PL/2100 AND HPCOM

Dedicated

to my

Parents

PROGRAMMING LANGUAGE 2100
AND
THE COMPILER HPCOM

By

DILIP K. ROY

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
November 1973

MASTER OF SCIENCE
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Programming Language 2100 and its Compiler HPCOM

AUTHOR: Dilip K. Roy

SUPERVISOR: Professor N. Solntseff

NUMBER OF PAGES: x, 264

Abstract

This report contains a description of the programming language PL/2100, which is a procedure oriented, block structured language with an extensive set of operators, including arithmetic, relational, logical, bit manipulation and shift operators. It is designed for writing PL/2100 programs that conform to the standard of structured programming and for efficiently expressing and implementing algorithms written in it.

This report, also, contains a description of the one-pass working compiler (HPCOM) (for PL/2100) written for the HP/2100A minicomputer.

ACKNOWLEDGEMENTS

It is a pleasure to thank Professor N. Solntseff for his conscientious and valuable supervision during the course of this work and many comments and suggestions regarding the form and content of this thesis.

I would particularly like to thank Professors D. Kenworthy and R. Rink for their critical reading of the manuscript of this thesis.

I thank Mrs. H. Kennelly for her excellent and careful typing of this thesis.

I am thankful to the Department of Applied Mathematics, McMaster University for the financial assistance made available to me during the course of this work.

I would like to thank my fellow graduate students, the staff and the faculty of the Department of Applied Mathematics, for their help to make my stay enjoyable.

Finally, I like to thank all my friends I have met through F.R.O.S., for their cooperation and help to make my stay enjoyable and fruitful.

TABLE OF CONTENTS

Page

CHAPTER I: INTRODUCTION

CHAPTER II: COMPILING AND HPCOM COMPILER

CHAPTER III: CODE GENERATION

Page

CHAPTER IV: I/O IN HPCOM, INTERFACE WITH OPERATING SYSTEM AND SEGMENTATION

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Comparison of High Level Language and Assembly Language

1.1.1 Assembler Language

An assembler language is a symbolic form of machine language. While machine language is numeric, assembler language allows alphabetic names for operation codes and storage locations.

1.1.1.1 Advantage of Assembler

Assembler language permits the symbolic writing of machine language instructions, thus contributing to the speed and accuracy of the programming and debugging processes.

Through an assembler, users can also access all registers available for programming for the machine.

1.1.1.2 Disadvantage of Assembler

Only one data type viz. WORD of a machine is available.

Operations and access on structured data need programming and required structure is built into the program.

Only two control structures are available for programming i) Unconditional jump instructions (ii) Conditional skip instructions.

1.1.1.3 Conclusion

Hence a program in assembler is necessarily built up of very small parts joined by jump instructions.

1.1.2 High Level Language

A high level language is one which is independent of the features of a particular machine. Hence it is more easily adaptable by a user who does not know any feature of a machine.

1.1.2.1 Advantages

i) Many data types (e.g. in PASCAL we have an infinite no. of possible data types) are available.

Structure can be built into data where it belongs and need not be built into programs (as in the case of an assembler language).

ii) Reasonable number of control statements could be made available in the language. So a program can be written in a realtively small number of parts (compared to an assembler program of similar size), the flow of control into and from which is easily discernible provided certain rules (as provided in the language) are followed.

1.1.2.2. Disadvantages

The program and data of a high level language are removed from the hardware of a machine. There may be machine instructions which cannot be used (even if they could simplify the processing of a particular problem), as problem-oriented machine codes cannot be produced by a compiler of a language.

1.2 Need for Amalgamation of High Level Language and Assembler Language

The obvious answer to the possibility of getting the benefits from both assembler and high level languages, is to merge the low-level access of registers and instructions of a machine with the availability of numerous data types and control structure of a high level language.

This has been done before viz PL/360(W1) and SUE/360(K1) but intentional hardware dependency of a language means that the design of such a language must be done right from the beginning for each class of computer.

PL/2100 (Programming Language 2100) has been written for HP/2100A (Hewlett-Packard 2100A) with the intention of incorporating the above-mentioned features in the language.

1.3 Minicomputers and Need for High Level Languages for Minicomputers

1.3.1 Minicomputers

Although minicomputers have been available for many years, the full range of their applicability to all aspects of computing is, only now beginning to be adequately explored.

The minicomputers are mini in several ways

i) mini wordlength

The wordlength of the most common mini machines (e.g. PDP, HP) is 16 bits, although minicomputers are also available with word sizes of 8, 12 and 18 bits.

ii) mini memory size

Memory size of minicomputers have been, traditionally, small. They usually have 4K or 8K (1K = 1024 words) of memory, although minicomputers can be expanded into larger memory sizes.

iii) mini cost

May be the most important and attractive feature is the low cost of the minicomputers.

Recent advances in solid state circuit technologies have allowed instruction sets of minicomputers to be sophisticated, keeping the cost within a reasonable range.

Even though internal speed of minicomputers are comparable to those of larger machines, the throughput is smaller than the larger processors because of the short word-lengths.

There are various usages of the minicomputers, e.g. process control, to give greater flexibility to larger machine by providing time sharing or remote job entry, teaching of machine organization because of the simplicity of the hardware etc.

A great enhancement in the use of minicomputers has been because of the possibility of using Disc Operating System for the minicomputers. This, not only gives a simple and flexible operating system from the programmer's point of view, but also gives a huge secondary storage (in disc) complementing the small memory size of the minicomputers.

1.3.2 Need for High Level Language for Minicomputers

The recent advances in the hardware design of minicomputers, have not been paralleled by the development of software. System programs for minicomputers have, generally, been written in the assembler language of the host machine.

Sammet(S1) has indicated that the advantages of high level language for software implementation:

i) Easy conversion to another machine.

ii) Greater ease for a person to pick up somebody else's work.

Except for a few exceptions, high level system languages for minicomputers are not available at present.

The reasons behind the use of low level languages for minicomputer software development were that

i) The software would have to be written only once and the best possible code sequence should be used.

ii) Compilers for high level languages could not produce as good a code as a programmer familiar with the idiosynchrasies of the machine.

iii) A good compilor for an acceptable high level language could not meet the memory size restraints of minicomputers.

The new generation of minicomputer however, provides the capability of using a high level language for software development as the minicomputers have sophisticated instruction sets and become versatile to handle large programs. This capability is enhanced because of the availability of discs.

1.4 Existing High Level Machine-oriented Language

The first of these two types of languages is PL/360 written for IBM/360 machines(W1).

The other languages have been SUE/360(K1) written also for IBM/360 machines, the BLISS system implementation language(W2) written for PDP-10 and SUE system language for PDP-11 family machines(K1).

The Burrough family machines are designed with ALGOL in mind. These machines have no symbolic machine language as such! ALGOL is the machine language and system software is implemented in an extended version of ALGOL.

1.5 Features of HP/2100A Machine

The section is devoted to a summary of the different features of HP/2100A, so that relevant instruction sets can be chosen from PL/2100 as well as to get a better understanding of the language PL/2100.

1.5.1 Word-Length

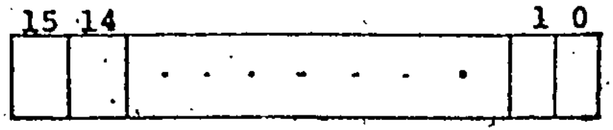


fig. 1.5 HP WORD

HP/2100A has a 16-bit word length and is only <u>word</u> addressable.

The lower order 10 bits (0-9) are used for addresses in simple memory instructions whereas extended memory instructions may have addresses up to 16 bits long.

1.5.2 Registers

The 2100A computer has six 16-bit working registers (including two accumulators A- and B-registers), two one-bit registers (E- and O-registers) and (on the operating panel) one 16-bit display register.

i) A- and B-registers

These are two accumulators which can hold the result of arithmetic operations (independent of each other). These are the absolute locations 000000B and 000001B in the machine memory and hence, can be accessed through the memory reference instructions (both single and extended memory reference instructions).

ii) E-register

This is a one-bit register which can be used with A- and B-registers for many shift and rotate instructions. This is known as the Extend register.

iii) O-register

This one-bit register is known as overflow-register and holds the overflow condition occurring from an arithmetic operation.

iv) M-register

It holds the address of the memory cell currently being read from or written into.

v) T-register

All data transferred into and out-of memory is routed through memory data register.

vi) S-register

It is a 16-bit utility register. In the halt mode of the machine, it can be manually loaded via display register on the panel. In the run mode it can be addressed as an I/ϕ device (select code 01).

vii) Registers available through microprogramming

There are other registers available in HP/2100A, which are available only through microprogramming, not through software programming.

For example:

a) Q-, F-registers

These are 16-bit accumulators. Special microprograms must be written in order to access these registers.

b) Scratch pad registers

Like the Q- and F-, the four scratch pad registers are available to software by special microprogramming.

The detailed discussion has been given in the booklet (H1) for microprogramming (Hewlett Packard).

1.5.3 Memory

The 2100A computer can be equipped with any of six memory configurations from 4K to 32K (1K = 1024 words). The available configurations, which determine the addressing range are: 4K, 8K, 12K, 16K, 24K and 32K.

1.5.3.1 Paging

The computer memory is logically divided into pages of 1024 words each. A page is defined as the largest block of memory which can be directly addressed by the memory address bits (0-9) of a memory reference instruction (single length).

Provision is made to address directly one of the two pages: page zero (base page) and the current page (in which instruction itself is located). A memory reference instruction word includes a bit (bit 10) to specify one or the other of these two pages. To address locations in any other page, indirect addressing is used. Page reference is specified by bit 10 as follows:

Logic 0 = Page Zero (Z)
Logic 1 = Current Page (C)

1.5.3.2 Addressing

All addressing in HP/2100A is done through the memory reference instructions. A HP/2100A memory reference instruction word contains

(i) for single length instructions [fig. 1.5.3.2.1]
  a) bit 15 for direct or indirect addressing
  b) bit 10 for addressing to one of two pages

viz page zero or current page

15	14 ..... 11 10	9 0
D/I	INSTRUCTION Z/C	MEMORY ADDRESS

fig. 1.5.3.2.1 Single Length Memory Instruction Word

c) bits 11-14 for instructions

d) bits 0-9 for addresses.

ii) <u>for extended arithmetic instructions</u> [fig. 1.5.3.2.2]

a) First word is the instruction itself.

b) Bit 15 of the memory address word is for direct or indirect addressing.

c) The second word is the address word.

INSTRUCTION	
D/I	MEMORY ADDRESS

fig. 1.5.3.2.2 Extended Length Memory Instruction Word

1.5.3.3 Indirect Addressing

There is no index register available but a bit is available in memory reference instruction to indicate indirect addressing.

For single length memory reference instructions, bit 15 of the instruction word is used; for extended arithmetic memory reference instructions, but 15 of the address word is used. Indirect addressing uses the address part of the instruction to access another word in memory, which is taken as a new memory reference for the same instruction. This new address word is a full 16-bits long, 15 bits of address plus another direct or indirect bit. 15-bit length of address permits access to any location in memory. The first address obtained in indirect phase which does not specify another indirect level becomes the effective address for the instruction.

Direct or Indirect addressing is specified by bit 15 is as follows:

Logic 0 = Direct

Logic 1 = Indirect.

1.5.4. Instruction Format

Instructions for the HP/2100A have four formats. Instructions are classified according to formats.

1.5.4.1 Memory Reference

Single length memory reference instructions: Format is given in fig. 1.5.4.1.

15	14 . . 11	10	9 0
D/I	INSTRUCTION	Z/C	MEMORY ADDRESS

fig. 1.5.4.1 Single Length Memory Instruction

Instruction is 4-bit long and is placed in bits 14-11 (inclusive).

1.5.4.2 Extended Arithmetic Memory Reference Instructions Instructions

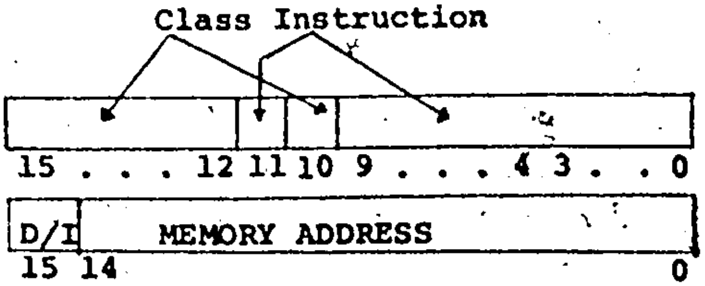


fig. 1.5.4.2 Extended Arithmetic Memory Instruction

Instruction code is given in the first word and the address is taken from the next word.

1.5.4.3 Register Reference

The 39 register reference instructions execute various functions on data contained in the A-, B-, E-registers.

The instructions are divided into two groups, the shift-rotate group and alter skip group. In each group, several instructions may be packed into one word (termed "micro-instructions" in the HP literature). Since two groups are separate and distinct the packed instructions from two groups cannot be mixed.

1.5.4.3.1 Shift-rotate Group

There are 20 instructions in the shift-rotate group. The bit 10 is zero for skip-rotate group.

1.5.4.3.2 Alter-skip Group

There are 19 instructions in the alter-skip group. This group is specified by "1" in bit 10.

A detailed discussion of these instructions and the rules of packing microinstruction have been given in Appendix B.

The figure 1.5.4.3.1 shows register reference instruction format.

15 12	11	10	0................0
CLASS	A/B	S/A	MICROINSTRUCTIONS

fig. 1.5.4.3.1 Register Reference Instruction Format

A/B → denotes A- or B-register

Logic 0 → A

Logic 1 → B

S/A → denotes Skip-rotate or Alter-skip groups

Logic 0 → S

Logic 1 → A .

1.5.5 Data Format

The basic data format for the 2100A computer is a 16 bit word. Bit positions are numbered from 0 through 15, in order of increasing significance. Data are stored in two's complement. Bit 15 of the data format is used for the sign bit, a "0" in this position indicates a positive number and a "1" indicates a negative number. The data is assumed to be a whole number,thus binary point is assumed to be the right of the number.

The basic word can be divided into two 8-bit bytes. The byte format is used for character-oriented I/O devices. Packing of two bytes into one word is accomplished by the software drivers. In I/O operations the higher order byte (Byte 1 viz 8-15) is the first to be transferred.

1.5.6 Conclusion

The section 1.5 has given briefly some of the salient features of the HP/2100A computer. The details of the other features may be obtained from the Hewlett-Packard reference manuals(H2,H3).

1.6 Programming Language 2100 (PL/2100)

1.6.1 Choice of High Level Language

The high level language chosen for PL/2100 is PASCAL(W2). The reasons for the choice of this most recently developed language are:

i) Beautiful and powerful data structure.

ii) Ease of extension for implementation by bootstrapping.

1.6.2 Choice of Features of HP/2100A Assembler

The choice of HP/2100A instructions which will appear as operators in PL/2100A is difficult. For simplicity, at present, only few shift instructions have been chosen as shift operators.

These instructions are ALS, ARS, RAL, RAR, and ALF and corresponding instructions for the B-register.

Other instructions may be included easily and the compiler can be modified, accordingly, without much difficulty. It should not take more than a month for a person who is familiar with the compiler.

The reason for the choice of these instructions as operators, is that they allow shifting to be done on the contents of a particular register.

The memory reference instructions chosen are only IOR and XOR as most of the other memory reference instructions could very well be substituted by different statements of the language PASCAL.

1.6.3 Form of PL/2100

The detailed description of the language in BNF is given in the Appendix A.

Several of the most salient design features are:

a) The main statement constructions are the assignment, while, if-then-else, case, go to, repeat and call statements.

b) Every program consists of a sequence of procedures which can access a set of global variables, parameters or local variables.

c) These are compound statement constructions as well as block constructions.

d) Procedure may be recursive if they are so declared.

e) An extensive set of operators are permitted in an expression. These are arithmetic, logical, relational and shift operators.

f) A wide variety of data types is allowed. Scalar, subrange, Array and Record types. Various other data types can be formed with these basic data types.

g) Another feature is the introduction of "Synonymy" between different simple variables ie. a number of simple variables can be declared to be "synonymous" (a term borrowed from PL/360$^{(W1)}$). In other words, two or more identifiers may refer to one storage location. This is close to "equivalence" statement used in FØRTRAN.

1.7 Purpose of This Project

This project is, mainly, concerned with the design of a high level language (PL/2100) for the HP/2100A computer and produces a working compiler for it.

The language PL/2100 has been, carefully, designed with the hope of:

i) Easy extension.

ii) Varieties of data structures.

The ease of extension and the various data structure make it attractive to be used for the minicomputer HP/2100A which, in the present installation, does not support a compiler for this type of high level language.

The compiler has been written in PASCAL (which is available on CDC-6400) and is one pass. It runs on the CDC-6400 and produces relocatable binary which can, easily, be interfaced with the Disc Operating System (DOSM) of the HP/2100A computer.

The compiler, at the present stage, is far from being an ideal one. It does all the basic things necessary to handle expressions and various statements, but does not produce codes for procedures.

HPCOM (the compiler for PL/2100) as a one-pass system will not fit into HP/2100A machine. [It takes about 55K at the present stage]. The DOSM system of HP/2100A allows segments of a program residing on the disc , to be brought onto the memory. Attempts have been made to segment the HPCOM logically, so that different segments could be brought in physically from the disc to memory. Logical segmentation was possible and it would have taken 9 or 10 passes to compile a program in PL/2100. This is time consuming and too complicated. It was thought to be wise to discard this approach. (A detailed discussion of segmentation will be given in Chapter IV). Hence there remain two possibilities viz either to take a more limited subset of PL/2100 or use the CDC-6400 computer for developing the software for HP/2100A. The latter is considered to be better and more feasible. HPCOM runs on CDC-6400.

Though more work has to be done to make the compiler better, the present work is a first important step towards a more complete system.

We will describe the method of compiling and different aspects of HPCOM in Chapter II. Chapter III deals with the code generation part of the compiler HPCOM. In Chapter IV, we will describe I/ϕ routines, the interface with the operating system and the segmentations of HPCOM. In Chapter V, we have given the concluding remarks along with the limitations and the possible modifications of HPCOM.

CHAPTER II

COMPILING AND THE HPCOM COMPILER

2.1 One-pass Compiler

The key to an efficient compiler for a fast computer with a relatively large main store is the one-pass scheme. It minimizes the number of references to secondary store which involve the operating system and exceed all other processes by orders of magnitude of time consumption. The restrictions imposed on the language due to the choice of a one-pass scheme, are minimal (viz the objects have usually to be declared textually prior to being referenced. Note that programming language 2100 has been designed keeping this in mind) and the complications due to unavoidable forward references are small$^{(W3)}$.

Though the HP/2100A does not have large core, a one-pass scheme was chosen. In the beginning it was supposed to be bootstrapped onto HP/2100A but later the idea was discarded as was indicated in Chapter I. The HPCOM (compiler) runs on the CDC-6400.

The HPCOM compiler generates relocatable binary code for HP/2100A. The gain in compilation time (on the CDC-6400) compared to the time in many-pass compiler is, somewhat, reduced by the use of standard relocatable loader (involving the operating system of HP/2100A). The advantage is the ability to merge "binary" programs after compilation and thus one can make full use of the library routines.

In a one-pass compiler, the preparation and the code generation parts are fused with semantic routines of the semantic analyser. A typical one pass scheme is given in fig. 2.1.

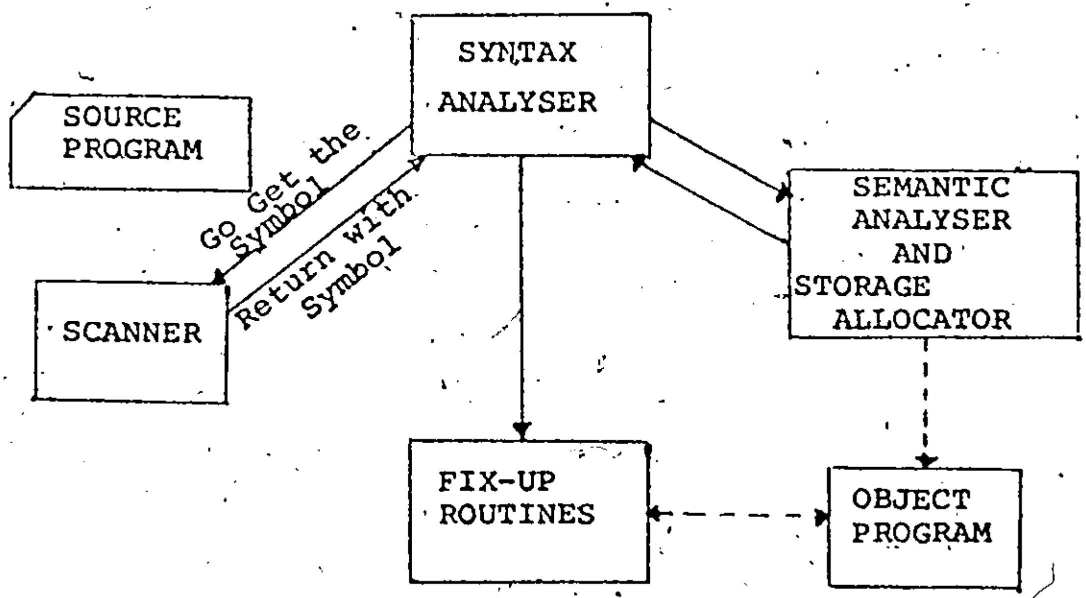


fig. 2.1 A Typical One-Pass Compiler

2.2 Tables of Reserved Words and Symbols of PL/2100

The programming language 2100 has several key words and symbols which are to be known at the compile time. These symbols and words are stored in different tables. The reserved words and symbols are associated with integer tokens which are stored in tables as well. The reserved words of different lengths are stored in different tables. This has been done to reduce search time. In the actual world of programming, the tables are represented by arrays of integer and characters. For details, see Appendix C.

2.3 Context Table

2.3.1 The Description of Objects During Compilation

All identifiers occurring in a program are stored in a table along with a description of the object they name. Since every object is characterized by various attributes with different ranges of values, the record is the appropriate data structure. Since various objects are described by different sets of attributes, the record has a variant part. The table itself is an array of such records. The context table contains description of all named objects. The definition of the table itself is given below; it uses the following data types:

```
TYPE

        AR = ARRAY[1..10] of CHAR;

        SHRINT = -1777B .. + 1777B;

        BITRANGE = 0..16;

        ADDRESS = 0..1777B;

        RG3 = 0..3;

        IDKLASS = (TYPES, KONST, PROC, VARS, FIELD,
                   TAGFIELD, DUMMYCLASS);
```

```
        TYPFORM = (NUMERIC, SYMBOLIC, ARRAYS, RECORDS,
                   FILES, REGISTERS);

        IDKINDS = (ACTUAL, FORMAL);

        OPTPWR = (NOOPT, PUREP, POSP, NEGP);

VAR

  CONTEXTTABLE:  ARRAY[0..250] OF
                 PACKED

        RECORD

        NAME: AR; NXTEL: SHRINT; SYNCELL: BOOLEAN;

          CASE  KLASS: IDKLASS OF

          TYPES : (SIZE: ADDRESS;
                   CASE FORM: TYPFORM OF

                  NUMERIC: (BITS: BITRANGE; MIN, MAX: INTEGER);

                  SYMBOLIC: (FCONST: INTEGER; BITSIZE: BITRANGE);

                  ARRAYS: (AELTYPE, INXTYPE: SHRINT;
                           LO,HI: SHRINT; SZE: BOOLEAN;
                           OPTTYP:OPTPWR;
                           EXP1, EXP2: BITRANGE);

                  RECORDS: (FSTFLD, RECVAR: INTEGER);

          KONST:  (CONTYPE: INTEGER;
                   CASE CONKIND: IDKINDS OF

                   ACTUAL: (SUCC: INTEGER; VALUES: INTEGER);
                   FORMAL: (CADDR: ADDRESS; CLEVEL: RG3) ;

          PROC:   (PROCTYPE, FORMALS: INTEGER;
                   PROCKIND:IDKINDS;
                   PROCADDR: ADDRESS; PROCLEVEL: RG3 ;
                   SEGSIZE: INTEGER);

          VARS:   (VTYPE: INTEGER; VKIND: IDKINDS;
                   SYNPTR: SHRINT;
                   VADDR: ADDRESS; VLEVEL: RG3);

          FIELD:  (FLDTYPE: INTEGER; FLDADDR: ADDRESS;
                   BITDISPL, BITWIDTH: BITRANGE);
```

```
TAGFIELD :  (CASESIZE: INTEGER; VARIANTS: INTEGER;
             CASE TAGVAL: BOOLEAN OF

             FALSE:  (CASETYPE:INTEGER);
             TRUE:   (CASEVAL: INTEGER));

END;       ┌→ END OF THE DEFINITION OF THE CONTEXT
             TABLE ↓
```

Anonymous objects which are generated during compilation and correspond to component variable denotations, primaries, expressions, etc. are described by variable local to the various processing procedures. These are specified as follows:

```
TYPE

    ATTRKIND = (VARBL, SVAL, LVAL, LCOND);

    ATTR = RECORD

               TYPTR: INTEGER;
               CASE KIND: ATTRKIND OF

               VARBL: (ACCESS: (DRCT, INDRCT, INXD);

                      BREG: RG3; DPLMT: INTEGER;

                      CASE  PCKD: BOOLEAN OF

                      FALSE :  ;
                      TRUE:   (BITADR, BITSZ: BITRANGE));

               SVAL:  (VAL: INTEGER);

               LVAL:  (CTERM: INTEGER);

               LCOND: (JMP:  0..3; ARITH:BOOLEAN);

               END;
```

The complete datatype defintions used by the compiler to describe objects are included, here, not only to convey an insight into the compiler organization, but also to demonstrate the power of PL/2100 data definition facilities (similar to PASCAL). They allow for a transparent and fully symbolic, machine independent form of data specifications, but at the same time make an economic usage of storage possible (the packed record has been used for that purpose).

2.3.2 Search Method

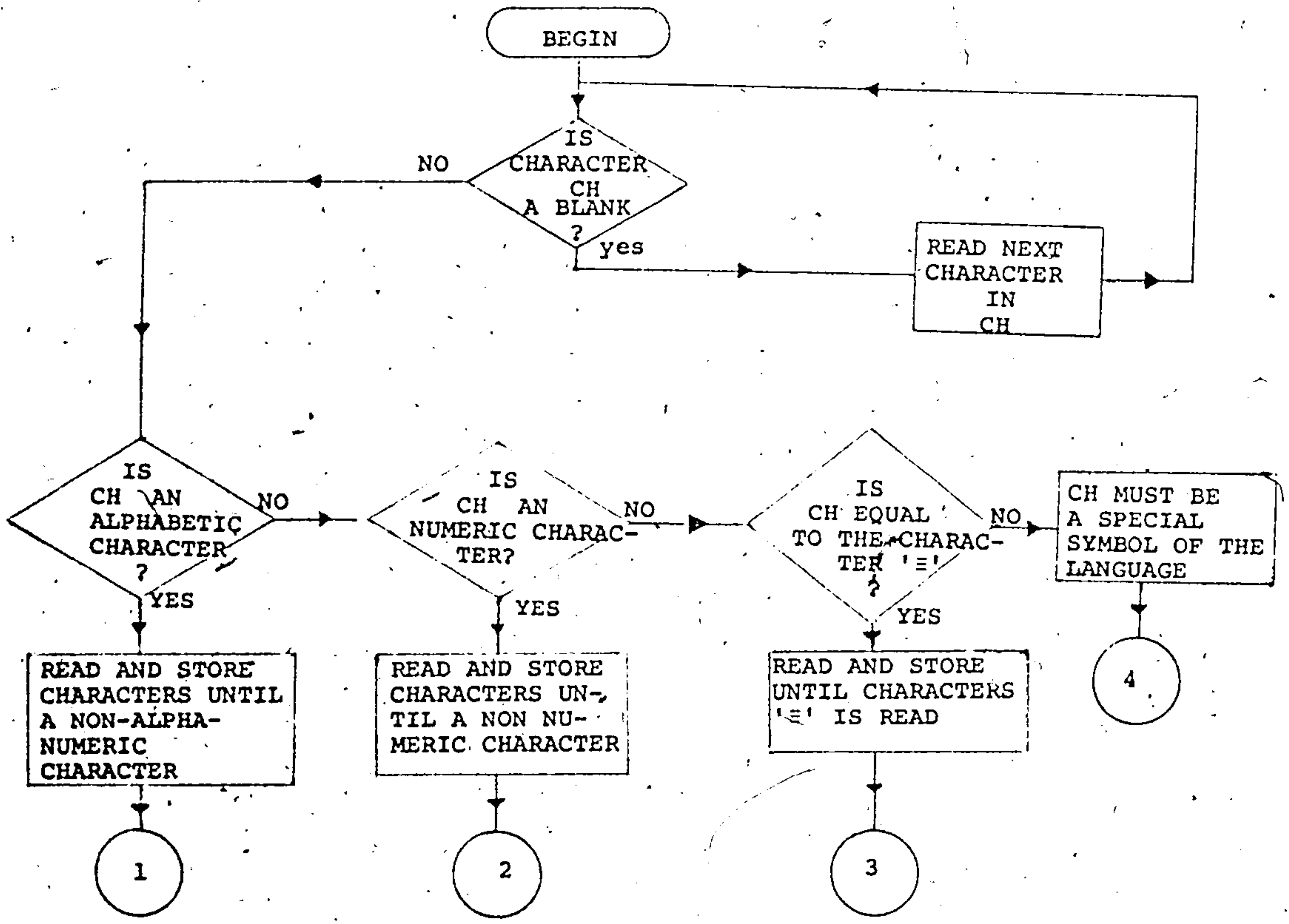
A linear search method has been used to search through the tables of reserve-words and the symbol table. The reason this method has been used, is because of its simplicity. In order to reduce the time for the search, the reserved words of different lengths are stored in different tables along with a table of pointers pointing to the first elements of the tables of reserved words. Thus search need not be made through all the tables at the same time. The identifiers are put into the objectable as they are encountered. A pointer is used, in the description of the identifier, to point to the previously encountered identifier. Identifiers stored are, obviously, all different.

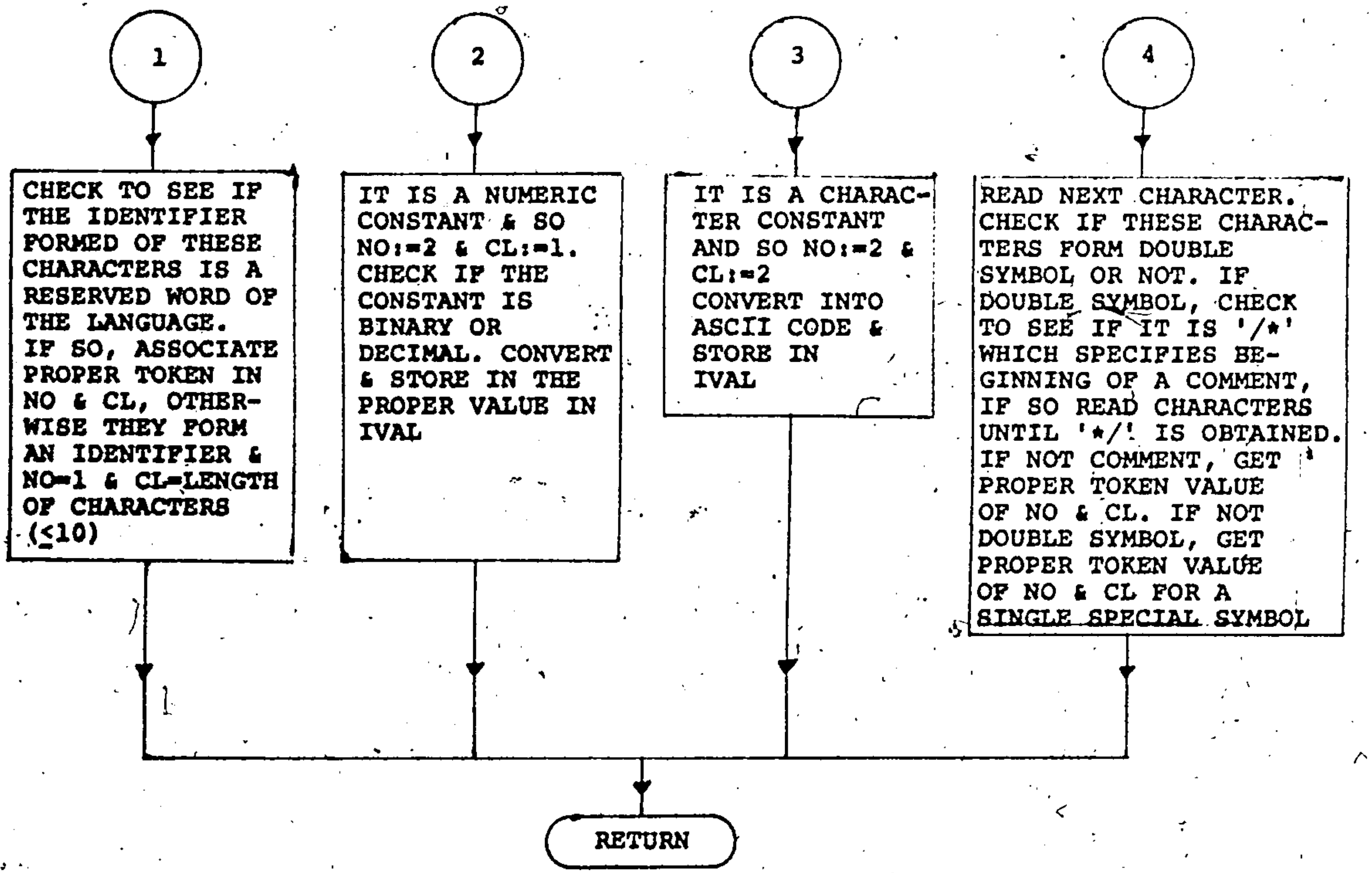
An indication has been given in this paragraph to show why binary search and hash-coded search methods have not been used. The binary search method has the disadvantage that identifier and the reserved words are to be put into alphabetic order. The variant parts of the record describing the objects in the object are of different lengths (in no. of bits) and packing would be lost as the identifiers are moved from one place to another as is necessary in binary search method. This will, naturally, cause a lot of troubles. The hash-coded search seems to be better but is more complicated and as such could be time-consuming.

The linear search method, compared to binary search and hash-coded search methods, seems to be a bit slow but much simpler. In fact, a proper study should be done with the tables of reserved words and the context table used in HPCOM compiler, to see which of these search methods is better and more efficient.

2.4 Lexical Scanner

The simplest part of a compiler is the lexical scanner used to scan the text (or the source program). In PL/2100, the word-delimeters (or reserved words e.g. begin) are represented like identifiers (without escape characters) and must be interpreted by the scanner. Identifier (and also numbers) are therefore, considered as basic symbols. It is a source-oriented scanner and has made full use of recursive definitions of procedures possible in PASCAL (the language in which the compiler has been written). The main controls of the scanner is given in fig. 2.4 and a listing of the program in Appendix D.


BEGIN
IS CHARACTER CH A BLANK ?
yes
NO
READ NEXT CHARACTER IN CH
IS CH AN ALPHABETIC CHARACTER ?
YES
NO
READ AND STORE CHARACTERS UNTIL A NON-ALPHA-NUMERIC CHARACTER
1
IS CH AN NUMERIC CHARACTER?
YES
NO
READ AND STORE CHARACTERS UNTIL A NON NUMERIC CHARACTER
2
IS CH EQUAL TO THE CHARACTER '≡' ?
YES
NO
READ AND STORE UNTIL CHARACTERS '≡' IS READ
3
CH MUST BE A SPECIAL SYMBOL OF THE LANGUAGE
4



<u>fig. 2.4</u> Main control section of lexical analyser

2.5 Syntactic Analysis of the Source Text

The design of the programming language 2100 is based on syntax allowing the application of a reasonably simple and perspicuous analysis technique. The method chosen was first used by Conway(C1) who called it the "SEPARABLE TRANSITION DIAGRAM" technique. This method has also been used by Wirth(W3) in the design of the PASCAL compiler. This method has been chosen for the design of the compiler (HPCOM) for PL/2100.

The syntax of the language is presented as a finite set of pseudo-finite state recognizers. The attribute "pseudo" is due to the fact that some of the basic symbols to be recognized are replaced by sentences recognizable by one of the members of this set. The recognizers may thus activate each other, possibly causing recursion. This top-down parsing technique has the following advantages:

i) Every single recognizer can be presented by a lucid finite graph directly representing the recognizer's program.

ii) If a programming system is available offering recursive procedures, no explicit stack mechanism need be programmed. (It is important to note that, as PASCAL has capability of recursive procedures, this has been done in HPCOM).

iii) Program paths introduced to handle syntactic errors can be represented in the syntax graph.

It is important to note that syntax analyser is top-down.

The syntax analyser has recursive procedures for nonterminal symbols of the language and these procedures parse phrases for the nonterminals. The procedures are told where in the program to begin looking for a phrase for all the nonterminals; hence syntax analyser is <u>goal-oriented</u> or predictive. This method is known as recursive descent method. The syntax analyser uses a <u>bottom-up</u> parsing principle to obtain input (viz integer tokens) via the <u>source oriented</u> lexical scanner (which associates the tokens with the different symbols and identifiers of the language).

The dependence relationships between the various main procedures of the syntax analyser are given in fig. 2.5.

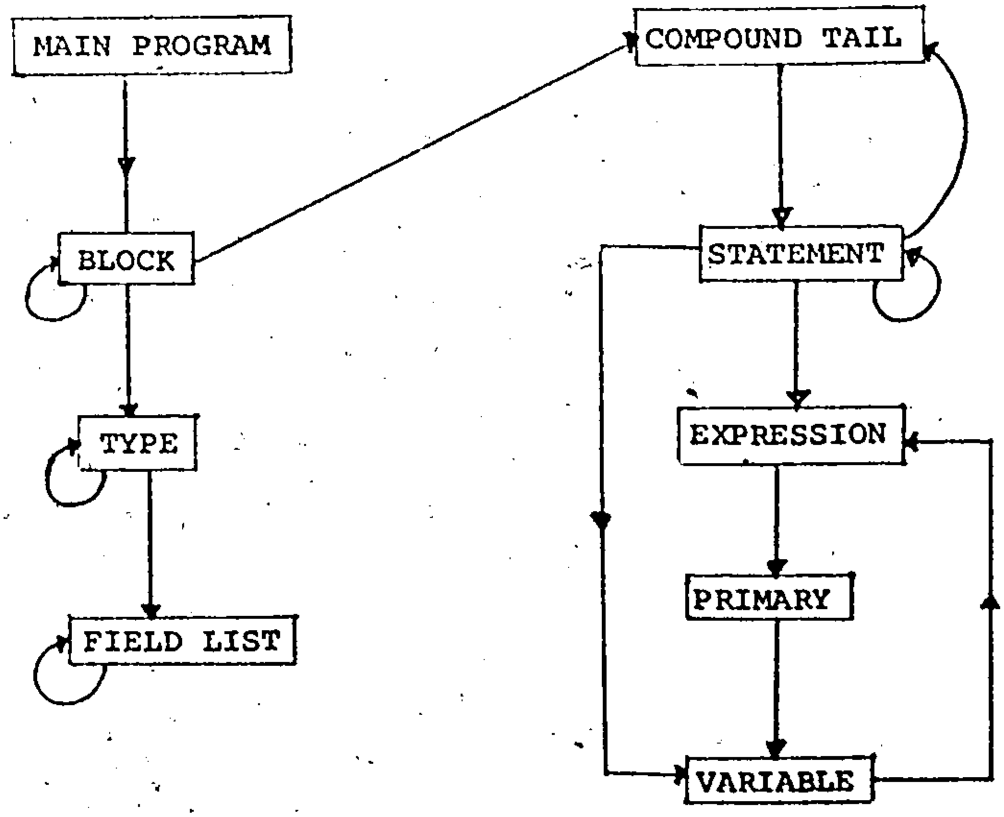


fig. 2.5 presents the block diagram of the syntax analyser

The detailed syntax of PL/2100 in terms of 'transition diagrams' is given in Appendix E.

2.6 Semantic Routines and Code Generators

The advantage of using recursive descent method becomes evident in semantic routines as one can insert code for a phrase anywhere within a procedure, not just at the end of it, when a phrase has been detected. No explicit stack mechanism is necessary to store the parsed phrase. In HPCOM semantic routines and code generators are fused with the syntax analyser which calls them whenever necessary. If coroutines are used, semantic routines and code generators need not be fused with the syntax analyser. The code generators are, of course, dependent on the features of the target machine. Since code generators are very important part of the compiler, they are discussed in detail in the next Chapter III.

2.7 Initialization Routine and Fix-up Routines

The initialization routine is the first routine to be called in the compiler and it initializes different variables and the symbol table.

The fix-up routines are called at the end of the code generations and are used to fix the codes up namely, placing the proper branch addresses, allocating the constants at the end of the data stack and assigning the temporaries used.

2.8 Routines Used to Produce Relocatable Binary in Proper Format

These routines are used to put the code in a form acceptable to the relocatable loader of the HP/2100A machine.

2.9 Overall View of the Different Parts of HPCOM

In this section, the following block diagram has displayed the various parts of the compiler as being called from the main procedure.

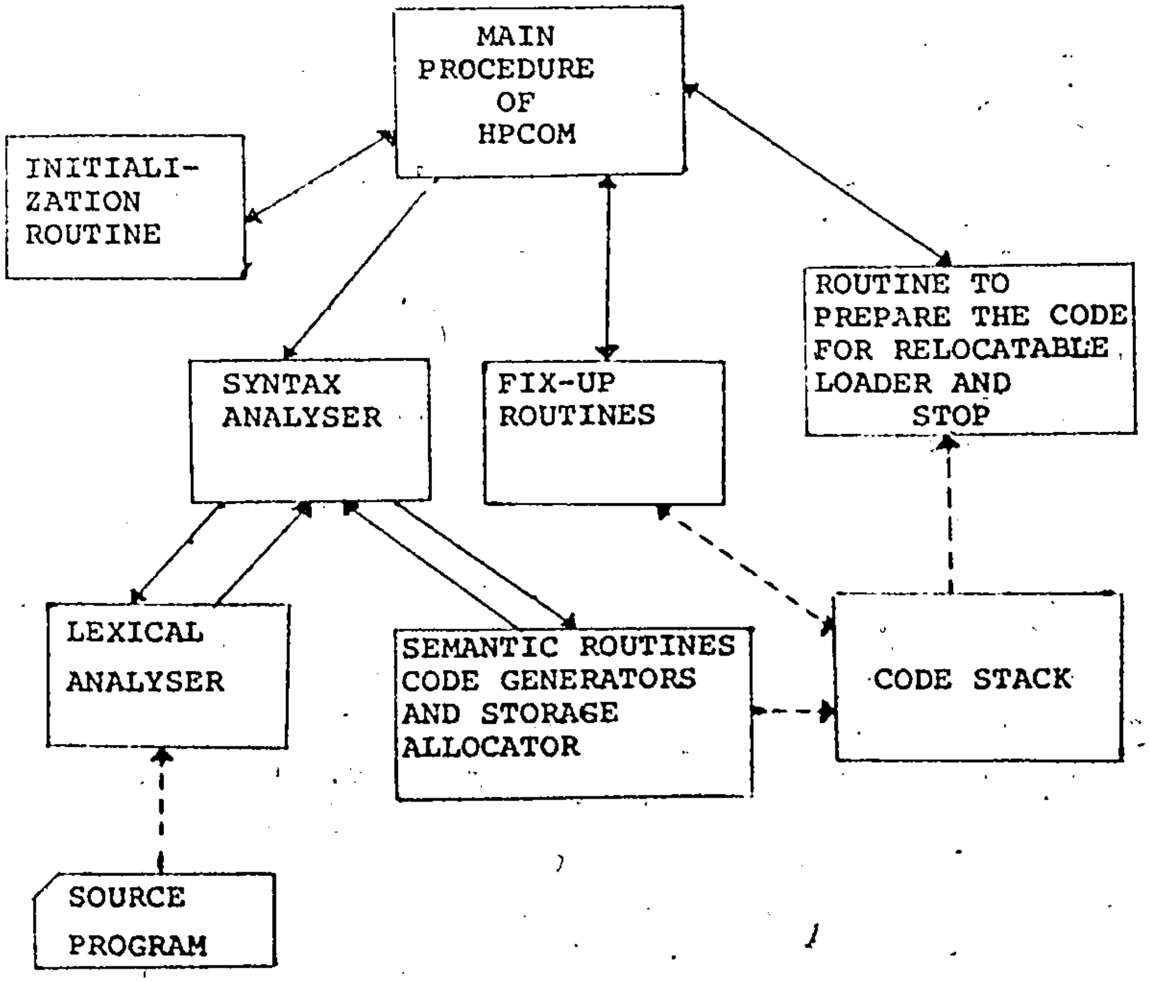


fig. 2.9 **Displays Different Parts of HPCOM Compiler**

2.10 Conclusion

The language PASCAL has been used to write the compiler HPCOM. The reasons are that the PASCAL compiler is fast and that the language PL/2100 closely resembles PASCAL in data structure. Not much effort need be spent to write the compiler in PL/2100, because the features available in both PASCAL and PL/2100 have been used (except the powerset or (set of) type which is not available in PL/2100). This would be advantageous to a person who wants to bootstrap the HPCOM compiler to HP/2100A machine.

The design of the compiler is governed by the fact that the compiler should produce efficient code. Efficiency, in the case of HPCOM, refers primarily to space (required to store the object code in the target machine),not time. This is because the HP/2100A has a small memory and as such a program in PL/2100 should not occupy too much space in the core of the target machine (HP/2100A).

CHAPTER III

CODE GENERATION

3.1 Code Generators

This part of the compiler is machine dependent i.e. their structure and the algorithms used depend on the target machine. The following diagram (fig. 3.1) gives a view in the organization of the code emitter. Needless to say that these routines check syntax and get the semantics as well. (This is apparent from the discussion in Chapter I).

