5 440 10c

Scale 2

1

63-4-2

THE INSTITUTE FOR COOPERATIVE RESEARCH

ICR

UNIVERSITY of PENNSYLVANIA

PHILADELPHIA, PENNSYLVANIA

$9.60

(5) 440 100

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC

STATUS REPORT

Dated: 30 June 1962

Covering: 1 July 1961 to 30 June 1962
Contract No. DA36-034-507-ORD-3347RD
Department of the Army Project No.
Ordnance Corps Project No.

Technical Supervisor: Frankford Arsenal

Prepared by:

(16) by C. Beckman, ~~Project Director~~
S. D. Bedrosian,
R. S. Berkowitz,
P. Z. Ingerman,
D. Prener and
R. L. Wexelblat.

The Institute for Cooperative Research
University of Pennsylvania

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC

30 June 1962

Approved:

C. Beckman

C. Beckman
Project Director

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC

TABLE OF CONTENTS

List of Figures and Tables

SPECIAL REPORT

ON ELEMENT VALUE SOLUTION OF SINGLE-ELEMENT-KIND NETWORKS

S. D. Bedrosian

TABLE OF CONTENTS

ABSTRACT

The present study is a continuation and extension of work done during 1959 and 1960 entitled "A Study of Piece Part Fault Isolation by Computer Logic." During the period June 1961 - June 1962 the emphasis has been on automatic checkout of Automotive Systems with special regard to the computer controlled tank checkout system developed by the Frankford Arsenal and on an extension of the work of Berkowitz in network element solvability.

Section A on Automatic Programming is divided into an introduction followed by two main parts and five appendices. The first of the main sections discusses the compiler pseudo-language, derived from the language developed for the Electronic Piece Part Fault Analysis system, is discussed giving a formal specification of the syntax of the language and showing that part of it is a general algebraic compiler and part is a series of problem oriented input-output statements. The second main section discusses the compiler. This compiler was first presented in the June 1961 report of this Project and the present discussion describes various additions and changes. Also given are the language syntax in a tabular format and again in a machine-oriented matrix format.

Appendix 1 presents a discussion of the methods of generating the tabular and matrix formulations of the language syntax. Appendix 2 presents a brief discussion of a set of symbolic strings used by the first section of the compiler and decoded by the second. Appendix 3 discusses some modifications that might make the compiler more efficient. Appendix 4 gives the coding of some of the service routines for the Libratrol-500 computer. In Appendix 5, Part 1 is a short manual to the

language, giving an informal presentation that might be the basis for a training manual to be written at some later date.

Appendix 5, Part II, gives some practical illustrations of the use of the language.

Section B, on Circuit Analysis and Network Element Value Solvability Studies consists of the Bedrosian Ph.D. dissertation on Element Value Solutions of Single-Element-Kind Networks.

In this study the work of Berkowitz is extended by finding explicit solution techniques for single-element-kind networks. This includes a computer programmable algorithm and discussion of various ramifications.

The original object of this effort, based on an optimistic attitude, was to consider element value solvability for major classes of networks. Subsequent investigation revealed unexpected subtleties and inherent difficulties associated with this problem. Consequently, the work has been restricted to the treatment of single-element-kind networks,

Section A. Automatic Programming

I. Introduction

This report describes the work done on automatic checkout on Project ROTEK from June 1961 to June 1962. The effort was devoted to a language and compiler oriented toward Automotive Systems (tanks, in particular) with the Libratrol-500 to be used as the testing computer and the compiling computer as yet unspecified.

Work was divided into three sections:

Language development
Further work on the compiler
Writing of utility programs for the Libratrol-500

These are described in detail in the succeeding sections of this report. Also included is a brief manual of the language which is designed to be the basis of a larger training manual. At various places in this report, reference is made to the June, 1961 status report. This is:

Beckman, et. al, Study of Piece Part Fault Isolation by Computer Logic - Status Report, June 1961, The Institute for Cooperative Research, University of Pennsylvania.

II. The Compiler Language

The language presented here is derived from FATAL (see June, 1961 report) and, like FATAL is based upon ALGOL-60. It is divided into two sections: an algebraic compiler language and a series of special statements having to do with input and output and using a problem oriented vocabulary.

The Basic Algebraic Compiler

The algebraic compiler is based upon ALGOL-60 and as put forward here is probably more powerful than would actually be required for an automatic checkout system.[1,23,29] The compiler discussed in section III of this report is, however, independent of the input or output languages and will just as readily handle any reasonable subset of the language. By reasonable subset we mean any subset that is unambiguous in meaning and in definition. And, what is more important, the speed and efficiency of compiling is dependant upon the language used so there will be no decrease in efficiency caused by hypothesizing a more general language than will actually be used. From outward appearances, this part of the language looks quite different from the algebraic parts of FATAL. These differences were due to an attempt to make the language as close to English as possible. There are those in the computer field today who are still of the opinion that English is a good programming language. This is not so and it should be emphasized that the unnecessary and redundant words and symbols added to the language increase ease of understanding at the cost of efficiency of compiling. Compare our VARY statement and FORTRAN's DO statement to see an example of this.[25]

As in last year's report, the language is given in the Backus Normal Form (B.N.F.). The formal specification is prefaced with a formal and an informal description of how the Backus Normal Form works.* The following specification is meant to be self-contained and can be used independent of the rest of this report.

The following is quoted from the paper "Specification Languages for Mechanical Languages and Their Processors, A Baker's Dozen" by Saul Gorn.**

> A Backus syntactic specification language is a linear sequential language over an alphabet (infinite) composed of the following symbols:
>
> a. The Bracket Symbols '<' and '>'.
>
> b. The production symbol '::='.
>
> c. The choice symbol '|'.
>
> d. Two alphabets called 'names' and 'symbols'. Both of these alphabets are often selections of words from a prior alphabet and can be infinite in number (but not in length).
>
> The individual 'names' are intended to denote auxiliary languages called 'syntactic types' of the language being specified; a string of symbols consisting of '<' followed on the right by a name and then followed by '>' is intended to designate the extent of the name, i.e. the set of strings in the specified language of which it is the name — or, put another way, surrounding a name by these brackets is an operation transforming the intent designated by the name to the set of processors which operate on it.
>
> The individual 'symbols' are designated strings of characters from the alphabet of the language being specified, and

*It should be noted that B.N.F. is a Chomsky type-2 Phrase Structure Grammar and languages specified in B.N.F. are either type-2 grammars or can often be made so by suitable minor modifications. There is a large body of literature pertaining to these subjects that is useful to this study.[4,7,8,11]

**Comm. of the A.C.M. V. 4, No. 12, Dec. 1961, p. 532.

often are designations by the objects themselves of auxiliary language with only one word, namely that string. The symbol '|' is used to chain together alternative forms, each of which yields composite syntactic types which are auxiliary languages formed by concatenation. Thus a symbol concatenated with a symbol designates another symbol, a bracketed name concatenated with a symbol designates the set of words obtained by concatenating that symbol with each word of the set designated by the name, a bracketed name concatenated with a bracketed name designates the 'product set' of all strings designated by the other.

By using a string of the form: 'bracketed name' concatenated on the right by '::=' concatenated on the right by 'a composite name' constructed with '|', bracketed names, symbols and concatenation we obtain in compact form a set of production rules for the sublanguage designated by the bracketed name at the extreme left of the string.

The foregoing quotation is a formal definition of a syntactic specification in the Backus normal form. A few examples are given here for purposes of illustration. Where examples are taken from our language the source is shown.

The simplest production rule consists of a bracketed name on the left of a ::= and a single symbol on the right:*

<author>::= SHAKESPEARE

This is to be interpreted: wherever the name 'author' appears in brackets the symbol 'SHAKESPEARE' can be substituted for it. The symbol '|' is interpreted 'or'. We redefine the name author as

<author>::= SHAKESPEAR|MARLOWE|BACON|DRYDEN|MILTON

which means that the name 'author' is defined as the name of the class which contains {SHAKESPEAR, MARLOWE, BACON, DRYDEN, MILTON}.

*In this paper we use lower case letters for names and upper case for symbols. A series of upper case letters (as: SHAKESPEARE) is considered a single symbol unless explicitly stated otherwise.

It is possible for bracketed names to appear on the right of ::=

<author>::=<blind author>|<elizabethan author>|<17th century author>

This defines 'author' as the name of the class which contains the union of the classes: {blind author, elizabethan author, 17th century author} .

These classes might also be defined.

<blind author> ::= MILTON
<elizabethan author> ::= SHAKESPEAR|MARLOWE|BACON
<17th century author> ::= DRYDEN|POPE|<blind author>

If we define:

<opus> ::= HAMLET|DUCHESS OF MALFI|PEYTON PLACE

and

<authorship statement> ::=<author>WROTE<opus>.*

We can generate a whole series of statements by substituting the symbols that <author> and <opus> name into the given statements.

For example:

SHAKESPEARE WROTE HAMLET.
BACON WROTE HAMLET.
DRYDEN WROTE PEYTON PLACE.
MARLOWE WROTE DUCHESS OF MALFI.

All of these statements are <u>syntactically</u> correct. The third is, however, <u>semantically</u> incorrect; while the correctness of the first two is disputed.

There is another concept to be considered: that of the empty (or null) set.

Definition:

<empty> ::=

*This defines an authorship statement as: a member of the set of authors followed by the symbol 'WROTE' followed by a member of the set of opera followed by a period.

If we re-define:

<opus> ::= HAMLET | DUCHESS OF MALFI | PEYTON PLACE | <empty>

then the following statements are syntactically (and semantically) correct:

SHAKESPEARE WROTE.
MILTON WROTE. etc.

For the concept of recursive definitions, take an example from the language (1.A).

If letters and digits are defined intuitively as primitive concepts we can define:

<symbol> ::= <letter> | <digit>
<symbol string> ::= <symbol> | <symbol string> <symbol>

<symbol string> is defined in terms of itself (i.e. recursively) this allows the building up of a symbol string of any length containing any combination of letters. Starting with the letter R, which is a symbol string by the first part of the definition of symbol string, second part of the definition (which says that a symbol string is any symbol string followed by any symbol) allows the formation of the string R1. Again by the second part of the definition R12; then R124; then R124C; then R124C4; then R124C41 and so on.

The syntax of the name 'number' is given by a combination of definitions, both simple and recursive, (1.B.1). The simplest form of a number is a digit. By the same sort of recursive definition used for symbol string an unsigned integer is a series of digits (of any length). A decimal fraction is an unsigned integer preceeded by a decimal point; while a decimal number is either an unsigned integer, a decimal fraction or else an unsigned integer followed by a decimal fraction (i.e. 3 or .14159 or else 3.14159). An exponent part is defined as 10** integer where the double star is the symbol for exponentiation and an integer is either an unsigned integer or else an unsigned integer preceeded by either '+' or '-'. An unsigned number is either a decimal number, an exponent part or the former followed by the latter with a star between (star is the sign for multiplication). Finally, a number is either an unsigned number or an unsigned number with a sign before it.

In this manner the entire syntax of the language is defined in terms of the basic symbols. A program is (by the rules of 6.A) a string of these basic symbols put together according to the rules of the definitions following.

1. Symbols and Definitions

1.A Symbols

```
<symbol> ::=<letter>|<digit>
<symbol string>::=<symbol>|<symbol string>|<symbol>
<letter>::=⎫
<digit>::= ⎭defined according to their intuitive meanings
```

1.B Numbers

1.B.1 Syntax

```
<number>::=<unsigned number>|+<unsigned number>|-<unsigned number>
<unsigned number>::=<decimal number>|<exponent part>|
                    <decimal number>*<exponent part>
<decimal number>::=<unsigned integer>|<decimal fraction>|
                   <unsigned integer><decimal fraction>
<unsigned integer>::=<digit>|<unsigned integer> <digit>
<decimal fraction>::=.<unsigned integer>
<exponent part>::=10**<integer>
<integer>::=<unsigned integer>|+<unsigned integer>|-<unsigned integer>
```

1.B.2 Semantics

Any number of the forms usually associated with decimal numbers can be expressed. The exponent part is shorthand for a scale factor expressed as an integral power of 10.

1.C Comments

At any point in a program, the programmer may put a comment which consists of any symbol string without a semi-colon. This will be ignored by the compiler. Comments are of use for making explanatory notes within a program.

```
<comment>::=COMMENT:<symbol string>;
```

2. Variables and Expressions

2.A Variables

2.A.1 Syntax

```
<value> ::= <number> | <variable name
<variable name> ::= <simple variable> | <vector> | <matrix>
<simple variable> ::= <variable identifier>
<vector> ::= <variable identifier> (<primary>)
<matrix> ::= <variable identifier> (<primary>, <primary>)
<variable identifier> ::= <letter> <symbol string>
```

2.A.2

The term variable name is used to denote a quantity referred to by name rather than by explicit appearance and which is able to take on a number of values. In this language as presently formulated, there are three types of variable names allowed: simple variables, vectors, and matrices. A vector is a subscripted variable having one subscript and a matrix is a subscripted variable having two subscripts. C.f., however, section 3.E.4.

Values of variable names can be changed by set statements or refer statements (q.v.)

Although any string of letters is syntactically a legitimate variable identifier, the use of the names of the standard functions (see 2.B.4.C) should be avoided.

2.A.3 Examples

Simple variables:

X
X3
FOOF
SUM

Vectors

X(i)	X(2)
Y(i+j/2*n)	Y(4)
SUM (SIN(3*PI/2*n))	SUM2(97)

Matrices

X(i, j)	X(1,3)
Y(X(i,j),COS(Z(k)))	Y(27,2)
SUM2(SIN(X),3*ARC TAN(THETA))	SUM2(1,5)

2.A.4 Subscripts

As can be seen from the above examples the subscripts of vectors or matrices can be expressed in terms of numbers, of other variables or arithmetic expressions, however, the actual component of the vector or matrix referred to is specified by the actual numerical value of the subscripts. (If this numerical value is not an integer, its value is taken as the nearest integer not greater than the actual value. Only positive (non-zero) subscripts are defined.)

If a subscripted variable is to be used, it must have previously been defined by a dimension statement (q.v.) or a refer statement (q.v.).

2.B Arithmetic Expressions

2.B.1 Syntax

```
<adding operator>::=+|-
<multiplying operator>::= *|/
<memory value>::=<number>|<simple variable>
<function name>::=ABS|SIGN|SQRT|SIN|COS|ARCTAN|LN|EXP|ENTIER|
                MEDIAN|LOG|MEAN
<function expression>::=<function name>(<arithmetic expression>)
<primary>::=<memory value>|<vector>|<matrix>|
           <procedure call>|<function expression>|
           (<arithmetic expression>)
```

```
<factor>::=<primary> <factor>**<primary>
<term>::=<factor>|<term> <multiplying operator> <factor>
<arithmetic expression>::=<term>|<adding operator> <term>|
                       <arithmetic expression><adding operator><term>
```

2.B.2 Semantics

An arithmetic expression is a rule for computing a numerical value. When the indicated arithmetic operations are performed on the actual numerical values of the primaries of the expression the value of the expression is obtained. For variables in an arithmetic expression the value of the expression is obtained from the current value while for function designators it is the value arising from the indicated operation. For arithmetic expressions in parenthesis the value must be expressed in terms of the values of the primaries within the parenthesis. A memory value is a convenient special case used when there exist non-subscripted variables. In this case, the coding generated is more efficient.

2.B.3 Examples

Primaries:

```
1732*10**-3          (3+Y*Z)              X(1)
X                    cos(X+3)             X(3,7)
omega                (cos(X)+3)           Y(3n+2)
```

Factors:

```
1.414**7             cos(X+3)**Y          X(i,j)**(3+K)
1.414**Y             cos((X+3)**Y)
```

Terms:

```
1.414**Y/cos(X+3)         1.414**Y/(cos(X+3)*Z     (A/B)*C
(1.414**Y/cos(X+3))*Z     A/B                      A/(B*C)
```

Arithmetic Expressions:

```
+ln(3-Y)             C*D+B**A
-(A+B)               3+4*cos(5+T)
```

2.B.4 Operators and Functions

2.B.4.a Definitions

The operators +, -, *, and / have their conventional meaning (addition, subtraction, multiplication, and division). The ** symbol is used to symbolize exponentiation. In the operation <factor>**<primary> the factor is the base and the primary is the exponent.

For example:

a**b**c means $(a^b)^c$

a**(b**c) means $a^{(b^c)}$

2.B.4.b Precedence

The following rules of precedence of evaluation hold:

first: **
second: */
third: + -

These follow from the definitions in 2.B.1.

2.B.4.c Standard Functions

Various standard functions are expressed in abbreviated notations; several of these are:

ABS(E)	The absolute value of expression E
SIGN(E)	The signum of E(+1 if $E>0$, -1 if $E<0$, 0 if $E=0$)
SQRT(E)	The square root of E
SIN(E)	The sine of E
COS(E)	The cosine of E
ARCTAN(E)	The principal value of the arc tangent of E
LN(E)	The natural logarithym of E: $\log_e(E)$
LOG(E)	The logarithym to the base 10 of E: $\log_{10}(E)$

EXP(E)	The exponential of E: (e^E)
ENTIER(E)	Largest integer not greater than E
MEDIAN(E)	a. If E has no free subscripts, MEDIAN (E) = E b. If E has one free subscript then, if the values of E be arranged in order of magnitude: 1. MEDIAN(E) is equal to the value in the central position (if the greatest value of the subscript is odd). 2. MEDIAN(E) is equal to the mean of the two values nearest to a central position (if the greatest value of the subscript is even). c. If E has more than one free subscript, MEDIAN (E) = 0
MEAN(E)	a. If E has no free subscripts, MEAN(E) = E b. If E has one free subscript then, if n is the greatest value of this subscript, MEAN(E) = (E(1) + E(2) + ... + E(n))/n. c. If E has more than one free subscript MEAN(E) = 0

E may be any arithmetic expression.

3. Basic Statements

3.A Let Statement

3.A.1 Syntax

```
<equation>::=<variable name>:=<arithmetic expression>
<equation list>::=<equation>|<equation list  ,<equation>
<let statement>::=LET<equation list>;
```

3.A.2 Semantics

The let statement is used for computations.

For example:

```
LET X:=3*COS(THETA)/SIN(SQRT (PHI)),
    Sum:=2*X - 5/X**Z,
    X(i+1):=X(i) + Y(j);
```

3.B Transfer Statement

3.B.1 Syntax

```
<location>::=<procedure call>|<statement label>
<transfer statement>::= GO TO<location>;|HALT<location>;
```

3.B.2 Semantics

The GO TO transfer can be used for a jump or unconditional transfer of control. The HALT is the same except that the computer stops before transferring.

If transfer is made to a statement (identified by a label), this statement must be in the procedure being run at the time. Transfer may be made to another procedure. This is discussed in 4.B and the syntax of <procedure call> is given in 4.A.

3.C Condition Statement

3.C.1 Syntax

```
<relation>::=<|≤|=|≥|>|≠
```

```
<condition>::=<primary><relation> <primary>
<condition part>::=<condition>|<condition part>AND<condition>
<condition list>::=<condition>|(<condition part>)
                   <condition list>OR<condition list>
<statement list>::=<statement>|<statement list> <statement>
<end>::=.|.<symbol string>
<condition tail>::=<statement list> <end>|
                   <statement list>ELSE<statement list> <end>
<condition statement>::=IF<condition list>THEN<condition tail>;
```

3.C.2 Semantics

A condition statement is of one of the two forms:

a. IF (some combination of conditions is fulfilled) THEN (do something) ELSE (do something else).

b. IF (some combination of conditions is fulfilled) THEN (do something).

