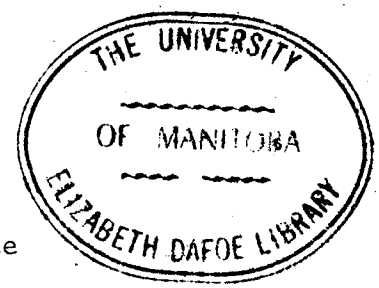
IPLAN - An Introductory Programming Language

A Thesis

Presented to

the Faculty of Graduate Studies and Research

The University of Manitoba



In Partial Fulfilment

of the requirements for the Degree

Master of Science

in the Institute for Computer Studies

---

by

Peter M. Finch

May 1969

ABSTRACT

The advantages and disadvantages of several existing methods of teaching programming are discussed and the basic requirements of an introductory programming language are considered. IPLAN (an Introductory Programming LANguage) is proposed to fulfill these requirements and its design goals are stated.

The syntax and semantics of IPLAN are specified and the details of the implementation on an IBM 360/65 are outlined. The appendices contain example programs illustrating the use and special features of the language.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to Professor S. R. Clark, my supervisor, for his advice, guidance and assistance, which have been essential to me in the preparation of this thesis.

Special thanks is extended to Professor T. A. Rourke for his comments and criticisms.

Finally, I wish to especially thank Miss R. E. Gould for all her help and for typing this thesis.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER I

Introductory Programming Languages

1.1 Introduction

The growing importance of computers in our society requires that a large proportion of future generations needs some knowledge of the operation of computers, so that the layman does not regard them with the mystical admiration or fear that is typical of today's attitude. In a few years it is reasonable to suppose that not only bright students in a few high schools, but many students in most high schools will be studying computer courses and until such time as computers become conversant in natural languages, students will be forced to learn programming languages.

The language taught in introductory programming courses could be selected from one of four groups :

scientific programming languages;

commercial programming languages;

assembly level programming languages and;

specially designed introductory programming languages.

1.2 Use of Scientific Programming Languages for Teaching

The present trend in teaching programming languages in universities is to start with a diagnostic form of a scientific language, for example, WATFOR [1], PUFFT [3], or DITRAN [2] if the main scientific language of the university is FORTRAN [4]. Although an honors student with a scientific background can normally master the complexities of such an initial programming language, the average student and the high school student may easily become confused and/or discouraged.

Both advantages and disadvantages of using a diagnostic scientific language as a first programming language are inherently related to its parent language. At this university, second year students are taught the WATFOR version of FORTRAN IV. From the author's experience of demonstrating this course, several points can be made.

1.2.1 Advantages of WATFOR

During the course the student may tackle problems of considerable complexity, which may strongly stimulate his interest.

If WATFOR is well known by the student at the end of the course, he has an almost complete knowledge of a standard and widely used scientific language.

1.2.2 Disadvantages of WATFOR

Most problems fall into one of three groups related to design faults in the parent language (i.e. FORTRAN), which WATFOR could easily have avoided but for the design goal of compatibility with FORTRAN.

The groups are :

a) Conceptual problems

b) Problems of detail

c) Errors.

a) Examples of conceptual problems are :

the meaning of the "=" sign;

the carriage control character for the printer;

the difference between array and function references and;

the relationship between input-output executable statements and format statements.

The problems created by the last one were severe enough for WATFOR to introduce a minimal field-free format input-output at the expense of full FORTRAN compatibility.

b) Problems of detail embedded in the parent language absorb too much time and effort and lead to unnecessary errors. For example, consider the format statement. There would seem to be no reason why a field width should not be allowed with the 'X' format code (e.g. I5 is correct, X5 is not). Also the action taken when the I/O list is longer than the

number of format elements is not simple. From the FORTRAN IV Language Specifications : "If there are more items in the I/O list than there are format codes in the FORMAT statement, control is transferred to the group repeat count of the group format specification terminated by the last right parenthesis that precedes the right parenthesis ending the format statement." [4]

c) The formidable problem of teaching students to find their own errors is complicated by two factors :

(i) The complexity of FORTRAN syntax allows "obvious" errors to be compiled and executed without comment.

Consider this example in which a period has been keypunched in place of a comma :

```
      DO 55 I = 1.12
      ..............
   55 CONTINUE
```

Both the FORTRAN IV level G and the WATFOR compiler declare by default a floating point variable "DO55I" and assign to it a value of 1.12. The intended DO loop is executed only once and with an invalid value of I.

(ii) The debug facilities of WATFOR are inadequate. Although it checks for the use of variables that have not been given a value and checks the range of subscripts, it lacks, in particular,

a trace. Even though the student knows his program is in an endless loop, the statement number in which his program was cancelled often is insufficient help for him to find his error. (The FORTRAN IV level G compiler [4] has several debug routines, but they have to be requested and are unnecessarily complex to use.)

1.2.3 Disadvantages of Scientific Languages in General

Although the above examples are selected from teaching a diagnostic form of FORTRAN, similar examples may be found in most other scientific programming languages (except that the ALGOL-like languages are relatively free from the problems related to complex syntax).

A further general disadvantage is that it is not easy to start by teaching a simple subset since certain of the problems, for example, complex syntax and input-output, cannot be avoided.

The last major disadvantage is the possibility that even after successfully learning a scientific programming language, the student is still not aware of the basic computer operations. He has learned how to control a powerful tool, but not how to use it most efficiently.

1.3 Use of Commercial Programming Languages for Teaching

Apart from the advantage that commercial programming languages

have in making the student familiar with a language used in business applications, the use of these languages for teaching seems to have more disadvantages and fewer advantages than the use of scientific programming languages.

For example, consider COBOL [5] since this is the main commercial language. In COBOL, like FORTRAN, input-output is not simple, the "=" sign raises conceptual problems, and the complex syntax causes unnecessary errors. The source program is very format sensitive (blanks used as delimiters, 'A' margins defining paragraphs and 'B' margins defining statements). The concept of the four divisions, identification, environment, data and procedure, is not basic and is not necessary for an introductory language. Finally there are no widely used, if any, diagnostic COBOL compilers and the manufacturer supplied compilers do not do the error checking necessary for an introductory language.

1.4 Use of Assembly level Programming Languages for Teaching

1.4.1 Actual Assembly Languages

Although assembly level languages demonstrate exactly how the computer operates and make available to the programmer its full capabilities, these languages are usually too complex for the beginner

to program in, especially with respect to input-output, and require a far too detailed knowledge of the machine operation to be suitable for an introductory programming language.

1.4.2 Assembly Languages for Simulated Computers

Such languages are very valuable as teaching aids, but are normally only taught as a brief introduction to teaching a high level language because of their clumsiness for advanced problems and their restricted input-output.

For example, consider the SPECTRE computer [6], which is similar to the hypothetical computer described in T. E. Hull's book "Introduction to Computing" [7]. The SPECTRE-MAP* instructions "INP" (input) and "OUT" (output) are the basic input-output instructions. "INP A" inputs into location A the signed ten digit number punched in columns 1 to 11 of the next data card. Sign and leading zeroes must be punched. Similarly "OUT A" prints the signed ten digit number in location A into a fixed field on a new line on the line printer.

Advanced problems in SPECTRE-MAP are complicated by the lack of index registers and hence address modification (i.e. run time code modification) has to be used to simulate indexed instructions.

---

* SPECTRE-MAP is the assembly level language of the SPECTRE computer.

1.5 Special Introductory Programming Languages

1.5.1 Existing Introductory Programming Languages

Several universities have written their own languages. Notable in this group is the Cornell Computing Language CORC [8], which has had a major influence on later diagnostic scientific languages. CORC goes to considerable lengths to correct the programmer errors, in particular, the incorrect spelling of variable names. Unfortunately CORC is badly restricted in source program format (blanks used as delimiters, FORTRAN type continuations, etc..) and has limited input-output.

Other teaching languages are mainly scientific language subsets and in particular are subsets of FORTRAN. Examples are FORGO [9] and MAD [10]. None of the existing teaching languages fulfills all of the basic requirements that the author feels are essential for an introductory programming language.

1.5.2 Basic Requirements of an Introductory Programming Language

These requirements are felt to be :

(i) Simple syntax.

(ii) Simple but powerful input-output.

(iii) Comprehensive error diagnostics.

(iv) Adequate debug facilities.

(v) Explicit data types. From early on, it is important that the intrinsic difference in types be understood and in particular the concepts of range and accuracy as related to real and integer numbers.

(vi) An arithmetic instruction set. These should be in a one-to-one relationship with basic machine instructions.

(vii) Conditional and unconditional branch statements (and hence statement labels).

(viii) Subscripting and loop facilities

(ix) Time, size and number of statements executed as measures of program efficiency.

In particular, it is felt to be important that the following are avoided :

(i) Implicit conversions.

(ii) Implicit declarations.

(iii) Input-output with format statements.

(iv) Field-dependent source statement format.

(v) A system of error analysis that allows an apparently correct statement to compile and perhaps even execute without warning.

The author's views are in keeping with the following extract from the detailed syllabus for a proposed course entitled "Introduction to Computer Science" as outlined in "Specimen Courses in Computer Science"

by a Computer Bulletin working party :

> "Detailed syllabus : Introduction to a programming language and the computer. The two facets should be integrated by allowing the student to write simple programs which demonstrate : basic input and output facilities; arithmetical operations; storage and program control. The language used should be a simple teaching language and not languages like FORTRAN and ALGOL ...." [11]

There are other features such as arithmetic expressions which might well be included in an introductory language. Indeed several additional features are discussed in Chapter 4. At this stage of the design and development of the language it is not possible to state categorically which features should and which should not be included in an introductory language. It is only possible to present a proposal and to be prepared to alter this as a result of experience gained through using the language to teach programming.

1.6 The Introductory Programming Language IPLAN

This thesis presents an introductory programming language called IPLAN (Introductory Programming LANguage) which is an attempt to fill the need for a specialised teaching language.

1.6.1 IPLAN design goals

These goals are :

(i) Ease of learning

The language must be simple to learn. There will be one statement to a card with a keyword (which will be a word in the English language and not a mnemonic) followed by an operand(s) and optionally preceded by a statement label. The source program and the data will be field-free format or, simply, free format.

(ii) Non logic errors

All errors will be flagged and a full diagnostic error message (not just an error code) given. In the case of a non terminal execution error, the number of the card in which the error condition arises will be given.

Compilation errors will not prevent execution and execution will continue as far as is thought to be useful.

When a terminal error does occur, a full dump of data areas and information about the state of the program will be given.

(iii) Logic errors

Adequate debug facilities to trace logic errors.

A further design goal arises in the implementation.

(iv) Fast throughput of student jobs.

1.6.2 Versions of IPLAN

Three versions of the IPLAN compiler and control program are envisaged :

The 'main' version described in this thesis.

An extended version as outlined in Chapter IV.

An integer version (see Appendix 6).

CHAPTER II

Semantic Definition of the Language

2.1 Introduction

Only one statement may appear on each card and there are no continuations. The compiler ignores all blanks and all characters between a left angle bracket (<) and a right angle bracket (>). (Exception : see PRINT TEXT). Character strings thus enclosed are called "comments" and are used to document the program. Comments may precede or follow a statement and if no right angle bracket is found, the comment ends at the end of the card.

For the remainder of this description blanks and comments are assumed to have been deleted.

A definition of the meta-language is given in Appendix 1.

2.2 Programs

<program> ::= BEGIN PROGRAM<declaration group><statement group>
END PROGRAM

A program is delimited by the keywords (or statements) BEGIN PROGRAM and END PROGRAM, and all the declaration statements must precede any of the executable statements.

2.3 Keywords

<keyword> ::= <opcode>|<conversion>|<debug>|<transfer opcode>|
STORE|STOP|NEW PAGE|NEW LINE|SPACE|PRINT TEXT|
READ INTEGER|READ REAL|PRINT INTEGER|PRINT REAL|
CYCLE|FOR|REPEAT|REAL SCALAR|INTEGER SCALAR|REAL
VECTOR|INTEGER VECTOR|REAL MATRIX|INTEGER MATRIX|
BEGIN PROGRAM|END PROGRAM

Keywords are not reserved names.

2.4 Statement numbers

<statement< ::= <unlabelled statement>|<integer>:<unlabelled statement>

<statement group> ::= <statement>|*<statement>

Any statement may be labelled by an integer. (See<integer>). If the statement is non executable, the statement number is considered to be attached to the first following executable statement.

