Runcible I
Vol. I Series V
Revised Edition
March, 1959
Staff
Computing Center

CASE INSTITUTE OF TECHNOLOGY
UNIVERSITY CIRCLE • CLEVELAND 6, OHIO
Revised Unified New Compiler with
IT-Basic Language Extended

"They dined upon mince and slices
of quince,
Which they ate with a runcible
spoon;
And hand in hand, on the edge of
the sand,
They danced by the light of the
moon,—
The moon;
They danced by the light of the
moon.” —Edward Lear

COMPUTING CENTER
Case Institute of Technology
Cleveland, Ohio
Abstract

In recent years several compilers have arisen for various computers as steps towards automation in programming. A compiler, such as RUNCIBLE I, comes as close as possible to the ideal of transforming a flow chart directly onto cards and into a finished program. Coding for Runcible I is done in "IT-language," originally developed by Dr. A. J. Perlis, Mr. J. W. Smith, and Mr. H. R. Van Zoeren at Carnegie Institute of Technology, Pittsburgh. Although the original IT-language will still work correctly with this compiler, it has been expanded and simplified somewhat for greater convenience.

Runcible I will run on a basic 2000-word 650 which has only an alphabetic attachment as an extra feature, but if desired it will give an output program which utilizes the 653 floating point hardware, index registers, and core storage. Neither the special character attachment or any additional selectors are needed by the 533 plugboard, and the input statements may be easily listed on a 402 tabulator. All routines will work with a 12-word buffer if necessary. This program is designed to work together with the SOAP III assembly program, but the user has the option of skipping the SOAP phase and obtaining a five-instruction-per-card machine language deck directly. RUNCIBLE I also introduces a number of devices which greatly facilitate the debugging of a program which is in its initial stages. Hand optimized subroutines used with the running program compare favorably in speed and accuracy with those of any alternative general purpose system for the 650 known to date.

The major purpose of the compiler is thus to enable a program to be run speedily with but a minimum of preparation and headache on the programmer's part. It is specifically designed to be used with algebraic equations arising in typical mathematical and engineering problems.
Acknowledgments

RUNCIBLE I is a project of such a magnitude that it is impossible to acknowledge everyone who has contributed to the effort. As with Compiler II, the entire blame for the whole operation belongs with Mr. Frederick Way III, Assistant Director of the Computing Center, who with George Haynam made the major decisions on the philosophy to be carried out. Bill Lynch did the greater part of the work of converting the compiler to floating point instructions and expanding the language, and he worked hand in hand with the author on producing the final deck in its multipurpose form; he has also done some of the work on extensions. George Haynam is responsible for the clocking feature and most of the coding in the basic packages, and he was also in charge of board wiring. He and George Petznick, Jr. have prepared a large number of extensions which produce extremely accurate results. Don Buyansky drew up the 533 board wiring diagram, and almost every other member of the staff at the computing center has had his hand in at one time or another. Donald Knuth, author of this manual, added the one pass feature and wrote the relocation routine; he also designed the 24-operation-mode program and deck, helped to hand optimize the basic packages and prepared the comments on the symbolic program listings. Basic packages were optimized with HAND SOAP, a modification of SOAP III.

NOTE: Any program which is written correctly in original IT-language or for the SOAP II Compiler (Case Institute, November 1957) will be processed correctly by this routine without any changes except for slight substitutions in the header card and the data cards. No major revisions of RUNCIBLE I are anticipated for several years; now in preparation is "Runcible O" which will process directly from statements to answers without any intermediate punching of a machine language program.
INTRODUCTION. The user of RUNCIBLE I need not be familiar with any features of the 650 other than those to be described in this manual. A compiler such as this program is probably the simplest and most convenient way to solve problems on a computer without actually studying the complexities of the machine. Those who state problems should not be required to code them; RUNCIBLE I takes the place of a professional programmer and does the job itself.

THE LANGUAGE. There are many types of languages which enable communication between people. Besides spoken tongues, there are the languages of music, of mathematics, and so on. RUNCIBLE I understands "IT-language," which is very close to both English and mathematical formulae. The program which will solve a problem is put onto cards in "IT-language," the compiler will transform these into an intermediate "SOAP language" and punch it onto cards in this form. These cards in turn are processed by the SOAP III program to produce the "machine language" which the computer understands directly. The compiler programmer need not know any more of SOAP language than is given in this manual. The first section of the manual deals with the vocabulary and syntax of IT-language.

NUMBERS: FIXED POINT AND FLOATING POINT. An IT-language program is a set of numbers, words, and symbols arranged to give a complete description of some problem. The numbers which are manipulated by RUNCIBLE I are of two categories: fixed point and floating point.

Fixed point numbers are integers whose numeric value is less than one billion. They may be positive or negative, but may never take on fractional values. They are used primarily as indices or subscripts and only rarely for arithmetic calculations.

Floating point numbers are normally used for arithmetical operations because of their greater flexibility. Floating point numbers can be zero or range from $10^{-50}$ to $10^{49}$ in numerical value, and are always rounded to eight significant figures.

1. If desired, the SOAP phase may be bypassed; see Operation Modes.
2. When using 653 instructions (see Operation Modes) they range from $10^{-51}$ to $10^{48}$. 

-3-
VARIABLES. Variables normally called $x$, $y$, $z$, $p$, etc. are always given the names $I$, $Y$, or $C$ with subscripts when using RUNCIBLE I; e.g., $I_1$, $Y_1$, $Y_2$, $C_{10}$, and so on. (The subscript is to be written next to the letter as $I_1$, $Y_2$, $C_{10}$.) Every variable must have a subscript.

$I$- variables are indexing variables and are always given fixed-point (integral) values. $Y$- and $C$- variables are the problem variables and always take on floating-point values. There is no difference in arithmetic between a $Y$- variable and a $C$- variable; any distinction is made by the programmer himself as an aid to keeping the names of his variables in order.

When working on the problem the machine keeps a table similar to the table below. If, for example, the program uses variable $Y_3$, the machine looks into the table and sees that the floating point number $-1.55$ is to be used in the calculation. It is possible to have variable subscripts, $Y_{I_1}$, $C_{I_4}$, etc. $Y_{I_1}$ (written $YII$) will be the variable $Y_2$ in the example of the table below, since $II$ equals 2. In a similar manner, $C_{I_4} = C_1 = 1000$.

Subscripts must be fixed point numbers or fixed point variables.

<table>
<thead>
<tr>
<th>I</th>
<th>Y</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+5</td>
<td>$+3.0000000$</td>
</tr>
<tr>
<td>1</td>
<td>+2</td>
<td>$-10.000000$</td>
</tr>
<tr>
<td>2</td>
<td>+1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>+12</td>
<td>$-1.5500000$</td>
</tr>
<tr>
<td>4</td>
<td>+1</td>
<td>$+6155397.0$</td>
</tr>
<tr>
<td>5</td>
<td>-1040</td>
<td>$+4.1400000 \times 10^{-34}$</td>
</tr>
<tr>
<td>6</td>
<td>+704</td>
<td></td>
</tr>
</tbody>
</table>

1. $YII$ means $Y_{I_1}$ and in this case $YII = YI_2 = Y1 = -10$. This hierarchy of subscripts may proceed to any depth. Compound subscripts of the form $Y(I_2+I_3)$ are also allowable; other examples of permissible subscripts are $I(I_3xI_9)$ and $C[I(I_1+2)]$, and even $Y"SINEF,C3"$. An alternate form for subscripts used in matrices is described in Appendix II. Note that all legal subscripts must be fixed point.
CONSTANTS. Constants used in a RUNCIBLE program are written as follows:
Floating-point constants are written as some number by a power of ten, with
a "B" to indicate what power of ten. Examples:

- 6.02 B 23 means 6.0200000 x 10^{23}
- 1066 B -11 means 1066.0000 x 10^{-11}

The B may be omitted as long as the number can be represented in eight digits
without multiplying by a power of ten; for example,

- 3.1415927; .15625; 0.

When the B is omitted the number must contain a decimal point.

Fixed-point constants are written in the ordinary manner; "123" for example.
Notice that "123." is a fixed point constant while "123." is floating point
because of the decimal point.

Any number of constants may be used in a program.

MATHMATICAL OPERATIONS. Mathematical expressions can be converted almost
at sight directly into IT-language, but it is best to describe this process
carefully so there will be no chance of a mistake.

Five binary operations are standard for RUNCIBLE expressions: addition,
subtraction, multiplication, division, and raising to a power; they are
called binary because they involve two quantities. These operations are
written in the following manner (letting Cl represent x and Y1 stand for y):

<table>
<thead>
<tr>
<th>Math. language</th>
<th>IT-language</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>Cl + Y1</td>
</tr>
<tr>
<td>x - y</td>
<td>Cl - Y1</td>
</tr>
<tr>
<td>xy</td>
<td>Cl x Y1</td>
</tr>
<tr>
<td>(\frac{x}{y})</td>
<td>Cl / Y1</td>
</tr>
<tr>
<td>(x^y) (x to y power)</td>
<td>Cl P Y1</td>
</tr>
</tbody>
</table>

Note how the IT-language scheme enables the writing of a formula as a string
of symbols all on one line. Putting the operations together, with a few
constants, we have:

Math. language: \(\frac{[(10.4x10^{28}) + x^4]y}{x - y}\)

IT-language: \([10.4828 + (Cl P 4)]x Y1^2 / (Cl - Y1)\)

1. A floating point exponent (such as 5 B 3.14) is not permitted.
2. Actually 700 is the maximum allowable number but this may safely be
   considered "infinite" for programs processed by RUNCIBLE. If more than
   100 constants are used the last 100 are punched out and the program then
   continues where it left off.
Notice the use of parentheses in the last example.

When more than two quantities are involved, parentheses are needed to avoid ambiguity. Parentheses are very important in IT-language because there is no difference in priority or scope between any of the binary operations as is usually understood in everyday mathematics. For example, the expression \( Y_1 \times Y_2 + C_3 \) would mean \( Y_1 Y_2 + C_3 \) to most people but RUNCIBLE understands it to mean \( Y_1(Y_2 + C_3) \). As another instance, in order to write \( x^4y \) we would have to write \( (C_1 P 4) \times Y_1 \) because \( C_1 P 4 \times Y_1 \) would mean \( x^{4y} \). There is a very simple MORAL to be learned from this: always place parentheses around the quantities in an IT language expression until it can mean only one thing. This cannot be stressed too heavily, for the vast majority of programming errors for RUNCIBLE are caused by neglecting to place parentheses in the proper way.

As a reference, these are the rules by which RUNCIBLE determines to which quantities each binary operation applies.

1. On the left-hand side of the 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.

2. On the right-hand side of the symbol, everything up to the end of the expression or to the first unmatched right parenthesis is used.

Here are some examples showing application of these rules.

<table>
<thead>
<tr>
<th>IT-language:</th>
<th>Math. language:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y_1 / Y_2 + Y_3 )</td>
<td>( Y_1 ) ( Y_2 + Y_3 )</td>
</tr>
<tr>
<td>( (Y_1/Y_2) + Y_3 )</td>
<td>( Y_1 ) ( Y_2 + Y_3 )</td>
</tr>
<tr>
<td>( Y_1 P Y_2 + 3x Y_3 )</td>
<td>( Y_1 Y_2 + 3Y_3 )</td>
</tr>
<tr>
<td>( (Y_1 P Y_2)+3xY_3 )</td>
<td>( Y_1 Y_2 + 3Y_3 )</td>
</tr>
<tr>
<td>( (Y_1 P Y_2+3)X Y_3 )</td>
<td>( (Y_1 Y_2 + 3)Y_3 )</td>
</tr>
<tr>
<td>( [(Y_1 P Y_2)+3]X Y_3 )</td>
<td>( (Y_1 Y_2 + 3)Y_3 )</td>
</tr>
<tr>
<td>( 4 / Y(11+12)-I_3 )</td>
<td>( 4 ) ( (I_1+I_2)^{-1} )</td>
</tr>
</tbody>
</table>

1. Parentheses, braces, and brackets may be nested within each other not more than nine deep but this limit is rarely met in practice.
2. Exception: the minus (-) is treated first as a unary operator (see below) and then as a plus (+) binary operator.
Remember, it is always best to add parentheses to make your intentions unquestionable—better safe than sorry. Adding parentheses where they are unnecessary will not bother RUNCIBLE and no harm will be done.

In addition to the five binary operations there are two standard "unary" operations: taking the absolute value and taking the negative of a quantity. Letting $Cl \leftrightarrow x$ and $Ml \leftrightarrow y$ as before, we have

Math. language:  
$$|x|$$  
$$|x+y|$$  
$$-x$$  
$$-(x+y)$$  
$$-|y|$$

IT-language:  
$$A Cl$$  
$$A( Cl + Ml )$$  
$$- Cl$$  
$$-( Cl + Ml )$$  
$$- A Ml$$

The unary operator $A$ or $-$ applies to the variable or constant at the immediate right of the symbol only, 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:

IT-language:  
$$AYl + Cl / C2$$  
$$A(Yl + Cl) / C2$$  
$$AYl P -(ACl-Y2)$$  
$$AYl(3+1) + 2$$

Math. language:  
$$|Yl| + \frac{Cl}{C2}$$  
$$\frac{|Yl + Cl|}{C2}$$  
$$|Yl| - (|Cl| - Y2)$$  
$$|Y_{3+1}| + 2$$

Arithmetic: Runcible always "unwinds" statements by doing the innermost parentheses first, just as in ordinary high school algebra. Once RUNCIBLE is into the innermost parenthesis it drops the "ordinary" rules and does

1. It is possible to intermingle fixed and floating constants and variables—RUNCIBLE will not get mixed up; however this is generally inefficient and of doubtful utility.
things from the RIGHT!

