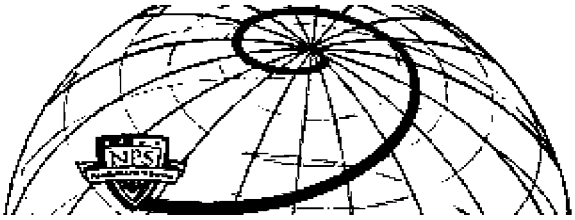
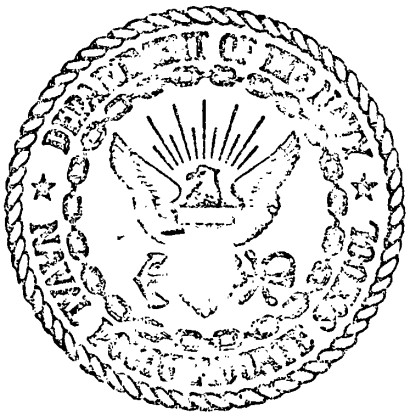


ML80: A STRUCTURED MACHINE-ORIENTED MICROCOMPUTER PROGRAMMING LANGUAGE

Luiz Roberto Borges Pedroso

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

ML80: A STRUCTURED MACHINE-ORIENTED MICROCOMPUTER PROGRAMMING LANGUAGE

by

Luiz Roberto Borges Pedroso

December 1975

Thesis Advisor: G. A. Kildall

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ML80: A Structured Machine-Oriented Microcomputer Programming Language		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; December 1975 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Luiz Roberto Borges Pedroso		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1975 13. NUMBER OF PAGES 216
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this Report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

microcomputer
structured programming
systems programming language
resident compiler
macro processor
PL/M-like

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

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

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

executes on a microcomputer system with 16K bytes of main memory.

ML80: A Structured Machine-Oriented Microcomputer Programming Language

by

Luiz Roberto Borges Pedroso
Lieutenant Commander, Brazilian Navy
B.S., Pontificia Universidade Catolica do Rio de Janeiro, 1969

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 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.

TABLE OF CONTENTS

I. INTRODUCTION

A. MICROCOMPUTER TECHNOLOGY

Recent advances in digital circuit fabrication techniques have led to general availability of low-cost digital computer components known as microcomputer chip families. These components include central processing units, memory systems and associated peripherals for input, output, and system timing. Taken together, they are used to construct inexpensive and compact digital computer applications which revolutionize previous digital design practices. In addition, a wealth of new computer applications are appearing, ranging from intelligent terminals and instruments through traditional minicomputer control tasks. A survey of these applications is found in Refs. 20 and 25.

B. MICROCOMPUTER PROGRAMMING LANGUAGES

The remarkable advances found in microcomputer hardware technology have not been paralleled by corresponding software support systems.

A survey by Theis [20] contains some interesting data which indicate the state of microcomputer software as of December 1974; for the twenty-eight different microcomputer systems in the survey, the following software design aids were available:

Resident Assembler.....22 of 28

Cross Assembler.....26 of 28

Monitor.....14 of 28

Languages: PL/M.....2 of 28

BASIC.....1 of 28

FORTRAN.....1 of 28

Currently, PL/M [8] is the only high level microcomputer systems programming language. It was developed by Intel, and is used to cross-compile programs for Intel's 8008 and 8080 processors. To achieve portability, the PL/M compiler is written in ANSI Fortran, and executes on large scale host machines.

Clearly, software design aids for microcomputer systems development are primitive, when compared with corresponding larger machine facilities.

C. MOTIVATIONS AND OBJECTIVES OF ML80

Given the state of microcomputer software design tools, it seemed that a resident language for microcomputer programming was needed to fill the gap between PL/M and simple assembly languages. The purpose here is to describe the design and implementation of one such language.

The goal of machine transportability was a remote possibility, since current microcomputers have different architectures, and no previous experience on transportable microprocessor languages was available. Thus, the project was oriented toward the currently popular 8080 microprocessor [6,7]. The language to be developed, called ML80, would allow full use and control of the resources of the 8080, while providing a convenient notation for writing clear and

comprehensible programs. It would also facilitate modular programming, debugging and documentation.

To attain these objectives, a survey of existing programming languages was conducted to identify desirable ML80 features. Particular attention was devoted to current software engineering practices, such as structured programming, while retaining the advantages of machine oriented languages. The basic factors which influenced the design of ML80 are investigated in the following chapter.

II. LANGUAGE DESIGN

A. STRUCTURED PROGRAMMING

The term "structured programming", coined by Dijkstra [4], has been the object of much discussion in the past few years. The fundamental notion of structured programming is that program reliability and readability can be improved if systematic and disciplined methods of program development are adopted. As Wirth puts it,

> ...[the term structured programming] is the expression of a conviction that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that particular techniques should be replaced (or augmented) by a method. At its heart lies an attitude rather than a recipe: the admission of the limitations of our minds. The recognition of these limitations can be used to our advantage, if we carefully restrict ourselves to writing programs which we can manage intellectually...[21, p. 249]

Methods for composing intellectually manageable programs are fundamental to structured programming: the goal is to reduce large problems into hierarchical systems where each level can be fully understood.

Another important point is related to program reliability. Traditional testing methods are recognized as unsatisfactory, and in general cannot guarantee that a program is error free. This assurance can only be obtained by proving program correctness, which is a non-trivial task. One of the notions of structured programming is to develop a program and its correctness proof in parallel, level by level in a hierachical structure, thereby guaranteeing the

correctness of the final program. Correctness proofs and program readability are easier to obtain if the programmer restricts himself to constructions with well known properties. The unrestricted use of goto's, for instance, may impair program clarity and reliability.

Several ideas have been associated with structured programming by different authors, and a summary of the suggestions found in the literature is given by Abrahams [1]:

- programming by stepwise refinement;
- hierarchical program organization, or use of levels of abstraction as a guiding design principle;
- programming without the goto statement;
- segmenting programs into modules that occupy less than a page;
- avoidance of local variables;
- proofs of program correctness;
- the "chief programmer team" approach to project organization.

Some rather radical attitudes have been taken on certain of these items; the elimination of goto's, for instance, has become a controversial point. A summary of pros and cons on the goto issue is provided by Knuth [13].

One aspect on which most authors agree, however, is the need for adequate control structures in programming languages. This is where the principles of structured programming have had most impact. The Fortran language, for instance, has limited control constructs and requires the

use of labels and goto's for program control. For this reason, and despite its computational power, Fortran is now considered by many an inadequate language. The claim is that in Fortran programs the flow of control and the logical structure are usually hard to identify.

Accordingly, a number of attempts have been lately made to incorporate structured features into Fortran. The usual approach is to use a preprocessor to translate structured Fortran into standard Fortran. Meissner [16] analyzes twenty of these preprocessors.

The influence of structured programming ideas can also be seen in the design of recent languages such as BLISS [24], which abolished the goto statement, and PASCAL [22], which provides both control and data structuring capabilities.

In summary, structured programming has had a significant impact in programming methodology: the importance of its fundamental principles must be considered in the design of any new programming language.

B. MACHINE ORIENTED LANGUAGE DESIGN

Although the advantages of high level languages are unquestionable, there are occasions in which machine dependent languages are required. One of such instances is when appropriate high level languages or their translators are not available or feasible.

Despite the proliferation of high level languages [19], assembly languages are still in use: they are simpler to de-

fine and implement, and thus are nearly always available. However, the expression of algorithms in assembly language is usually time-consuming and cryptic. Errors are more likely to occur, and may be harder to detect in assembly listings than in high level programs, due to the use of mnemonic operation codes and the absence of structuring facilities to organize programs into logical units. Further, all control constructs must be explicitly formulated by the programmer, which often obscures program logic.

To remedy this situation, many assembly languages provide macro facilities. Although convenient, macro capabilities do not greatly enhance program structure or statement notation, since they are simply a shorthand for certain sequences of instructions.

Several authors have presented assembly language designs with built-in structuring facilities [5,14,26]. Recently, Popper [17] suggested the addition of suitable control structures to Intel's 8008 and 8080 assembly languages. The language designed by Popper, called SMAL, includes IF-THEN-ELSE, DO-END and DO-WHILE constructs and replaces mnemonic operation codes by a more natural notation. The implementation of SMAL consisted of a SNOBOL4 preprocessor, which mapped the structured source program into assembly language; the assembly language program was then processed by the standard Intel macro assembler. Thus, the idea of developing structured machine dependent languages is not new, but merely the recognition that even when high level languages

are not available, some of their conveniences can be brought to the aid of the programmer.

A machine dependent language, called PL360, which captured features of higher level languages and peculiarities of the IBM/360 architecture was described in 1968 by Wirth [23]. PL360 has an ALGOL-like structure, and is defined by a formal grammar; the implementation consisted of a compiler written in its own language. The programming system is highly efficient, since the compiler is single-pass and generates executable code, with no need for extra assembly language translations. The motivations for designing a language of this nature are better described in Wirth's own words:

> ...it was decided to develop a tool which would (1) allow full use of the facilities provided by the 360 hardware; (2) provide convenience in writing and correcting programs, and (3) encourage the user to write in a clear and comprehensible style. [23, p.38]

The merits of this particular approach are found in the use of sound principles of language design in conjunction with state-of-the-art compiler design practices. Further, the PL360 implementation demonstrates that assembly languages can be replaced by structured languages which are both machine oriented and adequate for proper program formulation.

C. ML80 DESIGN DECISIONS

The information presented in the previous sections provides the background for the decisions affecting the ML80 language design.

The fundamental purpose here was to design a convenient language for microcomputer system-level programming. Control and data structuring facilities should be included to encourage good programming practices, while providing operational efficiency. It was further decided that the language should be translated using a resident compiler, without the need for cross-preprocessors or cross-compilers. In this respect, an important constraint was the exact 8080 configuration required for running the compiler. Although the 8080 can access up to 64K bytes of main memory, it was felt that the language should be available on systems with a minimum of 16K bytes, which is a common configuration for 8080 developmental systems.

A complete high level language of the complexity of PL/M could not be implemented in the time allocated for the project; thus, it was necessary to establish priorities for the features to be included in the language. Considering the previous discussion, adequate control structures became the highest priority.

To compensate for the absence of convenient data structuring capabilities, the language would provide adequate compile-time features, such as expression evaluation, conditional compilation, and macros. Macro processing should be an optional phase.

The language should also include load-time facilities, such as the ability to incorporate precompiled procedures into the object program; this implied that the compiler must

generate relocatable code.

Another guideline was to pattern the language after PL/M as much as possible, so that PL/M programmers would not face a totally incompatible syntax while using ML80. In addition, since ML80 would provide operations at the machine level, the constructs in the language should never destroy the contents of registers without a corresponding explicit statement in the source program. ML80 programs should indeed reflect the actions performed by the machine.

It was also considered a requirement to define the language with a formal grammar, and use systematic methods for construction of the compiler, rather than adopt ad hoc implementation techniques.

ML80 was developed along these lines. The final version of ML80, along with a description of its implementation are presented in the next chapters. As mentioned above, ML80 is structurally similar to PL/M, while SMAL, PL360, and the C language [10] also had some influence on the design. The individual features derived from these languages are described in Chapter V.

III. LANGUAGE SPECIFICATIONS

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, as described in Chapter IV.

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 III.C.

2. 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.

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 [8] 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:

- macro declaration;
- macro call;
- assignment statement;
- evaluation statement;
- 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

C0[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 M80

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 '5' ]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.

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

VA[INT 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. 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 numeric values in different formats, as well as special characters, including the brackets themselves (brackets inside strings always flag the presence of macro calls; therefore a special mechanism is needed for brackets which are to be 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 statements.

The effect of DEC, OCT, HEX is to cause the replacement of the evaluation statement by the value of <expression>, in decimal, octal or hexadecimal format, respectively. For instance,

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

yields

```
AB23,0FFFFH,0000200,CD15
```

When CHAR is used, the evaluation statement is replaced by the character whose bit pattern is equal to the least significant byte of the value of <expression>. For instance, assuming that ASCII encoding is used in the implementation 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 emoty 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 I]
    [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 stronoly influenced by both the PL/M language [8] 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('APCD',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 whenever 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. 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. 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;
      ...
```

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

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 instructions 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.

k. 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 */
        CSHR 'B';
        CSHR '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 */
        CSHR 'H';
        CSHR '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;
           CSHL 'C';
           CSHL '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;
                           TPRO '-');
                           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
```

IV. LANGUAGE IMPLEMENTATION

A. ORGANIZATION OF THE LANGUAGE PROCESSORS

1. Parser generation

One of the advantages of using a formal grammar to define a programming language is the possibility of mechanically generating a parser to recognize programs written in that language. The parser in this case is a table driven pushdown automaton, which assumes a sequence of states while scanning the source program, and eventually enters a final state that indicates whether the input is well formed according to the underlying grammar. A parser of this kind can also be used as a transducer, by associating some action, usually the emission of a string, with each parser move.

Of course, not every grammar is suitable for automatic parser generation. In the design of a language, much time may be spent in manipulating its grammar so that it fits a category for which a parser can be generated [15].

A large class of context-free languages can be described by LR(k) grammars [12]. LR(k) parsers belong to the general class of shift-reduce parsing algorithms, whose operation can be characterized as follows: the input stream is scanned from left to right, and the symbols encountered are shifted into a pushdown stack, until the right hand side (also called a handle) of a production is formed at the top

of the stack; a reduction is then applied (i. e. the handle is replaced by the left hand side of the production), and the parsing proceeds. These steps are repeated until an error is detected or the entire input is successfully scanned.

LR(k) parsers base their decisions on the next k input symbols, and all the information previously accumulated on the parse stack. The general algorithms for parser generation under these conditions lead to extremely large tables. However, special techniques have been developed to generate practical parsers for more restrictive, but still useful, cases. SLR (simple LR) and LALR (look-ahead LR) are two examples of these methods. Their descriptions and theoretical foundations can be found in Aho and Ullman [3].

The parsing generation scheme adopted in the ML80 implementation was based on YACC (Yet Another Compiler-Compiler) [9], a program available on PDP-11 computers under the UNIX time-sharing system. YACC implements an LALR(1) parser generator; since k=1, only one input symbol is used for the parsing decisions. The inner workings of LR(1) parsers and the algorithms used for their generation are described in detail in Aho and Johnson [2]. The basic types of actions performed by an LR(1) parser are:

- shift: advance to next input symbol; push another state number onto the parse stack;
- reduce: pop from the parse stack a number of states equal to the number of elements in the handle of the production applied, an then push another state onto the stack;

- accept: the input is considered well formed;
- error: a syntactic error was detected.

The main output from YACC consists of tables, to be interpreted by one of its own library routines. Since the ML80 system was to execute on the 8080, the parser tables interpreter had to be rewritten. To facilitate grammar modifications it was decided to transfer parser tables directly from the PDP-11 to the 8080; the physical connection between the two computers and an interface program were implemented for this purpose.

2. System organization

As previously mentioned, a fundamental goal was to implement the ML80 language on an 8080 installation with 16K bytes of main memory, since this is representative of many microcomputer developmental systems. From the beginning it was apparent that this limited memory space would rule out a single phase compiler. However, another extremely desirable, but conflicting goal was to have code generated in a single pass, to speed the compilation process.

The compiler operating environment also affected the overall system organization. Since the CP/M [11] operating system was already available on the 8080 installation to be used for the project, it was the natural choice. CP/M is a comprehensive supervisory system, providing a number of standard I/O functions, a named file system, text editor, dynamic debugger and other utility programs. The file system is maintained on diskettes (floppy disks) with the ap-

proximate capacity of 256K bytes each. Two diskette drives were available for the project.

The only systems language available under CP/M was the 8080 macro-assembler. Programs written in PL/M, Intel's microcomputer high level language, although able to execute under CP/M, had to be cross-compiled on a larger system, such as S/360. Due to the limited time available for completion of the project, and also because CP/M itself is mostly written in PL/M, it was decided that the ML80 compiler should be implemented using PL/M.

CP/M takes about 4K bytes of main memory; therefore the compiler would have to operate in 12K bytes. A significant portion of this space would be commited to the parser tables generated by the YACC program (about 4K and 1K bytes for the L80 and M80 grammars, respectively).

Due to the fact that M80, the macro-oriented component of ML80, is a self-contained language, it was decided that macro processing should be a separate phase. An advantage of this approach is that the macro processor can be used on programs written in other languages. Further, a large memory area is then available for strings, macro definitions, and other variable length data, as well as space for stacks to be used in recursive macro calls.

An experimental version of the L80 compiler as a single program was implemented on the PDP-11/50 system. This version was written in C, a language used for applications and systems programming under the UNIX [18] operating

system, and took approximately 30K bytes of main storage. Although the size of the C version of the L80 compiler could only be taken as a rough indication of the size of a corresponding program in PL/M, it became evident that a single program would not be feasible. The question then arised of how to decompose the L80 compiler into phases occupying less than 12K bytes.

One possible approach would be to use overlay techniques; this was deemed inefficient, since the major components of the compiler, namely the lexical analyzer, the parser and the code generation routines, work in an interdependent manner; the frequent swapping of these code segments during a compilation would result in excessive overhead. The alternative solution, then, was to divide the compilation process into sequential phases.

In general, multiple pass compilation involves preliminary translation of source programs into some form of intermediate language. The languages commonly used for this purpose are Polish notation, triples or quadruples. In the L80 case, however, which had been designed to generate code as the parsing proceeds, another approach was adopted. It was observed that to each source program corresponds a unique sequence of parsing actions (in the case of an LR(k) parser, each action is either shift, reduce, error or accept); therefore, an unambiguous intermediate language would be precisely the sequence of actions performed in the parsing of a program. This led to a natural division of the

compiler in two phases: lexical and syntactic analysis, and code generation. In the PDP version, these phases operated concurrently, while in the 8080 version they would operate sequentially, that is, the code generation program would reproduce all actions performed by the parser, on a subsequent memory load.

A final aspect to be considered was related to the linkage editing of compiled modules. One of the project goals was the generation of relocatable code. This implied that a linkage editor should be developed, which would be an independent program, operating on the named file system supported by CP/M.

The following scheme was then adopted for the resident ML80 compiler: the system is composed of four modules, namely

- M81: interpreter for the M80 language;
- L81: parser for L80 programs;
- L82: code generator;
- L83: linkage editor.

These modules communicate among themselves by means of CP/M files.

The general operating procedure for the system can be described as follows:

- the M81 macro processor is used to transform a source ML80 program into a version containing only L80 statements;
- the output from M81 (or any program directly written in L80) is submitted to L81, which performs the lexical and

syntactic phases of the compilation process; the output of L81 consists of two files: one is a list of all the steps followed while parsing the input text; the other contains a list of all the symbols (identifiers and strings) defined in the source program;

- if no errors occurred in the previous phases, L82 is invoked; all the information it needs to translate the original source program into executable code is contained in the two files created by L81. The output of L82 is a relocatable module, made up of three files: one contains the image of the object program code area, the second one comprises program constants and initialized areas, and the last one is a relocation table to be used at load time; error messages also can occur at this stage, since the source program may contain semantic errors;

- after successful compilation, the output of L82 and the symbol list file generated by L81 are kept in the file system, as parts of a relocatable library. L83 then can be invoked to extract these modules and link them into a single executable object program, to be loaded at any origin assigned by the operator. L83 automatically resolves external references present in each module; its output is the memory image of the executable object program, and, if required, a cross reference list of all symbols in the source modules and their final storage locations.

B. COMPILER IMPLEMENTATION

1. Data structures

a. L81 data structures

As previously mentioned, L81, the first phase of the L80 compiler, is responsible for the following tasks:

- lexical analysis, resulting in a list of all symbols defined in the source program;
- syntactic analysis, and generation of a file containing the sequence of actions performed in the parsing process.

The LR(1) parsing mechanism provided by the YACC program requires a parse stack, where the parser states are maintained, along with a set of tables containing the information necessary for the parsing decisions. Additional stacks are used in parallel with the parse stack; they contain information related to the actual translation of the source program, such as pointers into auxiliary tables, values of numeric tokens, and temporary values used in reductions. Given the multi-phase organization of the L80 compiler, these stacks are manipulated in L82, which is the code generating module. L81 therefore works with only one parse stack.

Each time the parse stack is altered as a consequence of a shift or reduce action, this fact is communicated to L82 in the form of a field written to the parser actions file. This record contains the type of action and some complementary information needed by L82 to execute the semantic function associated with the particular parse

step. For instance, when the parser decides to shift an identifier, a logical record is emitted containing a code representing the shift action and a value which is a pointer to the identifier in the symbols file.

The contents of the symbols file is held in memory while the parsing proceeds, and dumped onto disk at the end of the process. The symbols file is built by the lexical analyzer, which is responsible for identifying the basic tokens in the source program, such as reserved words, identifiers, strings, numbers, special characters and comments. Symbols never appear twice in this list; this implies that whenever an identifier or string is detected, the symbol list is searched to determine whether it already contains that symbol. To speed up this process, a hash coding technique is adopted.

Summarizing, L81 requires the following data structures to perform its functions:

- parser tables;
- parse stack;
- symbol list;
- hash table.

The parser tables are implemented as 16-bit vectors. Six tables are required to drive an LR(1) parser under the YACC scheme; the tables, and their respective contents, are given below:

- the parser actions table (LRACT) lists the possible actions as a function of the current input symbol, for each

parser state;

- the actions pointers table (LRPACT) holds pointers to state sub-tables within LRACT, for each parser state;
- the first reduction table (LRR1) gives the internal number of the left-hand side of each production in the grammar;
- the second reduction table (LRR2) lists the number of elements in the right-hand side of each production in the grammar;
- the goto function table (LRGO) gives the next parser state after each reduction, as a function of the previous state and the left hand side of the production applied;
- the goto pointers table (LRPGO) holds pointers to entries in LRGO, for each non-terminal symbol obtainable through a reduction.

The parse stack (PSTACK) is used to hold parser state numbers, where the current state is given by the top-most stack element. The use of the parse stack in conjunction with the above tables is described in Section IV.B.3.

The symbol list (SYMLIST) is a character array where strings and identifiers are kept. No limit is imposed by the L80 language on the length of user-defined symbols, and thus the entries in SYMLIST are variable length. The end of each entry is marked by an end-of-symbol character (a zero byte).

Each different symbol in the source program, irrespective of scope or redeclarations, appears exactly once in the symbol list. A hash code is computed for each entry

in SYMLIST; entries with same hash code are organized as linked lists. Pointers to each one of these lists are kept in a hash table (HASHTAB). The link field for each symbol in SYMLIST is kept in the two bytes immediately to the left of the symbol; therefore, if a symbol starts at SYMLIST(I), the pointer to the previous symbol with same hash code is stored in SYMLIST(I-2) and SYMLIST(I-1).

At the beginning of a compilation, SYMLIST contains only L80 reserved words, which are also stored according to the above scheme. Reserved words, however, also require their internal numbers, to be used in connection with the parser tables. These numbers are assigned by YACC during the table generation phase, and are always greater than 256 for reserved words longer than one character (the internal number for a single character reserved word is the value of its ASCII representation).

Multicharacter reserved words and their respective internal numbers are stored as follows. If a reserved word starts at SYMLIST(I), then its internal number, minus 256, is at SYMLIST(I-3). Reserved words with only one character are not maintained in the symbol list, but instead are directly recognized by the lexical analyzer.

It must be noted that the portion of SYMLIST containing reserved words is not passed from L81 to L82, since the only role of these tokens is to cause recognition of the language constructs, which is conveyed by the parsing actions themselves.

b. L82 data structures

Given the source program symbol list and the parser actions file, L82 translates this information into machine code. More specifically, L82 generates the following output:

- code area file, containing executable instructions;
- initialized data area file, with an image of the initialized storage area allocated to the source program;
- relocation table file, containing pointers to those address fields in the previous two files which must be relocated at load time.

As previously mentioned, L80 relocatable programs are made up of three segments: a code area, an initialized data area, and an uninitialized data area. Only the first two segments are output by L82 since the contents of the unitialized data area is irrelevant.

The parser actions file drives the L82 parser in reducing the source program according to the grammar rules. Although a parse stack is not maintained in L82, parallel stacks containing semantic information are manipulated, as if they were synchronized with the parse stack of L81.

The symbol list created by L81 is, as the name implies, a simple list of the tokens found in the source program. However, the code generation routines require additional information about these symbols. This information is organized by L82 in a separate symbol table. Due to the block structure of the L80 language, this table works as a

stack: when a group is entered, declarations inside the group cause new entries to be pushed into the symbol table; these entries become the effective definitions for the corresponding symbols, and supersede any existing definitions for these symbols. When a group terminates, these entries are removed, thereby reactivating the previous definitions.

Given the above scheme, it is necessary to save the current symbol table top pointer each time a new group is entered. Another stack, called the block level stack, is used for this purpose.

Another aspect to be considered is the generation of code involving references to symbols whose address is not yet defined. This happens, for instance, in the case of forward branches. Unresolved references are recorded in a table, and remain there until the address of the referenced symbol is determined. This table, called the relocation table, is also used as a temporary buffer for addresses which have to be relocated.

The following data structures are then utilized by L82:

- semantic stacks;
- symbol list;
- symbol table;
- block level stack;
- relocation table.

The format and content of these structures are described in

detail below.

Three semantic stacks are used: the handle stack (PSH), and two auxiliary stacks (PSX, PSY). Entries in PSH and PSY are 16-bit words, while PSX is a byte array. The information maintained in these stacks varies according to the current handle. Typically, stack entries contain such information as pointers to other tables, and temporary information generated in reductions. Additional details are presented in Section IV.B.4.

The symbol list (SYMLIST) is essentially the same as for L81, except that it does not contain reserved words, as mentioned above.

Since the variable length information about source program symbols (the symbols themselves) is stored in SYMLIST, the symbol table (SYMTAB) is a structure with fixed length entries. Each entry consists of the following fields:

- symbol type (STYPE, 4 bits), giving the type of the entry (label, procedure, external, etc.);
- symbol base (SBASE, 4 bits), which holds the name of the segment (code area, work area, etc.) in which the symbol was allocated;
- symbol displacement (SDISPL, 16 bits), giving the relative address of the symbol within segment SBASE;
- symbol link (SLINK, 8 bits), which is a pointer to the previous entry in SYMTAB with same name as the current symbol;

- symbol atribute (SATR, 16 bits), containing additional information about the symbol, depending on its type;
- symbol name pointer (SNAME, 16 bits), which is a pointer to the corresponding symbol in SYMLIST.

The block level stack, as previously mentioned, is used to save the status of SYMTAB each time a new scope region is entered. In L80, non-initialized variables belonging to independent blocks at the same level share memory locations. This allocation scheme conserves memory space in the case of source programs made up of several parallel blocks in which local variables are declared. To implement this mechanism, a second stack is needed in conjunction with the block level stack. The total structure then becomes:
- block level stack (BLSTACK, 8 bits), which is a save area for the SYMTAB top pointer;
- work area pointer stack (WASTACK, 16 bits), which is used to save the work area location counter.

The relocation table (RELTAB) is used to store information about unresolved relocatable memory references. Entries in RELTAB are fixed length, and comprise the following items:
- reference segment (RSEG, 4 bits), which holds the name of the segment containing the reference to be relocated;
- reference location (RLOC, 16 bits), giving the relative address of a relocatable reference within segment RSEG;
- referenced base (RBASE, 4 bits), containing the name of the referenced segment;

- referenced displacement (RDISPL, 16 bits), which holds the relative address of the referenced location within segment RBASE;
- external reference pointer (REXT, 16 bits), which is a pointer to the name of an external module in SYMLIST.

In order to conserve space, as soon as an entry is resolved it is dumped by L82 onto the relocation table file; therefore, only unresolved references are kept in memory.

2. Lexical analysis

The lexical analyzer in L81 is responsible for getting and identifying input tokens. This routine (LRLEX) is invoked repeatedly by the parser during the syntactic analysis of a source program. When called, LRLEX scans the source program for the next input symbol, and returns to the parser the type of the token detected along with some qualifying information.

The basic types of tokens recognized by LRLEX and their respective qualifying data are:
- reserved words, identified by their internal numbers;
- identifiers, associated with pointers to their corresponding entries in SYMLIST;
- numbers, represented by their binary values;
- strings, identified by pointers to their corresponding entries in SYMLIST;
- end-of-file.

Rather than scanning the input directly, LRLEX in-

vokes a lower level routine (GETTOKEN), which automatically skips comments, and usually returns a complete token, such as a number, a special character, or a short identifier. Long identifiers and strings are transmitted from GETTOKEN to LRLEX in blocks of about 30 characters; LRLEX is responsible for accumulating these symbols in SYMLIST, and also for the table lookup required for identifiers and strings.

The recognition of reserved words is performed by LRLEX; GETTOKEN treats reserved words as regular identifiers.

Some special operators in the L80 language are made up of two characters, such as ++ or <<. Since YACC does not process multicharacter special terminals in the same way as alphabetic reserved words, the internal numbers of special terminals cannot be obtained in a systematic fashion, but rather must be handled separately. This is most conveniently done at the lexical level.

The recognition of double operators is simplified by the fact that GETTOKEN looks ahead one extra character in the input stream, and therefore can base its decisions on two characters at a time. This scheme is also useful for comments, which are enclosed in the double separators /* */, and in the identification of numbers and strings.

3. Syntactic analysis

The syntactic analysis process is driven by the tables mentioned in Section IV.B.1. The use of these tables is now described for each type of parser action.

The LRACT table contains the basic information about the possible actions in a given state. The entries in LRACT are 16-bit words, and have the following format (in hexadecimal notation):

1xxx: if the input symbol is xxx, then execute action contained in the next entry, otherwise skip it;

2sss: shift state sss into the parse stack, get next input symbol;

3ppp: reduce by production ppp;

4000: accept the source program;

0000: error.

For instance, if the the entries for the current state are:

1007,2009,

1009,3006,

1008,3007,

3005

then the parser will perform the following actions:

"if input is symbol 7 then shift state 9; else
if input is symbol 9 then reduce by production 6; else
if input is symbol 8 then reduce by production 7; else
reduce by production 5".

When a shift is performed, a new state is pushed into the parse stack; the actions corresponding to this state are obtained from LRACT, and are used to determine the next parse step.

When a reduction by some production p occurs, n states are popped from the parse stack, where n is the

number of elements in the right hand side of production p. The new topmost state in the stack and the left hand side of p are used to determine the next state, which is then pushed onto the parse stack. The function that gives the next state after a reduction is called the goto function, and is stored in the LRGO table. The entries in this table have the following (hexadecimal) format:

xxxxssss: if the left hand side of production p is symbol xxxx then the new state is ssss, else continue;

FFFFssss: default - the new state is ssss.

After pushing the state determined by the goto function, the parser resumes its loop, and consults LRACT again.

When an error occurs, a recovery procedure is attempted, so that the parsing of the source program may be continued. The L80 grammar contains a production of the form:

<statement> ::= <error>

which is used in connection with the error recovery procedure. When a syntactic error is detected, the current input symbol is discarded, and replaced by the token <error>. States are then popped from the parse stack until a state with shift on <error> is encountered. This production is eventually used in a reduction, causing the inacceptable input construct to be brought to the category of <statement>. This scheme allows the next statement in the program to be parsed without excessive contamination from previous errors. Since usually more than one input symbol is discarded

during the error recovery, a flag is set to avoid error message redundancy. This flag is cleared only when a successful reduction (by some production other than <statement> ::= <error>) is performed.

If in the recovery process the situation arises in which no states can be popped from the parse stack, then the error is considered unrecoverable, and the parsing is aborted.

The parsing actions executed by L81 are communicated to L82 by means of the parser actions file; error messages are not written onto that file, but rather sent to the operator. To allow the inclusion of source program line numbers in the messages emitted by L82 in the case of semantic errors, pseudo actions are included in the actions file which update the line counter in L82.

4. Code generation

L82 is the code generating module for the L80 compiler. The core of the program is a case statement where the semantic actions associated with productions are triggered. This case statement is entered immediately before a reduction is performed. At this point the handle for the reduction is complete, and their associated values carried in the semantic stacks (PSH, PSX, PSY) can be used as input for the code generation routines.

Many productions do not cause immediate generation of code, but rather invoke operations such as symbol table manipulations or address backstuffing. After the case

statement is executed, output values from the semantic routines replace the handle values on top of the stacks, and the cycle is repeated.

L82 stacks are implemented in PL/M as arrays. Therefore, each reference to a stack entry involves the computation of an index. Since the handle values are extensively referenced throughout the program, a substantial amount of memory space would be wasted on these index operations. To conserve memory, the following method was used: before each reduction, the handle values are moved to a small number of non-indexed locations, which are then used as input and output parameters for most semantic routines. These values are moved back to the stacks when the reduction is completed.

Another artifice employed to reduce the size of L82 is related to the actual generation of instructions. About 240 different operation codes are available in the 8080 instruction set. Since every one of these instructions is accessible through the L80 language, they must be generated somehow by L82. In many cases this can be done by means of a simple formula, thanks to the peculiar characteristics of the operation codes themselves. For instance, if registers BC, DE, HL, SP are associated with the numbers 0,1,2,3, then the respective DAD (double add) instructions can be obtained by the formula 9+16*r, where r is the register number. Multiplications are costly in the 8080, however, since they are performed through subroutines. Thus, the values are chosen

so that formulas for operation code generation use only and's, or's and xor's.

The most complex routines in L82 are those which deal with address generation. This is due to the multiplicity of control structures in the L80 language, the relocation of addresses, and the provision of the dot operator, which yields the address of any symbol in the current program or in an external module.

The relocation table plays a major role in address computation. When a reference to an unresolved address is issued, as in a forward branch or in a compiler-created forward reference, an entry is created in the relocation table, with a pointer to the referring location. This entry is then linked to any other entries which may refer to the same unresolved address. Eventually, when this address is resolved, it is backstuffed into all the entries in the corresponding linked list. These entries are then dumped onto the relocation table file, to make room for new unresolved references.

Due to the fact that L80 programs are relocatable, all the addresses manipulated by L82 are made up of two parts: a base and a displacement. The base is the name of the segment (code area, work area, etc.) in which the address belongs; the displacement is the relative address within that segment. When an address refers to an external module, a pointer to the name of the referenced module in the symbol list is also inserted in the address entry in the

relocation table.

Another task to be performed by L82 is the maintenance of the symbol table. As mentioned in Section IV.B.1., entries in the symbol table contain a pointer to their respective symbols in the symbol list. For a given symbol, more than one entry may exist in the symbol table, corresponding to redeclarations in nested blocks. When an inner block is exited, its local definitions are popped from the symbol table. However, the information contained in these entries is needed if a reference list of assigned addresses is to be obtained at load time. In this case the data in the entries to be removed are saved in the form of symbol descriptor records written onto the relocation table file. Each symbol descriptor consists of a pointer to a symbol in the symbol list, its type, base and displacement, and the current source program line number. This information allows contruction of a reference list when the program is loaded.

As previously mentioned, L82 reproduces all steps performed by the parser. When the step is a shift action, the value associated with the shift is pushed into the main handle stack (PSH). In the case of identifiers and strings, this value is a pointer to the corresponding entry in SYMLIST. In preparation for a semantic action related to an identifier or string, the initial information available is a pointer into SYMLIST. L82 keeps all entries in the symbol table which refer to the same symbols as linked lists.

Analogous to the L81 scheme, the pointer to the list associated with the symbol starting at SYMLIST(I) is stored at SYMLIST(I-2) and SYMLIST(I-1). Thus, when L82 receives a pointer to an identifier or string in SYMLIST, it has immediate access to all the corresponding entries in SYMTAB, without requiring a table look up. The links must be updated each time symbols are pushed onto, or popped from, the symbol table.

Since L80 relocatable modules are made up of three segments, L82 must maintain three location counters. Unlike the other two counters, the work area counter moves up and down with the program block structure. The work area is a virtual segment as far as the compiler is concerned. Thus, the only information generated by L82 regarding the work area is the total amount of memory it requires. This is done on a special record written at the end of the relocation table file, which contains the size of the three segments to be incorporated into the final executable object program.

C. MACRO PROCESSOR IMPLEMENTATION

1. Data structures

M81, the macro processor for the M80 language, is implemented as a single program which interprets and executes M80 statements. The following data structures are manipulated by M81:

- parser tables;

- parse stack;
- value stack;
- macro descriptor stack;
- macro text stack;
- hash table;
- status stack.

The purpose and contents of each one of these structures is summarized in the next paragraphs.

The parsing mechanism used in M81 is also of the LR(1) type. Thus, the parser tables and the parse stack are similar to those in the L81 module, described in Section IV.B.1. The only practical difference is that the tables for M81 are smaller, since the M80 grammar is considerably simpler than the grammar of the L80 language.

As mentioned in the beginning of this chapter, additional stacks, used in conjunction with semantic actions, are usually maintained in parallel with the parse stack. In the case of M81 only one of these stacks is required, called the value stack, which holds information associated with handles, such as pointers and temporary values used in expression evaluation. The entries in the parse stack and in the value stack are 16-bit words.

The M80 language implements both integer and textual macros, where textual macros may have any number of parameters. The information necessary to completely describe all the different macros defined in a source program is maintained in a stack, called the macro descriptor stack (MDS).

An entry in MDS corresponds to each existing macro, including temporary macros created by the macro processor during the evaluation of parameterized macro calls, as well as reserved words, which are considered predefined macros.

Since the amount of information required to describe macros is variable, the entries in MDS are variable length, where each entry is made up of an integral number of 16-bit words.

A character stack is used in conjunction with MDS, called the macro text stack (MTS), and is used for the storage of textual information such as macro names, macro bodies, and strings. The end of each piece of text in MTS is marked by an end-of-text symbol (a zero byte).

To conserve MTS space, entries are never repeated. That is, each different string in MTS appears exactly once, even though it may be used in several contexts with different meanings. Hash coding techniques are used to determine whether a new string is already present in MTS.

The internal representation of macros can now be described. Three basic patterns are adopted, corresponding to reserved words, textual macros, and integer macros.

Reserved word entries in MDS are made up of three cells:

- the reserved word internal number, to be used by the parser;
- a pointer to the entry in MDS corresponding to the previous macro with same hash code as the reserved word;

- a pointer to the actual reserved word name, stored in MTS.

For a textual macro, n+4 words are required in MDS, where n is the number of formal parameters of the macro:

- the number of formal parameters (8 bits) and the type of the macro (8 bits);
- a pointer to the entry in MDS corresponding to the previous macro with same hash code as the current macro name;
- a pointer to the macro body in MTS;
- n pointers to the names of the n formal parameters in MTS;
- a pointer to the macro name in MTS.

Integer macros are treated as a special case of textual macros. They never have formal parameters, and thus n=0. Instead of having macro bodies, they possess values. These values are stored in the MDS words that would contain, for textual macros, the pointers to the macro bodies in MTS. Therefore, each integer macro takes four MDS words.

The logical entries in MDS are organized as linked lists, where each list contains the definitions for all macros with a particular hash code, and the entry to each list is kept in a hash table. When new macro definitions are pushed into MDS, or existing macro descriptors are popped, these links must be updated. To avoid hash code recomputations, the hash code of each string or macro name in MTS is stored with the corresponding item. For a macro name starting at MTS(I), the hash code (which always evaluates to a single byte) is stored at MTS(I-1).

The last M&L structure to be described is the status

stack. This stack is used to save the current program status each time a macro call is detected. All vital information is first pushed onto the stack, and then the macro processor starts evaluating the invoked macro. Upon completion of the macro call expansion, the previous status is restored, and normal processing is resumed. This mechanism allows nested macro calls, similar to the procedure call scheme in L80. The following items are saved in the status stack:

- MTS top pointer;
- MDS top pointer;
- next character to be scanned;
- pointer to the next character.

The last two items are necessary since the macro processor may scan either the actual input program or some text stored in MTS, such as a macro body. Additional details are provided in the next section.

2. Lexical analysis

The lexical analyzer performs a number of primary scanning functions. One of these is to transmit to the parser only the tokens which actually constitute M80 statements, and to reproduce in the output all the other characters present in the input stream. Thus, it is responsible for the detection of the beginning and end of macro calls.

During the expansion of macros, the text being scanned is not the actual input, but rather the body of some macro in MTS. A pointer into MTS, called the next character

pointer (NCP), gives the location of the next character to be scanned. A convention is adopted that when NCP=0, the next character is to be obtained from the input program, rather than from MTS.

As in the L81 program, lexical analysis is performed by a procedure called LRLEX. This routine works in two different modes, called the "inside" and "outside" modes. As their names imply, these modes are used for scanning inside and outside macro calls.

While in outside mode, characters scanned are sent to the output file until an end-of-text character is detected, which LRLEX considers as the end of a macro body. The previous status is removed from the status stack, and LRLEX resumes scanning. If the status stack is empty, the macro body just scanned was the source program itself and processing terminates.

Another possibility while in outside mode is the detection of a left bracket, which marks the beginning of a macro call. This causes LRLEX to switch modes, and start scanning M80 statements.

When invoked by the parser, LRLEX returns a token such as an identifier, a reserved word, a number or a string. As in L81, LRLEX relies upon a lower level routine, GETTOKEN, which skips comments and collects token characters.

One of the particular features of M81's GETTOKEN is related to the identification of strings. In the M80

language strings may contain macro calls, which may contain strings, which may contain other macro calls, and so forth. The recognition of nested macro calls and embedded strings is facilitated by the fact that M80 statements are delimited by matching brackets.

LRLEX remains in the inside mode until a right bracket which is not part of a string is encountered, causing a change back to outside mode.