2.5 Numbers

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

<sign> ::= +|-

<integer> ::= <digit>|*<digit>

<signed integer> ::= <sign><integer>|<integer>

<real> ::= <integer>.<integer>

<number> ::= <integer>|<real>|<sign>{<integer>|<real>}

Numbers have their conventional meaning.

2.6 Variables

<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

<identifier> ::= <letter>|*<letter>

<simple integer variable> ::= <identifier>

<subscript> ::= <integer>|<simple integer variable>|{<integer>|<simple integer variable>},{<integer>|<simple integer variable>}

<integer variable> ::= <simple integer variable>|<identifier>(<subscript>)

<real variable> ::= <identifier>|<identifier>(<subscript>)

<variable> ::= <integer variable>|<real variable>

Variable names serve for the identification of scalars, vectors (one dimensional arrays) and matrices (two dimensional arrays). Each variable name must have a unique first 16 characters. A variable has a type (real or integer) and a form of storage (scalar, vector or matrix) which are determined once in a declaration statement. Vectors and matrices cannot be referred to as a whole, but must be referenced via their elements as subscripted variables.

2.7 Declarations

```
<type scalar>  ::=  REAL SCALAR|INTEGER SCALAR

<scalar list>  ::=  <identifier>|*,<identifier>

<scalar
   declaration>::=  <type scalar><scalar list>

<type vector>  ::=  REAL VECTOR|INTEGER VECTOR

<vector list>  ::=  <identifier>(<integer>)|*,<identifier>(<integer>)

<vector
   declaration>::=  <type vector><vector list>

<type matrix>  ::=  REAL MATRIX|INTEGER MATRIX

<matrix list>  ::=  <identifier>(<integer>,<integer>)|*,<identifier>
                    (<integer>,<integer>)

<matrix
   declaration>::=  <type matrix><matrix list>

<declaration>  ::=  <scalar declaration>|<vector declaration>|
                    <matrix declaration>

<declaration
   group>      ::=  <declaration>|*<declaration>
```

Declarations define the properties of variables, i.e. their type (real or integer) and their form of storage (scalar, vector or matrix), which must be declared for all variables.

In vector and matrix declarations the lower subscript bound is one and the upper is specified. There are no dynamic declarations.

2.8 The Accumulator

All arithmetic and branch operations implicitly use the accumulator. The type of the accumulator (real, integer or undefined) is specified by the most recently executed load or conversion operation. Mixed type operations are not permitted. (At execution time, the operand type of an ADD, SUBTRACT, MULTIPLY, DIVIDE, or STORE instruction is checked against the accumulator type.)

2.9 Simple Statements

<opcode>	::=	LOAD|ADD|SUBTRACT|MULTIPLY|DIVIDE
<conversion>	::=	CONVERT TO REAL|CONVERT TO INTEGER
<debug>	::=	TRACE ON|TRACE OFF|MONITOR ON|MONITOR OFF| DUMP ALL
<transfer opcode>	::=	GO TO|IF POSITIVE GO TO|IF NEGATIVE GO TO| IF ZERO GO TO
<simple statement>	::=	<opcode>{<variable>|<number>}|STORE<variable>| <conversion>|<transfer opcode><integer>|<debug>| STOP
<unlabelled statement>	::=	<simple statement>|<printer statement>| <i-o statement>|<loop>|<declatation>|BEGIN PROGRAM| END PROGRAM

The LOAD statement copies into the accumulator the contents of the storage location specified by the operand, which remain unchanged.

The STORE statement copies into the storage location specified by the operand the contents of the accumulator, which remain unchanged.

The ADD (SUBTRACT) statement adds to (subtracts from) the accumulator the contents of the storage location specified by the operand.

The MULTIPLY (DIVIDE) statement multiplies (divides) the accumulator by the contents of the storage location specified by the operand.

The CONVERT TO INTEGER statement converts the type of the accumulator to integer and assigns an integral value to the accumulator, which is the nearest integer to the previous contents, i.e. in the terminology of the "Revised Report on ALGOL 60", [12] entier (contents of the accumulator +0.5). N.B. An error will occur if an attempt is made to convert a real number larger in magnitude than the largest integer number.

The CONVERT TO REAL statement converts the type of the accumulator to real and assigns a real value equal to the previous contents.

The GO TO statement transfers control unconditionally to the statement whose statement number is specified by the operand.

The conditional GO TO statements test the accumulator and if the condition is satisfied, control is transferred to the specified statement without altering the contents of the accumulator.

The STOP statement terminates the execution of the program.

The END PROGRAM statement, as well as delimiting the source program, terminates the execution of the program.

The TRACE ON (OFF) statement causes a trace to be printed (or not). After a TRACE ON statement has been executed and until a TRACE OFF statement is executed the card number of each statement executed is printed.

The MONITOR ON (OFF) statement causes the accumulator to be monitored (or not). After a MONITOR ON statement has been executed and until a MONITOR OFF statement is executed the card number and the type and contents of the accumulator are printed for each statement executed which changes the type or contents of the accumulator.

The DUMP ALL statement requests a program information dump. The card numbers of the last forty executed statements are printed together with the names and values of all variables.

2.10 Input-Output

Input is from punched cards and output is on the line printer. Both are basically considered as streamed, meaning that the records (cards on input, lines on output) are viewed as concatenated into a continuous stream. The exception to this concept is on input, since a number is considered to end at the end of a card. In printing, the

ability to split up the output into records exists with the NEW LINE and NEW PAGE instructions. A line on the printer is 132 characters long.

2.11 I-O Statements

<i-o statement>::= PRINT INTEGER{<integer variable>|<signed integer>}, <integer>|PRINT REAL{<real variable>|<real>|<sign> <real>},<integer>,<integer>|READ REAL<real variable>| READ INTEGER<integer variable>

The READ INTEGER statement reads the next integer number in the input stream into the integer storage location specified by the operand. Blanks are ignored and a number is terminated by a comma or the end of a data card, or both. An error arises if a comma is encountered before a digit is found. Any character other than a blank, digit, sign or comma encountered is invalid. The contents of the accumulator are unaltered.

The READ REAL statement reads the next number in the input stream into the real storage location specified by the operand. Blanks are ignored and a number is terminated by a comma or the end of a data card, or both. An error arises if a comma is encountered before a digit is found. Any character other than a blank, digit, sign, decimal point or comma encountered is invalid. The number does not have to contain a decimal point. The contents of the accumulator are unaltered.

The PRINT INTEGER statement prints either the contents of the integer storage location or the integer number specified by the first operand. The second operand, an integer in the range [2,16], specifies the number of spaces allowed to print these contents. If the second operand is omitted or if the stated value allows too few spaces to print the first operand, a value of 11 is assumed. A space must be allowed for the sign which is printed as a blank for positive operands.

The PRINT REAL statement prints either the contents of the real storage location or the real number specified by the first operand. The last printed digit is rounded up before printing if the first unprinted digit is a 5 or above. The second and third operands, integers in the range [2,16]and [0,16]respectively, specify the number of spaces before and after the decimal point allowed to print these contents. If the second and third operands are omitted, values of 11 and 3 are assumed respectively. A space must be allowed for the sign which is printed as a blank for positive operands. The decimal point is always printed.

2.12 Printer statements

<printer statement> ::= NEW PAGE|NEW LINE<integer>|SPACE<integer>|
PRINT TEXT 'any character string not containing quote'

The NEW PAGE statement causes the printer to skip to a new page.

The NEW LINE statement causes the printer to start at the beginning of a line after advancing n lines where n is the operand which is an integer in the range [1,4] and if omitted or outside that range is assumed to be 1.

The SPACE statement causes the printer to skip n spaces where n is the operand which is an integer in the range [1,100] and if omitted or outside that range is assumed to be 1.

The PRINT TEXT statement prints the character string enclosed in quotes (') which forms the operand.

2.13 Loops

```
<cycle
   statement>   ::=   CYCLE{<integer>|<simple integer variable>}TIMES

<for parameter>::=   <simple integer variable>|<signed integer>

<for statement>::=   FOR<simple integer variable>=<for parameter>(
                     <for parameter>)<for parameter>

<repeat>        ::=  REPEAT|<integer>:REPEAT

<loop >         ::=  {<cycle statement>|<for statement>}<statement
                     group><repeat>
```

The CYCLE REPEAT group of statements causes the group of statements enclosed between the CYCLE statement and the closest following REPEAT

statement to be repeatedly executed the number of times specified by the operand. The operand must be an integer or a simple integer variable (i.e. unsubscripted) and must have, at the first execution of the CYCLE statement, a value in the range [0,100]. This value specifies the number of repetitions which is independent of any change in the value of the operand inside the CYCLE REPEAT group. A value of zero means no repetitions. If the value is outside the allowable range, a value of one is substituted and an error message is given.

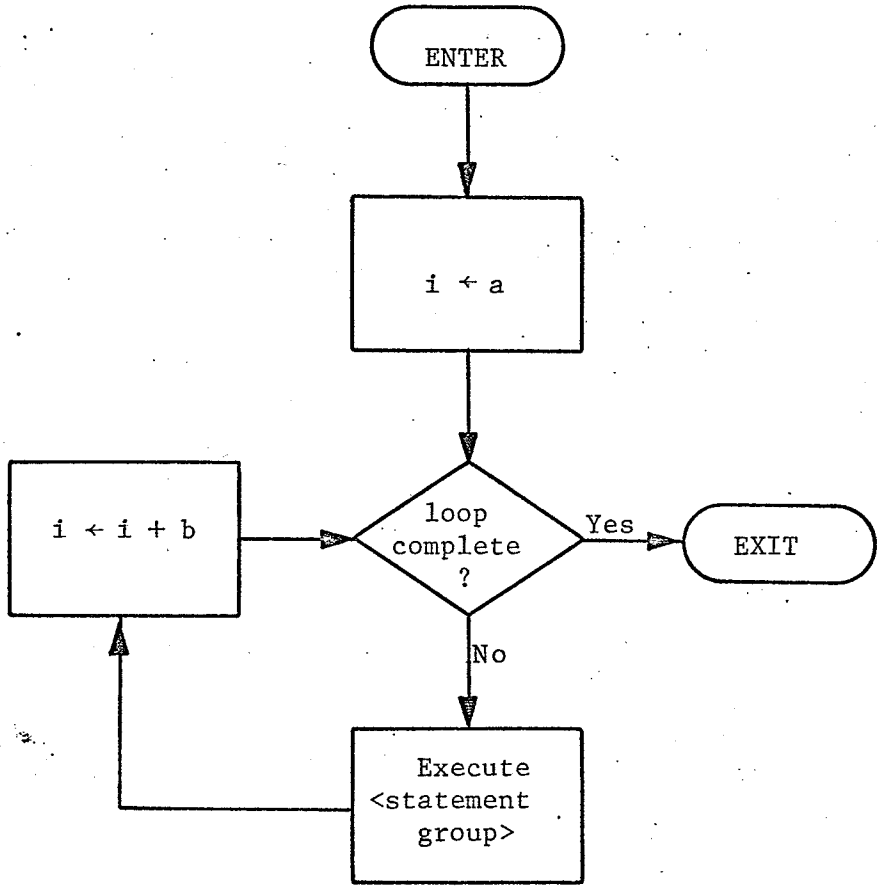
The FOR REPEAT group of statements causes the group of statements enclosed between the FOR statement and the REPEAT statement to be repeatedly executed zero or more times while the controlled variable is being changed.

Consider :

```
FOR i = a(b)c
statement group
REPEAT
```

where i is a simple integer variable and a,b,c are simple integer variables or integer numbers.

The process may be visualised by the following flow-chart :



The loop is complete if (i - c) * sign (b) > 0 where sign (b) has a value of +1 if b is positive, 0 if b is zero and -1 if b is negative.

All the FOR parameters may be varied inside the statement group.

Nesting of CYCLE REPEATS and FOR REPEATS is allowed to a depth of 10.

CHAPTER III

Description of the Control Program and the Compiler

3.1 Introduction

The purpose of the control program and the compiler, from hence forth called the compiler, is to batch process small jobs with high throughput. The jobs may not refer to peripherals apart from the card reader and the line printer. The compile phase is one pass and the compiler is resident in core at all times. Hence the job stream may be processed without reference to any secondary storage.

An assembly level language (IBM System 360 Assembler [13]) was used to write the compiler with the intention of making the object code both compact and efficient. The compiler program consists of 28 Assembler subroutines and occupies about 7500 32-bit words (or 30,000 bytes) of core.

Logically the compiler may be split up into the following parts :

Control program

Analysis of source program

Code generation

Input - Output routines

Execution time routines

Error handling

Program information dump

and will be discussed under these sub-headings.

