NPS COBOL

User's Guide

PART 1

NPS MICRO-COBOL Version 2.1

User's Guide

PART ONE

Revised and Updated

ORGANIZATION

The Cobol compiler and run-time program are designed to run on an 8080 based system in an interactive mode. They will happily operate under both CP/M 2.2 and CP/M 3.1 (plus) systems.

These programs need a miminum 24k of main memory and a disc drive.

The following programs are supplied in the Cobol package:

COBOL.COM COBOL.OVL	The Cobol compiler.
EXEC.COM EXEC.OVR	The 'Runtime' Execution program.
COBOL-n.DOC	The Cobol program notes.
SERIES.PTn	The Cobol programming series.
EXAM-nn.COB	Example programs.
MAG.nnn	A complete Cobol program that stores details of magazine articles.
ADDRESS.nnn	A complete Cobol program that stores names, addresses etc.

GENERAL INTRODUCTION

The compiler is composed of two programs COBOL.COM and COBOL.OVL, each of which reads in and compiles a portion of the input file.

COBOL.COM reads the input program to the end of the Data Division and builds the symbol table. At the end of the Data Division, COBOL.COM is overlayed by COBOL.OVL which uses the symbol table to produce the code.
The output code is written to disc as it is produced to minimize the use of internal storage.

The EXECution program consists of two parts EXEC.COM and EXEC.OVL, the EXEC.COM Program builds the core image for the intermediate code and performs such functions as backstuffing addresses and offsetting address in subroutines. EXEC.COM then loads EXEC.OVL (the interpreter) into memory and transfers control to it. The interpreter is controlled by a large case statement that decodes the instructions and performs the required actions.

MICRO-COBOL ELEMENTS

This section contains a description of each element in the language and shows simple examples of their use. The following conventions are used in explaining the formats:

Elements enclosed in <broken> brackets are themselves complete entites and are described elsewhere in the manual.

Elements enclosed in {squiggley} brackets are choices, one of the elements which is to be used.

Elements enclosed in [square] brackets are optional.

All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated in lower case. These names are unrestricted in length, however they must be unique within the first 15 characters. The only other restriction on user names is that the first character must be an alpha character. The remainder of the user name can have any combination of representable characters in it.

The input to the compiler does not need to conform to standard COBOL format. Free form input will be accepted as the default condition. If desired, sequence numbers can be entered in the first six positions of each line. However, a toggle needs to be set to cause the compiler to ignore the sequence numbers.

The first character position on any line is used to indicate the following:

```
*  -  indicates a comment entry.
:  -  indicates a debugging line.
/  -  indicates a page eject.
```

COMPILER TOGGLES

There are six compiler toggles which are controlled by an entry following the compiler activation command, COBOL <filename>. The format of the entry consists of following <filename> by one space and then entering a "$" followed immediately by the desired toggles. There must be only one space after <filename> and no spaces between the "$" and the toggles.

The following is an example of a typical entry:

```
COBOL EXAMPLE $S
```

This entry would cause the compiler to ignore the first six characters (used for sequence numbers) at the beginning of each input line. In each case the toggle reverses the default value.

$C - No intermediate code. Default is OFF.

Setting this toggle speeds initial compilation for syntax checking. When this toggle is set the "CIN" file is empty.

$D - Debugging mode. Default is OFF.

This toggle sets the debugging mode, which means all the debugging lines (those with a ':' in column one) are compiled. If this toggle is not set in the ENVIRONMENT DIVISION of the

source program all debugging lines are treated as comments.

$L - list the input code on the screen as the program is compiled. Default is ON.

Error messages are displayed at the terminal in any case.

$P - Productions. List productions as they occur. Default is OFF.

$S - sequence numbers are in the first six positions of each record Default is OFF.

$T - Tokens. List tokens from the scanner. Default is OFF.

$W - Create a list file. Default is OFF.

