ML80 LANGUAGE MANUAL

Luiz R. Pedroso

October 1975

ABSTRACT

A structured systems programming language for the 8080 microprocessor is described. The language provides an algebraic notation for machine-level register and data operations, while incorporating most control constructs available in block-structured high-level languages. Compile-time facilities include recursive macros, expression evaluation, and conditional compilation. Object programs are relocatable, and independently compiled procedures can be linked at load-time. The resident compiler executes on a microcomputer system with 16K bytes of main memory.

STO[illegible]

.M80

M81 - MACRO PR

.L80

L81 - PARSER

.80PAR

.80SYM

L82 - CODE GEN

.80CODE

.80DATA

.80RELOC

L83 - LINKER

TABLE OF CONTENTS

ML80 LANGUAGE MANUAL

A. AN INFORMAL DESCRIPTION OF ML80

1. General

ML80 is a language designed to encourage structured programming, and thus it incorporates many of the usual features of block structured languages. In particular, the following mechanisms are provided:

- nested block structure;
- controllable scope for variables and data;
- adequate control constructs, including:

  IF-THEN-ELSE,

  DO-WHILE,

  DO-BY,

  DO-CASE,

  REPEAT-UNTIL;
- procedures.

Another important feature, oriented toward modular programming, is the provision of EXTERNAL and COMMON declarations. These declarations allow program references to code or data, respectively, generated from previous compilations, thus allowing construction of procedure libraries and data bases.

Although the control structures of ML80 are similar to those found in high-level languages, data manipulation operations are provided at the machine level. All instructions available in the 8080 instruction set can be directly accessed by the programmer. The notation utilized is algebraic, rather than the usual mnemonic operation codes employed by assembly languages. Nested expressions also can be built using the basic arithmetic, logical and data transfer operations, thus contributing to program conciseness and readability.

Macro processing facilities are provided in ML80 to further aid formulation of complex operations using primitives at the machine level. These facilities include:

- compile-time expression evaluation;
- parameterized macros;
- conditional macro expansion;
- recursion.

ML80 is, in fact, composed of two independent languages: M80, a macro oriented language, and L80, a machine oriented language.

Although designed to work in conjunction with one another, these languages are completely independent and self-contained. A programmer can write and run L80 programs without any knowledge of M80. Similarly, programs (or any texts) containing statements in the M80 language can be input to the M80 macro processor without regard to the source language.

Given the modularity of M80 and L80, these two languages are described in a totally independent manner, in the following sections. Examples of programs written in ML80, which incorporate statements in both languages, are presented in Section C.

2. $\boxed{\text{M80:}}$ a macro processing language

a. Structure

M80 is a language intended for conventional macro processing (text replacement) along with arithmetic and logical expression evaluation.

An M80 program is any text containing 0 or more M80 statements, which are delimited by the bracketing symbols "[" and "]".

All M80 statements are free-format. They can be placed anywhere in the source text, and occupy any number of lines. In well formed M80 programs the brackets always appear in matching pairs, which can be nested in a way analogous to parenthesis nesting in arithmetic expressions.

All text outside brackets is meant to be simply reproduced without any modification, while the commands inside brackets are interpreted, triggering a macro action. A macro action always causes the corresponding statement (including its brackets) to be replaced by some string. In particular, the replacement string may be empty.

Since M80 is designed for both text and integer manipulations, its statements are centered around two basic types of entities: textual macros and integer macros.

$\boxed{\text{Integer macros}}$ behave as integer variables in most programming languages. In particular, they have numeric values which can be modified through assignment statements and used in expression evaluation and conditional statements. On the other hand, a textual macro is associated with a string, called its macro body. A textual macro can also have parameters.

Every macro has a name, which is some unique identifier (a sequence of any number of alphanumeric characters, starting with a letter - the $ sign is considered a letter in M80). The only restriction is that macro names must be different from the M80 reserved words, which are:

MACRO, INT, IF, THEN, ELSE, DEC, OCT, HEX, CHAR.

M80 adopts the same syntax as PL/M for strings, numbers and comments. For instance, 'ABCD' denotes the string ABCD, while 'AB''CD' represents the string AB'CD; 07F7H, 55Q, 121D, 000101B, 121 represent the numbers 07F7 hexadecimal, 55 octal, 121 decimal, 000101 binary and 121 decimal, respectively. The leading digit of a hexadecimal number must be a decimal digit, to avoid confusion with macro names. Thus, the hexadecimal number F3 is written 0F3H.

Comments are enclosed in /* */, and are ignored whenever they appear as part of an M80 statement.

M80 macros can be created, modified and invoked by means of the following statements:

1 - macro declaration;

2 - macro call;

3 - assignment statement;

4 - evaluation statement;

5 - conditional statement.

These statements are described in detail in the following paragraphs.

b. Macro declarations

Textual macros are created using statements in the following format:

```
[ MACRO <macro name> <formal param 1> <formal param 2> ...
  ... <formal param n> <macro body> ]
```

where <macro name>, <formal param i>, 1<=i<=n, n>=0, stand for unique identifiers, and <macro body> represents a string. The statement

```
[ MACRO C 'A=A++C' ]
```

defines a textual macro with name C, macro body A=A++C, and no formal parameters. Similarly,

```
[ MACRO A R1 R2 'A=A+[R1], +[R2], +[R3]' ]
```

creates a textual macro A, with formal parameters R1, R2, and a macro body equal to the string enclosed in the single quotes.

Integer macros are declared using statements of the form:

```
[ INT <macro name 1> <macro name 2> ... <macro name n> ]
```

where <macro name i>, 1<=i<=n, n>0, are unique identifiers. As an example,

```
[ INT X Y Z ]
```

defines three integer macros X, Y, Z. The values of integer macros are initialized to 0 upon declaration.

Both INT and MACRO statements evaluate to the empty string, and thus have only the side effect of creating new macros. For instance, the line

```
    CO[ MACRO P '125' ]ME TO TH[INT V W]E AID
```

evaluates to

```
    COME TO THE AID
```

although it causes the creation of textual macro P and integer macros V and W.

M80 macros are organized as a stack. That is, if more than one definition exists for a single macro, then only the most recent definition is effective. This fact is important in the evaluation of parameterized macro calls, as illustrated below.

c. Macro calls

After being defined, macros can be invoked by statements which have the following syntax:

```
    [ <macro name> <actual param 1> <actual param 2> ...
     ... <actual param n> ]
```

where <macro name> is the name of some integer or textual macro, and <actual param i>, 1<=i<=n, n>=0, are strings. It is important to notice that n=0 for integer macros, i. e. only textual macros can be called with actual parameters.

If the macro being called is an integer macro, then the macro call is replaced in the source text by the value of the macro, in decimal format and with suppression of leading zeros. If the value is negative, a minus sign is generated. For instance, assuming that Y is an integer macro with value 8, the line

```
    Y[Y] IS AN [Y]-BIT VARIABLE
```

will evaluate to

```
    Y8 IS AN 8-BIT VARIABLE
```

Similarly, if the value of integer macro X is -20, then

```
    A=A+[ X /* THIS IS A COMMENT */ ],-[X]
```

yields

```
    A=A+-20,--20
```

For textual macros two cases must be considered: assuming that a call to a textual macro contains m actual parameters, and the macro being invoked has n formal parameters, either m>n or m<=n. The former case is considered an error, while the latter has the following effect:

- m temporary textual macros are created, with macro names <formal param 1>, <formal param 2>, ..., <formal param m>, and macro bodies <actual param 1>, <actual param 2>, ..., <actual param m>, respectively;
- the macro call is replaced by the body of the macro being invoked; however, if macro calls are present in this macro body, they are themselves evaluated before the substitution takes place; the evaluation of macro calls embedded in a macro body is performed from left to right;
- after the replacement of the macro call is completed, all temporary macros are deleted.

Since the process described above is recursive, it may be useful to illustrate the evaluation of macro calls with some examples. Given the M80 program

```
    xxx[MACRO A B C '[B] IS THE [C]' ]yyy
    zzz[A 'NOW' 'TIME']www
```

execution proceeds as follows:

- the statement in the first line defines macro A, with formal parameters B and C; the statement itself evaluates to the empty string;

- upon recognition of the call to macro A in the second line, temporary macros B and C, with macro bodies NOW and TIME, respectively, are created;

- before the body of macro A is inserted in the text, it must be evaluated, since it contains calls to macros B and C;

- the evaluation of the call to macro B is performed according to the same steps described above; however, since no parameters are involved, and also because the body of macro B (the string NOW) does not contain other macro calls, the net effect is to replace [B] by NOW;

- the evaluation of [C] similarly yields the string TIME; therefore, the expanded version of the body of macro A is the string NOW IS THE TIME;

- the following output is generated:

```
    xxxyyy
    zzzNOW IS THE TIMEwww
```

- finally, the temporary definitions of macros B and C are deleted.

