F O R T R U N C I B L E

by

G. E. Haynam

April, 1960

COMPUTING CENTER

TABLE OF CONTENTS

	Page
ABSTRACT	1
ACKNOWLEDGEMENTS	3
I. INTRODUCTION	4
II. FORTRUNCIBLE LANGUAGE	5
A. WORDS	6
1. OPERANDS	6
a) VARIABLES	7
b) SUBSCRIPTED VARIABLES	8
c) CONSTANTS	9
2. OPERATORS	10
a) STANDARD MATHEMATICAL	10
b) ARITHMETIC	14
c) EXTENSIONS	15
3. CALLS	17
B. STATEMENTS	19
1. SUBSTITUTION	20
2. JUMP	20
3. INPUT	21
4. OUTPUT	21
5. HALT	23
6. BYPASS	24
7. EXTENSION	24
8. CONDITIONAL	24
9. ITERATION	25
10. ARRAY OPERATION	27
11. PROCEDURES	30
12. PROCEDURE DECLARATION	31

TABLE OF CONTENTS (Continued)

	Page
13. END	32
14. EXECUTE	32
15. INTEGER DEFINITION	34
16. ARRAY DEFINITION	34
C. EXAMPLE PROBLEMS	36
III. CODING DETAILS	41
A. BASIC PACKAGES	41
B. CARD FORMATS	42
1. HEADER CARD	42
2. COMMENTS CARD	43
3. STATEMENT CARDS	43
C. CORRESPONDENCE TABLE	46
D. DATA CARDS	47
IV. OPERATING PROCEDURES	48
A. MODES OF OPERATION	48
B. FORTRUNCIBLE PROGRAM	49
C. RUNCIBLE PROGRAM	50
D. OPERATOR INSTRUCTIONS	50
V. STOPS	53
A. NORMAL STOPS	53
B. ERROR STOPS	53
BIBLIOGRAPHY	54
APPENDIX I - SUMMARY OF STATEMENTS	55
APPENDIX II - PASSING "RUNCIBLE I" STATEMENTS	58
APPENDIX III - RELOADING A CORRESPONDENCE TABLE	59

TABLE OF CONTENTS (2nd Continued)

		Page
APPENDIX IV	- SAMPLE PROBLEMS	60
APPENDIX V	- "FAR"	66
APPENDIX VI	- ADDITIONS TO "SOAP III" - "RUNCIBLE I" 533 PLUGBOARD	74

ABSTRACT

The Fortruncible compiler extends the basic Runcible I compiler in the following ways:

1) Alphabetic names for simple variables and arrays,
2) Special characters, group II, (Fortran style) for designating the mathematic operations,
3) Matrix operations,
4) Convenient method for writing subroutines in the compiler language (Procedures).

In addition to the above extensions any number of Runcible statements may be intermixed with the Fortruncible statements to give complete flexibility and generality to the compiler.

This compiler uses the standard Runcible control panel with suitable additions to the wiring as described in this report. Since special characters, group II, have been used to designate the mathematical operations, the 533 read-punch unit must be equipped with the special characters device along with the alphabetic attachment. No attempt has been made to avoid the use of the special character device by complicated control panel wiring.

There are five forms of the compiler program available, each of which requires a different amount of additional 650 hardware as follows:

1) Basic 650.
2) Basic 650 with Table Look-up on Equal (TLE) operation code.
3) Augmented 650 (Indexing Registers and I.A.S.).
4) Augmented 650 with Table Look-up on Equal (TLE) operation code.
5) Augmented 650 with 355 RAMAC and all of the above additions.

ABSTRACT (continued)

The output of the Fortruncible compiler is standard Runcible statements that have been translated from the Fortruncible statements along with those reproduced from any input Runcible statements.

ACKNOWLEDGMENTS

No program the size of Fortruncible can be attributed to the efforts of any one individual, but it is the natural outgrowth of many other programs which themselves have been influenced by many individuals too numerous to mention. However, credit should be extented to Dr. Perlis for his basic effort in writing the first compiler for the IBM 650 (IT) [1]. Also the efforts of the entire Case Computing Center should not go unnoticed as this program is only a natural extension of the Runcible I compiler [2][3][5] as developed by the staff at the Center.

I should like to express my appreciation to Professor R.J. Nelson and Mr. F. Way III, the directors of the Computing Center, for their support of this project. I am also indebted to Donald Knuth for the many sections in this report that were extracted from the Runcible I manual and to Y.H. Rutenberg for his many corrections and additions to this report. Finally, this program could not have been put into final running form without the help of everyone connected with the Center who debugged the program by using it to solve problems.

F O R T R U N C I B L E

I. INTRODUCTION

In order to solve a complicated problem on a high-speed digital computer, a large amount of time is required by a programmer and/or coder to translate the specifications of the problem, which may be in the form of a flow chart, into the rather restricted language of the computer. Also if at a later time it is desired to solve the previously coded problem on a different computer, then it will again be necessary to spend an appreciable amount of time reprogramming and recoding the problem for the new machine. In order to simplify the coding procedure and to reduce the coding time substantially, several problem languages have been formulated. These closely resemble the common mathematical language used in constructing the flow chart that expresses the solution of a problem. Then, programming the solution of a problem in terms of any one of these languages becomes a simple matter. Even persons unfamiliar with digital computers can write programs without the help of a trained programmer.

Since the translation from one of these problem languages to a given machine language is well-defined, a translator or compiler program can be written to effect this translation automatically on the computer, thereby reducing the coding time from days to minutes. This saving in time reduces the cost of programming a given problem even though the cost of computer time required for the translation is relatively high when compared to personnel time.

The first problem-oriented language for use on an IBM 650 computer was developed by Dr. A.J. Perlis, Mr. J.W. Smith, and Mr. H.R. Van Zoeren of Carnegie Institute of Technology and was called "IT-language" [1]. This basic "IT-language" has been extended by the Staff at Case Institute of Technology in their "Runcible I" compiler [2] to yield still greater ease in programming. However, two serious shortcommings of "IT-language" are still retained in the "Runcible I" compiler, namely: 1) All problem variables must be expressed as one of the three simply subscripted

variables I, Y, or C rather than by problem related names, 2) A rather cumbersome form of matrix notation which bears little resemblance to that normally encountered in problems. In order to eliminate these deficiencies and to add further flexibilities in the problem language, a new problem-oriented language, called "Fortruncible", was developed. This language retains many of the basic ideas contained in "IT-language" along with these added flexibilities.

The basic philosophy of the Fortruncible compiler is to translate "Fortruncible" language into standard "Runcible I" language which may then be treated as any other Runcible program with all of the various options as described in the "Runcible I" programmer's manual. As can be inferred from what was said above, the use of "Fortruncible" language will necessitate an additional pass through the 650 to perform the Fortruncible to Runcible translation. However, the added flexibility of the "Fortruncible" language is well worth the additional computer time as considerably fewer programming mistakes will be encountered.

II. "FORTRUNCIBLE" LANGUAGE

If one wishes to formulate a language, the first thing to be done is to choose the list of symbols that will be used to construct the language. This list of symbols is referred to as the "alphabet" of the language. In the case of "Fortruncible" language the following list of symbols will be chosen as its alphabet:

FORTRUNCIBLE ALPHABET

A	J	S	0	9	=
B	K	T	1	;	>
C	L	U	2	.	$
D	M	V	3	(	
E	N	W	4	)	
F	O	X	5	+	
G	P	Y	6	-	
H	Q	Z	7	*	
I	R	Δ (space)	8	/	

Then, from this alphabet the words of the language will be constructed as strings of these symbols. The number of symbols in a word or string must be greater than or equal to one, and the rules for word formation will be described in section IIa. From the words of the language, one can form sentences or statements which convey a complete idea in the solution of a problem. The rules for constructing statements in "Fortruncible" language are described in section IIB. Finally, a properly ordered sequence of statements in the language will constitute a program, which - if it adequately describes the flow chart for a problem - will yield a program for solving the problem.

A. WORDS

The set of words in the language is divided into three categories: operands, operators, and calls. The set of operands consist of those words which represent numerical quantities such as constants and variables. The set of operators constitute all mathematical operations that are defined in the language, and the calls consist of all other acceptable words which are neither operators nor operands, such as IF, TO, ARRAY, and INTEGER.

1. Operands: The operands which represent numbers are of two categories: fixed point and floating point.

Fixed-point numbers are integers with a numeric value of less than one billion. They are either positive or negative, but may never take on fractional values. Their primary use is as indices or subscripts and they should be only rarely used for arithmetic calculations.

Floating point numbers are normally used for arithmetical operations because of their much greater range of values. Floating point numbers can be zero or range from 10^{-50} to 10^{49} in numerical value,[1] and are always rounded to eight significant figures. It should be noted that

---

[1] When using 653 instructions they range from 10^{-51} to 10^{48}.

large losses in accuracy may result from the operations of addition and subtraction with floating point numbers.

a) VARIABLES. Simple variables are designated by their alphabetic names such as X, Y, Z, W, ALPHA, TAU, MU, etc., and may represent either fixed or floating point numbers. However, all simple named variables will be assumed to represent floating point numbers, unless otherwise designated (see INTEGER statements). Any names which satisfy the following rules of name formation will be considered to represent simple variables.

Rule 1: The first character or symbol in a name must be alphabetic, ie. a symbol from A thru Z.

Rule 2: The remaining symbols must be alphanumeric, ie. consisting of the symbols from A thru Z and the numerics 0 thru 9.

Rule 3: Spaces and special characters may not occur within a given name.

Rule 4: The number of symbols in a name is arbitrary, but only the first five will be retained.

Rule 5: The first five symbols in a name must be unique to the name and should not occur as the first five characters in any other words in the program. This rule is a consequence of rule 4.

For example: BETA, X, X1, YBAR, Y2S are all formed properly according to rules 1 thru 5. However, 1ST, X-Y, *XF, VAR 1, 234 are not simple named variables since each one violates some of the above rules. The names RUMPLESTILTSKIN and AUFWIEDERSEHEN satisfy the rules but will be truncated to five characters and thus will be considered in the program to be RUMPL and AUFWI. But, the names ARCHYPERBOLICTANGENT and ARCHYPERBOLICSINE will be considered identically as the name ARCHY, and thus they violate rule 5.

It should also be noted that care should be exercised to spell the problem names consistently throughout the program; otherwise confusion may result.

The Fortruncible compiler program translates all floating point simple named variables into Runcible C-variables and all fixed point (integer) variables into Runcible I-variables. Due to a limitation of space in the translator, the following rule must be adhered to:

Rule 6: The maximum number of floating point simple named variables permitted in a single program is 48. Similarly, the upper bound on fixed point named variables is also 48.

b) SUBSCRIPTED VARIABLES. Subscripted variables are designated by their alphabetic names followed by their subscripts enclosed by parentheses, such as X(8), Y(2,3), RHO(N), THETA(I,J), etc., and may represent only floating point numbers. The variables which represent the subscripts must be fixed point (integer) variables as described in section IIA/a. The following rules must be satisfied by all subscripted variables:

Rule 7: The names of all subscripted variables must satisfy rules 1 thru 5 as described above.

Rule 8: All subscripted variables must be defined as ARRAYS before they are used in a program. (See ARRAY definitions.)