A listing file is created when this toggle is set. When this toggle is not set the "LST" file will only contain error messages.

RUN TIME CONVENTIONS

This section explains how to run the compiler on the current system.

The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes sequence numbers then the compiler must be notified by setting the appropriate toggle.

The compiler is started by typing COBOL <filename>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same <filename> as the input file and will be given a file type of CIN. Any previous copies of the file will be erased. As with the CIN file, a LST file will be created with the same file name as the input file and any previous LST files with that name will be erased.

The interpreter is started by typing EXEC <filename>. The first program is a loader, and it will display:

"NPS MICRO-COBOL LOADER VERS 2.1"

followed by the display:

"LOAD FINISHED"

to indicates successful completion. The run-time package will be brought in by the EXEC routine, and execution should continue without interruption. Successful transfer of control to the interpreter will be indicated by the display:

"NPS MICRO-COBOL INTERPRETER VERS 2.1."

Completion of program execution will be indicated by the display

"X EXECUTION ERROR(S),"

where "X" is the number of errors which occurred during execution.

FILE INTERACTIONS WITH CP/M

CP/M treats all data files as Random Access files, CP/M cannot distinguish the sequential files, from the random Access files created under Cobol. The distinction between the two file types is specified by the way in which the file is opened by the Cobol program that uses the file.

This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists.

This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention removes the capability of reading variable length files in a random mode.

All of the physical records are 128 bytes in length and the program supplies buffer space for these records in addition to the logical records. Logical records may be of any desired length.

PROGRAM STRUCTURE

Cobol programs must followed a pre-defined program structure, the general structure and command syntax is contained in COBOL.PT2, COBOL.PT3 and COBOL.PT4 document files.

Programs follow the outline structure of:

A. IDENTIFICATION DIVISION.

   (details about the program)

B. ENVIRONMENT DIVISION.

   (details on the environment the program was designed to run in, such as machine type, file types etc)

C. DATA DIVISION.

   (declaration of variables, file structures etc)

D. PROCEDURE DIVISION.

   (the program commands)

IDENTIFICATION DIVISION

ELEMENT: IDENTIFICATION DIVISION Format

FORMAT: IDENTIFICATION DIVISION.

```
             PROGRAM-ID. <comment>.

             [AUTHOR. <comment>.]

             [DATE-WRITTEN. <comment>.]

             [SECURITY. <comment>.]
```

NOTES:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

```
             IDENTIFICATION DIVISION.
             PROGRAM-ID. SAMPLE.
             AUTHOR. HAL R POWELL.


             IDENTIFICATION DIVISION.
             PROGRAM-ID. WAGES-CALCULATION.
             *
             * This program calculate the employees net wage from
             * data entered through the keyboard and stores the net
             * wage in the WAGE.DAT file
```

ENVIRONMENT DIVISION

ELEMENT: ENVIRONMENT DIVISION Format

FORMAT:

```
             [ENVIRONMENT DIVISION.

             CONFIGURATION SECTION.

             SOURCE-COMPUTER. <comment> [DEBUGGING MODE].

             OBJECT-COMPUTER. <comment>.

             [INPUT-OUTPUT SECTION.

             FILE-CONTROL.

             <file-control-entry> . . .

             [I-O-CONTROL.

             SAME file-name-1 file-name-2 [file-name-3]
                  [file-name-4] [file-name-5]. ] ] ]
```

NOTES:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

The DEBUGGING MODE clause is used in conjunction with the ':' to indicate conditional compilation. If this clause is specified, all debugging lines (those with a ':' in column one) are compiled. If this clause is not

specified, all debugging lines are treated as comments. In addition the DEBUGGING MODE can be specified by using the compiler toggle 'D.'

