ZZ20-6415
March 1972

PALO ALTO SCIENTIFIC CENTER

IBM
Data Processing Division

MPL/135 - A PROGRAMMING LANGUAGE FOR SYSTEM/370 MODEL 135 MICROPROGRAMMING

John R. Walters and Daniel L. McNabb

MPL/135

A Programming Language

for

System/370 Model 135

Microprogramming

Daniel L. McNabb

John R. Walters

IBM Palo Alto Scientific Center

Abstract

MPL/135 is a language and compiler for microprogramming the System/370 Model 135, employing compiler technology, rather than assembler lore. The level of detail in the language is purposely kept high in order to enable the microprogrammer complete freedom in the use of the machine. This report is intended as an MPL/135 language reference manual and is to be used in conjunction with the functional specifications of System/370 Model 135.

Terms for IBM Subject Index

Microprogramming

Compilers

Programming Languages

IBM System/370 Model 135

1. INTRODUCTION

MPL/135 is a language for microprogramming the System/370 Model 135, patterned after MPL/145 (A), used for the Model 145. Both languages adhere to, and have been constructed using compiler language technology, rather than assembler lore. Yet both languages are purposely highly detailed and, therefore, quite machine dependent, because the very nature of microprogramming dictates that explicit utilization of every facit of the micromachine be kept available for the microprogrammer. Whether one should term these languages assemblers or compilers may be subject to some debate; but the authors have chosen to term them machine-dependent compilers in order to emphasize the fact that they contain a compiler-like control structure in which operands, not operators, are more apparent. Such a structure is entirely lacking in a classical assembler.

The MPL/135 language has been derived from PL/I, ALGOL, PL/360 (B), as well as from MPL/145 (A), yet it differs considerably from MPL/145 because of the great dissimiliarities between the two machines at the microinstruction level. The basic form of the language is that of PL/I; an MPL/135 program is a procedure, which may contain wholly nested procedures, thereby establishing a basic subprogram structure with capabilities for name scoping. Names are declared using a PL/I-like DECLARE (DCL) statement, which has been tailored to the storage and data types available in the Model 135. Because of the intent to retain explicit control with the microprogrammer, there are no implicit naming conventions in MPL/135; all names except simple labels must be declared explicitly at the head of the procedure or a containing procedure.

As can be seen by comparing the architectures of the Models 135 and 145, the Model 135 is quite 'vertical' in nature, appearing very similar to many ordinary small, binary machines, such as the System/7. On the other hand, the Model 145 offers considerably more parallelism, the effect being that more than one action can be specified in the framework of a single microinstruction, thus making the Model 145 more 'horizontal' in nature. Considerable effort was spent in designing MPL/145 to permit the microprogrammer to specify exactly these parallel functions and still retain a structure to the resulting programs. In the Model 135 no

such facility exists; therefore, the computational statements in MPL/135 look less like those in MPL/145 and more like those found in a conventional programming language. In this way, computation is normally achieved using an assignment statement of the form:

```
ALPHA <- ALPHA |+| BETA ;
```

where the exact nature of ALPHA and BETA are limited by the capabilities of the Model 135 hardware, and the operation, |+|, is an actual microinstruction in the machine with precisely only those properties which the hardware provides.

Additionally, MPL/135 contains several control statements which form a control structure, so that tedious housekeeping can be performed automatically without either restricting the microprogrammer or forcing him to resort to branch logic and thereby be required to create many extraneous labels. Included among these statements are:

1. An IF/THEN statement
2. An IF/THEN/ELSE statement
3. A DO WHILE statement
4. A DO CASE statement
5. An iterative DO statement

The microprogrammer is not penalized should he choose not to use these facilities.

2. The MPL/135 Language

2.1 Basic Definitions

The MPL/135 language is described in terms of the System/370 Model 135 microarchitecture (C) which comprises a processing unit and groups of storage elements. Each of the storage elements holds a content, also called a value. At any given time, certain significant relationships may exist between storage elements and values. These relationships may be recognized and altered, and new values created by the processing unit. The actions taken by the processor are determined by a program. The set of possible programs forms the MPL/135 language. An MPL/135 program is composed of, and therefore can be decomposed into elementary constructions

according to the rules of a syntax or grammar. To each elementary construction there corresponds an elementary action specified as a semantic rule of the language. The action denoted by a program is defined as the sequence of elementary actions corresponding to the elementary constructions which are obtained as the program is decomposed (parsed) by reading it from left to right.

2.1.1 The Processor

At any given time, the state of the microprocessor is defined by a collection of switches and latches as described in the Model 135 Functional Specifications (C).

2.1.2 Storage Elements

Storage elements are classified into five different types of registers and two different types of storage. Information is represented in one of the six following types:

1. Bit, a single binary digit valued either 0 or 1,
2. Digit, a 4-bit unsigned numeric quantity,
3. Byte, an 8-bit unsigned numeric quantity,
4. Half, a 16-bit unsigned numeric quantity,
5. Word, a 32-bit unsigned numeric quantity,
6. Double, a 64-bit unsigned numeric quantity.

The registers or storage may contain single elements of these data types or structures of them, depending on their physical limitations.

2.1.3 Relationships

The most fundamental relationship is that which holds between a cell and its value. It is known as containment; the cell is said to contain the value.

Another relationship holds between the cells which are the components of a structured cell, called an array or structure, and the structured cell itself. This is known as subordination. Structured cells are regarded as containing

the ordered set of the values of the component cells.

A set of relationships between values is defined by monadic and dyadic functions or micromachine operations, which the processor is able to evaluate or perform. The relationships are defined by mappings between values (or pairs of values) known as the operands and values known as the results of the evaluation. These mappings are described in detail in the Model 135 Functional Specifications (C).

2.1.4 The Program

A program contains declarations and statements written in free format on arbitrarily many lines. Declarations serve to list the cells, registers, labels, and symbolic constants which are involved in the algorithm described by the program, and to associate names, so-called identifiers with them. Declarations must precede statements within a given procedure. The statements specify the operations to be performed on these quantities, to which they refer through the use of identifiers.

A program is a sequence of tokens, which are either basic symbols, strings, or comments. Each token is itself a sequence of one or more characters. The following conventions are used:

a. Basic symbols constitute the basic vocabulary of the language (cf. appendix ii). They are either single characters or sequences of characters.

b. Strings are sequences of one or more characters enclosed in quote marks ('). A string may not be extended across an input line.

c. Comments are sequences of one or more characters preceded by the characters /* and followed by the characters */ . Comments may appear anywhere in a program but may not be embedded in a basic symbol or string; it is understood that they have no effect on the execution of a program.

In order that a sequence of tokens be an executable program, it must be constructed according to the rules of the syntax.

2.1.5 Syntax

A sequence of tokens constitutes an instance of a syntactic entity (or construct), if that entity can be derived from the sequence by one or more applications of syntactic substitution rules. In each such application, the sequence equal to the right side of the rule is replaced by the symbol which is its left side.

Syntactic entities are denoted by English words enclosed in the brackets "<" and ">". These words are used to describe approximately the nature of the syntactic entity, and where these words are used elsewhere in the text, they refer to that syntactic entity.

Syntactic rules are of the form:

<a> ::= <sequence>

where <a> is a syntactic entity (called the left side) and <sequence> is a finite sequence of tokens or syntactic entities (called the right side of the rule). The notation:

<a> ::= <b>|<c>|<d>

is used as an abbreviation for the syntactic rules:

<a> ::= <b>, <a> ::= <c>, <a> ::= <d>.

2.1.6 Syntactic Entities

The syntactic entities are listed in appendix i.

2.1.7 Basic Symbols

The basic symbols of MPL/135 are listed in appendix ii.

2.2. Data Manipulation Facilities

2.2.1 Identifiers

2.2.1.1
```
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
             N|O|P|Q|R|S|T|U|V|W|X|Y|Z
```

2.2.1.2
```
<digit>  ::= 0|1|2|3|4|5|6|7|8|9
```

2.2.1.3
```
<underscore> ::= _
```

2.2.1.4
```
<identifier> ::= <letter>            |
                 <identifier> <letter> |
                 <identifier> <digit>  |
                 <identifier> <underscore>
```

Notes:
An identifier is restricted to 12 characters; it may be used to designate a register, a storage cell, a symbolic constant, a program label, or a procedure.

Examples:

A	A9
TWELVECHARSY	HYPHEN_ED
DIRTY_NAME_1	YOU_NAME_IT
MICRO_1	MICRO10

2.2.1.5
```
<label>              ::= <identifier>
```

Notes:
Only 5 characters of a label are used; if a label is written with more than 5 characters, the first 3 and the last 2 characters are used.

2.2.1.6
<procedure name> ::= <identifier>

Notes:
Only 3 characters of a procedure name are used; if a procedure name is written with more than 3 characters, the leftmost 3 characters are used.