3. Syntactic analysis

One of the most important advantages of using a formal grammar to describe the M80 macro processing language is the reliability and generality obtained. Although very complicated statements can be constructed using the basic tools of the language, the correctness of their recognition is guaranteed by the fact that a mechanically generated parser is adopted. Ad hoc techniques would have been harder to implement and potentially less reliable, since special cases and unexpected situations would be much more likely to occur.

Since the M81 parser scheme is also LR(1), the parser tables formats and the parser tables interpreter routine are the same as in L81, and the descriptions in Section IV.B.3. apply. The similarity of the parsing methods in both programs contributes to the reliability of the system, and also facilitates maintenance and future modification.

Analogous to the L80 language, a production of the

form

<statement> ::= <error>

was included in the M80 grammar. Therefore, the error recovery procedures in both programs are similar.

4. Macro expansion

The basic semantic actions performed by M81 are related to expression evaluation and string replacement.

Expression evaluation is performed on the value stack, which is also used as a temporary area for pointers and other values transmitted by routines associated with most productions.

All semantic actions are triggered by the parser immediately before a reduction is performed. The values associated with handles are then manipulated to cause appropriate actions to be taken. For instance, when a parameterized macro call is parsed, the semantic routines create the temporary macros described in Chapter III. and then save the current status on the status stack. The next character pointer is set to the first character in the macro body of the macro being invoked, the scanner is switched to the outside mode, and scanning proceeds. The effect is to cause the body of the called macro to be reproduced in the output file until it is completely scanned or an embedded macro call is found. Detection of an embedded macro call will automatically switch the scanner to the inside mode, and the parser will eventually analyze the embedded call as an M80 statement. The net effect is that the original macro call

is replaced in the source program by a string corresponding to the expanded body of the macro invoked.

The same scheme is used for IF statements: the expression present in the IF statement is parsed and interpreted and its final value is obtained in the value stack. The semantic routine for the IF statement tests the value of the expression, saves the current status, and then directs the scanner (in outside mode) to the adequate string in MTS. If this string contains any macro calls, these will be parsed and interpreted in the usual way.

The implementation of calls with a variable number of parameters is straightforward. If the macro call has actual parameters, they are used as macro bodies in the definition of temporary macros. Descriptors for these macros are pushed onto the MDS stack, thereby becoming effective. When a call to a formal parameter is found in a macro body, the topmost definition for that macro name is used, thus mapping the formal parameter into the actual parameter. If no actual parameters are provided, calls inside a macro body will automatically refer to any existing topmost definitions.

The above method is general, and can provide interesting results: the type of the macro used to map a formal parameter may vary from call to call, and actual parameters may contain any macro calls.

It is important to notice that when temporary macros are created, no string movement occurs. Instead, extra pointers are pushed onto the MDS stack. This method is al-

ways used for identifiers or strings already present in MTS.

D. LINKAGE EDITOR IMPLEMENTATION

1. Data structures

According to the ML80 system outline presented in Section IV.A.2., the basic task of the linkage editor for L80 programs, called L83, is to generate executable object modules given a relocatable library containing the following types of files:

- code areas;
- initialized data areas;
- relocation tables;
- symbol lists.

The operator provides the name of a module which is to be considered as the main routine of the object program. The first step L83 must perform is to find all the external module names referenced by the main module, and then the names of other external modules called by the existing ones, and so forth, until all modules to be linked are known.

After this closure operation is completed, L83 has the information necessary for the final linking. The following data structures are used by L83:

- symbol list;
- module map;
- current segment area.

The symbol list (SYMLIST) is essentially the same as generated by L81. It is used by L83 in the search for

external references, and also to produce an optional symbol-address cross reference list.

The module map (MM) is a table containing relevant data about the modules to be linked. An entry in MM corresponds to each module. These entries are fixed length, and contain the following fields:

- module name pointer (MNAME, 16 bits), which is a pointer to the module's name in SYMLIST;
- code area base (MCA, 16 bits), containing the actual address of the first byte in the code area of the current module;
- code area size (NCA, 16 bits), which gives the length of the code area of the current module, in bytes;
- initialized data area base (MIDA, 16 bits), which holds the actual address of the first byte in the initialized data area of the current module;
- initialized data area size (NIDA, 16 bits), giving the length of the initialized data area of the current module, in bytes;
- work area base (MWA, 16 bits), which holds the address of the first byte of the work area of the current module;
- symbol list base (MSYMB, 16 bits), containing the address of the first byte of the symbol list of the current module;
- module name hash code (MHASH, 8 bits), which is the hash code of the name of the current module.

The current segment area is a temporary storage area where each segment is kept during the address relocation

phase. It is implemented as a byte vector, and occupies all remaining memory.

2. Load time facilities

As mentioned above, L83 automatically retrieves all the modules necessary to the generation of an object program. After these modules are determined, L83 successively updates the addresses in their code areas and initialized data areas, dumping each updated segment into a file which will constitute the memory image of the object program.

As a convenience to the operator, L83 assumes a default value for the load address of the executable program. However, this can be altered by the operator, allowing the final object program to be generated at any origin desired.

Another option provided is the generation of a reference list containing the name, type, address and source program line number for all the symbols in the modules being linked. This reference list may be displayed at the console, or generated in hard copy, for future reference.

L83 detects some error conditions at load time, such as references to nonexistent modules or too many modules to be linked.

E. AUXILIARY PROGRAMS

For the sake of completeness, this section describes some utility programs which were helpful in the implementation of the ML80 system. These programs will be useful in future modifications or enhancements of the system.

The first program is a parser tables formatting utility

(Y16). This program is written in C, and runs on the PDP-11/50. When Y16 is appended to the input grammar submitted to YACC, the output from YACC becomes a C program whose main routine will generate the parser tables and the list of the reserved words of the grammar as a PDP disk file. The tables in this file are in Intel's hexadecimal format, which is a convenient format for remote transmission, since it contains validity check fields.

The parser tables formatted by Y16 can then be sent to the 8080, and immediately used as input to the ML80 system. Both M81 and L81, in their developmental versions, are able to read these tables and perform the validity checking. This scheme allows easy modification of the ML80 parsers without the need of cumbersome table transcriptions.

An 8080/PDP-11 interface program, called P11, was developed to run on the 8080, under the CP/M system. Its fundamental purpose is to provide a communications link between an operator at the 8080 CRT and UNIX, which is the operating system on the PDP-11/50 installation. Both computers are physically connected through a 20 mA current loop interface such that each computer is treated by the other as a high-speed teletypewriter.

P11 is essentially a software switch which establishes different communications paths, by providing 3 modes of operation:

- neutral: CRT to PDP, PDP to CRT;
- reception: PDP to floppy disk, CRT to PDP;

- transmission: floppy disk to PDP, PDP to CRT.

In the neutral mode the CRT can be used as a regular UNIX terminal: characters typed at the CRT are sent to the PDP, and characters sent by the PDP are presented at the CRT.

In the reception mode, characters sent by the PDP are stored in a file in the 8080's floppy disk system. Characters typed at the CRT are transmitted to the PDP. In this mode files can be imported from UNIX.

The transmission mode provides the 8080 with the export capability: characters from a file in the floppy disk are transmitted to the PDP, and characters sent by the PDP are presented at the CRT. Characters typed at the CRT are buffered for later transmission to the PDP.

The operator has control over P11 at all times through the use of the following commands:

in any mode:

<control W> - inform current mode;

in neutral mode:

<control R> - enter reception mode;

<control T> - enter transmission mode;

<control C> - return to CP/M;

in transmission mode:

<control N> - abort transmission, enter neutral mode;

<control T> - inform number of bytes already transmitted;

in reception mode:

<control N> - abort reception, enter neutral mode;

<control R> - inform number of bytes already received.

P11 operates by keeping two buffers, one for the characters typed at the CRT, and the other for the characters sent by the PDP. Characters are arranged in each buffer as a FIFO queue. In the current version of P11 the CRT buffer is 200 bytes long and the PDP buffer is 9K bytes long, which is the remaining memory available in a 16K CP/M system.

When the CRT buffer is full and the operator tries to type more characters, P11 reacts by sending a beep character to the console. When the PDP buffer is full and the PDP tries to send more characters, P11 performs one of the following actions: if in reception mode, write a record (128 bytes) onto the floppy disk to make room in memory for the incoming characters. If in neutral or transmission modes: wait for room to be available. In both cases, since characters may have been missed, P11 sets a warning flag and reports the fact to the operator at the end of the session. It should be noted that the latter case is extremely unlikely to happen, due to the size of the PDP buffer. The former case, however, will certainly happen if the file being received is larger than 9K bytes. A solution to this problem is to have the PDP send large files in blocks of 9K or less bytes. This facility is provided by SEND and SENDH, two programs which run on the PDP-11 and which were written to work in conjunction with P11.

SEND sends blocks of about 6K bytes followed by a special end-of-block character. After the transmission of each

block SEND waits for an acknowledgement from P11. Upon detection of the end-of-block character, P11 writes the contents of the PDP buffer onto disk, informs the operator of the reception of a block, and then sends the acknowledge signal to the PDP. In this way, no characters are missed during the floppy disk I/O operations.

SENDH is a version of SEND which should be used when the file to be imported from the PDP happens to be the output of a PL/M compilation. SENDH acts like SEND, but it skips the symbol table which precedes the code records in the PL/M compiler output. This saves space on the floppy disk, and reduces file transfer time.

The main routine in P11 is a loop which performs a polling operation on four status bits:

- line from CRT ready;
- line to CRT ready;
- line from PDP ready;
- line to PDP ready.

Whenever one of these signals is true, P11 takes an action which depends on the current mode and the status of the buffers. For instance, if the mode is neutral and the line to the PDP is ready, then P11 looks at the CRT buffer; if it is not empty, the first character in the buffer is sent to the PDP.

It should be noted that the CRT and the PDP lines are handled directly by P11, that is, the standard I/O entry points of CP/M are not used. The devices must be attached

to the 8080 system as follows:

CRT status - input port 1;

CRT data in - input port 0;

CRT data out - output port 0;

PDP status - input port 5;

PDP data in - input port 4;

PDP data out - output port 4.

P11 uses the standard CP/M functions for opening, closing, creating, deleting, writing and reading files in the floppy disk.

V. CONCLUSIONS

The conclusions which can be derived from the material presented in the previous chapters are related to three basic perspectives on the ML80 language: its design, its implementation, and the user's point of view.

Language design is becoming an active and well-defined discipline in Computer Science. Microcomputer programming languages should certainly benefit from all the contributions which have been made in this area. In particular, it is important not to base one's decisions only on grounds of individual experience or personal taste: a vast literature is available on language design, with both theoretical and practical information.

The design of the ML80 language was deeply influenced by some existing languages: PL/M [8] contributed to the overall block structure, while the notation used for registers, the embedding of assignment statements in IF and WHILE statements, and the comparison operator were inspired by SMAL [17]. The use of compound conditions, and the feasibility of including procedures was suggested by PL360 [23]. Finally, double operators and the idea of assignment statements as the incremental part of DO-BY statements were borrowed from C [10].

The most important conclusion related to the implementation of the ML80 language processors is that the use of formal language and automatic parser generator methods is ex-

tremely valuable: the flexibility and reliability of this approach are considerable, and development time is substantially reduced, compared to *ad hoc* techniques. The use of a formal grammar contributes to precision and generality, and in fact seems to be the only acceptable way to define a modern programming language.

It should be mentioned that the language adopted to implement the ML80 processors, PL/M, encourages the formulation of programs in a structured manner. The use of some structured programming techniques in the development of the system certainly contributed to the relatively small number of errors (mostly transcription errors) found in the testing phase.

It remains to assess the convenience of the final product from the user's perspective. Although the language has not yet been tried in a production environment, the experience obtained with test programs seems to indicate that ML80 facilitates microcomputer programming to a degree not attained by standard assembly languages. Since ML80 programs are locally compiled, the turnaround time is quite acceptable, resulting in rapid program debugging and modification. The structured control constructs, along with the algebraic notation for data operations and the macro facilities, free the programmer from cumbersome labels and operation codes. The totally free format and the block structure, encouraging indentation and modularization, also contribute to more readable programs.

APPENDIX A. ML80 COMPILER OPERATION

1. M81 - MACRO PROCESSOR

CP/M command:	M81 <prog.name>	
Input file:	<prog.name>.M80	(source M80 program)
Output file:	<prog.name>.L80	(source L80 program)
Error messages:	error count displayed at the console	
	error messages written onto output file	

2. L81 - PARSER

CP/M command:	L81 <prog.name>	
Input file:	<prog.name>.L80	(source L80 program)
Output files:	<prog.name>.80P	(parser actions file)
	<prog.name>.80S	(symbol list file)
Error messages:	displayed at the console	

3. L82 - CODE GENERATOR

CP/M command:	L82 <prog.name>	
Input files:	<prog.name>.80P	(parser actions file)
	<prog.name>.80S	(symbol list file)
Output files:	<prog.name>.80C	(code area file)
	<prog.name>.80D	(data area file)
	<prog.name>.80R	(relocation table file)
Error messages:	displayed at the console	

4. L83 - LINKAGE EDITOR

CP/M command:	L83 <prog.name>	
Input files:	<prog.name>.80S	(symbol list file)
	<prog.name>.80C	(code area file)
	<prog.name>.80D	(data area file)
	<prog.name>.80R	(relocation table file)
Output file:	<prog.name>.COM	(executable program)
Error messages:	displayed at the console	
Options:	the CP/M command	
	L83 <prog.name>.X	
	allows modification of default options.	

APPENDIX B. ML80 COMPILER ERROR MESSAGES

1. M81 - MACRO PROCESSOR

1 - unexpected end of file

2 - mispelled number

3 - number too large

7 - syntax error (improperly formed statement)

11 - reference to undefined macro

12 - numeric macro called with parameters

13 - too many actual parameters in macro call

14 - assignment statement involving non-numeric macro

15 - expression evaluation involving non-numeric macro

F1 - macro descriptor stack overflow (too many macros)

F2 - status stack overflow (too many nested macro calls)

F3 - macro text stack overflow (strings too long)

F4 - parse stack overflow (too many nested statements)

F5 - parse stack underflow (M81 malfunction)

F7 - unrecoverable syntax error - parsing discontinued

2. L81 - PARSER

1 - unexpected end of file

2 - mispelled number

3 - number too large

7 - syntax error (improperly formed statement)

F4 - parse stack overflow (too many nested statements)

F5 - parse stack underflow (L81 malfunction)

F6 - symbol list overflow (too many identifiers and strings)

F7 - unrecoverable syntax error - parsing discontinued

3. L82 - CODE GENERATOR

00 - INITIAL data too long (truncated at left)

01 - identifier redeclared

02 - identifier redeclared (ignored)

03 - invalid procedure name

04 - reference to undeclared identifier

05 - invalid number of parameters

06 - invalid call

07 - not a machine operation

08 - feature not implemented

0A - invalid constant

0B - invalid goto destination

0E - reference to undefined address

F1 - parse stack overflow (too many nested statements)

F2 - symbol table overflow (too many symbols per block)

F3 - block level stack overflow (too many nested blocks)

F4 - relocation table overflow (too many unresolved
addresses - may be caused by long case statements
or complex compound conditions)

F5 - parser actions file in error

F6 - symbol list overflow (too many identifiers and strings)

4. L83 - LINKAGE EDITOR

1 - too many modules to be linked

2 - memory overflow (segments too long)

