MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS CORPORATION

MCDONNELL DOUGLAS

COMPILER WRITING SYSTEM DETAIL DESIGN SPECIFICATION

VOLUME I

Language Specification

APRIL 1974

MDC G5359

PREPARED BY:

W. J. ARTHUR
MCDONNELL DOUGLAS ASTRONAUTICS COMPANY
ADVANCE INFORMATION SYSTEMS

PREPARED UNDER THE DIRECTION OF:

MR. B. C. HODGES
MARSHALL SPACE FLIGHT CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
UNDER CONTRACT NAS8-27202

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-WEST

5301 Bolsa Avenue, Huntington Beach, CA 92647

COMPILER WRITING SYSTEM

DETAIL DESIGN SPECIFICATION

VOLUME I - LANGUAGE SPECIFICATION

MDAC CONTRACT NUMBER NAS8-27202

"TECHNIQUES IN THE GENERATION
OF SUPPORT SOFTWARE"

30 APRIL 1974

PREFACE

This report is Volume I of the design specification for the Compiler Writing System. Its purpose is to introduce the compiler and target specification methodology in the form of a collection of formal languages. Other languages are presented having only a compilation support function, and are consequently invisible to the user.

Volume II should be referred to for the design of all system components.

This report has been prepared in compliance with the requirements of contract NAS8-27202, covering work done between 1 June 1973 and 30 April 1974. If additional information is required, please contact any of the following McDonnell Douglas or NASA representatives:

- Mr. G. M. Jones, Contract Negotiator/Administrator
  Huntington Beach, California
  Telephone: (714) 896-2795

- Mr. B. C. Hodges, Project COR, S&E-COMP-C
  Marshall Space Flight Center, Alabama
  Telephone: (205) 453-1385

TABLE OF CONTENTS

INTRODUCTION

Volume I of the Design Specification is contained herein. It consists of three sections plus appendices.

Section 1 describes the constructs within the Meta-Language for both Language and Target machine specification.

Section 2 presents the elements of the Function Language as to meaning and syntax.

Section 3 describes the structure of the Target Language, which represents the target-dependent object text representation of application programs.

I. META-LANGUAGE

This section describes the entire Meta-Language, encompassing statements to allow definition by the compiler writer of:

- Characteristics of the host machine on which the generated compiler is to execute;
- Characteristics of the source language input to the compiler;
- Specification of the target machine(s) on which the generated code from the compiler is to execute.

Various terms and constructs used throughout the description of the Meta-Language are defined below.

Term	Meaning
Almeric	Represents any alphanumeric character; i.e., a letter or a digit.
Letmeric	Represents a symbol beginning with a letter and followed by zero or more almerics.
[items]	This form indicates that the items enclosed within the brackets are optional, occurring once or not at all.
$[\text{items}]_n$	This form indicates optionality also, except the items within the brackets may be repeated up to 'n' times. If 'n' is left as a variable rather than specified with a digit the items are repeated an indefinite number of times.
$\left\{\begin{matrix}\text{items}\\ \text{items}\\ \vdots\\ \text{items}\end{matrix}\right\}$	Indicates selection of exactly one of the enclosed items from the vertical list.

1.0 General Meta-Language Conventions

Meta-language consists of sequence of statements in free-format syntax, each statement terminating with a period ('.') character. There are no column conventions and all 80 columns of each card image is available for a statement. No special indication of continuation is required, and successive cards (up to 25 per statement) are read until the end-of-statement delimeter ('.') is encountered. The next statement may follow immediately thereafter on the same card or the next card.

Blanks have the property that wherever one blank is allowed any number of blanks are allowed. Blanks serve as one type of delimeter between operands, and any other delimeter may optionally be preceded by blanks.

Comment strings may be inserted at any point for increased clarity or for documentation except within quoted literal strings, and are delineated by the paired sequence '/*' and '*/'. Comment strings are ignored by the Meta-Compiler but are included as part of the Meta-Language listing.

1.1 Compiler Identification

The statements described below serve to define the name of the generated compiler as well as certain parameters relating to the host computer on which its execution will take place.

Define Statement

DEFINE compilername (WORD=integer [,TABLE=integer] [,EXEC=integer] [,CARDS=integer] [,HASH=integer] [,TARGET=integer]).

compilername -- letmeric name of the generated compiler.

WORD=integer -- the number of bits in a word of the host computer.

TABLE=integer -- An optional parameter defining the size in host words of the dynamic table/stack region. If none is declared a value of 1000 is assumed.

EXEC=integer -- An optional parameter defining the size in host words of the execution control table. If none is declared a value is estimated based on the maximum syntax level encountered and the number of deferred procedure arguments encountered.

CARDS=integer -- An optional parameter specifying the size of the card image buffer (IMAGE). Its value should be 1 + maximum number of continuation lines allowed per source statement in the language being defined. If absent a value of 1 is assumed implying a language with no continuation lines allowed.

HASH=integer -- An optional parameter defining the size in host words of the storage area for packed symbol strings. The default value is 1000.

TARGET=integer-- An optional parameter defining the size of the target definition storage area. The default value is 1000.

The DEFINE statement is always the first statement of a Meta-Language definition.

Examples:

(1) DEFINE JOVIAL (HWORD=32).

The generated compiler is called 'JOVIAL' and runs on a 32 bit host computer.

(2) DEFINE FORTN (HWORD=60, CARDS=640).

The generated compiler is called 'FORTN', runs on a 60 bit host, and allows eight card images per language statement.

End Statement

END OF DEFINITION.

This statement is always the last statement of a Meta-Language compiler definition and indicates the end of processing.

Symbolic Equate

EQUATE name=integer [,name=integer]$_{n}$.

name -- a letmeric symbol

integer -- an integer (decimal, hex, or octal) constant

All occurrences of the equated names in any future Meta-Language statements are replaced by their designated value. An equated name may appear anywhere an integer value is allowed.

Example:

```
EQUATE  SIV=1, ARY=2, EXPANSION 22=22.
EQUATE MSK=X'2AF9',OCTALMASK=O'77235'.
/* NOTE HEX AND OCTAL CONSTANT SYNTAX */
```

1.2 Language Definition

Statements in this section deal with the specification of characteristics of the source language under definition. Language definition elements may be further classified as follows:

- Lexical Preprocessing Elements;
- Compiler Declarative Elements;
- Language Translation Rules.

1.2.1 Lexical Preprocessor Definition

Statements in this section serve to define distinct characters and character classes (terminals) as well as transformations to be performed on each input language card image to enable the formation of complete language statements prior to the beginning of statement parsing. The preprocessing definition causes the generation of driving program complete with all required I/O, continuation line handling, character transformations, etc., as well as the overall cycling involved in invoking parsers and deferred procedure execution.

1.2.1.1 Terminal/Character Definition

The following statement defines:

- The complete character set for the language under definition;
- The collection of symbols to be associated with subsets of the character set, called Terminals. Subsequent references to the terminal names in the body of the Meta-Language will cause a search for the associated character [s] to be made.

Terminal Statement

TERMINAL[S] (terminal name=operand[//operand]$_n$) [,(terminal name=operand[//operand]$_n$)].

terminal name -- the letmeric name of the terminal being defined within the opening and closing parens.

operand -- one of the following forms:

(1) A string of host characters delineated by two successive single quotes (''), or

(2) A previously defined terminal name, or

(3) NOT, followed by a blank, followed by an operand of type (1) or (2).

The operands define the alternative character[s] which satisfy the terminal definition. Thus, a terminal name matches a source language construct if the character specified by one of the terminal operands does.

An operand consisting of a string of host characters (type (1)) is satisfied if any one of the component characters of the string is found.

An operand which is a previously defined terminal name is satisfied if a character is found which satisfies the terminal name's definition.

NOT, followed by an operand, is satisfied if a character satisfying the operand is not detected.

The following example defines the default collection of terminals supplied by the system if no TERMINAL statements are present:

Example: [The standard default terminal definition]

```
TERMINALS  (SPECIAL = '',.=)($'*/-+''),
           (SPACE = '' ''),
           (DIGIT = ''0123456789''),
           (NONSPACE = NOT SPACE),
           (LETTER = ''ABCDEFGHIJKLMNOPQRSTUVWXYZ''),
           (ALMERIC = LETTER//DIGIT),
           (HEX = DIGIT//"ABCDEF"),
           (BINARY = "01"),
           (OCTAL = "01234567"),
           (NONQUOTE = NOT '''''),
           (CHARACTER = SPECIAL//LETTER//DIGIT//SPACE).
```

The character set of the language being defined is formed as the union of all characters found within all string operands for all defined terminals. Thus, every desired character must appear in at least one terminal definition.

1.2.1.2 Statement Structure Definition

Statements in this group define the statement fields (if any), noise and copy strings and continuation rules for the language being defined.

Field Definition

FIELD[S] fieldnumber (|*begin| - |*end|) [,fieldnumber (|*begin| - |*end|)]$_{n}$.

fieldnumber -- the field number, starting at one and incrementing by one for each defined field.

begin -- the starting column of a field on the first card of a language statement.

end -- the ending column of a field on the first card of a language statement.

* -- indicates the starting (or ending) column of a field is unknown and will be determined during the preprocessing scan for each statement.

All characters following the ending column of the last defined field on the first card image of a statement will be automatically ignored.

The isolation and detection of separate fields allows parsing of each field to take place separately in the generated parser (see 1.2.3, the NEXT FIELD statement).

Examples: [FORTRAN]

FIELDS 1 (1-5), 2(7-72). [Implies ignoring of columns 73-80]

[Meta-Assembler]

FIELDS 1 (1-*), 2(*-*), 3(*-*).

Continuation Definition

CONTINUATION FREE. (Case 1)

CONTINUATION CARD {1|2} , APPEND COLS begin-end, column operand [,column operand]$_{n}$. (Case 2)

begin -- integer starting column number.

end -- integer ending column number.

column operand -- a construct of the form:

[NOT] column number 'string'

This statement defines a continuation rule to be applied to the first and all following cards of each statement (···CARD 1···), or starting with the second card of the statement (···CARD 2...). For Case 1 a free-format card input is implied such that the end of a statement is determined at the time the statement parsing commences.

For Case 2 the APPEND...specification designates the extent of each continuation card (columns 'begin' to 'end') to be copied and to be included as part of the statement being built. Each continuation card is considered part of the last defined field in the FIELD directive (if any) and extends the ending column designator for that field. The remainder of the statement defines the continuation rule in terms of column operands separated by conjunctions (,).

A column operand specifies a string of characters, specified explicitly ('string'), that is to be found (or not found if preceded by NOT) in a designated card column position. All column operands must be satisfied before a given card image is considered to be a continuation of the previous card.

Examples:

```
    [FORTRAN]
CONTINUATION CARD 2, APPEND 7-72, NOT COLUMN 6 '', NOT COLUMN 6 '0'.
    [META-ASSEMBLER]
CONTINUATION CARD 1, APPEND 1-71, NOT COLUMN 72 ''.
    [SPL]
CONTINUATION FREE.
    [META-TRANSLATOR]
CONTINUATION FREE.
```

Prescan Activity

```
PRESCAN element [//element].
                 n
      element -- operand [,operand]
                        n

      operand -- [NOT] COLUMN number {'string'}                    (1)
                                     {terminal[s]}

                 [IF][NOT] 'string'                                (2)

                 [IF][NOT] terminal[s]                             (3)

                 IF iexp relop iexp                                (4)

                 [IF][NOT] VALUE                                   (5)

                 integer replacement                               (6)

                 COPY THRU {string  }                              (7)
                           {terminal}

                 COPY {integer [CHARACTERS]}                       (8)
                      {VALUE               }

                 COPY 'string'                                     (9)

                 SKIP {integer [CHARACTERS]}                       (10)
                      {VALUE               }

                 SKIP THRU {string  }                              (11)
                           {terminal[s]}

                 SET NEXT FIELD                                    (12)
```

A sequence of PRESCAN statements allow specific actions to take place during the preprocessing of a statement. At preprocessing time a single scan is made over each input card image in a left-to-right manner. Each character or character sequence is examined to see if it matches the conditions set forth in a PRESCAN statement. If so, the indicated action is performed. If not, the character is copied into the statement as is.

Each element of a PRESCAN is applied in turn to the current input character until the conditions described by the conditional element operands are detected. The SKIP or COPY action operands of the element (if any) are then applied, and the next PRESCAN activity is examined for applicability.

Operand types (1) - (3) above are satisfied whenever the indicated string or terminal occurs starting in a specific column (type 1) or the current cursor column of the pre-scan. A preceding 'NOT' requires the string or terminal to be absent. If the operand is satisfied the prescan cursor is advanced beyond it so that further operands can be checked for or actions can take place, unless the operand was preceded by 'IF,' in which case the precursor is not changed. If a terminal is searched for and the indicated terminal name is followed by an 's', the operand consists of zero or more occurrences of the characters satisfying the terminal definition until a character is found which does not satisfy it.

Operand type 4 is satisfied if the specified relationship exists between the two integer arithmetic expressions. The six relational operators allowed are: EQ, NE, LT, GT, GE, LE.

Operand type 5 searches for a sequence of digits and computes the integer value they represent. The computed value is associated with the VALUE designator which may subsequently be used in SKIP or COPY directives.