Rule 9: All subscripts in a subscripted variable must represent fixed point numbers (integers).

Rule 10: The maximum number of subscripts on a subscripted variable is two (2).

Rule 11: All subscripted variables will be considered to represent floating point numbers and will be defined by the translator to be Runcible Y-variables.

Rule 12: The maximum number of different subscripted variables is thirty (30).

It will be assumed that each of the following examples has been preceded by its proper ARRAY definition.

Mathematical Notation	Fortruncible Notation
$BESSEL_{I,J}$	BESSEL(I,J)
$A_{2,1}$	A(2,1)
V_6	V(6)
T_{2I+1}	T(1 + 2*I)

It should be noted that BESSEL(I,J) is considered as BESSE(I,J) as a consequence of rule 5. Also the subscript of an array may be computed but no parentheses may be included in the subscript expression. If it is necessary to use parentheses in the computation of a subscript value, then the computation should be done outside of the subscript notation.

c) CONSTANTS. Constants like variables may be either floating point or fixed point, with their form being determined by the manner in which they are written. Constants which are to be considered as floating point numbers must be written with decimal points or in "Power-of Ten" notation, and all other numbers will be considered as fixed point numbers (integers). The "Power-of-Ten" notation is indicated by enclosing the desired exponent of ten in parentheses immediately after the numerical constant. Some examples of the various constant forms are:

Mathematical Form	Fortruncible Form	Arithmetic
123	123	fixed point
1.	1.0	floating point
12.6×10^{14}	12.6(14)	floating point
69×10^{-6}	69(-6)	floating point
.00072	0.00072	floating point
7.2×10^{-4}	7.2(-4)	floating point
72×10^{-5}	72(-5)	floating point

The last three numbers in the above examples are the same and yield the same result in Fortruncible language even though they are written in a different form. Several rules must be adhered to when writing constants in Fortruncible language.

Rule 13: All floating point numbers written in the "Power-of-Ten" notation must represent acceptable floating point numbers. (i.e. must be in the range 10^{-50} to 10^{49}).

Rule 14: The exponent in the "Power-of-Ten" notation must be an integral constant (i.e. should not contain any decimal points).

Rule 15: Under no circumstances should a constant end with a decimal point, but an additional zero (0) should be supplied if necessary.

Any number of constants may be used in a program.[1]

2. Operators: The set of mathematical operations that can be performed in Fortruncible language are divided into two categories; namely, 1) The standard operations of addition, subtraction, multiplication, etc., and 2) Special functions such as SIN, EXP, LN, etc. which will be classified as extensions.

a) STANDARD MATHEMATICAL OPERATIONS. The mathematical operators which are directly acceptable to Fortruncible language are listed in the following table along with examples in mathematical notation and Fortruncible notation.

---

[1] Actually 700 is the maximum allowable number, but this may safely be considered "infinite" for programs processed by Fortruncible.

Symbol	Operation	Math. Notation	Fortruncible Notation
+	Addition	X + Y	X + Y
-	Subtraction	X - Y	X - Y
*	Multiplication	X Y	X * Y
/	Division	X / Y	X / Y
$	Absolute value	| X + Y |	$ X + Y $
=	Substitution	Y = X + 2	Y = X + 2
**	Power	X^Y	X ** Y
>	Greater than[1]	X > Y	X > Y
>=	Greater or equal[1]	X ⩾ Y	X >= Y
=	Logical equality[1]	X = Y	X = Y

Note the close resemblance of Fortruncible language to standard mathematical notation, and how it enables the writing of a formula as a string of symbols all on one line. The operations can be put together with a few constants to yield:

Mathematical Notation

$$Z = \frac{((10.4 \times 10^{28}) * X^4)Y}{X - Y}$$

Fortruncible Notation

```
Z = ((10.4(28) + (X ** 4)) * Y)/(X - Y)
```

Notice the use of parentheses in this last example.

When more than two operands are involved, parentheses are needed to avoid ambiguity. Parentheses are very important in Fortruncible language because there is no difference in priority or scope between any of the binary operations as usually understood in common mathematical

---

[1] These operations are valid only inside a conditional clause (see conditional statements).

notation. For example, the expression X * Y + Z would mean (X * Y) + Z to most people but Fortruncible would interpret it as X * (Y + Z). As another instance, in order to write X^4Y it is necessary to write (X ** 4) * Y since X ** 4 * Y would mean X^{4Y}. There is a very simple MORAL to be learned from this: always place parentheses around the operands in a Fortruncible language expression until it can mean only one thing. This cannot be stressed too heavily, for the vast majority of programming errors in Fortruncible language are caused by neglecting to place parentheses in the proper way.[1] It should be noted at this point that Fortruncible treats operators in a manner very similar to Runcible and "IT-language".

As a reference, these are the rules by which Fortruncible determines to which operands each binary operation applies:

Rule 16: On the left-hand side of the operator symbol, the binary operation applies to the variable or constant immediately at its left, unless the character next to the operator is a right parenthesis. In the latter case, the entire quantity between the right parenthesis and its matching left parenthesis is used.

Rule 17: On the right-hand side of the operator symbol, everything up to the end of the expression or to the first unmatched right parenthesis is used. One exception is that the minus (-) operator is treated first as a unary operator (see below) and then as a plus (+) binary operator.

The following examples illustrate the application of these rules.

---

[1] Parentheses may be nested within each other not more than nine (9) deep, but this limit is rarely met in practice.

Fortruncible Language		Mathematical Language
X / Y + Z	means	$\frac{X}{Y + Z}$
(X / Y) + Z	means	$\frac{X}{Y} + Z$
X ** Y + 3 * Z	means	$X^{Y} + 3Z$
(X ** Y) + 3 * Z	means	$X^{Y} + 3Z$
(X ** Y + 3) * Z	means	$(X^{Y + 3})\ Z$
((X ** Y) + 3) * Z	means	$(X^{Y} + 3)\ Z$
4/RHO(I + J) - K	means	$\frac{4}{RHO_{(I + J)} - K}$

Remember, it is always better to add parentheses to make your intentions unquestionable than to gamble that the compiler will interpret your expressions the same way you do. The additional parentheses although perhaps redundant will in no way harm the resultant machine language program.

As was noted above, the minus (-) operator is treated as a "unary" operator where the operator applies only to the variable or constant at the immediate right of the operator symbol, unless the character to its right is a left parenthesis. In the latter event, it operates on the entire quantity inside the left parenthesis and the matching right parenthesis. Examples:

Fortruncible Language	Mathematical Language
- X	- X
X - Y - Z	X - Y - Z
X - (Y + Z)	X - (Y + Z)
X ** - (Y - Z)	$X^{-(Y - Z)}$

b) ARITHMETIC. Fortruncible always analyzes expressions by doing the innermost parentheses first, following the rule of ordinary algebra. Once Fortruncible is into the innermost parenthesis it drops the "ordinary" rules of algebra and performs the operations starting from the right and working toward the left! The following examples which assume that X is a floating point variable, will illustrate these ideas:

EXPRESSION	RESULT
X = 3 x 4 + 6	X is given the value 30.0
X = 6 + 3 x 4	X is given the value 18.0
X = 6 + (3 x 4)	X is given the value 18.0
X = (3 x 4) + 6	X is given the value 18.0
X = 6 - 3 x 4 + 2	X is given the value -12.0
X = 6 - (3 x 4) + 2	X is given the value -4.0
X = 6 / 4 / 2	X is given the value 3.0
X = (6 / 4) / 2	X is given the value 0.0
X = (6.0 / 4.0) / 2.0	X is given the value 0.75
X = (6.0 / 4.0) / 2	X is given the value 0.75
X = (6 / 4) / 2.0	X is given the value 0.5

From the above examples we can see that FORTRUNCIBLE attempts to use the arithmetic (fixed or floating) of the innermost parentheses and then tries to keep on using that kind of arithmetic until it must do floating point - at this stage, and from this stage on, it does everything floating point at this parenthesis level. The final substitution into the left hand side is always forced to agree with the arithmetic of the left hand side. Consider that the variables named I, and J are fixed point and X is floating point in the following examples:

EXPRESSION	RESULT
I = 6.0 + 3 - 7 x 2.0	I is given the value -5
I = 6.0 + 3 - 7 x 2	I is given the value -5
X = 6.0 + 3 - 7 x 2	X is given the value -5.0
X = 6.0 + 3.0 - (1 / 4)	X is given the value 9.0
J = 6.0 + 3.0 - (1 / 4)	J is given the value 9
X = 3.0 - (1 / 4) + 6.0	X is given the value 9.0
X = (9 / 10) + 1.0 - (7 / 16)	X is given the value 1.0
J = (9 / 10) + 1.0 - (7 / 16)	J is given the value 1

In summary then, if you "mix" arithmetic by mixing fixed and floating constants or variables, the rule is that FORTRUNCIBLE always initializes its arithmetic to FIXED at EACH parenthesis level and continues that way until it encounters a floating variable or constant at the same level in which case the arithmetic stays floating at that parenthesis level. The search takes place from the RIGHT. The final substitution is done in the arithmetic of the left hand side variable. Floating point answers are always rounded to eight significant figures, but fixed point numbers are never rounded -- all figures to the right of the decimal point are dropped. (Thus, 8/9 = 0 in fixed point division!)

c) EXTENSIONS. In addition to these basic operations, a wide variety of special functions or "extensions" can be added to the compiler, making it extremely versatile. Rules for their use are given in the write-up supplied with each individual extension which might be in the subroutine library at your installation. If the needed routine is not available, then you can add your own coding in SOAP III form [4] following the rules described in the RUNCIBLE I - EXTENSIONS manual [3].

An extension is referred to in Fortruncible language as though it were a subscripted named variable, but without being defined as an

array. (ie. no ARRAY declaration statement). As a consequence of this fact, every subscripted variable which has not been defined as an ARRAY will be considered to be an EXTENSION.

Rule 18: The input arguments to an extension may be either floating point or fixed point as defined internally in the extension.

Rule 19: The rules for forming the name of the extension are the same as those for subscripted variables.

Rule 20: The maximum number of input arguments to an extension is ten. (10).[1]

Rule 21: The resulting output from an extension will be assumed to be floating point unless the name is preceded by a decimal point to indicate that the output is in fixed point form.

Several examples of extensions are given below to indicate the manner in which the names are formed.

Mathematical Notation	Fortruncible Notation
$\sqrt{X}$	RT2(X)
sin X	SIN(X)
e^X	EXP(X)
$J_n(X)$ (Bessel function)	BESSEL(N, X)
random number	.RNFX(1)

[1] A more complete rule for determining the maximum number of input arguments to an extension when several extensions are nested is described in the RUNCIBLE I - EXTENSIONS manual [3]

It should be noted that the fourth example above very closely resembles standard matrix notation and that CARE should be exercised in making certain that all subscripted variables are defined as ARRAYs. The last example shown above indicates how an extension whose output is in fixed point form should be written. The output in this example is a fixed point random number.

As a final example, consider the Fortruncible representation of the "Wolontis function":

$$y = f(x) = \frac{\sin x}{\sqrt{1 + e^{-x^3}}}$$

which when written in Fortruncible language becomes:

```
Y = SIN(X)/RT2(1.0 + EXP( - X ** 3.0))
```

3. Calls. All words in the Fortruncible language which are neither operands nor mathematical operators are defined as call words. There is a fixed set of these words that have been defined in the language, and a list of these words is included below for reference purposes.