<file-control-entry>

ELEMENT: <file-control-entry>

FORMAT:
```
1. SELECT file-name
       ASSIGN implementor-name
       [ORGANIZATION SEQUENTIAL]
       [ACCESS SEQUENTIAL].

2. SELECT file-name
       ASSIGN implementor-name
       ORGANIZATION RELATIVE
       [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
               {RANDOM RELATIVE data-name      }

3. SELECT file-name
       ASSIGN implementor-name
       ORGANIZATION INDEXED
       [ACCESS {SEQUENTIAL}].
               {RANDOM    }
```

NOTES:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ.

For CP/M the implementor name needs to conform to the normal specifications. Indexed is not implemented.

EXAMPLES:
```
SELECT CARDS
 ASSIGN CARD.FIL.
 SELECT RANDOM-FILE
 ASSIGN A.RAN
 ORGANIZATION RELATIVE
 ACCESS RANDOM RELATIVE RAND-FLAG.
```

DATA DIVISION

ELEMENT: DATA DIVISION Format

FORMAT:
```
DATA DIVISION.
 [FILE SECTION.
 [FD file-name
   [BLOCK integer-1 RECORDS]
   [RECORD [integer-1 TO] integer-3]
   LABEL RECORDS {STANDARD}]
                 {OMITTED}
   [VALUE OF implementor-name-1 literal-1
           [implementor-name-2 literal-2] . . . ].

   [<record-description-entry>] . . . ] . . .

 [WORKING-STORAGE SECTION.
```

```
    [<record-description-entry>] . . . ]

  [LINKAGE SECTION.
    [<record-description-entry>] . . . ]

    <comment>
```

DESCRIPTION:
This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following:

1. Label records make no sense on the diskette so no entry is required.

2. The VALUE OF clause has no meaning for CP/M. If a record is given two lengths as in RECORD 12 to 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

** END OF COBOL-1.DOC **

NPS COBOL

User's Guide

PART 2

NPS MICRO-COBOL Version 2.1

User's Guide

PART TWO

Revised and Updated

<comment>

ELEMENT: <comment>

FORMAT: any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES: this is a comment
anotheroneallruntogether
8080b 16K

NOTE : a '*' in column 1 to cause compiler to ignore sequence numbers.

<data-description-entry>

ELEMENT: <data-description-entry> format

FORMAT:
```
level-number {data-name}
             {FILLER   }
[REDEFINES data-name]
[PIC character-string]
[USAGE {COMP   }]
       {COMP-3}
       {COMPUTATIONAL}
       {DISPLAY}
[SIGN {LEADING} [SEPARATE]]
     {TRAILING}
[OCCURS integer]
[SYNC [LEFT]]
      [RIGHT]
[VALUE literal].
```

DESCRIPTION:
This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implemented, however existing programs that are transferred to MICRO-COBOL and use this feature will compile and execute successfully. All numeric data are maintained in DISPLAY format or packed BCD if the COMP-3 option is used.