3.2 The Control Program

The control program (see figure 3.1) initialises the job stream and then initialises the compilation phase of the first job and calls the analyser routine. On return the control program initialises the execution phase and executes the compiled code. When each job terminates, the control program repeats the process, starting with the initialisation of the compilation phase of the next job.

The control program cancels a job if its combined compilation and execution time exceeds 15 seconds, if it prints more than 300 lines of output (execution of a NEW PAGE instruction counts as 10 lines), executes more than 20,000 statements, or has more than 20 errors. These cancellation parameters may be overridden by the options card.

When the end of the card file is sensed the control program ends the processing of the job stream.

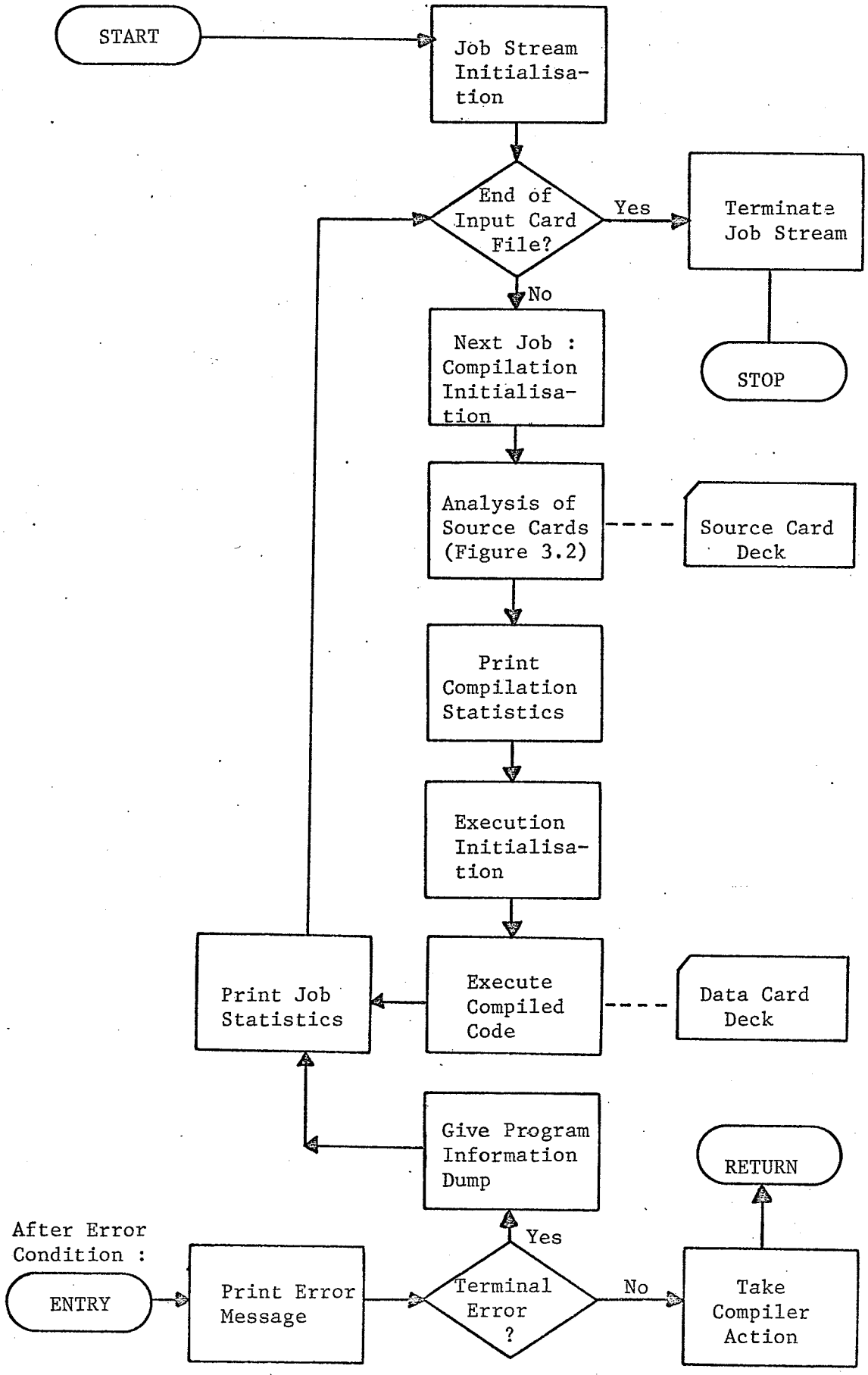
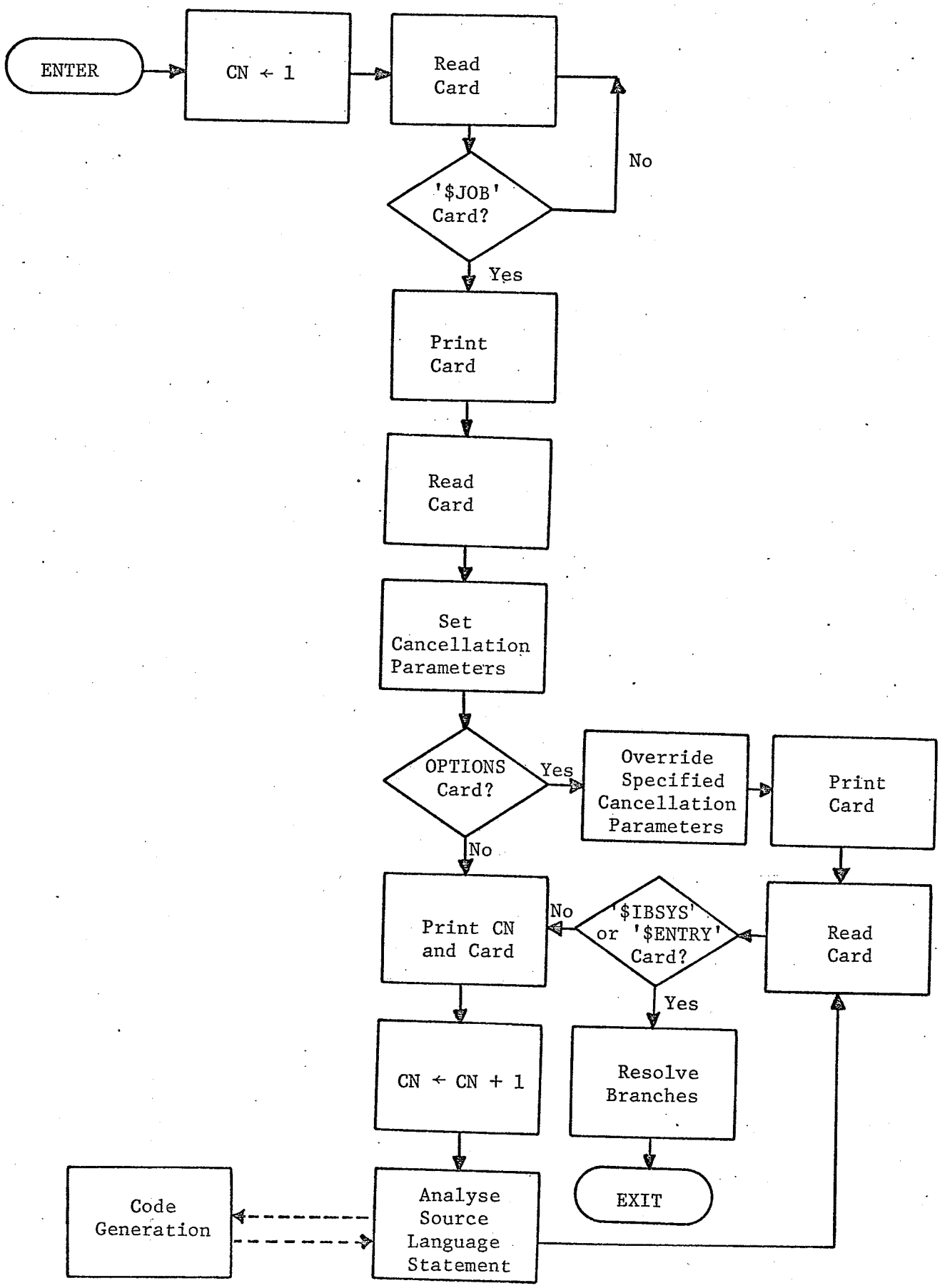


Figure 3.1 Compiler Flowchart



Where CN is the card number

Figure 3.2 Analysis of source cards

3.2.1 The Options Card

To override any or all of the cancellation parameters, an options card is inserted at the beginning of the source deck, immediately following the '$JOB' card. The first 9 spaces of this card contain the characters 'OPTIONS = (', followed by up to four parameters separated by commas and ended with a closing bracket. The parameters are : first, the job time in seconds; second, the number of lines of output printed by the executing job; third, the instruction count in multiples of 1000; and fourth, the error count. Parameters that are missing or have a value of zero are ignored.

For example :

```
OPTIONS = (15,300,20,20)
```

specifies the standard cancellation parameters and

```
OPTIONS = (0,,30)
```

specifies :

```
job time      = 15 seconds
output        = 300 lines
instructions  = 30,000
and error count = 20
```

since only the third parameter is changed.

3.3 Analysis of Source Program

3.3.1 Job Control

Jobs are divided in the job stream by WATFOR [1] control cards. A job is preceded by a card with the characters '$JOB' in the first four columns. A job is terminated by a '$IBSYS' card and if data is used, the source program and data are separated by a '$ENTRY' card.

3.3.2 The Analyser Routine

The analyser initially searches for the '$JOB' card, which is printed. If the next card is an options card, the specified parameters are overridden. Each source card is printed, preceded by a sequence number, known as the card number. One card is analysed at a time being scanned once from left to right with blanks and any characters enclosed between a left angle bracket (<) and a right angle bracket (>) being ignored. (These character strings are called "comments" and are used to document the program.)

Each card is scanned for a keyword and usually at least one operand. When a letter is detected it is added to a character string which, if a keyword has not yet been found, is then compared to all keywords of the same length. Error conditions occur when a card (unless it is blank or contains only a comment) does not have a keyword, or the

first keyword in a program is not 'BEGIN PROGRAM'.

If applicable, on successful analysis of a card, the coding routine is called. The normal parameters are the opcode, the operand, the address and the operand type. If the operand is subscripted, the parameters are as before plus two subscripts and two dimensions. Vector references are treated as matrix references with the second dimension and the value of the subscript being one.

Information about statement labels is stored in a set of tables with space for 100 entries and information about branch targets is stored in a second set of the same size. An error message is printed if the tables overflow and the compilation is terminated.

At the end of the source deck (after a '$IBSYS' or '$ENTRY' card has been found), the preceding keyword must have been 'END PROGRAM' otherwise an error message is printed and 'END PROGRAM' coded.

The branch targets and the statement labels are matched and the branch coding updated.

Finally, the compilation time, the object program size, and the number of scalars and numbers used in the compilation are calculated and printed.

3.3.2.1 Hashing and Identifier Tables

Scalar Numbers

The type and location of the storage element assigned to a scalar variable can be found by referencing four tables: the name table, where the identifier is stored; the type table; the address table; and the pointer table which points to the next free position in the overflow part of the tables.

In pseudo ALGOL, the building of the tables (initiated by a declaration statement) is as follows :

```
hash  :=  remainder (identifier/127);
loop  :   if  ptab  (hash) ≠  0
          then begin  hash  :=  ptab  (hash);
                      go to loop
               end;
          name tab  (hash)  :=  identifier;
          ptab  (hash)      :=  next;
          next              :=  next + 1;
          add tab  (hash)   :=  next add;
          next add          :=  next add + 1;
          type tab  (hash)  :=  type
```

where :

remainder is a pseudo function that returns the remainder when

the identifier, treated as a bit pattern, is divided by 127;

hash is the hashing index;

ptab is the pointer table;

name tab is the name table where the identifiers are stored;

add tab is the address table;

type tab is the type table;

next is the next free position in the overflow part of the tables;

and next add is the address of the next free element of storage.

The method of finding the address and type of a referenced identifier is, in pseudo ALGOL :

```
hash    :=  remainder (identifier/127);
loop    :   if identifier ≠ name tab  (hash)  then
                begin if ptab  (hash)  =  0  then go to error  end;
                    else begin hash  :=  ptab  (hash);
                                go to loop
                         end;
address :=  add tab  (hash);
type    :=  type tab  (hash);
```

The size of the tables allows for 192 entries, 127 for first entries and 65 for overflow entries. Therefore with this implementation it is possible to overflow the identifier tables if more than 65 scalars and numbers are used. If this happens compilation terminates with an error message. (N.B. Considerable storage can be saved by a

slightly more involved hashing algorithm, allowing the identifiers to be stored sequentially in the name table. Although storage conservation is in principle important, in this first implementation it was not considered critical.)