Rule 22: Under no circumstances should any problem variables or extensions be named the same as these call words in order to avoid confusing the translator. This also applies to the first five symbols in a name since they are the only ones retained by the compiler.

Call Word	Use
ARITHMETIC	Part of an arithmetic switch statement.
ARRAY	Defines a name as a subscripted variable.
BYPASS	A no-operation statement.
CVECTOR	Defines a column vector (See Appendix V).
DECIMAL	Part of an arithmetic switch statement.
EDIT	Special data edit subroutine.
END	Designates the end of a PROCEDURE declaration.
EXECUTE	Sets the link to a PROCEDURE section of coding.
FIXED	Defines a name as a fixed point variable.
FLOATING	Part of an arithmetic switch statement.
GO	Sets up a transfer of control to another section.
HALT	Sets up a computer stop instruction.
IF	Defines a conditional clause.
INTEGER	Defines a name as a fixed point variable.
JUMP	Same as GO.
PROCEDURE	Designates the beginning of a PROCEDURE declaration.
PROGRAM	Part of a program overlay statement.
PUNCH	Defines an output statement.
READ	Defines an input statement.
SET	Signifies a set error correction statement.
STATISTICAL	Designates a statistical read statement.
THRU	Varies.
TO	Varies.

B. STATEMENTS.

In section IIA the rules of word formation were described; in this section the rules for combining these words into statements or sentences will be described. A set of these statements will then constitute the program for solving the desired problem..

Rule 23: Each statement is given a number which is some integer less than 1000.

The order in which the statements are executed has no relation at all to the numbers on the statements -- they are eventually carried out in essentially the same order in which the original deck was compiled. It is sometimes helpful, however, to make the numbers consecutive in case the cards should get mixed up.

Rule 24: The statement number zero (0) is special, and is reserved for statements which are not going to be referred to in the program.

Since each non-zero statement essentially wastes one additional memory location as compared to zero statements, it is desirable for long programs to use as many zero statement numbers as possible in order to minimize memory space. It should also be emphasized that the largest statement number should be as small as possible consistent with rules 24 and 25 to further minimize space requirements.

Rule 25: Each non-zero statement number must be unique and must never appear on more than one statement in a program.

There are many types of statements in Fortruncible language, and the forms of those most frequently used are described below in detail, while less frequently used statements are described in the appendices.

1. Substitution statements. Perhaps the most frequently used statement in Fortruncible language is the substitution or replacement statement. In this statement, a variable has its current value replaced by the value of any mathematical expression; a new value is substituted for its former one. This substitution operation as in standard mathematical notation is denoted by an "equality" symbol (=). For example:

```
      Y = 0.0
      Z = Y / (X * X)
      I = I + 1
RHO(J) = N - 3        (N assumed to be fixed point)
```

In the first case Y is set to zero. The second example sets Z equal to Y divided by X squared. In the third illustration, I becomes equal to one greater than its former value. The last case computes the value of N minus 3 and assignes it to the RHO variable which has a subscript equal to the current value of J. Note that in this last example, Fortruncible will convert the fixed point right-hand side automatically into a floating point number before inserting it into RHO(J). Also RHO is assumed to have been defined as an ARRAY. Variables always retain their values until being changed by a substitution statement or a READ statement (see below) or perhaps an extension statement (see below).

2. Jump statements. Fortruncible normally executes statements in the order it receives them, but this sequence can be broken by a JUMP statement which tells the compiler to jump to a certain statement and continue from there. A JUMP statement is written simply as

```
JUMP TO k
```

where k is the number of the statement which should be executed next. A variable or a parenthesized arithmetic expression may also be used instead of k as long as the result is fixed point; e.g., if I and J

are fixed point variables, then JUMP TO J or JUMP TO (J + 3 * I) are valid statements.

Rule 26: k must be a positive fixed point constant.

3. Input statements. There are three different types of read statements is Fortruncible language. The most common form of read statement is written

READ

This statement will cause the computer to read in one or more data cards for the problem. All data for a program other than constants which are written directly in Fortruncible language enter the 650 via a READ statement. The form of the data cards is described later in the section on card formats.

A second form of read statement is

READ PROGRAM

This statement is used to overlay a portion of the program in memory when segmention is used for large programs. This routine is not standard and the extensions manual should be consulted on the use of this statement.

The third form of read statement is

STATISTICAL READ N, k

This statement is a special one for statistical problems and the Statistical Extensions for Runcible I manual should be consulted for the use of this statement.[5].

4. Output statements. The output statements are used to punch answers onto cards. There are three different kinds of PUNCH statements:

a) The first form of PUNCH statement is used to punch the current values of up to four (4) variables onto one card. The statement

PUNCH TAU

will cause the current value of the variable TAU to be punched on a card;

PUNCH RHO PUNCH J PUNCH OMEGA

will put the values of RHO, J, and OMEGA all on the same card. Up to four (4) variables can be punched at a time in this manner.

PUNCH BETA(I, J + 1)

where BETA has been defined as an array and I, J have been defined as integers is also allowable -- it will punch the element of the array BETA which is specified by I and J + 1. It is not legal, however, to give a statement like PUNCH (I + J) or PUNCH - RHO.

b) If a large number of consecutive variables, which are a portion of an array, are to be punched, then a statement such as

PUNCH BETA(2,1) THRU BETA(6,4)

may be given. When the word "THRU" is used like this, up to seven (7) answers will be put onto each card, and successive cards will be punched until the number of variables designated by the PUNCH statement have been exhausted.

c) If the entire set of variables in an array are to be punched out, then a statement such as

PUNCH BETA

where BETA has been defined as an array, may be given. This statement will cause the entire array BETA to be punched out in the seven-per-card

form as described in part b. ONLY one (1) array may be punched out on a single statement, and separate statements must be written for each array to be punched.

Formats of the output cards from a PUNCH statement are described in a later section. They are identical to the formats required by the READ statement, so answers from one program may be used as data for another.

An output statement with a zero (0) statement number has a special meaning:

> Rule 27: An output statement whose statement number is zero (0) will be considered as a conditional output statement. If the console switch of the 650 is set plus during the running phase, the statement will be skipped (omitted), but if it is minus, the statement will be executed normally.

This type of statement is very handy for obtaining intermediate answers when checking a program out in its first few trial runs.

5. Halt statement. A halt statement will stop the 650 if the programmed switch on the console is set to stop.[1] It is simply written as

HALT

A number may be written after the word HALT like this:

HALT 12

in this case the machine will stop displaying the number 12. This technique may be used to differentiate between several HALTs in the same program.

---

[1] Control will proceed to the next statement (if any) if the program start switch is depressed after a halt. A programmed halt can be identified by its data address of 8003.

6. Bypass statement. The BYPASS statement, written (as might be guessed)

BYPASS

performs no arithmetic operation, but is extremely useful as a common ending point for several of the other statements (see iteration statements).

7. Extension statements. Some extensions which may be used with the compiler have no specific output. The form taken on by such statements is specialized and varied; rules are given in the writeup for each individual subroutine. However the general form is

$$\text{NAME}(V_1, V_2, \ldots, V_n)$$

where V_1 through V_n are the input expressions to the extension.

8. Conditional clauses. Any of the above statements may be made conditional by adding a conditional clause to the end of the statement. A conditional clause is formed in the following manner:

> Rule 28: A conditional clause must begin with the word IF and must contain one and only one of the three allowable relations: =, >, >= (equals, greater than, and greater than or equal). Any number of arithmetic operations may be included within a conditional clause.

For example, consider:

```
IF X = Y
IF J >= 2
IF $ Z - W $ > 1(-8)
```

In the last example, the relation is satisfied if the magnitude of Z - W is greater than 10^{-8}. Several examples of conditional clauses added to statements are:

1. JUMP TO 1 IF TEST = 0.0
2. READ IF 1 >= KEY
3. HALT IF RHO(J + 2) = RHO(J + 1)
4. PUNCH XO IF -(J / 5) >= 4 * J
5. Y = X / 10.0 IF Z * Z > X ** 8.4
6. JUMP TO K IF N = X
7. X = 3.1415927 IF SIN(X) = 0.0 IF X > 1.571

The left-hand portion of the statement is executed only if the relation is satisfied; when the condition is not fulfilled, the statement is treated like a BYPASS. If N is assumed to be fixed point while X is floating point in example 6, then the example is still valid indicating that mixed arithmetic in conditional clauses is valid. The seventh case is interesting because the substitution will be done only if both conditions hold. Any number of conditional clauses may be used in one statement.

9. Iteration statement. An "iteration statement" is a convenience which frees the programmer from programming loops. It precedes the portion of the program to be iterated on and has the following form:[1]

n, v1, v2, v3, v4,

where v1 is the variable which is to be changed; v2 is its starting value and v4 is its finishing value; and v3 is the amount (increment) by which the variable v1 is to be changed before repeating the sequence of statements again.[2]

---

[1] This process can also be programmed without using an iteration statement.

[2] If v4 - v2 is not exactly divisable by v3, the iteration procedure will be discontinued just before the value of v4 is passed.

The n in this statement refers to a statement number: All statements after the iteration statement up to and including the statement n will be repeated for every value of the variable v1 as defined by the iteration statement.

Rule 29: The statement number represented by n must not be zero (0).

Rule 30: The statement number of the statement immediately following the iteration statement must not be zero (0).

Rule 31: The variable v1 must be a simple named variable.

Rule 32: The quantities v2, v3, and v4 must be either simple named variables or short constants (ie. constants which contain no more than five (5) symbols including punctuation).

Rule 33: To avoid difficulties, the statement whose number is n should be a BYPASS statement.[1]

Rule 34: All commas (,) must occur in an iteration statement exactly as indicated in the form, especially the one at the end of the statement.

For example, the statement

5, X, 1, 2, 13,

means: execute all statements from the next one through the statement numbered 5 for X taking on the values 1, 3, 5, 7, 9, 11, and 13. It is possible to include an iteration statement within the scope of another iteration statement -- all of the details will be handled automatically

---

[1] An iteration statement may end on any statement which is not related to a procedure or matrix operation statement.

by Fortruncible.[1]

Rule 35: Iteration statements may be nested no deeper than four (4).

Rule 36: n must be a positive fixed point constant.

Rule 37: If v3 is to be a negative increment then it must be preceded by a minus (-); if it is to be positive then it must not start with a minus.

For example:

2, COUNT, MAXIMUM, - DELTA, MINIMUM,

is an iteration statement with a negative increment. The variable COUNT starts out at its MAXIMUM value and is decremented by the amount DELTA until it reaches its MINIMUM value which closes off the iteration statement. An iteration statement of the form:

2, COUNT, MAXIMUM, DELTA, MINIMUM,

would be incorrect even though DELTA itself is negative, since the translator would not be able to recognize that a negative increment was being used. Hence, it would compile the wrong testing procedure for the end of the iteration statement.