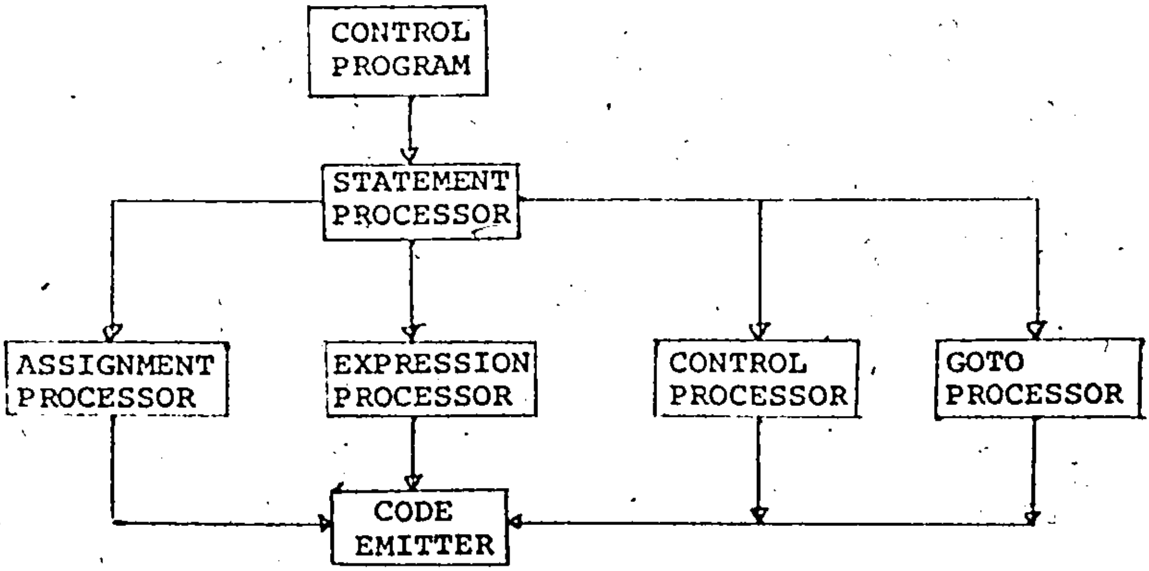


Fig. 3.1 Code Emitter Organization

3.1.1 Control Program

The control program is driven by syntactic productions recognized by the parser. Whenever a syntactic entity is recognized, the control program calls the statement processor which, in turn, calls the module responsible for the code emission of the class of productions to which the recognized production belongs. The compiler has extensive type checking capabilities, and it is the responsibility of the different processors to initiate type checking where necessary.

3.1.2 Expression Processor

Whenever an arithmetic expression is recognized, the statement processor activates the expression processor. The function of the arithmetic expression processor is to prepare all the operands of the expression for processing by generating proper codes for their values (at run time) as required. Thus the expression processor calls other modules (not shown in fig. 3.1) to get the address of the operands. Once the operator is recognized it finds out if more operands are necessary for the operator. The processor parses from left to right with equal precedence for all the operators. Once a meaningful sentence is recognized, it generates codes which yield the value of the expression.

There are only two hardware accumulators namely, A- and B-registers for the HP/2100A machine. Thus it might be necessary to store the partial value of the expression (during runtime) and hence temporary memory locations are necessary to store these values. At compile time, the arithmetic processor generates proper codes and assigns the locations necessary for this purpose. An attempt has been made to optimize the number of temporaries. The maximum number of temporaries necessary for an expression are also the maximum number of temporaries for the whole program, as these temporary locations are made free after an expression is executed (during runtime). The arithmetic processor calls two routines viz. LDTMP (load from the temporary) and STTMP (store into temporary) for this purpose. It also keeps a table of pointers so that fix-up routines can, properly, assign the locations.

3.1.3 Control Section

The control section processor handles the code generation for all the control structures of the programming language 2100. Both selections and exits from a loop are managed by the compile time tables. These tables are updated by the control section processor, and whenever all pertinent information is available, these tables are used to emit fix-ups and branch tables. Both the management and the use of the tables are machine independent.

3.1.4 Assignment Processor and GOTO Processor

Assignment processor is called whenever a value of an expression is to be assigned to a variable.

Whenever an unconditional branching is recognized in a program, the GOTO processor is activated. It updates and keeps track of the branching table (at compile time) so that proper codes are generated. Both forward and backward jumps are allowed.

3.1.5 Object Code Emitter

The object code emitter is responsible for the production of HP/2100A object code. The produced code is placed into a code stack containing all the codes generated for a program. The code stack provides the compiler the capability of peephole optimization but no attempt has been made to optimize the code produced.

3.2 Code Emitted for Different Statements and Types of Operands

In this section the code emitted for expressions, different statements and operands of different types will be discussed.

3.2.1 Expression Evaluation

The format of arithmetic operations on HP/2100A insists that one of the operands of the expression be located in one of the two accumulators viz A- and B-registers. The evaluation of expressions often results in values which must be stored temporarily. These values are stored in temporary memory locations, not in A- and B-registers as these registers could be used in evaluating the expressions. A great deal of effort has been spent on HPCOM to make it allocate temporary locations efficiently. These are not allocated dynamically at run time.

To evaluate an arithmetic expression, the first operand is loaded in the A-register (by convention, the first operand is always loaded in the A-register and also the result of the expression is always in the A-register) and the HP arithmetic operations (viz. MPY, ADA, etc.) are used to evaluate the expression. The code generated for different arithmetic operations, is given in terms of HP assembly language instructions (whose uses and meanings are described in Appendix B).

3.2.1.1 Codes Generated for Arithmetic Operators

3.2.1.1.1 Addition

e.g 1st operand + 2nd operand

```
LDA   1st operand /* Load first operand in the
                     A-register */

ADA   2nd operand /* Add 2nd operand to the
                     contents of A-register */
```

3.2.1.1.2 Subtraction

e.g. 1st operand - 2nd operand

```
LDA   1st operand /* Load first operand in
                     A-register */

LDB   2nd operand /* Load second operand in
                     B-register */

CMB,INB /* Take 2's complement of the 2nd operand */

ADA 1  /* Add contents of B-register to the contents
          of A-register, the result of subtraction is
          in A-register */
```

3.2.1.1.3 Multiplication

e.g. 1st operand * 2nd operand

```
LDA  1st operand /* load 1st operand in A-register */

MPY  2nd operand /* Multiply contents of A register
                    with the 2nd operand */

IOR 1    /* Inclusive or contents of B-register
            with the contents of A-register */
```

It is important to note that the last code viz IOR 1 is important, as the sign bit of the result of multiplication is in B-register. It is implicitly assumed that the result of multiplication is less than $(2^{15}-1)$ i.e. the 15th bit of A-register is always zero, and also that bits 0 to 14 of of B-register are zero.

3.2.1.1.4 Division

e.g. 1st operand div 2nd operand

```
LDA  1st operand /* load 1st operand in the A-register */
CLB
SSA               /* skip next instruction if the sign
                     bit of A-register is zero i.e.
                     the 1st operand is positive */


CMB,INB          /* 1st operand is negative, take 2s
                    complement of it and store in B-
                    and A-registers combined */

DIV  2nd operand /* Divide the contents of B- and A-
                    register by the 2nd operand. The
                    result is in A-register */
```

3.2.1.2 Codes for Relational And Logical Operators

3.2.1.2.1 OR (V)

e.g. 1st operand $\{{}^{V}_{OR}\}$ 2nd operand

```
LDA  1st operand /* Load 1st operand in the A-register */

IOR  2nd operand /* Inclusive or 2nd operand to the
                    contents of A-register. The result
                    is in A-register */
```

3.2.1.2.2 AND (Λ)

e.g. 1st operand $\{{}^{\Lambda}_{AND}\}$ 2nd operand

```
LDA  1st operand /* Load 1st operand in the A-register */

AND  2nd operand /* And 2nd operand to the contents of
                    the A-register. The result is in
                    A-register */
```

3.2.1.2.3 Relational operators

e.g. 1st operand $\begin{Bmatrix} LT \\ LE \\ GT \\ GE \\ NE \\ EQ \end{Bmatrix}$ 2nd operand

In cases of the relational operators, first operand is subtracted from the 2nd operand (for convenience) and the result (in the A-register) of the subtraction is checked (a check on the sign bit of the A-register and/or the contents of the A-register is necessary) to see if the relation is true or false, and the result is set to true (1 in the A-register) or false (0 in the A-register).

The following three words of instruction are common to all relational operations:

```
LDA  1st operand /* Load the 1st operand in the A-register */
CMA,INA  /* Take 2s  complement of the 1st operand */
ADA  2nd operand /*  Add 2nd operand to the contents of
                     the A-register. The result of the
                     subtraction is in the A-register */
```

3.2.1.2.3.1 LT(<)

e.g. 1st operand $\{^{<}_{LT}\}$ 2nd operand

```
SSA,RSS /* skip the next instruction, if the result of
           the subtraction is not positive */

SZA,RSS /* skip the next instruction, if the result of
           the subtraction is not zero */
```

3.2.1.2.3.2 LE($\leq$)

e.g. 1st operand $\{{}^{\leq}_{LE}\}$ 2nd operand

```
SSA,SZA /* skip the next instruction, if the result of the
           subtraction is not positive */
```

3.2.1.2.3.3 GE($\geq$)

e.g. 1st operand $\{{}^{\geq}_{GE}\}$ 2nd operand.

```
SSA,SZA /* skip the next instruction, if the result of the
           subtraction is positive or zero */
```

3.2.1.2.3.4 GT(>)

e.g. 1st operand $\{{}^{>}_{GT}\}$ 2nd operand

```
SSA,RSS /* skip the next instruction, if the result of the
           subtraction is not positive */
```

3.2.1.2.3.5 NE($\neq$)

e.g. 1st operand $\{{}^{\neq}_{NE}\}$ 2nd operand

```
SZA,RSS /* skip the next instruction if the result of the
           subtraction is not zero */
```

3.2.1.2.3.6 EQ(=)

e.g. 1st operand $\{{}^{=}_{EQ}\}$ 2nd operand

```
SZA /* skip the next instruction, if the result of the
       subtraction is zero */
```

Next two words of code are common to all relational operations:

```
CLA,RSS /* set A register to zero and skip the next instruction.
           That is to say that the result of relational
           operation is false */

CLA,INA /* set A register to 1, The result of relational
           operation is true */
```

3.2.1.3 Bit Operators

3.2.1.3.1 IOR

e.g. 1st operand ior 2nd operand

```
LDA 1st operand /* load 1st operand in the A-register */
```

IOR 2^{nd} operand /* inclusive OR 2^{nd} operand to the contents of the A-register. The result is in the A-register */

3.2.1.3.2 XOR

e.g. 1^{st} operand xor 2^{nd} operand

LDA 1^{st} operand /* Load 1^{st} operand in the A-register */

XOR 2^{nd} operand /* Exclusive OR 2^{nd} operand to the contents of the A-register. The result is in the B-register */

3.2.1.4 Shift Operators

e.g.

$$\text{operand}\left\{\begin{array}{c}\underline{\text{als}}\\ \underline{\text{ars}}\\ \underline{\text{ral}}\\ \underline{\text{rar}}\\ \underline{\text{alf}}\end{array}\right\}\langle\text{factor}\rangle$$

for the A-register, where <factor>, an integer, species the number of shifts to be made.

Similarly for the B-register, we have

$$\text{operand}\left\{\begin{array}{c}\underline{\text{bls}}\\ \underline{\text{brs}}\\ \underline{\text{rbl}}\\ \underline{\text{rbr}}\\ \underline{\text{blf}}\end{array}\right\}\langle\text{factor}\rangle$$

For the operator alf (or blf), the factor is determined as a modulo 4 (because 4 alf or blf) shifts are the same as no shift).

For other operators, the factor is determined as modulo 16. Following examples will make it clear.

3.2.1.4.1

(a) operand alf 6

At compile time, factor is determined as a modulus of 4 and the result is 2. The code generated is.

LDA operand /* load operand in the A-register */

ALF,ALF /* make alf shift twice */

(b) operand alf 7

Codes generated are

```
LDA operand

ALF,ALF

ALF
```

3.2.1.4.2

(a) operand rar 7

Codes generated are

```
LDA operand

RAR,RAR

RAR,RAR     /* code has been optimized
               in these cases */
RAR,RAR

RAR
```

(b) operand rar 19

The resultant factor is 3 and codes generated are

```
LDA operand

RAR,RAR     /* 3 RAR shift necessary */

RAR
```

3.2.1.4.3

(a) operand als 18

Codes generated are

```
LDA  operand

ALS,ALS  /* factor = 18 mod 16 = 2 */
```

(b) operand als 5

Codes generated are

```
     LDA operand
     ALS,ALS              /* two ALS shift allowed per
     ALS,ALS                 instruction word */
     ALS
```

3.2.1.4.4

(a) operand ars 17

Codes generated are

```
     LDA operand
     ARS                  /* one ARS shift necessary */
```

(b) operand ars 6

Codes generated are

```
     LDA operand
     ARS,ARS              /* operand is shifted
     ARS,ARS                 by six places */
     ARS,ARS
```

3.2.1.4.5

(a) operand ral 36

Codes generated are

```
     LDA operand
     ALF       /* resultant factor is 4 and 4 RAL
                  shifts are equal to one ALF shift */
```

(b) operand ral 3

Codes generated are

```
LDA operand
RAL,RAL   /* 3 shifts necessary */
RAL
```

3.2.1.4.6

Code generated for the shift operators for the B-register is similar except the the result is in the B-register.

for example

operand <shift operator> factor

Since the value of an operand is always in the A-register, we generate codes as follows,

```
LDA operand /* load operand in the A-register */
STA 1 /* store the value of the operand from the
              A-register in the absolute location 1 which
              is the B-register */
```

This is followed by similar codes for shift operators for the B-register. The result is in the B-register.

3.2.2 Assignment statement

e.g. <variable>: =<expression>

The result of expression is always in A register. Hence code generated is,

```
STA variable ad  /* store the contents of A-register
                    in location for the variable */
```

3.2.3 IF statement

IF {Boolean expression} THEN {statement} ELSE {statement}

The result of the necessarily boolean expression is either false (value 0 in A) or true (value 1 in A). Hence codes generated are:

for IF {expression} THEN {statement};

```
   SZA,RSS  /* skip next instruction,if the result of boolean
               expression is true (value 1) */

   JMP LAB1    /* jump to location LAB1 if the result
      .        of boolean expression is false (value 0) */.
      .
      .     }  /* code for the statement after THEN */
      .
LAB1 NOP       /* A no operation instruction */
 .
 .
```

Codes generated for

IF {expression} THEN {statement} ELSE {statement}

```
   SZA,RSS  /* if result of expression is true, we go to
               THEN part */

   JMP,LAB1 /* jump to else part of if statement */
      .
      .     /* code for statement for THEN part */
      .
   JMP,LAB2 /* jump around else part of the statement */

LAB1 NOP    /* else part starts here */
 .
 .          /* code generated for statement in else part */
 .
LAB2 NOP    /* A dummy no operation instruction used to know
               during compile time, where to jump around
               else part */
```

3.2.4 REPEAT statement

```
       e.g.  REPEAT

                        Statement(s)

             UNTIL (expression is true)

   LAB1 NOP             /* GENERATES DUMMY NO OP INSTRUCTION */
     .
     .                  /* code generated for statement(s) */
     .

     -
     -                  /* code generated for boolean expression
     -                     Result 'true' (value 1) or 'false'
     -                     (value 0) is in A */

        SZA,RSS     /* skip next instruction, if result of
                       boolean expression is true */

        JMP LAB1    /* result of expression is false, repeat the
                       REPEAT loop */

        NOP         /* Another dummy no operation instruction */
```

3.2.5 WHILE statement

```
        e.g.  WHILE <expression> DO <statement>

   LAB1 NOP         /* start of WHILE loop */
     .
     .              /* code for expression, result true or false
     .                 is in A-register */

        SZA,RSS     /* skip next instruction if the result of
                       expression is true */

        JMP LAB2    /* jump out of the while loop, the result
                       of expression being false */

              .     /* code for statement */
              .
              .

        JMP  LAB1   /* GO REPEAT WHILE LOOP */

   LAB2 NOP         /* END OF WHILE LOOP */
```

3.2.6 GOTO statement:

```
e.g. GOTO <integer>

JMP LABEL  /* JMP TO THE LABELLED LOCATION */
```

3.2.7 FOR statement

```
                                   e1                      e2
                                   ↓                       ↓
FOR <identifier> : = <expression>{   TO   }<expression>
 :                                 DOWNTO
 :
 code for 1st expression    DO   statement

STA <identifier> /* STORE the value of the first
                    expression in the location of
                    the identifier */

/* code for 2nd expression */

STA <TEMP> /* store the result of 2nd expression in
              a known location */

     LAB1 ( LDA <identifier> /* load value of identifier in A */
          |
          | CMA,INA          /* take 2s complement */
          |
{TO}      < ADA <TEMP>       /* calculate e2-el */
          |
          | SSA,SZA          /* if e2-el is -ve,for loop is
          |                     finished
          |
          | JMP LAB2         /* jump out of for loop */
          |   :
          (   :              /* statement codes */
              :
              :
            LDA <identifier> /* load value of identifier in
                                the A-register

            INA              /* INCREMENT IT */

            STA <identifier> /* store the new value of el in
                                identifier */

            JMP LAB1         /* GO    BACK TO CHECK FOR-LOOP */

     LAB2   NOP              /* dummy no operation to go out
                                of for loop */
```

For 'DOWNTO' the relation e1 ≥ e2 must hold for the loop to continue. Hence the difference in coding would be

```
LAB1  LDA <TEMP>  /* load e2 */
      CMA,INA     /* complement e2 */
      ADA <identifier> /* calculate e1-e2 */
```

3.3 Data representation

We now discuss the internal representation of the PL/2100 data types, and the way in which operations on them are implemented. One of the deficiencies of the HP/2100A instruction set is the lack of facility whereby structured data types may be easily manipulated. These deficiencies are the following.

1) The HP/2100A instruction set allows the manipulation of only 8 bit bytes (in case of character) and/or maximum 16 bit word with one instruction. The PL/2100 language however allows the manipulation of structured types (arrays, records) which in the majority of cases are greater than 16 bits in length. No problem arises as long as those subfields of the structured types are being handled that fit into a word or less. Restricting the data structures to those whose lengths are at 16 bits would, however, destroy one of the most elegant features of the PL/2100.

2) The HP/2100A does not have convenient instructions for address calculation. An address may be calculated through one of the accumulators at a time. When dealing with data types which involve either implicitly or explicitly a variable offset

from the base address of a variable of the data type (arrays), the use of an index register would be most useful. HP/2100A machine has no hardware index register as such (but indirect addressing is possible through one bit in the instruction word).

3.3.1 Numeric types

Numeric types in HP/2100 implementation are those whose internal representation is a sixteen bit word.

All numeric operations allowed by the PL/2100 system language, except modulo, are implemented. Arithmetic operations are evaluated in A- and B-register and result is always in A-register (unless stored).

3.3.2 Scalar types

The values of the programmer defined scalar types are ordered by the position of the identifier names in the defining list. Internal values are assigned in order starting with zero,and the base 2 logarithms of the largest value determines the number of bits required. The symbolic constants C0, C1, ... Cn of a scalar type, are represented internally as 0,1,2,...n.

3.3.3 Array types

An array is a structure consisting of a fixed number of components all of the same type. The dimensionability of an array is specified in the array declaration.

In PL/2100, the bounds of an array must be constant, and known at compile time. As a result, we can allocate (at compile time) to a variable of the array type the amount of storage required to hold all the elements of the array. The address of an array element consists of the base address of the array offset by the index of the element into the array.

The effective relative address (relative to the top of the data stack) of an array is calculated as follows:

Let us assume that the array is one dimensional, and the lower and upper bounds of the array are denoted by LO and HI respectively. The size of each element of the array is denoted by SIZE (the size is the number of HP words required to hold an array element of a particular type, e.g. if an array element is integer, SIZE = 1). The base address of the array variable is given by, say, DPLMT. The effective address of an array element is calculated according to the following formula,

effective address = DPLMT + (value of index-LO)*SIZE.

For a multidimensional array, the effective address can be calculated similarly. The HPCOM compiler is capable of handling multidimensional arrays.

We have given an example of the code required to access an array element. The declaration for an array variable is given below:

```
VAR
   A: ARRAY [1..100] OF INTEGER;
```

The above declaration reserves storage for 100 members (16-bit integer) array named A and declares the variable index to take a value from 1 to 100.

To access an array element, we emit the following sequence of code:

```
CLB            /* Clear the B-register */

STB <temp>     /* STORE the contents of B-register in a fixed
                  location <temp> known at compile time. This
                  location will contain the address of the
                  array element */

LDA <index>    /* Load the value of the index in the A-register;
                  if index is a constant, it can be calculated
                  at compile time */

MPY <size>     /* <size> is 1,in the particular example given
                  above, and is known at compile time */

ADA <disp>     /* <disp> is the displacement from the base
                  address and is = base address - LO * <size>.
                  This is calculated at compile time */

ADA <temp>     /* contents of location <temp> is added, this
                  allows multidimensional arrays to be compiled
                  as well */

STA <temp>     /* store the effective address of the array
                  element in the known location <temp> */
```

3.3.4 Record types

A record type is a structure consisting of a number of components, possibly of different types. The record definition specifies for each component of the record, called a field, the type of the field, and an identifier which denotes it.

A record type may have a variable format where one of the fields of the record, called the tagfield, indicate the chosen format of record at any time. For example,

```
TYPE
  person = RECORD name, firstname: ARRAY [1..10] OF char;
                  age: integer;
                  married: boolean;
```

```
    CASE s: sex OF
      male: (enlisted, bold:boolean);
      female: (pregnant:boolean;
                    size: ARRAY [1..3] OF integer);
  END
```

Since PL/2100 does not allow the definition of data types whose length may change at run time, we know at compile time the displacement of all subfields from the start of the record. As a result, no runtime index calculation from the start address is necessary. The address of the field of the record is the base address of the record variable, plus the displacement of the field from the start of the record. The maximum amount of storage allocated to a variable is the sum of the fixed part of the record and the largest variant.

3.4 Synonym Declaration

Through the synonym declaration, more than one simple variable (i.e. not of structured type) can occupy the same location at the run time.

In this section we present how these simple 'synonymous' variables are allocated to the same location. A pointer is used to link the synonymous variables circularly.

For example, if the synonym declaration is, namely,

```
SYN
        A = B,C,D;
```

the identifiers A,B.C.D are connected as given in fig. 3.4.1.

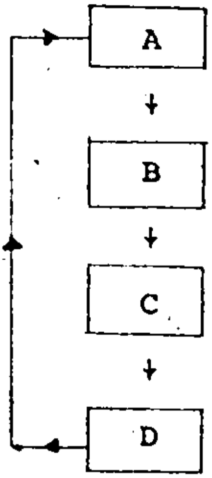


fig. 3.4.1 circular linkage of synonymous variables

If the above declaration is followed by,

B = E,F,G;

the linkage is changed as given in fig. 3.4.2

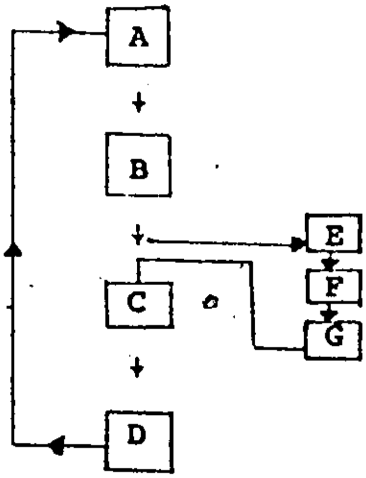


fig. 3.4.2 The changed circuilar linkage of synonymous variables

A boolean pointer is associated with each of these synonymous identifiers, and is false and remains false until the identifiers are allocated to a location.

All the synonymous identifiers are allocated to an address (relative to the top of the data stack) of the synonymous variable encountered first in the variable declaration part.

For example, if in the variable declaration part, we have

C : integer;

and its relative address is 3, then all the synonymous variables declared above will be given the address 3 and the boolean pointers will be set to true. These synonymous variables will be of type integer.

An attempt to allocate space to any of these synonymous variables (which have been allocated to a location already) will cause a compile time error.

3.5 Procedure

Though HPCOM compiler at the present stage is not capable of compiling procedures, an indication of how it could be done is given in this section.

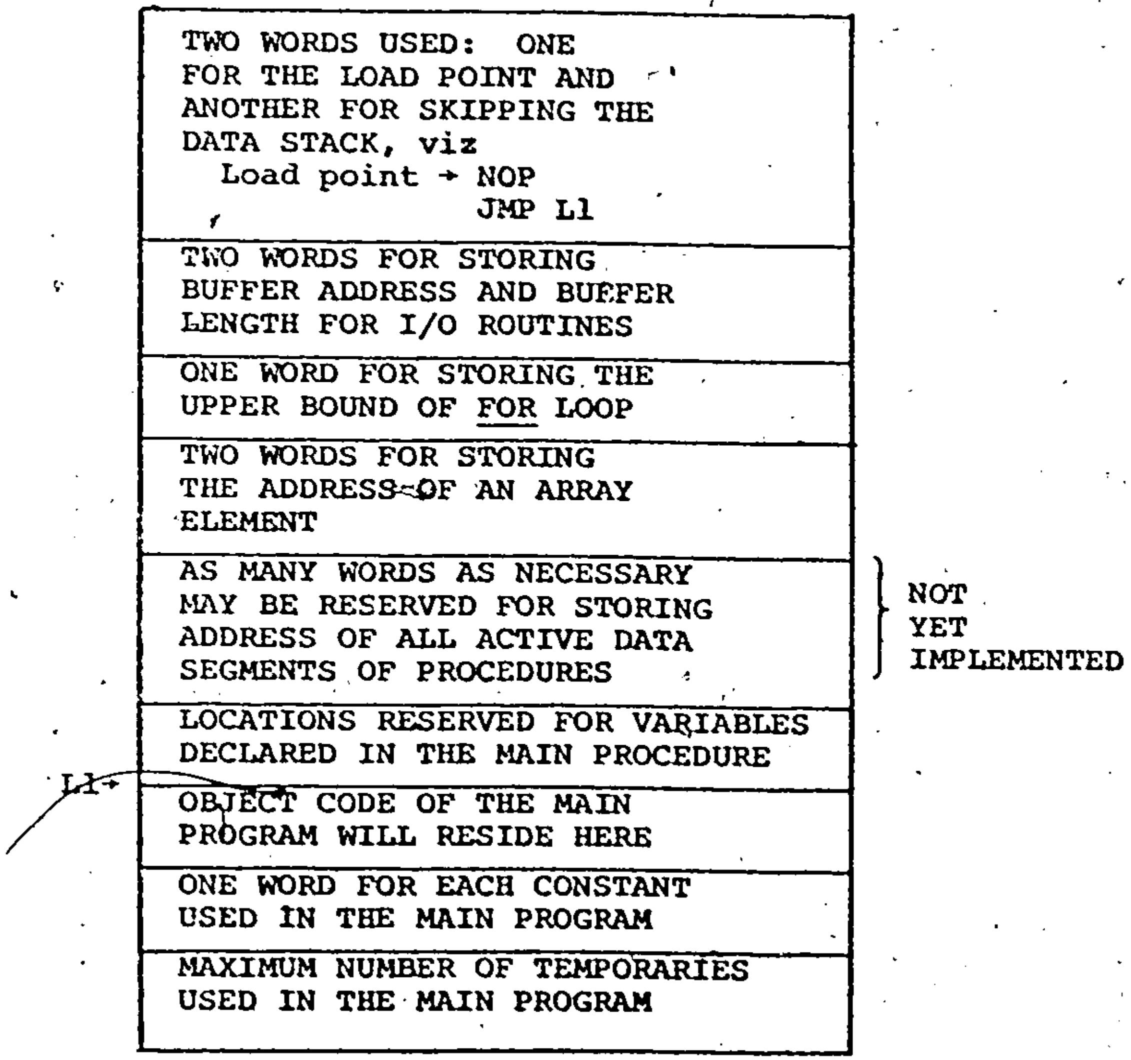
In the case of the PASCAL compiler, every procedure has associated with it a data segment consisting of a header and the local data space of the procedure. The data segments are linked by two chains namely, static and dynamic links, respectively(W3).

The PASCAL implementation report(W3) suggests that the base addresses of all active data segments (those that may be accessed from the presently executing procedure) be stored in a display. The display would then be contained in hardware register for quick access. But HP/2100A machine has only two hardware registers viz A- and B-registers and this method is not suitable. Another method used in the implementation of ALGOL for the Burroughs 5500 was considered. This method has also been used in the implementation of the SUE compiler (K1). In this method two pointers are stored in two registers; these two pointers point to data segments of global and the most recently activated procedures. The static link is stored in another register. Again this method would be unacceptable as the HP/2100A machine has two registers (viz A- and B-registers) which are used, mostly, for arithmetic calculations.

The suggestion put forward here, is to use complete display method as has been implemented in PASCAL compiler (for simplicity and convenience) and to store the information at the top locations of the data stack of the main procedures. These locations are known at compile time and can be set aside for this purpose only. These locations will act as a set of working registers. The disadvantage of the method is that it requires memory access quite often whereas the advantage is that a compiler-writer can have as many nesting of procedures as he likes, by allocating sufficient number of working registers.

3.6 Arrangement of the Code Stack for the Main Procedure

In this section, the following block diagram presents the arrangement of the codes as generated (along with the locations kept aside for variables, constants used in the program and temporaries used for expressions).



TWO WORDS USED: ONE FOR THE LOAD POINT AND ANOTHER FOR SKIPPING THE DATA STACK, viz

```
  Load point → NOP
               JMP L1
```

TWO WORDS FOR STORING BUFFER ADDRESS AND BUFFER LENGTH FOR I/O ROUTINES

ONE WORD FOR STORING THE UPPER BOUND OF FOR LOOP

TWO WORDS FOR STORING THE ADDRESS OF AN ARRAY ELEMENT

AS MANY WORDS AS NECESSARY MAY BE RESERVED FOR STORING ADDRESS OF ALL ACTIVE DATA SEGMENTS OF PROCEDURES

LOCATIONS RESERVED FOR VARIABLES DECLARED IN THE MAIN PROCEDURE

OBJECT CODE OF THE MAIN PROGRAM WILL RESIDE HERE

ONE WORD FOR EACH CONSTANT USED IN THE MAIN PROGRAM

MAXIMUM NUMBER OF TEMPORARIES USED IN THE MAIN PROGRAM

fig. 3.6 displays code stack of a PL/2100 main procedure

CHAPTER IV

I/O IN HPCOM, INTERFACE WITH OPERATING SYSTEM AND SEGMENTATION

4.1 Input/Output (I/O)

In order for a programmer to communicate with the computer, the computer is, normally, provided with external INPUT/OUTPUT devices. The HP/2100A machine is provided with several of these devices. These are as follows:

device	function	logical unit numbers
Teleprinter	Input/output	1
Teletype	Input/output	7
Line Printer	Output	8
Card Reader	Input	5

The logical unit number is associated with each device and this number distinguishes one device from the other, so that the machine knows which device it should read from or write onto.

The HP/2100A machine, at the present installation, is provided with the Disc Operating system (DOSM) which can perform input and output operations using EXEC calls.

4.1.1 Read/Write Calling Sequence in HP Assembler Language

A typical calling sequence to transfer information to or from an external I/0 device is given below in HP/2100A assembler language.

```
                              :
                              EXT EXEC
                              :
                              JSB EXEC  (Transfer control to DOS-M)
DEF RCODE (REQUEST code) }    DEF *+5   (Point of return from DOSM)
DEF CONWD (Control       }
           Information)  }    DEF BUFFER (Buffer Location)
                              DEF BUFL  (Buffer Length)
                              :

RCODE       DEC     1 (or 2)       (1 = READ, 2 = WRITE)
CONWD       OCT     conwd          (described later)
BUFFER      BSS     n              (Buffer of n words)
BUFFL       DEC     n (or -2n)     (same n; words (+) or character (-))
```

4.1.1.1 CONWD

The conwd, required in the calling sequence, contains the following fields:

0 0	W		K	V	M	LOCAL UNIT #
15 14	13	12 11 10 9	8	7	6	5 4 3 2 1 0

FIELD	FUNCTION
W	If 1, tells DOS-M to return to the calling program after starting the I/0 transfer. If W = 0, DOS-M waits until the transfer is complete before returning.
K	Used with keyboard input, specifies printing the input as received if K = 1. If K = 0, "no printing" is specified.
V	Used when reading variable length records from punched tape devices in binary format (M=1, below). If V = 0 the record length is determined by the word count in the first non-zero character which is read in.

M Determines the mode of data transfer. If M = 0 transfer is in ASCII character format, and if M = 1, binary format.

4.1.1.2 'Conwd' used in the HPCOM compiler

For all the I/0 devices,

W = 0 , i.e. DOS-M waits until the transfer is complete. The reason is that the execution of the latter (often a read statement, for example) part of the program may depend on the data received from the input.

K = 0 or 1 i.e K could be specified 1 if printing the input is required (in keyboard mode only).

V = 0 i.e. record length is determined by buffer length. In the HPCOM, a variable buffer length has been used as one does not need to print the whole buffer every time. At compile time the maximum buffer length is fixed to 72 HP words.

M = 0 i.e. All transfers are made in ASCll character mode.

The following table displays the conwd used for different I/0 devices.

Table 4.1.1.2 displays 'conwd' for different I/0 devices

Device	conwd INPUT printing of the input	conwd INPUT no printing of the input	OUTPUT
Teleprinter (oscilloscope)	401 (octal)	1	1
Teletype	407 (octal)	7	7
Card Reader	-	5	-
Line Printer	-	-	8

4.1.2 I/0 Routines

Two I/0 routines (one is GET which gets input from a particular device and the other is PUT which puts output on to a particular device) are written in the HP assembly language. These routines read input into the buffer or write the buffer on to a output device

Since the programs obtained from the compiler (HPCOM) would be working under the Disc Operating Systems (DOSM) of the HP/2100A, the routines GET and PUT are written keeping this in mind. The GET and PUT routines use the EXEC calling sequence for input and output.

Different information necessary for input and output are described below.

4.1.2.1 Buffer

The buffer for input/output in a program in PL/2100 is allocated by the compiler HPCOM.

4.1.2.1.1 Buffer Address

The buffer address is known at the compile time and is stored at the top of the data stack. This address is passed through the B-register into the GET and PUT routines and is immediately stored on entering the GET or PUT routine.

4.1.2.1.2 Buffer Length

The maximum buffer length (as specified by the compiler) is 72 HP words (i.e. 144 ASCII characters because two ASCII characters may be stored in a single HP word).

For input the buffer length is fixed and is 72. The DOSM returns from the input mode when line feed for a particular device is read.

For output the buffer length may be variable (but ≤ 72) and the DOSM returns from the output mode after the buffer of a particular length has been written onto a particular device specified in the program.

4.1.2.2 Logical Unit Number

The logical unit number of a particular device can be provided in a PL/2100 program. This has been described in the section 4.1.1.2.

4.1.3 Routines to Convert ASCII Characters to Integer and Vice Versa

Since input/output from or onto a particular device is in ASCII character mode two routines ALLOC and ALLOK have been used to convert ASCII character to integer and vice versa.

4.1.3.1 ALLOC

This routine is associated with the output routine PUT. This routine gets the value of a variable through the A-register. A pointer to the most recent vacant place in the buffer (initially the pointer points to the top of the buffer) is passed through the B-register. The routine ALLOC converts the integer into proper ASCII character code and allocates it to the buffer. A pointer pointing to the vacant place in the buffer is passed back to the main routine through the

B-register and is stored at a reserved place in the data stack and can be used to allocate the value of the next variable (in the same WRITE statement) in the buffer. Also is passed the length of the buffer for a particular WRITE statement. This information can be used by PUT routine.

After an output operation, the buffer is made free so that it can be used for further output or input.

The figure below gives an indication of the use of buffer pointer. For example, let us assume that we want to output two integer variables A and B.

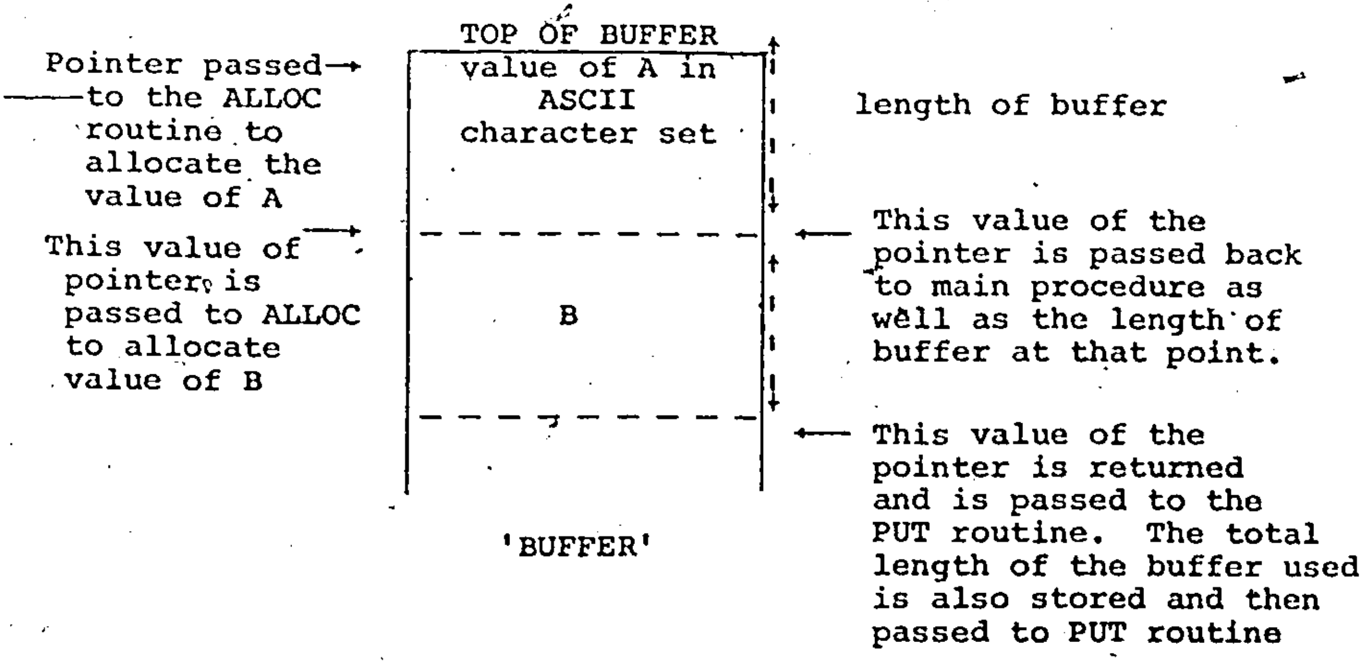


fig. 4.1.3.1 displays the use of buffer pointer.

4.1.3.2 ALLOK

The routine ALLOK works along with the input routine GET. It converts the ASCII character input to integer value and returns the value in the A-register to be stored in a variable location. A pointer to the buffer is passed to the routine through the B-register and is used to get different integers.

4.1.4 I/0 FORMAT

Only free format is used in both input and output. The free format is indicated by an asterisk (*) in the READ or WRITE statement.

The input data are all separated by commas (,) and the comma is used to differentiate between one data to another. Any number of blanks could be used between the data.

The output data are all written out followed by one or two blanks. Only free format is used.

The following examples will make these clear:

i). Input

If we want to read two numbers 4 and 45, the input should be

$$\underbrace{\text{ƀƀ...ƀ}}\ 4\ \underbrace{\text{ƀƀƀ..ƀ}},\underbrace{\text{ƀƀƀ...ƀ}}\ 45\ \underbrace{\text{ƀ...ƀ}},\underbrace{\text{ƀ...ƀ}}\ \ \underline{\text{line feed}}$$

The blanks (ƀ) are all optional

ii) Output

If we want to write two numbers 4 and 45, the output would look like

ƀƀ4ƀ45ƀƀ

These blanks (ƀ) would be provided by the routine ALLOC.

4.1.5 Code Generated for Read and Write Statement

In this section we describe the code generated by the compiler for READ and WRITE statements

4.1.5.1 Read statement

e.g. READ (401B,*,A,B);

where

i) 401B specifies 'conwd' i.e. it means that the input is printed out as received from the device with logical unit number 1.

ii) * specifies free format

iii) A,B are two integer variables.

Codes generated are:

```
LDA buffer address /* Load into A the buffer address
                      stored in a fixed location known
                      at the compile time */

LDB conwd          /* Load into B the value of 'conwd' */

JSB GET            /* JUMP to the routine GET, to get the
                      input */

LDB buffer pointer /* initially it points to the top of
                      the buffer */

JSB ALLOK          /* GO and GET the integer A from
                      ALLOK, integer is in the A-register;
                      the new buffer pointer in the B
                      register */

STA A              /* store the integer in the variable
                      location A */

STB buffer pointer /* store the new buffer pointer */ } actually
                                                         not
LDB buffer pointer /* load the new buffer pointer */  } necessary

JSB ALLOK          /* Go and get the integer B in the register
                      A, the buffer pointer is in the B-
                      register */

STA B              /* store the integer in the variable
                      location B */
```

4.1.5.2 WRITE statement

e.g. WRITE (7,*,A,B);

where

i) 7 is the 'conwd' i.e. write the values of the variables A and B on the device with the logical unit number 7

ii) * specifies free format

iii) A and B are two integers to be written out

Codes generated are:

```
          :
          CLA                 /* The content of the A-register is
                                 zero */

          STA buffer length   /* Initial buffer length is zero */

          LDB buffer address  /* Load into B the buffer address;
                                 initially it is the top of the
                                 buffer */

The value of an  { LDA A      /* Load into the A-register, the
expression may   {               value of the variable A */
be calculated    {
here; leaving    {
the result in the{
A-register
           JSB ALLOC             /* Returns with the pointer to the
                                 buffer vacant place in the B-
                                 register and the new buffer length
                                 in the B-register */

          ADA buffer length   /* get new buffer length */

          STA buffer length   /* store the new buffer length */

          LDA B               /* load the variable B into the
                                 A-register */

          JSB ALLOC           /* put B in the buffer */

          ADA buffer length

          STA buffer length
```

```
LDB buffer address   /* load buffer address in the
                        B-register, the A-register
                        contains the buffer length */

JSB  PUT             /* write the buffer;

RSS                     skip next instruction word

DEC 7                   on to the device with logical
 :                      unit no. 7 */
 :
```

It is important to note that the value of the variable is in A-register, the reason is that by convention the value of an expression is in the A-register and so the value of the expression can be passed to the ALLOC routine without the use of any other instruction. As both A and B are used to pass buffer address and buffer length, the logical unit number (or better conwd) has been passed to the routine PUT as shown above. Any arithmetic expression and/or constant (both integer and character constants) can be written out.

There is no check on the logical unit number and so the programmer would have to know which logical number to use.

4.2 Living with the Operating System

The requirements of a PL/2100 program and HPCOM in particular for interactions with their environment (the operating system) take several forms.

4.2.1 Interaction of HPCOM with its environment

The compiler HPCOM runs on the CDC-6400 machine under the SCOPE operating system. Since HPCOM is written in PASCAL, the PASCAL compiler residing on a file is called to compile and load the program HPCOM. Only thing the operating system SCOPE has to do is to attach the file containing the PASCAL compiler and then the PASCAL compiler compiles HPCOM and loads the compiled HPCOM onto the memory. Then CDC-6400 machine executes the HPCOM program (compiler). The input to the HPCOM compiler is a program in PL/2100 and the output, the relocatable binary format (suitable for the standard relocatable loader of the HP/2100A), is written on to a PASCAL file and then punched on to cards. These cards can then be read and loaded onto the HP/2100A memory for execution. The SCOPE operating system is, thus, called on to punch the output from the HPCOM compiler on to the cards.

4.2.2 Interaction of PL/2100 program with its environment

A PL/2100 program runs on the HP/2100A machine which works under the Disc Operating System (DOSM).

Since the PL/2100 program is already in the relocatable binary format and is on the cards, it may either be loaded on the disc by the relocatable loader or be stored on the user file on the disc and be loaded later on. This could be done either through a Prog loadr call or through a STORE directive by the DOSM.

The standard relocatable loader called by the DOSM will relocate the program and writes the core image (the absolute binary program) of the program on to the disc. This may be stored on the user file by the STORE directive.

Then in order to execute the program the RUN directive is used by the DOSM and the program is loaded onto the memory and executed.

The DOSM system handles the EXEC calls (which are the line of communication between an executing program and DOS-M)$^{(H4)}$. One of the operations of EXEC calls is to perform input and output from the external devices. Thus a program with the help of its environment can communicate with the outside world in a given format specified in the program.

4.2.3 Operations done by the DOSM for a PL/2100 program

In this subsection we present what the Disc Operating System (DOSM) does for a PL/2100 program, namely,

i) Loading already compiled PL/2100 program into the memory of the computer (HP/2100A) and then placing it in execution.

ii) String input (ASCll character input).

iii) String output (ASCll character output).

4.3 Segmentations of the Compiler HPCOM

One of early ideas in the implementation of the compiler was to run it on the HP/2100A machine. The idea was rejected (as discussed in section 1 in Chapter I) because of the inconvenience of the implementation. In this section, we look into the idea of what was thought should be done or could be done.

4.3.1 Necessity for segmentation

The HP/2100A machine, at the present installation (at the Department of Applied Mathematics, McMaster University), is provided with 12K memory (1K = 1024 machine words). Out of this core of memory, DOSM (the operating system) takes about 4K for its own routines, so a programmer is virtually left with 8K of memory. On the other hand, the HPCOM compiler for the PL/2100A language takes at the present time, about 55K. (A small subset of PL/2100 might be used to get a smaller compiler but most of the beautiful features of the language PL/2100 would be lost!). Hence to implement this compiler on the HP/2100A machine, it is necessary to segment the compiler.