The remaining type represent specific actions to be performed as follows:

(6) the integer replacement into a variable is performed.

(7) the string of input characters ending with the designated string or terminal is copied directly into the statement being formed.

(8) the designated number of characters (integer or previously computed VALUE) are copied directly.

(9) the indicated string of characters is copied directly.

(10) the designated number of characters (integer or previously computed VALUE) are ignored.

(11) all characters up to and including the occurrence of the designated string or terminal are ignored.

(12) the current column in the statement being formed is marked as the beginning column for the next field (see FIELD directive).

If a sequence of PRESCAN statements occur the conditions and actions they define are performed in the order presented. Thus, a SKIP action may cause characters to be ignored which satisfy a copy condition which occurs in a later PRESCAN.

Example:

```
    [FORTRAN]
PRESCAN  '(',LEV=LEV+1//')', LEV=LEV-1//
         '=', IF LEV EQ 0, NEQ=1//',', IF LEV EQ 0, NCOM=1.
PRESCAN IF ''', COPY THRU '''//SPECIAL, SPACES, VALUE, 'H',
    COPY VALUE CHARACTERS.
PRESCAN IF SPACE, SKIP SPACES.
```

The first PRESCAN above detects the presence of an equal sign or comma not enclosed within parens in each FORTRAN statement. The flagged information (NEQ and NCOM) may be interrogated by the statements in the body of the Metalanguage as an aid to statement recognition (i.e., DO, replacement statements, statement functions, ...).

The second PRESCAN statement causes quoted or FORTRAN hollerith strings to be copied as is.

The third PRESCAN statement causes spaces to be skipped which are not in quoted or hollerith strings.

Example:

```
[Metalanguage itself]
PRESCAN    SPACE, SKIP THRU SPACES//IF '/*', SKIP THRU '*/'.
PRESCAN    IF QUOTE, COPY THRU QUOTE.
```

Note that whenever a space is detected, all succeeding spaces are ignored.

1.2.2 Compiler Declaratives

Statements in this section appear together prior to the body of the Metalanguage rules. The declaratives define attributes of the language being defined and certain parameters affecting the physical structure of the generated compiler.

1.2.2.1 Expression Related Attributes

If the language being defined allows expressions of any type to occur, certain characteristics of their operators and operands should be specified.

Array Storage

STORAGE $\left|\begin{matrix}\text{COLUMN}\\\text{ROW}\end{matrix}\right|$ BIAS $\left|\begin{matrix}0\\1\end{matrix}\right|$.

The direction of variance of the subscripts for arrays utilized in the language is specified as well as computational bias. A COLUMN specification implies allocation of the array in the target memory such that the indices to the left vary most frequently, i.e., A(I,J,K,...) is followed by A(I+1,J,K,...), etc.. The ROW specification implies the opposite case of the rightmost indices varying most frequently.

The BIAS value specifies whether the first array cell is referenced by a subscript value of zero or one. Thus, the reference A(I) becomes the contents of the location of A plus I minus the bias value.

The absence of this statement causes a default of row allocation with a bias of zero.

Example:

```
[JOVIAL]
STORAGE ROW BIAS 0.
```

Mode Specification

MODES modenum [,modenum].

modenum -- integer value or equate name.

The various computational modes for expression operands in the language are specified by mode numbers having the following pre-defined meanings:

Mode Number	Computational Mode
0	Integer
1	Fixed Point
2	Floating Point, or Real
3	Complex
4	Logical
5	Boolean
6	Texual
7	Status
8	Contexual
9	Location
10	Binary (secondary mode)
11	Octal (secondary mode)
12	Hex (secondary mode)

Examples:

```
EQUATE  INT=0, FIX=1, REA=2, CMP=3, LOG=4, BOL=5, TEX=6, STA=7, TMP=8,
        LOC=9, BIN=10, OCT=11, HEX=12.
MODES   INT, REA, CMP, LOG.                          [FORTRAN]
MODES   INT, FIX, REA, LOG, BOL, TEX, STA, TMP, LOC, BIN, OCT, HEX.    [SPL]
```

Operator Definition

OPERATORS number[-[hierarchy]-[commut]-[assoc][,number[-[hierarchy]-[commut]-[assoc]]]$_{n}$.

number -- the operator number

hierarchy -- the order of computation of the operator relative to other operators, with higher valued operators computed first.

commut -- commutivity flag - 0 = not commutative
1 = commutative

assoc - associativity flag - 0 = not associative
1 = associative

This statement allows specifying the allowed language operators and defines attributes for each to control the order of operator execution as well as optimization processes (see 1.4). The list of available operators by number is as follows:

NUMBER	OPERATOR	DEFAULT: HIERARCHY	COMMUTIVITY	ASSOCIATIVITY
1	Addition (+)	8	1	1
2	Subtraction (-)	8	0	0
3	Multiply (*)	9	1	1
4	Divide (/)	9	0	0
5	Exponentiate (**)	10	0	0
6	Replacement (=)	0	0	0
7	Exchange (==)	0	0	0
8	Relational equal	5	1	-
9	Relational not equal	5	1	-

NUMBER	OPERATOR	DEFAULT: HIERARCHY	COMMUTIVITY	ASSOCIATIVITY
10	Relational greater or equal	5	*0	-
11	Relational greater	5	*0	-
12	Relational less than	5	*0	-
13	Relational less or equal	5	*0	-
14	Boolean and	3	1	1
15	Boolean or	2	1	1
16	Boolean exclusive or	2	1	1
17	Boolean equivalence	1	1	1
18	Boolean negate (not)	4	-	-
19	Logical and	7	1	1
20	Logical or	6	1	1
21	Logical exclusive or	6	1	1
22	Matrix inversion	-	-	-
23	Matrix transpose	-	-	-

* These operators can be considered commutative through suitable transformations, i.e., A GE B = B LT A.

The values shown in the above table are the defaults for missing hierarchy, commutivity, and associativity flags. The absence of this statement will cause all language operators to be activated with their default flags.

Example:

```
EQUATE  PLUS=1, MINUS=2, MULT=3, DIV=4, EXPON=5.
OPERATORS  PLUS-1-1-1, MINUS-1--0, MULT-2-1-1, DIV-2-0-0, EXPON-3-0-0.
```

This example defines a language with only arithmetic operators to be performed under the normal rules of operator hierarchy.

Invalid Mode Combination Specification

ILLEGAL MODES ([ms[-me]],os[-oe],[ms-[me]])[n,([ms-[me]],os[-oe],[ms-[me]])].

ms -- starting mode number.

me - ending mode number.

os -- operator starting number.

oe -- operator ending number.

Each parenthesized operand designates a collection of operators (os-oe) or a single operator by number which may not act upon two operands having the computational modes indicated. If either computational mode range indicator (ms-me) is missing the values 0-00 are implied, meaning any operand mode. For unary operators the left operand is not supplied.

Example: [Using operators and mode equates from previous examples]

```
ILLEGAL MODES  (LOG, PLUS-EXPON,),          /* LOGICAL WITH ARITHMETIC*/
               (,PLUS-EXPON,LOG),
               (INT-CMP,LAND-LXOR,),
               (,LAND-LXOR,INT-CMP),
               (INT, EXPON, REA-CMP),       /*INTEGER**NON INTEGER*/
               (LOG, REQ-RLE,),             /*LOGICAL WITH A
               (,REQ-RLE, LOG).             /*RELATIONAL OPERATOR*/
```

1.2.2.2 Symbol/Hash Table Declaration

SYMBOL ATTRIBUTES usagenum[,size][n,([attribname[([S,] word-bitstart-bitlen)],])

[n,usagenum...].

usagenum - symbol usage number

size - The estimated number of language symbols of type usagenum.

attribname -- A symbolic attribute name.

word -- optional host word position to store the attribute value.

bitstart -- optional bit starting position within the word.

bitlen -- optional bit length of the attribute field.
S -- indicates a signed attribute value.

This statement allows the attributes of a symbol of type 'usagenum' to be specified as well as the specific bit layout of the entry in contiguous memory words. The absence of this statement causes the default attributes and symbol table structure outlined in Appendix A according to the host memory word size on the DEFINE card (1.1).

If the size*option is absent a pre-defined value is chosen per symbol type as defined in Appendix A. Each attribute name must correspond to one of the universal system-defined names indicated in Appendix A. Any unspecified names are set to zero whenever a symbol of type 'usagenum' is unpacked. If no word and bit specifier are defined for a particular attribute name, the field is automatically allocated.

Examples:

```
EQUATE  SIV=1, ARY=2, PRO=4.
SYMBOL ATTRIBUTES  SIV, 500, ARY, 200.                                    (1)
SYMBOL ATTRIBUTES   SIV, 500,(IM, REL,PREC),ARY,25,(IM, REL, PREC).       (2)
SYMBOL ATTRIBUTES    SIV,200,(IM(0-0-4),REL(0-4-4),PREC(0-8-8)),ARY,20,PRO,22,
                     (Im(0-0-4),REL(0-4-4),PRE(0,8,8),TYPE(0-27-3)).      (3)
```

Example one specifies table space for 500 simple variables and 200 arrays with standard attributes and host word allocation. Example two specifies the relevant attributes for each array or variable. Example 3 defines array, procedure, and variable symbol usages with specific attributes and bit layouts.

*--All size estimates for symbol table or hash table declarations need only be approximate. The dynamic symbol storage area management provides for varying table lengths depending on encountered symbols.

Hash Table Declaration

TABLE[S],name[(size,ptr[,attribname([S,]word-bitstart-bitlen)]
                    n
    [,name...].
    n

name -- the letmeric hash table name.

size -- the estimated number of table entries.

ptr -- the name of a cell containing the symbol sequence number of the current symbol entry being manipulated.

attribname -- an optional letmeric attribute name.

word -- optional host word position to store the attribute field.

bitstart -- optional bit starting position in the word.

bitlen -- optional bit length of the attribute field.

s -- sign flag.

A collection of symbolic hash tables may be declared for storing arbitrary symbols. A corresponding parallel attribute table for each may also be specified with a variable number of attribute fields with specified allocation within host words. Both a hash table and its corresponding attribute table (if any) are indexed through the symbol sequence number, which ranges from 1 to 'size.' A signed attribute is indicated by a sign flag [S,].

Examples:

```
TABLES /*LOOP TERMINATORS*/,LTERM(100,LTPTR,LTERML).
TABLES/*SET SYMBOLS*/,SETS(250,SSPTR,SSL,VALUE(0-0-32),
    INITFL(1-0-1)),/*NAMES*/,NTAB(500,NPTR,NLEN).
```

1.2.2.3 Miscellaneous Declarations

Stack Definition

STACK[s],name(size,index)[,name...]
n

name -- the symbolic stack name.

size -- the estimated number of stack words.

index -- the symbolic name of a cell containing the offset pointer to the last (most recently entered) stack word.

A collection of last-in-first-out stacks consisting of contiguous host words may be defined. The two stacks PSTACK and QSTACK are predefined and need not be declared; if explicitly declared only their size estimate can be changed.

Example:

```
STACK/*SUBSCRIPTS*/,SUBST(50,SPTR),/*POLISH STRING*/,PSTACK(500,PPOINT).
```

Array Declaration

ARRAYS[name(dim[,dim])][,name(dim[,dim])].
n

name -- the name of a host array.

dim -- integer dimension specifier.

Arrays consisting of contiguous host words are defined with indicated dimensionality. The number of dimension specifiers for a given array must not exceed two.

Example:

```
ARRAY X(10),I(2,3).
```

Error Message Declaration

ERROR MESSAGE[S] 'message' [,'message'].
n

Each message imbedded within quotes represents an error diagnostic to printed out for an error element, described in section 1.2.3. The ith error number will be associated with the ith message declared within an ERROR MESSAGE statement.

Data Declaration

DATA name[,name]/integer[,integer]/[name....]
n n

name -- an array name or variable name.

integer -- the initialization values with an optional repeat factor:

integer * integer

The syntax and meaning of the DATA declarative is the same as for the data statement in FORTRAN IV. An array name appearing within a DATA declarative must have been previously declared with an ARRAY declarative.

Debug Aid Selection

```
DEBUG |ON|  { STEP |UP|  [TRUE]  }  [ ,  |ON|   ...]
      |OFF|  {      |DOWN|        }  n  |OFF|
             {
             { DUMP [FALSE]
             { REOCCUR
             { TERMINAL
             { FL
             { PROC
```

This declaration allows selection of a number of trouble shooting aids by the compiler writer to be imbedded into the generated compiler. Each option specifies a debug aid to be selected or de-selected by its preceding ON or OFF modifier; the default setting for each option is OFF. Once [de]selected the condition remains in effect until modified by a DEBUG element within a rule definition (see 1.2.3.3).

OPTION	*EFFECT
STEP {UP / DOWN} [TRUE]	Trace step-ups or step-downs. A print line is generated indicating a step-up or step-down has occurred with a display of the mlv number, syntax level, and cursor position. The TRUE option causes only monitoring of the TRUE step-ups.
DUMP [FALSE]	Provide a memory dump of all tables, stacks, arrays, and cells upon returning to the zero syntax level for each parsed statement. The FALSE option inhibits the dump unless the return is false; i.e., an invalid statement.
REOCCUR	Traces all step-into and step-out of reoccurrence elements.
TERMINAL	Prints for each detected terminal operand the terminal number, its cursor position, the reoccurrence level, and the syntax level.