In the first case, if the conditions are fulfilled, the list of statements following the THEN are performed and the list following the ELSE is skipped. If the conditions are not fulfilled, the list following the ELSE is performed and the one following the THEN is omitted. In the second case, if the conditions are not fulfilled, control is transferred to the next statement following the condition statement.

The condition list is made up of combinations of simple conditions of the form:

<primary><relation><primary>.

Some examples:

```
a>b
D(2,3) 12
J**2≠-1
3<A+B(j)
SIN(THETA*2*PI)=COS(OMEGA/2*PI).
```

The syntax of condition list allows the joining of these simple relations into more complex forms. A condition part can be any string of conditions connected by AND. For example:

a>b AND D(2,3) ⩾2 AND J**2 ≠ -1 AND ...

This is satisfied if and only if each of the individual conditions is satisfied. A condition part in parenthesis is a condition list and can be used in a condition statement.

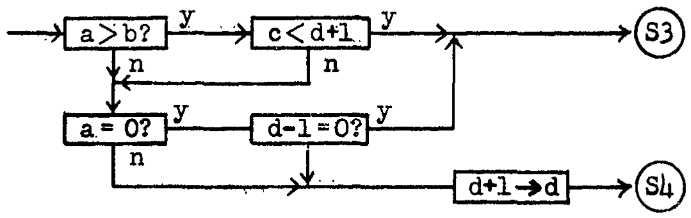
It is also possible to combine two or more condition lists by stringing them along with OR separating them. In this case the expression is satisfied if and only if at least one of the constituent conditions is satisfied. For example:

(a>b AND b⩽c) OR J**2 ≠ -1 OR
(A(1,j) = B(j+3) AND C = 0) OR ...

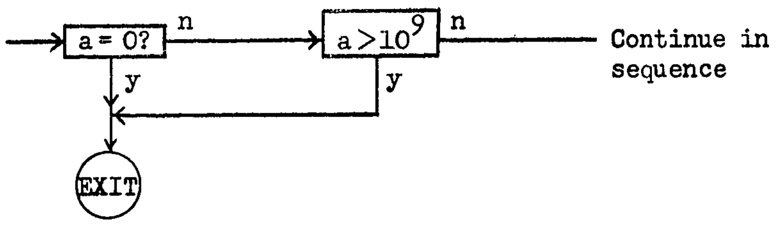
The statement list may consist of any combination of statements and ends in a period. A comment may be added after the period if desired without the necessity of writing "COMMENT:"

3.C.3 Examples:

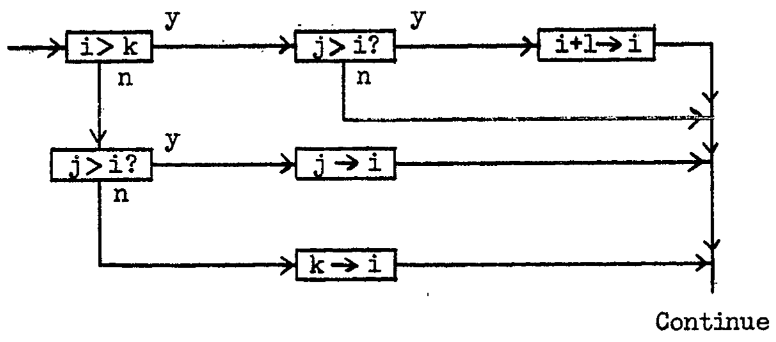
IF (a>b AND c<d+1) OR (a = 0 AND d-1 = 0) THEN
GO TO S3; ELSE LET d: = d+1; GO TO S4; .; represents:



IF a = 0 OR a>10**9 THEN GO TO EXIT; .; represents:



```
IF i>k THEN
   IF j>i THEN LET i:=i+1 ; . CASE 1;
         ELSE
   IF j>i THEN LET i:=j; ELSE LET i:=k; . CASE 2; represents
```



"CASE 1" and "CASE 2" are comments added for mnemonic purposes.

3.D Vary Statement

3.D.1 Syntax

```
<lower limit> ::=<primary>
<upper limit> ::=<primary>
<increment> ::= <primary>
<statement list> ::= <statement>|<statement list> <statement>
<vary by steps> ::=
          VARY<variable name>FROM<lower limit>
            BY<increment> UNTIL<upper limit>
            IN<statement list><end>;
<name list>::=<variable name>|<name list>,<variable name>
<vary list>::=<value>|<vary list>,<value>
<compound vary list>::=(<vary list>)|<compound vary list>(<vary list>)
```

```
<end> ::= . | .<symbol string>
<vary by list> ::=
            VARY<name list>OVER<compound vary list>
              IN<statement list><end>;
<vary statement> ::=<vary by steps>|<vary by list>
```

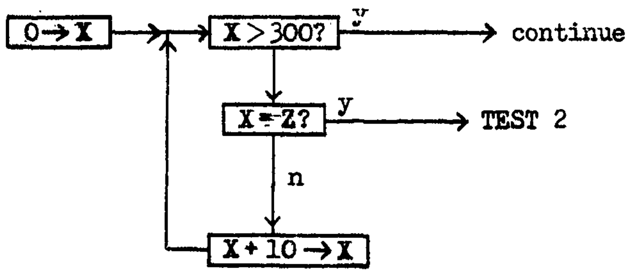
3.D.2 Semantics

The first vary statement will cause the value of a variable in a specified list of statements to be modified by increments over a given range. The second vary statement will cause one or more variables to be varied each over its own specified list of values. It is possible for the statement list of one vary statement to contain another vary statement. A period ends the statement list of a vary statement. As before, a comment (not containing a semi-colon) may be inserted after the period for purposes of explanation.

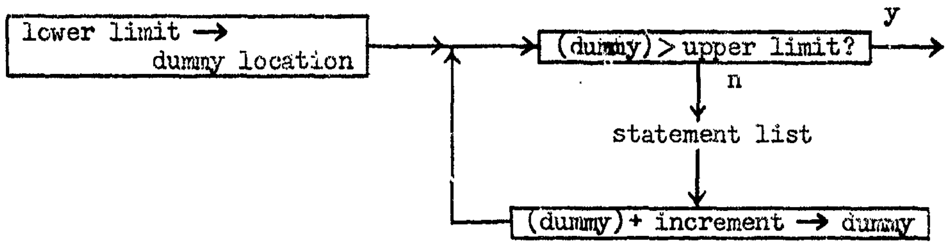
3.D.3 Examples

```
VARY X FROM 0 BY 10 UNTIL 300 IN
   IF X= Z THEN
   GO TO TEST 2; .; .;
```

In this example, Z is some previously defined simple variable and the simple variable X is varied by steps of 10 from 0 to 300 in the condition statement "IF X= Z THEN GO TO TEST 2; .;". The flow chart for this is:



The general form of a vary by steps statement is



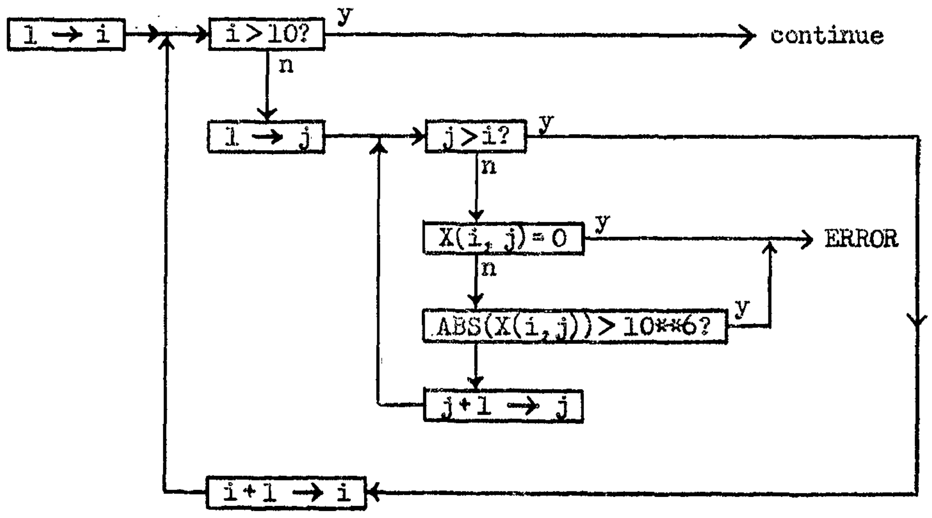
The limits and increment do not have to be constants:

```
VARY i FROM a BY ((a - b)/100) UNTIL b IN
   IF 3*i**2 > 10 THEN
      LET c: = i ;
      GO TO S9; .; . END OF i LOOP;
```

A compound vary statement:

```
VARY i FROM 1 BY 1 UNTIL 10 IN
   VARY j FROM 1 BY 1 UNTIL i IN
      IF X(i, j) = 0 OR
        ABS(X(i, j)) > 10**6 THEN
          GO TO ERROR; .; .; .;
```

can be represented:



wherein the symmetric matrix X_{ij} is taken term by term (actually, only those terms along or above the main diagonal are considered) and if any term equals 0 or is greater in magnitude than 10^6, control is transferred to ERROR.

In a vary by list statement, variables are ranged over lists. For example:

```
VARY K, X, Z OVER (1,2, 3)(A, B, C)(10**2), 10**3, 10) IN
   LET P(k) := Z*X; .;
```

This statement will perform the following operation:

```
LET P(1):=10**2*A
    P(2):=10**3*B
    P(3):=10*C;
```

3.E Dimension and Refer Statements

3.E.1 Dimension Statement Syntax

```
<vector limit>::=<variable identifier>(<unsigned integer>)
<matrix limit>::=<variable identifier>
                   (<unsigned integer>,<unsigned integer>)
<subscript limit>::=<vector limit>|<matrix limit>
<dimension list>::=<subscript limit>|
                     <dimension list>,<subscript limit>
<dimension statement>::=DIMENSION<dimension list>;
```

3.E.2 Refer Statement Syntax

```
<vector identifier>::=<variable identifier>
<value list>::=<value>|<value list>,<value>
<pair>::=<value list>AS<vector identifier>|
           <value>AS<variable identifier>
<pair string>::=<pair>|<pair string>,<pair>
<pair part>::=<pair string>AND |<empty>
<pair list>::=<pair part> <pair>
<refer statement>::=REFER TO<pair list>;
```

3.E.3 Semantics

The dimension statement is used to allocate storage for vectors and matrices.

For example if a program used the matrices SUM (i, j) and A (r,t) (where $1 \leqslant i \leqslant 10$, $1 \leqslant j \leqslant 20$, $1 \leqslant r \leqslant 3$ and $1 \leqslant t \leqslant 200$) and the vectors PRESS (k) and TEMP (s) (where $1 \leqslant k \leqslant 30$ and $1 \leqslant s \leqslant 5$), then the following statement would appear in the program (this must, of course come before the use of any of the dimensioned names):

```
DIMENSION SUM (10, 20), A (3, 200), PRESS (30), TEMP (5);
```

The refer statement has two uses, one is to allow the name of a variable or set of variables to be changed and the second to allow a series of values of variables or constants to be combined into a single vector for ease in reference. When a simple variable or an entire vector or an entire matrix is renamed by a refer statement, the original variable name becomes undefined and may be re-assigned at will.

For example, the following refer statement will put the combination of values of variables X(1,1), Y(1,1), Z(3), ALPHA and 13.75 into a vector, V:

```
REFER TO X(1,1), Y(1,1), Z(3), ALFHA, 13.75 AS V;
```

The statement

REFER TO X AS Y, PRESS AS FOOF AND GEB AS GBW; will give X, PRESS and GEB the new names Y, FOOF, GBW respectively. X, PRESS and GEB could be simple variables or arrays. The equivalents would have the same dimension.

3.E.4 General Subscripting

For the system as proposed, compilation time will be decreased (at the cost of decreased generality and wasted storage) by limiting arrays to two subscripts. Below is a reformulation of Dimension using the name ARRAY to identify a matrix of any dimension.

```
<array limit> ::= <arithmetic expression>
<limit list> ::= <array limit>|<limit list>,<array limit>
<array segment> ::= <variable identifier>(<limit list>)|
                    <variable identifier>
<array list> ::= <array segment>|<array list>,<array segment>
<array statement> ::=ARRAY <array list>;
```

In this case, the array statements do not have to appear at the beginning of a program but may appear anywhere. Note that this implies dynamic storage allocation. If this feature is not required, use

```
<array limit> ::= <unsigned integer>
```

The dynamic storage allocation implies the use of such concepts as global and local variables and block structure. We attempt to gain some of the advantages of this through the use of the RELEASE psuedo-statement (c.f. 3.I).

3.F Type Declarations

3.F.1 Syntax

```
<identifier list> ::= <variable identifier>|
                      <identifier list>,<variable identifier>
<declaration> ::=INTEGER <identifier list>;|
                 INTEGER VECTOR <identifier list>;|
                 INTEGER MATRIX <identifier list>;|
```

3.F.2 Semantics

In general, variables name floating point numbers. The use of an INTEGER declaration will make the value of the variable an integer which will be stored at some fixed magnitude in memory. INTEGER VECTOR and INTEGER MATRIX serve to make all the values of a vector or of a matrix integers. Declaring a variable used in a subscript to be an integer will facilitate the finding of the value desired.

Variables of either type (integer or floating point) may be used in any equation. The type of the result is the type of the variable found on the left of the ":=" .

3.G Dummy Statements

3.G.1 Syntax

```
<dummy statement>::=
```

3.G.2 Semantics

A dummy statement is occasionally useful for labelling the end of a procedure.

3.H Code Statements

3.H.1 Syntax

```
<code statement>::=CODE<symbol string>END;
```

3.H.2 Semantics

If it is wished to include a block of computer coding (numerical and/or symbolic) in the program, it can be placed between "CODE" and "END;" and the compiler will pass it through unchanged. N.B.: the symbol string "END;" must not appear within the block of coding or the compiler will take that to the end of the block.

3.I The Release Statement

3.I.1 Syntax

```
<identifier list>::=<variable identifier>|
                    <identifier list>,<variable identifier>
<release statement>::=RELEASE<identifier list>;
```

3.I.2 Semantics

This statement is actually a pseudo-statement since it will generate only pseudo-operations for the assembler. It is used to conserve storage... when a section of the program is finished with some simple variable, vector, or matrix, the use of RELEASE will permit the memory used for that variable, vector, or matrix to be reassigned.

4. Procedures and Program

4.A Syntax

```
<unlabelled statement>::=<code statement>|<condition statement>|
                         <declaration>|<dimension statement>|
                         <dummy statement>|<let statement>|
                         <read statement>|<refer statement>|
                         <release statement>|<set statement>|
                         <special statement>|<transfer statement>|
                         <vary statement>

<statement label>::=<letter> <symbol string>

<statement>::=<statement label>:<unlabelled statement>|
              <unlabelled statement>

<statement list>::=<statement>|<statement list> <statement>

<procedure name list>::=<function name>|<procedure identifier>|
                        <function name>,<procedure name list>|
                        <procedure identifier>,<procedure name list>

<subroutine list>::=(<procedure name list>)|E

<variable list>::=<variable name>|<variable list>,<variable name>

<formal parameter list>::=(<variable list>)|E

<procedure heading>::=<formal parameter list>,<subroutine list>

<procedure identifier>::=<letter> <symbol string>

<procedure>::=PROCEDURE<procedure identifier>
                       <procedure heading> <statement list>END

<procedure sequence>::=<procedure>|<procedure sequence> <procedure>

<program>::=START<procedure sequence> STOP

<parameter>::=<variable name>|<arithmetic expression>

<actual parameter list>::=<parameter>|
                          <actual parameter list>,<parameter>

<procedure call>::=<procedure identifier>(<actual parameter list>);
```

4.B. Semantics and Examples

A complete program begins with START and ends with STOP. Between these delimiters there may be several procedures, each with END to denote its finish. It is possible for one procedure to call upon another as a subroutine. If this is to be done, each procedure to be called is named in the heading of the calling procedure. It is emphasized that all variables first mentioned within a procedure are local to that procedure and if used by another procedure (except as allowed by the following) are treated as completely new names. The exception to this rule is in the case of a procedure being used as a subroutine. The sub-procedure may refer to any variables of the super-procedure (but not vice versa) which are mentioned in the procedure call. The symbolic assembly program will take care of this detail.

Consider the following procedure which will transpose a given matrix. It is assumed that this procedure is to be used only as a subroutine and the matrix to be transposed is to replace the original matrix:

```
PROCEDURE TRANSPOSE (n, m), E;
COMMENT: n is the order of the matrix to be transposed, and
         M is the matrix itself;
INTEGER  i, j;
VARY i FROM 1 BY 1 UNTIL n IN
VARY j FROM i BY 1 UNTIL n IN
LET TEMP:=M(i, j),
    M(i, j):= M(j, i),
    M(j, i):= TEMP; .END j LOOP; .END i LOOP;
COMMENT: This routine is actually short enough not to need to be
         used as a subroutine, but it will do for illustrative
         purposes; END
```

In the above procedure, the variables i, j and TEMP are local to the subroutine and if the symbolic names "i", "j" or "TEMP" were used by another routine, they would have an entirely different meaning. If it were now necessary to use this procedure as a subroutine the entry to the subroutine might be by a procedure call of one of the following forms:

```
a.) TRANSPOSE (n, M);
b.) TRANSPOSE (23, a);
c.) TRANSPOSE ((c + d)/e, L);
```

In a procedure heading, the variables are really dummy variables and, upon operation, the true values, i.e. those in the procedure call are used. In a.) above, the values of n and M in the subroutine are those found in the super; in b.) n is set to 23 and the values for M are taken as the values of the matrix a (which must have been defined in the super program); and, in c.) n is set equal to the value of (c + d)/e (where c, d, and e must be defined in the super program) and values for M are taken from the values of the matrix L.