EXAMPLES: 01 CARD-RECORD.

```
            02 PART PIC X(5).
            02 NEXT-PART PIC 99V99 USAGE DISPLAY.
            02 FILLER.
               03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.
               03 LONG-NUMB 9(15).
               03 STRING REDEFINES LONG-NUMB PIC X(15).
            02 ARRAY PIC 99 OCCURS 100.


PROCEDURE DIVISION
==================

ELEMENT:   PROCEDURE DIVISION Format

FORMAT:    1. PROCEDURE DIVISION [USING name1 [name2] . . . [name5]].
              section-name SECTION.
              [paragraph-name. <sentence> [<sentence> . . . ] . . . ] . . .

           2. PROCEDURE DIVISION [USING name1 [name2] . . . [name5]].
              paragraph-name. <sentence> [<sentence> . . . ] . . .

DESCRIPTION:
As is indicated, if the program is to contain sections, then the first
paragraph must be in a section.


<sentence>
==========

ELEMENT:   <sentence>

FORMAT:    <imperative-statement>
           <conditional-statement>


<imperative-statement>
======================

ELEMENT:   <imperative-statement>

FORMAT:    The following verbs are *always* imperative:

           ACCEPT . . . . . 15
           CALL . . . . . 17
           CLOSE  . . . . . 17
           DISPLAY  . . . . . 18
           EXIT . . . . . 19
           GO . . . . . 20
           MOVE . . . . . 22
           OPEN . . . . . 24
           PERFORM  . . . . . 25
           STOP . . . . . 27

           The following *may be* imperatives:

           ADD  . . . . . 16  }  *without* the SIZE ERROR
           DIVIDE . . . . . 19  }  statement
           MULTIPLY . . . . . 23  }
           SUBTRACT . . . . . 28  }

           DELETE . . . . . 18  }  *without* the INVALID option
```

```
          WRITE  . . . . . 29  }
          REWRITE  . . . . . 26  }


<conditional-statements>
========================

ELEMENT:   <conditional-statements>

FORMAT:    IF . . . . . 21
           READ . . . . . 26

           ADD  . . . . . 16  }  *with* the SIZE ERROR statement
           DIVIDE . . . . . 19  }
           MULTIPLY . . . . . 23  }
           SUBTRACT . . . . . 28  }

           DELETE . . . . . 18  }  *with* the INVALID option
           WRITE  . . . . . 29  }
           REWRITE  . . . . . 26  }


ACCEPT
======

ELEMENT:   ACCEPT

FORMAT:    ACCEPT <identifier>

DESCRIPTION:
This statement reads up to 255 characters from the console. The usage of
the item must be DISPLAY.

EXAMPLES:  ACCEPT IMAGE.
           ACCEPT NUM(9).


ADD
===

ELEMENT:   ADD

FORMAT:    ADD {identifier-1} [{identifier-2}]  . .   TO identifier-m
               {literal-1   }   {literal-2   }
           [ROUNDED] [SIZE ERROR <imperative-statement>]

DESCRIPTION:
This instruction adds either one number to a second with the result being
placed in the last location.  Multiple adds have not been implemented.

EXAMPLES:  ADD 10 TO NUM1
           ADD X TO Z ROUNDED.
           ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC


CALL
====

ELEMENT:   CALL

FORMAT:    CALL literal [USING name1 [name2] . . . [nameN]]
```

DESCRIPTION:
Control is transferred to the called procedure with an address of each of the parameters to be passed. The parameters map to those in the linkage section of the called program. The type and size of the parameters must match exactly.

```
EXAMPLES:  CALL 'NC152' USING DN1
           CALL 'PRINT'
           CALL 'ADDLIST' USING VAR1 VAR2 VAR3
```

CLOSE

ELEMENT: CLOSE

FORMAT: CLOSE file-name

DESCRIPTION:
Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

```
EXAMPLES:  CLOSE FILE1
           CLOSE RANDFILE
```

DELETE

ELEMENT: DELETE

FORMAT: DELETE file-name [INVALID <imperative-statement>]

DESCRIPTION:
This statement requires the file-name of the item to be deleted. The record is logically removed by filling it with a high value character, which is not displayable to the console or line printer. The logical record space can be used again by writing a valid record in its place.

```
EXAMPLES:  DELETE FILE-NAME
```

DISPLAY

ELEMENT: DISPLAY

```
FORMAT:    DISPLAY {identifier} [{identifier-1}] . . . [{identifier-N}]
                   {literal   }  {literal-1   }  . . .  {literal-N   }
```

DESCRIPTION:
This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 80 characters for literal values and 255 characters for identifiers.

```
EXAMPLES:  DISPLAY MESSAGE-1
           DISPLAY MESSAGE-3 10
           DISPLAY 'THIS MUST BE THE END'
```

DIVIDE
======

ELEMENT: DIVIDE

FORMAT: DIVIDE {identifier} INTO identifier-1 [ROUNDED] {literal }
         [SIZE ERROR <imperative-statement>]

DESCRIPTION:
The result of the division is stored in identifier-1; any remainder is lost.