4.3.2 Segmentation

Once we know why we need to segment the HPCOM compiler, we look into the matter of how segmentation could be done.

4.3.2.1 Physical Segmentation

Under the DISC operating system, the different parts of the same program may be brought into the memory. This is how it is done.

User programs may be structured into a main program and several segments, as shown in figure 4.3.2.1. The main program starts at the beginning of the user program area.

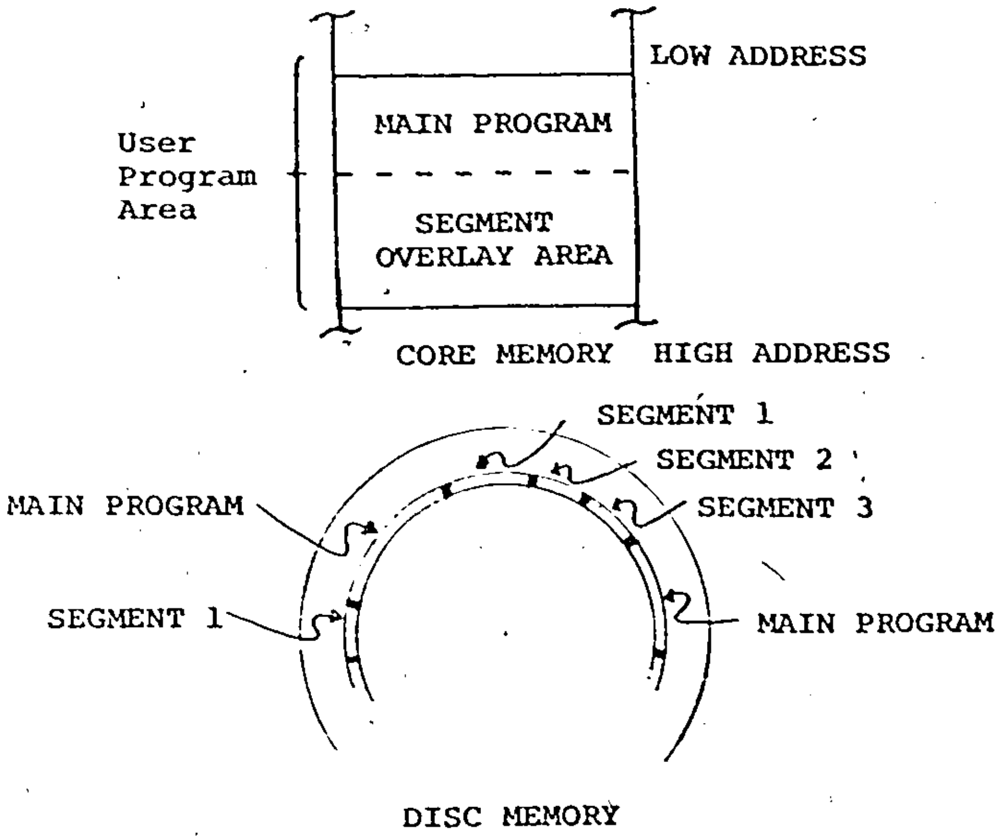


figure 4.3.2.1.a Segmented Programs

The area for the segments starts immediately following the last location of the main program. The segments reside on the disc, and are read into the core by an EXEC call, when needed. Only one segment may be in core at a time. When a segment is read into core, it overlays the segment previously in core. These EXEC calls may be generated by the compiler. This may be done by using a keyword SEG followed by an integer (to specify different segments) and whenever this keyword is encountered, the compiler would generate the proper EXEC calls. Each segment and the main program are distinct through their name (NAM name [,type]) associated with it. The main program must be type 3 and the segments must be type 5.

Each segmented program should use unique external reference symbols, otherwise the loader may link segments and main program incorrectly. The EXEC calling sequence (for loading main program and the segmented program) are given below in the HP assembler language:

```
            EXT EXEC
             .
             .
             .

            JSB EXEC  (transfer control to DOS-M)

            DEF *+3 (to 8)   (governed by the number of
                              parameters)

            DEF RCODE        (request code)

            DEF SNAME        (segment name)

            DEF PRAM 1       (first optional parameter)
             .
             .
             .
             .
            DEF PRAM 5       (fifth optional parameter)
             .
             .
RCODE       DEC 8 or 10      (8 = segmented programs,
                              10 = main program)

SNAME       ASC 3,xxxxx      (xxxxx is the segment name)

PRAM1          ---           (up to 5 words of parameter
PRAM2          ---            information are passed to
  .                           the segmented or the main
  .            ---            program)
PRAM5
```

When a main program and a segment are currently residing in core, they operate as a single program. Jumps from a segment to a main program (or vice versa) can be programmed by declaring an external symbol and referencing it via a JMP or JSB instruction (fig. 4.3.2.1.b). A matching entry symbol must be defined as the destination in the other program. It is the programmer's responsibility to make sure that the correct program is in core before any JMP instructions are executed.

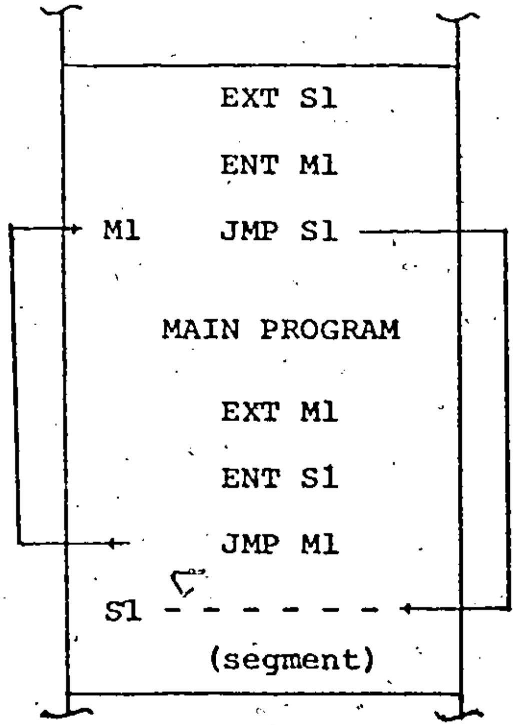
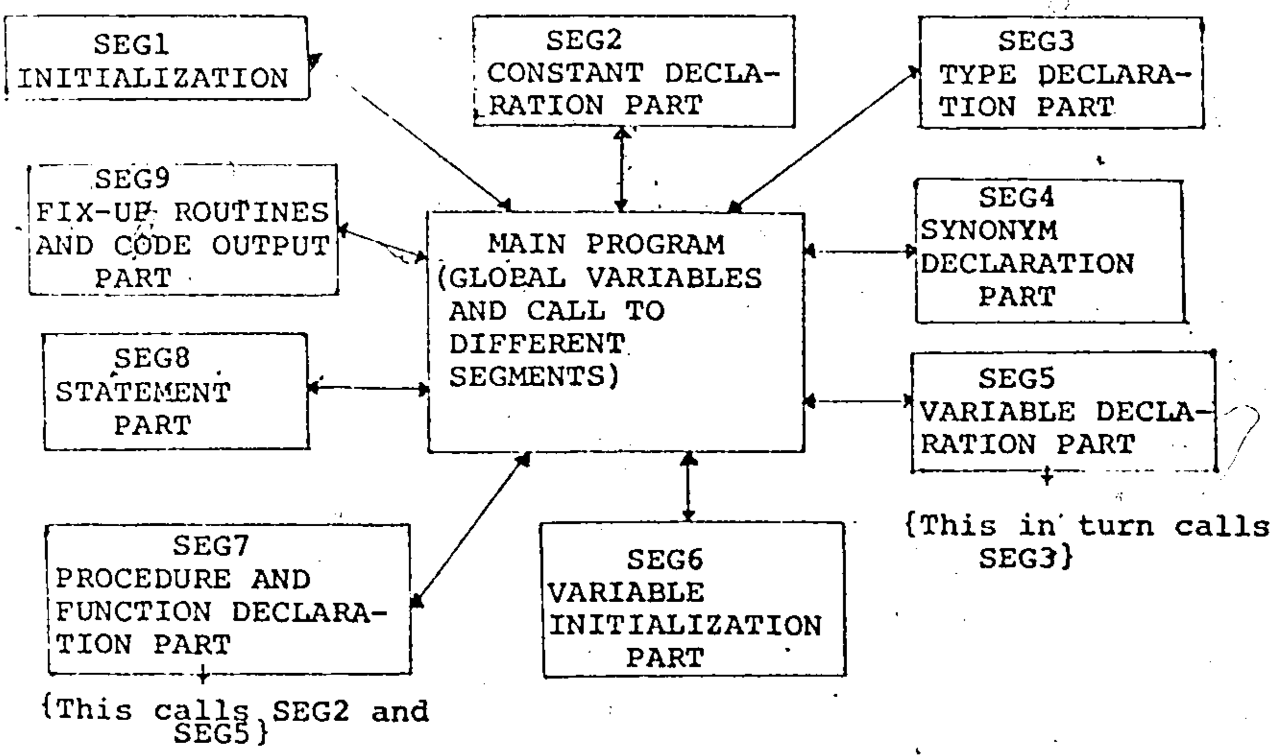


fig. 4.3.2.1.b main-to-segment jump

4.3.2.2. The logical segmentation of the compiler

Since the main program and only one segment could be in the core of memory at a particular time, it is necessary that the information that might be needed from one part of the compiler to another must be kept in the main program which is always present in the core. These informations include the symbol table, tables of reserved words in PL/2100 and all other global variables used in the compiler.

The other segments have been found conveniently from the syntax diagram of the PL/2100. The following block diagram shows the different segments that could be called by the main program and by each other.



{This in turn calls SEG3}

{This calls SEG2 and SEG5}

fig. 4.3.2.2 : Block diagram showing the segmented part of the compiler

The diagram above gives an indication of how HPCOM might be segmented into different parts. As one segment is overlayed by the other segment, it is necessary to have one segment calling another through the main program. For example, when seg5 calls seg3 it is necessary to jump back to the main program which in turns calls seg3. On returning from seg3, main procedure calls the seg5 and enters at the point in seg5 where it left before calling seg3. Another possibility is to segment the seg5 further into two parts viz.one before the call of seg3 and another after the call of seg3 and these two segments (viz seg51 and seg52) are called from the main program in the order namely,

```
      ⋮
Call seg51
Call seg3
Call seg52
      ⋮
```

All the necessary information is kept in the main program.

The main program, as one can see, is becoming larger in size and the other parts of the compiler need to be segmented further into smaller sizes. The method seems to be very complicated and cumbersome.

At the present stage as the HPCOM compiler does not produce codes for the procedure, it is not possible to check how big each segment would be.

An estimate of the amount of core required by the symbol table and other tables shows that the main program needs to be about 4.5K long (the size of symbol table is about 2.5K). Thus one is left with 3K for each segment. The compiler needs to be segmented further. For example, the statement part might be divided further into segments containing different statements (like <u>if-then-else</u>, <u>repeat</u>, <u>goto</u> etc.).

Because of the complexity of the segmentation, the method was discarded.

This section on segmentation at least tends to show the difficulty one might find in implementing a large compiler in a machine of small core. The swapping of one segment from another could be time-consuming and the segmentation becomes very complex.

CHAPTER V

CONCLUSION AND PROPOSALS FOR FURTHER WORK

5.1 Conclusion

The design of the language PL/2100 allows user-defined data types (which could be infinite in number). It, also, allows the use of some HP/2100A Assembler Instructions as operators; this allows the programmer to do bit-manipulations on the contents of a particular accumulator (namely, A- or B-register). The language allows "synonym declaration" (à la PL/360) on simple variables, only, so that more than one variable may use the same location in the core. The language, also, allows the variable initialization through the VALUE declaration.

The compiler HPCOM, written for PL/2100, is designed to be one-pass and runs on the CDC-6400 computer. The compiler HPCOM is written in PASCAL which is available on the CDC-6400 machine.

The compiler HPCOM, produces code for the various statements and expressions of the language PL/2100.

The input/output routines allow the transfer of information (to and from the external devices) in the ASCII character mode. Further modifications need to be done to have binary input/output; this is important if PL/2100 is to be used for writing the operating system of the HP/2100A computer.

The HPCOM compiler does not produce code for the procedure, so the compiler needs to have some routines to produce code for procedures (if procedures need be used in the language PL/2100).

The HPCOM compiler produces codes for the target machine HP/2100A in relocatable binary form, so that the standard relocatable loader could be used to link the object programs and load the object programs for execution.

The design of the language PL/2100 and its compiler HPCOM facilitates the extension and modifications to be done easily. We suggest a further modification in the language PL/2100 in the use of EXTERNAL symbols. This could be added in the declaration part of the language. The symbols could be followed by the key word <u>EXTERNAL</u> and parameters might be allowed, too. For example,

```
EXTERNAL
  IDIM(I,J)  ; ↱calculate K=I-MIN(I,J)↓
               ↱K in A-register ↓
  FLOAT(I)   ; ↱CONVERT I to real X ↓
               ↱result X in A- & B- register ↓
```

The addition of <u>externals</u> would facilitate the use of mathematical routines available in the HP/2100A library

The present work is mostly concerned with the design of the language PL/2100 and in producing a working compiler

HPCOM for it. Its merits rest on being the first very important step in building up a more complete system.

REFERENCES

C1. Conway, Melvin E. Design of a Separable Transition-Diagram Compiler, Communication of the ACM, volume 6, Number 7 (July 1963).

G1. Gries, David Compiler Construction for Digital Computer, John Wiley and Sons, Inc. (1971).

H1. Hewlett-Packard Microprogramming Guide Hewlett-Packard Company (Nov, 1971).

H2. Hewlett-Packard 2100A Computer Reference Manual, Hewlett-Packard Company (Dec. 1971).

H3. Hewlett-Packard HP Assembler Programmer's Reference Manual, Hewlett-Packard Company (June 1971).

H4. Hewlett-Packard Moving Head Disc Operating System Hewlett-Packard Company (March 1971).

H5. Hewlett-Packard Basic-Control System Hewlett-Packard Company (1971).

K1. Kalmer, Gabor G. An Implementation Language for Minicomputers, Technical Report, Computer System Research Group, University of Toronto (Jan. 1973).

M1. McKeeman, W.M., Horning, J.J. and Wortman, D.B. A Compiler Generator, Prentice-Hall, Inc., Englewood Cliffs, N.J., U.S.A. (1970).

S1. Sammet, J.E. A Brief Survey of Languages Used in Systems Implementation, Proceedings of ACM SIGPLAN Symposium on Languages for System Implementation, SIGPLAN Notices, volume 6, Number 9 (October 1971).

W1. Wirth, Niklaus PL/360, A Programming Language for the 360 Computers, Journal of the Association for Computing Machinery, Volume 15, Number 1, pp. 37-74 (January 1968).

W2. Wirth, Niklaus The Programming Language PASCAL Acta Information 1, pp. 35-63 (1971). OR Systematic Programming An Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A. (1973).

W3. Wirth, Niklaus — The Design of a PASCAL Compiler, Software - Practice and Experience, Volume 1, Number 1 (1971).

W4. Wulf — W. A. Wulf et al, BLISS Reference Manual, Carnegie-Mellon University, Computer Science Department (January 1970).

APPENDIX A

SYNTAX AND SEMANTICS OF PL/2100

A.1 Informal Definition of PL/2100

This section presents an informal introduction to the language. It is meant to give the reader an overview of the basic components of the language. A full syntactic and semantic definition of the language is contained in the next section.

A.1.1 The core of PL/2100

It is the core of the language which is recommended as an educational tool for teaching programming and as a tool for writing an operating system for the HP/2100A.

A.1.2 Basic Data

The basic data type of the language is scalar types. Their definition indicates an ordered set of values i.e. introduce an identifier as a constant standing for each value in the set. Apart from definable scalar types, there exist in PL/2100 two standard scalar types, whose values are not denoted by identifiers but instead by numbers and quotations respectively, which are syntactically distinct from identifiers. These types are: integer and char.

An integer may be written as a constant or it may be represented by a variable identifier.

A constant may be represented in integer, octal or character string form.

The set of values of type char is the character set available on a particular installation (which is ASCII character set for HP/2100A at the present installation). A character constant could only be two characters long for HP/2100A. A character is any element of the ASCII character set.

A scalar type may also be defined as a subrange of another scalar type by indicating the smallest and the largest value of the subrange.

A variable identifier is defined as any sequence of letters or digits beginning with a letter. Every integer and character variable may be initialized to a constant value at compile time.

A.1.3 Basic Operators

All the basic operators act only on integer values. They are divided into five classes:

a) the arithmetic operators of addition (+), subtraction (-), multiplication (*), division (div), and unary minus (-). The arithmetic operators return the integer value which results from the operation.

b) the logical operators and (and), or (or) and not (not). The logical operator returns a 1 if the result of the operation is true and a zero if the result of the operation is false. An individual operand is considered true if it is one and false if it is zero.

the relational operators greater than (>), greater than or equal to (≥), equal to (=), less than or equal to (≤), less than (<), and not equal to (≠). The relational operator returns a 1 if the relation is true and a 0 if the relation is false.

A.1.4 Data Structures

The data structures, available in PL/2100 are arrays and records. Array and record must be declared.

The bounds for an array are scalar type or subrange of type integer. Arrays may be initialized to any constant value. For referencing an array variable, the subscript or index may be any legal expression. The time needed for a selection of an array component does not depend on the value of the selector (index). The array structure is therefore called a random-access structure.

In a record structure, the components (called fields) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector does not contain a computable value, but instead consists of an identifier uniquely denoting the component to be selected. These component identifiers are defined in the record type definition. Again, the time needed to access a selected component does not depend on the selector, and the record structure is therefore also a random-access structure.

A record type may be specified as consisting of several variants. This implies that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a component field which is common to all variants and is called the tag field. Usually the part common to all variants will consist of several components, including the tag field.

A.1.5 Statement Structures

The choice of statement structures was motivated by the desire to promote structured programming. The basic statement structures of the language are

a) the assignment statement

<variable> :: = <expression>

b) the if-then-else statement

if <expression>
  then $<\text{statement}>_1$

if <expression> then $<\text{statement}>_1$
  else $<\text{statement}>_2$

which executes $<\text{statement}>_1$ if <expression> is true, or $<\text{statement}>_2$ if <expression> is false. The else part of the statement is optional.

c) the while statement

WHILE <expression> do <statement> which executes <statement> followed by the while statement, if necessarily Boolean <expression> is true. If <expression> is false, control passes to the statement following the while statement.

d) the repeat statement

repeat <statement> until <expression>

The expression controlling repetition must be of type Boolean. The sequence of statements between the symbols repeat and until is repeatedly (and at least once) executed until the expression becomes true.

e) the for statement

for <control variable> = <initial value> $\left\{\frac{\text{down to}}{\text{to}}\right\}$

<final value> do <statement>

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable of the for statement.

The control variable, the initial value and the final value must be of the same scalar type (or subrange thereof). The repeated statement must alter neither the value of the control variable nor the final value.

f) the CASE statement

The case statement consists of an expression (the selector) and a list of statements, each being labelled by a constant of the type of the selector. It specifies that one statement to be executed whose label is equal to the current value of the selector.

```
case <expression> of <case list element> end

<case list element> :: = {<case label>:}<statement>
                         |{<case label>:}
```

A.1.6 Program Structure

The choice of program structure was motivated by the desire to promote modular design of programs. A complete PL/2100 program consists of a program header card, followed by a nonempty sequence of segment defintions, followed by the main segment (or procedure) which is followed by the symbol progend. The program begins execution at the start of the main procedure.

A.1.7 Segment Definitions

A segment is either a procedure or a function. A procedure definition takes the form:

```
procedure  <procname> {(<parameter list>)}
           {<local declaration list>} <statement part>
```

where <procname> is the name of a procedure (an identifier), <parameter list> is an optional parameter list which consists of typed formal parameters, and <local declaration list>

is a possibly empty list of local variables that may not be initialized. A procedure could be recursive. Global variables may appear anywhere in the statement sequence.

A function definition takes the form

```
function <funcname> {(<parameter list>)}: <result type>
          {<local declaration list>} <statement part>
```

where <funcname> is the name of the function (an identifier), and <parameter list> and <local declaration list> are as defined above. The identifier representing the function name, returns the value of the function. <type> represents the type of the function (represented by the identifier).

A.1.8 Comments

A comment is any string of characters (except the symbol */) between /* and */ and may appear in the program wherever a blank may occur.

A.2 Syntax and Semantics of Pl/2100

This section contains the syntactic and semantic definition of the language.

A.2.1 Notation and Terminology

According to traditional BNF (Backus-Normal Form OR Backus Naur Form) notation, syntactic constructs are denoted by English words enclosed between angular brackets < and >.

These words also describe the nature or meaning of the construct, and are used in the accompanying descriptions of semantics.

Possible repetitions of a construct are indicated by an asterisk viz * (0 or more repetitions) or a circular plus sign viz ⊕ (1 or more repetitions).

If a sequence of construct to be repeated consists of more than one element, it is enclosed by the meta brackets { and }.

A.2.2 Program

A.2.2.1 Syntax

<program> :: = program <identifier> <option list>
<block> progend

<option list> :: = ,R|,P|,B|<option list>

<block> :: = <declaration list> <function or procedure declaration> <statement part>

A.2.2.2 Semantics

A program consists of a set of declarations which are global to the whole program , followed by a series of segment definitions (if any) which are either procedure or function declarations followed by the compound tail which is the main program and the symbol progend which specifies the end of a program.

The program starts with a symbol program followed by the identifier which is the name of the program, and followed by a list of options (if any).

In the list of options, R specifies the printing of the table containing object codes, B specifies the printing of the object code in relocatable binary format and P specifies the relocatable binary code to be punched on the card.

A.2.2.3 Example

```
            PROGRAM FACT, B, P, R
                 /* FINDS THE FACTORIAL OF INTEGERS UP TO 4*/
                    CONST   MAX = 4;
                    VAR     K,N,FACT:INTEGER;
/* this is the
 beginning of a
 <compound tail>*1  BEGIN

                          K: = 1; FACT: = 1
            1:            IF (K EQ MAX) THEN GO TO 2;
                          K: = K+1;
                          FACT: = FACT*K;
                          GO TO 1;
            2:            N: = FACT;
                     END;
/* this is  PROGEND
the end of
a compound
tail */
/* there is
no procedure
or function present */
```

A.2.3 Declarations

A.2.3.1 Syntax

<declaration list> :: =

<label declaration part>
|<constant definition part>

|<type definition part>

|<variable declaration part>
{<variable initialization part>}

|<synonym definition part>

<label declaration part> :: = <empty>|

label <label>{, <label>}

<constant definition part> :: = <empty>|

CONST <constant definition>{,<constant definition>};

<type definition part> :: = <empty>|

type <type definition>{;<type definition>};

<synonym definition part> :: = <empty>|

syn <synonym definition>{;<synony definition>};

<variable declaration part> :: = <empty>|

var <variable declaration>{;<variable declaration>};

<variable initialization part> :: = <empty>|

value <variable initialization>{;<variable initialization>};

A.2.3.1.1

<constant definition> :: = <identifier> = <constant>

<constant> :: = <unsigned constant>|<sign><number>
|<identifier>

<unsigned constant> : = <number>| ≡ <character> ≡

A.2.3.1.2

<type definition> :: = <identifier> = <type>

<type> :: = <scalar type>|<subrange type>|<array type>|
<record type>|<type identifier>

<type identifier> : = <identifier>

A.2.3.1.2.1

```
<scalar type> :: = (<identifier> {,<identifier>})
```

A.2.3.1.2.2

```
<subrange type> :: = <constant> .. <constant>
```

A.2.3.1.2.3

```
<array type> :: = array [<index type>{,<index type>}]
                  of <component type>
<index type> :: = <scalar type>|<subrange type>
<component type> :: = <type>
```

A.2.3.1.2.4

```
<record type> :: = record <field list> end
<field list> :: = <fixed part>|<fixed part>;<variant part>
                 |<variant part>
<fixed part> :: = <record section>{;<record section>}
                      : <type>
<variant part> :: = case <tag field> : <type identifier>
                    of <variant>{; <variant>}
<variant> :: ={<case label>:}⊕(<field list>)|{<case label>:}⊕
<case label> :: = <unsigned constant>
<tag field> :: = <identifier>
```

A.2.3.1.3

```
<variable declaration> :: = <identifier>{,<identifier>}
                                          : <type>
```

A.2.3.1.4

```
<variable initialization>
 :: = <identifier> =<constant>|(<constant>,{<constant>}⊕)
```

A.2.3.1.5

```
<synonym definition>
        :: = <identifier> = <identifier>{,<identifier>};
```

A.2.3.2 Semantics

The declaration list consists of all the constant and integer, character, array and record variables that are global to the program.

Integer variables and arrays, and character variables and arrays may be initialized with their values assigned at compile time. In order to facilitate the initializing several elements of an array with the same value, any initialization value may be followed by an asterisk (*) followed by a constant, implying that initial value should be assigned to the next consecutive sequence of elements whose length is defined by the constant in parentheses.

A.2.3.3 Examples

A.2.3.3.1

```
const A = 1, B = 2, C = 3;
```

A.2.3.3.2

```
type
      color = (red, orange, yellow, green, blue);
      cards = (club, diamond, heart, spade);
      index = -10 .. 10;
      days  =  Monday .. Friday;
      vector = array [1 .. 10] of integer;
      booltab = array [1..10] of boolean;
```

```
calendar = record day : 1..31;
                  month : 1..12;
                  year : 0..2000
           end
```

A.2.3.3.3 syn

```
D = A,B,C;
B = J,I;
```

A.2.3.3.4 var

```
A : vector;
B : char;
C : boolean ;
D : array [1..10] of integer;
```

A.2.4 Program Segments

A.2.4.1 Syntax

```
<procedure and function declaration part> :: =
  {<procedure or function declaration>;}
<procedure or function declaration> :: =
   <procedure declaration>|<function declaration>
<function declaration> :: =
   <function heading> <label declaration part>
   <constant definition part>  <type definition part>
   <variable declaration part>
   <procedure and function declaration part>  <statement part>
```

```
<function heading> :: =
  function <identifier> (<formal parameter section>
    {;<formal parameter section>}) : <result type>;
    <result type> :: = <type identifier>
<procedure declaration> :: = <procedure heading><label
                              declaration part>
  <constant definition part><type definition part>
     <variable declaration part>
<procedure and function declaration part><statement part>
<procedure heading> :: = procedure <identifier> ; |
   procedure <identifier>(<formal parameter section>
   {;<formal parameter section>});
<formal parameter section> :: =
   <parameter group> |
  const <parameter group>{;<parameter group>}|
  var   <parameter group>{;<parameter group>}|
  function <parameter group>|
  procedure <identifier>{,<identifier>}
<parameter group> : = <identifier>{,<identifier>}:
                          <type identifier>
```

A.2.4.2 Semantics

A PL/2100 program consists of a sequence of procedures and functions called program segments.

A.2.4.3 Example

A.2.4.3.1

```
procedure add (var x : integer; var y : integer);
    begin
         z: = x+y+1; /* z is a global variable */
    end;
```

A.2.4.3.2

```
function  sum (var x: integer; var y:  integer):
        integer;
    begin
         sum: = x+y
    end;
```

A.2.5 Statements

A.2.5.1 Syntax

```
<statement part> :: = <compound statement>
<compound statement> :: = begin <component statement>
    {;<component statement>} end
<component statement> :: = <statement> |
    <label definition> <statement>
      <label> :: = <integer>
```

A.2.5.1.1

```
<statement> :: = <simple statement>|
                 <structured statement>
<simple statement> :: = <assignment statement>|
               <procedure statement>|<go to statement>
<assignment statement> :: = <variable>: = <expression>|
               <function identifier>: = <expression>
<procedure statement> :: = <procedure identifier>|
 <procedure identifier> (<actual parameter>
  {,<actual parameter>})
```

```
<procedure identifier> :: = <identifier>
<actual parameter> :: <expression>|<variable>
         |<parameter identifier>|<function identifier>
<goto statement> ::= goto <label>
       <label> :: = <integer>
```

A.2.5.1.2

```
<structured statement>
:: = <compound statement>|<conditional
         statement>|<repetitive statement>
     <conditional statement> :: = <if statement>|
                                  <case statement>
     <repetitive statement> :: =
        <while statement>|<repeat statement>
            |<for statement>
```

A.2.5.1.2.1

```
<if statement> :: = if <expression> then <statement>
|if <expression> then <statement> else
<statement>
<case statement>:: = case <expression> of
  <case list element>{;<case list element>} end
<case list element> :: = {<case label>:}<statement>|
 {<case label>:}
```

A.2.5.1.2.2

```
<repeat statement> :: = repeat <statement>
  {;<statement>} until <expression>
<while statement> :: = while <expression> do <statement>
```

```
<for statement> :: = for <control variable>: =
                        <for list> do <statement>

<for list> :: = <initial value> to <final value>|

                <initial value> downto <final value>

<control variable> :: = <identifier>

<initial value> :: = <expression>

<final value> :: = <expression>
```

A.2.5.2 Semantics

A statement part is a sequence of statements.

A.2.5.3 Examples

A.2.5.3.1

```
y : = x
```

A.2.5,3.2

```
if x then
        y : = x
    else
        y : = z
```

A.2.5.3.3

```
case operator of
plus : x : = x+y ;
times : x : = x*y;
absval : if x < o then x : = x
end
```

A.2.5.3.4

```
while i > o  do
begin
      z : = z*x;
      i : = i div 2
end
```

```
A.2.5.3.5  repeat
                k : = i mod j;
                i = j;
                j : = k
           until j = 0;
```

```
A.2.5.3.6  for  i = 2 to 100 do if a[i] > max then
                                        max : = a[i]
```

A.2.6 Expression

A.2.6.1 Syntax

```
<expression> :: = <primary>|<operator><expression>
                  |<shift><operator><expression>
<primary> :: = <constant>|<variable>|
              (<expression>)|<unary op><primary>
<variable> :: = <identifier>|<identifier>•<identifier>
              |<identifier>[<expression>]
<shift> : = <shift operator><factor>
<factor> :: = <integer>
<operator> :: = <arithmetic operator>|<logical operator>
              |<relational operator>|<bit operator>
<arithmetic operator> :: = +|-|*|div
<logical operator> :: = and|or
<relational operator> :: =  LT|<|LE|≤|GE|≥|EQ|=|
                            NE|≠|GT|>
<shift operator> :: =  ALS|BLS|ARS|BRS|ALF|BLF|
                       RAL|RBL|RAR|RBR
```

<bit operator> :: = IOR|XOR

<unary op> :: = NOT|-

<constant> :: = <signed constant>|<unsigned constant>

<signed constant> := <sign><integer>

<sign> : = +|-

<unsigned constant> :: = <integer>|<octal>|<character constant>

<integer> :: = <digit>$^{\oplus}$

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

<octal> :: = <octal digit>$^{\oplus}$B

<octal digit> : = 0|1|2|3|4|5|6|7

<character constant> :: = ≡ <character string> ≡

<character string> :: = <letter or digit>$^{\oplus}$

<letter or digit> :: = <letter>|<digit>

<identifier> :: = <letter or digit>$^{\oplus}$

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

A.2.6.2 Semantics

An expression is a rule for computing a numerical value. A primary is either a constant, a variable or an expression enclosed in parentheses. The operators have the following meaning associated with them. Note that several of the operations are machine dependent.

A.2.6.2.1 Unary minus (-)

Returns the negative of the primary, it preceeds.

A.2.6.2.2 Logical not (not)

Returns a 1 if the value of the primary it preceeds is zero and returns a zero if the value of the primary it preceeds is one.

A.2.6.2.3 Left Arithmetic Shift (als)

The A-register is shifted left by the number of bits specified by the <factor>. Sign bit is not affected. Bit shifted out of bit 14 is lost. A "0" replaces vacated bits on the right.

A.2.6.2.4 Right Arithmetic Shift (ars)

The A-register is shifted right by the number of bits specified by the <factor>. Sign bit (bit 15) is not affected; copy of sign bit is shifted into the bits adjacently right to it. Bit shifted out of bit 0 is lost.

A.2.6.2.5 Rotate A Left (ral)

Rotate A-register left by number of places specified by the <factor>. Bits 15, 14 etc. are rotated respectively, into bit 1, 0 etc.

A.2.6.2.6 Rotate A Right (rar)

Rotate A-register right by number of places specified by the <factor>. Bits 0, 1 etc. are rotated, respectively, into bits 14, 15 etc.

A.2.6.2.7 Rotate A Left Four (alf)

Rotate A register left four places and <factor> denotes how many of this type of rotation is to be done. For each alf rotation bits 15, 14, 13, 12 are rotated around to bits 3, 2, 1,0 respectively.

A.2.6.2.8 Corresponding Shifts on the B-register

The operators brs, rbl, rbr, blr, bls, blf may be used to shift bits on the B-register and these correspond exactly to those operators described for the A-register.

A.2.6.2.9 Bit inclusive or (ior)

The contents of the A-register are combined with the contents of the memory location as an "inclusive or" logic operation.

A.2.6.2.10 Bit exclusive or (xor)

The contents of the memory location are combined with the contents of the A-register by an "exclusive or" operation.

A.2.6.2.11 Addition (+)

Returns the sum of the expression and the primary.

A.2.6.2.12 Subtraction (-)

Returns the result of subtracting the primary from the expression.

A.2.6.2.13 Multiplication (*)

Returns the result of the multiplication of the primary and the expression.

A.2.6.2.14 Division (div)

Returns the quotient of the primary and the expression.

A.2.6.2.15 Equal (=)

Returns a 1 if the relation and the expression are equal, and 0 otherwise.

A.2.6.2.16 Other Relations

Not equal (≠), greater than (>), less than (<), greater than or equal (≥) and less than or equal (≤) are similar to equal (=).

A.2.6.2.17 Logical And (and)

Returns a 1 if the both of the boolean expression and the primary are nonzero and a zero otherwise.

A.2.6.2.18 Logical or (or)

Returns a 1 if either or both of the expressions and the primary are non-zero and zero otherwise.

A.2.6.2.19 Precedence

The operators are of the same precedence. Expression is evaluated from left to right. Only an expression between two parentheses have higher precedence than the one not inside a pair of parentheses.

A.2.6.2.20 Character String

A character string constant occupies one machine word. The constant is padded on the right with blanks or truncated on the right as needed to fit into one word. Note this is machine dependent to the extent of both word size and internal character representation.

In HP/2100A, the word size is 16 bits and can have only two characters which are, internally, represented as ASCII character set.

A.2.6.2.21 Examples

i) x+y*z div U - 2

ii) A als 6 ior mask 1 xor mask 2

iii) A and B < C

iv) -x rar 5

v) A*(x+y*z) and A*x+y*z will have different results

vi) (x+y*z)*A and x+y*z*A will have the same results

vii) ≡ A2 ≡

A.2.7 Blanks

One or more blanks appear anywhere except within a symbol,identifier, or operator.

A.2.8 Comments

Comments are any string of characters (except the symbol */) between /* and */. A comment has no effect on the program and may appear anywhere that blanks may occur.

<comment> :: = <opening bracket><almost anything>
<closing bracket>

<opening bracket> :: = /*

<closing bracket> :: = */

<almost anything> :: = {any string of valid ASCII characters which does not contain a <closing bracket>}

APPENDIX B

HEWLETT PACKARD ASSEMBLY LANGUAGE INSTRUCTIONS AND THEIR MEANINGS

In this appendix the HP instructions (only those which have been used as operators or in compiler-generated codes) are given along with their meanings.

Notations used in representing the HP assembly language instruction is as follows:

m	Memory location
I	Indirect addressing locator
Comments	Optical comments
[ ]	Brackets defining a field or position of a field that is optional
{ }	Brackets indicating that one of the set may be selected
lit	literal

B.1 Memory Reference

Memory reference instructions perform arithmetic, logical and jump operations on the contents of the locations in core and the registers. An instruction may directly address the 2048 words of the current and base pages. If required, indirect addressing may be utilized to refer to all 27,777 words of memory. Expressions in the operand field may evaluate modulo 2^{10}.

If the program is in relocatable form (which is the case here as HPCOM produces relocatable binary code), the operand field may contain relocatable expressions or absolute expressions which are less than 100_8 in value.

B.1.1 Jump

Jump instructions may alter the normal sequence of program execution.

B.1.1.1 JMP

label JMP m[,I] comments

Jump to m. Jump indirect inhibits interrupt until the transfer of control is complete.

B.1.1.2 JSB

label JSB m[,I] comments

Jump to subroutine. The address for label+1 is placed into the location represented by m and control transfer to m+1. On completion of the subroutine, control may be returned to the normal sequence by performing a JMP m,I.

B.1.2 Add, Load and Store

Add, Load and Store instructions transmit and alter the contents of memory and of the A- and B-registers. A literal, indicated by "lit" may be either = B, = D, = A or = I type.

B.1.2.1 ADA

label ADA $\left\{\begin{matrix}\text{m[,i]} \\ \text{lit}\end{matrix}\right\}$ comments

Add the contents of m to A.

B.1.2.2 ADB

label ADB $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

Add the contents of m to B.

B.1.2.3 LDA

label LDA $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

Load A from m.

B.1.2.4 LDB

label LDB $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

Load B from m.

B.1.2.5 STA

label STA $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

Store contents of A in m.

B.1.2.6 STB

label STB $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

Store contents of B in m.

In each instruction, the contents of the sending location is unchanged after execution.

B.1.3 Logical Operations

The logical instructions allow bit manipulation and the comparison of two computer words.

B.1.3.1 AND

label AND $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

The logical product of the contents of m and the contents of A are placed in A.

B.1.3.2 XOR

label XOR $\left\{\begin{matrix} m[,I] \\ lit \end{matrix}\right\}$ comments

The modulo-two sum (exclusive "or") of the bits in m and the bits in A is placed in A.

B.1.3.3 IOR

label IOR $\left\{\begin{matrix} m[,I] \\ lit \end{matrix}\right\}$ comments

The logical sum (inclusive "or") of the bits in m and the bits in A is placed in A.

B.2 Register Reference

The register reference instructions include a shift-rotate group, an alter-skip group and NOP (no operation). With the exception of NOP, they have the capability of causing several actions to take place during one memory cycle. Multiple operations within a statement are separated by a comma.

B.2.1 Shift-Rotate Group

This group contains 19 basic instructions that can be combined to produce more than 500 different single cycle operations.

CLE	**Clear E to zero**
ALS	**Shift A left one bit, zero to least significant bit. Sign unaltered.**
BLS	**Shift B left one bit, zero to least significant bit. Sign unaltered.**
ARS	**Shift A right one bit, extend sign; sign unaltered.**
BRS	**Shift B right one bit, extend sign; sign unaltered.**

RAL	Rotate A left one bit
RBL	Rotate B left one bit
RAR	Rotate A right one bit
RBR	Rotate B right one bit
ALR	Shift A left one bit, clear sign, zero to least significant bit
BLR	Shift B left one bit, clear sign, zero to least significant bit
ERA	Rotate E and A right one bit
ERB	Rotate E and B right one bit
ELA	Rotate E and A left one bit
ELB	Rotate E and B left one bit
ALF	Rotate A left four bits
BLF	Rotate B left four bits
SLA	Skip next instruction if least significant bit in A is zero
SLB	Skip next instruction if least significant bit in B is zero.

These instructions may be combined as follows:

$$\text{label} \quad \left[\begin{Bmatrix}\text{ALS}\\ \text{ARS}\\ \text{RAL}\\ \text{RAR}\\ \text{ALR}\\ \text{ALF}\\ \text{ERA}\\ \text{ELA}\end{Bmatrix}\right] \quad [\text{,CLE}] \quad [\text{,SLA}] \quad \left[,\begin{Bmatrix}\text{ALS}\\ \text{ARS}\\ \text{RAL}\\ \text{RAR}\\ \text{ALR}\\ \text{ALF}\\ \text{ERA}\\ \text{ELA}\end{Bmatrix}\right] \quad \text{comments}$$

$$\text{label} \quad \left[\begin{Bmatrix}\text{BLS}\\ \text{BRS}\\ \text{RBL}\\ \text{RBR}\\ \text{BLR}\\ \text{BLF}\\ \text{ERB}\\ \text{ELB}\end{Bmatrix}\right] \quad [\text{,CLE}] \quad [\text{,SLB}] \quad \left[,\begin{Bmatrix}\text{BLS}\\ \text{BRS}\\ \text{RBL}\\ \text{RBR}\\ \text{BLR}\\ \text{BLF}\\ \text{ERB}\\ \text{ELB}\end{Bmatrix}\right] \quad \text{comments}$$

CLE,SLA, or SLB appearing alone or in any valid combinations with each other are assumed to be a shift-rotate machine instruction.

The shift-rotate instructions must be given in the order shown. At least one and up four are included in one statement. Instructions referring to the A-register may not be combined in the same statement with those referring to the B-register.

B.2.2 No-Operation Instruction

When a no-operation is encountered in a program, no action takes place; the computer goes on to the next instruction. A full memory cycle is used in executing a no-operation instruction.

label NOP comments

A subroutine to be entered by a JSB instruction should have a NOP as the first statement. The return address can be stored in the location coupled by the NOP during execution of the program. A NOP statement causes the assembler to generate a word of zeros.

B.2.3 Alter-Skip Group

The alter-skip group contains 19 basic instructions that can be combined to produce more than 700 different single cycle operations.

CLA	Clear the A-register
CLB	Clear the B-register
CMA	Complement the A-register
CMB	Complement the B-register
CCA	Clear, then complement ~~the A-~~register (set to ones)
CCB	Clear, then complement the B-register (set to ones)
CLE	Clear the E-register
CME	Complement the E-register
CCE	Clear, then complement the E-register
SEZ	Skip next instruction if E is zero
SSA	Skip if sign of A is positive (0)
SSB	Skip if sign of B is positive (0)
INA	Increment A by one
INB	Increment B by one
SZA	Skip if contents of A equals zero
SZB	Skip if contents of B equals zero
SLA	Skip if least significant bit of A is zero
SLB	Skip if least significant bit of B is zero
RSS	Reverse the sense of the skip instructions. If no skip instruction preceed in the statement, skip the next instruction.

These instructions may be combined as follows:

$$\text{label}\left[\begin{Bmatrix}\text{CLA}\\\text{CMA}\\\text{CCA}\end{Bmatrix}\right]\text{[,SEZ]}\left[,\begin{Bmatrix}\text{CLE}\\\text{CME}\\\text{CCE}\end{Bmatrix}\right]\text{[,SSA][,SLA][,INA][,SZA][,RSS] comments}$$

$$\text{label}\left[\begin{Bmatrix}\text{CLB}\\\text{CMB}\\\text{CCB}\end{Bmatrix}\right]\text{[,SEZ]}\left[,\begin{Bmatrix}\text{CLE}\\\text{CME}\\\text{CCE}\end{Bmatrix}\right]\text{[,SSB][,SLB][,INB][,SZB][,RSS] comments}$$

The alter-skip instructions must be given in the order shown. At least one and up to eight are included in one statement. Instructions referring to the A-register may not be combined in the same statement with those referring to the B-register. When two or more skip functions are combined in a single operation, a skip occurs if any of the conditions exist. If a word with RSS also includes both SSA and SLA (or SSB and SLB), a skip occurs only when sign and least significant bit are both set to (1).

B.3 Extended Arithmetic Instructions

These instructions may be used with the EAU version of the Assembler or Extended Assembler to increase the computer's overall efficiency. The computer (HP/2100A) must include the Extended Arithmetic Unit option to obtain the resulting increase in available core storage and decrease in program run time.

Only two of these instructions have been used in the HPCOM and are considered here. Both are memory reference instructions.

B.3.1 MPY

label MPY $\begin{Bmatrix} m[,I \\ \text{lit} \end{Bmatrix}$ comments

The MPY instruction multiplies the contents of the A-register by the contents of m. The product is stored in registers B and A. B contains the sign of the product and the 15 most significant bits; A contains the least significant bits.

B.3.2 DIV

label DIV $\left\{ \begin{matrix} m[,I] \\ lit \end{matrix} \right\}$ comments

The DIV instruction divides the contents of registers B and A by the contents of m. The quotient is stored in A and the remainder in B. Initially B contains the sign and the 15 most significant bits of the dividend. A contains the least significant bits.

APPENDIX C

COMPILE-TIME TABLES FOR RESERVED WORDS USED IN PL/2100

C.1 Reserved Symbols and Associated Integer Token

In this section we display in the following table, the reserved symbols (used in PL/2100) along with the integer tokens (NO and CL) associated with them.

SYMBOL	NO	CL	SYMBOL	NO	CL
ID	1	0	ALS	24	1
INTEGER	2	1	BLS	24	2
CHAR	2	2	ARS	24	3
	5	1	BRS	24	4
NOT	5	1	RAR	24	5
*	6	1	RBR	24	6
1	6	2	RAL	24	7
∧	6	3	RBL	24	8
AND	6	3	ALF	24	9
DIV	6	4	BLF	24	10
MOD	6	5	IOR	24	11
+	7	1	XOR	24	12
-	7	2	BEGIN	25	0
V	7	3	END	26	0
OR	7	3	IF	27	0
<	8	1	THEN	28	0
LT	8	1	ELSE	29	0
≤	8	2	CASE	30	0
LE	8	2	OF	31	0
≥	8	3	REPEAT	32	0
GE	8	3	UNTIL	33	0
>	8	4	WHILE	34	0
≠	8	5	DO	35	0
NE	8	5	FOR	36	0
=	8	6	TO	37	1
EQ	8	6	DOWNTO	37	2
IN	8	7	GOTO	38	0
(	9	0	NIL	39	0
)	10	0	TYPE	40	0
[	11	0	ARRAY	41	0
]	12	0	RECORD	41	2
,	15	0	FILE	41	3
;	16	0	LABEL	42	0
.	17	0	CONST	43	0
:	21	0	VAR	45	0
:=	22	0	FUNCTION	46	0
/*	23	0	PROCEDURE	47	0
			VALUE	48	0
			WITH	49	0
			REGISTER	50	0
			SYN	51	0
			REGA	52	1
			REGB	52	2

C.2 Tables of Reserved Symbols and Their Pointers

The reserved words of the language PL/2100A are stored in different tables according to the lengths of the words. The pointers associated with the top of each table, are also stored in a separate table. In the actual programming world, the tables of reserved words are stored in an array of characters and the table of pointers in an array of integer.

