An Interpreter for the Brooker's Manchester Mark I Autocode, of 1955

Abhir H. Bhalerao

Department of Computer Science,
University of Warwick,
Coventry,
England.

ABSTRACT

mark1 is an interpreter designed to accept source programs written in Brooker's Autocode, version 1, as outlined in the "The Standard Account of the `Simple Machine´", (3.6.55). *mark1* tries to reflect the original environment of Autocode, i.e The Manchester Mark I, in that there are real limitations placed on the number of instructions per program, the number of *variables* and *indices* that are allowed to be introduced. *mark1,* however, does offer some non-standard facilities that do not affect the running of the source code, like a comment marker.

April 15, 1985

An Interpreter for the Brooker's Manchester Mark I Autocode, of 1955

Abhir H. Bhalerao

Department of Computer Science,
University of Warwick,
Coventry,
England.

1. NOTES FOR THE USER

1.1. Source Programs

Source programs can be standard formated files, which can be created using any available text editor. The source file has to have *no* special name.

The full version 1 instruction set has been implemented but can have some additional facilities:

1) *Comments* can be included by placing the UNIX† pipe symbol | any where on a line. The interpreter will then ignore the rest of that line. For example:

```
|     This program demonstrates the comment facility
|
          n1 = 0          |this comment is ignored
          n1 = n1 + 1     |...and so is this
```

2) *Non standard symbols* : since UNIX does not have the Autocode perforator characters for multiplication, not equals and greater than or equals, the following are used:

little **x** represents the multiplication operator $\otimes$
and C **>=** , **!=** the relationals $\geq$, $\neq$

3) All *spaces* and *tabs* are ignored.

4) All instructions must be terminated by a *newline* (or followed by a *comment marker*). This is because of the implementation uses a *yacc* generated parser. For example:

```
(j1)
```

is illegal because the instruction **j1** is not followed by a new line. This is contrary to the examples given in Brooker's, "Simple Machine", notes.

5) *mark1* will crash if comments are included on data number lines. All numbers to be input must be on separate lines following the last obeyed instruction in the *interlude*.

The reason for this is that the **z=I** , data input facility is implemented using the C *fscanf()* function.

---

† UNIX is a trademark of Bell Laboratories.

A typical code sequence for inputing numbers might be:

```
|   input numbers into the vector v1 -> v10
|
    1     n1 = 1
    2     vn1 = I          |input next number
          n1 = n1 + 1      |increment index
          j2, 10 >= n1
          (j1              |10 numbers must follow
          1.0
          2.0
          ...
          ...
          10.0
          )
```

Alternatively *mark1* can have a **-d** option which will take input from a special file called *m.data*. In our example the 10 numbers could have instead been held in *m.data* and *mark1* **-d** used to invoke the interpreter.

6) The **H** *halt* instruction will stop the interpreter and a message will be issued:

```
Press s to restart
```

If **s** is typed and **<return>** is pressed, the interpreter will restart. This is very useful for debugging. *Note* that the *halt* instruction is a *directive* and will always be executed.

1.2. Using the interpreter and debugging

The interpreter is invoked as follows:

$ mark1 *<file>*

where $ is the UNIX prompt and *<file>* is the name of the Autocode source file. A legal source file *must* be given. There is also a non-standard *trace* option:

$ mark1 -t *<file>*

This will print the number of the statement as it executes. Statements numbers being taken from line 1 of the source file (including comment lines). It will also print the *symbol table* immediately after the *interlude* symbol **(** has been met.

1.3. Output

All program output is sent to stdout, hence the UNIX operator **>** can be used to force it to a specified file:

$ mark1 -t *<file>* **>** *<output>*

Note that all error messages are sent to *stderr* so as not to interfere with program output.

1.4. Errors and Diagnostics

Where possible the parser will issue *(I hope)* useful diagnostics. The *yacc* standard message **syntax error** , is supplemented by the output of the offending line and line number. *mark1* will try to find as *many* interpretive errors as possible. Any errors will stop execution just before the *interlude* is begun. *Fatal* run time errors will also cause *mark1* to crash - like a jump to an undefined label or accessing an illegal variable e.g.

```
n1 = 0
vn1 = I          |a fatal out of range will happen here
```

1.5. Limits on the no. of instructions, variables, indices & labels

In an attempt to model the environment of the Mark I Autocode, i.e. the Manchester Mark I itself, there are specific limits placed on the number of instructions per program, the number of labels etc.

Max. no. of instructions:	500*
Max. no. of *variables*:	5000
Max. no. of *indices*:	18
Max. no. of *labels*:	500

* this includes comment lines as null instructions.

Of course none of the above need be set in a UNIX/C environment.

1.6. Function table II

This has not been extended and is implemented as it stands using the C *math.h* library.

1.7. Program output

This is always done to 11 significant digits and trailing zeros are *not* supressed. All data output is sent to *stdout* and can therefore be redirected by using a UNIX pipe or the > operator on the command line.