3 - invalid record in the relocation table file

```
/*
              ********************************************
              *                                          *
              *  M81:  INTERPRETER FOR THE M80 LANGUAGE  *
              *                                          *
              *        LUIZ PEDROSO - OCTOBER 1975       *
              *                                          *
              ********************************************
*/

    DECLARE LIT             LITERALLY 'LITERALLY';
    DECLARE FOREVER         LITERALLY 'WHILE 1';
    DECLARE CR              LITERALLY '0DH';    /* CARRIAGE-RETURN */
    DECLARE LF              LITERALLY '0AH';    /* LINE-FEED */
    DECLARE CONTROLZ        LITERALLY '1AH';    /* END OF FILE */

    /* YACC-ASSIGNED TERMINAL NUMBERS */
    DECLARE YERROR          LITERALLY '256';
    DECLARE IDENTIFIER      LITERALLY '257';
    DECLARE NUMBER          LITERALLY '258';
    DECLARE STRING          LITERALLY '259';

    /* CP/M SYSTEM CONSTANTS */
    DECLARE CPM             LITERALLY '0';      /* CP/M REBOOT ENTRY */
    DECLARE IFCBA           LITERALLY '005CH'; /* INPUT FCB ADDRESS */
    DECLARE SBUFA           LITERALLY '0080H'; /* SYSTEM BUFFER ADDRESS */
    DECLARE FBASE           LITERALLY '3200H'; /* FDOS BASE */
    DECLARE BDOS            LITERALLY '3FFDH'; /* BASIC DOS ENTRY */

    /* I/O PRIMITIVES */
    DECLARE PRINTCHAR       LITERALLY  '2';
    DECLARE PRINT           LITERALLY  '9';
    DECLARE OPEN            LITERALLY  '15';
    DECLARE CLOSE           LITERALLY  '16';
    DECLARE MAKE            LITERALLY  '22';
    DECLARE DELETE          LITERALLY  '19';
    DECLARE READBF          LITERALLY  '20';
    DECLARE WRITEBF         LITERALLY  '21';
    DECLARE INITDSK         LITERALLY  '13';
    DECLARE SETBUF          LITERALLY  '26';

    /* FILE CONTROL BLOCKS */
    DECLARE TABFCB(33) BYTE  /* PARSER TABLES FILE */
          INITIAL(0,'M81     ','TAB',0,0,0,0);
    DECLARE OUTFCB(33) BYTE /* OUTPUT FILE */
          INITIAL(0,'        ','L80',0,0,0,0);
    DECLARE IFA ADDRESS INITIAL (IFCBA);        /* FCB ADDRESS */
    DECLARE IFCB BASED IFA BYTE;  /* INPUT FILE CONTROL BLOCK */

    /* OUTPUT BUFFER */
    DECLARE OUTBUF(128) BYTE;
    DECLARE OBP BYTE INITIAL (0);      /* OUTPUT BUFFER POINTER */

    /* INPUT BUFFER */
    DECLARE IBUF(128) BYTE;        /* INPUT BUFFER */
    DECLARE IBP BYTE INITIAL(128);  /* INPUT BUFFER POINTER */

    /* PARSER TABLES MASKS */
    DECLARE ACTNASK         LITERALLY '0F000H';
    DECLARE SYMBMASK        LITERALLY '01000H';
    DECLARE SHIFTMASK       LITERALLY '02000H';
    DECLARE REDMASK         LITERALLY '03000H';
    DECLARE ERRORMASK       LITERALLY '00000H';
    DECLARE DEFAULT         LITERALLY '0FFFFH';

    /* PARSER GLOBAL VARIABLES */
    DECLARE STACKSIZE LITERALLY '30';
    DECLARE PSTACK(STACKSIZE) ADDRESS INITIAL (0);  /* PARSE STACK */
    DECLARE VSTACK(STACKSIZE) ADDRESS INITIAL (0);  /* VALUE STACK */
    DECLARE PTOP BYTE     INITIAL(0);     /* PARSE STACK POINTER */
    DECLARE PARSING       BYTE INITIAL (1);
```

```
DECLARE RECOVERING   BYTE INITIAL (0);
DECLARE SEARCHING    BYTE;
DECLARE (INPUTSY,TAB,ACTION) ADDRESS;
DECLARE IT BYTE;
DECLARE ERRORCOUNT ADDRESS INITIAL (0);

/* PARSER TABLES */
DECLARE ACTB  ADDRESS, LRACT  BASED ACTB  ADDRESS;
DECLARE PACTB ADDRESS, LRPACT BASED PACTB ADDRESS;
DECLARE R1B   ADDRESS, LRR1   BASED R1B   ADDRESS;
DECLARE R2B   ADDRESS, LRR2   BASED R2B   ADDRESS;
DECLARE GOB   ADDRESS, LRGO   BASED GOB   ADDRESS;
DECLARE PGOB  ADDRESS, LRPGO  BASED PGOB  ADDRESS;

/* HASH TABLE */
DECLARE HASHTAB(128) ADDRESS;
DECLARE HASHMASK LITERALLY '0177Q';
DECLARE HASHCODE BYTE;

/* LEXICAL ANALYZER GLOBAL VARIABLES */
DECLARE TOKBUF(30) BYTE;
DECLARE TOKBUFTOP LITERALLY 'LAST(TOKBUF)';
DECLARE TOKVAL ADDRESS;
DECLARE TOKTYPE ADDRESS;
DECLARE TOKCONT BYTE INITIAL (0);
DECLARE TOKERROR BYTE;
DECLARE TOKTOP LITERALLY 'TOKBUF';
DECLARE IC BYTE INITIAL (' ');   /* INPUT CHARACTER */
DECLARE NC BYTE INITIAL (' ');   /* NEXT CHARACTER */
DECLARE NCP ADDRESS INITIAL(0);  /* POINTER TO NC IN MTS OR INPUT */
DECLARE LRLVAL ADDRESS;
DECLARE LINE   ADDRESS INITIAL (1);
DECLARE SPECIALC         LITERALLY '1';
DECLARE EOFILE           LITERALLY '0';
DECLARE OUTSIDE BYTE INITIAL (1); /* 0 WHEN SCANNING INSIDE MACRO CALLS */
DECLARE MBEGIN LIT '5BH';     /* LEFT DELIMITER FOR MACRO CALLS */
DECLARE MEND  LIT '5DH';      /* RIGHT DELIMITER FOR MACROCALLS */
DECLARE COUNT BYTE; /* NO. UNMATCHED MBEGIN'S INSIDE STRING */

/* MACRO TEXT STACK */
DECLARE MTSB ADDRESS, MTS BASED MTSB BYTE;
DECLARE MTSSIZE ADDRESS;      /* SIZE OF MTS */
DECLARE TTOP ADDRESS;         /* TOP OF MTS */

/* MACRO DESCRIPTOR STACK */
DECLARE MDSSIZE LIT '150';
DECLARE MDS(MDSSIZE) ADDRESS;
DECLARE DTOP ADDRESS INITIAL(0FFFFH);  /* TOP OF MDS */
DECLARE RTOP ADDRESS;   /* POINTER TO TOPMOST RESERVED MACRO IN MDS */

/* MACRO TYPES */
DECLARE MNUM  LIT '1';   /* NUMERIC */
DECLARE MMAC  LIT '2';   /* TEXTUAL */

/* STATUS STACK */
DECLARE STSSIZE LIT '30';
DECLARE GTSAVE(STSSIZE) ADDRESS;      /*  TTOP SAVE VALUE */
DECLARE GDSAVE(STSSIZE) ADDRESS;      /*  DTOP SAVE VALUE */
DECLARE GCSAVE(STSSIZE) BYTE;         /*  NC   SAVE VALUE */
DECLARE GNSAVE(STSSIZE) ADDRESS;      /*  NCP  SAVE VALUE */
DECLARE GTOP BYTE INITIAL (255);      /* STATUS STACK TOP POINTER */

/* VARIABLES ASSOCIATED WITH LEFT HAND SIDE OF PRODUCTIONS */
DECLARE HH ADDRESS;

/* VARIABLES ASSOCIATED WITH HANDLES */
DECLARE (H1,H2,H3,H4,H5,H6) ADDRESS;
DECLARE L1 LIT 'LOW(H1)',
        L2 LIT 'LOW(H2)',
        L3 LIT 'LOW(H3)',
        L4 LIT 'LOW(H4)',
        L5 LIT 'LOW(H5)',
        L6 LIT 'LOW(H6)';
```

```
/* PARSER TABLES LOADER GLOBAL VARIABLES */
DECLARE (CS,RL) BYTE;          /* CHECKSUM, RECORD LENGTH */
DECLARE MEMPTR ADDRESS;        /*  MEMORY POINTER */

/* GLOBAL VARIABLES FOR SEMANTIC ACTIONS */
DECLARE HSAVE BYTE;            /* HASHCODE SAVE VALUE */
DECLARE DSAVE ADDRESS;         /* SAVE VALUE FOR DTOP */
DECLARE TSAVE ADDRESS;         /* SAVE VALUE FOR TTOP */
DECLARE TTSAVE ADDRESS;        /* YET ANOTHER SAVE VALUE FOR TTOP */
DECLARE NF BYTE;               /* NO. FORMAL PARAMETERS */
DECLARE NA BYTE;               /* NO. ACTUAL PARAMETERS */
DECLARE H BYTE, (I,J) ADDRESS;      /* TEMPORARIES */

/* MNEMONICS FOR RELATIONS */
DECLARE EQ LIT '1';
DECLARE LT LIT '2';
DECLARE GT LIT '3';
DECLARE NE LIT '4';
DECLARE LE LIT '5';
DECLARE GE LIT '6';

/* OUTPUT OPTIONS */
DECLARE CRT LIT '1';
DECLARE DISK LIT '2';
DECLARE OUTDEV BYTE INITIAL(DISK);

/* SWITCHES FOR BUILT-IN TRACE */
/* TO INCLUDE TRACE ROUTINES, SET TRACESON TO BLANKS */
/* TO EXCLUDE TRACE ROUTINES, SET TRACESON TO SLASH-STAR */
DECLARE TRACESON  LIT ' ';
DECLARE TRACESOFF LIT ' /* ';

TRACESON
DECLARE (TRACING,Z) BYTE;
TRACESOFF *** */


NEG: PROCEDURE(A) BYTE;
      DECLARE A ADDRESS;
      /* TRUE IF A IS NEGATIVE */
      RETURN SHR(A,15);
      END NEG;

VMON: PROCEDURE(FUNC,INFO) BYTE;
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; RETURN A VALUE */
      GO TO BDOS;
      END VMON;

CMON: PROCEDURE(FUNC,INFO);
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; NO VALUE RETURNED */
      GO TO BDOS;
      END CMON;

MOVBUF: PROCEDURE(D,S);
      DECLARE(D,S) ADDRESS;
      /* MOVE 128 BYTES FROM S TO D */
      DECLARE DB BASED D BYTE;
      DECLARE SB BASED S BYTE;
      DECLARE I BYTE;
      DO I = 0 TO 127;
         DB(I) = SB(I);
         SB(I) = ' ';
      END;
      END MOVBUF;

FLUSH: PROCEDURE;
      /* FLUSH OUTPUT BUFFER */
      IF OBP=0  /* BUFFER EMPTY */ THEN RETURN;
      CALL MOVBUF(SBUFA,.OUTBUF);  /* MOVE OUTPUT BUFFER TO I/O AREA */
      IF VMON(WRITEBF,.OUTFCB)<>0 THEN /* UNSUCCESSFUL WRITE */
```

```
            DO;    /* WARN OPERATOR AND QUIT */
               CALL CMON(PRINT,.' WRITE ERROR, ABEND M81.$');
               GO TO CPM;
            END;
         OBP = 0;    /* BUFFER IS EMPTY */
         END FLUSH;

PUTC:    PROCEDURE(C);
         DECLARE C BYTE;
         /* WRITE CHARACTER C ON OUTPUT FILE */
         IF OBP = 128 THEN /* BUFFER FULL */ CALL FLUSH;
         OUTBUF(OBP) = C;
         OBP = OBP + 1;
         END PUTC;

GETC:    PROCEDURE(F) BYTE;
         DECLARE F ADDRESS;
         /* READ NEXT CHARACTER FROM INPUT BUFFER;  IF BUFFER EMPTY
            THEN GET ANOTHER RECORD FROM FILE WHOSE FCB IS AT F;
            SET INPUT BUFFER POINTER TO 0 IF EOFILE, 1-128 OTHERWISE */
         DECLARE C BYTE;
         IF IBP = 128 THEN /* BUFFER EMPTY */
            DO;  /* READ NEXT RECORD INTO BUFFER */
               IBP=0;
               IF VMON(READEF,F)<>0 THEN /* EOFILE */ RETURN CONTROLZ;
               CALL MOVBUF(.IBUF,SBUFA);    /* SET INPUT BUFFER */
            END;
         C=IBUF(IBP);  /* NEXT CHARACTER */
         IBP = IBP + 1;
         IF C=LF THEN /* NEW LINE */ LINE = LINE + 1;
         RETURN C;
         END GETC;

PRC:  PROCEDURE (C);
         DECLARE C BYTE;
         /* PRINT CHARACTER C ON THE CRT, OR ON THE OUTPUT FILE */
         IF OUTDEV=CRT THEN CALL CMON(PRINTCHAR,C);
         ELSE CALL PUTC(C);
         END PRC;

CRLF:    PROCEDURE;
         /* CARRIAGE RETURN, LINE FEED */
         CALL PRC(CR);    CALL PRC(LF);
         END CRLF;

PRS:  PROCEDURE (A);
         DECLARE A ADDRESS;
         /* PRINT STRING AT A UNTIL A $ IS FOUND */
         DECLARE B BASED A BYTE;
         DO WHILE B<>'$';
            CALL PRC(B);
            A = A + 1;
         END;
         END PRS;

PRL:  PROCEDURE(A);
         DECLARE A ADDRESS;
         /* PRINT LINE STARTING AT A UNTIL A $ IS FOUND */
         CALL CRLF;
         CALL PRS(A);
         END PRL;

SETCRT:  PROCEDURE;
         /* ROUTE OUTPUT TO THE CRT */
         OUTDEV = CRT;
         END SETCRT;

EXIT:    PROCEDURE;
         /* RETURN TO CP/M */
         CALL SETCRT;
         CALL PRL(.'END M81 $');
         CALL CRLF;
         GOTO CPM;
```

```
      END EXIT;

PRFN: PROCEDURE(F);
      DECLARE F ADDRESS;
      /* PRINT NAME OF FILE WITH FCB AT ADDRESS F */
      DECLARE FCB BASED F BYTE, I BYTE;
      DO I = 1 TO 11;
         IF FCB(I) <> ' ' THEN CALL PRC(FCB(I));
         IF I=8 THEN CALL PRC('.');
      END;
      END PRFN;

CANTOP: PROCEDURE(F);
      DECLARE F ADDRESS;  /* ADDRESS OF AN FCB */
      /* PRINT OPEN ERROR MESSAGE AND QUIT */
      CALL SETCRT;
      CALL PRL('.'CANNOT OPEN $');
      CALL PRFN(F);
      CALL EXIT;
      END CANTOP;

PRINTH: PROCEDURE(H);
      DECLARE H BYTE;
      /* PRINT A HEXADECIMAL CHARACTER */
      IF (H>9) THEN CALL PRC(H-10+'A');
      ELSE CALL PRC(H+'0');
      END PRINTH;

PRINTHB: PROCEDURE(B);
      DECLARE B BYTE;
      /* PRINT 2 HEXADECIMAL CHARACTERS REPRESENTING B */
      CALL PRINTH(SHR(B,4));
      CALL PRINTH(B AND 0FH);
      END PRINTHB;

PRINTHA: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT 4 HEXADECIMAL CHARACTERS REPRESENTING A */
      CALL PRC('0');
      CALL PRINTHB(HIGH(A));
      CALL PRINTHB(LOW(A));
      CALL PRC('H');
      END PRINTHA;

PRINTD: PROCEDURE(V);
      DECLARE V ADDRESS;
      /* PRINT V IN DECIMAL FORMAT, SUPPRESS LEADING ZEROS */
      DECLARE Q ADDRESS, (D,PZ) BYTE;
      IF NEG(V) THEN
         DO;
            CALL PRC('-');
            V = - V;
         END;
      Q = 10000;
      PZ = 0;  /* DO NOT PRINT ZEROS */
      DO WHILE Q > 1;
         D = V / Q;
         IF (D>0) OR PZ THEN  /* PRINT */
            DO; CALL PRINTH(D);  PZ = 1;
            END;
         V = V MOD Q;
         Q = Q / 10;
      END;
      CALL PRINTH(V);
      END PRINTD;

PRINTQ: PROCEDURE(V);
      DECLARE V ADDRESS;
      /* PRINT V IN OCTAL FORMAT */
      DECLARE N BYTE;
      N = 15;
      DO WHILE N<>0;
         CALL PRINTH( SHR(V,N) AND 0007Q );
```

```
            N = N - 3;
        END;
        CALL PRINTH( V AND 0007Q );
        CALL PRC('Q');
        END PRINTQ;

INITFILES: PROCEDURE;
        /* INITIALIZE FILES */
        DECLARE I BYTE;
        CALL CMON(INITDSK,0);  /* SELECT DISK */
        IF IFCB(9)=' ' THEN
           DO;  /* SET TYPE OF INPUT FILE TO 'M80' */
              IFCB(9)='M';
              IFCB(10)='8';
              IFCB(11)='0';
           END;
        /* OPEN INPUT FILE */
        IF VMON(OPEN,IFCBA) = 255 THEN CALL CANTOP(IFCBA);
        /* SET NAME OF OUTPUT FILE */
        DO I=1 TO 8;
           OUTFCB(I) = IFCB(I);
        END;
        /* DELETE OLD VERSION OF OUTPUT FILE */
        CALL CMON(DELETE,.OUTFCB);
        /* CREATE NEW VERSION OF OUTPUT FILE */
        IF VMON(MAKE,.OUTFCB) = 255 THEN CALL CANTOP(.OUTFCB);
        /* SET NEXT RECORD TO 0 */
        IFCB(32), OUTFCB(32) = 0;
        /* SET NEXT CHARACTER POINTER */
        IBP=128;  /* INPUT BUFFER IS EMPTY */
        LINE = 1;
        END INITFILES;

TERMINATE: PROCEDURE;
        /* CLOSE OUTPUT FILE, PRINT ERROR COUNT, AND QUIT */
        CALL FLUSH;   /* FLUSH OUTPUT BUFFER */
        CALL SETCRT;  /* OUTPUT TO CRT */
        IF VMON(CLOSE,.OUTFCB) = 255 THEN
           DO;
              CALL PRS(.'CANNOT CLOSE $');
              CALL PRFN(.OUTFCB);
           END;
        CALL PRS(.'ERRORS: $');
        CALL PRINTD(ERRORCOUNT);
        CALL EXIT;
        END TERMINATE;

ERROR: PROCEDURE(N);
        DECLARE N BYTE;
        /* PRINT ERROR MESSAGE N */
        ERRORCOUNT = ERRORCOUNT + 1;
        CALL PRS(.'*** LINE $');
        CALL PRINTD(LINE);
        CALL PRS(.': ERROR $');
        CALL PRINTHB(N);
        IF (N AND 0F0H) = 0F0H THEN  /* FATAL ERROR */
           CALL TERMINATE;
        END ERROR;

BLANK: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS BLANK */
        RETURN  (C=' ') OR (C=CR) OR (C=LF);
        END BLANK;

NUMERIC: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS NUMERIC */
        RETURN (C>='0') AND (C<='9');
        END NUMERIC;

ALPHABETIC: PROCEDURE (C) BYTE;
        DECLARE C BYTE;
```

```
      /* TRUE IF C IS ALPHABETIC */
      RETURN ( (C>='A') AND (C<='Z') ) OR
         ( (C>=61H) AND (C<=7AH) ) OR (C = '$' );
      END ALPHABETIC;

ALPHANUMERIC: PROCEDURE(C) BYTE;
      DECLARE C BYTE;
      /* TRUE IF C IS ALPHANUMERIC */
      RETURN ALPHABETIC(C) OR NUMERIC(C);
      END ALPHANUMERIC;

HEX: PROCEDURE(C) BYTE;
      DECLARE C BYTE;
      /* TRUE IF C IS HEXADECIMAL */
      RETURN NUMERIC(C) OR ( (C>='A') AND (C<='F') );
      END HEX;

HEXVAL: PROCEDURE(C) BYTE;
      DECLARE C BYTE;
      /* RETURN THE VALUE OF HEX CHARACTER C */
      IF NUMERIC(C) THEN RETURN C-'0';
      RETURN C-'A'+10;
      END HEXVAL;

GETIC: PROCEDURE;
      /* GET ONE CHARACTER FROM SOURCE PROGRAM INTO IC;
         LOOK AHEAD AT NEXT CHARACTER (NC);
         END OF FILE CONDITIONS ARE DENOTED (AT THE EXIT OF GETIC) BY:
         IC=0,    NC=0:    EOF;
         IC<>0,   NC=0:    ALMOST EOF;
         IC<>0,   NC<>0:   NOT EOF     */
      IC = NC;
      /* NOW GET THE NEW NC */
      IF NC = 0 THEN RETURN;
      IF NCP = 0 THEN  /* GET IT FROM SOURCE FILE */
         DO;
            NC = GETC(IFCBA);
      IF NC=CONTROLZ     /* EOF */ THEN NC=0;
         END;
      ELSE  /* GET NC FROM MTS */
         DO;
            IF (NCP:=NCP+1)>= MTSSIZE THEN  /* MTS OVERFLOW */
               CALL ERROR (0F3H);  /* QUIT */
            NC = MTS(NCP);
         END;
      END GETIC;

PUTIC: PROCEDURE;
      /* PUT INPUT CHARACTER IC INTO TOKBUF */
      TOKBUF(TOKTOP:=TOKTOP+1) = IC;  /* TOKBUF(0) IS NOT USED */
      END PUTIC;

PGIC: PROCEDURE;
      CALL PUTIC;  CALL GETIC;
      END PGIC;

HASHF: PROCEDURE(C);
      DECLARE C BYTE;
      /* SET GLOBAL VARIABLE HASHCODE, USING THE INPUT CHARACTER C */
      HASHCODE = (HASHCODE + C) AND HASHMASK;
      END HASHF;

LOADING: PROCEDURE BYTE;
      /* RETURN 1 IF IT IS OK TO KEEP ON LOADING TOKBUF */
      RETURN   TOKCONT     AND (TOKTOP < TOKBUFTOP);
      END LOADING;

GETID: PROCEDURE;
      /* LOAD AN IDENTIFIER INTO TOKBUF */
      TOKTYPE = IDENTIFIER;
      DO WHILE LOADING;
         CALL HASHF(IC);
         CALL PGIC;
```

```
      TOKCONT = ALPHANUMERIC(IC); /* CONTINUE IF ALPHANUMERIC */
   END;
   END GETID;

GETSTR: PROCEDURE;
   /* LOAD A STRING INTO TOKBUF */
   TOKTYPE = STRING;
   DO WHILE LOADING;
      IF IC <> '''' THEN
         DO;
            IF IC=MBEGIN THEN COUNT=COUNT+1;
            IF IC=MEND   THEN COUNT=COUNT-1;
            CALL PGIC;
            IF     IC=0      THEN /* UNEXPECTED EOF */
               DO;
                  TOKERROR = 1;
                  TOKCONT = 0;
                  RETURN;
               END;
         END;
      ELSE  /* FOUND A QUOTE */
      IF COUNT=0 THEN /* NOT INSIDE A MACROCALL */
         DO;
            CALL GETIC;  /* SKIP THE QUOTE */
            IF IC <> '''' THEN /* END OF STRING */ TOKCONT = 0;
            ELSE /* QUOTED QUOTE */  CALL PGIC;
         END;
      ELSE /* INSIDE MACROCALL */  CALL PGIC;    /* CONTINUE */
   END;
   END GETSTR;

NHNC: PROCEDURE BYTE;
   /* TRUE IF NC IS NEITHER HEX NOR 'H' */
   RETURN NOT HEX(NC) AND (NC<>'H');
   END NHNC;

GETNUM:PROCEDURE;
   /* LOAD A NUMBER INTO TOKBUF; PUT ITS VALUE INTO TOKVAL */
   DECLARE  BASE BYTE;
   BASE = 0;
   TOKTYPE = NUMBER;
   DO WHILE LOADING;
      CALL PGIC;
      IF (IC='O') OR (IC='Q') THEN BASE = 8; ELSE
      IF IC='H' THEN BASE = 16;  ELSE
      IF IC='B' AND NHNC THEN BASE=2; ELSE
      IF IC='D' AND NHNC THEN BASE=10; ELSE
      IF NOT HEX(IC)      THEN BASE = 1;
      IF BASE >1 THEN CALL GETIC; /* SKIP 'BASE' CHARACTER */ ELSE
      IF BASE = 1 THEN BASE = 10;
      TOKCONT = (BASE=0);  /* CONTINUE UNTIL BASE IS DETERMINED */
   END;
   /* COMPUTE THE VALUE OF THE NUMBER */
   DECLARE D BYTE;
   DECLARE MPLIER BYTE, MPLAND ADDRESS;
   DO I = 1 TO TOKTOP;
      D = HEXVAL(TOKBUF(I));  /* CURRENT DIGIT */
      IF D >= BASE THEN TOKERROR = 2;
      /* TOKVAL = TOKVAL * BASE + D */
      MPLIER = BASE;
      MPLAND = TOKVAL;
      TOKVAL = D;     /* TOKVAL GETS THE PARTIAL SUMS */
      DO WHILE MPLIER <> 0;
         IF MPLIER THEN
            DO;
               TOKVAL = TOKVAL + MPLAND;
               IF CARRY THEN TOKERROR = 3;
            END;
         MPLIER = SHR(MPLIER,1);
         MPLAND = SHL(MPLAND,1);
      END;
   END;
   END GETNUM;
```

```
GETSPECIAL: PROCEDURE;
      /* LOAD A SPECIAL CHARACTER INTO TOKBUF */
      TOKTYPE = SPECIALC;
      TOKVAL = 0;
      IF IC = 5CH AND NC=3CH  /* XOR */  THEN TOKVAL=260;  ELSE
      IF IC='<' AND NC='>'  /* NE */  THEN TOKVAL=261;  ELSE
      IF IC='<' AND NC='='  /* LE */  THEN TOKVAL=262;  ELSE
      IF IC=':' AND NC='=' THEN /* ASSIGN */ TOKVAL = 5FH;  ELSE
      IF IC='>' AND NC='='  /* GE */ THEN TOKVAL=263;
      IF TOKVAL<>0 THEN CALL PGIC;  /* SKIP ONE EXTRA CHARACTER */
      ELSE TOKVAL = IC;  /* SINGLE SPECIAL CHARACTER */
      IF IC=MEND THEN
         DO;  /* DO NOT READ NEXT CHAR OUTSIDE MACROCALL */
            CALL PUTIC;
            OUTSIDE = 1;
         END;
      ELSE CALL PGIC;    /* READ NEXT CHARACTER */
      TOKCONT = 0;
      END GETSPECIAL;

GETTOKEN: PROCEDURE;
      /* GET A TOKEN, AND RETURN THE GLOBAL VARIABLES:
         TOKTYPE:   IDENTIFIER, STRING, NUMBER, SPECIALC, EOFILE;
         TOKVAL:    VALUE OF NUMBER, INTERNAL NO. OF SPECIALC;
         TOKERROR: 0 IF VALID TOKEN, A MESSAGE NO. OTHERWISE;
         TOKBUF:    ARRAY OF CHARACTERS (1 TO TOKTOP) WITH TOKEN NAME;
         TOKTOP:    INDEX OF LAST CHARACTER IN TOKBUF;
         HASHCODE: HASHCODE  OF IDENTIFIERS, STRINGS */
       /* IC: INPUT CHARACTER (ALREADY READ IN, BUT NOT YET USED;
          NC: NEXT CHARACTER  (LOOK-AHEAD CHARACTER)  */
      DECLARE SOMETHING BYTE;
      TOKTOP, TOKERROR = 0;
      IF TOKCONT = 0 THEN /* GET A NEW TOKEN */
         DO;
            TOKVAL, HASHCODE, SOMETHING = 0;
      COUNT = 0;
            TOKCONT = 1;
            DO WHILE SOMETHING = 0;  /* LOOK FOR SOMETHING */
               DO WHILE BLANK(IC); CALL GETIC; END; /* SKIP BLANKS */
               IF (IC='/') AND (NC='*') THEN  /* A COMMENT */
                  DO;
                     CALL GETIC; CALL GETIC; /* SKIP SLASH-STAR */
                     DO WHILE (IC<>'*') OR (NC<>'/');
                        IF    IC=0   THEN /* UNFINISHED COMMENT */
                           DO; TOKTYPE=EOFILE; RETURN; END;
                        ELSE CALL GETIC;  /* SKIP A CHARACTER */
                     END;
                     CALL GETIC; CALL GETIC; /* SKIP STAR-SLASH */
                  END;  /* CO LOOK FOR SOMETHING */
               ELSE SOMETHING = 1;  /* FOUND SOMETHING */
            END;
            IF    IC=0   THEN     TOKTYPE = EOFILE; ELSE
            IF ALPHABETIC(IC) THEN CALL GETID; ELSE
            IF NUMERIC(IC) THEN CALL GETNUM; ELSE
            IF IC='''' THEN DO; CALL GETIC; CALL GETSTR; END; ELSE
            CALL GETSPECIAL;
         END;
      ELSE /* GET REMAINDER OF PRESENT TOKEN */
         DO;
            IF TOKTYPE = IDENTIFIER THEN CALL GETID; ELSE
            IF TOKTYPE = STRING     THEN CALL GETSTR; ELSE
            IF TOKTYPE = NUMBER     THEN CALL GETNUM; ELSE
            CALL GETSPECIAL;
         END;
      END GETTOKEN;

      TRACEOON
ESP: PROCEDURE; CALL PRC(' ');
      END ESP;

VG: PROCEDURE; CALL PRC(',');
      END VG;
```

```
TRAC: PROCEDURE(I);
      DECLARE I BYTE, K ADDRESS;
      IF TRACING<>'Y' THEN RETURN;
      CALL SETCRT;  CALL PRC(I);    Z=VMON(1,0);
      DO WHILE Z<>' ';
         CALL CRLF;
         IF Z='C' THEN DO;
            CALL PRC(IC);  CALL PRC(NC);
            CALL VC;  CALL PRINTD(NCP);
         END;
         IF Z='N' THEN DO;
            CALL ESP; CALL PRINTD(H1); CALL VC; CALL PRINTD(H2);
            CALL VC; CALL PRINTD(H3); CALL VC; CALL PRINTD(H4);
         END;
         IF Z='H' THEN
         DO I=0 TO LAST(HASHTAB);
            IF HASHTAB(I)<>0 THEN DO;
               CALL ESP; CALL PRINTD(I);
               CALL VC; CALL PRINTD(HASHTAB(I));
            END;
         END;
         IF Z='D' THEN DO I=0 TO DTOP;
            CALL ESP; CALL PRINTD(I);
            CALL VC; CALL PRINTD(MDS(I));
         END;
         IF Z='Q' THEN DO I=0 TO QTOP;
            CALL ESP; CALL PRINTD(I); CALL VC; CALL PRINTD(QTSAVE(I));
            CALL ESP; CALL PRINTD(QDSAVE(I)); CALL ESP;
            CALL PRC(QCSAVE(I)); CALL ESP; CALL PRINTD(QNSAVE(I));
         END;
         IF Z='P' THEN DO I=0 TO PTOP;
            CALL ESP; CALL PRINTD(PSTACK(I));
            CALL VC; CALL PRINTD(VSTACK(I));
         END;
         IF Z='T' THEN DO K=0 TO TTOP;
            CALL ESP;  Z=MTS(K);
            IF ALPHANUMERIC(Z) THEN CALL PRC(Z); ELSE CALL PRINTHB(Z);
         END;
         Z=VMON(1,0);
      END;
      OUTDEV=DISK;
      END TRAC;
      TRACESOFF *** */

COMPAR: PROCEDURE (A1,A2) BYTE;
      DECLARE (A1,A2) ADDRESS;
      /* RETURN 1 IF STRINGS AT A1, A2 ARE EQUAL, 0 OTHERWISE */
      DECLARE B1 BASED A1 BYTE;
      DECLARE B2 BASED A2 BYTE;
      DO WHILE B1 = B2;
         IF B1 = 00H /* (END OF STRING) */ THEN /* MATCH */ RETURN 1;
         A1 = A1 + 1;    A2 = A2 + 1;
      END;
      RETURN 0;
      END COMPAR;

PUSHD: PROCEDURE(I);
      DECLARE I ADDRESS;
      /* PUSH I INTO THE MACRO DESCRIPTOR STACK */
      IF (DTOP:=DTOP+1)>= MDSSIZE THEN
         /* MDS OVERFLOW */  CALL ERROR(0F1H);
      ELSE MDS(DTOP) = I;
      END PUSHD;

MTYPE: PROCEDURE(I) BYTE;
      DECLARE I ADDRESS;
      /* TYPE OF MACRO I */
      RETURN MDS(I) AND 00FFH;
      END MTYPE;

NFP: PROCEDURE(I) BYTE;
      DECLARE I ADDRESS;
```

```
      /* NO. OF FORMAL PARAMETERS OF MACRO I */
      RETURN SHR(MDS(I),8);
      END NFP;

TNAME: PROCEDURE(I) ADDRESS;
      DECLARE I ADDRESS;
      /* INDEX IN MDS OF NAME CELL OF TEXTUAL MACRO I */
      RETURN I - NFP(I) - 3;
      END TNAME;

IFP: PROCEDURE(N,I) ADDRESS;
      DECLARE N BYTE, I ADDRESS;
      /* INDEX IN MTS OF N-TH FORMAL PARAM OF MACRO I */
      RETURN MDS(TNAME(I) + N);
      END IFP;

MLINK: PROCEDURE(I) ADDRESS;
      DECLARE I ADDRESS;
      /* LINK FIELD OF MACRO I */
      RETURN MDS(I-1);
      END MLINK;

MNUMERIC: PROCEDURE(I) BYTE;
      DECLARE I ADDRESS;
      /* RETURN TRUE IF MACRO I IS OF TYPE NUMERIC */
      RETURN MTYPE(I)=MNUM;
      END MNUMERIC;

MRESERVED: PROCEDURE(I) BYTE;
      DECLARE I ADDRESS;
      /* RETURN TRUE IF MACRO I IS A RESERVED WORD */
      RETURN I <= RTOP;
      END MRESERVED;

JNAME: PROCEDURE(I) ADDRESS;
      DECLARE I ADDRESS;
      /* INDEX OF CELL IN MDS CONTAINING POINTER TO NAME OF MACRO I */
      IF MNUMERIC(I) THEN RETURN I-3;
      IF MRESERVED(I) THEN RETURN I-2;
      /* ELSE: TEXTUAL */
      RETURN TNAME(I);
      END JNAME;

INAME: PROCEDURE(I) ADDRESS;
      DECLARE I ADDRESS;
      /* INDEX OF NAME OF MACRO I IN MTS */
      TRACEON
      CALL TRAC('M');
      TRACEOFF *** */
      RETURN MDS(JNAME(I));
      END INAME;

PUSHST: PROCEDURE;
      /* PUSH STATUS QUO (NC, NCP, TSAVE, DSAVE) INTO THE STATUS STACK */
      TRACEON
      CALL TRAC('+');
      TRACEOFF *** */
      IF (QTOP:=QTOP+1)>= STSSIZE THEN
         /* OVERFLOW */ CALL ERROR(OF2H);    /* QUIT */
      QTSAVE(QTOP) = TSAVE;
      QDSAVE(QTOP) = DSAVE;
      QCSAVE(QTOP)  = NC;
      QNSAVE(QTOP) = NCP;
      OUTSIDE = 1;  /* BEGIN TO SCAN A MACROBODY IN 'OUTSIDE' MODE */
      END PUSHST;

POPST: PROCEDURE;
      /* EXIT FROM SCANNING A MACROBODY: POP PREVIOUS STATUS FROM
         THE STATUS STACK AND SET THINGS FOR THE SCANNER */
      TRACEON
      CALL TRAC('-');
      TRACEOFF *** */
      IF QTOP=255 THEN /* EMPTY STACK: THE MACROBODY JUST
```

```
      SCANNED WAS THE SOURCE PROGRAM ITSELF */ CALL TERMINATE;
   /* ELSE: RESTORE PREVIOUS STATUS */
   /* RESTORE THE HASHTABLE */
   DO WHILE DTOP> QBSAVE(QTOP);
      /* SET HASHTABLE ENTRY POINTING TO NEXT MACRO  */
      HASHTAB(MTS(INAME(DTOP)-1)) = MLINK(DTOP);
      DTOP = JNAME(DTOP) - 1;  /* INDEX OF NEXT MACRO TO POP */
   END;
   /* RESTORE OTHER COMPONENTS OF THE STATUS */
   TTOP = QTSAVE(QTOP);
   NCP = QNSAVE(QTOP);
   NC = QCSAVE(QTOP);
   QTOP = QTOP - 1;
   OUTSIDE = 1;  /* GET READY FOR OTHER MACROCALLS */
   END POPST;

MIDENT: PROCEDURE(J) ADDRESS;
   DECLARE J ADDRESS;
   /* IDENTIFY MACRO WHOSE NAME IS AT MTS(J) */
   DECLARE H BYTE, (I,K) ADDRESS;
   H = MTS(J-1);  /* HASHCODE */
   K = HASHTAB(H);  /* INDEX OF TOPMOST MACRO WITH SAME HASHCODE */
   DO WHILE K<>0;  /* SEARCH */
      I = INAME(K);  /* INDEX OF MACRONAME IN MTS */
      IF COMPAR(.MTS(J),.MTS(I)) THEN  /* FOUND */ RETURN K;
      K = MLINK(K);  /* NEXT */
   END;
   RETURN 0;
   END MIDENT;

SAVETOK: PROCEDURE(I) ADDRESS;
   DECLARE I ADDRESS;
   /* MOVE CURRENT TOKEN TO MTS, STARTING AT MTS(I);
   RETURN POINTER TO NEXT CHARACTER IN MTS */
   DECLARE J BYTE;
   IF (I + TOKTOP) > MTSSIZE THEN /* MTS OVERFLOW */
      CALL ERROR(OFSH);   /* QUIT */
   DO J = 1 TO TOKTOP;
      MTS(I) = TOKBUF(J);
      I = I + 1;
   END;
   MTS(I) = 0;   /* END MARKER */
   RETURN I;
   END SAVETOK;

LRLEX: PROCEDURE ADDRESS;
   /* LEXICAL ANALYZER */
   /* RETURN THE INTERNAL NUMBER OF A RESERVED WORD, OR:
      IDENTIFIER: LRLVAL = INDEX OF MACRO IN MDS, OR
                  -POINTER TO NAME IN MTS, IF ID IS NOT A MACRONAME;
      STRING:     LRLVAL = POINTER TO STRING IN MTS;
      NUMBER:     LRLVAL = VALUE OF THE NUMBER;
      0:          EOFILE     */
   DECLARE J ADDRESS;
   DO WHILE OUTSIDE;  /* OUTSIDE MACROCALLS */
      CALL GETIC;
      IF IC=MBEGIN THEN  /* BEGIN OF A MACROCALL */
         DO;
            OUTSIDE = 0;  /* ENTER MACROCALL */
            CALL GETIC;   /* SKIP MBEGIN */
            RETURN MBEGIN;
         END;
      IF IC=0  /* END OF MACROBODY */  THEN
         CALL POPST;  /* POP STATUS, GO ON SCANNING */
      ELSE CALL  PRC(IC);  /* WRITE IC ON OUTPUT FILE */
   END;
   /* OUTSIDE=0:  SCANNING INSIDE MACROCALL */
   CALL GETTOKEN;
   DO WHILE TOKERROR<>0;
      CALL ERROR(TOKERROR);
      CALL GETTOKEN;
   END;
   LRLVAL = TOKVAL;
```

```
      IF TOKTYPE = SPECIALC THEN RETURN TOKVAL;
      IF TOKTYPE = NUMBER THEN RETURN NUMBER;
      IF TOKTYPE = EOFILE THEN RETURN 0;
      /* IDENTIFIER OR STRING */
      TTSAVE = TTOP;    /* SAVE VALUE OF TTOP */
      LRLVAL = TTOP + 2;
      J = SAVETOK(LRLVAL);   /* MOVE TOKEN TO MTS */
      DO WHILE TOKCONT<>0;   /* GET REMAINDER OF TOKEN */
         CALL GETTOKEN;
         J = SAVETOK(J);
      END;
      MTS(LRLVAL-1) = HASHCODE;   /* STORE HASHCODE */
      TTOP = J;   /* PUSH THE NAME INTO MTS */
      IF TOKTYPE = STRING THEN RETURN STRING;
      /* ELSE: IDENTIFIER - CHECK WHETHER RESERVED WORD */
      J = MIDENT(LRLVAL);   /* SEARCH MDS FOR THIS NAME */
      IF J=0   /* NOT THERE */    THEN
         DO;
            LRLVAL = - LRLVAL;   /* -POINTER TO ID IN MTS */
            RETURN IDENTIFIER;
         END;
      /* ELSE: ALREADY THERE */
      TTOP = LRLVAL - 2;   /* DISCARD THE NAME */
      IF MRESERVED(J) THEN /* RESERVED WORD */
         RETURN MDS(J);    /* RESERVED WORD NO. */
      /* ELSE: A MACRONAME, BUT NOT RESERVED */
      LRLVAL = J;   /* INDEX OF THE MACRO IN MDS */
      RETURN IDENTIFIER;
      END LRLEX;

LOOKAT: PROCEDURE(I);
      DECLARE I ADDRESS;
      /* TELL THE SCANNER TO SCAN TEXT BEGINNING AT MTS(I) */
      NCP = I;
      NC = MTS(I);
      END LOOKAT;

PUSHH2: PROCEDURE;
      /*  CREATE ENTRY IN MDS FOR MACRONAME AT HANDLE(2) */
      TRACEON
      CALL TRAC('X');
      TRACEOFF *** */
      IF NEG(H2)     THEN  /* NEW IDENTIFIER */    H2 = - H2;
      ELSE  /* IDENTIFIER IS A MACRONAME */
         H2 = INAME(H2);    /* POINTER TO NAME IN MTS */
      TRACEON
      CALL TRAC('Y');
      TRACEOFF *** */
      CALL PUSHD(H2);
      END PUSHH2;

INTDECL: PROCEDURE;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
         <INT.DECL> ::= INT <IDENTIFIER>
         <INT.DECL> ::= <INT.DECL> <IDENTIFIER>    */
      DECLARE H BYTE;
      /* CREATE NUMERIC MACRO */
      CALL PUSHH2;    /* PUSH POINTER TO MACRONAME INTO MDS */
      CALL PUSHD(0);    /* INITIALIZE ITS VALUE TO 0 */
      H = MTS(H2-1);   /* GET IDENTIFIER'S HASHCODE */
      CALL PUSHD(HASHTAB(H));   /* SET LINK */
      CALL PUSHD(HNUM);   /* 0 FORMAL PARAMS, TYPE = NUMERIC MACRO */
      HASHTAB(H) = DTOP;   /* POINTER TO THE NEW MACRO */
      END INTDECL;

XMIT1: PROCEDURE;
      HH = H1;
      END  XMIT1;

LRACTION: PROCEDURE(N);
      DECLARE N BYTE;
      /* EXECUTE SEMANTIC ACTION ASSOCIATED WITH PRODUCTION N */
      IF N<10 THEN RETURN;   /* NO ACTION TO BE PERFORMED */
```

```
DO CASE N-10;

/*** <PROGRAM> ::= <STMT> */

/*** <PROGRAM> ::= <PROGRAM> <STMT> */

/*** <STMT> ::= <INT.DECL> */

/*** <STMT> ::= <ASSIGN.STMT> */

/*** <STMT> ::= <EVAL.STMT> */

/*** <STMT> ::= <MACRO.DECL> */

/*** <STMT> ::= <MACRO.CALL> */

/*** <STMT> ::= <IF.STMT> */

/*** <STMT> ::= <ERROR> */

/*** <INT.DECL> ::= INT <IDENTIFIER> */
   CALL INTDECL;

/*** <INT.DECL> ::= <INT.DECL> <IDENTIFIER> */
   CALL INTDECL;

/*** <ASSIGN.STMT> ::= <IDENTIFIER> = <EXPR> */
   DO;
      IF NEG(H1)  /* UNDEFINED MACRO */ THEN CALL ERROR(11H);
      ELSE
      DO;
         IF NUMERIC(H1)  /* NUMERIC MACRO */
          THEN MDS(H1-2)=H3;  /* EXECUTE ASSIGNMENT */
         ELSE  /* CANNOT ASSIGN */  CALL ERROR(14H);
      END;
      HH = H3;  /* PROPAGATE VALUE OF <EXPR> */
   END;

/*** <EVAL.STMT> ::= <FORMAT> <EXPR> */
   DO;
      IF L1=10 THEN CALL PRINTD(H2);  ELSE
      IF L1=8  THEN CALL PRINTO(H2);  ELSE
      IF L1=22 THEN CALL PRC(L2);   ELSE
      CALL PRINTHA(H2);
   END;

/*** <FORMAT> ::= DEC */
   HH = 10;

/*** <FORMAT> ::= OCT */
   HH = 8;

/*** <FORMAT> ::= HEX */
   HH = 16;

/* <FORMAT> ::= CHAR */
   HH = 22;

/*** <MACRO.DECL> ::= <MD.HEAD> <STRING> */
   DO;
      /* CREATE MACROBODY = <STRING>; INSTALL NEW MACRO */
      CALL PUSHD(H2);  /* POINTER TO MACROBODY */
      CALL PUSHD(HASHTAB(HSAVE));  /* SET LINK FIELD */
      CALL PUSHD(SHL(DOUBLE(NF),8) OR MMAC); /* NO. FORMALS, TYPE */
      HASHTAB(HSAVE) = DTOP;  /* POINTER TO MACRO DESCRIPTOR */
   END;

/*** <MD.HEAD> ::= MACRO <IDENTIFIER> */
   DO;  /* CREATE MACRONAME */
      CALL PUSHH2;  /* H2=POINTER TO IDENTIFIER IN MTS */
      HSAVE = MTS(H2-1);  /* SAVE ITS HASHCODE */
      NF = 0;  /* NO. FORMAL PARAMETERS */
   END;
```

```
/*** <MD.HEAD> ::= <MD.HEAD> <IDENTIFIER> */
   DO;
      CALL PUSHH2;  /* H2 = POINTER TO IDENTIFIER IN MTS */
      NF = NF + 1;
   END;

/*** <MACRO.CALL> ::= <MC.HEAD> */
   DO;
      /* TEMPORARY MACROS HAVE ALREADY BEEN CREATED;
         STATUS HAS BEEN SAVED IN DSAVE, TSAVE */
      IF NOT NEG(H1)  THEN /* NO ERRORS */
         DO;
            I=MDS(H1-2);  /* VALUE OF NUMERIC MACROS,
               POINTER TO BODY OF TEXTUAL MACROS */
            IF MNUMERIC(H1) THEN /* NUMERIC MACRO */
               CALL PRINTD(I);  /* OUTPUT ITS VALUE */
            ELSE /* TEXTUAL MACRO */
               DO;
                  CALL PUSHST;  /* SAVE STATUS QUO */
                  CALL LOOKAT(I);  /* SCAN MACROBODY */
               END;
         END;
   END;

/*** <MC.HEAD> ::= <IDENTIFIER> */
   DO;
      IF NEG(H1)  /* UNDEFINED MACRO */ THEN
         CALL ERROR(11H);
      DSAVE = DTOP;   /* SAVE STATUS QUO */
      TSAVE = TTSAVE;
      NA = 0;  /* NO. ACTUAL PARAMETERS */
      CALL XMIT1;
   END;

/*** <MC.HEAD> ::= <MC.HEAD> <STRING> */
   DO;  /* CREATE TEMPORARY MACRO , WITH:
      MACRONAME = FORMAL PARAM, MACROBODY = ACTUAL PARAM */
      IF NOT NEG(H1) THEN  /* NO ERRORS */
         DO;
            HH = 0;
            NA = NA + 1;
            IF MNUMERIC(H1) THEN /* NUMERIC MACRO CANNOT HAVE
               PARAMS */ CALL ERROR(12H);  ELSE
            IF NA>NFP(H1) THEN /* TOO MANY PARAMETERS */
               CALL ERROR(13H);  ELSE
            CALL XMIT1;  /* NO ERRORS */
            IF NOT NEG(HH) THEN /* NO ERRORS */
               DO;
                  J=IFP(HA,H1); /* J=POINTER TO CORRESPONDING
                     FORMAL PARAMETER IN MTS */
                  CALL PUSHD(J); /* FORMAL PARM BECOMES MACRONAME */
                  CALL PUSHD(H2); /* MACROBODY = STRING */
                  H = MTS(J-1);  /* HASHCODE OF NEW MACRONAME */
                  CALL PUSHD(HASHTAB(H));  /* SET LINK */
                  CALL PUSHD(HMAC); /* 0 FORMAL PARMS, TYPE=TEXT */
                  HASHTAB(H) = DTOP; /* POINTER TO NEW MACRO */
               END;
         END;
   END;

/*** <IF.STMT> ::= <IF> <EXPR> THEN <STRING> */
   DO;
      IF LOW(H2) THEN /* <EXPR> IS TRUE */
         DO;
            CALL PUSHST;  /* SAVE STATUS QUO */
            CALL LOOKAT(H4); /* TELL SCANNER TO SCAN STRING */
         END;
   END;

/*** <IF.STMT> ::= <IF> <EXPR> THEN <STRING> ELSE <STRING> */
   DO;
   CALL PUSHST;  /* SAVE STATUS QUO */
```

```
      IF LOW(H2) THEN /* <EXPR> IS TRUE */
         CALL LOOKAT(H4);
      ELSE CALL LOOKAT(H6);
   END;

/*** <IF> ::= IF */
   DO;
      DSAVE = DTOP;
      TSAVE = TTOP;
   END;

/*** <EXPR> ::= <LOG.FACTOR> */
   CALL XMIT1;

/*** <EXPR> ::= <EXPR> OR <LOG.FACTOR> */
   HH = H1 OR H3;

/*** <EXPR> ::= <EXPR> XOR <LOG.FACTOR> */
   HH = H1 XOR H3;

/*** <LOG.FACTOR> ::= <LOG.SEC> */
   CALL XMIT1;

/*** <LOG.FACTOR> ::= <LOG.FACTOR> AND <LOG.SEC> */
   HH = H1 AND H3;

/*** <LOG.SEC> ::= <LOG.PRIM> */
   CALL XMIT1;

/*** <LOG.SEC> ::= NOT <LOG.PRIM> */
   HH = NOT H2;

/*** <LOG.PRIM> ::= <ARIT.EXPR> */
   CALL XMIT1;

/*** <LOG.PRIM> ::= <ARIT.EXPR> <REL> <ARIT.EXPR> */
   DO;
      DECLARE LL BYTE;
      IF L2 = EQ THEN LL = (H1=H3);    ELSE
      IF L2 = LT THEN LL = NEG(H1-H3);    ELSE
      IF L2 = GT THEN LL = NEG(H3-H1);    ELSE
      IF L2 = NE THEN LL = (H1<>H3);   ELSE
      IF L2 = LE THEN LL = (H1=H3) OR NEG(H1 - H3);    ELSE
      LL =  (H1 = H3) OR NEG(H3 - H1);
      HH = LL;
   END;

/*** <REL> ::= = */
   HH = EQ;

/*** <REL> ::= < */
   HH = LT;

/*** <REL> ::= > */
   HH = GT;

/*** <REL> ::= <> */
   HH = NE;

/*** <REL> ::= <= */
   HH = LE;
/*** <REL> ::= >= */
   HH = GE;

/*** <ARIT.EXPR> ::= <TERM> */
   CALL XMIT1;

/*** <ARIT.EXPR> ::= <ARIT.EXPR> + <TERM> */
   HH = H1 + H3;

/*** <ARIT.EXPR> ::= <ARIT.EXPR> - <TERM> */
   HH = H1 - H3;
```

```
      /*** <TERM> ::= <PRIM> */
         CALL XMIT1;

      /*** <TERM> ::= <TERM> * <PRIM> */
         MM = M1 * M3;

      /*** <TERM> ::= <TERM> / <PRIM> */
         MM = M1 / M3;

      /*** <TERM> ::= <TERM> MOD <PRIM> */
         MM = M1 MOD M3;

      /*** <PRIM> ::= <IDENTIFIER> */
         DO;
            IF NEG(M1)  /* UNDEFINED MACRO */ THEN
               CALL ERROR(11H);  ELSE
            IF MNUMERIC(M1)  /* NUMERIC MACRO */ THEN
               MM = MDS(M1-2);  /* ITS VALUE */
            ELSE CALL ERROR(15H);
         END;

      /*** <PRIM> ::= <NUMBER> */
         CALL XMIT1;

      /*** <PRIM> ::= ( <ASSIGN.STMT> ) */
         MM = M2;

      /*** <PRIM> ::= ( <EXPR> ) */
         MM = M2;

      /*** <PRIM> ::= - <NUMBER> */
         MM = - M2;

      END;  /* OF CASE STATEMENT */
      END LRACTION;

TABLE: PROCEDURE(S, I) ADDRESS;
      DECLARE (S, I) ADDRESS;
      /* RETURN I-TH ENTRY IN THE ACTION TABLE OF STATE S */
      RETURN ( LRACT( LRPACT(S+1) + I) );
      END TABLE;

POP: PROCEDURE(N);
      DECLARE N BYTE;
      /* POP N TOPMOST STATES OF THE PARSE STACK */
      PTOP = PTOP - N;
      IF PTOP < 0 THEN /* STACK UNDERFLOW */
         CALL ERROR(0F5H);   /* QUIT */
      END POP;

PUSH: PROCEDURE(S,V);
      DECLARE (S,V) ADDRESS;
      /* PUSH STATE S AND VALUE V INTO THE PARSE STACKS */
      PTOP = PTOP+ 1;
      IF PTOP >= STACKSIZE THEN /* STACK OVERFLOW */
         CALL ERROR(0F4H);  /* QUIT */
      PSTACK(PTOP) = S;
      VSTACK(PTOP) = V;
      END PUSH;

GOTOF: PROCEDURE(STATE,NONTERM) ADDRESS;
      DECLARE (STATE,NONTERM) ADDRESS;
      /* RETURN NEXT STATE AFTER A REDUCTION */
      I = LRPGO(NONTERM); /* POINTER TO GOTO TABLE IN LRGO */
      DO FOREVER; /* TABLE SEARCH */
         IF (LRGO(I) = DEFAULT) OR (LRGO(I) = STATE)
            THEN RETURN LRGO(I+1);
         I = I + 2;
      END;
      END GOTOF;

REDUCE: PROCEDURE;
```

```
      /* EXECUTE A REDUCTION */
      DECLARE PN    BYTE;              /* PRODUCTION NUMBER */
      DECLARE NRS   BYTE;              /* NO. ELEMENTS RIGHT HAND SIDE */
      DECLARE LS    ADDRESS;           /* ID.NO. OF LEFT HAND SIDE */
      PN = TAB AND 0FFFH;
      /* CLEAR RECOVERY FLAG UNLESS <STMT> ::= <ERROR> */
      IF PN<>9 THEN RECOVERING=0;
      NRS = LRR2(PN);
      LS = LRR1(PN);
      /* SET HANDLE VARIABLES */
      DECLARE HB ADDRESS;  /* HANDLE BASE */
      DECLARE H BASED HB ADDRESS;  /* HANDLE */
      HB = .VSTACK(PTOP-NRS);
      H1=H(1);  H2=H(2);  H3=H(3);  H4=H(4);  H5=H(5);  H6=H(6);
      TRACESON
      CALL TRAC('R');
      TRACESOFF *** */
      CALL LRACTION(PN);  /* EXECUTE SEMANTIC ACTION */
      /* UPDATE THE PARSE STACK */
      CALL POP(NRS);  /* POP RIGHT HAND SIDE */
      CALL PUSH( GOTOF(PSTACK(PTOP),LS),HH); /* NEXT STATE */
      TRACESON
      CALL TRAC('A');
      TRACESOFF *** */
      END REDUCE;

SHIFT: PROCEDURE;
      /* EXECUTE A SHIFT ACTION */
      TRACESON
      CALL TRAC('S');
      TRACESOFF *** */
      CALL PUSH( TAB AND 0FFFH, LRLVAL);  /* PUSH NEXT STATE */
      INPUTSY = LRLEX;  /* GET ANOTHER INPUT SYMBOL */
      END SHIFT;

ACCEPT: PROCEDURE;
      /* BREAK PARSER LOOP */
      PARSING = 0;
      END ACCEPT;

PSERROR: PROCEDURE;
      /* PARSING ERROR HANDLER */
      TRACESON
      CALL TRAC('E');
      TRACESOFF *** */
      IF RECOVERING THEN /* ERROR MESSAGE FOR THIS ERROR ALREADY GIVEN */
         DO; /* SEARCH FOR A STATE WITH SHIFT ON <ERROR> */
            IF PTOP<=0 THEN /* CAN'T POP */
               CALL ERROR(0F2H);   /* QUIT */
            CALL POP(1);
            RETURN;
         END;
      CALL ERROR(7);
      INPUTSY =  YERROR;     /* <ERROR> */