C.2.1 Table of Reserved Words

Index	Reserved Words of PL/2100	Index	Reserved Words of PL/2100
1	IF	32	REGA
2	DO	33	REGB
3	TO	34	THEN
4	OF	35	ELSE
5	OR	36	GOTO
6	LT	37	CASE
7	LE	38	WITH
8	GT	39	TYPE
9	GE	40	FILE
10	NE	- - -	- - -
11	EQ	41	BEGIN
- - -	- - -	42	UNTIL
12	AND	43	WHILE
13	END	44	ARRAY
14	NIL	45	VALUE
15	FOR	46	CONST
16	DIV	47	LABEL
17	MOD	- - -	- - -
18	VAR	48	PACKED
19	SYN	49	REPEAT
20	ALS	50	DOWNTO
21	BLS	51	RECORD
22	ARS	- - -	- - -
23	BRS	52	FUNCTION
24	RAR	53	REGISTER
25	RBR	- - -	- - -
26	RAL	54	PROCEDURE
27	RBL		
28	ALF		
29	BLF		
30	IOR		
31	XOR		

C.2.2 Table of Pointers

Column 1 INDEX	Column 2 POINTER VALUE
1	1
2	1
3	12
4	32
5	41
6	48
7	52
8	52
9	54
10	55
11	55

It is important to note that the index values in the first column of this table correspond to the length of the reserved words stored in a table pointed to by the pointer value given in the second column.

This has been done to facilitate a faster search through the tables of reserved words.

C.2.3 Tables of Token Values of the Reserved Words

Two tables viz WNO and WCL (in the programming world, they are arrays of integer) have been used to store the token value NO and CL. These tables are given below.

```
WNO = (27,35,37,31,7,8,8,8,8,8,8,6,26
       39,36,6,6,45,51,24,24,24,24,24,24,24
       24,24,24,24,24,52,52,28,29,38,30,49,40,
       41,25,33,34,41,48,43,42,53,32,37,41,46,50,47)
```

```
WCL = (0,0,1,0,3,1,2,4,3,5,6,3,0,0,0,4,5,0,
       0,1,2,3,4,5,6,7,8,9,10,11,12,1,2,0,0,0,
       0,0,0,3,0,0,0,1,0,0,0,0,0,2,2,0,0,0)
```

APPENDIX D

LISTING OF LEXICAL ANALYSER

```
↗**********************************************************************↓
        PROCEDURE NEXTCH ;
            ↗ READS NEXT CHAR OF FILE INPUT,
              READS EOL TO BLANK ,
              PRINTS ERROR SUMMARY AFTER EACH LINE
              PRINTS ADDRESS AT THE BEGINNING OF LINE ↓
          BEGIN
            IF EOLFLAG THEN
            BEGIN
                IF ERRINX GT 0 THEN PRTERR ↗ PRINT ERROR ↓ ;
                EOLFLAG := FALSE ;  CHCNT:= 0 ;
                OUTPUT↑ := ≡ ≡ ; PUT(OUTPUT) ;
                      IF OP THEN OUTO(LC) ELSE OUTO(IC) ;
                OUTPUT↑ := ≡ ≡ ; PUT(OUTPUT) ;
            END ↗ EOLFLAG ↓ ;
            GET(INPUT) ; CH:= INPUT↑ ; CHCNT := CHCNT + 1 ;
            IF CHCNT  > 73 THEN
            BEGIN
                OUTPUT↑:= ≡*≡ ;
                PUT(OUTPUT) ;
                OUTPUT↑ := CH ;
                PUT(OUTPUT) ;
                REPEAT
                     GET(INPUT) ; CH:= INPUT ↑ ;
                     OUTPUT↑ := CH ;
                      PUT(OUTPUT) ;
             UNTIL CH=EOL ;
            END  ELSE
            BEGIN OUTPUT↑ := CH ; PUT(OUTPUT) ;
                  IF CH≠ EOL THEN  GOTO 10
            END ;
            CH:= ≡ ≡ ; EOLFLAG := TRUE ;
    10:   END ↗ NEXTCH↓ ;
↗******************          *********        ********      ********↓
```

```
PROCEDURE  INSYMBOL ;
     ↗ LEXICAL ANALYSER ↓
  VAR
  I,K : INTEGER ;
 DIGIT : ARRAY [0.. 4] OF INTEGER ;
 BEGIN
1:    WHILE CH=≡ ≡ DO NEXTCH ;
      IF CH≤ ≡Z≡ THEN        ↗ IDENTIFIER ↓
      BEGIN
            K:=0 ;
            REPEAT
               K:=K+1 ;
               A[K] := BLANK ;
            UNTIL  K=10 ;
            K:=0;
           REPEAT IF K<ALFALENG THEN
                BEGIN
                     K:=K+1;
                     A[K] :=CH ;
                END;
                     NEXTCH;
           UNTIL CH > ≡9≡ ;
               ↗ SEARCH FOR RESERVED WORDS ↓
           FOR I:=WL[K] TO WL[K+1] -1 DO
                BEGIN
                       CPARE(A,WD,I,COMPARE) ;
                     IF COMPARE THEN
                     BEGIN NO :=WNO[I] ;
                           CL:=WCL[I] ; IVAL:=0;
                           GOTO 2 ;
                      END ;
                END;
                NO:=1 ; CL:=K;
                I :=0 ;
                REPEAT
                   I:=I+1 ;
                   AVAL[I] := BLANK ;
                UNTIL  I=10 ;
                I :=0 ;
                REPEAT
                    I:=I+1 ;
                    AVAL[I] :=A[I] ;
                UNTIL I=K;
2:   END  ELSE
       IF CH ≤ ≡9≡ THEN      ↗ NUMBER ↓
       BEGIN NO:=2; CL:=1; I:=0;
```

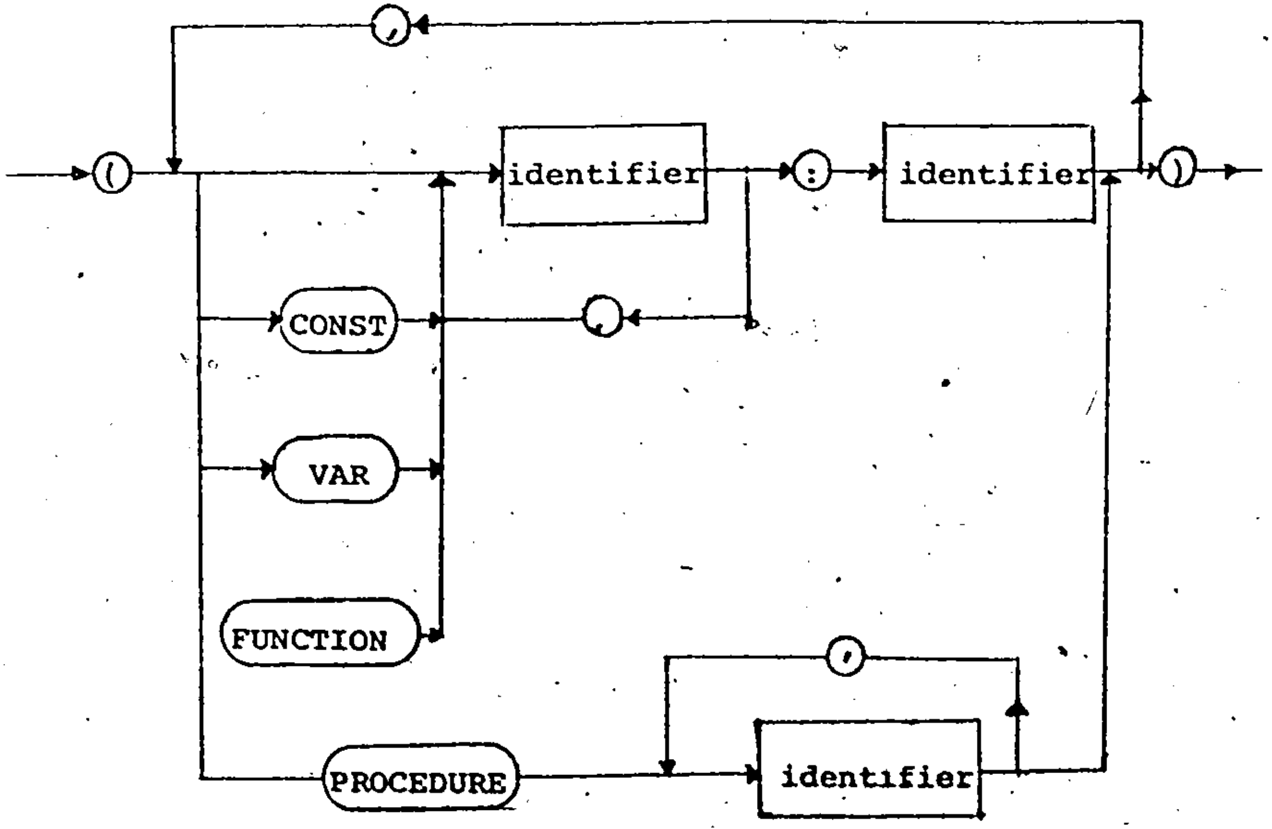
```
      WHILE CH IN DIGITS DO
      BEGIN  IF I< 5 THEN
          DIGIT[I] :=ORD(CH) - ORD(≡0≡) ;
          I:=I+1; NEXTCH ;
      END ;
      IVAL :=0;
      IF CH = ≡B≡ THEN   ↱OCTAL ↓
            BEGIN IF I > 5 THEN ERROR(2) ELSE
                    FOR K:=0 TO I-1 DO
                     IVAL :=8*IVAL + DIGIT[K] ;
                     NEXTCH ;
            END   ELSE
            BEGIN  IF I>4
                    THEN ERROR(2) ELSE
                    FOR K:=0 TO I-1 DO
                         BEGIN IF IVAL≤ MAX10 THEN
                              IVAL:=10*IVAL + DIGIT[K]
                              ELSE BEGIN ERROR(2) ;
                                         IVAL:=0; END ;
                         END;
           END ;
  END      ELSE
  BEGIN ↱ SPECIAL CHARACTER ↓
       IF CH = ≡≡≡≡ ↱ CHAR CONSTANT↓ THEN
            BEGIN NU:=2; K:=0; BT1:=FALSE ;
                 REPEAT NEXTCH ;
                      IF CH=≡≡≡≡ THEN
                      BEGIN NEXTCH ; BT1:= CH≠≡≡≡≡ END ;
                      IF ¬BT1 THEN
                         IF K=ALFALENG THEN
                         BEGIN  ERROR(54) ;
                                BT1:= TRUE ;
                         END  ELSE
                         BEGIN K:=K+1; A[K] := CH  END ;
                 UNTIL BT1 ;
              CL:=2; TECH:=A[1]; IVAL:=ASCIICODE[TECH] ;
              IF K EQ 2 THEN
              BEGIN
                TECH:=A[2]; TEMP:=ASCIICODE[TECH];
                APPEND(IVAL,8,TEMP)
              END ELSE APPEND(IVAL,8,40B) ;
            END ↓ ≡ ↓  ELSE
            BEGIN NO:=SYMNO(CH) ;  CL:= SYMCL[CH] ;
                  IVAL := 0 ;
                  ↱ TEST FOR TWO CHARRACTER  SYMBOL ↓
                  IF CH= ≡:≡ THEN
```

```
                             BEGIN  NEXTCH ;
                                  IF CH= ≡=≡ THEN
                                      BEGIN NO:= 22 ; NEXTCH END ;
                             END  ELSE
                             IF CH= ≡.≡ THEN
                             BEGIN  NEXTCH ;
                                    IF CH= ≡.≡ THEN ↱:↓
                                    BEGIN NO:=21 ; NEXTCH  END;
                             END  ELSE
                             IF CH= ≡/≡ THEN
                             BEGIN NEXTCH ;
                                   IF CH = ≡*≡ THEN  ↱ SKIP COMMENT ↓
                                   BEGIN  NEXTCH ;
                                   3 :    WHILE CH≠ ≡*≡  DO NEXTCH ;
                                          NEXTCH ;  IF CH≠≡/≡ THEN
                                          GOTO 3 ELSE
                                          NEXTCH ;  GOTO 1 ;
                                   END ;
                             END  ELSE
                             NEXTCH ;
                   END ;
          END ; ↱ SPECIAL CHARACTER ↓
     END ;     ↱ INSYMBOL↓
↱*************************************************************************↓
```

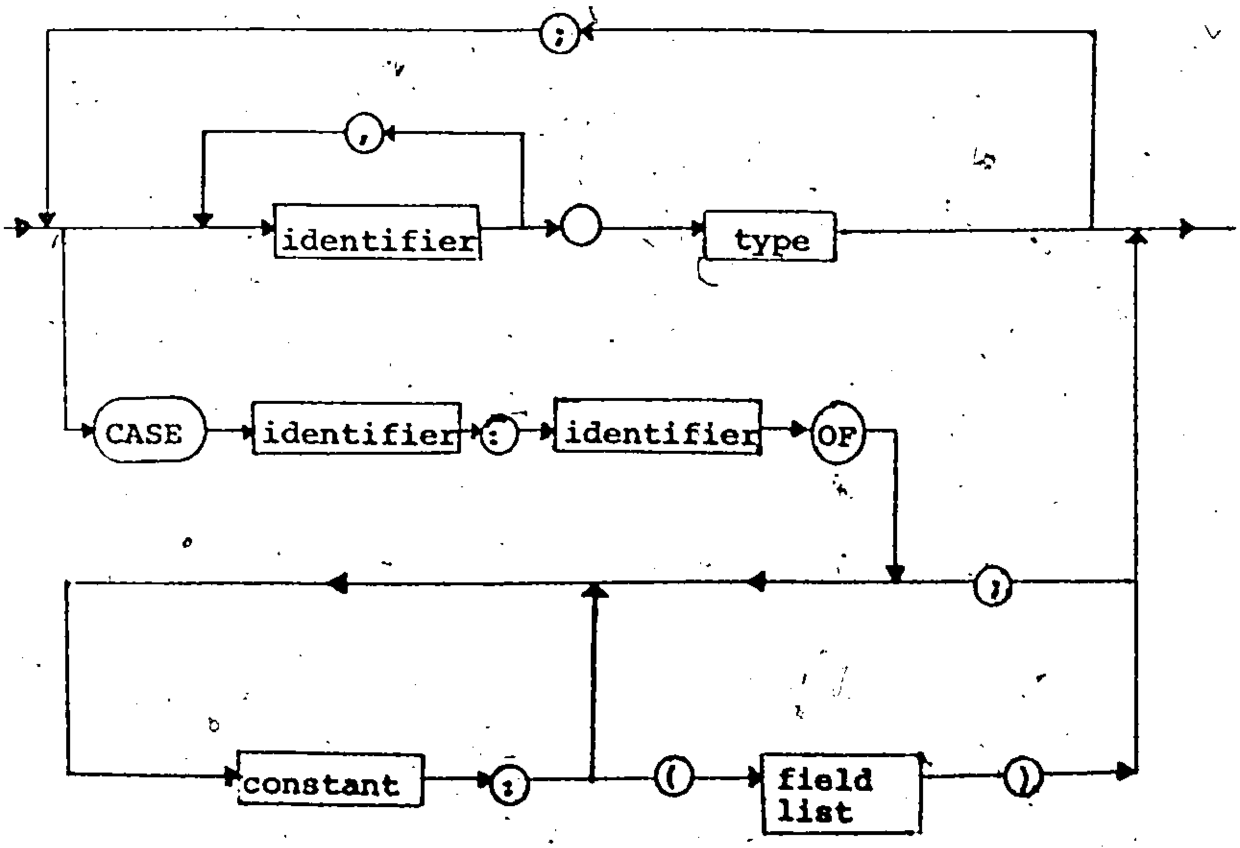
APPENDIX E

SYNTAX DIAGRAM OF PL/2100

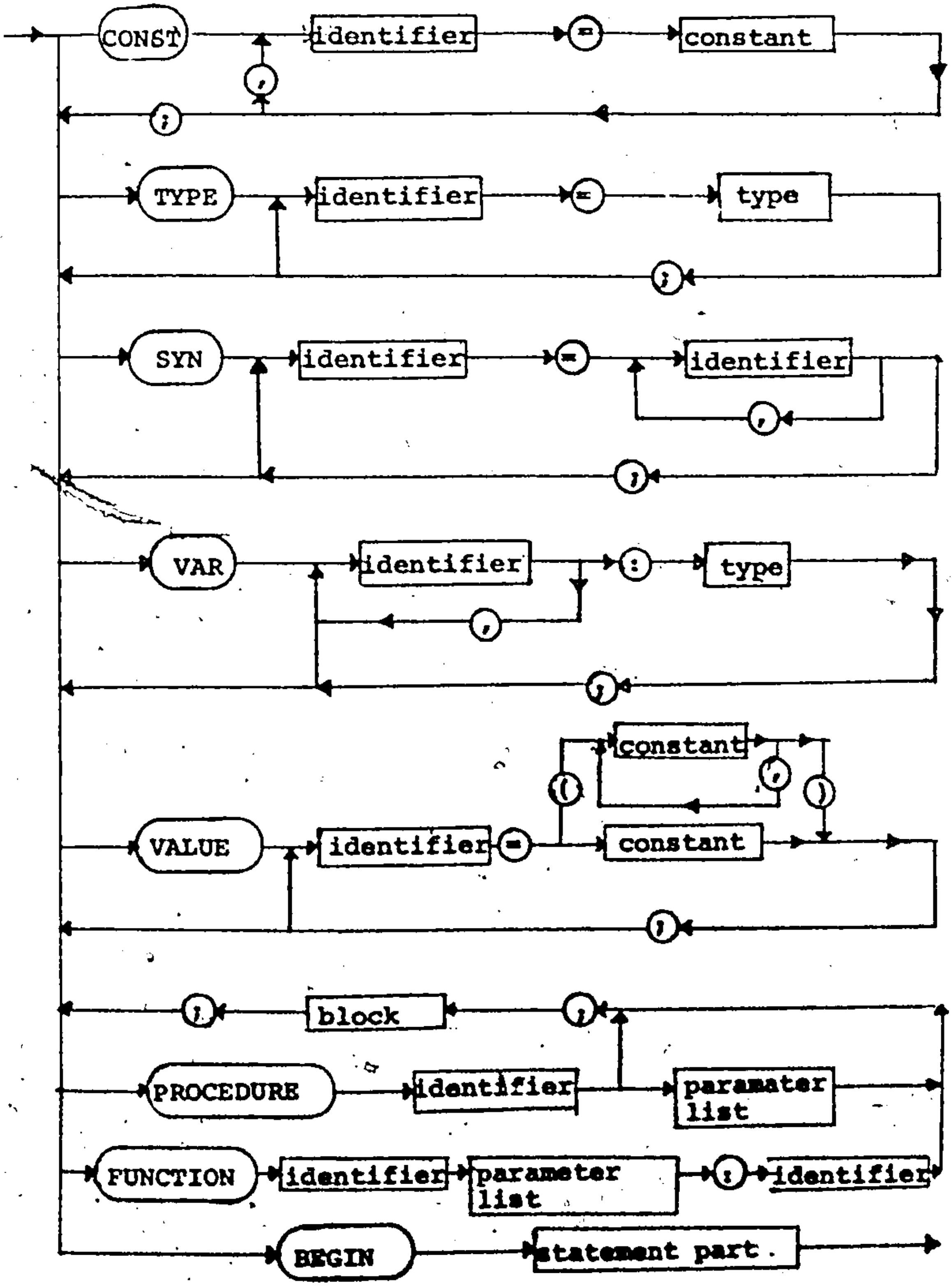
parameter list

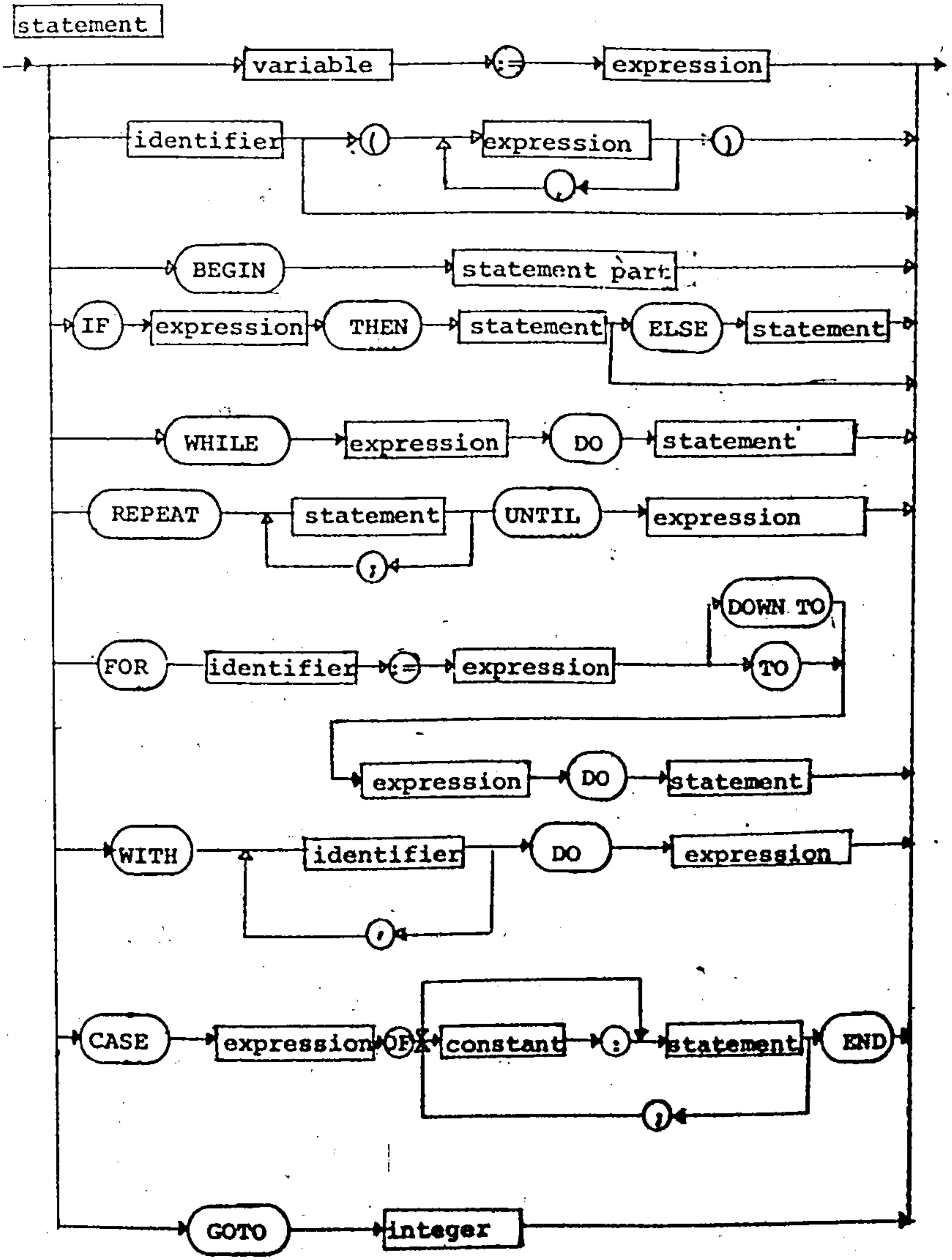


field list

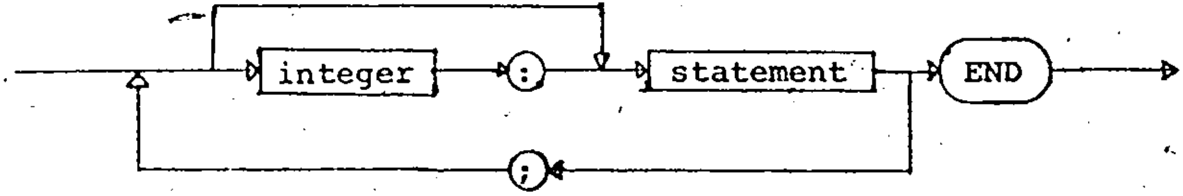

;
,
identifier
type
CASE
identifier
:
identifier
OF
;
constant
:
(
field list
)

block

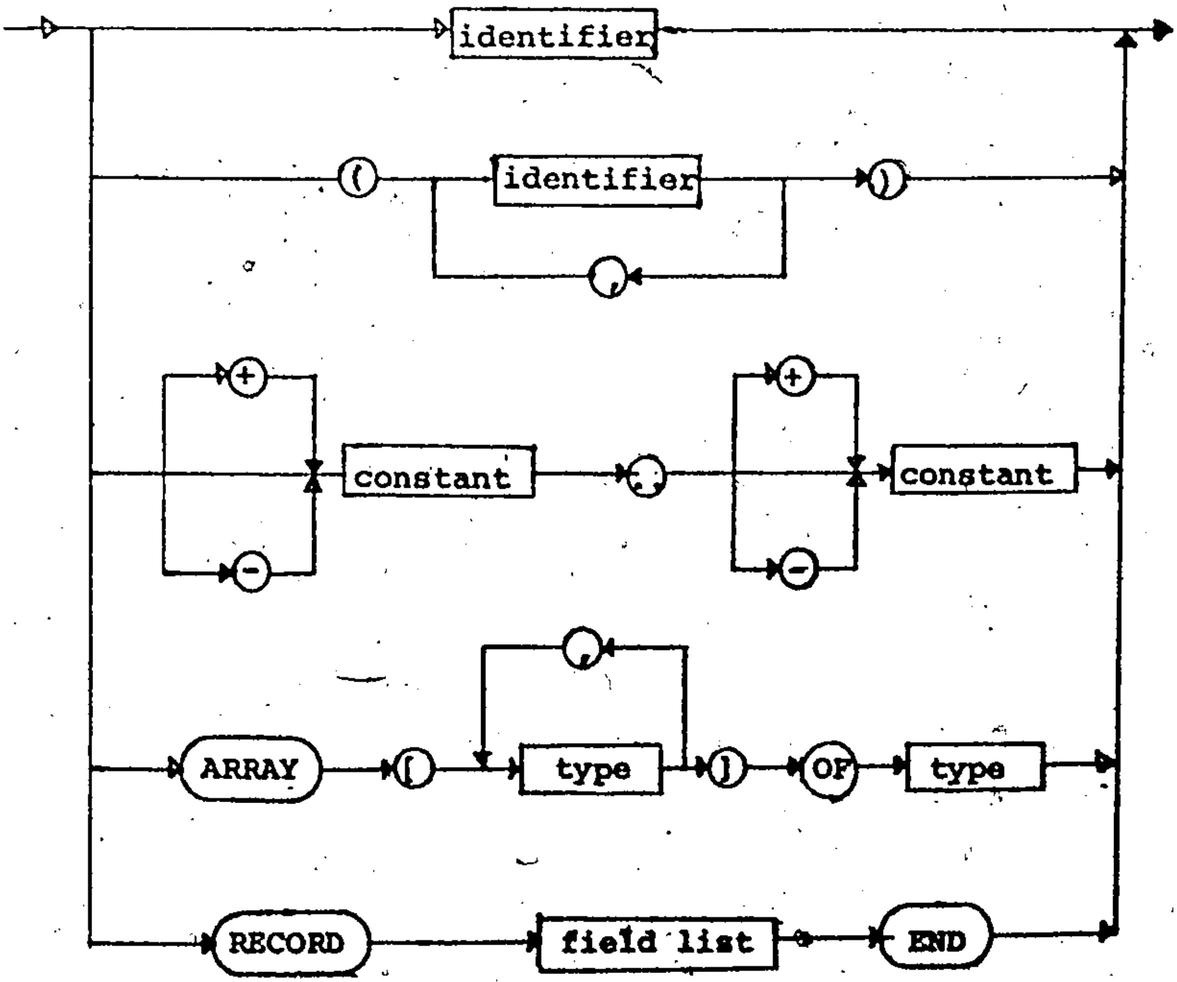

CONST
identifier
=
constant
,
;
TYPE
identifier
=
type
;
SYN
identifier
=
identifier
,
;
VAR
identifier
:
type
,
;
VALUE
identifier
=
(
constant
,
)
constant
;
;
block
;
PROCEDURE
identifier
paramater list
FUNCTION
identifier
parameter list
:
identifier
BEGIN
statement part.


statement
variable
:=
expression
identifier
(
expression
,
)
BEGIN
statement part
IF
expression
THEN
statement
ELSE
statement
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
identifier
:=
expression
DOWN TO
TO
expression
DO
statement
WITH
identifier
,
DO
expression
CASE
expression
OF
constant
:
statement
;
END
GOTO
integer

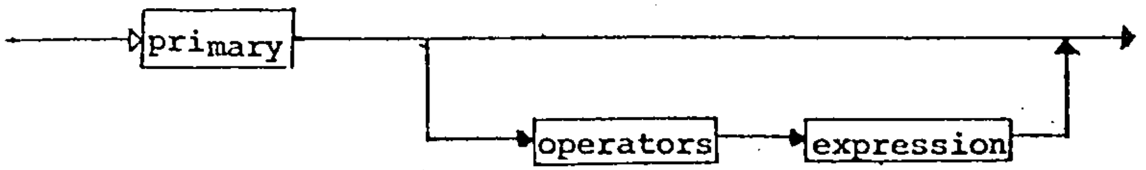
statement part



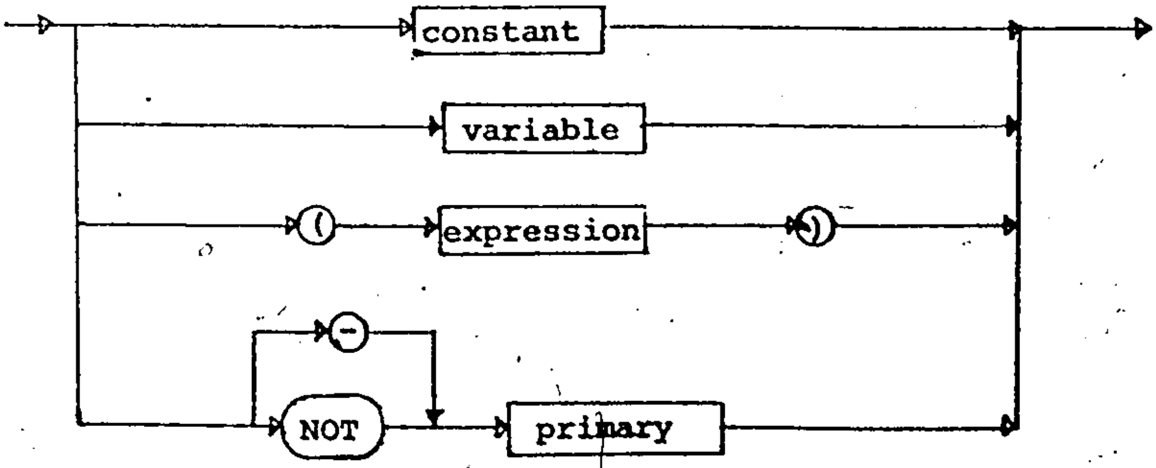
type



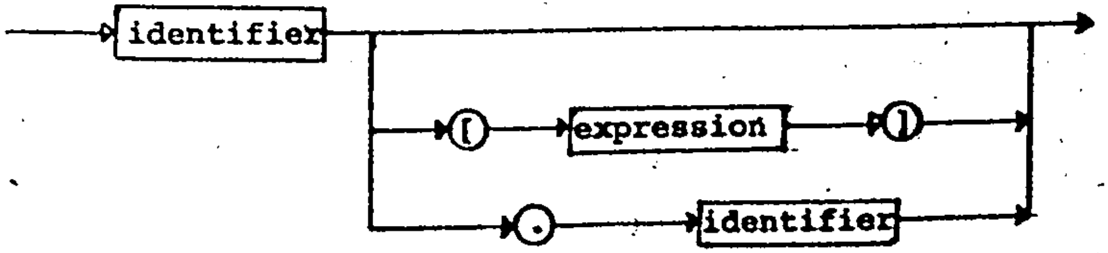
expression



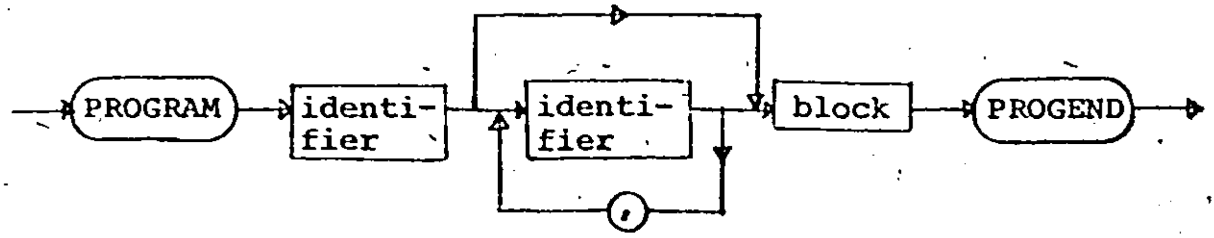
primary



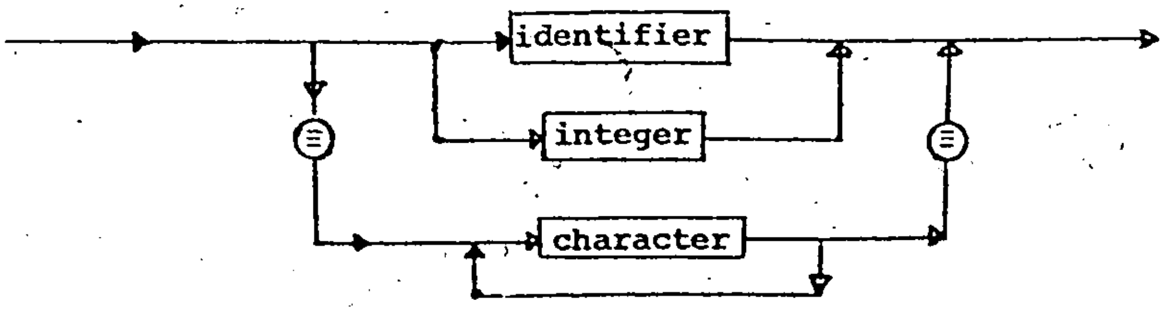
variable



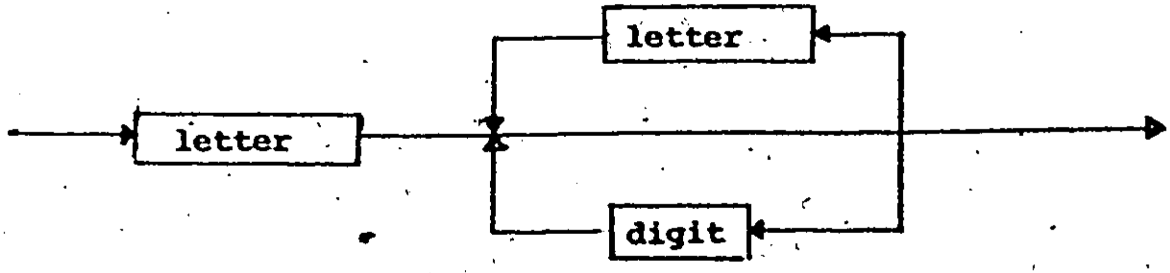
program



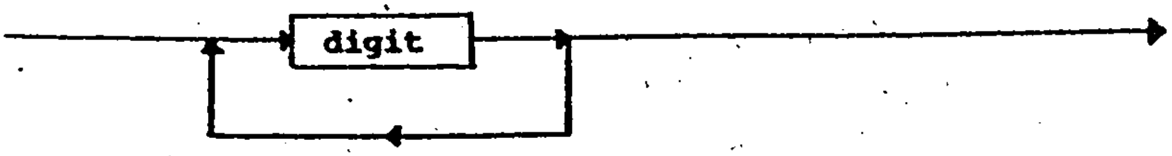
constant



identifier



integer



APPENDIX F

LISTING OF I/O ROUTINES

```
ASMB,L,R
            NAM  DR1,7
*                                                                 *
*           THIS ROUTINE GETS THE INPUT IN THE BUFFER             *
*           RDCON----CONWD                                        *
*           READQ----REQUEST CODE = 1                             *
*           BUFFS----BUFFER ADDRESS                               *
*           BUFLS----BUFFER LENGTH                                *
*           INPUT--------                                         *
*           B-REGISTER---ADDRESS OF THE TOP OF THE BUFFER         *
*                                                                 *
            EXT  EXEC
            ENT  GET
.           EQU  100B
OPFLG       EQU  .+113
GET         NOP
            STA  BUFFS            STORE BUFFER ADDRESS
            LDA  GET              GET THE
            INA                   LOGICAL UNIT
            LDB  0,I              NUMBER IN RDCON
            STB  RDCON
            JSB  EXSET
            JSB  EXEC
            DEF  *+5
            DEF  READQ
            DEF  RDCON
            DEF  BUFFS,I
            DEF  BUFLS
            JMP  GET,I
```

```
0030  EXSET    NOP
0031           JSB  EXEC
0032           DEF  *+2
0033           DEF  *+2
0034           JMP  EXSET,I
0035           DEC  -19
0036  READQ    DEC  1
0037  RDCON    BSS  1
0038  BUFFS    BSS  1
0039  BUFLS    OCT  72
0040           END  GET
**** LIST END ****
```

```
ASMB,L,R
              NAM  DR2,7
*                                                            *
*             THIS ROUTINE WRITES A VARIABLE LENGTH          *
*             BUFFER ONTO THE OUTPUT DEVICE                  *
*             INPUT-------                                   *
*                  A-REGISTER----BUFFER LENGTH               *
*                  B-REGISTER----BUFFER ADDRESS              *
*                                                            *
              EXT  EXEC
              ENT  PUT
              EQU  100B
OPFLG         EQU  .+113
PUT           NOP
              STA  BUFLS
*                                                            *
*             STORE BUFFER LENGTH                            *
*                                                            *
              STB  BUFFS
*                                                            *
*             STORE BUFFER ADDRESS                           *
*                                                            *
              LDA  PUT               GET THE
              INA                    LOGICAL UNIT
              LDB  0,I               NUMBER IN B-REGISTER
              STB  WRCON             STORE IN WRCON
              CLA
              CLB
              LDB  OPFLG
              JSB  EXSET             PREPARE EXEC FOR OUTPUT
              JSB  EXEC
              DEF  *+5               RETURN ADDRESS FROM EXEC
              DEF  WRITQ             WRITE CODE = 2
```

```
              DEF   WRCON           CONWD
              DEF   BUFFS,I         GET THE BUFFER ADDRESS
              DEF   BUFLS           GET BUFFER LENGTH
              JMP   PUT,I           RETURN TO CALLING PROGRAM
     EXSET    NOP
              JSB   EXEC
              DEF   *+2
              DEF   *+2
              JMP   EXSET,I
              DEC   -19
     WRITQ    DEC   2
     WRCON    BSS   1
     BUFFS    BSS   1
     BUFLS    BSS   1
              END   PUT
**** LIST END ****
```

```
ASMB,L,R
              NAM  DR3,7
              ENT  ALLOK
*                                                          *
              THIS ROUTINE GETS THE ASCII CHARACTER FROM INPUT   *
*             BUFFER AND CONVERTS IT INTO INTEGER IF NECESSARY   *
*             INPUT-----
*                  B-REGISTER--------BUFFER POINTER              *
*                  E-REGISTER--------ZERO IF INTEGER             *
*                         ONE OTHERWISE                          *
*             OUTPUT------                                       *
*             A-REGISTER--------CONTAINS INTEGER OR CHARACTER    *
*             B-REGISTER------------CONTAINS NEW BUFFER POINTER  *
*                                                                *
ALLOK         NOP
              STB  TEMPV
              LDA  BLANK
              STA  CHAR
              SEZ
              JMP  L4              CHARACTER INPUT
              CLA
              STA  SIGN
              STA  DIG
              LDA  CRSW
              STA  CRFG
L1            JSB  NTBLK           GET NON-BLANK CHARACTER
              CPA  COMMA
              JMP  L1              YES
              CPA  MINUS
              RSS                  YES
              JMP  L2
              LDB  MINUS
              STB  SIGN
```

```
              JSB  GETCR          GET NEXT CHARACTER
      L2      JSB  DIGIT          FORM THE DIGITS
              JSB  NTBLK
              CPA  COMMA
              JMP  L3             YES
              LDB  DIG            IF NOT COMMA
              BLS,BLS             MULTIPLY
              ADB  DIG            BY
              BLS                 10
              STB  DIG            STORE THE NUMBER IN DIG
              JMP  L2             GET MORE DIGIT
      L3      LDA  DIG
              LDB  SIGN
              SZB
              CMA,INA             IF NEGATIVE NUMBER,TAKE 2,S COMPLEMENT
              LDB  TEMPV
              JMP  ALLOK,I
      *       GETCR  GETS NEXT CHARACTER                      *
      GETCR   NOP
              LDB  CRFG           CRFG IS BOOLEAN FLAG
              SSB
              JMP  GETC2
              LDA  TEMPV,I
              AND  CH2
              ALF,ALF
      *       GET FIRST CHARACTER                             *
      GETC1   LDB  CRFG
              RBR
              STB  CRFG
              JMP  GETCR,I
```

```
*       GET NEXT SECOND CHARACTER                                *
GETC2   LDA  TEMPV,I
        AND  CH1
        LDB  TEMPV
        INB
        STB  TEMPV
        JMP  GETC1
*       NTBLK GET NON-BLANK CHARACTER                            *
NTBLK   NOP
NTBL1   JSB  GETCR
        CPA  BLANK
        RSS                       YES
        JMP  NTBLK,I
        JMP  NTBL1
*       DIGIT GETS THE DIGIT FROM ASCII CHARACTER                *
DIGIT   NOP
        LDB  SIXTY
        CMB,INB
        ADB  0
        ADB  DIG
        STB  DIG
        JMP  DIGIT,I
*       CHARACTER INPUT                                          *
L4      JSB  NTBLK
        CPA  COMMA
        JMP  L5
        STA  GHAR
L5      JSB  NTBLK
        CPA  COMMA
        JMP  L6
```

```
0094               STA  CHAR
0095               JMP  L5
0096  L6           LDA  CHAR
0097               LDB  TEMPV
0098               JMP  ALLOK,I
0099  CRSW         OCT  52525
0100  MINUS        OCT  55
0101  COMMA        OCT  54
0102  TEMPV        BSS  1
0103  SIGN         BSS  1
0104  DIG          BSS  1
0105  CRFG         BSS  1
0106  CHAR         BSS  1
0107  CH2          OCT  077400
0108  CH1          OCT  177
0109  BLANK        OCT  40
0110  SIXTY        OCT  60
0111               END  ALLOK
**** LIST END ****
```

```
ASMB,L,R
               NAM  DR4,7
               ENT  ALLOC
*                                                           *
*                                                           *
*                                                           *
*        THIS ROUTINE CONVERTS AN INTEGER INTO              *
*        ASCII CHARACTER AND PUT IT INTO THE                *
*        BUFFER. IF CHARACTER IS INPUT ,IT IS PUT  INTO     *
*        THE BUFFER STRAIGHT.                               *
*                                                           *
*                                                           *
*        INPUT........                                      *
*             A-REGISTER--------INTEGER OR CHARACTER        *
*                               TO BE WRITTEN OUT.          *
*             B-REGISTER--------BUFFER POINTER.             *
*             E-REGISTER--------IT IS ZERO FOR INTEGER      *
*                               ONE OTHERWISE.              *
*                                                           *
*        OUTPUT........                                     *
*             A-REGISTER--------BUFFER LENGTH               *
*             B-REGISTER--------NEW BUFFER POINTER          *
*                                                           *
*                                                           *
ALLOC          NOP
               STA  TEMPT    STORE INTEGER OR CHARACTER
               STB  TEMPA
               STB  TEMPA    STORE BUFFER POINTER
               CLB
               STB  BUFL     INITIALIZE BUFFER LENGTH
               STB  KOUNT    AND KOUNT TO ZERO
```

```
               SEZ
               JMP  LAB6     CHARACTER INPUT
               LDB  BLANK
               STB  SIGN
               SSA,RSS
               JMP  LAB1     SIGN OF INTEGER IS POSITIVE
               CMA,INA       WE WANT TO WRITE POSITIVE INTEGER
               STA  TEMPT
               LDB  F55      ALONG WITH ITS
               STB  SIGN     NEGATIVE SIGN
LAB1           LDA  TEMPT
               CLB           GET NEXT DIGIT
               DIV  TEN
               SZA,RSS
               JMP  LAB2     YES
               STA  TEMPT    GET THE ADDRESS OF A
               LDA  TEMP2    TEMPORARY LOCATION
               ADA  KOUNT
               ADB  SIXTY
               STB  0,I
*        INCREMENT THE COUNT OF THE DIGIT                          *
               LDA  KOUNT
               INA
               STA  KOUNT
               JMP  LAB1        GET NEXT DIGIT
LAB2           LDA  TEMP2
               ADA  KOUNT
               ADB  SIXTY
*        STORE LAST DIGIT IN TEMPORARY LOCATION                    *
               STB  0,I
*        STORE THE SIGN                                            *
               LDA  SIGN
```

```
LAB3           ALF,ALF
               STA  NUMB
               JSB  INBFL
               IOR  NUMB
*        PUT SIGN AND THE DIGITS IN THE BUFFER            *
*        DECREMENT THE KOUNT TO GET THE PROPER DIGITS     *
               JSB  PUTBL
               CCA
               ADA  KOUNT
               STA KOUNT
               SSA
               JMP  LAB5
               SZA,RSS
               JMP  LAB4
               JSB  INBFL
               CCB
               ADB  KOUNT
               STB  KOUNT
*        GET NEXT DIGIT TO BE PUT IN THE BUFFER           *
               JMP  LAB3
LAB4           JSB  INBFL
               ALF,ALF
               IOR  BLANK
               JSB  PUTBL
*        PUT TWO BLANKS AFTER EACH OUTPUT                 *
LAB5           LDA  BLANK
               ALF,ALF
               IOR  BLANK
               JSB  PUTBL
               JMP  ALLOC,I
*                                                         *
*        CHARACTER OUTPUT                                 *
```

```
*                                                                    *
LAB6         LDA  BLANK
             ALF,ALF
             IOR  BLANK
             JSB  PUTBL       STORE BLANK IN BUFFER
             LDA  TEMPT       LOAD THE CHARACTER
             JSB  PUTBL       STORE THE CHARACTER IN BUFFER
             JMP  LAB5
*                                                                    *
*         STORE THE CONTENTS OF A-REGISTER IN THE BUFFER             *
*         UPDATE BUFFER LENGTH AND BUFFER POINTER                    *
*                                                                    *
PUTBL        NOP
             LDB  BUFL
             ADB  TEMPA
             STA  1,I
             LDA  BUFL
             INA
             STA  BUFL
             JMP  PUTBL,I
*                                                                    *
*         INCREMENT THE COUNTER--KOUNT-- FOR THE DIGIT               *
*         ADJUST THE ADDRESS OF TEMPORARY LOCATION                   *
*                                                                    *
INBFL        NOP
             LDB  KOUNT
             ADB  TEMP2
             LDA  1,I
             JMP  INBFL,I
TEMP2        DEF  TEMP
TEMP         BSS  20
```

```
TEMPA          BSS  1
TEMPT          BSS  1
SIGN           BSS  1
KOUNT          BSS  1
BUFL           BSS  1
F55            OCT  55
SIXTY          OCT  60
TEN            DEC  10
BLANK          OCT  40
NUMB           BSS  1
               END  ALLOC
**** LIST END ****
```