10. Array operation statements. For the convenience of the programmer who is using vectors and matrices, a statement has been defined in Fortruncible language which permits the operations of plus (+), minus (-), and product (*) to be interpreted as the corresponding array

---

[1] The scope of every iteration statement must be contained in the scope of any other iteration which uses it; that is, if a certain statement iterates on statements 1 through 5, say, no meaningful iteration statement within these bounds will terminate at any statement after five.

operations in a special form of replacement statement. The general form of this type of statement is:

$$A = B \circ C$$

where A, B, and C have been defined as ARRAYs and o is any of the operations +, -, or *.

Rule 38: Only one (1) operation is permissible per statement.

Rule 39: The form of the statement must have a definite meaning in the matrix sense (i.e., the operation must be defined mathematically).

Rule 40: Under certain circumstances the variables A, B, or C may be interpreted as scalars or scalar matrices (see examples below).

Rule 41: No consistency check on the size of the arrays is made by the compiler during translation. It will be assumed during compilation that the programmer has defined the sizes of the arrays properly.

In order to illustrate the above ideas and to summarize the types of forms which can be written, consider the following examples: in which A, B, and C are considered as n x n matrices, I, J, and K considered as vectors of size n, and X, Y, and Z are considered as scalar variables:

1. Simple replacement:

   a) Matrix by matrix: A = B
   b) Matrix by scalar matrix: A = X or A = 2, where 2 is considered as 2 times the identity matrix.
   c) Vector by vector: I = J

2. Products:

a) Matrix product: A = B * C, A = 4 * X, A = B * 42, A = X * C, or A = B * Y.

b) Matrix vector product: A = I * J

c) Vector matrix product: I = B * J

d) Scalar product: Z = J * K

3. Sums:

a) Matrix sum: A = B + C, A = 39 + C, or A = B + 121

b) Vector sum: I = J + K

4. Differences:

a) Matrix difference: A = B - C, A = 1 - C, or A = B - 21

b) Vector difference: I = J - K

It can be noted from the above examples that constants may be used as scalar matrices.

Rule 42: If a constant is to be considered as a scalar matrix, then it must be written as an integer of no more than four (4) digits.

During the translation of these statements, the compiler requires additional non-zero statement numbers which it will generate as it needs them starting with the maximum statement number as the programmer has defined it. The purpose of these statement numbers is to provide re-entry points in the program for the iteration statements which are generated internally for the matrix operations.

Rule 43: Array operation statements should not be nested any deeper than one (1) in external iteration statements since under some conditions three internal iteration statements are generated.

Rule 44: No external iteration statement should terminate on an array operation statement.

11. Procedures. As a further convenience to the programmer, a special set of statements have been included in Fortruncible language which permit the programmer to write subroutines directly in Fortruncible language. These subroutines called procedures can then be linked up with the main program as often as desired. The rules governing the formation of these subroutines or procedures are listed below.

Rule 45: Every set of statements that are to be considered as a procedure must represent a subroutine which is complete in itself.

Rule 46: The first statement in a procedure must be a PROCEDURE declaration statement as defined below.

Rule 47: The last statement in a procedure must be an END of procedure statement.

Rule 48: All named variables used within a procedure will not be bound to the procedure but will carry the same meaning outside of the procedure. Consequently, care must be exercised when writing a procedure to prevent destroying any results needed outside the procedure.

Rule 49: All procedures are exited by transferring control to the END statement either by the natural sequence of statements or by means of a JUMP statement.

Rule 50: The set of statements which represent a procedure must occur in a program before any statement which links the procedure.

The special statements in Fortruncible language relating to procedures will be considered next along with their rules of formation.

12. Procedure declaration statement. The first statement in a procedure as mentioned above must be a procedure declaration statement which defines the procedure. The form of this statement is:

$$\text{PROCEDURE NAME}(I_1, I_2, \ldots, I_j) = (O_1, O_2, \ldots, O_k)$$

where a) NAME is a name for the procedure.

b) I_1 through I_j are the internal names of the input variables to the procedure.

c) O_1 through O_k are the internal names of the output variables of the procedure.

Rule 51: The name of a procedure may be any length but it will be truncated to four (4) characters that must be unique to the program.

Rule 52: The input variables must not contain parentheses or commas. Hence the only admissible variables are simple named variables or entire arrays written without parentheses or commas.

Rule 53: The output variables also must not contain parentheses or commas.

Rule 54: The maximum number of input and output variables combined is five (5). (i.e. the relation $j + k \leqslant 5$ must be satisfied.)

Rule 55: There must be at least one input and one output variable in every procedure. (i.e. $j \geqslant 1$ and $k \geqslant 1$.)

Rule 56: The statement number of every procedure declaration must be unique and hence different from zero.

Rule 57: Procedure declarations must not be nested, but other procedures may be executed from within a given procedure.

Rule 58: The maximum number of different procedures in a given program is four (4).

Rule 59: Each procedure declaration defines one fixed point named variable.

13. End statements. These statements are used to end each procedure and have the form

END NAME

where NAME is the name of the procedure that is being ended.

Rule 60: The name associated with each END statement should agree with the name of the corresponding PROCEDURE declaration.

Rule 61: Each END statement must have a unique statement number.

14. Execute statements. Whenever it is desired to link a procedure, an EXECUTE statement is used. This type of statement has the following form:

$$\text{EXECUTE NAME}(E_1, E_2, \ldots, E_j) = (V_1, V_2, \ldots, V_k)$$

where a) NAME is the name of a previously defined procedure.

b) E_1 through E_j are the current input expressions for the procedure.

c) V_1 through V_k are the external variable names for the results of the procedure after it is executed.

Rule 62: The number of input expressions (j) must agree with the number of input variables in the defined procedure.

Rule 63: The input expressions may contain any number of arithmetic operations but must not contain any parentheses or commas.

Rule 64: The number of output variables (k) must agree with the number of output variables in the defined procedure.

Rule 65: The names of output variables must be simple named variables or entire arrays and must not contain parentheses or commas.

Rule 66: Every EXECUTE for a procedure must follow physically the procedure declaration statement.

Rule 67: EXECUTE statements may be used within other procedure declarations but they must refer to previously defined procedures.

Rule 68: The statement number of an EXECUTE statement must be unique.

Rule 69: Any number of EXECUTE statements may be used in a program consistent with the memory space.

Rule 70: Array operations may be used as input expressions in an EXECUTE statement, but the corresponding variable in the procedure must also be an array.

Rule 71: No iteration statement should terminate on an EXECUTE statement, but EXECUTE statements may be contained within iteration statements consistent with the other rules governing the nesting of iteration statements.

This statement will first substitute the values of the input expressions into the input variables of the procedure before transferring to the procedure. After completing the procedure, the corresponding results are substituted back into the defined problem variables before returning to

the main program sequence. The third example in section II.C will illustrate the use of the above ideas, and further examples of this type can be found in Appendix IV.

15. Integer definition statements. An INTEGER definition statement is used to define a list of simple named variables as fixed point variables (integers). The form of this statement is:

$$\text{INTEGER } V_1 \; V_2 \; \ldots \; V_n$$

where V_1 through V_n is a list of simple named variables.

Rule 72: Each name in the list must be separated by a space or comma.

16. Array definition statements. All variables which are to be considered as subscripted variables must be defined as such by an ARRAY definition statement. There are several forms of the statement depending on whether the array is a matrix or vector.

a) If the subscripted variable is a vector then the definition statement has one of the forms

1. ARRAY NAME(n)
2. ARRAY NAME(N) k

where in the first form NAME is the name of the vector and n is an integer representing the size of the vector. In the second form, NAME is the name of the vector, N is a fixed point variable (integer) that represents the current size of the vector, and k is an integer that designates the amount of space in memory to allocate for the vector.

b) If the subscripted variable is a matrix then the definition statement has one of the forms

1. ARRAY NAME(m, n)
2. ARRAY NAME(M, N) k

where in the first form NAME is the name of the matrix and m, n are integers representing the size of the matrix (i.e. NAME is an m by n matrix, m is the row size and n is the column size). In the second form, NAME is the name of the matrix, M, N are fixed point variables (integers) which represent respectively the row size and column size of the matrix, and k is an integer which designates the amount of space in memory to allocate for storing the matrix.

Consider the following examples:

ARRAY RHO(19)

This defines the vector RHO to be of size 19.

ARRAY TAU(I) 100

This defines the vector TAU to be of current size I, but to allow a maximum size of 100 for TAU and 100 locations in memory will reserved for this vector.

ARRAY BETA(6, 9)

This statement defines the matrix BETA to be 6 by 9. But

ARRAY BETA(I, J) 225

defines the matrix to be currently of size I by J but reserving 225

locations in memory to store the matrix. In the above examples it has been assumed that the named variables I and J have been defined as fixed point variables (integers).

Rule 72: Every ARRAY definition must contain an integer or integers which designate the amount of spece to be reserved in memory at the time of translation.

Rule 73: Every variable which has the form of an array but has not been defined as such will be considered to be an extension.

Rule 74: If the size of an ARRAY is defined in terms of a fixed point named variable, then this variable must have an assigned value at running time when this array definition is encountered.

Rule 75: Every ARRAY definition generates coding in the machine language program.

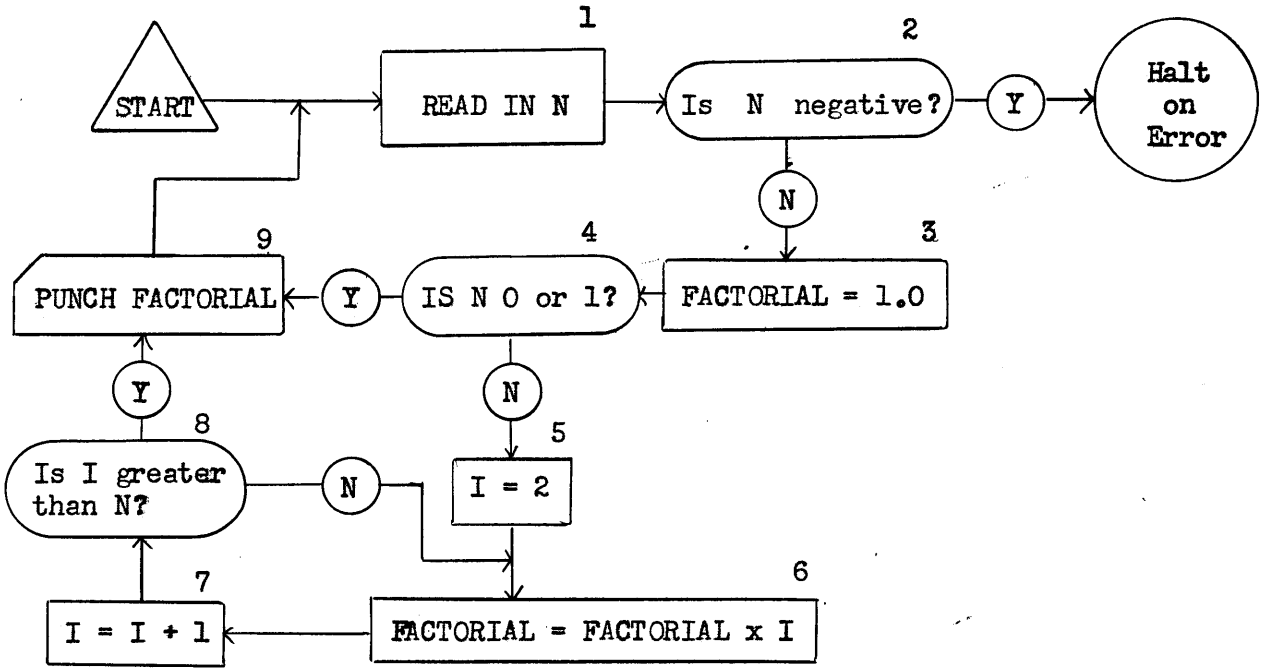
Rule 76: Every ARRAY definition generates a named fixed point variable which corresponds to the origin of the array.