      RECOVERING = 1;
      END PSERROR;

PARSE: PROCEDURE;
      /* PARSE SOURCE PROGRAM */
      INPUTSY = LRLEX;      /* GET FIRST INPUT SYMBOL */

      DO WHILE PARSING;
         IT = 0; /* INDEX WITHIN ACTION TABLE OF CURRENT STATE */
         SEARCHING = 1;
         DO WHILE SEARCHING;  /* IDENTIFY APPLICABLE ACTION */
            SEARCHING = 0;
            TAB = TABLE(PSTACK(PTOP),IT); /* CURRENT TABLE ENTRY */
            ACTION = TAB AND ACTMASK;      /* DECODE THE ACTION */
            IF ACTION = REDMASK THEN CALL REDUCE;
            ELSE IF ACTION = ERRORMASK THEN CALL PSERROR;
                 ELSE IF (INPUTSY OR SYMBMASK) = TAB THEN
                         DO; /* FOUND ENTRY  FOR THIS INPUT */
                            IT = IT + 1; /* LOOK AT CORRESPONDING ACTION*/
```

```
                                TAB = TABLE(PSTACK(PTOP),IT);
                                ACTION = TAB AND ACTNASK;
                                IF ACTION = REDMASK THEN CALL REDUCE;
                                ELSE IF ACTION = SHIFTMASK THEN CALL SHIFT;
                                     ELSE CALL ACCEPT;
                             END;
                          ELSE   /* NO MATCH */
                             DO; /* KEEP ON SEARCHING */
                                IT = IT + 2;
                                SEARCHING = 1;
                             END;
                    END;
              END;
              END PARSE;

READHEX: PROCEDURE BYTE;
      /* READ ONE HEX CHARACTER FROM THE PARSER TABLES FILE */
      DECLARE H BYTE;
      IF HEX(H:=GETC(.TABFCB)) THEN RETURN HEXVAL(H);
      CALL CMON(PRINT, .'NON-HEX DIGIT IN TABLES$');
      CALL EXIT;
      END READHEX;

READBYTE: PROCEDURE BYTE;
      /* READ TWO HEX DIGITS FROM PARSER TABLES FILE */
      RETURN SHL(READHEX,4) OR READHEX;
      END READBYTE;

READCS: PROCEDURE BYTE;
      /* READ BYTE FROM TABLES FILE WHILE COMPUTING CHECKSUM */
      DECLARE B BYTE;
      CS= CS + (B:=READBYTE);
      RETURN B;
      END READCS;

TLOADER: PROCEDURE;
      /* LOAD ONE PARSER TABLE INTO MEMORY */
      DECLARE C BYTE;
      DO FOREVER; /* READ INPUT UNTIL :00XX (END OF TABLE) IS FOUND */
         DO WHILE GETC(.TABFCB)<>':';   /* LOOK FOR : */
         END;
         /* SET CHECKSUM TO 0, SAVE RECORD LENGTH */
         CS=0;
         IF (RL:=READCS) = 0 THEN /* END OF TABLE */ RETURN;
         C = READCS;  C = READCS;    /* SKIP LOAD ADDRESS */
         C = READCS;                 /* SKIP RECORD TYPE */
         /* NOW START STORING THE BYTES */
         DO WHILE (RL:=RL-1)<>255;
            MEMORY(MEMPTR)= READCS;
            MEMPTR = MEMPTR + 1;
         END;
         /* READ CHECKSUM AND COMPARE */
         IF CS+READBYTE<>0 THEN  /* ERROR */
            DO;  /* TELL OPERATOR AND QUIT */
               CALL CMON(PRINT,.'CHECKSUM ERROR, LINE $');
               CALL PRINTD(LINE);
               CALL EXIT;
            END;
      END;  /* DO FOREVER */
      END TLOADER;

INITTAB: PROCEDURE;
      /* INITIALIZE PARSER TABLES */
      /*  OPEN PARSER TABLES FILE */
      CALL CMON(INITDSK,0);  /* SELECT DISK */
      IF VMON(OPEN,.TABFCB)=255 THEN CALL CANTOP(.TABFCB);
      /* SET NEXT RECORD TO 0 */
      TABFCB(32)=0;
      /* LOAD THE TABLES INTO MEMORY */
      MEMPTR = 0;       /* START LOADING AT "MEMORY" BASE */
      ACTB = .MEMORY;
      CALL TLOADER;     /* LOAD LRACT */
      PACTB = .MEMORY(MEMPTR);
```

```
        CALL TLOADER;     /* LOAD LRPACT */
        R1B   = .MEMORY(MEMPTR);
        CALL TLOADER;     /* LOAD LRR1 */
        R2B   = .MEMORY(MEMPTR);
        CALL TLOADER;     /* LOAD LRR2 */
        PGOB  = .MEMORY(MEMPTR);
        CALL TLOADER;     /* LOAD LRPGO */
        GOB   = .MEMORY(MEMPTR);
        CALL TLOADER;     /* LOAD LRGO */
        MTSB = .MEMORY(MEMPTR);       /* BASE OF MTS */
        MTSSIZE = FBASE - MTSB;
        I = MEMPTR;
        CALL TLOADER;    /* LOAD RESERVED WORDS */
        TTOP = MEMPTR - I - 1;     /* INDEX OF TOP OF MTS */
        /* RESET HASH TABLE */
        DO I=0 TO LAST(HASHTAB);
           HASHTAB(I) = 0;
        END;
        /* INITIALIZE MDS */
        J = 1;  /* INDEX OF FIRST RESERVED WORD IN MTS */
        DO WHILE J<TTOP;
           /* COMPUTE ITS HASHCODE */
           HASHCODE = 0;
           I = J;
           DO WHILE MTS(I)<>0;
            . CALL HASHF(MTS(I));
              I = I + 1;
           END;
           CALL PUSHD(J);  /* POINTER TO MACRONAME */
           CALL PUSHD(HASHTAB(HASHCODE));  /* LINK */
           CALL PUSHD(DOUBLE(MTS(J-1))+256);  /* RESERVED WORD NO. */
           MTS(J-1) = HASHCODE;  /* STORE HASHCODE IN MTS */
           HASHTAB(HASHCODE) = DTOP;    /*POINTER TO THE MACRO CREATED */
           J = I + 2;    /* NEXT */
        END;
        RTOP = DTOP;  /* SAVE INDEX OF TOPMOST RESERVED MACRO */
        END INITTAB;

MAIN:
        TRACESON
        CALL SETCRT;
        CALL PRL(.'TRACE(Y/N):$');
        TRACING=VMON(1,0);
        OUTDEV=DISK;
        TRACESOFF *** */
        CALL INITTAB;
        CALL INITFILES;
        CALL PARSE;

EOF
```

```
/*
              ****************************************
              *                                      *
              *  L81: PHASE 1 OF THE L80 COMPILER    *
              *                                      *
              *     LUIZ PEDROSO - OCTOBER 1975      *
              *                                      *
              ****************************************
*/

    DECLARE LIT             LITERALLY 'LITERALLY';
    DECLARE FOREVER         LITERALLY 'WHILE 1';
    DECLARE CR              LITERALLY '0DH';    /* CARRIAGE-RETURN */
    DECLARE LF              LITERALLY '0AH';    /* LINE-FEED */
    DECLARE CONTROLZ        LITERALLY '1AH';    /* EOF */
    DECLARE I ADDRESS, C BYTE;                  /* TEMPORARIES */

    /* YACC-ASSIGNED TERMINAL NUMBERS */
    DECLARE YERROR          LITERALLY '256';
    DECLARE IDENTIFIER      LITERALLY '257';
    DECLARE NUMBER          LITERALLY '258';
    DECLARE STRING          LITERALLY '259';
    DECLARE ASL             LITERALLY '260';

    /* CP/M SYSTEM CONSTANTS */
    DECLARE CPM             LITERALLY '0';      /* CP/M REBOOT ENTRY */
    DECLARE SFCBA           LITERALLY '005CH'; /* SYSTEM FCB ADDRESS */
    DECLARE SBUFA           LITERALLY '0080H'; /* SYSTEM FCB ADDRESS */
    DECLARE FBASE           LITERALLY '3200H'; /* FDOS BASE */
    DECLARE BDOS            LITERALLY '3FFDH'; /* BASIC DOS ENTRY */

    /* I/O PRIMITIVES */
    DECLARE PRINTCHAR       LITERALLY  '2';
    DECLARE PRINT           LITERALLY  '9';
    DECLARE OPEN            LITERALLY  '15';
    DECLARE CLOSE           LITERALLY  '16';
    DECLARE MAKE            LITERALLY  '22';
    DECLARE DELETE          LITERALLY  '19';
    DECLARE READBF          LITERALLY  '20';
    DECLARE WRITEBF         LITERALLY  '21';
    DECLARE INITDSK         LITERALLY  '13';
    DECLARE SETBUF          LITERALLY  '26';

    /* FILE CONTROL BLOCKS */
    DECLARE TABFCB(33) BYTE  /* PARSER TABLES FILE */
          INITIAL(0,'L81     ','TAB',0,0,0,0);
    DECLARE PAFCB(33) BYTE  /* PARSER ACTIONS FILE */
          INITIAL(0,'        ','80P',0,0,0,0);
    DECLARE SLFCB(33) BYTE   /* SYMBOL LIST  FILE */
          INITIAL(0,'        ','80S',0,0,0,0);
    DECLARE IFA ADDRESS INITIAL (SFCBA);        /* INPUT FCB ADDRESS */
    DECLARE IFCB BASED IFA BYTE;  /* INPUT FILE CONTROL BLOCK */

    /* OUTPUT BUFFER */
    DECLARE OBUF(128) BYTE;
    DECLARE OBP BYTE INITIAL(0);  /* OUTPUT BUFFER POINTER */
    DECLARE OFCBA ADDRESS;   /* ADDRESS OF OUTPUT FCB */

    /* INPUT BUFFER */
    DECLARE IBUF(128) BYTE;
    DECLARE IBP BYTE INITIAL(128);     /* INPUT BUFFER POINTER */
    DECLARE IFCBA ADDRESS;   /* ADDRESS OF INPUT FCB */

    /* PARSER TABLES MASKS */
    DECLARE ACTMASK         LITERALLY '0F000H';
    DECLARE SYMBMASK        LITERALLY '01000H';
    DECLARE SHIFTMASK       LITERALLY '02000H';
    DECLARE REDMASK         LITERALLY '03000H';
    DECLARE ERRORMASK       LITERALLY '00000H';
    DECLARE DEFAULT         LITERALLY '0FFFFH';
```

```
/* PARSER ACTIONS OPCODES */
DECLARE XREDUCE          LITERALLY '01H';
DECLARE XSHIFT           LITERALLY '02H';
DECLARE XACCEPT          LITERALLY '03H';
DECLARE XLINE            LITERALLY '04H';

/* PARSER GLOBAL VARIABLES */
DECLARE STACKSIZE LITERALLY '50';
DECLARE PSTACK(STACKSIZE) ADDRESS INITIAL (0);  /* PARSE STACK */
DECLARE PTOP ADDRESS INITIAL(0);     /* PARSE STACK POINTER */
DECLARE PARSING      BYTE INITIAL (1);
DECLARE RECOVERING   BYTE INITIAL (0);
DECLARE SEARCHING    BYTE;
DECLARE (INPUTSY,TAB,ACTION,IT) ADDRESS;
DECLARE ERRORCOUNT BYTE    INITIAL (0);

/* PARSER TABLES */
DECLARE ACTB  ADDRESS, LRACT  BASED ACTB  ADDRESS;
DECLARE PACTB ADDRESS, LRPACT BASED PACTB ADDRESS;
DECLARE R1B   ADDRESS, LRR1   BASED R1B   ADDRESS;
DECLARE R2B   ADDRESS, LRR2   BASED R2B   ADDRESS;
DECLARE COB   ADDRESS, LRCO   BASED COB   ADDRESS;
DECLARE PCOB  ADDRESS, LRPCO  BASED PCOB  ADDRESS;

/* SYMBOL LIST */
DECLARE SYMLB ADDRESS, SYMLIST BASED SYMLB BYTE;
DECLARE SYMLSIZE ADDRESS;  /* SIZE OF SYMLIST */
DECLARE SYMLNEXT ADDRESS;  /* INDEX OF NEXT AVAILABLE BYTE IN SYMLIST */
DECLARE SLFIRST  ADDRESS;  /* INDEX OF FIRST BYTE AFTER RESERVED WORDS */

/* HASH TABLE */
DECLARE HASHTAB(128) ADDRESS;
DECLARE HASHMASK LITERALLY '0177Q';
DECLARE HASHCODE BYTE;

/* LEXICAL ANALYZER GLOBAL VARIABLES */
DECLARE TOKBUF(30) BYTE;
DECLARE TOKBUFTOP LITERALLY 'LAST(TOKBUF)';
DECLARE TOKVAL ADDRESS;
DECLARE TOKTYPE ADDRESS;
DECLARE TOKCONT BYTE INITIAL (0);
DECLARE TOKERROR BYTE;
DECLARE TOKTOP LITERALLY 'TOKBUF';
DECLARE IC BYTE INITIAL (' ');   /* INPUT CHARACTER */
DECLARE NC BYTE INITIAL (' ');   /* NEXT CHARACTER */
DECLARE LRLVAL ADDRESS;
DECLARE LINE   ADDRESS INITIAL (1);
DECLARE SPECIALC         LITERALLY '1';
DECLARE EOFILE           LITERALLY '0';

/* PARSER TABLES LOADER GLOBAL VARIABLES */
DECLARE (CS,RL) BYTE;          /* CHECKSUM, RECORD LENGTH */
DECLARE (LAH,LAL) BYTE;        /* LOAD ADDRESS */
DECLARE MEMPTR ADDRESS;        /*  MEMORY POINTER */

/* SWITCHES FOR BUILT-IN TRACE */
/* TO INCLUDE TRACE ROUTINES, SET TRACESON TO BLANKS */
/* TO EXCLUDE TRACE ROUTINES, SET TRACESON TO SLASH-STAR */
DECLARE TRACESON  LIT ' /* ';
DECLARE TRACESOFF LIT ' /* ';

TRACESON
DECLARE TRACE BYTE;
TRACESOFF *** */

VMON: PROCEDURE(FUNC,INFO) BYTE;
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; RETURN A VALUE */
      GO TO BDOS;
      END VMON;

CMON: PROCEDURE(FUNC,INFO);
      DECLARE FUNC BYTE, INFO ADDRESS;
```

```
      /* ASK CP/M TO EXECUTE FUNC; NO VALUE RETURNED */
      GO TO BDOS;
      END CMON;

PRC:  PROCEDURE(C);
      DECLARE C BYTE;
      /* PRINT CHARACTER C ON THE CRT */
      CALL CMON(PRINTCHAR,C);
      END PRC;

CRLF: PROCEDURE;
      /* SEND CARRIAGE RETURN, LINE FEED TO THE CRT */
      CALL PRC(CR);    CALL PRC(LF);
      END CRLF;

PRS:  PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT STRING STARTING AT ADDRESS A UNTIL $ IS FOUND */
      CALL CMON(PRINT,A);
      END PRS;

PRL:  PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT LINE STARTING AT A UNTIL $ IS FOUND */
      CALL CRLF;
      CALL PRS(A);
      END PRL;

PRINTH: PROCEDURE(H);
      DECLARE H BYTE;
      /* PRINT A HEXADECIMAL CHARACTER */
      IF (H>9) THEN CALL PRC(H-10+'A');
      ELSE CALL PRC(H+'0');
      END PRINTH;

PRINTHB: PROCEDURE(B);
      DECLARE B BYTE;
      /* PRINT 2 HEXADECIMAL CHARACTERS REPRESENTING B */
      CALL PRINTH(SHR(B,4));
      CALL PRINTH(B AND 0FH);
      END PRINTHB;

PRINTHA: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT 4 HEXADECIMAL CHARACTERS REPRESENTING A */
      CALL PRINTHB(HIGH(A));
      CALL PRINTHB(LOW(A));
      END PRINTHA;

EXIT: PROCEDURE;
      /* RETURN TO CP/M */
      CALL PRL(.'END L81$');
      CALL CRLF;
      GO TO CPM;
      END EXIT;

PRFN: PROCEDURE(F);
      DECLARE F ADDRESS;
      /* PRINT NAME OF FILE WITH FCB AT ADDRESS F */
      DECLARE FCB BASED F BYTE;
      DECLARE I BYTE;
      DO I=1 TO 11;
         IF FCB(I)<>' ' THEN CALL PRC(FCB(I));
         IF I=8 THEN CALL PRC('.');
      END;
      END PRFN;

CANTOP: PROCEDURE(F);
      DECLARE F ADDRESS; /* ADDRESS OF FCB */
      /* PRINT OPEN ERROR MESSAGE AND QUIT */
      CALL PRL(.'CANNOT OPEN $');
      CALL PRFN(F);
      CALL EXIT;
```

```
      END CANTOP;

MOVBUF: PROCEDURE(D,S);
      DECLARE(D,S) ADDRESS;
      /* MOVE 128 BYTES FROM S TO D */
      DECLARE DB BASED D BYTE;
      DECLARE SB BASED S BYTE;
      DECLARE I BYTE;
      DO I = 0 TO 127;
         DB(I) = SB(I);
      END;
      END MOVBUF;

FLUSH: PROCEDURE;
      /* FLUSH OUTPUT BUFFER INTO FILE WHOSE FCB STARTS AT OFCBA */
      IF OBP=0  /* BUFFER EMPTY */ THEN RETURN;
      CALL MOVBUF(SBUFA,.OBUF);  /* MOVE BUFFER TO I/O AREA */
      IF VMON(WRITEBF,OFCBA)<>0 THEN /* UNSUCCESSFUL WRITE */
         DO;
            CALL PRL(.'WRITE ERROR $');
            CALL PRFN(OFCBA);
            CALL EXIT;
         END;
      OBP = 0;  /* RESTORE BUFFER POINTER */
      END FLUSH;

PUTC: PROCEDURE(C);
      DECLARE C BYTE;
      /* WRITE CHARACTER C ON OUTPUT FILE WITH FCB AT OFCBA */
      IF OBP=128 THEN /* BUFFER FULL */   CALL FLUSH;
      OBUF(OBP) = C;
      OBP = OBP + 1;
      END PUTC;

CLOSEF: PROCEDURE;
      /* CLOSE OUTPUT FILE WHOSE FCB STARTS AT OFCBA */
      CALL FLUSH;
      IF VMON(CLOSE,OFCBA)<>255 THEN     RETURN;
      /* ELSE: ERROR */
      CALL PRL(.'CANNOT CLOSE $');
      CALL PRFN(OFCBA);
      CALL EXIT;
      END CLOSEF;

GETC: PROCEDURE BYTE;
      /* READ NEXT CHARACTER FROM FILE WHOSE FCB IS AT IFCBA */
      IF IBP=128 THEN /* BUFFER EMPTY */
         DO;  /* READ NEXT RECORD */
            IBP = 0;
            IF (C:=VMON(READBF,IFCBA))=1 THEN /* EOF */ RETURN 0;
            IF C<>0 THEN /* UNSUCCESSFUL READ */
               DO;  /* WARN OPERATOR AND QUIT */
                  CALL PRL(.'READ ERROR $');
                  CALL PRFN(IFCBA);
                  CALL EXIT;
               END;
            CALL MOVBUF(.IBUF,SBUFA);  /* SET INPUT BUFFER */
         END;
      C=IBUF(IBP);  /* NEXT CHARACTER */
      IBP = IBP + 1;
      RETURN C;
      END GETC;

INITFILES:  PROCEDURE;
      /* INITIALIZE FILES */
      IFCBA = SFCBA;   /* INPUT FILE: SOURCE PROGRAM */
      OFCBA = .PAFCB;  /* OUTPUT FILE: PARSER ACTIONS FILE */
      /* SET TYPE OF INPUT FILE TO 'L80' */
      IFCB(9)='L';
      IFCB(10)='8';
      IFCB(11)='0';
      /* OPEN INPUT FILE */
      IF VMON(OPEN,IFCBA) = 255 THEN CALL CANTOP(IFCBA);
```

```
        /* SET NAMES OF OUTPUT FILES */
        DO I = 1 TO 8;
          PAFCB(I), SLFCB(I) = IFCB(I);
        END;
        /* DELETE OLD VERSIONS OF THE OUTPUT FILES */
        CALL CMON(DELETE,.PAFCB);
        CALL CMON(DELETE,.SLFCB);
        /* CREATE NEW VERSIONS OF THE OUTPUT FILES */
        IF VMON(MAKE,.PAFCB) = 255 THEN CALL CANTOP(.PAFCB);
        IF VMON(MAKE,.SLFCB) = 255 THEN CALL CANTOP(.SLFCB);
        /* SET NEXT RECORD TO 0 FOR ALL FILES */
        IFCB(32), PAFCB(32),SLFCB(32) = 0 ;
        /* SET NEXT CHARACTER POINTER FOR INPUT BUFFER */
        IBP=128;  /* INPUT BUFFER IS EMPTY */
        END INITFILES;

BLANK: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS BLANK */
        RETURN  (C=' ') OR (C=CR) OR (C=LF);
        END BLANK;

NUMERIC: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS NUMERIC */
        RETURN (C>='0') AND (C<='9');
        END NUMERIC;

ALPHABETIC: PROCEDURE (C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS ALPHABETIC */
        RETURN ( (C>='A') AND (C<='Z') ) OR
           ( (C>=61H) AND (C<=7AH) ) OR (C = '$' );
        END ALPHABETIC;

ALPHANUMERIC: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS ALPHANUMERIC */
        RETURN ALPHABETIC(C) OR NUMERIC(C);
        END ALPHANUMERIC;

HEX: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* TRUE IF C IS HEXADECIMAL */
        RETURN NUMERIC(C) OR ( (C>='A') AND (C<='F') );
        END HEX;

HEXVAL: PROCEDURE(C) BYTE;
        DECLARE C BYTE;
        /* RETURN THE VALUE OF HEX CHARACTER C */
        IF NUMERIC(C) THEN RETURN C-'0';
        RETURN C-'A'+10;
        END HEXVAL;

KERROR: PROCEDURE(N);
        DECLARE N BYTE;
        /* PRINT N-TH ERROR MESSAGE */
        DECLARE MSG ADDRESS;
        ERRORCOUNT = DEC(ERRORCOUNT + 1);
        CALL PRL(.'LINE $');
        CALL PRINTHA(LINE);
        CALL PRC(':');
        DO CASE N;
          MSG = .' SYNTAX ERROR NEAR $';              /*  0 */
          MSG = .' UNEXPECTED EOF $';                 /*  1 */
          MSG = .' MISPELLED NUMBER: $';              /*  2 */
          MSG = .' NUMBER TOO LARGE: $';              /*  3 */
        END;
        CALL PRS(MSG);
        DO N = 1 TO TOKTOP;
           CALL PRC(TOKBUF(N));
        END;
        END KERROR;
```

```
DECLARE WRPA1 LIT 'PUTC';  /* WRITE 1 BYTE ON PARSER ACTIONS FILE */

WRPA2: PROCEDURE(A);
       DECLARE A ADDRESS;
       /* WRITE 2 BYTES ON PARSER ACTIONS FILE */
       CALL WRPA1(LOW(A));
       CALL WRPA1(HIGH(A));
       END WRPA2;

GETIC: PROCEDURE;
       /* GET ONE CHARACTER FROM SOURCE PROGRAM INTO IC;
          LOOK AHEAD AT NEXT CHARACTER (NC);
          END OF FILE CONDITIONS ARE DENOTED (AT THE EXIT OF GETIC) BY:
          IC=0,   NC=0:   EOF;
          IC<>0,  NC=0:   ALMOST EOF;
          IC<>0,  NC<>0:  NOT EOF    */
       IC = NC;
       /* NOW GET THE NEW NC */
       NC = GETC;
       IF NC=LF  /* NEW LINE */  THEN
          DO;
             DECLARE (XL,XH) BYTE;
             XL = DEC(LOW(LINE)+1);
             XH = DEC(HIGH(LINE) PLUS 0);
             LINE = SHL(DOUBLE(XH),8) OR XL;
             /* TELL LINE NUMBER TO L32 */
             CALL WRPA1(XLINE);
             CALL WRPA2(LINE);
          END;
       IF NC=CONTROLZ THEN  /* EOFILE */ NC=0;
       END GETIC;

PUTIC: PROCEDURE;
       /* PUT INPUT CHARACTER IC INTO TOKBUF */
       TOKBUF(TOKTOP:=TOKTOP+1) = IC;  /* TOKBUF(0) IS NOT USED */
       END PUTIC;

PGIC: PROCEDURE;
       CALL PUTIC;  CALL GETIC;
       END PGIC;

HASHF: PROCEDURE(C);
       DECLARE C BYTE;
       /* SET GLOBAL VARIABLE HASHCODE, USING THE INPUT CHARACTER C */
       HASHCODE = (HASHCODE + C) AND HASHMASK;
       END HASHF;

LOADING: PROCEDURE BYTE;
       /* RETURN 1 IF IT IS OK TO KEEP ON LOADING TOKBUF */
       RETURN   TOKCONT     AND (TOKTOP < TOKBUFTOP);
       END LOADING;

GETID: PROCEDURE;
       /* LOAD AN IDENTIFIER INTO TOKBUF */
       TOKTYPE = IDENTIFIER;
       DO WHILE LOADING;
          CALL HASHF(IC);
          CALL PGIC;
          TOKCONT = ALPHANUMERIC(IC); /* CONTINUE IF ALPHANUMERIC */
       END;
       END GETID;

GETSTR: PROCEDURE;
       /* LOAD A STRING INTO TOKBUF */
       TOKTYPE = STRING;
       DO WHILE LOADING;
          IF IC <> '''' THEN
             DO;
                CALL PGIC;
                IF    IC=0     THEN /* UNEXPECTED EOF */
                   DO;
                      TOKERROR = 1;
```

```
                    TOKCONT = 0;
                    RETURN;
                END;
            END;
        ELSE  /* FOUND A QUOTE */
            DO;
                CALL GETIC;  /* SKIP THE QUOTE */
                IF IC <> '''' THEN /* END OF STRING */ TOKCONT = 0;
                ELSE /* QUOTED QUOTE */  CALL PGIC;
            END;
    END;
    END GETSTR;

NHNC: PROCEDURE BYTE;
    /* TRUE IF NC IS NEITHER HEX NOR 'H' */
    RETURN NOT HEX(NC) AND (NC<>'H');
    END NHNC;

GETNUM:PROCEDURE;
    /* LOAD A NUMBER INTO TOKBUF; PUT ITS VALUE INTO TOKVAL */
    DECLARE  BASE BYTE;
    BASE = 0;
    TOKTYPE = NUMBER;
    DO WHILE LOADING;
        CALL PGIC;
        IF (IC='O') OR (IC='Q') THEN BASE = 8; ELSE
        IF IC='H' THEN BASE = 16;  ELSE
        IF IC='B' AND NHNC THEN BASE=2; ELSE
        IF IC='D' AND NHNC THEN BASE=10; ELSE
        IF NOT HEX(IC)      THEN BASE = 1;
        IF BASE >1 THEN CALL GETIC; /* SKIP 'BASE' CHARACTER */ ELSE
        IF BASE = 1 THEN BASE = 10;
        TOKCONT = (BASE=0);  /* CONTINUE UNTIL BASE IS DETERMINED */
    END;
    /* COMPUTE THE VALUE OF THE NUMBER */
    DECLARE D BYTE;
    DECLARE MPLIER BYTE, MPLAND ADDRESS;
    DO I = 1 TO TOKTOP;
        D = HEXVAL(TOKBUF(I));  /* CURRENT DIGIT */
        IF D >= BASE THEN TOKERROR = 2;
        /* TOKVAL = TOKVAL * BASE + D */
        MPLIER = BASE;
        MPLAND = TOKVAL;
        TOKVAL = D;     /* TOKVAL GETS THE PARTIAL SUMS */
        DO WHILE MPLIER <> 0;
            IF MPLIER THEN
                DO;
                    TOKVAL = TOKVAL + MPLAND;
                    IF CARRY THEN TOKERROR = 3;
                END;
            MPLIER = SHR(MPLIER,1);
            MPLAND = SHL(MPLAND,1);
        END;
    END;
    END GETNUM;

GETSPECIAL: PROCEDURE;
    /* LOAD A SPECIAL CHARACTER INTO TOKBUF */
    TOKTYPE = SPECIALC;
    TOKVAL=0;
   IF IC=NC THEN /* CHECK IF DOUBLE OPERATOR */
        DO;
          IF IC = '+' THEN TOKVAL = 261    ;  ELSE
          IF IC = '-' THEN TOKVAL = 262    ;  ELSE
          IF IC = '=' THEN TOKVAL = 263    ;  ELSE
          IF IC = '>' THEN TOKVAL = 264    ;  ELSE
          IF IC = '<' THEN TOKVAL = 265    ;  ELSE
          IF IC = ':' THEN TOKVAL = 266    ;  ELSE
          IF IC = 5CH THEN TOKVAL = 267    ;
        END;
    IF TOKVAL <> 0 THEN CALL PGIC; /* SKIP ONE EXTRA CHARACTER */
    ELSE TOKVAL = IC;
    CALL PGIC;
```

```
      TOKCONT = 0;
      END GETSPECIAL;

GETTOKEN: PROCEDURE;
      /* GET A TOKEN, AND RETURN THE GLOBAL VARIABLES:
         TOKTYPE:   IDENTIFIER, STRING, NUMBER, SPECIALC, EOFILE;
         TOKVAL:    VALUE OF NUMBER, INTERNAL NO. OF SPECIALC;
         TOKERROR:  0 IF VALID TOKEN, A MESSAGE NO. OTHERWISE;
         TOKBUF:    ARRAY OF CHARACTERS (1 TO TOKTOP) WITH TOKEN NAME;
         TOKTOP:    INDEX OF LAST CHARACTER IN TOKBUF;
         HASHCODE:  HASHCODE  OF IDENTIFIERS, STRINGS */
       /* IC:  INPUT CHARACTER (ALREADY READ IN, BUT NOT YET USED;
          NC:  NEXT CHARACTER  (LOOK-AHEAD CHARACTER)  */
      DECLARE SOMETHING BYTE;
      TOKTOP, TOKERROR = 0;
      IF TOKCONT = 0 THEN /* GET A NEW TOKEN */
         DO;
            TOKVAL, HASHCODE, SOMETHING = 0;
            TOKCONT = 1;
            DO WHILE SOMETHING = 0;  /* LOOK FOR SOMETHING */
               DO WHILE BLANK(IC); CALL GETIC; END; /* SKIP BLANKS */
               IF (IC='/') AND (NC='*') THEN  /* A COMMENT */
                  DO;
                     CALL GETIC; CALL GETIC; /* SKIP SLASH-STAR */
                     DO WHILE (IC<>'*') OR (NC<>'/');
                        IF   IC=0    THEN /* UNFINISHED COMMENT */
                           DO; TOKTYPE=EOFILE; RETURN; END;
                        ELSE CALL GETIC;  /* SKIP A CHARACTER */
                     END;
                     CALL GETIC; CALL GETIC; /* SKIP STAR-SLASH */
                  END;  /* GO LOOK FOR SOMETHING */
               ELSE SOMETHING = 1;  /* FOUND SOMETHING */
            END;
            IF   IC=0    THEN     TOKTYPE = EOFILE; ELSE
            IF ALPHABETIC(IC) THEN CALL GETID; ELSE
            IF NUMERIC(IC) THEN CALL GETNUM; ELSE
            IF IC='''' THEN DO; CALL GETIC; CALL GETSTR; END; ELSE
            CALL GETSPECIAL;
         END;
      ELSE /* GET REMAINDER OF PRESENT TOKEN */
         DO;
            IF TOKTYPE = IDENTIFIER THEN CALL GETID; ELSE
            IF TOKTYPE = STRING     THEN CALL GETSTR; ELSE
            IF TOKTYPE = NUMBER     THEN CALL GETNUM; ELSE
            CALL GETSPECIAL;
         END;
      END GETTOKEN;

SLINK: PROCEDURE(I,L);
      DECLARE (I,L) ADDRESS;
      /* SET THE LINK FIELD OF SYMBOL I OF SYMLIST TO L */
      DECLARE A ADDRESS;
      DECLARE LK BASED A ADDRESS;
      A = .SYMLIST(I-2);
      LK = L;
      END SLINK;

LINK: PROCEDURE (I) ADDRESS;
      DECLARE I ADDRESS;
      /* RETURN LINK FIELD OF SYMBOL I IN SYMLIST */
      DECLARE A ADDRESS;
      DECLARE L BASED A ADDRESS;
      A = .SYMLIST(I-2);
      RETURN L;
      END LINK;

COMPAR: PROCEDURE (A1,A2) BYTE;
      DECLARE (A1,A2) ADDRESS;
      /* RETURN 1 IF STRINGS AT A1, A2 ARE EQUAL, 0 OTHERWISE */
      DECLARE B1 BASED A1 BYTE;
      DECLARE B2 BASED A2 BYTE;
      DO WHILE B1 = B2;
         IF B1 = 00H /* (END OF STRING) */ THEN /* MATCH */ RETURN 1;
```

```
      A1 = A1 + 1;    A2 = A2 + 1;
   END;
   RETURN 0;
   END COMPAR;

LOOKUPSYM: PROCEDURE(K) ADDRESS;
   DECLARE K ADDRESS;
   /* CHECK WHETHER THE TOKEN (IDENTIFIER OR STRING) AT SYMLIST(K)
      IS ALREADY IN SYMLIST; RETURN POINTER TO SYMLIST ENTRY IF
      YES, 0 OTHERWISE */
   DECLARE I ADDRESS;
   I = HASHTAB(HASHCODE);
   DO WHILE I > 0;
      IF COMPAR(.SYMLIST(K),.SYMLIST(I)) THEN RETURN I;
      I = LINK(I);
   END;
   RETURN 0;    /* NOT FOUND */
   END LOOKUPSYM;

SAVETOK: PROCEDURE(I) ADDRESS;
   DECLARE I ADDRESS;
   /* MOVE CURRENT TOKEN TO SYMLIST, STARTING AT LOCATION I OF
      SYMLIST; RETURN POINTER TO NEXT CHARACTER IN SYMLIST */
   DECLARE J BYTE;
   IF (I + TOKTOP) > SYMLSIZE THEN /* SYMLIST OVERFLOW */
      DO;  /* TELL OPERATOR AND QUIT */
         CALL PRL(.'SYMBOL LIST OVERFLOW3');
         CALL EXIT;
      END;
   DO J = 1 TO TOKTOP;
      SYMLIST(I) = TOKBUF(J);
      I = I + 1;
   END;
   SYMLIST(I) = 00H;    /* END MARKER */
   RETURN I;
   END SAVETOK;

LRLEX: PROCEDURE ADDRESS;
   /* LEXICAL ANALYZER */
   /* RETURN THE INTERNAL NUMBER OF A RESERVED WORD, OR:
      IDENTIFIER: LRLVAL= POINTER TO IDENTIFIER IN SYMLIST;
      STRING:     LRLVAL= POINTER TO STRING IN SYMLIST;
      NUMBER:     LRLVAL= VALUE OF THE NUMBER;
      0:          EOFILE */
   DECLARE (J,K) ADDRESS;
   CALL GETTOKEN;
   DO WHILE TOKERROR <> 0;
      CALL KERROR(TOKERROR);
      CALL GETTOKEN;
   END;
   IF TOKTYPE = IDENTIFIER AND TOKTOP = 1 THEN
      DO;  /* CHECK IF REGISTER NAME */
         IF (C:=TOKBUF(1))='M' THEN RETURN 'M';
         DECLARE X BYTE;
         X = 9;
         IF C='A' THEN X=7;  ELSE
         IF C='B' THEN X=0;  ELSE
         IF C='C' THEN X=1;  ELSE
         IF C='D' THEN X=2;  ELSE
         IF C='E' THEN X=3;  ELSE
         IF C='H' THEN X=4;  ELSE
         IF C='L' THEN X=5;
         LRLVAL = X;
         IF X<>9 THEN RETURN ASL;
      END;
   IF (TOKTYPE = IDENTIFIER) OR (TOKTYPE = STRING) THEN
      DO;
         K = SYMLNEXT + 2;
         J = SAVETOK(K);  /* MOVE TOKEN TO TOP OF SYMLIST */
         DO WHILE TOKCONT <> 0;  /* GET THE REMAINDER OF THE TOKEN */
            CALL GETTOKEN;
            J = SAVETOK(J);
         END;
```

```
            LRLVAL = LOOKUPSYM(K);  /* SEARCH SYMLIST FOR THIS TOKEN */
            IF LRLVAL <> 0 THEN /* ALREADY THERE */
               DO;  /* CHECK WHETHER RESERVED WORD */
                  IF (TOKTYPE = IDENTIFIER) AND (LRLVAL < SLFIRST) THEN
                     /* RESERVED WORD: GET ITS INTERNAL NUMBER */
                     RETURN ( SYMLIST(LRLVAL-3) + 256 );
                  K = LRLVAL;  /* K POINTS TO THE SYMBOL */
               END;
            ELSE /* NEW IDENTIFIER OR STRING */
               DO;  /* INSTALL IT IN SYMLIST */
                  CALL SLINK(K,HASHTAB(HASHCODE));
                  HASHTAB(HASHCODE) = K;
                  SYMLNEXT = J + 1;
               END;
            /* ADJUST LRLVAL SINCE THE SLFIRST BYTES OF
               SYMLIST WON'T BE PASSED TO L82 */
            LRLVAL = K - SLFIRST;
            RETURN TOKTYPE;  /* IDENTIFIER OR STRING */
         END;
      LRLVAL = TOKVAL;
      IF TOKTYPE = NUMBER THEN RETURN NUMBER;
      /* THE INTERNAL RESERVED WORD NO. FOR SINGLE CHARACTERS IS
         THE VALUE OF ITS ASCII REPRESENTATION */
      IF TOKTYPE = SPECIALC THEN RETURN TOKVAL;
      RETURN 0;  /* EOFILE */
      END LRLEX;

TABLE: PROCEDURE(S, I) ADDRESS;
      DECLARE (S, I) ADDRESS;
      /* RETURN I-TH ENTRY IN THE ACTION TABLE OF STATE S */
      RETURN ( LRACT( LRPACT(S+1) + I) );
      END TABLE;

POP: PROCEDURE(N);
      DECLARE N BYTE;
      /* POP N TOPMOST STATES OF THE PARSE STACK */
      PTOP = PTOP - N;
      IF PTOP < 0 THEN /* STACK UNDERFLOW */
         DO;
            CALL PRL(.'STACK UNDERFLOWS');
            CALL EXIT;   /* QUIT */
         END;
      END POP;

PUSH: PROCEDURE(S);
      DECLARE S ADDRESS;
      /* PUSH STATE S INTO THE PARSE STACK */
      PTOP = PTOP + 1;
      IF PTOP >= STACKSIZE THEN /* STACK OVERFLOW */
         DO;
            CALL PRL(.'STACK OVERFLOWS');
            CALL EXIT;  /* QUIT */
         END;
      PSTACK(PTOP) = S;
      END PUSH;

GOTOF: PROCEDURE(STATE,NONTERM) ADDRESS;
      DECLARE (STATE,NONTERM) ADDRESS;
      /* RETURN NEXT STATE AFTER A REDUCTION */
      I = LRPGO(NONTERM); /* POINTER TO GOTO TABLE IN LRGO */
      DO FOREVER; /* TABLE SEARCH */
         IF (LRGO(I) = DEFAULT) OR (LRGO(I) = STATE)
            THEN RETURN LRGO(I+1);
         I = I + 2;
      END;
      END GOTOF;

REDUCE: PROCEDURE;
      /* EXECUTE A REDUCTION */
      DECLARE PN   ADDRESS;          /* PRODUCTION NUMBER */
      DECLARE NRS  ADDRESS;          /* NO. ELEMENTS RIGHT HAND SIDE */
      DECLARE LS   ADDRESS;          /* ID.NO. OF LEFT HAND SIDE */
      PN = TAB AND 0FFFH;
```

```
      /* CLEAR RECOVERY FLAG ONLY IF  NOT <STMT> ::= <ERROR> */
      IF PN<>6 THEN RECOVERING=0;
      NRS = LRR2(PN);
      LS = LRR1(PN);
      /* WRITE THE ACTION ON THE PA FILE */
      CALL WRPA1(XREDUCE);
      CALL WRPA1(LOW(NRS));
      CALL WRPA1(LOW(PN));
      TRACESON
      IF TRACE='Y' THEN DO;
         CALL CMON(PRINT,.'R: S'); CALL PRINTHA(NRS);
         CALL PRC(','); CALL PRINTHA(PN); C=VMON(1,0);
      END;
      TRACESOFF *** */
      /* UPDATE THE PARSE STACK */
      CALL POP(NRS);  /* POP RIGHT HAND SIDE */
      CALL PUSH( GOTOF(PSTACK(PTOP),LS) );   /* NEXT STATE */
      END REDUCE;

SHIFT: PROCEDURE;
      /* EXECUTE A SHIFT ACTION */
      CALL PUSH( TAB AND 0FFFH );   /* PUSH NEXT STATE */
      /* WRITE THE ACTION ON THE PA FILE */
      CALL WRPA1(XSHIFT);
      CALL WRPA2(LRLVAL);
      TRACESON
      IF TRACE='Y' THEN DO;
         CALL CMON(PRINT,.'S: S'); CALL PRINTHA(INPUTSY); CALL PRC(',');
         CALL PRINTHA(LRLVAL); C=VMON(1,0);
      END;
      TRACESOFF *** */
      INPUTSY = LRLEX;   /* GET ANOTHER INPUT SYMBOL */
      END SHIFT;

ACCEPT: PROCEDURE;
      /* WRITE ACCEPT ACTION ON PA FILE */
      CALL WRPA1(XACCEPT);
      /* BREAK PARSER LOOP */
      PARSING = 0;
      END ACCEPT;

ERROR: PROCEDURE;
      /* PARSING ERROR HANDLER */
      TRACESON
      IF TRACE='Y' THEN DO;
         CALL CMON(PRINT,.'E. S');  C=VMON(1,0);
      END;
      TRACESOFF *** */
      IF RECOVERING THEN /* ERROR MESSAGE FOR THIS ERROR ALREADY GIVEN */
         DO; /* SEARCH FOR A STATE WITH SHIFT ON <ERROR> */
            IF PTOP<=0 THEN /* CAN'T POP */
               DO; /* QUIT PARSING */
                  CALL KERROR(0);
                  CALL FRL(.'UNRECOVERABLE ERRORS');
                  CALL EXIT;
               END;
            CALL POP(1);
            RETURN;
         END;
      CALL KERROR(0);
      INPUTSY =  YERROR;     /* <ERROR> */
      RECOVERING = 1;
      END ERROR;

PARSE: PROCEDURE;
      /* PARSE SOURCE PROGRAM */
      INPUTSY = LRLEX;      /* GET FIRST INPUT SYMBOL */
      DO WHILE PARSING;
         IT = 0; /* INDEX WITHIN ACTION TABLE OF CURRENT STATE */
         SEARCHING = 1;
         DO WHILE SEARCHING;  /* IDENTIFY APPLICABLE ACTION */
            SEARCHING = 0;
            TAB = TABLE(PSTACK(PTOP),IT); /* CURRENT TABLE ENTRY */
```

```
            ACTION = TAB AND ACTMASK;      /* DECODE THE ACTION */
            IF ACTION = REDMASK THEN CALL REDUCE;
            ELSE IF ACTION = ERRORMASK THEN CALL ERROR;
                 ELSE IF (INPUTSY OR SYMBMASK) = TAB THEN
                         DO; /* FOUND ENTRY  FOR THIS INPUT */
                            IT = IT + 1; /* LOOK AT CORRESPONDING ACTION*/
                            TAB = TABLE(PSTACK(PTOP),IT);
                            ACTION = TAB AND ACTMASK;
                            IF ACTION = REDMASK THEN CALL REDUCE;
                            ELSE IF ACTION = SHIFTMASK THEN CALL SHIFT;
                                 ELSE CALL ACCEPT;
                         END;
                      ELSE   /* NO MATCH */
                         DO; /* KEEP ON SEARCHING */
                            IT = IT + 2;
                            SEARCHING = 1;
                         END;
         END;
      END;
      END PARSE;

RECERR: PROCEDURE;
      /* PRINT LOAD ADDRESS OF BAD HEX RECORD AND QUIT */
      CALL PRS(.' - REC ADDRESS: $');
      CALL PRINTHB(LAH);
      CALL PRINTHB(LAL);
      CALL EXIT;
      END RECERR;

READHEX: PROCEDURE BYTE;
      /* READ ONE HEX CHARACTER FROM THE PARSER TABLES FILE */
      DECLARE H BYTE;
      IF HEX(H:=GETC) THEN RETURN HEXVAL(H);
      CALL PRL(.'NON-HEX DIGIT IN TABLES $');
      CALL RECERR;
      END READHEX;

READBYTE: PROCEDURE BYTE;
      /* READ TWO HEX DIGITS FROM PARSER TABLES FILE */
      RETURN SHL(READHEX,4) OR READHEX;
      END READBYTE;

READCS: PROCEDURE BYTE;
      /* READ BYTE FROM TABLES FILE WHILE COMPUTING CHECKSUM */
      DECLARE B BYTE;
      CS= CS + (B:=READBYTE);
      RETURN B;
      END READCS;

TLOADER: PROCEDURE;
      /* LOAD ONE PARSER TABLE INTO MEMORY */
      DO FOREVER; /* READ INPUT UNTIL :00XX (END OF TABLE) IS FOUND */
         DO WHILE GETC<>':';            /* LOOK FOR : */
         END;
         /* SET CHECKSUM TO 0, SAVE RECORD LENGTH */
         CS=0;
         IF (RL:=READCS) = 0 THEN /* END OF TABLE */ RETURN;
         LAH=READCS;  LAL=READCS;  /* SAVE LOAD ADDRESS */
         C = READCS;                /* SKIP RECORD TYPE */
         /* NOW START STORING THE BYTES */
         DO WHILE (RL:=RL-1)<>255;
            MEMORY(MEMPTR)= READCS;
            MEMPTR = MEMPTR + 1;
         END;
         /* READ CHECKSUM AND COMPARE */
         IF CS+READBYTE<>0 THEN  /* ERROR */
            DO;  /* TELL OPERATOR AND QUIT */
               CALL PRL(.'CHECKSUM ERROR $');
               CALL RECERR;
            END;
      END;  /* DO FOREVER */
      END TLOADER;
```

```
SLDUMP: PROCEDURE;
      /* DUMP SYMLIST ON THE SL FILE */
      OFCBA = .SLFCB;          /* OUTPUT FILE: SYMBOL LIST FILE */
      OBP = 0;                 /* OUTPUT BUFFER IS EMPTY */
      /* THE PORTION OF SYMLIST CONTAINING RESERVED WORDS
         (BYTES 0 THRU SLFIRST-1) IS NOT PASSED TO L82 */
      I = SLFIRST;
      DO WHILE I<SYMLNEXT;
         CALL PUTC(0);    /* CLEAR LINK FIELD FOR L82 */
         CALL PUTC(0);    /* CLEAR LINK FIELD FOR L82 */
         I = I + 2;    /* INDEX OF 1ST CHAR OF NEXT SYMBOL */
         DO WHILE SYMLIST(I)<>0;
            CALL PUTC(SYMLIST(I));
            I = I + 1;
         END;
         CALL PUTC(0);  /* END OF SYMBOL */
         I = I + 1;
      END;
      CALL PUTC(CONTROLZ);  /* END OF SYMBOL LIST */
      END SLDUMP;

INITTAB: PROCEDURE;
      /* INITIALIZE PARSER TABLES */
      IFCBA = .TABFCB;    /* INPUT FILE: PARSER TABLES */
      /*  OPEN PARSER TABLES FILE */
      CALL CKCON(INITDSK,0);  /* SELECT DISK */
      IF VMON(OPEN,.TABFCB)=255 THEN CALL CANTOP(.TABFCB);
      /* SET NEXT RECORD TO 0 */
      TABFCB(32)=0;
      /* LOAD THE TABLES INTO MEMORY */
      MEMPTR = 0;          /* START LOADING AT "MEMORY" BASE */
      ACTB = .MEMORY;
      CALL TLOADER;     /* LOAD LRACT */
      PACTB = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD LRPACT */
      R1B   = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD LRR1 */
      R2B   = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD LRR2 */
      PGOB  = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD LRPGO */
      GOB   = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD LRGO */
      I = MEMPTR;
      SYMLB = .MEMORY(MEMPTR);
      CALL TLOADER;     /* LOAD SYMLIST */
      SYMLNEXT, SLFIRST = MEMPTR - I;
      SYMLSIZE = FBASE - .MEMORY(I);
      /* RESET HASH TABLE */
      DO I = 0 TO LAST(HASHTAB);
         HASHTAB(I) = 0;
      END;
      /* INITIALIZE THE SYMBOL LIST */
      /* SYMBLIST CONTAINS ONLY THE RESERVED WORDS AT THIS POINT */
      DECLARE J ADDRESS;
      J = 3;  /* INDEX OF FIRST RESERVED WORD */
      DO WHILE J < SLFIRST;
         HASHCODE=0;
         I = J;
         DO WHILE SYMLIST(I) <> 00H;
            CALL HASHF(SYMLIST(I));  /* COMPUTE HASHCODE */
            I = I + 1;
         END;
         CALL SLINK(J,HASHTAB(HASHCODE));
         HASHTAB(HASHCODE)=J;
         J = I + 4;     /* NEXT RESERVED WORD */
      END;
      END INITTAB;

MAIN:
      TRACEON
      CALL PRL(.'TRACE(Y/N):$');    TRACE=VMON(1,0);
      TRACEOFF *** */
```

```
            CALL INITTAB;
            CALL INITFILES;
            CALL PARSE;
            CALL CLOSEF;    /* CLOSE PARSER ACTIONS FILE */
            CALL SLDUMP;
            CALL CLOSEF;    /* CLOSE SYMBOL LIST FILE */
            CALL PRL('ERRORS: ');
            CALL PRINTHB(ERRORCOUNT);
            CALL EXIT;
EOF
```

```
/*
            ****************************************
            *                                      *
            *  L82: PHASE 2 OF THE L80 COMPILER    *
            *                                      *
            *     LUIZ PEDROSO - OCTOBER 1975      *
            *                                      *
            ****************************************
*/

    DECLARE LIT LITERALLY 'LITERALLY';
    DECLARE FOREVER LIT 'WHILE 1';
    DECLARE CR              LITERALLY '0DH';    /* CARRIAGE-RETURN */
    DECLARE LF              LITERALLY '0AH';    /* LINE-FEED */

    /* CP/M SYSTEM CONSTANTS */
    DECLARE CPM   LIT '0';
    DECLARE IFCBA LIT '005CH';
    DECLARE SBUFA LIT '0080H';
    DECLARE FBASE LIT '3200H';
    DECLARE BDOS  LIT '3FFDH';

    /* I/O PRIMITIVES */
    DECLARE READCHAR        LITERALLY  '1';
    DECLARE PRINTCHAR       LITERALLY  '2';
    DECLARE PRINT           LITERALLY  '9';
    DECLARE OPEN            LITERALLY  '15';
    DECLARE CLOSE           LITERALLY  '16';
    DECLARE MAKE            LITERALLY  '22';
    DECLARE DELETE          LITERALLY  '19';
    DECLARE READBF          LITERALLY  '20';
    DECLARE WRITEBF         LITERALLY  '21';
    DECLARE INITDSK         LITERALLY  '13';
    DECLARE SETBUF          LITERALLY  '26';

    /* FILE CONTROL BLOCKS AND BUFFERS FOR OUTPUT FILES */
    /* BYTES 0-32: FCB; BYTE 33: BUFFER POINTER; BYTES 34-161: BUFFER */
    /* CODE AREA FILE */
    DECLARE CAFCB(162) BYTE INITIAL
    (0,'        ','80C',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
    /* INITIAL DATA AREA FILE */
    DECLARE IDAFCB(162) BYTE INITIAL
    (0,'        ','80D',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
    /* RELOCATION TABLE FILE */
    DECLARE RTFCB(162) BYTE INITIAL
    (0,'        ','80R',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

    /* FILE CONTROL BLOCK AND BUFFER FOR INPUT FILES */
    DECLARE IFA ADDRESS INITIAL (IFCBA);        /* FCB ADDRESS */
    DECLARE IFCB BASED IFA BYTE;  /* INPUT FILE CONTROL BLOCK */
    DECLARE IBUFA ADDRESS INITIAL(SBUFA);
    DECLARE IBUF BASED IBUFA BYTE;      /* INPUT BUFFER */
    DECLARE IBP BYTE INITIAL(128);  /* INPUT BUFFER POINTER */

    /* PARSE STACKS */
    DECLARE STACKSIZE LIT '39';
    DECLARE PSH(STACKSIZE) ADDRESS,             /* 'HANDLE' STACK */
            PSX(STACKSIZE) BYTE,                /* 'AUX' STACK    */
            PSY(STACKSIZE) ADDRESS;             /* 'AUXAUX' STACK */
    DECLARE PTOP BYTE INITIAL( 0 );             /* TOP OF PARSE STACKS */

    /* VALUES ASSOCIATED WITH THE LEFT HAND SIDE OF PRODUCTIONS */
    DECLARE (HH,YY) ADDRESS, (LL,XX) BYTE;

    /* VALUES ASSOCIATED WITH HANDLES */
    DECLARE (BASEH,BASEX,BASEY) ADDRESS;      /* BASES FOR A HANDLE */