Examples:

\[\begin{align*}
Y_1 & \leftarrow 5 \times 4 + 6 \quad ; \quad Y_1 \text{ is given the value } 30.0 \\
Y_1 & \leftarrow 6 + 3 \times 4 \quad ; \quad Y_1 \text{ is given the value } 18.0 \\
Y_1 & \leftarrow 6 + (3 \times 4) \quad ; \quad Y_1 \text{ is given the value } 18.0 \\
Y_1 & \leftarrow (3 \times 4) + 6 \quad ; \quad Y_1 \text{ is given the value } 18.0 \\
Y_1 & \leftarrow 6 - 3 \times 4 + 2 \quad ; \quad Y_1 \text{ is given the value } -12.0 \\
Y_1 & \leftarrow 6 - (3 \times 4) + 2 \quad ; \quad Y_1 \text{ is given the value } -8.0 \\
Y_1 & \leftarrow 6 / 4 / 2 \quad ; \quad Y_1 \text{ is given the value } 3.0 \\
Y_1 & \leftarrow (6 / 4) / 2 \quad ; \quad Y_1 \text{ is given the value } 0.75 \\
Y_1 & \leftarrow (6.0 / 4.0) / 2.0 \quad ; \quad Y_1 \text{ is given the value } 0.75 \\
Y_1 & \leftarrow (6 / 4) / 2.0 \quad ; \quad Y_1 \text{ is given the value } 0.5
\end{align*}\]

From the above examples we can see that RUNCIBLE 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.

Examples:

\[\begin{align*}
I_2 & \leftarrow 6.0 + 3 - 7 \times 2.0 \quad ; \quad I_2 \text{ is given the value } -5 \\
I_2 & \leftarrow 6.0 + 3 - 7 \times 2 \quad ; \quad I_2 \text{ is given the value } -5 \\
Y_2 & \leftarrow 6.0 + 3 - 7 \times 2 \quad ; \quad Y_2 \text{ is given the value } -5.0 \\
C_6 & \leftarrow 6.0 + 3.0 - (1 / 4) \quad ; \quad C_6 \text{ is given the value } 9.0 \\
I_6 & \leftarrow 6.0 - 3.0 - (1 / 4) \quad ; \quad I_6 \text{ is given the value } 9 \\
C_6 & \leftarrow 3.0 - (1 / 4) + 6.0 \quad ; \quad C_6 \text{ is given the value } 9.0 \\
C_6 & \leftarrow (9 / 10) + 1.0 - (7 / 16) \quad ; \quad C_6 \text{ is given the value } 1.0 \\
I_6 & \leftarrow (9 / 10) + 1.0 - (7 / 16) \quad ; \quad I_6 \text{ is given the value } 1
\end{align*}\]
In summary then, if you "mix" arithmetic by mixing fixed and floating constants or variables, the rule is that RUNGIBLE 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, 9/9 = 0 in fixed point division)!^2

2. The remainder of a fixed point division is usually dropped but it can be saved if desired as described in Appendix III.
Extensions: In addition to these basic operations, a wide variety of "extensions" can be added to the compiler, making it extremely versatile. Rules for their use are given in the writeup supplied with each individual extension which might be in the subroutine library at your installation, but a few general rules and examples will be illustrated here for clarity.

Reference to an extension is made with quotation marks followed by the name of the extension and a comma; these are followed by one or more inputs and closing quotation marks. (The quotation marks are treated just as parentheses by Runcible except that they identify extensions.) The name of each extension will end with either an "E" or "F," the former signifying floating point output and the latter signifying a fixed point result.

Three commonly used extensions are illustrated below, letting Cl stand for x:

Math. language:
\[ \sqrt{x} \]
\[ \sin x \]
\[ e^x \]

IT-language:
"RT2 E, Cl"
"SINE, Cl"
"EXP E, Cl"

As a final example for this section on operations, here is the IT-language representation of the "Wolontis function"

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

We have \[ f(x) = "SINE, Cl" / "RT2 E, 1 + EXP E, (-Cl) P 3.0" " \]

STATEMENTS. The numbers, variables, and operators we have now learned are put together into meaningful statements as instructions to Runcible. Each statement is given a number which is some integer less than 1000. The order in which 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. Statement number zero is special -- this number is reserved for statements which are not going to be referred to in the program, and as many statements can be numbered zero as desired. Each nonzero statement

1. When a subroutine is identified by number it is floating point if the number is below 500.
number, however, must never appear on more than one statement in a program — it must be unique.

Substitution statements. There are many types of statements; perhaps the most frequently used is a substitution statement. In this statement a variable is given the value of any mathematical expression; a new value is "substituted" for its former one. The substitution operation is denoted by a backwards arrow (\(\leftarrow\)). Examples:

- \(Y_1 \leftarrow 0\)
- \(C_{10} \leftarrow Y_1 / (Y_2 \times Y_2)\)
- \(I_3 \leftarrow I_3 + 1\)
- \(Y_{14} \leftarrow I_9 - 3\)

In the first case \(Y_1\) is set to zero. The second example sets \(C_{10}\) equal to \(Y_1\) divided by \(Y_2\) squared. In the third illustration, \(I_3\) becomes equal to one greater than its former value. The last case takes three less than the value of \(I_9\) and gives it to the \(Y\) variable which has subscript equal to the current value of \(I_4\). Note that in the last example RUNGIBLE will convert the fixed point right-hand side automatically into a floating point number before inserting it into \(Y_{14}\). 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).

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

\[\text{JUMP TO } k\]

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 it is fixed point; e.g., \(\text{JUMP TO } I_2\) or \(\text{JUMP TO } (I_2 + 3 \times I_1)\). \(k\) must never be zero or negative.

READ statements. A READ statement is written

\[\text{READ}\]

This statement will cause the machine to read in one or more cards of data for the problem. All data for a program other than constants enter
the 650 via a READ statement. The form of data cards is described later in the section on formats.

Output statements. Answers are punched onto cards when an output statement is given. There are two different kinds of PUNCH statements:

1. PUNCH statement type one will put the current values of up to four variables onto one card. The statement

   PUNCH Y3

will cause the current value of variable Y3 to be imprinted on a card;

   PUNCH C4 PUNCH I6 PUNCH Y9

will put the values of C4, I6, and Y9 all on the same card. Up to four variables can be punched at a time in this manner.

   PUNCH Y(I4+1)

is also allowable—it will type the value of the variable specified by (I4+1). It is not legal, however, to give a statement like PUNCH (Y1+Y2) or PUNCH AY7.

2. If a large number of consecutive variables are to be punched, a statement such as

   PUNCH Y1 THRU Y15

may be given. When the word "THRU" is used like this, up to seven answers will be put onto each card. The statement above, for example, will put Y1-Y7 on the first card, Y8-Y14 on the second card, and Y15 on a third card.

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 may be used as data to another program.

When an output statement has statement number zero it will be bypassed when the console switch of the 650 is set to plus during the running phase; it will be obeyed only when this switch is minus. This is a handy device for obtaining intermediate answers when checking a program out in its first few trial runs.

HALT statement. A HALT statement will stop the 650 if the programmed switch on the console is set to STOP. It is written simply

   HALT

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 8005.
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.

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

**BYPASS**

does absolutely nothing; it is simply bypassed during the running phase. (Believe it or not, this statement can be useful!)

**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.

**Conditions.** Any of the above statements may be made conditional and executed only if a certain relation holds true. There are three allowable relations: =, >, ≥ (equals, greater than, and greater than or equal). The condition is preceded by the word IF, thus:

\[
\text{IF } Y_9 = C_9 \\
\text{IF } I_4 \geq 2 \\
\text{IF } A(C_8-Y_8) > 1 B -8
\]

In the last example the relation is satisfied if the magnitude of \( C_8 - Y_8 \) is greater than \( 10^{-8} \). Here are some examples of statements made conditional:

1. JUMP TO 1 IF \( Y_3 = 0 \).
2. READ IF \( 1 \geq C_{10} \)
3. HALT IF \( Y(I_5+2) = Y(I_5+1) \)
4. PUNCH C2 IF \( -(I_5/5) \geq 4 \times I_5 \)
5. \( C_4 \leftarrow Y_4/10 \). IF \( C_3 \times C_3 > Y_4 \times 8.4 \)
6. JUMP TO I3 IF \( I_1 = C_1 \)
7. \( Y_5 \leftarrow 3.1415927 \) IF "SINE, \( Y_5" = 0 \). IF \( Y_5 > 1.571 \)

The lefthand 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. Note that fixed and floating point arithmetic may be mixed as in example 6. The seventh case is interesting because the substitution

\[1. \text{Examples of extension statements: STATISTICAL READ 1,2; READ PROGRAM; EDIT C1 THRU C2, 1231231234, 1112223334; CONVERT Y1 THRU Y50.}\]
Iteration statement. The last type of statement to be discussed in this section is a handy programmer's convenience for a sequence of operations which occurs quite often in typical problems. An "iteration statement" causes a number of other statements to be repeated over and over as a variable is changed by a specified amount. 1

Iteration statements look something like this:

\[ n, v_1, v_2, v_3, v_4, \]

\( v_1 \) is the variable (I, Y, or C) which is to be changed; \( v_2 \) is its starting value and \( v_4 \) is its finishing value; and \( v_3 \) is the amount by which the variable is to be changed before repeating the sequence of statements again. 2

The \( n \) in this statement is a certain statement number: All statements after the iteration statement up to and including statement \( n \) will be repeated for every value of the variable. (\( n \) must not be zero, nor should the statement immediately following the iteration statement have number zero.) Please note the comma after \( v_4 \) — it must be included!

For example, the statement

\[ 5, Y_1, 1, 2, 13, \]

means: execute all statements from the next one to statement 5 for \( Y_1 \) 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 the details will be handled automatically by RUNCIBLE; this "nesting" of iteration statements may proceed to four deep. 3

Iteration statements cannot be made conditional.

\( v_2, v_3, \) and \( v_4 \) need not be constants; they may be variables or even mathematical expressions. For example, the starting value might be something like \( Y_4/2 \); and the finishing value, \( C8P 13 \). A lot of liberties are allowable here, the only restrictions being:

1. No more than five characters (excluding spaces) may appear between any two adjacent commas in an iteration statement. Thus the statement

\[ 2, Y_1, (Y_2 \times Y_3), -6, 3.1415927, \]

is illegal on two counts: \( v_2 \) has seven characters — this may be cured by dropping the parentheses; \( v_4 \) has nine characters — the remedy is to let some

1. This process can also be programmed, of course, without using an iteration statement.
2. If \( v_4 - v_2 \) is not exactly divisible by \( v_3 \) the iteration procedure will be discontinued just before the value \( v_4 \) is passed.
3. 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.
### Example Problem:

**How can we reduce the cost?**

<table>
<thead>
<tr>
<th>Safe</th>
<th>Blew</th>
<th>No. of Stories</th>
<th>Contd</th>
<th>No. of Rooms</th>
<th>Area 1</th>
<th>Area 2</th>
<th>Area 3</th>
<th>Area 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C1</td>
<td>C1</td>
<td>C3</td>
<td>Cu</td>
<td>Y1</td>
<td>Y1</td>
<td>Y1</td>
<td>Y1</td>
</tr>
<tr>
<td>2</td>
<td>C6</td>
<td>C6</td>
<td>C7</td>
<td>Cu</td>
<td>Y3</td>
<td>Y4</td>
<td>Y4</td>
<td>Y4</td>
</tr>
<tr>
<td>3</td>
<td>C7</td>
<td>C10</td>
<td>C9</td>
<td>C12</td>
<td>Y3</td>
<td>Y4</td>
<td>Y4</td>
<td>Y4</td>
</tr>
<tr>
<td>4</td>
<td>C12</td>
<td>C4</td>
<td>C16</td>
<td>C16</td>
<td>Y1</td>
<td>Y1</td>
<td>Y1</td>
<td>Y1</td>
</tr>
</tbody>
</table>

**Answers:**

<table>
<thead>
<tr>
<th>Area 1</th>
<th>Area 2</th>
<th>Area 3</th>
<th>Area 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C11</td>
<td>C18</td>
<td>C19</td>
<td>C20</td>
</tr>
<tr>
<td>C21</td>
<td>C23</td>
<td>C26</td>
<td>C29</td>
</tr>
<tr>
<td>C25</td>
<td>C26</td>
<td>C27</td>
<td>C29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k = 1</th>
</tr>
</thead>
</table>

i.e., \( C_{34} = A_{31}b_{14} + A_{32}b_{24} + A_{33}b_{34} + A_{34}b_{44} \)

(3 iterations)

### Notes:
- Use these data to improve the design.
- Use these data to improve the design.
- Use these data to improve the design.
- Use these data to improve the design.

\[ A \cdot B = C \]
\[ C_{ij} = \sum \text{all } b_{kj} \]

- Use these data to improve the design.
variable, say Y4, be set equal to 3.1415927 first and then use Y4 in the iteration statement.