As another example, the program:

```
    xxx[MACRO B 'NOW' ][MACRO D '[B]']yyy
    zzz[MACRO A '[D] IS THE TIME' ][A]www
```

generates the same output as the previous one.

At this point it should be noticed that in MMG any macro calls can be present inside any strings. And every M80 statement behaves as a macro call. For instance,

```
    xxx[MACRO X 'ABC[MACRO M 'XYZ']DEF' ]yyy
```

defines a macro X such that its body contains another macro declaration, which can be thought of as a call to a predefined macro MACRO. Although XYZ is itself an individual string, no confusion arises from it being embedded in the body of macro X.

It should be observed, therefore, that brackets in M80 are also used to modify the scope of strings. For instance,

```
    'ABC[ MACRO M 'XY[MACRO N 'U''V'.]' ]DEF'
```

is a single string, in which two levels of embedding occur; the innermost macro declaration associates body U'V to macro N.

It is important to notice that formal parameters defined in a macro declaration can be activated only if they are invoked (directly or indirectly) in the body of their defining macro. A declaration like

```
    [MACRO X Y 'XYZ' ]
```

is syntactically correct, but the formal parameter Y will never be activated.

As previously mentioned, if a macro has more than one definition, then the most recent one is adopted when the macro is invoked. The consequences of this property are illustrated by the following example:

```
    [MACRO B 'S' ]B=[B]
```

```
[MACRO C '8' ]C=[C]

[MACRO A B C 'A=[B]+[C]+[C]+[B]' ][B]+[C]

[A '3' '4' ]

[A '7' ]

[A]
```

which evaluates to:

```
B=5

C=8

5+8

A=3+4+4+3

A=7+8+8+7

A=5+8+8+5
```

In the fourth line of the example above, macro A is called with two actual parameters; temporary definitions for macros B and C are established, so these are the most recent ones when the body of A is evaluated. After the processing of this call is completed, those definitions are deleted. This accounts for the results obtained for the calls in the last two lines. For the last line, the most recent definitions for macros B and C are those created in the beginning of the program (if these were not available, an error condition would arise). The net effect of this mechanism is the provision of default values for parameters.

Some additional examples of evaluation of macro calls are presented after the introduction of other features of M80.

d. Assignment statements

Assignment statements allow modification of the values of integer macros; they are of the form:

```
[ <macro name> := <expression> ]
```

where <macro name> is the name of some previously declared integer macro, and <expression> stands for any arithmetic or logical expression involving integer macros and/or constants.

NON-EMBEDDED

Assignment statements always evaluate to the empty string. For instance, the program

```
VA[ IN] Y ][ Y:=25+2*3 ]RIABLE[ Y ]=[Y:=Y+Y][Y]
```

generates the text

```
VARIABLE31=62
```

The following operators are available for expression evaluation:

```
*    /    %(mod)    (highest precedence)
+    -
=    <    >    <=    >=    <>
!(not)
&(and)
\(or)    \\(xor)    (lowest precedence)
```

The relational operators =, <, >, <=, >=, <> denote binary operations which yield the value 0 (false) or 255 (true); the logical operators !, &, \, \\ are applied on a bit by bit basis. The operator ! is unary; a unary minus operator (-) is also provided. As an example, the statement

```
[ Y := (-3>20) + 0FH&3*2 ]
```

assigns the value 6 to integer macro Y.

Whenever an integer macro is used in expression evaluation, an embedded assignment statement enclosed in parentheses (instead of brackets) can be used. Embedded assignments are treated as expressions; the value of an embedded assignment is the same as the value of the expression in its right hand side. For instance,

```
[ X := -(Y:=3) + (Z:=4) ]
```

assigns the values 1, 3, 4 to integer macros X, Y, Z, respectively.

Parenthesis can also be used freely to alter the precedence of arithmetic or logical operators, as in conventional algebraic notation.

Assignment statements, as any other bracketed M80 statements, can be thought of as macro calls. Therefore, if they are embedded in the body of a textual macro, they are executed every time the textual macro is invoked.

W/ BRACKETS IN THIS CASE!

For instance, the M80 program

```
[INT X]xxx
[MACRO C '[X][X:=X+1],' ]yyy
[C][C][C][C]zzz
```

generates the following output:

```
xxx
yyy
0,1,2,3,zzz
```

The same output is generated by:

```
[INT X]x[MACRO INCX '[X:=X+1]' ]xx
[MACRO C '[X][INCX],' ]yyy
[C][C][X][INCX][C]zzz
```

e. Evaluation statements

Evaluation statements allow generation of numer
ic values in different formats, as well as special charac-
ters, including the brackets themselves (brackets insid
strings always flag the presence of macro calls; therefore
special mechanism is needed for brackets which are to b
treated as text).

Evaluation statements have the following syntax:

```
[ <format> <expression> ]
```

where <format> is any of the reserved words DEC, OCT, HEX,
CHAR, and <expression> is defined as for assignment state-
ments.

The effect of DEC, OCT, HEX is to cause the re-
placement of the evaluation statement by the value of <ex-
pression>, in decimal, octal or hexadecimal format, respec-
tively. For instance,

```
AB[ DEC 15+8 ],[HEX -1],[INT X][OCT (X:=0FH)+1],CD[X]
```

yields

```
AB23,0FFFFH,000020Q,CD15
```

When CHAR is used, the evaluation statement i
replaced by the character whose bit pattern is equal to th
least significant byte of the value of <expression>. Fo
instance, assuming that ASCII encoding is used in the imple-
mentation of M80, the program

```
     xxx[MACRO B '[CHAR 5BH]' ]yyy

     zzz[MACRO E '[CHAR 5DH]' ]www

     uuu[B]MACRO X 'Y'[E]vvv
```

yields

```
     xxxyyy

     zzzwww

     uuu[MACRO X 'Y']vvv
```

Also assuming the ASCII code, the effect of

```
     [MACRO LF '[CHAR 0AH /* LINE FEED */]' ][LF][LF][LF]
```

is to generate three line-feed characters.

As any other M80 statements, evaluation statements can be embedded in any macro bodies or actual parameters. Unlike macro declarations and assignment statements, they do not evaluate to the empty string (except when [CHAR n], where n does not correspond to a printable character, is issued; the result in this case is uncertain).

f. Conditional statements

Conditional statements are used to create different execution paths in an M80 program; they also provide the ability to define recursive macros, i. e. macros which invoke themselves.

Conditional statements have the following syntax:

```
     [ IF <expression> THEN <string 1> ]

     or

     [ IF <expression> THEN <string 1> ELSE <string 2> ]
```

where <expression> is defined as for assignement statements, and <string 1>, <string 2> are strings enclosed in single quotes, and which may contain any number of well formed M80 statements (including conditional statements).

Conditional statements cause the <expression> to be evaluated. If the least significant bit of the result is is 1, then <string 1> replaces the conditional statement, otherwise <string 2> (or the empty string, in the case of the IF-THEN construct) is used as the replacement string. Of course, if macro calls (M80 statements) are embedded in the replacement string, they are evaluated before the replacement is performed, as for regular macro bodies. For instance,

```
    [IF 4>3 THEN 'UN' ELSE 'BIN']ARY OPERATOR
```

yields the string

```
    UNARY OPERATOR
```

while

```
    [INT X][IF X=0 THEN '**[DEC (X:=5)][X]**' ELSE '27']99
```

results in

```
    **55**99
```

Use of conditional statements in the programming of recursive textual macros is illustrated by the following example: suppose one wishes to create n groups of asterisks, such that the i-th group contains i asterisks, and terminates with the number 200+i. A possible solution would be:

```
    [INT N G ]]
    [MACRO STAR '[ I := I-1 ][IF I>0 THEN '*[STAR]']']
```

```
[MACRO GROUP '[IF (G:=G+1)<=N THEN
 '[I:=G+1][STAR] [DEC 200+G] [GROUP]' ]' ]
```

Then the output corresponding to the calls

```
[N:=5][GROUP]
[G:=0][N:=6][GROUP]
```

is:

```
* 201 ** 202 *** 203 **** 204 ***** 205
* 201 ** 202 *** 203 **** 204 ***** 205 ****** 206
```

Conditional statements can also be used for generating tailored programs from a general package which incorporates different versions of routines. Selected portions of code can be excluded from, or included into, the final product, without the need for extensive editing in the original package (usually only a small number of integer macros, used as switches, have to be set). This technique is known as conditional compilation.

The above description summarizes most of the important aspects of the M80 language. Additional details are provided in the last sections of this chapter.

3. L80: a machine oriented language

a. Structure

According to its design objectives, L80 is a language which attempts to bring adequate programming tools, characteristic of higher level languages, to the environment of machine-oriented programming.