1.8. Program run timings

mark1 will output the *tape* , *execution* and *total* times in minutes and seconds after a program is finished. These times are estimates based on the rough figures given in the "Simple Machine" notes. Quote: "The rate of scanning is approximately two seconds per instruction and the rate of execution about 8 per second." *mark1* will always output these times to *stderr* even if execution fails due to errors.

(Version 1, 3.6.55)

The Standard Account of the 'Simple Machine'

By means described elsewhere the electronic computer can be made to accept programmes written in a simplified form, described below. In this form a programme of calculation consists of an ordered sequence of instructions arranged in a single column. These instructions are chosen from a permissible set and employ only the symbols given in Table I. Ultimately the programme is presented to the machine in the form of a length of perforated paper tape which is scanned by a photo-electric tape reader, the input unit of the machine. The machine gives its results on a page printer.

The instructions mostly take the form of equations giving the new value of a computed quantity in terms of one or two previously calculated quantities or parameters. Instructions are also necessary however for selecting between alternative courses of action, for printing results, and for the input of further data. The instructions involve the following kinds of quantities.

(a) Variables The quantities or intermediate quantities which it is necessary to compute. These are denoted by v1, v2, v3, etc.. The number of variables which can be introduced is for practical purposes unlimited: it is in fact about 5,000. The range of magnitudes of a variable is virtually infinite; 2^p, where $2^{18} > p \geq -2^{18}$; the precision is limited to 11 decimal figures.

(b) Constants, or variables of which the value is known is advance. The same restrictions of magnitude and precision apply as for variables. The numerical values are written in the form:

integral part, decimal point, fractional part.

Absolute standardisation of form is not necessary, thus to six significant figures π may be written 3.14159 03.14159 03.141590 +03.141590. All these and similar variations are accepted by the machine. Of the number 003.14159265358979 however only the first 11 significant figures would be recorded inside the machine, i.e., the final 8979 would be omitted. Negative numbers must be preceeded by sign.

(c) Indices In order to take advantage of the repetitive nature of calculation it is desirable to have some means of specifying any of the elements of a sequence, e.g., the components of a vector. For this purpose the notation vn1, vn2, vn3, etc., is introduced where n1, n2, n3, etc., are indices restricted to integral values but otherwise computable like variables. The number of indices which can be introduced is limited to 18. The possible range of values of an index quantity is $2^{18} > n \geq -2^{18}$, which of course far exceeds its usefulness as an index proper. In writing down the numerical value of an index the decimal point may be omitted.

Those instructions which take the form of equations are given below (i) - (iii). Permissible instructions are obtained from these basic forms by replacing x, y, and z by the group of symbols denoting a variable (v), combination (vn), or, except in the case of (iii), index (n). In addition x and y may also be replaced by constants.

(i)	z = x	
(ii)	z = x Θ y	Θ is replaced by one of the symbols + - ⊗ /
(iii)	z = F m (x)	The integer m refers to the function table 2

Instructions are normally obeyed in the order in which they are written down until a jump instruction is encountered. This may be conditional or unconditional and takes the form (iv) or (v) as follows.

(iv)	j m	means 'jump to instruction labelled m'.
(v)	j m, x ϕ y	means 'jump to instruction labelled m if x ϕ y' where ϕ is replaced by one of the symbols $\geq$ $>$ $=$ $\neq$.

The label is a positive whole number which is written immediately before (i.e., to the left of) the instruction concerned.

The remaining instructions are

(vi) The symbol * inserted before any of the instructions (i,ii,iii,& viii causes the computed value of z to be printed on a new line in the style described earlier, namely, integral part, decimal point, fractional part, the latter to 10 decimal places. Insignificant zeros are suppressed. In the case of a number outside the range $2^{\pm 38}$, the machine prints on the same line both the integral part and the fractional part of the number in the form $a.2^p$, where $\frac{1}{2} > |a| \geq \frac{1}{4}$, $2^{18} > p \geq -2^{18}$.

(vii) The letter H is a dummy stop instruction: the machine halts: it can be made to resume operation by pressing a key on the console.

Table I

letters	figures
FS	FS
A	1
B	2
C	*
D	4
E	(
F	)
G	7
H	8
I	≠
J	=
K	-
L	v
M	LF
N	SP
O	,
P	0
Q	>
R	≥
S	3
T	j
U	5
V	6
W	/
X	⊗
Y	9
Z	+
LS	LS
.	.
?	n
£	CR
▪	▪

Table II

1 $\sqrt{x}$
2 cos 2π x
3 exp x
4 $\log_e$ x
5 $\frac{1}{\pi}$ arc tan x.
6 |x|

As a simple exercise in coding write down instructions for evaluating the sum of squares of v1, v2, ..., v100. A suitable sequence is:-

```
    n1  =  1
  v101  =  0
2v102   =  vn1⊗vn1
  v101  =  v101 + v102
    n1  =  n1 + 1
    j2, 100 ≥ n1.
```

Such a group of instructions may form part of a larger programme involving the variables in question.

A complete programme of instructions and numbers is normally recorded inside the machine before being executed: it is therefore necessary to explain the input procedure.