APPENDIX G

LISTING OF THE HPCOM COMPILER

```
**************** . HPCOM . *****************
*                                          *
*   AUTHOR  : DILIP K. ROY.                *
*   LANGUAGE: PASCAL(DECEMBER,1972 VERSION)*
*   MACHINE : CDC-6400                     *
*   PLACE   : MCMASTER UNIVERSITY          *
*             HAMILTON,ONTARIO.            *
*   DATE    : JUNE12,1973                  *
*                                          *
********************************************
```

```
HPCOM IS WRITTEN FOR PROGRAMMING LANGUAGE 2100 (WRITTEN
FOR  THE MINICOMPUTER HP/2100A).  THE COMPILER PRODUCES
RELOCATABLE BINARY CODE SUITABLE FOR STANDARD RELOCATABLE
LOADER OF THE HP/2100A COMPUTER.
          A PL/2100 PROGRAM ALWAYS STARTS WITH A PROGRAM
CARD,NAMELY,
       PROGRAM  <PROGRAMNAME>  (,<OPTIONAL PARAMETERS>)
WHERE <OPTIONANAL PARAMETERS> ARE  :
       L :   WRITES OUT CODE STACK.
       B :   WRITES OUT THE RELOCATABLE CODE
       P :   PUNCHES OUT THE RELOCATABLE CODE ON THE
             FILE  **HPDIP**.

TO RUN HPCOM ,CONTROL CARDS NEEDED ARE :
   1 :    JOB CARD.
   2:     ATTACH(PASCAL,ID=GGQPASCAL,MR=1,CY=20)
   3 :    PASCAL.
   4 :    REWIND(HPDIP)
   5 :    REQUEST(HPHP). HEWLETT PACKARD CARDS NEEDED.
   6 :    DISPOSE(HPDIP,P8=CAB)
             NOTE : LAST THREE CARDS ARE NEEDED
                    TO PUNCH OUT THE COMPILED
                    CODE ONTO HP CARDS.
TO RUN A PL/2100 PROGRAM COMMANDS NEEDED ARE :
   1:     :STORE--DIRECTIVE TO STORE THE RELOCATABLE
          PROGRAM AND LATER LOAD IT .
          OR USE :PROG,LOADR,--DIRECTIVE TO LOAD THE
          PROGRAM DIRECTLY.
   2:     :RUN,PROGRAMNAME---TO RUN THE PROGRAM.
```

```
◆$A◆,X◆◆
            CONST
                  CODMAX = 1000,
                  KPMAX=50,
                  MAXLEVEL = 10 ,
                  JMPMAX=49,
                  MAXEXLABS=10,
                  UNUMAX=100,
                  CSTMAX = 40 ,
                  FILLIMIT = 10 ,
                  WORDLENGTH = 16 ,
                  DISPLIMIT = 20,
                  ALFALENG = 10 ,
                  MAXLAB = 10 ,
                  MAX10 = 77777B ;
```

```
TYPE
      BITS= 0..1 ;
      RTYP = 0..5 ;
      BITRANGE = 0..16 ;
      RG3 = 0..3   ;
      ADDRESS = 0.. 1777B ;
      SHRINT = -1777B .. +1777B ;
      ARRINT3 = ARRAY [1..3] OF INTEGER ;
      AR = ARRAY [1..10] OF CHAR   ;
      DELARRY = ARRAY [1..54,1..10] OF CHAR ;
      INTARR = ARRAY [-1..39,1..10] OF CHAR ;
      OPTPWR = ( NOOPT,PUREP,POSP,NEGP) ;
      IDKLASS = (TYPES,KONST,PROC,VARS,FIELD,TAGFIELD,
                 DUMMYCLASS) ;
      TYPFORM = (NUMERIC,SYMBOLIC,ARRAYS,RECORDS,FILES,
                 REGISTERS) ;
      IDKINDS = (ACTUAL,FORMAL) ;
      WHERE = (BLOCK,CWITH,VWITH) ;
      ATTRKIND = (VARBL,SVAL,LVAL,LCOND) ;
      ARCODE = ARRAY [1..CODMAX] OF
               RECORD
               LOC,INST,ADRR : INTEGER; TYP:RTYP; MREF:BOOLEAN;
               END;
      ATTR = RECORD
               TYPTR : INTEGER ;
               CASE KIND : ATTRKIND OF
                 VARBL : (ACCESS:(DRCT,INDRCT,INXD);
                          BREG:RG3;DPLMT:INTEGER;
                          CASE PCKD : BOOLEAN OF
                            FALSE : ;
                            TRUE : (BITADR,BITSZ:BITRANGE));
                 SVAL : (VAL: INTEGER);
                 LVAL : (CTERM : INTEGER) ;
                 LCOND : (JMP: 0..3; ARITH:BOOLEAN)
             END;
```

```
VAR
     HPAS : ARCODE ;  ↱ CODE STACK ↓
     HCA : INTEGER ; ↱ USED TO INDEX CODE STACK ↓
     BFLC,BLC : INTEGER ;
     ↱ BFLC--IS THE LOCATION WHERE BUFFER ADDRESS IS STORED.
       BLC-- NUMBER OF RESERVED LOCATIONS AT THE TOP OF THE
             DATA STACK . ↓
     HPDIP[OUT] : FILE OF INTEGER ;
     ↱ FILE OF RELOCATABLE CODE TO BE PUNCHED OUT ↓
     ↱ USED BY DIFFERENT ROUTINES USED FOR THE RELOCATION
       OF THE CODE STACK . ↓
     PRNAM: AR ;
     W : ARRAY [1..60] OF INTEGER ;
     TRNSFR : ARRINT3 ;
     CERR,LISTBIN,PUNCHBIN,RLISTBIN : BOOLEAN ;
     ↱ OPTIONAL PARAMETERS ARE USED ALOG WITH THESE VARIABLES
       THESE VARIABLES ARE SET TRUE IF OPTIONAL PARAMETERS
       ARE PRESENT.    ↓

     XPTR,EXSYM : INTEGER ; ↱ USED FOR EXTERNALS ↓
     EXTX : ARRAY[1..30] OF
              RECORD XNAM :AR; XSYM:INTEGER   END ;
     ↱ EXTX IS USED TO STORE THE NAME OF THE EXTERNALS ↓

     LABTAB : ARRAY [1..MAXLAB] OF
           PACKED RECORD LABVAL,FLD2,FLD3 : INTEGER END ;
     CLABIX : INTEGER ;
     ↱ ALL LABELS MET SO FAR IN THE BODY OF THE PROCEDURE
     ACTUALLY BEING COMPILED,TOGETHER WITH INFORMATION WHETHER
     LABEL DEF. ([LABEL]:) ALREADY FOUND OR NOT (DLD2) .
     IN THE FORMER CASE FLD3 CONTAINS THE CORRESPONDING
     ADDRESS,WHERE IN THE LATTER CASE FLD3 CONTAINS AN INDEX
     INTO UNDLAB WHERE THE OCCURENCE ARE CHAINED  ↓

     STORE : ARRAY [1..KPMAX] OF
             RECORD
                STRP: INTEGER; STPL : INTEGER;
             END ;
     STRE : ARRAY [1..10] OF
             RECORD
                STOP :  INTEGER;  STCL: INTEGER   ;
             END ;
     RPMAX,RP,KP : INTEGER ; ↱ USED WITH STORE AND STRE ↓
     ↱ STORE AND STRE ARE USED BY ROUTINES STTMP AND LDTMP.
       THEY STORE COMPILE TIME INFORMATION TO BE USED TO
       UPDATE THE CODE STACK . ↓
```

```
ERRLIST : ARRAY [1..10] OF
          PACKED
          RECORD  POS,NMR : SHRINT END ;
ERRNRS : ARRAY [0..3] OF SET OF  0..3 ;
ERRINX , POS1 : SHRINT ;
ERRORLIST : ARRAY[1..10] OF INTEGER ;
ERRORKOUNT : INTEGER ;
ERR : BOOLEAN ;
CHCNT : INTEGER ;
↱ ERRLIST CONTAINS THE POSITION AND NUMBERS OF THE
  ERROR ON ONE LINE AND ;
  ERRINX = TOP OF ERRLIST,
  POS1 = POSITION OF LAST PRINTED ERRORMARK(↑),
  CHCNT = POSITION OF LAST READ CHARACTER,
  ERRLIST,ERRINX,POS1,CHCNT,EOLFLAG---- ALL ARE
  USED BY NEXTCH AND ERROR ;
  ERRORKOUNT --KEEPS TRACK OF THE NUMBER OF ERRORS
                 IN A PROGRAM.
ERRORLIST : KEEPS THE ERROR NUMBER .
  ERR---- IS SET WHENEVER ERROR HAS BEEN CALLED AND IS
          TESTED (AND AND RESET ) BY SEVERAL PROCEDURE ↓

JMPTAB : ARRAY [0..JMPMAX] OF INTEGER ;
JMPIX : SHRINT ;
↱ TRANSFER VECTOR FOR CALLS OF FORWORD DECLARED
   PROCEDURES AND GOTO STATEMENTS LEADING OUT OF PRO  ; ↓

CSTTB : ARRAY [1..CSTMAX] OF
        RECORD  VALU:INTEGER; INX:SHRINT END;
LCX : SHRINT ;
↱ CONTAINS ALL CONSTANTS C, OCCURING IN THE
  PROCEDURES ACTUALLY BEING COMPILED TOGETHER
  AN INDEX INTO UNDLAB WHERE THEIR OCCURENCE IN
  THE CODE OF THIS PROCEDURE ARE CHAINED ↓

EXTAB : ARRAY [1..MAXEXLABS] OF
        PACKED
        RECORD  EXVAL,JMPBIX : SHRINT END;
CXTABIX : SHRINT ;
↱ CONTAINS THE EXPLICITLY DECLARED LABELS OF ALL
  PROCEDURES NOT YET CLOSED , TOGETHER WITH THEIR
  CORRESPONDING INDEX INTO JMPTAB ↓

UNDLAB : ARRAY [1..UNDMAX] OF
         RECORD SUCC,PLACE:INTEGER END;
```

```
CHINX : SHRINT ;
↱ ACTS AS A LIST STRUCTURE , CHAINING
  OCCURENCE OF CONSTANTS AND JUMPS TO NOT  YET
  REACHED LABELS IN THE CODE OF THE PROCEDURES
  ACTUALLY BEING COMPILED ,
  CHINX= HEAD OF FREE LIST ↓

↱ FOLLOWING VARIABLES ARE USED BY PROCEDURES NEXTCH
  AND  INSYMBOL.
  NO,CL ARE THE TOKEN INTEGER ASSOCIATED WITH  IDENTIFIER
  AND RESERVED SYMBOLS OF THE LANGUAGE PL/2100. ↓
↱ LC,IC--LOCATION AND INSTRUCTION COUNTER ↓
LC,IC : INTEGER ;
DP,BI1,EOLFLAG : BOOLEAN ;
CH,TECH : CHAR ;
NO,CL : INTEGER ;
IVAL : INTEGER ;
A,AVAL : AR ;
DIGITS : SET OF   CHAR ;

↱ CONSTANT POINTERS ↓
INTPTR,CHARPTR,BOOLPTR,PREDEFP,NILPTR,UNOCLPTR : SHRINT ;
BYTPTR,WROPTR,PREDEFPTR,UNDECPTR : SHRINT ;
↱ USED BY VARIABLE AND SYNONYM DECLARATION PART ↓
↱ ALSO BY TYPE DECLARATION PART ↓
LL,LL1,VLC: INTEGER ;
AT,IT1 : INTEGER ;
P,N,Q,SYNT : SHRINT ;
FILTOP : INTEGER ;
TYPID : AR ;
TL : INTEGER ;
SURKPTR,PROCPTR,OLDLEV : INTEGER ;
↱ USED BY VALUE DECLARATION PART ↓
IT,PT : INTEGER ;
VALPT : BOOLEAN ;
VARS : ARRAY [1..100] OF INTEGER ;
↱ THESE ARE USED TO KEEP THE VARIABLE ATTRIBUTES ↓
GATTR,AATTR,LATTR,RATTR : ATTR ;
COMPAR,COMPARE : BOOLEAN ; ↱USED BY SEARCH ROUTINES↓
↱ USED BY EXPRESSION ROUTINES ↓
CTY,STY : BOOLEAN ;
OP,LADOPCL : INTEGER ;
WRFL : BOOLEAN ;  ↱ USED BY READ AND WRITE ROUTINES ↓
↱ USED FOR TEMPORARIES BY VARIOUS ROUTINES ↓
I,J,K,M,TEM,TEMP: INTEGER ;
CEXTABIX,CHNIX: INTEGER ;
```

```
FSTIX : INTEGER ;
↱ VARIABLES POINTING TO CONTEXTTABLE ↓
↱ AUXILIARY VARIABLES ,USED IN MAIN PROG. AND SEVERAL
 PROCEDURES ↓
INDEX : SHRINT ;
NEXT,CTPTR,MAXCTP : SHRINT ;
DISPLAY : ARRAY [0..DISPLIMIT] OF
       RECORD FNAME:INTEGER ;
             CASE OCCUR : WHERE OF
                   BLOCK: ;
                   CWITH : (CDISP:INTEGER; CLEV:RG3) ;
                   VWITH : ( VDSPL : INTEGER) ;
       END ;
TOP,DISX : INTEGER ;
LEVEL : INTEGER ;
↱ CONTEXTTABLE CONTAINS THE IDENTIFIERSALOG WITH
  THIER ATTRIBUTES                     ↓
```

```
CONTEXTTABLE : ARRAY[0..250] OF
          PACKED
    RECORD
      NAME : AR ; NXTEL : SHRINT ;
                 SYNCELL : BOOLEAN ;
       CASE KLASS : IDKLASS OF
         TYPES : (SIZE : ADDRESS ;
                    CASE FORM : TYPFORM OF
                     NUMERIC:(BITS:BITRANGE;MIN,MAX:INTEGER);
                    SYMBOLIC:(FCONST:INTEGER;BITSIZE:BITRANGE);
                    ARRAYS:(AELTYPE,INXTYPE:SHRINT;
                           LO,HI:SHRINT; SZE:BOOLEAN;OPTTYP:OPTPWR;
                           EXP1,EXP2:BITRANGE) ;
                    RECORDS : (FSTFLD,RECVAR : INTEGER ) ;
                     FILES : ;
                     REGISTERS : ) ;
         KONST : (CONTYPE:INTEGER ;
                CASE CUNKIND : IDKINDS OF
                ACTUAL :(SUCC:INTEGER;VALUES:INTEGER);
                FORMAL : (GADDR:ADDRESS;GLEVEL:RG3)) ;
         PROC : (PROCTYPE,FORMALS : INTEGER;
                 PROCKIND : IDKINDS ;
                 PROCADUR:ADDRESS ; PROCLEVEL : RG3;
                  SEGSIZE:INTEGER ) ;
         VARS  : (VTYPE:INTEGER; VKIND:IDKINDS;
                      SYNPTR : SHRINT ;
                  VADDR: ADDRESS ; VLEVEL : RG3 ) ;
         FIELD : (FLDTYPE : INTEGER ; FLDADDR : ADDRESS ;
                  BITDISPL , BITWIDTH : BITRANGE) ;
         TAGFIELD : (CASESIZE : INTEGER ; VARIANTS : INTEGER ;
                  CASE TAGVAD : BOOLEAN OF
                  FALSE : (CASETYPE : INTEGER) ;
                  TRUE : (CASEVAL : INTEGER )    ) ;
    END ;
```

```
PROG--- CONTAINS   PROGRAM
PROGEND--- CONTAINS PROGEND
WU--- INITIALIZE TO ALL THE RESERVED WORDS OF PL/2100
WNO,WCL-- CONTAINS INTEGER TOKENS NO AND CL
SYMNO,SYMCL--TOKENS NO AND CL FOR BCD CHARACTER SET
             OF THE RESERVED WORDS OF PL/2100
INITNAM---USED TO INITIALIZE THE CONTEXTTABLE
ASCIICODE--- ASCII  CODE CORRESPONDING TO THEBCD CHARS
DD--USED IN ROUTINES FOR RELOCATION

PROG : AR ;
PROGEND : AR ;
BLANCK : AR ;
BLANK : CHAR ;
SPLITSTAT : ARRAY [1..52] OF INTEGER ;
ERRCL,TERRCL : ARRAY [1..52] OF INTEGER ;
WD : DELARRY ;
WNO,WCL : ARRAY [1..54] OF INTEGER ;
WL : ARRAY[1..11] OF INTEGER ;
SYMNO,SYMCL : ARRAY[≡+≡ .. ≡;≡ ] OF INTEGER ;
INITNAM : INTARR ;
ASCIICODE : ARRAY [CHAR] OF INTEGER ;
DD : ARRINT3 ;
```

```
VALUE
    PROG = ('P','R','O','G','R','A','M',' ',' ',' ') ;
    PROGEND = ('P','R','O','G','E','N','D',' ',' ',' ');
    BLANCK = (' ',' ',' ',' ',' ',' ',' ',' ',' ',' ')      ;
    BLANK = ' ' ;
    SPLITSTAT = (2,1,22*1,3,1,4,2*1,5,1,6,1,7,1,8,1,9,7*1,2,
                 2,1,10,3*1) ;
    ERRCL = (25*0,1,2,1,0,2,1,0,2,2,1,0,1,0,
             1,0,2,0,2,2,0,2,2,5*0) ;
    TERRCL = (10*0,1,2,0,2,0,2,3*0,1,2,3*0,3*2,0,0,2,0,
              2,0,2,0,2,0,2,0,2,0,2,2,0,2,2,4*0,1,2) ;
   WD = ( 'I','F',' ',' ',' ',' ',' ',' ',' ',' ',
          'D','O',' ',' ',' ',' ',' ',' ',' ',' ',
          'T','O',' ',' ',' ',' ',' ',' ',' ',' ',
          'O','F',' ',' ',' ',' ',' ',' ',' ',' ',
          'O','R',' ',' ',' ',' ',' ',' ',' ',' ',
          'L','T',' ',' ',' ',' ',' ',' ',' ',' ',
          'L','E',' ',' ',' ',' ',' ',' ',' ',' ',
          'G','T',' ',' ',' ',' ',' ',' ',' ',' ',
          'G','E',' ',' ',' ',' ',' ',' ',' ',' ',
          'N','E',' ',' ',' ',' ',' ',' ',' ',' ',
          'E','Q',' ',' ',' ',' ',' ',' ',' ',' ',
          'A','N','D',' ',' ',' ',' ',' ',' ',' ',
          'E','N','D',' ',' ',' ',' ',' ',' ',' ',
          'N','I','L',' ',' ',' ',' ',' ',' ',' ',
          'F','O','R',' ',' ',' ',' ',' ',' ',' ',
          'D','I','V',' ',' ',' ',' ',' ',' ',' ',
          'M','O','D',' ',' ',' ',' ',' ',' ',' ',
          'V','A','R',' ',' ',' ',' ',' ',' ',' ',
          'S','Y','N',' ',' ',' ',' ',' ',' ',' ',
          'A','L','S',' ',' ',' ',' ',' ',' ',' ',
          'B','L','S',' ',' ',' ',' ',' ',' ',' ',
          'A','R','S',' ',' ',' ',' ',' ',' ',' ',
          'B','R','S',' ',' ',' ',' ',' ',' ',' ',
          'R','A','R',' ',' ',' ',' ',' ',' ',' ',
          'R','B','R',' ',' ',' ',' ',' ',' ',' ',
          'R','A','L',' ',' ',' ',' ',' ',' ',' ',
          'R','B','L',' ',' ',' ',' ',' ',' ',' ',
          'A','L','F',' ',' ',' ',' ',' ',' ',' ',
          'B','L','F',' ',' ',' ',' ',' ',' ',' ',
          'I','O','R',' ',' ',' ',' ',' ',' ',' ',
          'X','O','R',' ',' ',' ',' ',' ',' ',' ',
          'R','E','G','A',' ',' ',' ',' ',' ',' ',
          'R','E','G','B',' ',' ',' ',' ',' ',' ',
          'T','H','E','N',' ',' ',' ',' ',' ',' ',
          'E','L','S','E',' ',' ',' ',' ',' ',' ',
```

```
'G','O','T','O',' ',' ',' ',' ',' ',' ',
'C','A','S','E',' ',' ',' ',' ',' ',' ',
'W','I','T','H',' ',' ',' ',' ',' ',' ',
'T','Y','P','E',' ',' ',' ',' ',' ',' ',
'F','I','L','E',' ',' ',' ',' ',' ',' ',
'B','E','G','I','N',' ',' ',' ',' ',' ',
'U','N','T','I','L',' ',' ',' ',' ',' ',
'W','H','I','L','E',' ',' ',' ',' ',' ',
'A','R','R','A','Y',' ',' ',' ',' ',' ',
'V','A','L','U','E',' ',' ',' ',' ',' ',
'C','O','N','S','T',' ',' ',' ',' ',' ',
'L','A','B','E','L',' ',' ',' ',' ',' ',
'P','A','C','K','E','D',' ',' ',' ',' ',
'R','E','P','E','A','T',' ',' ',' ',' ',
'D','O','W','N','T','O',' ',' ',' ',' ',
'R','E','C','O','R','D',' ',' ',' ',' ',
'F','U','N','C','T','I','O','N',' ',' ',
'R','E','G','I','S','T','E','R',' ',' ',
'P','R','O','C','E','D','U','R','E',' ') ;
HNO = (27,35,37,31,7,6*8,6,26,39,36,6,6,45,51,12*24,
      52,52,28,29,38,30,49,40,41,25,33,34,41,48,43,42,
      53,32,37,41,46,50,47) ;
HCL = (2*0,1,0,3,1,2,4,3,5,6,3,3*0,4,5,0,0,1,2,
      3,4,5,6,7,8,9,10,11,12,1,2,
      6*0,3,0,0,0,1,5*0,2,2,0,0,0) ;
WL = (1,1,12,32,41,48,52,52,54,55,55) ;
SYMNO = (7,7,6,6,9,10,0,8,0,15,17,0,11,12,21,8,13,
        7,6,18,14,8,8,8,8,5,16) ;
SYMCL = (1,2,1,2,0,0,1,6,3,0,0,2,0,0,0,
        5,0,3,3,0,0,1,4,2,3,1,0 ) ;
INITNAM =  ('T','E','X','T',' ',' ',' ',' ',' ',' ',
            'H','E','O','R',' ',' ',' ',' ',' ',' ',
            'G','E','T',' ',' ',' ',' ',' ',' ',' ',
            'P','U','T',' ',' ',' ',' ',' ',' ',' ',
            'A','L','L','O','C',' ',' ',' ',' ',' ',
            'A','L','L','O','K',' ',' ',' ',' ',' ',
            'E','X','E','C',' ',' ',' ',' ',' ',' ',
            'U','N','P','A','C','K',' ',' ',' ',' ',
            'I','N','S','E','R','T',' ',' ',' ',' ',
            'A','P','P','E','N','D',' ',' ',' ',' ',
            'R','E','A','D',' ',' ',' ',' ',' ',' ',
            'W','R','I','T','E',' ',' ',' ',' ',' ',
            'O','O','D',' ',' ',' ',' ',' ',' ',' ',
            'I','N','T',' ',' ',' ',' ',' ',' ',' ',
            'C','H','R',' ',' ',' ',' ',' ',' ',' ',
            'E','O','F',' ',' ',' ',' ',' ',' ',' ',
```

```
ASTPSEAIOARCUFIIMSCELSBWA
BQPRUOLNULEHOARNEIOXNQYOR
SRUECLFPTFAAOLUTMNSP RTRC
  NUC AUPALRLSEE     TEUY
  C   LTU   EE G        A
      E T   A  E        N
      N     N  K
      G
ASCIICODE = (000B,101B,102B,103B,104B,105B,106B,107B,
             110B,111B,112B,113B,114B,115B,116B,117B,
             120B,121B,122B,123B,124B,125B,126B,127B,
             130B,131B,132B,060B,061B,062B,063B,064B,
             065B,066B,067B,070B,071B,053B,055B,052B,
             057B,050B,051B,044B,075B,040B,054B,056B,
             042B,133B,135B,072B,047B,043B,100B,046B,
             136B,041B,074B,076B,134B,077B,045B,073B) ;
DD = (20B,400B,10000B) ;
```

```
*****************************************
RESERVED WORD AND SYMBOLS USED IN  HP-2100
     NO   CL    SYMBOL           NUMBERS
      1         ID
      2    1    INTEGER          CONST

      2    2    CHAR             CONST
      5    1    ¬
      5    1    NOT
      6    1    *
      6    2    /
      6    3    ∧
      6    3    AND
      6    4    DIV
      6    5    MOD
      7    1    +
      7    2    -
      7    3    ∨
      7    3    OR
      8    1    <
      8    1    LT
      8    2    ≤
      8    2    LE
      8    3    ≥
      8    3    GE
      8    4    >
      8    4    GT
      8    5    ≠
      8    5    NE
      8    6    =
      8    6    EQ
      8    7    IN
      9         (
     10         )
     11         [
     12         ]
     15         ,
     16         ;
     17         .
     18         ↑
     21         :
     22         :=
     23         /*
     24    1    ALS
     24    2    BLS
     24    3    ARS
```

```
24   4   BRS
24   5   RAR
24   6   RBR
24   7   RAL
24   8   RBL
24   9   ALF
24  10   BLF
24  11   IOR
24  12   XOR
24
25       BEGIN
26       END
27       IF
28       THEN
29       ELSE
30       CASE
31       OF
32       REPEAT
33       UNTIL
34       WHILE
35       DO
36       FOR
37   1   TO
37   2   DOWNTO
38       GOTO
39       NIL
40       TYPE
41   1   ARRAY
41   2   RECORD
41   3   FILE

42       LABEL
43       CONST
45       VAR
46       FUNCTION
47       PROCEDURE
48       VALUE
49       WITH
50       REGISTER
51       SYN
52   1   REGA
24   2   REGB
*******************************************
```

```
↗**********************************************************↓
          PROCEDURE LINES (N: INTEGER) ;
            VAR I: INTEGER ;
            BEGIN
              FOR I:=1 TO N DO WRITE(EOL,≡ ≡)
            END ↗LINES↓ ;
↗**********************************************************↓
          PROCEDURE OUTCH(C:CHAR) ;
            BEGIN OUTPUT↑ := C ; PUT(OUTPUT) END ;
↗**********************************************************↓
          PROCEDURE OUTO(VAL:INTEGER) ;
            ↗PRINT VAL OCTAL,ONE BLANK AND 6 DIGITS↓
            VAR K,M : INTEGER ; J,S : INTEGER ;
             BEGIN
                IF VAL=0 THEN FOR J:=1 TO 6 DO OUTCH(≡0≡) ELSE
                BEGIN
                  M:=-7B; S:=45;
                  FOR J:=15 TO 20 DO
                  BEGIN K:=-VAL ; APPEND(K,S,M) ;
                        OUTCH(CHR(ORD(≡0≡)-K)) ;
                        S:=S+3 ;
                  END ;
                END;
            END ; ↗OUTO↓
↗**********************************************************↓
```

```
PROCEDURE COMP(VAR A,B:AR; VAR COMPAR:BOOLEAN) ;
   ↑ COMPARE TWO ARRAYS A AND B, COMPAR IS TRUE IF THEY ARE
     EQUAL, OTHERWISE IT IS FALSE. ↓
   VAR  K: INTEGER ;
  BEGIN
   K:= 1 ;
 1: IF A[K]=B[K]  THEN
                  BEGIN  K:= K+1 ;
                     IF K≤10 THEN GOTO 1
                             ELSE COMPAR := TRUE ;
                  END
                  ELSE COMPAR := FALSE ;
   END ↑ COMP ↓ ;
*************************************************************************↓
   PROCEDURE CPARE(VAR A:AR; VAR B:DELARRY; M: INTEGER ;
                                  VAR COMPARE:BOOLEAN) ;
   ↑ COMPARE TWO ARRAYS A AND B, COMPARE IS TRUE IF THEY ARE
     EQUAL, OTHERWISE IT IS FALSE. ↓
 VAR
  K: INTEGER;
  BEGIN
    K:=1;
 1 :  IF A[K] = B[M,K] THEN
                    BEGIN
                      K:=K+1;
                      IF K≤10 THEN GOTO 1
                              ELSE COMPARE := TRUE;
                    END
                    ELSE  COMPARE := FALSE ;
  END;    ↑CPARE↓
*************************************************************************↓
```

```
FUNCTION LOG2(VAL1 : INTEGER) : INTEGER ;
   VAR E: INTEGER ; VAL: INTEGER ;
   BEGIN  E:=0 ;
       VAL:= VAL1 ;
       WHILE VAL GT 0 DO
       BEGIN VAL := VAL DIV 2 ; E:=E+ 1  END ;
       LOG2 := E ;
    END ↱ LOG2 ↓ ;
↱*************************************************************************↓
      PROCEDURE MULOPT(VAL1:INTEGER ; VAR EXP1,EXP2 : INTEGER ;
                          VAR OPT : OPTPWR) ;
      ↱ MULOPT DESIDES IF VAL = 2**EXP1                              :OPT=PUREP
                                =2**EXP2*(2**EXP2+1)                 :OPT=POSP
                                =2**EXP1*(2**EXP2-1)                 :OPT=NEGP
                                OTHER                                 OPT=NOOPT ↓
         VAR E1,E2 : INTEGER ;
             VAL: INTEGER ;
         BEGIN
            EXP1:=0 ; EXP2:= 0 ;
            IF VAL1 GT 0 THEN
            BEGIN  E1:= 0 ;
                VAL := VAL1 ;
                WHILE ¬ODD(VAL) DO
                BEGIN VAL:=VAL DIV 2; E1:= E1+1 END ;
                IF VAL=1 THEN
                BEGIN OPT:=PUREP; EXP1:=E1 END
                ELSE
                BEGIN
                    E2 := 0 ;
                    VAL:=VAL DIV 2; E2:=E2+1 ;
                    IF ODD(VAL) THEN
                    BEGIN
                      REPEAT VAL:=VAL DIV 2; E2 := E2+1 ;
                       UNTIL ¬ODD(VAL) ;
                      IF VAL GT 0 THEN OPT:= NOOPT ELSE
                      BEGIN OPT := NEGP ;
                        EXP2:= E2 ; EXP1 := E1 ;
                      END ;
                    END ELSE
                    BEGIN
                      REPEAT VAL:= VAL DIV 2; E2:= E2+1 ;
                        UNTIL ODD(VAL) ;
                      IF VAL GT 1 THEN OPT := NOOPT ELSE
                      BEGIN OPT := POSP ;
                        EXP2 := E2 ; EXP1:= E1 ;
                      END ;
```

```
                END ;
            END ;
        END ELSE OPT := NOOPT ;
      END * MULOPT ↓ ;
*****************************************************************↓
```

```
PROCEDURE PRTERR ;
  VAR I,K,POSIT : SHRINT ;
  BEGIN
    OUTCH(≡ ≡);
    FOR I:=1 TO 4 DO OUTCH(≡*≡);
    I:=-1; K:=1 ;
    REPEAT
      POSIT := ERRLIST[K].POS ;
      WHILE I LT POSIT DO
      BEGIN
        OUTPUT↑ := ≡ ≡; PUT(OUTPUT) ;
        I := I+1 ; K := K+1 ;
      END ;
      OUTPUT↑ :=≡↑≡; PUT(OUTPUT) ;
      I := I+1 ; K := K+1 ;

    UNTIL K GT ERRINX ;
    WHILE I LE 80 DO
    BEGIN
      OUTPUT↑ := ≡ ≡; PUT(OUTPUT) ;  I :=I+1 ;
    END ;
    FOR I := 1 TO ERRINX DO WRITE(ERRLIST[I].NMR) ;
    OUTCH(EOL); ERRINX:=0; POS1:=0 ;
  END ↗ PRTERR ↓ ;
↗*********    *********      *********      ********      ********↓
PROCEDURE ERROR(I:SHRINT) ;
  BEGIN
    ERR := TRUE ;
    IF ERRORKOUNT EQ 0 THEN
    BEGIN
      ERRORKOUNT := ERRORKOUNT+1 ;
      ERRORLIST[ERRORKOUNT] :=I ;
    END ELSE
    BEGIN
      J:=0;
      REPEAT
        J:=J+1;
        IF ERRORLIST[J] EQ I THEN GOTO 1 ;
      UNTIL J EQ ERRORKOUNT ;
      ERRORKOUNT := ERRORKOUNT+1 ;
      IF ERRORKOUNT EQ 10 THEN GOTO 1 ;
      ERRORLIST[ERRORKOUNT] :=I ;
    END ;
1:  ERRNRS[I DIV 32] := ERRNRS[I DIV 32] OR [I MOD 32] ;
    IF ERRINX GT 9 THEN
    WITH ERRLIST[10] DO
```

```
        BEGIN
        POS :=0; NMR:=110;ERRINX:=10;
          ERRNRS[NMR DIV 32] := ERRNRS[NMR DIV 32] OR [NMR MOD 32]
        END-ELSE
        BEGIN
          IF CHCNT GT POS1 THEN POS1 := CHCNT;
          ERRINX := ERRINX + 1 ;
          WITH ERRLIST[ERRINX] DO
          BEGIN
            POS := POS1 ; NMR := I
          END ;
          POS1 := POS1+ 1 ;
        END ;
      END * ERROR * ;
**********    *********      *********     ********     ********
```

```
     PROCEDURE ERRMSSG ;
        VAR I,J,TEMP : INTEGER ;
**********          *********          *********          *********          *********
        PROCEDURE ERRMSSG1 ;
          BEGIN
            CASE J OF
 1:     WRITE(≡SCALAR ≡,≡TYPE ≡,≡EXPECTED≡,EOL) ;
 2:     WRITE(≡INTEGER ≡,≡TOO ≡,≡LARGE≡,EOL) ;
 3:     WRITE(≡ERROR ≡,≡IN ≡,≡CONSTANT≡,EOL) ;
 4:     WRITE(≡= ≡,≡EXPECTED≡,EOL) ;
 5:     WRITE(≡FIELD ≡,≡NAME ≡,≡DECLARED ≡,≡TWICE≡,EOL) ;
 6:     WRITE(≡BAD ≡,≡RANGE ≡,EOL) ;
 7:     WRITE(≡) ≡,≡EXPECTED≡,EOL) ;
 8:     WRITE(≡NAME ≡,≡DECLARED ≡,≡TWICE ≡,EOL) ;
 9:     WRITE(≡TAGFIELD ≡,≡TYPE ≡,≡BAD≡,EOL) ;
10:     WRITE(≡: ≡,≡EXPECTED≡,EOL) ;
11:     WRITE(≡IDENTIFIER≡,≡ EXPECTED≡,EOL) ;
12:     WRITE(≡IDENTIFIER≡,≡ NOT ≡,≡DECLARED≡,EOL) ;
13:     WRITE(≡INDEX ≡,≡MUST ≡,≡BE ≡,≡OF ≡,≡SCALAR≡,≡ TYPE≡,EOL) ;
14:     WRITE(≡OF ≡,≡EXPECTED≡,EOL) ;
15:     WRITE(≡END ≡,≡EXPECTED≡,EOL) ;
16:     WRITE(≡ERROR ≡,≡IN ≡,≡TYPE ≡,≡DECLARATIO≡,≡N≡,EOL) ;
17:     WRITE(≡ERROR ≡,≡IN ≡,≡VARIABLE ≡,≡DECLARATIO≡,≡N≡,EOL) ;
18:     WRITE(≡ERROR ≡,≡IN ≡,≡VALUE ≡,≡DECLARATIO≡,≡N≡,EOL) ;
19:     WRITE(≡LOWBOUND ≡,≡GT ≡,≡HIGHBOUND≡,EOL) ;
20:     WRITE(≡ERROR ≡,≡IN ≡,≡VALUE ≡,≡ASSIGNMENT≡,EOL) ;
21:     WRITE(≡NOT ≡,≡A ≡,≡VARIABLE ≡,≡IDENTIFIER≡,EOL) ;
22:     WRITE(≡SYMBOLIC ≡,≡SUBRANGE ≡,≡TYOE ≡,≡NOT ≡,≡ALLOWED≡,EOL) ;
23:     WRITE(≡UNDEFINED ≡,≡IDENTIFIER≡,EOL) ;
24:     WRITE(≡COMPONENT≡,≡ TYPE ≡,≡IS ≡,≡NOT ≡,≡ARRAY ≡,≡OR ≡,
               ≡RECORD ≡,≡TYPE ≡,EOL) ;
25:     WRITE(≡] ≡,≡EXPECTED≡,EOL) ;
26:     WRITE(≡TYPE ≡,≡OF ≡,≡VARIABLE ≡,≡SHOULD ≡,≡BE ≡,≡RECORD≡,EOL) ;
27:     WRITE(≡NO ≡,≡SUCH ≡,≡FIELD ≡,≡IN ≡,≡THIS ≡,≡RECORD≡,EOL) ;
28:     WRITE(≡FIELD ≡,≡NAME ≡,≡EXPECTED≡,EOL) ;
29:     WRITE(≡ILLEGAL ≡,≡SYMBOL ≡,≡IN ≡,≡EXPRESSION≡,EOL) ;
30:     WRITE(≡BEGIN ≡,≡EXPECTED≡,EOL) ;
            END ◆ CASE ◆ ;
          END ; ◆ ERRMSSG1 ◆
**********          *********          *********          *********          *********
        PROCEUURE ERRMSSG2 ;
          BEGIN
            CASE J OF
31:     WRITE(≡CONFLICT ≡,≡OF ≡,≡INDEX ≡,≡WITH ≡,≡DECLARATIO≡,≡N≡,EOL);
32:     WRITE(≡ILLEGAL ≡,≡TYPE ≡,≡OF ≡,≡OPERAND≡,EOL) ;
33:     WRITE(≡v ≡,≡CANNOT ≡,≡BE ≡,≡USED ≡,≡AS ≡,≡MONADIC ≡,
```

```
                ≡OPERATOR≡,EOL) ;
34:   WRITE(≡:= ≡,≡EXPECTED≡,EOL) ;
35:   WRITE(≡ASSIGNMENT≡,≡ NOT ≡,≡ALLOWED≡,EOL) ;
36:   WRITE(≡ILLEGAL ≡,≡SYMBOL ≡,≡IN ≡,≡EXPRESSION≡,EOL) ;
37:   WRITE(≡THEN ≡,≡EXPECTED≡,EOL) ;
38:   WRITE(≡TYPE ≡,≡OF ≡,≡EXPRESSION≡,≡ NOT ≡,≡BOOLEAN≡,EOL) ;
39:   WRITE(≡; ≡,≡EXPECTED≡,EOL) ;
40:   WRITE(≡DO ≡,≡EXPECTED≡,EOL) ;
41:   WRITE(≡LABEL ≡,≡EXPECTED≡,EOL) ;
42:   WRITE(≡CONSTANT ≡,≡EXPECTED≡,EOL) ;
43:   WRITE(≡UNTIL ≡,≡EXPECTED≡,EOL) ;
44:   WRITE(≡TO/DOWNTO ≡,≡EXPECTED≡,EOL) ;
45:   WRITE(≡MIXED ≡,≡TYPES ≡,EOL) ;
46:   WRITE(≡TOO ≡,≡MANY ≡,≡LABELS ≡,≡IN ≡,≡THE ≡,≡PROGRAM≡,EOL) ;
47:   WRITE(≡TOO ≡,≡MANY ≡,≡CONSTANTS ≡,≡OR ≡,≡YET ≡,≡UNDECLARED≡,
                ≡LABELS ≡,≡IN ≡,≡THIS ≡,≡PROGRAM≡,EOL) ;
48:   WRITE(≡LABEL ≡,≡DEFINED ≡,≡MORE ≡,≡THAN ≡,≡ONCE≡,EOL) ;
49:   WRITE(≡TOO ≡,≡MANY ≡,≡EXIT ≡,≡LABELS≡,EOL) ;
50:   WRITE(≡( ≡,≡EXPECTED≡,EOL) ;
51:   WRITE(≡, ≡,≡EXPECTED≡,EOL) ;
52:   WRITE(≡ASSIGNMENT ≡,≡TO ≡,≡FUNCTION ≡,≡IDENTIFIER≡,≡ MUST ≡,
                ≡OCCUR ≡,≡IN ≡,≡FUNCTION ≡,≡ITSELF≡,EOL) ;
53:   WRITE(≡TOO ≡,≡MANY ≡,≡EXIT ≡,≡LABELS ≡,≡OR ≡,≡FORWARD ≡,
                ≡PROCEDURE≡,EOL) ;
54:   WRITE(≡TOO ≡,≡LONG ≡,≡A ≡,≡CHARACTER≡,≡ CONSTANT≡,EOL) ;
55:   WRITE(≡TYPE ≡,≡DECLARED ≡,≡TWICE≡,EOL) ;
56:   WRITE(≡] ≡,≡EXPECTED≡,EOL) ;
57:   WRITE(≡INDEX ≡,≡OUT ≡,≡OF ≡,≡RANGE≡,EOL) ;
58:   WRITE(≡LABEL ≡,≡TOO ≡,≡LARGE≡,EOL) ;
59:   WRITE(≡LOGICAL ≡,≡UNIT ≡,≡NUMBER ≡,≡MISSING≡,EOL) ;
60:   WRITE(≡* ≡,≡EXPECTED≡,EOL) ;
          END •CASE ↓ ;
        END ; • ERRMSSG2 ↓
**********    *********      **********    ********    *********
        PROCEDURE ERRMSSG3 ;
        BEGIN
          CASE J OF
61:   WRITE(≡EXTERNAL ≡,≡SYMBOL ≡,≡NOT ≡,≡DEFINED≡,EOL) ;
62:   WRITE(≡ASSIGNMENT≡,≡ TO ≡,≡FORMAL ≡,≡FUNCTION ≡,≡IDENTIFIER≡,
              ≡ ILLEGAL ≡,EOL) ;
63:   WRITE(≡TYPE ≡,≡OF ≡,≡VARIABLE ≡,≡MUST ≡,≡BE ≡,≡ARRAY≡,EOL) ;
64:   WRITE(≡TYPE ≡,≡OF ≡,≡EXPRESSION≡,≡ MUST ≡,≡BE ≡,≡SCALAR≡,EOL);
          END • CASE ↓ ;
        END ; • ERRMSSG3 ↓
**********    *********      **********    ********    *********
```

```
      BEGIN

       FOR I := 1 TO ERRORKOUNT-1 DO
       FOR J:=I+1 TO ERRORKOUNT DO
       BEGIN
          IF ERRORLIST[I] LE ERRORLIST[J] THEN GOTO 1 ;
          TEMP:= ERRORLIST[I] ;
          ERRORLIST[I] := ERRORLIST[J] ;
          ERRORLIST[J]:= TEMP ;
 1:    END ;
       FOR I:=1 TO ERRORKOUNT DO
       BEGIN
          J := ERRORLIST[I] ;
          IF J LE 69 THEN ERRMSSG1 ELSE
          IF J LE 60 THEN ERRMSSG2 ELSE ERRMSSG3 ;
       END ; •FOR ↓
      END ; •ERRMSSG ↓
*****************************************************************↓
```

```
PROCEDURE NEXTCH ;
    ↗ READS NEXT CHAR OF FILE INPUT,
      READS EOL TO BLANK ,
      PRINTS ERROR SUMMARY AFTER EACH LINE
      PRINTS ADDRESS AT THE BEGINNING OF LINE ↓
  BEGIN
    IF EOLFLAG THEN
    BEGIN
        IF ERRINX GT 0 THEN PRTERR ↗ PRINT ERROR ↓ ;
        EOLFLAG := FALSE ; CHCNT:= 0 ;
        OUTPUT↑ := ≡ ≡ ; PUT(OUTPUT) ;
                 IF DP THEN OUTO(LC) ELSE OUTO(IC) ;
        OUTPUT↑ := ≡ ≡ ; PUT(OUTPUT) ;
    END ↗ EOLFLAG ↓ ;
    GET(INPUT) ; CH:= INPUT↑ ; CHCNT := CHCNT + 1 ;
    IF CHCNT  > 73 THEN
    BEGIN
        OUTPUT↑:= ≡*≡ ;
        PUT(OUTPUT) ;
        OUTPUT↑ := CH ;
        PUT(OUTPUT) ;
        REPEAT
              GET(INPUT) ; CH:= INPUT ↑ ;
              OUTPUT↑ := CH ;
               PUT(OUTPUT) ;
     UNTIL CH=EOL ;
    END  ELSE
    BEGIN OUTPUT↑ := CH ; PUT(OUTPUT) ;
           IF CH≠ EOL THEN  GOTO 10
    END ;
    CH:= ≡ ≡ ; EOLFLAG := TRUE ;
 10:  END ↗ NEXTCH↓ ;
↗*********          *********          *********          ********          *********↓
   PROCEDURE  INSYMBOL ;
          ↗ LEXICAL ANALYSER ↓
     VAR
     I,K : INTEGER ;
    DIGIT : ARRAY [0.. 4] OF INTEGER ;
    BEGIN
 1:    WHILE CH=≡ ≡ DO NEXTCH ;
       IF CH≤ ≡Z≡ THEN       ↗ IDENTIFIER ↓
       BEGIN
             K:=0 ;
             REPEAT
                K:=K+1 ;
```

```
              A[K] := BLANK ;

               UNTIL  K=10 ;
               K:=0;
             REPEAT IF K<ALFALENG THEN
                    BEGIN
                         K:=K+1;
                         A[K] :=CH ;
                    END;
                         NEXTCH;
             UNTIL CH > ≡9≡ ;
                    ↑ SEARCH FOR RESERVED WORDS ↓
             FOR I:=WL[K] TO WL[K+1] -1 DO
                    BEGIN
                           CPARE(A,WD,I,COMPARE) ;
                         IF COMPARE THEN
                         BEGIN NO :=WNO[I] ;
                                 CL:=WCL[I] ; IVAL:=0;
                                 GOTO 2 ;
                           END ;
                    END;
                    NO:=1  ; CL:=K;
                    I :=0 ;
                    REPEAT
                       I:=I+1 ;
                       AVAL[I] := BLANK ;
                    UNTIL   I=10 ;
                    I :=0 ;
                    REPEAT
                         I:=I+1 ;
                         AVAL[I] :=A[I] ;
                    UNTIL I=K;
     2:  END  ELSE
           IF CH ≤ ≡9≡ THEN      ↑ NUMBER ↓
           BEGIN NO:=2; CL:=1; I:=0;
                 WHILE CH IN DIGITS DO
                 BEGIN  IF I< 5 THEN
                    DIGIT[I] :=ORD(CH) - ORD(≡0≡) ;
                    I:=I+1; NEXTCH ;
                 END ;
                 IVAL :=0;
                 IF CH = ≡B≡ THEN    ↑OCTAL ↓
                      BEGIN IF I > 5 THEN ERROR(2) ELSE
                                FOR K:=0 TO I-1 DO
                                 IVAL :=8*IVAL + DIGIT[K] ;
                                 NEXTCH ;
                      END   ELSE
                      BEGIN  IF I>4
```

```
                   THEN ERROR(2) ELSE
                    FOR K:=0 TO I-1 DO
                         BEGIN IF IVAL≤ MAX10 THEN
                              IVAL:=10*IVAL + DIGIT[K]
                              ELSE BEGIN ERROR(2) ;
                                         IVAL:=0; END ;
                         END;
           END ;
 END       ELSE
BEGIN ↱ SPECIAL CHARACTER ↓
     IF CH = ≡≡≡≡ ↱ CHAR CONSTANT↓ THEN
          BEGIN NO:=2; K:=0; BT1:=FALSE ;
               REPEAT NEXTCH ;
                    IF CH=≡≡≡≡ THEN
                    BEGIN NEXTCH ; BT1:= CH≠≡≡≡≡ END ;
                    IF ¬BT1 THEN
                       IF K=ALFALENG THEN
                       BEGIN  ERROR(54) ;
                              BT1:= TRUE ;
                       END  ELSE
                       BEGIN K:=K+1; A[K] := CH  END ;
               UNTIL BT1 ;
            CL:=2; TECH:=A[1]; IVAL:=ASCIICODE[TECH] ;
            IF K EQ 2 THEN
            BEGIN
              TECH:=A[2]; TEMP:=ASCIICODE[TECH];
              APPEND(IVAL,8,TEMP)
            END ELSE APPEND(IVAL,8,40B) ;
          END ↓ ≡ ↓ ELSE
          BEGIN NO:=SYMNO[CH] ;  CL:= SYMCL[CH] ;
                IVAL := 0 ;
                ↱ TEST FOR TWO CHARRACTER  SYMBOL ↓
                IF CH= ≡:≡ THEN
                    BEGIN  NEXTCH ;
                         IF CH= ≡=≡ THEN
                             BEGIN NO:= 22 ; NEXTCH END ;
                    END  ELSE
                    IF CH= ≡.≡ THEN
                    BEGIN  NEXTCH ;
                           IF CH= ≡.≡ THEN ↱:↓
                           BEGIN NO:=21 ; NEXTCH  END;
                    END  ELSE
                    IF CH= ≡/≡ THEN
                    BEGIN NEXTCH ;
                          IF CH = ≡*≡ THEN  ↱ SKIP COMMENT ↓
```

```
                             BEGIN  NEXTCH ;

                             3 :    WHILE CH≠ ≡*≡  DO NEXTCH ;
                                    NEXTCH ;  IF CH≠≡/≡ THEN
                                    GOTO 3 ELSE
                                    NEXTCH ;  GOTO 1 ;
                             END ;
                        END  ELSE
                        NEXTCH ;
                END ;
        END ; ↱ SPECIAL CHARACTER ↓
    END ;       ↱ INSYMBOL↓
↱**********************************************************↓
```

```
PROCEDURE  SRCHREG (P:INTEGER ) ;
        ↱ SEARCHES ONE BLOCK ,RETURNS CTPTR ↓
         LABEL 1 ;
     BEGIN  CTPTR := P;
            WHILE CTPTR ≠ 0 DO
              BEGIN
                  COMP ( CONTEXTTABLE[CTPTR].NAME, AVAL, COMPAR ) ;
                  IF COMPAR THEN  GOTO 1
                  ELSE CTPTR := CONTEXTTABLE[CTPTR].NXTEL   ;
              END ;
  1:  END•SRCHREC↓ ;
↱*********  *********       *********      ********      ********↓
PROCEDURE SEARCH ;
     ↱ SEARCHES CONTEXTTABLE , RETURNS CTPTR  AND
            DISX=INDEX TO DISPLAY ↓
         LABEL 1 ;
     VAR I: INTEGER ;
     BEGIN
            FOR I:=TOP DOWNTO 0 DO
                BEGIN CTPTR := DISPLAY[I].FNAME ;
             WHILE CTPTR NE 0 DO
                                BEGIN
                    COMP(CONTEXTTABLE[CTPTR].NAME,AVAL,COMPAR) ;
                    IF COMPAR THEN GOTO 1
                    ELSE CTPTR := CONTEXTTABLE[CTPTR].NXTEL ;
                                END
               END ;
  1:     DISX := I;
     END ↱ SEARCH ↓ ;
↱*****************************************************************↓
```

```
PROCEDURE INCONST(VAR V,P:INTEGER; NXT:INTEGER);
VAR SIGN : BOOLEAN; PT: INTEGER ;
BEGIN
        SIGN := FALSE ; P:=0 ;
        IF NO=7 THEN
        BEGIN SIGN:=CL=2 ;
        IF CL≤2 THEN
                BEGIN INSYMBOL;IF NO=1 THEN ERROR(3) END ;
        END ;
        IF NO=2 THEN
        BEGIN CASE CL OF
1:      P:=INTPTR;
2:    P:= CHARPTR ;
        END ;
        IF SIGN THEN  V:=-IVAL ELSE V:=+IVAL ;
        INSYMBOL ;
        END   ELSE
        IF NO=1  THEN
        BEGIN  PT:= CTPTR ;
                  SRCHREC (NXT) ;
                  IF CTPTR=0 THEN SEARCH ;
                  IF CTPTR=u THEN ERROR(12) ELSE
                  WITH CONTEXTTABLE[CTPTR] DO
                  BEGIN  IF (KLASS≠KONST) ∨(CONKIND=FORMAL)
                                 THEN ERROR(42) ELSE
                                        BEGIN P:=CONTYPE;V:=VALUES END ;
                  END;
                  CTPTR:=PT ; INSYMBOL ;
        END ELSE ERROR(3) ;
      END ↱ INCONST    ↓ ;
↱**********************************************************↓
```

```
PROCEDURE SKIP(FNO: INTEGER) ;
     BEGIN
        WHILE (ERRCL[NO] EQ 2)          AND (FNO NE NO)  DO
        IF (NO EQ 41) AND (CL EQ 2) THEN   ↑RECORD↓
         BEGIN
           REPEAT INSYMBOL ;
                SKIP(49) ;
           UNTIL ¬(NO IN [16,30]) ;
             IF NO EQ 26 THEN INSYMBOL ;
         END ELSE INSYMBOL ;
     END ↑SKIP↓ ;
↑**********     **********     **********     *********     *********↓
   PROCEDURE FINDSEMICOLON ;
        BEGIN
          IF NO NE 16 THEN
           BEGIN ERROR(39) ; SKIP(16);
                 IF NO NE 16 THEN WRITE(≡ ≡,≡EXIT1≡) ;
         END ; INSYMBOL ;
        END ↑ FINDSEMICOLOR↓ ;
↑**********************************************************************↓
```

```
PROCEDURE SEG1 ; ↗INITIALIZATION ↓
  BEGIN
     DIGITS := [≡0≡,≡1≡,≡2≡,≡3≡,≡4≡,≡5≡,≡6≡,≡7≡,≡8≡,≡9≡ ] ;
    DP := TRUE ;
    ↗ INITIALIZE CONTEXTTABLE ↓
     WITH CONTEXTTABLE[0]  DO  ↗TYPE OF NIL ↓
     BEGIN  I:= 1 ;
            REPEAT
                 NAME[I] := BLANK ;
                 I:= I+1 ;
            UNTIL I=10 ;
             NXTEL := 0 ;  KLASS := TYPES ;
            SIZE := 0 ; FORM:= NUMERIC  ;
  END ;
  CTPTR := 1 ;
  WITH CONTEXTTABLE[CTPTR] DO ↗TUPE OF MEM ↓
  BEGIN
   I:= 1 ;
   REPEAT  NAME[I] := BLANK ;
           I:=I+1 ;
   UNTIL I=10 ;
   NXTEL := 0 ; KLASS:= TYPES ;
   SIZE := 136B ; FORM := ARRAYS ;
   LO:= 0 ; HI:= 177B    ; OPTTYP := PUREP ;
  END ;
  CHARPTR := 10   ;
  WITH CONTEXTTABLE[CHARPTR] DO ↗CHAR↓
  BEGIN  I:=0 ;
        REPEAT
           I:=I+1 ;
           NAME[I] := INITNAM[26,I] ;
        UNTIL I=10 ;
        NXTEL:= 5 ;     KLASS := TYPES    ;
        SIZE:=1 ; FORM:= SYMBOLIC ; BITSIZE:= 7;
  END;
  WITH CONTEXTTABLE[2] DO ↗ ; ↓
  BEGIN  I:=1 ;
        REPEAT  NAME[I] := BLANK ; I:=I+1 ;
         UNTIL I=10 ;
       NXTEL:=CHARPTR; CONKIND:=ACTUAL;
        VALUES:= 63 ; SUCC:= 0 ;
  END;
     CONTEXTTABLE[CHARPTR].FCONST := 2;
  BOOLPTR:= 11 ;
  WITH CONTEXTTABLE[BOOLPTR] DO ↗BOOLEAN ↓
      BEGIN
```

```
     I:=0 ;
     REPEAT
       I:=I+1; NAME[I] := INITNAM[27,I] ;
     UNTIL I EQ 10 ;
     NXTEL:= CHARPTR ; KLASS:= TYPES ; SIZE :=1 ;
     FORM := SYMBOLIC ; BITSIZE:= 2 ;
END ;
 WITH CONTEXTTABLE[3] DO ↗EOL↓
BEGIN I:=1;
     REPEAT NAME[I]:= INITNAM[20,I] ;
           I:=I+1;
      UNTIL I=10;
      NXTEL:= BOOLPTR ;
     KLASS:= KONST ; CONTYPE := CHARPTR ; CONKIND := ACTUAL;
     VALUES:= 0;  SUCC:= 0 ;
END;
CTPTR:= 0 ;
NEXT := 27 ;
   PT := 4 ;
FOR IT:=0 TO 1 DO ↗FALSE,TRUE ↓
BEGIN
  WITH  CONTEXTTABLE[PT+IT] DO
          BEGIN I:=0 ;
            REPEAT
              I:=I+1; NAME[I] := INITNAM[IT+28,I] ;
            UNTIL  I EQ 10 ;
           NXTEL:= NEXT ;
           KLASS:= KONST; CONTYPE:= BOOLPTR ;
           CONKIND:=ACTUAL; VALUES:=IT; SUCC:= 0;
       END;
    NEXT := 4 ;
END;
CONTEXTTABLE[BOOLPTR]. FCONST := 5 ;
INTPTR := 12 ;
WITH CONTEXTTABLE[INTPTR] DO  ↗INTEGER ↓
BEGIN  I:=0 ;
      REPEAT
         I:=I+1 ;
         NAME[I] := INITNAM[30,I] ;
      UNTIL I=10 ;
       NXTEL:= BOOLPTR ;  KLASS:= TYPES;
       SIZE:= 1;  FORM:= NUMERIC;
       MIN:=100000B; MAX:= 77777B ;
       BITS := WORDLENGTH ;
END;
BYTPTR := 13 ;
```

```
WITH CONTEXTTABLE[BYTPTR] DO ↗BYTE↓
BEGIN
  I := 0;
  REPEAT
    I:=I+1;
    NAME[I] := INITNAM[37,I] ;
  UNTIL I=10 ;
  NXTEL := INTPTR ;
  KLASS:= TYPES; SIZE :=1 ;
  FORM := SYMBOLIC;     BITSIZE := 4 ;
END;
WRDPTR := 14 ;
WITH CONTEXTTABLE [WRDPTR] DO ↗WORD ↓
BEGIN
  I:=0 ;
  REPEAT
    I:=I+1;
    NAME[I] := INITNAM[38,I] ;
  UNTIL I=10 ;
  NXTEL := BYTPTR; KLASS:= TYPES ;
  SIZE :=1 ; FORM :=SYMBOLIC ;
  BITSIZE := WORDLENGTH ;
END ;
UNDCLPTR:=15 ;
WITH CONTEXTTABLE[UNDCLPTR]  DO
BEGIN  I:=1 ;
       REPEAT  NAME[I]:= BLANK ;
               I:=I+1;
       UNTIL I=10 ;
      NXTEL:= 0 ;  KLASS:= VARS ; VTYPE:= 0 ;
      VKIND:= ACTUAL ; VADDR:= 0 ; VLEVEL:= 0;
END ;
PT := 6 ;
WITH CONTEXTTABLE[PT] DO ↗ MEM↓
BEGIN  I:=1 ;
       REPEAT NAME[I] := INITNAM[31,I] ;
              I:=I+1;
       UNTIL I=10 ;
       NXTEL:= 0; KLASS:= VARS ;
       VTYPE:= CTPTR ; VKIND:= ACTUAL ;
       VADDR:= 0 ; VLEVEL := 0 ;
END ;
CTPTR := 7 ;
WITH CONTEXTTABLE[CTPTR] DO  ↗TYPE OF INPUT AND OUTPUT↓
BEGIN I:=1;
      REPEAT NAME[I] :=BLANK ;
```

```
             I:=I+1 ;
      UNTIL I=10 ;
      NXTEL := 0 ;  KLASS := TYPES ;
      SIZE := 113 ;  FORM := FILES ;
END ;
PT := 8 ;
FOR IT := 22 TO 23 DO
BEGIN
  WITH CONTEXTTABLE[PT] DO
  BEGIN I:= 1 ;
        REPEAT  NAME[I] := INITNAM[IT,I] ;
                I:= I+ 1 ;
        UNTIL I=10 ;
        NXTEL:= CTPTR ; VKIND := ACTUAL ;
        VLEVEL := 0 ;
  END ;
  PT := PT + 1 ;
END ;
    NEXT := 1 ;
    PT := 15 ;
    FOR IT := 0 TO 11 DO ↱ TEXT  TO WRITE ↓
    BEGIN
      WITH CONTEXTTABLE[PT] DO
      BEGIN
        I:=0;
        REPEAT
               I:=I+1 ;
          NAME[I] := INITNAM[IT-1,I] ;
      UNTIL I EQ 10 ;
        NXTEL := NEXT ; KLASS:= PROC ; PROCTYPE:=PT ;
        FORMALS:=0; PROCADDR:=0; PROCLEVEL:=0;
        SEGSIZE:=IT; PROCKIND := ACTUAL ;
      END ;
     .NEXT :=PT; PT:=PT+1 ;
    END ;
    FOR IT := 1 TO 5 DO
    WITH EXTX[IT] DO
    BEGIN
      I:=0;
      REPEAT
        I:=I+1 ;
        XNAM[I] := INITNAM[IT,I] ;
      UNTIL I EQ 10 ;
      XSYM := IT ;
    END ;
    WITH DISPLAY[0] DO
```

```
BEGIN FNAME := 14 ; OCCUR := BLOCK END ;
FOR IT := 1 TO UNDMAX-1 DO