At the beginning of the compilation phase of each job, all of the pointer table is set to zero and the type table is set to 'Q', meaning unspecified type.

Arrays

The type, absolute address, and dimensions of vectors and matrices may be found by referencing five tables; the name, address, row, column and type tables. The size of the tables allows for 32 entries after which compilation terminates with an error message.

When an array is referenced a sequential search is employed to find its name in the name table since, in view of the anticipated small number of arrays used in each program, this is quicker than a hashing technique.

3.3.2.2 Storage of Variables, Numbers and Text

All storage elements are of word size.

Scalars

Scalars are stored together in an area called SCALSTOR, which is initialised throughout to hexadecimal '5050...'. At execution time a dedicated general purpose register points to SCALSTOR so that in the compiled code the address of scalars can be given in base-displacement form.

Numbers

Numbers are treated as scalars with their name, which is twelve blanks followed by their value, stored in the same identifier tables. The assigned storage element is initialised to their value.

Vectors and Matrices

Vectors are treated as single column matrices. On declaration the required storage area is obtained from main storage by the GETMAIN macro. All array storage is initialised to hexadecimal '5050...'. The absolute address and the dimensions are stored in the compiled code when a subscripted statement is compiled.

At the end of a job, all storage assigned to arrays is freed by use of the FREEMAIN macro.

Text

The operands of the PRINT TEXT instructions are stored

sequentially in an area of size 512 characters. If this is exceeded, an error message is printed and further text is not stored.

3.4 Code Generation

After successful analysis of a card containing an executable statement, the corresponding code is generated and added to the compiled code. If the card analysis failed, dummy code is added which on execution prints the message "THE STATEMENT ON CARD NUMBER ... HAS BEEN DELETED BY THE COMPILER".

If the analysed card contained a CYCLE or FOR keyword information is stored in a push down stack so that the following REPEAT may complete the coding belonging to that loop by accessing the information stored most recently.

If the compiled code exceeds the maximum allowed size, an error message is printed. Compilation continues but the location counter is reset so that new code overwrites the earlier generated code.

The compiled machine instructions are of three types : those that reference the accumulator; those that branch and those that link to execution time or I-O routines.

The accumulator referred to in the language definition is in fact

two registers, a floating point (FPR 4) for real numbers and a general purpose (GPR 10) for integer numbers. At all times one of the pair is flagged as unused by containing hexadecimal '50505050'. At any time the accumulator is defined by which register is unused.

As well as FPR 4 and GPR 10, general purpose registers 9, 12 and 13 are dedicated throughout execution. GPR 9 points to an area called 'SCALSTOR', where all scalars, numbers and address constants are stored. Hence these may be addressed in the compiled code in base-displacement form. The card numbers of the most recently executed forty statements are stored in an area called CARDNUMS. GPR 12 points to the last entry in CARDNUMS (i.e. the card number of the most recently executed statement). GPR 13 is the base register and the register save area pointer. Branches are coded in RX format, using the base register and a displacement.

The method of linking is to load GPR 15 with the address of the required entry point (stored in 'SCALSTOR') and to branch to that address linking GPR 14 to the following piece of code, normally a parameter.

3.4.1 The Basic Instruction

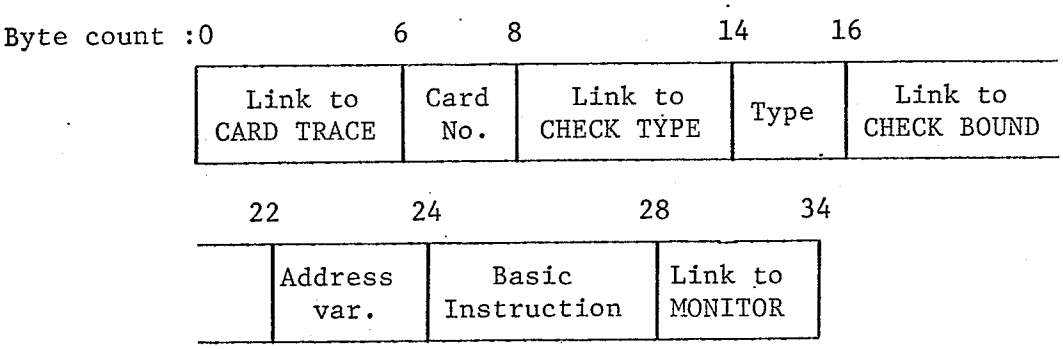
The basic instruction is a System 360 Assembler RX format instruction, using a particular general purpose register or floating point register.

For example, the basic fixed point ADD instruction is in hexadecimal representation of machine code :

5A A0 90 60

where '5A' is the opcode, 'A' specifies general purpose register 10, '0' specifies no index register and '9060' is the base and displacement (S type of address) of the operand.

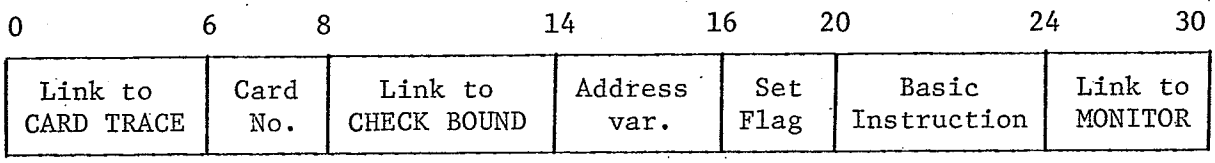
3.4.2 ADD, SUBTRACT, MULTIPLY, DIVIDE instructions



All the links are $1\frac{1}{2}$ words (or 6 bytes) long and the basic instruction is one word long. Since all 360 machine code instructions must be aligned on half-word boundaries, usually the parameters are half-words.

In the fixed point MULTIPLY and DIVIDE instructions, since the 360 machine code instructions use a register pair, two shift instructions are coded effectively making the basic instruction three words long.

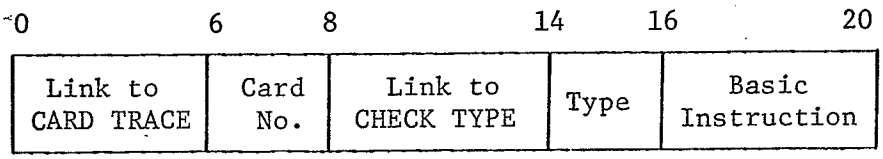
3.4.3 LOAD instruction



The set flag word in the LOAD instruction flags the unreferenced register, in the general purpose and floating point register pair, as unused.

There is no link to the CHECK TYPE routine.

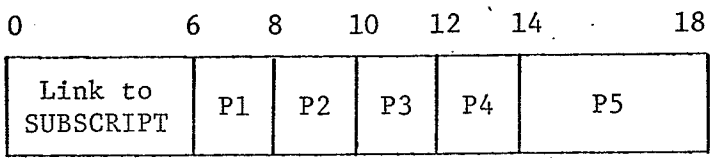
3.4.4 STORE instruction



There are no links to the CHECK BOUND or MONITOR routines.

3.4.5 Subscripted Instructions (ADD, SUBTRACT, MULTIPLY, DIVIDE, LOAD, STORE)

These instructions have this coding inserted immediately before the Basic Instruction :



where : P1 is the upper limit of the first subscript,

P3 is the upper limit of the second subscript,

P2 is the address of the first subscript,

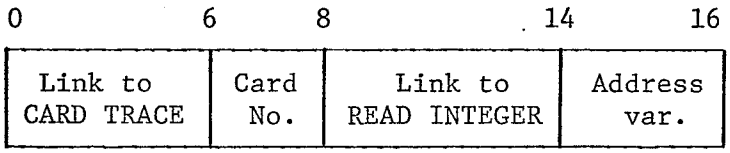
P4 is the address of the second subscript,

P5 is the address of the first element of the array.

If the array is singly subscripted, P3 and P4 are dummy parameters.

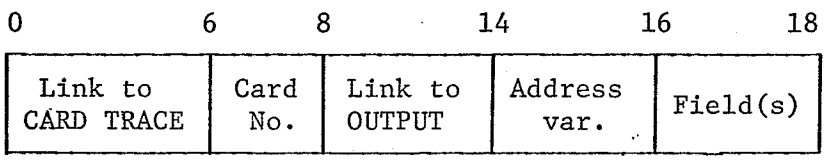
3.4.6 Input - Output instructions

Unsubscripted READ :



In the case of the READ REAL instruction, the link is to the entry point, READ REAL.

Unsubscripted PRINT :



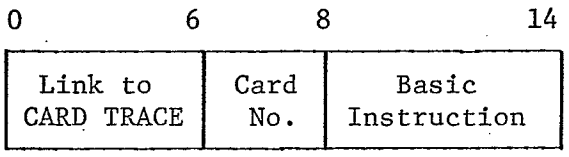
In the PRINT REAL instruction, there are two field parameters (spaces before and spaces after decimal point) which each take one byte in the 'fields' parameter.

If the instruction is subscripted the code specified in 'Subscripted

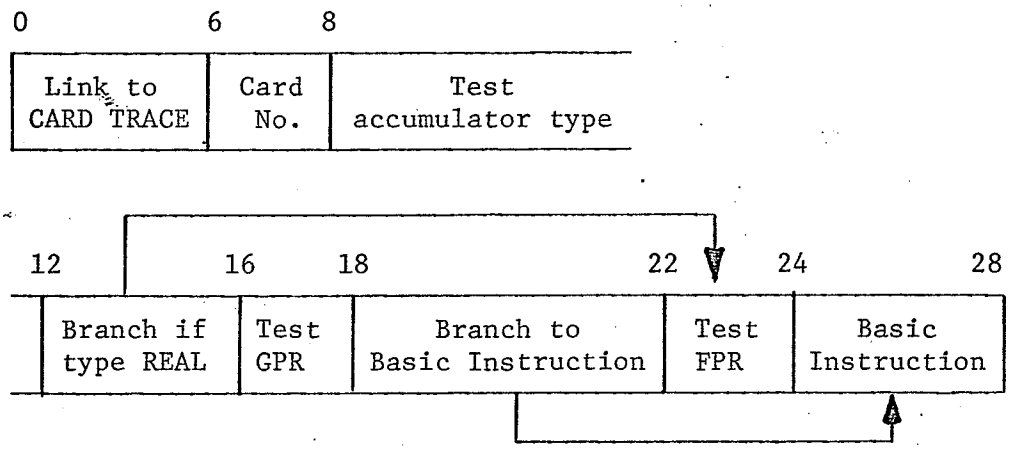
instructions' is inserted before the link to the I-O routine.

3.4.7 Branch Instructions

Unconditional branch :



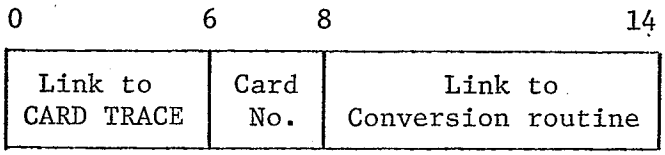
Conditional branch :



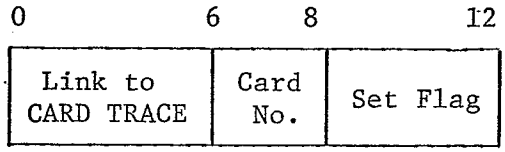
The accumulator type must be tested at execution time since it is not known at compile time, and the sign of the corresponding register (General Purpose or Floating Point) then found.

The basic instruction contains the condition for branching.

3.4.8 Conversion Instructions (CONVERT TO REAL, CONVERT TO INTEGER)

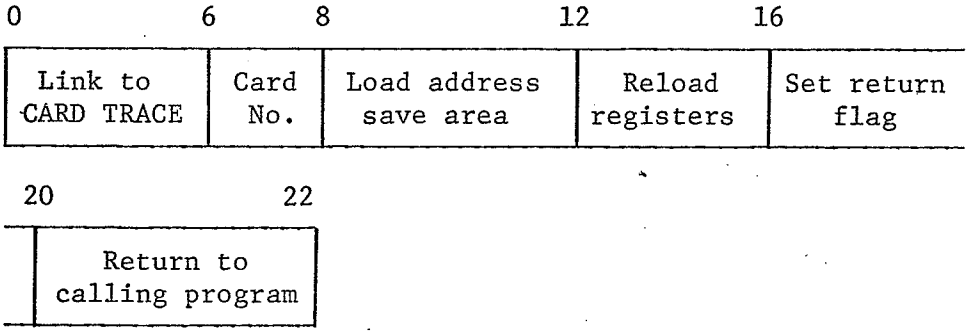


3.4.9 TRACE and MONITOR instructions



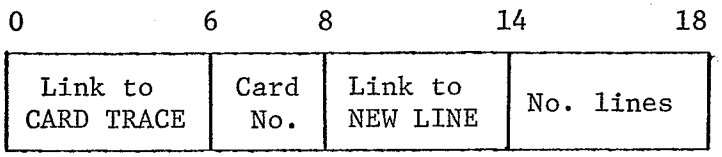
The 'Set Flag' is a Move Immediate instruction.