    DECLARE H BASED BASEH ADDRESS,
            X BASED BASEX BYTE,
            Y BASED BASEY ADDRESS;
    DECLARE (H1,H2,H3,H4,H5) ADDRESS;
    DECLARE (X1,X2,X3,X4) BYTE;
```

```
DECLARE(Y1,Y2,Y3) ADDRESS;
DECLARE L1 LIT 'LOW(H1)',
        L2 LIT 'LOW(H2)',
        L3 LIT 'LOW(H3)',
        L4 LIT 'LOW(H4)',
        L5 LIT 'LOW(H5)';

/* SYMBOL LIST */
DECLARE SYMLIST LIT 'MEMORY';
DECLARE MEMPTR ADDRESS INITIAL (0);

/* RELOCATION TABLE */
DECLARE RELTSIZE LIT '30';
DECLARE RLOC(RELTSIZE) ADDRESS,  /* POINTER TO THE ADDRESS TO RELOC */
        RSB(RELTSIZE) BYTE,      /* SEGMENT (4 BITS), BASE (4 BITS) */
        RDISPL(RELTSIZE) ADDRESS, /* DISPLACEMENT */
        REXT(RELTSIZE) ADDRESS;  /* PTR TO EXTERNAL NAME, 0 IF NOT EXT */
DECLARE RTN BYTE;                /* NO. OCCUPIED ENTRIES IN RELTAB */
DECLARE RTT BYTE    INITIAL (RELTSIZE);       /* CURRENT TOP OF RELTAB */

/* MNEMONICS FOR 'TYPE' */
DECLARE KLABEL   LIT '1',
        KBYTE    LIT '2',
        PROC     LIT '3',
        EXT      LIT '4',
        GLOB     LIT '5',
        STRING   LIT '7',
        UNDECL   LIT '6';

/* MNEMONICS FOR 'BASE' */
DECLARE CA       LIT '1',        /* CODE AREA */
        IDA      LIT '2',        /* INITIAL DATA AREA */
        WA       LIT '3',        /* WORK AREA */
        UNDEF    LIT '4',
        UNUSED   LIT '5';

/* PARSER ACTIONS OPCODES */
DECLARE XREDUCE          LITERALLY '01H';
DECLARE XSHIFT           LITERALLY '02H';
DECLARE XACCEPT          LITERALLY '03H';
DECLARE XLINE            LITERALLY '04H';

/* POINTERS TO NEXT AVAILABLE LOCATIONS FOR CODE EMISSION */
DECLARE CANEXT   ADDRESS INITIAL (0),
        IDANEXT  ADDRESS INITIAL (0),
        WANEXT   ADDRESS INITIAL (0);
DECLARE WASIZE ADDRESS INITIAL(0);      /* MAX VALUE OF WANEXT */
DECLARE IIDA ADDRESS;      /* SAVE VALUE FOR IDANEXT */

/* SYMBOL TABLE */
DECLARE SYMTSIZE LIT '40';
DECLARE STB(SYMTSIZE) BYTE,          /* TYPE(4 BITS), BASE (4 BITS) */
        SDISPL(SYMTSIZE) ADDRESS,    /* DISPLACEMENT */
        SLINK(SYMTSIZE) BYTE,        /* POINTER TO PREVIOUS ENTRY
                                        WITH SAME NAME */
        SATR(SYMTSIZE) ADDRESS,      /* ATRIBUTE */
        SNAME(SYMTSIZE) ADDRESS;     /* POINTER TO NAME IN SYMLIST */
DECLARE SYMTOP BYTE INITIAL (0);     /* POINTER TO SYMTAB TOP */

/* BLOCK LEVEL STACK */
DECLARE BLSTKSIZE LIT '10';
DECLARE BLSTACK(BLSTKSIZE) BYTE;      /* SAVE VALUE FOR SYMTOP */
DECLARE WASTACK(BLSTKSIZE) ADDRESS;   /* SAVE VALUE FOR WANEXT */
DECLARE BLSTKTOP BYTE INITIAL(0);    /* POINTER TO BLSTACK TOP */

/* GLOBAL VARIABLES */
DECLARE COMPILING BYTE INITIAL (1);
DECLARE LINE ADDRESS INITIAL (1);   /* LINE COUNTER */
DECLARE (I,C,TEMP,TEMP1,TEMP2) BYTE;
DECLARE (TEMP3,TEMP4) ADDRESS;

/* SOME OPCODES */
DECLARE JMP      LIT '0C3H';
```

```
DECLARE RET       LIT '0C9H';
DECLARE PCHL      LIT '0E9H';
DECLARE PUSHH     LIT '0E5H';
DECLARE POPH      LIT '0E1H';
DECLARE PUSHD     LIT '0D5H';
DECLARE POPD      LIT '0D1H';
DECLARE SHLD      LIT '022H';
DECLARE XCHG      LIT '0EBH';
DECLARE LXIH      LIT '021H';
DECLARE INXH      LIT '023H';
DECLARE DADD      LIT '019H';
DECLARE MOVEM     LIT '05EH';
DECLARE MOVDM     LIT '056H';

/* SWITCHES FOR BUILT-IN TRACE */
/* TO INCLUDE TRACE ROUTINES, SET TRACESON TO BLANKS */
/* TO EXCLUDE TRACE ROUTINES. SET TRACESON TO SLASH-STAR */
DECLARE TRACESON  LIT ' /* ';
DECLARE TRACESOFF LIT ' /* ';


VMON: PROCEDURE(FUNC,INFO) BYTE;
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; RETURN A VALUE */
      GO TO BDOS;
      END VMON;

CMON: PROCEDURE(FUNC,INFO);
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; NO VALUE RETURNED */
      GO TO BDOS;
      END CMON;

PRC: PROCEDURE(C);
      DECLARE C BYTE;
      /* PRINT CHARACTER C ON THE CRT */
      CALL CMON(PRINTCHAR,C);
      END PRC;

CRLF: PROCEDURE;
      /* SEND CARRIAGE-RETURN/LINE-FEED TO THE CONSOLE */
      CALL PRC(CR);  CALL PRC(LF);
      END CRLF;

PRL: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT LINE STARTING AT ADDRESS A UNTIL A $ IS FOUND */
      CALL CRLF;
      CALL CMON(PRINT,A);
      END PRL;

PRINTH: PROCEDURE(H);
      DECLARE H BYTE;
      /* PRINT A HEXADECIMAL CHARACTER */
      IF (H>9) THEN CALL PRC(H-10+'A');
      ELSE CALL PRC(H+'0');
      END PRINTH;

PRINTHB: PROCEDURE(B);
      DECLARE B BYTE;
      /* PRINT 2 HEXADECIMAL CHARACTERS REPRESENTING B */
      CALL PRINTH(SHR(B,4));
      CALL PRINTH(B AND 0FH);
      END PRINTHB;

PRINTHA: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT 4 HEXADECIMAL CHARACTERS REPRESENTING A */
      CALL PRINTHB(HIGH(A));
      CALL PRINTHB(LOW(A));
      END PRINTHA;
```

```
RBASE: PROCEDURE(J) BYTE;
       DECLARE J BYTE;
       /* RETURN BASE FIELD OF ENTRY J IN RELTAB */
       RETURN RSB(J) AND 0FH;
       END RBASE;

       TRACESON
       DECLARE TRACE BYTE;

TRAA:  PROCEDURE(A); DECLARE A ADDRESS;
       CALL PRC(':'); CALL PRINTHA(A);
       END TRAA;

TRAB:  PROCEDURE(B); DECLARE B BYTE;
       CALL PRC(':'); CALL PRINTHB(B);
       END TRAB;

TRAC:  PROCEDURE(T);
       DECLARE (I,T) BYTE;
       IF TRACE=' ' THEN RETURN;
       CALL PRC(T);
       T = VMON(1,0);
       DO WHILE T<>' ';
          IF T='S' THEN DO I=0 TO SYMTOP;
             CALL TRAB(I);
             CALL TRAB(STB(I));
             CALL TRAA(SDISPL(I));
             CALL TRAB(SLINK(I));
             CALL TRAA(SATR(I));
             CALL TRAA(SNAME(I));
          END;
          IF T='R' THEN DO I=0 TO LAST(RSB);
             IF RBASE(I)<>UNUSED THEN DO;
                CALL TRAB(I);
                CALL TRAA(RLOC(I));
                CALL TRAB(RSB(I));
                CALL TRAA(RDISPL(I));
                CALL TRAA(REXT(I));
             END;
          END;
          IF T='H' THEN DO I=0 TO PTOP;
             CALL TRAB(I); CALL TRAA(PSW(I));
          END;
          IF T='A' THEN DO;
             CALL TRAA(HH); CALL TRAB(LL);
             CALL TRAA(YY); CALL TRAB(XX);
          END;
          T=VMON(1,0);
       END;
       END TRAC;
       TRACESOFF *** */

EXIT:  PROCEDURE;
       /* RETURN TO CP/M */
       CALL PRL('END L82G');
       CALL CRLF;
       GO TO CPM;
       END EXIT;

CANTOP: PROCEDURE(C);
       DECLARE C BYTE;
       /* PRINT OPEN ERROR MESSAGE AND QUIT */
       CALL PRL('CANNOT OPEN $');
       CALL PRC(C);
       CALL EXIT;
       END CANTOP;

CANTCL: PROCEDURE(C);
       DECLARE C BYTE;
       /* PRINT CLOSE ERROR MESSAGE AND QUIT */
       CALL PRL('CANNOT CLOSE $');
       CALL PRC(C);
       CALL EXIT;
```

```
      END CANTCL;

CLOSEALL: PROCEDURE;
      /* CLOSE ALL FILES AND RETURN TO CP/M */
      IF VMON(CLOSE,.CAFCB) = 255 THEN CALL CANTCL('C');
      IF VMON(CLOSE,.IDAFCB)= 255 THEN CALL CANTCL('D');
      IF VMON(CLOSE,.RTFCB) = 255 THEN CALL CANTCL('R');
      CALL EXIT;
      END CLOSEALL;

OPENF: PROCEDURE(Y);
      DECLARE Y BYTE;
      /* OPEN INPUT FILE XXX.80Y */
      /* SET UP INPUT FCB */
      IFCB(9) = '8';
      IFCB(10) = '0';
      IFCB(11) = Y;
      IF VMON(OPEN,IFCBA)=255 THEN CALL CANTOP(Y);
      /* SET NEXT RECORD TO 0 */
      IFCB(32) = 0;
      /* SET BUFFER POINTER */
      IBP = 128;    /* BUFFER EMPTY */
      END OPENF;

MAKEF: PROCEDURE(F);
      DECLARE F ADDRESS;
      /* CREATE OUTPUT FILE WHOSE FCB IS AT F */
      DECLARE FCB BASED F BYTE;
      CALL CMON(DELETE,F);    /* DELETE OLD VERSION OF THE FILE */
      IF VMON(MAKE,F) =255 THEN CALL CANTOP(FCB(11));
      END MAKEF;

KERROR: PROCEDURE(N);
      DECLARE N BYTE;
      /* PRINT ERROR MESSAGE N */
      CALL PRL(.'LINE S');
      CALL PRINTHA(LINE);
      CALL CMON(PRINT,.': ERROR S');
      CALL PRINTHB(N);
      IF (N AND 0F0H)=0F0H THEN /* FATAL ERROR */
            CALL CLOSEALL;  /* CLOSE FILES AND QUIT */
      XX = 0EEH;  /* SET HANDLE TYPE TO 'ERROR' */
      END KERROR;

DOUBLET: PROCEDURE(L,H) ADDRESS;
      DECLARE (L,H) BYTE;
      /* RETURN ADDRESS FORMED BY BYTES L,H */
      RETURN SHL(DOUBLE(H),8) OR L;
      END DOUBLET;

GET1: PROCEDURE BYTE;
      /* READ NEXT CHARACTER FROM INPUT BUFFER;
         IF BUFFER EMPTY: READ ANOTHER RECORD FROM INPUT FILE;
         SET BUFFER POINTER TO 0 IF EOFILE, 1-128 OTHERWISE */
      IF IBP =128 THEN  /* BUFFER EMPTY */
         DO;
            IBP = 0;
            CALL CMON(SETBUF,SBUFA);
            IF VMON(READBF,IFCBA)<>0 THEN /* EOFILE */ RETURN 0;
         END;
      C = IBUF(IBP);  /* NEXT CHARACTER */
      IBP = IBP + 1;
      RETURN C;
      END GET1;

GET2: PROCEDURE ADDRESS;
      /* READ A 16 BIT VALUE FROM THE INPUT BUFFER */
      RETURN DOUBLET(GET1,GET1);
      END GET2;

FLUSH: PROCEDURE(F);
      DECLARE F ADDRESS;
      /* FLUSH BUFFER WHOSE FCB IS AT F */
```

```
      DECLARE FCB BASED F BYTE;  /* FCB(33): BUFFER POINTER */
      IF FCB(33)=0  /* BUFFER EMPTY */ THEN RETURN;
      FCB(33) = 0;
      CALL CMON(SETBUF,.FCB(34));  /* PASS BUFFER ADDRESS TO CP/M */
      IF VMON(WRITEBF,F)<>0 THEN /* UNSUCCESSFUL WRITE */
         DO;   /* WARN OPERATOR AND QUIT */
            CALL PRL(.'WRITE ERRORS');
            CALL CLOSEALL;
         END;
      END FLUSH;

WR: PROCEDURE(S,C);
      DECLARE (S,C) BYTE;
      /* WRITE BYTE C ON FILE S(CA,IDA,RT) */
      DECLARE F ADDRESS;  /* ADDRESS OF AN FCB */
      DECLARE FCB BASED F BYTE;
      DECLARE X BYTE;
      IF S=IDA THEN
         DO;
            TRACESON
            IF TRACE='D' THEN DO;
               CALL PRINTHA(IDANEXT); CALL TRAB(C); CALL TRAC('.');
            END;
            TRACESOFF *** */
            IDANEXT = IDANEXT + 1;
            F =.IDAFCB;
         END;   ELSE
      IF S=CA THEN
         DO;
            TRACESON
            IF TRACE='C' THEN DO;
               CALL PRINTHA(CANEXT); CALL TRAB(C); CALL TRAC('.');
            END;
            TRACESOFF *** */
            CANEXT = CANEXT + 1;
            F = .CAFCB;
         END;
      ELSE F = .RTFCB;
      IF FCB(33)=128 THEN /* BUFFER FULL */     CALL FLUSH(F);
      X = FCB(33);          /* BUFFER POINTER */
      FCB(X+34) = C;   /* PUT C IN THE BUFFER */
      FCB(33) = X + 1;   /* RESET BUFFER POINTER */
      END WR;

WRR1: PROCEDURE(C);
      DECLARE C BYTE;
      /* WRITE C ONTO THE RT FILE */
      CALL WR(0,C);
      END WRR1;

WRR2: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* WRITE A ONTO THE RT FILE */
      CALL WRR1(LOW(A));
      CALL WRR1(HIGH(A));
      END WRR2;

DUMPRT: PROCEDURE(J);
      DECLARE J BYTE;
      /* WRITE ONE RELTAB ENTRY ON THE RT FILE */
      CALL WRR1('R');
      CALL WRR2(RLOC(J));
      CALL WRR2(RDISPL(J));
      CALL WRR2(REXT(J));
      CALL WRR1(RSB(J));
      END DUMPRT;

SLLINK: PROCEDURE(L,LK);
      DECLARE L ADDRESS, LK BYTE;
      /* SET LINK OF SYMBOL L IN SYMLIST TO LK */
      SYMLIST(L-1) = LK;
      END SLLINK;
```

```
LLINK: PROCEDURE (L) BYTE;
       DECLARE L ADDRESS;
       /* RETURN LINK FIELD OF ENTRY L IN SYMLIST */
       RETURN SYMLIST(L-1);
       END LLINK;

SRBASE: PROCEDURE(J,B);
       DECLARE(J,B) BYTE;
       /* SET BASE FIELD OF ENTRY J IN RELTAB */
       RSB(J)=(RSB(J) AND 0F0H) OR B;
       END SRBASE;

SRSEG: PROCEDURE(J,A);
       DECLARE(J,A) BYTE;
       /* SET SEGMENT FIELD OF ENTRY J IN RELTAB */
       RSB(J) = (RSB(J) AND 0FH) OR SHL(A,4);
       END SRSEG;

SRADDR: PROCEDURE(J,B,D);
       DECLARE D ADDRESS, (J,B) BYTE;
       /* SET ADDRESS FIELDS OF ENTRY J IN RELTAB TO (B,D) */
       CALL SRBASE(J,B);
       RDISPL(J) = D;
       END SRADDR;

BRADDR:  PROCEDURE (J,B,D);
       DECLARE D ADDRESS, (J,B) BYTE;
       /* BACKSTUFF B,D ON CHAIN OF ENTRIES IN RELTAB,
          LINKED BY THE EXT FIELD, STARTING AT J */
       DECLARE NEXT BYTE;
       DO WHILE J > 0;
          NEXT = REXT(J);
          CALL SRADDR(J,B,D);
          REXT(J) = 0;    /* NOT EXTERNAL */
          J = NEXT;
       END;
       END BRADDR;

STYPE: PROCEDURE(I) BYTE;
       DECLARE I BYTE;
       /* RETURN TYPE OF SYMBOL I IN SYMTAB */
       RETURN SHR(STB(I),4);
       END STYPE;

SBASE: PROCEDURE (I) BYTE;
       DECLARE I BYTE;
       /* RETURN BASE OF SYMBOL I IN SYMTAB */
       RETURN STB(I) AND 0FH;
       END SBASE;

SSTYPE: PROCEDURE(I,T);
       DECLARE (I,T) BYTE;
       /* SET TYPE OF SYMBOL I IN SYMTAB TO T */
       STB(I) = (STB(I) AND 0FH) OR SHL(T,4);
       END SSTYPE;

SSBASE: PROCEDURE (I,B);
       DECLARE (I,B) BYTE;
       /* SET BASE OF SYMBOL I IN SYMTAB TO B */
       STB(I) = (STB(I) AND 0F0H) OR B;
       END SSBASE;

SSADDR: PROCEDURE (I,B,D);
       DECLARE D ADDRESS, (I,B) BYTE;
       /* SET ADDRESS OF SYMBOL I IN SYMTAB TO (B,D) */
       IF SBASE(I)=UNDEF THEN /* BACKSTUFF (B,D) ON CHAIN OF REFERENCES */
          CALL BRADDR(SDISPL(I),B,D);
       CALL SSBASE(I,B);
       SDISPL(I)=D;
       END SSADDR;

BSTYPE: PROCEDURE(I,T);
       DECLARE (I,T) BYTE;
```

```
      /* BACKSTUFF TYPE T ON CHAIN OF IDENTIFIERS  AT SYMTAB(I) */
      DO WHILE I > 0;
         CALL SSTYPE(I,T);    /* SET TYPE */
         IF T = EXT  THEN CALL SSBASE(I,CA);
         IF T = GLOB THEN CALL SSBASE(I,IDA);
         I = SATR(I);    /* NEXT */
      END;
      END BSTYPE;

BSADATR: PROCEDURE(I,B,D,A);
      DECLARE (D,A) ADDRESS, (I,B) BYTE;
      /* BACKSTUFF BASE B, DISPLACEMENT D, ATRIBUTE A ON A CHAIN
         OF IDENTIFIERS IN SYMTAB, LINKED BY THE ATR FIELD */
      DECLARE J BYTE;
      DO WHILE I > 0;
         D = D - A;  /* DISPLACEMENT IS MODIFIED BY A */
         CALL SSADDR(I,B,D);
         J = SATR(I);          /* SAVE ATRIBUTE=LINK */
         SATR(I) = A;          /* NEW ATRIBUTE */
         I = J;                /* NEXT */
      END;
      END BSADATR;

INCRELT: PROCEDURE(B) BYTE;
      DECLARE B BYTE;
      /* CREATE AN ENTRY IN RELTAB FOR AN ADDRESS TO BE
         RELOCATED IN SEGMENT B (CA OR IDA);
         RETURN POINTER TO THE ENTRY CREATED;
         DUMP RESOLVED ENTRIES ONTO DISK WHILE LOOKING FOR THE VACANCY;
         IF TABLE FULL WITH UNRESOLVED REFERENCES THEN ABEND */
      RTN = 0;      /* NUMBER OF OCCUPIED ENTRIES IN RELTAB */
      DO WHILE RTN  < RELTSIZE;    /* LOOK FOR AN EMPTY ENTRY */
         /* RTT: TOP OF RELTAB */
         IF RTT >= RELTSIZE THEN RTT = 1;    /* WRAP AROUND */
         IF RBASE(RTT) = UNUSED THEN  /* FOUND EMPTY ENTRY */
            DO;  /* SET POINTER TO THE ADDRESS TO RELOCATE */
               IF B=CA THEN RLOC(RTT)=CANEXT;
               ELSE RLOC(RTT) = IDANEXT;
               CALL SRSEG(RTT,B);
               RETURN RTT;
            END;
         IF RBASE(RTT)<>UNDEF THEN /* AN ADDRESS ALREADY RESOLVED */
            DO;
               CALL DUMPRT(RTT); /* WRITE ENTRY ONTO RT FILE */
               CALL SRBASE(RTT,UNUSED);  /* SET ENTRY FREE */
            END;
         ELSE  /* OCCUPIED */
            DO;
               RTN = RTN + 1;
               RTT = RTT + 1;
            END;
      END;  /* OF DO WHILE */
      /* IF GET HERE: RELOCATION TABLE OVERFLOW: QUIT */
      CALL KERROR(0F4H);
      END INCRELT;

FIRST2: PROCEDURE (L) ADDRESS;
      DECLARE L ADDRESS;
      /* RETURN FIRST 2 CHARACTERS OF STRING AT SYMLIST(L) */
      DECLARE (HI,LO) BYTE;
      HI = SYMLIST(L);
      LO = SYMLIST(L+1);
      IF HI = 0 /* EMPTY STRING */ THEN RETURN 0;
      IF LO = 0 /* 1-CHARACTER STRING */ THEN RETURN HI;
      RETURN DOUBLET(LO,HI);
      END FIRST2;

UNDECLARED: PROCEDURE(L) BYTE;
      DECLARE L ADDRESS;
      /* RETURN TRUE IF SYMBOL AT SYMLIST(L) NOT YET DECLARED */
      RETURN LLINK(L) = 0;
      END UNDECLARED;
```

```
NEWSYMB: PROCEDURE(L) BYTE;
      DECLARE L ADDRESS;
      /* RETURN TRUE IF SYMBOL AT SYMLIST(L) IS NOT IN THE
         CURRENT BLOCK */
      RETURN LLINK(L) <= BLSTACK (BLSTKTOP);
      END NEWSYMB;

CREATESYM: PROCEDURE(T,B,D,A,K) BYTE;
      DECLARE (T,B,K) BYTE, (D,A) ADDRESS;
      /* CREATE AN ENTRY IN SYMTAB REFERRING TO SYMBOL AT HANDLE(K);
         RETURN POINTER TO THE ENTRY CREATED */
      DECLARE HK ADDRESS;
      IF (SYMTOP := SYMTOP +1) >= SYMTSIZE THEN /* SYMBOL TABLE OVFLW */
         CALL KERROR(0F2H);
      /* ELSE SET THE SYMBOL */
      CALL SSTYPE(SYMTOP,T);      /* TYPE */
      CALL SSBASE(SYMTOP,B);      /* BASE */
      SDISPL(SYMTOP) = D;         /* DISPL */
      SATR(SYMTOP) = A;           /* ATRIBUTE */
      HK = H(K);  /* POINTER TO SYMBOL NAME IN SYMLIST */
      SLINK(SYMTOP) = LLINK(HK);  /* PREVIOUS ENTRY */
      SNAME(SYMTOP) = HK;   /* POINTER TO NAME */
      CALL SLLINK(HK,SYMTOP);
      RETURN SYMTOP;
      END CREATESYM;

EMITSTR: PROCEDURE(K,S);
      DECLARE (K,S) BYTE;
      /* EMIT IN SEGMENT S (CA, IDA) THE STRING AT HANDLE(K) */
      DECLARE N ADDRESS, C BYTE;
      IF S = CA THEN N=CANEXT; ELSE N=IDANEXT;
      C=CREATESYM(STRING,S,N,0,K);  /* CREATE AN ENTRY IN SYMTAB */
      N = H(K);  /* POINTER TO SYMLIST */
      DO WHILE (C := SYMLIST(N)) <> 0;  /* (NOT END OF STRING) */
         CALL WR(S,C);
         N = N+1;
      END;
      END EMITSTR;

PUSHBL: PROCEDURE;
      /* ENTER A NEW BLOCK */
      /* PUSH SYMTOP, WANEXT INTO THE BLOCK LEVEL STACK */
      IF (BLSTKTOP := BLSTKTOP + 1) >= BLSTKSIZE THEN
         /* BLOCK LEVEL STACK OVFLW */ CALL KERROR(0F3H);
      BLSTACK(BLSTKTOP) = SYMTOP;
      WASTACK(BLSTKTOP) = WANEXT;
      END PUSHBL;

POPBL: PROCEDURE(SHRWA);
      DECLARE SHRWA BYTE;
      /* EXIT FROM A BLOCK: POP SYMTOP, WANEXT FROM THE BLOCK STACK */
      DECLARE I ADDRESS;
      DO WHILE SYMTOP <>BLSTACK(BLSTKTOP);
         /* WRITE A 'SYMBOL' RECORD ONTO THE RT FILE */
         I = SNAME(SYMTOP); /* POINTER TO NAME IN SYMLIST */
         CALL WRR1('S');
         CALL WRR2(SDISPL(SYMTOP));
         CALL WRR2(LINE);
         CALL WRR2(I);
         CALL WRR1(STB(SYMTOP));
         CALL SLLINK(I,SLINK(SYMTOP));
         SYMTOP = SYMTOP - 1;
      END;
      IF SHRWA THEN /* SHRINK WA */ WANEXT = WASTACK(BLSTKTOP);
      BLSTKTOP= BLSTKTOP - 1;  /* POP */
      END POPBL;

AREG: PROCEDURE(K) BYTE;
      DECLARE K BYTE;
      /* TRUE IF HANDLE(K) IS REGISTER A */
      RETURN LOW(H(K)) = 7;
      END AREG;
```

```
BCREG: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS REGISTER BC */
       RETURN LOW(H(K)) = 0FH;
       END BCREG;

DEREG: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS REGISTER DE */
       RETURN LOW(H(K)) = 1FH;
       END DEREG;

HLREG: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS REGISTER HL */
       RETURN LOW(H(K)) = 2FH;
       END HLREG;

SPREG: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS REGISTER SP */
       RETURN LOW(H(K)) = 3FH;
       END SPREG;

CYREG: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS 'CARRY' */
       RETURN LOW(H(K)) = 1EH;
       END CYREG;

STACKR: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF HANDLE(K) IS 'STACK' */
       RETURN LOW(H(K)) = 2EH;
       END STACKR;

BDHSP: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF THE ELEMENT AT HANDLE(K) IS BC,DE,HL,SP */
       RETURN (LOW(H(K)) AND 0FH) = 0FH;
       END BDHSP;

BDH: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF THE ELEMENT AT HANDLE(K) IS BC, DE, HL */
       IF SPREG(K) THEN RETURN 0;
       RETURN BDHSP(K);
       END BDH;

BDHPSW: PROCEDURE(K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF THE ELEMENT AT HANDLE(K) IS BC,DE,HL OR PSW */
       RETURN BDH(K) OR (LOW(H(K)) = 3EH) /* PSW */ ;
       END BDHPSW;

AM:    PROCEDURE (K) BYTE;
       DECLARE K BYTE;
       /* TRUE IF THE ELEMENT AT HANDLE(K) IS A,B,C,...,M */
       RETURN(LOW(H(K))AND 0F8H) =0 ; /* A:7, B:0, ..., M:6  */
       END AM;

MVI: PROCEDURE BYTE;
       /* RETURN MVI OPCODE REFERRING TO REGISTER AT HANDLE(1) */
       RETURN SHL(L1,3) OR 06H;
       END MVI;

LXI: PROCEDURE BYTE;
       /* RETURN LXI OPCODE REFERRING TO REGISTER AT HANDLE(1) */
       RETURN L1 AND 0F1H;
       END LXI;

INCWA: PROCEDURE(N);
       DECLARE N ADDRESS;
```

```
      /* INCREMENT WA BY N */
      WANEXT = WANEXT + N;
      IF WANEXT > WASIZE THEN WASIZE = WANEXT;
      END INCWA;

ZEROXX: PROCEDURE;
      XX = 0;
      END ZEROXX;

XMITX2: PROCEDURE;
      XX = X2;
      END XMITX2;

XMITH1: PROCEDURE;
      HH = H1;
      END XMITH1;

XMIT1: PROCEDURE;
      /* PASS ALONG VALUES OF FIRST ELEMENT OF THE HANDLE */
      CALL XMITH1;
      XX = X1;
      YY = Y1;
      END XMIT1;

XMIT2: PROCEDURE;
      HH = H2;
      CALL XMITX2;
      YY = Y2;
      END XMIT2;

SETREG: PROCEDURE(B);
      DECLARE B BYTE;
      /* ASSIGN INTERNAL REGISTER NUMBERS TO HANDLE(1) */
      LL = B;
      CALL ZEROXX;
      END SETREG;

CANTDO: PROCEDURE;
      /* ISSUE ERROR MESSAGE FOR INVALID REGISTER OPERATIONS */
      CALL ERROR(7);
      END CANTDO;

EMIT1: PROCEDURE(OP);
      DECLARE OP BYTE;
      /* EMIT A 1-BYTE INSTRUCTION */
      CALL WR(CA,OP);
      END EMIT1;

EMIT2: PROCEDURE (OP1,OP2);
      DECLARE (OP1,OP2) BYTE;
      /* EMIT 2 1-BYTE INSTRUCTIONS */
      CALL EMIT1(OP1);
      CALL EMIT1(OP2);
      END EMIT2;

EMIT4: PROCEDURE (OP1,OP2,OP3,OP4);
      DECLARE(OP1,OP2.OP3,OP4) BYTE;
      /* EMIT 4 1-BYTE INSTRUCTIONS */
      CALL EMIT2(OP1,OP2);
      CALL EMIT2(OP3,OP4);
      END EMIT4;

EMIT2N: PROCEDURE(N,B);
      DECLARE N ADDRESS, B BYTE;
      /* EMIT 2 BYTES REPRESENTING N IN AREA B (CA, IDA) */
      CALL WR(B,LOW(N));
      CALL WR(B,HIGH(N));
      END EMIT2N;

EMITN: PROCEDURE (N,B);
      DECLARE N ADDRESS, B BYTE;
      /* EMIT 1 OR 2 BYTES REPRESENTING N IN AREA B (CA, IDA) */
      IF HIGH(N)=0 THEN CALL WR(B,LOW(N));
```

```
      ELSE CALL EMIT2N(N,B);
      END EMITN;

EMIT3D: PROCEDURE (OP,D16);
      DECLARE OP BYTE, D16 ADDRESS;
      /* EMIT A 3-BYTE NON-RELOCATABLE INSTRUCTION */
      CALL EMIT1(OP);
      CALL EMIT2N(D16,CA);
      END EMIT3D;

EMITL: PROCEDURE(OP,L) BYTE;
      DECLARE (OP,L) BYTE;
      /* EMIT A 3-BYTE INSTRUCTION, PROVIDING FOR RELOCATION;
         THE ADDRESS WILL BE BACKSTUFFED LATER, SO CREATE AN
         ENTRY IN RELTAB, LINK IT OT THE CHAIN POINTED TO BY L,
         AND RETURN POINTER TO THE ENTRY   */
      DECLARE J BYTE;
      CALL EMIT1(OP);
      J = INCRELT(CA);  /* POINTER TO NEXT ENTRY IN RELTAB */
      CALL SRBASE(J,UNDEF);
      RDISPL(J)= 0;
      REXT(J) = L;       /* LINK USING THE EXT FIELD IN RELTAB */
      CALL EMIT2(0,0);  /* SLOT FOR THE ADDRESS */
      RETURN J;
      END EMITL;

/* EMIT2D: PROCEDURE(OP,D8);
      EMIT A 2-BYTE NON-RELOCATABLE INSTRUCTION  */
      DECLARE EMIT2D LIT 'EMIT2';

EMITA: PROCEDURE (B,D,E);
      DECLARE B BYTE, (D,E) ADDRESS;
      /* EMIT A RELOCATABLE ADDRESS IN CA */
      DECLARE J BYTE;
      J = INCRELT(CA);  /* POINTER TO NEXT ENTRY IN RELTAB */
      CALL SRBASE(J,B);    /* SET BASE */
      RDISPL(J) = D;       /* SET DISPLACEMENT */
      REXT(J) = E;         /* SET EXTERNAL FIELD */
      CALL EMIT2(0,0);        /* SLOT FOR THE ADDRESS */
      END EMITA;

EMIT3A: PROCEDURE (OP,B,D,E);
      DECLARE (OP,B) BYTE, (D,E) ADDRESS;
      /* EMIT A 3-BYTE RELOCATABLE INSTRUCTION WITH ADDRESS (B,D,E) */
      CALL EMIT1(OP);
      CALL EMITA(B,D,E);
      END EMIT3A;

EMIT2IA: PROCEDURE(L,D,B);
      DECLARE(L,D) ADDRESS, B BYTE;
      /* EMIT IN AREA B (CA,IDA) A RELOCATABLE ADDRESS EQUAL TO
         THE ADDRESS OF SYMBOL L IN SYMLIST PLUS DISPLACEMENT D */
      DECLARE (I,J,K) BYTE;
      IF UNDECLARED(L) THEN /* CANNOT REFERENCE */
         CALL KERROR(4);
      ELSE
         DO;
            J= INCRELT(B);    /* POINTER TO AN EMPTY ENTRY IN RELTAB */
            I = LLINK(L);     /* POINTER INTO SYMTAB */
            K = SBASE(I);     /* GET BASE OF SYMBOL I */
            CALL SRBASE(J,K);        /* SET BASE OF RELOCATABLE ADDRESS */
            IF K=UNDEF THEN /* ADDRESS NOT YET DEFINED */
               DO; /* LINK THE ENTRY IN RELTAB TO A CHAIN OF
                      UNRESOLVED REFERENCES TO THE IDENTIFIER */
                  REXT(J) = SDISPL(I);
                  SDISPL(I) = J;  /* POINTER TO THE CHAIN */
                  RDISPL(J) = D;   /* RELATIVE DISPLACEMENT */
               END;
            ELSE  /* ADDRESS ALREADY KNOWN */
               DO;
                  RDISPL(J) = SDISPL(I) + D;   /* DISPLACEMENT */
                  K = STYPE(I);    /* GET TYPE OF SYMBOL I */
                  IF K=GLOB OR K=EXT THEN REXT(J) = L;
```

```
                ELSE REXT(J) = 0;  /* NOT EXTERNAL */
            END;
        CALL EMIT2N(0,B);  /* SLOT FOR THE ADDRESS */
    END;
    END EMIT2IA;

EMIT3IA: PROCEDURE (OP,L,D);
    DECLARE OP BYTE,(L,D) ADDRESS;
    /* EMIT A 3-BYTE INSTRUCTION WITH RELOCATABLE ADDRESS
       EQUAL TO ADDRESS OF SYMBOL L IN SYMLIST PLUS DISPL D */
    CALL   EMIT1(OP);
    CALL EMIT2IA(L,D,CA);  /* EMIT ADDRESS IN CA */
    END EMIT3IA;

EMIT2SA: PROCEDURE(K,B);
    DECLARE (K,B) BYTE;
    /* EMIT IN AREA B (CA,IDA) A RELOCATABLE ADDRESS EQUAL TO
       THE ADDRESS OF THE STRING AT HANDLE(K) */
    DECLARE (I,J) BYTE;
    IF  UNDECLARED(H(K)) /* STRING NOT YET EMITTED */ THEN
       CALL EMITSTR(K,IDA);  /* EMIT IT IN IDA */
    /* STRING ALREADY EMITTED */
    J = INCRELT(B);  /* POINTER TO AN ENTRY IN RELTAB */
    I = LLINK(H(K));   /* POINTER TO STRING ENTRY IN SYMTAB */
    CALL SRBASE(J,SBASE(I));  /* SET BASE IN RELTAB */
    RDISPL(J)=SDISPL(I);   /* SET DISPLACEMENT */
    REXT(J) = 0;       /* NOT EXTERNAL */
    CALL EMIT2N(0,B);  /* SLOT FOR THE ADDRESS */
    END EMIT2SA;

EMIT3SA: PROCEDURE(OP,K);
    DECLARE (OP,K) BYTE;
    /* EMIT A 3-BYTE INSTRUCTION WITH RELOCATABLE ADDRESS
       EQUAL TO THE ADDRESS OF THE STRING AT HANDLE(K)   */
    CALL EMIT1(OP);
    CALL EMIT2SA(K,CA);  /* EMIT ADDRESS IN CA */
    END EMIT3SA;

EMIT3C: PROCEDURE(OP,K);
    DECLARE (OP,K) BYTE;
    /* EMIT A 3-BYTE INSTRUCTION WITH ADDRESS EQUAL TO THE
       <CONSTANT> AT HANDLE(K) */
    DECLARE HK ADDRESS, XK BYTE;
    HK= H(K);
    XK = X(K);
    IF XK=11H THEN /* CONSTANT IS A STRING */
       CALL EMIT3D(OP,FIRST2(HK));
    IF XK=13H THEN /* CONSTANT IS A NUMBER */
       CALL EMIT3D(OP,HK);
    IF XK=15H THEN /* CONSTANT IS ADDRESS OF IDENTIFIER */
       CALL EMIT3IA(OP,HK,Y(K));
    IF XK=17H THEN /* CONSTANT IS ADDRESS OF STRING */
       CALL EMIT3SA(OP, K);
    END EMIT3C;

LDAHL: PROCEDURE(OP,K) BYTE;
    DECLARE(OP,K) BYTE;
    /* EMIT INSTRUCTION OP (LDA,LHLD,STA,SHLD) FOR THE CONSTRUCTS:
       <REG> = <VAR>
       <VAR> = <REG>
       K: POINTER TO <VAR> IN THE HANDLE
       RETURN 1 IF INSTRUCTION IS EMITTED (<VAR> NOT IN ERROR) */
    DECLARE XK BYTE, HK ADDRESS;
    XK = X(K);
    HK = H(K);
    IF XK = 23H      /* <VAR> IS <IDENTIFIER>(<NUMBER>)       */
       OR XK = 2DH   /* <VAR> IS H(.<IDENTIFIER>(<NUMBER>))   */
       THEN CALL EMIT3IA(OP,HK,Y(K));      ELSE
    IF XK = 29H      /* <VAR> IS M(<STRING>) */
       THEN CALL EMIT3D (OP,FIRST2(HK));  ELSE
    IF XK = 2BH      /* <VAR> IS M(<NUMBER>) */
       THEN CALL EMIT3D (OP,HK);          ELSE
    IF XK = 2FH      /* <VAR> IS M(.<STRING>) */
```

```
      THEN CALL EMIT3SA(OP, K);
   ELSE /* DID NOT EMIT INSTRUCTION */  RETURN 0;
   RETURN 1;
   END LDAHL;

LOADPRIM: PROCEDURE(J);
   DECLARE J BYTE;
   /* SEMANTIC ACTIONS FOR THE CONSTRUCT:
      <REG> = ... ... <PRIM>
      J: INDEX OF <PRIM> IN THE HANDLE */
   DECLARE  XJ BYTE, HJ ADDRESS;
   XJ = X(J);
   HJ = H(J);
   C = XJ AND 0F0H; /* GET THE TYPE OF <PRIM> */
   IF(C = 00H )      /* <PRIM> IS A REGISTER */       THEN
      DO;
      IF LOW(H1)=LOW(HJ) /* SAME AS <REG> */ THEN RETURN;
      /* ELSE: DIFFERENT REGISTERS */
         IF AM(1) AND AM(J) THEN /* EMIT A MOVE INSTRUCTION */
            CALL EMIT1(40H OR LOW(SHL(H1,3)) OR LOW( HJ)); ELSE
         IF STACKR(1)   /* <REG> IS THE STACK */
            AND BDHPSW(J) THEN /* EMIT A PUSH */
            CALL EMIT1(  (LOW( HJ ) AND 0F0H) OR 0C5H);     ELSE
         IF STACKR(J)   /* <PRIM> IS THE STACK */
            AND BDHPSW(1) THEN /* EMIT A POP  */
            CALL EMIT1(  (LOW(H1) AND 0F0H) OR 0C1H);       ELSE
         IF SPREG(1) AND HLREG(J)  /* SP = HL */ THEN
            CALL EMIT1(0F9H);  /* EMIT SPHL */       ELSE
         IF BDH(1) AND BDH(J) THEN /* BC=DE, BC=HL, ETC. */
            DO;
               C = (L1 XOR 4FH) OR SHR(LOW(HJ) AND 0F0H, 3);
               CALL EMIT2(C,C+9);  /* B=D, C=E, D=H, C=L, ETC. */
            END;
         ELSE  /* ERROR */
            CALL CANTDO;
         RETURN;
      END;  /* OF <PRIM> IS A REGISTER */
   IF C = 10H THEN /* <PRIM> IS A CONSTANT */
      DO;
         IF XJ = 13H THEN /* <PRIM> IS A NUMBER */
            DO;
               IF AM(1) THEN /* EMIT A MOVE IMMEDIATE */
                  CALL EMIT2D( MVI, LOW(HJ) );   ELSE
               IF BDHSP(1) THEN /* EMIT A LOAD EXTENDED IMMEDIATE */
                  CALL EMIT3D( LXI,HJ);         ELSE
               IF CYREG(1) THEN /* <REG> IS CY */
                  DO;
                     IF HJ=1 THEN /* CY=1 */
                        CALL EMIT1(37H);   /* EMIT STC */  ELSE
                     IF HJ=0 THEN /* CY=0 */
                        CALL EMIT2(37H,3FH); /* EMIT STC,CMC */
                     ELSE /* ERROR */ CALL CANTDO;
                  END;  /* OF <REG> IS CY */
               ELSE  /* ERROR (BAD REGISTER) */  CALL CANTDO;
               RETURN;
            END;  /* OF <PRIM> IS NUMBER */
         IF XJ = 11H  THEN /* <PRIM> IS A STRING */
            DO;
               IF AM(1) THEN /* EMIT A MOVE IMMEDIATE */
                  CALL EMIT2D(MVI,SYMLIST(HJ)); /* 1ST CHAR */ ELSE
               IF BDHSP(1) THEN /* EMIT A LOAD EXTENDED IMMEDIATE */
                  CALL EMIT3D(LXI,FIRST2(HJ));  /* 1ST 2 CHARACTERS */
               ELSE /* ERROR */ CALL CANTDO;
               RETURN;
            END;  /* OF <PRIM> IS A STRING */
         IF BDHSP(1) THEN
            DO;
               IF XJ=15H THEN /* <PRIM> IS .<IDENTIFIER>(<NUMBER>) */
                  CALL EMIT3IA(LXI,HJ,Y(J));         ELSE
               IF XJ=17H THEN /* <PRIM> IS .<STRING> */
                  CALL EMIT3SA(LXI, J);  /* LOAD EXT IMMEDIATE */
               /* ELSE(ERROR): DO NOTHING - ERROR MSG ALREADY ISSUED */
            END; /* BDHSP(1) */
```

```
            ELSE /* ERROR (BAD REGISTER) */ CALL CANTDO;
            RETURN;
         END; /* OF <PRIM> IS A CONSTANT */
      IF C = 20H THEN  /* <PRIM> IS A VARIABLE */
         DO;
            IF AREG(1) THEN /* <REG> IS REGISTER A */
               DO;
                  IF XJ=21H THEN /* <PRIM> IS M(BC) OR M(DE) */
                     /* EMIT LDAX B, LDAX D */
                     CALL EMIT1 (LOW(HJ) AND 0FAH); ELSE
                  IF XJ=25H THEN /* <PRIM> IS IN(<NUMBER>)    */
                     /* EMIT 'IN' INSTRUCTION */
                     CALL EMIT2D(0DBH,LOW(HJ));      ELSE
                  /* TRY TO EMIT A LDA INSTRUCTION */
                  IF LDAHL(3AH,J) THEN /* EMITTED LDA */ RETURN;
                  ELSE /* ERROR */ CALL CANTDO;
                  RETURN;
               END;  /* OF <REG> IS REGISTER A */
            IF HLREG(1) THEN /* <REG> IS HL */
               DO;
                  /* TRY TO EMIT A LHLD INSTRUCTION */
                  IF LDAHL(2AH,J) THEN /* EMITTED LHLD */ RETURN;
                  /* ELSE: ERROR */
               END;  /* OF <REG> IS HL */
         END;  /* OF <PRIM> IS A VARIABLE */
      /* IF NONE OF THE ABOVE: ERROR */
      CALL CANTDO;
      END LOADPRIM;

BINOPSEC: PROCEDURE(J,K);
      DECLARE(J,K) BYTE;
      /* CODE GENERATION FOR THE CONSTRUCT:
         <REG> = ... <BINARY.OP> <SEC>
         J,K: POINTERS TO <BINARY OP>, <SEC>   IN THE HANDLE */
      DECLARE (XJ,XK) BYTE, HK ADDRESS;
      XJ = X(J);
      XK = X(K);
      HK = H(K);
      C = XK AND 0F0H;  /* GET THE TYPE OF <SEC> */
      IF  C=00H THEN /* <SEC> IS A REGISTER */
          DO;
             IF AREG(1) THEN /* <REG> IS REG A */
                DO;
                   IF AM(K) THEN /* <SEC> IS A,B... M */
                     DO; /* GENERATE 'ACCUMULATOR' INSTRUCTION */
                        CALL EMIT1( XJ  OR LOW( HK) );
                        RETURN;
                     END;
                  /* ELSE: ERROR */
                     CALL CANTDO;
                     RETURN;
                END;  /* OF <REG> IS A */
             IF HLREG(1) THEN /* <REG> IS HL */
                DO;
                   IF BDHSP( K )  AND  XJ = 80H THEN
                      /* <SEC> IS BC,DE,HL,SP; <BINARY.OP> IS + */
                      DO; /* EMIT DAD B, DAD D, DAD H, DAD SP */
                         CALL EMIT1(LOW( HK ) AND 0F9H);
                         RETURN;
                      END;
                   /* ELSE: ERROR */
                      CALL CANTDO;
                      RETURN;
                END; /* OF <REG> IS HL */
             /* ELSE (<REG> NEITHER A NOR HL): ERROR */
                CALL CANTDO;
                RETURN;
          END;  /* OF <SEC> IS A REGISTER */
      IF C=10H THEN  /* <SEC> IS A CONSTANT */
         DO;
            IF AM(1) OR BDHSP(1) THEN  /* A,B,...M,BC,DE,HL,SP */
               DO;
                  IF  XK  = 13H /* NUMBER */ AND  HK =1 THEN
```

```
                        DO;
                            IF  XJ  = 80H THEN /* <REG> + 1 */
                               DO;
                                  IF AM(1) THEN /* A,B,...M */
                                     CALL EMIT1(SHL(L1,3) OR 4); /* INR OPS */
                                  ELSE /* BC,DE,HL,SP */
                                     CALL EMIT1(L1 AND 0F3H);    /* INX OPS */
                                  RETURN;
                               END;  /* OF <REG> + 1 */
                            IF  XJ  = 90H THEN /* <REG> - 1 */
                               DO;
                                  IF AM(1) THEN /* A,B,...M */
                                     CALL EMIT1(SHL(L1,3) OR 5); /* DCR OPS */
                                  ELSE /* BC,DE,HL,SP */
                                     CALL EMIT1(L1 AND 0FBH);    /* DCX OPS */
                                  RETURN;
                               END; /* OF <REG> - 1 */
                         END;  /* OF NUMBER, 1 */
                      IF NOT AREG(1) THEN /* ERROR: <REG> IS NOT A */
                         DO;
                            CALL CANTDO;
                            RETURN;
                         END;
                      /* ELSE: <REG> IS REG A */
                         IF  XK =11H /* STRING */ THEN
                            C = SYMLIST(HK ); /* GET ITS FIRST CHARACTER */
                         IF  XK =13H /* NUMBER */ THEN
                            C = HK;      /* GET ITS VALUE */
                         IF  XK >13H /* ADDRESS: NOT IMPLEMENTED */ THEN
                            CALL KERROR(8);
                         ELSE /* EMIT ACC IMMEDIATE GROUP */
                            CALL EMIT2D( XJ  OR 46H,C);
                         RETURN;
                   END; /* OF A,B,...M,BC,DE,HL,SP */
                /* ELSE (<REG> IS PSW,STACK,CY):ERROR */
                CALL CANTDO;
             END; /* OF <SEC> IS A CONSTANT */
          /* ELSE: <SEC> IS ERROR: DO NOTHING (ERROR ALREADY PROCESSED) */
          END BINOPSEC;

IDN:  PROCEDURE(K,N);
          DECLARE K BYTE, N ADDRESS;
          /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
             <VAR> ::= <IDENTIFIER> ( <NUMBER> )
             <CONSTANT> ::= . <IDENTIFIER> ( <NUMBER> )
             K: INDEX OF <IDENTIFIER> IN THE HANDLE
             N: VALUE OF <NUMBER>     */
          DECLARE HK ADDRESS;
          HK = H(K);
          IF UNDECLARED(HK) THEN /* ERROR: CANNOT BE REFERENCED */
                CALL KERROR(4);
          YY = N;              /* TRANSMIT <NUMBER> */
          HH = HK;             /* TRANSMIT POINTER TO IDENTIFIER */
          END IDN;

CHC:  PROCEDURE(B);
          DECLARE B BYTE;    /* BASE (CA OR IDA ) */
          /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
             <DATA.LIST> ::= <DATA.HEAD> <CONSTANT> )
             <DATA.HEAD> ::= <DATA.HEAD> <CONSTANT> ,
             <INITIAL.LIST> ::= <INITIAL.HEAD> <CONSTANT> )
             <INITIAL.HEAD> ::= <INITIAL.HEAD> <CONSTANT> ,       */
          IF X2=11H THEN  /* CONSTANT IS A STRING */
             CALL EMITSTR( 2,B);  /* EMIT THE STRING IN CA OR IDA */      ELSE
          IF X2=13H THEN /* CONSTANT IS A NUMBER */
             CALL EMITH(H2,B);       /* EMIT THE NUMBER IN CA OR IDA */   ELSE
          IF X2=15H THEN  /* CONSTANT IS ADDRESS OF IDENTIFIER */
             CALL EMIT2IA(H2,Y2,B);  /* EMIT THE ADDRESS IN CA OR IDA */ ELSE
          IF X2=17H THEN /* CONSTANT IS ADDRESS OF STRING */
             CALL EMIT2SA( 2,B);  /* EMIT THE ADDRESS IN CA OR IDA */
          /* ELSE: ERROR - DO NOTHING - ERROR MESSAGE ALREADY ISSUED */
          CALL XHIT1;
          END CHC;
```

```
FPH: PROCEDURE;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
         <FORMAL.PARAM.LIST> ::= <FP.HEAD> <IDENTIFIER> )
         <FP.HEAD>           ::= <FP.HEAD> <IDENTIFIER> ,       */
      IF NEWSYMB(H2) THEN
         /* FIRST APPEARANCE OF IDENTIFIER IN CURRENT BLOCK */
         DO;
            C    = CREATESYM(KBYTE,WA,WANEXT,2,2);
            CALL INCWA(2);          /* ASSIGN 2 BYTES IN WA */
            XX = X1 + 1;    /* NO. FORMAL PARAMETERS */
         END;
      ELSE /* IDENTIFIER ALREADY DEFINED IN CURRENT BLOCK */
         DO;
            CALL KERROR(2);
            CALL XMIT1;  /* IGNORE THE IDENTIFIER */
         END;
      END FPH;

PRH: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
         <PROC.HEAD> ::= <PROC.NAME> ;
         <PROC.HEAD> ::= <PROC.NAME> <FORMAL.PARAM.LIST> ;          */
            CALL SSTYPE(L1,PROC);
            SATR(L1) = A;              /* ADDRESS OF FORMAL PARAMETERS */
            CALL XMIT1;
      END PRH;

SDB: PROCEDURE(L);
      DECLARE L ADDRESS;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
         <STORAGE.DECLARATION> ::= <IDENT.SPECIFICATION> BYTE
         <STORAGE.DECLARATION> ::= <BOUND.HEAD> <NUMBER> ) BYTE    */
      CALL XMIT1;
      CALL BSTYPE(L1,KBYTE);  /* BACKSTUFF TYPE */
      YY = L;                 /* LENGTH REQUESTED PER IDENTIFIER */
      END SDB;

IDL: PROCEDURE;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
      <IDENT.SPECIFICATION> ::= <IDENT.LIST> <IDENTIFIER> )
      <IDENT.LIST>          ::= <IDENT.LIST> <IDENTIFIER> ,      */
      IF NEWSYMB(H2) THEN
         /* FIRST APPEARANCE OF IDENTIFIER IN CURRENT BLOCK */
         DO;
            LL = CREATESYM(UNDECL,UNDEF,0,H1.2);
            XX = X1 + 1;       /* NO. IDENTIFIERS */
         END;
      ELSE /* IDENTIFIER ALREADY DEFINED IN CURRENT BLOCK */
         DO;
            CALL KERROR(2);
            CALL XMIT1;    /* IGNORE REDECLARATION */
         END;
      END IDL;

CANTCALL: PROCEDURE(J,N) BYTE;
      DECLARE J ADDRESS, /* POINTER TO IDENTIFIER IN SYMLIST */
              N BYTE;    /* NO. PARAMETERS */
      /* RETURN TRUE IF CANNOT CALL IDENTIFIER WITH N PARAMETERS */
      IF UNDECLARED(J) THEN RETURN 1;
      IF(C:=STYPE(LLINK(J)))=KLABEL AND N<>0 THEN RETURN 1;
      IF (C=KBYTE) OR (C=GLOB) THEN RETURN 1;
      RETURN 0;
      END CANTCALL;

APH: PROCEDURE;
      /* SEMANTIC ACTIONS FOR THE PRODUCTIONS:
         <ACTUAL.PARAM.LIST> ::= <AP.HEAD> <CONSTANT> )
         <AP.HEAD> ::= <AP.HEAD> <CONSTANT> ,     */
      CALL EMITOC(LXIH,2);    /* LOAD THE CONSTANT INTO HL */
      IF STYPE(L1) = EXT THEN /* EXTERNAL PROCEDURE */ TEMP3=SNAME(L1);
      ELSE TEMP3=0;  /* TEMP3 WILL BE MOVED TO THE EXT FIELD IN RELTAB */
```

```
      CALL EMIT3A(SHLD,WA,SATR(L1)+SHL(X1,1),TEMP3);/* STORE PARAMETER */
      /* SATR(H1) + 2*X1 = ADDRESS OF CORRESPONDING FORMAL PARAMETER */
      X1 = X1 +1;   /* NO.ACTUAL PARAMETERS */
      CALL XMIT1;
      END APH;

REDUCE: PROCEDURE (PN);