* - Terms discussion herein are defined within section 1.2.3.

OPTION	*EFFECT
FL	Dump all generated Function Language. Provide a readable listing of all F.L. Terms generated for each language construct.
PROC	Prints all step-downs and step-ups from code generative PROC expansions.

Statement Level Declaration

STATEMENT[S][ONCE]([MODULE,][[ONCE] $\left\{\begin{matrix}\text{mlv} \\ \text{literal}\end{matrix}\right\}$ [=mlv][[MODULE,]$_{n}$

[,[ONCE]$_{n}$ $\left\{\begin{matrix}\text{mlv} \\ \text{literal}\end{matrix}\right\}$ [=mlv]][])[,[ONCE]...]$_{n}$.

mlv -- the name of a language rule (see 1.2.3 following).

literal -- a character string (1.2.3) identifying a language statement.

This directive allows the following to be specified:

- The processing level for each distinct language statement.

  Typically, a language statement may occur in a specified sequence relative to other statements in the language. A processing level, identified above by each othermost paren grouping, is composed of a collection of statement processors occurring at the same level. The right operand mlv is the name of the processing rule for a given statement; the left operand (mlv or literal) is the recognition criteria for the statement.

  A statement may occur at more than one level, such as the FORMAT statement in FORTRAN. In that case the associated processing rule will appear more than once in the rule definition section (1.2.3).

  The absence of this statement causes all statements to be recognizable at all levels, i.e., no order dependence is present in the language.

- The occurrence frequency for each statement.

  A ONCE prefix for a level means that only one enclosed statement will be allowed at that level. A ONCE prefix for an individual statement means only one occurrence of that statement is allowed at that level.

- The inclusion of an arbitrary mlv at a level.

  If a left operand mlv is specified with no right operand (processing mlv), the mlv is simply to be included in the generated module representing that level.

- The separation of mlvs into modules.

  Each processing level causes a distinct module (FORTRAN subroutine) to be generated composed of the code for the statement processor mlvs at that level. Further division of a level into further modules may be accomplished through a MODULE operand, which causes the following statements at that level (until the next MODULE operand) to be included in a separate module.

Examples:

```
[FORTRAN]
STATEMENTS ONCE ('FUNCTION'=$FUNCPROCESSOR,'SUBROUTINE'=
  $SUBPROCESSOR,'PROGRAM'=$PROGPROCESSOR,'BLOCKDATA'=$BCPROCESSOR,
  $NULL=$PROGPROCESSOR),ONCE('IMPLICIT'=$IMPLICIT),
  ('COMMON'=$COMPROCESSOR,'DIMENSION'=$DIMPROCESSOR,
  'EQUIVALENCE'=$EQUIVPROCESSOR,MODULE,$MODESPEC=$MODEPROCESSOR).
```

1.2.3 Language Definition Rules

This section describes the body of a language definition which consists of a collection of Meta-Language Variable (MLV) specifications. Each MLV typically corresponds to a construct of a language, defining both the syntax and semantics of the construct in terms of other MLV's, syntax operands, and elements. Each MLV is identified with a syntax level during the parsing process, such that an MLV called from another MLV is at a syntax level one greater than the calling MLV. The MLV at syntax level 0, termed the 'head node', is the point at which a language specifications begins.

The first two MLVs defined have special meaning differentiating them from the remaining definition rules. The first MLV must be named INITIALIZER and allows the compiler writer to specify certain semantic functions to be performed once at the beginning of a compilation. These tasks include clearing flags, reading predefined table or stack entries, and initializing arrays.

Following the INITIALIZER rule is the LABEL rule defining the parsing and handling of statement labels in the compiler input language being defined. This rule is automatically invoked at the start of each statement parse and is responsible for processing any label and positioning the cursor beyond it.

Following the label rule are the individual statement parser rules and their subordinate rules (see STATEMENTS statement).

Each MLV has associated with it certain attributes at any given time during a statement parse. When invoked from another MLV, for example, it is considered 'true' of the construct represented by its definition rule is detected in the input statement at the current cursor position. In that case the MLV becomes associated with the satisfying character string in the input statement.

One may think of the entire set of definition rules (MLV's) as being 'executed', or applied to each statement of a language starting at the head node and moving to deeper levels of syntax. The entire statement is 'true', or successfully parsed, if a true exit is made out of the head node.

1.2.3.1 MLV Operands

The following operand types may occur within the body of an MLV definition. They are manipulated within the elements and strings defined in the next section, being referred to therein by the indicated name.

Literal Operand--literal

Syntax: 'characters'

The literal operand consists of a set of language characters enclosed with quotes. A quote character within the literal is allowed only if it stands alone, i.e., '''' is the literal operand for a single quote character in the language.

Examples: 'ABC'

''''

'FORMAT C'

MLV Reference Operand--mlv

Syntax: $letter[almerics]
n

This operand is a symbol to be identified with the name of an MLV. Only the first 20 letmeric characters are used and must therefore be unique.

Examples: $VARIABLE
$FORMATPROCESSOR

Simple Variable Operand--svar

Syntax: letter[almerics]
5

This operand corresponds to an integer variable in FORTRAN, occupying one word or cell containing a value.

Examples: ITEMP
XYZ
T1

Data Variable Operand--dvar

Syntax: svar[(iexp [,iexp])]

This operand corresponds to a simple variable, an array reference, or a function reference in FORTRAN. The iexp operands are integer expressions as defined below.

Examples: ABC
I(2*J+5,K)
FUNC(5, K* IABS(J-K))

Procedure Name Operand--pname

Syntax: letter[almerics]
5

This operand is the name of a procedure, or semantic processing subprogram. It is referenced within an integer expression followed by a parameter list, or is linked to by the Procedure Reference element (1.2.3.3).

Examples: FUNC(I,J)

Hash Name Operand--hashname

Syntax: letter[almerics]
5

This operand is the name given to a declared hash table for storing symbol strings (1.2.2.2).

Stack Name Operand--stackname

Syntax: letter[almerics]
5

The stackname is the name of a declared program stack or list (1.2.2.3).

Terminal Name Operand--terminalname

Syntax: letter[almerics]
n

This operand is the name associated with a terminal representing a specific character string. Terminals are declared in the lexical preprocessing section (1.2.1.2).

Integer Expression Operand--iexp

Syntax: $\overset{+}{-}$eoperand $\left[\underset{n}{\left\{\begin{matrix}+\\-\\*\\/\\**\end{matrix}\right\}}\right.$ eoperand]

The integer expression operand is formed in the same manner as an integer expression within FORTRAN, except that the allowed component expression operands (eoperand) are extended to include the following:

- integers
- (iexp)
- dvar
- POSITION OF mlv
  The cursor position of the last input string satisfying the mlv is implied;
- SIZE OF mlv
  The number of characters in the last input string satisfying the mlv is implied;

- VALUE OF [{HEX / BINARY / OCTAL / iexp}] mlv

  A character to value conversion is performed on the string satisfying the mlv. The optional computational base may be specified as hexadecimal (HEX), binary (BINARY), octal (OCTAL), or other (iexp);

- INDEX OF {mlv / literal}

  The position of the literal operand in the literal vector, or the internal sequence number of the mlv operand, is implied;

- INT OF 'character'

  The internal value of the single enclosed language character is implied. All characters are converted from external character codes to an internal sequence number prior to any character manipulation;

- TYPE OF mlv

  The type of the symbol satisfying the mlv is implied. The symbol type is the sector number within the Symbol Table for the sector containing the symbol. If the symbol is not in the Symbol Table a value of zero is implied;

- NUMBER OF terminal[S]

  The number of terminals detected during the last terminal element request is implied.

Examples:
```
$EX1.=I=J-1+X(I*7-4,3)*ISIGN(I*J-K**2/3).
$EX2.=.L FUNCTION(2-(X-Y*J)+K,1).
$EX3.=$SYMBOL,I=POSITION OF $SYMBOL*2 + SIZE OF SYMBOL.
$EX4.=$GHEX,J=VALUE OF HEX $GHEX+1.
$EX5.=I=INDEX OF 'SINF'-INDEX OF 'COSF'.
$EX6.=$SYMBOL,IU=TYPE OF $SYMBOL.
$INTEGER.=1 TO 10 DIGITS,I=NUMBER OF DIGITS.
```

Procedure Call Argument Operand--arg

Syntax: iexp
mlv
literal
hashname (iexp)
ELEMENTS [iexp THRU iexp] OF stackname
stackname (iexp)

This operand is always used as an argument to an immediate or deferred semantic function. The meaning of each argument and the information passed to the called routine are described below.

- iexp
  The integer expression valued is passed;
- mlv
  The input character string associated with the mlv operand is passed;
- literal
  The literal operand character string is passed;
- hashname (iexp)
  The symbolic character string occupying the 'iexp' entry of the hash table named 'hashname' is passed;
- ELEMENTS [iexp THRU iexp] OF stackname
  The contents of the stack named 'stackname' is passed as a stream of integer values. A contiguous portion of the stack may be passed by specifying the lower and upper positions within the stack through the use of the two 'iexp' designators.
- stackname (iexp)
  The value contained in position 'iexp' of the stack named 'stackname' is passed.

1.2.3.2 MLV Definition Syntax

```
$mlv.=string[//string].
          n
     mlv--letmeric mlv name.
     string--element[,element]
                    n
     element--[NOT] Mlv stepdown element          (1)
              [NOT] Literal element               (2)
                    Scan element                  (3)
              [NOT] IF-test element               (4)
                    Reoccurrence element          (5)
                    Replacement element           (6)
                    Case element                  (7)
                    Procedure reference element   (8)
                    Put element                   (9)
                    Push element                 (10)
                    Pop element                  (11)
                    Set element                  (12)
              [NOT] Terminal element             (13)
                    Null element                 (14)
                    False element                (15)
                    Chain element                (16)
                    Fortran element              (17)
                    Test element                 (18)
                    Field element                (19)
                    Attribute element            (20)
                    Error element                (21)
                    Table element                (22)
                    Debug element                (23)
                    Free Table element           (24)
                    Save/Restore Element         (25)
```

An mlv is true if and only if one of its component strings is true. The strings are evaluated in a left-to-right manner. A string is true if every one of its elements is true. The elements consist of operators and operands optionally preceded by a 'NOT' modifier, which reverses the truth value for that element. If an element is true, the cursor is automatically moved beyond the statement input characters represented by the element. A false element causes the cursor to be set back to its value at the beginning of string containing that element. Thus, a false application of an mlv does not move the cursor, while a true application moves the cursor beyond the last element of its first true string. The elements are described below by number.

1.2.3.3 MLV Element Descriptions

(1) Mlv stepdown element

Syntax: $mlv

The mlv name identifies a definition which is to be applied, or 'stepped down into,' at another syntax level. The truth value of this element is that returned by the applied mlv. The stepdown may be recursive to the same mlv name as that being defined.

Examples:
```
$EX1.=$NAME//$INTEGER.
$EX2.='+', $EX2//'-',$EX2.
```

(2) Literal Element

Syntax: 'characters' [i.e., literal operand]

The enclosed language characters are searched for starting at the current cursor position in the statement image. The element is true if the characters are found exactly in the order specified.

Examples:
```
$EX1.='COMMON'//'DIMENSION'.
$EX2.='/',$SYMBOL,'/'//',.
```

(3) Scan Element

Syntax: SCAN FOR literal

The input statement being parsed is scanned from left-to-right for the indicated literal operand, starting at the current cursor position. If the literal is found, the element is true and the cell LTSCAN is set to its starting position in the input image, but the cursor is not moved for this element.

Example:
```
$EX.=SCAN FOR '=','A','=','B'//'SET'.
```
This mlv parses the constructs:
```
A = B
SET
```

(4) IF-Test Element

Syntax: IF { literal / terminalname } (a)

iexp relop iexp (b)
mlv IN hashname (c)
mlv { EQ mlv / NE } (d)
iexp ON stackname (e)
mlv TYPE iexp (f)
NEW SYMBOL (g)

This element tests for the validity of a relationship as applied to its operands. If the relationship is true the element is true.

(a) IF { literal / terminal }

The literal or terminal operand represents a character string tested for occurrence starting at the current cursor position. The element is true if the characters are found. No movement of the cursor is involved. Note that the IF operand may stand alone as an element type ((1),(13)), the difference in that case being the cursor movement if the operand character string is found.

Examples: IF 'ABC'
IF $SYMBOL
IF DIGIT

(b) IF iexp relop iexp

relop -- { LT / GT / NE / EQ / LE / GE }

The two integer expressions are algebraically compared for the indicated relational condition. No cursor movement is involved.

Examples: IF A-B GT C-D*E
IF SIZE OF $X LT 25-I

(c) If mlv IN hashname

The input character string corresponding to the mlv is searched for as an entry in the indicated hash table. The element is true if the string is found, and the symbol sequence number (SSN) and symbol length are placed in the two control cells defined in the hashtable declaration. The table number and SSN are also placed in the global cell SYMP. No cursor movement is involved.

Example: IF $VARIABLE IN SYMTAB

(d) IF mlv $\begin{cases} EQ \\ NE \end{cases}$ mlv

The two input character strings represented by the mlv's are compared for character-by-character identity (EQ) or non-identity (NE). No cursor movement is involved.