2. If v3 is negative it must start with the character -; if it is positive it must not start with a minus. The example above illustrates proper use of this rule; if v3 had been written (-6.) erroneous operation would have resulted.

3. n must be a positive fixed point constant and v1 must be a variable.

EXAMPLE PROBLEMS. Two sample problems will be given here to demonstrate some portions of IT-language as it has been described; more examples may be found in Appendix VI.

Example 1. Calculate n! where n is a non-negative integer.

Solution -- since is to be an integer we shall let II represent n in this case for input data to the problem; n! (the answer) will be floating point and we will call it Yl. We will do successive multiplications by a variable Cl which will run through the integers up to n. A flow chart to solve the problem would look something like this:

1. If v3 is zero the programmer should visit a psychiatrist.
Raw index $i$
Column index $j$

Subtract 1 from each subscript

$$\left[ (i-1) \times 4 \right] + g$$

$$C \left\{ \left[ 16 + \left[ (i-1) \times 4 \right] + g \right\} \text{ given subscript} \right.$$ 

$$C^\prime (16 + g + 4 \times (i-1))$$

Subscript column index $j$
Width $g$
New index $j$

Width $g$
Matrix depth $1-1$
Translation of the flow chart into a series of statements is almost automatic:

<table>
<thead>
<tr>
<th>Number</th>
<th>Statement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>READ</td>
<td>(Read in II)</td>
</tr>
<tr>
<td>2</td>
<td>HALT IF 0 &gt; II</td>
<td>(Error stop if n negative)</td>
</tr>
<tr>
<td>3</td>
<td>Yl ← 1.</td>
<td>(Initialize Yl)</td>
</tr>
<tr>
<td>4</td>
<td>JUMP TO 9 IF 1 ≥ II</td>
<td>(0! = 1! = 1.)</td>
</tr>
<tr>
<td>5</td>
<td>Cl ← 2.</td>
<td>(Initialize Cl)</td>
</tr>
<tr>
<td>6</td>
<td>Yl ← Yl x Cl</td>
<td>(Multiply successively until Cl = n)</td>
</tr>
<tr>
<td>7</td>
<td>Cl ← Cl + 1.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>JUMP TO 6 IF II ≥ Cl</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>PUNCH II PUNCH Yl</td>
<td>(Punch answer)</td>
</tr>
<tr>
<td>10</td>
<td>JUMP TO 1</td>
<td>(Start over again)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Number</th>
<th>Statement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>READ</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HALT IF 0 &gt; II</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Yl ← 1.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>JUMP TO 7 IF 1 ≥ II</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6, Cl, 2, 1, 1, II,</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Yl ← Yl x Cl</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PUNCH II PUNCH Yl</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>JUMP TO 1</td>
<td></td>
</tr>
</tbody>
</table>

Example 2. Suppose we want to evaluate the error function $Q(x) = \int_0^x e^{-t^2} dt$ for arbitrary values of $x$ using the approximation

$$Q(x) = 1 - \frac{2}{\sqrt{\pi}} \left( a_1 n + a_2 n^2 + a_3 n^3 + a_4 n^4 + a_5 n^5 \right) e^{-x^2}$$

where $n = 1/\sqrt{\pi}$ (p and the $a$'s are numerical coefficients which we will omit here). We will let $a_k <-> Yk$; $x <-> Cl$; $n <-> C2$; $Q(x) <-> C3$ (the answer); and $p <-> Y6$. $2/\sqrt{\pi} = 1.1283791$. We could simply evaluate the equations directly with the following program:

1. This person would be of below average mentality, but would be expert at arithmetic — just like a computer.
But the evaluation of the polynomial will be quite a bit more rapid if we rewrite the expression \( Q(x) = 1 - \frac{2}{\sqrt{\pi}} (n(a_1 + n(a_2 + n(a_3 + n(a_4 + n(a_5))))) e^{-x^2} \). Now we could evaluate this new equation 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.

Of course an even shorter program would be...

**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. For...

1. In this statement and the last not all of the parentheses are necessary. The \( P \) operator is rather slow when doing floating point calculations since it always requires finding a logarithm and antilogarithm — \( C2xX^2 \) would actually be much faster than \( C2P3 \) as written—but the following program speeds it up even more. The extension \( \text{EXP E} \) might have been used here as it is in the second version of the program.
instance, there are the "A" packages, designed for eight-digit accuracy and legible error display; and the "S" packages, stripped down for speed. The separate writeup for these packages goes into greater detail.

WHICH PACKAGE 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 RUNCIBLE 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:
- P operator
- Logarithm (base e) LNE
- Exponential (base e) EXPE

P3:
- P operator
- Log (base e) LNE
- Expon (base e) EXPE
- Sine (radian) SINE
- Cosine (radian) COSE
- Arctangent (rad) ARTE
- Square root RT2E

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.

CARD PREPARATION AND FORMATS.

1. Data card format.

A. Four entries per card. On the four-per-card input, data is entered on each card in "couples," each consisting of an "IDENT" — the name of a variable (17, XL, etc.) — and the VALUE of the variable itself. The "IDENT" looks like this:
where 01, 02, 03 are used for I, Y, or C respectively, and the number s is the subscript value. The number k is an arbitrary personal identification which may be attached by the programmer; it should be filled with zeroes if not used for identification.

The VALUE of the variable differs for fixed and floating point entries. For fixed point (I-) variables, write down the integer and add left hand zeroes until it is ten digits long; this is the correct form for VALUE. For floating point (Y- and C-) variables, first write the number as 

\[ a.bbbbbbb \times 10^c \]

where c is some exponent between -50 and 49, and the decimal point is as shown (with digit "a" non-zero). Then the correct form for the floating point VALUE is

\[(a) (bbbbbbbb) (50 + c)\]

Thus, five would be written 0000000005 in fixed point and 5000000050 in floating point; fifty would be 0000000050 fixed and 5000000051 floating. Zero is always written as ten zeroes, whether fixed or floating.

1. In more advanced RUNCIBLE programs, 04, 05, and 06 stand for S, T, and P regions, respectively.
2. These rules must be slightly modified when using the 653 floating point hardware: add fifty-one (51) to the exponent instead of 50; e.g., five is 5000000051 and fifty is 5000000052 for floating point VALUES.
The sign of the number must always be punched over the right-most (units) digit of the VALUE: a 12- (Y) overpunch indicates a plus sign and an 11-(X) punch indicates a minus sign. Signs must also be placed over the units digit of the IDENTs -- either plus or minus is permissible.

Columns 1-20 contain IDENT 1 and VALUE 1
Columns 21-40 contain IDENT 2 and VALUE 2
Columns 41-60 contain IDENT 3 and VALUE 3
Columns 61-80 contain IDENT 4 and VALUE 4

If less than four variables are to be entered on a card, the IDENT following the last VALUE must be filled with all zeroes and the remainder of the card may be blank.

A READ statement in RUNCIBLE will cause cards to be read until a card with IDENT 1 negative is found. Any number of cards having IDENT 1 positive may precede the card having negative IDENT 1, which stops the READ statement and signals for continuation of the program.

A 12- (Y) overpunch in column 4 is necessary to identify the four-per-card format. The sample card shown assigns I7 = 342, Y4 = -58.3, and C10 = .00125; observe the IDENTs and VALUES carefully.
B. Seven entries per card. Up to seven variables may be entered on one card if the subscripts are all consecutive. The IDENT for the first variable is put in columns 1-10 in the same form as noted above except that \( k \) now must be a number between 1 and 7 which specifies how many consecutive values are to be loaded by this card. Then VALUE 1 (corresponding to subscript \( s \) of the IDENT) is placed in columns 11-20, VALUE 2 (corresponding to subscript \( s + 1 \)) is placed in columns 21-30, VALUE 3 (for \( s + 2 \)) into cols. 31-40, VALUE 4 (for \( s + 3 \)) into cols. 41-50 and so on, leaving unused columns blank if desired.

A 12- (Y) overpunch in column 7 is necessary to identify the seven-per-card format. The sign of IDENT 1 again controls the reading of cards (see four-per-card format); when IDENT 1 is negative as it is above it signifies the last card of a series.

Seven-per-card and four-per-card formats may be used interchangeably. The sample card shown assigns \( C12 = 2.56 \times 10^{30} \), \( C13 = 0 \), and \( C14 = -1 \).

2. Output card format. Output card format is precisely the same as input card format and can therefore be used as input to another program. With four-per-card output, IDENT 1 is made negative on all cards, and the identification \( k \) is the statement number of the PUNCH statement being executed. IDENT 1 is negative only on the last card of the seven-per-card output (when the word THRU was used in the compiler statement).
3. **RUNCIBLE Statements.** Since the basic 650 for which this program is designed recognizes only numbers and alphabetic characters—_not_ equal signs or quotation marks—an alphabetic code is used for these symbols when punching the statements onto cards. Here is a complete list of symbols used and their alphabetic equivalents:

- ( | left parenthesis, bracket, or brace... L
- ) | right parenthesis, bracket, or brace... R
- \_ decimal point ................. J
- \(- substitution .................. Z
- = equals ......................... U
- > greater than .................... V
- \( greater than or equal .......... W
- , comma .......................... K
- " quotation marks (right or left) . Q
- + plus ............................ S
- - minus .......................... M
- \* times .......................... X
- / divided by ...................... D

Each statement must be punctuated by an F ("finish"), which is added at the end. The very last statement of a program must end with an FF ("Final finish").

The statement number, n, is punched as (0000 + n) in columns 1-4. Column 5 must contain both a 12(Y) punch and a 9 punch (this is the code for the letter "I") to identify the format, and a 9-punch may be added in column 5 to be picked up by tabulator board wiring for use in listing the deck. The statement itself is punched in columns 43 through 69, and column 70 must contain the F which terminates it. (If a statement is so long it does not fit on one card, up to five cards may be used. In this case the "F" should appear only on the very 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 148 characters, including the final F or FF as the case may be. Columns 1-4 need only be punched on the first card of each statement.) Columns 6-42 and 73-80 are ignored by RUNCIBLE except columns 7 and 41 must not contain 12- (Y) punches but these

1. It may prove convenient at your installation to put these symbols directly onto the keys of your keypunch keyboard; e.g.,
2. Column 10 should be blank if the 533 plugboard has the added statistical features.
columns are usually not used.

- Blank columns in the middle of a statement are always ignored, and they may be inserted or deleted at will. A substitution statement must start in column 43. Here is example program one as it would be punched onto cards; statement number three is shown above.

<table>
<thead>
<tr>
<th>STATEMENT NUMBER</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td>43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72</td>
</tr>
<tr>
<td>0 0 0 1 S</td>
<td>READ</td>
</tr>
<tr>
<td>0 0 0 2 S</td>
<td>HALT IF $ V I I</td>
</tr>
<tr>
<td>0 0 0 3 S</td>
<td>Y1 Z 1 J</td>
</tr>
<tr>
<td>0 0 0 4 S</td>
<td>JUMP TO 7 IF 1 W I I</td>
</tr>
<tr>
<td>0 0 0 5 S</td>
<td>6 K C 1 K 2 J K 1 J K I I K</td>
</tr>
<tr>
<td>0 0 0 6 S</td>
<td>Y1 Z 1 X C 1</td>
</tr>
<tr>
<td>0 0 0 7 S</td>
<td>PUNCH #1 PUNCH #Y</td>
</tr>
<tr>
<td>0 0 0 8 S</td>
<td>JUMP TO 1</td>
</tr>
</tbody>
</table>

Here $ represents zero and $ an alphabetic letter O. Appendix VII contains another sample program punched on cards including the Header and Comments Cards.

Rules:
1. The 1st character of a statement must be in col. 43.
2. Every statement must end with an "F".
3. The "F" when it appears must be in col. 70.
4. No statement may exceed 5 cards.
4. Header Card and Comments Card. Every RUNCIBLE program must be preceded by two cards — the Header Card and the Comments Card. The Header Card contains important control information, and its format is as follows (all numbers are in the form of fixed point VALUES):

- **COLUMNS 1 - 10:** the number of the maximum \( I \)- subscript
- **COLUMNS 11 - 20:** the number of the maximum \( Y \)- subscript
- **COLUMNS 21 - 30:** the number of the maximum \( C \)- subscript
- **COLUMNS 31 - 40:** the highest statement number used
- **COLUMNS 41 - 50:** the total number of special locations used by extensions—this may be calculated by consulting the writeup for each extension used. (These "special" locations are those which, for some reason or other, are supposed to be in sequential order.) Each extension now has two magic numbers associated with it: The number of special locations—this goes into columns 41-50—and the number of other locations—this goes into columns 61-70.