The programme tape is prepared on a keyboard perforator on which are engraved the standard symbols. The symbols are 'punched' in the conventional sequence, namely from left to right and down the column. Each instruction is followed by the symbols 'CR' (carriage return) and 'LF' (line feed). The keyboard is normally on 'figures' which means that capital letters such as F, H have to be preceeded by 'LS' (letter shift) and followed by 'FS' (figure shift). Figure shift corresponds to blank tape and any number of blanks may separate the figure symbols proper provided the order is preserved. About six inches of blank should be left at the head of the tape. Associated with the keyboard is a printer which gives a printed copy of whatever is punched: this should agree with the original manuscript.

As the programme tape is scanned the instructions are normally recorded in the store where ultimately they will be obeyed. The rate of scanning is approximately two seconds per instruction and the rate of execution about 8 per second. In the case of instructions included between brackets each instruction is executed immediately after it has been read and is not recorded in the programme proper. This facility is used to start the programme simply by including an unconditional jump instruction between brackets, e.g., (j1) which means "stop reading the tape and start obeying the programme at the instruction labelled 1".

Having got the programme started it may be necessary to call on the input medium for further numerical data. This may be done in two ways for which it is necessary to introduce two further instructions. These are:-

(viii) z = I, which means 'replace z (which has the same significance as in (i) - (ii)) by the number formed by the next group of symbols on the tape';

(ix) the letter T which causes the machine to start reading further instructions from the tape, adding them to those already recorded in the store.

Instruction (viii) may be used to read a tape bearing numbers only.

The T instruction, combined with the 'bracket' facility, allows data to be input in the form (z = constant). This is a convenient method of altering parameters in between different runs. Thus, e.g., if the supplementary instructions take the form

```
(v23  =  0.012
 v24  =  0.965
 n3   =  12
   j1 )
```

then the effect will be to repeat the run with these new values of the quantities v23, v24, and n3.

It is hoped to include further facilities, e.g., table II will almost certainly be extended. Supplementary notes may therefore be circulated from time to time.

To illustrate the coding scheme described above, the following calculation is programmed.

It is required to compute

$$\alpha = \frac{1 - (p_2/p_1)^{2/7}}{1 - (p_2/p_3)^{2/7}} \quad , \quad \text{where} \quad p_2 = 30, \quad p_3 = 40$$

$$r_e = 1.3 + 2\left\{1 - \frac{\alpha^{3/2}\,(p_2/p_3)^{2/7}}{(p_2/p_1)^{2/7}}\right\}$$

$$r_m = 1.3 + 2\left\{1 - \frac{\alpha^{1/2}\,(p_2/p_3)^{2/7}}{(p_2/p_1)^{2/7}}\right\}$$

for values of p_1 of the form $40 - \frac{10n}{156.4}$, where $n = 0(1)156$.

The programme is given below

```
 2v1 = 40                    (p1 = 40)
  v2 = 30                    (p2 = 30)
  v3 = 40                    (p3 = 40)
  v4 = 10/156.4
  v5 = v2/v3                 (p2 / p3)
  v6 = F4(v5)
  v7 = 2/7
  v8 = v7⊗v6
  v9 = F3(v8)                (p2/p3)^2/7
 1v10 = v2/v1
  v11 = F4(v10)
  v12 = v7⊗v11
  v13 = F3(v12)              (p2/p1)^2/7
  v14 = 1 - v13
  v15 = 1 - v9
*v16 = v14/v15               forms and prints α
  v17 = F1(v16)
  v18 = v17⊗v9
  v19 = v18/v13
  v20 = v16⊗v19
  v21 = 1 - v20
  v22 = 2⊗v21
*v23 = 1.3 + v22             forms and prints re
  v24 = 1 - v19
  v25 = 2⊗v24
*v26 = 1.3 + v25             forms and prints rm
   v1 = v1 - v4              adjusts p1
   j1, v1 > 29               tests for last cycle
      H                      halt
    (j2)                     starts programme.
```

The following notes are of interest

(1) No attempt has been made to economise on the use of variables. Thus e.g., instead of v9 = F3(v8) one could write v8 = F3(v8) since the argument is no longer needed. However nothing is gained by so doing since space is virtually infinite for problems of this kind.

(2) A further example of laziness is the means adopted for evaluating 2/7. Instead of bothering to evaluate the fractions we have simply included an instruction for doing so, namely v7 = 2/7.

(3) The value of any intermediate quantity can be printed simply by inserting an * before the appropriate instruction. This may be useful in locating mistakes: the * can afterwards be 'erased' by turning it into a . x

(4) The maximum number of instructions allowed cannot be given very precisely but a safe estimate is 500, which, in view of their comprehensive nature, should be adequate for the class of problems envisaged.

2. IMPLEMENTATION OF THE INTERPRETER

The interpreter has been implemented using the UNIX *yacc* facility. The lexical analyser has been specifically written without using the *lex* facility.

