks are labeled as such and printed using a block format. Elements of three and higher dimensional arrays will be labeled with the equivalent linear subscript. If dummy variables (in a function definition or expression) or elements in dynamic records are included in the list the specific values assigned to such variables or expressions during execution will not be labeled but simply preceded by three dots (...). An example statement is:

```
PRINT RESULTS X1, Y1, Z(1)...Z(N+1), MTX(1,1)...MTX(M,N)
```

Two other forms of the PRINT RESULTS statement are:

```
PRINT BCD RESULTS list
PRINT HEX RESULTS list
```

These statements have the same effect as "PRINT RESULTS list" except that the value for each list element is treated as character (or hex) information, and printed accordingly.

2.10 ALLOCATE and RELEASE

Two statements are available to allocate and release space dynamically.

The form of the ALLOCATE statement is one of:

```
(a)   ALLOCATE drcname
(b)   ALLOCATE drcname->ptrvar
(c)   ALLOCATE (intexp)->ptrvar
```

where "drcname" is the name of a dynamic record, "ptrvar" is the name of a scalar pointer variable, and "intexp" is an integer expression. Form (a) can be used if there is a "using" in effect for the dynamic record "drcname" (see the USING POINTER statement, Section 3.10). In this case, a call to GETSPACE is made for a region whose size is the size of the dynamic record, and the location returned is stored in the implicit pointer variable. Form (b) can be used anytime, and is the same as (a), except that the location is stored in the specified pointer

variable "ptrvar". Form (c) is used to allocate a region of the size (in bytes) specified by the integer expression "intexp".

To all three forms the expression ",RC->intvar" may be appended, in which case GETSPACE is called with a conditional call, and the return code is placed in "intvar", which must be a scalar integer variable. If this expression is not appended, the GETSPACE call is an unconditional one.

The release statement is of the form:

RELEASE ptrvar

and causes a call to FREESPAC to free the space pointed to by the pointer variable "ptrvar".

As an example showing allocating storage and use of the .SIZE. operator (described in Section 3.10), consider the following:

```
DYNAMIC RECORD (PAGE) LINK,REST(4092)
POINTER LINK    ;* link to next page allocated
CHARACTER REST  ;* remainder of page to be subdivided

DYNAMIC RECORD (TRIPLE) LPTR,LTYP,OP,RPTR,RTYP
INTEGER LPTR,LTYP,OP,RPTR,RTYP

POINTER P,ENDP
ALLOCATE (PAGE)->P          ;* P starts at front of region
ENDP = P + .SIZE.PAGE - 1   ;* ENDP points to end
P:LINK = 0@P                ;* zero link
P = P + 4                   ;* position pointer
USING POINTER P, FOR TRIPLE
LPTR = ...                  ;* make first entry
...
P = P + .SIZE.TRIPLE        ;* position for next entry
IF P+ .SIZE.TRIPLE > ENDP, GO TO ALLOCATE_MORE
...
```

3. Declarations (Non-executable statements)

Declarations are non-executable statements, and, except for function declarations, they may occur anywhere in the program. Their purpose is to furnish information to the translator program or to the reader of the program. Declarations may have statement labels, but names in the label field are ignored by the translator, and may not be referred to in other statements. (See Appendix A for allowable abbreviations.)

3.1 Comments

A comment "declaration" consists of any string of characters acceptable to the computer. This statement is completely ignored by the translator and furnishes information to the reader of the program. Every statement of the comment must have an "*" in column 1. An input line which is entirely blank is treated as a comment. A comment line may occur anywhere between statements; it may not occur between an input line and its continuation lines. A comment is terminated by the end of the input line in which it begins. This is unlike other statements, which are terminated by either end-of-record or a semicolon. Examples:

```
* This is a comment
          I = 24    ;* This is also a comment
```

3.2 Mode Declaration

All variables and function values are assumed to have the normal mode unless declared otherwise. The normal mode is FLOATING POINT by default. Any of the modes may be specified as the normal mode by writing the following declaration:

NORMAL MODE IS m

where m is one of the following phrases: INTEGER, BOOLEAN, STATEMENT LABEL, FUNCTION NAME, FLOATING POINT, CHARACTER, SHORT INTEGER, BYTE INTEGER, LONG INTEGER, or POINTER. (VARYING CHARACTER and LONG FLOATING POINT are also recognized, but no use may yet be made for variables of these modes. That is, they can be declared, but they cannot be used in any expressions, functions, or statements. The reason for this is that they are not fully implemented yet. See Appendix E, Number 1 for details.) Only one such declaration may appear in a program and it is in effect for the whole program, no matter where it occurs in the program. If a variable or function value is to have a mode different from the normal mode then its mode must be declared in a declaration of the form:

m varlist

where m is as defined above, and varlist is a list of variables or function names whose values are to be of mode m. For example,

BOOLEAN P, Q, DIGIT., TRUE

This example declares that P, Q, and TRUE are variables of Boolean mode, and that the function DIGIT. returns a value of Boolean mode.

3.2.1 Automatic mode assignment

All constants are automatically assigned modes by the translator (see Section 1.1). Other automatic assignments of modes are:

(a) A name appearing in the statement label field is assigned statement label mode (see Section 1.3).

(b) A function name constant is assigned function name mode (see Section 4.1).

(c) A vector appearing as the dimension vector of some array in a dimension declaration is assigned integer mode (see Section 3.3).

(d) A vector which is preset by a vector values or constant declaration is assigned a mode consistent with the first assigned value (see Section 3.6).

3.3 The DIMENSION Declaration

In order to be sure that that consecutive elements of a vector or array are stored in order in the computer, it is necessary to declare the ranges of the subscripts to be used in referring to elements of the array. If only one subscript is used (i.e., one is referring to the elements of a vector), it is understood that the lowest value a subscript may have is zero, so one declares the highest value the subscript may assume at any time during the computation. For example:

DIMENSION V(50)

In this case, consecutive storage locations will be assigned for 51 elements, e.g., V(0), V(1), V(2), ..., V(50). Negative subscripts may not be used with vectors. If the name V is used without any subscript, it is exactly the same as if V(0) had been written.

For arrays with two or more dimensions (i.e., each reference to an element requires two or more subscripts), one declares the range of each subscript. Thus, if the array B is two-dimensional, and if the first subscript used with B is

expected to take on values between -5 and 10 inclusive while the second subscript will vary between 1 and 15 inclusive, one would write:

```
DIMENSION B((-5...10),(1...15))
```

If 1 is the lower bound for a subscript, the one, the three dots, and the parentheses may be omitted, so that the last declaration above would more likely be written:

```
DIMENSION B((-5...10),15)
```

In this case, storage would be allocated to B(-5,1), B(-5,2), ..., B(-5,15), B(-4,1), ..., B(10,15). There are 10 - (-5) + 1 = 16 rows and 15 columns in this array, so 240 storage locations would be assigned to the array B. (16 X 15 = 240; the "first" element of the array has linear subscript 0 -- see below.)

Each array is always considered to have storage assigned to it as if it were a vector, regardless of the 2-dimensional (or higher-dimensional) structure declared for it as described here. References to elements of an array may therefore be made by using the appropriate number of subscripts, or by using a single subscript. The "first" element of any array (single or multidimensional) is automatically set to correspond to the single subscript 0, so that in the example above, B(-5,1) could also be referred to as B if desired. The single subscript is often called the "linear subscript", and the relationship between the subscripts i and j in B(i,j) and the corresponding linear subscript r in B(r) is (for two-dimensional arrays):

$$r = n(i - 1) + (j - 1) + b$$

where n is the number of columns and b is chosen so that the "first" element has linear subscript r = 0. For the example above, the first element is B(-5,1). Substituting, we have

$$0 = 15(-5 -1) + (1 - 1) + b$$
$$b = 90$$

For the array B, then, the relationship is

$$r = 15(i - 1) + (j - 1) + 90$$

Declarations may occur anywhere in the program in any order, and they may be combined into a single statement, so a typical declaration might be

```
DIMENSION P(20), Q(10,20), R((-5...10),10,20), B((-5...10),15)
```

Elements of arrays are assigned storage in the order determined by varying the last subscript first, then the next to last, etc., as indicated for the array B above. Thus, if one writes B(0,12), ..., B(1,3) in an output list, for example, with B dimensioned as above, then

```
B(0,12), B(0,13), B(0,14), B(0,15), B(1,1), B(1,2), B(1,3)
```

would be printed, because the declared ranges of the subscripts would be used. (These are kept for use during execution of the program.) In this way it will correctly happen that the successor to B(0,15) is B(1,1), rather than B(0,16).

A dimension function (see later) may be specified by putting "F=xxx" as the last item in the declaration list. For example:

```
DIMENSION ARR(3,4,F=DF.)
```

If the user wishes to construct his own dimension vector (mostly because he wants to twiddle its contents-- there's no SETDIM yet), then the dimension information in the declaration should be two items, the first of which is the total actual size of the array and the second is "D=dvarr" where dvarr is the name of the dimension vector. This must be an unsubscripted name; you may not put more than one dimension vector into an array. For example:

```
DIMENSION ARR(12,D=DVEC)
```

Array references are as you would expect:

```
ARR(I,J)
```

Note that arrays are stored row-wise in GOM. For a character array, substring notation may be mixed with multi-dimensioning, but since a substring represents a contiguous string of characters, only the right-most subscript can be a substring. For example:

```
ARR(2,1...3)
```

Currently, if the array has a dimension function, it may NOT use substrings.

In order to allow a GOM external function to use an array passed to it by a GOM program, the dimension vector location must be passed. Since (unlike the 7090) it is not possible to pass two parameters in one parameter, it is up to the user to explicitly pass the dimension vector as a parameter. To allow access to the dimension vector for the normal (implicit dimension vector) case, the .DIMVEC. unary operator is supplied. (Users must not use this operator to change the dimension vector because it won't necessarily work-- in the implicit case, various optimizations may be made at compile time based on the declaration.) An example for passing:

```
    ADD.(IARR, .DIMVEC.IARR, JARR, .DIMVEC.JARR)
         ... ... ...
*
    INTERNAL FUNCTION ADD.(ARR1, DARR1, ARR2, DARR2)
    INTEGER ARR1(1,D=DARR1), ARR2(1,D=DARR2)
    LOOP FOR I=2,1,I>4
       LOOP FOR J=2,1,J>9
          ARR1(I,J) = ARR1(I,J) + ARR2(I,J)
       END LOOP
    END LOOP
    FUNCTION RETURN
    END OF FUNCTION
```

RESTRICTION: READ DATA and PRINT RESULTS currently cannot handle multiple subscripts; you must use the equivalent linear subscript for READ DATA, and PRINT RESULTS will report values in terms of a linear subscript.

DIMENSION FUNCTIONS: Normally, GOM generates inline instructions to convert the multiple subscripts given to the equivalent linear subscript according to the standard formulas. If the user wishes to have arrays stored according to some other scheme, he may have GOM call a function he supplies to do this conversion. The function may be an internal or external function, and is called with the location of the dimension vector as the second argument, and the subscripts given on this reference as the third and following arguments. The function should return a value which is the linear subscript to be used in accessing an array.

DIMENSION VECTORS: The dimension vector for an array is a one-dimensional array that contains the information required to convert the multiple subscripts to a linear subscript or the converse. The information contained in it is redundant so as to allow minimal code to be generated. It consists of 3 + 3*N fullwords, where N is the number of dimensions, as follows:

First word	zero or address of dimension function
Second word	number of dimensions (but see below)
Third word	size of the array (maximum linear subscript)
Fourth word	base (the linear subscript that corresponds to A(1,1,1,...))
Fifth and up	N-1 words containing the spans (max-min+1) of the dimensions, from left to right, excluding the first.
Last	2*N words containing a 1 word lower bound followed by a 1 word upper bound, for each dimension, left to right.

For example, instead of

```
DIMENSION ARR(3,4)
```

one could say

```
DIMENSION ARR(12,D=DVEC)
VECTOR VALUES DVEC=0,2,12,1,4,1,3,1,4
```

If there was a dimension function DF, then instead of

```
DIMENSION ARR(3,4,F=DF)
```

it could be specified with:

```
DIMENSION ARR(12,D=DVEC)
VECTOR VALUES DVEC=DF.,2,12,1,4,1,3,1,4
```

and with the following definition it should do the same (albeit slowly) as not having a dimension function at all:

```
INTERNAL FUNCTION DF.(ARR,DV,I,J)
DIMENSION ARR(1), DV(8)
L = (I-1)*DV(4) + (J-1) + DV(3)
FUNCTION RETURN L
END OF FUNCTION
```

SECOND WORD: This word is split into 2 parts, the first which contains the halfword size of the element (e.g., 4 for integers), and the second which holds the halfword number of dimensions. Since the code GOM generates does not look at this word of the dimension vector at all, this should be user transparent. Only those writing their own dimension functions need watch out.

3.3.1 Including dimension information in other declarations

The word DIMENSION may also be replaced by any of the following: PROGRAM COMMON, INTEGER, BOOLEAN, FLOATING POINT, FUNCTION NAME, STATEMENT LABEL, CHARACTER, SHORT INTEGER, BYTE INTEGER, LONG INTEGER, POINTER, or DYNAMIC RECORD, with the effect determined by the specific declaration used. In any of these other cases, dimension information is not required for a name on the list. If given, as described above, the dimensioning is in addition to the declared effect. (For PROGRAM COMMON see Section 3.4. For DYNAMIC RECORD see Section 3.9.)