Example: IF $SYMBOL EQ $VARIABLE

(e) IF iexp ON stackname

The value indicated by 'iexp' is searched for on the stack indicated by stackname. The element is true if the value is found, and its stack position is placed in the cell IPOS.

Example: IF A-B+C ON PSTACK

(f) IF mlv TYPE iexp

The symbol represented by the mlv is searched for in the compiler Symbol Table segment associated with the type 'iexp'. The element is true if the string is found, and its position pointer is placed in the global cell SYMP. No cursor movement is involved.

Example: $EX.=$NAME,IF $NAME TYPE 2.

(g) IF NEW SYMBOL

If the previous PUT function (see element a) of a hash table or the Symbol Table dealt with a symbol not previously in that table, the element is true. No cursor movement is involved.

Example: ..., PUT $X IN TAB1, IF NEW SYMBOL...

(5) Reoccurrence Element

Syntax: $\left\{\begin{array}{l}\text{[LIST OF iexp] (string[//string]}_{n}\text{)} \\ \text{[iexp[TO iexp]](string[//string])}\end{array}\right.$

string--element$_{n}$[,element]

This element is in effect a 'sub-definition' in that arbitrary definition strings consisting of any element types, including other reoccurrence elements nested to any level, may be enclosed within the parentheses. A reoccurrence element standing alone with no prefix modifiers is true if one of its enclosed strings is true, as is the case for a full mlv definition. When prefixed by one integer expression (iexp), the enclosed entity is to be applied repeatedly exactly 'iexp' times, i.e., the reoccurrence is true only if 'iexp' successive occurrences of the enclosed entity are found in the input statement. When prefixed by two expressions both a minimum and maximum number of occurrences must be present as specified by the two expressions.

The 'LIST OF iexp' prefix is a short form for a commonly occurring type of reoccurrence, defined as: LIST OF iexp(...) = (...), 0 TO iexp-1(',',...). Thus, from 1 to 'iexp' occurrences of the enclosed entity is required, with a language comma (,) character separating each if more than one is to be found.

In all cases the cursor is moved beyond the source characters corresponding to last successful occurrence of the enclosed entity if the total reoccurrence is true; otherwise, the cursor is not moved.

Examples: (1) $EXAMPLE.='FOR', ('MAT'//'MULA'//'TRAN'//NULL)

This MLV parses the constructs: FORMAT
FORMULA
FORTRAN
FOR

(2) $ARGS.=1 TO 3 ('A',('1'//'2')//'B'//'C')

This MLV will parse: BCA1
A2
CBC
BA1B

(3) $ARGUMENT PROCESSOR.=LIST OF 10($SYMBOL,('C',LIST OF 3($INTEGER),
')'//NULL)).

This MLV parses the construct: A,B(3,4),D,E(1,2,3),I,J,K,L,M,N

(4) $X.=5('A')//'B',$X.

This MLV will parse: AAAAA
BAAAAA
BBAAAAA
BBBAAAAA
. . . . .
. . . . .

(6) Replacement Element

Syntax: dvar=iexp

The cell identified by 'dvar' is replaced by the value of the integer expression 'iexp'. This element is always considered true. No cursor movement is involved.

Examples: $EX1.=A=B+C,D=E,$X.
$EX2.=I=J*7+FUNC(3+K).
$EX3.=U(I*3+9)=J-1,J=POSITION OF $EX1-1.

(7) Case Element

Syntax: CASE iexp OF (element [,element])
n

The case element allows the selection of a particular element from a list of elements. The 'iexp' must range from 1 to the number of listed elements and is used to select the particular element to apply. The case element is true if the application of the selected element is true. Any cursor movement is tied to the cursor movement, if any, of the selected element. Thus,

```
CASE iexp OF (element[,element])=(IF iexp EQ 1,element1//
IF iexp EQ 2,element2 //....IF iexp EQ n,elementn)
```

Note that the comma (,) is a separator for each case element rather than an 'element truth conjunction' between two elements.

(8) Procedure Reference

Syntax: $\left\{\begin{matrix}\text{*pname} \\ \text{.L[INK]}\left[\left([\text{svar},][\text{iexp}]\right)\right]\ \text{pname}\end{matrix}\right\}\ \left[\left(\underset{n}{\text{arg}[,\text{arg}]}\right)\right]$

This element allows a semantic procedure to be called with optionally supplied arguments. The call may be made on an immediate or deferred basis.

An immediate call (*pname...) format causes a call to be made to routine 'pname' at the time the element is encountered. If execution of 'pname', which is typically a system-defined semantic routine, causes error conditions to be detected the element is considered false, otherwise it is considered true.

A deferred call (.L[INK]...) format causes a call to the routine 'pname' to be stacked away for future execution. All such deferred calls are ordered by chain association and priority within a chain. The two optional quantities in the deferred call syntax ('svar' and 'iexp' above) define a particular chain and a priority within the

chain, respectively. (See chain element, type 16). Those calls with the highest priority values within a deferred chain will be executed first. The absence of a priority parameter causes a priority of one to be used; the absence of a chain specifier causes the deferred call to be associated with the currently active chain.

The arguments passed to the 'pname' for future execution are always the current values of the arguments at the time the deferral request is made; the argument names may be subsequently modified without effecting the deferred call in any way.

The actual execution of the stacked calls for all deferred procedures is tied to exiting from the mlv corresponding to the head node, or first mlv defined, at the end of parsing a statement. If the head node mlv exits false, all the deferred calls are thrown away. If a true exit is made (i.e., the entire statement parse is true) the deferred calls set up by 'execution' of the definition string causing the mlv to be true are executed by priority within a chain, and by order of chain formation during the statement parse.

It should be noted that any deferred calls set up during a string 'execution' within an mlv or reoccurrence element are thrown away if the string becomes false. Thus, only deferred calls set up through the true paths (if any) of a statement parse are active at the time the calls are actually made.

Examples: (1) `$EX1.='A',.L FUNC(1),'B'//'A',.L FUNC(2),'C'.`

The above definition applied to:	produces the deferred calls:
AB	FUNC(1)
AC	FUNC(2)
AA	NONE

(2)
```
$EX2.=$SYMBOL,.L(2)F1($SYMBOL,7),.L(22)F2(SYMP,-1)//
$INTEGER,.L F3($INTEGER),.L(5)F4(0).
$SYMBOL.=LETTER,0 TO 5 ALMERICS.
$INTEGER.=1 TO 10 DIGITS.
```

The above definition applied to:	produces the deferred calls:
NAME	F2(SYMP,-1) F1('NAME',7)
12357	F4(0) F3('12357')

(9) Put Element

Syntax: PUT $\begin{Bmatrix} \text{mlv} \\ \text{literal}[,\text{literal}]_{n} \end{Bmatrix}$ $\begin{Bmatrix} \text{IN hashname} \\ \text{TYPE iexp} \end{Bmatrix}$

The character string[s] represented by the literal operand[s] or the mlv operand are placed either in the hash table indicated by 'hashname', or the Symbol Table segment identified by type 'iexp'. The element is true if the table does not overflow. For hash tables, the associated running pointer cell is set to the resulting symbol sequence number. The symbol sequence number and hash table or symbol segment number (iexp) are placed in the global cell SYMP, uniquely identifying the table and symbol position within.

If more than one literal operand is present, the effect is the same as for repeated PUT functions for each literal.

Examples:
```
$EX1.=$NAME,PUT $NAME IN LOCIDT.
$EX2.=$SYMBOL,PUT $SYMBOL TYPE 4.
$EX3.=PUT 'SIN','COS','TAN','SQRT' TYPE 4.
```

(10) Push Element

Syntax: PUSH $\begin{Bmatrix} \text{iexp} \\ \text{mlv} \end{Bmatrix}$ $\begin{Bmatrix} \text{IN} \\ \text{ON} \end{Bmatrix}$ stackname [(iexp)]

The operand is placed on the end of the declared stack identified by 'stackname', or is placed in a particular position on the stack (for 'stackname(iexp)'). For an expression operand (iexp) the single host word expression value is placed on the stack. For an mlv operand two words of information are stacked: the starting position of the characters satisfying the mlv in the statement image buffer, and the character length.

If the operand is placed on the end of the stack, the stack index cell specified in the stack declaration is incremented, and therefore always points to the last meaningful stack element. This element is true if the index does not exceed the declared stack length.

Examples:
```
PUSH -10 ON PSTACK
PUSH SIZE OF $MLV ON STACK1
PUSH $MLV ON S1(22)
PUSH $T2 IN S3
```

(11) POP Element

Syntax: POP stackname[,stackname]
n

This element causes the index cell of each 'stackname' argument to be decremented by one, thus effectively dropping the last stack entry for each. The element is true if the decremented index is not negative.

Examples:
```
POP PSTACK
POP S1,S2,S3
```

(12) Set Element

Syntax:
```
SET { mlv=mlv                                  }   (a)
    { mlv=STRING(arg[,arg])                    }   (b)
                       n
    { CASE svar=(element[,element])            }   (c)
                              n
```

There are three forms of the set element:

(a) SET mlv=mlv

This form allows the mlv on the left to be identified with the input characters satisfying the mlv on the right. This element establishes the mlv being set, which may or may not have a formal definition rule of its own.

Examples:
```
$X.=SET $VARIABLE=$SYMBOL.
$EX.=$INTEGER,SET $T1=$INTEGER,$INTEGER.
```

(b) SET mlv=STRING(arg[,arg])
n

The mlv is associated with a character string formed as the left-to-right concatenation of the strings represented by the arguments. An integer-valued argument (constant, iexp, or stack cell) is converted from binary to character format in base ten. This element is always true unless the formed string overflows the image buffer.

Examples: (1) SET $X=STRING('ABC',11,4*3+7,'','*')
The string associated with $X: ABC1119*

(2) Assuming $SYMBOL is associated with 'XT17':
SET $SYMBOL=STRING('ALPHA',$SYMBOL,'=',10)
The resulting string associated with $SYMBOL: ALPHAXT17=10

(3) PUT 'SIN'TYPE 4,T1=SYMP,PUT '+' IN TOPER,SET $E1=STRING(
'TEMP','=',SYMTAB(T1),'(X)',TOPER(SYMP),'TEMP')
The resulting string associated with E1: TEMP=SIN(X)+TEMP

(c) SET CASE svar=(element[,element])
n

This element is a short form for and is equivalent to the following:

svar=0,(element1,svar=1//element2,svar=2//...//element$_{n+1}$,svar=n+1)

Thus, the SET CASE element is true if one of its component elements is true, with the simple variable 'svar' being set to the true element's index. Each component element may be of any type. The SET CASE element is typically used to set the 'svar' for a subsequent CASE OF element usage (see element type 7).

Examples: (1) SET CASE I = (IF 'A','=',DIGITS)

Applied to:	The element result:
A	True; no cursor movement; I=1
=	True; cursor moves by one; I=2
123	True; cursor moves by three; I=3
Z	False; no cursor movement; I=0

(2) SET CASE I = (('A'//'B'),SET CASE J = ('C','D'),NULL)

Applied to:	The element result:
A	True; cursor moves by one; I=1
B	True; cursor moves by one; I=1
D	True; cursor moves by one; I=2,J=2
E17	True; no cursor move; I=3,J=0.

(13) Terminal Element

Syntax: [iexp[TO iexp]]terminalname[s]

The character[s] associated with the 'terminalname' are to be found starting at the current cursor position in the input statement. The terminalname must have been previously declared in the Lexical Pre-processor section of the Meta-Language definition (see section 1.2.1.2). The two optional integer expression prefixes define the minimum and maximum number of occurrences of the terminal to be found in order for the element to be true. If both are absent their assumed values are one; i.e., exactly one terminal is scanned. If only one prefix expression is present it specifies both the minimum and maximum values; i.e., 'iexp' successive occurrences of the terminal are scanned. The optional trailing 's' is ignored if either prefix is specified, but if the terminalname occurs alone followed by an 's', then 'zero to infinity' occurrences are implied resulting in repeated scanning for the terminal until no more are found.

The element is true if the stated or implied minimum to maximum occurrences are found, and the cursor is moved beyond the last found, and the cursor is moved beyond the last found terminal. A false situation does not change the cursor. Note that a zero minimum specification causes the element to be always true.

Examples: Assume the terminal definitions:
TERMINALS (DIGIT='0123456789',OPERATOR='+-/'//'*','*'//'*').

Applying the following examples to the input: 12*13-**5
causes the indicated results with the parsed portion of input underlined:

(1)	1 TO 10 DIGITS	-- true, 12*13....
(2)	0 TO 2 OPERATORS	-- true, no cursor movement
(3)	0 TO 2 DIGITS	-- true, 12*13....
(4)	DIGITS,OPERATOR,2 DIGITS, OPERATORS,1 DIGIT	-- true, 12*13-**5
(5)	DIGITS,OPERATORS,DIGITS, 3 OPERATORS	-- false, since '-**' is composed of the two operators '-' and '**'.

(14) Null Element

Syntax: NULL

This element is unconditionally true and causes no cursor movement. It is used primarily to keep a string true which has one or more optional elements.