2.2.1.7
<symbolic constant> ::= <identifier>

2.2.2 Values

2.2.2.1
<decimal constant> ::= <digit> |
<decimal constant> <digit>

Notes:
A decimal constant is treated as an unsigned numeric quantity between 0 and 65535, right-justified with zero fill in a 16-bit halfword. 65535 is the largest value permitted.
Examples:

0	1
16382	5
39	0001
0123	65530

2.2.2.2
<hexadecimal digit> ::= <digit>|A|B|C|D|E|F

2.2.2.3
<hexadecimal string> ::= <hexadecimal digit>|
<hexadecimal string> <hexadecimal digit>

2.2.2.4
<hexadecimal constant> ::= ' <hexadecimal string> '

Notes:
A hexadecimal constant is limited to 4 hexadecimal digits, right-justified with zero fill in a 16-bit halfword.
Examples:

'0027'	'100F'
'F0F0'	'7007'
'F'	'7F'

2.2.2.5
<bitstring> ::= 0|1|
<bitstring> 0 |
<bitstring> 1

2.2.2.6
<bit constant> ::= ' <bitstring> ' B

Notes:
A bit constant is limited to 16 bits, right-justified with zero fill in a 16-bit halfword.
Examples:

'0000111100001111'B '1111000011110000'B
'1010101010101010'B '0101010101010101'B
'110'B

2.2.2.7
<special character> ::= =|$|;|:|'|->|+|-|,|.|<|¬|(|)|<blank>

2.2.2.8
<character> ::= <letter>|<digit>|
<underscore>|<special character>

2.2.2.9
<character string> ::= <character> |
<character string> <character>

Notes:
A single quote is written as a double quote within a character string.

2.2.2.10
<character constant> ::= ' <character string> ' A

Notes:
A character constant appearing in an arithmetic statement is limited to 2 characters; otherwise, a character constant is limited to 16 characters; the value of each character is its EBCDIC representation left-justified with blank fill in multiples of 16-bit halfwords.
Examples:

'MP'A 'L/'A
'13'A '5!'A
'MPL/135 IS BEST'A

2.2.3 Register Declarations

The Model 135 has the following five types of registers available for the microprogrammer:

1. General Purpose Registers, designated GPR,
2. Floating Point Registers, designated FPR,
3. Control Work Registers, designated CWR,
4. External Registers, designated EXT,
5. Work Registers, designated W.

An identifier may be associated with one of these registers using a DECLARE (or DCL) statement. Subsequent use of that identifier will result in the use of the register specified.

2.2.3.1
<register type> ::= HALF | HIHALF | LOHALF | WORD | DOUBLE

Note:
HIHALF and LOHALF designate the high order or the low order 16 bits in a 32-bit word. These designations are used to select a half word within a given word.

2.2.3.2
<dcl head> ::= DECLARE <identifier> |
DCL <identifier> |
<declared element> , <identifier>

Notes:
A DECLARE or DCL statement may be used to declare a single identifier or a group of identifiers, each identifier separated by a comma. The purpose of the statement is to relate certain attributes with the identifier so named.

2.2.3.3
<work half> ::= W0 | W1 | W2 | W3 | W4 | W5 | W6 | W7

2.2.3.4
<work word> ::= W01 | W23 | W45 | W67

2.2.3.5
<work dbl> ::= W03

2.2.3.6
<work reg> ::= <work half> | <work word> | <work dbl>

2.2.3.7
<work subscript> ::= <work reg> (BIT <decimal constant>) |
<work reg> (DIGIT <decimal constant>) |
<work reg> (BYTE <decimal constant>)

Notes:
The decimal constants refer to the bit, digit, or byte position within the halfword, word, or doubleword, starting with position 0. The letter B may be used as an abbreviation for BIT, D for DIGIT, and BY for BYTE.

2.2.3.8
<pointer> ::= <identifier> |
<work subscript>

Notes:
A pointer is any digit field in a work register used to address indirectly a specific general purpose, floating, or control work register. If the identifier is used, it must have been declared previously as a digit in a work register; if a subscripted work register is specified, it must be a digit subscript, c.f.2.2.3.7.

2.2.3.9
<GPR declaration> ::= <dcl head> GPR |
<dcl head> GPR (<pointer>) |
<dcl head> <register type> GPR |
<dcl head> <register type> GPR (<pointer>)

Notes:
If either of the first two forms are used, the type is assumed to be WORD; if either the second or fourth form is used, a reference to the identifier will imply the use of the pointer register. The GPR identifier may be used with an explicit pointer other than the implied pointer by writing the explicit pointer, followed by a pointer operator, followed by the GPR identifier.

Examples:

```
DCL ADAM GPR;
DECLARE EVE GPR (W0(DIGIT 0));
DCL TEMP LOHALF GPR;
DECLARE TEMPERATURE WORD GPR (W4(DIGIT 3));
```

2.2.3.10

```
<FPR declaration> ::= <dcl head> FPR                          |
                      <dcl head> FPR ( <pointer> )            |
                      <dcl head> <register type> FPR          |
                      <dcl head> <register type> FPR ( <pointer> )
```

Notes:
If either of the first two forms are used, the type is assumed to be WORD; if either the second or fourth form is used, a reference to the identifier will imply the use of the pointer register. The FPR identifier may be used with an explicit pointer other than the implied pointer by writing the explicit pointer, followed by a pointer operator, followed by the FPR identifier.
Examples:

```
DECLARE NAME FPR;
DECLARE AREA FPR (PTR_1);
DCL COUNT WORD FPR;
DCL CTR WORD FPR (W23(DIGIT 6));
```

2.2.3.11

```
<indirect CWR declaration> ::= <dcl head> CWR                 |
                               <dcl head> CWR ( <pointer> )   |
                               <dcl head> <register type> CWR |
                               <dcl head> <register type> CWR ( <pointer> )
```

Notes:
If either of the first two forms are used, the type is assumed to be WORD; if either the second or fourth form is used, a reference to the identifier will imply the use of the pointer register. The CWR identifier may be used with an explicit pointer other than the implied pointer by writing the explicit pointer, followed by a pointer operator, followed by the CWR identifier.
Examples:

```
DCL PTR CWR;
DECLARE X CWR (ALPHA);
DCL ZED WORD CWR;
DCL FINAL_COUNT WORD CWR (0_PTR);
```

2.2.3.12
<direct CWR word> ::= CWR0 | CWR1 | CWR2 | CWR3 |
CWR4 | CWR5 | CWR6 | CWR7

2.2.3.13
<direct CWR dbl> ::= CWR01 | CWR23 | CWR45 |CWR67

2.2.3.14
<direct CWR name> ::= <direct CWR dbl> | <direct CWR word>

2.2.3.15
<direct CWR half> ::= <direct CWR name> (HALF <decimal digit>)

Notes:
The decimal digit is used to specify the 0th or 1st half of a CWR word, or the 0th, 1st, 2nd, or 3rd halfword of a double word.

2.2.3.16
<direct CWR syn> ::= SYN (<direct CWR name>) |
SYN (<direct CWR half>)

2.2.3.17
<direct CWR declaration> ::= <dcl head> <direct CWR syn> |
<dcl head> <register type> <direct CWR syn>

Notes:
If the register type is omitted, the type is assumed to be WORD; if the type HIHALF or LOHALF is specified, and a conflicting HALF subscript is specified, the HALF specification will prevail.

2.2.3.18
<ext register> ::= EXT0 | EXT1 | EXT2 | EXT3 | EXT4 |
EXT5 | EXT6 | EXT7 | EXT8 | EXT9 |
EXT10 | EXT11 | EXT12 | EXT13 | EXT14 |
EXT15

2.2.3.19
<ext syn> ::= SYN (<ext register>)

2.2.3.20
<ext declaration> ::= <dcl head> <ext syn> |
<dcl head> BYTE <ext syn> |
<dcl head> HALF <ext syn>

Notes:
If the type is specified, only BYTE or HALF is valid; if the type is not specified, BYTE is assumed.
Examples:

```
DECLARE F_PTR SYN (EXT0);
DCL KONST BYTE SYN (EXT9);
```

2.2.3.21
<work type> ::= BIT | DIGIT | BYTE | HALF | WORD | DOUBLE

2.2.3.22
<work declaration> ::= <dcl head> SYN (<work reg>) |
<dcl head> SYN (<work subscript>) |
<dcl head> <work type> SYN (<work reg>) |
<dcl head> <work type> SYN (<work subscript>)

Notes:
If the type specification is omitted, it is determined from the type of the register appearing as the synonym. If a specified type conflicts with the register type, an error will be indicated, and the type of the register appearing as the synonym will prevail.
Examples:

```
DCL SCRATCH SYN (W0);
DCL ONE_BIT SYN (W0(BIT 2));
DCL HALFWORD HALF SYN (W3);
DCL A_DIGIT DIGIT SYN (W23(DIGIT 5));
```

It is possible to relate more than one identifier to a register or portion of a register. This can be achieved by defining one identifier to be the synonym for another, or by writing identical declarations for two or more identifiers. The general synonym form is:

2.2.3.23
<synonym declaration> ::= <dcl head> SYN (<identifier-1>)

where identifier-1 refers to a previously defined

identifier.
Examples:

```
DECLARE BETA SYN ( ALPHA );
DCL THIS_ONE SYN (THAT_ONE);
```

2.2.4 Storage Declarations

The Model 135 has two classes of storage, namely MAIN and CONTROL storage.

