Development of a method for checking syntax in a conversational computing system

N W. Bennett
University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/theses

Recommended Citation

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
Development of a method for checking syntax in a conversational computing system

N.W. Bennett B.Sc., F.A.C.S.

Ph.D. Thesis

1974
ABSTRACT

Three systems in the field of conversational computing for small machines have been developed. The main theme has been the exploitation of the full duplex operation of a typewriter to validate the syntax of all expressions and statements entered by the user. A high degree of interaction between the user and his program has been a secondary theme.

The first of these systems, called ACTIV-E, was implemented on a D.E.C. PDP-8 computer. This system explored how a computer program might be used to validate the syntax of characters typed by the user. The language implemented by ACTIV-E was a very limited subset of FORTRAN, and once the concept of echo-checking was developed, its implementation was straightforward.

The second system is known as ACL-NOVA and was implemented on a D.G.C. NOVA computer. This system was developed by the author and is being used on several installations.

The last system has the generic name of XB and has been implemented on the D.E.C. PDP-11 computer. Three types of expression have been provided in a language structure designed to demonstrate the syntax and recovery techniques. A concept of a "set of allowable next characters" is developed. A broader view of the validity of expressions is explored. Very flexible means are available for the user to interact with the system and his program.
CONTENTS

INTRODUCTION 1
HISTORICAL PERSPECTIVE 3
MAIN THEME AND OUTLINE OF RECOVERY PROBLEM 6
A SOLUTION TO THE RECOVERY PROBLEM 8
SET OF ALLOWABLE NEXT CHARACTERS 17
OTHER ERRORS 20
FACILITIES FROM INTERNAL CHECKING 28
SYMBOL TABLE 29
SYNTAX 31
EDIT FACILITY 34
INTERACTION WITH PROGRAM 36
CALL and RETURN statements 38
THE PRESENT SYSTEM AND THE FUTURE 40
CONCLUSIONS 41
ACKNOWLEDGEMENTS 42
REFERENCES 43
Appendix A A Guide for Users of the XP system
Appendix B The Development of the ACL system
Appendix C The UCC FASTEXT system
Appendix D An Assembler/Simulator for the PDP-11
Appendix E Use of the XB paper tape system
Appendix F Sample flowchart - Arithmetic expressions
Appendix G Partial Source of the XP system
INTRODUCTION

The first system in this project was a simple conversational computing facility called ACTIV-8 developed for a PDP-8 computer in 1968. A feature of ACTIV-8 was a dynamic syntax checking scheme which exploited the properties of a full duplex typewriter. ACTIV-8 was readily accepted and a second system called ACL was developed, based on the experience gained from the earlier system and developing the theme of dynamic syntax checking. The ACL system was implemented on a NOVA computer by Dr. P.L. Sanger, an officer with the Australian Atomic Energy Commission after the candidate changed employment from the A.A.E.C. to Compunet Ltd. in August, 1969. The Director of the A.A.E.C. has acknowledged that the candidate was the originator of the ACL-NOVA project and of the major concepts behind it.

A paper describing these two systems has been published and this is reproduced as Appendix B. Although these two systems have been superceded by a third system, called XB, they were essential steps in the development of the XB system and the candidate requests that they be considered as part of this thesis.

The last system has been developed solely by the candidate whilst employed with Compunet Ltd. The XB system is being described here for the first time. The theme of syntax checking has been developed to include real, logical and string expressions.
A mechanism to exercise the syntax algorithms has been developed. Initially, this was intended solely to validate the routines used to check the syntax, but has been retained in the final system to fill an advisory role. A user may explore all the possible paths available at any place in expressions or statements.

Although the XB system has been mainly developed to explore and demonstrate the syntax checking, it is a powerful conversational system in its own right. A guide for users of the system has been included as Appendix A. A paper tape copy of the XB system is provided as part of this thesis. The candidate would be pleased to demonstrate this system, but if this is not possible, Appendix E gives instructions for its use on a 16K PDP-11. The body of the thesis builds upon the users guide and the demonstration of the system and it is concerned only with the broad philosophy behind the development and implementation of the XB system.

The XB system consists of over eight thousand statements and the most relevant part is reproduced as Appendix G. A set of sample flowcharts illustrating the techniques used is given in Appendix F. A cross assembler and simulator for the PDP-11 on a Univac 1108 has also been written in conjunction with the XB project and is described in Appendix D.