- **COLUMNS 51 - 60:** the number of locations used by the basic package.¹
  (This is 325 for P1A and 525 for P2A, 751 for P3A the most frequently used packages; for other packages, see their own write-up.)
- **COLUMNS 61 - 70:** the total number of locations used by extensions (excluding the basic package) — this may be calculated by consulting the writeup for each extension used.
- **COLUMNS 71 - 80:** all zeroes (unless using procedure of Appendix IV)

**COLUMNS 10, 20, 30, 40, 41, 50, 60, 70, 80:** 12- (Y) overpunches.

The punch in column 41 is necessary to identify the header card.

Any of these numbers may be made a little 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.

---

¹ When using B-processing (see Operation Modes) this number must be at least 75, or at least 175 if any extensions are used, because of the Re-location Package used to load the program.
The example Header Card which appears on the next page is the one determined by example program one, using basic package F1A.

The Comments Card is generally easier to prepare than the Header Card: columns 1 - 42 and 73 - 80 are blank and the remaining columns 43 - 72 may be filled with anything the programmer's heart desires (as long as it is acceptable to the alphabetic attachment) -- the title of the program, his own name, or his girlfriend's name; or he may choose to be evasive about the whole thing and leave the card completely blank.

**MODEL OF OPERATION.** There are twenty-four different modes of operation to choose from while compiling, and these are selected by setting switches on the 660 console.

1. **Clocking.** RUNDEX II's clocking feature is a major step forward for the testing of a compiler program during its trial runs. "Minimum" clocking stops information on the number of the statement which is being executed and will display this number if the computer stops during the running phase (see Steps). "Minimum" clocking does not require any more space in the 660's memory than the program would normally use; it does, however, add approximately .025 seconds making running time to every non-zero statement executed.

2. Additional available clocking features (stopping at certain statement numbers, line tracing, and full tracing) are explained in detail in Appendix 1. There is a three-way choice while compiling: to use no clocking, "minimum" clocking, or the "full" clocking, which allows any of all of the aforementioned

---

The Comments Card between a MLA V Ill comments card which is the first card of the symbolic output.

If it is non-safe.
the program, his own name, or his girlfriend's name; or he may choose to be
obstinate about the whole thing and leave the card completely blank.¹

MODES OF OPERATION. There are twenty-four different modes of operation to
choose from while compiling, and these are selected by setting switches on
the 650 console.

1. Clocking. RUNCIBLE I's clocking feature is a major step forward for
the testing of a compiler program during its trial runs. "Minimum" clocking
keeps tabs on the number of the statement which is being executed² and will
display this number if the computer stops during the running phase (see Stops).
"Minimum" clocking does not require any more space in the 650's memory than
the program would normally use; it does, however, add approximately .026
seconds machine running time to every non-zero statement executed.

Additional available clocking features (stopping at certain statement
numbers, flow tracing, and full tracing) are explained in detail in Appendix
I. There is a three-way choice while compiling: to use no clocking, minimum
clocking, or the full clocking, which allows any or all of the aforementioned

1. The Comments Card becomes a SOAP III comments card which is the first card
of the symbolic output.
2. If it is non-zero.
extra features. The extra features require an additional 98 memory locations.

2. X or Y operation. If a 653 floating point attachment is available the program will run much faster using this hardware directly rather than going through the floating arithmetic extensions. RUNCIBLE will put the 653 to work if desired -- this we will call "Y" operation for convenience, as opposed to ordinary compilation which we shall designate "X" operation. The floating point exponents on input data cards and output cards in the final program are slightly different (excess-51 rather than excess-50) when using Y-operation.

3. A or B operation. RUNCIBLE I normally turns out a SOAP III symbolic program which must first be assembled by SOAP before the finished deck is ready to run -- this is called "A" operation. If desired, however, the SOAP phase may be bypassed and RUNCIBLE will turn out a machine language program immediately with five instructions on each card ("B" operation).

B-operation is to be recommended for smaller scale problems and when conservation of cards is more important than conservation of time. A-operation is more versatile and is recommended for programs which will be long runs. A B-program requires less than one-sixth the cards of an A-type (multipass) program during the IT-language to machine language translation, and takes approximately one-half the time to assemble; however, it uses about 10 percent more time while running. (More accurate timing has yet to be made for this comparison.)

Extensions: Programming is exactly the same for A or B processing except that there are different maximum limits for extensions. In type A, no more than nine named extensions may be given in any one statement; type B restricts the programmer to a total of ten different extensions (named or not in P1 or P2) in his entire program.

4. Error searching. There are many checks on programming errors built into the RUNCIBLE program (see STOPS); these are sometimes awkward to correct without beginning compilation over again. The error search mode feature in RUNCIBLE I will check an entire program for errors.

1. Index register A (8005) is used to calculate variable subscripts when Y-operation is used, but all index registers are free for use by extensions.
for goofs without punching any of its normal output cards. In this case any error which would normally have halted compilation will not usually stop the computer -- it will be punched onto a card instead.

Selecting the modes. These various modes of operation are selected in the following manner: we will "build" a four-digit number klmn. Digit \( n \) is 9 if not clocking and 8 if clocking; and if clocking is used, digit \( m \) is 9 for minimum clocking only and 8 for the expanded clocking features described in Appendix I. For \( X \)-operation, digit \( l \) is set to 9, while it is 8 when \( Y \)-operation is desired. Digit \( k \) is 9 when multipass (A) operation is to be used and 8 when using the B (single pass) feature. The number \( kmn \) finally is plus when normal output is called for and minus when searching out errors.

All of these rules can be remembered more easily by the general pattern that all four digits are 9 for basic operation, and changing any one of them to 8 calls out a more special feature. Four eights is more or less asking RUNGIBLE for the "works." The rules are summarized in this chart:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>X</th>
<th>Minimum Clocking</th>
<th>Don't Clocking</th>
<th>Normal Clocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multipass</td>
<td>650</td>
<td>650</td>
<td>650</td>
<td>650</td>
<td>650</td>
</tr>
<tr>
<td>One Pass</td>
<td>653</td>
<td>653</td>
<td>653</td>
<td>653</td>
<td>653</td>
</tr>
</tbody>
</table>

OPERATOR INSTRUCTIONS.

Step 1. Insert RUNGIBLE 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 blank cards.

Step 3. Place the RUNGIBLE I deck in the read hopper, face down, and place your program deck on top of it. The program deck consists of 1) Header Card 2) Comments Card and 3) Statements (in that order).

1. When not clocking, digit \( m \) should also be a 9.
Step 4. Set the 650 console to the following:

<table>
<thead>
<tr>
<th>SWITCH</th>
<th>SETTING</th>
<th>(see Operation Modes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE ENTRY</td>
<td>70 1951 klmn +</td>
<td></td>
</tr>
<tr>
<td>PROGRAMMED</td>
<td>STOP</td>
<td></td>
</tr>
<tr>
<td>HALF CYCLE</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>ADDRESS SELECTION</td>
<td>anything -- 1888 recommended</td>
<td></td>
</tr>
<tr>
<td>CONTROL</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>DISPLAY</td>
<td>LOWER ACCUM</td>
<td></td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>SENSE</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>

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. 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. Clear out the read feed and put the RUNCIBLE I deck back into the file.

Step 10. If using B-operation (see Operation Modes) skip down to step 14. Otherwise repeat step 5 and then run the SOAP III deck into the machine (by pressing START and END OF FILE at the proper times).

Step 11. If your program uses extensions which are not included in the basic package you are using, lift the first few cards from the output of step 8 which have 12- (Y) punches in columns 76-80 and insert the extensions in symbolic form into the output deck at this place. Now, place in the read feed face down (in order):
1) The Reservation Package which accompanies your basic package; e.g., R1A corresponds to P1A.

2) The output of step 8 — with extensions inserted if they are used.

3) A BLANK CARD.

Step 12. Repeat steps 6, 7, and 8; clear out the read feed and put the SOAP deck, Reservation Package, and extensions back into the file. The output you now have is a running program assembled at five instructions per card.

Step 13. Repeat step 5 and run in your basic package followed by the output of step 12. You are doing fine. Skip down to step 16.

Step 14. Place in the read feed, face down, in order

1) The Relocation Package (standard for type B operation)
2) The output from step 8
3) Any extensions used in relocatable form
4) The basic package you want
5) A BLANK CARD

Step 15. Repeat steps 5, 6, and 7.

Step 16. When your deck is all in, the machine should be stopped with 1999 in the ADDRESS lights. Turn the CONTROL switch to ADDRESS STOP. Set the ADDRESS SELECTION switches to 1888. If you are using any of the expanded clocking special features, change the storage entry switches to the setting described in Appendix I; if you are running on ERROR SENSE (Appendix III) make the desired console setting; otherwise leave everything on the console alone. Now, push PROGRAM START (not COMPUTER RESET this time!).

Step 17. If your program contains any READ statements, place the data cards in the read feed in their proper order. Hit the START button.

Step 18. Congratulations. If you have gotten this far, any output may consist of the answers to your problem. Be sure to return all library decks to the files and to remove your other cards from the machine.
GENERAL INFORMATION FLOW
FOR RUNCIBLE I

Compiling Phase (A Mode)

INPUT
HEADER CARD

YOUR STATEMENTS

FIRST CARD RUNCIBLE I

RUNCIBLE DECK

OUTPUT

SOAP III (Symbolic) Deck
Compiling Phase (B Mode)

Input

Header Card

Your Statements

Runcible Deck

First Card
Runcible I

Output

Machine Language (5 Per Card) Deck
RUNNING PHASE
B MODE ONLY

FIRST CARD
RELOCATION PACKAGE

OUTPUT FROM
COMPILING PHASE

RELOCATABLE
EXTENSIONS

BASIC PACKAGE

BLANK CARD

DATA (IF ANY)

MACHINE SHOULD STOP
READING HERE. SEE
STEP 16, PAGE 26.

ANSWERS
**Soap Phase**

*(A Mode Only)*

- **First Few Cards** of Compiling Phase Output (i.e., all of the ones with Y punches in cols. 76-80)
- **First Card** SOAP III
- **First Card** R-- Deck
- **Soap III**
- **Reservations Package** (may be R1A, R2A, R3A, R1Y, R2Y)
- **Extensions (if any)** in symbolic Soap III form (i.e., A Mode Extensions)
- **Remainder of Compiling Phase Output**
- **Blank Card**

**Machine Language Program**

5 Instructions/Card
**Running Phase**

**A Mode Only**

Data (if any)

Output from Soap Phase

Basic "P" Package

Rerun Procedures. There are several cases where one does not wish to reload a large program deck when the program is already in the machine.

1. If RUNCIBLE is still in memory, and you are not changing from A to B or from X to Y operation modes, you may compile a second or successive program by setting the STORAGE ENTRY switches to 70 1999 klmn +, placing your program deck in the read feed, and continuing with step 5. Change the console back to 70 1951 klmn + when step 10 is reached.

2. If SOAP III is still in the machine, you may SOAP a second or successive program by setting storage entry to all zeroes, repeating step 5 and continuing with step 11. Return the console to its previous setting after the SOAP phase has been completed.

3. During the running phase of your program you may always start it over from the beginning by setting console to 00 0000 1999 + and hitting COMPUTER RESET, PROGRAM START.

STOPs. The above operator instructions work fine for the unusual case when the 650 does no stop. You have undoubtedly noticed the rules given thus far for programming; they are certainly not unreasonable and, of course, must be followed. Error checks are made by RUNCIBLE on almost every rule. When the machine stops, an indication of which error has been made appears in the display lights as described below.

If the DISTRIBUTOR light is on, you most likely have a mistake punching your card. Run the cards out of the hopper—it's the fourth last one out you want to check.

When the DISTRIBUTOR light is not lit, the next step is to look at the OPERATION lights. If they are

1. 00 (address 9998): The program deck has not been loaded properly; there is a card missing or out of order. Remedy: try again with a good deck. If the 00 9998 stop has occurred, your own five-per-card program may be out of order if the basic package is already loaded properly; check columns 7-10 on your cards. The numbers in these columns must start at 0001 and be consecutive. If you have dropped a deck, its loading routine may be unscrambled by watching the 12-overpunches in columns 71-79.

38
2. 70: The machine wants to read a card. Depress read START, or END OF FILE if the last card in the hopper is halfway in; however, if the CARD FEED STOP light is on, panic.¹

3. 71: The machine wants to punch a card. Depress punch START. If this doesn't work, there is a bent card in the punch feed or the feed is filled to capacity. In the latter case, relax and take all cards out of the feed as the first portion of your output. But if there was a bent card, remove it and play around until the unlabeled light goes out. You will have now one or more extra garbage cards (see step 8 in operator instructions) in the middle of your output which must be removed.

4. In the 90's: You have not set the storage entry switches correctly.

5. Blank: Now you must check the ADDRESS lights, as follows: **Compilation phase.**
   A. 1234: This is a correctable error in a RUNCIBLE statement; error cards are punched for this type of mistake when in error search mode (see Operation Modes). A list of these goofs will be given below. To restart, correct the offending statement, place it first into the read hopper, and depress PROGRAM START and read START.

   B. 1953, 1954: Turn the OVERFLOW switch to SENSE and run the deck again.

   C. 9500: Columns 31-40 of your Header Card are negative or zero; or storage exceeded and *not* in error search mode.² When tight for memory space in a long program, always check by an error search *first*. Remedy: change your header card and start over.