Example:

```
INTEGER A(10), N, P, Q(30*3)
```

3.3.2 Automatic dimensioning

Dimensioning is automatic in two situations:

(a) When a constant statement label vector, say L, is used

(see Section 1.3) and n is the highest subscript used on L in the statement label field, then n + 1 array elements are reserved for L. Of course, L may also appear in a dimension declaration, in which case the highest subscript is used.

(b) If part or all of a vector is set by a VECTOR VALUES or CONSTANT declaration (see Section 3.6), the vector need not appear in a dimension statement unless the maximum subscript implied by the initial values is not sufficiently high.

NOTE: There are two sub-sections for Section 3.3 which are not currently implemented. They deal with multiple dimensioning and modifying the declared range of array subscripts. See Appendix E, Numbers 4 and 5 for details.

ALSO NOTE: Another declaration section dealing with the EQUIVALENCE declaration form has yet to be implemented. It too is described in Appendix E, Number 6.

3.4 PROGRAM COMMON Declaration

This declaration has the form

```
PROGRAM COMMON a,b,c, ...
```

or

```
PROGRAM COMMON (name) a,b,c, ...
```

where "a,b,c, ..." is a list of one or more variables or array names. The first form assigns the variables to blank common; the second form assigns them to named common, where name is the (eight character maximum) name of the common section.

Examples:

```
PROGRAM COMMON (DATA1) A,B(10),C
PROGRAM COMMON MATRIX, X, Y1, BC
```

In these declarations, the arrays and individual variables listed after the words PROGRAM COMMON are non-overlapping in storage and are assigned (in the order in which they occur, from left to right) to a special section of storage which is separate from the usual storage of variables and arrays, and is in fact, not in the program at all. Dimension information may be included, if desired.

One use of this statement provides for several sections of a program to refer to variables and arrays by the same names, while being translated and checked out separately. A program divided up this way would have the form of a main program and several external function programs, with the main program being

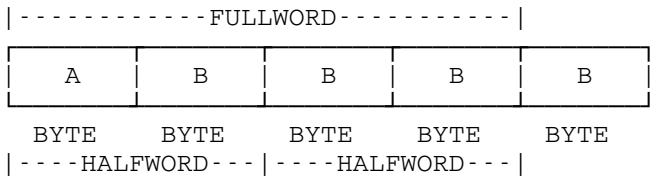
used primarily to call on each of the external functions in turn. Although variables and arrays to be used jointly by several external functions can be communicated as arguments to the functions, assigning them to PROGRAM COMMON makes them available to all sections which declare them as such.

The storage reserved for common sections, both named and blank, is reserved separately, and is not a part of any program (main program or external function program). Every program which refers to a variable in program common must have the same address assignment for that variable. This is usually done by including identical PROGRAM COMMON and DIMENSION declarations in all the programs which refer to program common.

Note that this declaration forces an ordering on the assignment of variables that will override the normal boundary alignment provided for variable storage. Each common section is allocated to begin on a doubleword boundary. Thus,

```
PROGRAM COMMON A,B
BYTE INTEGER A
INTEGER B
```

will force the four-byte integer field B, which would normally be allocated to start on a fullword boundary, to start on an odd byte boundary (thus, a packed representation of variables is used). Here is how these variables appear in storage (Note--this storage representation begins on a doubleword boundary):



PROGRAM COMMON declarations are cumulative for each common section defined. If another such declaration occurs in a program, the arrays and variables listed therein are considered appended to the previous list of PROGRAM COMMON arrays and variables. The amount of storage actually reserved for each common section is determined solely by the maximum amount used by all modules that refer to that section.

3.5 GLOBAL AREA Declaration

In order to provide a place for the repository of global information in a reentrant environment (where PROGRAM COMMON cannot be used), another entity, the Global Area, is available in GOM. A global area is defined with the GLOBAL AREA statement:

GLOBAL AREA list

or

GLOBAL AREA (areaname) list

This declaration is syntactically the same as the PROGRAM COMMON construct. The areaname is restricted to a name of four or fewer characters (eight or fewer if Linkage=CLSMTS is specified, see sub-section 4.3.3); if it is longer, its external form will be truncated and a warning message will be given. Each global area with a different name is a separate, independent storage area. Variables in a global area may not be preinitialized with the VECTOR VALUES statement.

3.6 Presetting Variables

Any scalar, vector, or portion of a vector (or array when considered as a vector, i.e., using linear suscripts) may be preset by declarations of one of the following two forms:

VECTOR VALUES A(n) = C^0, C^1, ..., Cr (vectors)
VECTOR VALUES A = C (scalars)

Here n is an integer constant, and A(n) may be written simply as A if n = 0. The entries C^0, ..., Cr may be any constants (not necessarily all of the same mode). Array A, starting with element A(n), is preset with the constants in the list, starting with C^0. A is automatically set to have the same mode as C^0; and A is automatically dimensioned large enough for the constants. If there are s constants d^0, ..., d(s-1) in the list, then the elements A(n), A(n + 1), ..., A(n + s - 1) are preset (in order) to the values d^0, ..., d(s-1). A is automatically set to have the same mode as d^0; and A is automatically given a storage reservation of n + s locations, which is the same as writing DIMENSION A(n + s - 1).

A may appear in a mode declaration as well, provided it is consistent with the mode of C^0. If A appears in other VECTOR VALUES or DIMENSION declarations the maximum length given or implied for A is used for storage assignment.

NOTE: There is an alternate form for the VECTOR VALUES statement which is not currently implemented. This form is described in Appendix E, Number 7.

3.7 Parameterization of Constants

Just as the VECTOR VALUES statement specifies the initialization of variables, the CONSTANT statement allows one to supply a name for a constant. The form of the statement is the same as for VECTOR VALUES, except for the statement name:

```
                CONSTANT A = C
        or      CONSTANT A(n) = C⁰, C¹, ..., Cr
```

The resultant name A, since it is just a name for a constant, can be used anywhere a constant may be used. For example,

```
                CONSTANT TSIZ=57
                DIMENSION TABLE1(TSIZ), TABLE2(TSIZ)
```

Important Note

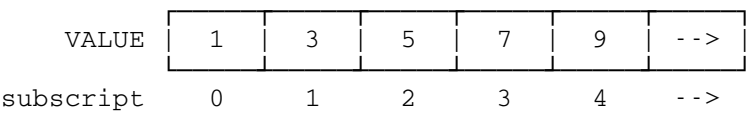
Unlike the VECTOR VALUES declaration, the CONSTANT declaration is effective at the point it occurs in the program. Thus the order of the statements in the above example is required.

Besides providing a parameterization for constants, the CONSTANT statement can be used for tables whose values are indeed constant. For example:

```
                CONSTANT ODDS = 1,3,5,7,9,11,13,15,17
```

Such names can be subscripted, just as if they were variables. Here is how ODDS looks graphically using the above example:

CONSTANT ODDS



Note that the mode of a constant (i.e., INTEGER, CHARACTER, etc.) is determined by the mode of the first constant value.

3.8 Format Variable Declaration

This declaration has the form

```
                FORMAT VARIABLE list
```

where list is a list of unsubscripted variable names. If this declaration is embedded within a function definition (see Section 4.5), then none of the names in list may be dummy variables or be used in dynamic records. All names that will be used as format variables in formats must be declared to be format variables in this way. This declaration does not imply anything at all about arithmetic or Boolean mode, or about dimension. There may be any number of such declarations, anywhere in the program. Format variables may be used in formats only with the READ FROM and WRITE ON statements.

3.9 Dynamic Record

A dynamic record declaration defines a structure which consists of a set of names and their spatial relationship to each other. It does not allocate any storage and does not provide an instance of this structure. In various other languages this would be called a one-level structure, a record, or a DSECT.

This is similar to the PROGRAM COMMON declaration, except that the PROGRAM COMMON declaration does cause the allocation (external to the program) of exactly one instance of the common section being defined. Because of this similarity, the syntax of the DYNAMIC RECORD declaration is the same as the PROGRAM COMMON declaration, except, of course, that the statement name differs. The form is:

```
DYNAMIC RECORD (drcname) list-of-elements
```

Examples are:

```
DYNAMIC RECORD (BLOCK) I,VAL(10),J
DYNAMIC RECORD (TRIPLE) LPTR,LTYP,OP,RPTR,RTYP
```

At any given time, zero or more instances of a given dynamic record may exist. Therefore, in order to refer to a given variable in a particular instance of a dynamic record, not only the variable must be given, but the location of the particular instance wanted must also be specified. There are two ways to provide this location. It can be done explicitly when the variable is mentioned by using the ":" operator (see next paragraph), or it may be done implicitly by setting up a "using" which tells the compiler where to find the location. This is described in the next section. Generally, more efficient code is produced when a "using" is in effect, particularly when compared with a statement that has more than one ":" operator in it.

The ":" operator is used to refer to variables in dynamic records. The left operand must be of pointer mode, and the right operand must be a variable (possibly subscripted) which is defined in a DYNAMIC RECORD statement. The pointer is assumed to contain the location of the first element in the dynamic record. Examples are:

```
X = P:J
P:LPTR = AB
P(I):VAL(J) = 4.3
```

The simplest way to produce an instance of a DYNAMIC RECORD is to map it onto something already allocated storage in the program. For example, given

```
DYNAMIC RECORD (BLOCK) I,VAL(10),J
```

then if

```
DIMENSION AREA(12)
```

```
POINTER P
P = .LOC.AREA
```

things are set up so that specifying

```
P:I
```

refers to an I in a BLOCK mapped onto AREA, and hence actually refers to the first word in AREA. This is a static usage of a dynamic record.

A more common usage is to use the ALLOCATE statement to dynamically obtain storage to use for an instance of a dynamic record. This manner of usage and the ALLOCATE statement are discussed in Section 2.10.

The .SIZE. unary operator is available to obtain the length (in bytes) of a dynamic record. The operand must be a dynamic record name, and the mode of the result is integer.

3.10 USING Statements

There are two statements available to set up and remove an implicit "using" for a dynamic record. These are declarations in that they produce no object code, but are merely directives to the compiler. However, unlike other declarations, they are not global, but take effect when they are encountered by the compiler. Thus they behave like the USING and DROP statements in assembly language. The forms are:

```
USING POINTER ptrvar, FOR drcname
STOP USING POINTER ptrvar, FOR drcname
```

where ptrvar is a scalar pointer variable, and drcname is a dynamic record name. The USING POINTER statement tells the compiler that from that point on, it can assume that the given pointer variable has the location of an instance of the specified dynamic record. Then if the name of a variable in that dynamic record appears in the program unqualified by the ":" operator, the compiler will use the pointer variable specified by the USING POINTER statement. Thus, given the declaration

```
DYNAMIC RECORD (BLOCK) I,VAL(10),J
```

instead of specifying

```
P:J = P:J + 1     (explicate method)
```

one could instead say

```
USING POINTER P, FOR BLOCK
J = J + 1         (abstract method)
```

If a pointer variable is explicitly specified by use of the ":" operator, the pointer specified is used, and any using in effect is ignored.

A using for a given dynamic record remains in effect until either a STOP USING POINTER statement for that dynamic record is given, or until a new using for that dynamic record is specified. (A little thought will show that while many pointers could be pointing at a given dynamic record, the compiler cannot use them all to address a given variable, and hence it uses only the most recent one specified, for each dynamic record.)

4. Functions

The name of a function consists of one to 24 letters, underscores, or digits, but the name must be followed by a period (.) so that the translator can recognize it as a function name. The first character of a function name must be a letter. A function name given explicitly in this form will be called a function name constant. If the function is single-valued, then the value of the function (that is, the result of making a call on the function) is represented by following the function name with the proper number of arguments separated by commas and enclosed in parentheses. Thus, ADD51., COS., POLY., and FUNCT3. are function names, while ADD51.(X,Y3,ADD.), POLY.(N,VJ,7) and COS.(X) are function values.

If the function referred to by the function name constant is not defined (as an INTERNAL FUNCTION) in the same source module as the reference, then function is assumed externally defined. The external name presented to the system loader will consist of the first eight characters of the function name, if it is greater than eight. Hence, such names must be unique in the first eight characters, and it is recommended that they not exceed eight characters.

4.1 Function Name Constants, Variables, and Expressions

(a) The notation for function evaluations, i.e., function references, always requires a period after the name of the function.

(b) Function name constants are the names which designate entry points in actual definitions, e.g., SIN., COS., F. --assuming F. is defined as either an internal or external function. Function name constants are never subscripted and never appear without a period; they may not stand alone on the left side of an assignment statement.

(c) Function name variables, i.e., variables of function name mode where the mode is either declared or is implicit from a VECTOR VALUES declaration, should not have a period when used as variables. Use on the left side of an assignment statement or as an argument of a function are examples. The following two restrictions apply to the use of function name variables when they are used as function names in function evaluations.

(1) Single function name variables, i.e., variables not normally considered to be an element of an array, must be written with a zero subscript preceding the period. For example, if G is a single function name variable, its use in a function evaluation would appear G(0).(A,B)

where A and B are the arguments. As with any single variable used only with a zero subscript, it does not need to be dimensioned.