The structure of L80 is strongly influenced by both the PL/M language and the architecture of the 8080 microprocessor. The influence of PL/M is easily recognized in the control constructs and data definition statements of L80; the dependence on the 8080 characteristics occurs mostly at the data manipulation level.

One of the original goals of L80 was to allow modular programming through the generation of relocatable modules which can be retrieved and linked into a single program. Some of the constructs in the language reflect this idea.

An L80 program is free-format, and consists of a sequence of statements separated by semicolons, and terminated by the word EOF. Both upper and lower case characters are recognized in L80, and the dollar-sign ($) is considered an alphabetic character. Except for these conventions, the syntax of identifiers, numbers, strings and comments is the same as in PL/M.

Identifiers can be defined by the programmer and used as symbolic names for variables, constants, and addresses of code segments. They are made up of any number of alphanumeric characters, beginning with a letter. Numeric constants can be written in decimal, hexadecimal, octal or binary formats; as an example, the number 16 can be denoted by 16D or 16, 10H, 20Q or 20O, 010000B, respectively.

Strings are enclosed in single quotes. Quotes which are to be interpreted as part of a string are denoted by two contiguous single quotes: 'XY''Z', for instance, represents the string XY'Z. Comments are enclosed in /* */, and may appear between any two basic tokens.

In addition to the symbol EOF, some other upper case identifiers are considered reserved words in L80; a complete list of these words is presented in Figure 1. Certain special characters are used as operators or statement delimiters; they are summarized in Figure 2.

The statements available in L80 can be divided into the following categories:

- storage declarations;
- assignment statements;
- groups;
- conditional statements;
- case statements;
- iterative statements;
- procedure declarations;
- procedure calls;
- label declarations;
- external declarations;
- control statements.

Each one of these categories is described in the following paragraphs.

b. Storage declarations

By means of storage declarations the programmer can set aside memory locations for future use in a program. For instance, the statement

```
DECLARE X(5) BYTE
```

defines a variable X, occupying a total of 5 bytes.

A	DE	H	OUT
B	DECLARE	HALT	PLUS
BC	DISABLE	HL	PROCEDURE
BY	DO	IF	PSW
BYTE	E	IN	PY
C	ELSE	INITIAL	REPEAT
CALL	ENABLE	L	RETURN
CASE	END	LABEL	SP
COMMON	EOF	M	STACK
CY	EVEN	MINUS	THEN
D	EXTERNAL	NOP	UNTIL
DATA	GOTO	ODD	WHILE
			ZERO

Figure 1. L80 reserved words.

+	(add)	<	(ral)	=	(assign)
++	(adc)	<<	(rlc)	==	(exchange)
-	(sub)	>	(rar)	:	(label def)
--	(sbb)	>>	(rrc)	::	(compare)
&	(and)	!	(not)	;	(separator)
\	(or)	#	(daa)	,	(separator)
\\	(xor)	.	(address)	()	

Figure 2. L80 special symbols.

Similarly,

```
    DECLARE (X,Y) BYTE
```

creates two variables X and Y, each one 1 byte long. Initial values can be assigned to variables, as in

```
    DECLARE X BYTE INITIAL(25), Y(3) BYTE INITIAL('ABC')
```

The above declarations are used to request memory locations in read/write memory (RAM); assignment of storage located in read-only memory (ROM) can be imposed by the programmer with a statement such as

```
    DECLARE X DATA('ABCD',0)
```

In this example, X is defined as a constant (since it is to be located in ROM), 5 bytes long, and initialized to the values enclosed in parenthesis. The length of the constant is implicit in the declaration. Storage allocated as DATA is usually interspersed with the code (executable instructions) in the object program, while BYTE storage is allocated outside the code region.

Properly declared variables and constants can be referenced by their symbolic names. For instance, after the declaration

```
    DECLARE (X,Y)(3) BYTE INITIAL('ABCD')
```

the contents of X and Y can be referenced in the following ways:

X or X(0): the contents of memory starting at the first byte assigned to X;

X(1): the contents of memory starting at the second byte assigned to X;

X(3) or Y: the contents of memory starting at the
first byte assigned to Y.

It is important to notice that the length of the referenced location depends on the operation being performed. Additional information on this aspect is provided in the next section.

A special operator, dot (.), is used to reference the address, rather than the contents, of memory locations. In the above example,

.X yields the address of the first byte allocated to X,

.Y(2) gives the address of the third byte of Y, and so on.

Since addresses are constants in L80 (their value being determined at load time), the following program segment is correct:

```
    DECLARE X(3) BYTE;
    DECLARE XA DATA(.X, .X(1), .X(2), .X(3));
     ....
```

Here XA is created as an array of pointers to the 3 elements of X and to the first byte after X. Since addresses in the 8080 are 16 bits long, XA occupies 8 bytes.

c. Assignment statements

Assignment statements are used to alter the contents of registers and memory locations; due to the fact that they define operations at the register level, assignment statements reflect most of the peculiarities of the 8080 architecture.

One of the guidelines of L80 is that modifications of the contents of registers are always explicit in the source program, i. e. the code emitted by the compiler should not destroy the contents of any registers, even when behind-the-scenes operations are performed.

An obvious advantage of this feature is that the source program actually reflects the sequence of states assumed by the machine. Another consequence is related to the structure of L80 assignment statements: since it would be inefficient to save and restore registers for every assignment operation, register allocation is left under programmer control.

L80 attempts to provide a convenient notation for register operations, by means of the following conventions:

- eight 1-byte registers (A, B, C, D, E, H, L, M) are available in the 8080 microprocessor; they are denoted in L80 by A, B, C, D, E, H, L, M(HL), respectively;
- register pairs B and C, D and E, H and L can work as individual 2-byte registers; these register pairs are represented in L80 by BC, DE, HL, respectively;
- operations provided by the 8080 CPU can be classified as binary and unary; binary operations are denoted in L80 by the operators:

    + (add)

    - (subtract)

    ++ (add with carry)

```
     -- (subtract with borrow)
     &  (and)
     \  (or)
     \\ (xor)
```

while the unary operators are:

```
     !  (not)
     <  (rotate left through carry)
     >  (rotate right through carry)
     << (rotate left into carry)
     >> (rotate right into carry)
     #  (decimal adjust)
```

Some examples of register operations in L80 can now be given:

```
     B = B + 1;        /* increment register B */
     A = C + B;        /* move C to A, add B to A */
     A = A + M(HL);    /* add the contents of memory location
                          pointed to by HL into A */
     A = A ++ 3;       /* add 3 plus the carry bit to A */
     A = A & 0FH;      /* and A with hexadecimal constant 0F */
```

The above program segment could also be written as:

```
     A=C+(B=B+1),+M(HL),++3,&0FH;
```

As can be concluded from the example above, commas are used in L80 to factor out the left hand side of assignments to the same register, and also imply that execution proceeds from left to right, with no operator precedence involved.

Parenthesized assignment statements can be used whenever a register is allowed in the right hand side of assignment statements; they also serve to force the sequence of execution, as illustrated above.

The commas in the above construct are also useful to prevent misconceptions about the results of operations. For instance,

```
    A = 1;
    A = A + A, + A;
```

leaves register A with contents 4, since it corresponds to

```
    A = 1;
    A = A + A;  /* A now is 2 */
    A = A + A;  /* A now is 4 */
```

while

```
    A = 1;
    A = A + A + A;  (this is syntactically incorrect in L80)
```

might give the impression that A gets 3 as a final result.

The use of unary operators is shown in the examples below:

```
    A = !C;          /* move C to A, complement A */
    A = <<B;         /* move B to A, rotate A left into the
                        carry bit */
    A = >(B=B+1);    /* increment B, move B to A, rotate
                        A right through carry */
    B = (A=>>B);     /* move B to A, rotate A right into carry,
                        move A to B */
```

Assignment statements can involve user declared memory locations. For instance,

```
    DECLARE (X,Y) BYTE;
    X = (A=Y);
```

causes the same effect as X=Y in a higher level language; however, here it is clear that register A was used as an intermediary, and presently holds the value of variables X and Y. This value can now be used, if necessary, without requiring an extra load operation. The statements

```
    DECLARE Y(2) BYTE;
    Y = (HL=Y+1);
```

cause the value of Y (2 bytes long) to be loaded into the HL register pair, incremented, and deposited back into the cells assigned to Y. Register HL remains with the new value of Y. On the other hand,

```
    DECLARE Y(2) BYTE;
    Y = (A=Y+1);
```

works as the previous case, except that only the first byte of Y is involved. Therefore, the number of bytes affected depends on the operation performed.