2.2.4.1
<based clause> ::= BASED (<pointer>)

2.2.4.2
<storage type> ::= BYTE | HALF | WORD

2.2.4.3
<main storage declaration> ::= <dcl head> MAIN |
<dcl head> <storage type> MAIN |
<dcl head> MAIN <based clause> |
<dcl head> <storage type> MAIN <based clause>

Notes:
If the storage type is not specified, it is assumed to be HALF. If a based clause is used, the pointer must be a previously-defined even-odd work register pair, c.f. 2.2.3.4; the pointer will be used implicitly unless it is overridden by a pointer operator and an explicit pointer register used in conjunction with the identifier.
Examples:

```
DCL STORAGE MAIN;
DECLARE CELLS WORD MAIN;
DECLARE AREA MAIN BASED (O_PTR);
DCL AREA_2 HALF MAIN BASED(W01);
```

2.2.4.4
<control storage declaration> ::= <dcl head> CONTROL |
<dcl head> <storage type> CONTROL |
<dcl head> CONTROL <based clause> |
<dcl head> <storage type> CONTROL <based clause>

Notes:
If the storage type is omitted, it is assumed to be HALF.
If the based clause is specified, the pointer must be a previously-defined halfword work register, c.f. 2.2.3.3; the pointer will be used implicitly unless it is overridden by a pointer operator and an explicit pointer register.
Examples:

```
DCL CSTG CONTROL;
DCL STORE HALF CONTROL;
DCL SPACE CONTROL BASED(W23);
DCL ROOM WORD CONTROL BASED(PTR);
```

2.2.4.5

```
<private control> ::= DSP0  | DSP1  | DSP2  | DSP3  |
                      DSP4  | DSP5  | DSP6  | DSP7  |
                      DSP8  | DSP9  | DSP10 | DSP11 |
                      DSP12 | DSP13 | DSP14 | DSP15 |
                      DSP16 | DSP17 | DSP18 | DSP19 |
                      DSP20 | DSP21 | DSP22 | DSP23 |
                      DSP24 | DSP25 | DSP26 | DSP27 |
                      DSP28 | DSP29 | DSP30 | DSP31
```

2.2.4.6

```
<private control subscript> ::=
              <private control> ( BYTE 0 ) |
              <private control> ( BYTE 1 )
```

2.2.4.8

```
<directly addressed private control> ::=
      <dcl head> SYN ( <private control> )                    |
      <dcl head> SYN ( <private control subscript> )          |
      <dcl head> <type> SYN ( <private control> )             |
      <dcl head> <type> SYN ( <private control subscript> )
```

Notes:
The permissible type specifications are BYTE or HALF. If the type specification is omitted, the type HALF is assumed.
Examples:

```
DCL PRIVATE SYN (DSP19(BYTE 1));
DCL OWN     SYN (DSP27);
DCL MEIN    BYTE SYN (DSP12(BYTE 0));
DCL DEIN    HALF SYN (DSP19);
```

2.2.4.9
<common storage> ::= DSC0 | DSC1 | DSC2 | DSC3 |
DSC4 | DSC5 | DSC6 | DSC7 |
DSC8 | DSC9 | DSC10 | DSC11 |
DSC12 | DSC13 | DSC14 | DSC15 |
DSC16 | DSC17 | DSC18 | DSC19 |
DSC20 | DSC21 | DSC22 | DSC23 |
DSC24 | DSC25 | DSC26 | DSC27 |
DSC28 | DSC29 | DSC30 | DSC31 |
DSC32 | DSC33 | DSC34 | DSC35 |
DSC36 | DSC37 | DSC38 | DSC39 |
DSC40 | DSC41 | DSC42 | DSC43 |
DSC44 | DSC45 | DSC46 | DSC47 |
DSC48 | DSC49 | DSC50 | DSC51 |
DSC52 | DSC53 | DSC54 | DSC55 |
DSC56 | DSC57 | DSC58 | DSC59 |
DSC60 | DSC61 | DSC62 | DSC63

2.2.4.10
<directly addressed common> ::=
<dcl head> SYN (<common storage>) |
<dcl head> HALF SYN (<common storage>)

Notes:
If the HALF specification is omitted, the type HALF is assumed.
Examples:

```
DCL AREANA SYN (DSC59);
DCL SCHPAZE HALF SYN (DSC2);
```

2.2.5 Symbolic Constants

2.2.5.1
<symbolic term> ::= <identifier> | <constant>

2.2.5.2
<symbolic expression> ::= <symbolic term> |
<symbolic expression> + <symbolic term> |
<symbolic expression> - <symbolic term>

2.2.5.3
<symbolic equate> ::= EQU (<symbolic expression>)

2.2.5.4
<symbolic constant> ::= <dcl head> <symbolic equate>

Notes:
Each identifier appearing in a symbolic expression must have appeared previously as a symbolic constant. The effect of the declaration of a symbolic constant is to produce a named constant, fixed at compilation time.
Examples:

```
DCL RELCON EQU (ALPHA+BETA-'3F');
DCL REGNO  EQU (13);
DCL BASE   EQU (BETA-4);
```

2.2.6 Offsets and Structures

2.2.6.1
<structure head> ::= DECLARE 1 <identifier> OFFSET |
DCL 1 <identifier> OFFSET

Notes:
The purpose of this portion of a structure declaration is to name an entire structure, using the level number one as the structure header. The keyword OFFSET must appear at this level.

2.2.6.2
<offset control> ::= OFFSET | <type> | <type> OFFSET |
<symbolic equate> | <null>

Notes:
The keyword, OFFSET, may optionally appear at any subitem level in the structure definition. Permissible types are BYTE, HALF, and WORD.

2.2.6.3
<structure subitem> ::=
<structure head> , <integer> <identifier> <offset control> |
<structure subitem> , <integer> <identifier> <offset control>

Notes:
As the structure is decomposed from left to right, byte offsets are determined from the top node and assigned as symbolic constants to the various subitem identifiers.

Integers used at the subitem levels must match, i.e. , an integer must be equal to one of its predicessors or be greater than its immediate predicessor. Elementary subitems represent the lowest elements within the structure, and each elementary subitem must have a type specification as part of the offset control.

2.2.6.4
The byte counter within a structure declaration may be set or reset by including a symbolic equate phrase, c.f., 2.2.5.3, at any level.

An example of a structure declaration is as follows:

```
DCL 1 COUNTERS OFFSET,
      2 WORD_1 WORD,
        3 BYTE_1 BYTE,
        3 BYTE_2 BYTE,
        3 BYTE_3 BYTE,
        3 BYTE_4 BYTE,
      2 WORD_2,
        4 FIRST  HALF,
        4 LASTT  HALF,
      /* THIS RESETS BYTE COUNTER TO OVERLAY WORD_2 */
      2 WORD_3 EQU(WORD_2),
        4 W_21 BYTE,
        4 W_22 BYTE,
        4 W_23 BYTE,
        4 W_24 BYTE;
```

2.2.7 Label Declarations

Simple labels may be written with any MPL/135 statement, except a DECLARE statement. A label definition is of the form:

2.2.7.1
<label definition> ::= <label> :

Note:
Only 5 characters of a label are used, c.f. 2.2.1.5.

Subscripted labels of 4 or 16 elements may be used for 4 or

16 way selective branching. In this case, each element of the label must appear as a subscripted label:

2.2.7.2
<subscripted label definition> ::= <label> (<integer>) :

Note:
Permissible values of the integer are 0 to 3 for a 4-element label array and 0-15 for a 16-element label array.

Subscripted labels must also appear in a DECLARE statement:

2.2.7.3
<label declaration>::=
DECLARE <identifier> <label dimension> LABEL; |
DECLARE <identifier> LABEL;

2.2.7.4
<label dimension> ::= (4) | (16)

Notes:
The purpose of the label declaration is to declare a label or a label array. Label arrays may be either 4 or 16 elements, and are further declared within the text of the program. These subscripted labels may then be used in a GO TO statement to perform 4 or 16-way branching.
Examples:

```
DCL LABELEG(4) LABEL;
DECLARE BRANCH_TAB(16) LABEL;
```

2.3 Specification Statements

2.3.1 Register to Register Specification

2.3.1.1
<work/reg statement> ::=
<work target> <- <register source> ; |
<work target> <- <work target> WITH <shifts> ;

Note:
In form two the target register on the left side of the assignment arrow must be the same register as the work

target register on the right side of the assignment arrow. The equal sign, "=", may be used in place of the assignment arrow, "<-".