   D. 9876: If your entire deck of statements has been processed, cheer up, for this is no mistake; the compilation mode has been successfully completed. If not, however, an FF occurred on the last statement processed; or you are using B-operation and columns 71-80 of your Header Card are not zero.

   Remedy: start again.

**CORRECTABLE ERRORS IN STATEMENTS:** (1234 stop or error search output) If a 1234 stop occurs and the DISPLAY switch is turned to LOWER ACCUM, your error indication will be as follows:

```
0 n n 0 a a 0 b b
```

where nnn is the number of the offending statement and aa and bb are

¹ You have a bent card. Clear the read feed, and replace the last two cards out into the hopper. Fix up the bent card and replace the deck; depress STOP and then START (both keys).

² On this error a display is given as in the 1234 stops (see 02 02 stop).
explained in the table below. You may correct your statement immediately by making up the new card or cards necessary and replacing the incorrect statement, putting your cards back in the read hopper, pushing read START on the 533 and PROGRAM START on the 650.

If you are in error search mode, list any output cards on a tabulator with the SOAP III control panel. Your statement number will appear, followed by an entry in the table below, followed by that portion of your statement which caused the trouble.

Here is the error table — if your error indication does not appear to make any sense, check to see if you have punched the F's correctly in column 70.

<table>
<thead>
<tr>
<th>Letters:</th>
<th>aa bb</th>
<th>Error type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>A zero statement number immediately following an iteration statement; or more than five cards in a statement; or statement too long—over 100 machine language instructions were necessary to execute your statement (remedy: break it into two statements).</td>
<td></td>
</tr>
<tr>
<td>01 01</td>
<td>Subscript or statement number over 999.</td>
<td></td>
</tr>
<tr>
<td>02 02</td>
<td>Problem too long—2000 memory locations exceeded. You must start over if not in error search mode. Either cut down numbers on header card to bare minimum or shorten or segment your program.</td>
<td></td>
</tr>
<tr>
<td>04 04</td>
<td>Constant larger or smaller than allowable. (fixed point ≥ 1 billion or floating point &lt; 10^-50 or ≥ 10^50)</td>
<td></td>
</tr>
<tr>
<td>06 06</td>
<td>More than ten subroutines used in addition to the basic package when using type B operation.</td>
<td></td>
</tr>
<tr>
<td>07 07</td>
<td>Column 43 on first card of a substitution statement not I, Y, or C; or too many named subroutines used in one statement; or parentheses and quotation marks nested over nine deep.</td>
<td></td>
</tr>
<tr>
<td>08 08</td>
<td>Unmatched parentheses or quotation marks; or a floating point number when it should have been fixed; i.e., in a statement number, an exponent, or a subscript.</td>
<td></td>
</tr>
<tr>
<td>E R 65 79</td>
<td>&quot;CONVERT&quot; when not using type A operation.</td>
<td></td>
</tr>
<tr>
<td>G F 67 66</td>
<td>&quot;ARITHMETIC FLOATING&quot; when not using type AX operation.</td>
<td></td>
</tr>
</tbody>
</table>
### Letters: \( \text{aa} \) \( \text{bb} \) Error type: (continued)

<table>
<thead>
<tr>
<th>I</th>
<th>F</th>
<th>69</th>
<th>66</th>
<th>&quot;IF&quot; not followed properly by one of the relations ( U, V, ) or ( W ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>F</td>
<td>72</td>
<td>66</td>
<td>Iteration statements nested over four deep; or more than five symbols between commas in an iteration statement; or too many commas in an iteration statement; or bad spelling at the beginning of a matrix definition statement.</td>
</tr>
<tr>
<td>L</td>
<td>F</td>
<td>73</td>
<td>66</td>
<td>&quot;ARITHMETIC DECIMAL&quot; when not using type AX operation.</td>
</tr>
<tr>
<td>R</td>
<td>L</td>
<td>79</td>
<td>73</td>
<td>Improper use of REMAINDER (see Appendix III); or the invalid pair }[ in a statement.</td>
</tr>
<tr>
<td>O</td>
<td>K</td>
<td>90</td>
<td>72</td>
<td>PUNCHing more than four variables in one statement.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>Erroneous substitution statement; or statement does nothing. Examples: ( \text{II} \leftarrow \text{II} ); ( \text{II} \leftarrow \text{II} ); ( \text{II} ).</td>
</tr>
</tbody>
</table>

...... and if not in the list so far, 

| ? | K | ?? | 72 | Too many multiple input extensions nested together; or HALT followed by a variable. |
| ? | ? | \( \text{aa} \) \( \text{bb} \) | | The code for \( \text{aa} \) and \( \text{bb} \) is explained in the following table; they are two letters which should not appear next to each other in any meaningful RUNCIBLE statement! |

| 61 | A | 76 | 0 | |
| 62 | B | 77 | P | |
| 63 | C | 78 | Q | " |
| 64 | D | 79 | R | } ] | or \( \text{U}, \text{V}, \text{W} \) |
| 65 | E | 82 | S | + | or \( \text{M}, \text{U}, \text{V}, \text{W} \) |
| 66 | F | 83 | T | or the word PUNCH |
| 67 | G | 84 | U | = | |
| 68 | H | 85 | V | > | |
| 69 | I | 86 | W | \( \geq \) | |
| 71 | J | 87 | X | \( \leq \) | |
| 72 | K | 88 | Y | \( \leq \) | |
| 73 | L | 89 | Z | \( \leq \) | |
| 74 | M | 90 | lighthand end of statement 1 | |
| 75 | N | 99 | any integer 0-9 or a J | |

SOAPing phase. Assuming that properly written extension are being used, no stops should ever occur until the end of the SOAP III phase. However, if the OPERATION lights are blank you may have a stop of

0444: You forgot the reservation package.
0666: You got the compiler output jumbled up somehow.
0777
9800: You have successfully finished SOAPing; or (if all cards are not processed) you forgot to remove a garbage card (caused by a punch jam) from the middle of your deck. See 71 stop.

1. In this case, \( \text{bb} \) represents the first symbol of the statement.
Running Phase. 1. OPERATION lights not blank. ADDRESS lights:

8011: Overflow or underflow has occurred somewhere during floating point arithmetic while using Y operation. The statement on which it occurred is in machine location 0018 if clocking.

1888: Programmed stop in basic package. When using "A" packages, the DISPLAY switch may be dialed to indicate:

DISTRIBUTOR May or may not contain the input argument to the extension; when it does not, it is the same as the UPPER ACCUM.

UPPER ACCUM The number of the statement being executed; zero if clocking not used.

LOWER ACCUM Displays the ten-digit number

\[ Oa \text{ Onnn AAAA} + \]

nnn is the number of the extension; \( a \) is the error type which may be found by referring to the writeup of the extension. In the basic packages,

\( a = 1 \): Floating point exponent less than -50; pushing PROGRAM START will continue calculation using zero as a result.

\( a = 2 \): Floating point exponent more than +49; or attempting to fix a number of more than eight digits.

\( a = 3 \): Division by zero or logarithm of a non-positive quantity.

When the OVERFLOW light is not on, the program may be restarted by setting the ADDRESS SELECTION switches equal to AAAA, turning CONTROL to MANUAL OPERATION, punching COMPUTER RESET and TRANSFER, turning CONTROL back to ADDRESS STOP, resetting address switches to 1888, and hitting PROGRAM START. Zero will be employed as an answer to the calculation.

xxxx: Operation lights not blank, address not 1888 or 8011: If Overflow light is on and DISTRIBUTOR contains zero, you are attempting fixed point division by zero.

2. OPERATION lights blank. ADDRESS lights:

8765: Wrong basic package for your program. Dial PROGRAM REGISTER -- it contains 01 Onnn 8765 where nnn is the extension missing from the package you are using.

8778: (Type B Operation) You are attempting to load a subroutine in relocatable form which you did not properly call for in your program, or HALT FF statement.

1. See next page for extension numbers in the basic packages, and extension writeups for the number of a named extension.
8888: You had a 1888 stop with OVERFLOW light not on and did not restart as described above, or you forgot to turn the switch to Address Stop at Step 16.

0000-1999: Stop caused by HALT statement in your program; if you are displaying a number it is in LOWER ACCUM. PROG REGISTER should contain 01 3005 xxxx.

Extensions in the basic packages which include error stops have the following numbers:

```
n00 = 001: Log to base 10—not in P1
       002: 10 to the X—not in P1
       006: Floating point division
       008: Floating point addition and subtraction
       009: Floating point multiplication
       010: Fl. pt. number to fl. pt. power (P)—not in P1
       011: Clocking
       018: Natural logarithm—not in P1
       019: e to the X—not in P1
       020: Square root—not in P1 or P2
       021: Sine—not in P1 or P2
       022: Cosine—not in P1 or P2
       023: Tangent—not in P1 or P2
       033: Arctangent—not in P1 or P2
       500: Fix a floating point number
       501: Fx. pt. number to fx. pt. power (P)—not in P1
```

"IF NOTHING ELSE WORKS, READ THE INSTRUCTIONS"

1. If this number is 01 0011 xxxx, you have a statement number stop (see Appendix I."
EXTRA CLOCKING FEATURES. While doing minimum clocking, the present statement number is kept in machine location 0017; the previous statement number may be found in location 0018. The following features may also be used while clocking — if any or all of these are used, an additional 98 drum locations are needed. (These features are primarily intended for those who have a more complete knowledge of 650 machine language.)

1. Statement number stopping. The computer may be stopped just before executing any statement whose number is nonzero.

2. Flow tracing. A card will be punched just before executing every non-zero statement showing the contents of the accumulators.

3. Complete tracing. A card will be punched for every machine language instruction executed except those in extensions, showing contents of the accumulators and distributor. Extensions will be run at full speed. Tracing may be stopped or started at any statement number.

CAUTION: Iteration statements generate several zero statements which when executed are not clocked or flow traced.

To use any of these extra features, digits m and n (see Operation Modes) should be set to 8 while compiling, and the following console setting is to be made while running the machine language program:

```
q r s f  A A  B B  B
```

The sign controls conditional PUNCH statements as before, and digit f is free for some use which is as yet unforeseen. Clocking is independent of the sign or of digit f. Digit q tells what features are to be used:

- $q = 0 \text{ or } q > 4$: Minimum clocking only.
- $q = 1$: Statement number stopping and clocking.
- $q = 2$: Flow tracing, statement number stopping, and clocking.
- $q = 3$: Complete tracing, statement number stopping, and clocking.
- $q = 4$: Complete tracing, flow tracing, statement number stopping, and clocking.

1. Complete tracing will correctly trace all RUNCIBLE output instructions but will not handle branch-distributor-8 commands, references to 8001 or 8003 for instructions, or division with opposite signs. When "doctoring" of compiler output is done, therefore, care should be used to stay away from these instructions if complete tracing is also desired.
Statement number stops: The machine will halt just before executing statement number AAA if digit r is 8, and r must be 9 when this stop is not desired. The machine will halt before executing statement BBB if digit g is 8, and g must always be 9 when this stop is not desired. The halt command will have data address 0011, and the DISTRIBUTOR will contain the statement number on which the stop occurred. The accumulators do not contain anything of interest. To continue, depress PROGRAM START.

Complete tracing will start up when statement AAA is encountered and will be discontinued just before executing statement BBB. (If AAA equals BBB, no tracing will occur.) Digit g may be changed while running the program, but its effect will be noticeable only after the present statement has been completed. Never change the console setting unless the 650 is stopped.

Card formats:

Flow tracing: Complete tracing:

| Word 1: | FLO the letters COM |
| Word 2: | 000000nnnn 00kkkkmmnn (nnnn is statement number; kkkk is location of instruction.) |
| Word 3: | 0000000000 Instruction |
| Word 4: | UPPER ACCUMULATOR |
| Word 5: | LOWER ACCUMULATOR |
| Word 6: | 0000000000 |
| Word 7: | 0000000000 DISTRIBUTOR |
| Word 8: | 0000000000 |

When flow tracing with type Y operation, word 7 of the output cards will contain the contents of the DISTRIBUTOR if complete tracing is not being used simultaneously. Results of every substitution statement may be found in the distributor, while only certain types of substitution statements leave the answer in the lower.
APPENDIX II

MATRIX NOTATION. There are two types of doubly-subscripted notation available in RUNCIBLE I, and since they have somewhat different rules, they will be discussed separately here. It is possible to program matrix problems without this notation, but the machine language output on matrix notation is usually much better and the notation is convenient and quite easy to use.