(2) Such function evaluations may not be embedded in a larger expression. They may appear by themselves or as the right side of an assignment statement.

Thus if G and H are function name variables, then

```
   H(0).(A,B,T)
or T = G(0).(A,B)
```

would be acceptable statements, but not

```
T = G(0).(A,B) + C*D
```

| The result of such a function call is assumed to be
| of normal mode. If a statement including such a call
| in a larger expression is used, then an infinite loop
| may occur when trying to run the object program. In order to protect yourself against this problem, make sure to specify a local time limit when running the object program.

(d) A single constant, variable, or function value of function name mode is an expression of function name mode. The only operations legal for operands of function name mode, besides calling the function, are substitution, and the .E. and .NE. comparatives (or their equivalents). There are no automatic conversions from function name mode to any other mode.

4.2 Function Call Statements

Normally, the value of a function will occur as part of an expression as in the statement:

```
Z = COS.(X)/SIN.(X + 2.)
```

Certain functions, however, may stand alone as separate statements. For example, a procedure to sort a list could be called by:

```
(a)     EXECUTE LSORT.(ARRAY,MAP,N)

        or alternatively as:

(b)     LSORT.(ARRAY, MAP, N)
```

Here ARRAY, MAP, and N are the "arguments" of the function (subroutine) LSORT. A function called as in (a) or (b) above

need not be followed by a list of arguments.

The return code produced by the subroutine that was called may be obtained by means of the "->" operator. The left operand of this operator must be the function call, and the right operand must be an integer variable. The resulting return code from the function call specified by the left operand is stored in the variable specified by the right operand. This operator may be used on either a stand alone call or when the function is evaluated as part of an expression. For example,

```
FUNCT1.(A,B)->I
Y = FUNCT2.(X,Z)->J
```

Normally, calls to subroutines are set up as standard S-type calls, wherein the addresses of the arguments are provided in storage in a list whose location is supplied in register 1 when the call is performed. On occasion, it may be necessary to provide an R-type calling sequence, wherein the parameters are provided in one or more registers, and results are returned by the subroutine in various registers.

For an R-type call, arguments that are to be input to the subroutine being called are specified as "Rn=value" in the argument list, where "n" is the number of the register. "n" may not specify registers 9 through 15. "value" must be of a mode that is reasonable in a general register; that is, four bytes in size and not floating point, which means the mode must be INTEGER, BOOLEAN, STATEMENT LABEL, FUNCTION NAME, or POINTER. (Of course, using the .AS. operator or the "@" qualifier, anything (or at least 32 bits of it) may be forced there.) Results returned by the subroutine in registers other than register 0 (which is the normal register used to return a value) are indicated by specifying output arguments in the argument list following the input arguments. Output arguments are specified as "Rn->variable", where "n" is the number of the register. "n" may not specify registers 9 through 14. "variable" must be a scalar variable with the same mode restrictions as for "value" on input arguments.

Examples of R-type calls:

```
INTEGER SIZE,RC
POINTER LOCAREA
GETSPACE.(R0=3,R1=SIZE,R1->LOCAREA)->RC

POINTER FDUB,GETFD.
VECTOR VALUES FILENAME = "INFILE "
FDUB = GETFD.(R1=.LOC.FILENAME)

GDINFO.(R0="SCAR"@I,R1="DS  "@I,R1->LOCAREA)->RC
```

The second type of function call that GOM programs can issue are SVC based instructions. Since a supervisor call is really just another sort of a subroutine call, the statement needed to issue one is the same as for producing an R-type

subroutine call, except for the statement verb. The form is:

```
SUPERVISOR CALL n.(arguments)
```

where arguments is a typical R-type call list. n must be either an integer constant or an integer CONSTANT. Examples are:

```
SUPERVISOR CALL 28.(R0->A,R1->B,R2->C,R3->D)

CONSTANT DORMNT=4
SUPERVISOR CALL DORMNT.

CONSTANT WAYT=35
SUPERVISOR CALL WAYT.(R0=.LOC.A) ;* Won't wait
```

The allowable "register" names for an R-type call allow CC for handling the condition code. Note that only SUPERVISOR CALL type function calls may use CC.

```
CONSTANT PROTON=26
SUPERVISOR CALL PROTON.(CC->SW)
```

Also, to enhance multiple contiguous register usage, instead of "Rn" you may specify "Rn...Rm", which means that a LM or STM instruction will be used with the registers specified. In this case, it is up to the user to make sure that the semantics of the instruction makes sense. Also, in this case it will be necessary to declare Rn and Rm as being INTEGER. Example:

```
INTEGER TODOUT(3)
INTEGER R0,R3
SUPERVISOR CALL TOD.(R0...R3->TODOUT)
```

Note-- Users not familiar with these hardware instructions should read the I.B.M. 370 Principles of Operations Manual.

4.3 Function Definitions

There are two types of functions: the internal function and the external function. Since these are quite similar in many ways, that part of the description which is specific to external functions will be given in sub-section 4.3.3, that which is specific to internal functions in sub-section 4.3.4, and that which is common to both types will follow in sub-section 4.3.6. The first two sub-sections (4.3.1 and 4.3.2) deal with function entry and exit. Sub-section 4.3.5 deals with variable length calling sequences for functions.

4.3.1 Entry to a Function

Alternate entry points to a function being defined are indicated by

```
ENTRY TO n.
```

where n is the name of the entry. If the argument list is to differ from that specified in the preceding INTERNAL FUNCTION or EXTERNAL FUNCTION for which this is an alternate entry, then the list desired must be specified:

```
ENTRY TO n.(list)
```

where list is zero or more dummy arguments, separated by commas.

R-Type Function Entries

It is also possible to define an R-type entry, in a manner very similar to an R-type call, as for example:

```
EXTERNAL FUNCTION F2.(R0->A,R1->B)
```

which says that on entry, the contents of register 0 are to be stored in local variable A, and register 1 in local variable B. As with R-type calls, the modes of the variable must be such that it makes sense to do a ST into, (ST is the hardware instruction for store) and the variables should be in the local storage module.

4.3.2 Function Return

The legal forms of this statement are:

```
FUNCTION  RETURN  e
FUNCTION  RETURN  e,RC=n
FUNCTUON  RETURN  RC=n
FUNCTION  RETURN
```

This statement is used in a function definition to indicate a return to the calling program. e is an expression whose value is to be the output of this function when the function is considered as a single valued function. e need not be specified if no value is to be returned. (e is the value returned in register 0)

The mode of the expression e must be identical to the mode of the value the function is supposed to return. There are no arithmatic conversions done on this expression.

n is an integer expression whose value is to be the return code from this function call. Return codes are usually multiples of 4. If RC=n is not specified, a return code of zero is returned. (The return code is returned in general register 15.)

4.3.3 External Function Definitions

The prototype for the EXTERNAL FUNCTION is:

```
[RECURSIVE] EXTERNAL FUNCTION {name.         } [keywords]
[REENTRANT]                   {name.(parlist)}
```

where:

[parlist] specifies a parameter list. See Section 4.3.6 for the discussion of parameter lists.

[keywords] can be any one of the following:

```
LINKAGE={MTS|CLSMTS}
STACKSIZE=nP
PROGRAM SIZE={1|2|3}P
```

RECURSIVE Specifies a recursive function.

REENTRANT Specifies a reentrant function; i.e., a function which in no way modifies itself.

If neither RECURSIVE nor REENTRANT is specified, there is a code csect and a data csect in each module, and all variables not explicitly placed elsewhere are in the data csect.

If REENTRANT is specified, there is one csect, a code-plus-invariants csect, and a storage space which is dynamically allocated and initialized once (on the first call) and retained over subsequent calls. Thus any module declared neither REENTRANT nor RECURSIVE can be made reentrant merely by placing "REENTRANT" on the first statment and it should continue to behave the same way. (This is implemented by having each module call GPSECT to obtain or find its data area, and if freshly allocated, it initializes it. It would have been possible to use pseudoregisters for this so that only the main module would have to call GPSECT or GETSPACE, but there is only one pseudoregister vector and since others use them, use of them in GOM was avoided to prevent conflict.)

Functions declared RECURSIVE likewise have one csect, and have storage space which is allocated on every entry and released on every exit. Thus values of variables are NOT preserved from one call to another, unless the variables happen to be in a global area.

Restrictions: If REENTRANT or RECURSIVE is used, the formatted I/O statements may not be used,

since IOH has is not reentrant. Also, the simple I/O statements, except for PRINT COMMENT, may not be used although the support routines are reentrant, because the symbol table needs to be moved to the data area and initialized if it is to be used. This was not deemed worth the work in view of the expectation no reentrant program would ever use READ DATA or PRINT RESULTS.

LINKAGE=MTS The standard MTS Coding Convention Linkages are used both for the entry to this function, and calls from it to other external functions. However, nothing is assumed about the contents of register 11 and no stack limit checking is done. If the module is an EXTERNAL FUNCTION or a REENTRANT EXTERNAL FUNCTION, only the registers upon entry and a pointer to the data area are saved on the stack. If the module is a RECURSIVE EXTERNAL FUNCTION, the whole data area is placed on the stack.

LINKAGE=CLSMTS The standard MTS Coding Convention Linkages are used. In addition, GR11 points to a Pseudo-Register Vector used for GLOBAL AREA storage.

This is available only with the RECURSIVE option, since that is only way it can meaningfully be used. Specification is:

```
RECURSIVE EXTERNAL FUNCTION F.(ARG) LINKAGE=CLSMTS
```

This type of module expects the "MTS" linkage conventions (reg 13 has stack pointer on entry) and also expects a CLS transfer vector pointer to be in the second word in the global area pointed to by reg 11 on entry. Calls to other external routines will, as you might expect, pass on the global area pointer in reg 11 and supply an updated stack pointer in reg 13.

The variable _CLSTV_ is a predefined pointer variable which is set to point to the CLS transfer vector upon entry to the function. It is up to the user to make sure it gets used for functions to be called via the transfer vector, by using a

```
INCLUDE COPY:CLSTV#SG
```

Statement following the EXTERNAL FUNCTION statement, or providing equivalent definitions.

Similarly, _SSDPTR_ is a pointer variable that will point to the short stack descriptor on entry to the function. Functions declared

| Linkage=CLSMTS will always check the new stack pointer against the stack limit in the short stack descriptor on entry. If the limit has been exceeded, a Plus-style stack overflow error interrupt will occur.

PROGRAM SIZE=nP (where n is either 1, 2, or 3) Allows the programmer to specify in advance an upper bound on the size of the external function. This parameter, if specified, allows GOM to generate more efficient code.

STACKSIZE=nP A regular OS S-type entry is made. A stack of the specified size is allocated and all external calls are assumed to be LINKAGE=MTS functions.

This is somewhat equivalent to writing an assembly language routine whose entry code is | generated with an RXENTER macro. The second csect (or dynamic storage obtained for psect) will be extended as specified. In addition, all | external subroutine calls will use the CC | calling conventions with GR13 updated to point | to the next "stack frame" (as for LINKAGE=MTS) and hence either OS or MTS subroutines may be called.

The designation "EXTERNAL" implies that the statements which follow are to be translated independently of the main program in which they are to be used. Because of this independence, this block of statements is to be considered entirely as a separate program, and must have its own DIMENSION and MODE declaration, etc. Names of variables, functions, and labels which denote (or "represent") arguments of the function being defined are designated "dummy variables" (or "bound variables"). The modes of these dummy variables (if other than the normal mode) must be declared in the usual way (see Section 3.2), and arrays which are dummy variables must be dimensioned, so that the translator knows that it is legal to apply subscripts. The word "EXTERNAL" also implies that names chosen for variables and functions in the current function definition program have no relation whatever with identically named variables and functions in the main program (or other external functions), and that no difficulties will be encountered because of the use of similar names.

4.3.4 Internal Function Definitions

The prototype of the INTERNAL function is written as

```
INTERNAL FUNCTION {name.|name.(parlist)}
```

where [parlist] is the same as given in Section 4.3.3.

The designation "INTERNAL" implies that the definition program which follows is to be translated as part of the main program. The word "INTERNAL" also implies that any variables or functions not listed as dummy variables in the definition of the function (but used in its evaluation), are understood to be the same as elsewhere in the main program, and the current values of these variables and functions will be used. Names of variables, functions, and labels which denote arguments of the function being defined are designated "dummy variables" (or "bound variables"). They must be distinct from those appearing elsewhere in the program. The modes of dummy variables (if other than normal mode) must be declared in the usual way, and arrays which are arguments must be dimensioned, so that the translator knows that it is legal to apply subscripts.

A note on the code produced for internal functions: As currently implemented, internal functions are designed to be called only from within the compilation in which they are embedded. In order to make them as efficient as possible, if none of the FUNCTION RETURN statements for an internal function has a value to be returned, then the function entry saves only the function-call register (GR14) and the function return merely loads it and branches (and sets a return code). If a value is returned for an internal function, then the call may be embedded in an expression and thus all registers have to be saved. In any case, on an internal function entry it is assumed that registers 9, 10, 11, 12, and 13 are properly set up, and hence it is not wise to supply these entries as function names to the external world. (Use external function entries instead.)