2.3.1.2
<reg/work statement> ::=
    <register target> <- <work source> ;

Notes:
The type of the work source must match that of the register target. The equal sign, "=", may be used in place of the assignment arrow, "<-".

2.3.1.3
<shifts> ::= SL2 | RSL2 | SR4 | RSR4 | RSR8

Notes:
These shift operations may only be used in conjunction with work registers. SL2 specifies a left shift of 2 bits, RSL2 specifies a left ring shift of 2 bits, SR4 specifies a right shift of 4 bits, RSR4 specifies a right ring shift of 4 bits, and RSR8 specifies a right ring shift of 8 bits.

2.3.1.4
<register source> ::= <identifier> |
<work reg> |
<direct CWR name>|
<constant> |
<identifier-1> -> <identifier>

Notes:
The identifier may name a work register, constant, indirect auxiliary register, or external register.
Examples:

```
    /* IDENTIFIERS USED ARE SELF-DESCRIBING */
     WORK_REG1 <- WORK_REG1 WITH SL2;
     WORK_REG1 <- WORK_REG2;
     WORK_REG_WD <- CONTROL_WORK0;
     WORK_REG2 <- WORK_DIG -> GEN_REG(HALF 0);
    /* NEXT, WORK_DIG IS ASSUMED POINTER */
     WORK_REG_DBL <- FLT_PT_REG(DB);
     WORK_REG1 <- EXTERN_3;
     WORK_REG1 <- 25-SYMB_CONSTANT;
```

2.3.1.5
<register target> ::= <direct CWR name> |
<identifier> |
<identifier-1> -> <identifier>

Notes:
The identifier may name a direct CWR register, an indirect auxiliary register, or an external register.

The source register may be of type HALF, WORD, or DOUBLE; if the constant option is specified, the constant may only be of type HALF. The target register must be of the same type as the source register.

The 2's complement of an indirect auxiliary identifier may be used as the source operand in conjunction with a work register as a target operand. This is specified by preceding the indirect auxiliary identifier by a minus sign "-".

A source work register of type HALF may be shifted using any of the shift specifications in the WITH field. A source work register of type WORD or DOUBLE may only be shifted with SL2 or SR4.

The only types permitted in conjunction with an external register operand are BYTE and HALF.

In half word operations using either an indirect auxiliary or CWR direct operand as the source or target and a work register as the other operand, the work register must be an even register for the HI half and an odd register for the LO half.
Examples:

```
GPR_AUX(HI) <- WORK_3;
FPR_AUX     <- WORK_23;
WORK_DIG -> CWR_AUX <- WORK_23;
CWR_DIR_3 <- WORK_45;
EXTERN_7 <- WORK_2;
```

2.3.2 Register to Storage Specification

2.3.2.1
<work source> ::= <identifier>

Notes:
The identifier may only represent a work register of type

HALF or WORD.

2.3.2.2

```
<storage name> ::=          <identifier-1>      |
        <identifier-2> ->   <identifier-1>      |
        <identifier-2> ->   <identifier-1>(B)   |
        <identifier-2> ->   <identifier-1>(H)   |
        <identifier-2> ->   <identifier-1>(W)
```

Notes:
The storage may be an identifier-1 representing an element located in Control Storage, Main Storage, Key Storage, Directly-addressed Private Control Storage, or Directly-addressed Common Control Storage. Permissible types for Control or Main Storage are BYTE, HALF, or WORD; the permissible type for Key Storage is BYTE; the permissible type for Common Control Storage is HALF; and the permissible types for Private Control Storage are either BYTE or HALF. In the second form, above, identifier-2 specifies an explicit pointer register used with the pointer operator "->" to locate the element in the designated storage area. If the first form is used, the implicit pointer must have been declared using a BASED attribute. Forms 3, 4, and 5, respectively, specify a type of BYTE, HALF, or WORD.

2.3.2.3

```
<register/storage specification> ::=
      <storage name> <- <work source> ;          |
      <storage name> <- <work source> WITH <options-1>  ;
```

2.3.2.4

```
<options-1> ::= TEST | INC | DEC
```

Notes:
The types of the work register source and the storage target must match. The equal sign, "=", may be used in place of the assignment arrow, "<-".
Examples:

```
WORK_PTR -> MAINST(H) <- WORK_REG2; /*HALFWORD*/
MAINST_WPTR(B) <- WORK_SRC_HLF WITH INC;
CONTROLST <- WORK_HALF WITH TEST;
PRIV_CONTRL <- WORK_REG2;
COMMON_CTL23 <- WORK_REG5;
```

2.3.3 Storage to Register Specification

2.3.3.1
<storage/register specification> ::=
        <work target> <- <storage name> ;
        <work target> <- <storage name> WITH <options-2> ;

2.3.3.2
<options-2> ::= SET | TSK | SKIP | INC | DEC

Notes:
The types of the storage/storage source and the work register target must match. The effect of this statement is to replace the contents of the target work register with the contents of the source storage cell. The equal sign, "=", may be used in place of the assignment arrow, "<-".
Examples:

```
WORK_REG(BY 1) <- MAINST(B) WITH INC;
WORK_WORD <- MAINST(W) WITH SKIP;
WORK_REG1 <- PTR -> CONTROL(H);
WORK_REG1 <- DIRECT_PRIV3;
WORK_REG  <- DIRECT_CTRL;
```

2.4 Arithmetic Operations

2.4.1 Work Register Arithmetic

2.4.1.1
<work source> ::= <identifier>

Notes:
The identifier may only represent a work register of type HALF, WORD, or DOUBLE.

2.4.1.2
<work op> ::= |+| | |-| | |O| | |A| | |X|

Notes:
The operator |+| denotes addition, |-| denotes subtraction, |O| denotes "or", |A| denotes "and", and |X| denotes "exclusive or".

2.4.1.4
<work/work register arithmetic statement> ::=
    <work target> <- <work target> <work op> <work source> ;

Notes:
The types of the work register target and the work register source must match. The equal sign, "=", may be used in place of the assignment arrow, "<-".
Examples:

```
WORK1 <- WORK1 <- |+| WORK2;
WORK0123 <- WORK0123 |-| WORK0123;
```

2.4.2.1
<constant op> ::= |+| | |-| | |O| | |A¬| | |A|

Notes:
The operator |+| denotes addition, |-| denotes subtraction, |O| denotes "or", |A¬| denotes "and not", and |A| denotes "and".

2.4.2.2
<work/constant arith statement> ::=
   <work target> <- <work target> <constant op> <constant> ;

Notes:
The type of the target and the constant must be HALF. The equal sign, "=", may be used in place of the assignment arrow, "<-". If the operators |+| or |-| are used, the hexadecimal format for the constant must be as follows, with K indicating a hexadecimal digit:

```
00OK
00K0
0K00
K000
K0K0
00KK
KKKK
```

Examples:

```
WORK1 <- WORK2 |A| 'FF11';
WORK1 <- WORK1 |+| '0303';
WORK1 <- WORK1 |-| 35-SYMB_CONST;
```

2.4.3 Work to CWR

2.4.3.1
<CWR op> ::= |+| | |-|

Note:
The operator |+| denotes addition and the operator |-| denotes subtraction.

2.4.3.2
<work/CWR arith statement> ::=
<work target> <- <work target> <CWR op> <identifier> ; |
<work target> <- <work target> <CWR op> <identifier>
WITH <shifts> ;

Notes:
The identifier may represent a CWR register of type HALF, WORD, or DOUBLE; this type must match that of the work register target. The equal sign, "=", may be used in place of the assignment arrow, "<-". The work register may be shifted before the operation is performed if either SL2 or SR4 is specified in the WITH field.
Example:

```
WORK23 <- WORK23 |+| CWR_REG0 WITH SL2;
```

2.4.4 Work to Auxiliary Register

The following operations operate between the general purpose register, the floating point registers, and the indirect CWR registers and the work registers.

2.4.4.1
<aux reg op> ::= |+| | |-| | |O| | |A| | |X|

Notes:
The operator |+| denotes addition, |-| denotes subtraction, |O| denotes "or", |A| denotes "and", and |X| denotes "exclusive or".