1. VARIABLE-SIZE MATRICES. Two matrices having variable row length may be used: the YN matrix and the CN matrix. The number of columns in the YN matrix, whenever it is used, is always contained in the current value of variable I1, and I2 contains the number of columns of the CN matrix. Notation used in statements is

\[ \text{YN}(v1, v2) \]

where \( v1 \) (the row) and \( v2 \) (the column) are any fixed point mathematical expressions. Variable-size matrices start with row zero and column zero, and they overlap the other Y- and C- variables. YO is the same as YN(0,0) and if I2 = 4, say, CN(1,0) is variable C4. This may be seen more clearly in the following equivalence diagram, showing in this case a 4 x 3 matrix:

\[
\begin{array}{ccc}
\text{YN}_{0,0} & \text{YN}_{0,1} & \text{YN}_{0,2} \\
\text{YN}_{1,0} & \text{YN}_{1,1} & \text{YN}_{1,2} \\
\text{YN}_{2,0} & \text{YN}_{2,1} & \text{YN}_{2,2} \\
\text{YN}_{3,0} & \text{YN}_{3,1} & \text{YN}_{3,2}
\end{array}
\]

\[
\begin{array}{ccc}
Y_0 & Y_1 & Y_2 \\
Y_3 & Y_4 & Y_5 \\
Y_6 & Y_7 & Y_8 \\
Y_9 & Y_{10} & Y_{11}
\end{array}
\]

I1 = 3

2. FIXED-SIZE MATRICES. Fixed-size matrices are the type most frequently found in compilers prepared for other computers; the row size must be specified by the statements of the program. There are ten of these matrices available: YN1, YN2, ..., YN5, CN1, ..., CN4, and CN5. These matrices start with the more conventional element 1,1 in the upper left-hand corner. Notation used in statements is

\[ \text{YN2}(v1, v2) \]

where \( v1 \) (the row) and \( v2 \) (the column) are any fixed point mathematical expressions. Y- and C- variables are overlapped by the fixed-size matrices; the subscript number where each matrix will start (element 1,1 of the matrix) is supplied by the programmer.
A matrix definition statement must precede the use of any fixed-size matrices, or erroneous (undetected) operation will occur. Such a statement looks like this:

\[ Y^n_n, \text{COLS}, c, \text{START}, s, \]

or

\[ C^n_n, \text{COLS}, c, \text{START}, s, \]

where \( n \) is a number from 1 through 5, \( c \) is the number of columns in the matrix, and \( s \) is the subscript number which should correspond to element \( Y^n_n(1,1) \). Note the commas and English words in the statement; \( c \) and \( s \) must be fixed-point numbers without leading zeroes. A matrix definition statement cannot be made conditional, of course; when the program is running it is treated like a BYPASS statement, and so it is usually given statement number zero.

The equivalence diagram below illustrates the overlay of a 4 x 3 matrix of fixed size.

\[
\begin{array}{ccc}
\text{CN}^3_{1,1} & \text{CN}^3_{1,2} & \text{CN}^3_{1,3} \\
\text{CN}^3_{2,1} & \text{CN}^3_{2,2} & \text{CN}^3_{2,3} \\
\text{CN}^3_{3,1} & \text{CN}^3_{3,2} & \text{CN}^3_{3,3} \\
\text{CN}^3_{4,1} & \text{CN}^3_{4,2} & \text{CN}^3_{4,3}
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{C}^3_{2,2} & \text{C}^3_{3,2} & \text{C}^3_{4,2} \\
\hline
\text{C}^3_{2,3} & \text{C}^3_{3,3} & \text{C}^3_{4,3} \\
\hline
\end{array}
\]

A sample matrix problem is given in Appendix VI.
EXTRA PROGRAMMING FEATURES. RUNCIBLE I contains, in addition to those features described in the main portion of this manual, some secret hidden tricks which are extra added attractions for the more advanced programmer.

1. Remainder. The remainder of a fixed point division may be saved instead of the quotient by enclosing the division and parentheses and preceding it with the word REMAINDER; e.g.,

```
REMAINDER (13/5)
```

The expression inside the parentheses must be some fixed point division—it may have a more complicated dividend and divisor as

```
REMAINDER [(18+2)/(156)]
```

When used in a statement it is treated by RUNCIBLE as though the entire thing were enclosed in parentheses, so parentheses need not be placed again around the entire remainder expression. REMDR or simply RR may be used as an abbreviation for REMAINDER.

2. Variables in the core. Your variables may be placed into immediate access (core) storage for faster operation by adjusting the Header Card. Word 1, 2, or 3 of the header may be made negative (an 11-punch replacing the 12-punch) and the corresponding variables, I, Y, or C respectively, will be placed into the core. There are a few restrictions, however: no matter which variables, I, Y, or C, are placed in core, that one with subscript zero is put into 9000, subscript one into 9001, etc., so that no subscript greater than 59 may be used. Also if more than one type of variable is put into core, overlapping occurs between those having equal subscripts, and so care must be taken that no subscripts are duplicated between these two variables. For example, if both the I's and the Y's are placed in core, and II is used in the program, Y1 may not be used at the same time since the two variables are both kept in location 9001.

3. Switching arithmetic. This is another step forward in compiler offerings — the floating point variables may be changed into a fixed decimal point form with the number of decimal places to be saved

```
1. The other numbers in a negative word are ignored.
```
left as a variable. This is useful in some calculations for speed of operation and for editing answers before punching, making them more easily read. Switching arithmetic can be done only when using type A and type X operation modes simultaneously. A group of extensions is supplied for this operation; a full description of how to use these features completely is given in their writeup. The compiler statements ARITHMETIC DECIMAL and ARITHMETIC FLOATING will change from floating decimal to fixed decimal arithmetic or back from fixed to floating, respectively; they may not be made conditional. Only the arithmetic of Y- and C- variables is affected. During the running phase, variable IO (I-zero) will specify the number of decimal digits desired; this can conveniently be changed from one run to another until the correct scale for the problem is determined.

4. Error Sense Running. For production runs it may be found economical to leave the ERROR switch on SENSE and then have a built-in error restarting procedure in the program to bypass machine errors when they occur. This may be done with RUNCIBLE, as follows:

The error correction procedure is programmed as a block of statements, and on the first statement of the restarting program the number of the statement on which the error occurred is by now in location 0018 if using minimum clocking. This value may be programmed by asking for variable I(-1) if the I's are not in core; if they are in core, call for I(3982)! For example, the first statement might be

\[ I6 \leftarrow I(-1) \]

The error restart routine which you program will continue then perhaps by attempting to correct any harm done, keeping count to see if the error occurs repeatedly, and then going back into the main program.

This operating mode is set up by the statement

SET ERROR CORRECTION TO n

where n is the (fixed-point constant) number of the statement at which the correction procedure should start if an error is sensed. This number may be reset during different portions of the program by other similar statements. The SET ERROR CORRECTION statement must
precede any part of the program it is supposed to be operative on; it may be made conditional. During the running phase of the program the console switches must be set to NOP to 1998 to utilize this error sense operation.¹

5. Spelling liberties. RUNCIBLE is not always very fussy about the programmer's spelling of English words (this will come as a relief to many people):

- **READ** is any statement ending with a D (e.g., READ A CARD, or DONT READ)
- **HALT** is any statement ending with a T or H; HALT n is identified by the T-integer combination.
- **BYPASS** is any statement ending with an S.
- **PUNCH** may be replaced by the letter T; it may also be any group of letters or numbers starting with P and ending with H as long as there is only one P. (e.g., PINCH, or PEANUT GOULASH)
- **REMAINDER** is anything beginning and ending with R without any R's in the middle.
- **JUMP TO** is anything ending in G or O not containing an E.
- **SET ERROR CORRECTION TO** is anything ending in G or O containing the letter E. (e.g., RESTARTING)
- **IF** and **THRU** must be spelled exactly.
- **COLS** and **START** are not looked at but they must be less than six symbols in length.
- **ARITHMETIC DECIMAL** and **ARITHMETIC FLOATING** are anything ending in L or G, respectively.
- **READ PROGRAM** is anything ending in M.²
- **STATISTICAL READ** is identified by the letters EAD or ED in sequence followed by a number or variable.²
- **EDIT** is anything ending in IT.²
- **CONVERT** is anything ending with the letters ERT in sequence.²

6. Passing Soap Cards. (Type A Operation Only) Any number of cards written in standard SOAP III coding may be placed just after the Comments Card of a RUNCIBLE input program to be reproduced in the output. They will appear in the place where extensions are normally to be inserted when Scaping (see Operator Instructions, Step II). Column 5 must not contain a 12-punch on these cards.

---

1. Notice that digit f (see clocking) still is arbitrary if we ever find a use for it.
2. See these extensions for use of the words.
APPENDIX IV

CORRECTING ERRORS AFTER COMPLETING COMPILATION. Note: This procedure may be used only with type A operation. If only a small number of statements in a large program have to be changed after compilation, first make a listing of the SOAP III symbolic program. At the very end will be a list of constants terminating with ABC00 (or ABDOO, etc.) — the symbolic locations will be in decreasing order. Take the largest number following the ABC and put it into columns 71-80 of a Header Card which is identical in all other respects with an ordinary Header Card. (The control information in the other columns of this Header Card may have to be changed; this information must apply to the whole program, not just the corrections.)

Now compile in the regular manner the statements which correct or supplement your original output. Put FF on the card of the final statement compiled. The first few cards of your output — identified by 12 punches in columns 76-80 — should now replace the corresponding cards of the original program. Remove also the old symbolic program cards corresponding to the statements you want to replace — they can be located by checking the comments portion of the listing — and insert the new material into its proper place. Each statement begins with the characteristic "statement dictionary entry" which has location address (SO000+n) where n is the statement number. Append any new constants (identified by location address ABC, ABD, ...) at the end of the program. Continue then with the SOAP phase.

Caution: iteration statements become three statements when compiling, one to initialize the variable and the other two (numbered zero) to increment and test it, occurring just after statement n, the end of the iteration scope.

When compiled with clocking, linkages from statement to statement are made with statement numbers, and so the first statement in any block of consecutive correcting statements must have the same number as the first statement it replaces, and the final statement in this block must be followed with a BYPASS statement having the

1. If it is AED, add an extra hundred; ABE, two hundred; and so on.
same number as the statement following the replaced statements. Therefore a single statement may not be merely added into the listing when compiled with clocking; one statement must always be removed (and recompiled in this case) whenever a change is made. Example: suppose it is desired to insert a new statement 20 between old statements 4 and 5. Remove statement 4 from the listing; compile statements 4 and 20 and a dummy statement 5: BYPASS in that order. Put the output from statements 4 and 20 into the place formerly occupied by statement 4.
SUMMARY OF STATEMENTS.

1. BYPASS statement.*
2. Error restarting.*
3. Extension statement.*
4. HALT statement.*
5. Iteration statement.*
6. JUMP statement.*
7. Matrix definition.
8. PUNCH statement.*
9. READ statement.*
10. Substitution statement.*
11. Switch of arithmetic.

* May be made conditional by adding IF expression > expression
+ May be made conditional on sign of
console switches by giving statement
number zero.

BYPASS
SET ERROR CORRECTION TO n

HALT n
n, vl, v2, v3, v4,

JUMP TO (fixed point expression)

PUNCH var1 PUNCH var2 PUNCH var3 PUNCH var4
PUNCH var1 THRU var2

READ
variable ← math. expression

ARITHMETIC DECIMAL
ARITHMETIC FLOATING
Arrange numbers into order, low to high.

Diagram:

1. $i \leq 1$
2. $k \leq 1$
3. $C_i + k \geq C_i$
4. $C_i \leftarrow C_i + k$
5. $k \leftarrow k + 1$
6. $(m \geq i + k)$
7. $M \leftarrow M + 1$
8. $(M > i)$
9. Repeat
10. Done

Note: The diagram illustrates a sorting algorithm.
APPENDIX VI

ADDITIONAL SAMPLE PROGRAMS. (These programs are partly pilfered from the original Carnegie Tech compiler manual; they are chosen for their educational value.)

Example Sub-Program 1. Separate the integral and fractional parts of a floating point variable Y1.

1. \( I_1 \leftarrow Y_1 \)
2. \( C_1 \leftarrow I_1 \)
3. \( C_2 \leftarrow Y_1 - C_1 \)

\( I_1 \) is the integral part of \( Y_1 \) in fixed point form; \( C_1 \) is the same in floating point form; and \( C_2 \) is the fractional (decimal) part of \( Y_1 \) in floating point form.

Example Sub-Program 2. Represent the eight significant digits of a floating point variable \( Y_1 \) as a fixed point integer \( I_1 \).

1. \( \text{JUMP TO 5 IF } Y_1 \geq 1 \text{ B 8} \)
2. \( \text{JUMP TO 7 IF } 1 \text{ B 7} > Y_1 \)
3. \( I_1 \leftarrow Y_1 \)
4. \( \text{HALT} \)
5. \( Y_1 \leftarrow Y_1 / 10. \)
6. \( \text{JUMP TO 1} \)
7. \( Y_1 \leftarrow Y_1 \times 10. \)
8. \( \text{JUMP TO 2} \)

Trickier, shorter, and probably slower, would be

1. \( \text{JUMP TO 5 IF } 1 \text{ B 8} > Y_1 \text{ IF } Y_1 \geq 1 \text{ B 7} \)
2. \( Y_1 \leftarrow Y_1 / 10. \text{ IF } Y_1 > 1 \text{ B 8} \)
3. \( Y_1 \leftarrow Y_1 \times 10. \text{ IF } 1 \text{ B 7} > Y_1 \)
4. \( \text{JUMP TO 1} \)
5. \( I_1 \leftarrow Y_1 \)
6. \( \text{HALT} \)
Example Sub-Program 3. Find the maximum of a set of \( \Pi \) numbers; the numbers are \( Y \)-variables with consecutive subscripts; \( Y_{I_2} \) is the first variable of the group. \( \Pi > 1 \).

\[
\begin{align*}
1 & \quad C_1 \leftarrow Y_{I_2} \\
2 & \quad I_3 \leftarrow I_2 \\
3 & \quad I_\Pi \leftarrow I_2 + I_\Pi - 1 \\
4 & \quad 8, I_4, I_2 + 1, 1, I_\Pi, \\
5 & \quad \text{JUMP TO 8 IF } C_1 \geq Y_{I_4} \\
6 & \quad C_1 \leftarrow Y_{I_4} \\
7 & \quad I_3 \leftarrow I_4 \\
8 & \quad \text{BYPASS} \\
9 & \quad \text{JUMP TO I_5}
\end{align*}
\]

\( C_1 \) is the desired maximum and \( I_3 \) is the subscript of the maximum \( Y \) of the set. Statement 8 was necessary to end each iteration on a common statement. Statement 3 was necessary because the expression \( I_2 + I_\Pi - 1 \) was too long to include in iteration statement 4. Note statement 9 at the end -- a variable JUMP statement enabling the main program to use this sub-program several times and exit to different statements for continuation.