          UNDLAB[IT].SUCC := IT+1 ;
UNDLAB[UNDMAX].SUCC := 0 ;
ERRINX := 0 ; POS1 := 0 ;
JMPIX :=0 ; UEXTABIX :=0 ; CHNIX :=1 ;
EXSYM := 5;
PREDEFPTR :=  27 ;
DISPLAY[1].FNAME := 0 ; DISPLAY[1].OCCUR := BLOCK ;
TOP := 1 ; LEVEL := 0 ;  NEXT := 0 ;
CHCNT :=0 ;
LC :=0 ;
IC := 0 ;
VLC := 0 ;
NEXT := 14 ;
P := 27 ;
INDEX := P ;
EOLFLAG := FALSE ;
CERR := FALSE ;
PUNCHBIN := FALSE; LISTBIN:= FALSE; RLISTBIN:= FALSE ;
TRNSFR[1] :=0 ; TRNSFR[2]:=1 ; TRNSFR[3]:= 0 ;
INSYMBOL ;
COMP(PROG,AVAL,COMPAR) ;
IF COMPAR THEN INSYMBOL ELSE CERR:=TRUE;
I:=0 ;
REPEAT
  I:=I+1; PRNAM[I] := AVAL[I]
UNTIL I EQ 5 ;
INSYMBOL ;
IF NO EQ 15 THEN
REPEAT  INSYMBOL;
   IF AVAL[1] EQ ≡B≡ THEN LISTBIN:=TRUE ELSE
   IF AVAL[1] EQ ≡P≡ THEN PUNCHBIN :=TRUE ELSE
   IF AVAL[1] EQ ≡L≡ THEN RLISTBIN:=TRUE ELSE
   CERR:=TRUE ;
   INSYMBOL;
UNTIL NO NE 15 ;
IF CERR THEN
BEGIN
  LINES(1); WRITE(≡CONTROL CARD ERROR≡) ; LINES(1)
END ;
KP:=0 ;
RPMAX := 0 ;
SURRPTR := 0 ;
UNDECPTR := 0 ;
HCA := 0 ;
```

```
                LCX :=0 ;
                VALPT:= FALSE ;
                CTY := TRUE ;
                ERRORKOUNT := 0 ;
                M := 28 ;
            END ; ↱SEG1 ↓
↱*********************************************************************↓
```

```
       PROCEDURE SEG2 ;
         ↱ LABEL AND CONSTANT DECLARATION ↓
     BEGIN
           IF NO EQ 42 THEN ↱ LABEL↓
           BEGIN
             REPEAT
               INSYMBOL;
               IF (NO EQ 2) AND (CL EQ 1) THEN
               BEGIN
                 FOR IT:= FSTIX TO CEXTABIX DO
                 IF EXTAB[IT].EXVAL = IVAL THEN
                 BEGIN ERROR(48) ; GOTO 2 END ;
                 IF CEXTABIX = MAXEXLABS THEN ERROR(49) ELSE
                 BEGIN
                   CEXTABIX := CEXTABIX+1 ;
                   IF JMPIX GT JMPMAX THEN ERROR(53) ELSE
                   BEGIN
                     WITH EXTAB[CEXTABIX] DO
                     BEGIN EXVAL:=IVAL; JMPBIX := JMPIX END;
                     JMPTAB[JMPIX] := LEVEL ;
                     JMPIX := JMPIX + 1 ;
                   END
                 END ;
2 :              INSYMBOL ;
               END ↱ IF (NO=2) AND (CL=1)↓ ELSE
               BEGIN ERROR(41) ; GOTO 3 END
             UNTIL NO NE 15 ↱,↓ ;
3 :          FINDSEMICOLON ;
           END ;
         IF NO=43 THEN ↱CONST↓
         BEGIN
            INSYMBOL;
             WHILE NO EQ 1 DO
             BEGIN
               REPEAT
                 SRCHREC(NEXT);
                 IF CTPTR NE 0 THEN ERROR(8) ;
               INDEX := INDEX+1 ;
                P := INDEX ;
                 WITH CONTEXTTABLE[P] DO
                 BEGIN
                   I:=0;
                   REPEAT
                     I:=I+1;
                     NAME[I]:=AVAL[I] ;
                   UNTIL I=10 ;
```

```
         NXTEL:=NEXT ;   KLASS:= KONST ;
         CONKIND := ACTUAL ;
      END ;
      NEXT := P ;   INSYMBOL ;
      IF (NO NE 8) OR (CL NE 6) THEN
       ERROR(4) ELSE INSYMBOL ;
      TEM := CONTEXTTABLE[NEXT].NXTEL ;
      INCONST(I,N,TEM) ;
      CONTEXTTABLE[P].CONTYPE := N;
      CONTEXTTABLE[P].VALUES := I ;
      WHILE NO EQ 15 DO
      BEGIN  INSYMBOL ;
          IF NO NE 1 THEN
          BEGIN ERROR(11) ; SKIP(15) END ;
      END ;
     UNTIL NO NE 1 ;
     FINDSEMICOLON ;
    END  * WHILE NO=1* ;
  END ;
  END ; *SEG2 *
**************************************************************
```

```
PROCEDURE SEG4;    ↑ SYNONYM DECLARATION ↓
  PROCEDURE SYNDECL ;
      VAR
        SY,TY,SNXT : SHRINT ;
        JJ : INTEGER ;
      BEGIN
        SY := P ;
        TY := 0 ;
        SNXT := SY ;
        INSYMBOL ;
        IF (NO NE 8) OR (CL NE 6) THEN ERROR(4) ;
        REPEAT
           INSYMBOL ;
           IF NO NE 1 THEN  BEGIN ERROR(11); GOTO 1 END ;
           SRCHREC(NEXT) ;
           IF CTPTR NE 0 THEN ERROR(8) ;
           INDEX := INDEX+1 ;
           P := INDEX ;
           WITH CONTEXTTABLE[P] DO
           BEGIN
              NXTEL := NEXT ;
              JJ:= 0 ;
              REPEAT
                 JJ := JJ+1 ;
                 NAME[JJ] := AVAL[JJ] ;
              UNTIL JJ=10 ;
              SYNCELL := TRUE ;
           END ;
           NEXT := P ;
           CONTEXTTABLE[SY].SYNPTR := P ;
           SY := P ;
           INSYMBOL ;
        UNTIL NO NE 15 ;
          IF TY NE 0 THEN
                WITH CONTEXTTABLE[SY] DO SYNPTR := TY
              ELSE
                WITH CONTEXTTABLE[SY] DO SYNPTR := SNXT ;
      END ;
↑*********  *********     *********     ********     ********↓
   BEGIN
      IF NO=51 THEN ↑SYN↓
      BEGIN
        INSYMBOL ;
        WHILE NO EQ 1 DO
        BEGIN
```

```
                  SRCHREC(NEXT) ;
                  IF CTPTR NE 0 THEN
                               BEGIN
                           IF NOT(CONTEXTTABLE[CTPTR].SYNCELL) THEN
                           ERROR(8) ELSE
                                  BEGIN
                                   P := CTPTR ;
                                   Q := CONTEXTTABLE[P].SYNPTR ;
                                    SYNDECL ;
                                    GOTO 6
                                 END;
                               END ;
                  INDEX := INDEX+1 ;
                  P := INDEX ;
                  WITH CONTEXTTABLE[P] DO
                  BEGIN
                     SYNCELL := TRUE ;
                      I:= 0 ;
                      REPEAT
                        I:=I+1 ;
                        NAME[I] := AVAL[I] ;
                      UNTIL I=10 ;
                       NXTEL := NEXT ;
                  END ;
                  NEXT := P ;
                    Q := 0 ;
                     SYNDECL ;
   6 :            IF NO NE 16 THEN  ERROR(39) ;
                  INSYMBOL ;
                END ;
              END *SYN* ;
          END ; *SEG4*
*********************************************************************
```

```
PROCEDURE TYPEDECL(VAR TL,P1 : INTEGER) ;
  ↑ TL RETURNS THE SIZE OF THE VARIABLE OF TYPE POINTED
    TO BY P1 ↓
VAR I,L,LL,J,GV,E1,E2,BDISPL  :  INTEGER ;
          TEMP : INTEGER ;
            CPTR : INTEGER ;
    JJ:  INTEGER ;
    OPT:OPTPWR:DISPL:ADDRESS;
    PACKFLAG,LERR:BOOLEAN ;
    LASTFLD,PP,  NXTF,NXTC,NXTA,RTYP: INTEGER ;
      NXTE : INTEGER ;
**********   **********      **********      *********      *********
```

```
PROCEDURE  SKIPT(FNO:INTEGER ) ;
   BEGIN
     WHILE (TERRCL[NO]=0) ^(FNO≠NO)   DO
          INSYMBOL ;
   END ρSKIPT ↓  ;
PROCEDURE TYPERR(I:INTEGER)  ;
   BEGIN
      TL:=0 ; P1:=0 ;
      ERROR(I) ; SKIPT(49) ;
      WRITE(≡ ≡,≡ERROR(I) ≡,EOL)  ;   SKIPT(49)  ;
   END  ρ  TYPERR  ↓  ;
ρ*********          ********         *********       ********        ********↓
PROCEDURE SUBRANGE(VAR VAL1,VAL2 :INTEGER;N1:INTEGER;
                                          P:INTEGER ;) ;
  ρTHE FIRST SYMBOL HAS BEEN READ .
  THE PROCEDURE RETURNS TWO BOUND VALUES IN VAL1,VAL2
  AND RETURNS N1=POINTER TO TYPE OF CONSTANTS.
  ERROR: TYPES DO NOT AGREE, TYPE IS NOT INTEGER ,CHAR,OR
  ERROR: SYMBOLIC, VAL1> VAL2.
  P INDICATES BEGINNING OF SEARCH FOR FIRST SYMBOL
  IF IT IS AN ID . ↓
VAR N2: INTEGER ; ρPOINTER  ↓
BEGIN  INCONST(VAL1,N1,P) ;
   IF N1=0 THEN ERR:=TRUE ELSE
   BEGIN IF NO≠21 THEN  ERROR(10) ELSE
      INSYMBOL ;
      INCONST(VAL2,N2,NEXT) ;
      IF N1≠N2 THEN ERR:=TRUE ELSE
      IF VAL1>VAL2 THEN  ERROR(1C)
  END ;
END ρSUBRANGE ↓ ;
ρ*********          ********         *********       ********        ********↓
PROCEDURE SUBTYPE( VAR I,J:INTEGER ; P: INTEGER) ;
  ρ EITHER A TYPE.ID FOR A SUBRANGE OR AN EXPLICIT SUBRANGE

  ARE PROCESSED .
  RETURNS I=LOWBOUND, J=HIGHBOUND,

  P=POINTER TO TYPE OF CONSTANTS. ↓

   VAR
     JTEM : INTEGER ;
BEGIN
   IF NO=1 THEN
   BEGIN  SRCHREC(NEXT) ;
      IF CTPTR =0 THEN SEARCH ;
```

```
          IF CTPTR EQ 0 THEN ERROR(12) ELSE
          BEGIN
             IF CONTEXTTABLE[CTPTR].KLASS = TYPES THEN
             BEGIN
               IF CONTEXTTABLE[CTPTR].FORM >SYMBOLIC THEN
                  ERROR(13) ELSE
                  BEGIN
                     CASE CONTEXTTABLE[CTPTR].FORM OF
NUMERIC    :         BEGIN I:=CONTEXTTABLE[CTPTR].MIN ;
                           J:=CONTEXTTABLE[CTPTR].MAX ;
                           P:= INTPTR ;
                     END ;
SYMBOLIC :           BEGIN
                        I:=0;
                        JTEM:=CONTEXTTABLE[CTPTR].FCONST ;
                        J:= CONTEXTTABLE[JTEM].VALUES ;
                        P:= CTPTR ;
                     END ;
                  END↑CASE↓ ;
                  INSYMBOL ;
               END ;
             END ↑ TYPES↓ ELSE
             IF CONTEXTTABLE[CTPTR].KLASS=KONST  THEN
               SUBRANGE(I,J,P,CTPTR) ELSE   ERROR(42)     ;
          END ;
       END ↑ID ↓ ELSE
       IF (NO=2) v (NO=7) THEN
          SUBRANGE(I,J,P,0) ELSE ERROR(1)
    END  ↑ SUBTYPE ↓ ;
↑*********  *********     *********     ********     *********↓
    PROCEDURE  SCALDECL(N:INTEGER ) ;
       BEGIN
          SUBRANGE(I,J,PP,N) ;
          IF ¬ERR THEN
          IF  CONTEXTTABLE[PP].FORM = SYMBOLIC THEN
          BEGIN  ERROR(22) ;
            P1:=0
          END  ELSE
          BEGIN
            INDEX := INDEX + 1 ;
            P:=INDEX ;
            WITH CONTEXTTABLE[P]  DO
            BEGIN JJ:=1 ;
              REPEAT
                NAME[JJ] := BLANK ;
                JJ := JJ + 1 ;
```

```
                    UNTIL JJ=10 ;
                    NXTEL:=0; KLASS:=TYPES ;
                    SIZE := 1 ;  FORM := NUMERIC ;
                    MIN := I;  MAX := J;
                    IF ABS(I) > ABS(J)  THEN
                      BITS := LOG2(ABS(I)) ELSE
                      BITS := LOG2(ABS(J))
                  END ;
                  TL:=1 ; P1:=P
               END  ELSE  P1:=0
            END  ↑SCALDECL ↓  ;
 **********       *********        *********      ********      *********
```

```
      PROCEDURE FIELDLIST(VAR MAXSIZE,VARPTR,NXTF:INTEGER) ;
        VAR MXL,L,LL,MINSIZE,CASEBITS,I: INTEGER ;
              P,PP,PP1,PP2,NXTC,NXT,CPTR: INTEGER ;
               TAGFLAG : BOOLEAN ;
+**********      *********     *********      *********      *********+
          PROCEDURE RECERR(I:INTEGER) ;
             BEGIN ERROK(I) ;
                      SKIPT(53) ;
             END ; +RECERR +
+**********      *********     *********      *********      *********+
      PROCEDURE  ADJUST  ;
          + MOVE LAST FIELD TO RIGHT . IF IT IS THE ONLY FIELD,
            CHANGE TO NONPACKED.  IF LAST FIELD IS TAGFIELD THEN
             DO NOT MOVE .
             INCREASE  DISPL, RESET BDISPL .  +
          BEGIN
             IF  ¬TAGFLAG  THEN
             WITH CONTEXTTABLE[LASTFLD] DO
             BEGIN
                IF BITDISPL=0  THEN
                  + ONLY ONE FIELD  IN WORD +
                BITWIDTH := 0   ELSE
                BITDISPL := WORDLENGTH - BITWIDTH ;
             END ;
             DISPL := DISPL+ 1 ;  BDISPL := 0 ;
          END  + ADJUST  + ;
+**********      *********     *********      *********      *********+
          BEGIN
             NXT := NXTF ;
             REPEAT
               IF NO=30 THEN  + CASE + GOTO 2 ;
               I:=0;
  11:          IF  NO≠1  THEN
               BEGIN ERROR(11) ;
                  IF TERRCL[NO]=1 THEN GOTO 11 ;
                  GOTO 12 ;
               END ;
               SRCHREC(NXT) ;
               IF CTPTR ≠ 0 THEN ERROR(5) ELSE
               BEGIN I:=I+1 ;
                  INDEX:= INDEX + 1 ;
                  P := INDEX ;
                  WITH CONTEXTTABLE[P] DO
                  BEGIN JJ:=1 ;
                   REPEAT   NAME[JJ] := AVAL[JJ]  ;
                            JJ:=JJ+1;
```

```
          UNTIL JJ=10 ;
         NXTEL :=NXT ;
          KLASS:= FIELD ;
          FLDTYPE := 0 ;
         END ;
         NXT:= P ;
       END ;
       INSYMBOL ;
       IF NO= 15 THEN ↑ ; ↓
       BEGIN  INSYMBOL; GOTO 1 END ;
       IF NQ≠ 21 THEN  ↑NOT  ↓  ERROR(10)
         ELSE INSYMBOL ;
11 :   TYPEDECL(L,P) ;
       IF P≠0 THEN
       IF CONTEXTTABLE[P].FORM > RECORDS THEN
         ERROR(24) ELSE
       BEGIN
        IF PACKFLAG THEN
        BEGIN
         IF I GT 1 ↑REVERSE POINTERS ↓  THEN
         BEGIN
          PP:=NXT ;
          FOR I:=I DOWNTO 1 DO
          BEGIN
           PP1 := CONTEXTTABLE[PP].NXTEL ;
           CONTEXTTABLE[PP].NXTEL := PP2;
           PP2:=PP;
           PP:=PP1 ;
           END;
           CONTEXTTABLE[NXT].NXTEL := PP;
           NXT := PP2 ;
          END ↑REVERSE ↓ ELSE
          PP:=CONTEXTTABLE[NXT].NXTEL ;
          PP1:= NXT ;
          IF  CONTEXTTABLE[P].FORM LE SYMBOLIC THEN
          BEGIN
           CASE CONTEXTTABLE[P].FORM OF
      NUMERIC:  LL:=CONTEXTTABLE[P].BITS ;
      SYMBOLIC :  LL:= CONTEXTTABLE[P].BITSIZE ;
           END ;
           REPEAT
            IF BDISPL+LL GT WORDLENGTH THEN ADJUST;
            IF LL EQ WORDLENGTH THEN
            BEGIN
             CONTEXTTABLE[PP1].BITWIDTH :=0 ;
             CONTEXTTABLE[PP1].FLDADDR:=DISPL ;
```

```
                        DISPL:=DISPL+1 ;
                      END ELSE
                      BEGIN
                        CONTEXTTABLE[PP1].BITWIDTH :=LL ;
                        CONTEXTTABLE[PP1].BITDISPL := BDISPL;
                        CONTEXTTABLE[PP1].FLDADDR:=DISPL;
                        BDISPL:=BDISPL+LL;
                      END ;
                      CONTEXTTABLE[PP1].FLDTYPE := P ;
                      LASTFLD :=PP1;TAGFLAG:=FALSE ;
                      PP1 := CONTEXTTABLE[PP1].NXTEL ;
                    UNTIL PP1=PP;
                  END •FORM LE SYMBOLIC• ELSE
                  BEGIN
                    IF BDISPL NE 0 THEN ADJUST ;
                    TAGFLAG := FALSE ;
                    REPEAT
                      WITH CONTEXTTABLE[PP1] DO
                      BEGIN
                        BITWIDTH :=0;
                        FLDTYPE:=P;
                        FLDADDR:=DISPL ;
                      END;
                      DISPL := DISPL+L;
                      LASTFLD := PP1;
                      PP1 := CONTEXTTABLE[PP1].NXTEL ;
                    UNTIL PP1=PP ;
                    END ;
              END •PACKFLAG• ELSE
            LL:= I*L+DISPL ;
            DISPL := LL ;
            PP := NXT ;
            FOR I:=I DOWNTO 1 DO
               BEGIN
                  CONTEXTTABLE[PP] .FLDTYPE := P ;
                  LL := LL- L ;
                  CONTEXTTABLE[PP].FLDADDR := LL ;
                  CONTEXTTABLE[PP].BITWIDTH := 0 ;
                  PP:= CONTEXTTABLE[PP].NXTEL ;
               END ;
            END •FORM ≤ RECORDS •;
12 :        IF NO=16 THEN INSYMBOL ;
          UNTIL (TERRCL[NO]=2) ∧( NO≠30) ;
                  IF BDISPL NE 0 THEN ADJUST;
                  MAXSIZE := DISPL ; VARPTR :=0 ;
          GOTO 9 ;
```

```
2:   * CASE ◆
             INSYMBOL;
           IF NO≠1 THEN ERROR(6) ELSE
           BEGIN  SRCHREC(NXT) ;
              IF CTPTR NE 0 THEN ERROR(5) ELSE
              BEGIN
                INDEX := INDEX + 1 ;
                P := INDEX ;
                WITH CONTEXTTABLE[P] DO
                BEGIN  JJ:=1 ;
                  REPEAT
                    NAME[JJ]:= AVAL[JJ] ;
                    JJ:=JJ+1 ;
                  UNTIL JJ=10 ;
                  NXTEL:= NXT ; KLASS:= FIELD ;
                  FLOTYPE := 0 ;
                END ; NXT := P ;
              END ;
              INSYMBOL ;
           END  * NO=1  ◆ ;
           IF NO NE 21 THEN ERROR(10) ELSE
           INSYMBOL ;
           IF NO≠ 1 THEN ERROR(11) ELSE
           BEGIN SRCHREC(NEXT) ;
              IF CTPTR=0 THEN SEARCH ;
              IF CTPTR=0 THEN ERROR(12) ELSE
              IF (CONTEXTTABLE[CTPTR].KLASS ≠ TYPES)
                 ∨ (CONTEXTTABLE[CTPTR].FORM > SYMBOLIC)
                THEN ERROR(7) ;
                INSYMBOL ;
                IF NO≠ 31 THEN ERROR(14) ;
                CPTR := CTPTR ;
                IF CPTR NE 0 THEN
                IF CPTR≠ 0 THEN
                IF PACKFLAG THEN
                BEGIN
                     CASE CONTEXTTABLE[CPTR].FORM OF
 NUMERIC :             LL:= CONTEXTTABLE[CPTR].BITS ;
 SYMBOLIC  :           LL:= CONTEXTTABLE[CPTR].BITSIZE ;
                     END ;
                   IF BDISPL + LL > WORDLENGTH THEN ADJUST ;
                   CONTEXTTABLE[P].BITDISPL := BDISPL;
                   CONTEXTTABLE[P].BITWIDTH := LL ;
                   CONTEXTTABLE[P].FLDADDR:= DISPL ;
                   BDISPL := BDISPL+LL ;
                   LASTFLD := P; CASEBITS:= BDISPL ;
```

```
  ENJ ↗  PACKFLAG  ↓ ELSE
     BEGIN
       CONTEXTTABLE[P].FLDADDR := DISPL ;
       CONTEXTTABLE[P].BITWIDTH :=0 ;
     END ;
  CONTEXTTABLE[P].FLOTYPE := CPTR  ;
  DISPL := DISPL + 1 ;
  MINSIZE := DISPL ;
  MAXSIZE := DISPL ;
  NXTC :=0 ; INSYMBOL ;
REPEAT I:= 0 ;
  REPEAT
    IF (NO>2) ∨ (NO=2) ∧ ¬((CL=1)∨(CL=2)) THEN
    BEGIN RECERR(63) ; GOTO 3 END ;
       IF CPTR≠0 THEN
       IF NO=1 THEN
       BEGIN SRCHREC(NEXT) ;
          IF CTPTR=0 THEN SEARCH ;
          IF CTPTR=0 THEN ERROR(12) ELSE
          WITH CONTEXTTABLE[CTPTR] DO
          IF KLASS≠KONST THEN ERROR(42)
          ELSE
           IF (CONTEXTTABLE[CPTR].FORM=SYMBOLIC) ∧
           (CONTYPE≠CPTR) ∨
           (CONTEXTTABLE[CPTR].FORM=NUMERIC) ∧
           (CONTYPE≠INTPTR)  THEN
           ERROR(13) ELSE
           II:=VALUES ;
       END  ↗ NO=1  ↓ ELSE
       IF (CL=1)∧(CONTEXTTABLE[CPTR].FORM≠NUMERIC)
       ∨(CL=2)∧(CPTR≠CHARPTR)  THEN
        ERROR(46) ELSE
       II:=IVAL ;  ↗ CPTR ≠ 0 ↓
       INDEX := INDEX+1 ;
  P := INDEX ;
       WITH CONTEXTTABLE[P] DO
       BEGIN  JJ:=1 ;
         REPEAT
           NAME[JJ] := BLANK ;
           JJ:= JJ+1 ;
         UNTIL  JJ=10 ;
         NXTEL := NXTC ; KLASS := TAGFIELD ;
         TAGVAL := TRUE ;  CASEVAL := II ;
       END ;
       NXTC := P ; I := I+1 ;
       INSYMBOL ;
```

```
      3 :                  IF NO≠ 21  THEN
                              ERROR(10) ELSE
                              INSYMBOL ;
                    UNTIL  NO > 2 ;
                    IF NO=9 THEN  ↗  FIELDLIST  ↓
                    BEGIN
                       IF PACKFLAG  THEN
                       BEGIN  DISPL := MINSIZE - 1 ;
                              BDISPL := CASEBITS ;
                       END  ELSE   DISPL := MINSIZE ;
                       INSYMBOL ; FIELDLIST(MXL,PP,NXT) ;
                       IF NO≠10  THEN
                         ERROR(9) ELSE INSYMBOL ;
                    END ELSE
                    BEGIN  PP:=0 ; MXL:=MINSIZE END ;
                    P:=NXTC ;
                    FOR  I:= I DOWNTO 1 DO
                    WITH CONTEXTTABLE[P]  DO
                    BEGIN  CASESIZE := MXL ; VARIANTS :=PP;
                            P:=NXTEL ;
                    END ; IF MAXSIZE  <MXL THEN
                             MAXSIZE :=MXL ;
                    IF NO≠16 THEN INSYMBOL ;
                  UNTIL NO > 2 ;
                  INDEX := INDEX+1 ;
                  P := INDEX ;
                  WITH CONTEXTTABLE[P] DO
                  BEGIN JJ:=1 ;
                     REPEAT
                       NAME[JJ] := BLANK ;
                       JJ:=JJ+1 ;
                     UNTIL JJ=10 ;
                     NXTEL:= 0 ; KLASS:= TAGFIELD ;
                     CASESIZE := MINSIZE ; VARIANTS := NXTC ;
                     TAGVAL := FALSE ; CASETYPE := CPTR ;
                  END ;
                  VARPTR := P ;
               END ↓  NO=1  ↓ ;
    9 :        NXTF := NXT ;
            END ↗  FIELDLIST  ↓ ;
**********  *********     *********     ********     *********↓
```

```
↱  TYPEDECL ↓
BEGIN
   PACKFLAG := FALSE ;
   IF NO.EQ 53 THEN ↱PACKED ↓
   BEGIN PACKFLAG := TRUE ; INSYMBOL END;
   IF NO=1 THEN
   BEGIN  SRCHREC(NEXT) ;
      IF CTPTR=0 THEN SEARCH ;
      IF CTPTR = 0 THEN
      BEGIN ERROR(12) ;
          P1:=0; SKIPT(16)
      END  ELSE
      BEGIN IF CONTEXTTABLE[CTPTR].KLASS=TYPES THEN
        ↱  TYPE-ID  ↓
         BEGIN TL:=CONTEXTTABLE[CTPTR].SIZE ;
            P1:=CTPTR ; INSYMBOL
         END  ELSE
         IF CONTEXTTABLE[CTPTR].KLASS ≠ KONST  THEN
         SCALDECL(CTPTR) ELSE
         TYPERR(11) ;
      END ;
   END ↱  ID  ↓  ELSE
   IF NO=9 THEN ↱  SYNBOLIC ↓
   BEGIN  CV:=0; LERR:= ERR; ERR:=FALSE ;
      INDEX := INDEX + 1 ;
      P := INDEX ;
      WITH CONTEXTTABLE[P] DO
      BEGIN  JJ:=1 ;
         REPEAT
           NAME[JJ] := BLANK ;
           JJ:= JJ+1 ;
         UNTIL JJ=10 ;
         NXTEL:= 0 ; KLASS:= TYPES ;
         FORM := SYMBOLIC ;
      END ;  RTYP := P ; NXTC := 0 ;
      REPEAT INSYMBOL ;
         IF NO≠1 THEN
         BEGIN ERROR(11) ;
                  SKIPT(15) ; GOTO 2 ;
         END ;
         SRCHREC (NEXT) ;
         IF CTPTR ≠ 0 THEN ERROR(8) ELSE
        BEGIN
        INDEX := INDEX +1 ;
        P := INDEX ;
        WITH CONTEXTTABLE [ P]  DO
```

```
                  BEGIN   JJ:=1 ;
                    REPEAT
                       NAME[JJ] := AVAL[JJ] ;
                        JJ:= JJ+1 ;
                    UNTIL JJ=10 ;
                    NXTEL:= NEXT ;    KLASS:= KONST ;
                    CONTYPE := RTYP ; CONKIND:= ACTUAL ;
                    VALUES:= CV ; SUCC:= NXTC ;
                  END ; CV:=CV+1 ; NEXT:= P; NXTC:=P;
                 END ;
                  INSYMBOL ;
    2 :         UNTIL NO≠15 ;
                WITH CONTEXTTABLE[RTYP] DO
                BEGIN FCONST := NEXT ; SIZE:= 1 ;
                   BITSIZE :=    2 ;
                END ;
                IF ERR  THEN P1:= 0   ELSE
                BEGIN  ERR:=LERR ; P1:= RTYP ;  END ;
                TL:= 1 ;
                IF NO≠ 10 THEN TYPERR(9) ELSE INSYMBOL ;
             END ⌐ SYMBOLIC ↓ ELSE
             IF (NO=2) ∨ (NO=7) THEN  ⌐ SUBRANGE ↓
              BEGIN LERR:=ERR ; ERR:= FALSE ;
                SCALDECL(NEXT) ;
                IF ERR THEN ERROR(6)
                ELSE  ERR:= LERR
             END ⌐ SUBRANGE  ↓ ELSE
             IF NO=41 THEN ⌐STRUCTURED TYPES ↓
               CASE  CL OF
 ⌐ ARRAY ↓ 1 :
                BEGIN INSYMBOL ;
                   IF NO≠11 THEN
                   BEGIN  TYPERR(57) ;
                      IF TERRCL[NO]=1  THEN TYPEDECL(I,CTPTR) ;
                      GOTO 19 ;
                   END ;
                   NXTA := 0 ;
                   REPEAT
             INDEX := INDEX + 1 ;
                 P:= INDEX ;
                     WITH CONTEXTTABLE[P] DO
                      BEGIN  JJ:=1 ;
                         REPEAT NAME[JJ] := BLANK ;
                             JJ:=JJ+1 ;
                         UNTIL JJ=10 ;
                         NXTEL:= 0 ; KLASS:= TYPES ;
```

```
               FORM:= ARRAYS ; AELTYPE := NXTA ;
                 AELTYPE TEMP. LINKS  SJBARRAYS
            END ; NXTA := P ;
            INSYMBOL ;
            LERR:= ERR; ERR:= FALSE ;
            SUBTYPE(I,J,P) ;
            IF ERR  THEN
            BEGIN ERROR(6) ;
                 SKIPT(15) ; I:=0; J:=0; P:=0
            END ELSE
            ERR:= LERR ;
            WITH CONTEXTTABLE[NXTA ] DO
            BEGIN
              IF I LT 127 THEN LO:=I ELSE
                          BEGIN
                            LO:=I DIV 10 +1 ;
                            SZE:=TRUE ;
                          END;
              IF J LT 127 THEN HI:=J ELSE
                          BEGIN
                            HI := J DIV 10 +1 ;
                            SZE := TRUE ;
                          END ;
              INXTYPE:= P ;
            END;
        UNTIL NO≠15 ;
        IF NO≠12 THEN
        BEGIN ERROR(31) ;
             SKIPT(31) ;
            IF TERRCL(NO)=0 THEN GOTO 11 ;
            IF NO=31 THEN
            BEGIN INSYMBOL ; GOTO 11 END ;
            IF NO≠ 12 THEN
            BEGIN P1:=0; TL:=0; GOTO 19 END ;
        END ;
       INSYMBOL ;
        IF NO≠31 THEN ERROR(14) ELSE INSYMBOL;
11:     TYPEDECL(TL,CTPTR) ;
        IF CTPTR≠0 THEN
        IF CONTEXTTABLE[CTPTR].FORM >REGISTERS
         THEN BEGIN ERROR(24) ; CTPTR:=0 END
       ELSE
        BEGIN
              CPTR := CTPTR ;
              REPEAT
                WITH CONTEXTTABLE[NXTA] DO
```

```
                       BEGIN
                         MULOPT(TL,E1,E2,OPT) ;
                         OPTTYP := OPT ;EXP1 :=E1 ;
                             EXP2 := E2 ;
                         TL := TL * (HI-LO+1) ;
                         SIZE := TL ;
                         P := AELTYPE ; AELTYPE := CTPTR ;
                       END ;
                       CTPTR := NXTA ; NXTA := P ;
                     UNTIL  NXTA=0 ;
                     ▸ NOW TL IS THE SIZE OF THE ARRAY,CTPTR
                      POINTS TO IT +
              END ;
              P1:=CTPTR ;
              IF PACKFLAG THEN
              BEGIN
                CTPTR := CONTEXTTABLE[CPTR].AELTYPE ;
                CASE CONTEXTTABLE[CTPTR].FORM OF
     NUMERIC:   TEMP := CONTEXTTABLE[CTPTR].BITS ;
     SYMBOLIC:   TEMP := CONTEXTTABLE[CTPTR].BITSIZE ;
                END ;
                TEMP := WORDLENGTH DIV TEMP ;
                WITH CONTEXTTABLE[P1] DO
                BEGIN
                  IF (SIZE MOD TEMP) EQ 0 THEN SIZE:=SIZE DIV TEMP
                  ELSE SIZE := SIZE DIV TEMP + 1
                END;
              END;
19 :       END ▸ARRAY +  ;
▸ RECORD  +
2:         BEGIN
              INDEX := INDEX+1 ;
              P := INDEX ;
              WITH CONTEXTTABLE[P] DO
             BEGIN  JJ:=1 ;
                REPEAT
                 NAME[JJ]:= BLANK ;
                 JJ:=JJ+1 ;
                UNTIL JJ=10 ;
                NXTEL:= 0 ; KLASS:= TYPES ;
                FORM:= RECORDS ;
              END ; RTYP:= P;
              INSYMBOL ; NXTF:= 0 ;
              DISPL:=0; BDISPL:=0; LERR:=ERR;
              ERR:=FALSE ; FIELDLIST(TL,P,NXTF) ;
              IF NO#26  THEN ERROR(15) ;
```

```
               IF ERR THEN TYPERR(16) ELSE
               WITH CONTEXTTABLE[RTYP] DO
               BEGIN  SIZE:=TL; FSTFLD:= NXTF;
                     RECVAR:=P; P1:=RTYP; ERR:=LERR;
                     INSYMBOL ;
               END ;
            END ρ RECORDS ↓ ;
ρ FILE ↓    3 :
            BEGIN
               WRITE(≡ ≡,≡FILE≡,EOL) ;
            END ;
         END ρ STRUCTURED TYPES ↓
         ELSE
         IF NO=50 THEN  ρ REGISTER ↓
         BEGIN  INSYMBOL ;
            WRITE(≡ ≡,≡REGISTER ≡,EOL) ;
         END
          ρ REGISTER ↓ ELSE TYPERR(16) ;
      END ρ  TYPEDECL↓  ;
ρ*********************************************************↓
```

```
PROCEDURE SEG3 ; (* VARIABLE DECLARATION *)
 VAR
  AT : ADDRESS ;
  I,J,K : SHRINT ;
  JJ : INTEGER ;
  AS : ARRAY [1..10] OF INTEGER ;
 BEGIN
  INSYMBOL ;
  WHILE NO=1 DO
  BEGIN I:=0;
   J:= 0 ;
   REPEAT
    SRCHREC(NEXT) ;
    IF CTPTR NE 0 THEN
    BEGIN
     IF CONTEXTTABLE[CTPTR].SYNCELL THEN
     BEGIN
      CJNTEXTTABLE[CTPTR].SYNCELL := FALSE ;
      J := J+1 ;
      AS[J] := CTPTR ;
      P := CTPTR ;
     END ELSE ERROR(8)
    BND ELSE
    BEGIN
     INDEX := INDEX+1 ;
     P:= INUEX ;
     WITH CONTEXTTABLE[P] DO
     BEGIN
      JJ:= 0 ;
      REPEAT
       JJ:=JJ+1 ;
       NAME[JJ] := AVAL[JJ] ;
      UNTIL JJ=10 ;
      NXTEL := NEXT ;
      KLASS := VARS ; VKIND := ACTUAL ;
      VTYPE := 0 ; VLEVEL := 0 ;
     END ;
     NEXT := P ;
     I := I+1 ;
    END ;
     INSYMBOL ;
     IF NJ EQ 15 THEN
     BEGIN
      INSYMBOL ;
      IF NO NE 1 THEN
      BEGIN ERROR(11); GOTO 10 END ;
```

```
            END ELSE
            IF NJ NE 21 THEN ERROR(10)    ;
        UNTIL NO NE 1 ;
        IF NO EQ 21 THEN INSYMBOL
        ELSE IF NOT(ERR) THEN ERROR(10) ;
        N := P ;
        ERR := FALSE ;
        TYPEDECL(TL,CTPTR) ;
        IF ERR THEN GOTO 10 ;
        LC := LC+(I+J)*TL ;
        LL:=LC ;
        FOR I:= I DOWNTO 1 DO
        BEGIN
          LL:=LL- TL ;
          WITH CONTEXTTABLE[N] DO
          BEGIN
          VTYPE := CTPTR ;
               VLEVEL := LEVEL ; VADDR := LL;
          N := NXTEL ;
          END ;      ↑WITH N ↓
        END ↑ FOR I ↓ ;
        FOR J:= J DOWNTO 1 DO
        BEGIN
          N:=AS[J] ; LL:= LL-TL ;
  1 :     WITH CONTEXTTABLE[N] DO
          BEGIN
            IF SYNPTR EQ 0 THEN GOTO 2
            ELSE
            BEGIN
              KLASS := VARS ; VKIND := ACTUAL ;
              VTYPE := CTPTR ; VLEVEL := 0 ;
                VLEVEL := LEVEL ; VADDR := LL
            END ↑ SYNPYR NE 0 ↓ ;
            N := SYNPTR ; SYNPTR := 0 ;
          END ↑ WITH N ↓ ; GOTO 1 ;
 2:     JJ:= JJ+1 ;
        END ↑ FOR J ↓ ;
10 :      FINDSEMICOLON ;
      END ↑ WHILE NO=1 ↓

    END ; ↑SEG3↓
↑*************************************************************************↓
```

```
PROCEDURE SEG5 ; ↗VALUE DECLARATION ↓
  VAR II: INTEGER ;
  BEGIN
    IF NO EQ 48 THEN ↗ VALUE ↓
    BEGIN
      VALPT:= TRUE ;
          II:=0;
      IF LEVEL NE 0 THEN ERROR(18) ;
      INSYMBOL ;
      WHILE NO EQ 1  DO
      BEGIN
        SRCHREC(NEXT) ;
        IF CTPTR EQ 0 THEN SEARCH ;
        IF CTPTR EQ 0 THEN
        BEGIN
          ERROR(12) ;
          FINOSEMICOLON ;
          GOTO 20
        END ;
        IF CONTEXTTABLE[CTPTR].KLASS NE VARS THEN
        BEGIN
          ERROR(22) ; FINOSEMICOLON ;
          GOTO 20
        END;
        WITH CONTEXTTABLE[CTPTR] DO
        BEGIN
          IT := VADOR ;
          IF VALPT THEN
          BEGIN
          IF VLC EQ 0 THEN
          BEGIN
            VLC := IT ; LL:= IT ; AT := IT
          END ;
          END ;
          IF IT LT LL THEN  ERROR(20) ELSE
          WHILE IT GT AT DO
          BEGIN  II:=II+1; VARB[II]:=0; END ;
          INSYMBOL ;
          IF (NO NE 8 ) OR ( CL NE 6) THEN  ERROR(4)
          ELSE  INSYMBOL ;
          IF NO EQ 9 THEN ↗LIST↓
          BEGIN
            REPEAT
              INSYMBOL ; INCONST(I,PT,NEXT) ; IT1:= 1 ;
              IF (NO EQ 6) AND (CL EQ 1) THEN ↗*↓
              BEGIN
```

```
                IT1:= I ; INSYMBOL ;
                INCONST(I,PT,NEXT) ;
             END ;

             IF NOT(ERR) THEN
             BEGIN
                FOR J:=1 TO IT1 DO
                BEGIN II:=II+1; VARB[II]:=I END ;
                AT := AT + IT1 ;
             END ↑ERR ↓;
             IF (NO NE 15) AND (NO NE 10) THEN
             ↑ NOT,NOR) ↓
             BEGIN
                ERROR(20) ;
                FINDSEMICOLON ;
                GOTO 20
             END ;
          UNTIL NO= 10 ;
          INSYMBOL ;
       END ELSE
       BEGIN
          INCONST(I,PT,NEXT) ;
          IF NOT(ERR) THEN
          BEGIN
             II:=II+1; VARB[II]:=I ;
             AT := AT+ 1 ;
          END ;
       END ;
       LL := II + CONTEXTTABLE[JTYPE].SIZE ;
       FINDSEMICOLON ;
     END ↑ WITH ↓ ;
   VALPT:=FALSE;
20 :   END ↑ WHILE NO=1 ↓ ;
   VALPT:=TRUE;
   END ↑ VALUE↓ ELSE VLC := LC ;
 END ; ↑ SEGS ↓
****************************************************************************
```

```
↑*************************************************************************↓
      PROCEDURE GENRE(A,B:INTEGER; C : RTYP; D:BOOLEAN) ;
       ↑ PUTS THE CODE GENERATED IN GLOBAL VARIABLE--HCODE↓
        BEGIN
          IC:=IC+1; HCA:=HCA+1;
            WITH HPAS[HCA] DO
          BEGIN
            LOC:=IC; INST:=A; ADRR:=B;  TYP:=C; MREF:=D
          END
        END ↑ GENRE↓ ;
↑*************************************************************************↓
```

```
      PROCEDURE RLEXP ;
        ρ GENERATES CODE FOR RELATIONAL OPERATORS ↓
        BEGIN
          CASE LADOPCL OF
  1: •LT•   BEGIN
              GENRE(2021B,0,0,FALSE) ;
              GENRE(2003B,0,0,FALSE) ;
              GATTR.JMP := 0
            END ;
  2: •LE•   BEGIN GENRE(2021B,0,0,FALSE); GATTR.JMP:=1 END ;
  3: •GE•   BEGIN GENRE(2022B,0,0,FALSE); GATTR.JMP:=1 END ;
  4: •GT•   BEGIN GENRE(2021B,0,0,FALSE) ; GATTR.JMP :=0 END ;
  5: •NE•   BEGIN GENRE(2003B,0,0,FALSE) ; GATTR.JMP:=0 END ;
  6: •EQ•   BEGIN GENRE(2002B,0,0,FALSE); GATTR.JMP:=1 END ;
            END ρ CASE↓ ;
            GENRE(2401B,0,0,FALSE) ; GENRE(2404B,0,0,FALSE) ;
        END ρ RLEXP↓ ;
ρ********************************************************↓
```

```
    PROCEDURE LOTMP ;
      VAR  RLX : BOOLEAN ;
         DLX: BOOLEAN ;
       BEGIN
         IF RP EQ 0 THEN GOTO 1 ;
         STY := FALSE ;
         RLX := FALSE ;
         DLX := FALSE ;
         WITH STRE[RP] DO
         BEGIN OP:= STOP ; LADOPCL := STCL ; END ;
         IF OP EQ 7 THEN
         BEGIN
           CASE LADOPCL OF
  1:          GENRE(042000B,0,1,TRUE) ;        ↑ADA↓
  2:          BEGIN
              GENRE(002004B,0,0,FALSE) ; ↑ CMA,INA↓
              GENRE(042000B,0,1,TRUE) ; ↑ ADA ↓
              END ;
  3:          GENRE(032000B,0,1,TRUE) ;    ↑IOR↓
           END ↑ CASE CL OF ↓
         END ↑ NO EQ 7 ↓ ELSE
         IF OP EQ 6 THEN
         BEGIN
           CASE LADOPCL OF
  1:          BEGIN
                GENRE(100200B,0,0,FALSE); ↑MPY↓
                GENRE(0,0,1,FALSE) ;  ↑LOCATION FOR MPY↓
              END ;
  2:          ;
  3:          BEGIN GENRE(012000B,0,1,TRUE) ↑AND↓ END ;
  4:          BEGIN
                GENRE(72000B,BFLC+4,1,TRUE); ↑STA↓
                GENRE(62000B,0,1,TRUE); ↑LDA↓
                DLX:=TRUE;
              END ;
  5:          ;
           END ↑ CASE CL OF ↓
         END ↑ NO EQ 6 ↓ ELSE
         IF OP EQ 8 THEN
         BEGIN
           GENRE(003004B,0,0,FALSE) ; ↑CMA,INA↓
           GENRE(042000B,0,1,TRUE) ; ↑ ADA↓
           RLX := TRUE ;
         END ELSE ERROR(54) ;
         KP := KP+1 ;
         IF KP GT KPMAX THEN BEGIN ERROR(48) END ;
```

```
          WITH STORE[KP] DO
          BEGIN STRP:=RP; STPL:= HCA END ;
      *   IF RLX THEN
          BEGIN
            RLX:=FALSE; GENRE(3004B,0,0,FALSE); ↑CHA,INA↓
          END ;
          IF DLX THEN
          BEGIN
            DLX:=FALSE;
            GENRE(6400B,0,0,FALSE); ↑CLB↓
            GENRE(2020B,0,0,FALSE); ↑SSA↓
            GENRE(7004B,0,0,FALSE); ↑CMB,INB↓
            GENRE(100400B,0,1,FALSE); ↑DIV↓
            GENRE(BFLC+4,0,1,FALSE) ; ↑OP FOR DIV↓
          END ;
            RP := RP-1 ;
    18     END ↑ LDTMP↓ ;
↑**********     *********      **********      *********      *********↓
        PROCEDURE STTMP ;
          BEGIN
            CRY := TRUE ;
          RP:=RP+1 ;
          IF RP GT RPMAX THEN RPMAX := RP ;
          KP := KP+1 ;
          IF KP GT KPMAX THEN BEGIN ERROR(24); GOTO 1 END ;
          GENRE(072000B,0,1,TRUE) ; ↑ STA ↓
          WITH STORE[KP] DO
          BEGIN STRP := RP ; STPL := HCA  END ;
          WITH STRE[RP] DO
          BEGIN STOP:= OP; STCL:= LADOPCL END ;
    18     END ↑ STTMP ↓ ;
↑*************************************************************************↓
```

```
        ↑ CALLED AT PROCEDURE END , UPDATING ITS CODE (INSERTING
          ADRESSES OF THE CONSTANTS ) AND WRITING OUT CODE TOGETHER
          WITH THE CONSTANTS USED IN THIS PROCEDURE ↓
↑**********      *********         *********       ********      ********↓
      PROCEDURE WRIEM ;
        VAR FLC: INTEGER ;
        BEGIN
          FLC := IC ;
          FOR I:=1 TO RPMAX DO IC:=IC+1 ;
          FOR I:=1 TO KP DO
          WITH STORE[I] DO  HPAS[STPL].ADRR := FLC+STRP ;
            HPAS[3].ADRR := IC+1 ;
          IC:=IC+72;↑STORAGE FOR BUFFER↓
          LC:=IC;
        END ↑ WRIEM ↓ ;
↑**********      *********         *********       ********      ********↓
      PROCEDURE  WRITOUT ;
        VAR IT2 : SHRINT ;
        BEGIN
          FOR IT :=1 TO LCX DO
          WITH CSTTB[IT] DO
          BEGIN
            GENRE(VALU,0,0,FALSE) ; ↑DEC↓
            IT1 := INX ;
            REPEAT
              WITH UNDLAB[IT1] DO
              BEGIN
                WITH HPAS[PLACE] DO ADRR:= IC ;
                IT2:=IT1; IT1:=SUCC
              END
            UNTIL IT1 EQ 0 ;
            UNDLAB[IT2].SUCC := CHNIX ; CHNIX := INX
          END ↑ WITH CSTTB ↓
        END ↑ WRITOUT↓ ;
↑************************************************************************↓
```

```
PROCEDURE LOCST(FVAL: INTEGER) ;
  ρ ENTER CONSTANTS WITH VALUE FVAL INTO CONSTANT TABLE CSTTB
  IFF NOT YET PRESENT ,ELSE CHAINS OCCURENCE OF FVAL IN CODE
   (THROUGH UNDLAB) +
  VAR IT: SHRINT ;
  BEGIN
    FOR IT := 1 TO LCX DO
    WITH CSTTB[IT] DO
    IF VALU = FVAL THEN
    BEGIN
      IF CHNIX=0 THEN ERROR(47) ELSE
      BEGIN
        WITH UNDLAB[CHNIX] DO
        BEGIN
          IT1:=SUCC; SUCC:=INX; INX:=CHNIX;PLACE:=HCA
        END;
        CHNIX := IT1
      END ;
      GOTO 10
    END;
    IF LCX EQ CSIMAX THEN BEGIN ERROR(52); GOTO 10 END;
    IF CHNIX EQ 0 THEN BEGIN ERROR(47) ; GOTO 10 END;
    LCX :=LCX+1 ;
    WITH CSTTB[LCX] DO
    BEGIN
      VALU:=FVAL; INX:=CHNIX
    END ;
    WITH UNDLAB[CHNIX] DO
    BEGIN IT1:=SUCC; SUCC:=0; PLACE:=HCA END;
    CHNIX :=IT1;
  10:    END ρ LOCST + ;
ρ*********************************************************************+
```

```
     PROCEDURE ADDRESSVAR(FCTP:INTEGER; VAR FATTR:ATTR) ;
↑BUILD UP ATTRIBUTE IN FATTR AND VARIABLE IS POINTED TO BY FCTP↓
       BEGIN
          WITH CONTEXTTABLE[FCTP],FATTR DO
         BEGIN
           KIND := VARBL;
           IF KLASS  = VARS THEN
           BEGIN
             TYPTR := VTYPE; PCKD := FALSE ;
             IF VKIND = ACTUAL THEN
             BEGIN
                 ACCESS:=DRCT; DPLMT:=VADDR+BLC+BFLC+1 ;
               BREG := VLEVEL
             END ELSE
             BEGIN
               WRITE(≡ ≡,≡FORMAL≡) ;
             END
           END ELSE
           IF KLASS = FIELD THEN
           BEGIN
             TYPTR := FLDTYPE ;
             WITH DISPLAY[DISX] DO
             BEGIN
               IF OCCUR=CWITH THEN
               BEGIN
                 ACCESS := DRCT ; BREG := CLEVEL ;
                 DPLMT := FLDADDR + CDISP
               END ELSE
               BEGIN
                 WRITE(≡ ≡,≡FORMAL≡)
               END ;
               IF BITWIDTH NE 0 THEN
               BEGIN
                 PCKD := TRUE; BITADR := BITDISPL ;
                 BITSIZE := BITWIDTH
               END ELSE  PCKD := FALSE
             END ↑ DISPLAY↓
           END ↑ FIELD ↓ ELSE
           BEGIN ↑ PROCEDURE ↓
             TYPTR := PROCTYPE ; ACCESS := DRCT ;
             BREG := PROCLEVEL + 1 ; DPLMT := 2 ; PCKD:= FALSE ;
             IF PROCKIND=FORMAL THEN ERROR(62)
             ELSE IF LEVEL NE  BREG THEN ERROR(52)
           END
         END ↑ WITH FCTP,FATTR ↓
       END ↑ ADDRESSVAR↓ ;
```

```
************************************************************************
        PROCEDURE STATEMENT  ! FORWARD !
       PROCEDURE EXPRESSION  ! FORWARD !
************************************************************************
```

```
PROCEDURE VARIABLE ;
  VAR LATTR : ATTR ;
    TEMP,TEMP1: INTEGER ;
  BEGIN
    ADDRESSVAR(CTPTR,LATTR) ;
    INSYMBOL ;
    IF NO EQ 17 ↑.↓ THEN
    BEGIN
      INSYMBOL ;
      IF NO EQ 1 THEN ↑ID↓
      BEGIN
        IF LATTR.TYPTR NE 0 THEN
        BEGIN
          TEMP := LATTR.TYPTR ;
          IF CONTEXTTABLE[TEMP].FORM EQ RECORDS THEN
          WITH LATTR DO
          BEGIN
            TEMP := CONTEXTTABLE[TYPTR].FSTFLD ;
            SRCHREC(TEMP) ;
            IF CTPTR EQ 0 THEN
            BEGIN ERROR(27); CTPTR := UNDECPTR END ;
            WITH CONTEXTTABLE[CTPTR] DO
            BEGIN
              TYPTR := FLDTYPE ;
              DPLMT := DPLMT+FLDADDR ;
              IF BITWIDTH NE 0 THEN
              BEGIN
                PCKD := TRUE ; BITADR := BITDISPL;
                BITSZ :=BITWIDTH;
              END ELSE PCKD := FALSE
            END
          END ↑WITH LATTR↓ ELSE
          BEGIN
            ERROR(26) ; LATTR.TYPTR := 0
          END
        END ↑ TYPTR NE 0 ↓ ELSE
        BEGIN ERROR(28); LATTR.TYPTR:=0 END;
        INSYMBOL
      END ↓ NO=1 ↓
    END ↑ IF NO=17 ↓ ELSE
    IF NC EQ 11 ↑[↓ THEN
    BEGIN
      GENRE(6400B,0,0,FALSE) ; ↑ CLB ↓
      GENRE(76000B,BFLC+4,1,TRUE) ; ↑ STB IN BFLC+4 ↓
      REPEAT
        WITH LATTR DO
```

```
            IF TYPTR NE 0 THEN
            IF CONTEXTTABLE[TYPTR].FORM NE ARRAYS THEN
            BEGIN  ERROR(63); TYPTR:=0 END ;
            INSYMBOL;
            IF CTY THEN EXPRESSION ELSE BEGIN STTMP;
                        EXPRESSION END ;
            IF GATTR.TYPTR NE 0 THEN
            BEGIN
              TEMP:= GATTR.TYPTR ;    TEMP1:= LATTR.TYPTR ;
             IF CONTEXTTABLE[TEMP].FORM GT SYMBOLIC THEN ERROR(64);
              IF TEMP1 NE 0 THEN
              BEGIN
                PT:= CONTEXTTABLE[TEMP1].INXTYPE;
                 IF ((CONTEXTTABLE[TEMP].FORM EQ SYMBOLIC )
                     OR (CONTEXTTABLE[TEMP1].FORM EQ SYMBOLIC))
                     AND (TEMP NE PT) THEN ERROR(31) ;
                WITH GATTR DO
                IF KIND EQ SVAL THEN
                BEGIN
                  IF (CONTEXTTABLE[TEMP1].LO GT VAL)
                     OR (CONTEXTTABLE[TEMP1].HI LE VAL)
                     THEN ERROR(57) ;
                END ;
                WITH LATTR,CONTEXTTABLE[TYPTR] DO
                BEGIN
                  TEMP:= CONTEXTTABLE[AELTYPE].SIZE;
                  GENRE(100200B,0,0,FALSE) ; ⌐ MPY↓
                  GENRE(0,0,1,FALSE); ⌐ BY SIZE↓