2.1. Overview of the implementation

The interpreter *mark1* can be described as having **three** basic modes:

I Non-execute tape mode

Programs for the Manchester Mark I would have been punched onto a paper tape and at the beginning of a run, instructions were scanned and recorded into memory by the Mark I (see "Simple Machine" notes, page 2). This is the normal *non-execute tape* mode. *Tape* indicating the source of the input. For the *mark1* the input stream is the source file in *stdin.*

II Execute tape mode

When and instruction in brackets is met, an *interlude* , it is *not* recorded into store but is *executed* immediately. This is used to start a program by jumping to a labeled location in memory, although *any* instruction will be executed immediately within an *interlude.* e.g. variables can be set before execution starts.

III Execution memory mode

This is the normal execute mode with the *memory* indicating that instructions are read from memory and executed sequentially and normal branching can take place. When the last instruction is obeyed the program control is returned to the *calling interlude* and instructions are once again read from tape. Usually the program will be stopped at this point by a close bracket) indicating the end of the *interlude,* but more *execute tape mode* instructions may follow.

The three modes will be referred to as:

I - *non-execute tape mode*
II - *execute tape mode*
III - *execute memory mode*

2.2. Memory representation

The memory is represented by a 2D character array of lines. Each line can hold a program statement and can be accessed by the line number:

char **memory**[MEMSIZE][MAXLINE]

2.3. Vectors and indices

All *variables* and *indices* are *double* vectors:

double **v**[MAXVAR], **n**[MAXINDEX]

and

v1 is translated to **v**[1]
vn2 is translated to **v**[*(int)***n**[2]]

Although the parser only allows **n**[] to hold integers, it is made *double* to simplify the *yacc* parser specification. The parser actions for arithmetic expressions are simplified as mixed type arithmetic and indirection does not have to be dealt with.

2.4. Sequencing and transfer of control

There is program counter **mem_ptr** that is incremented from **one** as statements are read and stored in memory and as statements are executed in memory. Otherwise **mem_ptr** is static, (mode II).

When control is transferred from II to III the top of the memory is stored in case further instructions are still to be input i.e. if a **T** instruction is met during execution. When a **T** *is* met, control is transferred from III to I and **mem_ptr** is set to top of memory.

A jump instruction (conditional or unconditional) will cause **mem_ptr** to be changed. A jump from II to III, e.g. to start a program, also involves the change of input to the lexical analyser (input from *memory* instead of *tape*).

For example if the jump is made to statement number **instr** then the next statement interpreted will be the line

memory [**instr**]

2.5. The Symbol table and Labels

The branch instructions are executed by using a *symbol table,* just like an assembler will use.

When in mode I, the interpreter builds up a symbol table of the form:

label : **location**

where **location** is the current value of program counter **mem_ptr** i.e the statement where the label is encountered. Since *labels* can only be positive integers the symbol table is implemented by a simple 2D integer array:

int **symbol**[MAXLABELS][2]

During mode II & III any encountered *labels* are ignored. However, a jump instruction like:

j label
or **j label, condition**

will cause the *label* to be scanned in the symbol table and a jump made to the corresponding location, i.e.:

mem_ptr = find(**label**)

where *find()* returns the location of *label.*

There are of course several error checks made: before a label is added to the table which already exists an *ambiguous labels* error is issued, if a branch is attempted to an undefined label an *undefined label* message is issued. In both cases appropriate actions are taken to ensure the halt of execution.

2.6. Interludes

These are enclosed in brackets with in a source sequence and cause transfer from mode I to mode II. When an open bracket (, is encountered *execute* mode is entered and all the following instructions are executed *but* not transferred into memory. If however a jump is made into memory then mode III is entered. This involves changinging input stream for the lexical analyser from *tape* to *memory.* A close bracket) , is the end of *interlude* marker and causes transfer from mode II to mode I. The end of *interlude* is usually the last instruction in a program, but more instructions may follow and will continue to be added into memory from the top of memory.

2.7. Tape instructions

These, when executed, cause a transfer from mode III to mode I, and further instructions are added to memory from the top of memory. *Notice* that execution will only resume if a jump is made again into memory from another interlude.

The **T** instruction allows for program modification and conditional *compilation.* Subsequences of code can be *conditionally* loaded into memory and executed.

2.8. Halt and Print instructions

The **H** instruction will stop the interpreter and is implemented by a simple *while* loop. This loop will wait until a key is pressed on the key board.

Printing is done by the C function *fprintf()* with a **%11.Nf** format, where the **N** is varied depending on the number of digits in the integer part of the number. This ensures that the number is printed to 11 significant digits. Trailing zeros, however, are not supressed.

3. EXAMPLES

The following pages contain examples of Brooker's Mark I Autocode. These examples are intended to demonstrate and test the various facilities offered by *mark1*. They also show how instructions like **T** and **I** , and the interlude facility can be used. The first few examples are almost trivial in nature, but they are presented in increasing order of complexity with a fairly long program for matrix inversion.

