BRUIN

BROWN UNIVERSITY INTERPRETER

for the

CAMBRIDGE MONITOR SYSTEM

CMS

written by

The Computing Laboratory
Brown University
Providence, Rhode Island

adapted to CMS by

IBM Cambridge Scientific Center
CP-67/CMS Support Group
Cambridge, Massachusetts

June 1, 1969

TABLE OF CONTENTS

INTRODUCTION

This manual describes the facilities of the Brown University Interpreter (BRUIN) as it is implemented within the Cambridge Monitor System.

The BRUIN language is modeled after the University of Pittsburgh Interpretive Language (PIL). PIL was chosen as a model because it contains features which are highly desirable in a console language. Unlike a batch processing language, BRUIN (and PIL) provide the user with direct man-machine interaction capabilities. This feature is most noticeable in error-recovery procedures. After BRUIN reports an error in either syntax or program logic, it allows the user to make corrections and continue without a new start. Another commendable BRUIN feature is the lack of processor-oriented statements such as array dimensions and variable types.

The major goal of the BRUIN designers is that BRUIN be a language which will allow a neophyte as well as an experienced programmer to expend his energies on problem-solving rather than on "programming". Towards this goal the syntax has been kept simple, the meaning of each statement clear and unambiguous. BRUIN can best be utilized as either a program development tool or as a small numeric problem-solving tool.

SYNTAX NOTATION

Throughout this manual, wherever a BRUIN statement is described that statement will be illustrated with a uniform system of notation. The keywords (special BRUIN words) will be in upper case. This is not a requirement of the language; it is merely a device to denote the literal occurrence of that word. Keywords, in fact, may appear in combinations of upper-case and lower-case letters. Braces (<>) are not an element of the language but are used to denote a position in a statement where the user must make a replacement.

Every BRUIN statement begins with a keyword or part and step number. (See the section "Direct and Indirect Mode", page 19, for a description of the form of the the part and step number.) A statement is at most 80 characters in length. Each statement begins on a new line and cannot be continued on a second line. A statement may, but need not end with a period.

Keywords, part-step numbers and user-defined elements (such as variables) must not be immediately adjacent to one another. They must be separated by a single symbol operator (e.g. + / *), specified punctuation (, : ;), an assignment symbol (=) or a blank. There must not be blank spaces within keywords, part numbers and user-defined elements.

Example 1. The syntax of a form of the conditional statement is

If <Boolean expression>, THEN <command>; ELSE <command>.

A correct BRUIN command then could be

IF a = b, Then TO STep 1.5;ELse to step 1.6.

but not

IFa=b, Then TOSTep 1.5;ELse to step 1.6

The above statement is in error because there is not a blank between the F in the keyword IF and the variable a, and between the keywords TO and Step. Notice that the keywords IF, THEN and ELSE appear in a combination of upper-case and lower-case letters.

also not

```
I F a=5.  11,THEN TO STEP 1.5;ELSE TO STEP 1.6.
```

This statement is incorrect because there is a blank in the keyword IF and in the constant 5.11.

SYSTEM PROCEDURES

Entering BRUIN

After logging into CP and IPL'ing CMS, type "BRUIN". The console will respond with the message "BRUIN READY".

Exiting from BRUIN

When you are finished using BRUIN, you should dismiss the BRUIN interpreter with the BRUIN command:

```
CANCEL
```

CANCEL does not log you off. In order to log off from the terminal, thereby preventing a second party from accessing the system under your number and password, type:

```
CP LOGOUT
```

To use the terminal system again, you must repeat the CP LOGIN procedure.

INPUT CONVENTIONS

CMS input conventions are used for the character delete and line delete functions.

However, input characters are not converted to upper case.

DATA ELEMENTS

Constants

There are three types of constants: real arithmetic constant, Boolean constant and character string constant.

An arithmetic constant is written with at most 7 decimal digits with an optional decimal point. The constant may be immediately followed by the letter E (must be upper case) and a one or two digit signed or unsigned integer specifying an integral power of ten (scientific notation). In either form of the constant there must not be embedded blanks. Some examples of proper arithmetic constants appear below:

12.34567	
3567 or 3567.	
56.5E-5	meaning 56.5×10^{-5}
125E4	meaning 125×10^{4}

A constant (and results of all operations) can have a maximum magnitude of 7×10^{75} and a minimum non-zero magnitude of 5.4×10^{-79}.

The Boolean constants are THE TRUE and THE FALSE witten in upper or lower case letters.

Character string constants consist of a string of characters enclosed in double quotes. The double quote character may not occur in a string constant. Some proper character string constants are: "$123.5" and "BRUIN". The maximum length of a character string constant is limited by the size of the line.

Variables

The name of a variable must be constructed according to the following rules:

1. The total number of characters must not exceed eight (8).
2. The first character must be one of the alphabetic characters.
3. The remaining characters may be letters or numerals.

Upper-case letters are distinguished from lower-case letters in variable names; therefore the variable name DET is different from the name Det.

Example 2. The following names are valid BRUIN variable names:

DATANAME, B, FXY, B12C11

In addition to the scalar variable which denotes one item, there is the subscripted variable which denotes a collection of items. A subscripted variable is written as a variable name followed by a list of subscripts enclosed in parentheses. The subscripts are separated by commas. For example, F1(1,2) could represent the first row, second column of a table called F1.

The rules for subscripted variables are summarized below:

1. Each subscript may be an arithmetic constant, arithmetic variable, or an arithmetic expression.
2. An array may have up to four subscripts.
3. A subscript may have any value between -32767 and 32767, but only the integer part is used to reference an element of the array.

Example 3.

z(i,j,2*j), TABLE(Z(I,J,J),I+J)

In Example 3, assume i is 1.7 and j is 2.4; because only the integer part of the subscript is used, the element referred to in the z array is z(1,2,4).

Storage is allocated dynamically for variables (both scalar variables and arrays) at the time that the variable is defined (i.e., assigned a value). Therefore, sparse arrays are kept in a reduced space. In no BRUIN statement is is possible to reference the whole array by using only the name; both name and subscript must be specified.

To use a variable which has no associated value is an error. The interpreter will notify the user that this type of error has occurred.

In addition to having a value, a variable has an associated data type. The data type may be arithmetic, Boolean, or character string, depending on the type of value assigned to it. To change the data type of a variable by assigning to it a value of a different data type is an error. The interpreter will notify the user that this sort of error has occurred.