EXAMPLES: DIVIDE NUMB INTO STORE
           DIVIDE 25 INTO RESULT

EXIT
====

ELEMENT: EXIT

FORMAT: EXIT [PROGRAM]

DESCRIPTION:
The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM terminates a subroutine and returns to the calling program. Its use in the main program causes no action to be taken.

EXAMPLES: EXIT PROGRAM
           EXIT

GO
==

ELEMENT: GO

FORMAT: 1. GO procedure-name
         2. GO procedure-1 [procedure-2] . . . procedure-20
              DEPENDING identifier

DESCRIPTION:
The GO command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES: GO READ-CARD.
           GO READ1 READ2 READ3 DEPENDING READ-INDEX

IF
==

ELEMENT: IF

FORMAT: IF <condition> {stmt-lst } END-IF
         IF <condition> {stmt-lst } ELSE {stmt-lst} END-IF
            {NEXT SENTENCE} {NEXT SENTENCE}

DESCRIPTION:
This is an enhanced version of the standard COBOL IF statement. Nesting of IF statement is allowed.

EXAMPLES:
```
IF A GREATER B ADD A TO C ELSE GO ERROR-ONE END-IF.

IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A END-IF.

IF A LESS B DISPLAY A DISPLAY B END-IF.

IF A GREATER B DISPLAY A DISPLAY B
  ELSE DISPLAY C DISPLAY D END-IF.

IF A GREATER B
  IF A GREATER C
    DISPLAY A
  ELSE
    DISPLAY C
END-IF
ELSE
  IF B GREATER C
     DISPLAY B
  ELSE
    DISPLAY C
  END-IF
END-IF.
```

NPS COBOL

User's Guide

PART 3

NPS MICRO-COBOL Version 2.1

User's Guide

PART THREE

Revised and Updated

MOVE

ELEMENT: MOVE

FORMAT:
```
MOVE {identifier-1} TO identifier-2
     {literal     }
```

DESCRIPTION:
The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alpha-numeric moves and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alpha-numeric move if the elements are redefined as alpha-numeric; also all group moves are alpha-numeric.

EXAMPLES:
```
MOVE SPACE TO PRINT-LINE.
MOVE A(10) TO B(PTR).
```

MULTIPLY

ELEMENT: MULTIPLY

FORMAT:
```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
         {literal   }
  [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:
The multiply routine uses a double length register to calculate the result. This allows the result generated to be of maximum precision. The actual value stored will be determined by the amount of storage allocated for the variable. Overflow will occur if the number in the register is larger than the variable. If the precision in the register is greater than the variable, truncation occurs unless the round option is specified.

EXAMPLES:
```
MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.
```

OPEN

ELEMENT: OPEN

FORMAT:
```
OPEN {INPUT file-name-1 } [{file-name-2}] . . .
```

```
                {OUTPUT file-name-1} [{file-name-2}] . . .
                {I-O file-name-1   } [{file-name-2}] . . .
```

DESCRIPTION:
The three types of OPENS have exactly the same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error. Multiple opens have not been implemented.

```
EXAMPLES:  OPEN INPUT CARDS.
           OPEN OUTPUT REPORT-FILE.
```

PERFORM

ELEMENT: PERFORM

```
FORMAT:    1. PERFORM procedure-name [THRU procedure-name-2]
           2. PERFORM procedure-name [THRU procedure-name-2]
                 {identifier} TIMES
                 {integer   }
           3. PERFORM procedure-name [THRU procedure-name-2]
                 UNTIL <condition>
           4. PERFORM procedure-name VARYING {identifier}
                 FROM {identifier} BY {identifier}
                 UNTIL <condition>
```

DESCRIPTION:
All four options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