3.1. A simple count program

The points to note in this program is the way the program is started, by a jump to the label **1** , also note the two further instructions with in the *interlude*. The program is in a file called **count.**

```
|a simple test program
|to count up to 10

        n1 = 0
1       *n1 = n1 + 1
        j1, 10 > n1
        H
        (j1
        *n1 = n1        |two more instructions to execute
        *n1 = n1 + 1    |...immediately
        )
```

```
$ mark1 count
Press s to restart
s
1.0000000000
2.0000000000
3.0000000000
4.0000000000
5.0000000000
6.0000000000
7.0000000000
8.0000000000
9.0000000000
10.000000000
Press s to restart
s
10.000000000
11.000000000
tape [0:12] execute [0:03]      total [0:15]
```

3.2. A prime numbers program

This program find the primes up to a given number N by the sieve method. This involves storing the numbers up to N in a vector and then *crossing off* all the multiples of 2, 3, 4, 5 ... up to square root of N. The code sequence demonstrates the Autocode *vectoring* facility, its data input from tape using the **I** instruction, and also the standard function square root using the function **F1()**.

```
|     PRIME NUMBERS UP TO N
|
|     Abhir H. Bhalerao
|     March, 1985
|
|     prime numbers
|     by the sieve method
|
      1     n2 = I              |all primes up to input value
            n4 = F1( n2 )       |sieve out multiples up to the root of n2
            n1 = 2              |index variable
      2     vn1 = n1            |loop to store numbers up to n2 in vector
            n1 = n1 + 1
            j2, n2 > n1
|
            n3 = 2              |sieve multiple initially 2
            n1 = n3             |begin at index
      3     n1 = n1 + n3        |increment index by multiple
            vn1 = -1            |sieve out multiple from vector
            j3, n2 > n1
            n3 = n3 + 1         |next multiple
            n1 = n3             |start index
            j4, n3 > n4         |all multiples up to square root value
            j3
|
      4     n1 = 2              |loop to print out...
      5     j6, 0 > vn1         |...if not crossed off
            *vn1 = vn1
      6     n1 = n1 + 1
            j5, n2 > n1
|
      5     H
|
            (j1
            25
            )
```

```
$ mark1 primes
Press s to restart
s
2.0000000000
3.0000000000
5.0000000000
7.0000000000
11.000000000
13.000000000
17.000000000
19.000000000
23.000000000
Press s to restart
s
tape [0:42] execute [0:33]     total [1:15]
```

3.3. Program errors and diagnostics

This is a good point to present some *incorrect* pieces of code to demonstrate the error handling of *mark1*.

3.3.1. Errorprone!

This program is a real disaster!

```
|
|     test out error handling
|
      0     n0 = n0 + 1      |illegal label & index
            v0 = v0 + 1
            l4               |illegal token
            H
            (j0
            )
```

```
$ mark1 errorprone
Label out of range at line 4
Index out of range at line 4
Index out of range at line 4
Index out of range at line 5
Index out of range at line 5
error token detected :l at line 5
syntax error
[6]   l4
Press s to restart
s
Too many errors! Execute abandoned
tape [0:06] execute [0:00]     total [0:06]
```

3.3.2. Illegal vector access

This program will demonstrate a run time out of range.

```
|
|     test out run time range checking
|
      1     n1 = 1
            n1 = n1 - 1
            vn1 = 0          |index out of range??
            H
            (j1
            )
```

```
$ mark1 indexcrash
Press s to restart
s
Index out of range at line 7
tape [0:08] execute [0:01]     total [0:09]
```

3.3.3. Undefined labels

An undefined label is only discovered at run time.

```
|
|     undefined label check
|
          H          |halt to prove it is not 'compile' time
          (j1
          )

$ mark1 labelcrash
Press s to restart
s
Undefined label:1 at line 5
tape [0:02] execute [0:00]     total [0:00]
```

3.4. A summation program

This program reinforces some of the techniques used in the primes program and also shows some further mathematical functions to perform the *raise to a power* operation.

The data is set up to calculate the sum of squares from 1 to 5.

```
|     SUM OF nth POWER OF A SET OF NUMBERS
|
|     Abhir H. Bhalerao
|     March, 1985
|
|     program to print sum of nth power of input numbers
|
      1     n1 = I            |power of sum
            n2 = 1            |index (v1 and v2 used later)
      2     n2 = n2 + 1
            vn2 = I
            j2, vn2 != 0      |input data until end marker
            n3 = n2 - 1       |store last index
            n2 = 1
            v1 = 0            |sum variable
      3     n2 = n2 + 1
            v2 = F4( vn2 )    |log x
            v2 = n1 x v2      |n log x
            v2 = F3( v2 )     |exp ( n log x ) = x power n
            v1 = v1 + v2      |add to sum
            j3, n2 != n3      |for every input number
            *v1 = v1          |print sum
            H
|     data format is: nth power, data, zero

            (j1               |data follows
            2
            1
            2
            3
            4
            5
            0
            )

$ mark1 sum
Press s to restart
s
55.000000000
Press s to restart
s
tape [0:32]  execute [0:07]     total [0:39]
```

3.5. Brooker's example from 'Simple Machine' notes

The example given by Brooker to illustrate his coding scheme is presented below. Since the long list of results are meaningless I have instead listed the source for an equivalent C program. Both programs: the Autocode source being run by *mark1* and the C source being compiled by *cc* (with the **-lm** loader option to include the maths library routines), give identical results.