It is important to notice that the limitations of the 8080 architecture are reflected in the semantics of L80. For instance, the following statements, although syntactically correct, are semantically incorrect, since they cannot be executed by the machine:

```
    B = B + 2;          /* a valid alternative: B=B+1,+1             */
    HL = HL - 2000; /* a valid alternative: HL=HL+(BC=-2000)  */
    C = X + 2;          /* valid alternatives:  C=(A=X)+1,+1
                           or  HL=.X; C=M(HL)+1,+1                    */
```

In addition to the registers used so far, L80 provides reserved words to designate the following elements in the 8080 microprocessor:

IN(<number>)	- input port designated by <number>
OUT(<number>)	- output port designated by <number>
STACK	- the topmost two bytes of the machine stack
SP	- stack pointer register
CY	- carry flag bit
PSW	- program status word

The above listed elements can be used in assignment statements, as the following examples demonstrate:

```
SP = HL - 1;          /* move HL to SP, decrement SP */
STACK = HL;           /* push HL into the stack */
HL = STACK+BC,+5;     /* pop HL, add BC to HL, add 5 to HL */
A = IN(1) \ 0FH;      /* read input port 1 into A, or A
                         with 0F hexadecimal */
OUT(4) = (A=X);       /* load A  with  the  contents  of  X,
                         output A into port 4 */
CY = 0;               /* reset the carry bit */
```

L80 provides a limited indexing capability: variables indexed by numeric constants can be used wherever a non-indexed variable is allowed; dynamic indexing can be achieved only by using register pairs BC, DE, HL as index registers. The symbols M(BC), M(DE), M(HL) are adopted in this case. For instance, the following statements are correct:

```
X(2) = (A=Y(3));     /* A and X(2) get the value of Y(3) */
```

```
    BC = .X(28);          /* load address of X(28) into BC */
    A = M(BC) + 15;       /* load into A the contents of memory
                             cell indexed by BC, add 15 to A */
    M(DE) = (A=C);        /* store the contents of C into memory
                             location pointed to by DE */
```

The 8080 CPU is able to exchange the contents of some registers in a single machine operation. This feature gives rise to a special operator in L80, the == (exchange) operator. Statements involving the exchange operator are special cases of assignment statements, and can be illustrated by the following examples:

```
    HL == DE              /* exchange the contents of the
                             HL and DE registers */
    HL == STACK           /* exchange the contents of HL
                             with the topmost stack element */
    HL == (STACK=BC)      /* push BC, then exchange HL with
                             the topmost stack element */
    HL == (DE=BC+1,+1) /* move BC to DE, add 2 to DE, then
                             exchange the contents of HL, DE */
```

Additional examples of L80 assignment statements are presented in the next sections, in conjunction with other L80 constructs.

d. Groups

Groups are sequences of statements which behave as programs within a program. One of their properties is to establish scope for variables and data: an identifier declared in a group is defined only within that group, and cannot be referenced by statements located outside the group.

The reserved words DO and END are used to delimit a group. For instance, the program

```
     DO;
         DECLARE X BYTE;
         DO;
             DECLARE Y BYTE;
             <statement 1>;
         END;
         <statement 2>;
     END;
     EOF
```

contains two groups; since the variable Y is declared in the innermost group, it cannot be referenced by <statement 2>, which is outside that group; <statement 1>, however, can reference both X and Y. As far as the innermost group is concerned, Y and X are said to be local and global variables, respectively.

Another property of groups is that they are considered single statements. Whenever an isolated statement is expected, a group may be used. One application of this property is related to conditional statements, which are introduced below.

e. Conditional statements

Conditional statements can be used to select different execution paths in a program. They are of the forms

```
     IF <condition> THEN <statement>

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

where <statement> stands for any L80 statement, and <basic statement> is any <statement> which is not a conditional statement. The reason for such restriction is to prevent ambiguities which arise in statements such as

```
     IF <cond 1> THEN IF <cond 2> THEN <st 1> ELSE <st 2>
```

which has two different interpretations, depending upon IF-ELSE matching conventions. Under the syntax of L80, the above construct is always interpreted as

```
     IF <cond 1> THEN
        DO;
            IF <cond 2> THEN <st 1> ELSE <st 2>;
        END
```

The alternative interpretation must be explicitly imposed as follows:

```
     IF <cond 1> THEN
        DO;
            IF <cond 2> THEN <st 1>;
        END
     ELSE <st 2>
```

It should be noted that the word ELSE is never preceded by a semicolon, since it is not the beginning of a new statement, but rather the continuation of the IF-THEN part.

The <condition>, which always appears after the word IF, is not a logical expression as commonly used in higher level languages, but rather a machine-dependent Boolean entity. In the 8080 CPU, four bits are used as flags which are set according to the result of some register operations. These flags are called: Carry, Sign, Zero, and Parity; their status can be sensed in L80 programs by means of conditional statements.

In the simplest case, the above mentioned <condition> is simply one of the possible flag status:

ZERO	(Zero flag on)
! ZERO	(Zero flag off)
CY	(Carry flag on)
! CY	(Carry flag off)
MINUS	(Sign flag on)
PLUS	(Sign flag off)
PY EVEN	(Parity flag on)
PY ODD	(Parity flag off)

As an example, the conditional statement

```
    IF !CY THEN A=A+1 ELSE A=A-1
```

causes A to be incremented if the Carry flag is 0, and decremented if it is 1.

Usually the flag bits have to be set by some specific operation in order to assure that correct results are obtained. For instance, to check if the contents of register A is 0, the programmer might perform a logical OR operation on register A with itself, which causes all flags to be set, and then the Zero flag can be tested. This may be accomplished by writing:

```
    A = A \ A;
    IF ZERO THEN ...
```

however, a more convenient notation is available in L80:

```
    IF (A=A\A) ZERO THEN ...
```

Constructs as the one above improve program readability; they show clearly what operations are used to set the flags for condition testing. Any number of statements can be enclosed in the parenthesis, as in:

```
    IF (A=X; A=A+B; A=A\A) ZERO THEN ...
```

All the properties of assignment statements hold, so the same statement could be rewritten as:.

```
    IF (A=X+B,\A) ZERO THEN ...
```

In many occasions program logic requires the testing of more than a single condition in order to decide upon the correct execution path. L80 allows conjunction or disjunction of conditions by means of the & (and) and \ (or) operators. For instance:

```
    IF (A=C-3) ZERO & (A=D-4) ZERO THEN HL=HL+1
```

causes HL to be incremented only if the contents of register C is 3 and the contents of D is 4. The statement

```
    IF (A=C-3) ZERO \ (A=D-3) ZERO \ (A=E-3) ZERO THEN B=B+1
```

increments register B if at least one of the registers C, D, E contains a 3.

The evaluation of compound conditions as the ones above is always performed from left to right. This implies that for a disjunctive compound condition the THEN part gets control as soon as an isolated condition evaluates to true, while for conjunctive compound conditions the statement after the ELSE is executed as soon as a false condition is sensed.

A restriction on compound conditions is that the & and \ operators cannot be intermixed at the same level; a compound condition must be either purely conjunctive or purely disjunctive. This eliminates questions about the relative precedence of the & and \ operators, and in fact does not introduce excessive restrictions, as can be concluded from the following example:

```
    IF (IF (A=C-3)ZERO & (A=D-3)ZERO THEN CY=1 ELSE CY=0) CY
          \ (A=E-3)ZERO THEN HL=HL+1
```

In this case HL is incremented either if E contains 3 or both C and D contain 3.

The 8080 provides an accumulator comparison operation which can be used in conditional statements; this operation is denoted in L80 by the operator :: (compare), and gives rise to statements such as:

```
    A::B            /* compute A-B, without disturbing either
                       register, set flags according to result */
    A::(B=C+1,+1)   /* compute B=C+2, compare A and B */
```

Comparison statements can be incorporated into conditional statements, as in

```
    IF (A=B; A::3) PLUS & (A::20) MINUS THEN HL=HL+1
```

which causes HL to be incremented only if the contents of register B is less than 20 and greater than or equal to 3.

Since comparison statements are not assignments, they cannot be factored by the comma mechanism. For instance,

```
    A = B+C,::D
```

is not correct, since it expands to

```
    A = B; A = A + C; A = A::D
```

and A::D is not a value but rather denotes the action of setting the machine flags.

f. $\boxed{\text{Case statements}}$

Case statements are a specialization of conditional statements; they are of the form

```
    DO CASE <register>;
        <statement 0>;
        <statement 1>;
        ...
        <statement n>;
    END
```

Case statements, as groups, are considered single statements. However, they act as multi-path switches in the following way: if the contents of <register> is the value i when the case statement is entered at run time, then only <statement i> is executed (if either i>n or i<0 then the result cannot be predicted).

To have more than a single statement executed, one can make use of the group construct, as in