Example Program 5. Generalized matrix multiplication of up to a 20 x 20 matrix. Multiply the \( I_6 \times I_2 \) matrix \( YN_1 \) by the \( I_4 \times I_6 \) matrix \( CN_1 \) obtaining the product elements by the relation

\[
Y_1 = \sum_{I_5=1}^{I_6} CN_1(I_3,I_5) \times YN_1(I_5,I_1)
\]

punching the elements as they are being computed.

\[
\begin{align*}
0 & \quad CN_1, \text{COLS, 20, START, 25,} \\
0 & \quad YN_1, \text{COLS, 20, START, 25,} \\
0 & \quad \text{READ} \\
7 & \quad 4, I_1, 1, 1, I_2, \\
1 & \quad 4, I_3, 1, 1, I_4, \\
6 & \quad Y_1 \leftarrow 0. \\
2 & \quad 3, I_5, 1, 1, I_6, \\
3 & \quad Y_1 \leftarrow Y_1 + CN_1(I_3,I_5) \times YN_1(I_5,I_1) \\
4 & \quad \text{PUNCH } Y_1 \text{ PUNCH } I_3 \text{ PUNCH } I_1 \\
8 & \quad \text{HALT}
\end{align*}
\]

The various outputs that may be obtained with this program are illustrated in Appendix VII. Notice nesting of iteration statements.
LISTINGS OF OUTPUT. On the following pages are shown the input to and output from RUNCIBLE I as it works on Example Problem 3. First the program deck is shown. Due to the shortness of the problem, no attempt was made to keep the number of variables down to a minimum. Word 1 of the header card was made negative for the type Y operation. Statement number 3 would have fit onto a single card but the extra spaces were added for clarity.

<table>
<thead>
<tr>
<th>000000001</th>
<th>000000004</th>
<th>000000045</th>
<th>00000001</th>
<th>00000000325000000000000000000000</th>
</tr>
</thead>
</table>

EXAMPLES MATRIX MULTIPLICATION

| 00001 | CNLK COLSK 20K STARTK 25K F |
| 00001 | YNLK COLSK 20K STARTK 25K F |
| 00001 | READ F |
| 00071 | 4K 11K 1K 1K 12K F |
| 00011 | 4K 13K 1K 1K 14K F |
| 00061 | YL Z OJ F |
| 00021 | 3K 15K 1K 1K 16K F |
| 00061 | YL Z YL S CN1LISK 15R X YN1LIS5 K I1R F |
| 00041 | PUNCH Y1 PUNCH I3 PUNCH II F |
| 00081 | HALT FF |
EXAMPLE 3: MATRIX MULTIPLICATION

BLR 1900 1997 FULL CLOCK
REG 10020 0030
REG Y0031 0481
REG C0482 0932
REG S0933 0942
REG T0943 0952
REG 10000 0030
REG 20000 0030
REG 30000 0030
REG 40000 0030

GO TO NOP 8003
1 S0000 LOD 1 E 011
1 S0000 NOP 8003
1 S0000 LOD 1 E 011
1 S0000 LOD 8003
1 S0000 LOD 1 E 011
1 S0007 LOD 8003
1 S0007 LOD S0007 E 016
1 S0007 LOD ABC00
1 T0007 LOD ABC00
1 T0007 STD I0001
1 S0001 LOD I0003
1 S0001 LOD I0001
1 S0001 E 011
1 T0006 LOD ABC01
1 T0006 STD Y0001
1 S0002 LOD ABC00
1 S0002 STD I0005
1 S0002 E 011
1 S0003 LOD 10005
1 S0003 LOD ABC02
1 T0003 RAID I0003
1 T0003 MPY ABC02
1 T0003 ALO I0001
1 T0003 SLT 0004
1 T0003 RAL C0004

8002 F
### Example 3: Matrix Multiplication

The following code snippet demonstrates matrix multiplication using the same conventions as in previous examples. The register file is assumed to be available with the appropriate register addresses.

#### Code Explanation
- **REG 19001:** This register is used to store the first element of the matrix multiplication.
- **REG Y0020:** This register is used to store the second element of the matrix multiplication.
- **REG C0471:** This register is used to store the third element of the matrix multiplication.
- **REG S0922:** This register is used to store the fourth element of the matrix multiplication.
- **REG T0932:** This register is used to store the result of the matrix multiplication.

#### Register Addresses

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG 19001</td>
<td>0001</td>
<td>9000</td>
</tr>
<tr>
<td>REG Y0020</td>
<td>0002</td>
<td>0470</td>
</tr>
<tr>
<td>REG C0471</td>
<td>0003</td>
<td>0019</td>
</tr>
<tr>
<td>REG S0922</td>
<td>0004</td>
<td>0470</td>
</tr>
<tr>
<td>REG T0932</td>
<td>0005</td>
<td>0931</td>
</tr>
</tbody>
</table>

#### Operations
- **GO TO NOP 8003 1:** This operation is used to return control to the main program.
- **S0000 RAA 1 E 011 CN1K COLSK:** This operation stores a constant to the register.
- **S0000 RAA 1 E 011 YN1K COLSK:** This operation also stores a constant to the register.
- **S0000 RAA 1 E 011 READ:** This operation reads from the register.
- **S0007 LOD S0007 E 016:** This operation loads an address into the register.
- **T0007 LOD ABC00:** This operation loads data into the register.
- **S0001 RAA 1 E 011 I 1 Z:** This operation performs an arithmetic operation.
- **T0001 LOD ABC00 STD I0001 S0001:** This operation loads and stores data between registers.
- **S0006 RAA 1 E 011 Y1 Z 0J:** This operation also performs an arithmetic operation.
- **T0002 LOD ABC00 STD I0005 S0003:** This operation loads and stores data between registers.
- **S0003 RAA 1 E 011 Y1 Z Y1 S:** This operation performs another arithmetic operation.
- **T0003 RAA I0005:** This operation also performs an arithmetic operation.

The remaining operations continue to load and store data, perform arithmetic operations, and handle input/output operations as necessary.
S0000 RAA 1 E 011 15 Z L 15 R S
1 RAL 10005 1F
RAL 10000
ALO ABC00
STL 10005
S0001 RAA 1 E 011 G0 T0 0003 IF L16
1 R S 47
RSL 10005
ALO 10006
BMI S0004
S0002 RAA 1 E 011 PUNCH Y1 P
1 R S 48
LOD ABC03
STD P0000
LOD ABC04
STD P0001
LOD ABC05
STD P0002
S0003 RAA 1 E 011 PUNCH 13 PU
1 R S 49
LOD ABC06
S0004 RAA 1 E 017 13 Z L 13 R S
1 R S 50
LOD ABC03
STD P0000
LOD ABC04
STD P0001
STL 10003
S0005 RAA 1 E 011 G0 T0 0006 IF L14
1 R S 51
RAL 10003
RAL 10004
ALO ABC00
BMI 1F
S0006 RAA 1 E 011 G0 T0 0006 IF L14
1 R S 52
RAL 10001
RAL 10000
ALO ABC00
STL 10001
S0007 RAA 1 E 011 G0 T0 0006 IF L12
1 R S 53
RAL 10001
RAL 10000
ALO ABC00
BMI S0008
S0008 RAA 1 E 011 HALT
1 R S 54
T0008 HLT 8003 8778
1 R S 55
ABC06 00 0003 0004
ABC05 00 0002 0001
ABC04 00 0001 0003
ABC03 00 0001 0001
ABC02 00 0000 0020
ABC01 00 0000 0000
ABC00 00 0000 0001

61
### Compiler III - SOAP III

#### Board Wiring

**CONVENTION:** All hubs (except CONTROL INFORMATION) are numbered from left to right, e.g.:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

- **READ CARD B**, Cols. 1-10 to **STORAGE ENTRY B**, Wd 1, Pos. 1-10
  - 11-20
  - 21-30
  - 31-40
  - 41-50
  - 51-60
  - 61-70
  - 71-80

- **JACKPLUG:**
  - DI-C on DIGIT SELECTOR READ (left hand side)
  - DI-C on DIGIT SELECTOR PUNCH (left hand side)
  - RSU
  - BG-OFF (if DPBC device is not used)

**BUS TOGETHER:** **FIRST READING**, Cols. 2-7, **HEAD COLUMN SPLIT 1**, 12-X and wire to **LOAD**.

- **FIRST READING**, Cols. 4 to **READ COLUMN SPLIT 2**, C
  - 5
  - 41
  - 42
  - 43-47
  - 48-50
  - 51-55
  - 56
  - 57-61
  - 62
  - 63-67
  - 68-72

- **ENTRY B**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

- **READ CARD C**, Cols. 1-4 to **PILOT SELECTOR 3, Upper T**
  - 41
  - 42
  - 43-47
  - 48-50
  - 51
  - 52-55
  - 56
  - 57
  - 58-61
  - 62
  - 63-67
  - 68-72

- **COSELECTOR 3, T**, Pos. 1-4
- **PILOT SELECTOR 1**, Upper T
- **STORAGE ENTRY C**, Wd 1, Pos. 6
- **COSELECTOR 3, N**, Pos. 1-4
- **STORAGE ENTRY C**, Wd 1, Pos. 7-10
- **STORAGE ENTRY C**, Wd 4, Pos. 6-8
- **STORAGE ENTRY C**, Wd 4, Pos. 7-10
- **STORAGE ENTRY C**, Wd 4, Pos. 8-10
- **STORAGE ENTRY C**, Wd 5, Pos. 6-10
- **STORAGE ENTRY C**, Wd 6, Pos. 6-10
STORAGE ENTRY C, Wd 10, Pos. 4 to COSELECTOR 3, C, Pos. 5
  " " " " Wd 10, Pos. 5 " "
  " " " " Wd 10, Pos. 6 " "
  " " " " Wd 10, Pos. 10 " "
  " " " " Wd 7, Pos. 7-10 " 

PILOT SELECTOR 2, Lower C
  " 1, Lower C "
  " 1, Upper C "

COSELECTOR 3, C, Pos. 1-4

CAI to PILOT SELECTOR 4, Upper N

ALPHA IN, Wd 1 to PILOT SELECTOR 4, Upper C

JACKPLUG: On ALPH IN, W1-W2; W2-W3; W3-W4; W4-W5; W5-W6

WORD SIZE 10 for B, Wds 1-8 C, Wds 1-6
  " 0 for B, Wd 9 to Wd 10 "
  " 4 for C, Wds 7-9 "
  " 7 for C, Wd 10 "

COSELECTOR 4, T, Pos. 4 to COSELECTOR PICKUP 6
  " " " " 5 " "
  " " " " C " 3 "
  " " " " C " 4 "
  " " " " C " 5 "

PILOT SELECTOR 3, I PU
  " 4, I PU § X PU "
  " 5, I PU "
  " 6, I PU "
  " 7, I PU "
  " 8, I PU "
  " 9, I PU "
  " 10, I PU "

Control INFORMATION 3(numbered from Rt to Lt.)
PUNCH DELAY 2, OUT

COSELECTOR PICKUP 3

READ COLUMN SPLIT 2, 12-X

COSELECTOR PICKUP 5

CONTROL INFORMATION 2

PUNCH B

Emit READ DIGIT 2 to PILOT SELECTOR 1, Upper N

PILOT SELECTOR 5, Upper N to PILOT SELECTOR 5, Lower N &
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " " " " 7, Upper C "
  " 5, Lower C "
  " 6, Lower C "
  " 10, Lower T "
  " Lower N "
  " Lower N "

PUNCH CARD A, Cols. 78-80
PUNCH COLUMN SPLIT 4, 0-9 &
STORAGE EXIT B, Wd 1, Pos. 10
PUNCH COLUMN SPLIT 4, 12-X
COSELECTOR 7, C, Pos. 2
PUNCH COLUMN SPLIT 7, 12-X
PUNCH SELECTOR 10, Lower C & PSU
ALPH OUT, Wd 2
ALPH OUT, Wd 1
P+
PUNCH CARD A, Cols. 75-77
PUNCH COLUMN SPLIT 6, 12-X
PSU
ALPH OUT, Wd 6
COSELECTOR PICKUP 4  
" " " 5  
" " " 6  
" " " 7  
" " " 8  