The following rather lengthy program is for inverting a matrix

```
PROCEDURE INVERT (A, n, s, A1), E;
     COMMENT:  1. This program is a translation from the original ALGOL
                  of Algorithm 42 by T. C. Wood, published in the
                  April 1961 issue of COMM. A.C.M. (p. 176).

               2. This procedure inverts the square matrix A of order n
                  by applying a series of elementary row operations to
                  the matrix to reduce it to the identity matrix. These
                  operations when applied to the identity matrix yield
                  the inverse A1. The case of a singular matrix is
                  indicated by the value s:=1 (upon exit).

               3. A and A1 are arrays that must have been previously
                  dimensioned. s and n were previously declared integer.

               4. This program uses variable size arrays. If these were
                  not allowed, an integer would be substituted for n;

     DIMENSION a(n,2*n);  INTEGER i, j, k, m, ind;
     COMMENT: Augment matrix A with the identity matrix;
     VARY i FROM 1 BY 1 UNTIL n IN
        VARY j FROM 1 BY 1 UNTIL 2*n IN
            IF j≤n THEN LET a(i, j):=A(i, j); ELSE
               IF j= n+ i THEN LET a(i, j):= 1.0; ELSE
                  LET a(i, j):= 0.0; .; .; .
```

```
   END j LOOP; .
END i LOOP;
COMMENT: Begin inversion;
VARY i FROM 1 BY 1 UNTIL n IN
   LET j:=i, m:=i, ind:=0, s:=0;
L1:  IF a(m, j)=0 THEN
        LET ind:=1;
        IF m<n THEN
           LET m:=m+1;
           GO TO L1;
        ELSE LET s:=1;
           GO TO L2; .; .;
     IF ind=1 THEN
        VARY k FROM 1 BY 1 UNTIL 2*n IN
           LET TEMP:=a(m, k),
              a(m, k):=a(i,k),
              a(i, k):=TEMP; END k LOOP; .;
     VARY k FROM 2*n BY -1 UNTIL i IN
        LET a(i,k):= a(i, k)/a(i, i); .;
     VARY m FROM 1 BY 1 UNTIL n IN
        IF m≠i THEN
           VARY k FROM 2*n BY -1 UNTIL i IN
               LET a(m, k):=a(m, k) - a(i, k)*a(m, i); .; .; .; .
END i LOOP;
VARY i FROM 1 BY 1 UNTIL n IN
   VARY j FROM 1 BY 1 UNTIL n IN
      LET A1(i, j):=a(i, n+j); .; .;
L2:; END
```

Two examples of programs in the English-based algebraic compiler language are given. In all cases, the first comment gives the source of the routine.

a.) Exponential of a complex number

```
PROCEDURE EXPC (a, b, c, d), (EXP, SIN);
   COMMENT:  1. By John R. Herndon, Algorithm 46, from Comm. A.C.M.,
                April 1961.
             2. This procedure computes the number c+ di, which is
                equal to e^(a+ bi);
   LET c:=EXP(a);
   LET d:=c*SIN(b);
       c:=c*COS(b); END
```

b.) Logarithm of a complex number

```
PROCEDURE LOGC (a, b, c, d), (SQRT, ARCTAN, LN);
   COMMENT:  1. Same author, same source, Algorithm 48.
             2. This procedure computes the number c+ di which is
                equal to log_e(a+ bi);
   LET c:=SQRT  (a*a+ b*b);
   LET d:=ARCTAN  (b/a),
       c:=LN(c);
   IF a<0 THEN LET d:=d+ 3.1415927; .; END
```

c.) An example of a complete program with input and output.*

This program will read in a 10 x 10 matrix from tape (or card) invert it and print the result on the typewriter. We assume the existance of three procedures in <u>machine language</u>:

PROCEDURE READ(R)	will read in one number in decimal, convert it to binary and leave it in location R.
PROCEDURE WRITE(W)	will take one word, convert it to decimal and print it on the typewriter.
PROCEDURE NEWLINE	will return the carriage of the typewriter.

*Adapted from "An Introduction to ALGOL" by H. R. Schwartz, Comm. A.C.M., February 1962, p. 94.

```
START
   PROCEDURE FLIPE, (READ, WRITE, NEWLINE, INVERT);
      COMMENT: READ, WRITE and NEWLINE are code programs, INVERT
               is given as an example in Section 6.B of the
               language description and is not repeated here;
      DIMENSION A(10, 10);
      INTEGER i, j;
      VARY j FROM 1 BY 1 UNTIL 10 IN
         VARY i FROM 1 BY 1 UNTIL 10 IN
         EXECUTE READ (A(i, j)); .; .;
      EXECUTE INVERT (A, 10, singular, A);
      IF singular = 1 THEN LET oops:=-1
                          EXECUTE NEWLINE
                          EXECUTE WRITE (oops);
                          GO TO done; .;
      VARY j FROM 1 BY 1 UNTIL 10 IN
         EXECUTE NEWLINE;
         VARY i FROM 1 BY 1 UNTIL 10 IN
         EXECUTE WRITE (A(i, j)); .; .;
      done:; END
   PROCEDURE READ (R), E;
      CODE: ... END; END
   PROCEDURE WRITE (W), E;
      CODE: ... END; END
   PROCEDURE NEWLINE E, E;
      CODE: ... END; END
   PROCEDURE INVERT E, E;
      COMMENT: (see 6.B); END
   STOP
```

The Input-Output Statements

The Libratrol-500 computer has, as well as standard programmed control input-output facility, the ability to give output and take input through groups of relays connected to a special track in the memory. The automotive checkout system is built around a specially modified Libratrol-500 and an EPUT meter. The programming and operation of the Libratrol-500 are described in the Instruction and Programming Manuals published by the Librascope Division of Royal Precision Corporation (Burbank, California) and in a special Instruction Manual Supplement prepared for the Frankford Arsenal by Librascope. The EPUT meter is described in considerable detail in the Instruction Manual for the MAIDS EPUT METER prepared for the Frankford Arsenal by California Computer Products, Inc. (Downey, California).

A series of special statements called input-output statements is used to program the EPUT meter and its associated analog to digital equipment, to read in from or out to the EPUT meter and analog-digital equipment and to output various standard messages to the operator during the running of any testing program.

The statements are part of the complete compiler language but are given separately because they are designed for a special purpose. The algebraic compiler of the previous section could be coupled with the CONNECT statement of FATAL to make a circuit testing language. See 1961 report, p. 21.

5. Input-Output

5.A Set Statement

5.A.1 Syntax

```
<set statement> ::=SET<device> ;
<device> ::= <adc> <output line>|<eput>
<adc> ::=ADC TO<range> <speed>
<range> ::=1 V SCALE 10 V SCALE
<speed> ::= , HIGH SPEED( <crankshaft angle> )|<empty>
<eput> ::=EPUT<mode> <a-setting> <b-setting> <slopes> <threshold>|
           EPUT MD<a-setting> <b-setting> <slopes> <resetting> <threshold>
<mode> ::=MA | MB | MC | ME | MF | MG
<a-setting> ::= , N=<value>|<empty>
<b-setting> ::= , M=<value> <delay>|<empty>
<delay> ::= , D |<empty>
<slopes> ::=<da 1 slope> <da 2 slope> <da 3 slope>
<da 1 slope> ::= ,<sign> DA1|<empty>
<da 2 slope> ::= ,<sign> DA2|<empty>
<da 3 slope> ::= ,<sign> DA3|<empty>
<sign> ::=+|-
<resetting> ::=THEN<da 3 slope>|<empty>
<threshold> ::= , DA3 =<number> VOLTS|<empty>
<crankshaft angle> ::=<integer> DEGREES
<output line> ::=<line identifier> TO<number> VOLTS
<line identifier> ::=<letter>.<digit> <digit> <digit>
```

5.A.2 Semantics

The set statement can be used to set the scale or the method of operation of the analog to digital converter. If the high speed mode is selected than a crankshaft angle giving the final reading desired must be specified. If the given angle is not a multiple of 10, it is taken as the greatest multiple of 10 less than the given value.

Setting an output line consists of making an appropriate series of relay closures. If a voltage is specified that is not available, the nearest value less in magnitude than the specified value is selected.

The EPUT meter can be set up for any desired reading (c.f. MAIDS Manual). The following assumptions are made:

when a slope is not specified, it is +

when an a-setting or a b-setting is not specified, it is 0

when a threshold is not specified, it is unchanged from the last time the EPUT meter was used.

5.B Read Statement

5.B.1 Syntax

```
<read statement>::=READ<source>INTO<variable identifier><space>;
<source>::=<relay list><line identifier>
<relay list>::=<list member>|<relay list>,<list member>
<list member>::=GROUP(<group number>)|GROUPS(<group list>)
                RELAY(<relay number>)|RELAYS(<relay number list>)
<group list>::=<group number>|<group list>,<group number>
<group number>::=<unsigned integer>
<relay number>::=<group number>.<unsigned integer>
<space>::=(<unsigned integer>)|(<unsigned integer>,FF)
<relay number list>::=<relay number>|<relay number list>,<relay number>
```

5.B.2 Semantics

The read statement can be used alone to read relays and relay groups or in conjunction with a set statement to read from the EPUT meter or from the ADC in the high speed mode. Relay groups are identified by a unsigned integers and individual relays by a group number and a relay number. Inputs are either read into a single storage location or into a vector (which must have been previously dimensioned) starting at the location identified by the integer following the variable identifier. For example:

READ GROUPS (1, 3, 5), RELAY (29.1) INTO G(9FF);

means that the 24 relays in groups 1, 3, and 5 and relay 29.1 are read into G(9), G(10), ... G(32), G(33).

When a read statement, containing a line identifier follows a set statement, the two are taken as a unit, is the read statement specifying the input to the EPUT meter (or ADC).

5.C.1 Syntax

```
<special statement> ::= <delay statement>|<typeout statement>|
                        <definition statement>|<close switch statement>|
                        <open switch statement>
<close switch statement> ::=CLOSE SWITCH<unsigned integer>;
<open switch statement> ::=OPEN SWITCH<unsigned integer>;
<delay statement> ::=DELAY<unsigned number>SECONDS;
<typeout statement> ::=TYPEOUT<typeout list>;
<typeout part> ::='<symbol string>' |<unsigned integer>|(<variable name>)
<typeout list> ::=<typeout part>|<typeout list><typeout part>
<definition statement> ::=DEFINE<definition list>;
<definition list> ::=<definition>|<definition list>,<definition>
<definition> ::=TYPEOUT<unsigned integer>AS'<symbol string>'
```

5.C.2 Semantics

The delay statement and the switch statements are used respectively to cause time delay in the operation of the program and to open and close switches under computer control.

There are three kinds of typeout statement:

a.) TYPEOUT '<symbol string>' - the symbol string between the quotation marks is typed out (this must not contain a ";").

b.) TYPEOUT <unsigned integer> - the symbol string corresponding to the given integer (which must have previously been defined by a definition statement) is typed out.

c.) TYPEOUT (<variable name>) - the current value of the variable is typed out.

III. The Compiler

The complete compiling program for the algebraic language will consist of four separate routines. Another routine will be appended to compile the input output statements.

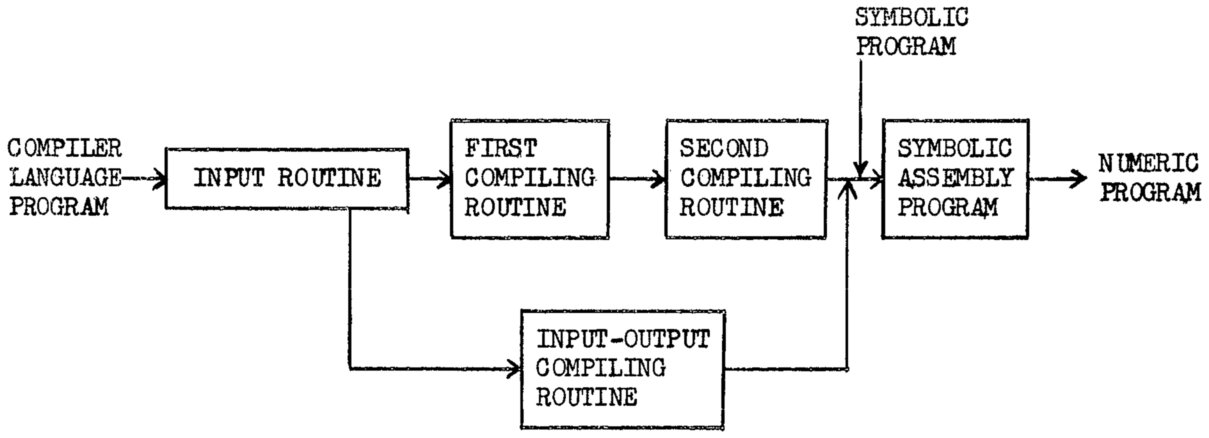


Figure 3.0 Block Diagram of Complete Compiler

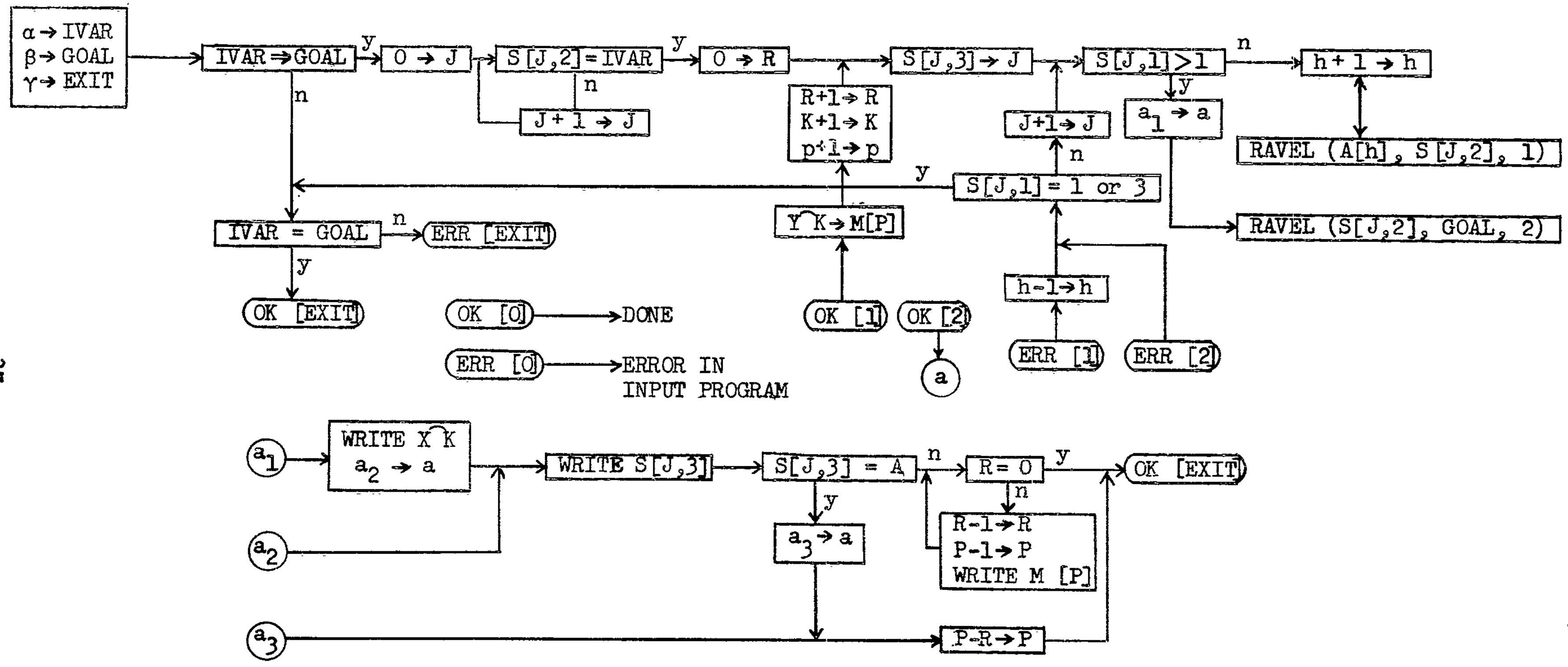
The input routine is a short one-to-one translator that will substitute numbers for the symbol strings in the input. It will also label input-output commands so that they will be processed by the special input-output compiler and it will label constants, variable identifiers, procedure identifiers, etc. as such to save the compiler proper having to process strings symbol by symbol. The substitution of fixed length numbers for variable length symbol strings will increase the efficiency of the various table look-ups needed during the compiling. The input routine is discussed in more detail later. The Symbolic Assembly Program is not a part of the compiler proper but is shown for completeness. It is emphasized that the compiler is able to generate either symbolic or absolute coding (or both, at some small loss of efficiency) and as such the Assembly Program can be eliminated entirely. For a computer such as the Libratrol-500 which is single address, it would be most efficient to generate numeric coding directly and omit the symbolic phase altogether. For a two address machine such as FADAC for which optimization or minimum latency programming is required, it would probably be best to compile symbolic coding and let the optimizer do the assembly.

RAVEL - The First Compiling Routine

In last June's report, RAVEL and SKEIN were the two compiling routines that made up the YARN compiler. This compiler was developed by P. Z. Ingerman on a contract sponsored jointly by the Air Force Office of Scientific Research and the National Science Foundation.* This work is well documented elsewhere and the discussion here centers mainly on the application of this compiler to the automatic checkout language.[14,15,17,19] Figure 3.1a shows the recursive subroutine RAVEL (α, β, γ) where α, β and γ are three parameters that must be assigned values before the routine is called. The program calls upon itself as a subroutine at two points. The first, RAVEL (A[h], S[J,2], 1) sets the values of α, β and γ to A[h], S[J,2] and 1, respectively; while the second RAVEL (S[J,2], GOAL, 2) sets α, β and γ to S[J,2], GOAL and 2. (The terms A[h] and S[J,2] are defined below).

*This work was done under Air Force contract AF-49(638)-951, "Research into Mechanical Languages" and N.S.F. Grant G-14096 placed with the Mechanical Languages Projects Office of the Moore School of Electrical Engineering, Univ. of Penna. Dr. Saul Gorn is the principal investigator on this contract.

Figure 3.1a RAVEL (α, β, γ) - recursive



Initially, α = A [0]
β = PROGRAM
γ = 0
K = 0
p = 0
h = 0

A[h]	is the input compiler language program.
A[0]	is the first character of the input.

Figure 3.1a is a modification of the flowchart on page 52 (Figure 6) of the June 1961 report. The mechanization of a recursive routine requires a lot of bookkeeping and the use of pushdown storage. Figure 3.1b shows the same routine in non-recursive form where the subscript "i" designates the level of the pushed down storage and symbols subscripted with "i" are the pushed down variables.

Table 3.0

Definition of Symbols Used in Figures 3.1a and 3.1b

i	Counter on the level of the main pushdown storage	
R_i	Stratification counter	Parameters in the main pushdown storage.
J_i	Syntax table address	
$IVAR_i$	Name of the current node	
$GOAL_i$	Name of the current goal	
$Exit_i$	Exit switch	
S	Syntax matrix	
K	Label counter	
p	Counter on the level of the label pushdown storage	
Mp	Label pushdown storage	
h	Counter on the input list	
A_h	The input list (A[0] is the first word, string or symbol input A[1] is the second... etc.)	
ϕ	The empty string (which may appear in the third column of the Syntax matrix)	
a	Output control switch	

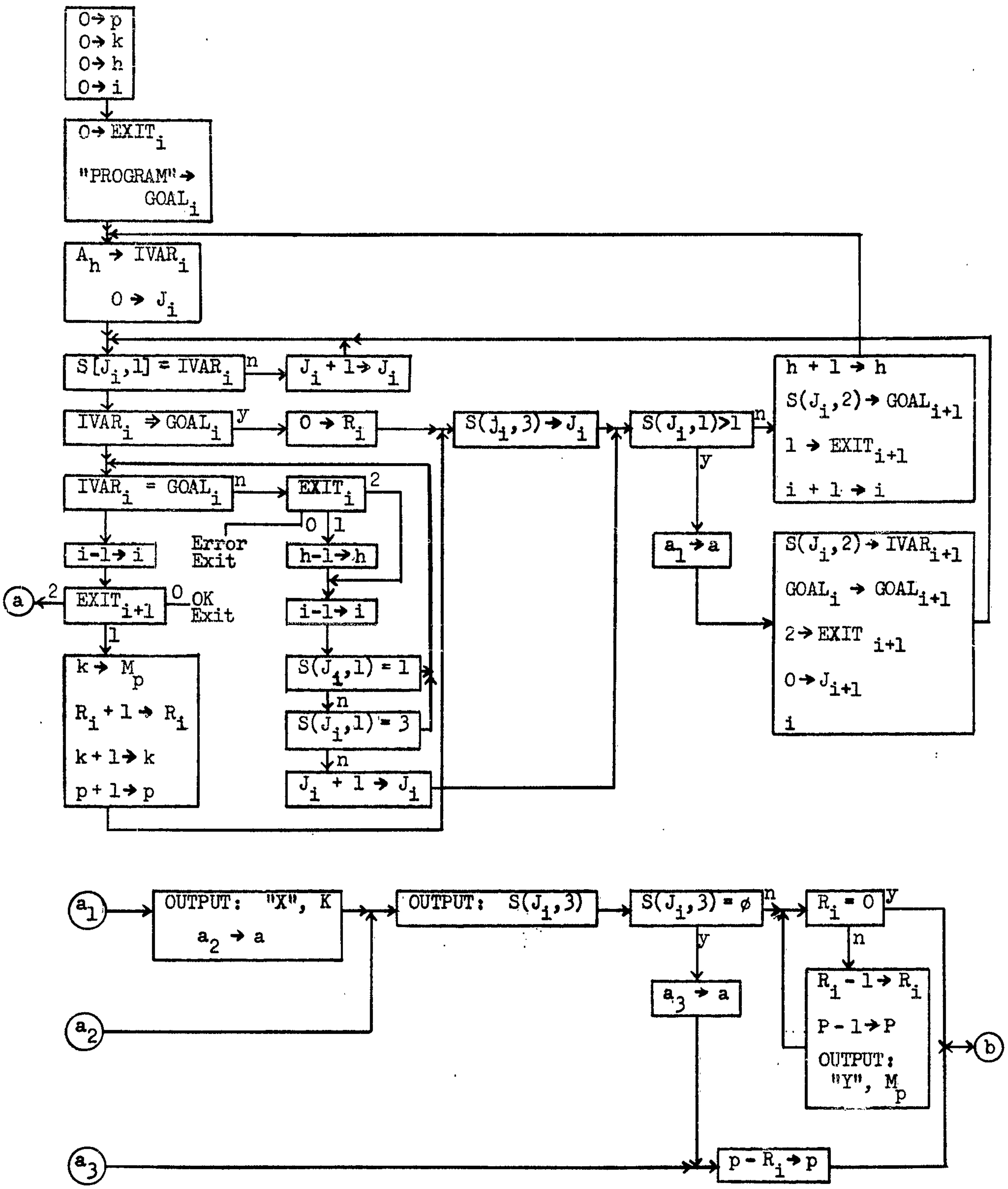
A language specified in the Backus Normal Form (BNF) can be diagrammed as a tree with recurrent nodes. For example the language:

<a> :=B|B<c>

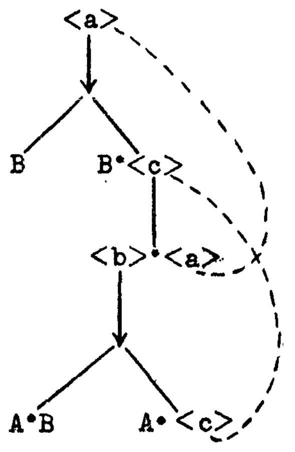
<b> :=AB|A<c>

<c> :=<b><a>

Figure 3.1b - RAVEL - non-recursive form



can be diagrammed



and, given a string such as, for example, BBABBAB which is of type <a> there is a path through the tree to generate this string. If the language is unambiguous, this path will be unique.[7,8,9,11]

The RAVEL routine takes an input program (or statement on procedure, etc.) and using the complete tree of the language, traces the path of generation. In this generation tree there is associated with each node a unique label or name and the output of RAVEL is a list of these node labels together with such local cross references as are necessary for the second compiling routine to be able to translate this list into computer code. Appendix 3 contains a discussion of a modification to RAVEL to increase efficiency.