2.4.4.2
<work/auxiliary arith statement> ::=
<work target> <- <work target> <aux reg op> <identifier> ;

Notes:
The identifier may represent a general purpose register, floating point register, or an indirect CWR register. The type of this register must match that of the work register. The equal sign, "=", may be used in place of the assignment arrow, "<-". Permissible types are: HALF, WORD, or DOUBLE. In half word operations using indirect auxiliary or direct CWR operands, the work register must be even for the HI half and odd for the LO half.
Examples:

```
WORK0123 <- WORK0123 |A| FPR_REG01;
WORK2    <- WORK2 |+| GPR_REG(H 0);
```

2.4.5 Set - Reset

2.4.5.1

<set reset target> ::= <identifier>

Notes:
The identifier may represent a work register of type HALF, or an external register of type BYTE.

2.4.5.2

<set statement> ::= SET <set reset target> BY <constant> ;

Notes:
The type of the constant must be BYTE for an external register target and HALF for a work register target. The effect of the statement is to "or" the target register with the value of the constant, so that each bit in the constant will be placed in the target register.
Examples:

```
SET WORK2 BY 'F0F0';
SET WORK2  BY 95-SYMB_CONST;
SET EXTERN15 BY '55';
```

2.4.5.3

<reset statement> ::= RESET <set reset target> BY <constant>;

Notes:
The type of the constant must be BYTE for an external register and HALF for a work register. The effect of this statement is to "and not" the constant to the target register, so that bits in the constant are used to reset

corresponding bits in the register.
Examples:

```
RESET WORK2 BY 35+SYMB1-SYMB2+CONST3;
RESET EXTERN11 BY 'FF';
```

2.5 Control Statements

2.5.1 Branch Statement

2.5.1.1

```
<label identifier> ::= <identifier>                    |
                       <qualifier>  .  <identifier>
```

Notes:
In the first form the identifier must be a label defined in the current procedure nest. In the second form, the qualifier must be a procedure name, and the identifier must be a label within that procedure.

2.5.1.2

```
<subscripted label> ::= <label identifier> ( <identifier> )|
                        <label identifier> ( <constant> )
```

Notes:
The label identifier must have been defined previously in a DECLARE statement with type LABEL and a subscript of 4 or 16. In form 1 the identifier within the parenthesis may represent a constant or a work register digit. The value of the constant or the work register digit must be 3 or less if the label identifier is dimensioned 4, or 15 or less if the label identifier is dimensioned 16.

2.5.1.3

```
<goto> ::= GO TO  |  GOTO
```

2.5.1.4

```
<goto statement> ::= <goto> <label identifier>   ;   |
                     <goto> <subscripted label>  ;
```

Notes:
The effect of the goto statement is to branch to the designated label.

```
Examples:
     GO TO FIXUP;        /* FIXUP IS A LOCAL LABEL */
     GO TO LABELEG(4);   /* LABELEG IS SUBSCRIPTED 16 */
     GOTO LABELEG(W4(D 3));
     GO TO LABELEG(INDEX);
```

2.5.2 Call Statement

2.5.2.1

```
<call label> ::= <identifier>                |
                 <qualifier> . <identifier>
```

2.5.2.2

```
<call statement> ::=
                 CALL <call label> USING <call register>  ;
```

Notes:
The call register may be either W5 or W6.

```
Examples:
     CALL SQRT USING W5;
     CALL SQRT USING LINKREG; /* LINKREG = W5 */
```

2.5.3 Return Statement

2.5.3.1

```
<return statement> ::=  RETURN  ;              |
                        RETURN USING <return register>  ;
```

Notes:
The return register may be any work register, W0 - W7. If form 1 is used, and the procedure statement specifies a return register, that register will be used. If the procedure statement does not specify a return register and form 1 is used, W6 will be used.

```
Examples:
     RETURN;  /* W6 IS THE LINK IF NOT DEFINED IN PROCEDURE */
     RETURN USING LINKREG;  /* LINKREG = W5 */
```

2.6 Blocks

Blocks are used to collect one or more statements and treat

the collection as a unit. Several types of blocks permit conditional, iterative, or selective execution of the statements contained within the block, each statement in which may also be a sub-block, etc.

2.6.1 Simple Blocks

Simple blocks are used to collect groups of statements. Conditional or iterative execution of the collection does not occur as part of the block itself. Simple blocks are headed by the simple DO statement and terminated by an END statement.

2.6.2 Simple DO

<simple do statement> ::= DO ;

2.6.3 END Statement

<end statement> ::= END ;

2.6.4 While Block

A while block is headed by a DO WHILE statement and terminated by an END statement. Its purpose is to execute the statements within the block 0 or more times, as long as the specified condition is met.

2.6.4.1

```
<condition> ::= <identifier>      |
                ¬ <identifier>    |
                  <identifier> = 0 |
                  <identifier> = 1 |
                  <identifier> ¬=0 |
                  <identifier> ¬=1 |
                  <special condition>
```

Notes:
The identifier must be defined as a bit in a work register.
Examples:

```
     A = 0      /* HERE A = W4(B 5) */
     TESTBIT    /* HERE TESTBIT = W3(B 2) */
```

2.6.4.2
<special condition> ::=
IRCH0 | PRI | MODE | RELOC | CARRY | ALU |
ALU1_7 | INVDEC | RI

Note:
These special conditions are described in the Model 135 functional specifications.

2.6.4.3
<do while statement> ::= DO WHILE (<condition>) ;

Notes:
The effect of this statement is to execute repeatedly the contents of the block headed by the DO WHILE statement and terminated by the matching END statement until the condition is no longer valid. If the condition is initially invalid, the statements will not be executed.
Examples:

```
DO WHILE (A = 1);
   . . .
   . . .
END;
. . .
DO WHILE (TESTBIT);
   . . .
   . . .
END;
. . .
DO WHILE (W3(B 4));
   . . .
   . . .
END;
```

2.6.5 Case Block

A case block is headed by a DO CASE statement and terminated by an END statement. Its purpose is to execute the i-th statement within the block. This statement may in turn be a sub-block.

2.6.5.1
<do case statement> ::=

```
DO CASE ( <identifier> ) ;                    |
DO CASE ( <identifier> ) OF 4 ;               |
DO CASE ( <identifier> ) OF 16 ;
```

Notes:
The identifier must represent a work register digit. The effect of the statement is to execute selectively one of the 4 or 16 statements which constitute the DO CASE block. The number of statements within the block must be exactly 4 or 16 (null statements designated by ";" may be used), each of which may also be a block, etc. In form 1 a case of 16 is assumed.
Examples:

```
DO CASE (W4(D 3)) OF 4;
     /*CASE 0*/ . . . ;
     /*CASE 1*/ . . . ;
     /*CASE 2*/ . . . ;
     /*CASE 3*/ . . . ;
END;
```

2.6.6 Count Block

A count block is headed by a DO COUNT statement and is terminated by an END statement. The effect of the statement is to use the count register to iterate the execution of the statements contained within the block. The count register counts down to 0 from any value between 0 and 255.

2.6.6.1
<do count statement> ::=

```
DO COUNT <- <identifier> ;                    |
DO COUNT <- <identifier> BY 1 ;               |
DO COUNT <- <identifier> BY 2 ;               |
DO COUNT <- COUNT ;                           |
DO COUNT <- COUNT BY 1 ;                      |
DO COUNT <- COUNT BY 2 ;
```

Notes:
The equal sign, "=", may be used in place of the assignment arrow, "<-". In forms 1, 2, and 3, the identifier represents a work register whose contents are placed in EXT15 while the low-order 8 bits are placed in the count register, which is then decremented by 1, 1, or 2, respectively, at the end of

the block. In forms 4, 5, and 6 the current value of the count register is decremented by 1, 1, or 2, respectively. The statements in the block are executed at least once until the count becomes zero or negative. Statements within the count block may not store from the work registers into external registers without altering the contents of the count register; similarly, a branch into a count block does not initialize the count register.
Examples:

```
DO COUNT <- A BY 2;
     . . .
     . . .
END;
 . . .
DO COUNT <- COUNT;
     . . .
     . . .
END;
```

2.7 IF Statement

2.7.1
<if statement> ::=

```
IF <condition> THEN <statement>  |
IF <condition> THEN <statement> ELSE <statement>
```

Notes:
Permissible conditions are those specified in sections 2.6.4.1 and 2.6.4.2, as well as the testing of any work register digit for zero or non-zero. The statements in either form 1 or form 2 may be simple non-if statements, or any of the blocks described in section 2.6. IF statements may not appear as statements within If statements, unless they are contained in blocks.