NOTE: There is another form for an INTERNAL FUNCTION definition known as the one-sentence definition. It is not, however, currently implemented. See Appendix E, Number 8 for details.

4.3.5 Variable Length Calling Sequences

All function calls generated by GOM are potentially variable length. In the case that there is a known number of arguments but some of them may not be present on the call, dummy variables for all possible arguments are listed in the dummy variable list as usual, but the ones that may be possibly absent are placed in another set of parentheses. For example,

```
EXTERNAL FUNCTION F.(A,B,(C,D))
```

defines function F with parameters A and B that will always be there, and parameters C and D that may be there. It is the responsibility of the user to determine if any of the optional arguments are present (see .NBRARG. below) before actually using them.

In the case that there is a truly indefinite number of arguments, the "..." notation at the end of the parameter list indicates this. Thus,

```
EXTERNAL FUNCTION F.(A,B,...)
```

indicates two known arguments followed by zero or more others. If present, the "..." must be last in the dummy variable list. If a parenthesized list of optional arguments occurs, it must be last in the dummy variable list, except for a "..." which may follow it.

In order to find out the number of arguments that a function was actually called with, the unary operator .NBRARG. is provided. An example use of this operator is:

```
N = .NBRARG.F
```

The argument is the function name, and the value is the integer number of arguments used on the most recent call to the function or any of its entries. The successful working of this operator depends on the call having been made with the high-order bit set in the adcon for the last parameter in the parameter list, or, in the case of no parameters, register 1 set to zero. (The latter is, of course, irrelevant in the case where there is at least one required argument.)

In order to access arguments that have no name, the binary operator .ARG. is provided. An example use of this operator is:

```
P = n.ARG.F
```

The right operand is the function name, the left operand is the integer expression specifying the number of the argument desired (starting at 1 for the first argument). The value is a pointer to the Nth argument. This pointer is actually the adcon out of the parameter list, and the last argument will therefore have the high-order bit set, a factor to note if any arithmetic is to be done on this pointer.

4.3.6 Internal and External Functions (Things they have in common)

Each function definition (except one-sentence definitions, described in Section 4.3.4) may define any number of functions and/or any number of procedures. The first entry is defined with the INTERNAL FUNCTION or EXTERNAL FUNCTION statement; the second and succeeding ones, if any, are defined with ENTRY TO statements. If an entry defined with an ENTRY TO statement has no parameter list given on the ENTRY TO statement, as for example

```
ENTRY TO F2.
```

then the parameter list (if any) specified on the INTERNAL FUNCTION or EXTERNAL FUNCTION statement is used for that entry. If an entry is to have a different parameter list, it should be specified in the ENTRY TO statement, as for example

ENTRY TO F2.(A,B,C)

It the user's responsiblilty to make sure that a given entry uses only the parameters it receives through the ENTRY TO statement; the compiler does no checking. The only restriction on names of the dummy variables for the entries is that if the same name is used in two (or more) entries, it must occupy the same ordinal position in each parameter list in which it appears. That is, the following is legal:

```
EXTERNAL FUNCTION F1.(X,Y,Z)
     ...
   ENTRY TO F2.(X,Y)
     ...
END OF FUNCTION
```

but the following is not legal:

```
EXTERNAL FUNCTION F1.(X,Y,Z)
     ...
   ENTRY TO F2.(Y,X)
     ...
END OF FUNCTION
```

In the use of a function (i.e., the call for it) the arguments may be constants, variables, function names, labels, or expressions. However, if one of the arguments appears to the left of an "=" sign in an assignment statement in the function definition it is not meaningful to use a constant or an expression for that argument in the call. As mentioned earlier, the arguments are not be checked for correspondence in mode and number to dummy variables.

An example function definition program is as follows:

```
INTERNAL FUNCTION COS.(X)
     ...
     ...
     ...
   ENTRY TO SIN.
     ...
     ...
     ...
   FUNCTION RETURN ALPHA + J - 3.
   ENTRY TO TAN.
     ...
     ...
     ...
   FUNCTON RETURN BETA/K5 - 4.* D
END OF FUNCTON
```

The first statement (INTERNAL FUNCTION COS.(X)) is a function declaration (i.e., declares that the following statements define a function called COS whose entry point is here). To define COS. as an external function, the declaration would be EXTERNAL FUNCTION COS.(X). Following the words

INTERNAL (or EXTERNAL) FUNCTION and the function name is the dummy variable list [(X) in this example]. The END OF FUNCTION declaration is the last statement in the function definition program. (In an EXTERNAL function definition this is also the last statement in the program.) An entry must be provided for each function being defined, but several functions may share any number of FUNCTION RETURN statements. An entry statement merely marks a point of entry, and does not affect the sequence of computation in any way. The expression after the phrase "FUNCTION RETURN" indicates that on this return the value of the function is to be the value of that expression. This expression must agree in mode with the function whose value it supplies, i.e., it must agree with the expected mode of the function value being called for in the calling program. This agreement is not checked. The function definition whose calls are intended to be included in expressions must have an expression following the FUNCTION RETURN statement. If the calls for a function are to appear in an EXECUTE statement (generally such functions have multiple outputs) the FUNCTION RETURN statement may appear without an accompanying expression. The FUNCTION RETURN statement may also specify a return code to be returned to the calling program (see Section 4.3.2); if not specified a return code of zero will be returned.

Return codes should be used to flag to the calling program unusual situtations. The practice, used in 7090 MAD, of supplying statement labels in calls to functions and then branching to those from inside the function definition will not work, due to the base-displacement nature of the 360/370/470 machines.

It is important to note that internal function definitions of any kind whatever (including the non-implemented single statement definition described in Appendix E) may occur anywhere in the program, except within another internal function definition. Internal function definitions may occur within external function definitions. However, external function definitions may not occur within any other programs, not even within other external function definitions. Each external function definition must be a complete, self-contained program.

Example of a function definition

The following is an example of a function whose value is $1/x$ if $0 < x \leq 1$ and $1/x^2$ if $x > 1$. If $x \leq 0$, one obtains an error return.

```
A  EXTERNAL FUNCTION INVSF.(X)
G      IF X > 0. .AND. X <= 1.
C        FUNCTION RETURN X .P. -1
H      ELSE IF X > 1.
D        FUNCTION RETURN X .P. -2
I      ELSE
E        FUNCTION RETURN 0, RC = 4
K      END IF
B  END OF FUNCTION
```

(Here the statements are all labeled only for reference in what follows.)

The list of dummy variables in the opening declaration (statement A in the preceding paragraph) may contain only nonsubscripted variable names (either individual or array) or function names (without arguments). Within the definition program itself (the statements between statement A and statement B), a function name will usually occur with arguments, and an array variable will usually occur with subscripts.

A few comments about the last example: This definition program defines a single-valued function of X, called INVSF . Since no mode declaration is given it is assumed by the translator that X is floating point. The value of INVSF.(X) is computed by the use of a compound conditional. If $0 < X \leq 1$, (statement G) then statement C is executed, causing a return to the calling program with the value 1/x. If the condition $0 < X \leq 1$ is not true, then the condition $X > 1$ is tested (statement H). If $X > 1$, statement D is executed. Finally, if neither of the conditions $0 < X \leq 1$ or $X > 1$ is true, then statement I finds that $X \leq 0$ and statement E (return with return code 4) is executed.

Suppose

```
          A    = B - D
          X    = T(I) + INVSF.(Y) * T(I-1)
          Y(I) = Z + R(J) * 2.5
```

is part of a program which calls on INVSF, and suppose the function return with RC=4 statement is executed during the evaluation of INVSF.(Y) (i.e., $Y \leq 0$). Then control is returned to the system in which the translated program is embedded, with a return code that is ignored. However, suppose instead these statements:

```
          A    = B - D
     F    Z    = T(I) + INVSF.(Y)->RCVR * T(I-1)
          IF RCVR=4, GO TO ER
     S    Y(I) = Z + R(J) * 2.5
          ...
          ...
          ...
     ER   Z    = 0
     L    Y(I) = 1.
```

are part of the calling program and $Y \leq 0$. When the function returns, the return code is stored in integer variable RCVR. The calling program then can test this returned value and act appropriately.

Appendix A

ALLOWABLE ABBREVIATIONS

> "That is not said right," said the caterpillar.
> "Not quite right, I'm afraid," said Alice timidly; "some of the words have got altered."
>
> Lewis Carroll, Alice in Wonderland

Abbreviations may be used for the key words or phrases for the most commonly used statements. These abbreviations are listed below. The source listing produced by the compiler will have the full phrase instead of the abbreviation, for easier reading. The form of an abbreviation is always the same; viz, the first and last letter of the phrase, with a prime (') between. An example of the use of these abbreviations is:

```
W'R X<Y, T'O ALPHA
W'R X=Y+1
     Z=J
O'R X=Y+2
     Z=J+2
O'E
     Z=J+3
E'L
```

The following is the list of abbreviations which are now available. Note that not all statements may be abbreviated. Although these abbrebiations are "nice" in the sense that they speed up the time required to enter a program in on a terminal and the full phrase is produced on a source listing, for your own benefit (and sanity), it is best not to use them. The use of these abbreviations was intended only for very small "grungy" programs.

```
A'E  ALLOCATE
B'N  BOOLEAN
B'R  BYTE INTEGER
C'E  CONTINUE

C'R  CHARACTER
C'T  CONSTANT
D'D  DYNAMIC RECORD
D'N  DIMENSION

E'L  END OF CONDITIONAL
E'M  END OF PROGRAM
E'N  END OF FUNCTION
E'O  ENTRY TO
```

F'F FOR VALUES OF
F'T FLOATING POINT
F'N FUNCTION RETURN
I'N INTERNAL FUNCTION

I'R INTEGER
L'R LONG INTEGER
L'T LONG FLOATING POINT
N'E NORMAL MODE IS FUNCTION NAME

N'L NORMAL MODE IS STATEMENT LABEL
N'N NORMAL MODE IS BOOLEAN
N'R NORMAL MODE IS INTEGER
N'S NORMAL MODE IS

O'E OTHERWISE
O'R OR WHENEVER
P'N PROGRAM COMMON
P'R POINTER

P'S PRINT RESULTS
P'T PRINT FORMAT
R'A READ AND PRINT DATA
R'E RELEASE

R'T READ FORMAT
S'L STATEMENT LABEL
S'R SHORT INTEGER
T'H THROUGH

T'O TRANSFER TO
U'R USING POINTER
V'R VARYING CHARACTER
V'S VECTOR VALUES

W'R WHENEVER

Appendix B

OPERATORS

Following table lists all operators with their allowable mode contents, arranged first by single character operators, then double, triple, and finally the "dotted" operators in alphabetical order. Note that the Prec heading is for use only by the Computing Center staff for diagnostics.

The modes are abbreviated as follows:

F		FLOATING POINT
	I	INTEGER
	BN	BOOLEAN
	SL	STATEMENT LABEL
	FN	FUNCTION NAME
	C	CHARACTER
	SI	SHORT INTEGER
	BI	BYTE INTEGER
	LI	LONG INTEGER
	P	POINTER
	drc	DYNAMIC RECORD name
	arith	F, I, SI, or BI
	funct	any function call
	rname	one of the names "R0" through "R10"
	mname	one of the names of the modes

Operator Graphic	Name	Prec	Allowable Modes: Left Operand	Right Operand	Result	Manual Refer.
-	negation	28	-	arith	(I or F)[1]	1.4(h)
-	subtraction	24	arith	arith	(I or F)[1]	1.4(b)
			LI	I,SI,BI	LI	
			I,SI,BI	LI	LI	
			LI	LI	LI	
			P	I,SI,BI	P	
			P	P	I	
+	addition	24	arith	arith	(I or F)[1]	1.4(a)
			LI	I,SI,BI	LI	
			I,SI,BI	LI	LI	
			LI	LI	LI	
			P	I,SI,BI	P	
			I,SI,BI	P	P	
*	multiplication	26	arith	arith	(I or F)[1]	1.4(c)
/	division	26	arith	arith	(I or F)[1]	1.4(d)
			LI	I,SI,BI	I	
=	substitution	6	arith,LI	arith,LI	left[2]	2.1
			C	C	C	
			FN	FN	FN	
			SL	SL	SL	
			P	P	P	
=	equality comparative	22	arith	arith	BN	1.7(b)
			C	C	BN	
			P	P	BN	
			FN	FN	BN	
			SL	SL	BN	
=	Rcall input	6	rname	I,BN,FN SL,P		4.2
<	less-than comparative	22	arith	arith	BN	1.7(b)
			C	C	BN	
			P	P	BN	
>	greater-than comparative	22	arith	arith	BN	1.7(b)
			C	C	BN	
			P	P	BN	
,	the comma is used as a separator between list elements.					
@	the at-sign is part of a constant. See Section 1.1.10					
:	selection	40	P	any	right[3]	3.9
<=	less-than or equal-to	22	arith	arith	BN	1.7(b)
			C	C	BN	

	comparative		P	P	BN	
>=	greater-than or equal-to comparative	22	arith C P	arith C P	BN BN BN	1.7(b)
¬=	not-equal comparative	22	arith C P FN SL	arith C P FN SL	BN BN BN BN BN	1.7(b)
->	return code	39	funct	I		4.2
->	Rcall output	6	rname	I,BN,FN SL,P		4.2
...	Block	10	any	any		1.9
...	Substring	10	I,SI,BI	I,SI,BI	I	1.13
.A.	Bitwise and	34	I,SI,BI LI I,SI,BI LI	I,SI,BI I,SI,BI LI LI	I LI LI LI	1.4(j)
.ABS.	Absolute value	36	-	F,I,SI	(I or F)[1]	1.4(g)
.AND.	Logical and	18	BN	BN	BN	1.6(d)
.ARG.	No-name Argument Access	36	I,SI,BI	any	P	4.3.5
.AS.	Mode override	37	any	mname	specified[4]	1.9
.BIT.	Test bit	22	I	I	BN	1.7(d)
.DIMVEC.	Location of the Dimension Vector	38	-	funct	I	3.3
.E.	Same as "=" (comparative)					
.EQ.	Same as "=" (comparative)					
.EQV.	Equivalence	12	BN	BN	BN	1.6(f)
.EV.	Bitwise exclusive or	34	I,SI,BI LI I,SI,BI LI	I,SI,BI I,SI,BI LI LI	I LI LI LI	1.4(i)