The complete tree of the language would be a quite large and unwieldy combination of lines, arrows and labels; rather than actually draw the tree, the syntax is put into a tabular form and then into a 3 by (many)-matrix machine oriented form. These are called the S-table and S-matrix and the method of their generation is given in Appendix 1. Table 3.1 is the S-table for the complete algebraic language and Table 3.2 is the Matrix Form of 3.1.

In Figure 3.1a there is a block containing the expression "IVAR $\Rightarrow$ GOAL" (read: does IVAR lead to GOAL). The answer to this is given in a binary matrix. The answer is "YES" if the space corresponding to IVAR on the left and GOAL at the top has a "1" otherwise, the answer is "NO". A sample table of this sort is given as Table 3 (p. 60) of the June 1961 report and the method of generation is given in Appendix I.

Table 3.1 Syntax - Linear Format - S-table

+	adop	A			
-	adop	B			
*	mulop	C			
/	mulop	D			
(	arex	)	prim	E	
(	condpart	)	condlist	F	
(	procnalist	)	sublist	G	
(	varlist	)	formparlist	H	
(	varylist	)	comvarlist	I	
<	rel	J			
≤	rel	K			
=	rel	L			
≥	rel	M			
>	rel	N			
≠	rel	O			
.	end	P			
.	string	end	Q		
;	dumstat		R		
ABS	fnam		S		
actparlist	,	param	actparlist	T	
adop	term	arex	U		
ARCTAN	fnam	U			
arex	adop	term	arex	W	
arex	param	X			
CODE	string	END	;	codestat	Y
codestat	unstat	Z			
comment	unstat	AA			
COMMENT	:	string	;	comment	AB
comvarlist	(	varylist	)	comvarlist	AC
cond	condlist	AD			
cond	condpart	AE			

condlist	OR	condlist	condlist	AF		
condpart	AND	cond	condpart	AG		
condstat	unstat	AH				
constant	number	AI				
COS	fnam	AJ				
dumstat	unstat	AK				
E	formparlist	AL				
E	sublist	AM				
ENTIER	fnam	AN				
eqlist	,	equation	eqlist	AO		
equation	eqlist	AP				
EXP	fnam	AQ				
fact	**	prim	fact	AR		
fact	term	AS				
fnam	(	arex	)	funex	AT	
fnam	,	proclist	procnalist	AU		
fnam	procnalist	AV				
formparlist	,	sublist	prochead	AW		
funex	prim	AX				
GO TO	loc	transtat	AY			
HALT	loc	transtat	AZ			
IF	condlist	THEN	condtail	;	condstat	BA
LET	eqlist	;	letstat	BB		
letstat	unstat	BC				
LN	fnam	BD				
LOG	fnam	BE				
matrix	prim	BF				
matrix	varnam	BG				
MEAN	fnam	BH				
MEDIAN	fnam	BI				
mval	prim	BJ				
namelist	,	varnam	namelist	BK		
number	mval	BL				

number	value	BM				
param	actparlist	BN				
prim	fact	BO				
prim	incr	BP				
prim	lowlim	BQ				
prim	rel	prim	cond	BR		
prim	uplim	BS				
procc	loc	BT				
procc	prim	BU				
procedure	procseq	BV				
PROCEDURE	procid	prochead	stlist	END	procedure	BW
procid	(	actparlist	)	;	procc	BX
procid	,	procnalist	procnalist	BY		
procid	procnalist	BZ				
procseq	procedure	procseq	CA			
SIGN	fnam	CB				
simvar	mval	CC				
simvar	varnam	CD				
SIN	fnam	CE				
SQRT	fnam	CF				
START	procseq	STOP	program	CG		
stat	stlist	CH				
statlab	:	unstat	stat	CI		
statlab	loc	CJ				
string	procid	CN				
string	statlab	CO				
string	varid	CP				
stlist	ELSE	stlist	end	condtail	CK	
stlist	end	condtail	CL			
stlist	stat	stlist	CM			
term	arex	CQ				
term	mulop	fact	term	CR		
transtat	unstat	CS				

unstat	stat	CT					
value	varylist	CU					
varid	(	prim	)	vector	CV		
varid	(	prim	,	prim	)	matrix	CW
varid	simvar	CX					
varlist	,	varnam	varlist	CY			
varnam	:=	arex	equation	CZ			
varnam	namelist	DA					
varnam	param	DB					
varnam	value	DC					
varnam	varlist	DD					
VARY	namelist	OVER	comvarlist	IN	stlist	end	
					;	vbyl	DE
VARY	varnam	FROM	lowlim	BY	incr	UNTIL	
			uplim	IN	stlist	end	
					;	vbyl	DF
varylist	,	value	varylist	DG			
varystat	unstat	DH					
vbyl	varystat	DI					
vbys	varystat	DJ					
vector	prim	DK					
vector	varnam	DL					

Table 3.2 Syntax - Matrix Format - S-matrix

0	0	+	81
1	0	-	82
2	0	*	83
3	0	/	84
4	0	(	85
5	0	<	90
6	0	≤	91
7	0	=	92
8	0	≥	93
9	0	>	94
10	0	≠	95
11	0	.	96
12	0	;	98
13	0	ABS	99
14	0	actparlist	100
15	0	adop	101
16	0	ARCTAN	102
17	0	arex	103
18	0	CODE	105
19	0	codestat	106
20	0	comment	107
21	0	COMMENT	108
22	0	comvarlist	109
23	0	cond	110
24	0	condlist	112
25	0	condpart	113
26	0	condstat	114
27	0	constant	115
28	0	COS	116
29	0	dumstat	117
30	0	E	118
31	0	ENTIER	120
32	0	eqlist	121
33	0	equation	122
34	0	EXP	123
35	0	fact	124
36	0	fnam	126
37	0	formparlist	129
38	0	funex	130
39	0	GO TO	131
40	0	HALT	132
41	0	IF	133
42	0	LET	134
43	0	letstat	135
44	0	LN	136
45	0	LOG	137
46	0	matrix	138
47	0	MEAN	140
48	0	MEDIAN	141
49	0	mval	142
50	0	namelist	143
51	0	number	144
52	0	param	146
53	0	prim	147
54	0	procc	152
55	0	procedure	154
56	0	PROCEDURE	155
57	0	procid	156
58	0	procseq	159
59	0	SIGN	160
60	0	simvar	161
61	0	SIN	163

Table 3.2 (Con't)

62	0	SQRT	164
63	0	START	165
64	0	stat	166
65	0	statlab	167
66	0	stlist	169
67	0	string	172
68	0	term	175
69	0	transtat	177
70	0	unstat	178
71	0	value	179
72	0	varid	180
73	0	varlist	182
74	0	varnam	183
75	0	VARY	188
76	0	varylist	190
77	0	varystat	191
78	0	vbyl	192
79	0	vbys	193
80	1	vector	194
81	3	adop	A
82	3	adop	B
83	3	mulop	C
84	3	mulop	D
85	0	arex	196
86	0	condpart	197
87	0	procnalist	198
88	0	varlist	199
89	1	varylist	200
90	3	rel	J
91	3	rel	K
92	3	rel	L
93	3	rel	M
94	3	rel	N
95	3	rel	O
96	2	end	P
97	1	string	201
98	3	dumstat	R
99	3	fnam	S
100	1	,	202
101	1	term	203
102	3	fnam	V
103	0	adop	204
104	3	param	X
105	1	string	205
106	3	unstat	Z
107	3	unstat	AA
108	1	:	206
109	1	(	207
110	2	condlist	AD
111	3	condpart	AE
112	1	OR	208
113	1	AND	209
114	3	unstat	AH
115	3	number	AI
116	3	fnam	AJ
117	3	unstat	AK
118	2	formparlist	AL
119	3	sublist	AM
120	3	fnam	AN
121	1	,	210
122	3	eqlist	AP
123	3	fnam	AQ
124	0	**	211
125	3	term	AS

Table 3.2 (Con't.)

126	0	(	212	158	3	procnalist	BZ
127	0	,	213	159	1	procedure	224
128	3	procnalist	AV	160	3	fnam	CB
129	1	,	214	161	2	mval	CC
130	3	prim	AX	162	3	varnam	CD
131	1	loc	215	163	3	fnam	CE
132	1	loc	216	164	3	fnam	CF
133	1	condlist	217	165	1	procseq	225
134	1	eqlist	218	166	3	stlist	CH
135	3	unstat	BC	167	0	:	226
136	3	fnam	BD	168	3	loc	CJ
137	3	fnam	BE	169	0	ELSE	227
138	2	prim	BF	170	0	end	228
139	3	varnam	BG	171	1	stat	229
140	3	fnam	BH	172	2	procid	CN
141	3	fnam	BI	173	2	statlab	CO
142	3	prim	BJ	174	3	varid	CP
143	1	,	219	175	2	arex	CQ
144	2	mval	BL	176	1	mulop	230
145	3	value	BM	177	3	unstat	CS
146	3	actparlist	BN	178	3	stat	CT
147	2	fact	BO	179	3	varylist	CU
148	2	incr	BP	180	0	(	231
149	2	lowlim	BQ	181	3	simvar	CX
150	0	rel	220	182	1	,	232
151	3	uplim	BS	183	0	:=	233
152	2	loc	BT	184	2	namelist	DA
153	3	prim	BU	185	2	param	DB
154	3	procseq	BV	186	2	value	DC
155	1	procid	221	187	2	varlist	DD
156	0	(	222	188	0	namelist	234
157	0	,	223	189	1	varnam	235

Table 3.2 (Con't.)

190	1	,	236
191	3	unstat	DH
192	3	varystat	DI
193	3	varystat	DJ
194	2	prim	DK
195	3	varnam	DL
196	1	)	237
197	1	)	238
198	1	)	239
199	1	)	240
200	1	)	241
201	3	end	Q
202	1	param	242
203	3	arex	U
204	1	term	243
205	1	END	244
206	1	string	245
207	1	varylist	246
208	1	condlist	247
209	1	cond	248
210	1	equation	249
211	1	prim	250
212	1	arex	251
213	1	procnalist	252
214	1	sublist	253
215	3	transtat	AY
216	3	transtat	AZ
217	1	THEN	254
218	1	;	255
219	1	varnam	256
220	1	prim	257
221	1	prochead	258
222	1	actparlist	259
223	1	procnalist	260
224	3	procseq	CA
225	1	STOP	261
226	1	unstat	262
227	1	stlist	263
228	3	condtail	CL
229	3	stlist	CM
230	1	fact	264
231	1	prim	265
232	1	varnam	267
233	1	arex	268
234	1	OVER	269
235	1	FROM	270
236	1	value	271
237	3	prim	E
238	3	condlist	F
239	3	sublist	G
240	3	formparlist	H
241	3	comvarlist	I
242	3	actparlist	T
243	3	arex	W
244	1	;	272
245	1	;	273
246	1	)	274
247	3	condlist	AF
248	3	condpart	AG
249	3	eqlist	AO
250	3	fact	AR
251	1	)	275
252	3	procnalist	AU
253	3	prochead	AW
254	1	condtail	276
255	3	letstat	BB
256	3	namelist	BK
257	3	cond	BR

Table 3.2 (Con't.)

258	1	stlist	277	279	3	condtail	CK
259	1	)	278	280	3	vector	CV
260	3	procnalist	BY	281	1	prim	287
261	3	program	CG	282	1	IN	288
262	3	stat	CI	283	1	BY	289
263	1	end	279	284	3	condstat	BA
264	3	term	CR	285	3	procedure	BW
265	0	)	280	286	3	procc	BX
266	1	,	281	287	1	)	290
267	3	varlist	CY	288	1	stlist	291
268	3	equation	CZ	289	1	incr	292
269	1	comvarlist	282	290	3	matrix	CW
270	1	lowlim	283	291	1	end	293
271	3	varylist	DG	292	1	UNTIL	294
272	3	codestat	Y	293	1	;	295
273	3	comment	AB	294	1	uplim	296
274	3	comvarlist	AC	295	3	vbyl	DE
275	3	funex	AT	296	1	IN	297
276	1	;	284	297	1	stlist	298
277	1	END	285	298	1	end	299
278	1	;	286	299	1	;	300
				300	3	vbys	DF

Table 3.3 Abbreviations Used in Tables 3.1 and 3.2

actparlist	actual parameter list	mval	memory value
adop	adding operator	param	parameter
arex	arithmetic expression	prim	primary
codestat	code statement	procc	procedure call
comvarlist	compound vary list	prochead	procedure heading
cond	condition	procid	procedure identifier
condlist	condition list	procnalist	procedure name list
condpart	condition part	procseq	procedure sequence
condstat	condition statement	rel	relation
condtail	condition tail	simvar	simple variable
dumstat	dummy statement	statlab	statement label
eqlist	equation list	stat	statement
fact	factor	stlist	statement list
formparlist	formal parameter list	sublist	subroutine list
fnam	function name	transtat	transfer statement
funex	function expression	unstat	unlabelled statement
incr	increment	uplim	upper limit
letstat	let statement	varlist	variable list
loc	location	varnam	variable name
lowlim	lower limit	varystat	vary statement
mulop	multiplying operator	vbyl	vary by list
		vbys	vary by steps

SKEIN - The Second Compiling Routine

As mentioned above, the output from RAVEL is a list of node labels and associated with each node is a symbolic string. SKEIN processes the strings named by the RAVEL output and from them, the coding is formed. The string language is briefly described on pages 40 to 43 of the June, 1961 report and in more detail in reference 19.

The flowchart of SKEIN given last year was in recursive format. Figure 3.3 shows SKEIN in non-recursive form with explicit pushdown storage. Table 3.4 shows changes in notation in the string language and SKEIN symbolism.

Table 3.4 Notation Changes

String language		Skein Symbolism	
New Notation	Old Notation	New Notation	Old Notation
[	$	r	θ
]	$\underline{3}$	S_{j2}	M_j
(	$\underline{6}$	S_{j3}	N_j
₵	$\underline{5}$		
)	$\underline{2}$		
e	¢		
.b	#		

The method of writing strings in heuristic rather than algorithmic and requires a detailed knowledge of the language being compiled, of the computer being compiled for, and of the method of compiling. Appendix 2 contains a brief discussion of this and several examples. Mr. P. Z. Ingerman who developed the compiler we are using is in the process of writing a paper on SKEIN which includes a detailed description of the string language and a section on the writing of strings.[14]

