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

CHAPTER I: INTRODUCTION

1.1 Comparison of High Level Language and Assembly Language 1
   1.1.1 Assembler Language 1
   1.1.2 High Level Language 2

1.2 Need for Amalgamation of High Level Language and Assembler Language 3

1.3 Minicomputers and Need for High Level Language for Minicomputers 3
   1.3.1 Minicomputers 3
   1.3.2 Need for High Level Language for Minicomputers 5

1.4 Existing High-Level Machine Oriented Languages 6

1.5 Features of HP/2100A Machine 6
   1.5.1 Word Length 6
   1.5.2 Registers 7
   1.5.3 Memory 9
   1.5.4 Instruction Format 11
   1.5.5 Data Format 13
   1.5.6 Conclusion 13

1.6 Programming Language 2100 (PL/2100) 14
   1.6.1 Choice of High Level Language 14
   1.6.2 Choice of Features of HP/2100A Assembler 14
   1.6.3 Form of PL/2100 15

1.7 Purpose of this Project 16
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1.5</td>
<td>HP Word</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 1.5.3.2.1</td>
<td>Single Length Memory Instruction Word</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 1.5.3.2.2</td>
<td>Extended Length Memory Instruction Word</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 1.5.4.1</td>
<td>Single Length Memory Instruction</td>
<td>11</td>
</tr>
<tr>
<td>Fig. 1.5.4.2</td>
<td>Extended Arithmetic Memory Instruction</td>
<td>11</td>
</tr>
<tr>
<td>Fig. 1.5.4.3.1</td>
<td>Register Reference Memory Instruction Format</td>
<td>12</td>
</tr>
<tr>
<td>Fig. 2.1</td>
<td>A Typical One-Pass Compiler</td>
<td>19</td>
</tr>
<tr>
<td>Fig. 2.4</td>
<td>Main Control Section of Lexical Analyser</td>
<td>25</td>
</tr>
<tr>
<td>Fig. 2.5</td>
<td>Presents the Block Diagram of Syntax Analyser</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 2.9</td>
<td>Displays Different Parts of HPCOM Compiler</td>
<td>30</td>
</tr>
<tr>
<td>Fig. 3.1</td>
<td>Code Emitter Organization</td>
<td>32</td>
</tr>
<tr>
<td>Fig. 3.4.1</td>
<td>Circular Linkage of Synonymous Variables</td>
<td>52</td>
</tr>
<tr>
<td>Fig. 3.4.2</td>
<td>The Changed Circular Linkage of Synonymous Variables</td>
<td>52</td>
</tr>
<tr>
<td>Fig. 3.6</td>
<td>Displays Code Stack of a PL/2100 Main Procedure</td>
<td>55</td>
</tr>
<tr>
<td>Fig. 4.1.3.1</td>
<td>Displays the Use of Buffer Pointer</td>
<td>61</td>
</tr>
<tr>
<td>Fig. 4.3.2.1.a</td>
<td>Segmented Programs</td>
<td>69</td>
</tr>
<tr>
<td>Fig. 4.3.2.1.b</td>
<td>Main-to-Segment Jump</td>
<td>71</td>
</tr>
<tr>
<td>Fig. 4.3.2.2</td>
<td>Block Diagram Showing the Segmented Parts of the Compiler</td>
<td>72</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 4.1.1.2 Displays CONWD for Different I/O Devices 58
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 relatively 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 idiosyncrasies of the machine.

iii) A good compiler 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

```
15 14

. . . . . . .
```

fig. 1.5 HP WORD
HP/2100A has a 16-bit word length and is only word 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/O 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 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
fig. 1.5.3.2.1 Single Length Memory Instruction Word

  c) bits 11-14 for instructions
  d) bits 0-9 for addresses.

ii) for extended arithmetic instructions [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.

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 2/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

Class Instruction

15 ... 12 11 10 9 .. .. 4 3 .. 0
D/I MEMORY ADDRESS
15 14 0

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 0 0 0 0 0 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 $\emptyset + A$
Logic $1 + B$

S/A + denotes Skip-rotate or Alter-skip groups
Logic $\emptyset + S$
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/\$ devices. Packing of two bytes into one word is accomplished by the software drivers. In I/\$ 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 fami-
liar 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 i.e. 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 FORTRAN.
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 structures 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/O 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.

![Diagram](image)

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:

```plaintext
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, NEG);

VAR

CONTEXTTABLE: ARRAY[0..250] OF

PACKED

RECORD

NAME: AR; NXTEL: SHRINT; SYNCALL: BOOLEAN;

CASE KLASS: IDKCLASS OF

TYPES : (SIZE: ADDRESS;
          CASE FORM: TYPFORM OF
          NUMERIC: (BITS: BITRANGE; MIN, MAX: INTEGER);
          SYMBOLIC: (FCONST: INTEGER; BITSIZE: BITRANGE);
          ARRAYS: (AECTYPE, INXTYPE: SHRINT;
                    LO, HI: SHRINT; SZE: BOOLEAN;
                    OPTYPE: OPTPWR;
                    EXPL, EXPS: 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;
                           PROCIND: IDKINDS;
                           PROCADDR: ADDRESS; PROCLEVEL: RG3;
                           SEGSIZE: INTEGER);
                   VARS: (VTYP: 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;

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 definitions 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 HP/COM 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-delimiters (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

READ NEXT CHARACTER IN CH

IS CHARACTER A BLANK?

YES

READ AND STORE CHARACTERS UNTIL A NON-ALPHABETIC CHARACTER

NO

READ AND STORE CHARACTERS UNTIL A NON-NUMERIC CHARACTER

CH MUST BE A SPECIAL SYMBOL OF THE LANGUAGE

IS CH EQUAL TO THE CHARACTER?

YES

READ AND STORE CHARACTERS UNTIL CHARACTERS ARE READ

NO

IS CH AN ALPHABETIC CHARACTER?

YES

IS CH AN NUMERIC CHARACTER?

YES

READ AND STORE CHARACTERS UNTIL A NON-NUMERIC CHARACTER
fig. 2.4 Main control section of lexical analyzer

1. Check to see if the identifier formed of these characters is a reserved word of the language. If so, associate proper token with no. cl., other- wise they form an identifier of characters (≤10).

2. It is a numeric constant & so no. cl. = 1. Check if the constant is decimal or binary. Convert & store in proper value inIVAL.

3. It is a character constant & so no. cl. = 2. Check if these characters form a double symbol or not. If check double symbol, or not, check to see if it is a comment specification. If so, read characters, get proper token value. If not, comment, get double symbol, or proper no. cl. single symbol.

4. Read next character. Check if these characters form double symbols or not. If check double symbol, or not, comment.
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 analyzer 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 here in the program to begin looking for a phrase for all the nonterminals; hence syntax analyser is goal-oriented or predictive. This method is known as recursive descent method. The syntax analyser uses a bottom-up parsing principle to obtain input (viz integer tokens) via the source oriented 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).

![Diagram of code emitter organization]

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 recog-
nized, 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

\[ \text{e.g.} \quad 1^{\text{st}} \text{ operand} + 2^{\text{nd}} \text{ operand} \]

LDA 1\text{st} \text{ operand} \quad */ \text{Load first operand in the A-register */} \\
ADA 2\text{nd} \text{ operand} \quad */ \text{Add 2\text{nd} operand to the contents of A-register */}

3.2.1.1.2 Subtraction

\[ \text{e.g.} \quad 1^{\text{st}} \text{ operand} - 2^{\text{nd}} \text{ operand} \]

LDA 1\text{st} \text{ operand} \quad */ \text{Load first operand in A-register */} \\
LDB 2\text{nd} \text{ operand} \quad */ \text{Load second operand in B-register */} \\
CMB,INB \quad */ \text{Take 2's complement of the 2\text{nd} operand */} \\
ADA \quad */ \text{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

\[ \text{e.g.} \quad 1^{\text{st}} \text{ operand} \times 2^{\text{nd}} \text{ operand} \]

LDA 1\text{st} \text{ operand} \quad */ \text{load 1\text{st} operand in A-register */} \\
MPY 2\text{nd} \text{ operand} \quad */ \text{Multiply contents of A register with the 2\text{nd} operand */} \\
TOR \quad */ \text{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 B-register are zero.

3.2.1.1.4 Division

\[ \text{e.g. } 1^{\text{st}} \text{ operand} \div 2^{\text{nd}} \text{ operand} \]

- 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 2's 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)

\[ \text{e.g. } 1^{\text{st}} \text{ operand} \{\text{OR} \} 2^{\text{nd}} \text{ operand} \]

- 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 (A)

\[ \text{e.g. } 1^{\text{st}} \text{ operand} \{\text{AND} \} 2^{\text{nd}} \text{ operand} \]

- 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

\[ \begin{align*}
& \text{e.g.} \quad 1^{\text{st}} \text{ operand} \quad 2^{\text{nd}} \text{ operand} \\
& \quad \{ LT \} \\
& \quad \{ LE \} \\
& \quad \{ GT \} \\
& \quad \{ GE \} \\
& \quad \{ NE \} \\
& \quad \{ EQ \} \\
\end{align*} \]

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 \( 1^{\text{st}} \) operand /* Load the \( 1^{\text{st}} \) operand in the A-register */

CMA,INA /* Take \( 2^n \) complement of the \( 1^{\text{st}} \) operand */

ADA \( 2^{\text{nd}} \) operand /* Add \( 2^{\text{nd}} \) 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(<)

\[ \text{e.g.} \quad 1^{\text{st}} \text{ operand} \quad 2^{\text{nd}} \text{ operand} \]

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(≤)
  e.g. 1st operand \( \leq \) 2nd operand
SZA,SZA /* skip the next instruction, if the result of the subtraction is not positive */

3.2.1.2.3.3 GE(≥)
  e.g. 1st operand \( \geq \) 2nd operand.
SZA,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 \( > \) 2nd operand
SZA,RSS /* skip the next instruction, if the result of the subtraction is not positive */

3.2.1.2.3.5 NE(≠)
  e.g. 1st operand \( ≠ \) 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 \( = \) 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 \( \text{ior} \) 2nd operand
LDA 1st operand /* load 1st operand in the A-register */
IOR 2\textsuperscript{nd} operand /* inclusive OR 2\textsuperscript{nd} operand to the contents of the A-register. The result is in the A-register */

3.2.1.3.2 XOR

\begin{align*}
\text{e.g.} & \quad 1\text{st} \text{ operand } \text{xor } 2\text{nd} \text{ operand} \\
\text{LDA} & \quad 1\text{st} \text{ operand } /* \text{Load } 1\text{st} \text{ operand in the A-register } */ \\
\text{XOR} & \quad 2\text{nd} \text{ operand } /* \text{Exclusive OR } 2\text{nd} \text{ operand to the contents of the A-register. The result is in the B-register } */
\end{align*}

3.2.1.4 Shift Operators
\begin{align*}
\text{e.g.} \\
\text{operand } \begin{bmatrix} \text{als} \\ \text{ars} \\ \text{ral} \\ \text{ralf} \end{bmatrix} <\text{factor} >
\end{align*}

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

Similarly for the B-register, we have
\begin{align*}
\text{operand } \begin{bmatrix} \text{bls} \\ \text{brs} \\ \text{rbs} \\ \text{rbl} \\ \text{rbr} \\ \text{blf} \end{bmatrix} <\text{factor} >
\end{align*}

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.

\begin{align*}
\text{LDA} & \quad \text{operand } /* \text{load operand in the A-register } */ \\
\text{ALF,ALF} & \quad /* \text{make alf shift twice } */
\end{align*}
(b) operand \texttt{alf} 7
Codes generated are
LDA operand
ALF,ALF
ALF

3.2.1.4.2

(a) operand \texttt{rar} 7
Codes generated are
LDA operand
RAR,RAR
RAR,RAR /* code has been optimized in these cases */
RAR,RAR
RAR
(b) operand \texttt{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 \texttt{als} 18
Codes generated are
LDA operand
ALS,ALS /* factor = 18 \text{ mod } 16 = 2 */
(b) operand \texttt{als} 5

Codes generated are
LDA operand
\texttt{ALS,ALS}
\texttt{ALS,ALS}
\texttt{ALS}

\texttt{3.2.1.4.4}

(a) operand \texttt{ars} 17

Codes generated are
LDA operand
\texttt{ARS}

\texttt{/* one ARS shift necessary */}

(b) operand \texttt{ars} 6

Codes generated are
LDA operand
\texttt{ARS,ARS}
\texttt{ARS,ARS}
\texttt{ARS,ARS}

\texttt{3.2.1.4.5}

(a) operand \texttt{ral} 36

Codes generated are
LDA operand
\texttt{ALF}

\texttt{/* resultant factor is 4 and 4 RAL shifts are equal to one ALF shift */}
(b) operand \texttt{ral} 3

Codes generated are

LDA operand

\texttt{RAL,RAL} /* 3 shifts necessary */

\texttt{RAL}

\subsection{3.2.1.4.6}

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

for example

operand \texttt{<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.

\subsection{3.2.2 Assignment statement}

e.g. \texttt{<variable> := <expression>}

The result of expression is always in A register.

Hence code generated is,

\texttt{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 */

/* 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

```
FOR <identifier> := <expression> { TO } <expression>:
  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 2's complement */
  ADA <TEMP> /* calculate e2-el */
  SSA,SZA /* if e2-el is -ve, for loop is finished
  JMP LAB2 /* jump out of for loop */
```

```
LAB2:
  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 */
  NOP /* dummy no operation to go out of for loop */
```
For 'DOWNTO' the relation el ≥ 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 el-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. Those 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 opera-
tions 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,

\[ \text{effective address} = \text{DPLMT} + (\text{value of index-LO}) \times \text{SIZE} \]

For a multidimensional array, the effective address can be calculated similarly. The HFCOM 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:

\[ \text{VAR} \]

\[ \text{A: ARRAY [1..100] OF INTEGER;} \]

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 defini-
tion 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.

![Diagram of identifiers A, B, C, D]

**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.

![Diagram of identifiers A, B, C, D, E, F, G]

**fig. 3.4.2** The changed circular 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

\[ C : \text{integer}; \]

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 (Kl). 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:

<table>
<thead>
<tr>
<th>device</th>
<th>function</th>
<th>logical unit numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teleprinter</td>
<td>Input/output</td>
<td>1</td>
</tr>
<tr>
<td>Teletype</td>
<td>Input/output</td>
<td>7</td>
</tr>
<tr>
<td>Line Printer</td>
<td>Output</td>
<td>8</td>
</tr>
<tr>
<td>Card Reader</td>
<td>Input</td>
<td>5</td>
</tr>
</tbody>
</table>

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/O 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  l (or 2)  (l = READ, 2 = WRITE)
CONWD  OCT  conwd  (described later)
BUFFER  BSS  n  (Buffer of n words)
BUFL   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:

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL UNIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIELD  FUNCTION

W  If 1, tells DOS-M to return to the calling program after starting the I/O transfer. If W = Φ, 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 = Φ, "no printing" is specified.

V  Used when reading variable length records from punched tape devices in binary format (M=1, below). If V = Φ the record length is determined by the word count in the first non-zero character which is read in.
Determines the mode of data transfer. If \( M = \emptyset \) 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/O devices,

\( W = \emptyset \), 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.

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

\( V = \emptyset \) 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.

\( M = \emptyset \) i.e. All transfers are made in ASCII character mode.

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

<table>
<thead>
<tr>
<th>Device</th>
<th>conwd</th>
<th>conwd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
</tr>
<tr>
<td>Teleprinter</td>
<td>401</td>
<td>1</td>
</tr>
<tr>
<td>(oscilloscope)</td>
<td>(octal)</td>
<td></td>
</tr>
<tr>
<td>Teletype</td>
<td>407</td>
<td>7</td>
</tr>
<tr>
<td>(octal)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Card Reader</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>Line Printer</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1.1.2 displays 'conwd' for different I/O devices
4.1.2 I/O Routines

Two I/O 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 \( \leq 72 \)) 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.

```
Pointer passed to the ALLOC routine to allocate the value of A

This value of pointer is passed to ALLOC to allocate value of B

TOP OF BUFFER
value of A in ASCII character set

---

BUFFERM

---

length of buffer

This value of the pointer is passed back to main procedure as well as the length of buffer at that point.

This value of the pointer is returned and is passed to the PUT routine. The total length of the buffer used is also stored and then passed to PUT routine.
```

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/O 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

\[ \text{\underline{BB}...\underline{B} 4 \underline{BB}...\underline{B}, \underline{BB}...\underline{B} 45 \underline{B}...\underline{B}...\underline{B} \text{ line feed} } \]

The blanks (\(B\)) are all optional

ii) Output

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

\[ \text{\underline{BB}4\underline{B}45\underline{BB}} \]

These blanks (\(B\)) 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 */

LDB buffer pointer /* load the new buffer pointer */

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 expression may be calculated here; leaving the result in the A-register

{ LDA A /* Load into the A-register, the value of the variable A */

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 T /* 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 (ASCII character input).

iii) String output (ASCII 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.
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 information are passed to the segmented or the main program)
    PRAM2  ---
    :      ---
    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.

![Diagram of program flow]

**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.

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 if-then-else, repeat, goto 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 EXTERNAL 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 externals 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

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

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

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

M1. McKeeman, W.M., Horning, J.J. and Wortman, D.B.
A Compiler Generator,
Prentice-Hall, Inc., Englewood

S1. Sammet, J.E. A Brief Survey of Languages Used
In Systems Implementation,
Proceedings of ACM SIGPLAN Symposium
on Languages for System Implementa-
tion, 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

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 (\geq), equal to (=), less than or equal to (\leq), less than (<), and not equal to (\neq). 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
   \[ \text{<variable>} :: = \text{<expression>} \]

b) the if-then-else statement
   \[ \text{if}< \text{expression}> \]
   \[ \text{then}< \text{statement}>_1 \]
   \[ \text{if}< \text{expression}> \text{then}< \text{statement}>_1 \]
   \[ \text{else}< \text{statement}>_2 \]

which executes \( <\text{statement}>_1 \) if \(<\text{expression}> \) is true, or \( <\text{statement}>_2 \) if \(<\text{expression}> \) 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> {down to} to <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 definitions, 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

\[
\text{function } \text{<funcname> } \{(\text{<parameter list>})\}: \text{<result type>}
\]
\[
\text{\{<local declaration list>\} <statement part>}
\]

where \text{<funcname>} is the name of the function (an identifier),
and \text{<parameter list> and <local declaration list> are as de-
} fined above. The identifier representing the function name,
returns the value of the function. \text{<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 PI/2100

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

A.2.1 Notation and Terminology

According to traditional BNF (Backus-Naur 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. Θ (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

\[ \text{<program>} ::= \text{program} \text{<identifier>} \text{<option list>}
\]

\[ \text{<option list>} ::= ,\text{R},,\text{P},,\text{B} \text{<option list>}
\]

\[ \text{<block>} ::= \text{<declaration list> <function or procedure declaration> <statement part>}
\]

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 \text{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*/
    BEGIN

    CONST MAX = 4;
    VAR K, N, FACT: INTEGER;

    /* this is the
    beginning of a
    <compound tail>*1
     
1:     K := 1; FACT := 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>{;<synonym 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> ΄= ΄<identifier>

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>:]θ(<field list>)|}<case label>:}θ
<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>)θ}
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 initiali-
zation 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
declaraiton 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>
\[ \text{for statement} ::= \text{for}<\text{control variable}>: = \text{for list}\ \text{do}\ <\text{statement}> \]

\[ \text{for list} ::= \text{initial value} \ \text{to} \ \text{final value} | \text{initial value} \ \text{downto} \ \text{final value} \]

\[ \text{control variable} ::= \text{identifier} \]

\[ \text{initial value} ::= \text{expression} \]

\[ \text{final value} ::= \text{expression} \]

A.2.5.2 Semantics

A statement part is a sequence of statements.

A.2.5.3 Examples

A.2.5.3.1 \hfill y : = x

A.2.5.3.2 \hfill if x then

\[ y : = x \]

\[ \text{else} \]

\[ y : = z \]

A.2.5.3.3 \hfill case operator of

plus: x : = x+y;

\[ \text{times} : x : = x*y; \]

absval: if x < 0 then x : = x

end

A.2.5.3.4 \hfill while i > 0 do

begin

\[ z : = z^2x; \]

\[ i : = i \ \text{div} \ 2 \]

end
A.2.5.3.5 \texttt{repeat}

\begin{align*}
  k & : = i \mod j; \\
  i & : = j; \\
  j & : = k \\
  \text{until} & \ j = 0; 
\end{align*}

A.2.5.3.6 \texttt{for} \ i = 2 \ \texttt{to} \ 100 \ \texttt{do} \ \texttt{if} \ a[i] > \texttt{max} \ \texttt{then} \\
  \quad \texttt{max} & : = a[i]

A.2.6 Expression

A.2.6.1 Syntax

\begin{align*}
\texttt{<expression>} & : = \texttt{<primary>} | \texttt{<operator>}\texttt{<expression>} \\
& | \texttt{<shift>}\texttt{<operator>}\texttt{<expression>} \\
\texttt{<primary>} & : = \texttt{<constant>} | \texttt{<variable>} | \\
& | (\texttt{<expression>}) | \texttt{<unary \ op>}\texttt{<primary>} \\
\texttt{<variable>} & : = \texttt{<identifier>} | \texttt{<identifier>} \cdot \texttt{<identifier>} \\
& | \texttt{<identifier>} [\texttt{<expression>}] \\
\texttt{<shift>} & : = \texttt{<shift \ operator>}\texttt{<factor>} \\
\texttt{<factor>} & : = \texttt{<integer>} \\
\texttt{<operator>} & : = \texttt{<arithmetic \ operator>} | \texttt{<logical \ operator>} \\
& | \texttt{<relational \ operator>} | \texttt{<bit \ operator>} \\
\texttt{<arithmetic \ operator>} & : = + | - | * | \div \\
\texttt{<logical \ operator>} & : = \texttt{and} | \texttt{or} \\
\texttt{<relational \ operator>} & : = \ \texttt{LT} | \texttt{LE} | \texttt{GE} | \texttt{LT} | \texttt{GT} | \\
\texttt{<shift \ operator>} & : = \texttt{ALS} | \texttt{BLT} | \texttt{ALG} | \texttt{BRS} | \texttt{ALF} | \texttt{BLF} \\
& | \texttt{RAL} | \texttt{RBL} | \texttt{RAR} | \texttt{RBR}
\end{align*}
\texttt{<bit\ operator> ::= \texttt{IOR}|\texttt{XOR}}
\texttt{<unary\ op> ::= \texttt{NOT}|\texttt{-}}
\texttt{<constant> ::= <signed\ constant>|<unsigned\ constant>}
\texttt{<signed\ constant> ::= <sign><integer>}
\texttt{<sign> ::= +|\texttt{-}}
\texttt{<unsigned\ constant> ::= <integer>|<octal>|<character\ constant>}
\texttt{<integer> ::= <digit>^6}
\texttt{<digit> ::= 0|1|2|3|4|5|6|7|8|9}
\texttt{<octal> ::= <octal\ digit>^6\_B}
\texttt{<octal\ digit> ::= 0|1|2|3|4|5|6|7}
\texttt{<character\ constant> ::= \texttt{\#}<character\ string>\texttt{\#}}
\texttt{<character\ string> ::= <letter\ or\ digit>^6}
\texttt{<letter\ or\ digit> ::= <letter>|<digit>}
\texttt{<identifier> ::= <letter\ or\ digit>^6}
\texttt{<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}

\textbf{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.

\textbf{A.2.6.2.1 Unary minus (\texttt{-})}

Returns the negative of the primary, it precedes.
A.2.6.2.2 Logical not (not)

Returns a 1 if the value of the primary it precedes is zero and returns a zero if the value of the primary it precedes 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 adjacent 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, vbr, 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 non-zero 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) \( \equiv A2 \equiv \)

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.

\(<\text{comment}> :: = <\text{opening bracket}> <\text{almost anything}> <\text{closing bracket}>\)

\(<\text{opening bracket}> :: = /\star\)

\(<\text{closing bracket}> :: = \star/\)

\(<\text{almost anything}> :: = (\text{any string of valid ASCII characters which does not contain a } <\text{closing bracket}>\)}
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
- \( \text{lit} \) 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 HFCOM produces relocatable binary code), the operand field may contain relocatable expressions or absolute expressions which are less than 100 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 {m[,I]} comments
lit
```