OPERATORS AND EXPRESSIONS

Arithmetic Expressions and Functions

The arithmetic operators of addition, subtraction, multiplication, division and exponentiation are represented symbolically by +, -, *, /, and ** respectively. (The vertical bar, |, also indicates exponentiation.)

	meaning	example
+	addition	a+b
-	subtraction	a-b
*	multiplication	a*b
/	division	a/b
**	exponentiation	a**b

Table 1. Arithmetic Operators

Arithmetic constants and variables are combined by the arithmetic operators to form arithmetic expressions. Arithmetic expressions, in turn, can be bracketed by parentheses to form other arithmetic expressions. The order in which expressions are evaluated is governed by the following rules of precedence with operations of a higher precedence performed before those of a lower precedence. Except for exponentiation, unary- and unary+, where operators of the same priority appear in an expression, there is a left-to-right evaluation of the expression. Therefore A/B/C produces the same result as (A/B)/C. If two or more of the operators of exponentiation, unary-, unary+ appear in an arithmetic expression, the order of evaluation is from right to left. Thus A**-2 produces the same result as A**(-2).

<table>
<tr><td></td><td></td><td><u>example</u></td></tr>
<tr><td>highest</td><td>functions</td><td>SQRT(x)</td></tr>
<tr><td></td><td>exponentiation,
unary -, unary +</td><td>A|B|-2</td></tr>
<tr><td></td><td>multiplication
and division</td><td>a*N/y</td></tr>
<tr><td>lowest</td><td>addition and
subtraction</td><td>z + p - 1</td></tr>
</table>

If an expression is enclosed in parentheses, it is evaluated before its associated operation is performed. For example, in the expression c*(a + b), a is added to b and this sum is multiplied by c. Thus parentheses modify the normal precedence rules. Parentheses can be used where there is a possibility of ambiguity.

Another way that arithmetic expressions can be formed is by taking functions of other arithmetic expressions. The set of predefined arithmetic functions is shown in Table 2 of Appendix B. The function name is fixed but may be written with upper-case or lower-case letters. Each of the functions has one argument and returns one value. The argument must be enclosed in parentheses and may be an arithmetic expression. The expression is evaluated before the function is called. Thus, if it is necessary to compute sin 2x, the BRUIN expression would be written SIN (2*x)

Examples:

<table>
<tr><th><u>Expression</u></th><th><u>Mathematical equivalent</u></th></tr>
<tr><td>c+a**8/b+5.E10</td><td>$c + a^8/b + 5x10^{10}$</td></tr>
<tr><td>TAN(LN(a*c))|2</td><td>$\tan^2(\ln(ac))$</td></tr>
<tr><td>-2**2*b</td><td>-2^2b</td></tr>
<tr><td>(-2)**2*b</td><td>$(-2)^2b$</td></tr>
</table>

Boolean Expressions

There are six relational operators and four Boolean operators defined in BRUIN. The symbolic representations of these operators and their definitions are found in Table 2 and Table 3.

short form	long form	meaning	example
<	$LT	a less than b	a < b
	$LE	a less than or equal to b	a $LE b
=	$EQ	a equal b	a = b
	$NE	a not equal b	a $NE b
>	$GT	a greater than b	a $GT b
	$GE	a greater than or equal to b	a $GE b

a and b are arithmetic expressions

Table 2. Relational Operators

The relational operators have two arithmetic expressions as operands; the result of such operations is a Boolean value (i.e., either true or false.) Thus, the expression B**2 - 4*A*C $GT 0 is an assertion that is either or false, or in BRUIN notation the result will be either THE TRUE or THE FALSE.

Boolean values or expressions may be combined with the Boolean operators or the two relational operators $EQ, $NE to produce a Boolean value.

short form	long form	meaning	example
&	$AND	logical product	a<b $AND c<d
	$OR	logical sum or inclusive or	a<b $OR c<d
¬	$NOT	complement	$NOT (a<b)
	$XOR	exclusive or	a<b $OR c<d

a, b, c and d are arithmetic expressions

Table 3. Boolean Operators

The Boolean operators (sometimes called logical operators) have their usual meaning. Thus, if P and Q are Boolean values the expression P $AND Q has the value THE TRUE, if and only if both P and Q have the value THE TRUE. The expression P $OR Q has the value THE TRUE if either P or Q or both P and Q have the value THE TRUE. Whereas, P $XOR Q has the value THE TRUE if either P or Q but <u>not</u> both have the value THE TRUE. Finally, $NOT P has the value THE TRUE if and only if P has the value THE FALSE.

The priorities of operations for arithmetic, Boolean and relational operations are now given by the following list:

highest	arithmetic function
	**, unary +, unary -, $NOT
	* /
	+ -
	relational operations
	$AND
lowest	$OR, $XOR

The priorities for arithmetic, Boolean and relational operators is similar to the priority of such operators in the PL/1 language but not the FORTRAN language. The one difference is in the operator $NOT. This difference becomes apparent in the expression $NOT a<b, where a and b have arithmetic values. Since $NOT has higher priority than the operator <, an attempt to evaluate $NOT a in this expression will be made. BRUIN will then stop with the error message "WRONG KIND OF OPERAND". Simply enclosing the expression a<b in parentheses, as in Table 3, will result in the Boolean expression a<b being the operand of $NOT.

The following example illustrates the evaluation of an expression according to the priority of the operators.

```
-a**b < c+2     $AND a+c=z
```

This expression is evaluated as if the various parts were enclosed in parentheses.

```
-(a**b)  (c+2)              (a+c)  (z)

(-(a**b))<(c+2)             (a+c)=(z)

((-(a**b))<(c+2))           $AND((a+c)=(z))
```

Character String Expressions and Functions

There are three special operators $FC, $LC and $CON for manipulating character string data. An expression of the two operators $FC and $LC is evaluated from right to left. In addition to these special operators the relational operators are valid for comparing character string data. The comparison will be made according to the 360 collating sequence. (i.e. blank <punctuation < a < b...<z<A<B...<Z<0.<9). the strings are compared character by character from left to right. If the strings are of different lengths the shorter string will be compared as if it were extended on the right with blanks. Notice that this implies that the character string "ab" will compare equal to the character string "ab ". Table 4 summarizes the valid character string operators. The concatenation operator, $CON, may be written _ (underline symbol) or || (two vertical bars).

<table>
<tr><th>Examples</th><th>Definition</th><th>Result</th></tr>
<tr><td>N $FC S</td><td>The N first characters of string S</td><td>a string of length N</td></tr>
<tr><td>N $LC S</td><td>The N last characters of string S</td><td>a string of length N</td></tr>
<tr><td>S $CON P
or
S||P</td><td>string S and P are joined in such a way that the first character of P immediately follows the last character of S</td><td>a string with the length = length S + length P</td></tr>
<tr><td>S $LT P
S $LE P
S $EQ P
S $NE P
S $GT P
S $GE P</td><td>compare S and P character by character from left to right</td><td>a Boolean value</td></tr>
<tr><td colspan="3">N has an arithmetic value
S and P are character strings.</td></tr>
</table>

Table 4. Character String Operators

There are four special functions which have a character string argument. They are UPPR, LOWR, LEN, AND CHAR. The function UPPR(S) produces a character string with alphabetic characters the upper case equivalent of the alphabetic characters of S. LOWR(S) produces a character string with alphabetic characters the lower case equivalent of the alphabetic characters of S. LEN(S) returns an arithmetic value which is the number of characters in string S. CHAR(X) returns the character representation of the arithmetic value X.