3.4.10 STOP and END PROGRAM instructions



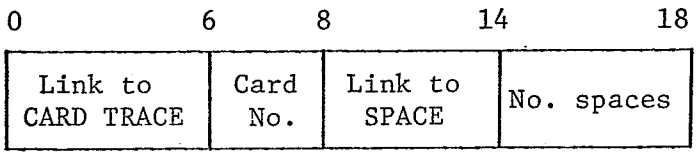
Bytes 8 - 22 are the same as the Assembler RETURN macro.

3.4.11 NEW LINE and NEW PAGE instructions

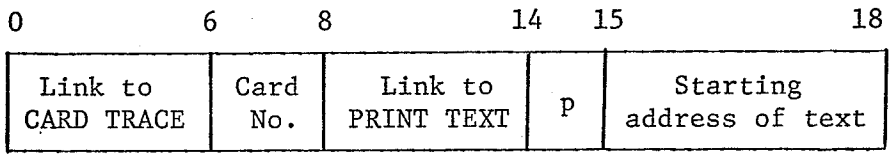


For the NEW PAGE instruction, a link is made to 'NEW LINE' but the number of lines is set to -1.

3.4.12 SPACE instruction



3.4.13 PRINT TEXT instruction



where P is the length of the text in bytes.

3.4.14 CYCLE REPEAT instructions

The code generated by the CYCLE instruction is most easily represented by the following System 360 Assembler instructions :

```
        L     Q,CV              CV = CYCLE VARIABLE
        LA    Q, 1(Q)           ADD 1 TO REGISTER Q
        LTR   Q,Q               TEST SIGN OF REGISTER Q
        BH    OK1               BRANCHES IF POSITIVE
ERR     L     15,=V(ERRMSG)     LINK TO ERROR ROUTINE
        BALR  14,15
        DC    H'120'            ERROR PARAMETER
        LA    Q,2               DUMMY 'CYCLE 1 TIMES'
OK1     C     Q,=F'100'         COMPARE CYCLE VARIABLE TO 100
        BH    ERR               TOO HIGH
        B     TEST              'TEST' IS IN THE REPEAT CODING
LOOP    ST    Q,AV              AV = ACTUAL VARIABLE
```

The CYCLE VARIABLE is the variable referred to in the CYCLE instruction, e.g. CYCLE variable TIMES. The ACTUAL VARIABLE initially has the value of the CYCLE VARIABLE, and is decreased by one each time the loop is executed.

The matching REPEAT instruction is represented by the following System 360 Assembler instructions :

```
        L     Q,AV              AV = ACTUAL VARIABLE
        LTR   Q,Q               TEST SIGN
        BH    TEST              ERROR IF NOT POSITIVE
        L     15,=V(ERRMSG)     ILLEGAL TRANSFER INTO CYCLE-REPEAT GROUP
        BALR  14,15             LINK TO ERROR ROUTINE
```

```
        DC    H'156'          ERROR PARAMETER
TEST    BCT   Q,LOOP          DECREMENT R.Q, IF POSITIVE LOOP
        ST    Q,AV            LOOP COMPLETED
```

3.4.15 FOR-REPEAT instructions

The code generated by the FOR instruction is represented by the following System 360 Assembler instructions :

```
        L     Q,START         'START' IS THE STARTING PARAMETER
        LA    P,TEST          'TEST' IS IN THE REPEAT CODING
        BR    P               BRANCH TO TEST
LOOP    ST    Q,AV            AV = ACTUAL VARIABLE
```

The code generated by the following REPEAT is represented by these System 360 Assembler instructions :

```
        L     Q,AV            AV = ACTUAL VARIABLE
        A     Q,STEP          ADD ON STEP
TEST    L     S,STEP
        LTR   S,S             TEST SIGN OF STEP
        BM    L1              BRANCH IF NEGATIVE TO L1
        OI    BC+1,x'DO'      INSERT 'BNH'
        B     L2
L1      OI    BC+1,x'BO'      INSERT 'BNL'
L2      C     Q,UNTIL         COMPARE R.Q. TO 'UNTIL'
BC      BC    O,LOOP          LOOP IF INSERTED CONDITION SATISFIED
```

```
S     Q,STEP          SUBTRACT STEP, LOOP COMPLETED
ST    Q,AV            ACTUAL VARIABLE HAS LAST USED VALUE
```

The variable, ACTUAL VARIABLE, START, STEP, UNTIL, refer to the parameters in the FOR instruction in the following way :

FOR actual variable = start, step, until

3.5 Input - Output Routines

Input is from cards and output is on the line printer. Both are stream except that on input a number is terminated by the end of a card. A line on the printer is 132 characters long.

3.5.1 Input Routines

Basically the input routines read integers and perform conversions. They are used at compile time and execution time by the READ INTEGER and READ REAL instructions.

The input buffer is examined and a decimal integer built up from the digits. If the end of the buffer is encountered before a digit is found the next card is read into the buffer and the scan continued. If a decimal point is found, the integer terminated by it is converted to floating point and stored, and the fractional part is treated as an integer with a scale factor. When the number terminates the fraction

part is converted to floating point and the two parts added.

3.5.2 Output Routines

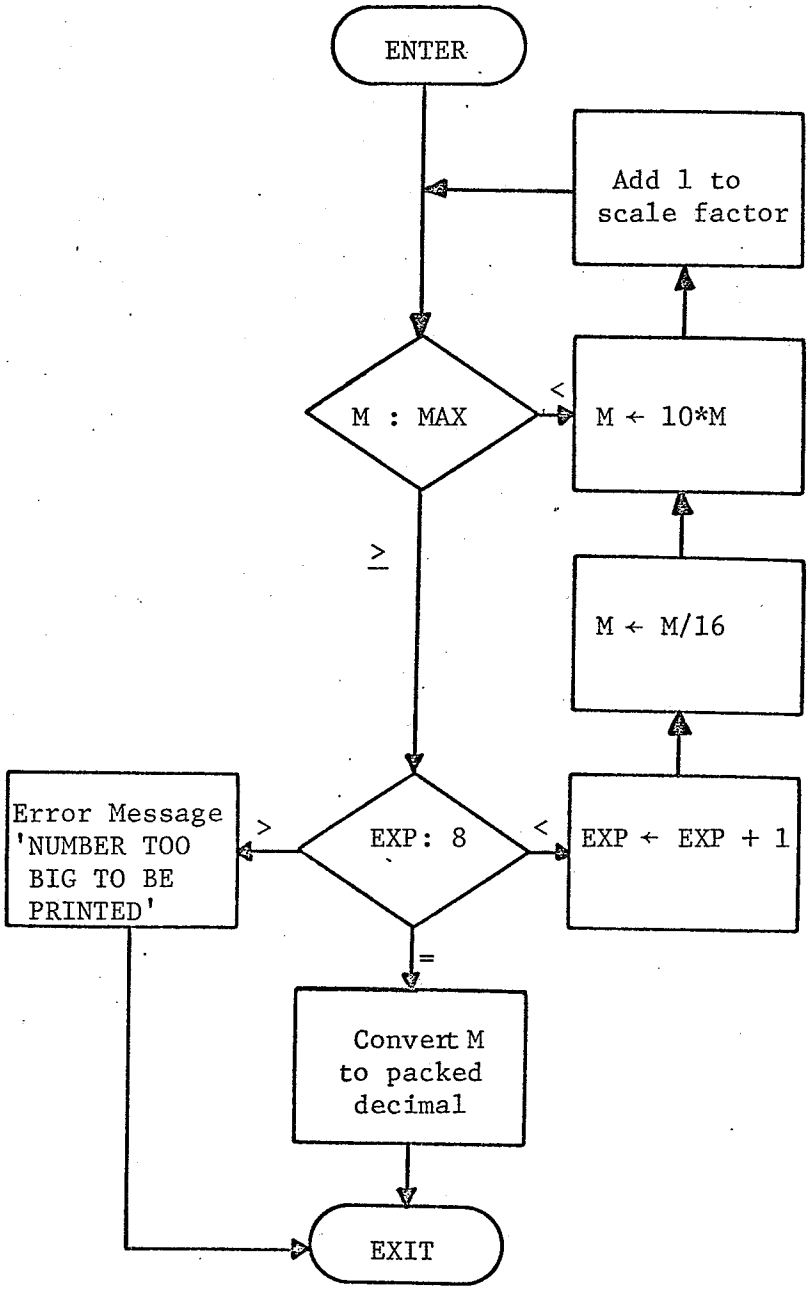
The output routines are used at execution time by the PRINT INTEGER, PRINT REAL, NEW LINE, NEW PAGE, SPACE and PRINT TEXT instructions.

The output buffer is 231 characters long. (The longest field length which can be specified by the operand in an output instruction is 100 characters in a "SPACE 100" statement and this could commence at the 132nd character position.) Whenever the output buffer pointer exceeds 131, the first 132 characters are printed and the next 99 left adjusted and the next 33 are blanked.

The PRINT INTEGER instruction requires a binary to packed decimal (internal IBM System 360 decimal integer representation) conversion and the PRINT REAL instruction requires a floating point to packed decimal conversion (See Figure 3.3). Both instructions create edit patterns of the specified format and edit the packed decimal into the output buffer.

The SPACE n instruction causes the output buffer pointer to be advanced n spaces.

The NEW LINE and NEW PAGE instructions cause the output buffer to be printed and cleared, resetting the buffer pointer to the beginning



Where :

M initially contains the hexadecimal mantissa;

EXP initially contains the hexadecimal exponent;

MAX is the maximum number that can be multiplied by 10 without causing an overflow interrupt;

and the scale factor specifies where the decimal point is to be considered.

Figure 3.3 Floating point to packed decimal conversion

of the buffer. If NEW LINE n is specified with n > 1 or if NEW PAGE is requested, the BSAM 'CNTRL' macro is simulated with the corresponding parameter.

The PRINT TEXT instruction causes the text to be copied from the text storage area into the output buffer.

3.6 Execution Time Routines

3.6.1 The Card Trace Routine

This routine records the card number of every statement as it is executed in an array which stores the last forty executed statements. If the trace has been requested (by a TRACE ON statement), this routine prints the trace.

3.6.2 The Check Bound Routine

The check bound routine checks whether the storage location specified by the parameter has been assigned a value. All variables are initialised at compile time to a particular value (hexadecimal '50505050'*) which has to be altered by either a READ or STORE instruction before being used in any other instruction.

---

* This value has no significance and of course could arise legally in the execution of a program.

3.6.3 Check Type Routine

The check type routine checks the type of the operand against the type of the accumulator.

3.6.4 Subscript Routine

The subscript routine checks the subscript range(s) and returns the referenced element's address in a general purpose register.

3.6.5 Monitor Routine

The monitor routine prints the contents and the type of the accumulator if there has been a monitoring request (by a MONITOR ON statement).

3.6.6. Convert to Integer Routine

The convert to integer routine is entered when a CONVERT TO INTEGER statement is executed. The real number in the floating point register is rounded to the nearest integer and loaded into the general purpose register. The accumulator type is changed to integer by flagging the floating point register as unused.

If the number is not within the range that can be converted to an integer ($-2^{31} \leq n < 2^{31}$), an error message is printed.

3.6.7 Convert to Real

The convert to real routine is entered when a CONVERT TO REAL statement is executed. The integer number in the general purpose register is converted to a real number and loaded into the floating point register. The accumulator type is changed to real by flagging the general purpose register as unused.

3.7 Error Handling

The philosophy of this language is to attempt to flag all errors, and to print an error message stating what the error is and what action the compiler has taken. Where it is feasible, this action is to make such assumptions as are necessary to continue the user's program. Otherwise the action is to terminate the program.