This document has been prepared using the UCC FASTEXT
HISTORICAL PERSPECTIVE

The I.B.M. 1620 was one of the standard machines of a decade ago. By modern standards this machine is very slow and cumbersome; it had a 20 microsecond cycle time per character and took about 500 microseconds to add two ten digit fields together. At the same time, a great virtue of the 1620 was its slowness and therefore, accessibility. One felt one could stop the machine and inspect the value of variables; the time spent in doing so would not greatly inconvenience the next user. At the A.A.E.C. time was booked in lots of half an hour.

After this, machines became faster, operating systems were used to speed the flow of jobs and the machine became remote from the user. Now there is a trend to use the larger, faster machines in a conversational mode in order to regain the advantages which existed a decade ago with slow machines. Rather than have just one user stopping and starting a whole machine, several users are able to have this ability via timesharing and typewriters.

The three systems in this Ph.D. project are part of this revolution; an attempt to regain the conversational facilities which existed in a crude form 10 years ago.

A second trend over recent years has been the use of machines in an "impedance matching" mode, or, to have
different machines linked together with each performing the
tasks to which it is best suited (in particular, input and
output tasks). A classic example is the UCCmpuert computer
network (Ref. 1). The central machine is a Univac 1108 which
has a high internal speed and also a lot of hardware devoted
to performing sophisticated instructions in as few machine
cycles as possible; for example, a floating point multiply
is performed in 3 cycles. All input and output to slow
peripherals such as card readers and printers are controlled
by a PDP-8 which is a fast mini-computer with a minimal
instruction set. At present 14 card readers and printers are
attached to this PDP-8 (2 on site and 12 remote). Cards are
read into the PDP-8 and buffered into large blocks. These
are sent into the 1108 at drum transfer speeds; several
hundred thousand characters per second. Similarly, output
from the 1108 is sent to the PDP-8 and then to printers at
much lower speed.

Another example in the same network is the FASBAC
system (Ref. 2). Here 3 machines are arranged in a
heirarchy. A PDP-8 computer is used to receive and send
characters from and to low speed asynchronous terminals
(110, 134.5 and 300 baud). The PDP-8 is also used to
automatically echo any character it may receive from a
keyboard back to its matching teleprinter. Whenever a
complete line is accumulated it is transmitted into a
larger, more sophisticated PDP-9 computer. In this regard
the PDP-8 acts as a pseudo card reader to the PDP-9.

The PDP-9 system is capable of building files, editing files and performing character oriented (vs. arithmetic) tasks. This document was formatted by the FASTEXT system on the PDP-9 computer (Ref. 2, Chapter 5).

A complete program may be built in the PDP-9 as a file and sent across to the 1108 for processing.

Thus there is a hierarchy:

a) The PDP-8 handles characters and builds lines.
b) The PDP-9 handles lines and builds files.
c) The 1108 handles files.

This approach of treating input and output in as simple a fashion as possible is usually carried over to single processor conversational systems. In this regard, the three systems in this project are a radical departure from the norm. The XB system treats input from the keyboard as a most important event; one requiring high priority and perhaps a lot of computing. The behaviour of the system could not be duplicated with a simple machine to receive and echo characters and a more powerful machine to perform arithmetic and logic.

It will be demonstrated that this approach taken in the XB system is worthwhile; that although the techniques employed are inefficient, they are effective.
MAIN THEME AND OUTLINE OF RECOVERY PROBLEM

The major theme of this project has been to explore the possibilities of utilizing the full duplex behaviour of a typewriter communicating with a computer, mainly to check syntax.

Firstly, the term full duplex has several usages and its meaning in this context should be clarified. On a teletypewriter attached to a D.E.C. PDP-11 the keyboard and teleprinter are two distinct devices. When a character is keyed, the character is transmitted (actually the ASCII representation) to the computer and an indication given in the machine. The character is not printed automatically by the action of striking the keyboard; it is the responsibility of the program in the computer to do this.

Some machines, such as the I.B.M. golf-ball typewriters are half-duplex; each character keyed by the user is typed locally without reference to the computer system. Others, such as the Memorex 1280 are switch selectable between half-duplex and full-duplex.

This degree of freedom of the program leads to the possibility of checking the incoming character for validity in context and, in particular, syntax. If a character is entered which cannot possibly be correct (such as two decimal points in a number) the simplest way of indicating the error is to not echo the character onto the teleprinter.