In a mixed expression of string operators, binary operators and Boolean operators the usual rules of priority as well as the conventions regarding left to right evaluation are observed. The string operators are placed in the priority table as follows:

highest	functions
	** unary + unary - $NOT
	*/
	+-
	$FC $LC
	$CON
	$EQ $GE $LE $NE $GT $LT $EQ
	$AND
lowest	$OR $XOR

The following example will serve to illustrate the use of the string operators and the priority rules for string operators:

<u>Example:</u>

Assume that D is a character string variable which is equal to "PROVIDENCE, R.I." and one wishes to make every letter except P in Providence a lower case letter. This could be done with the expression

1 $FC D $CON LOWR(9 $LC 10 $FC D)$CON 6 $LC D

This expression results in a character string of length 16 which is equivalent to "Providence, R.I.".

In building character strings by concatenation one must observe a maximum length of 252. If this length is exceeded BRUIN will issue an error message to that effect.

DIRECT AND INDIRECT MODE

To facilitate man-machine interaction, BRUIN provides two modes of operation, the desk calculator or direct mode and the stored program or indirect mode.

In the direct mode the console can be thought of as a desk calculator in that the statement is executed immediately and the text of the statement is not retained in memory. This mode of operation allows the user to evaluate expressions, store results of expressions for later use, and direct the interpreter to execute a stored program.

Errors are reported immediately in the direct mode. The user merely retypes the correct statement. The statement in error is not retained. For example,

```
TYPE SIN(8*3.14/16
```

would result in a BRUIN response "MISFORMED EXPRESSION" because of the missing right parenthesis. The user retypes

```
TYPE SIN(8*3.14)/16
```

and the result of the calculation would appear as

```
SIN(8*3.14)/16 = -0.7966093E-03
```

The assignment statement in the direct mode

```
SET L=A|2-4*A*C
```

is likewise executed immediately, causing the variable L to be given a value which could be used either in the direct mode or stored program mode. Notice here that the statement itself is not retained but the result of the calculation is retained in L.

In the indirect mode, statements are stored and executed under program control in a sequence defined by part and step numbers. In other words, indirect statements make up a stored program. The user may go back and forth between the two modes. The manner by which this is accomplished will be seen in later examples.

A statement in the indirect mode is always preceded by a part and step number. It is this number which tells BRUIN that this is a statement of the indirect mode. The part and step number are each at most 3 digits in length and are separated by a period. A blank must follow this number but must not be imbedded in the part-step number. The above TYPE statement, written in the indirect mode with part number 11, step number 01, would appear as

```
11.01 TYPE SIN(8*3.14)/16
```

This statement is stored without being executed immediately.

A part is a collection of one or more steps with the identical part number. Since steps are arranged in ascending step number by the interpreter, it is not necessary to type steps in sequential order. During the execution phase of that part, the statements will be executed in the sorted order of step numbers in that part.

Step numbers are treated as decimal fractions and may have any increment between them. For example, a part 1 could be written with steps 0, 1 and 999 (written with part numbers as 1.0, 1.1 and 1.999). This would be executed by the interpreter in the order 1.0, 1.1 and 1.999. If it is necessary to make an insertion, say between statements 1.0 and 1.1, simply choose any number (e.g 1.05) between 1.0 and 1.1. Then part 1 would consist of statements with numbers 1.0, 1.05, 1.1, 1.999 and would be executed in that order. For such program changes it is wise to have gaps in the step sequence.

During the execution of a program in the stored program mode, a sequence of BRUIN statements with the same part number are executed sequentially (assuming there is no transfer out of that part). This means that a statement with the number 2.000 is not executed after the statement with the number 1.999, unless these parts are logically connected by the user. This can be accomplished in a variety of ways. To give just one example, statement 1.999 could be the instruction

```
1.999 TO STEP 2.000
```

ASSIGNMENT STATEMENT

SET statement

SET <variable name> = <expression>,<variable name> = <expression>,etc.

The expression on the right-hand side of the assignment symbol (=) may be an arithmetic expression, Boolean expression or string expression. The type of expression will determine the data type assigned to the variable. The keyword SET is optional and may be omitted; for clarity, every example in the manual will use the keyword.

The SET statement can have a list of assignments of the form <variable name> = <expression>; each assignment in the list must be separated by commas. If there is only one assignment in the statement, omit the comma. The interpreter will make assignments in a list beginning with the leftmost element in the list and proceeding to the right.

Example 4.

```
SET   w = SIN(x+y),c=w*x,w=w*y
```

Using the most recently assigned values of x and y, the variable w is assigned the value of the arithmetic expression sin(x+y). Then c is assigned the value of w*x, where the value of w is the previously computed value, sin(x+ y). Finally w is given the newly computed value w*y. Variables w, and c are arithmetic variables because they were given arithmetic values.

Example 5.

```
SET BOOL = The True
```

The variable BOOL is assigned the constant Boolean value The True; therefore the variable BOOL is a Boolean variable.

Example 6.

```
SET ray(1,1)=$NOT(2 = 2)
```

The first = symbol on the left is the assignment symbol. The second = symbol is the relational operator. Therefore the value "false" is assigned to the subscripted variable ray(1,1). ray therefore is a boolean array.

Example 7.

```
101.98 SET Num = "1234567890"
```

The double quotes around 1234567890 cause Num to be a string variable. Therefore Num can be combined by string operations with other string operands. The number 101.98 designates part 101, step 98.

Example 6.

SET ray(1,1)=$NOT(2 = 2)

The first = symbol on the left is the assignment symbol. The second = symbol is the relational operator. Therefore the value "false" is assigned to the subscripted variable ray(1,1). ray therefore is a boolean array.

Example 7.

101.98 SET Num = "1234567890"

The double quotes around 1234567890 cause Num to be a string variable. Therefore Num can be combined by string operations with other string operands. The number 101.98 designates part 101, step 98.

SIMPLE CONSOLE INPUT-OUTPUT

OUTPUT statements

```
PUT <list of items separated by commas>*

        or

PUT LIST <list of items separated by commas>*
```

*(The keyword TYPE may be used in place of PUT).

An item in the output list of the first PUT statement may be one of four types of elements:

1. a variable name whose associated value is to be written,
2. an expression to be evaluated and written out,
3. a character string enclosed in double quotes(")
4. the special keywords, ALL, ALL VALUES, ALL PARTS, PART,STEP.

The PUT LIST form of the output statement differs from the simple put form in that, the list of items of the PUT LIST may not contain the special keywords described in 4 above. In addition, the PUT LIST writes the items in columns of 5 per line until the list is exhausted; a simple PUT writes the items one per line until the list is exhausted. Character strings in the PUT LIST form should not have more than 15 characters; only the leftmost 15 characters of a string are written.