Example: $X.=('A'//'B'//NULL),'C'

This mlv will parse the construct: [A]C
or: [B]C

(15) False Element

Syntax: FALSE

This element is unconditionally false, causing an alternate string to be applied to the input.

Example: $ISITADO.='DO',$LABEL,$NAME,'=',$INTEGER,',',FALSE //
$DOPROCESSOR.

(16) Chain Element

Syntax: .C [svar]

A new chain is initiated and will consist of all deferred procedure calls requested from this point on until a new chain element is 'executed'. If a simple variable (svar) is specified it will be associated with the new chain and may be referenced by name in a deferred procedure call (see element type 8). This element is true and causes no cursor movement.

Example: $EX.=$X,.C T1,.L(T1,10)PROG($X).

(17) Fortran Element

Syntax: .F[ORTRAN][iexp]literal

This element causes the insertion of the literal string, which should be the body of a Fortran statement, into the parser being generated. It is primarily used to insert semantics which are not directly supported by other Meta-Language statements. The literal is inserted into the generated compiler starting in column seven, optionally preceded by a label indicated by 'iexp'. The literal is not checked for Fortran compatible syntax and may be of any length (line continuation logic is automatically inserted). This element is unconditionally true and causes no cursor movement.

Examples: (1) .F10'GO TO 9001' generates 10----GO TO 9001
(2) .F CALL DUMPER generates CALL DUMPER

(18) Test Element

Syntax: TEST literal

This element is a short form for the following:
(literal//FATAL('SYNTAX ERROR AT',CURSOR))

If the literal operand is not found a fatal syntax error is displayed (see element type (22)), while if found the element is true and the cursor is moved beyond the literal.

Example: TEST '*'.

(19) Field Element

Syntax: NEXT FIELD [iexp]

The input cursor is positioned to the beginning of the next field. All fields have been previously declared in FIELD statements in the Lexical Pre-Processing section (see 1.2.1.3). If 'iexp' is present it specifies the field number to move to; otherwise, the next sequential field is chosen.

If positioning is made to the next sequential field, the cursor should be pointing to the end of the current field. If this is not the case the element is false, otherwise it is true unless an attempt is made to position to an undefined field.

Example: FIELDS 1(1-5),2(7-72).

```
$LABGET.=($LABEL//NULL),NEXT FIELD.
$LABEL.=1 TO 5 DIGITS.
```

This example shows the parse of the label field of a FORTRAN statement, leaving the cursor at column seven when done.

(20) Attribute Element

Syntax:

```
{PACK  }   {hashname [(iexp)]          }  (a)
{UNPACK}   {hashname svar[(iexp)]      }  (b)
           {svar                       }  (c)
           {TYPE iexp1[(iexp2)]        }  (d)
           {TYPE iexp1 svar[(iexp2)]   }  (e)
```

This element allows attributes associated with hash tables or the Symbol Table to be packed from, or unpacked into, the attribute cells associated with the table. Each hash table and the Symbol Table may have a parallel attribute table containing the same number of entries as the declared table size, each entry occupying a certain number of host words. Each entry contains a set of symbol attributes packed together; corresponding to each attribute is a global cell occupying a host word from which the attribute can be packed from or unpacked into.

The first two forms above allow hash table attributes to be manipulated. The first form [un]packs the entire entry located at entry number 'iexp', if present, or located at the entry contained in the declared running pointer associated with the table (see 1.2.2.2, Symbol/Hash Declarations). The second form allows a particular attribute, designated by the name 'svar', to be [un]packed.

Examples:
```
TABLE 'EQUATE NAMES', TEQU(100,TPTR,TLEN,
FLAG1(0-0-12),FLAG2(0-13-5),FLAG3(S,0-18-14)).
:
$ENAMP,=$SYMBOL, PUT $SYMBOL IN TEQU, FLAG1=0,
 FLAG2=1,FLAG3=7,PACK TEQU.
$UNAME.=$SYMBOL,IF $SYMBOL IN TEQU, UNPACK TEQU,
 FLAG3= -44, PACK TEQU FLAG3, S1=TPTR.
$X.=UNPACK TEQU(S1),FLAG1=0,PACK TEQU FLAG1(S1).
```

The third through fifth forms of the attribute element manipulate Symbol Table attributes only, with predefined cell names and the running symbol pointer SYMP. The third form allows [un]packing the attribute named 'svar'. The Symbol Table segment and entry sequence number are assumed to be contained in SYMP. The fourth form [un]packs an entire Symbol Table entry of type 'iexp1' for entry number 'iexp2' (if present), or SYMP. Form five has the same meaning except the single attribute named 'svar' is [un]packed.

Examples:
```
SYMBOL USAGES SIV,ARY,PRO.
SYMBOL ATTRIBUTES SIV,500,ARY,200,PRO,25.
:
$NAME.=$SYMBOL,PUT $SYMBOL TYPE SIV,IM=2,
  R=1,S=1,PACK TYPE SIV.
$UNAME.=$SYMBOL,IF $SYMBOL TYPE SIV,UNPACK IM,
  IM=IM+1,PACK IM,TEMP=SYMP.
$EXAMPLE.=UNPACK TYPE SIV(TEMP),R=0,
  PACK TYPE SIV R (TEMP).
```

(21) Error Element

Syntax: $\left\{\begin{matrix}\text{WARNING} \\ \text{ERROR} \\ \text{FATAL[ERROR]}\end{matrix}\right\}$ $\left\{\begin{matrix}\text{[(arg[,arg])]} \\ n \\ \end{matrix}\right\}$ (a) (b) (c)

Three distinct types of error handling capabilities are provided. The arguments represent information strings to be included in an error message line displayed on the print file. If an integer-valued first argument is provided it is associated with the corresponding error message declared in the ERROR MESSAGE directive (see 1.2.2.3).

The warning error (type (a)) element causes the specified diagnostic to be printed but retains a true status to allow parsing to continue. The message is printed at the time a step-up is made from the definition rule containing the warning element.

The normal error (type (b)) element is used whenever an error is found of such magnitude that parsing cannot continue at the current level. The truth value of this element is always false. The error message is stacked for later printout at the time the parsing of the current statement is terminated (i.e., the step-up from the head node definition is made). However, if any step-up true is made from a definition after encountering the error, its associated message is not printed and no further action is taken. Only one error message of type (b) may be active for a given statement being parsed; any subsequent ERROR requests replace the previous request.

The fatal error element (type (c)) causes an immediate exit false from the parsing of the current statement with a printout of the indicated message. Any deferred procedure calls are negated as well as any previous error conditions of type (b).

Examples: (1) `$EX1.=$NAME,IF $NAME TYPE 1//WARNING('INVALID NAME AT',CURSOR).`

(2)
```
ERROR MESSAGES 'INTEGER SUBSCRIPT REQUIRED' /*ERROR1*/
               'MISSING RIGHT PAREN AT'.    /*ERROR2*/
   .
   .
   .
$EX2.=NOT IF DIGIT,ERROR(1)//DIGITS,(')'//ERROR(2,CURSOR).
```

This example applied to:	Yields:
123)	True-no errors.
)	True-no errors.
X	INTEGER SUBSCRIPT REQUIRED
12*	MISSING RIGHT PAREN AT 3

(3) `$EX3.=$NAME//$INTEGER//FATAL('INVALID OPERAND').`

(22) Table Initialization Element

```
{READ }
{WRITE}  TABLES
```

The WRITE TABLES element allows the contents of all hash tables and the symbol table to be copied onto the library data file. This file will be subsequently read back into the tables as part of the compiler initialization during actual compilation runs. This element is typically included in the definition of a scaled-down 'compiler' whose only function is to initialize the symbol handling tables, as in the following:

```
$INITIALIZER.=PUT 'SIN','COS','ABS',...,'SQRT','EXIT'
TYPE FUNC,WRITE TABLES.
```

The generated table data is associated with the compiler name on the DEFINE card.

The READ TABLES element causes previously generated symbol and hash table entries to be read into the table area. It is typically placed within the initialization phase at the beginning of the production compiler as in the following:

```
$INITIALIZER.=READ TABLES,...
```

This element is true if the tables identified by the compiler name on the DEFINE card are found on the library data file.

(23) Debug Element

```
DEBUG {ON }  STEP {UP  } [TRUE]
      {OFF}       {DOWN}
             DUMP [FALSE]          [ , {ON } ...]
             REOCCUR                 n {OFF}
             TERMINAL
             FL
             PROC
```

This element allows various debugging aids as described in the DEBUG declarative (1.2.2.2) to be [de-] activated. The next selected debug options settings remain activated until the next 'executed' occurrence of the DEBUG element. This element is always true.

(24) Free Table Element

FREE $\left\{\begin{matrix}\text{hashname}\\ \text{iexp}\end{matrix}\right\}$ $[,_{n}$ $\left\{\begin{matrix}\text{hashname}\\ \text{iexp}\end{matrix}\right\}$]

The symbol storage area occupied by the indicated hash tables or Symbol Table segments (iexp) are made available for use by any other tables.

(25) Save/Restore Element

$\left\{\begin{matrix}\text{SAVE} & \text{iexp} & [\text{iexp}]_{n} \\ \text{RESTORE} & \text{svar} & [\text{svar}]_{n}\end{matrix}\right\}$

The SAVE element causes the expression operands to be placed on the top of the particular stack PSTACK. This element is equivalent to: PUSH iexp$[\text{iexp}]_{n}$ ON PSTACK.

The RESTORE element replaces the svar operands with the operands from the top of stack PSTACK. This element is equivalent to:

svar_1 = PSTACK(PPOINT)
OPO PSTACK
svar_2 = PSTACK(PPOINT)
POP PSTACK
⋮

1.3 TARGET DEFINITION

Statements in this section allow the compiler writer to define the characteristics of a variety of target computers for which specific code may be generated. The target definition section is optional and has no dependence with other sections of metalanguage dealing with language specification.

A target definition causes an entry to be made on the Target Library in such a manner as to allow the selection of a target at compile time, by the user, through compile time options (1.4).

1.3.1 Target Section Identification

Statements in this group identify and bound a target definition by name.

Target Initiation

START TARGET 'identification'.

identification -- the identifying string of letmerics for the machine description. The first eight characters must be unique.

This is the first statement of each target definition. The identifying string may be of any length but only the first 8 characters are used and must therefore be unique.

Target Termination

END TARGET.

This is the last statement of a target definition. The definition is placed on the Target Library with its identifying name. Any previous definition with the same name is replaced.

1.3.2 Machine Environment Definition

Statements in this section define certain global system parameters relating to characteristics of the target machine that affect code generation.

Parameter Setting

PSET name = value[,name=value].

name -- a letmeric symbol representing a system parameter name.

integer -- an integer, octal, hexadecimal*constant or equate name defining the value of the associated parameter name.

This statement serves to allow definition of parameters controlling certain aspects of code generation. The following table defines all the parameter names and their meaning:

---

* Hexadecimal and octal values are designed by X'digits' and O'digits," respectively.

PARAMETER NAME	MEANING	DEFAULT VALUE
WORD*	Bit size of single-precision target accessing word	none
BYTE	Bit size of a target byte	8
ADDR*	Bit size of the basic location counter addressing unit	none
BOUND	0 - Bound instructions, 1 - No bounding of instructions	0
RESULT	The register number of the result register upon return from subprograms	0
CALLR	The target call register	0
WORKR	A general work register	0

* The values of these global cells must be defined via the PSET command. All other parameters are optional.

Example: [For an IBM 360]

```
PSET    WORD=32, BYTE=8, ADDR=8, BOUND=1, RESULT=0, CALLR=15, WORKR=0.
```

Register Definition

REGISTERS Type = Size(Low-[High][,Low[-High]])[,Type...]
n

Size -- bit size of each register in a set.

Low -- lowest numbered register in a set.

High -- highest numbered register in a set.

Type -- register type for a given set:

A - Arithmetic accumulator (for logical, fixed point, or integer arithmetic)

G - General purpose

X - Index register

B - Base register

F - Floating point accumulator

E - Even/odd accumulators (for multi-precision operations)

C,D,H,I,... - Other single character labels designating selected subsets of registers referenced via IFORM directives.

This statement allows all target registers available to the code generators for assignment to be defined as to type, size, and specific number. The types A,G,X,B,F, and E have the indicated predefined meaning to the code generation process.

The compiler writer may specify his own register subsets by using any type designation character other than A,G,X,B,F,E.

Register types are referenced within instruction format specifiers (see IFORM directive) and as parameters to PROC support functions (see 1.3.4.4). All register maintenance facilities for saving or restoring registers appearing in this statement are automatically provided within C.W.S.

Example: [For an IBM 360]

```
REGISTERS   A=32(0-15),G=32(0-15),X=32(1-7),B=32(1-15),F=32(0,2,4,6,8,10,12,14),
            E=32(0,2,4,6,8,10,12,14).
```

Arithmetic Data - Fixed Point

FIXED POINT = {SMAG / 1COMP / 2COMP} [,SBIT={MAG / SIGN / ZERO}].

The characteristics of integer and fixed point data computations on the target machine are specified. Integer arithmetic may be specified as signed magnitude (SM), one's complement (1C), or two's complement (2C); in all cases the sign bit of an integer data item must be bit 0. The SBIT operand defines the utilization of the sign bit of low order words of multi-word values as follows:

MAG -- include as part of the fixed point value;

SIGN -- set to the sign of the fixed point value;

ZERO -- set to zero, unconditionally.