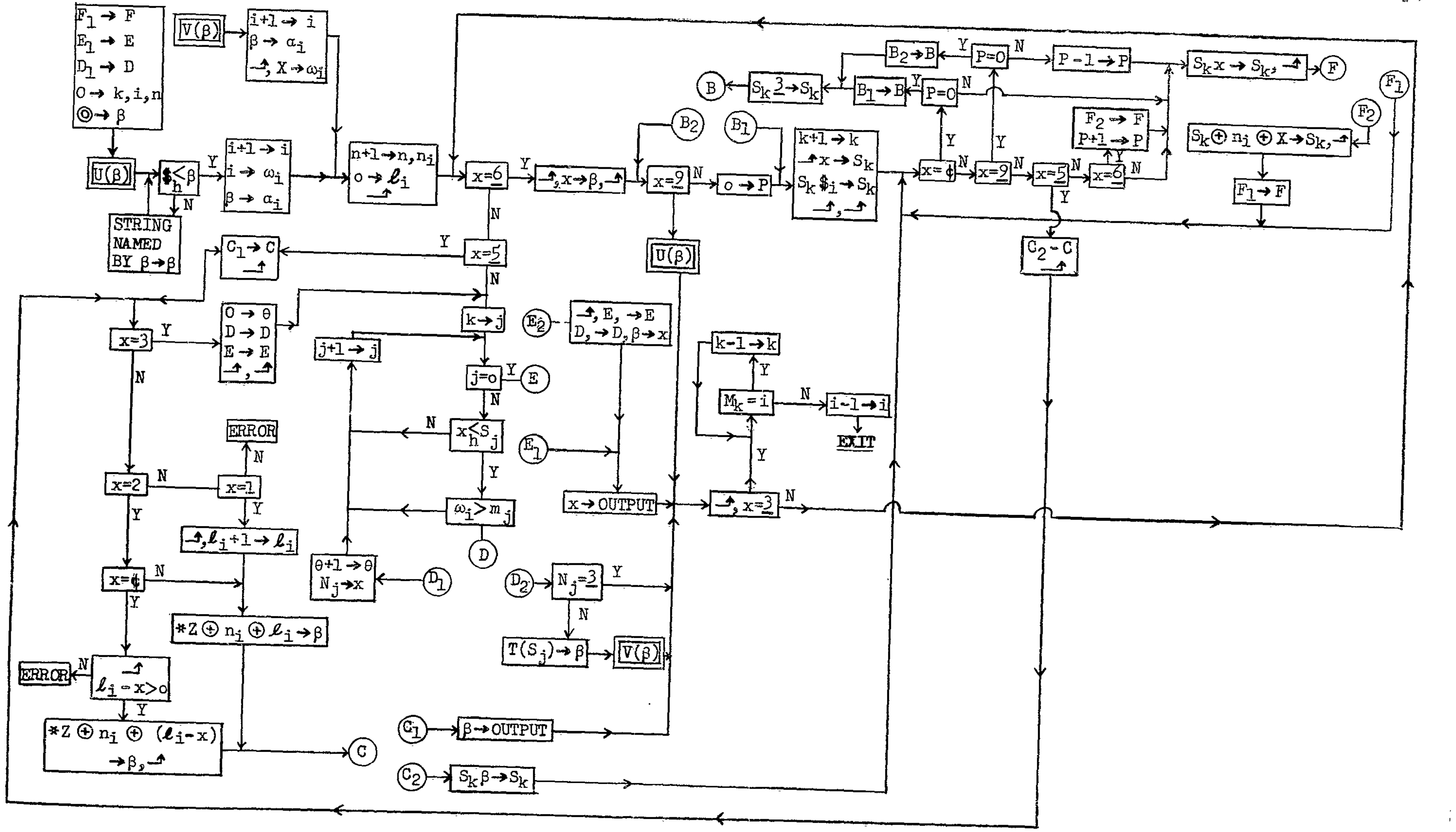


Figure 3.2a SKEIN - recursive

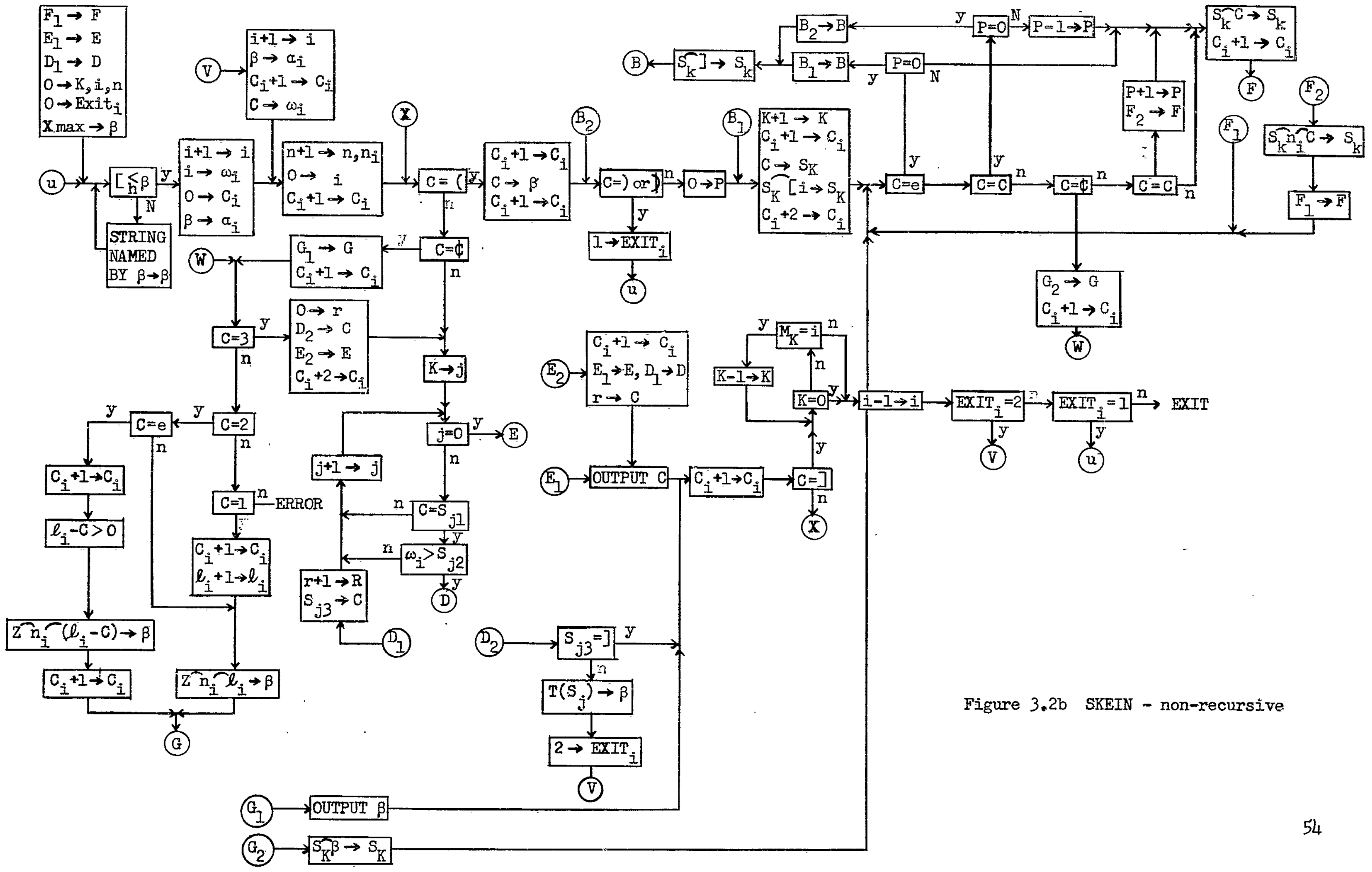


Figure 3.2b SKEIN - non-recursive

The Input Routine

Input to the compiling computer will most likely be in the form of punched card or punched tape. It will be necessary to delimit in some manner the special symbol strings which are to be treated as single characters (for example: "LET", "VARY", "ARCTAN", ":=", etc.) One possible way would be to place special symbols before and after these strings. This is however, dependant upon the input characteristics of the specific computer used for compiling.

The primary purpose of the input routine is to substitute numbers for letters and words so that the processing and table lookup will be more efficient.

Usually, numbers will be assigned to characters in the language in some orderly manner, for example, numbering the characters according to their first appearance in the S-matrix will simplify table look-up; and then a range of numbers is set aside for variable and procedure identifiers, another for statement labels, another for comments and another for constants. As words or characters are read into the computer, they are looked up in a table of allowed words or characters and if found the equivalent number is stored in the input list. If not found, they are identified (comment? constant? etc.) and assigned numbers. These numbers are then stored in the input list. Figure 3.3 roughly diagrams the procedure. Once a character string has been identified the same number is assigned to each succeeding occurrence of the same string.

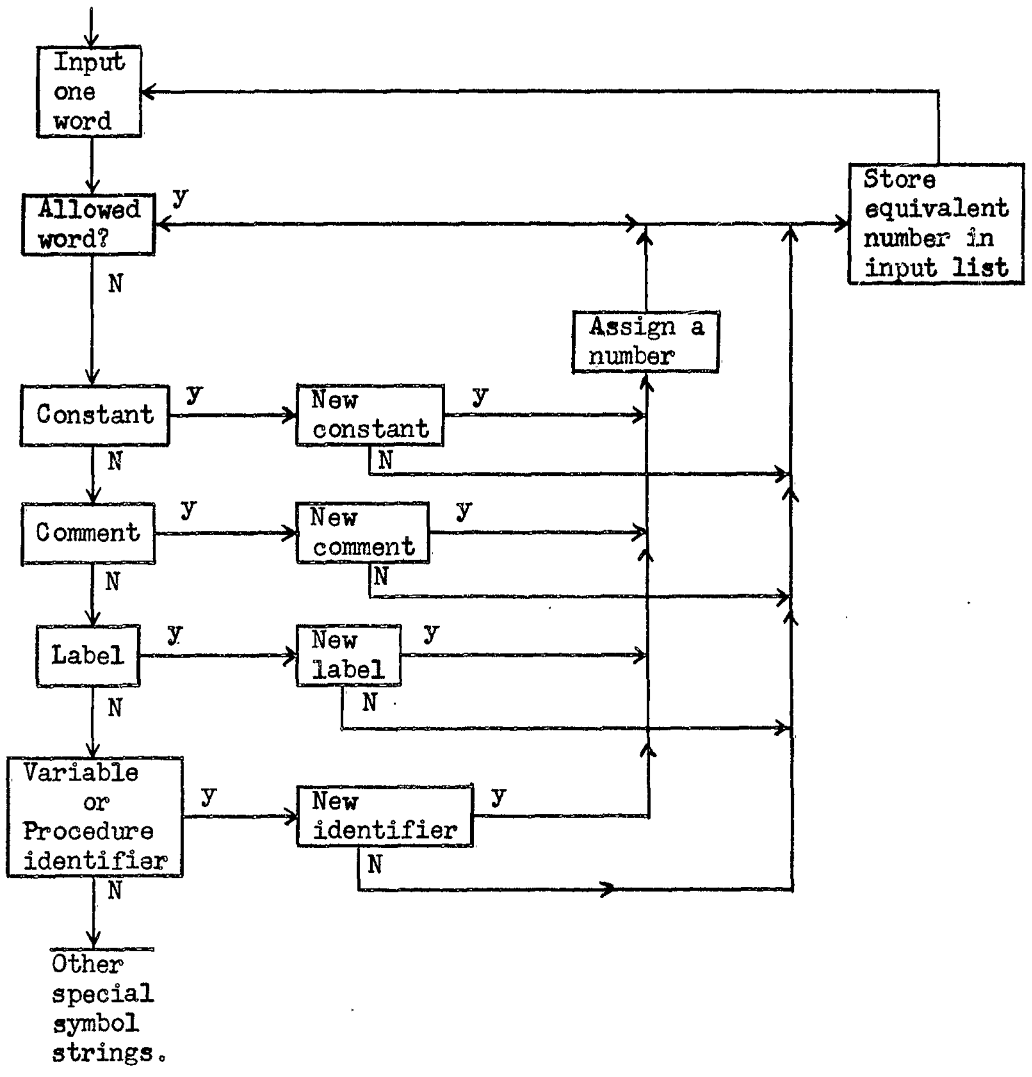


Figure 3.3 Schematic of Input Routine

Appendix 1

The Generation of the S-table, S-matrix and Binary Table for the "⇒" operation

The S-table is formed by re-writing the syntax of the language in a linear format and then alphabetising. Consider the language shown in Figure A1.0.

Figure A1.0 Sample Language

```
<a> ::= A B C
<b> ::= A B D | A B | B A
<c> ::= A<b> | B<a> | D
```

This is rewritten breaking up the "or" possibilities into a series of statements, dropping the metalinguistic brackets and writing the name on the left at the right, as in Figure A1.1.

Figure A1.1 Rewritten Language

A	B	C	a
A	B	D	b
A	B	b	
B	A	b	
A	b	c	
B	a	c	
D	c		

This is then alphabetised as in Figure A1.2 and the formulae are given symbols identifying them

Figure A1.2 S-table for Sample Languages

A	b	c	S1	
A	B	b	S2	
A	B	C	a	S3
A	B	D	b	S4
B	a	c	S5	
B	A	b	S6	
D	c	S7		

The S-matrix is formed from the S-table by an orderly transformation which is shown in progressive steps in Figure A1.3 (The numbers in the O'th column are for reference only and are not part of the S-matrix).

a) All names in the first column of the S-table are listed in the second column of the S-matrix (duplicated names are entered only once) to form the first <u>local list</u>.

b) Tag numbers are entered in the first column of the matrix. A tag of <u>1</u> means that this entry ends a local list and a tag of <u>0</u> means that the entry does not end a local list.

c) The names from the second column of the S-table are listed in the second column of the S-matrix. Duplicated names are entered only once only if all S-table entries to the left of them are identical. E.g. the three entries of <u>B</u> in the second column of the S-table all have the same entries to their left so they are only listed once.

d) The third column of the matrix will contain either an S-table identifying symbol (e.g. S1, S2, S3, etc. in this example; or A, B, C, D. ... etc. in the S-table for the language) or a number referring to another row in the matrix. The <u>A</u> in the first column of the S-table is followed by <u>b</u> in one row and <u>B</u> in three others. Hence, the <u>A</u> in the first row of the S-matrix is said to "refer to" and <u>b</u> and the <u>B</u> in rows 4 and 5 of the matrix. The <u>b</u> and <u>B</u> in rows 4 and 5 form a second local list and are tagged accordingly. Column 3 of row 1 has a 4 to indicate that the <u>A</u> is followed by the members of the local list beginning in row 4.

Similarly, the B in row 2 is followed by the local list consisting of a and A beginning in row 6 and the D is followed by the single entry local list in row 8. Whenever a character is entered into column 2 that is followed in the S-table by an identifier, that identifier is entered into column 3 and two is added to the tag digit to indicate the end of a production rule. (The separate lines in the S-table are known as production rules.)

e) The next set of entries in the second column come from the third column of the S-table. In this case every name is entered since no two identical names have identical entries to their left.

f) As above, local lists are formed, tagged and appropriate entries are put in column 3. The b of row 4 leads to the single entry list of row 9 which is c and this ends the production rule identified by S1. The B of row 5 leads to the 3 entry local list in 10, 11, 12. Row 10 column 3 contains an identifier so the tag digit of row 10 is increased by two.

g and h) The above processes are continued, using the next (in this case, the last) column of the S-table and this completes the S-matrix.

In the S-table for the complete compiler language, there are entries that have up to 13 items. In forming the S-matrix, the above procedures must be repeated twelve times.

The table for the "$\Rightarrow$" operation is formed from the S-table. Figure A1.4 contains a part of the S-table of Table 3.1. $\alpha \Rightarrow \beta$ is read "does α lead to β" and α is called the antecedent and β, the consequent. In the table the antecedents are listed along the left margin

Figure A1.3 Steps in Formation of the S-matrix

a)

	1	2	3
1		A	
2		B	
3		D	

b)

	1	2	3
1	0	A	
2	0	B	
3	1	D	

c)

	1	2	3
1	0	A	
2	0	B	
3	1	D	
4		b	
5		B	
6		a	
7		A	
8		c	

d)

	1	2	3
1	0	A	4
2	0	B	6
3	1	D	8
4	0	b	
5	1	B	
6	0	a	
7	1	A	
8	3	c	S7

e)

	1	2	3
1	0	A	4
2	0	B	6
3	1	D	8
4	0	b	
5	1	B	
6	0	a	
7	1	A	
8	3	c	S7
9		c	
10		b	
11		C	
12		D	
13		c	
14		b	

f)

	1	2	3
1	0	A	4
2	0	B	6
3	1	D	8
4	0	b	9
5	1	B	10
6	0	a	13
7	1	A	14
8	3	c	S7
9	3	c	S1
10	2	b	S2
11	0	C	
12	1	D	
13	3	c	S5
14	3	b	S6

g)

	1	2	3
11	0	C	
12	1	D	
13	3	c	S5
14	3	b	S6
15		a	
16		b	

h)

	1	2	3
11	0	C	15
12	1	D	10
13	3	c	S5
14	3	6	S6
15	3	a	S3
16	3	6	S4

and the consequents across the top. Antecedents are the names and symbols that appear in the first column of the S-table and consequents are the names that appear at the end of the rules. The $\Rightarrow$ table is begun by entering a 1 into each place where antecedent and consequent correspond to the beginning and end of a rule in the S-table; as in Figure A1.5.

Figure A1.4 Part of Table 3.1 Used for Example in Text

+	adop			
-	adop			
*	mulop			
/	mulop			
(	arex	)	prim	
ABS	fnam			
adop	term	arex		
arex	adop	term	arex	
COS	fnam			
fact	**	prim	fact	
fact	term			
fnam	(	arex	)	funex
funex	prim			
matrix	prim			
mval	prim			
numb	mval			
prim	fact			
term	arex			
term	mulop			
simvar	mval			
vector	prim			

Next the rule: $(\alpha \Rightarrow \beta)\cdot(\beta \Rightarrow \gamma) > (\alpha \Rightarrow \gamma)$ is applied to Figure A1.5 one row at a time to give Figure A1.6. For example (+ $\Rightarrow$ adop) and (adop $\Rightarrow$ arex) therefore (+ $\Rightarrow$ arex); (COS $\Rightarrow$ fnam) and (fnam $\Rightarrow$ funex) therefore (COS $\Rightarrow$ funex).

Figure A1.5 Initial Entries in ➔ table

	adop	arex	fact	fnam	funex	mulop	mval	prim	term
+	1								
-	1								
*						1			
/						1			
(								1	
ABS				1					
adop		1							
arex		1							
COS				1					
fact			1						1
fram					1				
funex								1	
matrix								1	
mval								1	
numb							1		
prim			1						
term		1							1
simvar							1		
vector								1	

Figure A1.6 Extension of ➔ table

	adop	arex	fact	fnam	funex	mulop	mval	prim	term
+	1	1							
-	1	1							
*						1			
/						1			
(			1					1	
ABS				1	1				
adop		1							
arex		1							
COS				1	1				
fact		1	1						1
fram					1			1	
funex			1					1	
matrix			1					1	
mval			1					1	
numb			1				1	1	
prim		1	1						1
term		1							1
simvar			1				1	1	
vector		1	1					1	1

The rule is then applied to the rows in the modified table and repeated until there is no further change. Figure A1.7 shows the complete table.

Figure A1.7 Complete ⇛ table for the S-table of Figure A1.4

	adop	arex	fact	fnam	funex	mulop	mval	prim	term
+	1	1							
-	1	1							
*						1			
/						1			
(		1	1						
ABS		1	1	1	1			1	1
adop		1							
arex		1							
COS		1	1	1	1			1	1
fact		1	1						1
fram		1	1		1			1	1
funex		1	1					1	1
matrix		1	1					1	1
mval		1	1					1	1
numb		1	1				1	1	1
prim		1	1						1
term		1							1
simvar		1	1				1	1	1
vector		1	1					1	1

Appendix 2

Strings and Their Interpretation

The following strings have thus far been written

```
A:  ø

E:  [(2e*Tb*T*I)]
U:  [(1)(2)]
W:  [(3)HT(β3e*T)(1e*Tb*T*I)
         (2)H0128 BT(β3e*T)R0012 U0000 A0000]

AO:  [(3)(1)]
AS:  [(1)]
BJ:  [B(1)]
BO:  [(1)]
CC:  [(1)]
CD:  [H(1)]
CQ:  [(1)]
CX:  [(1)]
CZ:  [(1)(3)]
```

Where: (n) means "replace this string by what comes n strings below"

(neαbβ) means "replace this string by what comes n strings below except where α appears, put β."

(β3eα) means replace this string by an integer one less than the number of characters in α ("*T" and "*I" are considered to be single characters).

The method of generation of strings that has been used is to simulate RAVEL for various input statements and, knowing the coding we wish to have generated for the given statement, write strings that will do just that.

Consider string "AS": when a factor becomes a term according to the definition in 2.B.1 of the language, no coding need be generated since both factors and terms are held in the accumulator. The string is [(1)] which essentially says "go on to the next string" when a memory value becomes a primary, however, this entails the bringing to the accumulator of a word in memory. The string for this is [B(1)] where the B is the Libratrol-500 "Bring" code. For more examples of strings, including a completely simulated statement, see the June 1961 report.

Appendix 3

Modifications to RAVEL

The use of an input routine to label and identify the strings and symbols in the input simplifies the processing. In fact, sections 1.A and 1.B.1 of the language can be omitted from the S-table and S-matrix. In consequence RAVEL must be modified so that when a symbol string that is identified as a variable identifier, statement label, procedure identifier, etc. is entered into the pushdown storage, the following occurs: R_i is set equal to zero, some special symbol is placed in the J_i column and the i is increased by 1; $GOAL_{i-1}$ goes into $GOAL_i$ the identifier associated with the symbol string (i.e. varid, statlab, etc.) is put into $IVAR_i$, "2" into $EXIT_i$ and control transfers to the $IVAR_i \Rightarrow GOAL_i$ section. It is understood that $IVAR_i \Rightarrow GOAL_i$ is <u>never</u> true when $IVAR_i$ is a symbol string.