Example 8.

```
PUT "The value of A is" , A
```

Example 8 illustrates the first type of PUT statement. Each item on the list will be written on a separate line. Assuming A has the value 2, this statement will cause the two lines to be written,

```
The value of A is
A= 2.000000
```

The interpreter always writes the variable name followed by the value in this PUT form.

Example 9.

```
PUT LIST "The value of A" ,A
```

Example 9 illustrates the second form of the PUT statement. This statement will cause the line to be written

```
The value of A  2.000000
```

Example 10.

```
PUT 3*SIN(3.14159/16 + SQRT(35.9))
```

This statement will result in the line

```
3*sin(3.14159/16 + sqrt(35.9)) = -.2850968
```

The special-purpose forms and their use are described below:

1. To write out a copy of special parts sorted in step order,

   ```
   PUT  PART <part number>
   ```

   Example:

   ```
   PUT PART 5, PART 6
   ```

2. To write out a copy of the entire program,

   ```
   PUT ALL PARTS
   ```

3. To list all defined variables and their current values,

   ```
   PUT  ALL VALUES.
   ```

4. To list the entire program and all variables in storage,

   ```
   PUT  ALL.
   ```

5. To write out a step,

PUT STEP <part and step number>

Example:

PUT STEP 1.3, STEP 50.1

Since the typewriter is a relatively slow device, these features should be used sparingly.

LINE statement

The line statement causes the console typewriter to space one line down the page.

Example 11.

```
1.1 Put (EXP(1.5)+EXP(-1.5))/2
1.2 Line
1.3 Put (EXP(1.5)-EXP(-1.5))/2
CALL PART 1
```

In example 11, the command LINE will produce a blank line between the two output lines. The output therefore will be

```
(EXP(1.5)+EXP(-1.5))/2=  2.352406

(EXP(1.5)-EXP(-1.5))/2=  2.129278
```

INPUT statements

<part and step number> GET <list of variables separated by commas>*

or

<part and step number> GET LIST <list of variables separated by commas>*

*(The keyword DEMAND may be used in place of GET)

On execution of the GET statement, BRUIN requests the user to provide values for the variables in the list following the keywords GET or GET LIST. The response by the user may be one of four types of data:

1. constants,
2. an expression in terms of previously defined variables,
3. function,
4. any combination of the above.

In the simple GET statement BRUIN prompts the user on each variable in the list by writing the variable name followed by =>; it then waits for a response from the user.

Example 12.

```
1.5 GET a,b,c
```

To the command GET a,b,c, the interpreter will respond with

```
a=>
```

The user may then type a number (e.g. 4.0) after the > symbol. Then the interpreter types

```
b=>
```

Again the user may type 3.5*a. The value of a has already been defined; therefore the value of the product of 3.5 and a will be assigned to b. Finally the interpreter types

```
c=>
```

The user may respond with SQRT(a+b).

Example 13. Assume that the value of i is 1 and j is 3.
3.11 DEMAND B(i,i+j)

BRUIN will return the value of the subscripts with the array name

B(1,4)=>

On execution of the GET LIST statement, BRUIN will prompt only with a greater than (>) symbol. The user may then enter any number of data items on a line; the data items must be separated by commas. Any number of blanks may surround the commas. If the list is not satisfied after the transmission of one line of data, BRUIN will prompt the user for more data. If (n+3) number of data items is transmitted and the list of varables only requests n items, the last 3 data items will be ignored.

Example 14. Assume that A should have the value 5.1, B the value 4.3, C the value $-8x10^{-5}$, and J the value 1.

81.05 GET LIST A,B,C,J

On this command BRUIN will prompt once with the symbol>. The user may then enter on a line

5.1, 4.3, -8.E5, 1.

Notice that the values are assigned to the variables from left to right.

In examples 12, 13, and 14 the variables in the list were given arithmetic values but this need not be the case. Boolean values or character strings may also be read in as values. Example 15 is an illustration of a GET LIST statement with variable A being given an arithmetic value, B a Boolean value and C a character string value.

Example 15.

80.4 GET LIST A,B,C

On execution of statement 80.4, BRUIN transmits the symbol >. Enter

4.125*SQRT(3.14159), THE TRUE, "STRING"

A variable is assigned a data type through the GET statement as well as the SET statement. In example 15 then, the variable A is an arithmetic variable, B is a Boolean variable and C is a string variable.

CONTROL STATEMENTS

In the preceeding sections you were shown how to refer to variable and constant quantities, process input and output and assign values to variables. In most cases a problem cannot be solved by a simple sequence of assignment statements and input/output statements. Statements to permit decision making are often required. These decision processing statements permit the programmer to vary the order in which statements are executed. Such statements which provide the programmer with the ability to alter program flow are here called CONTROL statements.

CALL statement

```
CALL PART <part number>
```

The CALL statement causes a part to be executed starting with its lowest step number.

Example 16. Assume that the following procedure to calculate an expression involving input parameters X, Y, DELTAX, DELTAY is stored in the indirect mode.

```
15.11 SET X = X + DELTAX, Y = Y + DELTAY
15.15 SET FXY  = X*SIND(Y)/COSD(X)+3*X
```

In order to execute the two-step procedure beginning with its first step (here 11) and terminating with its last step (here 15), enter the statement

```
CALL PART 15
```

Since the CALL statement is in the direct mode, it will cause execution of part 15 to take place immediately. Assuming that the variables X, Y, DELTAX and DELTAY have been defined, X, Y, and FXY will be assigned new values. Following the execution of a part (here part 15) invoked by a direct command, BRUIN will halt execution with the statement

```
"EXECUTION HALTED AT END OF PART 15"
```

In the indirect mode, at the termination of the specified part, control passes to the statement following the CALL.

Example 17. Assume part 25 of example 16.

```
 . . .
2.51 CALL PART 15
2.91 PUT X,Y,FXY
 . . .
```

On entering a CALL PART 2 in the direct mode, the order of execution here will be 2.51, 15.11, 15.15 and 2.91. The point to observe here is that control of execution is returned to the statement following the CALL statement (here 2.91). If no errors have been encountered, execution will halt with the statement

"EXECUTION HALTED AT END OF PART 2."

GO statement

It was stated earlier that BRUIN will stop execution of a stored program when an error is encountered. The user may correct his error and continue execution at the point of error. This is done by typing in the correction and then issuing the direct command GO.

Assume that in Example 16 X is initially 0, DELTAX is 1, DELTAY is 1 but that Y was not given an initial value. When the statement of step 15.11 is executed, BRUIN will issue the error statement

"ERROR AT STEP 15.11: UNDEFINED SYMBOL"

Entering the statements

```
SET Y = 1
GO
```

Will cause execution to resume by executing step 15.11 again. A word of caution is necessary here. Because X was already evaluated as 1 before BRUIN discovered the error, X will take on the value 2 when execution resumes at step 15.11

The GO command will not cause execution to resume when BRUIN interrupts because of an infinite loop.