PILOT SELECTOR HOLD 1  
" " " 5  

READ HOLD TO PILOT SELECTOR HOLD 4  
PUNCH HOLD TO PILOT  " " 10  

JACKPLUG: PILOT SELECTOR HOLD, 1-2; 2-3; 3-4; 5-6; 7-8; 8-9; 9-10;  
COSELECTOR HOLD, 4-5; 5-6; 6-7; 7-8  

COSELECTOR HOLD 2 to COSELECTOR HOLD 9  

PUNCH CARD A, Cols. 1-10  
" " " Cols. 17-20  
42  
" " " 43-47  
" " " 48-50  
" " " 51-55  
" " " 56  
" " " 57-61  
" " " 62  
" " " 63-67  
" " " 68-72  
" " " 74  

STORAGE EXIT A, Wd 4, Pos. 9  
" " " Wd 4, Pos. 10  

PUNCH CARD B, Cols. 1  
" " " 2  
" " " 3  
" " " 4  
" " " 5-6  
" " " 7  
" " " 8-10  
" " " 11-20  
" " " 21-30  
" " " 31-40  
" " " 42-50  
" " " 51-60  
" " " 61-70  
" " " 71-80  

STORAGE EXIT B, Wd 1, Pos. 1  
" " " Wd 1, Pos. 4  
" " " Wd 1, Pos. 7-9  
" " " Wd 5, Pos. 1  
" " " Wd 9, Pos. 7-10  

PUNCH COLUMN SPLIT 5, C  
STORAGE EXIT B, Wd 1, Pos. 2  
" " " Wd 1, Pos. 3  
PUNCH COLUMN SPLIT 6, C  
STORAGE EXIT B, Wd 1, Pos. 5-6  
PUNCH COLUMN SPLIT 7, C  
COSELECTOR 5, C, Pos. 3-5  
PUNCH COLUMN SPLIT 5, C  
STORAGE EXIT B, Wd 2, Pos. 1-10  
" " " Wd 3, Pos. 1-10  
" " " Wd 4, Pos. 1-10  
" " " Wd 5, Pos. 2-10  
" " " Wd 6, Pos. 1-10  
" " " Wd 7, Pos. 1-10  
" " " Wd 8, Pos. 1-10  

PUNCH COLUMN SPLIT 8, 0-9  
" " " 6, 0-9  
COSELECTOR 5, N, Pos. 2-4  
PUNCH COLUMN SPLIT 5, 0-9  
COSELECTOR 5, T, Pos. 2-5  

CONTROL INFORMATION 5  
" " " 7  
" " " 4  
" " " 1  
" " " 6
COSELECTOR 5, N, Pos. 5 to
  " " C, Pos. 2
  " 6, T, Pos. 1
  " " T, Pos. 5
  " " N, Pos. 5
  " " C, Pos. 2-5
  " 8, C, Pos. 3
  " " C, Pos. 4

PUNCH CARD C, Cols. 1
  " " " " 17-20
  " " " " 23-26
  " " " " 31-39
  " " " " 40
  " " " " 43-47
  " " " " 48-50
  " " " " 51-55
  " " " " 56
  " " " " 57-61
  " " " " 62
  " " " " 63-67
  " " " " 68-72

STORAGE EXIT C, Wd 7, Pos 10

PUNCH COLUMN SPLIT 4, C
  " " " " 7, 0-9
PUNCH CARD C, Cols. 74
PUNCH COLUMN SPLIT 1, C
  " " " 2, C
PUNCH CARD C, Cols. 7-10
PUNCH COLUMN SPLIT 5, 12-X
  " " " 10, 12-X
PUNCH COLUMN SPLIT 3, C
STORAGE EXIT C, Wd 9, Pos. 7-10
  " " " Wd 8, Pos. 3-6
  " " " Wd 7, Pos. 1-9
PUNCH COLUMN SPLIT 10, C
STORAGE EXIT C, Wd 1, Pos. 6-10
  " " " Wd 4, Pos. 6-8
  " " " Wd 2, Pos. 6-10
  " " " Wd 4, Pos. 9
  " " " Wd 3, Pos. 6-10
  " " " Wd 4, Pos. 10
  " " " Wd 5, Pos. 6-10
  " " " Wd 6, Pos. 6-10
PUNCH COLUMN SPLIT 10, 0-9

PUNCH DELAY 1, IN to CONTROL INFORMATION 3
PUNCH A to CONTROL INFORMATION 10
PUNCH B to CONTROL INFORMATION 8

JACKPLUG: ALPH OUT, W2-W3; W3-W4; W4-W5; W5-W6

EMIT READ-TIMED DIGITS:

8 in COSELECTOR 3, T, Pos. 5
8 in PILOT SELECTOR 1 4 2, Lower T
9 in COSELECTOR 3, N, Pos. 5
9 in PILOT SELECTOR 1 4 2, Lower N
X in PILOT SELECTOR 4, Lower C
0 in STORAGE ENTRY C, Wd 10, Pos. 7-9

EMIT PUNCH-TIMED DIGITS:

Y in COSELECTOR 4, T, Pos. 3
Y in PILOT SELECTOR 5, 7 4 8, Upper T
Y in " " 6, Lower T
Y in COSELECTOR 7, T, Pos. 2
Y in " 8, T, Pos. 3
Y in " " N, Pos. 4
Y in PUNCH COLUMN SPLIT 1, 2 5, 12-X
X in PILOT SELECTOR 6, Upper T
X in COSELECTOR 8, T, Pos. 4 5
X in " 6, C, Pos. 1
9 in PUNCH CARD A, Cols. 73
9 in COSELECTOR 7, T, Pos. 3
In PUNCH CARD A, Cols. 2-9 emit the following punch-timed digits—00000800
In PUNCH CARD C, 2-6 --91954
21-22 --24
27-29 --800

In COSELECTOR 6, T, Pos. 2-4 emit the following punch-timed digits—800
N, Pos. 2-4 --195
PUNCH COLUMN SPLIT 1, 2 3, (0-9) --036
INDEX

A Operation .................................. 28
Absolute Value .............................. 7
Addition ..................................... 5
Arithmetic .................................. 7
Arithmetic switching ....................... 18ff

B Operation .................................. 28
B, Use of ..................................... 5
Basic Packages ............................... 17ff, 43
Binary Operations ......................... 5ff
Buyansky, D. V ............................... 2
BYPASS Statement ........................... 13

C- Variables ................................. 4
Card Preparation ........................... 18ff
Cheating ..................................... 17
Clocking ..................................... 27ff, 44ff
Comma ........................................ 14
Comments Card .............................. 26ff
Conditional Punch ........................... 12
Conditions ................................... 13ff, 53
Constants .................................... 5
Core, Variables in ........................... 48

D .................................................. 22
Data Cards ................................... 18ff
Decimal Point ............................... 5, 22
Division ....................................... 5
Division, Fixed Point ....................... 8ff

Equals ........................................ 13
Error Correction ............................. 39ff, 51ff
Error Search Operation ..................... 28ff, 60ff
Error Sense Running ......................... 49ff
Example Program 1 ......................... 15ff, 23
Example Program 2 ......................... 16ff
Example Program 3 .......................... 55ff
Example Subprograms ........................ 54ff
Exponentiation ................................ 5
Extension Statements ........................ 13
Extensions .................................... 10, 13, 28

F .................................................. 22
FF .............................................. 22
Fixed Point Constants ........................ 5
Fixed Point Division ........................ 8ff
Fixed Point Numbers ........................ 3
Fixed Point VALUE ........................... 19
Fixed Point Variables ........................ 4

Floating Point Attachment .................. 28
Floating Point Constants .................... 5
Floating Point Numbers ...................... 3
Floating Point VALUE ......................... 19
Floating Point Variables .................... 4
Flow Chart .................................... 15
Formats ....................................... 18ff

Garbage Cards ................................ 30
Greater Than .................................. 13
Greater Than Or Equal ....................... 15

HALT Statements ............................. 12
Haynam, G. E ................................. 2
Header Card ................................... 25ff

I- Variables ................................... 4
IDENT ......................................... 18ff
Input Statement Cards ........................ 22ff
Intermixed Fixed and Floating .............. 7, 11, 13
IT-Language ................................... 3ff
Iteration Statements ........................ 1ff

J .................................................. 22
JUMP Statement ................................ 11

K .................................................. 22
Klam ............................................ 29
Knuth, D. E .................................... 2

L .................................................. 22
Listings ....................................... 55ff
Lynch, W. C ...................................... 2

M .................................................. 22
Machine Operating ............................ 29ff
Mathematical Operations ...................... 5ff
Matrix Notation ................................ 46ff, 55
Modes of Operation ........................... 27ff
Multipass ...................................... 28
Multiplication ................................ 5

Negation ....................................... 7
Numbers ....................................... 3

Operation Modes .............................. 27ff
Operations .................................... 5ff
Operator Instructions ......................... 29ff
Output Cards .................................. 21
Output Statements ............................. 12, 21
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P, Use of</td>
<td>5, 18</td>
</tr>
<tr>
<td>Parentheses</td>
<td>5ff</td>
</tr>
<tr>
<td>Parentheses, Importance of</td>
<td>6ff</td>
</tr>
<tr>
<td>Perlis, Dr. A. J.</td>
<td>1</td>
</tr>
<tr>
<td>Petznick, G. W., Jr.</td>
<td>2</td>
</tr>
<tr>
<td>Polynomial Evaluation</td>
<td>17</td>
</tr>
<tr>
<td>Priority of Binary Operators</td>
<td>6</td>
</tr>
<tr>
<td>Psychiatrist</td>
<td>15</td>
</tr>
<tr>
<td>PUNCH Statements</td>
<td>12, 21</td>
</tr>
<tr>
<td>P1</td>
<td>18, 25, 43</td>
</tr>
<tr>
<td>P2</td>
<td>18, 25, 43</td>
</tr>
<tr>
<td>P3</td>
<td>18, 25, 43</td>
</tr>
<tr>
<td>Q</td>
<td>22</td>
</tr>
<tr>
<td>Quotation Marks</td>
<td>10</td>
</tr>
<tr>
<td>R</td>
<td>22</td>
</tr>
<tr>
<td>Raising to Power</td>
<td>5</td>
</tr>
<tr>
<td>READ Statement</td>
<td>11f, 20</td>
</tr>
<tr>
<td>Relations</td>
<td>13f, 53</td>
</tr>
<tr>
<td>Remainder</td>
<td>48</td>
</tr>
<tr>
<td>Rerun Procedures</td>
<td>38</td>
</tr>
<tr>
<td>Runcible O</td>
<td>2</td>
</tr>
<tr>
<td>Scope of Binary Operators</td>
<td>6</td>
</tr>
<tr>
<td>Sine</td>
<td>10</td>
</tr>
<tr>
<td>Single Pass Operation</td>
<td>28</td>
</tr>
<tr>
<td>Smith, J. W.</td>
<td>1</td>
</tr>
<tr>
<td>Spelling Liberties</td>
<td>50</td>
</tr>
<tr>
<td>Square Root</td>
<td>10</td>
</tr>
<tr>
<td>Statement Numbers</td>
<td>10, 12, 22</td>
</tr>
<tr>
<td>Statements</td>
<td>10ff, 22f</td>
</tr>
<tr>
<td>Statements on Cards</td>
<td>22f</td>
</tr>
<tr>
<td>Stops</td>
<td>37ff</td>
</tr>
<tr>
<td>Subscripts</td>
<td>4</td>
</tr>
<tr>
<td>Substitution Statement</td>
<td>11</td>
</tr>
<tr>
<td>Subtraction</td>
<td>5</td>
</tr>
<tr>
<td>Summary of Statements</td>
<td>53</td>
</tr>
<tr>
<td>Switching Arithmetic</td>
<td>48f</td>
</tr>
<tr>
<td>Symbols, Code for</td>
<td>22</td>
</tr>
<tr>
<td>THRU</td>
<td>12, 21</td>
</tr>
<tr>
<td>Tracing</td>
<td>4ff</td>
</tr>
<tr>
<td>U</td>
<td>22</td>
</tr>
<tr>
<td>Unary Operators</td>
<td>7</td>
</tr>
<tr>
<td>V</td>
<td>22</td>
</tr>
<tr>
<td>Value</td>
<td>18f</td>
</tr>
<tr>
<td>Van Zoeren, H. R.</td>
<td>1</td>
</tr>
<tr>
<td>Variable Subscripts</td>
<td>4</td>
</tr>
<tr>
<td>Variables</td>
<td>4</td>
</tr>
<tr>
<td>Variables in Core Storage</td>
<td>48</td>
</tr>
</tbody>
</table>

66
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>22</td>
</tr>
<tr>
<td>Way III, F</td>
<td>2</td>
</tr>
<tr>
<td>Wolontis Function</td>
<td>10</td>
</tr>
<tr>
<td>X and Y Operation</td>
<td>28</td>
</tr>
<tr>
<td>Y- Variables</td>
<td>4</td>
</tr>
<tr>
<td>Zero Statements</td>
<td>10</td>
</tr>
<tr>
<td>1234 Stops</td>
<td>37, 39ff</td>
</tr>
</tbody>
</table>