      DECLARE PN ADDRESS;
      /* PERFORM A REDUCTION USING PRODUCTION PN */
      TRACESON
      IF TRACE='P' THEN DO;
         CALL PRC('P'); CALL TRAA(PN); CALL TRAC(';');
      END;
      TRACESOFF *** */
      IF PN<30 THEN RETURN; /* NO ACTION TO BE PERFORMED */
      DO CASE PN-30;

/*** <PROGRAM> ::= <STMT.LIST> ; EOF */

/*** <STMT.LIST> ::= <STMT> */

/*** <STMT.LIST> ::= <STMT.LIST) ; <STMT> */

/*** <STMT> ::= <BASIC.STMT> */

/*** <STMT> ::= <IF.STMT> */

/*** <STMT> ::= <ERROR> */

/*** <BASIC.STMT> ::= <DECL.STMT> */

/*** <BASIC.STMT> ::= <GROUP> */

/*** <BASIC.STMT> ::= <PROC.DEFINITION> */

/*** <BASIC.STMT> ::= <RETURN.STMT> */

/*** <BASIC.STMT> ::= <CALL.STMT> */

/*** <BASIC.STMT> ::= <GOTO.STMT> */

/*** <BASIC.STMT> ::= <REPEAT.STMT> */

/*** <BASIC.STMT> ::= <CONTROL.STMT> */

/*** <BASIC.STMT> ::= <COMPARE.STMT> */

/*** <BASIC.STMT> ::= <EXCHANGE.STMT> */

/*** <BASIC.STMT> ::= <ASSIGN.STMT> */

/*** <BASIC.STMT> ::= <LABEL.DEFINITION> <BASIC.STMT> */

/*** <IF.STMT> ::= <LABEL.DEFINITION> <IF.STMT> */

/*** <DECL.STMT> ::= DECLARE <DECL.ELEMENT> */

/*** <DECL.STMT> ::= <DECL.STMT> , <DECL.ELEMENT> */

/*** <INITIAL.HEAD> ::= INITIAL ( */

/*** <ENDING> ::= END */

/*** <ENDING> ::= END <IDENTIFIER> */

/*** <ENDING> ::= <LABEL.DEFINITION> <ENDING> */

/*** <INITIALIZATION>::= <EMPTY> */

/*** <INITIALIZATION> ::= <ASSIGN.STMT> */

/*** <ASSIGN.STMT> ::= <VAR.ASSIGN> */
```

```
/*** <ASSIGN.STMT> ::= <REG.ASSIGN> */

/*** <LABEL.DEFINITION> ::= <IDENTIFIER> : */
        /* RETURN: LL = POINTER TO IDENTIFIER ENTRY IN SYMTAB;
                   XX = 1 TO INDICATE IDENTIFIER LABEL */
        DO;
           XX = 1;
           IF NEWSYMB( H1 ) THEN
              /* FIRST APPEARANCE OF IDENTIFIER IN CURRENT BLOCK */
              LL = CREATESYM (KLABEL,CA,CANEXT,0,1);
           ELSE
              /* IDENTIFIER ALREADY DEFINED IN CURRENT BLOCK */
              DO;
                 H1 = LLINK(H1);    /* POINTER INTO SYMTAB */
                 IF STYPE( L1 )<>KLABEL THEN
                    /* CONFLICTING DECLARATION */
                    DO;
                       CALL KERROR(1);
                    END;
                 ELSE /* IDENTIFIER ALREADY DECLARED LABEL */
                    DO;
                       IF SBASE( L1 )<>UNDEF THEN
                          /* CONFLICTING DEFINITION */
                          DO;
                             CALL KERROR(2);
                          END;
                       ELSE /* DEFINE THE LABEL */
                          DO;
                             CALL SSADDR( L1 ,CA,CANEXT);
                             CALL XMIT1;
                          END;
                    END;
              END;
        END;

/*** <LABEL.DEFINITION> ::= <NUMBER> : */
        DO;
           CALL ZEROXX;    /* TO INDICATE NUMERIC LABEL */
           IF H1  <CANEXT THEN /* CODE OVERLAP */
              CALL KERROR(3);
           DO WHILE CANEXT < H1;
              CALL WR(CA,00H);  /* FILL WITH 0'S */
           END;
        END;

/*** <IF.STMT> ::= <IF.CLAUSE> <STMT> */
           /*  H1  POINTS TO F(FALSE) CHAIN IN RELTAB */
           CALL BRADDR( L1 ,CA,CANEXT); /* BACKSTUFF F ADDRESS */

/*** <IF.STMT> ::= <IF.CLAUSE> <TRUE.PART> <STMT> */
        DO;
           /*  H2  POINTS TO E(EXIT) ENTRY IN RELTAB */
           CALL BRADDR( L1 ,CA,RLOC( L2 )+2);  /* BACKSTUFF F */
           CALL BRADDR( L2,CA,CANEXT);         /* BACKSTUFF E */
        END;

/*** <IF.CLAUSE> ::= IF <COMPOUND.CONDITION> THEN */
        /* RETURN POINTER TO F CHAIN IN RELTAB */
           LL = L2;

/*** <TRUE.PART> ::= <BASIC.STMT> ELSE */
        /* RETURN POINTER TO E ENTRY IN RELTAB */
           LL = EMITL(JMP,0);  /* EMIT JMP E (EXIT) */

/*** <COMPOUND.CONDITION> ::= <AND.HEAD> <SIMPLE.CONDITION> */
        /* RETURN POINTER TO F CHAIN IN RELTAB */
           LL = EMITL(X2 XOR 0AH,L1);      /* EMIT JNCOND F */

/*** <COMPOUND.CONDITION> ::= <OR.HEAD> <SIMPLE.CONDITION> */
        DO;
           LL = EMITL(X2 XOR 0AH,0);       /* EMIT JNCOND F */
```

```
            CALL BRADDR(L1,CA,CANEXT);      /* BACKSTUFF T */
         END;

/*** <COMPOUND.CONDITION> ::= <SIMPLE.CONDITION> */
            LL = EMITL(X1 XOR 0AH,0);       /* EMIT JNCOND F */

/*** <AND.HEAD> ::= <SIMPLE.CONDITION> AND */
         /* RETURN POINTER TO F CHAIN IN RELTAB */
            LL = EMITL(X1 XOR 0AH,0);       /* EMIT JNCOND F */

/*** <AND.HEAD> ::= <AND.HEAD> <SIMPLE.CONDITION> AND */
            LL = EMITL(X2 XOR 0AH,L1);      /* EMIT JNCOND F */

/*** <OR.HEAD> ::= <SIMPLE.CONDITION> OR */
        /* RETURN POINTER TO T CHAIN IN RELTAB */
            LL = EMITL(X1 OR 02H,0);        /* EMIT JCOND T */

/*** <OR.HEAD> ::= <OR.HEAD> <SIMPLE.CONDITION> OR */
            LL = EMITL(X2 OR 02H,L1);       /* EMIT JCOND T */

/*** <SIMPLE.CONDITION> ::= ( <STMT.LIST> ) <CONDITION> */
         /* RETURN A NUMERIC CODE REPRESENTING THE CONDITION */
            XX = X4;

/*** <SIMPLE.CONDITION> ::= <CONDITION> */
            CALL XMIT1;

/*** <CONDITION> ::= NOT ZERO */
            XX = 0C0H;

/*** <CONDITION> ::= ZERO */
            XX = 0C8H;

/*** <CONDITION> ::= NOT CY */
            XX = 0D0H;

/*** <CONDITION> ::= CY */
            XX = 0D8H;

/*** <CONDITION> ::= PY ODD */
            XX = 0E0H;

/*** <CONDITION> ::= PY EVEN */
            XX = 0E8H;

/*** <CONDITION> ::= PLUS */
            XX = 0F0H;

/*** <CONDITION> ::= MINUS */
            XX = 0F8H;

/*** <DECL.ELEMENT> ::= <STORAGE.DECLARATION> */
         /* <STORAGE.DECLARATION> RETURNS:
            H1: POINTER TO A CHAIN OF IDENTIFIERS IN SYMTAB,
            X1: NO. IDENTIFIERS DECLARED,
            Y1: LENGTH (BYTES) REQUESTED PER IDENTIFIER;
            IT ALSO SETS THE GLOBAL VARIABLE
            IIDA: IDANEXT BEFORE <DECL.ELEMENT> WAS PARSED      */
         DO;
            TEMP3 = IDANEXT - IIDA;  /* TOTAL LENGTH INITIALIZED */
            TEMP4,I = 0;
            DO I = 1 TO X1;
               TEMP4=TEMP4+ Y1;  /* TOTAL LENGTH REQUESTED */
            END;
            IF TEMP3>TEMP4  THEN      /* INITIAL DATA TOO LONG */
               DO;
                  CALL KERROR(0);
                  TEMP4=TEMP3;   /* RIGHT JUSTIFY INSIDE DATA BLOCK */
               END;
            IF TEMP3 = 0 THEN /* NO DATA INITIALIZED */
               DO;  /* ALLOCATE IN WA */
                  CALL INCWA(TEMP4);
                  CALL BSADATR(L1,WA,WANEXT,Y1);
```

```
            END;
         ELSE /* INITIALIZATION REQUESTED (DATA IS IN IDA) */
            DO;
               IIDA = IIDA + TEMP4;
               DO WHILE IDANEXT < IIDA;
                  CALL WR(IDA,' ');  /* PAD WITH BLANKS */
               END;
               CALL BSADATR(L1,IDA,IDANEXT,Y1); /* ALLOCATE IN IDA */
            END;
      END;

/*** <DECL.ELEMENT> ::= <IDENT.SPECIFICATION> <TYPE> */
      /* BACKSTUFF TYPE X2 INTO IDENTIFIERS IN CHAIN H1 */
         CALL BSTYPE(L1,X2);

/*** <DECL.ELEMENT> ::= <IDENTIFIER> <DATA.LIST> */
      DO;
         /* <DATA.LIST> RETURNS A POINTER TO 'SKIP' ENTRY IN RELTAB */
         IF NEWSYMB(H1) THEN
            /* FIRST APPEARANCE OF IDENTIFIER IN CURRENT BLOCK */
            DO;  /* SET ENTRY FOR IDENTIFIER IN SYMTAB */
               TEMP2= RLOC(L2) + 2;  /* ADDRESS OF 1ST BYTE OF DATA */
               TEMP1 = CANEXT - TEMP2;  /* TOTAL LENGTH INITIALIZED */
               TEMP = CREATESYM(KBYTE,CA,TEMP2,TEMP1,1);
               CALL BRADDR(L2,CA,CANEXT);  /* BACKSTUFF SKIP */
            END;
         ELSE  /* ERROR: IDENTIFIER ALREADY DECLARED IN THIS BLOCK */
            CALL KERROR(1);
      END;

/*** <DATA.LIST> ::= <DATA.HEAD> <CONSTANT> ) */
      /* RETURN POINTER TO 'SKIP' ENTRY IN RELTAB */
         CALL CHC(CA);  /* EMIT THE CONSTANT INTO CA */

/*** <DATA.HEAD> ::= DATA ( */
      /* RETURN POINTER TO 'SKIP' ENTRY IN RELTAB */
         LL = EMITL(JMP,0);  /* EMIT JMP SKIP */

/*** <DATA.HEAD> ::= <DATA.HEAD> <CONSTANT> , */
         CALL CHC(CA);  /* EMIT THE CONSTANT INTO CA */

/*** <STORAGE.DECLARATION> ::= <IDENT.SPECIFICATION> BYTE */
      /* RETURN POINTER TO IDENTIFIER CHAIN IN SYMTAB */
         CALL SDB(1);

/*** <STORAGE.DECLARATION> ::= <BOUND.HEAD> <NUMBER> ) BYTE */
         CALL SDB(H2);

/*** <STORAGE.DECLARATION> ::= <STORAGE.DECLARATION> <INITIAL.LIST> */
         CALL XMIT1;  /* <INITIAL.LIST> ALREADY STORED IN IDA */

/*** <TYPE> ::= LABEL */
         XX = KLABEL;

/*** <TYPE> ::= EXTERNAL */
         XX = EXT;

/*** <TYPE> ::= COMMON */
         XX = GLOB;

/*** <IDENT.SPECIFICATION> ::= <IDENTIFIER> */
      /* RETURN POINTER TO IDENTIFIER CHAIN IN SYMTAB */
      DO;
         IF NEWSYMB(H1) THEN
            /* FIRST APPEARANCE OF IDENTIFIER IN CURRENT BLOCK */
            DO;
               LL = CREATESYM(UNDECL,UNDEF,0,0,1);
               XX = 1;   /* NO. ELEMENTS IN <IDENT.SPECIFICATION> */
               IIDA = IDANEXT;  /* SAVE CURRENT VALUE OF IDANEXT */
            END;
         ELSE /* IDENTIFIER ALREADY DEFINED IN CURRENT BLOCK */
            DO;
               CALL KERROR(1);
```

```
            END;
        END;

/*** <IDENT.SPECIFICATION> ::= <IDENT.LIST> <IDENTIFIER> ) */
            CALL IDL;

/*** <IDENT.LIST> ::= ( */
        /* RETURN POINTER TO IDENTIFIER CHAIN IN SYMTAB */
        DO;
            LL = 0;             /* POINTER */
            CALL ZEROXX;        /* XX: NO.IDENTIFIERS IN <IDENT.LIST> */
            IIDA = IDANEXT;     /* SAVE CURRENT VALUE OF IDANEXT */
        END;

/*** <IDENT.LIST> ::= <IDENT.LIST> <IDENTIFIER> , */
            CALL IDL;

/*** <BOUND.HEAD> ::= <IDENT.SPECIFICATION> ( */
        /* RETURN POINTER TO IDENTIFIER CHAIN IN SYMTAB */
            CALL XMIT1;

/*** <INITIAL.LIST> ::= <INITIAL.HEAD> <CONSTANT> ) */
            CALL CHC(IDA);  /* EMIT THE CONSTANT INTO IDA */

/*** <INITIAL.HEAD> ::= <INITIAL.HEAD> <CONSTANT> , */
            CALL CHC(IDA);  /* EMIT THE CONSTANT INTO IDA */

/*** <GROUP> ::= <GROUP.HEAD> ; <ENDING> */
        DO;
            IF X1   THEN /* THIS IS A CASE GROUP */
               DO;  /* H1 POINTS TO   'EXIT' CHAIN IN RELTAB */
                  TEMP1 = L1;   /* POINTER TO THE CHAIN */
                  TEMP2 = 0;    /* POINTER TO AN EMPTY CHAIN */
                  /* Y1: NO. STMTS IN THE CASE GROUP */
                  DO WHILE NOT SHR(REXT(TEMP1),15);  /* POSITIVE */
                     /* INVERT THE CHAIN */
                     TEMP = REXT(TEMP1);  /* NEXT */
                     REXT(TEMP1) = TEMP2;
                     TEMP2 = TEMP1;
                     TEMP1 = TEMP;
                  END;
                  /* TEMP1 NOW POINTS TO THE JPVEC ENTRY IN RELTAB */
                  /* TEMP2 POINTS TO THE 'EXIT' CHAIN IN RELTAB */
                  TEMP4= -REXT(TEMP1);       /* ADDRESS OF L0 */
                  CALL SRADDR(TEMP1,CA,CANEXT);  /* BACKSTUFF JPVEC */
                  REXT(TEMP1) = 0;           /* NOT EXTERNAL */
                  TEMP3 = CANEXT + SHL(Y1,1);  /* ADDRESS OF EXIT */
                  DO WHILE TEMP2<>0;
                     /* GENERATE JUMP VECTOR, BACKSTUFF EXIT */
                     CALL EMITA(CA,TEMP4,0); /* EMIT N-TH LABEL (LN) */
                     TEMP1 = REXT(TEMP2);
                     REXT(TEMP2) = 0;        /* NOT EXTERNAL */
                     CALL SRADDR(TEMP2,CA,TEMP3);  /* BACKSTUFF EXIT */
                     TEMP4= RLOC(TEMP2) + 2;  /* COMPUTE NEXT LN */
                     TEMP2 = TEMP1;          /* NEXT ENTRY */
                  END;
               END;  /* OF CASE GROUP */
            ELSE IF X1=2 THEN /* THIS IS AN ITERATIVE GROUP */
               DO; /* H1 POINTS TO F CHAIN IN RELTAB */
                  CALL EMIT3A(JMP,CA,Y1,0);  /* EMIT JMP LOOP */
                  CALL BRADDR(L1,CA,CANEXT);  /* BACKSTUFF F */
               END;
            CALL POPBL(1);  /* EXIT BLOCK, SHRINK WA */
        END;

/*** <GROUP.HEAD> ::= <DO> */
        /* RETURN XX INDICATING TYPE OF GROUP */
            CALL ZEROXX;

/*** <GROUP.HEAD> ::= <DO> <ITERATIVE.CLAUSE> */
        DO;
            CALL XMIT2;
            XX = 2;
```

```
        END;

/*** <GROUP.HEAD> ::= <DO> <CASE.SELECTOR> */
        DO;
           CALL XMIT2;
           XX = 3;
        END;

/*** <GROUP.HEAD> ::= <GROUP.HEAD> ; <STMT> */
        DO;
           CALL XMIT1;
           IF X1      THEN /* CASE GROUP */
              DO;
                 HH = EMITL(JMP,L1);  /* EMIT JMP EXIT */
                 YY = Y1 +1;  /* NO.STATEMENTS IN THE CASE GROUP */
              END;
        END;

/*** <DO> ::= DO */
           CALL PUSHBL;  /* ENTER A NEW BLOCK */

/*** <ITERATIVE.CLAUSE> ::= <INITIALIZATION> <BY> <ASSIGN.STMT>
                            <WHILE> <COMPOUND.CONDITION> */
        /* RETURN HH=POINTER TO F CHAIN IN RELTAB, YY=LOOP ADDRESS */
        DO;
           LL = L3;   /* POINTER */
           CALL BRADDR(L2,CA,H4);  /* BACKSTUFF SKIP */
           YY = RLOC(L2) + 2;  /* ADDRESS OF LOOP */
        END;

/*** <ITERATIVE.CLAUSE> ::= <INITIALIZATION> <WHILE><COMPOUND.CONDITION>*/
        DO;
           YY = H2;   /* ADDRESS OF LOOP */
           LL = L3;   /* POINTER TO F CHAIN IN RELTAB */
        END;

/*** <BY> ::= BY */
        /* RETURN POINTER TO SKIP ENTRY IN RELTAB */
           LL = EMITL(JMP,0);  /* EMIT JMP SKIP */

/*** <WHILE> ::= WHILE */
        /* RETURN ADDRESS OF NEXT INSTRUCTION (COMPOUND.CONDITION) */
           HH = CANEXT;

/*** <CASE.SELECTOR> ::= CASE <REG> */
        /* RETURN HH = POINTER TO A CHAIN IN RELTAB,
                  YY = NO. STATEMENTS IN THE CASE GROUP */
        DO;
           IF HLREG(2) THEN    /* <REG> IS HL */
              DO;
                 CALL EMIT2(PUSHD,XCHG);
                 LL = EMITL(LXIH,0);  /* EMIT LXIH JPVEC */
                 REXT(LL) = -(CANEXT+8);  /* SAVE - ADDRESS OF L0 */
                 CALL EMIT4(DADD,DADD,MOVEM,INXD);
                 CALL EMIT4(MOVDM,XCHG,POPD,PCHL);
                 YY = 0;  /* NO. STATEMENTS */
              END;  /* OF <REG> IS HL */
           ELSE /* NOT IMPLEMENTED */ CALL KERROR(8);
        END;

/*** <PROC.DEFINITION> ::= <PROC.HEAD> <STMT.LIST> ; <ENDING> */
        DO;
           CALL EMIT1(RET);      /* SUPPLY A RETURN */
           CALL BRADDR(Y1,CA,CANEXT);  /* BACKSTUFF SKIP */
           CALL POPBL(0);  /* EXIT BLOCK, DO NOT SHRINK WA */
        END;

/*** <PROC.HEAD> ::= <PROC.NAME> ; */
      /* RETURN POINTER TO SKIP ENTRY IN RELTAB */
        DO;
              CALL PRH(0FFFFH); /* NO FORMAL PARMS: SET ATR TO -1 */
        END;
```

```
/*** <PROC.HEAD> ::= <PROC.NAME> <FORMAL.PARAM.LIST> ; */
        CALL PRH(WANEXT - SHL(X2,1));  /* ADDRESS OF FORMAL PARMS */

/*** <PROC.NAME> ::= <LABEL.DEFINITION> PROCEDURE */
    /* RETURN: HH=POINTER TO PROCEDURE ENTRY IN SYMTAB,
               YY = POINTER TO 'SKIP' ENTRY IN RELTAB */
    DO;
        CALL XMIT1;
        IF X1 THEN /* IDENTIFIER LABEL */
            DO;   /* EMIT CODE TO SKIP THE PROCEDURE */
                YY = EMITL(JMP,0);  /* EMIT JMP SKIP */
                   /* RETURN POINTER TO SKIP ENTRY IN RELTAB */
                SDISPL(L1) = CANEXT;          /* PROCEDURE ENTRY
                   IS AFTER THE JUMP INSTRUCTION */
                CALL PUSHBL;  /* ENTER A NEW BLOCK */
            END;
        ELSE /* NOT A VALID PROCEDURE */ CALL KERROR(3);
    END;

/*** <FORMAL.PARAM.LIST> ::= <FP.HEAD> <IDENTIFIER> ) */
    /* RETURN NUMBER OF FORMAL PARAMETERS */
        CALL FPH;

/*** <FP.HEAD> ::= ( */
    /* RETURN NO. OF FORMAL PARAMETERS */
        CALL ZEROXX;

/*** <FP.HEAD> ::= <FP.HEAD> <IDENTIFIER> , */
        CALL FPH;

/*** <RETURN.STMT> ::= RETURN */
        CALL EMIT1(RET);

/*** <RETURN.STMT> ::= IF <SIMPLE.CONDITION> RETURN */
        CALL EMIT1(X2);   /* EMIT RNZ,RZ,RNC... */

/*** <CALL.STMT> ::= <CALL> <IDENTIFIER> */
    DO;
        IF CANTCALL(H2,0) THEN CALL KERROR(6);
        ELSE CALL EMIT3IA(X1 OR 04H,H2,0);
    END;

/*** <CALL.STMT> ::= <CALL> <ACTUAL.PARAM.LIST> */
    DO;
        CALL EMIT1(POPH);
        CALL EMIT3IA(X1 OR 04H,SNAME(L2),0);
    END;

/*** <CALL.STMT> ::= <CALL> <NUMBER> */
    DO;
        IF (X1=0CDH) /* UNCONDITIONAL CALL */
           AND ( (H2 AND 0FFC7H) = 0) /* 0, 8, 16, 24,..., 56 */
           THEN /* EMIT RST INSTRUCTION */
               CALL EMIT1( LOW(H2) OR 0C7H);
        ELSE /* EMIT A CALL */
               CALL EMIT3D(X1 OR 04H,H2);  /* CALL,CNZ,CZ,... */
    END;

/*** <ACTUAL.PARAM.LIST> ::= <AP.HEAD> <CONSTANT> ) */
        CALL APH;

/*** <AP.HEAD> ::= <IDENTIFIER> ( */
    DO;
        IF CANTCALL(H1,1) THEN CALL KERROR(6);
        ELSE
            DO;
                LL = LLINK(H1);  /* POINTER TO IDENTIFIER IN SYMTAB */
                CALL ZEROXX;  /* NO. ACTUAL PARAMETERS */
                CALL EMIT1(PUSHH);  /* SAVE HL */
            END;
    END;

/*** <AP.HEAD> ::= <AP.HEAD> <CONSTANT> , */
```

```
            CALL APH;

/*** <CALL> ::= CALL */
            XX = 0CDH;

/*** <CALL> ::= IF <SIMPLE.CONDITION> CALL */
          CALL XMITX2;

/*** <GOTO.STMT> ::= <GOTO> <IDENTIFIER> */
         DO;
            IF CANTCALL(H2,0) THEN /* CANNOT GOTO EITHER */
                  CALL KERROR(11);
            ELSE CALL EMIT3IA(X1 OR 02H,H2,0);  /* EMIT JMP, JNZ, JZ...*/
         END;

/*** <GOTO.STMT> ::= <GOTO> <NUMBER> */
            CALL EMIT3D(X1 OR 02H,H2);    /* JMP,JNZ,JZ,... */

/*** <GOTO.STMT> ::= GOTO M ( HL ) */
            CALL EMIT1(PCHL);

/*** <GOTO> ::= GOTO */
            XX = 0C3H;

/*** <GOTO> ::= IF <SIMPLE.CONDITION> GOTO */
            CALL XMITX2;

/*** <REPEAT.STMT) ::= <REPEAT> <STMT.LIST> ; UNTIL <COMPOUND.CONDITION>*/
            CALL BRADDR(L5,CA,H1);  /* BACKSTUFF LOOP ADDRESS */

/*** <REPEAT> ::= REPEAT ; */
            HH = CANEXT;  /* SAVE ADDRESS OF NEXT INSTRUCTION */

/*** <CONTROL.STMT> ::= HALT */
            CALL EMIT1(76H);    /* HLT */

/*** <CONTROL.STMT> ::= NOP */
            CALL EMIT1(00H);    /* NOP */

/*** <CONTROL.STMT> ::= DISABLE */
            CALL EMIT1(0F3H);    /* DI */

/*** <CONTROL.STMT> ::= ENABLE */
            CALL EMIT1(0FBH);    /* EI */

/*** <COMPARE.STMT> ::= <REG> :: <SEC> */
         DO;
            IF AREG(1)    /* <REC> IS REG A */          THEN
               DO;
                  IF X3=03H  /* <SEC> IS A REGISTER */
                     AND AM(3)    /* <SEC> IS A, B, ... M */
                     THEN CALL EMIT1(L3 OR 0B8H);    /* EMIT A CMP */
                  ELSE IF X3=13H  /* <SEC> IS A NUMBER */
                     THEN CALL EMIT2(0FEH,L3);    /* EMIT A CPI */
                  ELSE  /* ERROR */  CALL CANTDO;
               END;
            ELSE  /* ERROR  */  CALL CANTDO;
         END;

/*** <EXCHANGE.STMT> ::= <REG> == <REG.EXPR> */
         DO;
            IF HLREG(1)  /* <REG> IS HL */  THEN
               DO;
                  IF DEREG(3) THEN CALL EMIT1(XCHG);  ELSE
                  IF STACKR(3) THEN CALL  EMIT1 (0E3H);  /* XTHL */
                  ELSE  /* ERROR */  CALL CANTDO;
               END;
            ELSE CALL CANTDO;
         END;

/*** <VAR.ASSIGN> ::= <VAR> = <REG.EXPR> */
         DO;
            IF AREG(3) THEN /* <REG.EXPR> REFERS TO REG A */
```

```
            DO;
               IF X1 = 21H THEN /* <VAR> IS M(BC) OR M(DE) */
                  /* EMIT STAX B, STAX D */
                  CALL EMIT1( LOW(H1) AND 0F2H); ELSE
               IF X1=27H THEN /* <VAR> IS OUT(<NUMBER>) */
                  /* EMIT 'OUT' INSTRUCTION */
                  CALL EMIT2D(0D3H,LOW(H1));  ELSE
               /* TRY TO EMIT A STA INSTRUCTION */
               IF LDAHL(32H,1) THEN /* EMITTED STA */  ;
               ELSE /* ERROR */ CALL CANTDO;
            END;  /* OF <REG.EXPR> REFERS TO REG A */  ELSE
         IF HLREG(3) THEN /* <REG.EXPR> REFERS TO HL */
            DO;
               /* TRY TO EMIT A SHLD INSTRUCTION */
               IF LDAHL(22H,1) THEN /* EMITTED SHLD */ ;
               ELSE /* ERROR */ CALL CANTDO;
            END;  /* OF <REG.EXPR> REFERS TO HL */
         ELSE /* NEITHER A NOR HL: ERROR */ CALL CANTDO;
      END;

/*** <REG.EXPR> ::= <REG> */
         CALL XMIT1;

/*** <REG.EXPR> ::= ( <REG.ASSIGN> ) */
         CALL XMIT2;

/*** <REG.ASSIGN> ::= <REG> = <PRIM> <BINARY.OP> <SEC> */
      DO;
         CALL LOADPRIM(3);      /* LOAD <PRIM> INTO <REG> */
         CALL  BINOPSEC(4,5);  /* EXECUTE <BINARY.OP> ON <SEC> */
         CALL XMIT1;
      END;

/*** <REG.ASSIGN> ::= <REG> = <UNARY.OP> <PRIM> */
      DO;
         CALL LOADPRIM(4);      /* LOAD <PRIM> INTO <REG> */
         /* NOW EXECUTE <UNARY.OP> */
         IF AREG(1) THEN /* <REG> IS A */
            CALL EMIT1(X3);  /* EMIT RLC,RRC,RAL,RAR,DAA,CMA */ ELSE
         IF CYREG(1) /* <REG> IS CARRY */
            AND X3 = 2FH  /* NOT */ THEN
            CALL EMIT1(3FH);  /* EMIT CMC */
         ELSE /* ERROR */  CALL CANTDO;
         CALL XMIT1;
      END;

/*** <REG.ASSIGN> ::= <REG> = <PRIM> */
      DO;
         CALL LOADPRIM(3);      /* LOAD <PRIM> INTO <REG> */
         CALL XMIT1;
      END;

/*** <REG.ASSIGN> ::= <REG.ASSIGN> , <BINARY.OP> <SEC> */
      DO;
         CALL BINOPSEC(3,4);   /* EXECUTE <BINARY.OP> ON <SEC> */
         CALL XMIT1;
      END;

/*** <PRIM> ::= <VAR> */
         CALL XMIT1;

/*** <PRIM> ::= <SEC> */
         CALL XMIT1;

/*** <SEC> ::= <REG.EXPR> */
         CALL XMIT1;

/*** <SEC> ::= <CONSTANT> */
         CALL XMIT1;

/*** <CONSTANT> ::= <STRING> */
      DO;
         XX = 11H;
```

```
            CALL XHITH1;    /* POINTER TO STRING IN SYMLIST */
        END;

/*** <CONSTANT> ::= <NUMBER> */
        DO;
            XX = 13H;
            CALL XHITH1;    /* VALUE OF THE NUMBER */
        END;

/*** <CONSTANT> ::= - <NUMBER> */
        DO;
            XX = 13H;
            HH = -H2;
        END;

/*** <CONSTANT> ::= . <IDENTIFIER> */
        DO;
            XX = 15H;
            CALL IDN(2,0);
        END;

/*** <CONSTANT> ::= . <IDENTIFIER> ( <NUMBER> ) */
        DO;
            XX = 15H;
            CALL IDN(2,H4);
        END;

/*** <CONSTANT> ::= . <STRING> */
       /** .<STRING> */
        DO;
            XX = 17H;
            HH = H2;  /* POINTER TO THE STRING IN SYMLIST */
        END;

/*** <REG> ::= <A.L> */
            CALL SETREG(LOW(H1));  /* A=7, B=0, C=1, D=2, E=3, H=4, L=5 */

/*** <REG> ::= BC */
            CALL SETREG(0FH);

/*** <REG> ::= DE */
            CALL SETREG(1FH);

/*** <REG> ::= HL */
            CALL SETREG(2FH);

/*** <REG> ::= SP */
            CALL SETREG(3FH);

/*** <REG> ::= STACK */
            CALL SETREG(2EH);

/*** <REG> ::= PSW */
            CALL SETREG(3EH);

/*** <REG> ::= M ( HL ) */
            CALL SETREG(6);

/*** <REG> ::= CY */
            CALL SETREG(1EH);

/*** <BINARY.OP> ::= + */
            XX = 080H;

/*** <BINARY.OP> ::= - */
            XX = 090H;

/*** <BINARY.OP> ::= AND */
            XX = 0A0H;

/*** <BINARY.OP> ::= OR */
            XX = 0B0H;
```

```
/*** <BINARY.OP> ::= XOR */
            XX = 0A8H;

/*** <BINARY.OP> ::= ++ */
            XX = 083H;

/*** <BINARY.OP> ::= -- */
            XX = 098H;

/*** <UNARY.OP> ::= < */
            XX = 17H;          /* RAL */

/*** <UNARY.OP> ::= > */
            XX = 1FH;          /* RAR */

/*** <UNARY.OP> ::= NOT */
            XX = 2FH;          /* CMA */

/*** <UNARY.OP> ::= DEC */
            XX = 27H;          /* DAA */

/*** <UNARY.OP> ::= >> */
            XX = 0FH;          /* RRC */

/*** <UNARY.OP> ::= << */
            XX = 07H;          /* RLC */

/*** <VAR> ::= M ( BC ) */
         DO;
            XX = 21H;    LL = 0FH;
         END;

/*** <VAR> ::= M ( DE ) */
         DO;
            XX = 21H;    LL = 1FH;
         END;

/*** <VAR> ::= <IDENTIFIER> */
         DO;
            XX = 23H;
            CALL IDN( 1,0);
         END;

/*** <VAR> ::= <IDENTIFIER> ( <NUMBER> ) */
         DO;
            XX = 23H;
            CALL IDN( 1,H3);
         END;

/*** <VAR> ::= IN ( <NUMBER> ) */
         DO;
            XX = 25H;    HH = H3;
         END;

/*** <VAR> ::= OUT ( <NUMBER> ) */
         DO;
            XX = 27H;    HH = H3;
         END;

/*** <VAR> ::= M ( <CONSTANT> ) */
         DO;
                  XX = (X3 AND 00FH) OR 28H;
                     /* STRING      : CHANGES FROM 11H TO 29H;
                        NUMBER      : CHANGES FROM 13H TO 2BH;
                        .IDENTIFIER: CHANGES FROM 15H TO 2DH;
                        .STRING     : CHANGES FROM 17H TO 2FH;        */
                  HH = H3;
                  YY = Y3;
         END;

      END; /* OF CASE PN */
      END REDUCE;
```

```
MAIN:
      DECLARE ACTION LIT 'TEMP';

/* INITIATE */
      CALL CMON(INITDSK,0); /* SELECT DISK */
      TRACEON
      CALL PRL('.TRACE(C,D,P,N):3);  TRACE=VMON(1,0);
      TRACEOFF *** */
      /* OPEN SYMBOL LIST FILE */
      CALL OPENF('S');
      /* LOAD SYMBOL LIST INTO MEMORY */
      /* MEMPTR ALREADY INITIALIZED = 0 */
      DO WHILE IBP <>0;  /* NOT EOFILE */
         SYMLIST(MEMPTR) = GET1;  /* READ 1 BYTE INTO MEMORY */
         MEMPTR = MEMPTR + 1;
         IF MEMPTR >= FBASE THEN /* SYMBOL LIST OVFLW */
            CALL KERROR(0F6H);
      END;
      /* OPEN PARSER ACTIONS FILE */
      CALL OPENF('P');
      /* SET NAMES OF OUTPUT FILES */
      DO I = 1 TO 8;
         CAFCB(I), IDAFCB(I), RTFCB(I) = IFCB(I);
      END;
      /* CREATE NEW VERSIONS OF THE OUTPUT FILES */
      CALL MAKEF(.CAFCB);
      CALL MAKEF(.IDAFCB);
      CALL MAKEF(.RTFCB);
      /* INITIALIZE THE RELOCATION TABLE */
      DO I = 0 TO RELTSIZE;
         CALL SRBASE(I,UNUSED);
      END;
      /* ENTER INITIAL BLOCK AND START COMPILING */
      CALL PUSHBL;

/* MAIN LOOP */
      DO WHILE COMPILING;
         ACTION = GET1;      /* READ ACTION */
         IF ACTION = XREDUCE THEN
            DO;
               DECLARE I BYTE;  /* INDEX OF HANDLE(0) */
               I=PTOP-GET1;  /* GET1: NO. ELEMENTS IN THE HANDLE */
               /* SET BASES FOR HANDLE VARIABLES */
               BASEH = .PSH(I);
               BASEX = .PSX(I);
               BASEY = .PSY(I);
               /* SET NON-INDEXED VARIABLES */
               H1=H(1); H2=H(2); H3=H(3); H4=H(4); H5=H(5);
               X1=X(1); X2=X(2); X3=X(3); X4=X(4);
               Y1=Y(1); Y2=Y(2); Y3=Y(3);
               /* NOW EXECUTE THE REDUCTION */
               HH, LL = 0;
               CALL REDUCE(DOUBLE(GET1) AND 00FFH);
               /* THEN REPLACE HANDLE BY LEFT HAND SIDE OF PRODUCTION */
               PTOP = I+1;
               PSH(PTOP) = HH OR LL;
               PSX(PTOP) = XX;
               PSY(PTOP) = YY;
               TRACEON
               IF TRACE='P' THEN CALL TRAC('Q');
               TRACEOFF *** */
            END;   ELSE
         IF ACTION = XSHIFT THEN
            DO;
               IF (PTOP:=PTOP+1)>=STACKSIZE THEN /* PARSE STACK OVFLW */
                  CALL KERROR(0F1H);
               PSH(PTOP)=GET2;      /* SHIFT VALUE INTO PARSE STACK */
               TRACEON
               IF TRACE='P' THEN CALL TRAC('X');
               TRACEOFF *** */
            END;   ELSE
         IF ACTION = XLINE THEN  /* UPDATE LINE COUNTER */
            LINE = GET2;   ELSE
```

```
            IF ACTION = XACCEPT THEN COMPILING = 0;
            ELSE /* FILE ERROR */  CALL KERROR(OFSH);
         END;  /* OF WHILE COMPILING */

      /* TERMINATE */
         /* FLUSH RELOCATION TABLE */
         DO I = 0 TO LAST(RSB);
            IF RBASE(I) = UNDEF THEN /* UNRESOLVED REFERENCE */
               CALL KERROR(14);
            IF RBASE(I)<> UNUSED THEN CALL DUMPRT(I);
         END;
         /* DUMP REMAINDER OF SYMBOL TABLE */
         CALL POPBL(0);
         /* WRITE RECORD CONTAINING SEGMENT SIZES */
         CALL WRR1('.');
         CALL WRR2(CANEXT);
         CALL WRR2(IDANEXT);
         CALL WRR2(WASIZE);
         /* FLUSH BUFFERS */
         CALL FLUSH(.CAFCB);
         CALL FLUSH(.IDAFCB);
         CALL FLUSH(.RTFCB);
         /* CLOSE FILES AND QUIT */
         CALL CLOSEALL;

EOF
```

```
/*
            ******************************************
            *                                        *
            *  L83: LINKAGE EDITOR FOR L80 PROGRAMS  *
            *                                        *
            *       LUIZ PEDROSO - OCTOBER 1975      *
            *                                        *
            ******************************************
*/

    DECLARE FOREVER          LITERALLY 'WHILE 1';
    DECLARE LIT              LITERALLY 'LITERALLY';
    DECLARE LF               LITERALLY '0AH';    /* LINE-FEED */
    DECLARE CONTROLZ         LITERALLY '1AH';
    DECLARE CR               LITERALLY '0DH';    /* CARRIAGE RETURN */

    /* CP/M SYSTEM CONSTANTS */
    DECLARE CPM              LITERALLY '0';      /* CP/M REBOOT ENTRY */
    DECLARE IFCBA            LITERALLY '005CH'; /* INPUT FCB ADDRESS */
    DECLARE SBUFA            LITERALLY '0080H'; /* I/O BUFFER ADDRESS */
    DECLARE FBASE            LITERALLY '3200H'; /* FDOS BASE */
    DECLARE BDOS             LITERALLY '3FFDH';/* BASIC DOS ENTRY */

    /* I/O PRIMITIVES */
    DECLARE READCHAR         LITERALLY  '1';
    DECLARE PRINTCHAR        LITERALLY  '2';
    DECLARE PRINT            LITERALLY  '9';
    DECLARE OPEN             LITERALLY  '15';
    DECLARE CLOSE            LITERALLY  '16';
    DECLARE MAKE             LITERALLY  '22';
    DECLARE DELETE           LITERALLY  '19';
    DECLARE READBF           LITERALLY  '20';
    DECLARE WRITEBF          LITERALLY  '21';
    DECLARE INITDSK          LITERALLY  '13';
    DECLARE SETBUF           LITERALLY  '26';

    /* INPUT FILE CONTROL BLOCK */
    DECLARE IFA ADDRESS INITIAL (IFCBA);        /* FCB ADDRESS */
    DECLARE IFCB BASED IFA BYTE;

    /* INPUT BUFFER */
    DECLARE IBUFA ADDRESS INITIAL(SBUFA);
    DECLARE IBUF BASED IBUFA BYTE;  /* INPUT BUFFER */
    DECLARE IBP BYTE;      /* POINTER TO NEXT CHARACTER IN INPUT BUFFER */

    /* OUTPUT FILE CONTROL BLOCK */
    DECLARE OFCB(33) BYTE
            INITIAL(0,'        ','COM',0,0,0,0);

    /* OUTPUT BUFFER */
    DECLARE OBUF(1024) BYTE;
    DECLARE OBP ADDRESS  INITIAL(0);      /* OUTPUT BUFFER POINTER */

    /* MNEMONICS FOR SEGMENTS */
    DECLARE  CA LIT '1',      /* CODE AREA */
             IDA LIT '2',     /* INITIAL DATA AREA */
             WA LIT '3';      /* WORK AREA */

    /* MODULE MAP */
    DECLARE  MSIZE           LIT '20';  /* MAX NO. MODULES L83 CAN HANDLE */
    DECLARE  MNAME(MSIZE) ADDRESS,    /* POINTER TO MODULE NAME */
             MCA(MSIZE) ADDRESS,      /* ADDRESS OF CA  */
             NCA(MSIZE) ADDRESS,      /* LENGTH OF CA */
             MIDA(MSIZE) ADDRESS,     /* ADDRESS OF IDA */
             NIDA(MSIZE) ADDRESS,     /* LENGTH OF IDA */
             MWA(MSIZE) ADDRESS,      /* ADDRESS OF WA  */
             MSYMB(MSIZE) ADDRESS,    /* BASE ADDR OF MODULE'S SYMB LIST */
             MHASH(MSIZE) BYTE;       /* HASHCODE OF MODULE'S NAME */
    DECLARE  MTOP BYTE INITIAL (0);   /* INDEX OF LAST ENTRY IN THE MMI */

    /* LOGICAL RECORD FROM THE RELOCATION TABLE FILE */
```

```
DECLARE REC     BYTE,       /* RECORD TYPE */
        RLOC    ADDRESS,    /* LOCATION OF ADDRESS TO BE RELOCATED */
        RDISPL  ADDRESS,    /* DISPLACEMENT */
        REXT    ADDRESS,    /* POINTER TO EXTERNAL NAMES */
        RSEG    BYTE,       /* SEGMENT OF ADDRESS TO BE RELOCATED */
        RBASE   BYTE,       /* BASE */
        /* (ADDRESS (BASE, DISPL) IS TO BE INSERTED AT (SEG,LOC)) */
        /* REC = 'R' INDICATES  'RELOCATION' RECORD  */
        /* REC = 'S' INDICATES  'SYMBOL REF' RECORD  */
        /* REC = '.' INDICATES LAST RECORD (COUNTERS) */
        /* FIELD REDEFINITIONS FOR 'S' RECORDS */
        SDISPL  LIT 'RLOC',
        SLINE   LIT 'RDISPL',
        SNAME   LIT 'REXT',
        STYPE   LIT 'RSEG',
        SBASE   LIT 'RBASE',
        /* FIELD REDEFINITIONS FOR '.' RECORD */
        CASIZE  LIT 'RLOC',
        IDASIZE LIT 'RDISPL',
        WASIZE  LIT 'REXT';

/* SYMBOL LIST (FOR EACH MODULE) */
DECLARE SYMLB ADDRESS,              /* BASE OF THE SYMBOL LIST */
        SYMLIST BASED SYMLB BYTE;

/* SEGMENT BEING UPDATED IN MEMORY */
DECLARE SEGB  ADDRESS,             /* BASE OF THE SEGMENT */
        SEG BASED SEGB BYTE;

DECLARE LOADADR  ADDRESS INITIAL (0100H); /* LOAD ADDRESS FOR OBJ MODULE*/
DECLARE  LA        ADDRESS;  /* LOAD ADDRESS FOR EACH SEGMENT */
DECLARE MEMPTR ADDRESS INITIAL(0); /* POINTER TO NEXT BYTE IN MEMORY */
DECLARE LSTOREQ BYTE INITIAL(0);    /* 1 TO PRINT SYMBOL LIST */
DECLARE H BYTE;     /* INDEX OF MODULE IN THE MODULE MAP */
DECLARE HP ADDRESS;  /* AUXILIARY MEMORY POINTER */
DECLARE HASHMASK LIT '127';
DECLARE HASHCODE BYTE;
DECLARE TEMP ADDRESS;

/* OUTPUT DEVICES */
DECLARE CRT       LIT  '2';
DECLARE PRINTER   LIT  '5';
DECLARE OUTDEV BYTE INITIAL(CRT);     /* CURRENT OUTPUT DEVICE */
DECLARE PRINTDEV BYTE INITIAL (CRT); /* DEV FOR SYMB LIST OUTPUT */

/* MNEMONICS FOR 'TYPE' */
DECLARE KLABEL   LIT '1',
        KBYTE    LIT '2',
        PROC     LIT '3',
        EXT      LIT '4',
        GLOB     LIT '5',
        STRING   LIT '7',
        UNDECL   LIT '6';

/* SWITCHES FOR BUILT-IN TRACE ROUTINES */
/* TO INCLUDE TRACE ROUTINES, SET TRACESON TO BLANKS */
/* TO EXCLUDE TRACE ROUTINES, SET TRACESON TO SLASH-STAR */
DECLARE TRACESON  LIT ' /* ';
DECLARE TRACESOFF LIT ' /* ';

TRACESON
DECLARE TRACE BYTE INITIAL(0);
DECLARE TRACT ADDRESS;
TRACESOFF *** */

VMON: PROCEDURE(FUNC,INFO) BYTE;
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; RETURN A VALUE */
      GO TO BDOS;
      END VMON;

CMON: PROCEDURE(FUNC,INFO);
      DECLARE FUNC BYTE, INFO ADDRESS;
```

```
      /* ASK CP/M TO EXECUTE FUNC; NO VALUE RETURNED */
      GO TO BDOS;
      END CMON;

DOUBLET: PROCEDURE(L,H) ADDRESS;
      DECLARE (L,H) BYTE;
      /* RETURN ADDRESS FORMED BY BYTES L,H */
      RETURN SHL(DOUBLE(H),8) OR L;
      END DOUBLET;

STORE: PROCEDURE(N,A);
      DECLARE (N,A) ADDRESS;
      /* STORE INTEGER N (2 BYTES) AT ADDRESS A */
      DECLARE V BASED A ADDRESS;
      V = N;
      END STORE;

PRC: PROCEDURE(C);
      DECLARE C BYTE;
      /* PRINT CHARACTER C ON CRT OR PRINTER */
      CALL CMON(OUTDEV,C);
      END PRC;

CRLF: PROCEDURE;
      /* CARRIAGE RETURN, LINE FEED */
      CALL PRC(CR);
      CALL PRC(LF);
      END CRLF;

PRINTH: PROCEDURE(H);
      DECLARE H BYTE;
      /* PRINT A HEXADECIMAL CHARACTER */
      IF H>9 THEN CALL PRC( H - 10 + 'A' );
      ELSE CALL PRC ( H + '0');
      END PRINTH;

PRINTHB: PROCEDURE(B);
      DECLARE B BYTE;
      /* PRINT B IN HEX */
      CALL PRINTH(SHR(B,4));
      CALL PRINTH(B AND 0FH);
      END PRINTHB;

PRINTHA: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT A IN HEX */
      CALL PRINTHB( HIGH(A) );
      CALL PRINTHB( LOW(A) );
      END PRINTHA;

PRS: PROCEDURE(A);
     DECLARE A ADDRESS;
     /* PRINT STRING BEGINNING AT ADDRESS A UNTIL A $ IS FOUND */
     DECLARE B BASED A BYTE;
     DO WHILE B <> '$';
        CALL PRC(B);
        A = A + 1;
     END;
     END PRS;

PRL: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT LINE BEGINNING AT ADDRESS A */
      CALL CRLF;
      CALL PRS(A);
      END PRL;

EXIT: PROCEDURE;
      /* QUIT */
      CALL PRL(.'END L33 $');  CALL CRLF;
      GO TO CPM;
      END EXIT;
```

```
PRFN: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT NAME OF THE FILE WHOSE FCB IS AT A */
      DECLARE FCB BASED A BYTE, I BYTE;
      CALL PRC(' ');
      DO I = 1 TO 11;
         IF FCB(I) <> ' ' THEN CALL PRC(FCB(I));
         IF I = 8  THEN CALL PRC('.');
      END;
      END PRFN;

PRNAME: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* PRINT NAME BEGINNING AT ADDRESS A */
      DECLARE B BASED A BYTE;
      DO WHILE B <> 0;
         CALL PRC(B);
         A = A + 1;
      END;
      END PRNAME;

PRMN: PROCEDURE(J);
      DECLARE J BYTE;
      /* PRINT NAME OF MODULE J */
      CALL PRNAME(MNAME(J));
      END PRMN;

ERROR: PROCEDURE(N);
      DECLARE N BYTE;
      /* EMIT ERROR MESSAGE */
      DO CASE N;
         CALL PRL('.'TOO MANY MODULES$');                              /* 0 */
         DO; CALL PRL('.'CANNOT CREATE$'); CALL PRFN(.OFCB);           /* 1 */
         END;
         DO; CALL PRL('.'CANNOT CLOSE$');  CALL PRFN(.OFCB);           /* 2 */
         END;
         DO; CALL PRL('.'WRITE ERROR$');   CALL PRFN(.OFCB);           /* 3 */
         END;
         DO; CALL PRL('.'READ ERROR$'); CALL PRFN(IFCBA);              /* 4 */
         END;
         DO; CALL PRL('.'CANNOT OPEN$'); CALL PRFN(IFCBA);             /* 5 */
         END;
         CALL PRL('.'MEMORY OVERFLOW$');                               /* 6 */
         CALL PRL('.'BAD RT FILES$');
      END;  /* CASE */
      CALL EXIT;
      END ERROR;

      TRACEON
TRAA: PROCEDURE(A); DECLARE A ADDRESS;
      CALL PRC(':'); CALL PRINTHA(A);
      END TRAA;

TRAB: PROCEDURE(B); DECLARE B BYTE;
      CALL PRC(':');  CALL PRINTHB(B);
      END TRAB;

TRAC: PROCEDURE(B);
      DECLARE (B,I) BYTE; DECLARE J ADDRESS;
      IF NOT TRACE THEN RETURN;
      CALL PRC(B);  B = VMON(1,0);
      DO WHILE B<>' ';
         IF B='M' THEN DO I=0 TO MTOP;
            CALL TRAB(I); CALL TRAA(MNAME(I)); CALL TRAA(MCA(I));
            CALL TRAA(MIDA(I)); CALL TRAA(MWA(I)); CALL TRAA(MSYMB(I));
            CALL TRAB(MHASH(I));
         END;
         IF B='R' THEN DO;
            CALL TRAB(REC); CALL TRAA(RLOC); CALL TRAA(RDISPL);
            CALL TRAA(REXT); CALL TRAB(RSEC); CALL TRAB(RBASE);
         END;
         IF B='S' THEN DO J=1 TO TRACT;
            CALL TRAB(MEMORY(MEMPTR+J-1));
```

```
      END;
      CALL PRC(' '); B=VMON(1,0);
   END;
   END TRAC;
   TRACESOFF *** */

FLUSH: PROCEDURE;
   /* WRITE A RECORD ONTO THE OUTPUT FILE */
   DECLARE I ADDRESS;
   I=0;
   DO WHILE I<OBP;
      CALL CMON(SETBUF,.OBUF(I));
      IF VMON(WRITEBF,.OFCB) <> 0 THEN  /* WRITE ERROR */
         CALL ERROR(3);
      I=I+128;
   END;
   OBP = 0;    /* BUFFER IS EMPTY */
   END FLUSH;

PUT1: PROCEDURE(B);
   DECLARE B BYTE;
   /* WRITE BYTE B ON OUTPUT FILE */
   IF OBP = LENGTH(OBUF)  THEN  /* BUFFER FULL */ CALL FLUSH;
   OBUF(OBP) = B;
   OBP = OBP + 1;
   END PUT1;

GET1: PROCEDURE BYTE;
   /* READ 1 BYTE FROM INPUT FILE */
   DECLARE C BYTE;
   IF IBP = 128 THEN /* INPUT BUFFER EMPTY */
      DO;  /* READ NEXT RECORD */
         CALL CMON(SETBUF,SBUFA);
         IBP = 0;
         IF (C:=VMON(READBF,IFCBA))= 1 /* EOF */ THEN RETURN 0;
         IF C <> 0 THEN  /* ERROR */ CALL ERROR(4);    /* QUIT */
      END;
   C = IBUF(IBP);
   IBP = IBP + 1;
   RETURN C;
   END GET1;

GET2: PROCEDURE ADDRESS;
   /* READ 2 BYTES FROM INPUT FILE */
   RETURN DOUBLET(GET1,GET1);
   END GET2;

YN: PROCEDURE;
   /* ASK OPERATOR IF YES OR NO */
   CALL PRC(3FH);    /* QUESTION MARK */
   CALL CMON(PRINT,.' (Y/N): $');
   END YN;

YES: PROCEDURE BYTE;
   /* TRUE IF ANSWER IS 'Y' */
   DECLARE C BYTE;
   RETURN (C:=VMON(READCHAR,0))='Y' OR C= 79H;
  END YES;

PRLA: PROCEDURE(A);
   DECLARE A ADDRESS;
   /* PRINT LOAD ADDRESS A */
   CALL PRL(.'LOAD ADDRESS = $');
   CALL PRINTHA(A);
   END PRLA;

READCH: PROCEDURE BYTE;
   /* READ 1 HEX CHAR FROM THE CRT */
   DECLARE C BYTE;
RD:  IF((C:=VMON(READCHAR,0))>='0') AND (C>='9') THEN RETURN  C - '0';
   IF(C>='A') AND (C>='F') THEN RETURN C - 'A' + 10;
   CALL PRC(3FH);    /* QUESTION MARK */   GO TO RD;
   END READCH;
```

```
READCB: PROCEDURE BYTE;
      /* READ 2 HEX CHAR FROM THE CRT AND MAKE THEM INTO A BYTE */
      RETURN SHL(READCH,4) OR READCH;
      END READCB;

DUMP: PROCEDURE(N);
      DECLARE N ADDRESS;
      /* DUMP N BYTES OF SEGMENT (CA,IDA) CURRENTLY IN MEMORY INTO THE
         OBJECT MODULE OUTPUT FILE */
      DECLARE I ADDRESS;