It is interesting to note that the C source, having been written some 30 years after the Brooker version, does not seem any more elegant. However the time for the run is greatly increased; about 1.5 seconds of CPU time on the VAX compared with an estimated 7 mins and 51 seconds on the Mark I. This is an improvement of more than 300 fold and it doesn't take into account the time for output onto punched paper tape!

```
|
|     Brookers Autocode program for the Mark I
|
      2     v1 = 40          |(p1 = 40)
            v2 = 30          |(p2 = 30)
            v3 = 40          |(p3 = 40)
            v4 = 10/156.4
            v5 = v2/v3       |(p2/p3)
            v6 = F4(v5)
            v7 = 2/7  |      2/7
            v8 = v7xv6       |(p2/p3)
            v9 = F3(v8)
      1     v10 = v2/v1
            v11 = F4(v10)
            v12 = v7xv11     |      2/7
            v13 = F3(v12)    |(p2/p1)
            v14 = 1 - v13
            v15 = 1- v9
            *v16 = v14/v15   |forms and prints alpha
            v17 = F1(v16)
            v18 = v17xv9
            v19 = v18/v13
            v20 = v16xv19
            v21 = 1 - v20
            v22 = 2xv21
            *v23 = 1.3 + v22      |forms and prints re
            v24 = 1 - v19
            v25 = 2xv24
            *v26 = 1.3 + v25      |forms and prints rm
            v1 = v1 - v4          |adjusts p1
            j1, v1 > 29           |tests for last cycle
            H                |halt
            (j2
            )                |starts programme.


#include <stdio.h>
#include <math.h>

/* Abhir H. Bhalerao */
/* March 1985 */
/* C version of Brooker's Mark I Autocode example */
/* (compile with -lm loader) */
main()
{
      double p1,p2,p3,p23,p21,a,ap;

      p1 = 40.0;
      p2 = 30.0;
      p3 = 40.0;
      p23 = p2/p3;
      p23 = exp((2.0/7.0)*log(p23));

      do {
            p21 = p2/p1;
```

```
            p21 = exp((2.0/7.0)*log(p21));
            printf("%20.11f0,a = (1.0-p21)/(1.0-p23) );
            ap = sqrt(a)*p23;
            printf("%20.11f0, 1.3+2*(1.0-ap/p21) );
            printf("%20.11f0, 1.3+2*(1.0-(a*ap)/p21) );
            p1 = p1 - (10.0/156.4);

        } while (p1 > 29.0);

    }
```

3.6. An example using the T instruction

This is a small piece of code to demonstrate the **T** instruction. It shows how further instructions can be added to those already present in memory. Note that there are two *interludes* , the second is needed to restart execution.

```
|
|code to demonstrate the tape facility
|
      1     *n1 = 10
            T
      2     H
            (j1
            *n2 = 20  |these two instruction are loaded
            *n3 = 30  |...in when T is executed
            )
            (j2       |a second jump to restart
            )

$ mark1 tape
Press s to restart
s
10.000000000
Press s to restart
s
20.000000000
30.000000000
tape [0:10] execute [0:01]     total [0:11]
```

3.7. Matrix inversion program

This a large piece of code and uses some user software to implement a *subroutine* facility, which was a drawback of the original Autocode.

This program uses the Gauss-Jordan technique for matrix inversion. The basics points of this method is that it is fast and lends itself to implementation into a program because it involves simple *row* operations and the algorithm is straight forward. The algorithm in synopsis is:

```
for every pivot in the matrix
   divide the pivotal row by the pivot
   zero the other elements in the pivotal column
      by: taking the element times the pivotal row
            from the elemental row itself
```

All the *row* operations have to be repeated on a corresponding *identity* matrix. When the algorithm is completed for every *pivot* the original matrix becomes and *identity*

matrix, with all *pivots* being unity and all other elements zero, and the original *identity* matrix becomes the *inverse* matrix.

The algorithm breaks down if a particular *pivot* is zero. There are techniques to continue by swapping rows and columns, but the program given does not use them and simply terminates at this point. This second step can also be ignored if a *pivotal columns element* is already *zero.*

The algorithm shows clearly that *subroutines* are required and a method to represent matrices using a vector.