TO statement

There are two forms of the TO statement. One form is

<part and step number> TO PART <part number> *

*(In place of TO, GO TO may be used)

This TO statement causes control to be transferred to the first statement in the specified part. To illustrate the difference between a TO statement and CALL statement Example 17 is rewritten replacing statement 2.51 with a TO statement.

Example 18.

```
15.11 SET X=X+DELTAX,Y=Y+DELTAY
15.15 SET FXY=X*SIND(Y)/COSD(X)+3*X
. . .
2.51 To part 15
2.91 Put X, Y, FXY
```

On a CALL PART 2, the order of execution will be 2.51, 15.11, 15.15. Control of execution is not returned to the statement following the TO part 15 statement (here 2.91) as it is with a CALL.

Another form of the TO statement is

<part and step number> TO STEP <part and step number> *

*(In place of TO, GO TO may be used)

This TO statement causes control to be transferred to the statement with the specified part and step number.

Example 19.

```
15.11 SET X=X+DELTAX,Y=Y+DELTAY
15.15 SET FXY=X*SIND(Y)/COSD(X)+3*X
2.51 TO STEP 15.15
2.91 PUT X,Y,FXY
```

On a CALL PART 2, the order of execution will be 2.51, 15.15. This form of TO permits transfer of control within a part.

The TO statement is valid only in the indirect mode.

IF statement

The IF statement causes BRUIN to test a value and proceed in one or two possible paths. The Boolean expression in the IF statement is the value that is tested. The clauses THEN and ELSE describe the two possible actions. The simplest form is

```
IF <Boolean expression>, THEN <command>
```

If the Boolean expression here has the value true, the THEN clause is executed. If the expression is false the THEN clause is not executed. Execution proceeds with the statement following the IF statement. The word THEN may be omitted but not the punctuation (,).

Example 20.

```
1.5 IF b|2-4*a*c<0, TYPE "ROOTS COMPLEX"
1.6         . . .
```

If the expression b^2-4ac is less than zero, the TYPE command is executed followed by execution of statement 1.6. If the expression b^2-4ac is greater than or equal to zero, control passes directly to statement 1.6.

The other form of IF has an ELSE clause as well as a THEN clause.

```
IF <Boolean expression>, THEN <command>; ELSE <command>
```

If the Boolean expression has the value true, the THEN clause is executed. If the expression is false, the ELSE clause is executed. The words THEN and ELSE may be omitted but not the punctuation (,;).

Example 21.

```
3.9   SET A = x<3 $OR X>5
3.91  IF A, TO STEP 4.0; ELSE TO STEP 4.3
```

In this example A is a Boolean variable. If A has the value TRUE, transfer is made to step 4.0. If A has the value FALSE, transfer is made to step 4.3.

Example 22. Assume that the user wished to accomplish a three-way branch depending on whether x is less than, equal to or greater than y and then wished to return to the statement following the IF statement.

```
1.998  IF x<y, CALL PART 3;  ELSE IF x=y, CALL PART 4;
ELSE CALL PART 5
1.999  . . .

3.1    . . .
       . . .
3.999  SET FXY = SIN(x-y)/x

4.6    . . .
       . . .
4.999  SET FXY = 1.

5.998  . . .
5.999  SET FXY = x/(x|2-y|2)
```

In Example 22 the interpreter compares x and y. If the less than relationship is true, part 3 is executed beginning with step 3.1 and ending with 3.999; control then returns to statement 1.999 by virtue of the interpretation of a CALL command. If x is not less than y, the ELSE clause causes x to be compared with y again in an equality relationship. If x is equal to y, part 4 is done; otherwise part 5. Following both part 4 and part 5, step 1.999 is executed.

The interpreter will take any BRUIN command in the THEN, ELSE clauses but care must be exercised in using another IF in the THEN clause. If the expression is false, the interpreter looks for the command following the first semicolon. For example,

```
1.1 IF a>b, THEN IF c>d, THEN TO STEP 1.4; ELSE TO STEP 4.2;
    ELSE TO STEP 4.3
```

is interpreted as if the statement were written

```
1.1 IF a>b, THEN IF c>d, THEN TO STEP 1.4; ELSE TO STEP 4.2
```

In other words, if both a is greater than b and c is greater than d, branch to step 1.4; otherwise branch to step 4.2. Another way of writing this statement is

```
1.1 IF a>b $AND c>d, THEN TO STEP 1.4; ELSE TO STEP 4.2
```

FOR statement

The FOR statement specifies that a command is to be repeatedly executed until a specified criterion is satisfied. The forms of the FOR statement vary in the stopping conditions and vary in the manner in which the values of a control variable are stated. All forms will take any BRUIN command except a TO command.

The simplest form repeats an object command for a list of values:

FOR <control variable> = <list of arithmetic expressions separated by commas>:command>*

*(The keyword DO may be used in place of FOR; DO and FOR are equivalent).

Example 23.

```
101.5 FOR SUB = J,2*J,4*J,8*J: GET A(SUB)
```

This statement will cause the GET command to be executed four times. Assuming J is 1, the interpreter will request values for A(1), A(2), A(4), A(8). This statement could have been written

```
101.5 FOR SUB = 1,2,4,8: GET A(SUB)
```

Another form of the FOR statement specifies an initial value, increment and final value for the control variable. The general form is:

FOR <control variable> = <initial value> BY <increment> TO

<final value> : <command>

Example 24.

```
101.5 FOR sub = 1 BY i TO 3*i: GET A(sub)
```

Assuming i is 2, the variable sub will take on the values 1, 3, 5. In this form of the FOR statement, the command GET A(sub) is executed for the initial value. Then the control variable (here sub) is incremented (here by i) and compared to the final value (here 3*i). When the value of the control variable is greater than the final value, the loop is terminated. If no increment is specified, an increment of one (1) is used. If the final value specified is less than the initial value, the command is executed once. An infinite loop can occur if an increment is chosen which will cause the limit never to be reached. (e.g. initial value =1, increment = -2 and final value 2).

The BY,TO form of the FOR statement may be combined with the list form as in the following example:

```
1.1 FOR ID =0, 1, 2 BY 2 TO 8,9: CALL PART 8
```

ID here takes on the values 0,1,2,4,6,8 and 9. Any number of BY,TO combinations may be used in the list. If BY is omitted the increment is assumed to be one (1) until the TO limit is reached.

There exist two additional FOR list forms. They are

FOR <variable>=<initial value> BY <increment> UNTIL <Boolean expression>:<command>

FOR <variable>=<initial value> BY <increment> WHILE <Boolean expression>:<command>

Example 25.

```
FOR a=b BY 2 UNTIL a>z:  DELETE X(a)
```

The command DELETE X(a) will be repeated for successive values of a, until a is greater than z.