```
    DO CASE HL;
        DO;  /* case 0 */
        A = A+1;
        B = B+1;
     END; /* of case 0 */
     DO;  /* case 1 */
        C = C+1;
        D = D+1;
     END; /* of case 1 */
     ...
  END
```

Case statements are in general more readable than an equivalent sequence of IF-THEN-ELSE statements.

g. $\boxed{\text{Iterative statements}}$

As their name implies, iterative statements are useful in repetitive operations. These statements are available in L80 in two different forms:

```
DO <assignment statement 1> BY <assignment statement 2>
WHILE <condition> ; <statement list> ; END
```

or

```
REPEAT; <statement list> ; UNTIL <condition>
```

where <statement list> is a sequence of statements separated by semicolons, and <condition> is any simple or compound condition as described for conditional statements.

The effect of the DO-BY-WHILE construct is the following:

- <assignment statement 1> is executed once; it is usually an initialization statement;
- <condition> is tested; if it is true then <statement list>

is executed, otherwise control is transferred to the statement after END;

- after each execution of <statement list>, <assignment statement 2> is performed (usually this statement increments or decrements a loop counter) and then the previous step (condition testing) is applied.

As an example, the statement

```
    DO B=1 BY B=B+1 WHILE (A=B-31)!ZERO;
        <statement list>;
    END
```

will cause the execution of <statement list> 30 times (as long as the contents of B is not altered inside the loop).

Some variations are possible in the above pattern; for instance, the increment part can be omitted, as in

```
    DO B=1 WHILE (A=B-31)!ZERO & (A=C-15)PLUS;
        <statement list>;
    END
```

If both the initialization and the increment are omitted, then the statement becomes a simple DO-WHILE statement. Another possibility is illustrated by

```
    DO BY A=A+5 WHILE (A::31)MINUS;
        <statement list>;
    END
```

This is useful if the initialization has already been obtained as a consequence of some other statement.

The second kind of iterative statement, the REPEAT-UNTIL construct, also behaves as a single statement;

however it is not considered a group, and consequently does not create additional scope for identifiers. Another major difference between REPEAT-UNTIL and DO-WHILE is that the former always executes <statement list> at least once, while the latter is able to execute no iterations at all. The reason for this is that <condition> is tested at the beginning of a WHILE loop, and at the end of REPEAT loops.

Compound conditions can be used in REPEAT statements, as in

```
    REPEAT;
        <statement list>;
    UNTIL (A=A\\A;A::L)ZERO & (A::H)ZERO
```

In this example, the loop is repeated until the contents of register pair HL is 0.

h. Procedure declarations

A good programming technique, applicable in many cases, is to organize a program as a hierarchy of smaller units which perform specific tasks, and which can be coded and tested independently.

Procedures are a useful tool in the programming of such units. A procedure is a section of code which implements a certain function. It is usually invoked by other procedures at a logically higher level.

Procedures are most useful when they are able to handle parameters. Several techniques for parameter passing, such as call by value, call by address, call by name, etc. are adopted in high level languages. In a language which allows full control of the machine, such as L80, it is natural and efficient to pass parameters in registers. L80 obviously supports this kind of procedure, but also provides parameterized procedures, using the call by value modality of parameter passing.

Procedures, as groups, define their own scope for variables and data. Every procedure has a name, which can be any legal identifier. For instance:

```
    HEX: PROCEDURE;
            /* convert the contents of A (assumed in the
            range 0-15) to a printable hex character */
            IF (A::10) MINUS THEN A=A+'0'
            ELSE A=A-10,+'A';
            END
```

defines a procedure with name HEX, and no explicit parameters (the parameter in this case is passed in register A, as the comments reveal).

A procedure with parameters might look like:

```
    MOVBUF: PROCEDURE(SOURCE,DEST);
                /* move 128 bytes from source to dest */
                BC = (HL=SOURCE); ?
                HL = DEST;
                D = 128;
                REPEAT;
                    M(HL) = (A=M(BC));
                    HL=HL+1; BC=BC+1;
                UNTIL (D=D-1) ZERO;
```

```
        END MOVBUF;

        ....
```

Some points have to be clarified about this example. Parameters do not have to be declared in the procedure - they are implicitly considered local variables, and are assigned two bytes each. The repetition of the name of the procedure after the word END is not mandatory, but simply a convenience to the user to improve program readability. This feature applies to groups and iterative statements as well: the word END can always be followed by an user defined identifier, which is treated as a comment.

A procedure can contain any statements, including the definition of other procedures. Control is never transferred to a procedure as a mere consequence of sequential execution of statements. For instance, in the code segment:

```
     A = A+1;
 XX: PROCEDURE;
     C = C+1;
     ...
     END XX;
     B = B+1;
     ...
```

the next executable statement after A=A+1 is B=B+1; the procedure declaration is skipped automatically. Execution of procedures is only through call statements, described in the next paragraphs.

i. Procedure calls

After being defined, procedures can be invoked by means of call statements. Referring to the previous examples, the following statements could be issued:

```
A=5; CALL HEX;
 ...
DECLARE X(128) BYTE;
CALL MOVBUF(080H,.X);
 ...
DECLARE Z(256) BYTE;
CALL MOVBUF(.Z(0),.Z(128));
 ...
```

As mentioned previously, L80 constructs do not destroy the contents of registers without an explicit endorsement by the programmer. This feature is somewhat hard to reconcile with parameterized procedure calls, since parameter passing always involves some register manipulations. To avoid excessive save-restore overhead, L80 actual parameters are restricted to constants. This restriction is easily overcome by noting that the address of a variable is always a constant in L80 (since no dynamic allocation of storage at run time is provided). Therefore, if the address of a variable is used as an actual parameter, the called procedure gets all the information necessary to work with the variable itself. Thus, L80 procedures can function in a call by address mode. This is the reason why formal parameters are always assigned two bytes: they are assumed to hold addresses, which are 16 bits long in the 8080.

Procedure calls can be nested, which is considerably facilitated by the stack available in the 8080. The depth of procedure call nesting is limited only by the amount of memory allocated to the stack. This allocation is not automatically performed in L80, but rather it is left to the programmer.

A procedure cannot be called before it is defined, and recursive calls are not directly supported. A call to any absolute memory location is permitted, however, as in

```
    CALL 3FFDH
```

Parameter passing is not allowed in this case.

A peculiarity of the 8080 is the provision of restart instructions which are 1-byte calls to absolute memory addresses 0, 8, 16, 24, ..., 56. These instructions can be invoked in L80 in the obvious way:

```
    CALL 0;
     ...
    CALL 8;
     ...
    CALL 56;
     ...
```

The 8080 instruction set also contains special conditional call instructions, which are represented in L80 by statements of the form:

```
    IF <simple condition> CALL ...
```

These conditional statements have a slightly different syntax, since the word THEN is omitted. Compound conditions are not permitted in this class of statements, referred to as conditional calls. The following are examples of conditional calls:

```
     IF (A::(B=B+1)) ZERO CALL HEX;
      ...
     IF (D=E+5) PLUS CALL MOVBUF(.X,080H);
      ...
```

The same results would be obtained with conditional statements (IF-THEN) but with slightly less efficiency.

When a call statement is executed, control is transferred to the called procedure, and the return address is saved in the machine stack. The instruction addressed by the topmost stack element gets control when a RETURN statement is executed.

Procedures may contain any number of return statements, as dictated by program logic. A return statement is always provided by L80 at the end of each procedure.

Analogous to conditional call instructions, the 8080 provides conditional returns. In L80, these are denoted by:

```
     IF <simple condition> RETURN
```

as in

```
     IF (A=B-3) ZERO RETURN;
     IF (A=(C=C+1)-5) PLUS RETURN;
      ...
```

Compound conditions are not allowed in this class of statements.

j. Label declarations

Although the L80 control constructs presented so far are sufficient for most programming needs, there may be occasions in which goto's are required. Both symbolic and absolute addresses can be used in L80 in connection with goto's. Symbolic addresses are established by means of labels, which can be any user defined identifiers. For instance, in the program segment:

```
    LOOP: A = A+1;

          ...

          GOTO LOOP;

          ...
```

WHERE 'DECLARE LOOP LABEL' NOT NEEDED

the label LOOP is associated with the address of the statement A=A+1.

<IF

If a label is to be referenced before it actually appears in the source program, it must be declared, as in:

```
          DECLARE LOOP LABEL;

          GOTO LOOP;

          ...