When J_i is a special symbol and the "Output $S(J_i, 3)$" box is reached, the actual symbol string identified by the number in $IVAR_i$ is put out.

Appendix 4

Libratrol-500 Floating Point Arithmetic and Square Root

The coding is given in relocatable form. For the arithmetic the first operand is in the accumulator while the second is stored in the base address + 128. For the square root, the operand is in the accumulator.*

The format is qq.mmmmm where mmmmm is the normalized mantissa and qq is the exponent of z (complement notation is used for both.)

The output is a number in the accumulator in the above format.

The routine uses four tracks and no additional storage.

Calling sequences:

```
Arithmetic:  R Lo+12
             U Lo
             X 0000 (where X is A, S, M or D, depending upon
                     the operation desired.)

Square Root:  R Lo+330
              J Lo+300
```

*Square Root is done using Newton's Method.

Coding of Arithmetic Routines

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address & Notes
		0000	H	0132	1	a
		0001	E	0219	1	Extract exp a
		0002	H	0226	1	Exp a
		0003	B	0132	1	a
		0004	E	0220	1	Extract sign-mantissa a
		0005	U	0336	1	
		0006	B	0128	1	b
		0007	E	0219	1	Extract exp b
		0008	H	0227	1	Exp b
		0009	B	0128	1	b
		0010	E	0220	1	Extract sign-mantissa b
		0011	U	0341	1	
		0012	B	[ ]	1	Operation code
		0013	U	0015	1	
		0014	Z	0211	1	
		0015	S	0223	1	6 at 15
		0016	T	0150	1	if neg divide
		0017	S	0224	1	2 at 15
		0018	T	0113	1	if neg multiply
		0019	H	0130	1	if + add or subtract
		0020	B	0226	1	exp a
		0021	S	0227	1	exp b
		0022	T	0033	1	if neg b>a
		0023	S	0215	1	1 at 7
		0024	T	0044	1	if neg b=a
		0025	H	0129	1	a-b-1 = counter a>b
		0026	B	0229	1	b mantissa
		0027	M	0214	1	1 at 1
		0028	H	0229	1	b mantissa
		0029	B	0129	1	counter

Program Input Codes	Stop	Location	Instruction		Stop	Contents of Address & Notes
			Operation	Address		
		0030	S	0215	1	1 at 7
		0031	T	0044	1	if neg a = b
		0032	U	0025	1	
		0033	B	0227	1	b exp
		0034	S	0226	1	a exp
		0035	S	0215	1	1 at 7
		0036	H	0129	1	b-a-1 = counter b>a
		0037	B	0228	1	a mantissa
		0038	M	0214	1	1 at 1
		0039	H	0228	1	a mantissa
		0040	B	0129	1	counter
		0041	S	0215	1	1 at 7
		0042	T	0047	1	if neg a = b
		0043	U	0036	1	
		0044	B	0226	1	a exp
		0045	H	0230	1	final exp
		0046	U	0049	1	
		0047	B	0227	1	b exp
		0048	H	0230	1	final exp
		0049	B	0228	1	a mantissa
		0050	M	0214	1	1 at 1
		0051	H	0228	1	a shifted right one place
		0052	B	0229	1	b mantissa
		0053	M	0214	1	1 at 1
		0054	H	0229	1	b shifted right one place
		0055	B	0130	1	code (-8)
		0056	S	0225	1	7 at 15
		0057	T	0062	1	if neg add - if + subtract
		0058	B	0228	1	a mantissa
		0059	S	0229	1	b mantissa
		0060	H	0231	1	a - b = final mantissa
		0061	U	0101	1	
		0062	B	0228	1	a mantissa
		0063	A	0229	1	b mantissa

Program Input Codes	Stop	Location	Instruction		Stop	Contents of Address & Notes
			Operation	Address		
		0100	H	0231	1	a+b = final mantissa
		0101	U	0104	1	
		0102			1	
		0103			1	
		0104	B	0230	1	final exp
		0105	A	0215	1	1 at 7
		0106	T		1	if neg overflow exponent!
		0107	H	0230	1	final exp
		0108	U	0207	1	
Multiply		0109	XC	6363	1	
		0110	S	0131	1	
		0111	U	0134	1	
		0112			1	
		0113	B	0226	1	exp a
		0114	S	0221	1	64 at 7
		0115	A	0227	1	exp b
		0116	T		1	if neg overflow exponent!
		0117	H	0230	1	final exp
		0118	B	0228	1	a mantissa
		0119	M	0229	1	b mantissa = ab at 14
		0120	N	0348	1	1 at 24
		0121	H	0231	1	final mantissa
		0122	B	0228	1	a mantissa
		0123	N	0229	1	b mantissa = ab at 0 (n part)
		0124	M	0216	1	1 at 24
		0125	H	0131	1	ab mantissa at 24 (n part)
		0126	T	0109	1	if neg = |ab| + 1 at 24
		0127	U	0136	1	
		0128			1	b
		0129			1	counter
		0130			1	code (-8)
		0131			1	ab mantissa - (n part)

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address and Notes
		0132			1	a
		0133			1	b exp (-63)
		0134	A	0348	1	1 at 24
		0135	H	0131	1	ab mantissa at 24 (n part)
		0136	B	0231	1	final mantissa
		0137	A	0131	1	ab at 24 (n part)
		0138	H	0231	1	final mantissa
		0139	E	0218	1	extract all but 1st bit
		0140	S	0218	1	1 at 8
		0141	T	0143	1	if neg shift
		0142	U	0207	1	
		0143	B	0231	1	final mantissa
		0144	N	0217	1	1 at 30
		0145	H	0231	1	final mantissa
		0146	B	0230	1	final exp
		0147	S	0215	1	1 at 7
		0148	H	0230	1	final exp
		0149	U	0207	1	
Divide		0150	B	0227	1	b exp
		0151	S	0222	1	65 at 7
		0152	H	0133	1	b exp (-65)
		0153	B	0226	1	a exp
		0154	S	0133	1	b exp (-65)
		0155	T		1	if neg underflow exponent!
		0156	H	0230	1	final exp
		0157	B	0228	1	a mantissa
		0158	M	0214	1	1 at 1
		0159	D	0229	1	b mantissa
		0160	U	0352	1	
		0161	E	0218	1	extract all but 1st bit
		0162	S	0218	1	1 at 8
		0163	T	0201	1	if neg shift

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address and Notes
		0200	U	0207	1	
		0201	B	0231	1	final mantissa
		0202	N	0217	1	1 at 30
		0203	H	0231	1	final mantissa
		0204	B	0230	1	final exp.
		0205	S	0215	1	1 at 7
		0206	H	0230	1	final exp.
		0207	B	0012	1	
		0208	A	0213	1	Z0001
		0209	Y	0212	1	
		0210	U	0232	1	
		0211	A	0230	1	
		0212	U	[ ]	1	EXIT
		0213	XZ	0001	1	
)0000012		0214	4000	0000	1	1 at 1
		0215	100	0000	1	1 at 7
		0216		40	1	1 at 25
		0217		2	1	1 at 30
		0218	80	0000	1	1 at 8 = 1st bit extractor
		0219	7W00	0000	1	Exp extractor
		0220	80WW	WWWW	1	Sign - mantissa extractor
		0221	4000	0000	1	64 at 7
		0222	4100	0000	1	65 at 7
		0223	6	0000	1	6 at 15
		0224	2	0000	1	2 at 15
		0225	7	0000	1	7 at 15
		0226			1	a exp
		0227			1	b exp
		0228			1	a mantissa - sign
		0229			1	b mantissa - sign
		0230			1	final exp
		0231			1	final mantissa - sign

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address and Notes
		0232	B	0231	1	Mantissa
		0233	T	0249	1	No. is neg?
		0234	S	0217	1	1 at 30
		0235	T	0262	1	if neg no. = 0
		0236	U	0349	1	
		0237	S	0218	1	1 at 8
		0238	T	0241	1	if neg shift
		0239	B	0231	1	
		0240	U	0211	1	
		0241	B	0230	1	exp
		0242	S	0215	1	1 at 7
		0243	T		1	if neg overflow exponent!
		0244	H	0230	1	
		0245	B	0231	1	mantissa
		0246	N	0217	1	1 at 30
		0247	H	0231	1	
		0248	U	0236	1	
		0249	B	0261	1	Z0255
		0250	Y	0240	1	
		0251	XC	6363	1	
		0252	S	0231	1	-(mantissa)
		0253	H	0231	1	-(mantissa)
		0254	U	0236	1	
		0255	B	0014	1	Z0211
		0256	Y	0240	1	
		0257	XC	6363	1	
		0258	S	0231	1	+(mantissa)
		0259	E	0220	1	
		0260	U	0211	1	
		0261	Z	0255	1	
		0262	B	0214	1	64.00000
		0263	U	0212	1	TO EXIT

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address and Notes
Square Root		0300	H	0132	1	a
		0301	T	0333	1	halt if neg.
		0302	S	0217	1	1 at 30
		0303	T	0333	1	halt if neg.
		0304	B	0132	1	a
		0305	E	0219	1	Extract exp. a
		0306	H	0226	1	Exp a
		0307	B	0132	1	a
		0308	E	0220	1	Extract sign - mantissa a
		0309	H	0228	1	sign - mantissa a
		0310	B	0226	1	
		0311	E	0215	1	Extract of = 7 bit
		0312	S	0215	1	1 at 7
		0313	T	0320	1	if neg characteristic is even
		0314	B	0226	1	
		0315	A	0215	1	1 at 7
		0316	H	0226	1	
		0317	B	0228	1	
		0318	M	0214	1	1 at 1
		0319	H	0228	1	
		0320	B	0215	1	1 at 7
		0321	C	0347	1	X_i
		0322	S	0228	1	
		0323	D	0347	1	X_i
		0324	A	0347	1	X_i
		0325	M	0346	1	-1/2 at 0
		0326	T	0331	1	If neg $X_{i+1} = X_i + \left[-\frac{1}{2}\right]\left[-\frac{a}{X_i} + X_i\right]$
		0327	B	0226	1	if $+X_i = \sqrt{a}$
		0328	M	0214	1	5 at 8
		0329	A	0347	1	X_i
		0330	U	[ ]	1	
		0331	A	0347	1	X_i

Program Input Codes	Stop	Location	Instruction Operation	Instruction Address	Stop	Contents of Address and Notes
		0332	U	0321	1	
		0333	XZ	0000	1	a $\leqslant$ 0 To continue START
		0334	B	0214	1	bring 64.00000 = 0
		0335	U	0330	1	
		0336	T	0338	1	if neg add fill in
		0337	U	0339	1	
		0338	A	0219	1	2's complement fill in
		0339	H	0228	1	
		0340	U	0006	1	
		0341	T	0343	1	if neg add fill in
		0342	U	0344	1	
		0343	A	0219	1	2's complement fill in
		0344	H	0229	1	
		0345	U	0012	1	
)0000001		0346	J000	0000	1	-1/2 } for sq. root
		0347			1	X_i } for sq. root
)0000001		0348		80	1	1 at 24
		0349	B	0231	1	
		0350	E	0218	1	
		0351	U	0237	1	
		0352	M	0215	1	
		0353	H	0231	1	
		0354	U	0161	1	
		0355			1	
		0356			1	
		0357			1	
		0358			1	
		0359			1	
		0360			1	
		0361			1	
		0362			1	
		0363			1	

Appendix 5 - Part I

Algebraic Compiler Language Manual

1. General

A computer program written in this language consists of a series of statements each containing words and symbols. Each statement ends with a semicolon. This manual describes the allowable statements and discusses how they are combined into meaningful routines, subroutines and procedures.

2. The LET statement, variables and constants

2.1 The general form

Computations are programmed as LET statements which are of the form[1]:

```
LET "a variable":="an arithmetic expression"; for example:
LET B:=C+D ;
```

Before discussing this more fully, we shall describe the structure of the variables and constants that may appear in the statements.

2.11 Variables

A variable is a symbol which names a number. Simple variables are formed by strings of letters and digits beginning with a letter. For example:

A	AFL	S3T958
C26947	ROOT	ROOT1
RESULT	ALPHA	GAMMA60

2.12 Constants

A constant is a number and may be expressed either as an integer or as a decimal number. A decimal number may contain a decimal point

[1]In this manual, the quote symbols are often used to denote examples and class names. They do not appear in the actual program.

and may have a power of ten attached. A plus or minus sign may preceed the number if desired.

Examples:

Integers: 1 +3 -27 +7125 197654975

Decimals without powers of ten:

1.0 +3.0 -2.700 +.7125 .000197654975

Decimals with powers of ten[1]

1.0*10**3 = $1.0*10^3$

+3*10**-8 = $+3*10^{-8}$

-10**16 = -10^{16}

-.005*10**12 = $-.005*10^{12}$

Note that numbers do not have to be in normal form.

I.e., 30 = 30.0 = 3*10**1

= .3*10**2 = .03*10**3 etc.

2.13 Arithmetic expressions

An arithmetic expression is an algebraic formula and gives a rule for computing a numerical value. It is made up of variables, constants and the signs:

+	addition
-	subtraction
*	multiplication
/	division[2]
**	exponentiation.

[1]An asterisk denotes multiplication and a double asterisk denotes exponentiation.

[2]It is helpful to think of division as multiplication by the reciprocal i.e., $A/B = A*(B)^{-1}$ and $A/B/C = A*(B)^{-1}*(C)^{-1}$.

Left and right parenthesis are used for setting off parts of the expression.

Examples:

```
A+B         A+B*C          (A+B)*C
A/B+C/D     (A+B+C*D)/E    (A+B)**E
```

Operations are performed in the following orders:

First:	**
Second:	* and /
Third:	+ and -

and, where no order is expressed (as in A+B+C - D) evaluation is performed from left to right.

Examples:

A*B**C/D	means	$\frac{AB^C}{D}$
(A*B)**C/D	"	$\frac{(AB)^C}{D}$
(A*B)**(C/D)	"	$(AB)^{C/D}$
A*B**(C/D)	"	$AB^{C/D}$
A*B+C**E-D	"	$AB+C^E-D$
A*(B+C)**E-D	"	$A(B+C)^E-D$
A*B+C**(E-D)	"	$AB+C^{(E-D)}$
A+B**C-D/E*F	"	$A+B^C-D(E)^{-1} = A+B^C-\frac{DF}{E}$
A+B**((C-D)/E)*F	"	$A+B^{(C-D)/E}F$
(A+B**C-D/E*F	"	$(A+B)^C-\frac{DF}{E}$

2.14 Functions

There are a series of standard functions available for use in arithmetic expressions. In the following definitions Y stands for any arithmetic expression.

ABS(Y)	means	the absolute value of Y: -Y if Y is negative, otherwise Y.
SIGN(Y)	"	the signum of Y: +1 if Y is positive, -1 if Y is negative, 0 if Y=0.
SQRT(Y)	"	the positive square root of Y.
SIN(Y)	"	the sine of the number Y, or of the angle whose radian measure is Y.
COS(Y)	"	the cosine of the number Y.
ARCTAN(Y)	"	the principal value (in radians) of the arctangent of Y.
LN(Y)	"	the natural logarithm of Y: $\log_e Y$.
LOG(Y)	"	the logarithm to the base 10 of Y: $\log_{10} Y$.
EXP(Y)	"	the exponential of Y: e^Y.
ENTIER(Y)	"	the greatest integer not greater than Y.
MEDIAN(Y) MEAN(Y)	are used	primarily in conjunction with subscripted variables and will be discussed later. They are, however, defined for an expression Y which contains no free subscripts as follows:

MEDIAN(Y) = Y
MEAN(Y) = Y.

Examples:

```
SIN(THETA)          COS(3*J*PI)
SQRT(B*B-4*A*C)     LN(EXP(X))
```

2.15 The replacement Operator

The symbol ":=" is called the replacement operator and is used in equations. It is not the same as "=" but rather means "assign the value resulting from the computation on the right side to the variable on the left."

I.E. n:=n+1 is a valid operation,
but n=n+1 is an invalid statement.*

2.2 Computing, using the LET Statement

The basic let statement, as given above is of the form: LET "variable":=" arithmetic expression"; If more than one computation is to be performed, several equations may be combined into one LET statement by separating them with commas:

```
LET "variable":="arithmetic expression",
    "variable":="arithmetic expression",
    "variable":="arithmetic expression";
```

2.21 Examples

```
LET A:= B;
LET i:= i+2;
LET ROOT:=(-b+SQRT(b*b-4*a*c))/2*a;
LET T1:= A,
    A:= B,
    B:= T1;
LET RESULT:= (.5*SIN(PI*3*OMEGA)+COS(ALPHA))
              *(LN(EXP(SQRT(ARG))));
```

2.3 Subscription of Variables

It is sometimes useful and often necessary to refer to groups of numbers by a single name, using subscript numbers to denote the individual numbers. For example; matrices and determinants. This language allows the use of variables having single and double subscripts called respectively vectors and matrices. Since true subscripts cannot be conveniently written, the convention has been established that subscripts will be enclosed by parenthesis and placed behind the variable

*However some Programming languages (cf FORTRAN) allow it.

name. The two subscripts of a matrix are separated by a comma. E.g.

```
A(i)    FOOF(3)
A(i,j)  FOOF(3,7)
```

Subscripted variables may be used any place non-subscripted variables appear so long as the values of the subscripts are defined. I.e. A(i,j) does not mean anything unless numerical values have been assigned to i and to j.

An arithmetic expression may appear as a subscript e.g.

```
A(i+j, i-j)
FOOF((C+D/G, SQRT(P1))
```

When these subscripted variables are used, the expression or expressions in the subscript will be evaluated and if it is not an integer, the nearest smaller integer will be used (note that 4.001 and 4.999 will both be taken as 4). Memory space for vectors and matrices must be allocated by a special compiler-directing command, the DIMENSION statement, discussed below.

2.31 Subscript Functions

The MEAN(Y) and MEDIAN(Y) functions are meant to operate on expressions containing subscripted variables with one subscript undefined. The following are the complete definitions:

MEDIAN(Y) means a. If Y has no free subscripts, MEDIAN(Y) = Y.

b. If Y has one free subscript, then, if the values of Y be arranged in order of magnitude:

(1) If the greatest value of the subscript is odd, MEDIAN(Y) is equal to the value in the central position.

(2) If the greatest value of the subscript is even, MEDIAN(Y) is equal to the mean of the two values surrounding the central position.

c. If Y has more than one free subscript, MEDIAN(Y) = 0.

MEAN(Y) means a. If Y has no free subscripts, MEAN(Y) = Y.

b. If Y has one free subscript, then if n is the greatest value of this subscript, MEAN(Y) = (Y(1) + Y(2) + ... + Y(n))/n.

c. If Y has more than one free subscript MEAN(Y) = 0.

An arithmetic expression may be used as a subscript, provided the variables used in that expression all have values at the time of evaluation.

3. Control

A variety of statements are available to control the order in which computations are performed, and to facilitate repetitive calculations.

3.1 VARY-statements

VARY-statements are used to repeat certain calculations (or control statements) with changing values of one or several variables.

A VARY-statement contains *within it* statements to be repeated. The VARY-statement *and* each statement within it must end with a semicolon. In addition to this, the list of statements to be repeated must end with a period. The semicolon which denotes the end of the VARY-statement comes after this period. There are two types of VARY-statements: vary-by-steps and vary-over-list.