Example 26.

```
1.6 FOR a=b BY 1 WHILE R<5:  CALL PART 14
```

The command, CALL PART 14, will be repeated for a = b, b+1,...as long as R is less than 5. In the UNTIL and WHILE forms, if no increment is specified the control variable (here a) is not incremented. Care must be taken that the Boolean expression (R<5) in example 26 is not always true. The value of R must at some time in part 14 be set greater than 5 or an infinite loop will result.

STOP statement

If the user wishes to stop the execution of his program and perhaps check some values before continuing, he may use a STOP statement in the indirect mode. Assume that the following part is stored:

```
1.66 SET DIS = B|2 - 4.*A*C
1.67  Stop
1.68  SET D = SQRT(DIS)
1.67  TYPE (-B+D)/(2.*A)
```

On the command CALL Part 1, statement 1.66 followed by statement 1.67 will be executed. The message "STOP AT STEP 1.67" will be issued by BRUIN. BRUIN will then wait for further instructions. At this point the user may make changes or look at D to see if it is negative by typing in the direct mode.

```
PUT  D
```

A GO would cause the program to resume execution at statement 1.68.

DONE statement

<part and step number> DONE

The DONE statement causes the interpreter to halt execution of a part by signalling a logical end. It differs from the STOP in that execution of the stored program does not stop. Execution continues as it would at the physical end of a part.

The DONE statement is valid only in the indirect mode.

Example 27.

```
1.5   FOR I = 1 TO 8:  CALL PART 3
3.3   SET B(I) = 0
3.4   IF A(I) $LE 0, DONE
3.5   SET B(I) = A(I)
3.6   DONE
CALL PART 1
```

In Example 27 the values of A(1) through A(8) will be compared with 0. If the value of A(I), where I = 1,2,...8, is greater than 0, step 3.5 will be executed. If A(I) is less than or equal to 0 steps 3.5 and 3.6 will not be executed. An equivalent process could be accomplished by replacing step 3.4 with

```
3.4   IF A(I) $LE 0, TO STEP 3.6
```

PROGRAM CHANGES

When BRUIN is waiting for a command, the user may enter either a direct or indirect statement. If a statement with a new step is entered, that statement will be inserted in proper sequence in a part. If a statement is entered using an old step number, the old statement will be removed and the new statement will replace it.

DELETE statement

If in further execution certain variables, parts or steps are no longer needed these may be deleted, thereby reducing storage requirements.

Steps, parts and variables may be deleted selectively by statements of the form:

DELETE STEP <part and step number>

DELETE PART <part number>

DELETE <list of variables separated by commas>

Examples:

```
5.7  DELETE STEP 5.1

DELETE PART 4

DELETE X,Z(6),Z(1)
```

After the execution of a DELETE command, for example DELETE X, the variable X is no longer defined and reference to it will generate an error report.

All values or parts may be deleted by the following statements:

```
DELETE ALL VALUES

DELETE ALL PARTS
```

The first statement will leave the defined parts and delete all variables. The second statement will leave the defined variables and delete all parts.

To delete everything belonging to the user (parts and values) the statement

```
DELETE ALL
```

is used.

STORAGE REQUEST

When a user makes a request to run a BRUIN job he is initially allocated a fixed amount of core. If more space than has been allocated is needed at some point in a job, the interpreter will send a message

"NEED MORE SPACE"

The user may free part of the fixed space allocated to him by issuing a form of the delete command. If this is not feasible he may request more space with the ALLOCATE command.

ALLOCATE statement

ALLOCATE <number of blocks(1 to 9)>

Example:

ALLOCATE 3

ALLOCATE 3 requests 3 blocks of storage, each of which will hold about 120 values. If there is insufficient core to fulfill this request, BRUIN will issue the message

"LAST N NOT ALLOCATED"

For example, if BRUIN were unable to allocate one of the three blocks requested, the message would be

"LAST 1 NOT ALLOCATED"

The user may continue with the 2 blocks allocated. If BRUIN were unable to allocate any of the 3 blocks the message issued would be

"LAST 3 NOT ALLOCATED"

The user should request space at a later time.

ALLOCATE may be used in the direct or indirect mode.

FILE MAINTENANCE

SAVE Statement

The SAVE statement enables the user to save BRUIN programs and/or values as a file which can be loaded into core at a later time.

The list of items to be saved may contain:

1. names of variables,
2. the keywords ALL PARTS,
3. the keywords ALL VALUES,
4. the keyword ALL,
5. the keyword PART followed by a part number.

The filename is a name of from one to eight alphabetic or numeric characters, the first of which is alphabetic. If the alphabetic characters in the filename are not upper case, BRUIN makes them upper case. This change in name prevents a user from creating files which cannot be accessed or manipulated by CMS commands.

```
SAVE as <filename>  <list of items separated by commas>
```

Example 28:

```
V (1) = 0
FOR I = 1 BY 1 WHILE I<10: SET A(I) = EXP (V(I) ), V (I + 1) =
        V (I) + .1
SAVE AS EXPTBL ALL VALUES
```

In example 28, the SAVE command causes the variable names and values of A (1) through A (9), V (1) through V (10) and the final value of I to be saved in a file called EXPTBL BRUIN. In order to use the file at a later time, you must use the same filename in a LOAD statement.

```
LOAD EXPTBL
```

The SAVE command will replace with the new file if a previously saved file exists with the same name.

SAVE is valid in both the direct and indirect mode.

All files created by BRUIN have a filetype of "BRUIN".

LOAD Statement

LOAD <filename>

The LOAD command causes a file to be loaded into the users core area. If the user has defined parts or variables before issuing the LOAD command, the parts and variables will be merged with the file being loaded. Merging is done in the following manner:

1. If a part or step defined in core has the same number as a part or step in the file, that part or step in core will be replaced by the part or step in the file; otherwise both parts and steps will be retained in core.

2. If a variable defined in core has the same name as a variable defined in a file, the value of the variable in core will be replaced by the value of the variable from the file.

Example 29:

```
X (1) = .1
FOR I=1 BY 1 WHILE I <10: SET A(I)=SIN (X(I)), X(I+1)=X(I)+.4
LOAD EXPTBL
```

Assume that EXPTBL refers to the file in Example 28. Due to the nature of the LOAD merge, A(1) through A(9) in core will be replaced by A(1) through A(9) from the data set EXPTBL. In addition, the variables V(1) through V(10), X(1) through X(10) and I will be defined in core.

LOAD is valid in the direct and indirect mode.

OTHER FEATURES

Keywords Operands

Various keyword operands such as ALL, ALL PARTS, ALL VALUES, STEP and PART are valid in the DELETE statement, simple PUT statement and SAVE statement. In addition to the above operands there are three operands which are acceptable in either the PUT statements or as operands in expressions. They are THE SIZE, THE TIME, and THE DATE.