Add the contents of m to A.
B.1.2.2  ADB

    label  ADB \{m[,I]\}  \textit{lit}  \textit{comments}

    Add the contents of \textit{m} to \textit{B}.

B.1.2.3  LDA

    label  LDA \{m[,I]\}  \textit{lit}  \textit{comments}

    Load \textit{A} from \textit{m}.

B.1.2.4  LDB

    label  LDB \{m[,I]\}  \textit{lit}  \textit{comments}

    Load \textit{B} from \textit{m}.

B.1.2.5  STA

    label  STA \{m[,I]\}  \textit{lit}  \textit{comments}

    Store contents of \textit{A} in \textit{m}.

B.1.2.6  STB

    label  STB \{m[,I]\}  \textit{lit}  \textit{comments}

    Store contents of \textit{B} in \textit{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 \{m[,I]\}  \textit{lit}  \textit{comments}

    The logical product of the contents of \textit{m} and the contents of \textit{A} are placed in \textit{A}.
B.1.3.2 XOR

label XOR \{m[I]\, lit\} 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 \{m[I]\, lit\} 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:

\[
\begin{align*}
\text{label} & \quad \text{[ALS} \quad \text{ARS} \quad \text{RAL} \quad \text{RAR} \quad \text{ALR} \quad \text{ALF} \quad \text{ERA} \quad \text{ELA} \quad \text{] } \quad \text{[,CLE] [,SLA]} \quad \text{[,CLE] [,SLB]} \\
\end{align*}
\]

\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 to 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 precede in the statement, skip the next instruction.

These instructions may be combined as follows:

\[
\begin{align*}
\text{label} & \left[ \begin{array}{c}
\text{CLA} \\
\text{CMA} \\
\text{CCA}
\end{array} \right] \left[ \begin{array}{c}
\text{SEZ} \\
\text{CLE} \\
\text{CMCE}
\end{array} \right] \left[ \begin{array}{c}
\text{CMA} \\
\text{CME} \\
\text{CMCE}
\end{array} \right] \left[ \begin{array}{c}
\text{SSA} \\
\text{SSA} \\
\text{SSS}
\end{array} \right] \left[ \begin{array}{c}
\text{INA} \\
\text{INB} \\
\text{INB}
\end{array} \right] \left[ \begin{array}{c}
\text{SZA} \\
\text{SZA} \\
\text{SZA}
\end{array} \right] \left[ \begin{array}{c}
\text{RSS} \\
\text{RSS} \\
\text{RSS}
\end{array} \right]
\end{align*}
\]
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 {m[I]} . comments
lit
```

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**

\[
\text{label DIV } \left\{ \begin{array}{l}
\text{m[1]} \\
\text{lit}
\end{array} \right. \text{ comments}
\]

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.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>NO</th>
<th>CL</th>
<th>SYMBOL</th>
<th>NO</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>1</td>
<td>0</td>
<td>ALS</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>INTEGER</td>
<td>2</td>
<td>1</td>
<td>BLS</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>CHAR</td>
<td>2</td>
<td>2</td>
<td>ARS</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>BRS</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>NOT</td>
<td>5</td>
<td>1</td>
<td>RAR</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>1</td>
<td>RBR</td>
<td>24</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>RAL</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>&amp;</td>
<td>6</td>
<td>3</td>
<td>RBL</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>AND</td>
<td>6</td>
<td>3</td>
<td>ALF</td>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>DIV</td>
<td>6</td>
<td>4</td>
<td>BLF</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>MOD</td>
<td>6</td>
<td>5</td>
<td>IOR</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>XOR</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>2</td>
<td>BEGIN</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>7</td>
<td>3</td>
<td>END</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>7</td>
<td>3</td>
<td>IF</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>&lt;</td>
<td>8</td>
<td>1</td>
<td>THEN</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>LT</td>
<td>8</td>
<td>1</td>
<td>ELSE</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>LE</td>
<td>8</td>
<td>2</td>
<td>CASE</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>8</td>
<td>3</td>
<td>REPEAT</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>GE</td>
<td>8</td>
<td>3</td>
<td>UNTIL</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>&gt;</td>
<td>8</td>
<td>4</td>
<td>WHILE</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>#</td>
<td>8</td>
<td>5</td>
<td>DO</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>=</td>
<td>8</td>
<td>6</td>
<td>FOR</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>EQ</td>
<td>8</td>
<td>6</td>
<td>DOWNTO</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>IN</td>
<td>8</td>
<td>7</td>
<td>GOTO</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>()</td>
<td>9</td>
<td>0</td>
<td>NIL</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>[]</td>
<td>10</td>
<td>0</td>
<td>TYPE</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>[ ]</td>
<td>11</td>
<td>0</td>
<td>ARRAY</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>[ ]</td>
<td>12</td>
<td>0</td>
<td>RECORD</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>15</td>
<td>0</td>
<td>FILE</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>16</td>
<td>0</td>
<td>LABEL</td>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td>.</td>
<td>17</td>
<td>0</td>
<td>CONST</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>:</td>
<td>21</td>
<td>0</td>
<td>VAR</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>:=</td>
<td>22</td>
<td>0</td>
<td>FUNCTION</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>/*</td>
<td>23</td>
<td>0</td>
<td>PROCEDURE</td>
<td>47</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: The table continues with more entries, but they are not included in this snippet.
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

<table>
<thead>
<tr>
<th>Index</th>
<th>Reserved Words of PL/2100</th>
<th>Index</th>
<th>Reserved Words of PL/2100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>32</td>
<td>REGA</td>
</tr>
<tr>
<td>2</td>
<td>DO</td>
<td>33</td>
<td>REGB</td>
</tr>
<tr>
<td>3</td>
<td>TO</td>
<td>34</td>
<td>THEN</td>
</tr>
<tr>
<td>4</td>
<td>OF</td>
<td>35</td>
<td>ELSE</td>
</tr>
<tr>
<td>5</td>
<td>OR</td>
<td>36</td>
<td>GOTO</td>
</tr>
<tr>
<td>6</td>
<td>LT</td>
<td>37</td>
<td>CASE</td>
</tr>
<tr>
<td>7</td>
<td>LE</td>
<td>38</td>
<td>WITH</td>
</tr>
<tr>
<td>8</td>
<td>GT</td>
<td>39</td>
<td>TYPE</td>
</tr>
<tr>
<td>9</td>
<td>GE</td>
<td>40</td>
<td>FILE</td>
</tr>
<tr>
<td>10</td>
<td>NE</td>
<td>41</td>
<td>BEGIN</td>
</tr>
<tr>
<td>11</td>
<td>EQ</td>
<td>42</td>
<td>UNTIL</td>
</tr>
<tr>
<td>12</td>
<td>AND</td>
<td>43</td>
<td>WHILE</td>
</tr>
<tr>
<td>13</td>
<td>END</td>
<td>44</td>
<td>ARRAY</td>
</tr>
<tr>
<td>14</td>
<td>NIL</td>
<td>45</td>
<td>VALUE</td>
</tr>
<tr>
<td>15</td>
<td>FOR</td>
<td>46</td>
<td>CONST</td>
</tr>
<tr>
<td>16</td>
<td>DIV</td>
<td>47</td>
<td>LABEL</td>
</tr>
<tr>
<td>17</td>
<td>MOD</td>
<td>48</td>
<td>PACKED</td>
</tr>
<tr>
<td>18</td>
<td>VAR</td>
<td>49</td>
<td>REPEAT</td>
</tr>
<tr>
<td>19</td>
<td>SYN</td>
<td>50</td>
<td>DOWNTO</td>
</tr>
<tr>
<td>20</td>
<td>ALS</td>
<td>51</td>
<td>RECORD</td>
</tr>
<tr>
<td>21</td>
<td>BLS</td>
<td>52</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>22</td>
<td>ARS</td>
<td>53</td>
<td>REGISTER</td>
</tr>
<tr>
<td>23</td>
<td>BRS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>RAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>RBR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>RAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>RBL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>ALF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>BLF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>XOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C.2.2 Table of Pointers

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>POINTER VALUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>55</td>
</tr>
</tbody>
</table>

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.

\[
\text{WNO} = (27,35,37,31,7,8,8,8,8,8,8,6,26,39,36,6,6,45,51,24,24,24,24,24,24,24,24)
\]

\[
24,24,24,24,24,52,52,28,29,38,30,49,40,41,25,33,34,41,48,43,42,53,32,37,41,46,50,47)
\]
\[ WCL = (0, 0, 1, 0, 3, 1, 2, 4, 3, 5, 6, 3, 0, 0, 0, 0, 0, 4, 5, 0, \\
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 0, 0, 0, \\
0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0) \]
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 EOF THEN  
BEGIN  
IF ERRINX GT 0 THEN PRTERR & PRINT ERROR +;  
EOLFLAG := FALSE ; CHCNT := 0 ;  
OUTPUT +1 := ; PUT(OUTPUT);  
IF OP THEN OUTF(OC) ELSE OUTF(OC) ;  
OUTPUT +1 := ; PUT(OUTPUT);  
END ;  
EOLFLAG := ; GET(INPUT) ; CH := INPUT ; CHCNT := CHCNT + 1 ;  
IF CHCNT GT 73 THEN  
BEGIN  
OUTPUT +1 :=  
PUT(OUTPUT);  
OUTPUT +1 := CH ;  
PUT(OUTPUT);  
REPEAT  
GET(INPUT) ; CH := INPUT ;  
OUTPUT +1 := CH ;  
PUT(OUTPUT);  
UNTIL CH = EOL ;  
END ELSE  
BEGIN OUTPUT +1 := CH ; PUT(OUTPUT);  
IF CH = EOF THEN GOTO 10 ;  
END ;  
CH := ; EOLFLAG := TRUE ;  
101 END ; NEXTCH +
*************  *************  *************
PROCEDURE INSYMBOL;
    * LEXICAL ANALYSER *
    VAR
    I, K : INTEGER;
    DIGIT & ARRAY [0..4] OF INTEGER;
    BEGIN
        WHILE CH # 32 DO NEXTCH;
        IF CH # 25 THEN * IDENTIFIER *
            BEGIN
                K1 := 0;
                REPEAT
                    K1 := K1 + 1;
                    A[K1] := BLANK;
                UNTIL K1 = 10;
                K1 := 0;
                REPEAT IF K < ALFALEN THEN
                    BEGIN
                        K := K + 1;
                        A[K] := CH;
                        END;
                    END;
                    NEXTCH;
        UNTIL CH = 9.
        SEARCH FOR RESERVED WORDS;
        FOR I1 := ML(K) TO ML(K + 1) - 1 DO;
            BEGIN
                CPARE(A, HD, I, COMPARE);
                IF COMPARE THEN
                    BEGIN
                        NO := WNO[I1];
                        CL1 := WCL[I1]; IVAL1 := 0;
                        GO TO 2;
                    END;
        END;
        NO := 0;
        CL1 := K1;
        I := 0;
        REPEAT
            I := I + 1;
            AVAL[I] := BLANK;
        UNTIL I = 10;
        I := 0;
        REPEAT
            I := I + 1;
            A[I] := A[I1];
        UNTIL I = K;
        IF CH # 9 THEN * NUMBER *
            BEGIN
                NO := 0;
                CL1 := 1;
                I := 0;
            END;
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 = '0' 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
    FOR K = 0 TO I - 1 DO
        IF IVAL > MAX10 THEN
            IVAL = 10*IVAL + DIGIT[K];
        ELSE BEGIN ERROR(2) ;
            IVAL = 0; ENJ ;
        END;
    END;
END;
ELSE
BEGIN SPECIAL CHARACTER +
IF CH = ' ' THEN CHAR CONSTANT + THEN
    BEGIN NO = 2;
    BT1 = FALSE;
    REPEAT NEXTCH;
    IF CH = ' ' THEN
        BEGIN NEXTCH; BT1 = CH = ' ' ENJ ;
    IF BT1 THEN
        IF K = ALPHALG THEN
            BEGIN ERROR(54) ;
            BT1 = TRUE ;
        END;
    ELSE BEGIN K = K + 1; A[K] = CH END ;
    UNTIL BT1;
    CL = 2; TECH = A[1]; IVAL = ASCIIODEITECH ;
    IF K EQ 2 THEN
    BEGIN TECH = A[2]; TEMPI = ASCIIODEITECH ;
        APPEND(IVAL, TECH);
    END ELSE APPEND(IVAL, 8, 408) ;
END = '
ELSE
BEGIN NO1 = SYMNO(CH) ; CL = SYMCL(CH) ;
    IVAL = 0 ;
    TEST FOR TWO CHARACTER SYMBOL +
    IF CH = ' ' THEN
BEGIN
NEXTCCH:
  IF CH = "=" THEN
  BEGIN NO1 = 22 ; NEXTCCH END ;
  END ELSE
  IF CH = "*" THEN
  BEGIN NEXTCCH : IF CH = "*" THEN + SKIP COMMENT +
  BEGIN NEXTCCH:
  WHILE CH = "*" DO NEXTCCH ;
  NEXTCCH : IF CH = "/" THEN
  GOTO 3 ELSE
  NEXTCCH ; GOTO 1 ;
END ELSE
NEXTCCH ;
END
END " SPECIAL CHARACTER "
END " INSAYMBOL "

***************************************************************
APPENDIX E

SYNTAX DIAGRAM OF PL/2100
parameter list

identifier

identifier

CONST

VAR

FUNCTION

PROCEDURE

identifier
field list

CASE identifier -> identifier OR
constant -> field list
statement part

integer : statement END

; type

identifier

(1) identifier

(1) constant + constant

constant - constant

ARRAY (1) type (1) OF type

RECORD field list END
APPENDIX F

LISTING OF I/O ROUTINES
ASMB, L, R

0002 HAM DR1, 7
0003 *
0004 * THIS ROUTINE GETS THE INPUT IN THE BUFFER
0005 * RDCON----CONWD
0006 * READQ----REQUEST CODE = I
0007 * BUFFS----BUFFER ADDRESS
0008 * BUFLS----BUFFER LENGTH
0009 * INPUT------
0010 *
0011 * B-REGISTER----ADDRESS OF THE TOP OF THE BUFFER

0012 EXT EXEC
0013 ENT GET
0014 EQU 1000
0015 OPFLG EQU .+113
0016 GET NOP
0017 STA BUFFS STORE BUFFER ADDRESS
0018 LDA GET GET THE
0019 INA LOGICAL UNIT
0020 LDB 0, I NUMBER IN RDCON
0021 STB RDCON
0022 JSP EXSET
0023 JSB EXEC
0024 DEF ++5
0025 DEF READQ
0026 DEF RDCON
0027 DEF BUFFS, I
0028 DEF BUFLS
0029 JMP GET, I
0001  ASMB, L, R
0002  HAM  DR2, 7
0003  *
0004  *  THIS ROUTINE WRITES A VARIABLE LENGTH. *
0005  *  BUFFER ONTO THE OUTPUT DEVICE *
0006  *  INPUT----- *
0007  *  A-REGISTER----BUFFER LENGTH *
0008  *  B-REGISTER----BUFFER ADDRESS *
0009  *
0010  EXT  EXEC
0011  ENT  PUT
0012  EQU  100B
0013  OPFLG  EQU  +113
0014  PUT  NOP
0015  STA  BUFLS
0016  *
0017  *  STORE BUFFER LENGTH *
0018  *
0019  STB  BUFFS
0020  *
0021  *  STORE BUFFER ADDRESS *
0022  *
0023  LDA  PUT  GET THE
0024  IHA  LOGICAL UNIT
0025  LDB  0, I  NUMBER IN B-REGISTER
0026  STB  WRCON  STORE IN WRCON
0027  CLA
0028  CLB
0029  LDB  OPFLG
0030  JSB  ESXSET  PREPARE EXEC FOR OUTPUT
0031  JSB  EXEC
0032  DEF  *+5  RETURN ADDRESS FROM EXEC
0033  DEF  WRITQ  WRITE CODE = 2
0001 ASMB, L, R
0002 NAM DR3, 7
0003 ENT ALOK
0004 *
0005 *
0006 *
0007 *
0008 *
0009 *
0010 *
0011 *
0012 *
0013 *
0014 *
0015 ALOK, N0P
0016 STB TEMPY
0017 LDA BLANK
0018 STA CHAR
0019 SEZ
0020 JMP L4 * CHARACTER INPUT
0021 CLA
0022 STA SIGN
0023 STA DIG
0024 LDA CRSW
0025 STA CRFG
0026 L1 JSB NTBLK * GET NON-BLANK CHARACTER
0027 CPA COMMA
0028 JMP L1 * YES
0029 CPA MINUS
0030 RSS * YES
0031 JMP L2
0032 LDB MINUS
0033 STB SIGN
0034  JSB  GETCR  GET NEXT CHARACTER
0035   L2  JSB  DIGIT FORM THE DIGITS
0036  JSB  NTBLK
0037  CPA  COMMA
0038  JMP  L3  YES
0039  LDB  DIG IF 'NOT' COMMA
0040  BLS, BLS  MULTIPLY
0041  ADB  DIG BY
0042  BLS  ,  10
0043  STB  DIG STORE THE NUMBER IN DIG
0044  JMP  L2  GET MORE DIGIT
0045   L3  LDA  DIG
0046  LDB  SGN
0047  SZA
0048  CMP, IHA IF NEGATIVE NUMBER, TAKE 2'S COMPLEMENT
0049  LDB  TEMPY
0050  JMP  ALL0K, I
0051  GETCR  GETS NEXT CHARACTER
0052  GETCR  NOP
0053  LDB  CRFG CRFG IS BOOLEAN FLAG
0054  SSB
0055  JMP  GETC2
0056  LDA  TEMPY, I
0057  AND  CH2
0058  ALF, ALF
0059  GETC1  GET THE FIRST CHARACTER
0060  GETC1  LDB  CRFG
0061  RBR
0062  STB  CRFG
0063  JMP  GETCR, I
0064  *  GET NEXT SECOND CHARACTER
0065     GETC2  LDA  .TEMPP, I
0066        AND  CH1
0067        LDB  TEMPP
0068        INB
0069        STB  TEMPP
0070        JMP  GETC1
0071  *  NTBLK GET NON-BLANK CHARACTER
0072     NTBLK  NOP
0073     NTBL1  JSB  GETCR
0074        CPA  BLANK
0075        RSS  YES
0076        JMP  NTBLK, I
0077        JMP  NTBL1
0078  *  DIGIT GETS THE DIGIT FROM ASCII CHARACTER
0079     DIGIT  NOP
0080        LDB  SIXTY
0081        CMB, INB
0082        ADB  0
0083        ADB  DIG
0084        STB  DIG
0085        JMP  DIGIT, I
0086  *  CHARACTER INPUT
0087     L4  JSB  NTBLK
0088        CPA  COMMA
0089        JMP  L5
0090        STA  GCHAR
0091     L5  JSB  NTBLK
0092        CPA  COMMA
0093        JMP  L6
0094 STA CHAR
0095 JMP L5
0096 L6 LDA CHAR
0097 LDB TEMPY
0098 JMP ALLOK, I
0099 CRSW OCT 52525
0100 MINUS OCT 55
0101 COMMA OCT 54
0102 TEMPY 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 ****
0001 ASMB, L, R
0002 HAM DR4, 7
0003 ENT ALLOC
0004 *
0005 *
0006 *
0007 *
0008 *
0009 *
0010 *
0011 *
0012 *
0013 *
0014 *
0015 *
0016 *
0017 *
0018 *
0019 *
0020 *
0021 *
0022 *
0023 *
0024 *
0025 ALLOC
0026 NOP
0027 STA TEMPT
0028 STB TEMPA
0029 CLB
0030 STB BUFL
0031 STB KOUNT

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

STORE INTEGER OR CHARACTER
STORE BUFFER POINTER
INITIALIZE BUFFER LENGTH
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
ADD SIXTY
STB 0, I
INCREMNET THE COUNT OF THE DIGIT
LDA KOUNT
INA
STA KOUNT
LAB1
JMP LAB2  GET NEXT DIGIT
LDA TEMP2
ADA KOUNT
ADD SIXTY
LAST DIGIT IN TEMPORARY LOCATION
STB 0, I
STORE THE SIGN
LDA SIGN
0064 LAB3      ALF, ALF
0065      STA HUMB
0066      JSB INBFL
0067      IOR HUMB
0068 *      PUT SIGN AND THE DIGITS IN THE BUFFER
0069 *      DECREMENT THE KOUNT TO GET THE PROPER DIGITS
0070      JSB PUTBL
0071      CCA
0072      ADA KOUNT
0073      STA KOUNT
0074      SSA
0075      JMP LAB5
0076      SZA, RSS
0077      JMP LAB4
0078      JSB INBFL
0079      CCB
0080      ADD KOUNT
0081      STB KOUNT
0082 *      GET NEXT DIGIT TO BE PUT IN THE BUFFER
0083      JMP LAB3
0084 LAB4      JSB INBFL
0085      ALF, ALF
0086      IOR BLANK
0087      JSB PUTBL
0088 *      PUT TWO BLANKS AFTER EACH OUTPUT
0089 LAB5      LDA BLANK
0090      ALF, ALF
0091      IOR BLANK
0092      JSB PUTBL
0093      JMP ALLOC, I
0094 *      CHARACTER OUTPUT
0095 *      

140
0096  *  
0097  LAB6  LDA BLANK  
0098  ALF ALF  
0099  IOR BLANK  
0100  JSB PUTBL  STORE BLANK IN BUFFER  
0101  LDA TEMPT  LOAD THE CHARACTER  
0102  JSB PUTBL  STORE THE CHARACTER IN BUFFER  
0103  JMP LAB5  
0104  *  STORE THE CONTENTS OF A-REGISTER IN THE BUFFER  *  
0105  *  UPDATE BUFFER LENGTH AND BUFFER POINTER  *  
0106  *  
0107  
0108  PUTBL  NOP  
0109  LDB BUFL  
0110  ADB TEMPA  
0111  STA 1,I  
0112  LDA BUFL  
0113  INH  
0114  STA BUFL  
0115  JMP PUTBL,I  
0116  *  INCREMENT THE COUNTER--KOUNT-- FOR THE DIGIT  *  
0117  *  ADJUST THE ADDRESS OF TEMPORARY LOCATION  *  
0118  *  
0119  
0120  INBFL  NOP  
0121  LDB KOUNT  
0122  ADB TEMP2  
0123  LDA 1,I  
0124  JMP INBFL,I  
0125  TEMP2  DEF TEMP  
0126  TEMP  BSG 20
APPENDIX G

LISTING OF THE HPCOM COMPILER
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 NAME 'V' PROGRAM <PROGRAMNAME> (, <OPTIONAL PARAMETERS>) WHERE <OPTIONAL PARAMETERS> ARE:

A = WRITES OUT CODE STACK.
B = WRITES OUT THE RELOCATABLE CODE.
C = PUNCHES OUT THE RELOCATABLE CODE ON THE FILE "HPDIP".

TO RUN HPCOM, CONTROL CARDS NEEDED ARE:

1. JOB CARD
2. ATTACH (PASCAL, ID=GQPASCAL, MR=1, CY=20)
3. PASCAL
4. REQUEST (HPMPL, NEWCOM-PACKARD CARDS NEEDED)
5. DISPOSE (HPDIP, P=0, C=0)

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 LOADER" DIRECTION TO LOAD THE PROGRAM DIRECTLY.

RUN <PROGRAMNAME> --- TO RUN THE PROGRAM.
TYPE
BITS = 0..1;
RTP = 0..5;
BITRANGE = 0..16;
RG3 = 0..3;
ADDRESS = 0..1,7778;
SRMINT = -1,7778 .. +1,7778;
AHRINT = ARRAY [1..3] OF INTEGER;
AR = ARRAY [1..10] OF CHAR;
DELARY = ARRAY [1..3, 1..10] OF CHAR;
INTARR = ARRAY [1..32, 1..10] OF CHAR;
OPTWR = (+ NOOPT, PUREP, POSP, NEGP);
IDKLASS = (TYPES, KONST, PROC, VAR, FIELD, TAGFIELD);
TYPFORM = (NUMERIC, SYMBOLIC, ARRAYS, RECORDS, FILES, REGISTERS);
IDKINDS = (ACTUAL, FORMAL);
WHERE = (BLOCK, WITH, VITH);
ATTRKIND = (VARBL, SVNL, LVAL, LCND);
ARGCODE = ARRAY [1..CODMAX] OF RECORD
  LOG, INST, ADRR & INTEGER; TYPRTYP, HREF: BOOLEAN;
END;
ATTR = RECORD
  TYPIR & INTEGER;
  CASE KIND : ATTRKIND OF
   VARBL : (ACCESS, IDRCFT, INDRCT, INXC);
   BREGGR : (VARLMT, INTEGER);
   CASE PCKD : BOOLEAN OF
      FALSE ;
      TRUE : (BITADR, BITSZ, BITRANGE));
   SVNL : (VAL, INTEGER);
   LVAL : (CTERM, INTEGER);
   LCND : (JMP, 0..3) ARITH: BOOLEAN)
END;

VAR

HPAS, ARCO, CODE STACK
MCA, INTEGER; USED TO INDEX CODE STACK
BFLY, BLC, INTEGER
BFLG -- 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;
FILG OF RELOCATABLE CODE TO BE PUNCHED OUT;
R, USED BY DIFFERENT ROUTINES USED FOR THE RELOCATION
OF THE CODE STACK
RNAH; A ARRAY [1..60] OF INTEGER;
TRANS; I, ARRINT3
CERR, LISTLEN, PUNCHBIN, XLISTBIN, BOOLEAN
* OPTIONAL PARAMETERS ARE USED ALONG WITH THESE VARIABLES
* THESE VARIABLES ARE SET TRUE IF OPTIONAL PARAMETERS
* ARE PRESENT

XPT, EXSYM, INTEGER; -- USED FOR EXTERNALS
EXTX; ARRAY [1..30] OF
RECORD XNAH (AR1), XSYM (INTEGER), END;
* EXTX IS USED TO STORE THE NAME OF THE EXTERNALS
LABTAB; ARRAY [1..MAXLAB] OF
PACKED RECORD LABVAL, FLO2, FLO3, INTEGER, END;
CLABX; INTEGER
* ALL LABELS MEASURING FROM THE BODY OF THE PROCEDURE
* ACTUALLY BEING COMPILED TOGETHER WITH INFORMATION WHETHER
* LABEL DEF. ([LABEL]) ALREADY FOUND OR NOT (DLO2).
* IN THE FORMER CASE FLO3 CONTAINS THE CORRESPONDING
* ADDRESS WHERE IN THE LATTER CASE FLO3 CONTAINS AN INDEX
* INTO UNOlab WHERE THE OCCURRENCE ARE CHAINED

STORE; ARRAY [1..KMAX] OF
RECORD
SP; INTEGER; STPL; INTEGER;
END;
STRE; ARRAY [1..10] OF
RECORD
STP; INTEGER; STCL; INTEGER;
END;
RMAX, RP, KP; INTEGER; -- USED WITH STORE AND STRE
* STORE AND STRE ARE USED BY ROUTINES STMP AND CLMP
* 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..5;
ERRINX: POS & SHRINT;
ERRORTLIST: ARRAY [1..10] OF INTEGER;
ERRKOUNT: INTEGER;
ERR: BOOLEAN;
CHNT: INTEGER;
* ERRLIST CONTAINS THE POSITION AND NUMBERS OF THE
  ERROR ON ONE LINE.
ERRINX = TOP OF ERRLIST;
POS1 = POSITION OF LAST PRINTED ERRORMARK(*);
CHNT1 = POSITION OF LAST READ CHARACTER;
ERRLIST, ERRINX, POS1, CHNT, EOLFLAG---- ALL ARE
USED BY NEXTIE AND ERROR;
ERRKOUNT = KEPS TRACK OF THE NUMBER OF ERRORS
IN A PROGRAM.
ERRORTLIST = KEPS 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 FORWARD DECLARED
  PROCEDURES AND GOTO STATEMENTS LEADING OUT OF PRO;

CSFIB: ARRAY [1..CSFMAX] OF RECORD VAL: INTEGER; INX: SHRINT END;
LFX: SHRINT;
* CONTAINS ALL CONSTANTS C. OCCURRING IN THE
  PROCEDURES ACTUALLY BEING COMPILED TOGETHER.
AN INDEX INTO UNLAB WHERE THEIR OCCURENCE IN
THE CODE OF THIS PROCEDURE ARE CHAINED

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

UNDLAB: ARRAY [1..UNDMAXI] OF RECORD SUC, PLACE: INTEGER END;
CHIX | SHRINT
ACTS AS A LIST STRUCTURE, CHAINING
OCURRENCE OF CONSTANTS AND JUMPS TO NOT YET
RESOLVED SYMBOLS IN THE CODE OF THE PROCEDURES
ACTUALLY BEING COMPILED
CHIX = HEAD OF FREE LIST

Following variables are used by procedures NEXTCH
AND INSsymbol:
NOCL ARE THE TOKEN INTEGER ASSOCIATED WITH IDENTIFIER
AND RESERVED SYMBOLS OF THE LANGUAGE PL/2100.

LC,L1 INTEGER
DP,B1,EOLFLAG BOOLEAN
CH,TECH CHAR
NOC,C INTEGER
VAL,AL INTEGER
A1,VALAR AR
DIGITS SET OF CHAR

CONSTANT POINTERS
INIPTR,CHARPTR,BOOLPTR,PREDEFP NILPTR,UNOCLPTR | SHRINT
BYTPTR,ROSPTR,INDEFP,UNDECPTR | SHRINT
* USED BY VARIABLE AND SYNONYM DECLARATION PART
* ALSO BY TYPE DECLARATION PART
LL,L21,WLC INTEGER
AI,I11 INTEGER
PINPTR SYNT | SHRINT
FLIPTR SYNT | INTEGER
TPTR AR
TL INTEGER
SUPPTR,PROCPT | OLDENV INTEGER
* USED BY VALUE DECLARATION PART
IT,PI INTEGER
VLPT | BOOLEAN
VARS ARRAY[1...100] OF INTEGER
* THESE ARE USED TO KEEP THE VARIABLE ATTRIBUTES
GATTR,ARRAY,ARRAY | ARRAY |\ ATTR
COMPAR,COMPARE BOOLEAN | * USED BY SEARCH ROUTINES
* USED BY EXPRESSION ROUTINES
CITY BOOLEAN
GP,LADOPTL INTEGER
WRFL BOOLEAN | * USED BY READ AND WRITE ROUTINES
* USED FOR TEMPORARIES BY VARIOUS ROUTINES
I,J,K,M,EM,EMPT | INTEGER
CEXIBIX,CHIX | INTEGER
FSTIX  INTEGER
* VARIABLES POINTING TO CONTEXTTABLE
* AUXILIARY VARIABLES USED IN MAIN PROG. AND SEVERAL
PROCEDURES
INDEX  SHRINT
NEXT, CIPTR, MAXCIP, SHRINT
DISPLAY ARRAY [0..DISPLIMIT] OF
RECORD FNAME, INTEGER
CASE OCCUR WHERE OF
BLOCK:
WITH (COSP, INTEGER, CLEVEL3)
WITH (VOSPL, INTEGER)

END
TOP, JISX  INTEGER
LEVEL  INTEGER
CONTEXTTABLE CONTAINS THE IDENTIFIERS ALONG WITH
THEIR ATTRIBUTES
CONTEXTABLE ARRAY(0..250) OF PACKED

RECORD NAME: AR $ NXTEL: SHRINT $ SYMCLASS: BOOLEAN $ IDKCLASS OF:

TYPES { (SIZE $ ADDRESS $ CASE FORM $ DYPFORM OF:

NUMERIC { BIT1: BITRANGE; MIN, MAX: INTEGER };
SYMBOLIC { FCONST: INTEGER; BITSIZE: BITRANGE };
ARRAYS { AETYPE: INXTYPE; SHRINT } $
LOHT: SHRINT; SIZE: BOOLEAN, OPTYPE: OPTYPE $
EXP1, EXP2: BITRANGE } $
RECORDS { IFSTFLO, RECVAR: INTEGER } $
FILES $ $ REGISTERS } $

KONST { (CONTYPE: INTEGER) $ CASE CUNKIND: IDKINDS OF:

ACTUAL { (SUCC: INTEGER; VALUES: INTEGER) } $
FORMAL: { (CADDR: ADDRESS; LEVEL: RG3) } } $
PROC { (PROCTYPE: FORMALS $ INTEGER) $
PROCKind: IDKIND $ PROCADUR: ADDRESS $ PROCELEVEL: RG3; SEGSIZE: INTEGER } $