                  LOCST(TEMP);
                  DPLMT := DPLMT - LO * TEMP + 10000B ;
                  GENRE(42000B,0,1,TRUE); ⌐ ADA DISPLACEMENT ↓
                  LOCST(DPLMT) ;
                  GENRE(42000B,BFLC+4,1,TRUE) ; ⌐ ADA TO BFLC+4 ↓
                  GENRE(72000B,BFLC+4,1,TRUE) ; ⌐ STA IN BFLC+4 ↓
                  IF ACCESS EQ DRCT THEN ACCESS := INXD ;
                END ⌐ WITH LATTR ↓
              END ⌐ LATTR.TYPTR NE 0 ↓
            END ⌐ GATTR.TYPTR NE 0 ↓ ;
            IF LATTR.TYPTR NE 0 THEN
            LATTR.TYPTR := CONTEXTTABLE[TEMP1].AELTYPE ;
          UNTIL NO NE 15 ; ⌐ ] ↓
            IF NO NE 12 THEN ⌐ ] ↓ ERROR(25) ELSE INSYMBOL ;
        CTY := TRUE ;
        END ⌐ IF NO EQ 11 ↓ ;
        GATTR:=LATTR
      END⌐VARIABLE↓ ;
```

```
           *GATTR CONTAINS THE ATTRIBUTE OF A VARIABLE *
************************************************************************
```

```
    PROCEDURE PRIMARY ;
      BEGIN
        IF NO EQ 1 THEN
        BEGIN
          SRCHREC(NEXT) ;
          IF CTPTR EQ 0 THEN
          BEGIN ERROR(31); CTPTR := UNDECPTR END ;
          CASE CONTEXTTABLE[CTPTR].KLASS OF
TYPES:      BEGIN ERROR(45);
                  GATTR.TYPTR:=0;INSYMBOL
            END;
KONST:      WITH GATTR,CONTEXTTABLE[CTPTR] DO
            BEGIN
              TYPTR:=CONTYPE;
              IF CONKIND=ACTUAL THEN
              BEGIN
              KIND :=SVAL; VAL := VALUES
              END ELSE
              BEGIN
                KIND := VARBL;ACCESS:=DRCT; BREG:=CLEVEL;
                DPLMT:=CADDR; PCKD:=FALSE
              END;
              INSYMBOL;
            END;
PROC:       ;
VARS,FIELD:  VARIABLE;
          END ◆CASE KLASS OF◆ ;
        END ◆IF NO=1 ◆ ELSE
        IF NO EQ 2 THEN
        BEGIN
          WITH GATTR DO
          BEGIN
            KIND:=SVAL; CTY:=TRUE; VAL:=IVAL  ;
          CASE GL OF
1:          TYPTR:=INTPTR;
2:          TYPTR:= CHARPTR;
          END ◆ CASE ◆ ;
          END ◆ GATTR◆ ;
          INSYMBOL ;
        END ◆IF NO=2◆ ELSE
        IF NO EQ 9 THEN
        BEGIN
          WHILE NO EQ 9 DO INSYMBOL ;
          IF CTY THEN EXPRESSION  ELSE
          BEGIN SIIMP; EXPRESSION END ;
          IF NO NE 10 THEN ERROR(7) ELSE
```

```
            BEGIN
              LOTMP ;
              IF RP EQ 0 THEN BEGIN WHILE NO EQ 10 DO INSYMBOL END
                          ELSE INSYMBOL ;
            END ;
         END  ↑IF NO=9↓ ELSE
         BEGIN ERROR(29); GATTR.TYPTR:=0 END
       END ↑PRIMARY↓ ;
************************************************************↓
```

```
          PROCEDURE  ASSMB(A,B,C: INTEGER) ;
            ≠ PRODUCES CODE FOR THE SHIFT OPERATOR↓
            BEGIN
              IF A EQ 3 THEN
              BEGIN
                GENRE(B,0,0,FALSE);
                GENRE(C,0,0,FALSE);
              END ELSE
              IF A EQ 2 THEN GENRE(B,0,0,FALSE) ELSE
              IF A EQ 1 THEN GENRE(C,0,0,FALSE) ;
            END ≠ ASSMB↓ ;
≠*********        *********        *********        ********        ********↓
          PROCEDURE MODE(A,B:INTEGER; VAR C,D: INTEGER) ;
            ≠ FINDS C:=A MOD B AND D := A DIV B↓
            BEGIN
              C:= A MOD B;
              D:= A DIV B ;
            END ≠ MODE ↓ ;
≠*********        *********        *********        ********        ********↓
```

```
PROCEDURE SYSOP ;
   VAR TEMP,TEMP1,I,J,II: INTEGER ;
   BEGIN
      OP:=NO; LADOPCL:=CL; INSYMBOL;
      IF (LADOPCL EQ 11) OR (LADOPCL EQ 12) THEN
      BEGIN
         RATTR.TYPTR := GATTR.TYPTR;
         IF NO NE 1 THEN ERROR(28) ELSE
         BEGIN
            PRIMARY;
            IF (RATTR.TYPTR NE 0) AND (GATTR.TYPTR NE 0) THEN
            BEGIN
               IF RATTR.TYPTR EQ GATTR.TYPTR THEN
               BEGIN
                  WITH GATTR DO
                  BEGIN
                     CASE KIND OF
VARBL:               BEGIN
                        IF LADOPCL EQ 11 THEN
                        GENRE(32000B,DPLMT,1,TRUE) ↑ IOR↓ ELSE
                        GENRE(22000B,DPLMT,1,TRUE) ↑ XOR↓ ;
                     END;
SVAL:                BEGIN
                        IF LADOPCL EQ 11 THEN
                        GENRE(32000B,0,1,TRUE) ↑ IOR↓ ELSE
                        GENRE(22000B,0,1,TRUE) ↑ XOR↓ ;
                        LDCST(VAL) ;
                     END;
                     END ↑ CASE ↓;
                  END ↑ WITH GATTR↓
               END ELSE ERROR(45) ;
            END ↑TYPTR NE 0↓ ELSE ERROR(32) ;
         END ↑ NO EQ 1↓
      END ↑ CL=11 OR CL=12↓ ELSE
      BEGIN
         IF (NO NE 2) AND (CL NE 1) THEN ERROR(33) ELSE
         BEGIN
            IF (LADOPCL EQ 9) OR (LADOPCL EQ 10) THEN
            BEGIN
               MODE(IVAL,4,TEMP,TEMP1) ;
               IF LADOPCL EQ 9 THEN ↑ALF↓
               ASSMB(TEMP,1727B,1700B) ELSE
               ASSMB(TEMP,5727B,5700B) ;
            END ↑ CL=9 OR CL=10 ↓ ELSE
            BEGIN
               MODE(IVAL,16,TEMP,TEMP1) ;
```

```
                    IF (LADOPCL EQ 7) OR (LADOPCL EQ 8) THEN
                    BEGIN
                      MODE(TEMP,4,I,J) ;
                      IF LADOPCL EQ 7 THEN
                      BEGIN  ASSMB(J,1727B,1700B) ↑ALE↓ ;
                             ASSMB(I,5727B,5700B) ↑RAL↓
                      END ELSE ↑ CL=8↓
                      BEGIN
                        GENRE(33000B,0,0,FALSE); ↑LDB 0↓
                        ASSMB(J,5727B,5700B) ; ↑ BLF↓
                        ASSMB(I,5222B,5200B) ; ↑ RBL↓
                      END ;
                    END ↑ CL=7 OR CL=8↓    ELSE
                    BEGIN
                      MODE(TEMP,2,I,J) ;
                      FOR II:=1 TO J DO
                      BEGIN
                        CASE LADOPCL OF
1:                        GENRE(1020B,0,0,FALSE) ; ↑ALS,ALS↓
2:                        BEGIN
                            GENRE(33000B,0,0,FALSE); ↑LDB 0↓
                             GENRE(5020B,0,0,FALSE); ↑BLS,BLS↓
                          END;
3:                        GENRE(1121B,0,0,FALSE); ↑ARS,ARS↓
4:                        BEGIN
                            GENRE(33000B,0,0,FALSE); ↑LDB 0↓
                            GENRE(5121B,0,0,FALSE) ; ↑BRS,BRS↓
                          END ;
5:                        GENRE(1323B,0,0,FALSE); ↑ RAR,RAR↓
6:                        BEGIN
                            GENRE(33000B,0,0,FALSE) ;
                            GENRE(5323B,0,0,FALSE) ; ↑ RBR,RBR↓
                          END ;
                        END ↑ CASE ↓ ;
                      END ↑ FOR ↓ ;
                    END ↑ CL=1,2,3,4,5,6↓ ;
                  END ↑CL=9 OR CL=10 ↓
                END ↑ NO EQ 2 AND NO EQ 1 ↓
              END ↑ SHIFT OPERATOR ↓ ;
            END ↑ SYSOP ↓ ;
   ************************************************************
```

```
PROCEDURE EXPRESSION ;
   VAR LFG:BOOLEAN;
   3T1,BT2 : BOOLEAN ;
   BEGIN
     LFG:=FALSE;
     IF NO EQ 7 THEN ↗ADDOP↓
     BEGIN
        OP:=NO;
        IF CL EQ 2 THEN LFG:=TRUE
        ELSE IF CL EQ 3 THEN ERROR(33);
        INSYMBOL ;
     END ;
     IF NO EQ 5 THEN  ↗NOT↓
     BEGIN
        OP := NO; IF CL EQ 1 THEN LFG:= TRUE ELSE ERROR(33);
        INSYMBOL;
     END ;
     PRIMARY ;
     IF CTY THEN
     BEGIN
     IF  GATTR.TYPTR NE 0 THEN
     BEGIN
        CTY:= FALSE ;
        WITH GATTR DO
        BEGIN
          CASE KIND OF
 SVAL:    BEGIN
             GENRE(062000B,0,1,TRUE) ; ↗LDA↓
                LOCSTI(JAL) ; CTY := FALSE ;
          END;
 VARBL:      BEGIN
               IF ACCESS EQ ORCT THEN GENRE(62000B,CPLMT,1,TRUE)
               ELSE GENRE(162000B,BFLC+4,1, TRUE) ;
             END ;
          END ↗ CASE ↓
        END↗GATTR↓
     END ↗ TYPTR↓ ;
     END ↗ STY↓ ;
     IF LFG THEN
     BEGIN
        IF OP EQ 5 THEN GENRE(3000B,0,0,FALSE) ↗CMA↓ ELSE
        GENRE(003004B,0,0,FALSE) ; ↗ CMA,INA↓
     END ;
     LATTR := GATTR;
     WHILE (NO EQ 7) OR (NO EQ 6) OR (NO EQ 8) OR (NO EQ 24) DO
     BEGIN
```

```
        WHILE  (NO EQ 7) OR (NO EQ 6) DO
        BEGIN
          OP:=NO; LADOPCL:=CL;
          INSYMBOL ;
          PRIMARY ;
          IF STY THEN
          BEGIN
          IF (LATTR.TYPTR NE 0) AND (GATTR.TYPTR NE 0) THEN
          BEGIN
            TEMP:=LATTR.TYPTR ;
            BT1:=CONTEXTTABLE[TEMP].FORM = NUMERIC ;
            TEMP:= GATTR.TYPTR ;
            BT2 := CONTEXTTABLE[TEMP].FORM = NUMERIC ;
            IF BT1 AND BT2 THEN
            BEGIN
              WITH GATTR DO
              BEGIN
                CASE KIND OF
VARBL :           BEGIN
                   IF OP EQ 7 THEN
                   BEGIN
                     CASE LADOPCL OF
1:                      BEGIN
                          IF ACCESS EQ ORCT THEN
                          GENRE(42000B,DPLMT,1,TRUE) ELSE
                          BEGIN
                            GENRE(16200B,BFLC+4,1,TRUE); ↑LDA↓
                            LDTMP ;
                          END;
                        END ;
2:                      BEGIN
                          IF ACCESS EQ ORCT THEN
                          BEGIN
                            GENRE(66000B,DPLMT,1,TRUE);↑ LDB↓
                            GENRE(7000B,0,1,FALSE); ↑CMB,INB↓
                            GENRE(40001B,0,0,FALSE) ; ↑ADA TO B ↓
                          END ELSE
                          BEGIN
                            GENRE(162000B,BFLC+4,1,TRUE); ↑LDA ↓
                            LDTMP ;
                          END;
                        END ;
3:               ;
                     END↑CASE CL↓
                   END ↑NO=7↓ ELSE
                   BEGIN
```

```
                    CASE LADOPCL OF
   1:                  BEGIN
                          IF ACCESS EQ DRCT THEN
                          BEGIN
                             GENRE(100200B,0,0,FALSE);↗MPY↓
                             GENRE(DPLMT,0,1,FALSE);↗AD.FOR MPY↓
                             GENRE(30001B,0,0,FALSE); ↗IOR↓
                          END ELSE
                          BEGIN
                             GENRE(162000B,DFLC+4,1,TRUE);↗LDA↓
                             LDTMP;
                          END ;
                       END ;
   2:           ;
   3:                  ERROR(32) ;
   4:                  BEGIN
                          IF ACCESS EQ DRCT THEN
                          BEGIN
                             GENRE(6400B,0,0,FALSE);↗CLB↓
                             GENRE(2020B,0,0,FALSE) ; ↗SSA↓
                             GENRE(7004B,0,0,FALSE); ↗CMB,INB↓
                             GENRE(100400B,0,0,FALSE);↗ DIV↓
                             GENRE(DPLMT,0,1,FALSE); ↗OP. FOR DIV↓
                          END ELSE
                          BEGIN
                             GENRE(162000B,0,1,TRUE); ↗LDA↓
                             LDTMP ;
                          END ;
                       END ;
   5:                  ;
                    END↗CASE CL↓
                 END↗NO NE 7↓;
              END ↗VARBL↓ ;
   SVAL:      BEGIN
                 IF OP EQ 7 THEN
                 BEGIN
                    CASE LADOPCL OF
   1:                  BEGIN GENRE(042000B,0,1,TRUE);    ↗ADA↓
                             LDCST(VAL)
                       END ;
   2:                  BEGIN
                          GENRE(66000B,0,1,TRUE); ↗LDB↓ LDCST(VAL);
                          GENRE(007004B,0,0,FALSE); ↗CMB,INB↓
                       GENRE(040 01B,0,0,FALSE) ; ↗ ADA TO B ↓
                       END ;
   3:                  ERROR(32) ;
```

```
                  END ↱ CASE CL ↓
                END ↱ NO=7↓ ELSE
                BEGIN
                  CASE LADOPCL OF
    1:                BEGIN
                        GENRE(100200B,0,0,FALSE); ↱MPY↓
                        GENRE(0,0,1,FALSE); ↱OPERAND FOR MPY ↓
                        LDCST(VAL) ;
                        GENRE(030001B,0,0,FALSE) ; ↱IOR↓
                     END;
    2:               ;
    3:               ERROR(32) ;
    4:                BEGIN
                        GENRE(006400B,0,0,FALSE) ; ↱CLB↓
                        GENRE(002020B,0,0,FALSE); ↱SSA↓
                        GENRE(007004B,0,0,FALSE) ; ↱CMB,INB↓
                        GENRE(100400B,0,0,FALSE) ; ↱DIV↓
                        GENRE(0,0,1,FALSE); ↱OPERAND FOR DIV ↓
                        LDCST(VAL) ;
                      END ;
    5:               .
                  END ↱ CASE CL ↓
                END ↱ NO NE 7 ↓ ;
              END ↱ SVAL↓ ;
              END ↱ CASE KIND OF ↓;
             END ↱ GATTR ↓ ;
           END ↱ BT1 AND BT2 ↓ ELSE
           BEGIN
             IF LATTR.TYPTR = GATTR.TYPTR THEN
             BEGIN
               IF (GATTR.TYPTR=BOOLPTR) AND (LADOPCL EQ 3 ) THEN
               BEGIN
                 WITH GATTR DO
                 BEGIN
                   CASE KIND OF
    VARBL :          BEGIN
                       IF OP EQ 7 THEN
                       BEGIN
                      GENRE(032000B,DPLMT,1,TRUE) ; ↱IOR↓
                       END ELSE
                       BEGIN
                      GENRE(012000B,DPLMT,1,TRUE) ; ↱ AND ↓
                       END;
                     END;
    SVAL :           BEGIN
                       IF OP EQ 7 THEN
```

```
                          BEGIN
                          GENRE(032000B,U,1,TRUE) ; ↱IOR↓
                          LOCSE(VAL) ;
                          END ELSF
                          BEGIN
                         GENRE(012000B,0,1,TRUE) ; ↱AND ↓
                         LOCST(VAL) ;
                          END
                        END ↱ SVAL↓ ;
                           END↱ CASE KIND OF ↓
                     END ↱ GATTR↓
                   END ↱ BOOLPTR↓  ELSE ERROR(50)
                 END  ↱TYPTR=TYPTR↓ ELSE ERROR(50)
               END;
           END ↱ TYPTR NE 0 ↓ ;
           STY := TRUE ;
           END ↱ STY=TRUE ↓ ;
         END ↱ NO=7 OR NU=6 ↓ ;
         IF NO EQ 8 THEN ↱RELOP↓
         BEGIN
           OP:=NO; LADOPCL := CL ;
           WITH RATTR DO
           BEGIN  TYPTR:= GATTR.TYPTR; KIND:=LCOND END ;
           INSYMBOL ; IF NO EQ 9 THEN EXPRESSION ELSE PRIMARY ;
           IF STY THEN BEGIN
           IF (RATTR.TYPTR NE 0 ) AND (GATTR.TYPTR NE 0 ) THEN
           BEGIN
             IF RATTR.TYPTR EQ BOOLPTR THEN RATTR.ARITH := FALSE
                     ELSE RATTR.ARITH := TRUE ↱ ARITHMETIC ↓ ;
             IF RATTR.TYPTR EQ GATTR.TYPTR THEN
             BEGIN
               WITH GATTR DO
               BEGIN
                 CASE KIND OF
VARBL:          BEGIN
                 IF ACCESS EQ DRCT THEN
                 BEGIN
                     GENRE(003004B,0,0,FALSE) ; ↱ CMA,INA↓
                     GENRE(042000B,DPLMT,1,TRUE) ; ↱ ADA ↓
                 END ELSE
                 BEGIN
                   GENRE(162000B,BFLC+4,1,TRUE) ;
                   LDTMP
                 END ;
                END ;
  SVAL:            BEGIN
```

```
                    GENRE(003004B,0,0,FALSE) ; ↱ CHA,INA↓
                    GENRE(042000B,0,1,TRUE) ; ↱ ADA ↓
                    LOCST(VAL) ;
                  END ;
                END ↱ CASE KIND OF ↓ ;
                RLEXP;
              END ↱ WITH GATTR ↓ ;
            END ELSE ERROR(45) ;
          END ELSE ERROR(32) ;
                      END ↱ STY ↓ ;
                      STY := TRUE ;
          RATTR.TYPTR := BOOLPTR ;
          GATTR:= RATTR ;
        END ↱ NO=8 ↓ ;
        IF NO EQ 24 THEN BEGIN SYSOP; INSYMBOL END ;
        END  ↱ NO=7 OR NO=6 OR NO=8 OR NO=24↓ ;
      END ↱ EXPRESSION↓ ;
↱************************************************************↓
```

```
PROCEDURE ASSIGN ;
  BEGIN
    IF NO EQ 1 THEN SRCHREC(NEXT) ELSE
    BEGIN ERROR(33); SKIP(53) END ;
    VARIABLE ; AATTR := GATTR ;
    IF NO NE 22 THEN ↑1=↓
    BEGIN
      IF GATTR.TYPTR NE 0 THEN ERROR(34);
      SKIP(22);
      IF NO NE 22 THEN
      BEGIN
        IF GATTR.TYPTR EQ 0 THEN ERROR(34) ;
        GOTO 1
      END;
    END ;
    WITH AATTR DO
    IF ACCESS EQ INXD THEN GENRE(72000B,BFLC+5,1,TRUE);
    INSYMBOL ; EXPRESSION ;
    IF GATTR.TYPTR EQ 0 THEN SKIP(53) ELSE
    IF AATTR.TYPTR NE 0 THEN
    BEGIN
      IF AATTR.TYPTR NE GATTR.TYPTR THEN ERROR(35)
      ELSE BEGIN
      WITH AATTR DO
      BEGIN
        IF ACCESS EQ INXD THEN GENRE(172000B,BFLC+5,1,TRUE)
            ELSE GENRE(72000B,DPLMT,1,TRUE) ↑STA↓ ;
      END ;
                END
    END ELSE ERROR(32)    ;
 1:         END ↑ ASSIGN ↓ ;
↑*****************************************************************↓
```

```
PROCEDURE IFSTAT ;
  VAR LCA1,LCA2: SHRINT ;
  BEGIN
    INSYMBOL; EXPRESSION ;
    IF GATTR.TYPTR NE 0 THEN
    BEGIN
      IF GATTR.TYPTR NE BOOLPTR THEN ERROR(38) ;
      GENRE(002003B,0,0,FALSE); ↗SZA,RSS↓
      GENRE(026000B,0,1,TRUE) ; ↗JMP↓
      LCA1:=HCA ;
    END ;
    IF NO NE 28 THEN ↗ SY≠THEN↓
    BEGIN
      IF GATTR.TYPTR NE 0 THEN ERROR(37); SKIP(28);
      IF NO NE 28 THEN
      BEGIN
        IF GATTR.TYPTR NE 0 THEN ERROR(37);
        IF ERRCL[NO] EQ 2 THEN GOTO 30; GOTO 20;
      END
    END ;
    INSYMBOL ;
20:    STATEMENT ;
30:    IF NO NE 29 THEN ↗ SY≠ELSE ↓
    BEGIN
      GENRE(0,0,0,FALSE) ;     ↗NOP↓
      HPAS[LCA1].ADRR := IC ;
    END ELSE
    BEGIN
      GENRE(026000B,0,1,TRUE); ↗JMP↓
      LCA2:=HCA ;
      GENRE(0,0,0,FALSE) ; ↗NOP↓
      HPAS[LCA1].ADRR :=IC ;
      INSYMBOL; STATEMENT ;
      GENRE(0,0,0,FALSE) ; ↗NOP↓
      HPAS[LCA2].ADRR := IC ;
    END ;
  END ↗ IFSTAT↓ ;
↗*****************************************************************↓
```

```
        PROCEDURE REPEATSTAT ;
          VAR LJPADDR : ADDRESS ;
          BEGIN
            GENRE(0,0,0,FALSE) ; ↗ NOP ↓
            LJPADDR :=IC ;
            REPEAT
              BEGIN
                INSYMBOL ;
   20:        STATEMENT ;
              IF ERRCL[NO] EQ 1 THEN BEGIN ERROR(39); GOTO 20 END ;
              IF NO EQ 29 THEN ↗ ELSE ↓
              BEGIN  ERROR(36); INSYMBOL; GOTO 20 END ;
              END
            UNTIL NO NE 16 ; ↗ SY≠ ;↓
            IF NO NE 33 THEN ↗ SY≠UNTIL↓ ERROR(44) ELSE
            BEGIN
              INSYMBOL ; EXPRESSION ;
              IF GATTR.TYPTR NE 0 THEN
              BEGIN
                GENRE(002003B,0,0,FALSE) ; ↗ SZA,RSS ↓
                GENRE(026000B,LJPADDR,1,TRUE); ↗JMP TO LJPADDR↓
                GENRE(0,0,0,FALSE) ; ↗ NOP ↓
              END ELSE SKIP(53)
            END
          END ↗ REPEATSTAT↓ ;
↗*************************************************************↓
```

```
PROCEDURE WHILESTAT ;
  VAR LJPADDR : ADDRESS ; LCA : SHRINT ;
  BEGIN
    GENRE(0,0,0,FALSE) ; ↑ NOP ↓
    LJPADDR :=IC ;
    INSYMBOL; EXPRESSION ;
    IF GATTR.TYPTR NE 0 THEN
    BEGIN
      GENRE(002003B,0,0,FALSE) ; ↑SZA,RSS↓
      GENRE(026000B,0,1,TRUE) ; ↑ JMP↓
      LCA := HCA ;
    END ;
    IF NO NE 35 THEN ↑SY≠DO↓
    BEGIN
      IF GATTR.TYPTR EQ 0 THEN ERROR(40) ; SKIP(35);
      IF NO NE 35 THEN
      BEGIN
        IF GATTR.TYPTR EQ 0 THEN ERROR(40);
        IF ERRCL[NO] EQ 2 THEN GOTO 20; GOTO 10
      END ;
    END ;
    INSYMBOL ;
20:   STATEMENT ;
      GENRE(026000B,LJPADDR,1,TRUE) ; ↑ JMP TO LJPADOR↓
10:   GENRE(0,0,0,FALSE) ; ↑ NOP ↓
    HPAS[LCA].ADRR := IC ;
  END ↑ WHILESTAT↓ ;
↑*************************************************************↓
```

```
PROCEDURE GOTOSTAT ;
  BEGIN
    INSYMBOL ;
    IF (NO NE 2) OR (CL NE 1) THEN
    BEGIN ERROR(41); SKIP(53) END ELSE
    BEGIN
      IF IVAL GE MAX10 THEN ERROR(58) ;
      ↗ SEARCH THRU LABELTABLE OF CURRENT BLOCK↓
      FOR IT := 1 TO CLABIX DO
      WITH LABTAB[IT] DO
      BEGIN
        IF LABVAL EQ IVAL THEN ↗ LABEL ALREADY OCCURED ↓
        BEGIN ↗IF DECL.OCC.,GENERATE CODE ELSE CHAIN OCC.↓
          IF FLD2 EQ 0 THEN
          GENRE(0260000B,FLD3,1,TRUE)    ↗ JMP TO FLD3↓ ELSE
          BEGIN
            GENRE(0260000B,0,1,TRUE) ;    ↗JMP↓
            IF CHNIX EQ 0 THEN
            BEGIN ERROR(47); GOTO 20; END ;
            WITH UNDLAB[CHNIX] DO
            BEGIN
              IT1:=SUCC; SUCC:=FLD3; FLD3:=CHNIX;PLACE:=HCA
            END ;
            CHNIX := IT1;
          END ;
          GOTO 20 ;
        END ↗ LABVAL = IVAL↓
      END ↗ WITH LABTAB↓ ;
      ↗ LABEL NOT YET MET, ENTER IT INTO LABELTABLE ↓
      GENRE(0260000B,0,1,TRUE); ↗ JMP↓
      IF CLABIX EQ MAXLAB THEN BEGIN ERROR(46);GOTO 20 END ;
      IF CHNIX EQ 0 THEN BEGIN ERROR(47); GOTO 20 END ;

      CLABIX := CLABIX+ 1 ;
      WITH LABTAB[CLABIX],UNDLAB[CHNIX] DO
      BEGIN
        IT1:=SUCC; LABVAL:=IVAL; SUCC:=0; FLD2:=CHNIX;
        FLD3:= CHNIX ; PLACE := HCA ;
      END ;
      CHNIX := IT1;
  20: INSYMBOL ;
    END ↗ IF NO=2 AND CL=1 ↓ ;
  END ↗ GOTOSTAT↓ ;
↗**********************************************************↓
```

```
PROCEDURE FORSTAT;
   VAR LATTR:ATTR;LCCLASS,LCA: INTEGER;
       LJPADDR: ADDRESS; LOF: BOOLEAN;
   BEGIN
     INSYMBOL;
     IF NO NE 1 THEN
     BEGIN ERROR(11); GATTR.TYPTR:=0 END ELSE
     BEGIN
       SRCHREC(NEXT);
       IF CTPTR EQ 0 THEN BEGIN ERROR(23); CTPTR:= UNDECPTR END;
       VARIABLE;
     END ;
     LATTR:= GATTR;
       IF NO NE 22 THEN ↗ SY: NE := ↓
       BEGIN
         IF GATTR.TYPTR NE 0 THEN ERROR(33);
         SKIP(22) ;
         IF NO NE 22 THEN
         BEGIN
           IF GATTR.TYPTR EQ 0 THEN ERROR(33);
           IF ERRCL[NO] EQ 0 THEN GOTO 20; GOTO 10;
         END
       END ;
     INSYMBOL; EXPRESSION ;
     WITH LATTR DO GENRE(72000B,DPLMT,1,TRUE); ↗ STA IDENTIFIER↓
     IF NO NE 37 THEN ↗SY NE TO/DOWNTO↓
     BEGIN
       IF GATTR.TYPTR NE 0 THEN ERROR(70);     SKIP(37);
       IF NO NE 37 THEN
       BEGIN
         IF GATTR.TYPTR EQ 0 THEN ERROR(44);
         IF ERRCL[NO] EQ 0 THEN GOTO 20 ; GOTO 10;
       END
     END ;
     LCCLASS:=CL; INSYMBOL; EXPRESSION;
     IF (GATTR.TYPTR NE 0) AND (LATTR.TYPTR NE 0) THEN
     BEGIN
       GENRE(72000B,BFLC+3,1,TRUE); ↗STA IN BFLC+3↓
       LJPADDR:= IC+1 ;
       WITH LATTR DO
       BEGIN
         IF LCCLASS EQ 1 THEN
         BEGIN
           GENRE(62000B,DPLMT,1,TRUE); ↗LDA IDENTIFIER↓
           GENRE(3004B,0,0,FALSE); ↗ CMA,INA↓
           GENRE(42000B,BFLC+3,1,TRUE); ↗ADA TEMP↓
```

```
           END ELSE ↗ DOWNTO↓
           BEGIN
             GENRE(62000B,OFLC+3,1,TRUE); ↗LDA TEMP↓
             GENRE(3004B,0,0,FALSE); ↗ CMA,INA↓
             GENRE(42000B,DPLMT,1,TRUE); ↗ ADA IDENTIFIER↓
           END
         END ↗ LATTR↓   ;
         GENRE(2022B,0,0, FALSE); ↗ SSA,SZA↓
         GENRE(26000B,0,1,TRUE); ↗JMP OUT OF THE LOOP↓
         LCA := HCA ;
       END ↗ LATTR AND GATTR NE 0 ↓;
         IF NO NE 35 THEN ↗ SY NE DO ↓
         BEGIN
           IF GATTR.TYPTR NE 0 THEN ERROR(40); SKIP(31);
           IF NO NE 31 THEN
           BEGIN
             IF GATTR.TYPTR EQ 0 THEN ERROR(40);
             IF ERRCL[NO] EQ 0 THEN GOTO 20 ; GOTO 10;
           END
         END  ↗ DO↓ ;
         INSYMBOL;
20:      STATEMENT ;
         IF LATTR.TYPTR NE 0 THEN
         BEGIN
           WITH LATTR DO
           BEGIN
             GENRE(62000B,DPLMT,1,TRUE) ; ↗LDA IDENTIFIER↓
             IF LCCLASS EQ 1 THEN GENRE(2004B,0,0,FALSE) ELSE
             BEGIN GENRE(7400B,0,0,FALSE); GENRE(40001B,0,0,FALSE)
                     ↗ ADA TO B REGISTER↓ END;
             GENRE(72000B,DPLMT,1,TRUE); ↗STA IDENTIFIER↓
           END ;
           GENRE(26000B,LJPADDR,1,TRUE); ↗JMP BACK TO LOOP↓
           GENRE(0,0,0,FALSE) ; ↗ NOP ↓
           HPAS[LCA].ADRR := IC ;
         END ↗ LATTR.TYPTR NE 0 ↓ ;
10:    END ↗ FORSTAT ↓   ;
↗*******************************************************************↓
```

```
      PROCEDURE SRCHX (VAR P: INTEGER) ;
        ⌐ SEARCHES TABLES OF EXTERNALS ↓
        BEGIN
          CTPTR := P ;
          WHILE CTPTR NE 0 DO
          BEGIN
            COMP(EXTX[CTPTR].XNAM,AVAL,COMPAR);
            IF COMPAR THEN GOTO 1 ELSE CTPTR := CTPTR-1 ;
          END ;
    1:  END ⌐SRCHX↓ ;
⌐*********      *********        *********      ********     *********↓
      PROCEDURE EXTERN ;
        BEGIN
          SRCHX(XPTR) ;
          IF COMPAR THEN
          BEGIN
            IF CTPTR EQ 0 THEN BEGIN ERROR(61) ; GOTO 1  END;
          END ELSE
          BEGIN
            XPTR:=XPTR+1;
            WITH EXTX[XPTR] DO
            BEGIN
              I:=0;
              REPEAT
                I:=I+1; XNAM[I]:=AVAL[I] ;
              UNTIL I EQ 10 ;
            EXSYM := EXSYM+1 ;
            XSYM := EXSYM ;
            END
          END ⌐ COMPAR ↓ ;
    1:  END ⌐ EXTERN↓ ;
⌐*************************************************************************↓
```

```
PROCEDURE WRITEIR ;
  VAR  STWR : INTEGER ;
  BEGIN
    WRFL := TRUE ;
    INSYMBOL ;
    IF NO NE 9 THEN BEGIN ERROR(50); SKIP(53) ; GOTO 1 END ;
    INSYMBOL ;
    IF (NO NE 2) OR (CL NE 1) THEN BEGIN ERROR(59) ;SKIP(53);
      GOTO 1 END ELSE BEGIN STWR:=IVAL;INSYMBOL END ;
    IF NO NE 15 THEN ↑,↓ BEGIN  ERROR(51);SKIP(53); GOTO 1
      END ELSE INSYMBOL;
    IF (NO NE 6) OR(CL NE 1) THEN BEGIN ERROR(60) ;  SKIP(53);
      GOTO 1;END ELSE BEGIN INSYMBOL; IF NO NE 15 THEN
           ERROR(113) END ;
    GENRE(002600B,0,0,FALSE) ; ↑ CLA ↓
    GENRE(072000B,BFLC+2,1,TRUE) ; ↑ STA ↓
    REPEAT
      CTY := TRUE ;
      INSYMBOL; EXPRESSION ;
      IF GATTR.TYPTR EQ 0 THEN ERROR(75) ELSE
      BEGIN
        GENRE(066000B,BFLC,1,TRUE) ; ↑ LDB ↓
        GENRE(046000B,BFLC+2,1,TRUE) ; ↑ ADB ↓
        IF GATTR.TYPTR EQ CHARPTR THEN
        GENRE(002200B,0,0,FALSE)   ↑ CME ↓
        ELSE GENRE(002100B,0,0,FALSE) ; ↑ CLE ↓
        GENRE(016003B,0,4,FALSE) ; ↑ JSB ↓
        GENRE(042000B,BFLC+2,1,TRUE) ; ↑ ADA↓
        GENRE(072000B,BFLC+2,1,TRUE) ; ↑ STA↓
      END ↑ TYPTR NE 0 ↓ ;
    UNTIL NO NE 15 ;
    IF NO NE 10 THEN ↑)↓ ERROR(78) ELSE INSYMBOL;
    GENRE(042000B,BFLC+2,1,TRUE) ; ↑ADA↓
    GENRE(066000B,BFLC,1,TRUE); ↑ LDB↓
    GENRE(016002B,0,4,FALSE) ; ↑ JSB ↓
   GENRE(2001B,0,0,FALSE) ; ↑ RSS ↓
   GENRE(STWR,0,0,FALSE) ; ↑ LOGICAL UNIT NO. ↓
1: END ↑ WRITEIR ↓ ;
*************************************************************
```

```
PROCEDURE VARIAB ;
   BEGIN
     IF NO NE 1 THEN
     BEGIN  ERROR(11); GATTR.TYPTR:=0 END ELSE
     BEGIN
        SRCHREC(NEXT);
        IF CTPTR EQ 0 THEN
        BEGIN ERROR(31); CTPTR:= UNDECPTR END ELSE VARIABLE;
     END ;
   END ↑ VARIAB↓ ;
↑*********   *********.         *********      ********     ********↓
 PROCEDURE READIR ;
   VAR STHR : INTEGER ;
   BEGIN
     WRFL:=TRUE;   INSYMBOL;
     IF NO NE 9 THEN BEGIN ERROR(50); SKIP(53); GOTO 1 END;
     INSYMBOL;
     IF (NO NE 2) OR (CL NE 1) THEN BEGIN ERROR(59) ; SKIP(53);
         GOTO 1 END ELSE BEGIN STHR:=IVAL; INSYMBOL END;
     IF NO NE 15 THEN ↑,↓ BEGIN ERROR(51); SKIP(53); GOTO 1
         END ELSE INSYMBOL;
     IF (NO NE 6) OR (CL NE 1) THEN BEGIN ERROR(60) ; SKIP(53);
         GOTO 1 END ELSE BEGIN INSYMBOL; IF NO NE 15 THEN
         ERROR(51) END ;
     GENRE(66000B,BFLC,1,TRUE); ↑ LDB BUFFER ADDRESS↓
     GENRE(16001B,0,4,FALSE); ↑ JSB GET↓
     GENRE(2001B,0,0,FALSE); ↑ RSS↓
     GENRE(STHR,0,0,FALSE); ↑ LOGICAL UNIT NUMBER↓
     GENRE(66000B,BFLC,1,TRUE);↑LDB BUFFER POINTER↓
     GENRE(76000B,BFLC+1,1,TRUE) ; ↑STB BUFFER POINTER ↓
     REPEAT
        CTY := TRUE;
        INSYMBOL; VARIAB;
        IF GATTR.TYPTR EQ 0 THEN ERROR(47) ELSE
        BEGIN
          IF GATTR.TYPTR EQ CHARPTR THEN GENRE(2200B,0,0,FALSE)
             ↑CME↓ ELSE GENRE(2100B,0,0,FALSE); ↑ CLE↓
        GENRE(66000B,BFLC+1,1,TRUE) ; ↑ LDB BUFFR POINTR ↓
        GENRE(15004B,0,4,FALSE); ↑ JSB ALLOK↓
        WITH GATTR DO
        BEGIN
          IF ACCESS EQ DRCT THEN
          GENRE(72000B,DPLMT,1,TRUE) ↑STA IN VARIABLE↓ ELSE
          GENRE(172000B,BFLC+4,1,TRUE); ↑ STA ARRAY ELEMENT ↓
        END ;
        GENRE(76000B,BFLC+1,1,TRUE) ; ↑STB THE BUFFER POINTER ↓
```

```
                END ↑ TYPTR NE 0↓
              UNTIU NO NE 15 ↑,↓ !
              IF NO NE 10 THEN ↑!↓ ERROR(7)  ELSE INSYMBOL;
      !!    END ↑ READIR↓ !
*****!*******!*****************!**************************************************
```

```
         PROCEDURE COMPSTAT ;
           BEGIN
             REPEAT
               BEGIN
                 INSYMBOL ;
    1:           STATEMENT ;
                 IF NO EQ 29 THEN ↑ELSE↓
                 BEGIN ERROR(36) ; INSYMBOL; GOTO 1 END ;
               END;
             UNTIU NO NE 16 ;
             IF NO NE 26↑END↓ THEN ERROR(15) ELSE INSYMBOL ;
           END ↑ COMPSTAT↓ ;
************************************************************************
```

```
PROCEDURE STATEMENT ;
  VAR LPSW : SHRINT ;
  BEGIN
    IF (NO EQ 2) AND (CL EQ 1) THEN ↗ LABEL ↓
    BEGIN
      GENRE(0,0,0,FALSE) ; ↗NOP↓
      IF IVAL GE MAX10 THEN ERROR(58)  ;
      FOR IT :=1 TO CLABIX DO
      WITH LABTAB[IT] DO
      IF LABVAL EQ IVAL THEN ↗ FOUND↓
      BEGIN
        IF FLD2 EQ 0 THEN ↗MULTIDEF↓ ERROR(48) ELSE ↗FIXUP↓
        BEGIN
        IT1:= FLD3;
        REPEAT
          WITH UNDLAB[IT1] DO
          BEGIN
            WITH HPAS[PLACE] DO ADRR := IC ;  IT1:=SUCC
          END
        UNTIL IT1 EQ 0 ;
        IT1:=FLD2; UNDLAB[IT1].SUCC :=CHNIX ;
        CHNIX := FLD3; FLD2:=0; FLD3:=IC ;
        END ;
        GOTO 1 ;
      END ↗ IF,FOR,WITH ↓ ;
    ↗ NEW LABEL ↓
    IF CLABIX EQ MAXLAB THEN ERROR(46) ELSE
    BEGIN
      CLABIX := CLABIX +1 ;
      WITH LABTAB[CLABIX] DO
      BEGIN LABVAL :=IVAL; FLD2:=0; FLD3:=IC END ;
    END ;
1:    INSYMBOL ;
      IF NO NE 21 THEN ↗:↓
      BEGIN ERROR(10) ; SKIP(53); END ELSE INSYMBOL ;
    END ↗ IF NO=2 AND CL=1 ↓ ;
    RM:=0; SIY:= TRUE; CIY:= TRUE ;
    CASE SPLITSTAT[NO] OF
    ↗PASS↓ 1:  ↗0 OR 2↓ ;
          2:   BEGIN
               SRCHREC(NEXT) ;
                 IF CTPTR EQ 0 THEN BEGIN ERROR(23) END ;
                 WITH CONTEXTTABLE[CTPTR] DO
                 IF KLASS LE KONST THEN ERROR(64) ELSE
                 IF (KLASS=PROC) AND ((PROCTYPE=CTPTR) OR
                    (PROCTYPE EQ 0)) THEN
```

```
                BEGIN
                IF CTPTR LE PREDEFPTR THEN
                BEGIN
                  LPSW := CONTEXTTABLE[CTPTR].SEGSIZE ;
                  CASE LPSW OF
  0,1,2,3,4,5,6,7,8,9 : ;
   10:          READIR ;
   11:          WRITEIR ;
                  END ;
                END ELSE WRITE(≡PROCEDURE≡)      ;
                  END ρ PROCCALL↓ ELSE ASSIGN ;
                 END ρ IDENT↓ ;
          ρBEGIN↓     3:  COMPSTAT;
          ρIF↓        4:  IFSTAT ;
          ρCASE↓      5:  ;
          ρREPEAT↓    6:  REPEATSTAT ;
          ρWHILE↓     7:  WHILESTAT;
          ρFOR↓       8:  FORSTAT ;
          ρGOTO↓      9:  GOTOSTAT ;
          ρWITH↓      10:  ;
            END ρ CASE OF ↓ ;
            RP:=0; STY:=TRUE; CTY:= TRUE;
          END ρ STATEMENT ↓ ;
ρ************************************************************↓
ρ************************************************************↓
```

```
      ↙ FOLLOWING ROUTINES ARE USED TO PROVIDE CODES IN RELOCATABLE
        FORMAT ↓
↙***********************************************************************↓
      PROCEDURE PUNCHRECORD(RECNAME:ALFA; RL:INTEGER) ;
        VAR I,N,J,K,C,REM: INTEGER ;
        ↙GLOBAL W ↓
        BEGIN
          IF LISTBIN THEN ↙LIST BINARY RECORD ↓
          BEGIN
            N:=0; WRITE(RECNAME);LINES(2);
            FOR I:=1 TO RL DO
            BEGIN
              N:=N+1; OUTO(W[I]); WRITE(= =) ;
              IF N EQ 8 THEN BEGIN N:=0; LINES(1) END ;
            END; LINES(4);
          END ;
          IF PUNCHBIN THEN ↙ PUNCH BINARY RECORD, RELOCATABLE FORMAT↓
          BEGIN
            FOR I:=RL+1 TO 60 DO W[I]:=0;
            C:=0;
            REPEAT
              REM:=0; K:=0;
              FOR I:=1 TO 4 DO
              BEGIN
                N:=REM;
                FOR J:=C+1 TO C+3 DO APPEND(N,16,W[J]) ;
                K:=K+4 ;
                IF K LT 16 THEN
                BEGIN
                  C:=C+4;
                  J:=W[C] DIV DD[I] ;
                  REM:= W[C]-J*DD[I] ;
                  APPEND(N,16-K,J)
                END ELSE C:=C+3 ;
                HPDIP↑ :=N; PUT(HPDIP)
              END
            UNTIL C EQ 60 ;
          END
        END ↙ PUNCHRECORD ↓ ;
↙**********     *********      **********     ********     ********↓
```

```
      FUNCTION CHECKSUM(RL : INTEGER) : INTEGER ;
        VAR I,CS :INTEGER ; ⍴GLOBAL--W ↓
        BEGIN
          CS := W[2] ;
          FOR I:=4 TO RL DO CS := CS+W[I] ;
          CHECKSUM := CS MOD  200000B
        END ⍴ CHECKSUM ↓ ;
⍴**********       *********     **********      *********      *********↓
      PROCEDURE PACKIN(VAR NIM: AR; VAR B : ARRINT3) ;
        VAR I,J,N : INTEGER ;
        BEGIN
          NIM[6] := ≡ ≡;
          FOR I := 1 TO 3 DO
          BEGIN
            J := 2*I - 1 ;
          B[I] := ASCIICODE[NIM[J]] ;
          IF ((J+1) EQ 6) THEN N:=0 ELSE N:=ASCIICODE[NIM[J+1]] ;
          APPEND(B[I],8,N) ;
          END
        END ⍴ PACKIN↓ ;
⍴**********       *********     **********      *********      *********↓
      PROCEDURE NAMRECORD ;
        VAR P: ARRINT3; ⍴ GLOBAL -  W,PRNAM,LC↓
          BEGIN
            W[1] := 010400B; ⍴ RECORD LENGTH,LEFT SHIFTED 8 BITS↓
            W[2] := 200000B; ⍴ IDENT↓
            PACKIN(PRNAM,P) ; ⍴W[4..6] CONTAINS PROG. NAME↓
            APPEND(P[3],6,40B);
            FOR I:=1 TO 3 DO W[I+3]:=P[I] ;
            W[7] := LC;
            FOR I:=8 TO 17 DO W[I] :=0 ;
            W[3] := CHECKSUM(17);
            PUNCHRECORD(≡NAM RECORD≡,17) ;
          END ⍴ NAM RECORD ↓ ;
⍴**********       *********     **********      *********      *********↓
      PROCEDURE EXTRECORD ;
        VAR I,J,K,RL,M1,M2 : INTEGER ; MORE: BOOLEAN; P:ARRINT3;
        ⍴GLOBAL W,EXTX,EXSYM↓
        BEGIN
          M1:=1;
          IF EXSYM LE 19 THEN BEGIN MORE:=FALSE;M2:=EXSYM END ELSE
                              BEGIN MORE:=TRUE;M2:=19 END ;
   10     ⍴ NEW EXT RECORD ↓
          RL:= 3*M2+3 ⍴ RECORD LENGTH↓ ;
          W[1] :=0; INSERT(RL,8,W[1]) ;
          W[2] := M2+100000B; ⍴ ENTRIES + IDENT↓
```

```
           K:=3;
           FOR I:=M1 TO M2 DO
           BEGIN
             PACKIN(EXTX[I].XNAM,P) ; APPEND(P[3],0,I);
             FOR J:=1 TO 3 DO W[K+J] :=P[J] ;   K:=K+3;
           END ;
           W[3] := CHECKSUM(RL); ↑ CHECKSUM ↓
           PUNCHRECORD(≡EXT RECORD≡,RL);
           IF MORE THEN
           BEGIN
             M1:=M2+1 ; M2:=M2+20;
             IF M2 GE EXSYM THEN BEGIN M2:=EXSYM;MORE:=FALSE END ;
             GOTO 1 ;
           END ;
         END ↑ EXT RECORD ↓ ;     *********      *********     ********↓
↑**********    *********
        PROCEDURE OBLRECORD(PG:BITS) ;
         VAR M,I,J,K,L,N,NI:INTEGER; ENDC:BOOLEAN;
         BEGIN
           I:=1;
           REPEAT↑NEW RECORD EVERY LOOP↓
             L:=4; NI:=0; ENDC:=FALSE ;
             W[4] := HPAS[I].LOC ;↑RELOCATABLE LOAD ADDRESS↓
             WHILE (L LT 58) AND (NOT ENDC) DO
             BEGIN
               M:=0; L:=L+1; J:=L; W[J] :=0;
               ↑ W[J] CONTAINS RELOCATION INDICES FOR UPTO
                 NEXT FIVE WORDS ↓
               REPEAT
                 M:=M+1 ; L:=L+1 ;
                 IF HPAS[I].MREF THEN ↑MEMORY REFERENCE↓
                 BEGIN
                   APPEND(W[J],3,5) ;
                   W[L] := HPAS[I].INST ;
                   CASE HPAS[I].TYP OF
                      1: K:=0; 2: K:=1; 3: K:=2;
                   END ;
                   APPEND(W[L],0,K) ;
                   L:=L+1; W[L]:= HPAS[I].ADRR ;
                 END ELSE ↑NO MEMORY REFERENCE↓
                 BEGIN
                   IF HPAS[I].INST EQ 0 THEN W[L] := HPAS[I].ADRR ELSE
                   W[L] :=HPAS[I].INST;
                   APPEND(W[J],3,HPAS[I].TYP) ;
                 END ;
                 I:=I+1; IF I GT HCA THEN ENDC:=TRUE;
```

```
              UNTIL (M EQ 5) OR ENDC OR (L GE 59) ;
              APPEND(H[J],(5-M)*3+1,0) ;
              NI:=NI+M ;
            END ;
            H[1]:=0; INSERT(L,8,H[1]) ;
            H[2] := 60000B+PG*100B+NI ;
            H[3] := CHECKSUM(L) ;
            PUNCHRECORD(≡DBL RECORD≡,L) ;
          UNTIL ENDC ;
        END ↗ DBLRECORD ↓ ;
↗**********    *********       **********        *********        *********↓
       PROCEDURE ENDRECORD ;
         VAR I:INTEGER; ↗ GLOBAL --H,TRNSFR↓
         BEGIN
           H[1]:=2000B; ↗RECORD LENGTH=4,SHIFTED TO 8 BITS↓
           H[2]:= 120000B+TRNSFR[1]+ TRNSFR[2] ;
           H[4] := TRNSFR[3]; ↗ FOR MAIN PROGRAM IT IS 0 ↓
           H[3] := (H[2]+H[4]) MOD 200000B ;
           PUNCHRECORD(≡END RECORD≡,4) ;
         END ↗ ENDRECORD↓ ;
↗*****************************************************************************↓
```

```
PROCEDURE ENTERBODY ;
  ↱ GENERATES PROCEDURE PROLOG CODE AND
    PROCEDURE ENTRY CODE ↓
  VAR I: INTEGER ;
  BEGIN
    LCX :=0 ; HCA :=0 ;
       CLABIX := 0 ;
    IF LEVEL NE 0 THEN
    BEGIN ↱ PROCEDURE ENTRY CODE ↓
      WRITE(≡ ≡,≡DOLATER≡,EOL)
    END  ELSE
    BEGIN ↱ MAIN ↓
        BFLC:=2; BLC:=5;
      GENRE(0,0,0,FALSE) ; ↱ NOP↓
      GENRE(26000B,LC+BLC+BFLC+1,1,TRUE) ; ↱JMP↓
      GENRE(0,0,1,FALSE) ;
      FOR I:=1 TO BLC+LC DO GENRE(0,0,0,FALSE) ;
      IF VALP THEN
      BEGIN
        I:=1;
        FOR J:=VLC TO AT-1 DO
        BEGIN GENRE(0,VARB[I],0,FALSE); I:=I+1 END;
        FOR I:=AT+1 TO LC DO GENRE(0,0,0,FALSE) ;
      END ;
    END
  END ↱ ENTERBODY↓ ;
↱**********  **********    **********    *********    *********↓
```

```
PROCEDURE EXITBODY ;
  ↱ GENERATES CODE TO HALT THE EXECUTION OF THE PROGRAM↓
  BEGIN
       GENRE(160050,0,4,FALSE) ; ↱ JSB EXEC ↓
       GENRE(0,IC+1,1,FALSE)  ;   ↱ DEF *+2↓
       GENRE(0,IC+3,1,FALSE)  ;   ↱ DEF *+2↓
       GENRE(0,0,0,FALSE) ; ↱ NOP ↓
       GENRE(6,0,0,FALSE) ; ↱DEC 6↓
     END ; ↱ EXITBODY ↓
↱***********************************************************************↓
```

```
**********************************
*                                *
*                                *
*         MAIN    PROGRAM        *
*         ***************        *
*                                *
*                                *
**********************************
```

```
BEGIN
  ↱ MAIN PROCEDURE OF HPCOM CALLING DIFFERENT PARTS OF THE
    COMPILER AS NECESSARY ↓
       CH := INPUT↑ ;
11     SEG1; ↱ INIIILIAZIIION ↓
       IF (NO EQ 42) OR (NO FQ 43) THEN SEG2;
        ↱CONSTANT OR LABEL DECLARATION PART ↓
       ↱******     ******      ******      *****↓
       IF NO EQ 40 THEN ↱ TYPE ↓
          BEGIN INSYMBOL ;
            WHILE NO EQ 1 DO
            BEGIN   SRCHREC(NEXT) ;
                 IF (CIPIR NE 0 ) THEN ERROR(8);
                 I := 0 ;
                 REPEAT  I:= I+1 ;
                   TYPID[I] := AVAL[I] ;
                 UNTIL  I=10 ;
                 INSYMBOL ;
                 IF (NO NE 8) OR ( CL NE 6) THEN ERROR(4)
                                         ELSE INSYMBOL ;
                 ERR := FALSE ; TYPEDECL(IL,P) ;
                 IF ¬ERR THEN
                 BEGIN
                    COMP(CONTEXTTABLE[P].NAME,BLANCK,COMPAR) ;
                    IF NOT(COMPAR) THEN ERROR(55) ELSE
                       BEGIN WITH CONTEXTTABLE[P] DO
                          BEGIN   I:=0 ;
                                  REPEAT
                                     I:=I+1 ;
                                     NAME[I] := TYPID[I] ;
                                  UNTIL I=10 ;
                                  NXTEL := NEXT ;
                          END;
                          NEXT := P ;
                       END ;
                 END ;
                       FINDSEMICOLON ;
            END ;
          END ↱TYPE↓ ;
        ↱******         ******       ******       *****↓
        IF NO EQ 51 THEN SEG4 ; ↱SYNONYM DECLARATION↓
        IF NO EQ 45 THEN SEG3; ↱ VARIABLE DECLARATION ↓
         ↱ VALUE DECLARATION ↓
        IF NO EQ 48 THEN SEG5 ELSE VLC:=LC;
        ↱ IC IS INITIALIZED TO -1 ,SO THAT LOAD POINT IS ZERO ↓
        IC := - 1 ;
```

```
        ENTERBODY ;
        OP := FALSE ;
        IF NO EQ 25 THEN COMPSTAT ELSE ERROR(30) ;
        ↱ START COMPILING STATEMENT PART ↓
        EXITBODY ;
        WRITOUT ;
        WRITEM ;
        INSYMBOL;
        COMP(PROGEND,AVAL,COMPAR) ;
        IF NOT(COMPAR) THEN WRITE(EOL,≡ ≡,≡END CARD ERROR≡) ;
        IF ERR THEN BEGIN ERRMSSG; GOTO 2 END ;
        IF KLISTBIN THEN
        BEGIN
          WRITE(EOL,≡1≡) ; WRITE(≡ ≡,≡***≡,≡CODE STACK≡,≡***≡) ;
          LINES(4) ;
          FOR I:=1 TO HCA DO
          WITH HPAS[I] DO
          BEGIN
          LINES(1) ; OUTO(LOC); WRITE(≡ ≡);
             OUTO(INSI); WRITE(≡ ≡); OUTO(ADRR) ;
             WRITE(≡ ≡) ;
             OUTO(TYP); LINES(1);
          END
        END ;
        NAMRECORD;
        EXTRECORD;
        DBLRECORD(1) ;
        ENDRECORD;
        IF PUNCHBIN THEN ↱ PUNCH A BLANK CARD ↓
        BEGIN
          LISTBIN := FALSE;
          PUNCHRECORD(≡BLANK CARD≡,0);
        END ;
   2:   WHILE ((INPUT↑ EQ ≡/≡) OR (INPUT↑ EQ EOL))
                 AND NOT(EOF(INPUT)) DO GET(INPUT) ;
        IF NOT(EOF(INPUT)) THEN ↱ COMPILE NEXT PROGRAM↓
        BEGIN WRITE(EOL,≡1≡) ↱ START A NEW PROGRAM↓ ;
              CH := INPUT↑ ;
              WRITE(CH); GOTO 1
            END ;
    END.
```