C. EXAMPLE PROBLEMS.

Three sample problems will be given here to demonstrate some portions of Fortruncible language as it has been described; more examples may be found in Appendix IV.

Example 1. Calculate FACTORIAL N where N is a non-negative integer.

Solution - let N be a fixed point variable which will be the input data to this problem; FACTORIAL (the answer) will be floating point; and the computation will proceed by successive multiplications of the result by a variable I which will run through the integers up to N. A flow chart to solve the problem would look something like this:



The translation of the flow chart into a series of statements is almost automatic:

Number	Statement
0	INTEGER N I
1	READ
2	HALT IF 0 > N
3	FACTORIAL = 1.0
4	JUMP TO 9 IF 1 >= N
5	I = 2
6	FACTORIAL = FACTORIAL * I
7	I = I + 1
8	JUMP TO 6 IF N >= I
9	PUNCH N PUNCH FACTORIAL
10	JUMP TO 1

Observe that the program parallels almost exactly the instructions you would give to a person telling him what you wish to be done. The following is the same program using an iteration statement:

```
0    INTEGER N I
1    READ
2    HALT IF 0 > N
3    FACTORIAL = 1.0
4    JUMP TO 7 IF 1 >= N
5    6, I, 2, 1, N,
6    FACTORIAL = FACTORIAL * I
7    PUNCH N PUNCH FACTORIAL
8    JUMP TO 1
```

Example 2. Suppose we want to evaluate the error function

$$Q(X) = (2/\sqrt{\pi}) \int_0^X e^{-t^2}\, dt$$

for arbitrary values of X using the approximation

$$Q(X) = 1 - (2/\sqrt{\pi})(a_1 n + a_2 n^2 + a_3 n^3 + a_4 n^4 + a_5 n^5)\, e^{-X^2}$$

where $n = 1/(1 + pX)$ (p and the a's are numerical coefficients which we will omit here). We will let a_k = A(K); X = X; n = N; Q(X) = ERF (the answer): and p = P. $2/\sqrt{\pi}$ = 1.1283791. We could simply evaluate the equations directly with the following program:

Number	Statement	Comments
0	ARRAY A(5)	(define array A)
1	READ	(read in a_k, p, x)
2	N = 1.0/(1.0 + (P * X))	(calculate n)

```
3       ERF = 1.0 - (1.1283791 * ((A(1) * N)+(A(2) * (N**2))+
            (A(3) * (N**3))+(A(4) * (N**4))+(A(5) * (N**5))) *
            EXP(-X * X)                    (calculate Q(X))
4       PUNCH X PUNCH ERF                  (punch answer)
5       JUMP TO 1                          (read in another X
                                             and continue)
```

But the evaluation of the polynomial will be quite a bit more rapid if we rewrite the expression

$$Q(X) = 1 - (2/\sqrt{\pi})(N(a_1 + N(a_2 + N(a_3 + N(a_4 + Na_5))))) e^{-X^2} .$$

Now we could evaluate this new expression directly or set up a "loop" type of routine which calculates the polynomial from the inside out. Careful study of the program below will be very instructive.

```
0       INTEGER K
0       ARRAY A(5)
1       READ
2       N = 1.0/(1.0 + (P * X))
3       TEMP = A(5)
4       5, K, 4, -1, 1,
5       TEMP = A(K) + (N * TEMP)
6       ERF = 1.0 - (1.1283791 * N * TEMP * EXP(-X * X))
7       PUNCH X PUNCH ERF
8       JUMP TO 1
```

Of course an even shorter program would be

```
1       READ
2       NORMAL = ERF(X)          (error function extension)
3       PUNCH X PUNCH NORMAL
4       JUMP TO 1
```

.... but this is cheating.

Example 3. Suppose that we wish to compute the area under the Normal probability curve between the limits C and D. This area is given by

$$Q(C,D) = (2/\sqrt{\pi}) \int_C^D e^{-t^2}\, dt$$

One method of evaluating the integral would be to perform a direct numerical integration, but in this case a much simplier method is to use the results of Example 2 as a subroutine. This latter method is the one we will use in this example.

Number	Statement	Comments
0	INTEGER K	
0	ARRAY A(5)	
1	READ	(read in a's and p)
2	PROCEDURE NORMAL(X) = (ERF)	(define subroutine NORMAL)
3	N = 1.0/(1.0 + (P * $ X $))	
4	TEMP = A(5)	
5	6, K, 4, -1, 1,	
6	TEMP = A(K) + (N * TEMP)	
7	ERF = 1.0-(1.1283791 * N * TEMP * EXP(-X * X))	
8	ERF = 0.5 + ((X/$ X $) * ERF)	
9	END NORMAL	
10	READ	(read in C and D)
11	EXECUTE NORMAL(D) = (UPPER)	
12	EXECUTE NORMAL(C) = (LOWER)	
13	AREA = UPPER - LOWER	
14	PUNCH C PUNCH D PUNCH AREA	
15	JUMP TO 10	

For a more extensive example which uses both procedures and matrix operations see Appendix IV.

III. CODING DETAILS.

A. BASIC PACKAGES.

Standard subroutines such as floating-point arithmetic and input-output operations have been incorporated into "basic packages" which augment the finished program in its running stage. There are many of these packages, each of which has its own special purpose or goal, and the requirements of each individual program will determine just which one to use.

If you are going to run your program on an "ordinary" 650, then you must use one of the "A" packages; P1A, P2A, or P3A. If your program is to be run on an augmented 650 (floating point, index registers, etc.) then you use one of the "Y" packages; P1Y, P2Y, or P3Y. All of the packages mentioned so far include the necessary routines for READ, PUNCH, and other various and sundry necessities of life for FORTRUNCIBLE programs. The P1 packages contain a bare minimum of things and are used in most cases. The extra things included in the other two flavors are listed below:

P2 :	P3 :
P operator Logarithm (base e) LN Exponential (base e) EXP	P operator Log (base e) LN Expon (base e) EXP
	Sine (radian) SIN Cosine (radian) COS Arctangent (rad) ATAN Square root RT2

The decision as to which one of the packages to use is now simply made by examining the above lists. If your program does not use any of the things in either list -- then use P1. If your program uses only things listed above the dashed line -- then use P2, if your program uses features below the line, then use P3.

The sizes of the commonly available basic packages are listed below

Basic 650	Augmented 650
P1A - 325 locations	P1Y - 190 locations
P2A - 525 locations	P2Y - 365 locations
P3A - 751 locations	P3Y - 480 locations

B. CARD FORMATS.

Since the Fortruncible compiler translates the Fortruncible program into a standard Runcible program, only those card formats specifically related to the Fortruncible part of the program will be described and the programmer should refer to the RUNCIBLE manual for details concerning the other card formats. There are three types of cards used directly in the Fortruncible program: the HEADER card, the COMMENTS card, and the FORTRUNCIBLE STATEMENT cards.

1. Header card. The header card for Fortruncible has the same format as that used in Runcible. The format is as follows:

Columns 1 - 30: all zero (0) filled.
Columns 31 - 40: the highest statement number used.
Columns 41 - 50: all zero (0) filled.
Columns 51 - 60: the number of locations used by the basic package (see above).
Columns 61 - 80: all zero (0) filled.

In addition Columns 10, 20, 30, 40, 41, 50, 60, 70, 80: have 12 (Y) overpunches. The punch in column 41 is necessary to identify the header card.

Any of these numbers may be made a little bit larger than the actual value (for safety) but it is extremely important that none of them are smaller than the true values, for this is an unchecked error which can lead to mysterious and unfortunate results.

2. Comments card. The comments card is generally easier to prepare than the header card. Its format is:

Columns 1 - 40:	all blank (no punches)
Column 41:	a one (1) punch
Column 42:	blank
Columns 43 - 72:	may be filled with a title or anything else the programmer's heart may desire (as long as it is acceptable to the alphabetic attachment). If he is lazy he may leave it blank.

3. Fortruncible statement cards. The Fortruncible program can be compiled only on a 650 with a special character attachment, group II: as a consequence, only those characters recognizable by the attachment can be used in punching the statements onto cards. IBM has defined two sets of characters for the same set of card punches, called FORTRAN and COMMERCIAL 407 characters. For convenience, the Fortruncible programs are written with FORTRAN symbols with suitable modifications.[1] A list of the Fortruncible characters along with

---

[1] To further add to the convenience of the user, IBM, when defining the FORTRAN set of characters, decided to include two minus symbols, each with a different card definition, due to the great abundance of available symbols. However, Fortruncible uses the "minus" symbol corresponding to the commercial @ symbol as the greater than (>) symbol.

their FORTRAN and COMMERCIAL equivalents is given below, and all programs described in this manual will be written in the Fortruncible notation.

Fortruncible	Fortran	Commercial	Meaning
(	(	%	Left parenthesis
)	)	¤	Right parenthesis
.	.	.	Decimal point
=	=	#	Substitution
=	=	#	Equals
>	-	@	Greater
>=	-=	@ #	Greater than or equal
,	,	,	Comma
+	+	&	Plus
-	-	-	Minus
*	*	*	Times
/	/	/	Divided by
$	$	$	Absolute value
**	**	**	Power

Each statement must be punctuated by a period (.), which is added at the end of the statement. The very last statement must end with a double period (..). The rules for punching the statements on cards are:

Rule 77: The statement number, n, is punched as (0000 + n) in columns 1 - 4.

Rule 78: Column 5 must have a letter "R" punched in it to identify the card as a Fortruncible card.

Rule 79: The statement itself is punched in columns 43 through 71, and column 72 must contain the period

which terminates it. (If a statement is so long it does not fit on one card, up to five (5) cards may be used. In this case the period should appear only on the last card of the statement, and columns 43 through 72 may be used for characters of the statement on all other cards. A statement may thus contain up to 150 characters, including the final period (.) or double period (..) as the case may be. Each of the cards in a multiple card statement must have the same statement number punched in columns 1-4.)

Rule 80: Columns 6 - 42 and 73 - 80 are ignored by Fortruncible except that columns 7 and 41 must not contain a 12 (Y) punch and column 10 must be blank. For the convenience of the programmer, columns 17 - 20 may contain a serial card number -- helpful for reordering the cards if 52 pick-up has been played. This serial number may be listed on the 407 along with the statements to facilitate inserting corrections.

Rule 81: Blank columns will be ignored in the middle of a statement as long as they do not occur in the middle of a name or constant. (Care must be exercised on multiple card statements to insure that no additional blanks are inserted in the middle of a name or that two names are spaced properly.)

Rule 82: The very last statement in the program is ended with a double period (..) which must occur on the last card of the statement in columns 71 and 72.