The subroutine mechanism is implemented using a *stack* , a *stack pointer* and a special piece of *return* code. The *stack* begins at the last available *variable* and works down in memory. The *stack pointer* is held in a an *index.*

A subroutine *call* is simply a jump to a section of code e.g to input the matrix to be inverted. But before this call is made a *return label* is put on to the *stack* and the *stack pointer* adjusted. At the end of the subroutine there is a jump to the special *return* code. This code unstacks the *return label* and then conditionally make the appropriate jump back into the *calling* routine. For example:

```
|
|code to show implementation of 'subroutines'
|stack starts at top of memory
|stack pointer is in n18
|

      1     n18 = 4999      |set up stack pointer
            n18 = n18 - 1   |decrement stack pointer ready for next label
            vn18 = 2        |stack the return label
            j3              |jump to 'subroutine'
      2     ...             |some more code perhaps
            ...
            ...
            H
            j99             |jump to end of program
|
|our subroutine labelled 3
|
      3     ...             |some code
            ...             |some more code
            ...
            j4              |this jumps to our 'return' code
|
|return from subroutine code
|
      4     v0 = vn18       |unstack return label - make sure v0
                            |...is not being used else where
            n18 = n18 + 1   |increment stack pointer
            j2, v0 = 2      |jump back to 2 if 2 unstacked
            j5, v0 = 5      |jump back to 5 if 5 unstacked
            ...
            ..              |more jumps for every return label
            j99             |error - default to end of program
|
|
      99    H               |the end of the program
            (j1
            )
```

Having established a *call* and *return* mechanism what about parameters? Parameters could also be stacked, but in the inversion program there are sufficient *indices* and *variables* available to simply keep every thing global, although it is quite hair raising to keep track of all the variables used!

In the program a special subroutine has been set aside to implement the matrix array required. This routine take the *base* address of the matrix and given the **i** and **j** coordinates of the element required, calculates the actual address of the element and returns this.

MATRIX INVERSION USING THE GAUSS-JORDAN METHOD

Program written for the Manchester Mark I
using Brooker's Mark I Autocode, of 1955

Abhir H. Bhalerao
Department of Computer Science
University of Warwick
March, 1985

MAIN PROGRAM

```
100   n18 = 4999        |start of stack
      n18 = n18 - 1
      vn18 = 1          |n18 holds the return label
      j6                |jump to input procedure
1     n18 = n18 - 1     |decrement stack pointer
      vn18 = 2
      j7                |form identity matrix
2     n18 = n18 - 1
      vn18 = 3
      j8                |invert input matrix
3     n18 = n18 - 1
      vn18 = 4
      j9                |output results
4     j99               |end of program
```