Arithmetic Data - Floating Point

FLOATING POINT= (EXP=(start-end)[, {BIN / OCT / HEX}] [, {BIAS / SMAG / 1COMP / 2COMP}],

(MANTISSA=(start-end) [,signpos] [, {SMAG / 1COMP / 2COMP}]).

start -- bit starting position in word;

end -- bit ending position in word;

signpos-- bit mantissa sign position

This specification defines the exponent position, type (binary, hex, or octal), and representation:

BIAS -- exponent is biased by the value 2 start-end;

SMAG -- signed magnitude
1COMP -- one's complement
2COMP -- two's complement
} with sign at start bit.

The mantissa is also defined as to position, sign position (signpos), and representation. For 1COMP or 2COMP mantissas, all bits from bit signpos to the right are complemented for negative numbers.

Examples:

```
FIXED POINT = 1COMP, SBIT=SIGN.
  [one's complement integers with signs copied to multi-words]
FLOATING POINT=(EXP=(1-7),HEX,BIAS),
  (MANTISSA=(8-32),0,SMAG).
     [360 floating point format]
```

Target Character Set

CHARACTERS CODE//GRAPHICS//[,CODE//GRAPHICS//]$_{n}$.

code -- integer, octal, or hexadecimal value giving the internal target representation for the first character in the following graphics.

graphics-- a collection of characters having sequential internal code values on the target machine.

This statement specifies the conversion from the host characters utilized within the compiler input language to other internal character representation within the target machine. The code value is incremented by one for each graphic character.

Example: [IBM 360]

```
CHARACTERS X'A0'// //,X'A7'//'$( )*+,-./0123456789//,
    X'BD'//=//,X'C1'//ABCDEFGHIJKLMNOPQRSTUVWXYZ//.
```

1.3.3 Instruction Definition

The two statements in this section allow for definition of all required machine instructions and their formats. Only those instructions interacting with the code generation process need be defined; typically only a subset of a target instruction repetroire is utilized through the PROC expansions.

Operating Mnemonic Definition

OPERATION (opn,code,formp$_{n}$[,formp$_{n}$])[,(opn,code,formp$_{n}$[,formp$_{n}$])].

opn -- an operation mnemonic name.

code -- the hex, octal, or integer operation code for opn.

formp -- the instruction format (IFORM) name defining the structure of the operation.

This statement allows definition of target machine operations by name and numeric code. The opn is printed in the operation field of the generated assembly listing.

Each operation is associated with one or more instruction formats by name through the formp specifiers. When operation references are made by invoking the CODE generation function (see 1.3.4.3), the operation code and the CODE supplied parameters are substituted into the instruction formats indicated by the formp names, from left-to-right, until a successful instruction is constructed.

Instruction Formats

IFORM formp,length,O=(start-end)[,type=(struct-end)][,M=(subtype=(start-end)[,subtype=(start-end])]$_n$.

formp -- instruction format name referenced by OPERATION commands.

length -- the bit length of the generated instruction on the target machine.

O -- indicates the operation code field, the value of which (defined via an OPERATION) is to be placed in the indicated position in the instruction.

start -- starting bit position for a field.

end -- ending bit position for a field

type -- field descriptor of the types indicated below.

M -- indicates a special field defining an addressing mode, or memory reference, indicated by the enclosed subtypes.

subtype -- an addressing mode field descriptor of the types indicated below.

The statement allows the definition of an instruction format specifying the bit positions of the generated object text for an OPERATION reference having the instruction pointer name formp.

Each field of the IFORM directive generally corresponds to an actual parameter value supplied to the corresponding OPERATION reference through a CODE generation call. Missing or null actual parameters are considered to have a zero field value.

The field descriptor types include:

RS - REGISTER DESIGNATOR

S -- A,G,X,B,F,E or other character designating a register class (see the REGISTER statement).

The corresponding actual parameter is considered a positive register value of class 'S'. A diagnostic occurs if the register value is invalid for that class.

C - CONTROL FIELD

The corresponding actual parameter is considered an ordinary signed control value to be placed in the indicated field, such as a shift count, mask, etc.

Constant - An integer, hex, or octal constant

There is no corresponding actual parameter in this case. The constant becomes the field value to place in the designated position. Zero fields need not be specified since the instruction skeleton is initialized to zero prior to IFORM processing.

M - Memory Reference Field

This field defines via subtype specifications the type of memory reference permitted for this instruction format as well as the bit positions within the generated instruction for each memory reference subtype.

The actual parameter[s] corresponding to as M memory reference field are as follows:

(1) A data item, consisting of:

(a) A symbol pointer; and/or,

(b) A constant offset value for arrays and tables.

(2) An optional index register.

(3) An optional base register.

(4) An optional indirect flag.

The actual parameter items are supplied directly through the CODE request (see code generation function 1.3.4.3) to the corresponding OPERATION, and thus to each optional IFORM skeleton referenced therein. The various memory reference actual parameter combinations supplied may be summarized symbolically as: S[,X][,B][,*]. They are corollated with and substituted for the M subtypes, which are:

L -- Location counter value for a direct memory reference.

SD -- A signed displacement value.

D -- Unsigned displacement value.

X -- An index register.

B -- A base register.

* -- Indirect flag

The above subtypes may be specified in the following combinations with the indicated meaning, and may be correlated with the indicated actual parameter combinations. The order of the subtypes within a given combination must correspond to that of the supplied actual parameters.

(1) L[,X][,*] Direct memory reference with possible indexing and indirect addressing.

The computed effective address is assumed to be L plus the contents of register X (if present). If an indirect flag is present, the indirectness is assumed to be applied after computing L+(X), i.e., (L+(X)) is the effective memory address.

Allowed Actual Parameter Types: S[,X][,*]

(2) [S]D[,X][,*] Location counter displaced with possible indexing and indirect addressing.

The value [S]D is computed as a [signed] displacement from the current value of the location counter by subtraction from the data item location. Any indirectness is assumed to be performed last, i.e., ([S]D+current location counter + (X)) is the effective memory address.

Allowed Actual Parameter Type: S[,X][,*]

(3) [S]D,B[,X][,*] Base register displaced with optional indexing and indirect.

The value [S]D is assumed to be a [signed] displacement from an address contained in base register B, indexed by (X). Any indexing is assumed to be performed last, i.e., ([S]D+(B)+(X)) is the effective memory address.

Allowed Actual Parameter Type: S[,X][,*]

1.3.4 PROC Expansions

The PROC mechanism allows for code expansion by the compiler writer for all required and optional Operation Language terms. PROC definitions appear together as the last section of a target definition following the identification, environment, and operation definition sections. A PROC definition has one of the forms:

PROC $name. = operationname.

PROC $name[(argname[,argname]$_{n}$)].=string[//string]$_{n}$. (II)

name -- a letmeric symbol of up to 20 characters defining the unique PROC name.

argname -- letmeric symbol of up to 6 characters representing an argument to the PROC.

string-- a collection of elements, operands, and operators as defined in the following sections.

operationname -- the name of a single target operation defined by an OPERATION command which is equivalent in function to the Operation Language term represented by the PROC name.

The case (I) definition allows PROCs which have a one-to-one relationship with target operations to be written in short form. All PROC parameters are carried directly to the operation name and its instruction format expansions.

The syntax and semantics of a target definition PROC are similar to those of an MLV definition in the Metalanguage. Thus, PROCs are invoked in a top-to-bottom fashion as a collection of closed code generative functions. Each PROC has the attribute of being true or false upon return from its invokation. A PROC is true if and only if one of its designated alternative strings (separated by the '//' operator) is true.

1.3.4.1 PROC Operands

TYPE	SYNTAX	EXAMPLES
o Constants	[Signed] Decimal integer	10
	Hexadecimal String	X'7F'
	Octal String	O'1234567'
	Literal String	'ABC'
o Variable	One to six letmeric characters	NAM
	[followed by the modifier .sym]	N.SYM

PROC variables are analogous to SET symbols in ordinary assembler macros, i.e., they are associated with a value (always fitting within a host word) which may be interrogated or modified at PROC invokation time. Variables typically contain switches, symbol pointers, register numbers, or control values.

A referenced variable known to contain a symbol pointer should be flagged by a trailing .SYM modifier. Its subsequent use as an operand will cause its value to represent the assigned address field of the pointed-to symbol.

o Location Counter	$	$+1
		$+A-2

The current value of the target machine location counter is implied.

o PROC Argument	PROC argument name	PROC SP(REG,V).=..
		T= REG +1

This operand is a symbol which is a formal parameter representation of an actual parameter to the PROC to be supplied at PROC expansion time. Each parameter name has an associated attribute type which may be interrogated at expansion time (see below). An argument name may be used wherever an ordinary variable is legal.

The use of a PROC argument name as an operand need never be flagged with the modifier SYM, even if it is known to contain a symbol pointer. The attribute type for the actual parameter at expansion time determines the meaning of the parameter value.

o Expression	Operands combined with the operators +,-,/,**.	N+X'7F'/3*J-1 0'23'+17-X**2 3-2/1+4**2 -1+$-2 CELL.SYM+2

Expressions represent values which are to be computed at PROC invocation time as a function of their constituent constant or variable values. Expressions are evaluated according to a string left-to-right rule. Thus, the value of example expression three is 25.

If any operands in an expression represent the value of any location counter (i.e., location counter($) operands, PROC parameter symbols of type '.SYM', or variables with the modifier '.SYM'), they should be the only such operand in the entire expression with the following exception - any number of pairs of such operands may occur, each pair separated with a 'minus' operator, and each pair having the same relocatability.

Examples:

$+1	Valid	
$-KLOC+2/7	Valid	
NAME.SYM-SAM*2-3	Valid	[Only one pointer symbol]
2+N.SYM-M.SYM/3	Valid	
$+N.SYM-2	Invalid	[Multiple loc counter symbols]

o Derived type TYPE OF PROC argname $LDA(REG,A).=
...TYPE OF A...

The type of the indicated PROC argument name is the value of this operand. The types are:

1 - integer valued simple item
2 - literal string
3 - location counter
4 - operand symbol pointer
5 - current code table operator pointer
6 - null, or missing

1.3.4.2 PROC Elements/Operators

A PROC element consists of a PROC operator acting upon a PROC operand as defined in the previous section. The result is a specific action having a truth attribute. The elements are separated by conjunctions (,), allowing PROC expansion to proceed from one element to the next if and only if that element is true. Thus, an entire string is true only if all its elements are true.

The PROC elements are:

ELEMENT TYPE	SYNTAX	EXAMPLE
o Replacement	variable=expression	A = B+C.SYM-$+1

The variable name operand on the left is replaced by the expression operand on the right. The result is true if the expression can be validly computed at PROC expansion time.

o Condition Testing	IF expression1 relop expression2	IF B+2 GE $-2

The two expression operands are compared as signed quantities as indicated by the relational operator (EQ, NE, GE, LE, GT, or LT). The result is true if the condition is satisfied. Note that the result is automatically false if expression1 - expression2 cannot be validly computed.

o Truth Negation	NOT element	NOT IF A GT B NOT $LDA (REG, V)

The PROC element following the NOT operation is evaluated and the resulting truth value is reversed.

o Substring Element	(substring)	...,(IF I GT 7, $LDA ($,A)// $LDA (R,B)), ...

This element actually consists of an enclosed string, or substring of the form: (element[,element][//element]). Elements within the substring may be of any type defined in this section, including substring elements. A substring element is true only if its enclosed substring is evaluated to be true.

Examples:

```
...((IF A GT 2//IF L EQ J), T=5//T=6)...
$MOV(M1,M2).=(IF TYPE OF M1 EQ1, $LI(R,M1)//
    $LOAD ($,M1)),$STORE(R,M2).
```

o Truth Element	NULL	...,(IF I EQ 0,I=1//NULL),...

This element performs no action. Its truth attribute is simply set true.

o PROC Invokation	$procname[(arg[,arg])]	...,$ABC('T',U+1),...

A PROC may be called from any other PROC including itself. The procname is the symbolic name of a PROC and the arguments are optional actual parameters to be substituted for the corresponding formal parameters in the PROC definition.

Since there is a one-to-one correspondence between actual and formal parameters, a missing actual parameter should be indicated by successive commas if any additional parameters are to be supplied.

A PROC argument may be any valid operand as defined in the previous section. If an argument in a PROC call is a variable which is known to be a symbol pointer, it should be flagged as such with the modifier '.SYM'. This convention allows a TYPE interrogation in the referenced PROC to distinguish between a formal parameter which is a value and one which is a symbol pointer.

A PROC invokation element is considered true if the expansion of that PROC is true.

o Code Generation Function	CODE (op[,arg])	See 1.3.4.3
o Support Function Call	*name[(arg[,arg])]	See 1.3.4.4

These elements are defined in the following two sections.

1.3.4.3 Code Generation Function

The actual generation of target-specific machine operations is accomplished through utilization of the code generation function from a PROC:

*CODE(Opname[,arg])

opname -- a machine operation mnemonic name, defined via an OPERATION command.

arg -- the corresponding actual values to be substituted into the referenced operation's IFORM object text fields.