.EXOR.	logical exclusive or	16	BN	BN	BN	1.6(e)
.G.	Same as ">"					
.GE.	Same as ">="					
.GT.	Same as ">"					
.IND.	Indirection	42	P	mname	specified[4]	1.14
.L.	Same as "<"					
.LE.	Same as "<="					
.LOC.	Location	38	-	any	P	1.14
.LS.	Left shift	36	I,SI,BI LI	I,SI,BI I,SI,BI	I LI	1.4(k)
.LT.	Same as "<"					
.MPYLI.	same as "*" but LI result	26	I,SI,BI	I,SI,BI	LI	1.4(c)
.N.	Bitwise negation	36	-	I,SI,BI LI	I LI	1.4(i)
.NBRARG.	number of arguments in a function	36	-	funct	I	4.3.5
.NE.	Same as "¬="					
.NOT.	Logical not	20	-	BN	BN	1.6(a)
.OR.	Logical or	16	BN	BN	BN	1.6(c)
.P.	Exponentiation	30	arith	arith	(I or F)[1]	1.4(f)
.REM.	Remainder	26	arith I,SI,BI	arith I,SI,BI	I I	1.4(e)
.RESETBIT.	resetting a bit	6	I	I	-	1.4(l)
.RS.	Right shift	36	I,SI,BI LI	I,SI,BI I,SI,BI	I LI	1.4(k)
.SETBIT.	setting a bit	6	I	I	-	1.4(l)
.SIZE.	size of	40	-	drc	I	3.9

.V.	Bitwise or	32	I,SI,BI LI I,SI,BI LI	I,SI,BI I,SI,BI LI LI	I LI LI LI	1.4(j)
".("	function call	40	FN	any	declared[5]	4.1 4.2
"opnd("	subscription	40	any	arith	left[2]	1.10

Notes:

1. "(I or F)" means that the result is F if either or both of the operands is F; otherwise the result is I.

2. "left" means that the resultant mode is the mode of the left operand.

3. "right" means that the mode of the result is the mode of the right operand.

4. "specified" means that the mode of the result is the mode whose name is the right operand.

5. "declared" means that the mode is what the function was declared as returning (eg, "INTEGER FUNCT.").

Appendix C

List of Statements

Note: The internal statement-type number is for use only by the Computing Center staff for diagnostics.

Internal Statement-type Number	Abbrev-iation	Statement	Comments
41		(iterated statement)	unimplemented
11		(substitution statement)	
56	A'E	ALLOCATE	
31	B'N	BOOLEAN	
47	B'R	BYTE INTEGER	
45	C'R	CHARACTER	
58	C'T	CONSTANT	
17	C'E	CONTINUE	
3		DIAGNOSTIC CONDENSED OFF	COMPILER DEBUG
2		DIAGNOSTIC CONDENSED ON	COMPILER DEBUG
5		DIAGNOSTIC DSCAN OFF	COMPILER DEBUG
4		DIAGNOSTIC DSCAN ON	COMPILER DEBUG
8		DIAGNOSTIC I TABLES	COMPILER DEBUG
9		DIAGNOSTIC II TABLES	COMPILER DEBUG
10		DIAGNOSTIC III TABLES	COMPILER DEBUG
7		DIAGNOSTIC MTRX OFF	COMPILER DEBUG
6		DIAGNOSTIC MTRX ON	COMPILER DEBUG
35	D'N	DIMENSION	
53	D'D	DYNAMIC RECORD	
15		ELSE	
14		ELSE IF b	
16		END IF	
67		END LOOP	
16	E'L	END OF CONDITIONAL	
44	E'N	END OF FUNCTION	
67		END OF ITERATION	
1	E'M	END OF PROGRAM	
21	E'O	ENTRY TO	
36		EQUIVALENCE	unimplemented
19		EXECUTE	
40		EXTERNAL FUNCTION	
34	F'T	FLOATING POINT	
61		FORMAT VARIABLE	
33		FUNCTION NAME	
20	F'N	FUNCTION RETURN	
65		GLOBAL AREA	
13		IF b	
30	I'R	INTEGER	
39	I'N	INTERNAL FUNCTION	
66		ITERATE FOR v=e1,e2,b	FOR VALUES OF not done yet.
12		GO TO l	

50	L'T	LONG FLOATING POINT	
48	L'R	LONG INTEGER	
52		LOOK AT FORMAT	
66		LOOP FOR v=e1,e2,b	FOR VALUES OF not done yet.
29	N'S	NORMAL MODE IS	
14	O'R	OR WHENEVER b	
15	O'E	OTHERWISE	
51	P'R	POINTER	
42		PRINT BCD RESULTS	
27		PRINT COMMENT	
22	P'T	PRINT FORMAT	
43		PRINT HEX RESULTS	
28	P'S	PRINT RESULTS	
37	P'N	PROGRAM COMMON	
24		PUNCH FORMAT	
26	R'A	READ AND PRINT DATA	
25		READ DATA	
23	R'T	READ FORMAT	
59		READ FROM	
64		RECURSIVE EXTERNAL FUNCTION	
63		REENTRANT EXTERNAL FUNCTION	
57	R'E	RELEASE	
46	S'R	SHORT INTEGER	
32	S'L	STATEMENT LABEL	
55		STOP USING POINTER	
62		SUPERVISOR CALL	
18	T'H	THROUGH l, FOR v=e1,e2,b	FOR VALUES OF not done yet.
12	T'O	TRANSFER TO l	
54	U'R	USING POINTER	
49	V'R	VARYING CHARACTER	
38	V'S	VECTOR VALUES	
13	W'R	WHENEVER b	
		WHENEVER b, s	
60		WRITE ON	

Appendix D

Structure of the Object Program Produced

The structure of the program is subject to the following constraints:

1. The program should be reentrant.

2. The compiler is two-pass, with all code generated on the second (triples-to-output) pass.

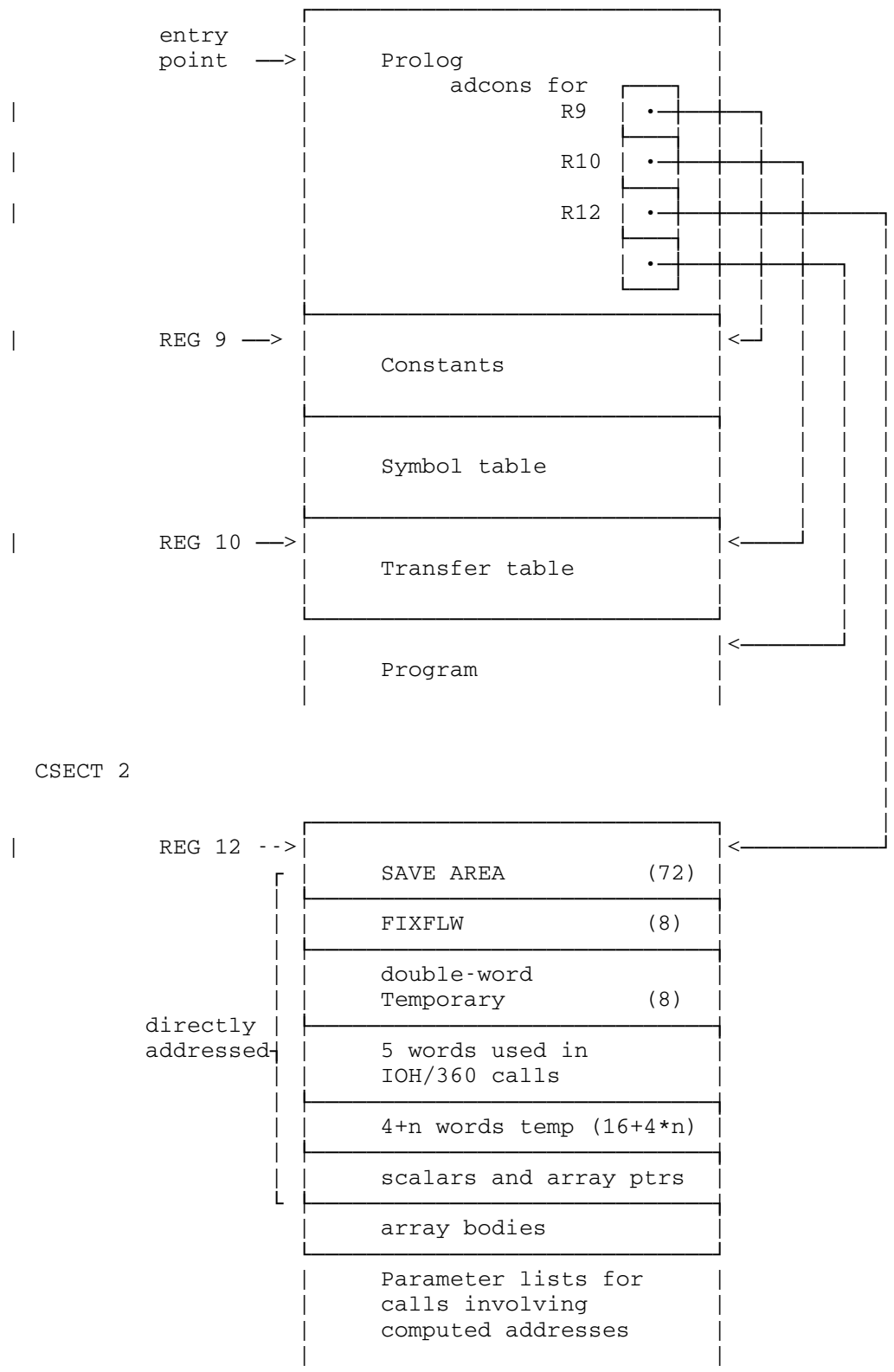
The first requirement dictated separation of the program into an invarient part and a data part. Provision is made for optionally dynamically allocating the data part whne the program is entered. The second constraint, coupled with the base-displacement nature of a 370 type machine, dictates some of the ordering of the output pieces.

General Structure

Each compilation of a main program (or external function) normally generates a module with two control sections. CSECT 1 (name is MAIN for main program) is called the code csect and contains the invariant material; CSECT 2 (name is #MAIN for main program) is the data csect. If a REENTRANT or RECURSIVE external function is being defined, then this area is allocated dynamically instead of produced as the second CSECT. The structure is still the same, and so further references to CSECT2 can be assumed to this case as well.

In general, everything that can be put into CSECT 1 is put there. Only items that will (possibly) change are put in CSECT 2.

The following figures display the general structure of a GOM program when it is compiled assuming OS linkage (i.e., no LINKAGE= on the External Function statement).


CSECT 1
entry point
Prolog
adcons for
R9
R10
R12
REG 9
Constants
Symbol table
REG 10
Transfer table
Program
CSECT 2
REG 12
SAVE AREA (72)
FIXFLW (8)
double-word
Temporary (8)
directly addressed
5 words used in IOH/360 calls
4+n words temp (16+4*n)
scalars and array ptrs
array bodies
Parameter lists for calls involving computed addresses

Register Usage

Gen Reg	0	used in subr calls
	1	used in subr calls
	2	free for temporary assignment
	3	free for temporary assignment
	4	free for temporary assignment
	5	free for temporary assignment
	6	free for temporary assignment
	7	free for temporary assignment
	8	3rd Base Register if PROGRAM SIZE=3P
	9	1st Base Register if PROGRAM SIZE≥1P Also covers constants
	10	2nd Base Register if PROGRAM SIZE≥2P. If the PROGRAM SIZE option isn't used, this covers the transfer table.
	11	Global area pointer if LINKAGE=CLSMTS
	12	Perm. Assigned. Cover local frame (Csect 2)
	13	Same as 12 if OS linkage (i.e., no Linkage= given), pointer to next stack frame otherwise.
	14	Perm. Assigned. Used for transfers and subroutine calls.
	15	Used in subroutine calls
float reg	0	Assigned as needed. Any in use
	2	forced to temporaries
	4	on function call.
	6	on function call.

CSECT 1

PROLOG: If the call is OS linkage, the prolog does the standard register save into save area, and then loads registers from four adcons at the end of the prolog, links the save areas, and branches to the first location of the program. If it is a CC call, then the regsiters are saved on the stack and register 12 and 13 are set to point to the current and next stack frame.