WARS { (WTYPE: INTEGER; WIND: IDKINDS) $
SYNPRI: SHRINT $ WADDD: ADDRESS $ WLEVEL: RG3 } $ FIELD { (FITYPE: INTEGER: FADDR: ADDRESS) $
FITDISPL: BITWIDTH $ BITRANGE } $
TAGFIELD { (CASESIZE: INTEGER $ VARIANTS: INTEGER) $ CASE TAGVAL: BOOLEAN $
FALSE: { (CASETYPE: INTEGER) } $ TRUE: { (CASEVAL: INTEGER) } } $ END;
PROG --- CONTAINS PROGRAM
PROGEND --- CONTAINS PROGEND
MU --- INITIALIZE TO ALL THE RESERVED WORDS OF PL/2100
WNJ,WCL --- CONTAINS INTEGER TOKENS NJ AND CL
SYMNO,SYMCL --- TOKENS NJ AND CL FOR BCD CHARACTER SET
OF THE RESERVED WORDS OF PL/2100
INITNAM --- USED TO INITIALIZE THE CONTEXTABLE
ASCIICODE --- ASCII CODE CORRESPONDING TO THE BCD CHARS
DE --- USED IN ROUTINES FOR RELOCATION

PROG  AR
PROGEND  AR
BLANK  AR
SPLITSTAT  ARRAY [1..52] OF INTEGER
ERRCL, TERRCL  ARRAY [1..52] OF INTEGER
WD  DELARRY
WNJ,WCL  ARRAY [1..54] OF INTEGER
WL  ARRAY [1..11] OF INTEGER
SYMNO,SYMCL  ARRAY [1..2;= 1 OF INTEGER
INITNAM  INTARR
ASCIICODE  ARRAY [CHAR] OF INTEGER
DE  ARRINT3
<table>
<thead>
<tr>
<th>RESERVED WORD AND SYMBOLS USED IN HP-2100</th>
<th>NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>CONST</td>
</tr>
<tr>
<td>CHAR</td>
<td>CONST</td>
</tr>
<tr>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td></td>
</tr>
<tr>
<td>MOD</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td></td>
</tr>
<tr>
<td>GE</td>
<td></td>
</tr>
<tr>
<td>NE</td>
<td></td>
</tr>
<tr>
<td>EQ</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>ALS</td>
<td></td>
</tr>
<tr>
<td>BLK</td>
<td></td>
</tr>
<tr>
<td>ARS</td>
<td></td>
</tr>
</tbody>
</table>
Diagram of a computer program or algorithm, showing flowchart-like structure with procedures and variables.
PROCEDURE CMP (VAR A, B: ARRAY; VAR COMPARE: BOOLEAN) ;
   * COMPARE THE ARRAYS A AND B. COMPARE IS TRUE IF THEY ARE
   * EQUAL, OTHERWISE IT IS FALSE.

VAR K: INTEGER
BEGIN
   K := 1
      BEGIN
         K := K + 1
         IF K <= 10 THEN GOTO 1
         ELSE COMPARE := TRUE
      END
   ELSE COMPARE := FALSE
END CMP

PROCEDURE COMPARE (VAR A, B: ARRAY; VAR B: DELARRAY; M: INTEGER;
                     VAR COMPARE: BOOLEAN) ;
   * COMPARE THE ARRAYS A AND B. COMPARE IS TRUE IF THEY ARE
   * EQUAL, OTHERWISE IT IS FALSE.

VAR K: INTEGER
BEGIN
   K := 1
      BEGIN
         K := K + 1
         IF K <= 10 THEN GOTO 1
         ELSE COMPARE := TRUE
      END
   ELSE COMPARE := FALSE
END COMPARE
FUNCTION LOG2 (VAL1 : INTEGER) : INTEGER;
VAR E1 : INTEGER;
BEGIN
  E1 := 0;
  VAL1 := VAL1;
  WHILE VAL1 GT 0 DO
    BEGIN
      VAL1 := VAL1 DIV 2;
      E1 := E1 + 1
    END;
  END;
END LOG2;

PROCEDURE MULOPT (VAL1 : INTEGER; VAR EXP1, EXP2 : INTEGER);
VAR OPT : OPTPHR;
BEGIN
  IF VAL1 = 2**EXP1
    THEN
      BEGIN
        OPT := PUREP;
        EXP1 := EXP1 ;
      END
    ELSE
      BEGIN
        OPT := OTHER;
      END;
  END;
BEGIN
  E1 := 0;
  EXP1 := 0;
  IF VAL1 = 0 THEN
    BEGIN
      E1 := 0;
      VAL1 := VAL1;
      WHILE NOT ODD (VAL1) DO
        BEGIN
          VAL1 := VAL1 DIV 2;
          E1 := E1 + 1
        END;
    END
  ELSE
    BEGIN
      OPT := PEP;
      EXP1 := EXP1;
    END;
  END;
END;
BEGIN
  E2 := 0;
  VAL1 := VAL1 DIV 2;
  E2 := E2 + 1;
  IF ODD (VAL1) THEN
    BEGIN
      REPEAT
        BEGIN
          VAL1 := VAL1 DIV 2;
          E2 := E2 + 1
        END
      UNTIL NOT ODD (VAL1);
      IF VAL1 GT 0 THEN OPT := NOOPPT ELSE
        BEGIN
          OPT := NEGEP;
          EXP1 := EXP1;
          E1 := E1
        END
    END
  ELSE
    BEGIN
      REPEAT
        BEGIN
          VAL1 := VAL1 DIV 2;
          E2 := E2 + 1
        END
      UNTIL ODD (VAL1);
      IF VAL1 GT 0 THEN OPT := NOOPPT ELSE
        BEGIN
          OPT := POSP;
          EXP2 := EXP2;
          EXP1 := EXP1;
        END
    END
END;
PROCEDURE PR_TERR;
VAR I, K, POSIT, SHRINT;
BEGIN
OUTCH("E=E");
FOR I:=1 TO 4 DO OUTCH("E=E");
I:=I+1; K:=K+1;
REPEAT
POSIT:=ERRLIST[K].POS;
WHILE I LT POSIT DO
BEGIN
OUTPUT* I:=E=E; PUT(OUTPUT);
I:=I+1; K:=K+1;
END;
OUTPUT* I:=E=E; PUT(OUTPUT);
I:=I+1; K:=K+1;
UNTIL K GT ERRINX;
WHILE I LE 60 DO
BEGIN
OUTPUT* I:=E=E; PUT(OUTPUT);
I:=I+1;
END;
FOR I:=1 TO ERRINX DO WRITE(ERRLIST[I].NMR);
OUTCH("EJ"); ERRINX:=0; POSIT:=0;
END; PR_TERR;**********
**********
**********
**********
PROCEDURE ERROR(I, SHRINT);
BEGIN
ERR:=TRUE;
IF ERRRKJN EQ 0 THEN
BEGIN
ERRKCOUNT := ERRKCOUNT + 1;
ERRLIST[ERRKCOUNT] := I;
END ELSE
BEGIN
J:=0;
REPEAT
J:=J+1;
IF ERRLIST[J] EQ I THEN GOTO 1;
UNTIL J EQ ERRKCOUNT;
ERRKCOUNT := ERRKCOUNT + 1;
IF ERRKCOUNT EQ 10 THEN GOTO 1;
ERRLIST[ERRKCOUNT] := I;
END;
ERRNL[I DIV 32] := ERRNL[I DIV 32] OR (I MOD 32);
IF ERRINX GT 9 THEN
WITH ERRLIST[10] DO
PROCEDURE ERMHSSG1;
VAR I,J:INTEGER;
**********
PROCEDURE ERMHSSG1;
BEGIN
CASE J OF
   18 WRITE("SCALAR=,ETYPE=,EXPECTED=EOL");
   12 WRITE("INTEGER=,LARGE,EOL");
   54 WRITE("ERROR: IN = CONSTANT,EOL");
   41 WRITE("=,EXPECTED,EOL");
   51 WRITE("FIELD,DECLARED=,ETHICE,EOL");
   56 WRITE("BAD,EXCEPLATE,EOL");
   74 WRITE("=,EXPECTED,EOL");
   88 WRITE("=,DECLARED,ETYPE,BAD,EOL");
   104 WRITE("=,EXPECTED,EOL");
   118 WRITE("=,DECLARED,ETYPE,NOT,EOL");
   170 WRITE("=,EXPECTED,EOL");
   134 WRITE("INDEX,MUST,OF,EOL");
   148 WRITE("END,EXPECTED,EOL");
   161 WRITE("=,EXPECTED,EOL");
   173 WRITE("=,EXPECTED,EOL");
   164 WRITE("=,EXPECTED,EOL");
   139 WRITE("=,EXPECTED,EOL");
   152 WRITE("=,EXPECTED,EOL");
   174 WRITE("=,EXPECTED,EOL");
   157 WRITE("=,EXPECTED,EOL");
   170 WRITE("=,EXPECTED,EOL");
   181 WRITE("=,EXPECTED,EOL");
   162 WRITE("=,EXPECTED,EOL");
   178 WRITE("=,EXPECTED,EOL");
   191 WRITE("=,EXPECTED,EOL");
   204 WRITE("=,EXPECTED,EOL");
   216 WRITE("=,EXPECTED,EOL");
   228 WRITE("=,EXPECTED,EOL");
   240 WRITE("=,EXPECTED,EOL");
   252 WRITE("=,EXPECTED,EOL");
   264 WRITE("=,EXPECTED,EOL");
   276 WRITE("=,EXPECTED,EOL");
   288 WRITE("=,EXPECTED,EOL");
   301 WRITE("BEGIN",CASE,EOL");
END;
**********
PROCEDURE ERMHSGG2;
BEGIN
CASE J OF
   31 WRITE("CONFLICT=,OF,INDEX=,WITH=,EXPECTED=EOL");
   32 WRITE("=,EXPECTED=EOL");
   33 WRITE("=,EXPECTED=EOL");
END;
**********
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 ;
18
END;
FOR I = 1 TO ERRORKOUNT DO
BEGIN
    J := ERRORLIST[I] ;
    IF J LE 40 THEN ERRMSSG1 ELSE
    IF J LE 80 THEN ERRMSSG2 ELSE ERRMSSG3 ;
END ;
END #ERRMSSG +
PROCEDURE NEXTCCH ;
READS NEXT CHAR OF FILE INPUT;
READ 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 ;
INPUT ← E ; PUT (OUTPUT) ↓
END WHEN → EOLFLAG ↓
END
GET (INPUT) ; CH ← INPUT ; CHCNT ← CHCNT + 1 ;
IF CHCNT → 73 THEN
BEGIN
OUTPUT ← CH ; PUT (OUTPUT) ↓
END
ELSE
BEGIN
OUTPUT ← CH ; PUT (OUTPUT) ↓
END
IF CH ← EOL THEN GOTO 10
END
CH ← E ; EOLFLAG ← TRUE ↓

PROCEDURE INSsymbol ↓
LEXICAL ANALYSER ↓

VAR
DIGIT : ARRAY [0... 4] OF INTEGER ↓

BEGIN
WHILE CH ← EDO NEXTCCH ↓
BEGIN
IF CH ← EZ THEN IDENTIFIER ↓
BEGIN
K ← 0 ↓
REPEAT
K ← K + 1 ↓
END
A[K] i = BLANK;
UNTIL K = 10;
K := 0;
REPEAT IF K < ALFALENG THEN
BEGIN
K := K + 1;
A[K] := CH;
END;
NEXTCH;
UNTIL CH >= 99;
SEARCH FOR RESERVED WORDS +
FOR I = HL[K] TO HL[K+1] - 1 DO
BEGIN
COMPARE(A,NO,I,COMPARE);
IF COMPARE THEN
BEGIN NO := NNO[I];
CL := HCL[I]; IVAL := 0;
GOTO '2';
END;
END;

i := K;
REPEAT
i := i + 1;
AVAL[i] := BLANK;
UNTIL i = 10;
i := 0;
REPEAT
i := i + 1;
AVAL[i] := A[i];
UNTIL i = K;

21 END ELSE
IF CH < 99 THEN
BEGIN
NO := 2;
CL := 1;
i := 0;
WHILE CH IN DIGITS DO
BEGIN IF I < 5 THEN
DIGIT[I] := ORU(CH) := ORU("0" - 1);
i := i + 1;
NEXTCH;
END;
IVAL := 0;
IF CH = 32 THEN
BEGIN IF I > 5 THEN ERROR(2) ELSE
FOR K := 0 TO i - 1 DO
IVAL := 0;IVAL + DIGIT[K];
NEXTCH;
END ELSE
BEGIN IF i > 4


THEN ERROR(2) ELSE
FOR K1=0 TO I=1 DO
BEGIN IF IVAL< MAX10 THEN
IVAL=10*IVAL+DIGIT(K)
ELSE BEGIN ERROR(2) I
IVAL=0 I END ;
END
BEGIN IF CHAR+ THEN
BEGIN NO=21 K1=0 BT11=FALSE ;
REPEAT NEXTCH 
IF CHAR= THEN BT11= CHAR END ;
IF BT11 THEN
IF K=ALFALEN THEN
BEGIN ERROR(54) ;
BT11= TRUE ;
END ELSE
BEGIN K1=K1+1 A(K)= CH END ;
UNTIL BT11
CL1=21 TECH1=A(1) I VAL1=ASCII_CODE[TECH1] ;
IF * EQ 2 THEN
BEGIN
TECH1=A(2) TEMPI=ASCII_CODE[TECH1] ;
APPEND(IVAL,8,TEMPI) ;
END ELSE
BEGIN NO1=SYMBOL1 CL1= SYMBOL1 ; I VAL1= 0 ;
* TEST FOR NOW CHARACTER SYMBOL *
BEGIN NEXTCH
IF CH= THEN
BEGIN NEXTCH
IF CH= THEN
BEGIN NO1=22 NEXTCH END ;
END ELSE
IF CH= THEN
BEGIN NEXTCH
IF CH= THEN
BEGIN NO1=21 NEXTCH END
END ELSE
IF CH= THEN
BEGIN NEXTCH
IF CH= THEN SKIP COMMENT 

BEGIN NEXTCH

31 WHILE CHEX DO NEXTCH

END ELSE

END NEXTCH

END SIGNAL CHARACTER
PROCEDURE SRCHREC (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
END SRCHREC;

PROCEDURE SEARCH;
/*/ SEARCHES CONTEXTABLE, RETURNS CTPTR AND DISX=INDEX TO DISPLAY +

LABEL 1;
VAR II:INTEGER;
BEGIN
FOR II = TOP DOWNTO 0 DO
BEGIN CTPTR := DISPLAY[II].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 DISX := II;
END SEARCH;
PROCEDURE INCONST(VAR V:INTEGER; NXT:INTEGER);
VAR SIGN: BOOLEAN; PT INTEGER;
BEGIN
   SIGN := FALSE; PT := 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
         + : = INTPtr;
      END;
      IF SIGN THEN V := -IVAL ELSE V := +IVAL;
   END;
   END;
   ELSE IF NO = 1 THEN
      BEGIN PTR := CTPtr;
      BEGIN Search(Nxt); IF CTPtr = 0 THEN SEARCH;
      IF CTPtr = 0 THEN ERROR(12) ELSE WITH CONTEXTABLE(CTPtr) DO
         BEGIN IF (Klass#KONST) V (CONKind=FORMAL) THEN ERROR(42) ELSE
            BEGIN PTR := CONType; V := VALUES END;
         END;
      END;
      END ELSE ERROR(3);
   END;
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  
                    ELSE INSYMBOL  
                END *SKIP*
        END
    PROCEDURE FINDSEMICOLON;
        BEGIN
            IF NO NE 16 THEN
                BEGIN
                    ERROR(39)  
                    SKIP(16)  
                    IF NO NE 16 THEN WRITE(* E, EEXITIE)  
                    INSYMBOL  
                    END *FINDSemicolon*  
    END

PROCEDURE SEG1 ; INITIALIZE +
BEGIN
DIGITS := [\(E, E1, E2, E3, E4, E5, E6, E7, E8, E9\) ;
DP := TRUE ;
INITIALIZE CONEXITABLE +
WITH CONEXITABLE[0] DO TYPE OF NIL +
BEGIN II := 1 ;
REPEAT
NAME[II] := BLANK ;
II := II + 1 ;
UNTIL II = 10 ;
NXTEL := 0 ; KLA$$ := TYPES ;
SIZE := 0 ; FORM := NUMERIC ;
END ;
CPtr := 1 ;
WITH CONEXITABLE[CPtr] DO TYPE OF MEM +
BEGIN
NAME := BLANK ;
II := 1 ;
REPEAT
NAME[II] := BLANK ;
II := II + 1 ;
UNTIL II = 10 ;
NXTEL := 0 ; KLA$$ := TYPES ;
SIZE := 156B ; FORM := ARRAYS ;
LO := 0 ; HI := 177B ; OPTYP := PUREP ;
END ;
CHARP := 10 ;
WITH CONEXITABLE[CHARP] DO CHAR +
BEGIN II := 0 ;
REPEAT
NAME[II] := INITNAM[26, I] ;
UNTIL II = 10 ;
NXTEL := 5 ; KLA$$ := TYPES ;
SIZE := 1 ; FORM := SYMBOLIC ; BITSIZE := 7 ;
END ;
WITH CONEXITABLE[2] DO +
BEGIN II := 1 ;
REPEAT
NAME[II] := BLANK ; II := II + 1 ;
UNTIL II = 10 ;
VALUES := 63 ; SUCCE := 0 ;
END ;
CONEXITABLE[CHARP1].FCONST := 2 ;
BOOLPTR := 11 ;
WITH CONEXITABLE[BOOLPTR] DO BOOLEAN +
BEGIN
I1 = 0;
REPEAT
I1 = I1 + 1;
NAME[I1] = INITNAM[27, I1];
UNTIL I1 = EQ 10;
NXTIEL = CHARPTR; KLASS = TYPES; SIZE = 1;
FORM = SYMBOLIC; BTNSIZE = 2;
END;
WITH CONTEXTTABLE[3] DO *EOL*
BEGIN I4 = 1;
REPEAT NAME[I4] = INITNAM[20, I4];
I4 = I4 + 1;
UNTIL I4 = 101;
NXTIEL = BOOLPTR;
KLASS = KONST; CONTYPE = CHARPTR; CONKIND = ACTUAL;
VALUES = 0; SUCC = 0;
END;
CPTR = 0;
NEXT I = 27;
PT I = 4;
FOR IT = 0 TO 1 DO *FALSE, TRJE +
BEGIN
WITH CONTEXTTABLE[PT + IT] DO
BEGIN IT = 0;
REPEAT
IT = IT + 1;
NAME[IT] = INITNAM[IT + 29, IT];
UNTIL IT = EQ 10;
NXTIEL = NEXT I;
KLASS = KONST; CONTYPE = BOOLPTR;
CONKIND = ACTUAL; VALUES = IT; SUCC = 0;
END;
NEXT I = 4;
END;
CONTEXTTABLE[BOOLPTR], FCONST = 5;
INPTR I = 12;
WITH CONTEXTTABLE[INPTR] DO *INTEGER +
BEGIN IT = 0;
REPEAT
IT = IT + 1;
NAME[IT] = INITNAM[30, IT];
UNTIL IT = 10;
NXTIEL = BOOLPTR;
KLASS = TYPES;
SIZE = 1;
FORM = NUMERIC;
MIN = 1000000; MAX = 77777778;
BIT = WORDLENGTH +
END;
BYTPTR I = 13;
WITH CONTEXTTABLE[BYTPIR] DO *BYTE*
BEGIN
  I; = 0;
  REPEAT
    I; = I; + 1;
    NAME(I;) = INITNAM[37, I;]
  UNTIL I; = 15;
  NXTL = INTPR;
  KCLASS = TYPES; SIZE = 1;
  FORM = SYMBOLIC; BITSIZE = 4;
END;

WITH CONTEXTTABLE[WRDPRTR] DO *WORD*
BEGIN
  I; = 0;
  REPEAT
    I; = I; + 1;
    NAME(I;) = INITNAM[38, I;]
  UNTIL I; = 15;
  NXTL = BYTPR; KCLASS = TYPES;
  SIZE = 1; FORM = SYMBOLIC;
  BITSIZE = WORDLENGTH;
END;

WITH CONTEXTTABLE[UNDCLPTR] DO
BEGIN
  I; = 1;
  REPEAT
    NAME(I;) = BLANK;
    I; = I; + 1;
  UNTIL I; = 10;
  NXTL = 0;
  KCLASS = VARS; VTYPE = 0;
  VKiND = ACTUAL; VADDR = 0; VLEVEL = 0;
END;

WITH CONTEXTTABLE[PT] DO *MEM*
BEGIN
  I; = 1;
  REPEAT
    NAME(I;) = INITNAM[31, I;]
    I; = I; + 1;
  UNTIL I; = 10;
  NXTL = 0;
  KCLASS = VARS;
  VTYPE = CTPR; VKiND = ACTUAL;
  VADDR = 0; VLEVEL = 0;
END;

WITH CONTEXTTABLE[CIPTR] DO *TYPE OF INPUT AND OUTPUT*
BEGIN
  I; = 1;
  REPEAT
    NAME(I;) = BLANK;
\[ \text{I} = \text{I} + 1 \]
\[ \text{UNTIL I} = 10 \]
\[ \text{NXTEL} = 0 \]
\[ \text{KCLASS} = \text{TYPES} \]
\[ \text{SIZE} = 113 \]
\[ \text{FORM} = \text{FILES} \]
\[ \text{END} \]
\[ \text{PT} = 8 \]
\[ \text{FOR IT} = 22 \text{ TO } 23 \text{ DO} \]
\[ \text{BEGIN} \]
\[ \text{WITH CONTEXTTABLE(PT)} \text{ DO} \]
\[ \text{BEGIN I} = 1 \]
\[ \text{REPEAT} \]
\[ \text{NAME(I)} = \text{INITNAM(IT, I)} \]
\[ \text{UNTIL I} = 10 \]
\[ \text{NXTEL} = \text{CTPTR} \]
\[ \text{VKIND} = \text{ACTUAL} \]
\[ \text{VLEVEL} = 0 \]
\[ \text{END} \]
\[ \text{PT} = \text{PT} + 1 \]
\[ \text{END} \]
\[ \text{NEXT I} = 1 \]
\[ \text{PT} = 15 \]
\[ \text{FOR IT} = 0 \text{ TO } 11 \text{ DO} \]
\[ \text{BEGIN} \]
\[ \text{WITH CONTEXTTABLE(PT)} \text{ DO} \]
\[ \text{BEGIN I} = 0 \]
\[ \text{REPEAT} \]
\[ \text{NAME(I)} = \text{INITNAM(IT - 1, I)} \]
\[ \text{UNTIL I} = 10 \]
\[ \text{NXTEL} = \text{NEXT} \]
\[ \text{KCLASS} = \text{PROC} \]
\[ \text{PROCYPE} = \text{PT} \]
\[ \text{FORMALS} = 0 \]
\[ \text{PROCADR} = 0 \]
\[ \text{PROCLEVEL} = 0 \]
\[ \text{SEGSIZE} = \text{IT} \]
\[ \text{PROCIND} = \text{ACTUAL} \]
\[ \text{END} \]
\[ \text{NEXT I} = \text{PT} \]
\[ \text{PT} = \text{PT} + 1 \]
\[ \text{END} \]
\[ \text{END} \]
\[ \text{FOR IT} = 10 \text{ TO 5 \text{ DO}} \]
\[ \text{BEGIN} \]
\[ \text{I} = 0 \]
\[ \text{REPEAT} \]
\[ \text{XNAM(I)} = \text{INITNAM(IT, I)} \]
\[ \text{UNTIL I} = 10 \]
\[ \text{KSYM} = \text{IT} \]
\[ \text{END} \]
\[ \text{WITH DISPLAY[0]} \text{ DO} \]
BEGIN FILE NNAME = 14 ; OCCUR = BLOCK END ;
FOR IT I = 1 TO UNDMAX - 1 DO
  UNDLAB(I).SUCG I = IT + 1 ;
ENDLAB(UNDMAX).SUCG I = 0 ;
EXRINS I = 0 ; POS I = 0 ;
JMPINS I = 0 ; LEXTAB INS I = 0 ; CHNINS I = 1 ;
EXSY I = 5 ;
PREDEFP I = 27 ;
DISPLAY I I FNAME I = 0 ; DISPLAY I I OCCUR I = BLOCK ;
TOP I = 1 ; LEVEL I = 0 ; NEXT I = 0 ;
CHNNT I = 0 ;
LC I = 0 ;
IC I = 0 ;
VLCS I = 0 ;
NEXT I = 14 ;
P I = 27 ;
INDEX I = P ;
EOLF I = FALSE ;
CERR I = FALSE ;
PUNCHBIN I = FALSE ; LISING I = FALSE ; RLISTBIN I = FALSE ;
TRNSFR(I) I = 0 ; TRNSFR(I) I = 1 ; TRNSFR(I) I = 0 ;
INSBSYM ;
COMP (PROG, AVAIL, COMPR) ;
IF COMPAR THEN INSBSYM ELSE CERR I = TRUE ;
I I = 0 ;
REPEAT I I = I I + 1 ; PRNAM(I I) = AVAIL(I I) ;
UNTIL I I EQ 5 ;
INSBSYM ;
IF NO EQ 15 THEN
  REPEAT INSBSYM ;
  IF AVAIL I I EQ .BE THEN LISTBIN I = TRUE ELSE
    IF AVAIL I I EQ .BE THEN PUNCHBIN I = TRUE ELSE
      CERR I = TRUE ;
    INSBSYM ;
  UNTIL NO ME 15 ;
END IF
END IF
BEGIN
  LINES(I I) ; WRITE (ECONTROL CARD ERROR) ; LINES(I I) ;
END ;
KP I = 0 ;
LPM I = 0 ;
SUMPTR I = 0 ;
UNDERR I = 0 ;
HQA I = 0 ;
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[I].EXVAL = IVAL THEN
              BEGIN
                ERROR(48) ; GOTO 2 ; END
          IF CEXTABIX = MAXEXLABS THEN ERROR(49) ELSE
            BEGIN
              CEXTABIX := CEXTABIX + 1 ;
              IF JHPIX GT JHPMAX THEN ERROR(53) ELSE
                BEGIN
                  WITH EXTABLE[CEXTABIX] DO
                    BEGIN
                      EXVAL[I] = IVAL ; JHPBIX := JHPIX END
                  JHPBIX := JHPBIX + 1 ;
                END
          END
          INSYMBOL;
        END + IF (NO EQ 2) AND (CL EQ 1) THEN ELSE
          BEGIN ERROR(41) ; GOTO 3 END
        UNTIL NO NE 15 ;
    END SEMICOLON ;
  END + IF NO NE 43 THEN CONST +
  BEGIN
    INSYMBOL;
    WHILE NO EQ 1 DO
      BEGIN
        BEGIN
        REPEAT
          PROCSEC(NEXT) ;
          IF CIPTR 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 ;
          END
        END
      END
    END
  END
NXTEL = NEXT ; KLAS = KONST ;
CONSUM = ACTUAL ;
END 1
NXTEL = P ; INS YMBOL ;
IF (NO ME 6) OR (CL ME 6) THEN
ERROR 4 ELSE INS YMBOL
TEH 1 = CONTEXT [NEXT] . NXTEL ;
I (CON S I I.N . I E M ) ;
CONTEXT [P] . CONTY P = N ;
CONTEXT [P] . VALUES = 1 ;
WHILE NO EQ 15 DO
BEGIN INS YMBOL ;
IF NO ME 1 THEN
BEGIN ERROR 11 ; SKIP(15) END ;
END 1
UNTIL NO ME 1 ;
FINDSEMICALON ;
END WHILE NO = 1 ;
PROCEDURE SEG4; /* SYNONYM DECLARATION */

VAR
  SY, TY, SNXT, SHRINT;
  JJ : INTEGER;

BEGIN
  SY := P;
  TY := 1;
  SNXT := SY;
  INSYMBOL :=
  IF (NO NE 8) OR (CL NE 6) THEN ERROR(4);

  INSYMBOL :=
  IF NO NE 1 THEN BEGIN ERROR(11); GOTO 1 END;
  IF SYRCHREG(NEXT) THEN ERROR(8);
  INDEX := INDEX+1;
  P := INDEX;

  repeat
    INSYMBOL :=
    IF NO NE 1 THEN BEGIN ERROR(11); GOTO 1 END;
    RCHREG(NEXT) THEN ERROR(8);
    INDEX := INDEX+1;
    P := INDEX;
    WITH CONTEXTTABLE[P] DO
    BEGIN
      NEXT := NEXT;
      JJ := 0;
      repeat
        JJ := JJ+1;
        NAME[JJ] := AVAL[JJ];
        UNTIL JJ=10;
      end;
      INGELL := TRUE;
    end;
  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 NOTCONTEXTTABLE(CTPTR),SYNCELL) THEN
      ERROR(18) ELSE
      BEGIN
        P := CTPTR ;
        Q := CONTEXTTABLE[P].SYNPTR ;
        SYNECL ;
        GOTO 6
      END;
    END;
  END;
INDEX I := INDEX+1 ;
I := INDEX ;
WITH CONTEXTTABLE(I) DO
BEGIN
  SYNCELL := TRUE ;
  I := 0 ;
  REPEAT
    I := I+1 ;
    NAME[I] := AVAL[I] ;
  UNTIL I=10 ;
  NEXT I := NEXT ;
END ;
NEXT I := P ;
Q := 0 ;
SYNDECL ;
INSYMBOL ;
END ;
END ;
END ;
END ;#SEG4


PROCEDURE TYPEcola (VAR TL, P1: INTEGER);
  TO RETURN THE SIZE OF THE VARIABLE OF TYPE POINTED
  TO BY P1.

VAR ITL: INTEGER;
  G: INTEGER;
  E1, E2: EIDISPL:
  INTEGER;
  TEMP:
  INTEGER;
  CPTR:
  INTEGER;
  Q:
  INTEGER;
  OPT:
  OPT;
  H: INTEGER;
  ADD:
  ADDRESS;
  P:<;
  KV;
  BOOLEAN;
  L:
  INTEGER;
  PP;
  NXTF;
  NXTO;
  NXTA;
  RTYP:
  INTEGER;
  NXTIE:
  INTEGER;
PROCEDURE SKIPF(FN0#INTEGER) ;
BEGIN
WHILE (FERRCL[NO]=0) ^ (FN0=NO) DO

END *SKIPF + 
PROCEDURE TYPERR(I#INTEGER) ;
BEGIN
I0=0 ; P0=0 ;
error(I) ; SKIPT(49) ;
WRITE(E=ERROR(I) ; E=EOL) ; SKIPT(49) ;
END *TYPERR + 

PROCEDURE SUBRANGE(VAR VAL1, VAL2 #INTEGER; I1 #INTEGER; P #INTEGER) ;

*THE FIRST SYMBOL HAS BEEN READ*
THE PROCEDURE RETURNS TWO BOUND VALUES IN VAL1, VAL2*
AND RETURNS I1=POINTER TO TYPE OF CONSTANTS.
ERROR: TYPES DO NOT AGREE, TYPE IS NOT INTEGER, CHAR, OR
ERROR: SYMBOLIC, VAL1> VAL2.
I P INDICATES BEGINNING OF SEARCH FOR FIRST SYMBOL
IF IT IS AN ID.
VAR N21 #INTEGER; I #POINTER +
BEGIN inconst(VAL1, N1, P) ;
IF N1=0 THEN ERROR(TRUE ELSE
BEGIN IF N2=1 THEN ERROR(10) ELSE
insymbol ;
inconst(VAL2, N2, NEXT) ;
IF N2=1 THEN ERROR(TRUE ELSE
IF VAL1=VAL2 THEN ERROR(I1);
END I
END *SUBRANGE + |

PROCEDURE SUBTYPE( VAR J #INTEGER; P #INTEGER) ;
*EITHER A TYPE ID FOR A SUBRANGE OR AN EXPLICIT SUBRANGE
ARE PROCESSED.*
RETURNS J=LOWBOUND, J=HIGHBOUND,
P=POINTER TO TYPE OF CONSTANTS. +
VAR
ITEM #INTEGER
BEGIN
IF NO=1 THEN
BEGIN search(NEXT) ;
IF CTPTR =0 THEN SEARCH ;
IF CTPTL EQ 0 THEN ERROR(12) ELSE
BEGIN
IF CONTEXTTABLE[CTPTL].KLASS = TYPES THEN
BEGIN
IF CONTEXTTABLE[CTPTL].FORM > SYMBOLIC THEN
ERROR(13) ELSE
BEGIN
CASE CONTEXTTABLE[CTPTL].FORM OF
BEGIN
I1 = CONTEXTTABLE[CTPTL].MIN ;
J1 = CONTEXTTABLE[CTPTL].MAX ;
P1 = INITP1 ;
END ;
END = CASE+ ;
END =象征 ;
END = TYPES ELSE
BEGIN
IF CONTEXTTABLE[CTPTL].KLASS = KONST THEN
SUBRANGE(I1, J1, P1, CTPTL) ELSE ERROR(42) ;
END ;
END = ID ELSE
IF (P0=0) \ (P0=7) THEN
BEGIN
SUBRANGE(I1, J1, P1, CTPTL) ELSE ERROR(11)
END = SUBRANGE+ ;
******
PROCEDURE SCALDEG(N: INTEGER) ;
BEGIN
SUBRANGE(I1, J1, P1, N) ;
IF ERR THEN
BEGIN
ERROR(22) ;
P1 = #0
END ELSE
BEGIN
INDEX = INDEX + 1 ;
P1 = INDEX I1 ;
WITH CONTEXTTABLE[P1] DO
BEGIN
J1 = 1
END
END;
REPEAT
NAME[JJ] := BLANK ;
JJ := JJ + 1 ;
JNITL JJ=10

NXTL3 = 0; KLASS1 = TYPES;
SIZE 3 = 1; FORM J = NUMERIC;
MIN 4 = H; MAX 5 = J;
IF ABS(I) > ABS(J) THEN;
BIT 6 = LOG2 (ABS(J)); ELSE
BIT 6 = LOG2 (ABS(J));
END J = J + 1; P11 = P
END ELSE P11 = 0
END #SCALDECL + ;
PROCEDURE FIELDLIST(VAR MAXSIZE, VAR PTR, NXTF: INTEGER);
    VAR MAXL, ALL, MINSIZE, CASEBIN: INTEGER;
    PP, PIP, PP2, XCT, NXT, CPTR: INTEGER;
    FLAG: BOOLEAN;
    **********
    PROCEDURE REQUERY(I: INTEGER);
    BEGIN ERROR(I);
        SKIPT(53);  
    END;  REQUERY
    **********
    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 CONTEXTABLE(LASTFLU) 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=3 THEN CASE GOTO 2;
            IF NO=1 THEN
                BEGIN ERROR(11);
                    IF TERR(N)=1 THEN GOTO 11;
                    GOTO 12;
                END;
            SMHREQ(NXT) :=
                IF CPTR = 0 THEN ERROR(5) ELSE
                    BEGIN I:=I+1;
                        INDEX := INDEX + 1;
                    END;
                    WITH CONTEXTABLE(I) DO
                        BEGIN JJJ:=J;
                            REPEAT NAME[J]= AVAL[J];
                                JJJ=JJ+1;
                        END;
            END;
            IF PTR THEN
                IF NAME[J]=AVAL[J]
                    THEN BEGIN JJJ:=J;
                        REPEAT NAME[J]= AVAL[J];
                        JJJ=JJ+1;
                    END;
            END;
    END;

UNTIL JJ=10 ;
NXTEL = NXT ;
KLSASS = FIELD ;
FLTYPE = 0 ;
END ;
NX[1] = P ;
END ;
INSYM ;
IF NO = 15 THEN ;
BEGIN INSYM ; GOTO 1 ; END ;
IF NO = 21 THEN NOT ; ERROR(10) ;
ELSE INSYM ;
END ;

11 ;

TYPEDECL(LP) ;
IF P=0 THEN
IF CONTEXTABLE(P).FORM = RECORDS THEN
ERROR(24) ELSE
BEGIN
IF PACKFLAG THEN
BEGIN
IF I GT 1 REVERSE POINTERS THEN
BEGIN
PPI = NXT ;
FOR I = 1 DOWNTO 1 DO
BEGIN
PPI = CONTEXTABLE(P).NXTEL ;
CONTEXTABLE[PP1].NXTEL = PPI ;
PP2 = PPI ;
PP1 = PP1 + 1 ;
END ;
CONTEXTABLE(NXT).NXTEL = PP ;
NX = PP2 ;
END = REVERSE ELSE
PP1 = CONTEXTABLE(NXT).NXTEL ;
END = NEXT ;
IF CONTEXTABLE(P).FORM LE SYMBOLIC THEN
BEGIN
CASE CONTEXTABLE(P).FORM OF
NUMERIC ;
SYMBOLIC ;
END ;
REPEAT
IF BB = DISPL AND LL GT WORLENGHT THEN ADJUST ;
IF LL EQ WORLENGHT THEN
BEGIN
CONTEXTABLE[PP1].BITWIDTH = 0 ;
CONTEXTABLE[PP1].FLOADDB = DISPL ;
END ;
DISPL = DISPL + 1;
END ELSE
BEGIN
  CONTEXTABLE[PP1].BITWIDTH = LL;
  CONTEXTABLE[PP1].BITDISPL = BOISPL;
  CONTEXTABLE[PP1].FLOATYPE = P;
  LASTFLOD[PP1].TAGFLAG = FALSE;
  PP1 = CONTEXTABLE[PP1].NXTEL;
UNTIL PP1 = PP;
END *FORM = SYMBOIC+ ELSE
BEGIN