This function provides the link between PROCs and operations on the target machine as specified by OPERATION and IFORM commands.

The actual arguments must agree in type, order, and number (except for implicit address conventions noted below for M subfields) with those specified by the IFORM fields. A corresponding assembly listing line is also generated (if the option is specified).

The CODE function reference returns a truth value in the same manner as a PROC invokation. If all field substitutions are found to be valid, the machine operation is generated and processing continues along a true path. If a field error is detected, the code generation is inhibited, and a false return is made to the calling PROC.

The following actual-formal parameter matchings are allowable between the arguments supplied to CODE and the corresponding IFORM fields:

IFORM FIELD TYPE	ALLOWABLE ACTUAL CODE ARGUMENTS
RS	Any integer-valued operand. The value is checked for the allowed range as specified by register class 's'.
C	Any integer-valued operand.
Constant	[No corresponding actual parameter]
M	Explicit or implicit address vector.

Substitution into address (M) subfields within IFORMs proceed according to the following rules depending on the defined address type:

(1) M subfield of the form L[,X][,*] -- direct memory reference

The actual parameters for each field ('*',L, and X) are optional but when present may be any valid PROC argument. Any missing parameters are assumed to have the value zero. The actual parameter for the L field is typically an address expression or a symbol pointer.

(2) M subfield of the form [S]D[,X][,*] -- location counter displaced

The same conditions as for type (1) apply, except whenever the [S]D field is a symbol pointer or address expression the computed field value is the difference between its assigned address and the current location counter. if the computed offset is beyond the allowed range the system will generate an indirect reference (which thus must be allowed in the IFORM) through a future literal which is later dropped within range.

(3) M subfield of the form [S]D,B[,X][,*] -- base register displaced

The * and X fields are optional and obey the same rules as for case (1). If the B input argument is present, explicit basing is performed and the value of the argument is the base register to be used. Then, if the [S]D argument is a simple value (or is missing), its value (or zero) is used for the [S]D field; if the [S]D argument is a symbol pointer, the difference between its assigned address and the current contents of the location counter is the field value to substitute. If the B argument is missing the system

will supply one based on current base register values (defined via previous USING support calls, 1.3.4.4). In this case the [S]D field is required. If no base registers currently contain a value within range of the sought after address, one is fetched from the register pool and is loaded with an appropriate base address.

1.3.4.4 Support Function Calls

A collection of support functions may be invoked within a PROC to support the code generation and register maintenance processes. A support function call has the form:

```
*Name[(Argument[,Argument])]
```

A truth value is returned to determine whether to continue expanding along this path. Appendix B defines the calling sequence for all code generative support functions.

1.4 Compile-Time Option Selection

Compiler options are supplied by the user to a C.W.S. generated compiler in such a way as to be independent of the J.C.L. of the host operating system. This is accomplished by supplying the following as the first card image of a compilation source deck:

OPTIONS[,T=targetname][,P= {1 / 2}][,NS] [,A][,NO] [,L= {0 / 1 / 2}] [,D=digit[,digit]].
5

This card is immediately followed by the first card of the source language program[s] being compiled. The meaning and default value for each compiler option is as follows:

OPTION	MEANING	DEFAULT VALUE
° T=targetname	Target Selection	First Target Library Entry

The indicated targetname, which matches a target machine definition previously created by the compiler writer, is selected for object output code generation. The absence of this option causes the first definition on the Target Library to be selected.

° P = {1 / 2} One (two) Pass Selection Default=2

A choice between a one or two pass compiler execution is provided. One pass selection permits faster compiler execution, but expands the generated object text and results in incomplete address field values in the generated assembly listing.

° NS Source Listing Suppression Default=listing provided

A listing of each source line image is printed on the standard list output device[1] unless this option is utilized to suppress it. Any error conditions will still cause error messages to be printed.

° A Generate Assembly Listing Default=none provided

The presence of this option causes an assembly listing of the generated target machine code to be generated on the standard list output device[1].

- NO — Object Code Suppression — Default=object generated

This option suppresses the generation of object text output, permitting faster compiler execution.

- L=$\left\{\begin{matrix}0\\1\\2\end{matrix}\right\}$ — Optimization Level — Default=0

Three levels of global optimization are available for selection as follows:

Level 0 -- Full global optimization is performed;

Level 1 -- Restricted optimization is performed;

Level 2 -- No global optimization is performed (fastest compiler execution).

- D=digit $\underset{5}{[,digit]}$ — Defaults=DEBUG declaration

The various debug options (see DEBUG statement, 1.2.2) may be turned on or off to monitor compiler execution. Each digit corresponds to a particular option as follows:

digit 1: -1=STEP DOWN, 1=STEP UP, 2=STEP UP TRUE

digit 2: 1=DUMP, -1=DUMP FALSE

digit 3: 1=REOCCUR

digit 4: 1=TERMINAL

digit 5: 1=FL

digit 6: 1=PROC

A zero setting for any option turns that option off.

2. FUNCTION LANGUAGE

This section describes the Function Language, which is produced as output by the Source Processor and is written on a data file in the format specified in section 5.2 of Volume II.

The general form of a FL term is as follows:

function-verb [operand][$_{n}$,operand]

function-verb -- The symbolic name associated with the F.L. operator.

operand -- an operand derived from the source program having one of the following forms:

- Symbol Pointer

  A symbol table or hash table pointer to a symbolic operand. A symbol pointer has the following form:

  Table-number, sequence-number

  Table-number -- The Symbol Table segment (if less than 10) or hash table (>10) to which the symbol belongs.

  sequence-number -- the symbol sequence number (1,2,...) identifying the symbol position.

- Integer Values

  Single-valued quantities used as flags, parameter values, etc...

The above defined descriptive notation is used throughout this section to describe each FL term. The terms are grouped by semantic function category.

2.1 Data Declaratives

° Data Initialization

Form: DATA $V_n[,I],O,M,C[,M,C]_n$

V -- variable, array, table, table item pointer

I -- integer constant pointers

M -- integer values

C -- general constant pointers

° Assigned Data

Form: ADATA label,arg

label -- program label to associate with the following label.

arg -- compressed string pointer (in QTABLE) or constant pointer.

2.2 Expression Manipulation

° Expression Generation

Form: GEN E

E -- expression stream in reversed polish notation. The form of the operands and operators within E are exactly as described for the Polish Expression Stack (PSTACK - see Volume II, section 2.3).

2.3 Compilation Support

° Statement Identification

Form: STAT

Use: Identifies the start of a new language statement.

° Label Definition

Form: LDEF label[$_{n}$,label]

label -- program label pointer

Use: Defines the start of the program block associated with the argument labels.

° Block Initiation

Form: BEGIN label

label -- program label pointer

Use: All code following up to the next BEGIN term is to be placed in the block associated with the label.

° Modify a Name

Form: MODIFY var[$_{n}$,var]

var -- variable name pointer

Use: The designated variables are changed through execution of any block composing the current statement.

° End of Compilation

Form: END

Use: This is the last FL term generated. It signals the end of a compilation.

2.4 Program Definition

° Main Program Initiation

Form: MAININ NAME,REC,REN

NAME -- main program name pointer

REC -- recursive flag (0=no, 1=yes)

REN -- reentrant flag (0=no, 1=yes)

Use: Defines the start of a global main program.

° Global Subprogram Initiation

Form: ESUBIN NAME,REC,REN

NAME -- subprogram name pointer

REC -- recursive flag

REN -- reentrant flag

Use: Defines the start of a global function or subroutine.

° Internal Subprogram Initiation

Form: ISUBIN NAME,REC,REN

NAME -- subprogram name pointer

REC -- recursive flag

REN -- reentrant flag

Use: Defines the start of an internal subprogram (subroutine or function).

° Global Entry Point Initiation

Form: EENTIN NAME,REC,REN

NAME -- Entry point name pointer for a globally defined subprogram entry point.

REC -- recursive flag

REN -- reentrant flag

Use: Defines the start of an entry point to the active global subprogram.

° Internal Entry Point Initiation

Form: IENTIN NAME,REC,REN

NAME -- Entry point name pointer for an internally defined subprogram entry point.

REC -- recursive flag

REN -- reentrant flag

Use: Defines the start of an entry point to the currently active internal subprogram.

° External Argument Transfer

Form: EXDUMMY i,Pi,Ri

i -- the argument number

Pi -- argument name pointer

Ri -- argument type (1=output,2=input,3=both,4=neither)

Use: Defines the i'th argument to the currently active global subprogram.

° Closed Subroutine Initiation

Form: CLOSP NAME

NAME -- closed subroutine name pointer

Use: Defines the start of a closed subroutine, a special form of an internal subprogram having no arguments.

° Internal Argument Transfer

Form: INDUMMY i,pi,ri

i -- the argument number

pi -- argument name pointer

ri -- argument type

Use: Defines the i'th argument to the currently active internal subprogram.

° Return From Subprogram

Form: RETURN

Use: Signals a return point from the currently active subprogram.

° External Argument Return

Form: EXRETDUM i,p

i -- the argument number

p -- argument name pointer

Use: Allows returning the value of an output argument to the active global subprogram.

° Internal Argument Return

Form: INRETDUM i,p

i -- the argument number

p -- argument name pointer

Use: Allows returning the value of an output argument to the active internal subprogram.

° Main Routine Exit

Form: MAINEX n,REC,REN

n -- subprogram name pointer

REC -- recursive flag

REN -- reentrant flag

Use: Terminates the active main program

° External Subprogram Exit

Form: ESUBEX n,REC,REN

n -- subprogram name pointer

REC-- recursive flag

REN-- reentrant flag

Use: Terminates the active global subprogram.

° Internal Subprogram Exit

Form: ISUBEX n,REC,REN

n -- subprogram name pointer

REC-- recursive flag

REN-- reentrant flag

Use: Terminates the active internal subprogram

° Closed Subroutine Exit

Form: CLOSEX

Use: Terminates the active closed subprogram

2.5 Program Transfer of Control

° Switch Definition

Form: IDXSW NAME[,-1,S,I]$_n$[,L]$_n$[,C]$_n$,T

NAME -- switch list name

L -- program location variable or label pointers

C -- CLOSE name

S -- the name of another switch list

I -- an index to the other switch list

T -- test flag (0=No checking)

Use: A list of transfer locations are defined to allow switch transfers at run-time. The switch list will be generated as a parameter list in the target code.

Form: ITMSW NAME,SIP[,K,L]$_n$[,K,C]$_n$[K,-1,SNAME,IND]$_n$,T

NAME -- Switch name

SIP -- Switch item pointer

K -- comparison constant pointer

L -- Label or location variable pointer

C -- CLOSE name

SNAME - Indexed or item switch name pointer

IND -- Index to SNAME

T -- Test flag (0=No checking)

Use: Defines an item switch containing labels, location variables, CLOSES, or other item or indexed switch references.

° Switch Transfer

Form: INSXFR NAME[,E]

NAME -- indexed switch name pointer

E -- index position value

Use: A indexed switch transfer through the switch list NAME using index E is implied.

Form: ITSXFR NAME[,E]

NAME -- item switch name pointer

E -- item switch name reference expression

° Direct Transfer

Form: GO L

L -- label or location variable pointer

Use: Implies a control transfer to label L, or through variable L.

° Indexed Transfer

Form: IXFER E_n[,L],TEST

where E -- an integer expression

L -- label pointers

TEST-- Test flag (0=No test code)

Use: Implies transfer to the E'th label.

° Computed Transfer

Form: CXFER I,[$_n$,L],TEST

where I -- a location variable

L -- location variable value candidates

TEST-- Branch test flag (0=No test)

Use: Implies transfer to the contents of I. The L's are the allowable values I may contain at run-time.

2.6 Loop Control

° Loop Initiation

Form: Loop T,LV,ℓb,ℓe

T -- loop type:

0=no initial test

1=test before initiating loop

LV -- loop variable pointer

ℓb -- loop body block number

ℓe -- loop successor block number

Use: Defines the start of a program loop with the loop induction variable LV.

° Parallel Loop Definition

Form: PLOOP LV

LV -- parallel loop variable pointer

Use: Implies the existence of a parallel loop to the currently active loop with a secondary induction variable LV.

° Loop Increment

Form: LINCR LV

LV -- an active loop variable pointer

Use: Implies the presence of incrementation code for the [parallel] loop associated with variable LV.

° Loop Test

Form: LTEST LV

LV -- an active loop variable pointer

Use: Implies the presence of incrementation code for the [parallel] loop associated with variable LV.

2.7 Special Directives

Terms in this group allow special functions such as:

- Hardware register manipulation;
- Compiler writer defined mappings;
- Interfacing to run-time support routines.

- Hardware Register Assignment

  Form: ASSIGN X,I

  I -- integer value

  X -- a variable pointer

  Use: The variable X is associated with the hardware register I.

- Freeing Hardware Registers

  Form: FREE $I_n[,I]$

  I -- integer value

  Use: The hardware registers, I, are freed for general usage from previous definition.

- Locking Hardware Registers

  Form: LOCK $I_n[,I]$

  I -- integer value

  Use: Hardware registers, I, are prohibited from general register usage.

° Memory Protect

Form: PROTECT L1, L2

L1, L2 -- location variables

Use: Implies the target memory area from location L1 to location L2 is to be protected from modification.