CONSTANTS: The following constants are generated for every program.

1. Doubleword fix-float conversion constant (4E00000000000000)
2. Fullword one (00000001)
3. Halfword four (0004)
4. Halfword minus two (FFFE)
5. FW length of Local Frame (Csect 2) if REENTRANT or RECURSIVE
6. FW address of Local Frame Data Initializer area if REENTRANT or RECURSIVE
7. If the .N. operator was used, fullword minus

one (FFFFFFFF)

8. Fullword 4096 (00001000)
9. If substring notation was used, a fullword with a blank in the high-order byte and zeros elsewhere (40000000)

Constants 2, 3, and 4 are used in I/O lists; constant 2 is also used in several logical operations.

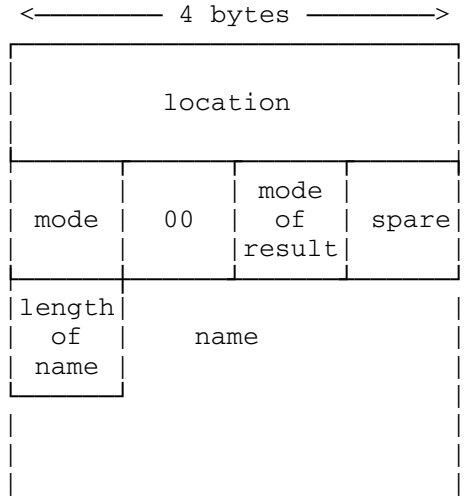
These constants are followed by all 8-byte constants, then all 4-byte (fullword) constants, then all 2-byte constants, and then by all the other-length constants.

SYMBOL TABLE: This is produced only if one of the simple I/O statements (READ DATA, PRINT RESULTS, ...) was used. Current form of the table is:

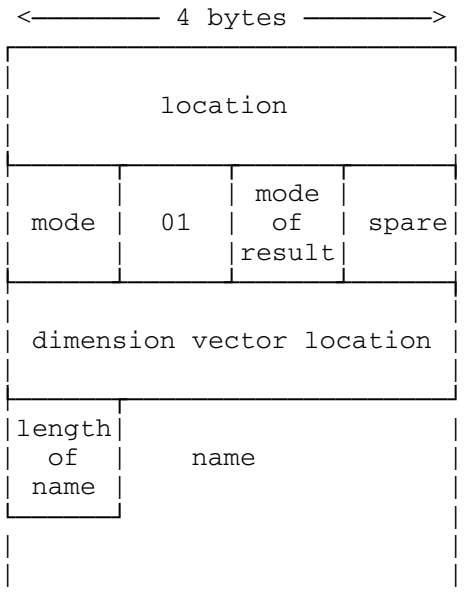
First word is full-word count of number of entries.

Second and following words are entries in one of the two forms:

(a)



(b)



The name is padded with blanks to the next fullword boundary, so that each entry is fullword aligned. "Dimension vector location" is currently the maximum subscript, as only one-dimensional arrays are supported. "Mode of result" has meaning only for function names. The symbol table entries are ordered by increasing location, and the location given for an array is the location of the actual array, not the location of the pointer to the array.

TRANSFER TABLE: Entries for this are produced for internally generated floating addresses. This table consists of a number of fullword adcons. There is a current restriction is a maximum of one page of the (i.e., 1024 max.). This table is placed before the program as far as storage layout goes, but the contents are not produced (and not printed in the print-object) until after the program code has been produced. A transfer table is not needed if the PROGRAM SIZE option is used.

PROGRAM: The remainder of the first Csect is the program. Each instruction or data item produced is printed in the print-object output, in the general form:

```
location [mnemonic] value[/relocated-by]
```

Some examples are:

```
000124  BAL  4510C022
000030       4E000000
```

000128 0000004C/01

For out-of-line or out-of-csect code:

(csect location [mnemonic] value[/relocated-by])

example:

(02 000094 0000013C/01)

CSECT 2

The contents of the data csect are as follows:

1. The first 72 bytes (18 words) of Csect 2 are a standard 18-word save area or the first 64 bytes are the register save area, depending on the linkage type.
2. A double-word aligned fix-float work area, preset to 4E000000 00000000 (only the second word is ever changed).
3. A double-word temporary.
4. 4 words used internally for scratch storage during external function calls.
5. Five words used to build the second parameter list for IOH/360 calls, for the case of a block parameter in which one or both of the ends of the block is a computed address. (E.g., A...A(I))
6. 4+n words of temporaries. Four words are needed to save floating-point registers if necessary (around function calls); more may be needed if computed values arise in parameter lists of subroutine calls.
7. Scalars and array pointers. Scalars are allocated space for their values; arrays are allocated a word which is preset to the location of the array body. 8-byte scalars are allocated first, then 4-byte scalars and array pointers, then 2-byte scalars, and then remaining scalars.

The above seven must be directly addressable. (That is, within a dispacement of some register.) The remaining items are accessed via adcons (such as the array pointers mentioned above).

8. Array bodies. Each array is aligned as required.

9. Parameter lists for subroutine calls with more than one argument, where one or more of the arguments have a computed address.

Note: Wherever possible, I/O parameter lists and subroutine parameter lists are generated in Csect 1 in-line with the code. This cuts down on the addressability problems and helps limit the amount of of initializing that will be required if Cesct 2 is dynamically allocated. (It also makes it easier to read the object listing.)

Appendix E

Unimplemented Statements

This appendix describes all of the unimplemented GOM statements as of 4/20/81. Most likely, a statement will be implemented differently than described here. If you have any questions, please call either Don Boettner or Jim Sterken at the Computing Center (313-764-2121).

1) MODES

The following two modes follow the same principles as those described in Section 1. See Section 1 for details about modes.

Mode Number	Name	Size of Element	Default Alignment
9	VARYING CHARACTER	1	byte
10	LONG FLOATING POINT	8	double word

2) LOOP STATEMENT

This alternate form of a LOOP FOR statement looks like:

LOOP FOR VALUES OF V = E^1, E^2, ..., Em

END LOOP will mark the last executable statement in the block to be repeated. The block of statements following (and not including) the LOOP statement, up to the END LOOP statement will be called the "scope" of the LOOP statement. Following the word OF appears the name of the iteration variable (in the illustration: V), which may be either an individual variable or subscripted array variable of any mode. To the right of the "=" sign may appear any number, i.e., a list, of expressions E^1, ..., Em. The modes of the E's bear the same relationship to the mode of V as they would in the statement V = Ej (see Section 2.1). Thus, if V is an integer or a floating point variable, then each of the Ej must be an integer or floating point expression. Similarly, if V is Boolean, then each of the Ej must be a Boolean expression.

The execution of this statement causes the statements within its "scope" to be executed, first with V = E^1, then again with V = E^2, and so on, until the list of expressions is exhausted. Computation then proceeds with the statement immediately following statement END LOOP. At this time the iteration variable will have the value of the expression Em unless its value was changed during the final iteration. Should a transfer be made to another part of the program at any time during the iteration, V will have its current value. An example of this type of statement is:

```
LOOP FOR VALUES OF BETA = 3, 4, X5, Y(6+I)+3
    J = 5 * BETA + 6
    J1 = J .P..5 - 1
    X(BETA) = J1 * COS.(2.*THETA)
END LOOP
```

3) INPUT/OUTPUT STATEMENTS
(list parameter option)

Another option for the list parameter of input/output statements (described in Section 2.8) follows below. This option follows the same rules as the others in Section 2.8. Recall that the statements are:

```
PRINT FORMAT   f [,RC->v][,list]
PUNCH FORMAT   f [,RC->v][,list]
READ FORMAT    f [,RC->v][,list]
LOOK AT FORMAT f [,RC->v][,list]

READ FROM   i, f [,RC->v][,list]
WRITE ON    o, f [,RC->v][,list]
```

We will not redescribe all the parameters above. The list parameter has another form for an input/output list of elements which follows below. Elements in a list parameter may be iteration elements of the form:

(v=e1, e2, b, list)

where v is the name of a variable, e1 and e2 are arithmetic expressions, b is a Boolean expression, and list is a non-empty input list of the type being presently defined. The interpretation of such an element is exactly analogous to the execution of an interation statment in that the values designated on the list are transmitted (as input or output) until b has the value 1B, with v being initialized to e1 and being incremented by e2 after each transmission of the list. These iterations may be nested (just as iteration statements).

4) MODIFYING THE DECLARED RANGE OF ARRAY SUBSCRIPTS
(Use of the SETDIM operation)

It may be that the declared range for a subscript should be modified during execution of the program to reflect the storage requirements of different sets of data. In other words, the need can arise to keep the dimension current.

For example, a program may be written which deals with an M x N array called D, and the largest values which M and N may have are are 30 and 20, respectively. Suppose we employ the declaration

DIMENSION D(30*20)

For a particular set of data, it might happen that M = 6 and N = 4. Unless this were reflected in the "dimension information", the output list element D(1,1), ..., D(M,N) would cause the values of D(1,1), ..., D(1,20), D(2,1), ..., D(2,20), ..., D(5,1), ..., D(5,20), D(6,1), ..., D(6,4) to be printed, and many of these values would be meaningless. A library subroutine (i.e., external function) is available called SETDIM, which will update dimension information when executed. The arguments to SETDIM are the name of the array, followed by the new ranges of all the subscripts, in order. In the example used here, one would write (after the values of M and N have been read as data):

```
SETDIM.(D,M,N)
```

The arguments giving the ranges of the subscripts may be any integer valued expressions. The block notation must be used if the lower limit is not 1, so that one might write

```
SETDIM.(D,3...N,M)
```

Expressions of integer mode may be written as part of a block designation. If a subscript is to range from N to 2*N, one would write:

```
SETDIM.(D,N...2*N,M)
```

5) DUPLICATE (OR MULTIPLE) DIMENSIONING

Several variables with the same dimension information may be grouped in a declaration. Any form of the dimension declaration may be used (see Appendix B). Variables so grouped will actually refer to the same dimension vector, and any change to the dimension information, such as a call for SETDIM, will be a change for all arrays in the group.

Examples:

```
(1)  INTEGER (A,B,C)(10),(D,E,F,G)(10*15)
(2)  DIMENSION (U,S,P)(25)
```

6) EQUIVALENCE DECLARATION

One of the major uses of equivalence in the past was to treat a variable of one mode as if it were another mode (by equivalencing names of different modes to the same storage locations). This effect can still be obtained in GOM via the .AS. operator which is implemented. The .AS. operator makes it clearer what is actually intended. This operator is described in Section 1.9.

The EQUIVALENCE declaration has the form

```
EQUIVALENCE (a¹,a²,...,am), (b¹,b²,...,bn),...
```

where the a's and b's are individual variables or variables shown with constant linear subscripts.

Example:

```
EQUIVALENCE (A,B), (MATRIX, XARRAY), (C, D(3))
```

which implies that the variables A and B are to represent the same storage location throughout the program, that MATRIX and XARRAY are to represent the same storage location through the program, etc. (Two variables which represent the same location always have the same value at any given time.) Thus, any number of equivalences may be established by one EQUIVALENCE declaration, and any number of such declarations may occur (at any place) in a program.

Variables whose names appear within the same set of parentheses need not have the same mode. The mode must be established by the appropriate MODE declaration for each of the variables. Within an EQUIVALENCE declaration do not establish mode. Occurrences

A nonsubscripted array variable name in an EQUIVALENCE declaration represents that element of the array (considered as a one-dimensional vector) whose subscript is zero. Reference in an EQUIVALENCE declaration to an array element of any number of dimensions may be made by linear-subscript only (i.e., as an element of a vector). Note that occurrence of any elements from any two arrays in the same parentheses implies equating the entire arrays accordingly.

7) VECTOR VALUES statement (alternate form)

The alternate form of a VECTOR VALUES statement takes the form:

```
VECTOR VALUES A(m) ... A(n) = k
```

Here m and n are integer constants, with $m \leq n$, and k is any constant (not a sequence of constants). This statement is treated exactly the same as the implemented VECTOR VALUES form described in Section 3.6 except that A(m), A(m + 1), ..., A(n - 1), A(n) all are preset with the value k. The storage reservation for A is equivalent to DIMENSION A(n) in this case, and A is set to the mode of k.

These declarations are useful for presetting tables, dimension vectors, format descriptions, etc. The presetting is done at the time of translation. The constants are loaded (as part of the translated program) into A. These declarations produce no computation at execution time. However, the values of A may be modified later by other statements in the program during execution.

Note that VECTOR VALUES defines initialized variables. If the values of the items being initialized will never be changed,

it is better to use the CONSTANT declaration (described in Section 3.7) which defines named constants. As an example, consider

```
                VECTOR VALUES PI = 3.14159265
        and     CONSTANT PI = 3.14159265
```

In the first case, a variable PI is defined which starts out (before execution begins) with a value of 3.14159265. In the second case, PI is defined as a name for the constant 3.14159265.

Variables which have been assigned to PROGRAM COMMON storage (see Section 3.4) may not be preset by a VECTOR VALUES statement.

8) INTERNAL FUNCTION DEFINITION (One-Sentence Definition)

Note: This one sentence form can and must be implemented as a three-statement regular INTERNAL FUNCTION as described in Sections 4.3.4 and 4.3.6.