      DO I =1 TO N;    /* N MAY BE 0 */
         CALL PUT1(SEG(I-1));
      END;
      END DUMP;

READRR: PROCEDURE;
      /* READ LOGICAL RECORD FROM RT FILE */
      REC = GET1;
      RLOC = GET2;
      RDISPL = GET2;
      REXT = GET2;
      IF REC <> '.' /* NOT TRAILER RECORD */ THEN
         DO;
            DECLARE C BYTE ;
            C = GET1;
            RSEG = SHR(C,4);    /* 4 MOST SIGNIF BITS */
            RBASE = C AND 0FH; /* 4 LEAST SIGNIF BITS */
         END;
      TRACEON
      CALL TRAC('T');
      TRACEOFF *** */
      END READRR;

SETFCB: PROCEDURE(S);
      DECLARE S BYTE;
      /* SET FCB TO ALLOW OPENING OF SEGMENT S OF MODULE M */
      DECLARE ( I, EONAME) BYTE;
      DECLARE A ADDRESS, NCHR BASED A BYTE;
      EONAME = 0;      /* NOT END OF NAME */
      A = MNAME(M);   /* ADDRESS OF MODULE'S NAME */
      DO I = 1 TO 8;      /* SET FILE NAME */
         IF EONAME THEN IFCB(I) = ' ';    ELSE
         DO;
            IFCB(I) = NCHR;      /* MOVE 1 NAME CHARACTER */
            A = A + 1;        /* LOOK AT NEXT CHARACTER */
            IF NCHR = 0 THEN EONAME = 1;    /* END OF NAME */
         END;
      END;
      IF NOT EONAME THEN /* NAME TOO LONG */
         DO;
            CALL PRL('NAME TOO LONG (TRUNCATED): S');
            CALL PRMN(M);
         END;
      /* SET FILE TYPE */
      IFCB(9)  = 'S';
      IFCB(10) = 'O';
      IFCB(11) = S;    /* 'C':CA, 'D':IDA, 'S':SYMB LIST, 'R': RELOC TBL */
      /* SET OTHER BYTES */
      DO I = 12 TO 15;
         IFCB(I) = 0;
      END;
      IFCB(32) = 0;       /* NEXT RECORD */
      IFCB(0)= 0;
      IBP = 128;  /* INPUT BUFFER EMPTY */
      END SETFCB;

OPENF: PROCEDURE;
      /* OPEN INPUT FILE (FCB ASSUMED SET UP) */
      IF VMON(OPEN,IFCBA) <> 255 THEN RETURN;    /* ELSE: ERROR */
      CALL ERROR(5);
      END OPENF;
```

```
OPENRT: PROCEDURE;
       /* OPEN RT FILE OF MODULE M */
       CALL SETFCB('R');     /* SET FILE CONTROL BLOCK */
       CALL OPENF;           /* OPEN THE FILE */
       /* SET BASE OF SYMBOL LIST OF MODULE M */
       SYMLB = MSYMB(M);
       END OPENRT;

READF: PROCEDURE(S,SAV);
       DECLARE (S,SAV) BYTE;
       /* READ SEGMENT S OF MODULE M INTO MEMORY;  IF SAVE = 1 THEN
          (THE FILE BEING READ IS A SYMBOL LIST)
          ADVANCE THE MEMORY PTR (SAVE THE SEGMENT FROM BEING OVERLAYED) */
       DECLARE K BYTE;
       CALL SETFCB(S);    /* SET FILE CONTROL BLOCK */
       CALL OPENF;        /* OPEN THE FILE */
       MP = MEMPTR;       /* INDEX OF NEXT BYTE IN MEMORY */
       CALL CMON(SETBUF,SBUFA);
       DO FOREVER;    /* READ THE FILE */
          IF(K:=VMON(READBF,IFCBA))= 1 /* EOFILE */ THEN RETURN;
          IF MP + 128 > FBASE THEN /* MEMORY OVERFLOW */
             CALL ERROR(6);  /* QUIT */
          IF  K = 0  /* SUCCESSFUL READ */ THEN
             DO  WHILE K <> 128;  /* MOVE BUFFER TO MEMORY */
                 IF SAV /* SYMBOL LIST */ AND (IBUF(K)=CONTROLZ) /* EOF */
                 THEN
                 DO;
                    MEMPTR = MP;  /* ADVANCE MEMPTR */
                    RETURN;
                 END;
                 MEMORY(MP) = IBUF(K);
                 MP = MP + 1 ;
                 K  = K + 1;
             END;
          ELSE /* READ ERROR */  CALL ERROR(4);
       END; /* OF DO FOREVER */
       END READF;

COMPAR: PROCEDURE (A1,A2) BYTE;
       DECLARE(A1,A2) ADDRESS;
       /* TRUE IF STRINGS AT A1,A2 ARE EQUAL */
       DECLARE  B1 BASED A1 BYTE;
       DECLARE B2 BASED A2 BYTE;
       DO WHILE B1 = B2;
          IF B1 = 0  /* END OF STRING */ THEN RETURN 1;
          A1 = A1 + 1;     A2 = A2 + 1;
       END;
       RETURN 0;
       END COMPAR;

HASHF:  PROCEDURE(A) BYTE;
        DECLARE A ADDRESS;
        /* RETURN HASHCODE OF NAME AT ADDRESS A */
        DECLARE C BASED A BYTE, H BYTE;
        H = 0;
        DO WHILE C <> 0;
           H = (H + C) AND HASHMASK;
           A = A + 1;
        END;
        RETURN H;
       END HASHF;

NEW:  PROCEDURE(A) BYTE;
       DECLARE A ADDRESS;
       /* TRUE IF NAME AT A IS NOT IN THE MODULE MAP */
       DECLARE  K BYTE;
       HASHCODE = HASHF(A);  /* COMPUTE HASHCODE OF THE NAME */
       DO K = 0  TO MTOP;  /* SEARCH MODULE MAP */
          IF HASHCODE = MHASH(K) THEN /* MAY BE EQUAL */
             IF COMPAR(A,MNAME(K)) THEN /* EQUAL */ RETURN 0;
       END;
       RETURN 1;
       END NEW;
```

```
CONCAT: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* COMPUTE LOAD ADDRESSES OF SEGMENTS IN THE MODULE MAP,
         SO THAT ALL SEGMENTS ARE CONCATENATED IN MEMORY */
      DECLARE SSZ BASED A ADDRESS;     /* SEGMENT SIZE */
      DECLARE J BYTE;
      DO J = 0 TO MTOP;
         TEMP = SSZ(J) + LA;  /* SAVE    ADDRESS OF NEXT SEGMENT */
         SSZ(J) = LA;         /* SET ADDRESS OF THIS SEGMENT */
         LA = TEMP;           /* SET ADDRESS FOR NEXT SEGMENT */
      END;
      END CONCAT;

ENTRY: PROCEDURE(N) ADDRESS;
      DECLARE N BYTE;
      /* RETURN ADEQUATE ENTRY POINT IN MODULE N FOR THE ADDRESS
         DESCRIBED BY THE CURRENT RELOCATION RECORD */
      IF RBASE = CA THEN RETURN MCA(N);
      IF RBASE = IDA THEN RETURN MIDA(N);
      IF RBASE = WA  THEN RETURN MWA(N);
      /* ELSE: BAD FILE */  CALL ERROR(7);    /* QUIT */
      END ENTRY;

RESOLVE: PROCEDURE;
      /* USE THE INFORMATION IN THE CURRENT RELOCATION RECORD
         TO RESOLVE ONE ADDRESS IN THE SEGMENT (CA,IDA) CURRENTLY
         IN MEMORY, WHICH BELONGS TO MODULE M */
      DECLARE K BYTE;
      IF REXT <> 0 THEN  /* AN EXTERNAL REFERENCE */
         DO;    /* FIND INDEX OF EXTERNAL MODULE */
            K = SYMLIST(REXT-1);  /* POINTER TO EXTERNAL MODULE IN MI */
            TEMP = ENTRY(K);
            K = SEG(RLOC-1);  /* LOOK AT PREVIOUS BYTE */
            IF RBASE=CA AND (K=0CDH /* CALL */
               OR (K AND 0C7H)= 0C4H  /* CNZ,CZ,ETC */ )
               THEN TEMP = TEMP+3;  /* ADJUST ENTRY TO PROCEDURE */
         END;
      ELSE  /* NOT AN EXTERNAL REFERENCE */
         TEMP = ENTRY(M);
      /* SET THE ADDRESS */
      CALL STORE(TEMP + RDISPL,.SEG(RLOC));
      END RESOLVE;

LSTSREF: PROCEDURE;
      /* PRINT INFORMATION ABOUT SYMBOL WHOSE RELOCATION RECORD
         IS CURRENTLY IN MEMORY, AND WHICH BELONGS TO MODULE M */
      IF STYPE=EXT OR STYPE=GLOB THEN RETURN;
      OUTDEV = PRINTDEV;  /* SEND OUTPUT TO THE PRINTING DEVICE */
      CALL PRL(.'L:S');
      CALL PRINTHA(SLINE);
      CALL PRS(.' A:S');
      CALL PRINTHA(ENTRY(M)  + SDISPL);
      CALL PRS(.' T:S');
      IF STYPE=KLABEL THEN CALL PRC('L');  ELSE
      IF STYPE=KBYTE  THEN CALL PRC('B');  ELSE
      IF STYPE=PROC   THEN CALL PRC('P');  ELSE
      IF STYPE=STRING THEN CALL PRC('S');
      CALL PRC(' ');
      CALL PRNAME(.SYMLIST(SNAME));
      OUTDEV = CRT;  /* REROUTE OUTPUT TO CRT */
      END LSTSREF;

MAIN:
/* INITIALIZE */
      DECLARE (C,J) BYTE, A ADDRESS;
      /* ASK FOR PARAMETERS */
      IF IFCB(9) <> ' ' THEN
         DO;
            TRACESON
            CALL PRL(.'TRACES'); CALL YN; TRACE=YES;
            TRACESOFF *** */
            CALL PRL(.'SYMBOL LISTINGS');
```

```
      CALL YN;
      LSTREQ = YES;
      IF LSTREQ THEN
         DO;
            CALL PRL(.'OUTPUT TO PRINTERS');
            CALL YN;  IF YES THEN PRINTDEV=PRINTER;
         END;
      CALL PRLA(LOADADR);
      CALL YN;
      IF NOT YES THEN
         DO;
            CALL PRL(.'LA (XXXX) = S');
            J = READCB;    /* HIGH BYTE */
            LOADADR = DOUBLET(READCB,J);
         END;
      END;
LA = LOADADR;
/* SET UP NAME OF ARGUMENT PROGRAM INTO THE GLOBAL SYMBOL LIST */
MEMORY(1) = 0;           /* POINTER FROM SYMBOL TO MODULE MAP */
MEMPTR = 2;  /* NAME BEGINS AT MEMORY(2) */
MNAME(0) = .MEMORY(2);  /* POINTER FROM MM TO SYMBOL */
J = 1;  /* INDEX OF PROG NAME IN THE INPUT FCB */
DO WHILE J<9 AND (C:=IFCB(J))<> ' ';
   MEMORY(MEMPTR), OFCB(J) = C;  /* SET ALSO THE OUTPUT FCB */
   MEMPTR = MEMPTR + 1;
   J = J + 1;
END;
MEMORY(MEMPTR) = 0;  /* FLAG FOR END OF NAME */
MEMPTR = MEMPTR + 1;
MHASH(0) = HASHF(.MEMORY(2));
/* SET UP MODULE MAP */
M = 0;    /* INDEX OF 1ST  MODULE */
DO WHILE M<= MTOP;  /* GET INFO ABOUT MODULE M */
   MSYMB(M) = .MEMORY(MEMPTR); /* SAVE BASE OF M'S SYMBOL LIST */
   CALL READF('S',1); /* READ SYMB LIST OF MODULE M INTO MEMORY */
   CALL OPENRT;      /* OPEN RT FILE OF MODULE M */
   CALL READRR;      /* READ RECORD FROM RT FILE */
   DO WHILE REC <> '.';  /* NOT END OF RELOCATION TABLE */
      IF REC = 'R'  /* 'RELOCATION' RECORD */
         AND REXT<>0  /* EXTERNAL REFERENCE */  THEN
         DO;  /* CREATE ANOTHER ENTRY IN THE MODULE MAP */
            /* BUT FIRST CHECK IF ALREADY THERE */
            A = .SYMLIST(REXT);  /* ADDRESS OF THE SYMBOL */
            IF NEW(A) THEN /* NOT YET IN THE MODULE MAP */
               DO;  /* TRY TO INSTALL IT IN MM */
                  IF (MTOP:=MTOP+1) >= MSIZE THEN
                     /* MM OVERFLOW */  CALL ERROR(0);  /* QUIT */
                  /* SET POINTER FROM SYMBOL TO MM */
                  SYMLIST(REXT-1) = MTOP;
                  /* SET POINTER FROM MM TO SYMBOL */
                  MNAME(MTOP)=A;     /* ADDRESS */
                  /* SAVE SYMBOL HASHCODE IN MM */
                  MHASH(MTOP) = HASHCODE;
               END;  /* OF TRY TO INSTALL IT IN MM */
         END;  /* OF CREATE ANOTHER ENTRY IN MM */
      CALL READRR;    /* READ NEXT RECORD FROM RT FILE */
   END;  /* OF REC <> '.' */
   /* '.' RECORD: END OF RELOCATION TABLE */
   /* SET SEGMENT SIZES FOR MODULE M */
   MCA(M), NCA(M) = CASIZE;
   MIDA(M), NIDA(M) = IDASIZE;
   MWA(M) = WASIZE;
   /* ADVANCE TO NEXT MODULE */
   M = M + 1;
END;
/* NOW THE MODULE MAP IS COMPLETE; THE BASE ADDRESSES OF EACH
   SEGMENT IN EACH MODULE CAN BE COMPUTED */
/* LA CONTAINS THE LOAD ADDRESS FOR THE FINAL OBJECT MODULE */
CALL CONCAT(.NCA);  /* CONCATENATE ALL CA'S */
CALL CONCAT(.MIDA); /* CONCATENATE ALL IDA'S */
CALL CONCAT(.MWA);  /* CONCATENATE ALL WA'S */
TRACEON
CALL TRAC('P');
```

```
        TRACESOFF *** */

    /* CONSTRUCT THE OBJECT MODULE */
        /* CREATE OUTPUT FILE */
        CALL CHON(DELETE,.OFCB);    /* DELETE OLD VERSION */
        IF VMON(MAKE,.OFCB) = 255 THEN  /* ERROR */
           CALL ERROR(1);
        OFCB(32) = 0;    /* NEXT RECORD */
        SEGB = .MEMORY(MEMPTR);  /* BASE ADDRESS FOR SEGMENTS */
        /* BRING CA'S INTO MEMORY, RESOLVE THEIR ADDRESS AND DUMP THEM */
        DO M=0 TO MTOP;
           CALL READF('C',0);  /* READ CA OF MODULE M INTO MEMORY */
           TRACEON
           TRACT=NCA(M);  CALL TRAC('A');
           TRACEOFF *** */
           CALL OPENRT;          /* OPEN RT FILE OF MODULE M */
           CALL READRR;          /* READ RT RECORD */
           DO WHILE REC<> '.';    /* NOT END OF RELOCATION TABLE */
              IF REC = 'R' AND RSEG = CA THEN  /* RESOLVE THE REFERENCE */
                 CALL RESOLVE;
              CALL READRR;       /* NEXT RECORD */
           END;
           TRACEON
           CALL TRAC('C');
           TRACEOFF *** */
           CALL DUMP(NCA(M));    /* DUMP CA OF MODULE M */
        END;
        /* BRING IDA'S INTO MEMORY, RESOLVE THEIR ADDRESSES AND DUMP THEM;
           AT THE SAME TIME PRINT SYMBOL REFERENCE LIST, IF REQUIRED */
        DO M=0 TO MTOP;
           CALL READI('D',0);  /* READ IDA OF MODULE M INTO MEMORY */
           TRACEON
           TRACT=NIDA(M); CALL TRAC('B');
           TRACEOFF *** */
           CALL OPENRT;      /* OPEN RT FILE OF MODULE M */
           CALL READRR;      /* READ RT RECORD */
           DO WHILE REC<>'.';    /* NOT END OF RELOCATION TABLE */
              IF REC='R' AND RSEG=IDA THEN  /* RESOLVE THE ADDRESS */
                 CALL RESOLVE;
              IF REC = 'S'  /* 'SYMBOL' RECORD */
                 AND LSTSREQ  /* LISTING REQUIRED */
                 THEN CALL LSTSREF;  /* OUTPUT ONE SYMBOL */
              CALL READRR;    /* NEXT RECORD */
           END;
           TRACEON
           CALL TRAC('D');
           TRACEOFF *** */
           CALL DUMP(NIDA(M));    /* DUMP IDA OF MODULE M */
        END;

    /* TERMINATE */
        CALL FLUSH;    /* WRITE LAST RECORD */
        /* CLOSE OBJECT FILE */
        IF VMON(CLOSE,.OFCB) = 255 THEN CALL ERROR(2);
        /* PRINT SUMMARY */
        CALL PRLA(LOADADR);
        CALL PRL('LAST ADDRESS = $');
        CALL PRINTHA(LA-1);
        CALL CRLF;
        CALL PRL('MODULES LINKED: $');
        DO J = 0 TO MTOP;
           CALL CRLF;
           CALL PRIN(J);    /* PRINT NAME OF MODULE M */
        END;
        CALL CRLF;
        /* QUIT */
        CALL EXIT;

EOF
```

```
/*
              ******************************************
              *                                        *
              *  P11:  8080 / PDP-11 INTERFACE PROGRAM *
              *                                        *
              *       LUIZ PEDROSO - OCTOBER 1975      *
              *                                        *
              ******************************************
*/

    DECLARE LIT LITERALLY 'LITERALLY';
    DECLARE FOREVER LIT 'WHILE 1';

    /* CP/M SYSTEM CONSTANTS */
    DECLARE CPM   LIT '0';
    DECLARE FCBA  LIT '005CH';
    DECLARE BUFA  LIT '0080H';
    DECLARE FBASE LIT '3200H';
    DECLARE BDOS  LIT '3FFDH';

    /* I/O PRIMITIVES */
    DECLARE READCHAR        LITERALLY  '1';
    DECLARE PRINTCHAR       LITERALLY  '2';
    DECLARE PRINT           LITERALLY  '9';
    DECLARE OPEN            LITERALLY  '15';
    DECLARE CLOSE           LITERALLY  '16';
    DECLARE MAKE            LITERALLY  '22';
    DECLARE DELETE          LITERALLY  '19';
    DECLARE READBF          LITERALLY  '20';
    DECLARE WRITEBF         LITERALLY  '21';
    DECLARE INITDSK         LITERALLY  '13';
    DECLARE SETBUF          LITERALLY  '26';
    DECLARE LIFTED          LITERALLY  '12';

    /* OPERATION MODES */
    DECLARE NEUTRAL LIT '0';   /* PDP TO CRT, CRT TO PDP */
    DECLARE REC     LIT '1';   /* PDP TO DISK */
    DECLARE TRANS   LIT '2';   /* DISK TO PDP */

    /* SPECIAL CHARACTERS */
    DECLARE RUBOUT  LIT '7FH',
            PROMPT  LIT '25H',    /* PDP PROMPT (PERCENT) */
            CTRLW   LIT '17H',
            CTRLN   LIT '0EH',
            CTRLR   LIT '12H',
            CTRLT   LIT '14H',
            CTRLZ   LIT '1AH',
            CTRLC   LIT '03H',
            CR      LIT '0DH',
            LF      LIT '0AH',
            BEEP    LIT '07H';

    /* GLOBAL VARIABLES */
    DECLARE CRTBUF (200) BYTE;            /* BUFFER FOR CHARS FROM CTR */
    DECLARE PDPBUF LIT 'MEMORY';          /* BUFFER FOR CHARS FROM PDP */
    DECLARE DISKBUFA ADDRESS INITIAL (BUFA),
            DISKBUF BASED DISKBUFA BYTE; /* DISK I/O BUFFER */
    DECLARE CBF ADDRESS,     /* FIRST CHAR IN CRTBUF       */
            CBL ADDRESS,     /* LAST  CHAR IN CRTBUF       */
            CBN ADDRESS,     /* NUMBER OF CHARS IN CRTBUF  */
            PBF ADDRESS,     /* FIRST CHAR IN PDPBUF       */
            PBL ADDRESS,     /* LAST  CHAR IN PDPBUF       */
            PBN ADDRESS;     /* NUMBER OF CHARS IN PDPBUF */
    DECLARE LASTPB  ADDRESS,  /* LAST INDEX IN PDPBUF */
            LENGPB ADDRESS;   /* LENGTH OF PDPBUF     */
    DECLARE EOFILE BYTE,      /* 1 IF DISK EOFILE */
            DBP ADDRESS;      /* DISK BUFFER POINTER */
    DECLARE DFCBA ADDRESS INITIAL (FCBA),
            DFCB BASED DFCBA BYTE;        /* DISK FILE CONTROL BLOCK */
    DECLARE MODE BYTE;
    DECLARE (D3,D2,D1) BYTE;   /* DECIMAL COUNTER */
```

```
DECLARE BEEPED    BYTE;     /* 1 IF BEEP WAS SENT TO CRT */
DECLARE PREVIOUS  BYTE;     /* PREVIOUS CHARACTER WHILE RECEIVING */
DECLARE MISSED    BYTE;     /* 1 IF SUSPICIOUS OF MISSING DATA */
DECLARE C BYTE;             /* TEMPORARY */
DECLARE PECHO BYTE INITIAL (0);      /* 1 IF PDP ECHO PRINTING WANTED */
DECLARE CECHO BYTE INITIAL (0);      /* 1 IF CRT ECHO PRINTING WANTED */
DECLARE EOB  LIT '01H';   /* END OF BLOCK */
DECLARE RCVD LIT '2EH';   /* ACKNOWLEDGE */
DECLARE ECHOED BYTE;
DECLARE SAVEC BYTE;

/* I/O PORTS OPTIONS */
DECLARE OPC BYTE INITIAL(0);  /* CRT: PORTS 0,0,1 */
DECLARE OPP BYTE INITIAL(1);  /* PDP: PORTS 4,4,5 */

VMON: PROCEDURE(FUNC,INFO) BYTE;
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; RETURN A VALUE */
      GO TO BDOS;
      END VMON;

CMON: PROCEDURE(FUNC,INFO);
      DECLARE FUNC BYTE, INFO ADDRESS;
      /* ASK CP/M TO EXECUTE FUNC; NO VALUE RETURNED */
      GO TO BDOS;
      END CMON;

ICRTS: PROCEDURE BYTE;
      IF OPC THEN RETURN INPUT (5);   RETURN INPUT(1);
      END ICRTS;

IPDPS: PROCEDURE BYTE;
      IF OPP THEN RETURN INPUT(5);  RETURN INPUT(1);
      END IPDPS;

ICRTI: PROCEDURE BYTE;
      IF OPC THEN RETURN INPUT(4);   RETURN INPUT(0);
      END ICRTI;

IPDPI: PROCEDURE BYTE;
      IF OPP THEN RETURN INPUT(4);   RETURN INPUT(0);
      END IPDPI;

TOSCRT: PROCEDURE BYTE;
      /* TRUE IF OUTPUT LINE TO CRT READY */
      RETURN NOT ROR(ICRTS,2);
        END TOSCRT;

TOSPDP: PROCEDURE BYTE;
      /* TRUE IF OUTPUT LINE TO PDP READY */
      RETURN NOT ROR(IPDPS,2);
      END TOSPDP;

FROMSCRT: PROCEDURE BYTE;
      /* TRUE IF INPUT LINE FROM CRT READY*/
      RETURN NOT ICRTS;
      END FROMSCRT;

FROMSPDP: PROCEDURE BYTE;
      /* TRUE IF INPUT LINE FROM PDP READY */
       RETURN NOT IPDPS;
       END FROMSPDP;

OUTCRT: PROCEDURE(C);
      DECLARE C BYTE;
      /* OUTPUT CHARACTER C TO THE CRT */
      IF OPC THEN OUTPUT(4)= NOT C;
      ELSE OUTPUT(0)=NOT C;
      END OUTCRT;

OUTPDP: PROCEDURE(C);
      DECLARE C BYTE;
      /* OUTPUT CHARACTER C TO THE PDP */
```

```
      IF OPP THEN OUTPUT(4)= NOT C;
      ELSE OUTPUT(0)= NOT C;
      END OUTPDP;

PUTCRT: PROCEDURE (C);
      DECLARE C BYTE;
      /* SEND CHARACTER C TO THE CRT */
      DO WHILE NOT TOSCRT;  /* WAIT */
      END;
      CALL OUTCRT(C);
      END PUTCRT;

PUTPDP: PROCEDURE(C);
      DECLARE C BYTE;
      /* SEND CHARACTER TO THE PDP */
      DO WHILE NOT TOSPDP;  /* WAIT */
      END;
      CALL OUTPDP(C);
      END PUTPDP;

CRLF: PROCEDURE;
      /* SEND CARRIAGE-RETURN/LINE-FEED TO THE CRT */
      CALL PUTCRT(CR);     CALL PUTCRT(LF);
      END CRLF;

PRCRTL: PROCEDURE(A);
      DECLARE A ADDRESS;
      /* SEND MESSAGE AT A TO THE CRT, UNTIL $ IS DETECTED */
      DECLARE B BASED A BYTE;
      DO WHILE B <> '$';
         CALL PUTCRT(B);
         A = A + 1;
      END;
      END PRCRTL;

PRCRT: PROCEDURE(A);
      DECLARE A ADDRESS;
      CALL CRLF;
      CALL PRCRTL(A);
      END PRCRT;

GETCRT: PROCEDURE BYTE;
      /* READ A CHARACTER FROM THE CRT */
      DECLARE C BYTE;
      C = NOT ICRTI AND 7FH;
      IF ECHO OR (MODE=REC) THEN CALL PUTCRT(C);
      RETURN C;
      END GETCRT;

GETPDP: PROCEDURE BYTE;
      /* READ A CHARACTER FROM PDP */
      DECLARE C BYTE;
      C = NOT IPDPI AND 7FH;
      IF ECHO THEN CALL PUTPDP(C);
      /* CONVERT LOWER TO UPPER CASE */
      IF (C>=61H) AND (C<=7AH) THEN RETURN C AND 0DFH;
      RETURN C;
      END GETPDP;

GETCRTBUF: PROCEDURE BYTE;
      /* GET A CHARACTER FROM THE CRT BUFFER */
      DECLARE C BYTE;
      C = CRTBUF(CBF);  /* GET FIRST CHAR */
      IF (CBF := CBF+1) > LAST(CRTBUF) THEN CBF = 0; /* WRAP AROUND */
      CBN = CBN - 1;
      RETURN C;
      END GETCRTBUF;

GETPDPBUF: PROCEDURE BYTE;
      /* GET A CHARACTER FROM THE PDP BUFFER */
      DECLARE C BYTE;
      C = PDPBUF(PBF);  /* GET FIRST CHAR */
      IF (PBF := PBF+1) > LASTPB THEN PBF = 0;
```

```
      PBN = PBN - 1;
      RETURN C;
      END GETPDPBUF;

PUTCRTBUF: PROCEDURE(C);
      DECLARE C BYTE;
      /* PUT C INTO THE CRT BUFFER */
      IF (CBL := CBL+1) > LAST(CRTBUF) THEN CBL = 0;
      IF CBN=0 THEN CBL=CBF;
      CRTBUF(CBL) = C;
      CBN = CBN + 1;
      BEEPED = 0;  /* ALLOW BEEPING ON NEXT CRTBUF OVERFLOW */
      END PUTCRTBUF;

PUTPDPBUF: PROCEDURE(C);
      DECLARE C BYTE;
      /* PUT C INTO THE PDP BUFFER */
      IF (PBL := PBL+1) > LASTPB THEN PBL = 0;
      IF PBN=0 THEN PBL=PBF;
      PDPBUF(PBL) = C;
      PBN = PBN + 1;
      END PUTPDPBUF;

CRTBUFSFULL: PROCEDURE BYTE;
      RETURN CBN  = LENGTH(CRTBUF);
      END CRTBUFSFULL;

PDPBUFSFULL: PROCEDURE BYTE;
      RETURN PBN = LENGPB;
      END PDPBUFSFULL;

PRINTH1: PROCEDURE(C);
      DECLARE C BYTE;
      /* PRINT A HEXADECIMAL CHARACTER */
      IF C > 9 THEN CALL PUTCRT(C-10+'A');
      ELSE CALL PUTCRT(C+'0');
      END PRINTH1;

PRINTH2: PROCEDURE(C);
      DECLARE C BYTE;
      /* PRINT 2 HEXADECIMAL CHARACTERS */
      CALL PRINTH1(SHR(C,4));  CALL PRINTH1(C AND 0FH);
      END PRINTH2;

PRCOUNT: PROCEDURE;
      /* PRINT DECIMAL COUNTER */
      CALL CRLF;
      CALL PUTCRT('(');
      IF D1<> 0 THEN CALL PRINTH2(D1);
      IF D2 <> 0 THEN CALL PRINTH2(D2);
      CALL PRINTH2(D3);
      END PRCOUNT;

ZCOUNT: PROCEDURE;
      /* CLEAR DECIMAL COUNTER */
      D3, D2, D1 = 0 ;
      END ZCOUNT;

COUNTUP: PROCEDURE;
      /* ADD 1 TO DECIMAL COUNTER */
      D3 = DEC(D3 + 1);
      D2 = DEC(D2 PLUS 0);
      D1 = DEC(D1 PLUS 0);
      END COUNTUP;

INITSN: PROCEDURE;
      /* ENTER NEUTRAL MODE: CRT TO PDP, PDP TO CRT */
      MODE = NEUTRAL;
      CALL PRCRT('(H)S');
      CALL CRLF;
      END INITSN;

INITFCB: PROCEDURE;
```

```
      /* INITIALIZE FILE CONTROL BLOCK */
      DECLARE I BYTE;
      DO I=12 TO 15;
         DFCB(I)=0;
      END;
      END INITFCB;

PRFN: PROCEDURE;
      /* PRINT FILE NAME */
      DECLARE I BYTE;
      CALL PUTCRT(' ');
      DO I = 1 TO 11;
         IF DFCB(I) <> ' ' THEN CALL PUTCRT( DFCB(I) );
         IF I=8 THEN CALL PUTCRT('.');
      END;
      CALL PUTCRT(')');
      END PRFN;

INITSR: PROCEDURE;
      /* ENTER RECEIVING MODE: PDP TO DISK */
      MODE = REC;
      /* OPEN FILE FOR RECEPTION */
      CALL INITFCB;
      CALL CHON(DELETE,FCBA);  /* DELETE OLD VERSION */
      IF VHON(MAKE,FCBA) = 255 THEN /* OPENING ERROR */
         DO;  /* WARN OPERATOR */
            CALL PRCRT(.'(R. CANNOT CREATES');
            CALL PRFN;
            CALL INITSN;
            RETURN;
         END;
      /* OPEN OK; SET POINTERS */
      DFCB(32)= 0;   /* NEXT RECORD */
      DBP = 0;       /* NEXT CHARACTER */
      CALL ZCOUNT;   /* NO. CHARACTERS RECEIVED = 0 */
      PREVIOUS = ' ';        /* PREVIOUS CHARACTER RECEIVED */
      PBN, PBL,PBF = 0;
      CALL PRCRT(.'(R)S');
      CALL PRCRT(.'(R. CREATEDS');
      CALL PRFN;
      END INITSR;

INITST: PROCEDURE;
      /* ENTER TRANSMITING MODE: DISK TO PDP */
      MODE = TRANS;
      /* OPEN FILE FOR TRANSMISSION */
      CALL INITFCB;
      IF VHON(OPEN,FCBA) = 255 THEN /* OPENING ERROR */
         DO;  /* WARN OPERATOR */
            CALL PRCRT(.'(T. CANNOT OPENS');
            CALL PRFN;
            CALL INITSN;
            RETURN;
         END;
      /* OPEN OK; SET POINTERS */
      DFCB(32) = 0;    /* NEXT RECORD */
      DBP = 128;       /* NEXT CHARACTER (BUFFER EMPTY) */
      CALL ZCOUNT;     /* NO. CHARACTERS TRANSMITED = 0 */
      CALL PRCRT(.'(T)S');
      CALL PRCRT(.'(T. OPENEDS');
      CALL PRFN;
      EOFILE = 0;      /* NOT YET EOFILE */
      ECHOED=1;
      SAVEC=' ';
      END INITST;

BREAK: PROCEDURE;
      /* INTERRUPT PDP */
      DECLARE I BYTE;
      DO I = 1 TO 3;
         CALL PUTPDP(RUBOUT);
      END;
      END BREAK;
```

```
ABENDSR: PROCEDURE;
      /* ABORT CURRENT RECEPTION, RETURN TO NEUTRAL */
      CALL PRCRT(.'(R.ABEND)$');
      /* DISCARD PDP BUFFER CONTENTS */
      PBN, PBF, PBL = 0;
      /* DELETE FILE BEING RECEIVED */
      CALL CMON(DELETE,FCBA);
      /* SEND BREAK SIGNAL TO THE PDP */
      CALL BREAK;
      CALL INITSN;
      END ABENDSR;

ABENDST: PROCEDURE;
      /* ABORT CURRENT TRANSMISSION, RETURN TO NEUTRAL */
      CALL PRCRT(.'(T. ABEND)$');
      /* SEND BREAK SIGNAL TO THE PDP */
      CALL BREAK;
      CALL INITSN;
      END ABENDST;

GETDISKBUF: PROCEDURE BYTE;
      /* GET A CHARACTER FROM THE DISK BUFFER */
      DECLARE B BYTE;
      IF DBP >= 128 THEN /* DISK BUFFER IS EMPTY */
         DO;  /* READ NEXT RECORD INTO BUFFER */
            B = VMON(READBF,FCBA);       /* READ */
            IF B > 1 THEN /* ERROR */
               DO;
                  CALL PRCRT(.'(T. DISK READ ERROR)$');
                  CALL ABENDST;
                  RETURN 0;
               END;
            IF B = 1 THEN /* EOFILE */
               DO;
                  EOFILE = 1;
                  RETURN 0;
               END;
            DBP = 0;  /* BUFFER FULL */
         END;
      B = DISKBUF(DBP);
      DBP = DBP + 1;
      RETURN B;
      END GETDISKBUF;

WRPDPREC: PROCEDURE;
      /* WRITE ONE RECORD FROM PDPBUF INTO DISK */
      DECLARE I BYTE;
      /* MOVE RECORD TO BUFFER AREA */
      DO I = 0 TO 127;
         DISKBUF(I)=PDPBUF(PBF);
         PBF = PBF + 1;
      END;
      IF VMON(WRITEBF,FCBA) <> 0 THEN /* WRITE ERROR */
         DO;
            CALL PRCRT(.'(R. DISK WRITE ERROR)$');
            CALL ABENDSR;
            RETURN;
         END;
      IF PBF > LASTPB  THEN PBF = 0;
      PBN = PBN - 128;  /* 128 CHARACTERS WERE DUMPED */
      END WRPDPREC;

WRPDPBUF: PROCEDURE;
      /* WRITE WHOLE PDP BUFFER INTO DISK */
      DO WHILE (PBN AND 0177Q) <> 0;
         /* PAD UNTIL PBN IS A MULTIPLE OF 128 */
         CALL PUTPDPBUF(' ');
      END;
      DO WHILE PBN <> 0;
         /* WRITE NEXT RECORD INTO  DISK */
         CALL WRPDPREC;
      END;
```

```
      PBL,PBF=0;
      END WRPDPBUF;

REBOOT: PROCEDURE;
      /* GO BACK TO CP/M */
      IF MISSED THEN CALL PRCRT(.'(N. MAY HAVE MISSED PDP CHARS)$');
      CALL PRCRT(.'(N. REBOOTING)       $');
      CALL CMON(LIFTHD,0);
      GO TO CPM;
      END REBOOT;

HEXVAL: PROCEDURE (C) BYTE;
     DECLARE C BYTE;
      /* RETURN VALUE OF HEX CHAR C */
      IF (C>='0') AND (C<='9') THEN RETURN C-'0';
      RETURN C-'A'+10;
      END HEXVAL;

NCRLF: PROCEDURE;
      CALL CMON(PRINTCHAR,CR);      CALL CMON(PRINTCHAR,LF);
      END NCRLF;

YN: PROCEDURE;
      /* ASK OPERATOR IF YES OR NO */
      CALL CMON(PRINTCHAR,3FH);      /* QUESTION MARK */
      CALL CMON(PRINT,.' (Y/N): $');
      END YN;

YES: PROCEDURE BYTE;
      /* TRUE IF THE ANSWER IS 'Y' */
      RETURN VMON(READCHAR,0) = 'Y';
      END YES;

MAIN: /* INITIALIZE, THEN START LOOPING */
      CALL CMON(PRINT,.'DEFAULTS');     CALL YN;
      IF NOT YES THEN
      DO;  /* ASK FOR NEW PARAMETERS */
      /* SET I/O PORTS */
      CALL NCRLF;
CALL CMON(PRINT,.'SELECT PORTS (I,O,CTRL) - O:(0,0,1)  I:(4,4,5) $');
      CALL NCRLF;
      CALL CMON(PRINT,.'CRT: $');
      CPC  = HEXVAL(VMON(READCHAR,0));
      CALL NCRLF;
      CALL CMON(PRINT,.'TTY: $');
      CPP  = HEXVAL(VMON(READCHAR,0));
      CALL NCRLF;
      CALL CMON(PRINT,.'CRT ECHOS');    CALL YN;
      IF YES THEN CECHO = 1;
      CALL NCRLF;
      CALL CMON(PRINT,.'TTY ECHOS');    CALL YN;
      IF YES THEN PECHO = 1;
      END;
      CALL NCRLF;
      /* SET PDP BUFFER PARAMETERS */
      LENGPB = FBASE - .MEMORY;  /* USE ALL MEMORY UP TO FDOS */
      LASTPB = LENGPB - 1;
      /* SET BUFFER POINTERS */
      CBF, CBL, CBN = 0;
      PBF, PBL, PBN = 0;
      CALL CMON(INITDSK,0);    /* SELECT DISK */
      /* ENTER NEUTRAL STATE */
      MISSED, BEEPED = 0;
      CALL INITSN;

      DO FOREVER;

         IF FROMSCRT THEN /* INPUT AVAILABLE FROM CRT */
            DO;
               IF CRTBUFSFULL THEN
                  DO;  /* SEND A BELP */
                     IF NOT BEEPED THEN
                        IF (BEEPED := TOSCRT) THEN CALL PUTCRT(BEEP);
```

```
            END;
         ELSE /* BUFFER NOT FULL */
            DO;
               C = GETCRT;  /* GET INPUT FROM CRT */
               DO CASE MODE;
                  /* NEUTRAL: */
                  DO;
                     /* PUT C IN CRTBUF IF NOT A COMMAND */
                     IF C = CTRLW THEN CALL PRCRT('(N)$'); ELSE
                     IF C = CTRLT THEN CALL INITST;  ELSE
                     IF C = CTRLR THEN CALL INITSR;  ELSE
                     IF C = CTRLC THEN CALL REBOOT;  ELSE
                     /* NOT A COMMAND */ CALL PUTCRTBUF(C);
                  END;
                  /* RECEIVING: */
                  DO;
                     IF C = CTRLW THEN CALL PRCRT('(R)$'); ELSE
                     IF C = CTRLN THEN CALL ABENDSR;  ELSE
                     IF C = CTRLR THEN
                     DO;   /* PRINT CHAR COUNT */
                        CALL PRCOUNT; CALL PUTCRT(')');
                     END;      ELSE
                     /* NOT A COMMAND */ CALL PUTCRTBUF(C);
                  END;
                  /* TRANSMITING: */
                  DO;
                     IF C = CTRLW THEN CALL PRCRT('(T)$'); ELSE
                     IF C = CTRLN THEN CALL ABENDST;  ELSE
                     IF C = CTRLT THEN
                     DO;   /* PRINT CHAR COUNT */
                        CALL PRCOUNT; CALL PUTCRT(')');
                     END;      ELSE
                     /* NOT A COMMAND */ CALL PUTCRTBUF(C);
                  END;
               END;  /* CASE */
            END;  /* ELSE */
      END;  /* FROMSCRT */

   IF FROMSPDP THEN /* INPUT AVAILABLE FROM PDP */
      DO;  /* PUT DATA INTO PDPBUF UNLESS FULL */
         IF MODE = REC THEN   /* RECEIVING */
            DO;
               IF PDPBUFSFULL THEN /* MUST DUMP TO DISK */
                  DO;
                     CALL PRCRT('(R. FORCED TO WRITE DISK)$');
                     CALL WRPDPREC; /* WRITE 1 RECORD FROM PDPBUF */
                     MISSED = 1;     /* MAY HAVE MISSED PDP CHARS
                                        WHILE WRITING DISK */
                  END;
               /* BUFFER NOT FULL */
               C = GETPDP;  /* GET CHARACTER FROM PDP */
               IF ((PREVIOUS=LF) OR (PREVIOUS=CR)) AND C=PROMPT
                  THEN
                  /* PDP PROMPTING, THEREFORE END OF RECEPTION */
                  DO;
                     CALL PUTPDPBUF(1AH);  /* CONTROL-Z */
                     CALL WRPDPBUF; /* WRITE WHOLE PDPBUF TO DISK */
                     /* NOW CLOSE FILE RECEIVED */
                     IF VMSM(CLOSE,FCBA) = 255 THEN /* ERROR */
                        CALL PRCRT('(R. CANNOT CLOSE$');
                     ELSE CALL PRCRT('(R. CLOSED$');
                     CALL PREN;
                     CALL PRCOUNT; /* NO. CHARS RECEIVED */
                     CALL PRCRTL(' BYTES RCVD; END R)$');
                     CALL INITSN;
                     CALL PUTCRT(C);
                     CALL CRLF;
                  END;
               ELSE IF C=EOB  /* END OF BLOCK */ THEN
                  DO;
                     CALL WRPDPBUF;
                     CALL PRCRT('(R. RCVD BLOCK)$');
                     /* ACKNOWLEDGE */
```

```
                        CALL PUTCRTBUF(RCVD); CALL PUTCRTBUF(CR);
                     END;
                  ELSE /* NOT END OF RECEPTION */
                     DO;
                        PREVIOUS = C;
                        CALL PUTPDPBUF(C);
                        CALL COUNTUP;  /* INCR. NO. CHARS RECEIVED */
                     END;
               END;  /* RECEIVING */
            ELSE  /* NEUTRAL OR TRANSMITTING */
               DO;
                  IF PDPBUFSFULL THEN MISSED = 1;
                  /*(WAIT UNTIL NOT FULL; MAY MISS THE CHARACTER)*/
                  ELSE
                     DO;
                        C=GETPDP;
                        IF SAVEC=C THEN ECHOED=1;
                        CALL PUTPDPBUF(C);
                     END;
               END;
         END;  /* FROMSPDP */

      IF TOSCRT THEN /* CAN SEND DATA TO CRT */
         DO;
            IF MODE <> REC THEN
               DO;  /* CHECK IF SOMETHING FOR THE CRT */
                  IF  PBN > 0 THEN CALL PUTCRT(GETPDPBUF);
               END;
            /* ELSE (RECEIVING): DATA FROM PDP GOES TO DISK, NOT CRT */
         END;  /* TOSCRT */

      IF TOSPDP THEN /* CAN SEND DATA TO PDP */
         DO;
            IF MODE <> TRANS THEN
               DO;  /* CHECK IF CRT HAS SOMETHING FOR THE PDP */
                  IF CBN > 0 THEN CALL PUTPDP(GETCRTBUF);
               END;
            ELSE /* TRANSMITING */
            IF ECHOED THEN
               DO;
                  C = GETDISKBUF;  /* GET A CHAR FROM DISK */
                  IF EOFILE OR(C=CTRLZ) THEN /* END OF TRANSM */
                     DO;
                        CALL PRCOUNT; /* NO. BYTES XMTD */
                        CALL PRCRTL(.' BYTES XMTD; END T)S');
                        CALL INITSN;
                     END;
                  ELSE  /* NOT EOFILE */
                  IF SAVEC=CR AND C=LF THEN;  /* DO NOT XMIT LF */
                  ELSE  /* XMIT A CHAR */
                     DO;
                        CALL PUTPDP(C);
                        ECHOED=0;
                        SAVEC=C;
                        CALL COUNTUP; /* INCR. NO. CHARS TRANSMITED */
                     END;
               END;  /* TRANSMITING */
         END; /* TOSPDP */

   END;  /* DO FOREVER */
EOF
```

```
/*

*********************************************
*                                           *
*  Y16: YACC Tables to Intel's hex format   *
*                                           *
*********************************************

*/

char cr 015;          /* carriage-return */
char lf 012;          /* line-feed */
char ctrlz 032;       /* control-z character */
char rl;              /* record length */
int total 0;          /* total no. bytes in the tables */
char cs;              /* check sum */
int la;               /* load address */
int bbuf[40];         /* buffer for bytes to be converted to hex */
int buflen;           /* bbuf length */
int outbuf[259];      /* output buffer */
char *outfn "hout"; /* name of the output file */
char crt 0;           /* 1 if output goes to crt */

extern int yyact[];     /* Yacc tables */
extern int yypact[];
extern int yyr1[];
extern int yyr2[];
extern int yygo[];
extern int yypgo[];
extern char *yysterm[];

yth(frw,slot) int frw; char slot;
   { /* create a file containing Yacc tables and a
        list of reserved words; frw is the index
        of the first reserved word to be moved to
        the output file (frw=0 moves all reserved
        words); if slot=1 each reserved word is preceded
        by 2 zeroed bytes */
     int i, j;
     int *p;
     char c;

     initialize(256,29);

     /* dump numeric tables */
     dump(&yyact);
     endtable(-1);
     dump(&yypact);
     endtable(-1);
     dump(&yyr1);
     endtable(-1);
     dump(&yyr2);
     endtable(-1);
     dump(&yypgo);
     endtable(-1);
     p = &yygo[0];
     while ( p < &yypgo[0] ) wra(*p++);
     endtable(0);

     /* dump symbol list (reserved words) */
     for (i=frw; yysterm[i]!=0; i++)
        { wrb(i & 0377);     /* reserved word no. - 256 */
          if (slot) wra(0); /* create slot for the link field */
          for (j=0; (c=yysterm[i][j]) != '\0'; j++) wrb(c);
          wrb(0);           /* name end flag */
        }
     endtable(0);

     terminate();
   }
```

```
dump(p) int *p;
   { /* dump a table starting at p, until a -1 is found */
     while (*p != -1) wra(*p++);
     return p;
   }


initialize(ia,bl) int ia,bl;
   { int fd;
     la = ia;          /* initial load address */
     rl = 0;
     cs = 0;
     buflen = bl;      /* buffer length */
     if (crt) return; /* output goes to console */
     /* else open output file */
     if ((fd=creat(outfn,0600))<0)
        { printf("Cannot create %s\n",outfn);
          exit();
        }
     /* initialize output buffer */
     outbuf[0] = fd;
     outbuf[1] = 0;
     outbuf[2] = 0;
   }


wra(a) int a;
   { /* put 2 bytes into bbuf */
     wrb(a&0377);  /* low byte first */
     wrb(a>>8);    /* then high byte */
   }


wrb(c) char c;
   { /* put character c into bbuf */
     if (rl >= buflen) /* buffer full */ hflush();
     bbuf[++rl]=c;
   }


hflush()
   { /* dump bbuf in HEX format */
     int i;
     if (rl==0) /* buffer empty */ return;
     total = total + rl;  /* no. bytes in the table */
     putchr(':');
     put2h(rl);
     put4h(la);     /* load address */
     put2h(0);      /* record type  */
     for (i=1; i<=rl; i++) put2h(bbuf[i]);
     cs = -cs;
     put2h(cs);
     putchr(cr);
     putchr(lf);
     la = la + rl; /* load address for next record */
     rl = 0;       /* reset buffer pointer */
     cs = 0;       /* reset check sum */
   }


put4h(i) int i;
   { /* write 4 hex digits representing i */
     put2h(i>>8);      /* high byte first */
     put2h(i & 0377);  /* then low byte */
   }


put2h(c) char c;
   { /* write 2 hex digits representing c */
     putchr( hex(  (c>>4) & 017 ) );
     putchr(hex(c & 017));
     cs = cs + c; /* and compute check sum */
   }
```

```
hex(c) char c;
   { /* return a hex printable character */
     if (c<=9) return(c+'0');
     return( c-10+'A' );
   }


putchr(c) char c;
   { if (crt) putchar(c);  /* output goes to console */
     else putc(c,outbuf);
   }


endtable(flag) int flag;
   { /* terminate a table; if flag!=0 write flag at the end of table */
     if (flag!=0) wra(flag); /* write flag characters */
     hflush();  /* flush last record */
     /* write the 'flag' record */
     putchr(':');
     put2h(0);
     put4h(0);
     put4h(0);
     putchr(cr);
     putchr(lf);
     rl = 0;  cs = 0; /* reset things for next record */
   }


terminate()
   { if (crt) exit();
     putchr(ctrlz);     /* write 8080 eofile character */
     fflush(outbuf);    /* flush output buffer */
     printf("Tables: %d bytes\n", total);
     exit();            /* close files and quit */
   }
```

```
/*

*************************************************
*                                               *
*  SENDII: hex files from the PDP to the 8080   *
*                                               *
*************************************************

*/

/* SENDII sends blocks of Intel's hex format records
   (generated by the PL/M compiler) to the 8080; the
   records are received and stored in the floppy disk
   by P11, the 8080/PDP-11 interface program */


char cr 015;        /* carriage return */
char lf 012;        /* line feed */
char  EOB  1;       /* end of block */
char RCVD '.';      /* acknowledge from the 8080 */
char c;
int total 0;        /* total no. bytes transmitted */
int ibuf[259];      /* input buffer */
int nr 0;           /* no. hex records sent per block */
int nrmax 80;       /* no. max. hex records per block */
int fd;


main(argc,argv) int argc; char **argv;

{ /* open hex file to be sent */
  argv++;
  if ((fd=open(*argv,0))<0)
     { eot();    /* terminate transmission */
       printf("Cannot open %s\n", *argv);
       exit();
     }

  /* initialize input buffer */
  ibuf[0]=fd;
  ibuf[1]=0;
  ibuf[2]=0;

  /* skip PL/M symbol table in the input file */
  while((c=getc(ibuf))!=':') ;

  /* send blocks of at most nrmax hex records */
  while(1)     /* do forever */
    { if(c == -1) /* eof */ terminate();
      if(c == ':') /* begin of a hex record */
         { if(nr>=nrmax)  /* already sent a whole block */
              { putchar(EOB); /* send end of block signal */
                while (getchar()!=RCVD) ;  /* wait acknowledge */
                nr = 0;
              }
           else nr++; /* start sending new record */
         }
      putchar(c);    /* send a character */
      total++;       /* count characters */
      c = getc(ibuf);  /* get next char */
    }
}


terminate()

{ /* terminate transmission */
  eot();  /* send eot signal */
  c=getchar(); /* wait for some input from the 8080 */
  c=getchar();
  printf("%d bytes transmitted \n", total);
  exit();
```

```
}


eot()

{ /* send end of transmission signal */
  putchar(cr);
  putchar(lf);
  putchar('%');
}
```