3.11 Vary-Over-List

A VARY-statement of the vary-over-list type will cause any specified list of statements to be repeated, with a specified variable or variables being assigned a new value from a list for each repetition of the list of statements. A VARY-statement of this type consists of the word VARY, followed by the variable(s) to be varied -- if there is more than one, they are separated by commas --, followed by the word OVER, followed by the list or lists of values over which the variable(s) is (are) to be varied, followed by the word IN followed by the list of statements to be repeated, followed by a period, followed by a semicolon. Remember that each of the statements within the list must also end with a semicolon.

```
          "variable"             "list of values"
VARY          or          OVER          or          IN  "statement list".;
       "list of variables"       "lists of values"
```

The list or lists of values are given in the following format. There is a separate list of values for each variable. If more than one variable is used, the separate lists should all contain the same number of values, for it is this number which determines the number of times the indicated statements are to be repeated. The lists are each enclosed in parenthesis, and are given in the same order in which the variables were given. They are <u>not</u> separated by commas; they follow one another directly. Each list, within its parenthesis, consists of a set of values - either numbers or variables but not arithmetic expressions - separated by commas. Any variables used in these lists of values must currently have numerical values.

Examples:

```
VARY J OVER (1, 2, 3, 4) IN
     LET X(J):=3*J; .;
```

After execution of this statement, the vector X would be:

(3, 6, 9, 12)

that is, for the first execution of the LET-statement (in this case the entire list of statements contains just this one statement) J would be set to 1; for the second J would be 2; for the third, 3; and for the fourth, 4. The vector X should have been dimensioned with a DIMENSION-statement at the beginning of the program. The first semicolon ends the LET-statement; the period ends the statement list; and the following semicolon ends the VARY-statement.

```
VARY K, M, N OVER (2, 3, 4)(A, B, C)(1.6, 1.7, 1.8) IN
     LET X(K):= X((K-1))/N;
     LET Y(K):= M*X(K); .
```

This statement would result in the following two vectors,

X: (R, R/1.6, R/((1.6)(1.7)), R/((1.6)(1.7)(1.8)))

where R was previously given a value, and

Y: (Y(1), AR/1.6, BR/((k,6)(1.7)), CR/((1.6)(1.7)(1.8)))

where Y(1) remains whatever it was before the execution of this statement. A, B, and C must have been assigned values before this statement was reached.

3.12 Vary-by-Steps

A VARY-statement of the vary-by-steps type will cause any specified list of statements to be repeated, with a specified variable being incremented or decremented by a certain amount for each repetition. A VARY-statement of this type consists of the word VARY, followed by the name of the variable to be incremented, followed by the word FROM, followed by the desired value of the variable for the first execution of the statement, followed by the word BY, followed by the amount of the increment, followed by the word UNTIL, followed by the final value the variable is to attain, followed by the word IN, followed by the list of statements to be repeated. The punctuation following this list is the same as for a vary over list VARY-statement.

```
    VARY "variable" FROM "lower limit" BY "increment" UNTIL "upper limit"
                                          IN "statement list".;
```

The quantities to be specified; initial value, increment, and final value, may be expressed as numbers or variables, provided any variables so used have numerical values at the time this statement is executed. In addition, any arithmetic expression may be used to specify these quantities.

Examples:

```
                    VARY J FROM 1 BY 1 UNTIL 4 IN
                       LET X(J):= 3*J; .;
```

This would yield the same result as the first example in 3.11.

In order to help make the vary statements more readable, a comment consisting of any symbol string not containing a semicolon may follow the period and preceed the semicolon. For example:

```
VARY X FROM 2 BY 1 UNTIL N IN
     VARY I FROM 1 BY 1 UNTIL (X-1) IN
          LET Z(X,I):=X+I,
              Z(I,X):=X*I; . end I loop;
     LET Z((X-1), (X-1)):=X-1; . end X loop;
```

This would create a matrix Z of order N wherein any entry below the major diagonal would be the sum of the subscripts of that entry; any entry above the major diagonal would be the product of its subscripts; any entry on the major diagonal would be one of its two equal subscripts. Notice that this statement fails to assign a value to Z(N,N). The VARY-statement, like any other statement, can be used in the statement list of a VARY-statement. No computation or other statement within the range of a VARY-statement should ever change the value of the variable being incremented by that statement. When nesting one vary statement within another, one must be particularly careful of the punctuation. In the above example, if the second LET-statement had been omitted and both VARY-statements had ended with the first LET-statement, the punctuation following it would have been as follows.

; .; .; (excluding the comments)

The first semicolon ends the LET-statement; the first period ends the statement list of the second, or "inner" VARY-statement; the second semicolon ends the inner VARY-statement; the second period ends the statement list of the first, or "outer" VARY-statement; the third semicolon ends this VARY-statement.

After a VARY-statement has been performed, that is after the statement list has been executed the proper number of times, the next instruction, in the statement sequence is executed.

3.2 Transfer Statements

A transfer statement is a statement which alters the sequence of execution from sequential order. In order that statements within a program may be easily referred to, and for mnemonic purposes, it is possible to label a statement by assigning a name to it. It is legitimate to label one or several statements without labeling them all. The only place a statement label is required is where it is referred to by a transfer statement, as described below. To name a statement, one precedes the statement by a statement label followed by a colon. A statement label is any string of letters and digits which begins with a letter.

3.21 Unconditional Transfers

There are two types of unconditional transfers, both of which unconditionally transfer control to the indicated location. A location is either a statement label, or a procedure (discussed in section 4.1).

3.211 GO TO-statements

A GO TO-statement is of the following form: the words GO TO, followed by a location (a statement label or a procedure call), followed by a semicolon. It has the effect of immediately transfering control to the indicated statement or procedure. For example:

```
GO TO B76Y;
```

would, as soon as it was reached, transfer control to the statement labeled B76Y.

3.212 HALT-statements

A HALT-statement consists of the word HALT, followed by a location, followed by a semicolon. Its function is identical to that of a GO TO-statement, except that, computer will stop before transferring.

3.22 Condition statements

It is possible to make the flow of a program conditional upon certain arithmetic conditions. These arithmetic conditions are of the form: an arithmetic expression, followed by a relation, followed by another arithmetic expression. The relation is one of the following six:

$$<, \leqslant, =, \geqslant, >, \neq$$

Some examples of conditions:

```
A>B
D(2, 3)<12
J**2≠-1
3=(A+B(J))
SIN(THETA*2*PI)⩽COS(OMEGA/2*PI).
```

A condition statement is of one of two forms. The first type consists of the word IF, followed by a condition or a combination of conditions, followed by the word THEN, followed by a list of statements to be performed if the specified conditions are satisfied (if the conditions are not satisfied, the statement after the condition statement is performed next).

```
IF "condition(s)" THEN "statement list".;
```

This statement list is of the same form as for a VARY-statement: each statement ends with a semicolon, the list ends with a period, which may be followed by a comment (the comment may not contain a semicolon), and the condition statement itself ends with a semicolon.

The other type of condition statement consists of the word IF, followed by a condition or a combination of conditions, followed by the word THEN, followed by a list of statements to be executed if the the conditions are satisfied, followed by the word ELSE, followed by a list of statements to be performed if the conditions are <u>not</u> satisfied, followed by a period (in a condition statement of this type, the first statement list, after the word THEN, is <u>not</u> followed by a period) and a comment if desired, followed by a semicolon.

```
IF "condition(s)" THEN "statement list" ELSE "statement list".;
```

It is possible to combine conditions, merely by separating them by the word OR, in which case the combined condition is considered to be satisfied if and only if one or more of its component conditions is satisfied. It is also possible to combine conditions, by separating them by the word AND, so that the combined condition is satisfied if and only if all the component conditions are satisfied. Any number of conditions may be strung together in either, or a combination of both, of these two ways. Any group of conditions connected by the word AND (or by several AND's) should be enclosed in parenthesis.

Examples:

```
    IF (A>B AND C<D+1) OR (A=0 AND D-1<0)  THEN
          GO TO S3;
          ELSE LET D:=D+1
             GO TO S4; .;
```

Notice that the above condition combination is changed if the parenthesis are placed differently:

```
    IF (A>B AND C<D+1  OR A=0 AND D-1<0) THEN etc.
          is satisfied if A>B and D-1<0 and either
          C<D+1 or A=0.

    IF I>K THEN
          IF J>I THEN
            LET I:=I+1; .;
          ELSE IF J>I THEN
            LET I:=J;
            ELSE LET I:=K; .; .;
```

3.3 Dummy Statements

A dummy statement is a convenience sometimes used for ending a vary statement list or a subroutine. A dummy statement is always labelled, and is usually referred to by at least one transfer statement and serves as an exit point, for example

```
VARY INDEX FROM LLIM BY INCR TO ULIM IN
  IF THING>27.5 THEN GO TO END; .;
  LET THING:=THING+ALPHA (INDEX/3);
END:; .;
```

In this case the dummy statement END:; gives a place to jump to within the VARY loop just in case the value of THING is greater than 27.5.

4. Compiler Directing Statements

4.1 DIMENSION Statement

At the beginning of a program, one must state the maximum size attained by each vector (one subscript) or matrix (two subscripts). This is done by means of a DIMENSION-statement. A DIMENSION-statement consists of the word "DIMENSION", followed by the name of a variable (a vector or a matrix only), followed – in parenthesis – by the maximum value(s) its subscript(s) can attain. This maximum value (or, these values) are expressed as unsigned integers. A DIMENSION-statement ends with a semicolon.

Example:

```
DIMENSION A(5, 103);
DIMENSION B(7);
DIMENSION A(5, 103), B(7), X2Z(3, 2), SUM(5);
```

The first of these statements defines A as a matrix whose maximum size is 5x103. The second defines B as a vector of maximum order 7. The third does both of the above, and also defines X2Z as a 3x2 (maximum) matrix, and SUM as a vector of (maximum) order 5. One may combine any number of matrices and vectors, in any order, into one DIMENSION-statement by separating them by commas, and ending the statement with a semicolon.

4.2 REFER Statement

The REFER statement has two uses. It allows the name of a variable or set of variables to be changed; and it allows a set of simple variables to be combined into a single vector. As a special case of the first of these, it also allows one to assign names to constants, and, thus, to assign values - as for initialization - to constants. Examples:

```
     REFER TO A AS B;
     REFER TO F(1, 2) AS G;
     REFER TO A AS B AND GRMP AS C;
     REFER TO J(3) AS K;
     REFER TO GRMP AS C, A AS B, SUM 2 AS SUM 3 AND RDC AS Y;
     REFER TO A AS B AND F(1, 2) AS G;
     REFER TO 3.0 AS X;
     REFER TO F(1, 2) AS G, J(3) AS K AND A AS B;
     REFER TO A AS B, GRMP AS C, 3.0 AS X, SUM2 AS SUM3, 7 AS Z AND RDC AS Y;
```

These are all examples of the first use of the REFER statement. No variable should appear more than once during any REFER statement. A, GRMP, SUM2, may be either simple variables, or arrays. Their equivalents are assigned the same dimension as the original. When a simple variable, a vector, or an entire matrix is renamed, the original name becomes undefined, and may be reassigned. When more than two pairs are given in a REFER statement, all but the last two are separated by commas. The last two are separated by "AND" which is not preceded by a comma. If a number is used, it must be as the first half of a pair. The statement must end with a semicolon. The second use of the REFER statement is to define vectors:

```
            REFER TO X(1,1), Y(2, 3), Z(3), ALPHA, 13.75 AS V;
```

ALPHA must be a simple variable. V must have been previously dimensioned by a DIMENSION statement. Commas separate all values, and the word "AND" is not used.

5. Comments

Any string of symbols not containing a semicolon may be included in the program in a statement which consists of the word COMMENT followed by the symbol string and ending with a semicolon. This may be used for adding explanatory comments within the body of a program. These comments will be printed in the output listing but will not affect the program in any way.

6. Code

In certain cases it might be efficient to include in a program a block of machine language (symbolic or numeric) coding. This may be done (so long as the block does not contain the string END; by prefixing the block with CODE and finishing it with END followed by a semicolon.

7. Procedures and Programs

Within a program, groups of statements are blocked into procedures and a procedure may serve as a routine or as a subroutine. A complete program begins with the word START which serves to inform the compiler that a complete program follows. The program ends with STOP which informs the compiler that the run is over. Between these delimiters may appear one or more procedures each of which begins with PROCEDURE and ends with END.

7.1 Structure of a Procedure Heading

Each procedure begins with a heading of the following form.

PROCEDURE "procedure name" ("parameter list"),("subroutine list");

The procedure name is a symbol string that starts with a letter and serves to identify the procedure. It increases the efficiency of the final program if the names of the library subroutines to be used in the procedure body are listed in the heading in the subroutine list. If none are to be used, this list is left blank.

Frequently procedures are defined for use as subroutines and as functions of one or more variables or parameters. In these cases, the parameters that must be set before (or upon) input to the subroutine are listed in the heading. If there are no parameters this list may be left blank.

Example of procedure heading

```
PROCEDURE SUMMATION 1(n, x), (SIN);
```

this heading might be a subroutine which computes the value of $\sum_{1}^{n} \sin(nx)$ and leaves the result in x. "n" and "x" are the parameters to be set upon input and SIN is the only subroutine used.

7.2 Executing Subroutines

When, within a routine, it is desired to execute a subroutine and EXECUTE statement is used. This consists of the word EXECUTE, followed by the name of the subroutine, a list of either values of parameters or a list of locations where the values can be found, ending with a semicolon. E.g.

```
EXECUTE SUMMATION 1(m, arg);
EXECUTE SUMMATION 1(27, vee);
EXECUTE SUMMATION 1(34, 10.74);
```

are all calls on the SUMMATION 1 subroutine.

The first example will enter the subroutine with the address of "m" replacing the dummy variable "n" and the address of "arg" replacing the dummy variable "x". The result will be placed in location "arg". In the second case, the value 27 will replace "n" and "vee" will replace "x". In the third case, 34 will be used for n and 10.75 for x.

N.B.: The result in case 3 will never be used, for after the computation, the result will be stored in the location where 10.75 used to be but, not having been given a symbolic name, it can never be referred to again.

7.3 Program Structure

A compiled program always begins running at the first statement of the first procedure in the program. Within a procedure transfers (by means of GO TO) may be made to any labelled statement or to the <u>beginning</u> of any other procedure in the program. Once a procedure has been left, by a GO TO, return may be made only to the beginning of the procedure. If, however a procedure is left by an EXECUTE control will be returned to the statement following the EXECUTE after the executed subprocedure has been operated.

7.4 Inter-routine communication

It should be emphasized that there is no way to communicate between routine and subroutine <u>except</u> by procedure heading parameters. I.e. symbolic addresses used in more than one routine will be assigned different numeric addresses and will, in general have different values.

8. Sample Program

```
START
   PROCEDURE COMPLEX POLYNOMIAL EVALUATION ( ), (ENTIER);
      COMMENT  This procedure has no parameters and uses the ENTIER
               subroutine. It reads in values of a_1, a_2, ... a_N, x,
               y and N, computing the function a_1+a_2z+a_3z^2+...a_nZ^{N-1} = w
               where z = x + jy and w = u + jv. If N is not a positive
               integer 0≤N≤21, or if |w|>10^10 an error printout
               followed by an absolute halt will occur. If no error,
              the result will be printed and the computer will halt.
              If "start compute" is depressed after print of w, new
              x, y and N will be read and a new w computed.;
   DIMENSION A(21), B(4);
   REFER TO B(1) AS Y, B(2) AS Y AND B(3) AS N;
   EXECUTE READ (21, A);
   COMMENT procedure READ (n, v) is assumed to be a machine language
            subroutine that reads n values into the previously
            dimensioned vector v;
INPUT:  EXECUTE READ (3, B);
   IF N≤0  OR  N≥21  OR  N≠ENTIER (N) THEN
            GO TO ERROR ( ); . CHECK ON N;
   COMMENT:  ERROR is a procedure with no input parameters
   LET u:=a(1), v:=0;
   VARY i FROM 1 BY 1 UNTIL N IN
        EXECUTE COMP EXP INT (x, y, i-1, x1, y1);
        EXECUTE COMP MULT (a(i), 0, x1, y1, x1, y1);
        EXECUTE COMP ADD (u, v, x1, y1, u, v); .
           THIS LOOP HAS COMPUTED
           $w:=\sum_{1}^{N} a_1 z^{i-1}$;
        IF SQRT (u*u+ v*v) 10**10 THEN
           GO TO ERROR ( ); .;
        EXECUTE PRINT (u, v);
        COMMENT procedure PRINT (a, b) is assumed to be a
           machine language subroutine that prints a (signed)
           followed by b(signed) followed by the letter "j".
           E.g.: +10-37j.;
        HALT INPUT;
END
PROCEDURE COMP ADD (a, b, c, d, e, f), ( );
   LET e:=a+c, f:=b+d;
   COMMENT This subroutine computes e+jf:=(a+jb)+(c+jd);
END
```

```
    PROCEDURE COMP MULT (a, b, c, d, e, f), ( );
       LET e:= a*c - b*d,  f:= b*c + a*d;
       COMMENT This subroutine computes e+jf:= (a+jb)*(c+jd);
    END
    PROCEDURE COMP EXP INT (a, b, n, c, d), ( );
       LET c:= 1, d:= 0;
       VARY m FROM 1 BY 1 UNTIL n IN
            EXECUTE COMP MULT (a, b, c, d, c, d); .;
       COMMENT This subroutine computes (c+jd):= (a+jb)^n
            where n is an integer. If n is negative the
            program will cycle.  If n is not an integer,
            the nearest smaller integer will be used;
    END
    PROCEDURE READ (n, V), ( );
       CODE .... END;
    END
    PROCEDURE PRINT (a, b), ( );
       CODE .... END;
    END
    PROCEDURE ERROR ( ), ( );
       CODE ....
       HALT: HALT HALT;
       COMMENT The odd looking thing one line above is
               a self-addressed halt loop;
    END
STOP
```

Appendix 5 - Part II

Setting External Equipment

The Analog-Digital Converter.

The range of the ADC can be set by either of the following commands:

```
SET ADC TO 1V SCALE;
SET ADC TO 10V SCALE;
```

If the high speed mode is desired, the phrase ", HIGH SPEED (X DEGREES") (where X is some integer, usually a multiple of ten) may be inserted before the semicolon. The number of degrees in the parenthesis determines the setting of the crankshaft. If X is not a multiple of 10, it is taken to be the nearest multiple of 10 smaller than the given value.

Examples:

```
SET ADC TO 1V SCALE, HIGH SPEED (110 DEGREES);
SET ADC TO 10V SCALE, HIGH SPEED (20 DEGREES);
```

Output lines.

A voltage can be put on an output line by a statement of the following sort:

```
SET ℓ.ddd TO X VOLTS;
```

where ℓ.ddd is a code to identify a line consisting of a letter followed by a period followed by a three digit number (A.003, B.227, etc.) and X is a number. If a statement is used that has an undefined line identifier (i.e. there is no line corresponding to the given identifier) the compiler will not accept the statement. If a voltage is specified that is not available, the nearest voltage, smaller in magnitude, will be used.

Example:

```
SET C.003 TO -10 VOLTS;
```

The EPUT Meter.

There are many settings that must be made in programming the EPUT meter. These have been combined into one lengthy statement, however, for the sake of brievity certain portions may be omitted. The statement used to set the EPUT meter is of the following form:

```
SET EPUT mode, N=X, M=Y, delay, slope DA1, slope DA2,
          slope DA3 THEN, slope DA3, DA3= Z VOLTS;
```

The "mode" portion may be MA, MB, MC, MD, ME, MF or MG. It may not be omitted.

The "N=X," portion is called the A-setting and X may be either a number or a variable. If the A-setting is omitted entirely, N is set equal to zero. Similarly the "M= Y," portion is called the B-setting, Y may be any number or variable and if "M= Y," is omitted, M will be equal to zero.