    LOOP: A = A+1;

          ...
```

Labels, as any other identifiers, are subject to scope rules. Therefore, in the program

```
    DO;
```

```
          DECLARE (LAB1,LAB2) LABEL;
          GOTO LAB1;
    LAB3: DO;
             LAB1: C = C+1;
             LAB3: GOTO LAB2;
          END;
    LAB1: A = A+1;
          GOTO LAB3;
    LAB2: B = B+1;
END;
EOF
```

the sequence of execution is: A=A+1, C=C+1, B=B+1.

More than one one label can be defined for a single address. The actual address of a label can be obtained by the dot operator. Both features are illustrated by:

```
 ...
 LAB2: LAB3: HL = .LAB2;
 ...
 LAB4: LAB5: BC = .LAB2(3);
 ...
```

The first statement loads the address of LAB2 into register HL, while the last statement loads the address of LAB2 plus 3 into BC.

Numeric labels are also allowed in L80; they force code to be generated at a relative address equal to the value of the number. For instance,

```
200: A = A+1
```

specifies that the code emitted for A=A+1 and its subsequent statements begin at address 200, relative to the origin of the program. The reason for this address to be relative is that L80 programs are relocatable (however, when numeric values are used in GOTO or CALL statements, they refer to absolute memory locations).

As for calls and returns, conditional jump in-structions are available on the 8080. In L80 they take the form:

```
IF <simple condition> GOTO ...
```

and can involve labels or absolute addresses, as in

```
IF (A::3) PLUS GOTO 3FFDH
```

Labels can also be used in conjunction with call statements. However, this is not advisable, unless return paths are provided by the programmer. Thus, the following is permitted:

```
LOOP: A = A+1;
        ...
        CALL LOOP;
        ...
        GOTO LOOP;
        ...
        GOTO LOOP(3);
        ...
        CALL LOOP(5);
        ...
```

```
HL = .LOOP(6);
GOTO M(HL);
...
```

The last statement corresponds to another special feature of the 8080: control is transferred to the instruction located at memory location addressed by the HL register. PCHL

k. $\boxed{\text{External declarations}}$

The declarations examined so far apply to variables, data and labels which belong to the current program. It is also possible for L80 programs to contain references to storage allocated to modules generated in independent compilations, causing retrieval of the referenced modules and their incorporation into the current program at load time.

At this point it should be mentioned that the product of each L80 compilation is an independent module, made up of three segments (any of which may be empty):

- a code area, containing executable code, and constants (declared as DATA) to be located in ROM;
- an initialized data area, which contains program constants, such as strings, and variables to which initial values were assigned in the source program;
- a work area, comprising storage assigned to non-initialized variables.

When several modules are linked into a single executable object program, all code areas are concatenated, followed by the initialized data areas, and all the work areas. As a consequence of this scheme, control could be passed from one module to the next by mere sequential execution. This method, however, is not recommended. A better technique is to transfer control from module to module by means of calls or goto's. For instance, in the program segment

```
    DECLARE (X,Y) EXTERNAL;
     ...
    CALL X(.Y,2900H);
     ...
    GOTO Y;
     ...
```

X and Y are names of external modules, and are referenced in two different ways. The expression .Y gives the address of the first instruction in the code area of module Y, and is used in the example as an argument for procedure X.

It is also possible to reference the initialized data area of an external module, which allows intermodule data sharing. As an example,

```
    DECLARE Z COMMON, W EXTERNAL;
     ...
    CALL W(.Z(3));
     ...
```

would pass to external procedure W the address of the fourth byte in the initialized data area of module Z.

It must be mentioned that a module name cannot be declared as EXTERNAL and COMMON at the same time, because then it is not clear to which segment external references apply. Nevertheless, given the block structure of L80, the problem is overcome as in the following example:

```
    DO;
        DECLARE X EXTERNAL;
        ...
        CALL X;
        ...
        DO;
            DECLARE X COMMON;
            HL = .X(4);
            ...
        END;
        ...
    END
```

In this example, both the code area and the data area of external module X are accessed.

1. Control statements

The 8080 provides some special instructions which can be classified as machine control instructions. These operations are denoted in L80 by the following reserved words:

```
    HALT            /* stop the CPU */
    NOP             /* no operation */
    ENABLE          /* enable  interrupts */
    DISABLE         /* disable interrupts */
```

These words can be used as regular L80 statements.

B. THE SYNTAX OF ML80

Both M80 and L80, the macro oriented and machine oriented components of the ML80 language, are defined by recursive grammars. These grammars are presented in the following sections, using BNF notation.

1. M80 grammar

In this section, the symbols <identifier>, <number>, <string> are considered terminal grammar symbols. The syntax for these symbols is informally described in III.A.2. The syntax of the M80 language is defined as follows:

```
<program> ::= [ <statement> ]
            ¦ <program> [ <statement> ]
<statement> ::= <integer.declaration>
              ¦ <assignment.statement>
              ¦ <evaluation.statement>
              ¦ <macro.declaration>
              ¦ <macro.call>
              ¦ <if.statement>
<integer.declaration> ::= INT <identifier>
                        ¦ <integer.declaration> <identifier>
<assignment.statement> ::= <identifier> := <expression>
<evaluation.statement> ::= <format> <expression>
<format> ::= DEC
           ¦ OCT
           ¦ HEX
           ¦ CHAR
<macro.declaration> ::= <macro.decl.head> <string>
```

```
<macro.decl.head> ::= MACRO <identifier>

                    ¦ <macro.decl.head> <identifier>

<macro.call> ::= <macro.call.head>

<macro.call.head> ::= <identifier>

                    ¦ <macro.call.head> <string>

<if.statement> ::= IF <expression> THEN <string>

                 ¦ IF <expression> THEN <string> ELSE <string>

<expression> ::= <logical.factor>

               ¦ <expression> \ <logical.factor>

               ¦ <expression> \\ <logical.factor>

<logical.factor> ::= <logical.secondary>

                   ¦ <logical.factor> & <logical.secondary>

<logical.secondary> ::= <logical.primary>

                      ¦ ! <logical.primary>

<logical.primary> ::= <arith.expr>

                    ¦ <arith.expr> <relation> <arith.expr>

<relation> ::=  =  ¦  <  ¦  >  ¦  <=  ¦  >=  ¦  <>

<arith.expr> ::= <term>

               ¦ <arith.expr> + <term>

               ¦ <arith.expr> - <term>

<term> ::= <primary>

         ¦ <term> * <primary>

         ¦ <term> / <primary>

         ¦ <term> % <primary>

<primary> ::= <identifier>

            ¦ <number>

            ¦ - <number>
```

```
        ¦ ( <expression> )

        ¦ ( <assignment.statement> )
```

2. L80 grammar

In this section, the symbols <identifier>, <number>, <string>, <empty> are considered terminal symbols. The syntax of the first three is informally described in III.A.3; the symbol <empty> represents the empty string. The syntax of the L80 language is defined as follows:

```
<program> ::= <statement.list> ; EOF

<statement.list> ::= <statement>

                   ¦ <statement.list> ; <statement>

<statement> ::= <basic.statement>

              ¦ <if.statement>

<basic.statement> ::= <decl.statement>

                    ¦ <group>

                    ¦ <procedure.definition>

                    ¦ <return.statement>

                    ¦ <call.statement>

                    ¦ <goto.statement>

                    ¦ <repeat.statement>

                    ¦ <control.statement>

                    ¦ <compare.statement>

                    ¦ <exchange.statement>

                    ¦ <assignment.statement>

                    ¦ <label.definition> <basic.statement>

<label.definition> ::= <identifier> :

                     ¦ <number> :
```

```
<if.statement> ::= <if.clause> <statement>
                 ¦ <if.clause> <true.part> <statement>
                 ¦ <label.definition> <if.statement>
<if.clause> ::= IF <compound.condition> THEN
<true.part> ::= <basic.statement> ELSE
<compound.condition> ::= <and.head> <simple.condition>
                       ¦ <or.head> <simple.condition>
                       ¦ <simple.condition>
<and.head> ::= <simple.condition> &
             ¦ <and.head> <simple.condition> &
<or.head> ::= <simple.condition> \
            ¦ <or.head> <simple.condition> \
<simple.condition> ::= ( <statement.list> ) <condition>
                     ¦ <condition>
<condition> ::= ! ZERO
              ¦ ZERO
              ¦ ! CY
              ¦ CY
              ¦ PY ODD
              ¦ PY EVEN
              ¦ PLUS
              ¦ MINUS
<decl.statement> ::= DECLARE <decl.element>
                   ¦ <decl.statement> , <decl.element>
<decl.element> ::= <storage.declaration>
                 ¦ <ident.specification> <type>
                 ¦ <identifier> <data.list>
```

```
<data.list> ::= <data.head> <constant> )

<data.head> ::= DATA (

              ¦ <data.head> <constant> ,

<storage.declaration> ::= <ident.specification> BYTE

                        ¦ <bound.head> <number> ) BYTE

                        ¦ <storage.declaration> <initial.list>

<type> ::= LABEL

         ¦ EXTERNAL

         ¦ COMMON

<ident.specification> ::= <identifier>

                        ¦ <ident.list> <identifier> )