In this way all syntax errors can be prevented. This
leads to two savings. Firstly, no messages relating to errors of syntax are needed (a saving of space) and, secondly, interpretation of statements may assume that no errors of syntax are possible (an increase of speed). Each character is checked dynamically for syntax as it is keyed, rather than considering a complete line after the line has been entered, or a complete program after a program has been entered.

The main problem with syntax checking is one of recovery. It is not very difficult to devise a scheme to check the syntax of (say) an arithmetic expression. The real problem lies in being able to continue with the same line after rejecting an error (i.e., for the program to recover to the state in which it was before the invalid character).

Another aspect of the recovery problem is the deletion of characters already entered. Recovery after an invalid character has been entered means regaining just one state; deleting several characters means regaining any one of several states. If twenty characters had been entered in a line, deletion could call for recovery of any one of these states. Suppose the state of the machine is $S_n$ while waiting for the $n$th character of a line. The problem is to recover to state $S_1$, $S_2$ etc.
A SOLUTION TO THE RECOVERY PROBLEM

The first attempt, in ACTIV-8, to program such a behaviour avoided the problem of recovery to a large extent. At each stage precautions were taken to recover the state Sn if the nth character proved invalid. The simplest way of doing this was to place restrictions on the syntax. For example, all variables were two characters in length; a letter followed by a number. If the second character entered was not a number the program simply looped until it received one.

With hindsight the solution to the recovery problem in the XB system seems straightforward, but many schemes were devised and discarded in the process. The evolution of the solution lay only partly in meeting the requirements already outlined. A major part of the evolution of the recovery techniques for the XB system lay in adapting the solution to meet new ideas which looked as though they should be able to be incorporated easily. If they proved difficult, the recovery technique was modified until it met the new requirements while still satisfying the old. During this evolution the recovery technique grew both more powerful and simpler; always a good sign! To aid the explanation of the recovery mechanism, etc., reference will be made to the source listing of XB (Appendix G). Such references will be to the decimal serial numbers in the left hand column. Thus a reference to the subroutine to accept arithmetic
The first decision was to have a recovery mechanism which had the minimum impact upon the program checking the syntax of the incoming characters. The simplest way of doing this was to have a single location in the machine to which a jump could be made (1434). Because this jump could be made from anywhere in the syntax, the only common point from which recovery could be made was the start of the line. Obviously this meant storing all the input characters in a buffer (INPUT) and having a knowledge of the number of characters which had been accepted (a pointer, INPA1, pointing to the next free position in INPUT). During recovery from the start a pointer (L) was needed to indicate which character should be taken next. Once the pointer L had caught up to the pointer INPA1 recovery was complete and the next character was needed from the user via the keyboard. Hence to recover after an invalid character, INPA1 was decreased by one, L reset to point to the start of INPUT and the program worked its way through all the characters at internal speed. Similarly, to backspace one character, INPA1 was decreased by two.

Another problem was the typing of characters. In the normal case, valid characters had to be typed back to the teleprinter once they were accepted, but during recovery the program had to treat them as internal data and certainly not to be typed once more. The backspace using the backslash (\) expressions will be to 1116-1349.
required the line to continue, while the multiple backspace (<nn) called for a new line and the reduced line to be retyped. The control of the typing of characters was handled by another pointer (INPBl) to the INPUT buffer. If no extra typing was needed INPBl was altered to match INPA1. If a new line was needed, the new line was taken and INPBl reset to the start of INPUT. In this way the line was retyped during recovery.

Some output from the system may be suppressed (Topic 7.2, page A-47, Appendix A) by the user giving an unsolicited character from the keyboard. Output which may be suppressed is indicated by a variable, OPTION, set to non-zero (5443, 5467). The output is by-passed in the subroutine which prints individual characters if OPTION is non-zero and the keyboard flag is set (4040-4045). This approach is used to ensure that the program follows the same path whether the printing is active or not. The alternative of breaking into the program stream was not seriously considered as data may be modified while the output is being prepared.