```
EXAMPLES:  PERFORM OPEN-ROUTINE.
           PERFORM TOTALS THRU END-REPORT.
           PERFORM SUM 10 TIMES.
           PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.
           PERFORM REPEAT-AGAIN VARYING COUNTER FROM 1 BY 2
                 UNTIL COUNTER EQUAL 10.
```

READ

ELEMENT: READ

```
FORMAT:    1. READ file-name INVALID <imperative-statement>
           2. READ file-name END <imperative-statement>
```

DESCRIPTION:
The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

```
EXAMPLES:  READ CARDS END GO END-OF-FILE.
           READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.
```

REWRITE

ELEMENT: REWRITE

FORMAT: REWRITE record-name [INVALID <imperative>]

DESCRIPTION:
REWRITE is only valid for files that are open in the I-O mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from, the last executed read.

EXAMPLES: REWRITE CARDS.
REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.

STOP

ELEMENT: STOP

```
FORMAT:    STOP {RUN    }
                {literal}
```

DESCRIPTION:
This statement stops execution of the program. If a literal is specified, then the literal is displayed on the console and a prompt is displayed giving the operator the option of terminating or continuing program execution.

```
EXAMPLES:  STOP RUN.
           STOP 1.
           STOP 'INVALID FINISH'.
```

For the last two examples the following prompt is displayed:

```
     OPERATOR ENTER A <CR> TO CONTINUE
         OR ENTER AN "S" TO TERMINATE.
```

SUBTRACT

ELEMENT: SUBTRACT

```
FORMAT:    SUBTRACT {identifier-1} [identifier-2] . . . FROM identifier-m
                    {literal-1   } [literal-2   ]
                    [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:
Identifier-m, is decremented by the value of identifier/literal one. The results are stored back in identifier-m. Rounding and size error options are available if desired. Multiple subtracts have not been implemented.

```
EXAMPLES:  SUBTRACT 10 FROM SUB(12).
           SUBTRACT A FROM C ROUNDED.
```

WRITE

ELEMENT: WRITE

```
FORMAT:    1. WRITE record-name [{BEFORE} ADVANCING {INTEGER}]
                                {AFTER }           {PAGE   }
           2. WRITE record-name INVALID <imperative-statement>
```

DESCRIPTION:
The record specified is written to the file specified in the file section of the source program. The INVALID option only applies to random files.

```
EXAMPLES:  WRITE OUT-FILE.
           WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```

<condition>
==========

ELEMENT: <condition>

FORMAT: RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
   {literal-1}       {LESS   }  {literal-2   }
                     {EQUAL  }
```

CLASS CONDITION:

```
    identifier [NOT] {NUMERIC  }
                     {ALPHABETIC}
```

DESCRIPTION:
It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:
```
A NOT LESS 10.
LINE GREATER 'C'.
NUMB1 NOT NUMERIC
```

Subscripting
==========

ELEMENT: subscripting

FORMAT: data-name (subscript)

DESCRIPTION:
Any item defined with an OCCURS may be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:
```
A(10)
ITEM(SUB)
```

VII. ERROR MESSAGES

A. Compiler Fatal Messages

BR Bad read - disk error, no corrective action can be taken in the program.

CL Close error - unable to close the output file.

MA Make error - could not create the output file.

MO Memory overflow - the code and constants generated will not fit in the allotted memory space.

OP Open error - can not open the input file, or no such file present.

SO Stack overflow - LALR (1) parsing stack has exceeded its maximum

allowable size.

ST Symbol table overflow - the table is too large for the allocated space.

WR Write error - disk error, could not write a code record to the disk.

B. Compiler Warnings

DD Carriage Control error - The WRITE BEFORE/AFTER ADVANCING option can only be used with sequential files.

CE Close error - attempted to close a non-existing file.

DD Duplicate Declaration - the identifier name has been previously declared.

EL Extra levels - only 10 are allowed.

FT File type - the data element used in a read or write statement is not a file name.

IA Invalid access - the specified options are not an allowable combination.

ID Identifier stack overflow - more than 20 items in a GO - DEPENDING statement.