  IF BOISPL NE 0 THEN ADJUST;
  TAGFLAG = FALSE;
  REPEAT
    WITH CONTEXTABLE[PP1] DO
      BEGIN
        BITWIDTH = 0;
        FLOATYPE = P;
        FLOATYP = DISPL;
      END;
    BOXFLOD = DISPL + 1;
    LASTFLOD = PP1;
    PP1 = CONTEXTABLE[PP1].NXTEL;
  UNTIL PP1 = PP;
END *
PACKFLAG = ELSE
  LL1 = I + DISPL;
  DISPL = LL1;
  PP = NXT;
FOR I = 1 DOUNTO 1 DO
BEGIN
  CONTEXTABLE[PP1].FLOATYPE = P;
  LL = LL - 1;
  CONTEXTABLE[PP1].FLOATYP = LL;
  CONTEXTABLE[PP1].BITWIDTH = 0;
  PP = CONTEXTABLE[PP1].NXTEL;
END *
END *FORM = RECORDS *+
IF NO = 16 THEN INSYMBOIC;
UNTIL (ERRORCL[NO] = 2) AND (NO = 30);
IF BOISPL NE 0 THEN ADJUST;
MAXSIZE = DISPL; VARPTR = 0;
GOTO 9;
21 CASE

INSYMBOFL

IF N0>1 THEN ERROR(6) ELSE
BEGIN SRCHECINEXT

IF CPTR NE 0 THEN ERROR(5) ELSE
BEGIN
INDEX I = INDEX + 1;
3 = INDEX;
WITH CONTEXTABLE(P) DO
BEGIN
JJ1 = 1;
REPEAT
NAME(JJ1) = AVAL(JJ1);
JJ1 = JJ1 + 1;
UNTIL JJ1 = 10;
NEXTL = NXT; KLASS = FIELD;
FLOATYPE = 0;
END; NXT = P;
END;
INSYMBOFL
END AND NO>1 + 1;
IF NO NE 21 THEN ERROR(10) ELSE
INSYMBOFL
IF NO>1 THEN ERROR(11) ELSE
BEGIN SRCHECINEXT

IF CPTR = 0 THEN SEARCH;
IF CPTR = 0 THEN ERROR(12) ELSE
IF (CONTEXTABLE(CPTR).KLASS # TYPES)
 THEN ERROR(7);
END;
INSYMBOFL
IF NO > 31 THEN ERROR(14);
CPTR = CPTR + 1;
IF CPTR NE 0 THEN
IF CPTR > 0 THEN
IF PACKFLG THEN
BEGIN
CASE CONTEXTABLE(CPTR).FORM OF
LL = CONTEXTABLE(CPTR).BITS;
LL = CONTEXTABLE(CPTR).BITSIZE;
END;

IF BDISPL + LL > WORDLENGTH THEN ADJUST;
CONTEXTABLE(P).BITDISPL = BDISPL;
CONTEXTABLE(P).BITWIDTH = LL;
CONTEXTABLE(P).FLDADJ = DISPL;
BDISPL += BDISPL + LL;
LASTFLO += P; CASEBITS += BDISPL;
ENJ = PACKFLAG = ELSE
BEGIN
CONTEXTTABLE[(P].FLADDR = DISPL ;
CONTEXTTABLE[(P].BISWIDTH = 0 ;
END
CONTEXTTABLE[(P].FLTYPE = CPTTR ;
DISPL = DISPL + 1 ;
MINSIZE = DISPL ;
MAXSIZE = DISPL ;
NXTC = 0 ; INSsymbol ;
REPEAT I = 0 ;
REPEAT
IF (NU>21) = (NO=2) = (CL=1) THEN
BEGIN RECERR(63) ; GUTH 3 ENO ;
IF CPTTR#0 THEN
IF NU=1 THEN
BEGIN SREGREC(NEXT) ;
IF CPTTR=0 THEN SEARCH ;
IF CPTTR=0 THEN ERROR(12) ELSE
WITH CONTEXTTABLE(CPTTR) DO
IF KLASS*KONST THEN ERROR(42)
ELSE
IF (CONTEXTTABLE(CPTTR).FORM=SIMALIC) =
(CONTYPE=CPTR) OR
(CONTEXTTABLE(CPTTR).FORM=NUMERIC) =
(CONTYPE=INTCPR) THEN
ERROR(13) ELSE
END
IF NU=1 ELSE
IF (CL=1) = (CONTEXTTABLE[CPTTR].FORM=NUMERIC)
AND (I(L=2) = (CPTTR#CHARPTR) THEN
ERROR(46) ELSE
END
P = INDEX ;
WITH CONTEXTTABLE[(P) DO
BEGIN J(J1) = 1 ;
REPEAT
NAME[(JJ) = BLANK ;
JJ = JJ+1 ;
UNTIL JJ=1001
NXTL = NXTC ; KLASS = TAGFIELD ;
TAGVAL = TRUE ; CASEVAL = IT ;
END
NXTC = P ; I = I+1 ;
INSymbol ;
IF NO = 21 THEN
    ERROR(10) ELSE INSYMBOL;
UNTIL NO > 2:
    IF NO = 9 THEN FIELDLIST
    BEGIN
        IF PACKFLAG THEN
            BEGIN DISPL I = MINSIZE - 1;
                BDISPL I = CASEBITS;
            END ELSE DISPL I = 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 = 1 DOWNTO 1 DO
    WITH CONTEXTABLE(P) DO
    BEGIN CASESIZE I = MXL; VARIANTS I = PP;
        P = NXTEL
    END; IF MAXSIZE < MXL THEN
    MAXSIZE I = MXL;
    IF NO = 16 THEN INSYMBOL;
UNTIL NO > 2:
INDEX I = INDEX + 1;
P = INDEX;
WITH CONTEXTABLE(P) DO
BEGIN JI = 1:
    REPEAT
        NAME(JJ) = BLANK;
        JJ = JJ + 1;
    UNTIL JJ = 10;
    NXTEL I = 0; KLAS = TAGFIELD;
    CASESIZE I = MINSIZE; VARIANTS I = NXTC;
    Val I = FALSE; CASETYPE I = CPTR;
END;
VARPTR I = P;
END;
END = NO = 1;
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=NULL THEN SEARCH ;
IF CTPTR = 0 THEN
BEGIN ERROR(12) ;
P11=0 ; SKIPI(16) ;
END ELSE
BEGIN IF CONTEXTABLE (CTPTR).KLAAS=TYPES THEN
* TYPE-ID +
BEGIN TVL=CONTEXTABLE (CTPTR).SIZE ;
P11=CTPTR ; INSYMBOL
END ELSE
IF CONTEXTABLE (CTPTR).KLAAS = KONST THEN
SCALDEC (CTPTR) ELSE
TYPEERR(111) ;
END
END * ID + ELSE
IF NO=9 THEN * SYMBOLES +
BEGIN CVL=0 ; LERR= ERR; ERR:=FALSE ;
INDEX := INDEX + 1 ;
P := INDEX + 1 ;
WITH CONTEXTABLE (P) DO
BEGIN JJ=1 ;
REPEAT
NAME [JJ] := BLANK ;
JJ := JJ+1 ;
UNTIL JJ=10 ;
NEXTL = 0 ; KLAAS := TYPES ;
FORM := SYMBOLES ;
END ; RYP := P ; NXTC := 0 ;
REPEAT INSYMBOL
IF NO.1 THEN
BEGIN ERROR(11) ;
SKIPI(15) ; GOTO 2 ;
END ;
SRCHREC (NEXT) ;
IF CTPTR = 0 THEN ERROR(8) ELSE
BEGIN
INDEX := INDEX + 1 ;
P := INDEX + 1 ;
WITH CONTEXTABLE [ P] DO
BEGIN J1=1
REPEAT
   NAME(JJ) = AVAL(JJ);
   JJ= JJ+1;
UNTIL JJ=10;
NXTL1= NEXT 1 KLAAS= KONST;
CONTYPE = RYP; CONKIND = ACTUAL;
VALUE = CV; SUC(1) = NXTC; END;
CV=CV+1; NEXT = P; NXTC = P;
END;

2 1 INSymb 1
UNTIL NO=15;
WITH CONTEXTABLE[RTYP] DO
BEGIN FC0NS= NEXT; SIZE= 1;
BITSIZE = 2;
END;
IF ERR THEN P1= 0 ELSE
BEGIN ERR= LERR; P1= RYP; END;
TL= 1;
IF NO= 10 THEN TYPERR(9) ELSE INSymb;
END;
SYMBOLIC 1 ELSE
IF (NO=2) OR (NO=7) THEN SUBRANGE +
BEGIN LERR= ERR; ERR= FALSE;
SCALDEC(NEXT);
END;
IF ERR THEN ERROR(6) ELSE ERR= LERR
END * SUBRANGE ELSE
IF NO=41 THEN *STRUCTURED TYPES +
CASE CL OF
    ARRAY + 1 I
BEGIN INSymb;
IF NO=11 THEN
BEGIN TYPERR(57) 1
IF TERR= 1 THEN TYPDECL(I, CTPTR); GO TO 19;
END;
SUBL= 0;
REPEAT
INDEX = INDEX + 1;
P1= INDEX;
WITH CONTEXTABLE[P] DO
BEGIN JJ=1;
   REPEAT NAME(JJ) = BLANK;
   JJ= JJ+1;
UNTIL JJ=10;
END;
NXTL1= 0 * KLAAS= TYPES;
FORM1= ARRAYS 1, AETYPE I= NXTA 1
  * AETYPE TEMP. LINKS SJBARRAYS
END 1 NRTA 1= P 1
INSYMBOL 1
LERR1= ERR1 1, ERR1= FALSE 1
SUBTYPE(I, J, P) 1
IF ERR THEN
   BEGIN 1 ERROR(16) 1
      SKP(15) 1 II=0 1 JJ=0 1 PII=0 1
   END ELSE
   ERR1= LERR1
WITH CONTEXTTABLE[NXTA] DO
BEGIN
IF I LT 127 THEN 1 LO1=I ELSE
   BEGIN
      LO1=I 1 DIV 10 1 1
      SZE1=TRUE 1
   END
IF J LT 127 THEN 1 HI1=J ELSE
BEGIN
   HI1=1 1 DIV 10 1 1
   SZE1=TRUE 1
END 1
INXTYPE1= P 1
END
UNTIL NO1=15 1
IF NO1=12 THEN
BEGIN 1 ERROR(31) 1
   SKP(31) 1
   IF TERRCL(NO1)=0 THEN 1 GOTO 11 1
   IF NO1=31 THEN
      BEGIN 1 INSYMBOL 1
      GOTO 11 1 END
   IF NO1=12 THEN
      BEGIN 1 PI1=0 1 TL1=0 1 GOTO 19 1 END
   END
   INSYMBOL 1
   IF NO1=31 THEN 1 ERROR(14) ELSE 1 INSYMBOL 1
   TYPECL(TK, CTPT1 1)
   IF CTPT1=0 THEN
      IF CONTEXTTABLE[CTPT1].FORM > REGISTERS
      THEN BEGIN 1 ERROR(24) 1 CTPT1=0 1 END
   ELSE 1 BEGIN
   CTPT1 = CTPT1 1
   REPEAT
      WITH CONTEXTTABLE[NXTA] DO

BEGIN
MULOPT(TL,E1,E2,OPT) ;
OPTYPE 1 = OPT ; EXP1 = E1 ;
EXP2 = E2 ;
TL = TL + (HI-LO+1) ;
SIZE = TL + P ;
AETYPE = CELPTR ;
END ;
CEPTR = NXTA ; NXTA = P ;
UNTIL NXTA = 0 ;
* NOW TL IS THE SIZE OF THE ARRAY, CEPTR.
POINTS TO IT +

END ;
PI = CEPTR ;
IF PACKFLAG THEN
BEGIN
CEPTR = CONTEXTTABLE[CEPTR].AETYPE ;
CASE CONTEXTTABLE[CEPTR].FORM OF
TEMP = CONTEXTTABLE[CEPTR].BITSIZE ;
BEGIN
TEMP = WORDLENGTH DIV TEMP ;
WITH CONTEXTTABLE[PI] DO
BEGIN
IF (SIZE MOD TEMP) EQ 0 THEN SIZE = SIZE DIV TEMP
ELSE SIZE = SIZE DIV TEMP + 1
END ;
END ;
END ;
END ARRAY ;

BEGIN
INDEX = INDEX + 1 ;
P = INDEX ;
WITH CONTEXTTABLE[PI] DO
BEGIN
JJI = 1 ;
REPEAT
NAME[JJI] = BLANK ;
JJI = JJI + 1 ;
UNTIL JJI = 10 ;
NXFIEL = 0 ; KLAS = TYPES ;
FORM = RECORDS ;
END ;
RIPTYP = PI ;
INSYM = 0 ;
DISPL = 01 ; DISPLX = 01 ; LERR = 11 ;
ERR = FALSE ; FIELDTST(TL,P,NXF) ;
IF NO = 26 THEN ERROR(15) ;
PROCEDURE SEG3 ;  * VARIABLE DECLARATION *
VAR
AT : ADDRESS ;
I, J, K, L : SHRINT ;
JJ : INTEGER ;
AS : ARRAY [1..10] OF INTEGER ;
BEGIN
INSymbol
WHILE NO = 1 DO
BEGIN JJ = 0
REPEAT
SEARCH (NEXT) ;
IF CPT  0 THEN
BEGIN
IF CONTEXTABLE [CPT].SYNCELL THEN
BEGIN
J = J + 1
AS [J] = CPT ;
P = CTRIQ ;
END
ELSE ERROR (8)
END
INDEX = INDEX + 1
P = INDEX ;
WITH CONTEXTABLE [P] DO
BEGIN
JJ = 0
REPEAT
JJ = JJ + 1
NAME [JJ] = AVL [JJ] ;
UNTIL JJ = 10
NEXT ;
KLAB = VARS
VIND = ACTUAL
TYPE = 0
LEVEL = 0
END
NEXT = P
JJ = JJ + 1
END
INSymbol
IF NO = 10 THEN
BEGIN
INSymbol
END
IF NO = 0 THEN
BEGIN ERROR (11) ; GOTO 10 END ;
END
END ELSE
IF NJ NE 21 THEN ERROR(10) ;
UNTIL NO NE 1 ;
IF NO EQ 1 THEN INSYM_EOL
ELSE IF NOT(ERR) THEN ERROR(10) ;
N IS = 1
ERR IS = FALSE ;
TYPEDECL(TL, CTPLTR) ;
IF ERR THEN GOTO 10 ;
LC IS = LC + (I + J) * TL ;
LL IS = LC ;
FOR I IS = 1 DOWNTO 1 DO
BEGIN
LL IS = LL - TL ;
WITH CONTEXTTABLE[N] DO BEGIN
VTYP IS = CTPLTR ;
3LEVEL IS = LEVEL ; VADDR IS = LL ;
N IS = NXTEL IS ;
END ; WITH N +
END * FOR I * ;
END * FOR J * ;
BEGIN
N IS = AS(j) ; LL IS = LL - TL ;
WITH CONTEXTTABLE[N] DO BEGIN
IF SYNPTR EQ 0 THEN GOTO 2
ELSE BEGIN
KCLASS IS = VARS ; VKIND IS = ACTUAL ;
VTYP IS = CTPLTR ; VLEVEL IS = 0 ;
LEVEL IS = LEVEL ; VADDR IS = LL
END * SYNPTR NE 0 * ;
N IS = SYNPTR ; SYNPTR IS = 0 ;
END * WITH N * ; GOTO 1.1
21 J IS = J + 1
END * FOR J * ;
END * FOR I * ;
10 8 FINDSENCOLON ;
END * WHILE, NO=1 * ;
END 1 SEG3+
PROCEDURE SEGS  /* VALUE DECLARATION */
VAR II: INTEGER;
BEGIN
  IF NO EQ 40 THEN /* VALUE */
  BEGIN
    VALPT[1] = TRUE;
    II := 0;
    IF LEVEL NE 0 THEN ERROR(18);
    INSYM;
    WHILE NO EQ 1 DO
      BEGIN
        SRCREC(NEXT);
        IF CTPTR EQ 0 THEN SEARCH;
      END;
      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
        IF U = VADR THEN
          IF VALPT THEN
          BEGIN
            IF VLC EQ 0 THEN
              BEGIN
                VLC := IT;
                LLI := IT;
                AT := IT;
              END;
            END;
            IF IT LI LL THEN ERROR(20) ELSE
            WHILE IT LI AT DO
            BEGIN
              IT := IT + 1;
            END;
          END;
          ELSE INSYM;
          IF NO EQ 9 THEN /* LIST */
          BEGIN
            REPEAT
            BEGIN
              INSYM;
              INCONST(I, PT, NEXT); IT := 1;
            END;
            IF NO EQ 61 AND (CL EQ 1) THEN /* $$ */
            BEGIN
            END;
          END;
        END;
      END;
    END;
  END;
END.
IT1 = I ; INSYMBOI
I.NCONSt(I, PT, NEXT) 
END ;

IF NOT(ERR) THEN
BEGIN
FOR J1 = 1 TO IT1 DO
BEGIN IT1 = IT1 + 1 ; VARB[III] = I END ;
AT1 = AT + IT1 ;
END ;
ERR1 = ;
IF (NO NE 15) AND (NO NE 10) THEN
* NOT, NOR) =
BEGIN
ERROR(20) =
FINOSEMICOLON ;
GOTO 20
END ;
UNTIL NO = 10 ;
INSYMBOI ;
END ELSE
BEGIN
I.NCONSt(I, PT, NEXT) ;
IF NOT(ERR) THEN
BEGIN
IT1 = IT1 + 1 ; VARB[III] = I ;
AT1 = AT + 1 ;
END ;
END =
IT = CONTEXTABLE[TYPE] SIZE ;
FINOSEMICOLON ;
END # WITH ;
VALPT = FALSE ;
20 :
VALPT = TRUE;
END # VALUE; ELSE VLC = LC ;
END # SEGS ;


PROCEDURE GENRE(A, B: INTEGER; C: R; TYP: 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;
      ADDR := B;
      TYP := C;
      MREF := D;
    END
END.
END - GENRE -
PROCEDURE RLEXP 1
* GENERATES CODE FOR RELATIONAL OPERATORS *
BEGIN
CASE LAOPOCL OF
41 LT BEGIN
GENRE(2021B,0,0, FALSE) 
GENRE(2003B,0,0, FALSE) 
GATTR.JMP I = 0
END 
21 LE BEGIN
GENRE(2021B,0,0, FALSE) 
GATTR.JMP I = 1 END 
31 GE BEGIN
GENRE(2022B,0,0, FALSE) 
GATTR.JMP I = 1 END 
41 NE BEGIN
GENRE(2031D,0,0, FALSE) 
GATTR.JMP I = 0 END 
61 EQ BEGIN
GENRE(2002B,0,0, FALSE) 
GATTR.JMP I = 1 END 
END + CASE +
GENRE(2048B,0,0, FALSE) 
END + RLEXP +
PROCEDURE L0TMP;
VAR RLX : BOOLEAN;
DLX : BOOLEAN;
BEGIN
IF R&B EQ 0 THEN GOTO 1;
STY := FALSE;
RLX := FALSE;
DLX := FALSE;
WITH STRRIP1 DO
BEGIN ORP := STOP; LAD0PCL := STCL; END;
IF OP EQ 7 THEN
BEGIN
CASE LAD0PCL OF
  11 GENRE(0420000,0,1,TRUE); /*ADA+
END;
BEGIN
  GENRE(020000,0,0,FALSE); /*CHA,INA+
  GENRE(0420000,0,1,TRUE); /*ADA+
END;
  GENRE(0320000,0,1,TRUE); /*IOR+
END = CASE CL OF ++
END = NO EQ 7 + ELSE
IF OP EQ 6 THEN
BEGIN
CASE LAD0PCL OF
BEGIN
  GENRE(0020000,0,0,FALSE); /*MPY+
  GENRE(0,0,1,FALSE); /*LOCATION FOR MPY+
END;
BEGIN
  GENRE(0120000,0,1,TRUE) /*AND+ END;
BEGIN
  GENRE(020000,0,1,TRUE); /*STA+
  GENRE(020000,0,1,TRUE); /*LODA+
  DLX := TRUE;
END;
END = CASE CL OF ++
END = NO EQ 6 + ELSE
IF OP EQ 8 THEN
BEGIN
  GENRE(0030000,0,0,FALSE); /*CHA,INA+
  GENRE(0420000,0,1,TRUE); /*ADA+
  RLX := TRUE;
END ELSE ERROR(54);
KP := KP+1;
IF KP GT KP0MAX THEN BEGIN ERROR(48) END;
WITH STORE[KP] DO
BEGIN STRP:=RP; STPL:= HCA END;
IF RLX THEN
BEGIN
RLX:=FALSE; GENC[3004B,0,0,FALSE]; *CHA,INA*
END;
IF DLX THEN
BEGIN
DLX:=FALSE;
GENC[6400B,0,0,FALSE]; *CLB*
GENC[2020B,0,0,FALSE]; *SSA*
GENC[7004B,0,0,FALSE]; *CMB,INB*
GENC[100006B,0,1,FALSE]; *DIV*
END;
RP:=RP-1;
END * LDTMP*;
************  ************  ************  ************
PROCEDURE STTMP;
BEGIN
CLV:=true;
KP:=RP+1;
IF RP # RMAX THEN RMPAX:=RP;
KP:=KP+1;
IF KP < RMAX THEN BEGIN ERROR(24); GOTO 1 END;
GENC[072008B,0,1,TRUE]; * STA *
WITH STORE[KP] DO
BEGIN STRP:=RP; STPL:= HCA END;
WITH SIG[RP] DO
BEGIN STOP:=OP; STCL:= LADOPCL END;
END * STTMP*;
* CALLED AT PROCEDURE END, UPDATING ITS CODE (INSERTING ADDRESSES OF THE CONSTANTS) AND WRITING OUT CODE TOGETHER WITH THE CONSTANTS USED IN THIS PROCEDURE *

**PROCEDURE WRITEM **
VAR FLG# INTEGER ;
BEGIN
  FLG := IC ;
  FOR I := 1 TO RMAX DO IC := IC+1 ;
  FOR I := 1 TO KP DO
    WITH STORC[I] DO HPAS(STPL).ADDR := FLG+STRP;
    HPAS[3].ADDR := IC+1 ;
    IC := IC+1 ;
  STORAGE FOR BUFFER
  IC := IC+1 ;
END * WRITEM * ;
**PROCEDURE WRITOUT **
VAR IT2 : SHRINT ;
BEGIN
  FOR IT := 1 TO LCX DO
    WITH CSITB[IT] DO
      BEGIN
        GENRELVALU[0,0,0,FALSE] ; DEC
        IT := INX ;
      REPEAT
      WITH UNDOLAB[IT] DO
        BEGIN
          WITH HPAS[PLACE] DO ADDR := IC ;
          IT := IT+1 ;
        UNTIL IT EQ 0 ;
        UNDOLAB[IT] := SUCU INX;
      END ; WITH CSITB ;
    END * WRITOUT * ;
**PROCEDURE WRITEM **
VAR FLG# INTEGER ;
BEGIN
  FLG := IC ;
  FOR I := 1 TO RMAX DO IC := IC+1 ;
  FOR I := 1 TO KP DO
    WITH STORC[I] DO HPAS(STPL).ADDR := FLG+STRP;
    HPAS[3].ADDR := IC+1 ;
    IC := IC+1 ;
  STORAGE FOR BUFFER
  IC := IC+1 ;
END * WRITEM * ;
**PROCEDURE WRITOUT **
VAR IT2 : SHRINT ;
BEGIN
  FOR IT := 1 TO LCX DO
    WITH CSITB[IT] DO
      BEGIN
        GENRELVALU[0,0,0,FALSE] ; DEC
        IT := INX ;
      REPEAT
      WITH UNDOLAB[IT] DO
        BEGIN
          WITH HPAS[PLACE] DO ADDR := IC ;
          IT := IT+1 ;
        UNTIL IT EQ 0 ;
        UNDOLAB[IT] := SUCU INX;
      END ; WITH CSITB ;
    END * WRITOUT * ;
PROCEDURE LOCST(FVAL: INTEGER):

ENTER CONSTANTS WITH VALUE FVAL INTO CONSTANT TABLE CSTTB

IF FVAL NOT YET PRESENT, ELSE CHAIN OCCURRENCE OF FVAL IN CODE

THROUGH UNLABEL:

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 UNLABEL(CHNIX) DO

BEGIN

IT1 = SUCC; SUCC = INX; INX = CHNIX; PLACE = HCA

END

END

CHNIX := IT1

END;

GOTO 10

END;

IF (LCX EQ CSTMAX) 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 UNLABEL(CHNIX) DO

BEGIN IT1 := SUCC; SUCC := 0; PLACE := HCA END;

CHNIX := IT1

END; LOCST;

100 END; LOCST;
PROCEDURE ADDRESSVAR(FCTP:INTEGER; VAR FATTRIAATTR); 
*BUILD UP ATTRIBUTE IN FATTR AND VARIABLE IS POINTED TO BY FCTP*
BEGIN
  WITH CONTEXTTABLE(FCTP), FATTR DO
  BEGIN
    KIND := VARB1;
    IF KCLASS = VARS THEN
      IF TYPE = VTYPE; PCKO := FALSE;
      IF KIND = ACTUAL THEN
        ACCESS := ORCT;
        DPLMT := VAADDR + BLC + BFLC + 1;
        BREG := VLEVEL;
      END ELSE
        WRITE(E =, EFORMALE);
      END ELSE
        IF KCLASS = FIELD THEN
          IF TYPE = FTYPE;
            WITH DISPLAY(DISX) DO
              BEGIN
                IF OCCUR=WITH THEN
                  BEGIN
                    ACCESS := ORCT;
                    BREG := CLEVEL;
                    DPLMT := FLOADDR + CDISP;
                  END ELSE
                    WRITE(E =, EFORMALE);
                  END IF BITWIDTH NE 0 THEN
                    BEGIN
                      PCKO := TRUE;
                      BITADR := BITDISPL;
                      BSIZE := BITWIDTH;
                      PCKO := FALSE;
                      END *DISPLAY*;
                  END *FIELD ELSE
                    BEGIN *PROCEDURE*;
                      IF PROC=TYPE THEN
                        ACCESS := ORCT;
                        BREG := PROCLEVEL + 1;
                        DPLMT := 2;
                        PCKO := FALSE;
                      ELSE IF PROC=FORMAL THEN ERROR(62);
                      END IF LEVEL NE BREG THEN ERROR(52);
                    END
                  END *WITH FCTP, FATTR*;
      END *ADDRESSVAR*;
PROCEDURE VARIABLE 
VAR LATIR, LATTR, TEMP, TEMP1, INTEGER;
BEGIN
ADDRESSVAR(CIPTR, LATIR); 
INSYMBOL;
IF NC EQ 1
BEGIN
INSYMBOL;
IF NC EQ 1 THEN +ID+
BEGIN
IF LATIR.TYPTR NE 0 THEN
BEGIN
TEMP := LATIR.TYPTR;
IF CONTEXTTABLE[TEMP].FORM EQ RECORDS THEN
WITH LATIR DO
BEGIN
TEMP := CONTEXTTABLE[TYPTR].FSTFLOD;
SRCHRCD(TEMP);
IF CIPTNR EQ 0 THEN
BEGIN ERROR(27); CIPTR := UNDECPTR END;
WITH CONTEXTTABLE[CIPTR] DO
BEGIN
TYPTR := FLOATY;
OPLMT := OPLMT+FLOADDR;
IF BITWIDTH NE 0 THEN
BEGIN
PCKO := TRUE; BITADR := BITDISPL;
BITSZ := BITWIDTH;
END ELSE PCKO := FALSE
END
END WITH LATIR+ ELSE
BEGIN
ERROR(26); LATIR.TYPTR := 0
END
END + TYPTR NE 0 + ELSE
BEGIN ERROR(26); LATIR.TYPTR := 0 END;
END
END + NO=1 +
END + IF NO=17 + ELSE
IF NC EQ 11 "(** THEN
BEGIN
GENRE(64008, 0, 0, FALSE) + CLB +
GENRE(760008, 8FLEC+4, 1, TRUE) + STB IN 8FLEC+4 +
REPEAL
WITH LATIR DO
IF TYPTR NE 0 THEN
  IF CONTEXTTABLE(TYPTR).FORM NE ARRAYS THEN
    BEGIN ERROR(63); TYPTR = 0 END;
  INSymbol;
  IF GIV THEN EXPRESSION ELSE BEGIN STHM;
    EXPRESSION END;
  IF GATR.TYPTR NE 0 THEN
    BEGIN
      TEMP1 = GATR.TYPTR ; TEMP11 = LATR.TYPTR
      IF CONTEXTTABLE(TEMP1).FORM GI SYMBOLIC THEN ERROR(64);
      IF TEMP1 NE 0 THEN
        BEGIN
          TEMP1 = CONTEXTTABLE(TEMP11).ITYPETYPE;
          IF ((CONTEXTTABLE(TEMP1).FORM EQ SYMBOLIC) OR
              (CONTEXTTABLE(TEMP1).FORM EQ SYMBOLIC))
            AND (TEMP1 NE PTI) THEN ERROR(31) ;
          WITH GATR DO
            IF KIND EQ SVAL THEN
              BEGIN
                IF (CONTEXTTABLE(TEMP11).LO GT VAL)
                  OR (CONTEXTTABLE(TEMP11).HI LE VAL)
                THEN ERROR(157) ;
              END WITH LATR, CONTEXTTABLE(TYPTR) DO
              BEGIN
                TEMP1 = CONTEXTTABLE(AELTYPE).SIZE;
                GENRE(100200B, 0, 0, FALSE) 1 = MPY
                GENRE(0, 0, 0, FALSE) 1 = BY SIZE;
                LOCST(TEMP1);
                DPLMT = DPLMT - LO * TEMP + 100000B 1
                GENRE(420000B, 0, 1, TRUE) 1 = ADA DISPLACEMENT
                LOCST(DPLMT) 1
                GENRE(420000B, BFLC+4, 0, TRUE) 1 = ADA TO BFLC+4
                GENRE(720000B, BFLC+4, 0, TRUE) 1 = STA IN BFLC+4
                IF ACCESS EQ DRC THEN ACCESS = INX0 ;
              END WITH LATR, TYPTR NE 0 ;
    END WITH LATR, TYPTR NE 0 ;
  IF LATR.TYPTR NE 0 THEN
    LATR.TYPTR = CONTEXTTABLE(TEMP11).AELTYPE
    UNTIL NO NE 151 ;
  IF NO NE 17 THEN =) ERROR(25) ELSE INSymbol ;
  END IF NO, EQ 11 + 1
  CATTR, LATR
  END VARIABLE

PROCEDURE PRIMARY;
BEGIN
    IF NO EQ 1 THEN
    BEGIN
        SCHREC(NEXT); 
        IF CTPTR EQ 0 THEN
        BEGIN ERROR(134); CTPTR = UNDECPTR END;
        CASE CONTEXTABLE[CTPTR].KLAASS OF
        BEGIN ERROR(45); 
        GATTR.TYPTR = 0; INSYMBO;
        END;
        KONST; 
        WITH GATTR, CONTEXTABLE[CTPTR] DO
        BEGIN
            TYPTR = CONTYPE;
            IF CONKIND = ACTUAL THEN
            BEGIN
                KIND = SVAL; VAL = VALUES  
                END ELSE
            BEGIN
                KIND = VARBL; ACCESS = DRECT; BREG = CLEVEL;
                DATA = CADDR; PCKD = FALSE
            END;
        END;
        INSYMBO;
    END;
    PROCI;
    VARS, FIELD; VARIABLE;
    END *CASE KLAASS OF*;
    END *IF NO*1 ELSE
    IF NO EQ 2 THEN
    BEGIN
    WITH GATTR DO
    BEGIN
      KIND = SVAL; CTY = TRUE; VAL = IVAL ;
      CASE CL OF
      TYPTR = ITPTR;
      TYPTR = CHARPTR;
      END *CASE*  
    END *
    GATTR*  
    INSYMBO ;
    END *IF NO*2 ELSE
    IF NO EQ 9 THEN
    BEGIN
    WHILE NO EQ 9 DO INSYMBO  
    IF CTY THEN EXPRESSION ELSE
    BEGIN SETHE; EXPRESSION END ;
    IF NO NE 10 THEN ERROR(7) ELSE
    END;
BEGIN
  LOTMP i
  IF RP EQ 0 THEN BEGIN WHILE NO EQ 10 DO INSymbOL END
  ELSE INSymbOL ;
END i
END IF NO=9+ ELSE
BEGIN ERROR(29); GATTR.TYPTRI=0 END
END *PRIMARi*
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
ELSE IF A EQ 1 THEN GENRE(C,0,0,FALSE) ;
END "ASSMB";
BEGIN
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";
END
```
PROCEDURE SYSOP;
VAR TEMP, TEMP1, I, J, III: INTEGER;
BEGIN
  NODE = NO1; LAADOCPL = CL1 INSYMBOL;
  IF (LAADOCPL EQ 11) OR (LAADOCPL EQ 12) THEN
  BEGIN
    RAIR, TPTR = GAIR, TPTR;
    IF NO NE 1 THEN ERROR(20) ELSE
    BEGIN
      PRIMARY;
      IF (RAIR, TPTR NE 1) AND (GAIR, TPTR NE 0) THEN
      BEGIN
        IF RAIR, TPTR EQ GAIR, TPTR THEN
        BEGIN
          WITH GAIR DO
          BEGIN
            CASE KIND OF
            BEGIN
              IF LAADOCPL EQ 11 THEN
              BEGIN
                GENRE(132000, 0, 1, TRUE) * IOR* ELSE
                GENRE(220000, 0, 1, TRUE) * XOR*
              END;
            END;
          END;
        END;
      END;
    END;
  END;
  END;
BEGIN
  IF LAADOCPL EQ 11 THEN
  BEGIN
    GENRE(132000, 0, 1, TRUE) * IOR* ELSE
    GENRE(220000, 0, 1, TRUE) * XOR*
  END;
  END;
BEGIN
  IF LAADOCPL EQ 9 OR LAADOCPL EQ 10 THEN
  BEGIN
    NODE(IVAL, 6, TEMP, TEMP1)
    IF LAADOCPL EQ 9 THEN
    BEGIN
      ASSEMBLY(17090, 17000) ELSE
      ASSEMBLY(57090, 57000)
    END;
    IF CL = 9 OR CL = 10 THEN
    BEGIN
      NODE(IVAL, 16, TEMP, TEMP1)
    END;
  END;
END.
IF (LADOPCL EQ 7) OR (LADOPCL EQ 8) THEN
BEGIN
  MODE(TMP,4,I,J);
  IF LADOPCL EQ 7 THEN
  BEGIN
    ASSMB(J,17278,17008)  + ALE+;
    ASSMB(I,57278,57008)  + RAL+;
  END ELSE + CL=3+
  BEGIN
    GENRE(330008,0,0,FALSE)  + LDB 0+
    ASSMB(J,57278,57008)  + BLF+
    ASSMB(I,52228,52008)  + RBL+
  END I
  ELSE + CL=7 OR CL=8+
  BEGIN
    MODE(TMP,2,I,J);
    FOR III=1 TO J DO
    BEGIN
      CASE LADOPCL OF
        10208,0,0,FALSE: + ALS,ALS+
        BEGIN
          GENRE(13008,0,0,FALSE)  + LDB 0+
          GENRE(50208,0,0,FALSE)  + BLS,BLS+
        END;
        11210,0,0,FALSE: + ARS,ARS+
        BEGIN
          GENRE(33008,0,0,FALSE)  + LDB 0+
          GENRE(51210,0,0,FALSE)  + BRS,BRS+
        END;
        12230,0,0,FALSE: + RAR,RAR+
        BEGIN
          GENRE(33008,0,0,FALSE)  + RBR,RBR+
          GENRE(53230,0,0,FALSE)  + RBR,RBR+
        END;
      END CASE +
      END FOR +
    END + CL=1,
    END + NO EQ 2 AND NO EQ 1 +
    END + SHIFT OPERATOR +
END + SYSOP +
PROCEDURE EXPRESSION;
VAR LFG BOOLEAN;
311, 612 I BOOLEAN;
BEGIN
LFG := FALSE;
IF NOT EQ 7 THEN ADDOP1
BEGIN
OP := NOT;
IF CL EQ 2 THEN LFG := TRUE
ELSE IF CL EQ 3 THEN ERROR(33);
INSYM1;
END;
IF NOT EQ 6 THEN NOT1
BEGIN
OP := NOT;
IF CL EQ 1 THEN LFG := TRUE;
ELSE ERROR(33);
INSYM1;
END;
END1;
PRIM1;
IF C1Y THEN
BEGIN
GAITR = TYPR NE 0 THEN
BEGIN
CIV1 := FALSE;
WITH GAITR DO
BEGIN
CASE KIND-OF
SVALL.
BEGIN
GENRE(62000B, 0, 1, TRUE); LDA1;
LJ12ST(SVALL1); CIV1 := FALSE;
END;
END;
END;
END1;
END; CASE;
ENDD; CASE1;
END1 ;
END1; TYPR1 ;
END1 ;
END1; STY11 ;
IF LFG THEN
BEGIN
IF OP EQ 5 THEN GENRE(3000B, 0, 0, FALSE) *CHA* ELSE
GENRE(0030048, 0, 0, FALSE) *CHA, IN11
END1;
LATIR := GAITR1
WHILE (NO EQ 71) OR (NO EQ 61) OR (NO EQ 81) OR (NO EQ 21) DO
BEGIN