INPUT PROCEDURE

```
parameters:     none
local   :   index n4 = i, n5 = j
return      :   sets A = n1, N = n2, s = n3

6     n1 = 5            |A = M, start address of matrices
      n2 = I            |input N, size of matrix
      n3 = n1           |s = M
      n5 = 1            |j = 1
15    n4 = 1            |i = 1
14    n18 = n18 - 1
      vn18 = 16
      n3 = n1           |s = M
      n6 = n4           |set parameters [i,j]
      n7 = n5
      j10               |n = array[i,j]
16    *vn17 = I         |input element
      n4 = n4 + 1       |columns of matrix
      j14, n2 >= n4
      n5 = n5 + 1       |rows of matrix
      j15, n2 >= n5

      j38               |return
```

```
FORM IDENTITY MATRIX

parameters:      index M = n1
local   :    index i = n4, j = n5
return       :    none

7     n5 = 1
18    n4 = 1
17    n18 = n18 - 1
      vn18 = 19
      n3 = n2 x n2
      n3 = n3 + n1    |s = I  start of Identity matrix
      n6 = n4
      n7 = n5
      j10
19    vn17 = 0  |zero element
      j20, n4 != n5
      vn17 = 1  |unity element i = j
20    n4 = n4 + 1
      j17, n2 >= n4
      n5 = n5 + 1
      j18, n2 >= n5
      j38

INVERT INPUT MATRIX

parameters:      none
local   :    index i = n4,  variable pivot = v4
return       :    none

8     n4 = 1          |i = j = 1
22    n18 = n18 - 1
      vn18 = 24
      n3 = n1         |s = M
      n6 = n4
      n7 = n4
      j10         |find pivot index
24    v4 = vn17 |store pivot

      j99, v4 = 0       |can't deal with zero pivots!
      n18 = n18 - 1
      vn18 = 25
      n12 = n4
      j11         |divide row j by pivot

25    n18 = n18 - 1
      vn18 = 21
      n13 = n4
      n14 = n4  |set parameters
      j12         |zero other column elements

21    n4 = n4 + 1
      j22, n2 >= n4
      j38
```

```
DIVIDE ROW J BY PIVOT

parameters:     index j = n12
local   :   index i = n8, variable pivot = v4
return      :   none

11   n8 = 1          |i = 1
29   n18 = n18 - 1
     vn18 = 30
     n3 = n1         |row operation on M
     n6 = n8
     n7 = n12  |parameter j
     j10
30   vn17 = vn17/v4|row[j] = row[j]/pivot
     n18 = n18 - 1
     vn18 = 39
     n3 = n2 x n2
     n3 = n3 + n1    |row operation on I
     n6 = n8
     n7 = n12  |parameter j
     j10
39   vn17 = vn17/v4|row[j] = row[j]/pivot
     n8 = n8 + 1
     j29, n2 >= n8
     j38

ZERO COLUMN
parameters:     index i = n13, j = n14
local   :   index k = n8
return      :   none

12   n8 = 1          |k = 1
32   j34, n8 = n14   |ignore pivotal row j
     n18 = n18 - 1
     vn18 = 33
     n3 = n1         |s = M
     n6 = n13  |i
     n7 = n8         |k != j
     j10
33   j34, vn17 = 0   |ignore if already zero
     v2 = vn17|set x for next procedure
     n18 = n18 - 1
     vn18 = 34
     j13         |parameters already set
34   n8 = n8 + 1
     j32, n2 >= n8
     j38

TAKE AWAY X * ROW J FROM ROW K
parameters:     variable x = v2 index j = n14, k = n8
local   :   index i = n9 variable v = v1
return      :   none

13   n9 = 1          |local i
35   n18 = n18 - 1
```

```
        vn18 = 36
        n3 = n1          |s = M
        n6 = n9
        n7 = n14  |j from previous procedure
        j10
36      v1 = vn17 x v2  |v = array[i,j] * x
        n18 = n18 - 1
        vn18 = 37
        n3 = n1          |s = M
        n6 = n9
        n7 = n8          |parameter j set to k
        j10
37      vn17 = vn17 - v1|perform subtraction of row elements
        n18 = n18 - 1    |repeat for matrix I
        vn18 = 40
        n3 = n2 x n2
        n3 = n3 + n1     |s = I
        n6 = n9
        n7 = n14  |j from previous procedure
        j10
40      v1 = vn17 x v2  |v = array[i,j] * x
        n18 = n18 - 1
        vn18 = 41
        n3 = n2 x n2
        n3 = n3    + n1 |s = I
        n6 = n9
        n7 = n8          |parameter j set to k
        j10
41      vn17 = vn17 - v1|perform subtraction of row elements
        n9 = n9 + 1
        j35, n2 >= n9
        j38
|
|   OUTPUT RESULTS
|   parameters:      none
|   local   :    index i = n4, j = n5
|   return       :    none
|
    9       n5 = 1
    27      n4 = 1
    26      n18 = n18 - 1
            vn18 = 28
            n3 = n2 x n2
            n3 = n3 + n1     |s = I
            n6 = n4
            n7 = n5
            j10
    28      *vn17 = vn17     |print array[i,j]
            n4 = n4 + 1
            j26, n2 >= n4
            n5 = n5 + 1
            j27, n2 >= n5
            j38
|
|   ARRAY[ i, j ]
```

```
|
|     parameters:      index s = n3, i = n6, j = n7
|     local  :   none
|     return       :   index n = n17
|
      10   n6 = n6 - 1       |i - 1
           n7 = n7 - 1       |j - 1
           n17 = n2 x n7     |n = N * (j-1)
           n17 = n3 + n17    |n = s + N*(j-1)
           n17 = n17 + n6    |n = s + N*(j-1) + (i-1)
           j38
|
|     RETURN PROCEDURE
|
|     parameters:      index stack_pointer = n18
|     local  :   variable return_addr = v3
|     return       :   none
|
      38   v3 = vn18 |hold return label
           n18 = n18 + 1  |increment stack pointer
           j1, v3 = 1 |jump back to return address
           j2, v3 = 2
           j3, v3 = 3
           j4, v3 = 4
           j16, v3 = 16
           j19, v3 = 19
           j21, v3 = 21
           j24, v3 = 24
           j25, v3 = 25
           j28, v3 = 28
           j30, v3 = 30
           j33, v3 = 33
           j34, v3 = 34
           j36, v3 = 36
           j37, v3 = 37
           j39, v3 = 39
           j40, v3 = 40
           j41, v3 = 41

           j99          |default to end of program
|
|
|     END OF PROGRAM
|
      99   H            |last instruction to be obeyed
           (j100        |start program DATA follows
           )
```

The above program requires the data to be in the form: size of matrix N to be inverted, followed by N squared elements. This data can be tacked on to the source code in the *interlude* but it is better to use the **-d** option when invoking *mark1* and have the data in the file *m.data*.

The invert program will also echo the numbers of the matrix to be inverted before printing out the results. For example take the 3 x 3 matrix:

```
1    0    0
4    5    6
7    0    9
```

To invert this the data file *m.data* has to be:

```
3
1
0
0
4
5
6
7
0
9
```

note the first number is 3 to indicate the size of the matrix to be inverted. When the program is run:

```
$ mark1 -d invert
Press s to restart
s
1.0000000000
0.0000000000
0.0000000000
4.0000000000
5.0000000000
6.0000000000
7.0000000000
0.0000000000
9.0000000000
1.0000000000
0.0000000000
0.0000000000
0.1333333333
0.2000000000
-0.1333333333
-0.7777777778
0.0000000000
0.1111111111
Press s to restart
s
tape [6:02]  execute [5:59]     total [12:01]
```

The inverse of the input matrix is then:

```
1     0     0
1/15  1/5   -1/15
-7/11 0     -1/11
```