IS Invalid subscript - an item was subscripted but it was not defined by an OCCURS.

IT Invalid type - the field types do not match for this statement.

LE Literal error - a literal value was assigned to an item that is part of a group item previously assigned a value.

LV Literal value error - the PICTURE clause field type does not match the VALUE clause literal type.

L7 Level 77 error - level 77 used incorrectly.

MD Multiple decimals - a numeric literal in a VALUE clause contains more than one sign.

NF No file assigned - there was no SELECT clause for this file.

NI Not implemented - a production was used that is not implemented.

NP No production - no production exists for the current parser configuration; error recovery will automatically occur.

NV Numeric value - a numeric value was assigned to a non-numeric item.

OE Open error - attempt to open a file that was not declared; or attempted to open a file for I-O that was not a RELATIVE file.

OL OCCURS LEVEL - 01 and 77 levels can not contain an occurs clause.

PC Picture clause - a pic clause exceeds 30? characters.

P1 More than one float symbol declared.

P2 Non-numeric data in repetition clause or missing right parenthesis.

P3 Invalid or incompatible symbol in pic clause.

P4 Invalid symbol(s) embedded within a float symbol only /,0,B,',' allowed.

P5 Invalid combination of symbols in pic clause, type cannot be determined.

P6 Number of possible numeric entries exceeds register length max is 18.

PF Paragraph first - a section header was produced after a paragraph header, which is not in a section.

R1 Redefine nesting - a redefinition was made for an item which is part of a redefined item.

R2 Redefine length - the length of the redefinition item was greater than the item that is redefined. That is only allowed at the 01 level. This error message may be printed out one identifier past the redefining identifier record in which it occurred.

R3 Redefines misplaced - a redefines was attempted in the FILE SECTION of the source program.

SE Scanner error - the scanner was unable to read an identifier due to an invalid character.

SG Sign error - either a sign was expected and not found, or a sign was present when not valid.

SL Significance loss - the number assigned as a value is larger than the field defined.

TE Type error - the type of a subscript index is not integer numeric.

UD Undeclared identifier - the identifier was not declared.

UL Unresolved label - label has not been referenced. This warning will be given to all references to external subroutines.

VE Value error - a value statement was assigned to an item in the file section.

WL Wrong level error - program attempted to write a record other than an 01 level record to an output file.

C. Interpreter Fatal Errors

CL Close error - the system was unable to close an output file.

CO Call stack Overflow - insufficient memory available to transfer variable address' and/or return location for a subroutine call.

ME Make error - the system was unable to make an output file on the disk.

NF No file - an input file with the given name could not be opened.

OE Open Error - attempt to open a file which was already open.

OP Open Error - the system was unable to open a file.

PS Procedure Stack - not enough memory to load all subroutines.

SO Subroutine Overflow - subroutine symbol table overflow.

W1 Write non-sequential - attempted to WRITE to a file opened for INPUT or a file opened for I-O when ACCESS was SEQUENTIAL.

W2 Wrong key - attempted to change the key value to a lower value than the number of the last record written.

W3 Write input - attempted to WRITE to a file opened for INPUT.

W4 Write non-empty - attempted to WRITE to a non-empty record.

W5 Read output - attempted to READ a file opened for OUTPUT.

W6 Rewrite error - attempted to REWRITE to a file not opened for I-O.

W7 Rewrite error - attempted to REWRITE a record before reading the file; or multiple REWRITE attempts without doing a READ between each.

D. Interpreter Warning Messages

EM End mark - a record that was read did not have a carriage return or a line feed in the expected location.

GD Go to depending - the value of the depending indicator was greater than the number of available branch addresses.

IC Invalid character - an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.

NE Numeric Error - non-numeric data in an arithmetic operation.

W8 Write Error - the system was unable to write to an output file on the disk. Disk may be full.

SI Sign Invalid - the sign is not a "+" or a "-."