APPENDIX H

SAMPLE COMPILED PROGRAMS IN PL/2100

```
PROGRAM FACT,L,8,P
000000                  CONST   MAX=4;
000000                  VAR
000000                     K,N,FACT: INTEGER ;
000003                  BEGIN
000012                     K:=1;FACT:=1;
000016          1:         IF  (K EQ MAX)  THEN GOTO 2 ;
000031                     K:=K+1;
000034                     FACT:=FACT*K;
000041                     WRITE(7,*,K,FACT);
000066                     GOTO 1;
000067          2:         N:=FACT ;
000072                       WRITE(7,*,N) ;
000110                  END ;
000227                     PROGEND
```

```
PROGRAM BIN,B,P,L
000000                CONST
000000                   FIND=5;
000000                VAR
000000                   X,I,J,K : INTEGER ;
000004                   A: ARRAY [1..15] OF INTEGER ;
000023                VALUE
000023                   A=(5*2,3,4,5,1,8,9,7,11,2*4)   ;
000023                BEGIN
000032                   I:=1 ; J:=15;
000036                   X:=FIND;
000040                   REPEAT
000041                      K:= (I+J) DIV 2 ;
000051                      IF A[K] LE X THEN I:=K+1 ;
000075                      IF A[K] GE X THEN J:=K-1 ;
000123                   UNTIL I GT J ;
000134                   WRITE(1,*,X,A[K]) ;
000171                END ;
000313              PROGEND
```

```
PROGRAM GCD,L,P,B
000000                /* TO FIND GCD(X,Y) */
000000                VAR
000000                  X,Y,A,B: INTEGER ;
000004                BEGIN
000016                  READ(401B,*,X,Y);
000033                  A:=X; B:=Y;
000037                  REPEAT  /* A>0,B>0 */
000040                    WHILE A GT B DO A:=A-B;
000060                    WHILE B GT A DO B:=B-A;
000100                  UNTIL A EQ B ;
000111                  /* A=B=GCD(X,Y) */
000111                  WRITE(8,*,X,Y,A,B) ;
000154                END ;
000271              PROGEND
```

```
               PROGRAM DIV,B,L,P
000000                   /* PROGRAM TO DIVIDE NATURAL NUMBER X BY Y */
000000                   VAR
000000                     X,Y,R,Q,W : INTEGER ;
000005                   BEGIN
000014                     READ(401B,*,X,Y);
000034                     R:=X; Q:=0; W:=Y;
000042                     WHILE W LE R DO W:=2*W ;
000062                       /* W= 2**N*Y GT X */
000062                     WHILE W NE Y DO
000073                     BEGIN /* Q*W+R=X,R GE 0 */
000073                       Q:=2*Q; W:=W DIV 2;
000107                       IF W LE R THEN
000117                       BEGIN
000117                         R:=R-W; Q:=Q+1;
000127                       END;
000130                     END;
000132                     /* Q*Y+R=X,0≤R≤W,Q=X DIV W */
000132                     WRITE(8,*,X,Y,Q,R,W);
000204                   END ;
000324             PROGEND
```

```
PROGRAM LST,P,L,B
000000
000000          /* FINDING THE LEAST NUMBER THAT IS EQUAL TO TWO DIFFERENT     *
000000             SUMS 3F TWO NATURAL NUMBERS RAISED TO THE THIRD POWER */    *
000000          VAR
000000            I,IL,IH,MIN,A,B,K: INTEGER ;
000007            J,P,S: ARRAY [1..12] OF INTEGER ;
000053            /* P[K]=K**3,S[K]=P[K]+P[J[K]] FOR K:=1,2,..IH */
000053          BEGIN
000062            I:=1; IL:=1; IH:=2;
000070            J[1]:=1; P[1]:=1; S[1]:=2; J[2]:=1;
000144            P[2]:=8; S[2]:=9;
000172            REPEAT
000173              MIN:=S[I]; A:=I; B:=J[I] ;
000221              IF J[I] EQ I THEN IL:=IL+1 ELSE
000246              BEGIN IH:=IH+1; P[IH]:=IH*IH*IH;
000272                    J[IH]:=1; S(IH]:= P(IH]+1 ;
000331              END ;
000332              I:=IL; K:=I;
000336              WHILE K LT IH DO
000350              BEGIN
000350                K:=K+1;
000353                IF S[K] LT S[I] THEN I:=K;
000412              END ;
000414            UNTIL S(I] EQ MIN ;
000435            WRITE(8,*,A,B,I,J[I]) ;
000510          END ;
000635          /* TO FIND LEAST X SUCH THAT
000635                      X=A**3+B**3=C**3+D**3
000635             WHERE A,B,C,D ARE NATURAL NUMBERS
000635             SUCH THAT A≠C AND A≠D */
000635          PROGEND
```

```
PROGRAM  TST1,L,B,P
000000             CONST  MIN=4, MAX=9 ;
000000             TYPE
000000               R = RECORD
000000                     A,B : INTEGER
000000                   END ;
000000             VAR
000000               C : R ;
000002               D,E : INTEGER ;
000004              BEGIN
000013               C.A := 2 ;
000015               C.B := 5 ;
000017               D := C.A + MIN ;
000022               E := C.B + MAX ;
000025               WRITE( 1,*,D,E) ;
000052              END ;
000173            PROGEND
```

```
PROGRAM TST2,B,P,L
000000                 CONST  MIN=4,MAX=10 ;
000000                 SYN
000000                   A = B,C ;
000000                   D = E,F ;
000000                 VAR
000000                     A,D,L : INTEGER ;
000003                   BEGIN
000012                     A := MIN ; WRITE(1,*,MIN ) ;
000032                     B:= MAX; E:= MIN ;
000036                     L := B+E ;
000041                     WRITE(1,*,L,B,E) ;
000075                   END ;
000214               PROGEND
```

```
PROGRAM TSI6,B,P,L
000000             CONST SH=5,SHT=10;
000000             VAR A,B,C,D : INTEGER ;
000004             BEGIN
000013               A:=4; B:=2; D:=4;
000021               C:= A ALS 2 + B ALF 7 + A RAL 11 ;
000033               D := C BLF 10 - A BLS 9 - C RBL 10 + D RBR 0 ;
000060             END ;
000177           PROGEND
```

```
PROGRAM TST4,B,P,L
000000               VAR A,B,C,D : INTEGER ;
000004               BEGIN
000013                  A:=2; B:=3; C:=4;
000021                  REPEAT
000022                     A := A+1 ;
000025                     D := B+C;
000030                  UNTIL A EQ 10 ;
000041                  WRITE(1,*,A,D) ;
000066               END ;
000210             PROGEND
```

```
PROGRAM TST5,B,P,L
000000                VAR A,B,C,D : INTEGER ;
000004                BEGIN
000013                   A:=1; B:=3; C:=5;
000021                   WHILE A NE 10 DO
000032                   BEGIN
000032                      D :=C*B; D:=D+5-B ;
000045                      A := A+1
000046                   END ;
000052                   WRITE(1,*,A,D) ;
000077                END ;
000220              PROGEND
```

```
PROGRAM TST6,P,L,B
000000                    VAR A,B,C,D : INTEGER ;
000004                       BEGIN
000013                          READ(1,*,A,B) ;
000033                          C := A+B*A +B ;
000042                          READ(4018,*,C,D) ;
000062                          A := B+C+D ;
000066                          WRITE(8,*,A,B,C,D) ;
000131                       END ;
000246                 PROGEND
```

```
PROGRAM TST6,L,P,B
000000                VAR
000000              A: ARRAY [1..10] OF INTEGER ;
000012               B,C,D : INTEGER ;
000015                BEGIN
000024                  A[1]:=10;  A[5]:= 5;
000053                    B:=2; C:=5; D:=3;
000060               D := B+A[5]- C;
000100                 A[2]:= B+ A[3] ;
000127               WRITE(1,*,D,A[2]) ;
000164              END ;
000310            PROGEND
```

```
PROGRAM TST9,8,P,L
000000          VAR
000000             A,B : INTEGER ;
000002             C : ARRAY [1..10] OF INTEGER ;
000014             D : CHAR ;
000015          VALUE
000015             C = (2,3,4,5*1,5,6) ;
000015             D = ≡ A≡ ;
000015          BEGIN
000024              A := C[3] + C[4] ;
000051            B := C[5] * 5;
000066            WRITE(8,*,A,B,D) ;
000122          END ;
000245         PROGEND
```

```
PROGRAM TST11,B,P,L
000000              VAR
000000                A : ARRAY (1..5,1..10) OF INTEGER ;
000062                C : ARRAY [1..5] OF CHAR ;
000067                I : INTEGER ;
000070               VALUE
000070                 A = (10*1,10*2,10*3,10*4,10*5) ;
000070                 C = (≡A≡,≡B≡,≡C≡,≡D≡,≡E≡) ;
000070                 BEGIN
000077                  I := 0;
000101                 REPEAT
000102                   I := I+1 ;
000105                   WRITE(1,*,C[I]) ;
000133                   WRITE(8,*,A[I,I]) ;
000170                 UNTIL I EQ 5 ;
000201               END ;
000326          PROGEND
```

```
PROGRAM TST12,B,P,L
000000                CONST A=10,B=11;
000000                TYPE
000000                  C= RECORD
000000                      D : ARRAY[1..10] OF INTEGER ;
000000                   END ;
000000                VAR
000000                  M,N : CHAR ;
000002                  I,J,K : INTEGER;
000005                    L: C ;
000017                    VALUE
000017                   M= ≡E≡;
000017                    N=≡N≡;
000017                     I= 200; J=5;
000017                BEGIN
000026                    WRITE(1,*,I,J) ;
000053                    WRITE(1,*,I,J,M,N) ;
000116                END ;
000233              PROGEND
```