3.7.1 Hardware Interrupts

In the control program there is a trap to monitor hardware interrupts in the form of a SPIE macro (Specify Program Interrupt Exit). Since no specific error message can be given for most of these interrupts, the error is termed as catastrophic. The exceptions are divide by zero and overflows and underflows.

3.7.2 Compile Time Errors

These fall into four classes, of which the first three give specific error messages :

a) Minor : Most of the information about the statement has been successfully analysed. An assumption is made, normally to ignore further information, and the statement is coded.

b) Major : The statement cannot be successfully analysed. Dummy code is inserted in the compiled code in place of the statement. On execution of this dummy code, the message : THE STATEMENT ON CARD NUMBER nnn HAS BEEN DELETED BY THE COMPILER is printed. (nnn is the card number of the deleted statement).

c) Terminal : After certain errors (e.g. an unresolved branch), it is not considered worthwhile to execute the compiled program. The program is terminated at the end of compilation.

d) Catastrophic : The compiler is unable to continue compiling the program. Compilation is terminated and a partial program information dump given. The program is flushed and the next program in the job stream started.

3.7.3 Execution Time Errors

These fall into three classes, those in the first two giving specific error messages :

a) Non terminal : The error message gives the card number of the statement whose execution caused the error. Some assumption is made and execution of the program continued.

b) Terminal : Execution of the program beyond this point is not possible or not considered useful. The program is terminated, a program information dump given, and the program flushed.

c) Catastrophic : The program terminates, a program information dump is given and the program flushed.

3.8 Program Information Dump

A program information dump occurs after three types of condition :

a terminal or catastrophic execution time error;

a catastrophic compilation time error or;

the execution of a DUMP ALL statement.

The dump gives :

the card number of the statement causing the dump;

the reason for the dump, i.e. execution or compilation error, or DUMP ALL statement;

the contents and type of the accumulator;

after an execution error or DUMP ALL statement a partial card number trace is printed;

the names, values and types of all scalars;

the names, values, types and dimensions of vectors and matrices.

If a scalar or an element of an array has not been assigned a value, the 'value' printed is "**UNUSED**".

For an actual example of the Program Information Dump see Appendix 4.

3.8.1 The Partial Card Number Trace

In the program information dump the card numbers of the forty most recently executed statements are printed, in the order in which the statements were executed. Initally the relevant storage area is filled with zeroes and hence if the program has executed fewer than forty statements, the earliest card numbers printed are zero.

CHAPTER IV

Further Developments, Compilation Time Statistics and Conclusions

The following extensions to the IPLAN language are to be implemented.

4.1 New Classes of Variables and Constants

The new classes BINARY and CHARACTER will be introduced. These classes will not involve the declaration of new types, but will involve a different interpretation of variables declared as either REAL or INTEGER.

4.1.1 BINARY constants

A BINARY constant is a 32-bit string of 1's and 0's enclosed in $ signs. If there are fewer than 32 bits (i.e. 1's and 0's) the string is right justified and padded on the left with 0's. If there are more than 32 bits, an error message is given and the rightmost 32 accepted.

Any character between the bracketing $ signs other than 0,1 or blank is invalid.

4.1.2 CHARACTER constants

A CHARACTER constant is a single character enclosed in quotes ('). For example 'A', '1', ' ', or '''. Each CHARACTER constant is right justified in a 32-bit word with zeroes on the left. The internal representation of characters in storage is shown in Appendix 7.

4.1.3 BINARY Input - Output

The READ BINARY statement reads into the specified storage location the next string of 1's and 0's, which is terminated by a comma or the end of a data card, or both. The same length and validity conditions apply as in paragraph 4.1.1 on BINARY constants.

The PRINT BINARY statement prints all 32 bits from the specified location, on the line printer.

4.1.4 CHARACTER Input - Output

The READ CHARACTER statement reads the next character in the input stream into the bottom (rightmost) 8 bits of the specified storage location. The top (leftmost) 24 bits are set to zero.

The PRINT CHARACTER statement prints the bottom (rightmost) 8 bits of the specified storage location as a character on the line printer. If the 8 bits do not represent a character then an invalid character

symbol, a zero overprinted by a plus sign, is printed.

4.1.5 Logical Operations

The logical keywords "AND", "OR", "NOT", "SHIFT LEFT", "SHIFT RIGHT" will be introduced.

AND and OR

The contents of the accumulator after the operation is the Boolean union or intersection of the previous contents and the operand, with both accumulator and operand being regarded as 32-bit binary strings. The accumulator type is unchanged.

NOT

There is no operand for this keyword.

The contents of the accumulator after the operation are the previous contents with '0' bits replaced by '1' bits and '1' bits replaced by '0' bits. The accumulator type is unchanged.

SHIFT LEFT and SHIFT RIGHT

The contents of the accumulator after the operation are the previous contents treated as a 32-bit binary string logically shifted left or right the number of bits specified by the operand. Bits are dropped off at the left (right) end of the accumulator and zeroes introduced

at the right (left) end for a left (right) shift. If the operand has a value $\geq$ 32 the result is zero. If the operand has a value < 0 an error message is printed and no shifting performed.

4.2 Dynamic Transfer of Control

The Backus Normal Form definition of <simple statement> (see Appendix 1) will be extended to include :

<simple statement> ::= <transfer opcode><simple integer variable>

e.g. the statement :

GO TO I

where I is declared as an integer scalar, will be valid.

At execution time if the value of I corresponds to a statement number then control will be transferred to that statement, otherwise an error message will be printed.

This feature will allow simple 'procedures' or 'subroutines' to be coded.

For example :

```
BEGIN PROGRAM
INTEGER SCALAR RETURN
.............
```

```
          .............
          LOAD 10
          STORE RETURN                  <SET UP RETURN ADDRESS
          GO TO 20                      <LINK TO ROUTINE
    10:   .............
          .............
          .............
          <ROUTINE
    20:   .............
          .............
          GO TO RETURN
```

4.3 Compilation Time Statistics

IPLAN was developed in a multiprogrammed environment in which the clock measures system time. Therefore it was almost impossible to obtain accurate timing statistics and the best alternative was to take the fastest time, after several runs of a job, as the true time. This was done to find the compilation time for the example program in Appendix 3. 62 statements were compiled in 0.45 seconds indicating compilation of at least 135 statements/second. The final version of the compiler should give slightly faster times since the version that was being used at the time of writing contained some unnecessary steps which had been used for debugging.

4.4 Conclusions

At the time of writing only a very limited amount of experience had been gained using IPLAN. The need for an integer version (see Appendix 6) resulted from teaching IPLAN to an adult education group where the level of programming proficiency attempted did not exceed the capabilities of this version.

IPLAN is designed for teaching and therefore must ultimately be judged on this basis. Further experience is necessary before conclusions can be drawn on whether or not the requirements of an introductory programming language as outlined in Chapter I are correct. If IPLAN does prove to be an acceptable introductory language in its present implementation then the next step in its development would be to implement it in a conversational mode.

APPENDIX 1

Backus Normal Form Definition of the IPLAN Language

Definition of the Meta-Language

The meta-character set

::= | < > * { }

The symbol ::= reads "is defined as".

The symbol | reads "or".

The symbols < > bracketing a mnemonic represent a syntactic unit (or a member of the non terminal vocabulary).

The symbol * represents the syntactic unit being defined (i.e. to the left of the ::= symbol).

The symbols { } mean select from the bracketed list one of the alternatives separated by the | symbol.

Basic symbols, numbers and variables

<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

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

<sign> ::= +|-

<identifier> ::= <letter>|*<letter>

<integer> ::= <digit>|*<digit>

<signed integer> ::= <sign><integer>|<integer>

<real> ::= <integer>.<integer>

<number> ::= <integer>|<real>|<sign>{<integer>|<real>}

<simple integer variable> ::= <identifier>

<subscript> ::= <integer>|<simple integer variable>|{<integer>|<simple integer variable>},{<integer>|<simple integer variable>}

<integer variable> ::= <simple integer variable>|<identifier>(<subscript>)

<real variable> ::= <identifier>|<identifier>(<subscript>)

<variable> ::= <integer variable>|<real variable>

<keyword> ::= <opcode>|<conversion>|<debug>|<transfer opcode>|STORE|STOP|NEW PAGE|NEW LINE|SPACE|PRINT TEXT|READ INTEGER|READ REAL|PRINT INTEGER|PRINT REAL|CYCLE|FOR|REPEAT|REAL SCALAR|INTEGER SCALAR|REAL VECTOR|INTEGER VECTOR|REAL MATRIX|INTEGER MATRIX|BEGIN PROGRAM|END PROGRAM

Statements

<opcode> ::= LOAD|ADD|SUBTRACT|MULTIPLY|DIVIDE

<conversion> ::= CONVERT TO REAL|CONVERT TO INTEGER

<debug> ::= TRACE ON|TRACE OFF|MONITOR ON|MONITOR OFF|DUMP ALL

<transfer opcode> ::= GO TO|IF POSITIVE GO TO|IF NEGATIVE GO TO|IF ZERO GO TO

<simple statement> ::= <opcode>{<variable>|<number>}|STORE<variable>|<conversion>|<transfer opcode><integer>|<debug>|STOP

<printer statement> ::= NEW PAGE|NEW LINE<integer>|SPACE<integer>|PRINT TEXT 'any character string not containing quote'

<i-o statement>::= PRINT INTEGER{<integer variable>|<signed integer>},<integer>|PRINT REAL{<real variable>|<real>|<sign><real>},<integer>,<integer>|READ REAL<real variable>|READ INTEGER<integer variable>

<cycle statement> ::= CYCLE{<integer>|<simple integer variable>}TIMES

<for parameter>::= <simple integer variable>|<signed integer>

<for statement> ::= FOR<simple integer variable>=<for parameter>(<for parameter>)<for parameter>

<repeat> ::= REPEAT|<integer>:REPEAT

<loop> ::= {<cycle statement>|<for statement>}<statement group><repeat>

<unlabelled statement> ::= <simple statement>|<printer statement>|<i-o statement>|<loop>|<declaration>|BEGIN PROGRAM|END PROGRAM

<statement> ::= <unlabelled statement>|<integer>:<unlabelled statement>

<statement group> ::= <statement>|*<statement>

Declarations

<type scalar> ::= REAL SCALAR|INTEGER SCALAR

<scalar list> ::= <identifier>|*,<identifier>

<scalar declaration>::= <type scalar><scalar list>

```
<type vector>   ::=   REAL VECTOR|INTEGER VECTOR

<vector list>   ::=   <identifier>(<integer>)|*,<identifier>(<integer>)

<vector
   declaration>::=    <type vector><vector list>

<type matrix>   ::=   REAL MATRIX|INTEGER MATRIX

<matrix list>   ::=   <identifier>(<integer>,<integer>)|*,<identifier>
                      (<integer>,<integer>)

<matrix
   declaration>::=    <type matrix><matrix list>

<declaration>   ::=   <scalar declaration>|<vector declaration>|
                      <matrix declaration>

<declaration
   group>       ::=   <declaration>|*<declaration>
```

Program

```
<program>       ::=   BEGIN PROGRAM<declaration group><statement group>
                      END PROGRAM
```

APPENDIX 2

Implementation Restrictions and Control Program Cancellations

The exceeding of any of these restrictions causes an error message to be printed.

A2.1 Size of numbers

Internal to the machine

Internal numbers an values of integer variables may not be outside the range R :

$$-2^{31} \leq R < 2^{31} \quad (2^{31} = 2,147,483,648)$$

Real numbers and values of real variables may not have a magnitude outside the range R :

$$16^{-63} < R < 16^{63} \quad (16^{63} = \text{approx } 10^{75})$$

In Source or Data Deck

A number may not have an integral magnitude > 2,147,483,647 $(=2^{31}-1)$

On Output

Any integer number can be printed.

Real numbers may not have a magnitude > 2^{32}(= 4,294,967,296).

A2.2 Source Program

Size

The compiled instructions (i.e. excluding variable and number storage) may not occupy more than 4038 bytes.

Scalars and Numbers

The number of scalars and numbers may not exceed 65.

Statement numbers and branch statements

The number of statement numbers or branch statements (i.e. GO TO's) may not exceed 100.

Arrays

The number of arrays may not exceed 32. The number of elements in any single array may not exceed 100. Array storage is not within

the compiler program and hence the total amount available depends on the environment.

Text

The total character length of the operands of the PRINT TEXT instructions may not exceed 512 characters.

A2.3 Control Program Cancellations

The control program cancels a job if it compiles and executes for more than 15 seconds, prints more than 300 lines of output (execution of a NEW PAGE instruction counts as 10 lines), executes more than 20,000 statements, or has more than 20 errors. These parameters may be overridden by the options card.