In form 1, the effect of the IF statement is to execute the statement or block following the keyword THEN only if the condition specified is true. In form 2, the statement or block following the keyword THEN is executed if the condition is true, while the statement or block following the keyword ELSE is executed if the condition is false.

```
Examples:
     IF A=1 THEN WORK2 <- WORK2 |+| CONST;
     IF CARRY THEN WORK2 <- WORK3; ELSE WORK2 <- WORK4;
     IF MODE THEN
        DO;
           IF ALU THEN WORK4 <- WORK1;
              ELSE WORK4 <- WORK5;
        END;
     ELSE
        DO;
           IF CARRY1 THEN WORK4 <- WORK3;
              ELSE WORK4 <- WORK2;
        END;
```

2.8 Miscellaneous Statements

2.8.1 IFETCH Statement

2.8.1.1

```
<ifetch statement> ::=
        IFETCH ;                                          |
        IFETCH WITH <ifetch conditions> ;                 |
        IFETCH USING <work reg> WITH <ifetch conditions> ;
```

Notes:
The purpose of this statement is to generate a System/370 instruction fetch microinstruction as indicated. For further details, the reader is referred to the Model 135 functional specifications.

2.8.1.2

```
<ifetch conditions> ::=
         NODBUG | NOIRPT | SUCCBR | CC= <integer>
```

Note:
The integer may only be a 0, 1, 2, or 3.

2.8.2 Special SET Statement

2.8.2.1

```
<special set statement>::=
      SET SELECT ( <work register digit>  ) ;
      SET INTERRUPT ( <work register digit> ) ;
```

Note:
The work register digit may be specified directly or with a predefined identifier. For further reference, the reader is referred to the Model 135 functional specifications.

2.8.3 DATA Statement

Constants and addresses may be entered as data by means of the DATA statement.

2.8.3.1
<datum> ::= <constant> | <label>

2.8.3.2
<data instruction head> ::=
DATA :
DATA (<hexadecimal location>) :

Note:
In form 2 the hexadecimal location may be specified as any constant or constant expression. This will become the starting address of the subsequent data.

2.8.3.3
<data instruction body> ::=
<data instruction head> <datum> |
<data instruction body> , <datum>

2.8.3.4
<data instruction> ::= <data instruction body> ;

Note:
Constants are formatted as indicated in section 2.2.2.

2.9 Procedural Blocks

An MPL/135 program must be contained within at least one procedural block, to which the labels in the program and identifiers appearing in the program and in DECLARE

statements are attached. Within this procedural block there may be wholly contained subprocedural blocks nested arbitrarily many levels. Such subprocedures must appear at the end of the containing procedures.

Declaration statements must appear first within a procedural block, and in subprocedural blocks declaration statements may be used to specify identifiers local to that block. Local identifier names may be the same as names in other procedural blocks, but with other attributes. Their scope is limited to the sub-block in which they are defined as well as any other subprocedural blocks contained in that defining block, in which the identifier is not explicitly defined.

2.9.1 Procedure Block

A procedure block is headed by a procedure statement and terminated by a matching END statement, cf., 2.6.3.

2.9.1.1
<procedure statement> ::=
<label> : PROC ; |
<label> : PROCEDURE ; |
<label> : PROC (<identifier>) ; |
<label> : PROCEDURE (<identifier>) ;

Notes:
The label is used to name the procedure and may be up to 3 characters long, cf., 2.2.1.6. In form 3 and form 4 the identifier is used to specify the standard return work register, cf., 2.5.3.1.

3. The Compiler

The MPL/135 compiler is written in PL/1 and translates MPL/135 source programs into Model 135 assembler source text in two passes. While the entire compiler constitutes a single PL/1 procedure (separate procedures are used to initialize the read-only syntax tables), it is essentially separated into four sections, namely:

a. a syntax scanner, which collects meta symbols, or tokens, to obtain the various syntax rules of the language. Having collected tokens for a given production, the syntax scanner invokes the code generation routines to apply the appropiate semantic interpretation, whereupon it then substitutes the left hand meta symbol for the sequence of tokens which constitute the right hand of the production. This process continues until the entire program has been parsed.

b. a lexical scanner, which accepts the source program and produces from it the meta symbols as required by the syntax scanner. A meta symbol number is returned to the syntax scanner for each terminal symbol of the language, c.f., Appendix II.

c. a code generation routine, which is invoked by the syntax scanner whenever a syntax rule is located, so that it may produce the proper semantics in the form of table entries, object text generation, etc. for each grammar rule.

d. a table adjustment routine and second text pass to produce proper labels for the Model 135 Assembler.

The diagram in Appendix III shows the relation between these four sections.

3.1 Syntax Scanner

The LALR syntax techniques described by LaLonde (D) are employed. Appendix I contains the actual syntax employed in the compiler, and Appendix II lists the Terminal symbols with a cross-reference to the productions in Appendix I.

3.2 Lexical Scanner

The lexical scanner reads the source text and returns meta symbol numbers, or tokens, for identifiers, constants, strings, or reserved words. These are shown in Appendix II.

3.3 Code Generation

The code generation routines generate text suitable for the microassembler.

3.4 Second Pass

The purpose of the second pass is to delete superfluous labels, which may have been generated in the first pass. At the end of the first pass, the symbol table is examined for the presence of such multiple labels, and the extraneous ones are marked; so that they can be replaced by the proper label. The object text is then read, and the marked labels deleted and replaced by the appropiate ones.