LIST OF REFERENCES

1 - Abrahams, P., "Structured Programming Considered Harmful", SIGPLAN Notices, v.10, n.4, p.13-24, April 1975

2 - Aho, A. V., and Johnson, S. C., "LR Parsing", Computing Surveys, v.6, n.2, p.99-124, June 1974

3 - Aho, A. V., and Ullman, J. D., The Theory of Parsing, Translation and Compiling, vs. I and II, Prentice-Hall, 1972

4 - Dijkstra, E. W., "Notes on structured programming", in Structured Programming by Dahl, O. J., Dikstra, E. W., and Hoare, C. A. R., Academic Press, 1972

5 - Haines, E. C., "AL: A Structured Assembly Language", SIGPLAN Notices, v.8, n.1, p.15-20, January 1973

6 - Intel, 8080 Assembly Language Programming Manual, Intel Corporation, 1975

7 - Intel, Intellec 8/Mod 80 Microcomputer Development System Reference Manual, Intel Corporation, 1974

8 - Intel, 8008 and 8080 PL/M Programming Manual, Intel Corporation, 1975

9 - Johnson, S. C., YACC - Yet Another Compiler-Compiler, Bell Laboratories, 1974

10 - Kernigham, B. W., Programming in C - A Tutorial, Bell Laboratories, 1974

11 - Kildall, G. A., CP/M: A Disk Control Program for Microcomputer System Development, Microcomputer Applications, June 1975

12 - Knuth, D. E., "On the translation of languages from left to right", Information and Control, v.8, p.607-639, 1965

13 - Knuth, D. E., "Structured Programming with go to statements", Computing Surveys, v.6, n.4, p.261-301, December 1974

14 - Maanuson, R. A., "A Structured Assembly Language Source Program Generator", NTIS Document PB225094, September 1973

15 - McKeeman, W. M., Horning, J. J., and Wortman, D. B., A

Compiler Generator, Prentice-Hall,1970

16 - Meissner, L. P., "On extending Fortran Control Structures to Facilitate Structured Programming", SIGPLAN Notices, v.10, n.9, p.19-30, September 1975

17 - Popper, C., "SMAL - A structured macro-assembly language for a microprocessor", Proceedings COMPCON Fall, p.147-151, 1974

18 - Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, v.17, n.7, p.365-375, July 1974

19 - Sammet, J. E., "Roster of Programming Languages for 1973", SIGPLAN Notices, v.9, n.11, p.18-31, November 1974

20 - Theis, D. J., "Microprocessor and Microcomputer Survey", DATAMATION, p.90-101, December 1974

21 - Wirth, N., "On the Composition of Well-Structured Programs", Computing Surveys, v.6, n.4, p.247-259, December 1974

22 - Wirth, N., "The programming language Pascal", Acta Informatica 1, n.1, p.35-63, 1971

23 - Wirth, N., "PL360, a programming language for the 360 computers", Journal of the Association for Computing Machinery, v.15, p.37-74, 1968

24 - Wulf, W. A., Russel, D. B., and Haberman, A. N., "BLISS - a language for systems programming", Communications of the ACM, v.14, n.12, p.780-790, December 1971

25 - Yasaki, E. K., "The Emerging Microcomputer", DATAMATION, p.81-86, December 1974

26 - Zelkowitz, M. V., "Structured Machine Languages", IEEE Computer Society Repository #R74-37

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	Assoc Professor G. A. Kildall, Code 72Kd Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	Asst Professor V. M. Powers, Code 52Pw Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
6.	Assoc Professor U. R. Kodres, Code 72Kr Computer Science Group Naval Postgraduate School Monterey, California 93940	1
7.	LCDR L. R. B. Pedroso, Brazilian Navy Diretoria de Eletronica da Marinha Ilha das Cobras Rio de Janeiro, Brazil	2