APPENDIX 3

An Example Program

The example program will sort up to 10 real numbers into ascending order. The logical steps in the program are related to the source deck cards as shown in figure A3.1.

The variables used are :

N is the number of numbers to be sorted;

LIST is the vector into which they are read an in which they are sorted;

J is a pointer to the biggest number found at a particular moment in a sorting pass through LIST and BIG has the value of that biggest number (e.g. BIG = LIST(J)) and;

I and M are indices.

The sort algorithm is, in ALGOL;

```
for M := N step -1 until 2 do
begin  BIG := LIST [1];   J := 1;
       for I = 1 step 1 until M do
       if BIG < LIST [I] then
          begin BIG := LIST [I];
                J := I;
          end;
       LIST [J] := LIST [M];
       LIST [M] := BIG
end;
```

Card numbers in Example Program

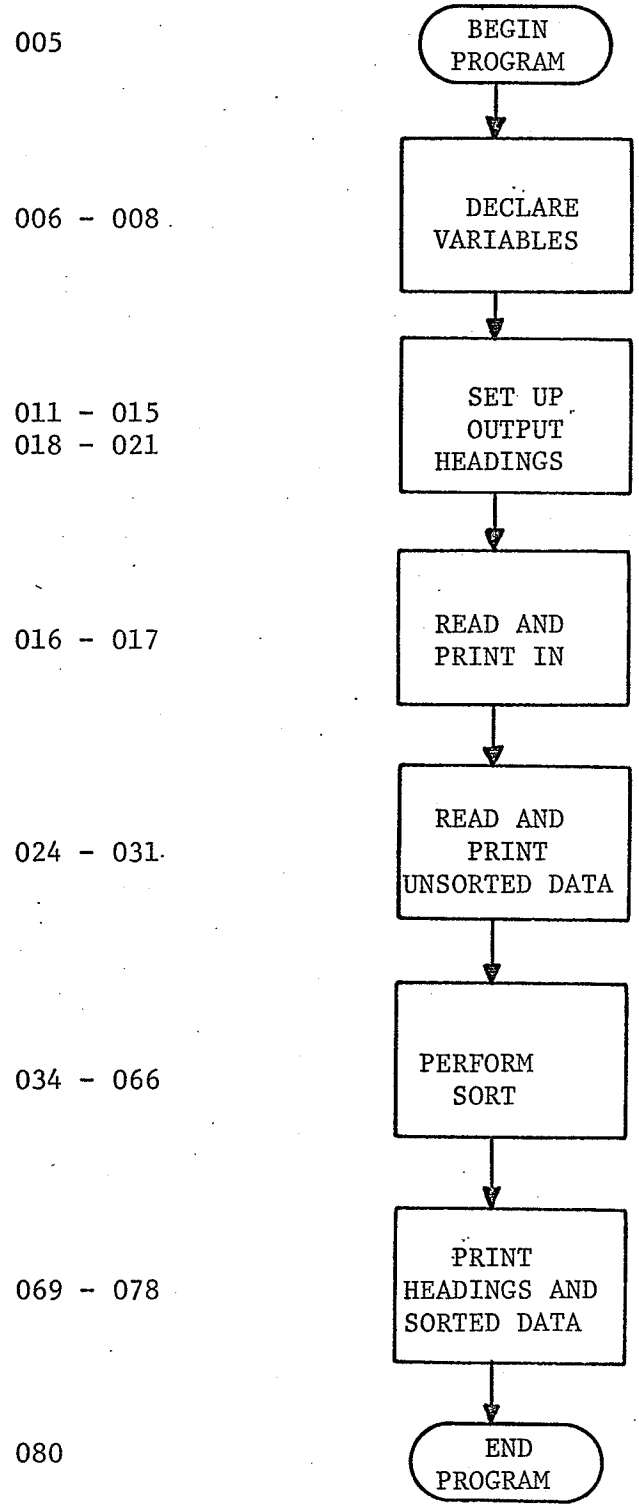


Figure A3.1 Logical Steps related to cards of Example Program

```
      $JOB

001   <                          SORT PROGRAM
002
003   <     READS IN 'N' UNSORTED REAL NUMBERS INTO 'LIST', SORTS AND PRINTS THEM
004
005   BEGIN PROGRAM
006         REAL VECTOR LIST (10)
007         REAL SCALAR  BIG
008         INTEGER SCALAR N,I,J,M
009
010   < SET UP OUTPUT HEADINGS
011      NEW PAGE
012      SPACE 40
013      PRINT TEXT  'OUTPUT FROM SORT PROGRAM'
014      NEW LINE 4
015      PRINT TEXT 'THERE ARE'
016      READ INTEGER N
017      PRINT INTEGER N,2
018      PRINT TEXT ' NUMBERS TO BE SORTED'
019      NEW LINE 2
020      PRINT TEXT  'UNSORTED ELEMENTS:'
021      NEW LINE
022
023   < READ IN AND PRINT OUT UNSORTED DATA >
024      LOAD 1
025      STORE I
026      CYCLE N TIMES
027        READ REAL LIST(I)
028        PRINT REAL LIST(I) 7,2
029        ADD 1
030        STORE I
031      REPEAT
032
033   <  THE FIRST M ELEMENTS IN LIST ARE UNSORTED
034      LOAD N
035      STORE M                          < M = SIZE OF UNSORTED LIST
036   40: LOAD 1
037       STORE J                         < J := 1 SINCE BIG = LIST(1)
038       STORE I                         < INITIALLY I := 1
039       LOAD LIST(1)
040       STORE BIG                       < BIG := LIST(1) >
041       CYCLE N TIMES
042         LOAD BIG
043         SUBTRACT LIST(I)
044         IF POSITIVE GO TO 50          < BRANCH IF BIG =< LIST(I)
045         < BIG :=  LIST(I) >
046           LOAD LIST (I)
047           STORE BIG
048           LOAD I
049           STORE J                     < J := I OF MAX LIST(I) >
050         50: LOAD I
051         ADD 1
052         STORE I                       < I := I+1 >
053       REPEAT
054
055   <  REVERSE BIGGEST AND M'TH ELEMENTS :
056       LOAD LIST(M)
057       STORE LIST(J)                   < LIST(J) IS BIGGEST ELEMENT >
058       LOAD BIG
059       STORE LIST(M)
060


061   <  UNSORTED LENGTH OF LIST NOW ONE SHALLER :
062       LOAD M
063       SUBTRACT 1
064       STORE M
065       SUBTRACT 1
066       IF POSITIVE GO TO 40            < CONTINUE SORTING IF M MORE THAN 1 >
067
068   <  PRINT SORTED LIST :
069      NEW LINE 4
070      PRINT TEXT 'ELEMENTS SORTED INTO ASCENDING ORDER:'
071      NEW LINE
072      LOAD 1
073      STORE I
074      CYCLE N TIMES
075        PRINT REAL LIST(I) 7,2
076        ADD 1
077        STORE I
078      REPEAT
079
080     END PROGRAM

      $ENTRY
TIME ELAPSED:  000.45 SECONDS    COMPILATION STATISTICS:   6  SCALARS AND NUMBER CONSTANTS USED,  OBJECT CODE =1674 BYTES
```

Figure A3.2 Listing of Example Program

```
                    OUTPUT FROM SORT PROGRAM

THERE ARE 8 NUMBERS TO BE SORTED

UNSORTED ELEMENTS:
    12.29   -31.50     4.79     9.59    -4.29     0.50     8.79    -0.09

ELEMENTS SORTED INTO ASCENDING ORDER:
   -31.50    -4.29    -0.09     0.50     4.79     8.79     9.59    12.29

TIME ELAPSED:  000.52 SECONDS          PROGRAM EXECUTED  497 STATEMENTS AND PRINTED  24 LINES OF OUTPUT
```

Figure A3.3 Output from Example Program

APPENDIX 4

Error Programs and Debug Facilities

A4.1 Example Error Program showing Program Information Dump

The example program shown in Figure A4.1 illustrates :

(i) The use of the OPTIONS card. This has set the maximum number of instructions that the program will execute to 1000. The other cancellation parameters were unchanged.

(ii) An undeclared variable on card 011. This caused two error messages and the latter prevented the statement from compiling (see (iv)).

(iii) A minor syntax error on card 014. This was accepted without comment by the compiler.

(iv) The execution of a deleted statement. Due to an error (see (ii)) the statement on card 011 was not compiled and in the statement's place in the object code, a call to a routine to produce an error message was planted.

(v) The execution of an invalid CYCLE statement. When the CYCLE statement on card 013 was executed, its cycle parameter (I) did

```
         $JOB     ERROR PROGRAM SHOWING PROGRAM INFORMATION DUMP

         OPTIONS=(,,1)                          < SETS INSTRUCTION MAX TO 1000
001
002      BEGIN PROGRAM
003      INTEGER SCALAR I, VAR
004      INTEGER VECTOR V(6)
005
006      NEW LINE 4
007      PRINT TEXT 'OUTPUT FROM ERROR PROGRAM'
008      NEW LINE 3
009      LOAD -1
010      STORE I                                < I := -1
011      STORE J
***  ERROR  *** 050 IDENTIFIER "J               " NOT DECLARED
***  ERROR  *** 029  NO/INVALID OPERAND
012      STORE V(I)
013      CYCLE I TIMES
014        PRINT INTEGER I 2                    < MISSING COMMA BEFORE 2
015      REPEAT
016      <  PUT INTO ENDLESS LOOP:
017      10: LOAD I
018          ADD 1
019          STORE I
020          GO TO 10
021      END PROGRAM

         $ENTRY
TIME ELAPSED:  000.20 SECONDS    COMPILATION STATISTICS:   4  SCALARS AND NUMBER CONSTANTS USED;  OBJECT CODE = 382 BYTES


OUTPUT FROM ERROR PROGRAM


THE STATEMENT ON CARD NUMBER 011 HAS BEEN DELETED BY THE COMPILER
***  ERROR  *** 030  ON CARD NUMBER 013  CYCLE PARAMETER <0 OR >100. 1 SUBSTITUTED
***  ERROR  *** 035  INSTRUCTION COUNT TOO HIGH. PROGRAM TERMINATED
-1
```

Figure A4.1 (part 1) Error Program showing Program Information Dump

```
********                    PROGRAM INFORMATION DUMP                    ********

THE STATEMENT ON CARD NUMBER 018 WAS BEING EXECUTED WHEN THE PROGRAM WAS TERMINATED

CONTENTS OF THE ACCUMULATOR          247                    TYPE IS INTEGER

LAST 40 EXECUTED CARD NUMBERS (EARLIEST FIRST):

019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;
019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;  019;  020;  017;  018;

 IDENTIFIER      TYPE              VALUE
I                INTEGER             246
VAR              INTEGER     ** UNUSED **

 ARRAY DUMP

ARRAY NAME          ROWS  COLS    TYPE
V                    6     1   INTEGER VECTOR
          -1         ** UNUSED **    ** UNUSED **    ** UNUSED **    ** UNUSED **    ** UNUSED **


TIME ELAPSED:  000.26 SECONDS          PROGRAM EXECUTED  1001 STATEMENTS AND PRINTED  12 LINES OF OUTPUT
```

Figure A4.1 (part 2) Error Program showing Program Information Dump

not have a valid value ($0 \leq I \leq 100$). The CYCLE loop was executed once.

(vi) The termination of the program in an endless loop. The statements on card 017 - 020 were executed until the maximum instruction count was exceeded. This caused a terminal error which provided a program information dump.

The "-1" appears after the terminal error message since although it was placed in the output buffer when the statement on card 014 was executed, the output buffer was never filled and could only be printed when the program had finished execution.

(vii) The program information dump. This gives a full data area dump and a trace of the last forty executed statements.