° Compiler Writer Constructs

Form: USERn [(P[,P]$_n$)]

P -- user parameters of any type

Use: The compiler writer may specifically map into a collection of special verbs (USER's) referencing parameters of any general nature. He must define procedure expansions for them in the Meta-language.

3. TARGET LANGUAGE

This section describes the format of the Target Language, which is the object text representation of a source program. An object program module is created including the following information:

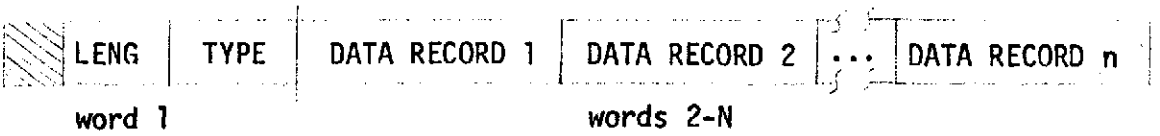
- object module description (DSC)
- global symbol dictionary (GSD)
- object program text (TXT)
- object module end (END)

This information is sufficient for the linkage editor program to perform the following processes:

- relocate object program text and any relocatable address values that it contains;
- complete the construction of any object program text that contains references to external linkage symbols;
- bind internal and external linkage addresses between separately assembled object program modules;
- prepare an absolute target program module.

A common binary object record format is used for each of the four above mentioned object module data types. In general, this format consists of one word of description followed by a variable number of words of data. The descriptive word includes the data type and the number of data words in the binary object record. The data words are formatted as data records, according to the binary object record type.

The following general format applies to each binary object record:



Length

Number of data words in the binary object record.

Type

Binary object record type: 1=DSC, 2=ESD, 3=ISD, 4=TXT, 5=END

Data Records (words 2-N)

See the following data record descriptions.

3.1 Object Module Description (DSC)

The object module description record containing information pertinent to the entire binary object program module. This information is useful to the post assembly linkage editor in order to identify and process the remaining ESD, ISD, TXT and END binary object records.

DSC records are formatted as follows:

Description Data Record (words 2-10)

The following fields are included in the description data record:

Object Module Name (words 2-4)

The main program or subprogram name.

Target Computer (words 5-7)

Specified in the compile options.

Memory Word Size (word 8)

Specified in the target description.

Low Memory Location (word 9)

The lowest assembled memory location that is used in this object program module.

Total Memory Locations (word 10)

The total number of memory locations that are used by this object program module (two pass assembly only).

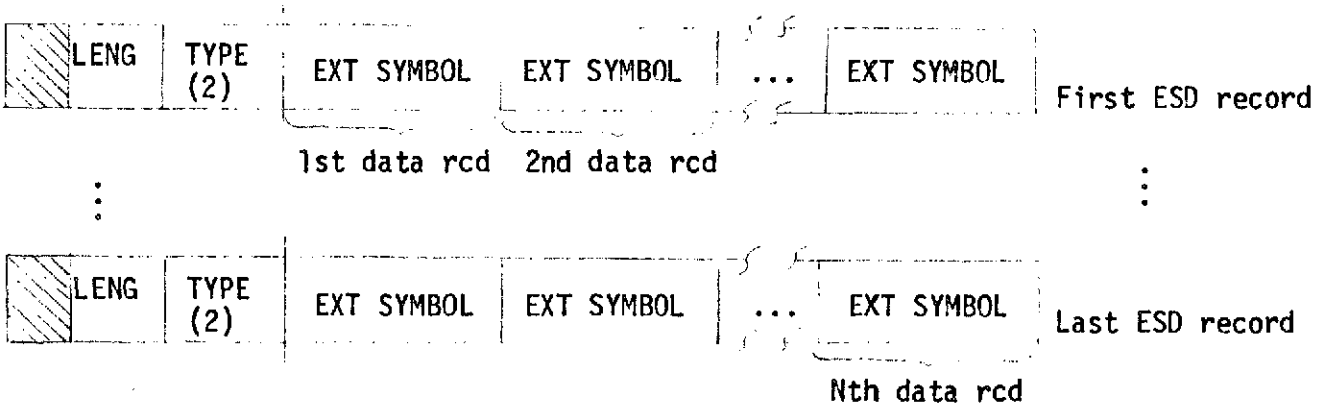
3.2 Global Symbol Dictionary (GSD)

The global symbol dictionary contains all of the external linkage symbols that are used by the object program module. Global symbol records contain external linkage symbol character strings. The sequential position of each record determines the sequence number of the global symbol character string that it contains. This information is used to reference global symbols in the symbol dictionary from within the object program text. As many GSD records are prepared as are required to contain all of the global symbol character strings.

External Symbols (3 words) - ESD

External linkage symbols are coded in packed integer code format. They are left justified in the assigned 3-word field and may use one, two or all three of the available words. External symbols correspond to externally defined procedure or data block names.

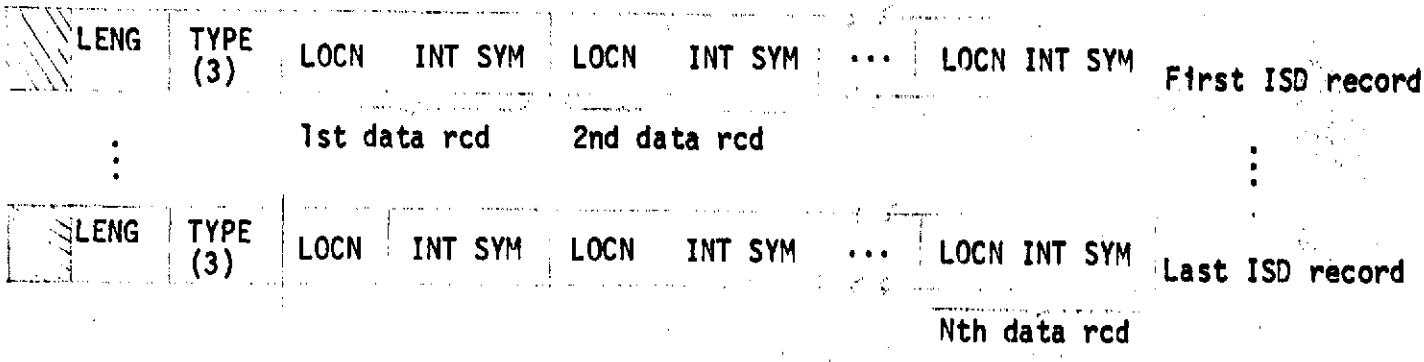
ESD records are formatted as follows:



Internal Symbols (4 words) - ISD

The internal symbol dictionary contains all of the internal linkage symbols (plus their addresses in the object program text) that are used by the object program module. Internal symbol records contain internal symbol character strings. Internal symbols may be located within this dictionary by character string comparison. As many ISD records are prepared as are required to contain all of the internal symbol character strings. An internal symbol corresponds to an entry point.

ISD records are formatted as follows:



Location (1 word)

The location counter value assigned by the assembler to the internal linkage symbol.

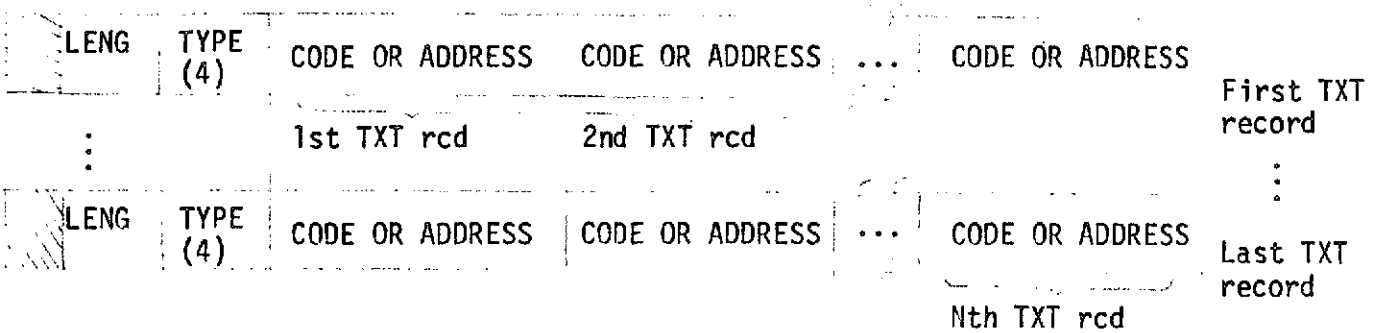
Internal Symbol (3 words)

Internal linkage symbols are coded in packed integer code format. They are left justified in the assigned 3-word field and may use one, two or all three of the available words.

3.3 Object Program Text (TXT)

Object program text is comprised of address and code records. Addresses may be relocated. Code is presented as fields of assembled core image bit strings that contain absolute or relocatable values and references to external symbols within the external symbol dictionary.

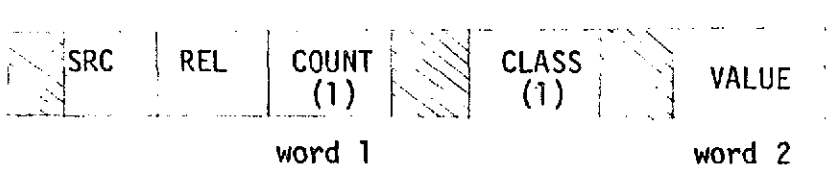
TXT records are formatted as follows:



Address Records (2 words)

All location counter values (addresses) are computed based on the number of bits in the addressing unit of the target computer main memory. TXT address records are output to the object program module for each non-sequential movement of the location counter. Upon recognizing a TXT address record, the linkage editor must replace its own location counter with the value provided and assign resulting absolute (or relocated) address as the absolute loader origin address of the set of TXT code records that follow.

Each TXT address record is formatted as follows:



Source

The source attribute of the address record: 0=origin change, 1=reserve.

Relocation

The relocation attribute of the location counter value: 0=absolute, 1=relocatable.

Count

The number of words remaining in the current data record: 1 (a constant).

Class

The text address data record sub-type: 1.

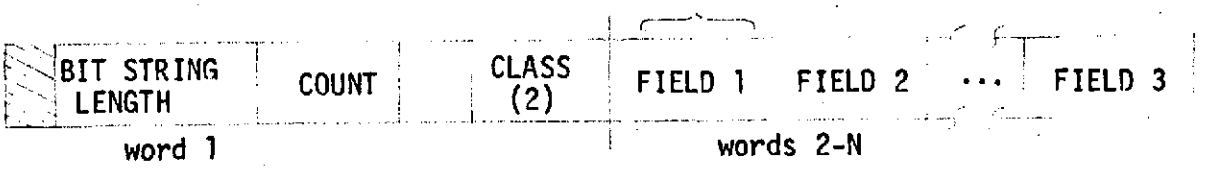
Value

The beginning location counter value of the code data records that follow.

Code Records and Fields

Code records contain instruction or data fields - some of which are absolute core image text, and some of which require internal address relocation and external address definition.

Each TXT code record is formatted as follows:



Bit String Length

The length in bits of the core image bit string that is to be constructed from information contained in the attached value fields.

Count

The number of words remaining in the current data record: the sum of all words in each value field.

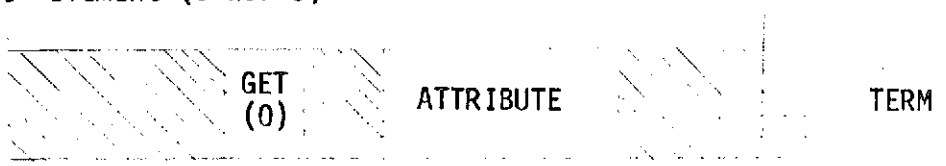
Class

The text code data record sub-type: 2.

Code Fields (words 2-N)

Code field elements are classified as terms ("get"), operators and end-of-string ("put"). They are arranged in 'PUT'/'GET' sequences as described below:

Term or "get" element (3 words):



Get

"Get" the following term.

Attribute

Evaluate the current term as being either: external, relocatable, absolute, or future for future symbols within intermediate language text.

Term

An absolute value, a symbol pointer, or an ESD symbol sequence number.

End-of-string or "put" element (1 word):

Start Bit

The starting bit position of the receiving field in the core image bit string.

Bit Length

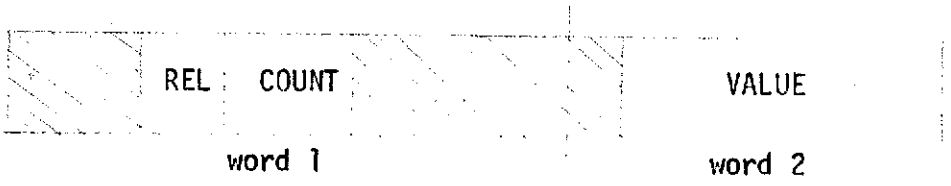
The number of bits that are contained in the receiving field in the core image bit string.

Put

Put the value of the current expression in the specified field of the current core image bit string.

3.4 Object Module End (END)

The object module end record designates the physical end of the object module. It may contain an optional execution start address. The END record is formatted as follows:



Relocation

The relocation attribute of the location counter value: 0=absolute, 1=relocatable.

Count

The number of words remaining in the current data record: 0=execution start address is not available, 1=execution start address is available.

Value

Value of the execution start address, if available.