THE SIZE is a floating point number which is the number of free elements left in users core area. It is approximately the number of new values that users core area can still hold.

THE TIME is a floating point number which gives the time since midnight in hundredths of a second.

THE DATE is a character string value of the form YYDDD. YY is the tens and units digit of the year and DDD is number of the day with January 1 as day 1.

Example: The statement

```
PUT THE TIME, THE DATE
```

produces two lines of output as follows:

```
THE TIME=  0.5391414E+07
THE DATE="68220       "
```

Comment Statements

A statement beginning with a C followed by one or more blanks is accepted by BRUIN as a user comment line. To place such a comment in a part, write the part-step number followed by a blank followed by the letter C. Comment statements can be used anywhere in the program. (See sample program, Appendix A).

Stopping Program Loops

The SPACE key in addition to producing blanks serves another important function in BRUIN. If a user program appears to BRUIN to be doing too much computing without writing or reading, BRUIN will stop computing and interrogate you simply with a question mark:

" ? "

If the user wishes to continue at the point where BRUIN stopped the program, he must enter at least one blank (press the space bar at least once) and then press carriage return. If the user does not wish to continue at the point where BRUIN stopped the program, he may type any command. BRUIN will process that command.

MATRIX STATEMENTS

In order to treat doubly subscripted variables in a notation similar to a mathematical matrix notation, a set of matrix statements has been included in the BRUIN language. The matrix statements begin with the keyword MAT and may be used in either the direct or indirect mode.

An mxn dimensional BRUIN matrix A is a subscripted variable A(M,N) which is defined for all combinations of the first subscript varying from 1 to M and the second subscript from 1 to N. A column vector which is the result of a MAT instruction has an explicit second dimension of 1; a row vector has an explicit first dimension of 1. A matrix or scalar which appears in a MAT instruction may appear in non MAT instructions as a doubly subscripted variable or in the case of the scalar as a simple variable. The interpreter makes no attempt to keep the dimensions of a matrix fixed to the original dimensions. Therefore a matrix which was input with row size 3, column size 2 may as the resultant matrix of an arithmetic operation have a different row and column size.

Matrix Input statements

There are two matrix input statements comparable to the ordinary Bruin input statements:

MAT GET <matrix name> (<row size> , <column size>) *

or

MAT GET LIST <matrix name> (<row size> , <column size>) *

*(The keyword DEMAND may be used in place of GET)

The expression for row size and column size must be an arithmetic expression or arithmetic constant greater than or equal to 1. If the expression is not an integer, it is truncated to the nearest integer.

The MAT GET statement prompts the user for all elements of the matrix. Let m be the row size and n be the column size of matrix a; then the MAT GET statement is equivalent to the non matrix BRUIN statement:

```
FOR i=1 TO m: FOR j=1 TO n: GET a(i,j))
```

The MAT GET list statement behaves as its non-matrix counterpart in that the input may be in the form of a list of elements separated by commas.

The elements of the matrix must be listed row-major order; that is, the value for a must be listed in the order

a (1, 1) ,a (1,2) ,...a (1,n) ,a (2,1) ,...a (m,n)

In both forms of the MAT GET statements the matrix, if previously defined, must specify a subscripted variable. The subscripted variable will be redefined as a matrix where the dimension of the matrix will be changed to the row size and column size specified.

<u>Example 30:</u>

```
MAT GET a(2,2)
```

The interpreter will prompt with

```
a (1,1) =>
```

The user then types the value he wishes to enter for this matrix element. Then the interpreter types

```
a (1,2) =>
```

The user responds again with a value. Similarly, the interpreter prompts for a (2,1) and a (2,2). Just as in a non MAT statement, the user response may be an arithmetic expression involving a previously defined variable. For example, the response for a (1,2) could have been a (1,1) **2.

<u>Example 31:</u> Assume that matrix B= 11 12 13
21 22 23

To input this in a list form the command is

```
MAT GET LIST B(2,3)
```

The interpreter responds with a >. The user may then input

```
11,12,13,21,22,23
```

Matrix Output Statements

MAT PUT <matrix name> (<row size> , <column size>) *

or

MAT PUT LIST <matrix name> (<row size> , <column size>) *

*(the keyword TYPE may be used in place of PUT)

The expressions for row size and column size, give the dimension of the matrix to be printed. This need not be the full dimensions of the matrix.

Let B be a matrix with row size m and column size n. The simple MAT PUT is equivalent to the BRUIN statement:

```
FOR i=1 TO m; FOR j=1 TO n:  PUT B(i,j)
```

One element of B per line will be printed along with its identification.

The MAT PUT LIST closely resembles its PUT LIST counterpart in that the output will be in columns with a maximum of 5 columns per line. However, only one matrix row is written per line. Thus, a 3x3 matrix is written as 3 lines with 3 entries per line. If a matrix has more than 5 columns the remaining elements of the row are printed on a following line or lines, indented to the second column.

Example 32: To output the matrix of example 31, execute the following statement:

```
MAT PUT B(2,3)
```

The interpreter responds with

```
B(1,1)=    11.0
B(1,2)=    12.0
B(1,3)=    13.0
B(2,1)=    21.0
B(2,2)=    22.0
B(2,3)=    23.0
```

Example 33:

```
MAT PUT LIST B(2,3)
```

The output will be

```
11.0            12.0            13.0
21.0            22.0            23.0
```

It is permissable to specify a smaller row size, j, or column size, k, in either MAT PUT statement than is defined for the matrix. BRUIN will simply output the upper right hand jxk partition of the matrix. Assuming the matrix B of example 34, the statement

```
MAT PUT LIST B(1,2)
```

will produce the line

```
11.0            12.0
```

Matrix Expressions

A BRUIN matrix expression consists of at most one operation. The operation may be addition, subtraction or multiplication of two matrices, multiplication of a matrix and a scalar, multiplication of a matrix and a vector (singly subscribed variable), or the evaluation of special BRUIN matrix functions. In some cases the dimensions of the matrix operands are required to fulfill certain constraints such as is required in conventional matrix algebra. In the case of the binary operations the two matrix operands are examined to insure that the operation is conformable. In some of the matrix functions such as determinant evaluation the matrix must be square.

Table 5 describes in detail the permissable matrix expressions; Table 6 describes the available BRUIN matrix functions.

Expression	Definition	Dimension of A	Dimension of B	Dimension of Result
A+B	matrix add	mxn	mxn	mxn
A-B	matrix subtract	mxn	mxn	mxn
A*B	matrix multiply	mxn	nxk	mxk
A*s	multiplication			
	of a	mxn		mxn
s*A	scalar and matrix			
V*A		mxn		1xn
A*V		nxm		nx1
Function(A)				
A=B	assignment		mxn	mxn

A and B are Bruin matrices; s is a constant or simple variable;
V is a singly subscripted variable of dimension M.