A4.2 Example Program showing Debug Facilities

The example program shown in figure A4.2 illustrates the use of the TRACE and MONITOR facilities.

```
          $JOB          EXAMPLE PROGRAM SHOWING DEBUG FACILITIES

001       BEGIN PROGRAM
002       INTEGER SCALAR I
003       REAL SCALAR Z
004
005          NEW LINE
006       MONITOR ON
007       TRACE ON
008       LOAD 1.5
009       CYCLE 3 TIMES
010          STORE Z
011          CONVERT TO INTEGER
012          STORE I
013          LOAD Z
014          ADD 1.5
015       REPEAT
016       MONITOR OFF
017       TRACE OFF
018          PRINT REAL Z
019          PRINT INTEGER I
020       END PROGRAM

          $ENTRY
TIME ELAPSED:  000.17 SECONDS    COMPILATION STATISTICS:  4  SCALARS AND NUMBER CONSTANTS USED,  OBJECT CODE = 364 BYTES

008  TRACE
008  ACCUMULATOR =          1.500000000000          TYPE IS REAL
009  TRACE
010  TRACE
011  TRACE
011  ACCUMULATOR =          2                       TYPE IS INTEGER
012  TRACE
013  TRACE
013  ACCUMULATOR =          1.500000000000          TYPE IS REAL
014  TRACE
014  ACCUMULATOR =          3.000000000000          TYPE IS REAL
015  TRACE
010  TRACE
011  TRACE
011  ACCUMULATOR =          3                       TYPE IS INTEGER
012  TRACE
013  TRACE
013  ACCUMULATOR =          3.000000000000          TYPE IS REAL
014  TRACE
014  ACCUMULATOR =          4.500000000000          TYPE IS REAL
015  TRACE
010  TRACE
011  TRACE
011  ACCUMULATOR =          5                       TYPE IS INTEGER
012  TRACE
013  TRACE
013  ACCUMULATOR =          4.500000000000          TYPE IS REAL
014  TRACE
014  ACCUMULATOR =          6.000000000000          TYPE IS REAL
015  TRACE
016  TRACE
017  TRACE
      4.500000          5


TIME ELAPSED:  000.30 SECONDS          PROGRAM EXECUTED    28 STATEMENTS AND PRINTED  34 LINES OF OUTPUT

        ********                              END OF JOB STREAM                              ********
```

Figure A4.2 Example Program showing Debug Facilities

APPENDIX 5

IPLAN Error Messages

The errors are listed in figure A5.1.

a) Compile Time Errors

Errors 001, 041, 046, 047 and 048 prevent execution. Errors 024 and 029 prevent the statement from compiling and cause the dummy statement coding to be planted in the statement's place in the object code. Error 050 prints the undeclared identifier.

b) Non terminal Execution Time Errors

Errors 018 and 027 occur in read operations and print the data card on which the error condition arose. Error 027 differs from error 018 by arising after a decimal point has been found in the number being read in. After both errors the input printer is advanced to the next terminator (i.e. a comma or an end of a card). Error 037 occurs when the specified field allows too few spaces to print the operand. The specified field is overridden. If the instruction is PRINT INTEGER, 11 spaces are allowed to print the operand. If the instruction is PRINT REAL the new fields are 11 and 3 (i.e. 11 places before the decimal point and 3 after).

COMPILE TIME ERRORS

```
***  ERROR  *** 001  UNDEFINED BRANCH, PROGRAM EXECUTION NOT ATTEMPTED
***  ERROR  *** 002  IDENTIFIER TRUNCATED TO 16 CHARACTERS
***  ERROR  *** 003  ILLEGAL CHARACTER, SCAN STOPPED
***  ERROR  *** 004  ILLEGAL COMMA
***  ERROR  *** 006  ILLEGAL LEFT BRACKET
***  ERROR  *** 007  ILLEGAL RIGHT BRACKET
***  ERROR  *** 008  CHARACTER STORE OVERFLOW
***  ERROR  *** 010  DIMENSION TOO LARGE
***  ERROR  *** 011  MORE 'REPEAT'S THAN 'CYCLES'
***  ERROR  *** 012  ARRAY NOT DECLARED
***  ERROR  *** 013  UNEXPECTED CHARACTER(S) FOUND. IGNORED
***  ERROR  *** 023  ILLEGAL FIELD. OVERRIDDEN
***  ERROR  *** 024  NO/INVALID KEYWORD
***  ERROR  *** 028  'BEGIN PROGRAM' NOT FIRST KEYWORD
***  ERROR  *** 029  NO/INVALID OPERAND
***  ERROR  *** 033  ILLEGAL CHARACTER IN REAL CONSTANT, SCAN STOPPED
***  ERROR  *** 034  ILLEGAL DIGIT, SCAN STOPPED
***  ERROR  *** 038  TYPE CONFLICT
***  ERROR  *** 040  INVALID OPTIONS CARD
***  ERROR  *** 041  ARRAYS DECLARED EXCEEDS 32, PROGRAM EXECUTION NOT ATTEMPTED
***  ERROR  *** 042  NUMBER TOO LARGE, 1 OR 1.0 SUBSTITUTED
***  ERROR  *** 046  TOO MUCH OBJECT CODE GENERATED, PROGRAM EXECUTION NOT ATTEMPTED
***  ERROR  *** 047  STATEMENT NUMBERS EXCEEDS 100, PROGRAM EXECUTION NOT ATTEMPTED
***  ERROR  *** 048  GO TO STATEMENTS EXCEEDS 100, PROGRAM EXECUTION NOT ATTEMPTED
***  ERROR  *** 050  IDENTIFIER NOT DECLARED
```

EXECUTION TIME ERRORS (NON-TERMINAL)

```
***  ERROR  *** 005  ON CARD NUMBER 055  REAL NUMBER TOO BIG TO BE CONVERTED TO INTEGER, 1 SUBSTITUTED
***  ERROR  *** 009  ON CARD NUMBER 055  OPERAND HAS NOT BEEN ASSIGNED A VALUE, 1 SUBSTITUTED
***  ERROR  *** 014  ON CARD NUMBER 055  SUBSCRIPT 1 TOO HIGH, 1 SUBSTITUTED
***  ERROR  *** 015  ON CARD NUMBER 055  SUBSCRIPT 2 TOO HIGH, 1 SUBSTITUTED
***  ERROR  *** 016  ON CARD NUMBER 055  SUBSCRIPT 1 TOO LOW, 1 SUBSTITUTED
***  ERROR  *** 017  ON CARD NUMBER 055  SUBSCRIPT 2 TOO LOW, 1 SUBSTITUTED
***  ERROR  *** 018  ON CARD NUMBER 055  ILLEGAL CHARACTER OR NUMBER TOO LARGE, OPERAND UNCHANGED
***  ERROR  *** 019  ON CARD NUMBER 055  REAL OPERAND, ACCUMULATOR IS INTEGER AND IS CONVERTED TO REAL
***  ERROR  *** 020  ON CARD NUMBER 055  INTEGER OPERAND, ACCUMULATOR IS REAL AND IS CONVERTED TO INTEGER
***  ERROR  *** 026  NO/MISPLACED $ENTRY CARD
***  ERROR  *** 027  ON CARD NUMBER 055  ILLEGAL CHARACTER, OPERAND HAS VALUE OF PRECEDING DIGITS
***  ERROR  *** 030  ON CARD NUMBER 055  CYCLE PARAMETER <0 OR >100. 1 SUBSTITUTED
***  ERROR  *** 031  ON CARD NUMBER 055  REAL NUMBER TOO BIG TO BE PRINTED
***  ERROR  *** 032  ON CARD NUMBER 055  NUMBER TO BE PRINTED HAS NOT BEEN ASSIGNED A VALUE
***  ERROR  *** 037  ON CARD NUMBER 055  FIELD TOO SMALL. OVERRIDDEN
```

EXECUTION TIME ERRORS (TERMINAL)

```
***  ERROR  *** 021  TIME EXCEEDED
***  ERROR  *** 025  END OF DATA
***  ERROR  *** 035  STATEMENT COUNT EXCEEDED
***  ERROR  *** 036  OUTPUT EXCEEDED
***  ERROR  *** 039  ILLEGAL TRANSFER HAS CAUSED INVALID EXECUTION OF "REPEAT"
***  ERROR  *** 043  CATASTROPHIC ERROR. COMPILER INFORMATION ON FOLLOWING TWO PAGES
***  ERROR  *** 044  ATTEMPT TO DIVIDE BY ZERO
***  ERROR  *** 045  NUMBER IS OUT OF RANGE
***  ERROR  *** 049  ERROR COUNT EXCEEDED
```

Figure A5.1 IPLAN Error Messages

c) Terminal Execution Time Errors

Errors 021, 035, 036 and 049 can be overridden at compile time by use of the OPTIONS card (see paragraph 3.2.1).

APPENDIX 6

IPLAN Integer Version

This version of IPLAN has been introduced for situations where the level of programming skill achieved will not exceed the capabilities provided by working only with integer scalars.

Modification of keywords

The new keywords introduced are :

DECLARE which replaces INTEGER SCALAR

READ which replaces READ INTEGER

PRINT which replaces PRINT INTEGER

TITLE which replaces PRINT TEXT.

The following keywords are deleted from the IPLAN specifications :

INTEGER SCALAR	REAL SCALAR
INTEGER VECTOR	REAL VECTOR
INTEGER MATRIX	REAL MATRIX
READ INTEGER	READ REAL
PRINT INTEGER	PRINT REAL
CONVERT TO INTEGER	PRINT TEXT
CONVERT TO REAL	

Undeclared variables are declared by default to a scalar integer with an error message.

The integer version is not a subset of IPLAN since the keywords DECLARE, READ, PRINT and TITLE do not exist in the main version. The implementation is such that if the first declaration statement after BEGIN PROGRAM is a DECLARE statement then the integer version is assumed, otherwise the main version is assumed.

APPENDIX 7

Character Map

Character	Decimal Integer	Binary Integer
A	1	1
B	2	10
C	3	11
.	.	.
.	.	.
Z	26	11010
a	27	11011
b	28	11100
c	29	11101
.	.	.
.	.	.
z	52	110100
0	53	110101
1	54	110110
2	55	110111
.	.	.
.	.	.
9	62	111110
+	63	111111

Character	Decimal Integer	Binary Integer
-	64	1000000
space	0	0
,	66	1000010
.	67	1000011
:	68	1000100
;	69	1000101
'	70	1000110
"	71	1000111
!	72	1001000
?	73	1001001
(	74	1001010
)	75	1001011
=	76	1001100
-	77	1001101
>	78	1001110
<	79	1001111
¢	80	1010000
$	81	1010001
%	82	1010010
*	83	1010011
&	84	1010100
#	85	1010101
@	86	1010110
|	87	1010111
/	88	1011000
_	89	1011001

References

1. Shantz, P.W., German, R.A., Mitchell, J.G., Shirly, R.S.K., and Zarnke, C.R. WATFOR - The University of Waterloo FORTRAN IV compiler. Comm. ACM 10, 1 (Jan. 1967), 41 - 44.

2. Moulton, P.G., and Muller, M.E. DITRAN - A compiler emphasizing diagnostics. Comm. ACM 10, 1 (Jan. 1967), 45 - 52.

3. Rosen, S., Spurgeon, R.A., and Donelly, J.K. PUFFT - The Purdue University fast Fortran translator. Comm. ACM 8, 11 (Nov. 1965), 661 - 666.

4. IBM Corporation. FORTRAN IV Language. File S360 - 25, Form C28 - 6515 - 5 (1966). IBM Systems Reference Library, Poughkeepsie, N.Y.

5. IBM Corporation. COBOL Language. File S360 - 24, Form C28 - 6516 - 6 (1965). IBM Systems Reference Library, Poughkeepsie, N.Y.

6. Brillinger, P.C., Ehle, B.H., and Graham, J.W. An Introduction to the SPECTRE COMPUTER. University of Waterloo, Ontario. (1968)

7. Hull, T.E. Introduction to Computing. Prentice Hall.

8. Conway, R.W., and Maxwell, W.L. CORC : The Cornell Computing Language. Comm. ACM 6,6 (June 1963), 317.

9. McClure, C.W., Sanderson, K., and Davis, J. FORGO COMMON Program Library 1620 - 02.0.008, IBM Data Processing Program Information Dept., Hawthorne, N.Y.

   Shantz, P.W. Notes on the FORGO System. University of Waterloo.

10. Arden, B.W., Galler, B.A., and Graham, R.M. The internal organisation of the MAD translator. Comm. ACM 4,1 (Jan. 1961), 28 - 31.

11. Computer Bulletin Working Party 7. Specimen courses in computer science. Computer Bulletin (June 1967), 43 - 50

12. Naur, P. (Ed) Revised report on the algorithmic language ALGOL 60.
In :
Rosen, S. (Ed) Programming Systems and Languages.
McGraw Hill (1967), 79 - 118.

13 IBM Corporation. Assembler Language. File S360 - 21, Form C28 - 6514 - 5 (1967)

IBM Corporation. Supervisor and Data Management Macro Instructions. File S360 - 36, Form C28 - 6647 - 0 (1967)

IBM Corporation. System 360 Principles of Operation. File S360 - 01, Form A22 - 6821 - 7 (1968). IBM Systems Reference Library, Poughkeepsie, N.Y.