WHILE (NO EQ 7) OR (NO EQ 6) DO
BEGIN
  OP1 = NO;
  LATOPCL = CL;
  INSYMBO;
  PRIMARY;
  IF SIV THEN
  BEGIN
    IF (LATR. TYPTR NE 0) AND (GATR. TYPTR NE 0) THEN
      BEGIN
        TEMP = LATR. TYPTR;
        BT1 = CONTEXTABLE(TEMP). FORM = NUMERIC;
        TEMP = GATR. TYPTR;
        BT2 = CONTEXTABLE(TEMP). FORM = NUMERIC;
      END;
      IF BT1 AND BT2 THEN
      BEGIN
        WITH GATR DO
        BEGIN
          CASE KIND OF
          BEGIN
            IF OP EQ 7 THEN
            BEGIN
              CASE LATOPCL OF
              BEGIN
                IF ACCESS EQ ORCI THEN
                BEGIN
                  GENRE(#20000B, DPLMT, 1, TRUE) ELSE
                  BEGIN
                    GENRE(#20000B, BFLC+4, 1, TRUE) ; LDA++
                    LTMP ;
                  END;
                  END;
                END;
                IF ACCESS EQ URCI THEN
                BEGIN
                  GENRE(#60000B, DPLMT, 1, TRUE) ; LDA++
                  GENRE(#70000B, 0, 1, FALSE) ; CH, IN+
                  GENRE(#40000B, 0, 0, FALSE) ; ADA TO B ;
                  BEGIN
                    GENRE(#162000B, BFLC+4, 1, TRUE) ; LDA++
                    LTMP ;
                  END;
                  END;
                END;
              END;
            END;
          END;
        END;
      END;
    END;
  END;
END;
CASE LANOPCL OF
BEGIN
  IF ACCESS EQ DRCT THEN
  BEGIN
    GENRE(100200B, 0, 0, FALSE); MPY
    GENRE(100200B, 0, 0, FALSE); AD FOR MPY
    GENRE(100200B, 0, 0, FALSE); IOR
  END ELSE
  BEGIN
    GENRE(162000B, DFLC*4, 1, TRUE); LDA
    LDMP; 
  END;
END;