This form of the internal function definition (not available as an external function definition because the latter must be a complete, independent program) has the form:

```
                INTERNAL FUNCTION Name.(List) = E
```

where Name is the name of the function being defined and E is an expression (arithmetic or Boolean) involving the variables in the List of dummy variables.

As a three-statement regular INTERNAL FUNCTION, this one-sentence definition looks like:

```
                INTERNAL FUNCTION NAME.(LIST)
                FUNCTION RETURN
                END OF FUNCTION
```

Example of the one-sentence form:

```
    INTERNAL FUNCTION SUMSQ.(X, Y, Z) = X*X + Y*Y + Z*Z - T*T
```

As indicated above, X, Y, and Z, as they occur on the right side of the equals sign, are dummy variables, and "(X, Y, Z)" is the dummy variable list. The current value of T, however, will be obtained and used each time the value of the function is needed. An example of the use of the function so defined would be:

```
                A = 1. - SUMSQ.(U, V + 3, W) .P. .5
```

In the one-sentence internal function definition, at least one dummy variable must be indicated, even if the function does not use arguments. The reason for this is that a function such

as "FUN." represents a FUNCTION NAME, not a FUNCTION CALL, but on the other hand, a function such as "FUN.(X)" (where X is the dummy variable) does represent a FUNCTION CALL. In GOM, the statement "X = FUN." is in FUNCTION NAME mode, but in a statement such as "X = FUN.(X)", the mode is determined by the value of the statement. For example:

```
INTERNAL FUNCTION F.  = 2*Y + 1
```

is not a legal definition, but

```
INTERNAL FUNCTION F.(X) = 2*Y + 1
```

is a legal definition.

Only nonsubscripted names of variables (either individual or array) or names of functions (without arguments) may appear in the dummy variable list. In the use of the function in an expression, the arguments may be any expressions that agree in mode with the corresponding dummy variable in the declaration.

The modes of dummy variables and "actual" arguments must correspond. Thus, in the example definition

```
INTERNAL FUNCTION POLY.(N, X, FN.) = FN.(J*X).P.N - X/XBAR
```

which might be used in the statement

```
BETA   ZQ = POLY.(M + 1, Y, SIN.) + POLY.(M - 1, Z, COS.)
```

it is understood that if N is in the integer mode, then so is M, and if X is in the floating point mode, then so are Y and Z. It is, of course, presumed that both M and N have been declared to be in the integer mode. Similarly, the values of SIN. and COS. must be the same mode as the values of FN. Moreover, in the use of functions, this mode correspondence cannot be checked by the translator since the machine uses independent routines for processing functions.

The function POLY has as one of its arguments, the name of a function. In the statement BETA, the function used in the first term to the right of the "=" sign is SIN, and in the second term, COS is used. Hence, statement BETA is then equivalent to:

```
BETA  ZQ = SIN.(J*Y).P.(M+1) - Y/XBAR + COS.(J*Z).P.(M-1) -
                                Z/XBAR
```

Appendix F

Example Programs

The following programs show how to get started in writing GOM programs. All of the programs were run under the *GOM compiler, and the object program was run under $DEBUG. The data used in all of these programs comes from a data file. Since Sections 2.8 and 2.9 don't really state how to correctly set up a data file for reading, the first program AND it's data file are listed. For the first example, the $RUN and $DEBUG commands that produced the output are shown. Note that the data file UNIT number is specified as 5, and the output listing is specified as SPRINT.

How to set up the $RUN and $DEBUG commands

For the first example program, this is how the $RUN command looked:

```
$RUN *GOM SCARDS=program SPRINT=printout T=1 PAR=TEST
```

Note that SPUNCH defaults to -LOAD, and that the PAR=TEST option is put in so that we can $DEBUG the program. Now, this is how the $DEBUG command looked:

```
$DEBUG -LOAD 5=datafile SPRINT=printout(*L+1) T=1
```

Once again note that the convention here is to use UNIT 5 for the data file.

Example Program #1: Program Listing

This program computes the winning average of a baseball team given it's won-lost record. Note that 2 sets of data are given so that the LOOP statement can be shown.

```
****************************************************************
*                                                              *
*                     EXAMPLE PROGRAM #1                       *
*                                                              *
*     This program calculates a team's winning average from    *
* it's won-lost record. The number of games won and lost are   *
* read in from a data file (shown below).                      *
*                                                              *
****************************************************************

        INTEGER GAMES_WON, GAMES_LOST, GAMES_PLAYED, COUNTER
        FLOATING POINT TEAM_AVERAGE

*  Set the loop counter to 1 and begin the loop

        COUNTER = 1

        LOOP UNTIL COUNTER = 3
           PRINT COMMENT "     TEAM STATISTICS PROGRAM"

*  Now read in the data

           READ DATA FROM UNIT 5

*  Now compute the games played, and then the team's average

           GAMES_PLAYED = GAMES_WON + GAMES_LOST
           TEAM_AVERAGE = (GAMES_WON * 100) / GAMES_PLAYED

*  Now print out the statistics

           PRINT RESULTS GAMES_WON
           PRINT RESULTS GAMES_LOST
           PRINT RESULTS GAMES_PLAYED
           PRINT RESULTS TEAM_AVERAGE

*  Now get ready for another set of data

           PRINT COMMENT "-"

           COUNTER = COUNTER + 1

        END LOOP
        END OF PROGRAM
```

The data file for this program looks likes this:

```
      1     GAMES_WON=38, GAMES_LOST=12, *
      2     GAMES_WON=25, GAMES_LOST=75, *
END OF FILE
```

Note that the asterisk (*) tells the program when to quit reading when the READ FROM UNIT 5 statement is encountered (see Section 2.9).

Example Program #1: Output

```
       TEAM STATISTICS PROGRAM

   GAMES_WON       =          38

   GAMES_LOST      =          12

   GAMES_PLAYED    =          50

   TEAM_AVERAGE    =    76.0000


       TEAM STATISTICS PROGRAM

   GAMES_WON       =          25

   GAMES_LOST      =          75

   GAMES_PLAYED    =         100

   TEAM_AVERAGE    =    25.0000
```

Example Program #2: Program Listing

This program demonstrates how the IF statement works. The program determines whether numbers are even or odd via this IF statement. Note that in this program it would have been better to use the IF...ELSE statement because the GOTOs and statement labels would be eliminated. For an example of IF...ELSE, see Example Program #4.

```
*****************************************************************
*                                                               *
*                      EXAMPLE PROGRAM #2                       *
*                                                               *
*      This program will read a value from the data file and    *
*  determine if that integer value is either an even or odd     *
*  number. the purpose of this program is to demonstrate the    *
*  IF statement construct.                                      *
*                                                               *
*****************************************************************

          INTEGER INPUT_VALUE, COUNTER

*  Set the loop counter to 1 and begin the iteration

          COUNTER = 1

          LOOP UNTIL COUNTER = 6

*  Now read in an integer value and echo it on the output

             READ DATA FROM UNIT 5

             PRINT RESULTS INPUT_VALUE

*  Now determine if it is even or odd

             IF INPUT_VALUE.REM.2 .EQ. 0, GO TO EVEN

             PRINT COMMENT "INPUT_VALUE IS ODD"
             GO TO REPEAT

EVEN         PRINT COMMENT "INPUT_VALUE IS EVEN"

REPEAT       PRINT COMMENT " "
             COUNTER = COUNTER + 1
          END LOOP
          END OF PROGRAM
```

Example Program #2: Output

```
INPUT_VALUE     =         123
INPUT_VALUE IS ODD


INPUT_VALUE     =        -222
INPUT_VALUE IS EVEN


INPUT_VALUE     =           0
INPUT_VALUE IS EVEN


INPUT_VALUE     =        9191
INPUT_VALUE IS ODD


INPUT_VALUE     =        -114
INPUT_VALUE IS EVEN
```

Example Program #3: Program Listing

This program demonstrates the LOOP FOR and LOOP WHILE constructs. Note that two nested loops are used. The program reads in numbers and prints out the largest power of two less than or equal to that number.

```
*****************************************************************
*                                                               *
*                     EXAMPLE PROGRAM #3                        *
*                                                               *
*       This program reads in a positive integer value from     *
*  the data file and then calculates the largest power of 2     *
*  that is less than or equal to the number that was read in.   *
*  The purpose of this program is to demonstrate the LOOP FOR   *
*  and LOOP WHILE statement constructs. Note that this program  *
*  uses 2 nested loops.                                         *
*                                                               *
*****************************************************************

         INTEGER DATA_VALUE, POWER_OF_TWO, COUNTER

*  Write a heading for this program and then begin the iteration

         PRINT COMMENT $      POWER OF TWO PROGRAM$

         LOOP FOR COUNTER=1, 1, COUNTER=6

*  Now read in a value and echo it on the output

            READ DATA FROM UNIT 5
            PRINT RESULTS DATA_VALUE

*  Initialize POWER_OF_TWO to 1 and then double it until it
*  finally exceeds the value in DATA_VALUE, at which time we
*  will have gone one power too far.

            POWER_OF_TWO = 1

            LOOP WHILE DATA_VALUE .GE. POWER_OF_TWO
               POWER_OF_TWO = POWER_OF_TWO * 2
            END LOOP

*  Adjust POWER_OF_TWO to the correct value and print it out

            POWER_OF_TWO = POWER_OF_TWO / 2

            PRINT RESULTS POWER_OF_TWO
            PRINT COMMENT " "
         END LOOP
         END OF PROGRAM
```

Example Program #3: Output

```
     POWER OF TWO PROGRAM

DATA_VALUE      =       54321

POWER_OF_TWO    =       32768


DATA_VALUE      =          89

POWER_OF_TWO    =          64


DATA_VALUE      =        4095

POWER_OF_TWO    =        2048


DATA_VALUE      =        1024

POWER_OF_TWO    =        1024


DATA_VALUE      =           3

POWER_OF_TWO    =           2
```

Example Program #4: Program Listing

This program demostrates the use of the IF...ELSE statement construct. The program computes square roots WITHOUT using a SQRT subroutine.

```
*****************************************************************
*                                                               *
*                      EXAMPLE PROGRAM #4                       *
*                                                               *
*      This program computes the approximate square root of     *
*  a number read from the data file. This program shows you     *
*  how to write a square root routine without using the SQRT    *
*  subroutine. The successive approximations method is used.    *
*  The algorithm terminates when the aprroximation is within    *
*  0.01% of the correct answer. This program demonstrates the   *
*  IF...ELSE statement construct.                               *
*                                                               *
*****************************************************************

         FLOATING POINT INPUT_NUMBER, GUESS, QUOTIENT, LOWER,
+                       APPROX_SQR_ROOT

         PRINT COMMENT "     SQUARE ROOT PROGRAM"

*  Read in the first value from the data file and begin the
*  looping.

         READ DATA FROM UNIT 5

         LOOP WHILE INPUT_NUMBER >= 0
            PRINT RESULTS INPUT_NUMBER

*  Take care of the special case: Zero

            IF INPUT_NUMBER .EQ. 0, GO TO SPECIAL

*  Begin the routine for finding the square root

            GUESS = 3
            QUOTIENT = INPUT_NUMBER / GUESS

*  Now set LOWER to the "right" value

            IF GUESS .LT. QUOTIENT
               LOWER = GUESS

            ELSE
               LOWER = QUOTIENT

            END IF

*  Now get better approximations...
```

```
            LOOP WHILE .ABS.(GUESS-QUOTIENT) > 0.0001*LOWER
               GUESS = (GUESS+QUOTIENT) / 2.0
               QUOTIENT = INPUT_NUMBER / GUESS

*  Set LOWER to an even more "right" value

               IF GUESS < QUOTIENT
                  LOWER = GUESS

               ELSE
                  LOWER = QUOTIENT

               END IF
            END LOOP

*  Skip over the special case

            GO TO OUTPUT

SPECIAL     GUESS = 0

OUTPUT      APPROX_SQR_ROOT = GUESS
            PRINT RESULTS APPROX_SQR_ROOT
            PRINT COMMENT " "

*  Read in another number and loop again

               READ DATA FROM UNIT 5
         END LOOP
         END OF PROGRAM
```

Example Program #4: Output

```
    SQUARE ROOT PROGRAM

INPUT_NUMBER    =    10.0000

APPROX_SQR_ROOT =    3.16228


INPUT_NUMBER    =    1234.00

APPROX_SQR_ROOT =    35.1295


INPUT_NUMBER    =   0.460000

APPROX_SQR_ROOT =   0.678233


INPUT_NUMBER    =    3.00000

APPROX_SQR_ROOT =    1.73205


INPUT_NUMBER    =   0.0

APPROX_SQR_ROOT =   0.0
```