APPENDIX I

MPL/135

CONCRETE SYNTAX

```
 1  <PROGRAM>  ::=  _|_   <PROCEDURE NEST>   _|_

 2  <PROCEDURE NEST>  ::=  <PROCEDURE DEFINITION>
 3                       |  <PROCEDURE NEST> <PROCEDURE DEFINITION>

 4  <PROCEDURE DEFINITION>  ::=  <PROCEDURE HEAD> <STATEMENT LIST> <ENDING>

 5  <PROCEDURE HEAD>  ::=  <PROCEDURE NAME> ;
 6                       |  <PROC ARG HEAD> <VARIABLE-A> ) ;
 7                       |  <PROCEDURE HEAD> <DECLARE STATEMENT>

 8  <PROC ARG HEAD> ::=  <PROCEDURE NAME> (

 9  <PROCEDURE NAME> ::=  <LABEL DEFINITION> PROCEDURE
10                      |  <LABEL DEFINITION> PROC

11  <STATEMENT LIST>  ::=  <STATEMENT>
12                       |  <STATEMENT LIST>  <STATEMENT>

13  <DECLARE STATEMENT> ::=  <DECLARE STATEMENT-A>  ;

14  <DECLARE STATEMENT-A> ::=  <DECLARE HEAD> <ATTRIBUTE LIST>
15                           |  <DECLARE HEAD>

16  <DECLARE HEAD> ::=  <DECLARE HEAD-C> <IDENTIFIER-A>
17                    |  <DECLARE HEAD LIST> )

18  <DECLARE HEAD LIST> ::=  <DECLARE HEAD-BB> <IDENTIFIER-A>

19  <DECLARE HEAD-BB> ::=  <DECLARE HEAD-C> (
20                       |  <DECLARE HEAD LIST> ,

21  <DECLARE HEAD-C> ::=  <DECLARE HEAD-D> <INTEGER>
22                      |  <DECLARE HEAD-D>

23  <DECLARE HEAD-D> ::=  DCL
24                      |  DECLARE
25                      |  <DECLARE STATEMENT-A>  ,

26  <ATTRIBUTE LIST> ::=  <MAPPING ATTRIBUTE>
27                      |  <STYPE-1> <MAPPING ATTRIBUTE>
28                      |  <STYPE-1>

29  <STYPE-1> ::=  BIT
30              |  DIGIT
31              |  BYTE
32              |  HALF
33              |  WORD
34              |  DOUBLE
35              |  HIHALF
36              |  LOHALF
```

```
37  <STYPE> ::=  <STYPE-1>
38             |  <ABBREVIATION>
39             |  BY

40  <MAPPING ATTRIBUTE> ::=  <STORAGE ATTRIBUTE>
41                         |  <SYN HEAD>  <VARIABLE-B>  )
42                         |  LABEL  ( <INTEGER> )
43                         |  <EQU HEAD>  <PRIMARY EXPR> )
44                         |  OFFSET

45  <STORAGE ATTRIBUTE> ::=  <STORAGE HEAD> <VARIABLE-B>  )
46                         |  <STORAGE>

47  <SYN HEAD> ::=  SYN  (

48  <STORAGE> ::=  GPR
49               |  FPR
50               |  CWR
51               |  MAIN
52               |  CONTROL

53  <STORAGE HEAD> ::=  <STORAGE>  (

54  <EQU HEAD> ::=  EQU (

55  <LABEL DEFINITION> ::=  <VARIABLE-B>  :

56  <STATEMENT> ::=  <BASIC-GR STATEMENT>
57                 |  <IF STATEMENT>
58                 |  <ADDR FIX>  :  <BASIC-GR STATEMENT>
59                 |  <ADDR FIX>  :  <IF STATEMENT>
60                 |  <DATA STATEMENT>
61                 |  <PROCEDURE DEFINITION>

62  <IF STATEMENT> ::=  <IF STATEMENT-A>
63                    |  <IF STATEMENT-A> <ELSE> <BASIC-GR STATEMENT>

64  <IF STATEMENT-A> ::=  <IF HEAD> <BASIC-GR STATEMENT>

65  <ELSE> ::=  ELSE

66  <IF HEAD> ::=  <IF0>  THEN
67               |  <IF1>  THEN
68               |  <IF2>  THEN
69               |  <LABEL DEFINITION> <IF HEAD>

70  <IF0> ::=  IF ~ <VARIABLE-B>
71           |  IF ~ <LATCHES>
72           |  IF <VARIABLE-B>
73           |  IF <LATCHES>

74  <IF2> ::=  <IF1> <RELATION> <PRIMARY EXPR>
```

```
 75  <BASIC-GR STATEMENT> ::=  <BASIC STATEMENT>
 76                          |  <GROUP>

 77  <BASIC STATEMENT> ::=  <ASSIGNMENT> ;
 78                       |  <SET-RESET> ;
 79                       |  <CALL STATEMENT> ;
 80                       |  <RETURN STATEMENT> ;
 81                       |  <GOTO STATEMENT> ;
 82                       |  <IFETCH STATEMENT> ;
 83                       |  <SELECT/INTERRUPT> ;
 84                       |  RESTORE ;
 85                       |  ;
 86                       |  <UNTRANS STATEMENT>
 87                       |  <LABEL DEFINITION> <BASIC STATEMENT>

 88  <GROUP> ::= <GROUP HEAD> <ENDING>

 89  <GROUP HEAD> ::=  <DO> ;
 90                  |  <DO WHILE> ) ;
 91                  |  <DO CASE> ;
 92                  |  <DOFOR> BY <INTEGER> ;
 93                  |  <DOFOR> ;
 94                  |  <GROUP HEAD> <STATEMENT>

 95  <DO CASE> ::= <DO CASE-A> )
 96              | <DO CASE-A> ) OF <INTEGER>

 97  <DO CASE-A> ::=  <DO CASE HEAD> <VARIABLE-B>

 98  <DO CASE HEAD> ::= <DO> CASE (

 99  <DO WHILE> ::=  <WHILE HEAD-1>
100               |  <WHILE HEAD-2>
101               |  <WHILE HEAD-3>

102  <WHILE HEAD-1> ::=  <DO WHILE HEAD> <VARIABLE-B>
103                   |  <DO WHILE HEAD> <LATCHES>

104  <WHILE HEAD-2> ::=  <DO WHILE HEAD> ~ <VARIABLE-B>
105                   |  <DO WHILE HEAD> ~ <LATCHES>

106  <WHILE HEAD-3> ::=  <WHILE HEAD-1> <RELATION> <PRIMARY EXPR>

107  <DO WHILE HEAD> ::= <DO> WHILE (

108  <DOFOR> ::=  <DO COUNT HEAD> <VARIABLE-B>
109            |  <DO COUNT HEAD> COUNT

110  <DO COUNT HEAD> ::= <DO> COUNT <REPLACE>

111  <DO> ::=  DO
112         |  <LABEL DEFINITION>  DO
```

```
113  <ENDING> ::=  <END HEAD> ;
114              |  <LABEL DEFINITION> <ENDING>

115  <END HEAD> ::=  END <IDENTIFIER-A>
116                |  END

117  <LATCHES> ::=  IRCH0
118              |  PRI
119              |  MODE
120              |  RELOC
121              |  CARRY
122              |  ALU
123              |  CARRY1
124              |  CARRY8
125              |  FPO
126              |  ALU1_7
127              |  INVDEC
128              |  RI

129  <SET-RESET> ::=  <SET-RESET-A> <PRIMARY EXPR>

130  <SET-RESET-A> ::=  SET <VARIABLE-B> BY
131                   |  RESET <VARIABLE-B> BY

132  <CALL STATEMENT> ::= <CALL HEAD-A> <VARIABLE-B>

133  <CALL HEAD-A> ::=  CALL <VARIABLE-B> USING

134  <RETURN STATEMENT> ::=  RETURN USING <VARIABLE-B>
135                       |  RETURN

136  <GOTO STATEMENT> ::=  GO TO <VARIABLE-B>
137                     |  GOTO <VARIABLE-B>

138  <IFETCH STATEMENT> ::=  <IFETCH HEAD> <IFETCH OPTIONS>
139                       |  IFETCH

140  <IFETCH HEAD> ::=  IFETCH USING <VARIABLE-B> WITH
141                   |  IFETCH WITH

142  <IFETCH OPTIONS> ::=  CC = <INTEGER>
143                     |  NODBUG
144                     |  NOIRPT
145                     |  SUCCBR

146  <SELECT/INTERRUPT> ::=  <SEL/INT HEAD> <VARIABLE-B> )

147  <SEL/INT HEAD> ::=  SET SELECT (
148                   |  SET INTERRUPT (

149  <ASSIGNMENT> ::=  <ASSIGNMENT-A>
150                  |  <ASSIGNMENT-A> <WITH CLAUSE>

151  <ASSIGNMENT-A> ::=  <VARIABLE> <REPLACE> <EXPRESSION>
```

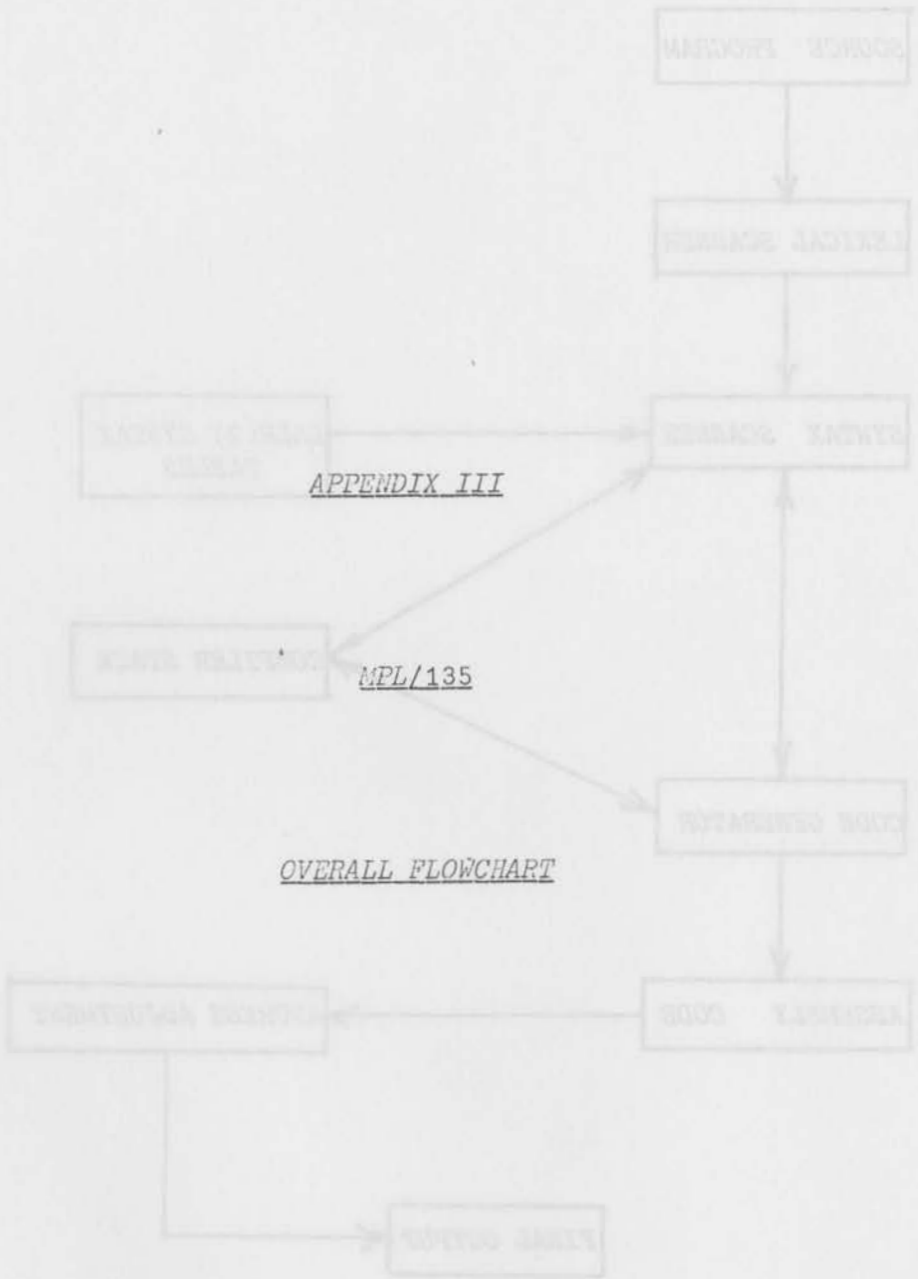
```
<WITH CLAUSE> ::= WITH <WITH OPTIONS>
                | <WITH CLAUSE> , <WITH OPTIONS>

<WITH OPTIONS> ::= SET
                 | TEST
                 | TSK
                 | SKIP
                 | INC
                 | DEC
                 | SL2
                 | SR4
                 | RSL2
                 | RSR4
                 | RSR8

<REPLACE> ::= < -
            | =

<RELATION> ::= =
             | ~ =

<EXPRESSION> ::= <TERM>
               | <EXPRESSION-A> <PRIMARY EXPR>

<EXPRESSION-A> ::= <VARIABLE> <MACHINE OPN>

<MACHINE OPN> ::= |+|
                | |-|
                | |A|
                | |O|
                | |X|
                | |A~|

<DATA STATEMENT> ::= <DATA LIST> ;

<DATA LIST> ::= <DATA HEAD> <PRIMARY EXPR>
              | <DATA LIST-A> <PRIMARY EXPR>

<DATA LIST-A> ::= <DATA LIST> ,

<DATA HEAD> ::= DATA <ADDR FIX> :
              | DATA :
              | <DATA HEAD> <LABEL DEFINITION>

<ADDR FIX> ::= <ADDR FIX-A> )

<ADDR FIX-A> ::= <AT HEAD> <PRIMARY EXPR>

<AT HEAD> ::= AT (

<PRIMARY EXPR> ::= <PRIMARY EXPR-A>
```

```
189  <PRIMARY EXPR-A> ::=  <TERM>
190                      |  <PRIMARY EXPR-A> + <TERM>
191                      |  <PRIMARY EXPR-A> - <TERM>

192  <TERM> ::=  -  <VARIABLE>
193           |  -  <CONSTANT>
194           |  <VARIABLE>
195           |  <CONSTANT>

196  <VARIABLE> ::=  <VARIABLE-B>
197               |  <VARIABLE-B> <PT ARROW> <VARIABLE-B>

198  <VARIABLE-B> ::=  <VARIABLE-A>
199                 |  <SUBSCRIPT HEAD> )

200  <SUBSCRIPT HEAD> ::=  <SUBSCRIPT HEAD-AA> <STYPE> <INTEGER>
201                     |  <SUBSCRIPT HEAD-AA> <PRIMARY EXPR>

202  <SUBSCRIPT HEAD-AA> ::=  <VARIABLE-A> (

203  <VARIABLE-A> ::=  <IDENTIFIER-A>
204                 |  <IDENTIFIER-B> <VARIABLE-A>

205  <IDENTIFIER-B> ::=  <IDENTIFIER-A> .

206  <IDENTIFIER-A> ::=  <IDENTIFIER>
207                   |  <ABBREVIATION>
208                   |  <WITH OPTIONS>

209  <CONSTANT> ::=  <INTEGER>
210               |  <NUMERIC STRING>
211               |  <STRING>
```