<ident.list> ::= (

               ¦ <ident.list> <identifier> ,

<bound.head> ::= <ident.specification> (

<initial.list> ::= <initial.head> <constant> )

<initial.head> ::= INITIAL (

                 ¦ <initial.head> <constant> ,

<group> ::= <group.head> ; <ending>

<ending> ::= END

           ¦ END <identifier>

           ¦ <label.definition> <ending>

<group.head> ::= DO

               ¦ DO <iterative.clause>

               ¦ DO <case.selector>

               ¦ <group.head> ; <statement>

<iterative.clause> ::=

     <initialization> BY <assignment.statement> WHILE
```

```
          <compound.condition>

        ¦ <initialization> WHILE <compound.condition>

<initialization> ::= <assignment.statement>

                   ¦ <empty>

<case.selector> ::= CASE <register>

<procedure.definition> ::=

          <proc.head> <statement.list> ; <ending>

<proc.head> ::= <proc.name> ;

              ¦ <proc.name> <formal.param.list> ;

<proc.name> ::= <label.definition> PROCEDURE

<formal.param.list> ::= <formal.param.head> <identifier> )

<formal.param.head> ::= (

                      ¦ <formal.param.head> <identifier> ,

<return.statement> ::= RETURN

                     ¦ IF <simple.condition> RETURN

<call.statement> ::= <call> <identifier>

                   ¦ <call> <actual.param.list>

                   ¦ <call> <number>

<actual.param.list> ::= <actual.param.head> <constant> )

<actual.param.head> ::= <identifier> (

                      ¦ <actual.param.head> <constant> ,

<call> ::= CALL

         ¦ IF <simple.condition> CALL

<goto.statement> ::= <goto> <identifier>

                   ¦ <goto> <number>

                   ¦ GOTO M ( HL )

<goto> ::= GOTO
```

```
        ¦ IF <simple.condition> GOTO
<repeat.statement> ::=
     REPEAT ; <statement.list> ; UNTIL <compound.condition>
<control.statement> ::= HALT
                        ¦ NOP
                        ¦ DISABLE
                        ¦ ENABLE
<compare.statement> ::= <register> :: <secondary>
<exchange.statement> ::= <register> == <register.expression>
<assignment.statement> ::= <variable.assignment>
                           ¦ <register.assignment>
<variable.assignment> ::= <variable> := <register.expression>
<register.expression> ::= <register>
                          ¦ ( <register.assignment> )
<register.assignment> ::=
     <register> = <primary> <binary.op> <secondary>
   ¦ <register> = <unary.op> <primary>
   ¦ <register> = <primary>
   ¦ <register.assignment> , <binary.op> <secondary>
<primary> ::= <variable>
            ¦ <secondary>
<secondary> ::= <register.expression>
              ¦ <constant>
<constant> ::= <string>
             ¦ <number>
             ¦ - <number>
             ¦ . <identifier>
```

```
              ¦ . <identifier> ( <number> )

              ¦ . <string>

<register> ::=  A  ¦  B  ¦  C  ¦  D  ¦  E  ¦  H  ¦  L

              ¦  M ( HL )  ¦  BC  ¦  DE  ¦  HL  ¦  SP

              ¦  STACK  ¦  PSW  ¦  CY

<binary.op> ::=  +  ¦  ++  ¦  -  ¦  --  ¦  &  ¦  \  ¦  \\

<unary.op>  ::=  <  ¦  <<  ¦  >  ¦  >>  ¦  !  ¦  #

<variable> ::= M ( BC )

              ¦ M ( DE )

              ¦ <identifier>

              ¦ <identifier> ( <number> )

              ¦ IN  ( <number> )

              ¦ OUT ( <number> )

              ¦ M ( <constant> )
```

C. PROGRAMMING EXAMPLES

Sample ML80 programs are presented in this section, which are intended to illustrate the usage of most language constructs.

1. A bubble sort procedure

```
/* A BUBBLE SORT PROGRAM IN ML80 */

EXCH: PROCEDURE;
      /* EXCHANGE THE CONTENTS OF M(BC), M(HL) */
      D=(A=M(BC));
      M(BC)=(A=M(HL));
      M(HL)=D;
      END EXCH;

SORT: PROCEDURE(N,VEC);
      /* SORT A VECTOR OF AT MOST 255 2-BYTE ELEMENTS */
      /* VEC: ADDRESS OF THE ARRAY TO BE SORTED */
      /* N: ADDRESS OF THE BYTE CONTAINING NO. ELEMENTS */
```

```
        D = 1; /* 'SWITCHED' FLAG: 1 IF SORT NOT YET DONE */
        DO WHILE (A=D,+0) ! ZERO;   /* ADD 0 TO SET FLAGS */
            D = 0;
            HL=N; E=M(HL); /* NO. ELEMENTS TO SORT */
            HL = VEC; /* HL POINTS TO FIRST VEC ELEMENT */
            DO WHILE (E=E-1) ! ZERO;
                BC = HL+1,+1; /* POINT TO SUBSEQUENT ELEMENT */
                A = M(BC) - M(HL); /* SUBTRACT LOW ORDER BYTES */
                HL=HL+1; BC=BC+1; /* POINT TO HIGH ORDER BYTES */
                IF (A=M(BC)--M(HL)) MINUS THEN /* REVERSED */
                    DO; /* EXCHANGE */
                        CALL EXCH; /* HIGH ORDER BYTES */
                        HL = HL - 1; BC = BC - 1;
                        CALL EXCH; /* LOW ORDER BYTES */
                        D = 1; /* SORT NOT YET DONE */
                        HL = BC; /* NEXT ELEMENT */
                    END
                ELSE HL = BC - 1;   /* NEXT ELEMENT */
            END;
        END;
        END SORT;

[MACRO CR '0DH' ]
[MACRO LF '0AH' ]
[MACRO BDOS '3FFDH' ]
[MACRO CPM  '0' ]

PRC: PROCEDURE;
      /* PRINT A CHARACTER (ASSUMED IN REG E) */
      C = 2;
      CALL [BDOS];
      END PRC;

CRLF: PROCEDURE;
      /* SEND CARRIAGE RETURN, LINE FEED TO THE CONSOLE */
      E = [CR]; CALL PRC;
      E = [LF]; CALL PRC;
      END CRLF;

PRL: PROCEDURE;
      /* PRINT A LINE (ADDRESS OF TEXT ASSUMED AT REG HL) */
      DECLARE SAVEHL(2) BYTE;
      SAVEHL = HL;
      CALL CRLF;
      C = 9;
      DE = (HL = SAVEHL);
      CALL [BDOS];
      END PRL;

MAIN: /* A TEST FOR THE SORT PROCEDURE */
      DECLARE ARRAY(31) BYTE INITIAL('AAXXDDEE1144GGHH',
        'JJFFWWPPQQNN55$');
      DECLARE N BYTE INITIAL(15);
      SP = 2900H;
```

```
     HL = .ARRAY; CALL PRL;
     CALL SORT(.N,.ARRAY);
     HL = .ARRAY; CALL PRL;
     CALL CRLF;
     GOTO [CPM];
EOF
```

2. Multiplication and division routines

```
ARITH: PROCEDURE;
       /* SOME BASIC ARITHMETIC FUNCTIONS */

       /* USAGE:    DECLARE ARITH EXTERNAL;
                    L = <FUNCTION NO.>;
                    CALL ARITH;

          FUNCTIONS AVAILABLE:
          1 - BYTE MULTIPLICATION          BC = C * D
          2 - ADDRESS MULTIPLICATION       HLBC = BC * DE
          3 - BYTE DIVISION                C=C/D; B=C%D
          4 - ADDRESS DIVISION             BC=BC/DE; HL=BC%DE
          5 - PRINT A IN HEX FORMAT
          6 - PRINT BC IN HEX FORMAT
          7 - PRINT BC IN UNSIGNED DECIMAL FORMAT
          8 - PRINT BC IN SIGNED DECIMAL FORMAT
                                                               */

[MACRO BDOS '3FFDH' ]
[MACRO PRC CHR 'E=[CHR]; CALL PRC' /* PRINT CHR */ ]
[MACRO PRQ CHR 'E=''[CHR]''; CALL PRC' ]
[MACRO SHR REG '[REG]=(A=>[REG])' /* ROT REG RIGHT THRU CY */]
[MACRO SHL REG '[REG]=(A=<[REG])' /* ROT REG LEFT  THRU CY */]

BMULT: PROCEDURE;
       /* BYTE MULTIPLICATION ROUTINE */
       /* INPUT:  C:  MULTIPLIER
                  D:  MULTIPLICAND
          OUTPUT: BC: UNSIGNED 16 BIT PRODUCT
                  C:  SIGNED 8 BIT PRODUCT      */