C. CORRESPONDENCE TABLE.

As previously mentioned, the Fortruncible program is translated into a Runcible program by the Fortruncible compiler. Since Runcible accepts only three types of singly subscripted variables, (I, Y, or C), it is necessary that the Fortruncible compiler translate all of the named variables into the above three types of variables. This translation is accomplished by setting up a correspondence between the fixed point named variables and Runcible "I" variables, the floating point subscripted named variables and Runcible "Y" variables, and the simple named variables with the Runcible "C" variables. This correspondence is stored in memory during the translation, and at the end of the translation it can be punched out for checking purposes. The card format of the correspondence table can be best described by considering the following sample correspondence table which has been printed in standard Runcible form.

Card no.	Statement no.		Statement			
0001	1800I	N	I0001			
0002	1801I	I	I0002			
0003	1802I	J	I0003			
0004	1803I	A 1	I0004			
0005	1804I	A 2	I0005			
0006	1805I	D 1	I0006			
0007	1806I	D 2	I0007			
0008	1807I	D 3	I0008			
0009	1808I	E 1	I0009			
0010	1850I	TAU	Y0001	I 4	10	10
0011	1851I	RHO	Y0002	I 5		10
0012	1900I	X	00001			
0013	1901I	MAX	00002			

The first three entries in the correspondence table define the equivalents of three simple named fixed point "I" variables. The next

two entries define the origins for two arrays, the first one is a matrix, and the second one is a vector. The next three fixed point names are "dummy" variables used by the array operations. Card 9 defines an exit location for a procedure. Cards 10 and 11 define arrays, the array TAU starts at Y0001 and uses 10 x 10 or 100 locations, with its origin defined by I 4. Thus, array RHO starts at Y0001 + 100 or Y0101 (the Y0002 is there only for indication purposes and 0002 means the second array, not the starting "Y" variable) and is a vector of size 10 with its origin defined by I 5. Cards 12 and 13 define the equivalents of the floating point variables X and MAX.

Another very important use for the correspondence table is in preparing the data cards for use in the final Machine language program. Since the intermediate program is in "Runcible" language, all data must be entered in standard Runcible format as described in the Runcible I manual. This means that the Runcible equivalents for each of the named variables has to be known in order to punch the data cards.

The correspondence table can also be used in rerun procedures (see Appendix III).[1]

D. DATA CARD FORMAT.

As mentioned above, all data used by the final running program must be entered in standard Runcible format.

Rule 83: All data is punched in the standard Runcible format where the equivalent Runcible variables are determined from the correspondence table.

Rule 84: All output results from a Fortruncible program are

---

[1] The same variable definitions can be retained between two programs by following the rules outlined in Appendix III for rerun procedures.

punched out on cards in the standard Runcible format using the equivalent Runcible variables as identification. This output can be used as input to another program, provided the variable definitions or correspondences have remained consistent.

IV. OPERATING PROCEDURES.

A. MODES OF OPERATION.

There are four (4) normal and one special modes of operation available for the Fortruncible to Runcible translation which depend upon the special equipment that is available on the 650. However, all the modes require that the 650 has the complete special character device, group II, on the 533 alphabetic attachment. The special mode of operation is called FAR (Fortruncible Automatic Runcible) and is described in detail in Appendix V. The four normal modes of operation are described below.

1. Basic 650. This mode uses only the basic 650 with the full alphabetic attachment as mentioned above. This mode is obtained by loading the Fortruncible deck with the storage entry switches set plus. As might be guessed, this form of operation is the slowest and should be used only when none of the other modes is applicable.

2. TLE 650. This mode is a modification of mode 1 in which the special operation code of Table Look-up on Equal is used to greatly increase the speed of compilation. Only those 650's which have this added special order code can use this version, which is obtained by loading the Fortruncible deck with the storage entry switches set minus. A test is made at the start of this program to see if the TLE 63 operation code is operative.

3. Augmented 650. This mode of operation is available for use on a 650 with the following added equipment:

1) Immediate Access Storage (Core Storage)
2) Indexing Registers

and is obtained by loading the Fortruncible S deck with the storage entry switches set plus. All rules of operation are the same as those for the other modes and the only difference is that the above equipment is used internally by the compiler.

4. Augmented 650 with TLE. This version is similiar to mode 3 except that the Table Look-up on Equal operation code has been used. This mode is obtained by loading the Fortruncible S deck with the storage entry switches set minus. The additional equipment required by this mode of operation is:

1) Immediate Access Storage (Core Storage)
2) Indexing Registers
3) Table Look-up on Equal (63)

B. FORTRUNCIBLE PROGRAM.

Your Fortruncible program consists of three parts:

1) Header Card
2) Comments Card
3) Fortruncible Statement Cards (last card has a double period (..) in columns 71 and 72.

which are assembled together in the above order (i.e. first the header card, second the comments card, and last the Fortruncible statements). The above collection of cards in the proper order constitutes a Fortruncible program.

C. RUNCIBLE PROGRAM.

The Fortruncible compiler translates your Fortruncible program into a standard Runcible program which uses none of the special alphabetic characters. The card order in the resulting Runcible program is:

1) Comments Card
2) Runcible Statement Cards
3) New Header Card (modified by the Fortruncible compiler)
4) Correspondence Table Cards

This order is not proper for the Runcible compiler which requires that the cards have the following order:

1) Header Card
2) Comments Card
3) Runcible Statement Cards

Consequently, the cards have to be rearranged to agree with the above order. To facilitate finding the new Header card in the output cards, a programmed stop is encountered immediately after punching the new Header card, and before punching the correspondence table. If the output cards are cleared out of the 533 punch unit at this point, the last card, which is the new Header card, can be placed first to form a properly ordered Runcible program, and then the correspondence table can be punched. This Runcible program can now be processed by the Runcible compiler in the usual manner following the rules described in the RUNCIBLE I manual.

D. OPERATOR INSTRUCTIONS.

Step 1. Insert FORTRUNCIBLE - RUNCIBLE I plugboard into 533 unit. This board will be used during the entire operation.

Step 2. Clear any cards out of the read feed and punch feed

and ready the punch feed with ample blank cards.

Step 3. Place either the FORTRUNCIBLE or FORTRUNCIBLE S deck in the read hopper, 12 edge in, face down. Which deck you use depends upon the extra hardware that is available on the 650 being used. (See above discussion.) Place your Fortruncible program on top of the FORTRUNCIBLE deck in the read feed, where your program consist of 1) Header Card 2) Comments Card and 3) Statements (in that order).

Step 4. Set the 650 console to the following:

SWITCH	SETTING
STORAGE ENTRY	70 1951 klmn ± (see Modes of Operation as described in the Runcible I manual)
PROGRAMMED	STOP
HALF CYCLE	RUN
ADDRESS SELECTION	1888
CONTROL	RUN
DISPLAY	LOWER ACCUM
OVERFLOW	SENSE
ERROR	STOP

Step 5. Depress the following buttons in order:

1) COMPUTER RESET
2) PROGRAM START

Step 6. Press both START buttons on the 533 unit.

Step 7. When the last card in the read hopper is halfway into the machine, depress END OF FILE. Do not push the START button again until the "End of File" light goes out.

Step 8. When the computer stops with the "Address Lights" displaying 0999, run the cards out of the punch feed and throw away the top and bottom cards. These cards are "garbage cards" and should have been punched identically.

Step 9. Now move the bottom card to the top of the set of output cards. This card is the new Header card and the resulting set of cards constitutes the Runcible program which should be saved for the next part of the operation.

Step 10. If the correspondence table is not desired skip down to step 12. Otherwise, depress the START button on the punch side of the 533 unit (adding blank cards if necessary) and depress PROGRAM START.

Step 11. This part of the translation is completed when the "Address Lights" display 9876. Run the cards out of the punch feed and again throw away the top and bottom cards. The remaining cards are the correspondence table and should be saved for future reference.

Step 12. Now perform the Runcible to machine language translation by following the rules set forth in the RUNCIBLE I manual.

V. STOPS.

(indicated in the "Address Lights")

A. NORMAL STOPS.

1) 0999 End of compiling phase. Remove cards before depressing program start and punching out the correspondence table.

2) 9876 This is the end-of-Fortruncible indication after the correspondence table has been punched out.

B. ERROR STOPS.

1) 9999 The Header card has been omitted from the program.

2) 9901 There are more than five (5) cards to a statement.

3) 1234 This is the normal error indication for all errors (correctable) detected during compilation. The statement number of the current statement being processed (may be a newly generated statement number) is displayed in the Upper Accumulator. The cards should be run out of the read feed on the 533 and the offending card/or cards may be corrected and reinserted in the read feed. Depressing the PROGRAM START button will initiate a card read and continue the translation process.

Sorry to say, most errors in statement formation will not be detected by the Fortruncible compiler but will be passed on to the Runcible I phase in erroneous form to be detected by Runcible I. One word of caution: Array operations and Execute statements can not be corrected but must be recompiled.

BIBLIOGRAPHY

1. A.J. Perlis, J.W. Smith, H.R. Van Zoeren, "Internal Translator (IT) A Compiler for the IBM 650", Computation Center, Carnegie Institute of Technology, January 1957.

2. Computing Center Staff, "Runcible I", Vol 1 Series V Revised Edition, Computing Center, Case Institute of Technology, March 1959.

3. Computing Center Staff, "Runcible I Extensions - Part I", Computing Center, Case Institute of Technology, March 1959.

4. D.E. Knuth, "SOAP III", Vol 1 Series IV, Computing Center, Case Institute of Technology, February 1958.

5. Computing Center Staff, "Statistical Extensions for Runcible I", Computing Center, Case Institute of Technology, September 1959.

A P P E N D I X I.

SUMMARY OF STATEMENTS.

1. Substitution: NAME = E_1 , where NAME is any named variable and E_1 is any expression.

2. Jump: JUMP TO k , where k is the number of the statement to be executed next. k may be a parenthesized fixed point expression.

3. Input: READ , normal data read.
READ PROGRAM , for program overlay.
STATISTICAL READ n, k , statistical data read.

4. Output: PUNCH $NAME_1$ PUNCH $NAME_2$ PUNCH $NAME_3$ PUNCH $NAME_4$, punches four per card form for any four named variables.
PUNCH $ARRAY_1$(I,J) THRU $ARRAY_1$(K,L) , punches any portion of an array in seven per card form.
PUNCH $ARRAY_1$, punches the entire array in seven per card form.

5. Halt: HALT k , stops the computer with the number k displayed.

6. Bypass: BYPASS , does absolutely nothing.

7. Extension: NAME(E_1, ... E_n) , where NAME is the name of the extension and E_1 through E_n are input expressions.

8. Conditional: IF E_1 r E_2 , where E_1 and E_2 are expressions and r is one of the relations >, >=, or =.

9. Iteration: n, V_1, V_2, V_3, V_4, , where V_1 is the variable to be changed, V_2 its initial value, V_3 its increment, and V_4 its final value. n is the statement number of the last statement to be included in the iteration.

10. Array operation: $ARRAY_1$ = $ARRAY_2$ o $ARRAY_3$, where $ARRAY_1$, $ARRAY_2$ and $ARRAY_3$ are names of arrays without subscripts and o is one of the operations + - or *. The statement must make sense.