Example Program #5: Program Listing

This program checks to see if a number is a palidrome. The numbers are inputted from a data file. Note that in this program we must make a copy of the information in order to test it later. By performing operations on the number, we are destroying its original contents.

```
*****************************************************************
*                                                               *
*                       EXAMPLE PROGRAM #5                      *
*                                                               *
*       This program takes some data values read from data      *
*  and determines whether the number is a palidrome; i.e.,      *
*  the reverse of the number is the number. The algorithm used  *
*  will be to construct the reverse number from the given       *
*  number and then test to see if the two are equal. This       *
*  program gives you an idea of about number-crunching          *
*  techniques used in more detailed programs.                   *
*                                                               *
*****************************************************************

         INTEGER ORIGINAL_NUMBER, REVERSE_NUMBER, DIGIT,
+                COPY_OF_1ST

         PRINT COMMENT "     NUMERIC PALIDROME PROGRAM"

*  Get the value for the ORIG_NUMBER and begin the iteration

         READ DATA FROM UNIT 5

         LOOP WHILE ORIGINAL_NUMBER > 0
            PRINT RESULTS ORIGINAL_NUMBER

*  Initialize the reverse number to zero, and make a copy of
*  the number so we don't destroy it when operating on it.

            REVERSE_NUMBER = 0
            COPY_OF_1ST = ORIGINAL_NUMBER

*  Reverse the number.

            LOOP WHILE COPY_OF_1ST > 0
               DIGIT = COPY_OF_1ST .REM. 10
               COPY_OF_1ST = COPY_OF_1ST / 10

               REVERSE_NUMBER = REVERSE_NUMBER*10 + DIGIT
            END LOOP

*  Now determine if it's a palidrome and print out an answer

            IF ORIGINAL_NUMBER = REVERSE_NUMBER,
               PRINT COMMENT "THE NUMBER IS A PALIDROME"
```

```
            ELSE
               PRINT COMMENT "THE NUMBER ISN'T A PALIDROME"

            END IF

            PRINT COMMENT " "      ;* Added spacing control
            READ DATA FROM UNIT 5  ;* Read in another number
         END LOOP
         END OF PROGRAM
```

Example Program #5: Output

```
     NUMERIC PALIDROME PROGRAM

ORIGINAL_NUMBER =        1221
THE NUMBER IS A PALIDROME


ORIGINAL_NUMBER =      123123
THE NUMBER ISN'T A PALIDROME


ORIGINAL_NUMBER =       24642
THE NUMBER IS A PALIDROME


ORIGINAL_NUMBER =          33
THE NUMBER IS A PALIDROME


ORIGINAL_NUMBER =    99889988
THE NUMBER ISN'T A PALIDROME


ORIGINAL_NUMBER =           1
THE NUMBER IS A PALIDROME
```

Example Program #6: Program Listing

This program demonstrates the use of INTERNAL FUNCTIONs in a basic GOM program. This program is basically a rewrite of Example Program #6 except that functions are used to read in and echo the data, and to determine if the number is even or odd. Note the conventions used to pass and retrieve parameters in GOM.

```
*****************************************************************
*                                                               *
*                     EXAMPLE PROGRAM #6                        *
*                                                               *
*        This program is a rewrite of Example Program #2        *
*   except that this program makes use of functions in GOM.     *
*   A value is read from data, printed out, and then deter-     *
*   mined if it is even or odd number. The purpose of this      *
*   program is to demonstrate simple function usage in GOM.     *
*                                                               *
*****************************************************************

          INTEGER INPUT_VALUE, COUNTER

*  Set the loop counter to 1 and begin the iteration

          COUNTER = 1

          LOOP UNTIL COUNTER = 6

*  Now call the INTERNAL FUNCTION READ. which will read a data
*  value from the data file and echo it on the output. This
*  value is then passed back into INPUT_VALUE.

             INPUT_VALUE = READ.(X)

*  Now call the INTERNAL FUNCTION EVEN_ODD. which will
*  determine if the number is even or odd, and it will print out
*  the appropriate message.

             EVEN_ODD.(INPUT_VALUE)

             PRINT COMMENT " "       ;* Added spacing control
             COUNTER = COUNTER + 1   ;* Bump the loop counter
          END LOOP
```

```
*****************************************************************
*****************************************************************
***                                                           ***
***                 INTERNAL FUNCTION READ.                   ***
***                                                           ***
***      This function just reads in a number from data and   ***
***  prints it out. Note that the value is passed back to     ***
***  calling statement. Also take note that we must over-      ***
***  ride the default normal mode (FLOATING POINT) since      ***
***  we are passing back an INTEGER.                          ***
***                                                           ***
*****************************************************************
*****************************************************************

          INTERNAL FUNCTION READ.

          NORMAL MODE IS INTEGER      ;* Overriding the default

          INTEGER DATA_VALUE

          READ DATA FROM UNIT 5       ;* Read in a value
          PRINT RESULTS DATA_VALUE    ;* Print the value

          FUNCTION RETURN DATA_VALUE  ;* Return the value
          END OF FUNCTION


*****************************************************************
*****************************************************************
***                                                           ***
***              INTERNAL FUNCTION EVEN_ODD.                  ***
***                                                           ***
***       This internal function determines whether the       ***
***  input value is even or odd, and prints out the approp-   ***
***  riate message. Note that INPUT_VALUE is passed to this   ***
***  function.                                                ***
***                                                           ***
*****************************************************************
*****************************************************************

          INTERNAL FUNCTION EVEN_ODD.(NUMBER)

          INTEGER NUMBER

          IF NUMBER .REM.2 .EQ. 0
             PRINT COMMENT "INPUT_VALUE IS EVEN"

          ELSE
             PRINT COMMENT "INPUT_VALUE IS ODD"

          END IF

          FUNCTION RETURN
          END OF FUNCTION

          END OF PROGRAM
```

Example Program #6: Output

```
DATA_VALUE      =         123
INPUT_VALUE IS ODD


DATA_VALUE      =        -222
INPUT_VALUE IS EVEN


DATA_VALUE      =           0
INPUT_VALUE IS EVEN


DATA_VALUE      =        9191
INPUT_VALUE IS ODD


DATA_VALUE      =        -114
INPUT_VALUE IS EVEN
```

Example Program #7: Program Listing

This program calculates the square root of a number. This is essentially a rewrite of Example Program #4 except that an EXTERNAL FUNCTION is used and demonstrated. One difference between the use of INTERNAL FUNCTIONs in the last program (#6) and this EXTERNAL FUNCTION is that we must put an END OF PROGRAM statement at the end of the EXTERNAL FUNCTION as well as at the end of the main program. This program could have very well used an INTERNAL function as well but obviously the purpose of this example program is to show the use of an EXTERNAL FUNCTION.

```
*****************************************************************
*                                                               *
*                      EXAMPLE PROGRAM #7                       *
*                                                               *
*      This program computes the approximate square root of a   *
*  number read from the data file. This program is just a       *
*  rewrite of Example Program #4 written with an EXTERNAL       *
*  FUNCTION subroutine that calculates the approximate square   *
*  root of a given number. This approximation is within 0.01%   *
*  of the correct answer. This program demonstrates EXTERNAL    *
*  FUNCTIONs in a GOM program.                                  *
*                                                               *
*****************************************************************

         FLOATING POINT INPUT_NUMBER, APPROX_SQR_ROOT

         PRINT COMMENT "     SQUARE ROOT PROGRAM"

*  Read in the first value from the data file and begin the
*  looping.

         READ DATA FROM UNIT 5

         LOOP WHILE INPUT_NUMBER >= 0
            PRINT RESULTS INPUT_NUMBER

*  First see if the number is zero. If so then just set
*  APPROX_SQR_ROOT to zero and don't bother with the subrout-
*  ine... just branch to OUTPUT.

            IF INPUT_NUMBER = 0
               APPROX_SQR_ROOT = 0     ;* The sqrt of 0 is 0
               GO TO OUTPUT            ;* Skip over SQROOT.
            END IF

*  Call EXTERNAL FUNCTION SQROOT. to get the square root. Note
*  that APPROX_SQR_ROOT will get the returned value.

            APPROX_SQR_ROOT = SQROOT.(INPUT_NUMBER)

*  Now print out the answer and skip a line
```

```
OUTPUT      PRINT RESULTS APPROX_SQR_ROOT
            PRINT COMMENT " "

*  Read in another number and loop again

               READ DATA FROM UNIT 5
            END LOOP
            END OF PROGRAM


****************************************************************
****************************************************************
***                                                          ***
***                EXTERNAL FUNCTION SQROOT.                 ***
***                                                          ***
***        This External Function computes the square root   ***
***   value of the number passed to this subroutine. The     ***
***   square root value is then passed back to the calling   ***
***   statement. First we must declare some variables used   ***
***   to make "better" approximations, and then start calc-  ***
***   ulating....                                            ***
***                                                          ***
****************************************************************
****************************************************************

            EXTERNAL FUNCTION SQROOT.(DATA_VALUE)

            FLOATING POINT GUESS, QUOTIENT, LOWER

            GUESS = 3
            QUOTIENT = DATA_VALUE / GUESS

*  Now set LOWER to the "right" value

            IF GUESS .LT. QUOTIENT
               LOWER = GUESS

            ELSE
               LOWER = QUOTIENT

            END IF

*  Now get better approximations...

            LOOP WHILE .ABS.(GUESS-QUOTIENT) > 0.0001*LOWER
               GUESS = (GUESS+QUOTIENT) / 2.0
               QUOTIENT = DATA_VALUE / GUESS

*  Set LOWER to an even more "right" value

               IF GUESS < QUOTIENT
                  LOWER = GUESS

               ELSE
                  LOWER = QUOTIENT
```

```
           END IF
         END LOOP

         FUNCTION RETURN GUESS   ;* Returning the sqrt. value

      END OF FUNCTION
   END OF PROGRAM
```

Example Program #7: Output

```
       SQUARE ROOT PROGRAM

   INPUT_NUMBER    =    10.0000

   APPROX_SQR_ROOT =    3.16228


   INPUT_NUMBER    =    1234.00

   APPROX_SQR_ROOT =    35.1295


   INPUT_NUMBER    =   0.460000

   APPROX_SQR_ROOT =   0.678233


   INPUT_NUMBER    =    3.00000

   APPROX_SQR_ROOT =    1.73205


   INPUT_NUMBER    =   0.0

   APPROX_SQR_ROOT =   0.0
```

Example Program #8: Program Listing

This program calculates the largest power of two for a value read in from data. This program is a rewrite of Example Program #3 except that I/O is not done using the simplified statements described in Section 2.9. Up until now, all I/O has been handled this way but for a general case, I/O will not coincide with the "simplified" format. The I/O demonstrated in this program calls on the IOH format routines described in MTS Volume #5 (System Services). For debugging purposes, simplified I/O in GOM will suffice, but when fancy I/O is required, it is a neccesity to use the IOH I/O formats (see Section 2.8).

```
*****************************************************************
*                                                               *
*                      EXAMPLE PROGRAM #8                       *
*                                                               *
*          This program calculates the largest power of two     *
*   that is less than or equal to a value read from data.       *
*   This is essentially a rewrite of Example Program #3         *
*   except that the IOH input/output statement constructs are   *
*   used. Up to now, all I/O was done using the simplified      *
*   I/O statements described in Section 2.9. The purpose of     *
*   this program is to demonstrate the IOH I/O statement        *
*   types.                                                      *
*                                                               *
*****************************************************************

         INTEGER DATA_VALUE, POWER_OF_TWO, COUNTER
         CHARACTER TEXT(27), TEXTFMT(13), READFMT(2)

*  Set up the heading for this program

         TEXT(0|20) = "POWER OF TWO PROGRAM"   ;* The header...
         TEXTFMT(0|12) = $"1",RC26.20*$        ;* and its format

         WRITE ON UNIT 6, TEXTFMT, TEXT        ;* Print it out

*  Now begin the iteration. First read in a value and echo it
*  on the output.

         LOOP FOR COUNTER=1, 1, COUNTER=6

            TEXT(0|25) = "THE INPUT DATA VALUE WAS:"
            TEXTFMT(0|13) = $"0",C27.25,I*$

            READFMT(0|2) = "I*"

            READ FROM UNIT 5, READFMT, DATA_VALUE

            WRITE ON UNIT 6, TEXTFMT, TEXT, DATA_VALUE

*  Initialize POWER_OF_TWO to 1 and then double it until it
*  finally exceeds the value in DATA_VALUE, at which time we
```

```
*  will have gone one power too far.

           POWER_OF_TWO = 1

           LOOP WHILE DATA_VALUE .GE. POWER_OF_TWO
              POWER_OF_TWO = POWER_OF_TWO * 2
           END LOOP

*  Adjust POWER_OF_TWO to the correct value and print it out

           POWER_OF_TWO = POWER_OF_TWO / 2

           TEXT(0|26) = "THE LARGEST POWER OF 2 IS:"
           TEXTFMT(0|9) = "C27.26,I*"

           WRITE ON UNIT 6, TEXTFMT, TEXT, POWER_OF_TWO
        END LOOP
        END OF PROGRAM
```

Example Program #8: Output

```
      POWER OF TWO PROGRAM

THE INPUT DATA VALUE WAS:   54321
THE LARGEST POWER OF 2 IS:  32768

THE INPUT DATA VALUE WAS:      89
THE LARGEST POWER OF 2 IS:     64

THE INPUT DATA VALUE WAS:    4095
THE LARGEST POWER OF 2 IS:   2048

THE INPUT DATA VALUE WAS:    1024
THE LARGEST POWER OF 2 IS:   1024

THE INPUT DATA VALUE WAS:       3
THE LARGEST POWER OF 2 IS:      2
```