       B=0;     /* INITIALIZE PARTIAL SUM */
       E=9;     /* INITIALIZE LOOP COUNTER */
  SHR: /* SHIFT LOW ORDER BYTE OF PARTIAL SUM */
       [SHR 'C'];    /* CY GETS LOW ORDER BIT OF C */
       IF (E=E-1) ZERO /* DONE */ RETURN;
       A=B;     /* HIGH ORDER BYTE OF PARTIAL SUM */
       IF CY THEN A=A+D;     /* ADD MULTIPLICAND */
       B=(A=>A);  /* SHIFT HIGH ORDER BYTE OF PARTIAL SUM */
       GOTO SHR;
       END BMULT;

AMULT: PROCEDURE;
       /* ADDRESS MULTIPLICATION ROUTINE */
```

```
        /* INPUT:   BC: MULTIPLIER (HIGH,LOW)
                    DE: MULTIPLICAND (HIGH,LOW)
           OUTPUT: HLBC: 32 BIT UNSIGNED PRODUCT
                    BC: 16 BIT SIGNED PRODUCT    */

        DECLARE I BYTE;
        HL=0;     /* INITIALIZE PARTIAL SUM */
        I=(A=17);     /* INITIALIZE LOOP COUNTER */
   SHR: /* SHIFT LOW ORDER 16 BITS OF PARTIAL SUM */
        [SHR 'B'];
        [SHR 'C'];
        IF (A=I-1) ZERO RETURN;
        I=A;      /* UPDATE LOOP COUNTER */
        IF CY THEN HL=HL+DE;     /* ADD MULTIPLICAND */
        /* SHIFT HIGH ORDER 16 BITS OF PARTIAL SUM */
        [SHR 'H'];
        [SHR 'L'];
        GOTO SHR;
        END AMULT;

BDIV:   PROCEDURE;
        /* BYTE DIVISION ROUTINE */
        /* INPUT:   C: DIVIDEND
                    D: DIVISOR
           OUTPUT:  C: QUOTIENT
                    D: DIVISOR
                    B: REMAINDER   */

        /* INITIALIZE */
        B = 0;
        L = 8;     /* LOOP COUNTER */
        REPEAT;
           /* SHIFT REM,QUOT LEFT */
           CY = 0;
           [SHL 'C'];
           [SHL 'B'];
           /* SUBTRACT DIVISOR */
           IF (A=B-D) PLUS THEN
              DO;
                 B = A;    /* UPDATE B */
                 C = (A=C\1);    /* SET BIT 0 OF QUOTIENT */
              END;
        UNTIL (L=L-1) ZERO;
        END BDIV;

ADIV:   PROCEDURE;
        /* ADDRESS DIVISION ROUTINE */
        /* INPUT:   BC: DIVIDEND (HIGH,LOW)
                    DE: DIVISOR  (HIGH,LOW)
           OUTPUT:  BC: QUOTIENT
                    DE: DIVISOR
                    HL: REMAINDER     */

        DECLARE N BYTE;
        HL = 0;
        DO N=(A=1) BY N=(A=N+1) WHILE (A=N; A::17) !ZERO;
           /* SHIFT REM,QUOT LEFT */
           CY = 0;
           [SHL 'C'];
           [SHL 'B'];
           [SHL 'L'];
           [SHL 'H'];
           /* SUBTRACT DIVISOR */
           IF ( L=(A=L-E); H=(A=H--D) ) PLUS
              THEN C=(A=C\1)   /* SET BIT 0 OF QUOTIENT */
           ELSE HL=HL+DE;    /* RESTORE */
        END;
        END ADIV;

SHRA:   PROCEDURE;
        /* SHIFT A RIGHT L BITS */
        REPEAT;
           CY = 0;
           A = >A;
        UNTIL (L=L-1) ZERO;
        END SHRA;

PRC:    PROCEDURE;
        /* PRINT A CHARACTER (ASSUMED IN REG E) */
        C = 2;
        CALL [BDOS];
        END PRC;

PRAH:   PROCEDURE;
        /* PRINT VALUE OF A (ASSUMED IN THE RANGE 0-15)
           AS A HEXADECIMAL CHARACTER */
        IF (A::10) MINUS THEN A=A+'0'
        ELSE A=A-10,+'A';
        [PRC 'A'];
        END PRAH;

PRINT$A: PROCEDURE;
        /* PRINT 2 HEX CHARS REPRESENTING THE VALUE OF A */
        STACK = PSW;        /* SAVE A */
        L=4; CALL SHRA;     /* GET HIGH NIBBLE */
        CALL PRAH;          /* PRINT HIGH NIBBLE */
        PSW = STACK;        /* RESTORE A */
        A = A & 0FH;        /* GET LOW NIBBLE */
        CALL PRAH;          /* PRINT LOW NIBBLE */
        END PRINT$A;

PRINT$BC: PROCEDURE;
        /* PRINT 4 HEX CHARS REPRESENTING THE VALUE OF BC */
        STACK = BC;    /* SAVE BC */
        A = B; CALL PRINT$A;
        BC = STACK;    /* RESTORE BC */
        A = C; CALL PRINT$A;
        END PRINT$BC;
```

```
PRDHL: PROCEDURE;
        /* PRINT VALUE OF HL IN UNSIGNED DECIMAL FORMAT,
           SUPPRESSING LEADING ZEROS */
        DECLARE (Q,$HL)(2) BYTE, PRZ BYTE;
        $HL=HL;    /* SAVE HL */
        Q=(HL=10000);
        PRZ=(A=A\\A);   /* PRZ=0: DO NOT PRINT ZEROS */
        REPEAT;
           /* DIVIDE HL BY Q */
           BC=(HL=$HL); DE=(HL=Q); CALL ADIV;
           $HL=HL;  /* SAVE REMAINDER */
           IF (A=C+0) !ZERO       /* QUOTIENT IS NOT ZERO */
              \ (A=PRZ+0) !ZERO THEN
              DO;    /* PRINT QUOTIENT */
                 A=C; CALL PRAH;
                 PRZ=(A=1);      /* STOP SUPPRESSING ZEROS */
              END;
           /* DIVIDE Q BY 10 */
           BC=(HL=Q); DE=10; CALL ADIV; Q=(HL=BC);
        UNTIL (A=Q; A::1) ZERO;
        HL=$HL;      /* LAST REMAINDER */
        A=L; CALL PRAH;
        END PRDHL;


/* MAIN: SELECT THE ADEQUATE FUNCTION */
        H=0; L=L-1;
        DO CASE HL;
           /* 1 */ CALL BMULT;
           /* 2 */ CALL AMULT;
           /* 3 */ CALL BDIV;
           /* 4 */ CALL ADIV;
           /* 5 */ CALL PRINT$A;
           /* 6 */ CALL PRINT$BC;
           /* 7 */ DO;
                      HL=BC; CALL PRDHL;
                   END;
           /* 8 */ DO;
                      IF (A=B+0) MINUS THEN   /* BC<0 */
                         DO;
                            STACK=BC;
                            [PRQ '-'];
                            BC=STACK;
                            L=(A=0-C); H=(A=0--B);
                         END
                      ELSE HL=BC;
                      CALL PRDHL;
                   END;
        END; /* CASE */
        END ARITH;
EOF
```

3. Usage of external procedures

```
/* TEST PROGRAM FOR ARITH PROCEDURES */

[MACRO CPM '0' ]
[MACRO BDOS '3FFDH' ]
[MACRO CR '0DH' ]
[MACRO LF '0AH' ]
[MACRO PRC CHR 'E=[CHR]; CALL PRC' ]
[MACRO PRQ CHR 'E=''[CHR]''; CALL PRC' ]
[MACRO PRV VAR 'BC=(HL=[VAR]); L=8; CALL ARITH' ]

PRC: PROCEDURE;
     /* PRINT A CHARACTER (ASSUMED IN REG E) */
     C=2; CALL [BDOS];
     END PRC;

CRLF: PROCEDURE;
     /* CARRIAGE RETURN, LINE FEED */
     [PRC '[CR]' ];
     [PRC '[LF]' ];
     END CRLF;

/* MAIN: */
     DECLARE ARITH EXTERNAL;
     DECLARE (X,Z)(2) BYTE, Y(2) BYTE INITIAL (-9);
     SP=2900H;
     DO X=(HL=-50) BY X=(HL=X+(BC=10)) WHILE (A=X-100) !ZERO;
        /* TEST ADDRESS MULTIPLICATION */
        BC=(HL=X); DE=(HL=Y+1,+1); Y=HL;
        L=2; CALL ARITH; Z=(HL=BC);
        CALL CRLF;
        [PRV 'X'];
        [PRQ '*'];
        [PRV 'Y'];
        [PRQ '='];
        [PRV 'Z'];
     END;
     CALL CRLF;
     GOTO [CPM];
EOF
```