The delay portion is used to determine whether delayed or immediate operation is desired. If delay is wished "D" is written. If immediate operation is to be used, the "D" and the following comma are deleted.

Each of the three amplifiers must have a slope set for its input. This is done by putting "+" or "-" for "slope" before "DA1", "DA2" and "DA3". It is possible to omit any or all of the slope parts, in which case the omitted slope(s) will be positive. In the case of MD, it may be desired to reset the slope of DA3 during operation. In this case the "THEN, slope DA3" is used. This portion must be omitted for modes other than MD and in Mode MD, may be omitted if the slope of DA3 is to remain unchanged.

The final setting is of the threshold of the DA3 amplifier which is set by the "DA3= Z VOLTS" part of the statement. Z may be any number and if this part of the statement is omitted the threshold will be the same as the previous setting within the program. (Caution: If a SET EPUT statement without a threshold specification is used and there was no previous setting of the threshold, the value will be undefined)

Examples:

```
          SET EPUT MD, N=128, M=32,768, D, +DA1, -DA2,
              -DA3 THEN, +DA3, DA3=-10 VOLTS;
          SET EPUT MA, N=2000;
          SET EPUT MB, M=12763, -DA1, -DA3;
          SET EPUT ME, N=100, M=200, D, -DA3;
          SET EPUT ME, N=2056, D, -DA2, DA3=26 VOLTS;
```

Relay Inputs.

It is possible to read a series of values from relay inputs into the computer memory. This statement is of the form:

```
          READ something INTO somewhere;
```

where the "something" is an input relay name or list of names and the "somewhere" is a vector which must previously have been dimensioned.

The "something" may take any of the following forms or may be a list of as many as are needed (separated by commas):

```
          GROUP (group number)
          GROUPS (group number, group number, ..., etc.)
          RELAY (relay number)
          RELAYS (relay number, relay number, ..., etc.)
```

A group number is a one or two digit integer while a relay number is a one or two digit integer followed by a period, followed by an integer from 1 to 8.

The "somewhere" can be of the form:

vector name (integer)

vector name (integer, FF)

The vector name can be any string of letters and digits (beginning with a letter). In the first case, a single value is read into a single entry in the vector. In the second case, several values are read in and the integer specifies the location of the first value. The ",FF" is for typographical purposes and means "and the following". It is not necessary and may be omitted at will.

Examples:

```
READ RELAY (29.2) INTO FOOF (28);
READ RELAYS (8.3, 8.4, 29.1) INTO FOOF (25,FF);
READ GROUP (1) INTO FOOF (1,FF);
READ GROUPS (2,3) INTO FOOF (9,FF);
READ GROUPS (1,2,3), RELAYS (8.3, 8.4, 29.1, 29.2)
      INTO FOOF (1,FF);
```

The following two statements are equivalent although the second is more efficient

```
READ GROUPS (1,2), RELAYS (3.1, 3.2, 3.3, 3.4,
      3.5, 3.6), RELAY (3.7), RELAY (3.8), GROUP (4)
            INTO GBW (7,FF);
READ GROUPS (1,2,3,4) INTO GBW (7,FF);
```

If a relay or group is named that does not exist, the compiler will give an error indication. If a vector is used that is either undimensioned or is too small, the <u>program</u> will give an error indication at run time.

Switch Opening and Closings.

There are a series of toggle switches which are identified by numbers and may be opened or closed under computer control. The following statements will achieve this purpose:

OPEN SWITCH switch number;

CLOSE SWITCH switch number;

The switch number may be any integer but if a switch is named that does not exist, the compiler will indicate an error.

Typeouts During Program Running.

It is possible to define and use standard typeouts during the operation of a program. These standard typeouts are given numbers when they are defined and the statement: TYPEOUT integer; will cause the typeout message identified by that integer to be typed out. It is also possible to typeout the current value of a variable using: TYPEOUT (variable); . For brief symbolic typeouts, the symbol string to be typed can be placed in the statement itself, enclosed in single quotes: TYPEOUT 'string';

The various typeouts may be combined in a single statement.

Examples:

We might have defined typeout 37 as "The current value of" and if we wish to type "The current value of Z(1,1) is ..." (where ... means the actual value of Z(1,1)) we write

TYPEOUT 39 'Z(1,1) is' (Z(1,1));

If typeout 2 is "Calibrate temp. probe:" the statement TYPEOUT 2 (i); will print the typeout, followed by the value of the variable i.

To define standard typeouts a statement of the form: DEFINE TYPEOUT number AS symbol string; is used. Several typeouts can be defined in the same statement.

For example:

```
                    DEFINE TYPEOUT 1 AS 'FOOF',
                           TYPEOUT 4 AS 'Gratch is nit.',
                           TYPEOUT 3 AS 'X(3,4) = ';
```

It is important to note that the symbol string must not contain single quotes (').

<u>Delays</u>.

Since the programs put together with this compiler will have to run in real time, there might be times when it would be useful to have the computer wait a fixed amount of time while some external event is occurring. The statement: DELAY number SECONDS; where "number" is any positive number will cause the computer to delay that amount of time before continuing.

Another type of DELAY might be useful. The EPUT meter reads directly into the accumulator of the computer and the statement "DELAY FOR EPUT;" would make the computer delay until the number in the accumulator stops changing (i.e., until the EPUT has stopped). This statement does not appear in the formal syntax.

Example: Sample program from the Controller Program Flow Diagrams

```
START PROCEDURE TRANSDUCER CALIBRATION E, (ABS, MEDIAN)
    COMMENT: In the procedure heading, the E indicates that there are
             no input parameters. (ABS, MEDIAN) indicates that the ABS
             and MEDIAN subroutines are used;
    DIMENSION INPUT (58), T(31), P(5), V(2);
    INTEGER  i, j;
    DEFINE TYPEOUT 1 AS 'If calibration is required - replace those probes.',
           TYPEOUT 2 AS 'Calibrate Temp. probe',
           TYPEOUT 4 AS 'Calibrate Press. probe',
           TYPEOUT 5 AS 'START ENGINE - SET 1000 RPM',
           TYPEOUT 40 AS 'Calibrate velocity probe';
    COMMENT: This program is an approximation to the transducer
             calibration flowchart of the controller flow diagrams.
    S11: SET ADC TO 10V SCALE;
    S12: READ RELAY GROUPS (1, 2, 3, 4, 5, 7, 8),
              RELAYS (29.1, 29.2) INTO INPUT (1,FF);
    S14: LET T(1):=96.0*INPUT(2) - 313.0,
             T(26):=96.0*INPUT(33) - 313.0,
             T(27):=96.0*INPUT(34) - 313.0;
         VARY i FROM 2 BY 1 UNTIL 25 IN
             LET T(i):=211.0*INPUT (i+7) - 328.0; .;
         VARY i FROM 28 BY 1 UNTIL 31 IN
             LET T(i):=83.3*INPUT (i+21) - 344.2; .;
    S16: LET P(1):=490*(INPUT (42) - 3.5)/(23.5 - INPUT (42));
             VARY i, a, b, j OVER (2,3,4,5)(56.5, 37.7, 37.7, 141.4)
                                  (283, 101.8, 101.8, 381.8)(43,44,45,53) IN
               LET P(i):=a - b/INPUT(j); .;
    S18: LET V(1):=        x            ;
             V(2):=        y            ;
         COMMENT: These computations are not specified in the flowchart;
    S110:  LET TMED:= MEDIAN (T(i));
           COMMENT:  i is a dummy subscript;
    S118:  VARY i FROM 1 BY 1 UNTIL 31 IN
              IF ABS (T(i) - TMED) > 5 THEN TYPEOUT 2(i); .; .;
    S121:  LET PMED:= MEDIAN (P(i));
    S129:  VARY i FROM 1 BY 1 UNTIL 5 IN
              IF ABS (P(i) - PMED) > 5 THEN TYPEOUT 4(i); .; .;
    S132:  IF V(1) < CONST THEN TYPEOUT 40; TYPEOUT '1'; .;
           IF V(2) < CONST THEN TYPEOUT 40; TYPEOUT '2'; .;
           COMMENT:  The value of "CONST" was not specified on the flowchart.;
    S134:  HALT NEXT;
    NEXT:  TYPEOUT 1;
    S136:  CLOSE SWITCH K.009;
           COMMENT:  START ENGINE;
    S137:  CLOSE SWITCH Z.009;
           COMMENT:  SET 1000 RPM;
    S138:  TYPEOUT 5;
    S139:  OPEN SWITCH K.009;
    S140:  OPEN SWITCH Z.009;
    S141:  SET ALPHA TO BETA;
    S142:  SET GAMMA TO DELTA;
           COMMENT:  The flowchart says "NO LOAD" for S141 and "1000 RPM"
                     for S142;
    S143:  HALT A2; END
STOP
```

Bibliography and References on Compiling Languages and Compilers

1. Bottenbruch, H., "Structure and Use of ALGOL 60," JACM Vol. 9, Nr. 2.

2. Brooker, R. A. and Morris, D., "A General Translation Program for Phrase Structure Languages," JACM Vol. 9, Nr. 1, 1962.

3. Burroughs Algebraic Compiler. Burroughs Corporation Bulletin 220-21011-P, 1961.

4. Chomsky, N., "On Certain Formal Properties of Grammars," Information and Control, Vol. 2, Nr. 2, 1959.

5. Evans, A. Jr., Perlis, A. J. and VanZoren, H., "The Use of Threaded Lists etc.," CACM, Vol. 4, Nr. 1, 1961.

6. Glennie, A. E., "On the Syntax Machine and the Construction of a Universal Compiler," Carnegie Institute of Technology, 1960.

7. Gorn, S. et al. "Common Programming Language Task," Part I University of Pennsylvania, 1959.

8. Gorn, S. and Parker, E. J., "Common Programming Language Task," Part I University of Pennsylvania, 1960.

9. Gorn, S., "Processors for Infinite Codes of the Shannon-Fano Type," Proceedings of the Symposium on Mathematical Theory of Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn. To be published.

10. Gorn, S., "Some Basic Terminology Connected With Mechanical Languages and Their Processors," CACM Vol. 4, Nr. 8, 1961.

11. Gorn, S., "Specification Languages for Mechanical Languages and Their Processors. A Baker's Dozen," CACM Vol. 4, Nr. 12, 1961.

12. Holt, A. W. and Turanski, W. J., "Common Programming Language Task," Part II University of Pennsylvania, 1959.

13. Holt, A. W., Turanski, W. J. and Parker, E. J., "Common Programming Language Task," Part II University of Pennsylvania, 1960.

14. Ingerman, P. Z., "A String-Manipulative Language for Use in Compilers of the Syntax-Oriented Type." Paper in progress.

15. Ingerman, P. Z., "A Translation Technique for Languages Whose Syntax Is Expressable in Extended Backus Normal Form," Proc. of the Symposium on Symbolic Languages in Data Processing, Rome, Italy. To be published 1962.

16. Ingerman, P. Z., "Dynamic Declarations," CACM Vol. 4, Nr. 1, 1961.

17. Irons, E. T., "A Syntax Directed Compiler for Algol 60," CACM Vol. 4, Nr. 1, 1961.

18. Irons, E. T. and Feurzeig, W., "Comments on the Implementation of Recursive Procedures etc.," CACM Vol. 4, Nr. 1, 1961.

19. Irons, E. T., "Maintenance Manual for PSYCO (The Princeton Syntax Compiler)," Part I. I.D.A. C.R.D., Von Neumann Hall, Princeton, N.J., January 1961.

20. Leonard, G. F., "The CL-1 Programming System Users Manual," Technical Operations, Inc., Report TO-B-61-3, 1961.

21. McCarthy, J. dt al. "LISP Programmers' Manual," Computation Center, Massachusetts Institute of Technology, 1960.

22. Mechanical Translation Group. "A New Approach to the Mechanical Syntactical Analysis of Russian," National Bureau of Standards Report 6505 and Supplement, 1959.

23. Naur, P. ed., "Report on the Algorithmic Language Algol 60," CACM Vol. 3, Nr. 5, 1960.

24. Newell, A. et al., "The Elements of IPL Programming," The Rand Corporation report P-1897, 1960.

25. "Reference Manual 709/7090 FORTRAN Programming System," I.B.M., New York, 1961.

26. Rhodes, I., "Hindsight in Predictive Syntactic Analysis," National Bureau of Standards report 7034, 1960.

27. Sattley, K., "Allocation of Storage for Arrays in Algol 60," CACM Vol. 4, Nr. 1, 1961.

28. Warshall, S., "A Theorem on Boolean Matrices," JACM Vol. 9, NR 1, 1962.

29. Woodger, M., "An Introduction to Algol 60," Computer Journal, p. 67, 1960.

JACM = Journal of the Association for Computing Machinery
CACM = Communications of the Association for Computing Machinery

Section B. Circuit Analysis and Network Element Value Solvability Study

The work done on this subject from July 1, 1961 to June 30, 1962 consists primarily of S. D. Bedrosian's Ph.D. dissertation which is attached to this report.

AD- ACCESSION NO.

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC, by
The Institute for Cooperative Research

Status Report, 106 pp incl. tables and illus., 30 June 1962, UNCLASSIFIED

The present study is a continuation and extension of work done during 1959-60, "A Study of Piece Part Fault Isolation by Computer Logic". During June 1961-June 1962, emphasis has been on automatic checkout of Automotive Systems with special regard to the computer controlled tank checkout system developed by Frankford Arsenal and on an extension of work of Berkowitz in network element solvability.

Section A - Automatic Programming is divided into an introduction followed by two main parts and five appendices. The 1st main section discusses the compiler pseudo-language, derived from the language developed for Electronic Piece Part Fault Analysis system, giving a formal specification of the syntax of the language and showing that part of it is a general algebraic compiler and part is a series of problem oriented input-output statements. The 2nd main section discusses the compiler. This compiler was first presented in June '61 report of this Project and the present discussion describes various additions and changes. Also given are the language syntax in a tabular format and in machine-oriented matrix format.

Section B - Circuit Analysis and Network Element Value Solvability Studies consists of Bedrosian Ph.D. dissetation on Element Value Solution of Single-Element-Kind Networks.

In this study the work of Berkowitz is extended by finding explicit solution techniques for single-element-kind networks. This includes a computer programmable algorithm and discussion of various ramifications.

I. The Institute for Cooperative Research (University of Pennsylvania)

DISTRIBUTION LIMITATION:
None, obtain copies from ASTIA.

UNCLASSIFIED

AD- ACCESSION NO.

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC, by
The Institute for Cooperative Research

Status Report, 106 pp incl. tables and illus., 30 June 1962, UNCLASSIFIED

The present study is a continuation and extension of work done during 1959-60, "A Study of Piece Part Fault Isolation by Computer Logic". During June 1961-June 1962, emphasis has been on automatic checkout of Automotive Systems with special regard to the computer controlled tank checkout system developed by Frankford Arsenal and on an extension of work of Berkowitz in network element solvability.

Section A - Automatic Programming is divided into an introduction followed by two main parts and five appendices. The 1st main section discusses the compiler pseudo-language, derived from the language developed for Electronic Piece Part Fault Analysis system, giving a formal specification of the syntax of the language and showing that part of it is a general algebraic compiler and part is a series of problem oriented input-output statements. The 2nd main section discusses the compiler. This compiler was first presented in June '61 report of this Project and the present discussion describes various additions and changes. Also given are the language syntax in a tabular format and in machine-oriented matrix format.

Section B - Circuit Analysis and Network Element Value Solvability Studies consists of Bedrosian Ph.D. dissertation on Element Value Solution of Single-Element-Kind Networks.

In this study the work of Berkowitz is extended by finding explicit solution techniques for single-element-kind networks. This includes a computer programmable algorithm and discussion of various ramifications.

I. The Institute for Cooperative Research (University of Pennsylvania)

DISTRIBUTION LIMITATION:
None, obtain copies from ASTIA.

UNCLASSIFIED

AD- ACCESSION NO.

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC, by
The Institute for Cooperative Research

Status Report, 106 pp incl. tables and illus., 30 June 1962, UNCLASSIFIED

The present study is a continuation and extension of work done during 1959-60, "A Study of Piece Part Fault Isolation by Computer Logic". During June 1961-June 1962, emphasis has been on automatic checkout of Automotive Systems with special regard to the computer controlled tank checkout system developed by Frankford Arsenal and on an extension of work of Berkowitz in network element solvability.

Section A - Automatic Programming is divided into an introduction followed by two main parts and five appendices. The 1st main section discusses the compiler pseudo-language, derived from the language developed for Electronic Piece Part Fault Analysis system, giving a formal specification of the syntax of the language and showing that part of it is a general algebraic compiler and part is a series of problem oriented input-output statements. The 2nd main section discusses the compiler. This compiler was first presented in June '61 report of this Project and the present discussion describes various additions and changes. Also given are the language syntax in a tabular format and in machine-oriented matrix format.

Section B - Circuit Analysis and Network Element Value Solvability Studies consists of Bedrosian Ph.D. dissertation on Element Value Solution of Single-Element-Kind Networks.

In this study the work of Berkowitz is extended by finding explicit solution techniques for single-element-kind networks. This includes a computer programmable algorithm and discussion of various ramifications.

I. The Institute for Cooperative Research (University of Pennsylvania)

DISTRIBUTION LIMITATION:
None, obtain copies from ASTIA.

UNCLASSIFIED

AD- ACCESSION NO.

A STUDY OF PIECE PART FAULT ISOLATION BY COMPUTER LOGIC, by
The Institute for Cooperative Research

Status Report, 106 pp incl. tables and illus., 30 June 1962, UNCLASSIFIED

The present study is a continuation and extension of work done during 1959-60, "A Study of Piece Part Fault Isolation by Computer Logic". During June 1961-June 1962, emphasis has been on automatic checkout of Automotive Systems with special regard to the computer controlled tank checkout system developed by Frankford Arsenal and on an extension of work of Berkowitz in network element solvability.

Section A - Automatic Programming is divided into an introduction followed by two main parts and five appendices. The 1st main section discusses the compiler pseudo-language, derived from the language developed for Electronic Piece Part Fault Analysis system, giving a formal specification of the syntax of the language and showing that part of it is a general algebraic compiler and part is a series of problem oriented input-output statements. The 2nd main section discusses the compiler. This compiler was first presented in June '61 report of this Project and the present discussion describes various additions and changes. Also given are the language syntax in a tabular format and in machine-oriented matrix format.

Section B - Circuit Analysis and Network Element Value Solvability Studies consists of Bedrosian Ph.D. dissertation on Element Value Solution of Single-Element-Kind Networks.

In this study the work of Berkowitz is extended by finding explicit solution techniques for single-element-kind networks. This includes a computer programmable algorithm and discussion of various ramifications.

I. The Institute for Cooperative Research (University of Pennsylvania)

DISTRIBUTION LIMITATION::
None, obtain copies from ASTIA.

UNCLASSIFIED

DISTRIBUTION LIST

10 copies.	Commander ASTIA Arlington Hall Station Arlington 12, Virginia ATTN: TIPDR
25 copies.	Commanding Officer U.S. Army Munitions Command Frankford Arsenal Philadelphia 37, Pennsylvania ATTN: Mr. A. Chalfin
1 copy	Commanding Officer U.S. Army Ordnance District-Phila. 128 North Broad Street Philadelphia 2, Pennsylvania ATTN: OD, R and D Branch
1 copy	Commanding Officer Diamond Ordnance Fuze Laboratories Washington 25, D. C. ATTN: Technical Reference Section
25 copies.	Commanding Officer U.S. Army Research Office (Durham) Duke Station, North Carolina ATTN: Dr. J. Gergen