ERROR(32);

BEGIN
  IF ACCESS EQ DRCT THEN
  BEGIN
    GENRE(6400B, 0, 0, FALSE); CLB
    GENRE(20200B, 0, 0, FALSE); SSA
    GENRE(70040B, 0, 0, FALSE); CMN INB
    GENRE(100400B, 0, 0, FALSE); DIV
    GENRE(100400B, 0, 0, FALSE); OP FOR DIV
  END ELSE
  BEGIN
    GENRE(162000B, 0, 1, TRUE); LDA
    LDMP; 
  END;
END;

END CASE CL+
END NO NE 7+
END VARAL +

SVAL1
BEGIN
  IF OP EQ 7 THEN
  BEGIN
    CASE LANOPCL OF
    BEGIN
      BEGIN
        GENRE(100200B, 0, 1, TRUE); ADA
        LDCST(VAL)
      END;
    END;
    BEGIN
      GENRE(65000B, 0, 1, TRUE); LDB+ LDCST(VAL); 
      GENRE(100700B, 0, 0, FALSE); CMN INB
      GENRE(040 01B, 0, 0, FALSE); ADA TO B
    END;
  END;
ERROR(32);
END CASE CL
END NO=7 ELSE
BEGIN CASE LADOPCL OF
BEGIN
  GENRE(1002000B,0,0,TRUE); +MPY+
  GENRE(0,0,1,TRUE); +OPERAND FOR MPY +
  LOCST(VAL1)
  GENRE(030001B,0,0,TRUE); +IOR+
END;
BEGIN ERROR(32);
BEGIN
  GENRE(0064000B,0,0,TRUE); +CLB+
  GENRE(0026000B,0,0,TRUE); +SSA+
  GENRE(0010000B,0,0,TRUE); +CMQ1 ING+
  GENRE(1004000B,0,0,TRUE); +DIV+
  GENRE(0,0,1,TRUE); +OPERAND FOR DIV +
  LOCST(VAL1)
END
END CASE CL
END NO=7 ELSE
END SVAL+1
END CASE KIND OF ;
END CASE ATTR+1
END BT1 AND BT2 ELSE
BEGIN IF LATR.TYPR = GATR.TYPR THEN
BEGIN IF (GATR.TYPR=800LPR) AND (LADOPCL EQ 3 ) THEN
BEGIN WITH GATR DO
BEGIN CASE KIND OF
BEGIN IF OP EQ 7 THEN
BEGIN GENRE(032000B,DPLMT,1,TRUE); +IOR+
END ELSE
BEGIN GENRE(012000B,DPLMT,1,TRUE); +AND +
END
BEGIN IF OP EQ 7 THEN

GENRE(003004B,0,0,TRUE) ;  "MA:A", NIA
GENRE(042000B,0,1,TRUE) ;  "ADA"
LOCST(VAL)

END  ; CASE KIND OF

RLEXPI
END  ; WITH GATIR
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; INSYMBO; 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(331) ; SKIP(53) END ;
  VARIABLE 1 AATTR IF GATTR 1
  IF NO NE 22 THEN
  BEGIN
    IF GATTR..YPTTR NE 0 THEN ERROR(34) ;
    SKIP(22) ;
    IF NO NE 22 THEN
    BEGIN
      IF GATTR..YPTTR EQ 0 THEN ERROR(34) ;
      GOTO 1
    END ;
  END ;
  WITH AATTR DO
  IF ACCESS EQ INXD THEN GENRE(172000B,BFLC+5,1,TRUE) ;
  INSYMBO1 EXPRESSION 1
  IF GATTR..YPTTR EQ 0 THEN SKIP(53) ELSE
  IF AATTR..YPTTR NE 0 THEN
  BEGIN
    IF AATTR..YPTTR NE GATTR..YPTTR THEN ERROR(35)
    ELSE BEGIN
      WITH AATTR DO
      BEGIN
        IF ACCESS EQ INXD THEN GENRE(172000B,BFLC+5,1,TRUE)
        ELSE GENRE(172000B,0PLMT,1,TRUE) ; STA+ 1
      END ;
      END ;
    END ELSE ERROR(32) ;
END * ASSIGN * ;
PROCEDURE REPEATSTAT;
VAR LJPAADDR ADDRESS;
BEGIN
  GENRE(0, 0, FALSE) = NOP;
  LJPAADDR = 10;
  REPEAT
    INSYMBOL = STATEMENT;
    IF ERROR <= 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 = SYS 1.
IF NO NE 33 THEN SYS UNILT+ ERROR(44) ELSE BEGIN
  INSYMBOL = EXPRESSION;
  IF GATR.ITYPR NE 0 THEN BEGIN
    GENRE(002003, 0, FALSE) = SZA, RSS;
    GENRE(025088, LJPAADDR, TRUE);
    JAP TO LJPAADDR;
    GENRE(0, 0, FALSE) = NOP;
  END ELSE SKIP(53);
END = REPEATSTAT;
PROCEDURE WHILESTAT:
VAR LJPADDR ADDRESS; LCA SHRINT:
BEGIN
  GENRE(0,0,0,FALSE); NOP +
  LJPADDR = IC;
  INSYMBOF EXPRESSION:
  IF GATIR.TYPTR NE 0 THEN
  BEGIN
    GENRE(00200038,0,0,FALSE); SZA,RSS+
    GENRE(026000B,0,1,TRUE); JMP+
    LCA = HCA;
  END;
  IF NO NE 35 THEN +SY#00+
  BEGIN
    IF GATIR.TYPTR EQ 0 THEN ERROR(40); SKIP(35);
    IF NO NE 35 THEN
    BEGIN
      IF GATIR.TYPTR EQ 0 THEN ERROR(40);
      IF ERRCLASS(O) EQ 2 THEN GOTO 20; GOTO 10
    END;
    END;
END:

201 INSYMBOF STATEMENT:
  GENRE(026000B,LJPADDR,1,TRUE); JMP TO LJPADDR+
101 GENRE(0,0,0,FALSE); NOP +
  HPASILCA; ADDR = IC;
END WHILESTAT+;
PROCEDURE GOTOSTAT;
BEGIN

  INSYMBOL;
  IF (NO NE 2) OR (CL NE 1) THEN
    BEGIN ERROR(44) 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 LASTTAB[IT] DO
    BEGIN

      IF LASTVAL EQ IVAL THEN LABEL ALREADY OCCURED !
      BEGIN IF DECL-OCC. GENERATE CODE ELSE CHAIN OCC.+

      IF FLDO EQ 0 THEN
        GENRE(0260006,FL03,1,TRUE) ! JMP TO FD03 ELSE
        BEGIN
          GENRE(0260008,0,1,TRUE) ; JMP
          IF CHNIX EQ 0 THEN
            BEGIN ERROR(47) GOTO 20 END
          END WITH UNDLAB[CHNIX] DO

      IT11=SUCG; SUCG=FL03; FL031=CHNIX;PLACE1=HCA
      END
      CHNIX = IT11;
      END
      GOTO 20 ;
      END "LABVAL = IVAL"
      END WITH LASTTAB+ ;
      "LABEL NOT YET MET, ENTER IT INTO LABELTABLE"
      BEGIN
      GENRE(0260008,0,1,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 LASTTAB[CLABIX]; UNDLAB[CHNIX] DO

      IT11=SUCG; LABVAL=IVAL; SUCG=0; FL021=CHNIX;
      FL031=CHNIX; PLACE1=HCA;
      END
      CHNIX = IT11;
      INSYMBOL;
  END "IF NO=2 AND CL=1 +

END "GOTOSTAT" ;
PROCEDURE FORSTATI:
VAR LATR, ATTR, LCLASS, LCA INTEGER,
LUPADOR ADDRESS, LOF BOOLEAN
BEGIN
  INSYMBOL
  IF NO NE 1 THEN BEGIN ERROR(11); GATTR.TYPTR=0 END ELSE
  BEGIN
    SCHRREC(NEXT)
    IF CTPTR EQ 0 THEN BEGIN ERROR(23); CTPTR=UNDECPTER END
    VARIABLE
    END
  LATR.GATTR
  IF NO NE 22 THEN BEGIN SY: NE 1 THEN
  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 ERRCLINO EQ 0 THEN GOTO 20; GOTO 10
  END
  END
  INSYMBOL EXPRESSION
  WITH LATR DO GENREC(720000, DPLMT, 1, TRUE)
  IF NO NE 37 THEN SY: NE 10/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 ERRCLINO EQ 0 THEN GOTO 20; GOTO 10
  END
  END
  LCLASS=GL; INSYMBOL EXPRESSION
  WITH LATR DO GENREC(720000, BFLC+3, 1, TRUE)
  STA IN BFLC+3
  LUPADOR=IC+1
  WITH LATR DO:
  BEGIN
    IF LCLASS EQ 1 THEN BEGIN
    GENREC(620000, DPLMT, 1, TRUE)
    GENREC(100000, 0, FALSE)
    GENREC(420000, BFLC+3, 1, TRUE)
    STA TEMP
    END
END ELSE \* DOWNTO+ 
BEGIN 
GENRE(62000B,0FLC+3,1,TRUE) \* LOA TEMP+ 
GENRE(30040,0,0,FALSE)+ \* CMA,INA+ 
GENRE(42000D,0PLMT+1,TRUE)+ \* ADA IDENTIFIER+ 
END 
END \* LATTR+ \* 
GENRE(20220,0,0,FALSE)+ \* SIA,SZAI+ 
GENRE(26000U,0,0,TRUE)+ \* JMP OUT OF THE LOOP+ 
LCA+ HCA+ 
END \* LATTR AND GATTR NE 0+1 
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[N0] EQ 0 THEN GOTO 20+ GOTO 10+ 
END \* DO+ 
END \* DOWNT+ 
STATEMENT+ 
200 
IF LATR.TYPTR NE 0 THEN 
BEGIN 
WITH LATR DO 
BEGIN 
GENRE(62000B,0PLMT+1,TRUE)+ \* LOA IDENTIFIER+ 
IF CCLASS EQ 1 THEN GENRE(20040,0,0,FALSE) ELSE 
BEGIN GENRE(20000B,0,0,FALSE)+ GENRE(400018,0,0,FALSE)+ 
END \* ADA TO B REGISTER+ END; 
GENRE(72000B,0PLMT+1,TRUE)+ \* SIA IDENTIFIER+ 
END \* 
GENRE(26000B,LJPDAR+1,TRUE)+ \* JMP BACK TO LOOP+ 
GENRE(0,0,FALSE)+ \* NOP+ 
HPAS[CA]+ AOR+ = IC+ 
END \* LATR.TYPTR NE 0+1 
100 \* ENDO FORSTAT+ 