Table 5. Matrix Expression

Function	Argument	Result	Dimension of result
DET(A)	nxn matrix A	determinant (A)	simple scalar
INV(A)	nxn matrix A	inverse of A	nxn
TRA(A)	nxn matrix A	trace of A	simple scalar
TRS(A)	nxm matrix A	transpose of A	mxn
IDN(s)	positive constant or variable	identify matrix	pxp where p is the integer part of s

A is a Bruin matrix; s is a constant or simple variable.

Table 6. Matrix Functions

Matrix Assignment Statements

MAT SET <variable name>=<matrix expression>,<variable name>=<matrix expression),etc.

The MAT assignment statement assigns the value of the matrix expression on the right hand side of the equal symbol to the variable on the left. Not only does the matrix expression give the variable a value but defines its type (e.g. simple scalar, matrix). If the variable on the left has already been defined as an arithmetic subscripted variable, it can be redefined as a matrix. No attempt is made in MAT instruction to keep the dimension of a matrix fixed. Example 37 is given as an illustration of changing dimension sizes. The keyword SET is optional.

<u>Example 34:</u> Assume that matrix B is a 3x2 matrix, G is a 2x2 matrix, and R is a 2x2 matrix. The object is to compute $GR^{-1}B'$.

```
MAT SET B=TRS(B),RI=INV(R),B=RI*B
MAT B=G*B
```

In example 37, according to the rule of matrix expressions, only one matrix operation per assignment is made. The first statement is a multiple SET statement with the assignments being performed from left to right. First the transpose of B is stored into B. The dimension of B is now 2x3. Secondly, the inverse of R is stored into RI and the product of RI and B transpose is computed. Finally, RI*B is premultiplied by G and the result stored in B.

SUMMARY

For a summary of the BRUIN instruction set, see Appendix A, part 1.

A complete sample program as typed on the 2741 console appears as part 2 of Appendix A.

Any comments about the organization or content of this manual or improvements to the language will be appreciated.

APPENDIX A

1. Table of Instructions

command	mode
ALLOCATE	direct/indirect
CANCEL	direct
DELETE	direct/indirect
GET (DEMAND)	indirect
CALL	direct/indirect
DONE	indirect
FOR (DO)	direct/indirect
GO	direct
IF	direct/indirect
LINE	direct/indirect
LOAD	direct/indirect
MAT GET	direct/indirect
MAT PUT	direct/indirect
MAT SET	direct/indirect
SAVE	direct/indirect
SET	direct/indirect
STOP	indirect
TO	indirect
PUT (TYPE)	direct/indirect

2. Sample Program and Output

```
>1.0 C sample program to solve quadratic equations
>1.1 put  "Program to Solve Quadratic Equations"
>1.2 put  "      a*x*x+b*x+c=0"
>1.3 for i=1 to 3:  line
>1.4 put  "Enter Coefficients"
>1.5 get a,b,c
>1.6 set discrim=b*b-4*a*c
>1.7 if discrim<0, then call part 2; else call part 3
>1.8 line
>1.9 put  "Enter 0 to terminate, 1 to continue"
>1.91 get ans
>1.92 if ans=1, then to step 1.3; else if ans=0, done; to step 1.9
>2.1 put "roots are complex"
>3.1 set root1=(-b+sqrt(discrim))/(2*a)
>3.2 set root2=(-b-sqrt(discrim))/(2*a)
>3.3 type " "," ", root1,root2
>CALL part 1
Program to Solve Quadratic Equations
      a*x*x+b*x+c=0



Enter Coefficients
a=>1.
b=>10.
c=>-40.


root1= 3.062255
root2=-13.06225

Enter 0 to terminate, 1 to continue
ans=>1.


Enter Coefficients
a=>1.
b=>0.
c=>1.
Roots are complex

Enter 0 to terminate, 1 to continue
ans=>0.
EXECUTION HALTED AT END OF PART 1.
```

APPENDIX B

1. BRUIN Character Set

name	special character
blank (space bar on console)	
equal	=
plus sign	+
minus sign	-
asterisk	*
slash	/
left parenthesis	(
right parenthesis	)
comma	,
point or period	.
semicolon	;
colon	:
"NOT" symbol	¬
"AND" symbol	&
vertical bar	|
"greater than"	>
"less than"	<
double quotation	"
apostrophe	'
dollar sign	$
cent sign	¢
exclamation point	!
percent sign	%
underscore	_
question mark	?
pound sign	#
at sign	@

2. Built-In Mathematical Functions

function	value returned	error condition	example
ABS	$\|x\|$	–	Abs(x)
LN	log (x) base e	$x \le 0$	Ln(3*z)
LOG2	log (x) base 2	$x \le 0$	log2(3*z+5)
LOG	log(x) base 10	$x \le 0$	LOG(x)
EXP	e to the power x	x>174.6731	EXP(5.1)
SQRT	square root of x	x < 0	sqrt(3.14/5)
SIN	sin(x), x in radians	$\|x\| \le 2^{18}$ x pi	sin(a*b)
COS	cos(x), x in radians	$\|x\| \le 2^{18}$ x pi	cos(3.1/8)
SIND	sin(d), d in degrees	$\|x\| \le 2^{18}$ x 180	sind(360/4)
COSD	cos(d), d in degrees	$\|x\| \le 2^{18}$ x 180	COSD(360/8)
TAN	tan(x), x in radians	$\|x\| \le 2^{18}$ x pi	TAN(3.14/4)
TAND	tan(d), d in degrees	$\|x\| \le 2^{18}$ x 180	TAND(a*b)
ATAN	arctan(x) in radians, -pi/2<ATAN(x)<pi/2	–	ATAN(z)
ATND	arctan(x) in degrees, -90<ATAN(x)<90	–	ATND(z)
ERF	$2/\sqrt{\pi}\int_0^x EXP(-u^2)\,du$	–	ERF(.8)
ERFC	1 - ERF(x)	–	ERFC(2.*x)
SINH	sinh(x)	x>174.6731	SINH(3.1*x)

	COSH	cosh(x)	x>174.6731	COSH(x)
	TANH	tanh(x)	-	TANH(z)
	ATNH	inverse hyperbolic tangent of x	ABS(x)≤1	atnh(x)
	IP	integer part of a number x sign of x times largest integer ≤|x|	-	IP(3.15679)
	FP	fractional part of a number	-	FP(x*z)
*	RAND	uniformly distributed random numbers between 0 and 1.	x ≤ 0	RAND (X)
	GAMA	$\int_0^\infty u^{x-1}$ EXP(-u) du	$x \le 2^{-252}$ or x>57.5744	GAMA(3*X)
	**FACT	x!where x!=x(x-1)(x-2)...	$x \le 2^{-252}$ or x>57.5744	FACT (3)

*For the first entry, x should be an odd integer. After the first entry x should be set equal to the previous result of the RAND function.

** x! is computed using the relationship x!=GAMA (x+1)