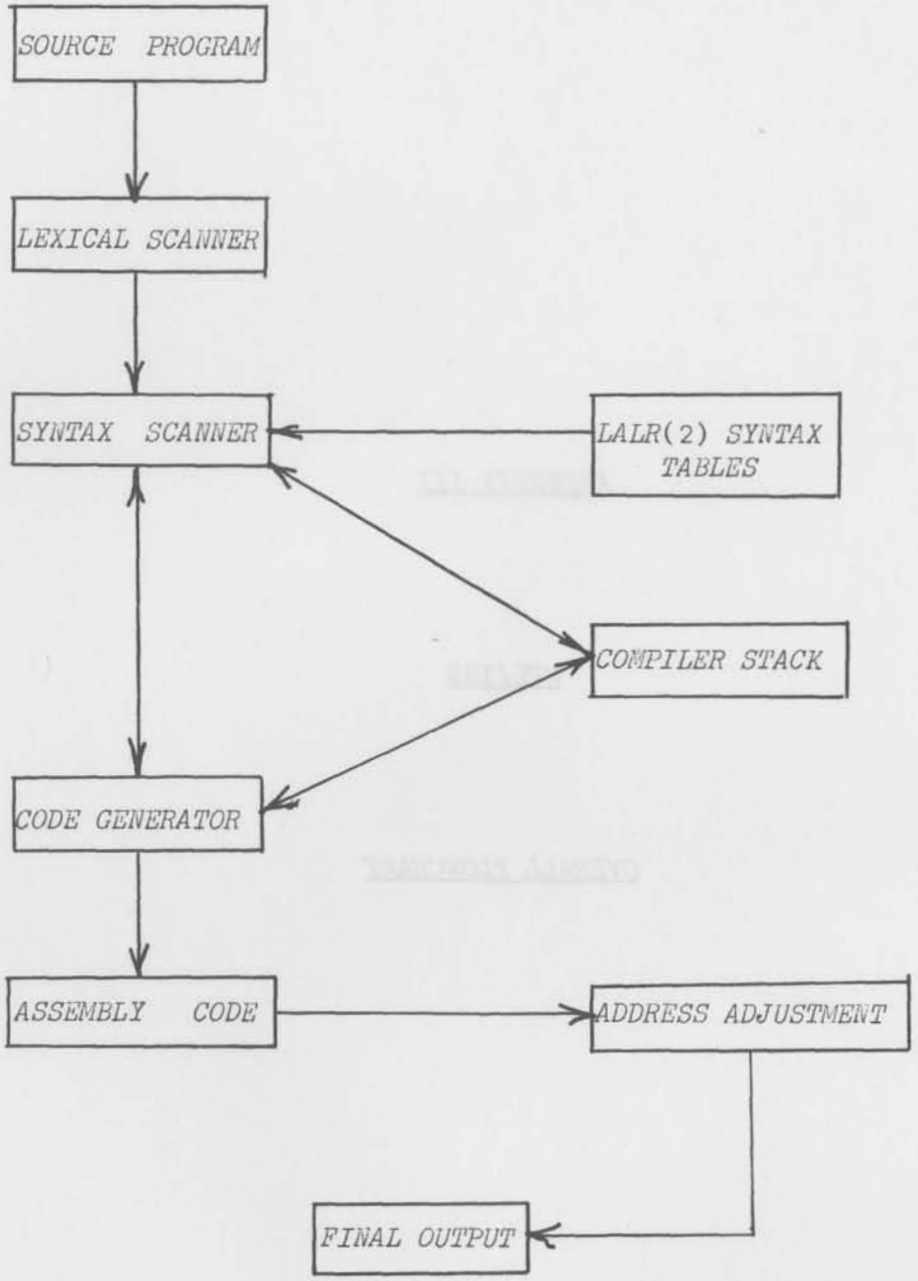
APPENDIX II

TERMINAL SYMBOLS

SYMBOL	USED IN PRODUCTION
.	205
<	165
(	8,19,42,47,53,54,98,107,147, 148,187,202
+	190
)	6,17,41,42,43,45,90,95,96, 146,185,199
;	5,6,13,77,78,79,80,81,82,83, 84,85,89,90,91,92,93,113,178
~	70,71,104,105,168
-	165,191,192,193
,	20,25,153,181
:	55,58,59,182,183
=	142,166,167,168
AT	187
BY	39,92,130,131
CC	142
DO	111,112
GO	136
IF	70,71,72,73
OF	96
RI	128
TO	136
|+|	172
|-|	173
|*A*|	174
|*O*|	175
|*X*|	176
_|_	1
ALU	122
BIT	29
CWR	50
DCL	23
DEC	159
END	115,116
EQU	54
FPO	125
FPR	49
GPR	48
INC	158
PRI	118
SET	130,147,148,154
SL2	160



OVERALL FLOWCHART


SOURCE PROGRAM
LEXICAL SCANNER
SYNTAX SCANNER
LALR(2) SYNTAX TABLES
COMPILER STACK
CODE GENERATOR
ASSEMBLY CODE
ADDRESS ADJUSTMENT
FINAL OUTPUT

APPENDIX IV

REFERENCES AND BIBLIOGRAPHY

References

(A). McNabb, D.L. and Walters, J.R., "MPL/145 A Langauge and Compiler for System/370 Model 145 Microprogramming", IBM Palo Alto Scientific Center, (May 1971).

(B). Wirth, Niklaus, "A Programming Language for the 360 Computers", Technical Report No CS 53, Computer Science Department, Stanford University, (June 1967).

(C). IBM Corporation, "System/370 Model 135 System Specifications", IBM Corporation, (1971).

(D). Lalonde, W., "An Efficient LALR Parser Generator", Technical Report CSRG-2, University of Toronto, (1970).

Bibliography

(1). Husson, S., "Microprogramming: Practices and Principles", Prentice-Hall, (1970).

(2). Rosin, R., "Contemporary Concepts of Microprogramming and Emulation", Computing Surveys of the ACM, Vol 1, No 4, (December 1969), pps. 197-212.