PROCEDURE SRCHX (VAR P1 INTEGER);.AdapterView
A SEARCHES TABLES OF EXTERNALS

BEGIN
  CTPTR := P1;
  WHILE CTPTR NE 0 DD
  BEGIN
    EXTPR XEXTN[CTPTR].XName, Aval, Compar;
    IF Compar THEN GOTO 1 ELSE CTPTR := CTPTR - 1;
  END;
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 EXTN[XPTR] DO
      BEGIN
        I := 0;
        REPEAT
          I := I + 1;
          XName[I].I.Aval[I];
        UNTIL I EQ 10;
        EXSYM := EXSYM + 1;
        XSYM := EXSYM;
      END;
    END
    END COMPAR + 1;
END EXTERN+;
PROCEDURE WRITEIR ;
VAR STM $ INTEGER ;
BEGIN
WRFL $ TRUE ;
INSYM $ ;
IF NO NE 9 THEN BEGIN ERROR(50) ; SKIP(53) ; GOTO 1 END ;
INSYM $ ;
IF (NO NE 2) OR (CL : NE 1) THEN BEGIN ERROR(59) ; SKIP(53) ; GOTO 1 END ELSE BEGIN STM $ IFAL ; INSYM $ END ;
IF NO NE 15 THEN BEGIN ERROR(151) ; SKIP(53) ; GOTO 1 END ELSE INSYM $ ;
IF (NO NE 5) OR (CL NE 1) THEN BEGIN ERROR(60) ; SKIP(53) ; GOTO 1 END ELSE BEGIN INSYM $ IF NO NE 15 THEN ERROR(113) END ;
GENRE(0028008,0,0,FALSE) ; CLA +
GENRE(0720008,BFLC+2,1,TRUE) ; STA +
REPEAT
CIT $ TRUE ;
INSYM $ EXPRESSION ;
IF GATIR.TYPTR EQ 0 THEN ERROR(75) ELSE
BEGIN
GENRE(0660000,BFLC+1,TRUE) ; LDB +
GENRE(0460000.BFLC+2,1,TRUE) ; ADB +
IF GATIR.TYPTR EQ CHARPTR THEN
GENRE(0020000,0,0,FALSE) ; ENM +
ELSE GENRE(0021000,0,0,FALSE) ; CLE +
GENRE(0160038,0,4,FALSE) ; JSB +
GENRE(0420008,BFLC+2,1,TRUE) ; ADA +
GENRE(0720008,BFLC+2,1,TRUE) ; STA +
END ; TYPTR NE 0 ;
UNTIL NO NE 15 ;
IF NO NE 10 THEN (+) ERROR(78) ELSE INSYM $ ;
GENRE(0420008,BFLC+2,1,TRUE) ; ADA +
GENRE(0660008,BFLC+1,TRUE) ; LDB +
GENRE(0160028,J,0,FALSE) ; JSB +
GENRE(2000100,0,0,FALSE) ; DSS +
GENRE(STM $ ADA ; FALSE + LOGICAL UNIT NO. ;
END ; WRITEIR ;
PROCEDURE VARIAB;
BEGIN
IF NO NE 1 THEN
BEGIN ERROR(11) GATTR.TYPR = 0 END ELSE
BEGIN
SRCHREC(NEXT);
END;
IF CTIPTR EQ 0 THEN
BEGIN ERROR(31) CTIPTR = UNDECPTR END ELSE VARIABLE;
END;
****
PROCEDURE READIR;
VAR STHR, 1, INTEGER;
BEGIN
HWFL = TRUE;
INSsymbol:
IF NO NE 9 THEN BEGIN ERROR(50) SKIP(53) GOTO 1 END;
INSsymbol:
IF (NO NE 2) OR (CL NE 1) THEN BEGIN ERROR(59) SKIP(53) GOTO 1 END ELSE BEGIN STHR = IVAL INSsymbol END;
IF NO NE 15 THEN BEGIN ERROR(51) SKIP(53) GOTO 1 END ELSE INSsymbol:
IF (NO NE 6) OR (CL NE 1) THEN BEGIN ERROR(60) SKIP(53) GOTO 1 END ELSE BEGIN INSsymbol:
IF NO NE 15 THEN ERROR(51) END;
GER(86, 0003, BFLC, 1, TRUE) SDB BUFFER ADDRESS
GER(160018, 0, 0, FALSE) JSB GET
GER(20018, 0, 0, FALSE) RSS
GER(20018, 0, 0, FALSE) LOGICAL UNIT NUMBER
GER(86, 0003, BFLC, 1, TRUE) SDB BUFFER POINTER
REP
CLT = TRUE
INSsymbol: VARIAB
IF GATTR.TYPR EQ 0 THEN ERROR(47) ELSE
BEGIN
IF GATTR.TYPR EQ CHARPTR THEN GENR(22008, 0, 0, FALSE)
GENR(21008, 0, 0, FALSE) CLE
GENR(86, 0003, BFLC, 1, TRUE) SDB BUFFER POINTER
GENR(160048, 0, 0, FALSE) JSB ALLOK
END;
BEGIN
IF ACCESS EQ ORCT THEN
GENR(72006B, OPLAT, 1, TRUE) STA IN VARIAB ELSE
GENR(172000B, BFLC, 1, TRUE) STA ARRAY ELEMENT
END;
GENR(760008, BFLC, 1, TRUE) STB THE BUFFER POINTER
END;
END * TYPTR NE 0*
UNTIL NO NE 15 **
IF NO NE 10 THEN ** ERROR(7) ELSE INSsymbol;***
** END * REadir; **
PROCEDURE COMPSTAT;
BEGIN
REPEAT
BEGIN
INSYMBO1;
STATEMENT;
IF NO EQ 29 THEN ELSE;
BEGIN ERROR(36); INSYMBO1; GOTO 1 END;
END;
UNTIL NO NE 16;
IF NO NE 26 END THEN ERROR(15) ELSE INSYMBO1;
END * COMPSTAT*;
PROCEDURE STATEMENT
VAR LPSH : 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(56) /
  FOR IT 1=1 TO CLABIX DO
  WITH LATAB[I] DO
  IF LABVAL EQ IVAL THEN *FOUND*
BEGIN
  IF FLDO EQ 0 THEN *MULTIDEF* ERROR(48) ELSE *FIXUP*
BEGIN
  IT1 = FLDO;
  REPEAT
  WITH UNDLAB[IT1] DO
  BEGIN
  WITH HPAS[PLACE] DO ADRR := IC ; IT1 := SUCG
  END
  UNTIL IT1 EQ 0
  IF IT1 = FLDO THEN UNDLAB[IT1] SUCG := CHNIX
  CHNIX := FLDO; FLDO := 0; FLDO1 := IC
  END
  GOTO 1
BEGIN
ENDIF FOR, WITH +
NEW LABEL +
IF CLABIX EQ MAXLAB THEN ERROR(46) ELSE
BEGIN
CLABIX := CLABIX + 1
WITH LATAB[CLABIX] DO
BEGIN LABVAL := IVAL; FLDO := 0; FLDO1 := IC END
END
END INSYMBO
IF NO NE 21 THEN +1
BEGIN ERROR(10) *SKIP(53) END ELSE INSYMBO
ENDIF IF NO=4 AND CL=1 *
BEGIN TRUE ; CTY1 = TRUE ;
CASE SPLITSTAT(0) OF
SPLASH +1 * DO OR 2 * 1
SRCHREGNEXT ;
IF CTIDR EQ 0 THEN BEGIN ERROR(23) END
WITH CONTEXTABLE[CTIDR] DO
IF KLASSE KONST THEN ERROR(54) ELSE
IF KLASSEBRG AND (PROCSTATE EQ 0) OR
(PROCTYPE EQ 0)) THEN

FOLLOWING ROUTINES ARE USED TO PROVIDE CODES IN RELOCATABLE FORMAT

PROCEDURE PUNCHRECORD(RENNAME: ALFA; RL: INTEGER);
VAR I, N, J, K, C, REM: INTEGER;
GLOBAL N, M;
BEGIN
IF LISTBIN THEN /* LIST BINARY RECORD */
BEGIN
N := 0;
WRITE(RENNAME); LINES(2);
FOR I := 1 TO RL DO
BEGIN
N := N + 1;
OUTI(W[I]): WRITE(E); 
END;
END;
LINES(4);
IF PUNCHBIN THEN /* PUNCH BINARY RECORD, RELOCATABLE FORMAT */
BEGIN
FOR I := RL + 1 TO 60 DO W[I] := 0;
END;
REPEAT
REM := 0; K := 0;
FOR I := 1 TO 4 DO
BEGIN
REM := REM + 100;
FOR J := C + 1 TO C + 3 DO APPEND(N, 16, W[J]);
K := K + 4;
IF K LT 16 THEN
BEGIN
C := C + 1;
J := W[C] DIV 10000;
REM := W[C] - J * 10000;
END
APPEND(N, 16 - K, J);
END;
END;
END;
END /* PUNCHRECORD */
FUNCTION CHECKSUM(RL INTEGER) INTEGER;
VAR I, CS INTEGER; *GLOBAL -- NL +
BEGIN
CS := MFI; 1
FOR I := 0 TO RL DO CS := CS + MFI; 1
CHECKSUM := CS MOD 200000B
END * CHECKSUM +;

************
************
************
************
PROCEDURE PACKIN(VAR NIM: AR; VAR B: ARRINT3);
VAR I, J, N: INTEGER;
BEGIN
NIM[6] := N :=
FOR I := 4 TO 3 DO
BEGIN
J := 2*I - 1
B[i] := ASCIICODE[NIM(J) - 1]
IF (((J + 1) EQ 6) THEN N := 0 ELSE N := ASCIICODE[NIM(J + 1)] - 1
APPEND(B[i], 0, N); 1
END
END * PACKIN +;

************
************
************
************
PROCEDURE NAMECORD;
VAR P: ARRINT3; *GLOBAL - W, PNAM, LC
BEGIN
W[1] := 010;00B; 1 RECORD LENGTH, LEFT SHIFTED 8 BITS +
W[2] := 208;00B; 1 IDENT +
APPEND(P, 0, 80); 1 FOR I := 1 TO 3 DO
W[I] := 80;
END * NAMECORD +;

************
************
************
************
PROCEDURE EXTRECORD;
VAR I, J, K, N1, N2: INTEGER; *GLOBAL - W, EXSYM *
BEGIN
N1 := 11
IF EXSYM LE 19 THEN BEGIN MORE := FALSE; N2 := EXSYM END ELSE
BEGIN MORE := TRUE; N2 := 19 END;
NEW EXTRECORD;
PL := 3; M2 := RECORD LENGTH; 1
FOR I := 1 TO N2 DO
W[I] := 80;
END * NAMECORD +;

************
************
************
************
PROCEDURE SEDQ;
VAR W: ARRINT3;
BEGIN
W := SOME ARRINT3
END * SEDQ +;
K1=31
FOR I1=M1 TO M2 DO
BEGIN
PACKING(X111,1) XNAM,P) ; APPEND(P[3],0,I1)
FOR I1=1 TO 4 DO W11=I1+PL[I1] ; K1=K1*3
END
W[3] = CHECKSUM(RL) ; CHECKSUM +
PUNCOREC(RECORD,RL)
IF MORE THEN
BEGIN
M1=M1+1 ; M2=M2+201
IF M2 GE EXSYM THEN BEGIN M2=EXSYM; MORE=FALSE END
GO TO 1
END
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******

************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +

************ ******* ******* *******
PROCEDURE OBLRECORD(PGIBITS)
VAR M1,J,K,L,N1:INTEGER; ENDC=BOOLEAN
BEGIN
************ ******* ******* *******
REPEAT NEW RECORD EVERY LOOP*
I1=1; N1=0; ENDC=FALSE
WHILE (L/LT 58) AND (NOT ENDC) DO
BEGIN
M0=0; L1=L+11; J1=J+L; W[J1]=0;
WHILE (W[J1] CONTAINS RELLOCATION, INDICES FOR UPTO NEXT FIVE WORDS
REPEAT
N1=N1+1 ; L1=L1+1
IF HPAS[I].HREF THEN MEMORY REFERENCE
BEGIN
APPEND(W[I],3,5)
W[I11]=HPAS[I11].INST
CASE HPAS[I11].TYP OF
11 K1=0; 21 K1=1; 31 K1=2
END
APPEND(W[I],0,K1)
L1=L1+1; W[I1]=HPAS[I11].ADDR
END ELSE NO MEMORY REFERENCE
BEGIN
IF HPAS[I11].INST EQ 0 THEN W[I1]=HPAS[I11].ADDR ELSE
W[I1]=HPAS[I11].INST
APPEND(W[I],3,HPAS[I11].TYP)
END
I1=I1+1
IF I GT MCA THEN ENDC=TRUE
************ ******* ******* *******
BEGIN
************ ******* ******* *******
END = EXT RECORD +
UNTIL (M EQ 5) OR ENDC OR (L GE 59); 
APPEND(W[I],(5-M)*3 + 1,0); 
NI=NI+M; 
END; 
W[I]=0; INSERT(L,8,W[I]); 
W[2]= 600000+PG*100+NI; 
W[3]= CHECKSUM(L); 
PUNCHRECORD(EDBL RECORD,L); 
UNTIL ENDC; 
END * DBLRECORD *; 
********** 
PROCEDURE ENDRECORD ; 
VAR II:INTEGER * GLOBAL — M,TRNSFR 
BEGIN 
W[I]=20000; * RECORD LENGTH=4, SHIFIED TO 8 BITS; 
W[4]= TRNSFR[3]; * FOR MAIN PROGRAM IT IS 0; 
PUNCHRECORD(=END RECORD=,4); 
END * ENDRECORD *;
PROCEDURE ENTERBODY:

* Generates procedure prolog code and
* procedure entry code

VAR:

VAR II INTEGER;
BEGIN

LGX := 0; HCA := 0;
CLABIX := 0;
IF LEVEL NE 0 THEN
END ELSE
BEGIN MAIN
BEGIN

IF ValP THEN
BEGIN
II := 1;
FOR II TO LC DO
BEGIN GENRE(0,VAR(II),0,FALSE); II := II + 1 END
END;
END
END
END}

************ ENTERBODY

************
PROCEDURE EXITBODY
* GENERATES CODE TO HALT THE EXECUTION OF THE PROGRAM
BEGIN
    GENRE(160050, 0, 0, FALSE); JSB EXEC
    GENRE(0, 0, 0, FALSE); DEF **2
    GENRE(0, 0, 0, FALSE); DEF **2
    GENRE(0, 0, 0, FALSE); NOP
    GENRE(0, 0, 0, FALSE); DEC 16
END; EXITBODY

END ** END

END ** END
BEGIN
* MAIN PROCEDURE OF HPCCOM CALLING DIFFERENT PARTS OF THE
* COMPILER AS NECESSARY
   CH = INPUT
   SEG1 = INITIALIZATION
   IF (NO EQ 43) OR (NO EQ 43) THEN SEG2
   * CONSTANT OR LABEL DECLARATION PART
   ********
   IF NO EQ 40 THEN = TYPE
   BEGIN INSYM
   WHILE NO EQ 1 DO
   BEGIN SRCCHREG(NEXT)
   IF (CIPTR NE 0) THEN ERROR(8)
   I = 0
   REPEAT I = I + 1
   IYPDII = AVAL(I)
   UNTIL I = 10
   INSYM
   IF (NO EQ 8) OR (CL NE 6) THEN ERROR(9) ELSE INSYM
   FOR ERR = FALSE; TYPEDECL(I, P) DO
   IF ERR THEN
   BEGIN
   COMPCONTEXTABLE(P, NAME, BLANK, COMPARE)
   IF NOT(COMPARE) THEN ERROR(55) ELSE
   BEGIN WITH CONTEXTABLE(P) DO
   BEGIN REPEAT
   IYPIII = IYPDII
   NAME(I) = IYPDII
   UNTIL I = 10
   NXTE = NEXT
   END
   NEXT P
   END
   END TYPE
   ********
   IF NO EQ 51 THEN SEG4
   * SYNONYM DECLARATION
   IF NO EQ 42 THEN SEG31
   * VALUE DECLARATION
   IF NO EQ 40 THEN SEG5 ELSE VLC = LC1;
   LC1 = 1
   IF LC IS INITIALIZED TO -1, SO THAT LOAD POINT IS ZERO
END

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 $125 ; START A NEW PROGRAM *
        CH$ = INPUT$ ; WRITE(CH$) GOTO 1
    END
END
APPENDIX H

SAMPLE COMPILED PROGRAMS IN PL/2100
PROGRAM GCD, L, P, B /* TO FIND GCD(X, Y) */
VAR X, Y, A, B: INTEGER;
BEGIN
READ(*, X, Y); /* X, Y */
A := X; B := Y;
REPEAT /* A > 0, B > 0 */
WHILE A > 0 AND B > 0 DO A := A - B;
END  /* A = B = GCD(X, Y) */
WRITE(*, X, Y, A, B);
END
PROGRAM DIV,B,L,P

VAR X,Y,R,Q,H : INTEGER;
BEGIN
  READ(*,01B,*+X,Y);
  R:=X; J:=0; W:=Y;
  WHILE H LE P DO W:=2*W;
  WHILE W NE Y DO
    BEGIN /* Q*H+R=X,R GE 0 */
      Q:=2*Q; W:=W DIV 2;
    END;
  IF H LE R THEN BEGIN
    R:=R-H; Q:=Q+1;
  END;
  END;
  /* Q*Y+R=X,05R>W,Q=X DIV W */
  WRITE(*,01B,*+X,Y,Q,R,H);
END.
PROGRAM LST,P,L,B

PROGRAM LST,P,L,B

/* FINDING THE LEAST NUMBER THAT IS EQUAL TO TWO DIFFERENT SUMS OF TWO NATURAL NUMBERS RAISED TO THE THIRD POWER */

VAR

I,IL,II,MIN,A,B,K* INTEGER

J,P,PI ARRAY [1..12] OF INTEGER


BEGIN

IL=1; IL=1; IH=2;
REPEAT

MIN=S[I]+I++; D=J[I];
IF J[I] EQ I THEN IL=IL+1 ELSE
BEGIN IH=IH+1; P[IH]=IH*IH*IH;
J[IH]=I; S[IH]=P[IH]+1;
END;

II=IL; K=II
WHILE K LT IH DO
BEGIN
K=K+1;
IF S[K] LT S[I] THEN II=K;
END;

END;

UNTIL S[I] EQ MIN;

WRITE(9,*A,B,L,J[I]);

END;

/* TO FIND LEAST X SUCH THAT */

X=A**3+B**3=C**3+D**3

WHERE A,B,C,D ARE NATURAL NUMBERS

SUCH THAT A*D AND A+D */

PROCEND
PROGRAM TST1, L, B, P

CONST MIN=4, MAX=9

TYPE R = RECORD
    A, B : INTEGER
END

VAR C, D, E : INTEGER

BEGIN
    A := 2
    B := 5
    C := A + MIN
    D := C + MAX
    WRITE(1, *, D, E)
END

END

PROGEN
PROGRAM TST2,B,P,L

CONST  MIN=4, MAX=10 ;
 SYN    A = B+2 ;
        D = E+4 ;
 VAR    A,D,E : INTEGER ;

BEGIN  A := MIN ; WRITE(1,*,MIN) ;
        B := MAX; E := MIN ;
        L := B+E ;
        WRITE(1,*,L,B,E) ;
 END ;

END
PROGRAM TSJ.BPL

000000  CONST SH=5, SHT=101
000000  VAR A, B, C, D : INTEGER ;
000004  BEGIN
000006  A := 4; B := 2; C := 4;
000013  J := A; LFS := 0; ALF := 7; A RAL := 14;
000021  X := C; BLF := 10; A BLS := 9; C RBL := 10; D RBR := 0 ;
000033  END ;
000050  END ;
000177  PROGENE
PROGRAM TST4, B, P, L
VAR A, B, C, D : INTEGER;
BEGIN
  A := 2;  B := 3;  C := 4;
  REPEAT
    A := A + 1;
    D := B + C;
  UNTIL A = 10;
  WRITE(1, *, A, D);
END
END
```plaintext
PROGRAM TEST;
VAR A, B, C, D: INTEGER;
BEGIN
  A := 1;
  B := 3;
  C := 5;
  D := 10;
  E := A + C * (D - B);
END.
```
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(4010):C,D
000052 A := B+C+D
000062 WRITE(8,*):A,B,C,D
000071 END;
0000246 PROGRAM
VAR
  A; ARRAY [1..10] OF INTEGER;
  B, C, D: INTEGER;

BEGIN
  A[1] := 10;
  B := 2;
  C := 5;
  WRITE(1, *, D, A[2]);
END;
PROGRAM TST9, B, D, L
VAR
    A, B : INTEGER;
    C : ARRAY [1..10] OF INTEGER;
    D : CHAR;
VALUE
    C := [2, 3, 4.5, 1, 5, 6];
    D := 'A';
BEGIN
    A := C[3] + C[4];
    B := C[7] + 5;
    WRITE(*, A, B, D);
END;
PROGEND
PROGRAM TST11, B, P, L
VAR
  A : ARRAY [1..5, 1..10] OF INTEGER ;
  C : ARRAY [1..5] OF CHAR ;
  I : INTEGER ;

VALUE
  A = (10*1, 10*2, 10*3, 10*4, 10*5) ;
  C = (EA, EB, EC, ED, EE) ;

BEGIN
  I := 0;
  REPEAT
    I := I + 1 ;
    WRITE(1, *, C[I]) ;
    WRITE(8, *, A[I, I]) ;
  UNTIL I EQ 5 ;
END ;
PROGEND
PROGRAM TST12, B, P, K

CONST A = 10, B = 11;

TYPE
  G = RECORD
      D : ARRAY[1..10] OF INTEGER;
   END;

VAR
  M, N : CHAR;
  I, J, K : INTEGER;
  L : G;
  VALUE
    M = 'E',
    N = 'E',
    I = 200;
  J = 5;
BEGIN
  WRITE(I, 'I', J);
  WRITE(I, 'I', J, M, N);
END.
PROCEND