11. Procedure declaration: PROCEDURE NAME(I_1, ..., I_j) = (O_1, ..., O_k) defines the subroutine named NAME with I_1 thru I_j as input variables and O_1 thru O_k as output variables.

12. End: END NAME , defines the end of a procedure or subroutine.

13. Execute: EXECUTE NAME(E_1, ..., E_j) = (V_1, ..., V_k) , performs the procedure named NAME with E_1 thru E_j as input expressions and V_1 thru V_k as output variables.

14. Integer: INTEGER $NAME_1$ ... $NAME_n$, defines the names $NAME_1$ thru $NAME_n$ as fixed point variables (integers).

15. Array: ARRAY $NAME_1$(m,n) , defines the name $NAME_1$ to be an array of size m by n.

16. Special: ARITHMETIC DECIMAL , changes arithmetic to decimal mode.

ARITHMETIC FLOATING , changes arithmetic to floating point mode.

SET ERROR CORRECTION TO n , defines an error correcting routine to start at statement n.

EDIT $NAME_1$ THRU $NAME_2$, used by a special editing routine.

A P P E N D I X I I .

PASSING "RUNCIBLE I" STATEMENTS.

Any number of Runcible statements in standard Runcible format can be intermixed with the Fortruncible statements in a Fortruncible program provided that they are not the first and last statements in the program. The first and last statements must be Fortruncible statements. These Runcible statements will be ignored by the Fortruncible compiler and passed on directly to the Runcible output program in unaltered form. However, there is a strong possibility that these Runcible statements will use standard I, Y, and C Runcible variables, so in order to avoid confusion with those defined by the Fortruncible compiler as it translates the Fortruncible named variables into Runcible I, Y, and C variables, the Header Card for the Fortruncible program must be modified in the following manner:

Columns 1 - 10	maximum subscript of the passed I variables.
Columns 11 - 20	maximum subscript of the passed Y variables.
Columns 21 - 30	maximum subscript of the passed C variables.
Columns 31 - 40	maximum statement number in program including both Fortruncible and Runcible statements. Note: The Runcible and Fortruncible statements should have unique statement numbers if they are non-zero.
Columns 41 - 80	same as before.

For efficiency, all of the variables used by the Runcible portion of the program should be clustered together and should have as small a subscript as possible. The number of variables used by the Runcible portion of the program does not affect the restrictions on those used by the Fortruncible portion except that the total number of resulting Runcible "I" variables must be less than 99. (i.e. I 99 is the largest subscripted I variable that is permissible.)

A P P E N D I X I I I .

RELOADING A CORRESPONDENCE TABLE.

Occasionally the situation arises where it is desirable to use the correspondences for one program with another for compatibility. This is especially true when segmenting a program. Under these circumstances it is desirable to be able to reload the correspondence table from one program to define the correspondences for another. This operation is possible provided certain rules are followed.

Rule 85: The old correspondence table if it is to be reloaded must follow immediately the Comments Card of the new program but before any of the statements.

Rule 86: The correspondence table if reloaded will take precedence over any information on the Header Card.

Rule 87: The card order of the correspondence table should be retained for proper reloading.

Rule 88: All array definitions will be retained, but these arrays will not be initialized in this program since the initialization steps are generated from an ARRAY statement. This rule is very important. If it is necessary to reinitialize an array then the definition should be removed from the correspondence table and an ARRAY definition statement inserted in the program to define and initialize the array.

A P P E N D I X I V .

SAMPLE PROBLEMS.

Example Sub-Program 1. Separate the integral and fractional parts of a floating point variable X.

```
0        INTEGER N                          .
1        N = X                              .
2        INTGR = N                          .
3        FRACT = X - INTGR                  .
```

N is the integral part of X in fixed point form; INTGR is the same in floating point form; and FRACT is the fractional (decimal) part of X in floating point form.

Example Sub-Program 2. Represent the eight significant digits of a floating point variable X as a fixed point integer N.

```
0        INTEGER N                          .
1        JUMP TO 5 IF X >= 1(8)             .
2        JUMP TO 7 IF 1(7) > X              .
3        N = X                              .
4        HALT                               .
5        X = X/10.0                         .
6        JUMP TO 1                          .
7        X = X * 10.0                       .
8        JUMP TO 2                          ..
```

A shorter and probably slower program would be:

```
0        INTEGER N                          .
1        JUMP TO 5 IF 1(8)>X IF X>=1(7)     .
2        X = X / 10.0 IF X >= 1(8)          .
3        X = X 10.0 IF 1(7) > X             .
4        JUMP TO 1                          .
5        N = X                              .
6        HALT                               ..
```

Example Sub-Program 3. Find the maximum of a set of N numbers; the numbers are stored as elements of the array BETA; and BETA(J) is the first variable of the group. Assume N > 1, N + J < 100.

```
0        INTEGER N J I K EXIT                .
0        ARRAY BETA(100)                     .
1        MAX = BETA(J)                       .
2        I = J                               .
3        N = J + N - 1                       .
4        8, K, J + 1, 1, N,                  .
5        JUMP TO 8 IF MAX >= BETA (K)        .
6        MAX = BETA(K)                       .
7        I = K                               .
8        BYPASS                              .
9        JUMP TO EXIT                        .
```

MAX is the desired maximum and I is the subscript of the maximum BETA of the set. Statement 8 was necessary to end the iteration statement properly. Note statement 9 at the end - a variable JUMP statement- enabling the main program to use this subprogram several times and to exit to different statements for continuation.

Example Program 4. Generalized matrix multiplication of up to a 20 x 20 matrix. The I x J matrix ALPHA will multiply the J x K matrix RHO and the result will be placed in the I x K matrix GAMMA. Then the resulting matrix GAMMA will be punched all at once.

```
0        READ                                .
0        ARRAY ALPHA(I,J) 400                .
0        ARRAY RHO(J,K) 400                  .
0        ARRAY GAMMA(I,K) 400                .
1        GAMMA = ALPHA * RHO                 .
2        PUNCH GAMMA                         .
3        HALT                                ..
```

The way this program would actually look when punched on cards and with the Header Card and Comments Card inserted is as follows:

```
         +10       +20       +30       +40       +50       +60       +70       +80
00000000000000000000000000000000000000300000000000000000365000000000000000000000
```

```
                                  1 EXAMPLE4 MATRIX MULTIPLICATION
0000R                               READ                          .
0000R                               ARRAY ALPHA(I,J) 400          .
0000R                               ARRAY RHO(J,K) 400            .
0000R                               ARRAY GAMMA(I,K) 400          .
0001R                               GAMMA = ALPHA * RHO           .
0002R                               PUNCH GAMMA                   .
0003R                               HALT                          ..
```

The resulting Runcible program after proper order has been restored by moving the new Header card into the first position is listed below along with the five per card machine language turned out by Runcible. Note that package P2Y is employed.

RESULTING RUNCIBLE PROGRAM FOR EXAMPLE 4.

```
+0000000009  +0000001200  +0000000000  +0000000006  +0000000000  +0000000365  +0000000000  +0000000000
0000  1        EXAMPLE4 MATRIX MULTIPLICATION
0001    0000I  READ                          F
0002    0000I  I 3ZSOMI 1                    F
0003    0000I  I 5ZI3SI 1SLI 1X1 2RMI 4      F
0004    0000I  I 6ZI5SI 4SLI 4XI 1RMI 4      F
0005    0001I  4KI 7K1K1KI 2K                F
0006    0005I  4KI 9K1K1KI 4K                F
0007    0006I  YLI 6SLI 4XI 7RSI 9RZOJ       F
0008    0000I  4KI 8K1K1KI 4K                F
0009    0004I  YLI 6SLI 4XI 7RSI 9RZYLI 6SLI
0010    0004I  4XI 7RSI 9RSYLI3SLI 1XI 7RSI 8
0011    0004I  RXYLI5SLI 4XI 8RSI 9R         F
0012    0002I  PUNCHYLI 6SLI 4X1RS1RTHRUYLI 6
0013    0002I  SLI 4XI 4RSI 2R               F
0014    0003I  HALTF                         F
```

CORRESPONDENCE TABLE FOR EXAMPLE 4

0001	1800I	J	I0001			
0002	1801I	I	I0002			
0003	1802I	A 1	I0003			
0004	1803I	K	I0004			
0005	1804I	A 2	I0005			
0006	1805I	A 3	I0006			
0007	1806I	D 1	I0007			
0008	1807I	D 2	I0008			
0009	1808I	D 3	I0009			
0010	1850I	ALPHA	Y0001	I 3	I 2	I 1
0011	1851I	RHO	Y0002	I 5	I 1	I 4
0012	1852I	GAMMA	Y0003	I 6	I 2	I 4

MACHINE LANGUAGE PROGRAM FOR EXAMPLE 4 (*)

0001	0001	+0000000019	0002	+0000000029	0003	+0000001230	0004	+0000001231	0005	+0000001238
0002	0000	+0000001246	1999	+6915341790	1534	+6600201533	1533	+1512461532	1532	+2000221531
0003	1531	+6000211530	1530	+1900201529	1529	+1600231528	1528	+1500201527	1527	+1500221526
0004	1526	+2000241525	1525	+6000201524	1524	+1900231523	1523	+1600231522	1522	+1500231521
0005	1521	+1500241520	1520	+2000251232	1232	+8012321891	1239	+6912471519	1519	+2400261236
0006	1236	+8012361891	1243	+6912471518	1518	+2400281237	1237	+8012371891	1244	+6000261517
0007	1517	+1900231516	1516	+1500281515	1515	+1500251514	1514	+8080021513	1513	+6912461512
0008	1512	+2420291511	1511	+6912471510	1510	+2400271235	1235	+8012351891	1242	+6000271509
0009	1509	+1900231508	1508	+1500281507	1507	+1500241506	1506	+8080021505	1505	+6920291504
0010	1504	+2418771503	1503	+6000261502	1502	+1900201501	1501	+1500271500	1500	+1500221461
0011	1461	+8080021472	1472	+6020291483	1483	+3918771494	1494	+2118771455	1455	+6000261466
0012	1466	+1900231477	1477	+1500281488	1488	+1500251499	1499	+8080021460	1460	+6020291471
0013	1471	+3218771482	1482	+2118771493	1493	+6000261454	1454	+1900231465	1465	+1500281476
0014	1476	+1500251487	1487	+8080021498	1498	+6918771459	1459	+2420291470	1470	+6500271481
0015	1481	+1512471492	1492	+2000271453	1453	+6600271464	1464	+1500231475	1475	+4614861235
0016	1486	+6500281497	1497	+1512471458	1458	+2000281469	1469	+6600281480	1480	+1500231491
0017	1491	+4614521237	1452	+6500261463	1463	+1512471474	1474	+2000261485	1485	+6600261496
0018	1496	+1500211457	1457	+4612331236	1233	+8012331891	1240	+6000231468	1468	+1900231479
0019	1479	+1500211490	1490	+1500251451	1451	+1512481462	1462	+2018501473	1473	+6012471484
0020	1484	+1900231495	1495	+1512471456	1456	+1500251467	1467	+1512481478	1478	+2018511489
0021	1489	+6512491450	1450	+6912341840	1234	+8012341891	1241	+0180038778	1249	+0000000002
0022	1248	+0000020000	1247	+0000000001	1246	+0000000000	1246	+0000000000	1246	+0000000000

(*) To be used with P2Y package.

A P P E N D I X V.

FAR

This program is a modification of the Fortruncible and Runcible programs to automatically compile from the Fortruncible input to the machine language running program without card output. All intermediate results, as well as the compiler programs themselves, are stored in the RAM 355 file. As a consequence of this automatic operation, the following equipment is necessary in order to use this program:

1) Alphabetic Device with Special Characters, group II
2) Table Look-up on Equal (Code 63)
3) Indexing Registers
4) Immediate Access Storage (Core Storage)
5) Floating Decimal Hardware
6) 355 RAMAC File

These programs are stored in RAM addresses 4000 to 4200 and use RAM addresses 0000 to 1100 for temporary storage. In addition, parts of this program are designed to operate with the CASE Omnibus - Librarian program which uses RAM addresses 9950 through 9999 and the special KLIK card, and the PIP program which occupies RAM addresses 4200 to 4400. A card image of these programs is available for loading the RAM file. All the rules for writing programs in Fortruncible and Runcible will be assumed to be known and the following information is a brief summary of the changes in the rules for operating and coding with FAR. A special card (FAR) is used to start the program.

This program is an automatic Fortruncible to machine language translator in one pass via a special Runcible program and PIP. The form of the program is the same as the regular Fortruncible but no intermediate cards are punched except for errors detected during compilation. The program starts out in standard compiling mode until an error is detected.

At this point the program shifts into error sense mode until the end of the phase is reached. This terminates the program. Some special features have been added to Fortruncible to facilitate the operation. This program also requires a 650 with indexing registers, cores, ram file, and special characters with a table look-up on equal. The automatic floating point attachment is required only if compiling in Y mode.

I. HEADER CARD.

The use of a header card is optional. If a header card is supplied then the number of I, Y, and C variables specified will be assumed to be used by passed Runcible statements and any variables assigned by Fortruncible will have larger subscripts than those specified by the header card. Actually Fortruncible will start assigning new variables in the order in which they occur in the program starting with the next larger subscript.

The maximum statement number designated by the header card will be checked against the actual program and if it is too small it will be corrected; however if the number specified is too large it will be permitted to remain.

If the number of locations in the P package is specified then it will be assumed to be correct; otherwise, if the number of locations is zero, then the translator will assign the correct number depending upon the operations used by the program. If any named extension is spelled incorrectly or the function is not contained in any of the P packages then an error will be indicated.

If no header card is supplied, then the translator will supply all of the above information automatically; however all ARRAY statements must specify the number of locations required in storage.

II. SPECIAL STATEMENTS.

In order to facilitate the punching of the data cards before compiling the program and to reduce the need for a correspondence table, the following statement type has been added to Fortruncible language:

FLOATING $NAME_1$ $NAME_2$ $NAME_3$ $NAME_4$ ... $NAME_n$.

where $NAME_1$ through $NAME_n$ are unscripted floating point name variables which are presently undefined. Then this statement will define these variables in the order named assigning them to sequential Runcible C variables starting with the next subscript in ascending order. This statement will not produce any output coding in machine language. Another use for this statement is to arrange certain variables in sequential order for use by a PUNCH $NAME_1$ THRU $NAME_n$ statement. Any name beginning with the letters FLOAT can not be used as the name of a variable.

As an example consider:

```
OOOOR                    FLOATING HOUSE DOG CAT
```

If no header card has been used or the number of C variables specified by the header card is zero and if this is the first statement in the program, then HOUSE will be assigned to C1, DOG to C2, and CAT to C3.

A second type of statement that has been added to the language is a matrix partition or redefinition statement. This statement has the form:

ARRAY B(I,J) = A(K,L)

where array A has already been defined. Then array B of size I x J

will be defined to start with B(1,1) equal to the element A(K,L) in array A. The array B should not have been previously defined. Also the partition should make sense, but the arrays may be vectors as well as matrices. A column vector may be defined as part of a matrix by the statement:

CVECTOR C(I) = A(K,L)

which defines the column vector C of length I to be equivalent to the column L of matrix A with C(1) equal to A(K,L), i.e. starting with element K in column L.

Consider the examples:

```
0006R                ARRAY CAR(10,10) = HOUSE (6,6)      .
0023R                CVECTOR COLUMN(10) = HOUSE (1,3)    .
```

In the first example, matrix CAR is defined to be of size 10 by 10 with CAR(1,1) equal to the element HOUSE(6,6) in the array HOUSE. In the second example, the column vector COLUMN is defined to be the third column in the array HOUSE with COLUMN(1) equal to the element HOUSE(1,3) in the array HOUSE.

All integers should be defined by means of the INTEGER statement:

INTEGER $NAME_1$ $NAME_2$ ... $NAME_n$

where $NAME_1$ through $NAME_n$ will be defined as fixed point integers.

III. PASSING RUNCIBLE STATEMENTS.

Any number of Runcible statements may be passed through Fortruncible by punching them in the standard Runcible format EXCEPT

that each statement should be ended with a period (.) in column 72 of the card instead of an F in column 70. Also a header card should be included to allow space for the Runcible variables used by the passed Runcible statements, if any.

IV. ENDING THE FORTRUNCIBLE PROGRAM.

The Fortruncible program MUST be ended by a double period on the last card. These periods should be punched in columns 71 and 72 in the last card of the last statement of the program.

V. ERROR CARDS.

Any errors in the formation of the statements detected during the running of the Fortruncible part of the translator will be punched on cards which will display an error number along with a portion of the statement being translated. However, the operator should be cautioned that only a few types of errors can be detected in the Fortruncible pass, and all other errors will be indicated by the standard Runcible error cards. The following errors will be detected in Fortruncible:

Type	Meaning
1	Too many named extensions.
2	Operator symbol is not defined.
3	An improperly formed power of ten notation.
4	The first character in the statement is a special character.
5	Unmatched parentheses.
6	Too many floating point named variables (> 48).
7	Too many fixed point named variables (> 48).

8	Too many named arrays (>30).
9	The array statement not properly formed.
10	The array redefinition statement not properly formed.
11	Too many redefined arrays (> 10).
12	Improperly formed statement.
13	Too many different procedures (> 4).
14	Too many variables in a procedure declaration (> 5).
15	An end of procedure without a procedure declaration.
16	The execute statement does not agree with the procedure definition.
17	An improper matrix equation.
18	The I subscript exceeds two digits.
19	The space required by the arrays has not been defined.
20	The translation has resulted in too long a Runcible statement.
21	A named extension has been requested that is not in any of the basic packages.

VI. STOPS.

9901	The Fortruncible statement is too long (> five cards).
9991	No comments card or improper correspondence table.
9902	Console set for multipass operation.
XXXX	Console not set properly.

VII. OPERATING PROCEDURES.

The console should be set for B-mode (One pass mode) as described in the Runcible manual. The translator is loaded from the RAM file by means of a FAR card. If the data address of 8000 is 1951, then arms 0 and 1 will be used by the translator. If it is set to 1952, arms 1 and 2 will be used, and a setting of 1953 will cause arms 2 and 0 to be used. Thus if one of the arms in the RAM is inoperative, the console can be set so that the program will still operate.

The ordering of the cards in the Fortruncible program is as follows:

1. FAR card
2. HEADER card (If present)
3. CORRESPONDENCE TABLE cards (If present)
4. COMMENTS card (Must be present)
5. FORTRUNCIBLE or RUNCIBLE STATEMENTS
6. LAST STATEMENT (periods in cols. 71 and 72)
7. RUNCIBLE DATA cards (If needed)

A. Machine language punchout.

Set the console to +70 1951 7777 read in the KLIK card.

B. Runcible statement punchout.

Set the console to +70 1951 6666 and read in the KLIK card.

C. Correspondence table punchout.

Set the console to +70 1951 5555 and read in the KLIK card. A stop of 0999 indicates that the correspondence table is ready to be punched out.

D. Rerunning the machine language program.

1. PIP card

2. Machine language program from A above
3. BLANK card
4. DATA cards (If necessary)

A P P E N D I X V I .

ADDITIONS TO SOAP III --- RUNCIBLE 533 PLUGBOARD to USE FORTRUNCIBLE.

N.B. All numbering is from LEFT to RIGHT (except control information)

Read Side

Remove: Wire from Co Sel 3 pos 4 to Rd Cd C Col 4
Add : Wire from Co Sel 3 pos 4 to Rd Col Split # 3 common
Add : Wire from Rd Col Split (0 - 9) to Rd Cd C Col 4
Add : Wire from Rd Col Split (11 - 12) to Rd Cd C Col 5

Pch Side

Remove: All wires to Cols 1⟶10 of Pch Card A.
Remove: Wire from Punch Delay # 1 out to Punch Delay # 2 in.
Add : All the following wires:
From Wd 7 Pos 7 Storage Exit A to Cosel 1 Pos 1 Normal
From Wd 7 Pos 8 Storage Exit A to Cosel 1 Pos 2 Normal
From Wd 7 Pos 9 Storage Exit A to Cosel 1 Pos 3 Normal
From Wd 7 Pos 10 Storage Exit A to Cosel 1 Pos 4 Normal
From Punch Col Split # 9 Common to Cosel 1 Pos 5 Normal
From Punch Col Split # 9 (0 - 9) to Emitted 9 timed to punch
From Punch Col Split # 9 (11 - 12) to Emitted 12 timed to punch
From Cosel 1 Pos 1 Common to Cosel 2 Pos 1 Transfer
From Cosel 1 Pos 2 Common to Cosel 2 Pos 2 Transfer
From Cosel 1 Pos 3 Common to Cosel 2 Pos 3 Transfer
From Cosel 1 Pos 4 Common to Cosel 2 Pos 4 Transfer
From Cosel 1 Pos 5 Common to Cosel 2 Pos 5 Transfer

From Cosel 2 Pos 1 Common to Pch Card A Col 1
From Cosel 2 Pos 2 Common to Pch Card A Col 2
From Cosel 2 Pos 3 Common to Pch Card A Col 3
From Cosel 2 Pos 4 Common to Pch Card A Col 4
From Cosel 2 Pos 5 Common to Pch Card A Col 5
From Cosel 2 Pos 1 Normal to Pch Col Split # 1
From Cosel 2 Pos 2 Normal to Emitted Zero Timed to Pch
From Cosel 2 Pos 3 Normal to Emitted Zero Timed to Pch
From Cosel 2 Pos 4 Normal to Emitted Zero Timed to Pch
From Cosel 2 Pos 5 Normal to Emitted Zero Timed to Pch
From Cosel 9 Pos 1 Common to Pch Card A Col 6 and 8 and 9
From Cosel 9 Pos 2 Common to Pch Card A Col 7
From Cosel 9 Pos 3 Common to Pch Card A Col 10
From Cosel 9 Pos 1 Normal to Emitted Zero Timed to Pch
From Cosel 9 Pos 2 Normal to Emitted 8 Timed to Pch
From Cosel 9 Pos 3 Normal to Pch Col. Split # 1 Common
From Control Information* to Cosel Pick 2 and 9
From Control Information* 9 to Cosel Pick 1
From Cosel Hold 1 and 2 and 9 to Pch Hold

*Note: Control Inf Only Is Numbered 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.