ACTIV-8 simply did not echo the input character to indicate an error. On a teletype this is quite satisfactory. There is an audible difference between a key pressed followed quickly by a character being typed, and a key being pressed followed by silence. However, on a video terminal it proved to be most unsatisfactory. There is no sound involved in the character appearing on the screen and one loses the
sense that there is something very wrong. To overcome this the bell character (control-G) was introduced. If the character was not valid, the bell character was echoed instead. This restored the impression of an error, even on the video. At first the bell was initiated by the error routine directly and was satisfactory for syntax errors, but characters were rejected for other reasons (see the EDIT facility, Topic 4.7, page A-16, Appendix A) and it seemed a pity not to use the same technique for all invalid characters, irrespective of the cause of their rejection. This was eventually solved by sounding the bell if two successive calls were made for an input character without a call for a character to be typed; if the first character was not echoed, something was wrong.

This emphasised the problem of speed of recovery. When one is simply not echoing, the program gives no signal that it has completed recovery. When the error routine signals the bell, it does so when the error has been detected and before recovery. When the bell is sounded by the second call for an input character, it does so after recovery is complete. So that the user does not lose rapport with the system, the time difference between echoing a valid character and sounding the bell must be minimal. This is the main reason for the programmed check (e.g. 1116-1349, arithmetic expressions) for syntax errors rather than the more compact but slower methods available with a table-
driven scheme. However, a compromise is used in the acceptance of keywords (e.g. 4723-4729).

In essence, the technique is very simple. Suppose the string

1.34E+

has been entered. The only character valid as the next character is a number (1204, and 1976-1983 via 882 and 75-84).

If the next character is a number, it is stored in INPUT (1201 via 891 and 75-84 to 3741-3745), typed (3746-3751) and another character solicited from the keyboard (3753-4036). If it is not, an exit is made (1207 via 960 and 85-92 to 1434-1438) to the error routine. The pointer INPA1 is decreased by one, and a jump made to one of several restart points (say 5049, see 5047). From here the program is able to progress from state S1 at the start of the line to Sn (say) where it was just before the invalid character. Note that (5045, 5046) INPA1 and INPB1 are not reset, thus forcing the program to use the internal characters and not retype them. The bell character is sounded (3841-3846) since no character has been typed (4047) between input characters.

The next point of interest is the expansion of a character in some situations (Topic 7.1, page A-46, Appendix A). An example of this is the compulsory space at the end of the keyword "TRUE". If a non-space is entered instead of the terminating space, it is saved in INPUT (1531-1539) so
that it is treated as the next character and a space substituted. The value of HBECHO (which controls the bell resulting from two successive calls for an input character) is increased so that the bell sounds unless there are two characters echoed between calls for an input character (1538). This is to allow for the non-space being an invalid character. Similarly the expansion of a special character which has been associated with a compound statement via an ASYNCH statement (4361-4386, Topic 5.18, page A-39, Appendix A) is performed by expanding the string into INPUT, increasing INPA1 to include the new characters, but leaving the pointers L and INPB1 so that the extra characters are both picked up and typed.

An expansion of particular interest is the null statement number (5321-5338). Here a character is used to indicate a null statement number of three spaces, and the most obvious choice is the space character itself. The difficulty arises when a recovery is made. If the space character causes an expansion of one space into three, care must be taken that the system does not end up with five spaces after recovery. This would most likely cause a syntax error, another recovery and a tight loop. This was solved by distinguishing between keyboard characters and recovery characters (5330). A new problem arises where a line containing a null statement number is punched onto paper tape via the SUSPEND statement. One must punch all three
spaces on the paper tape or the typing of the line would not
match the line printed via a LIST. However, the first space
must not cause an expansion either, else one would end up
with five spaces instead of three. This was solved by
inserting a special non-printing character (control-7, 5321)
in front of the three spaces to suppress the expansion.

Because the syntax of incoming characters is checked,
there is an opportunity to store additional information to
simplify and speed the process of evaluation. An output
buffer (OUTPUT with a pointer SL) receives a copy of the
input characters as well as special characters to act as
sign-posts for the evaluation. For example, a bracket for
subscripts (1226-1231), a bracket for grouping (1131-1134)
and a bracket enclosing an argument for a function (1320-
1335) are all stored with a different special character
imbedded in front of them. Similarly a unary plus (1121-
1125), an exponent plus (1191-1195) and an operator plus
(1171-1175) are all stored with different indicators. During
evaluation the special characters are sought rather than the
original characters. Since this "seeded" string must be used
to recover the original string at some future time (for
program listings, etc), the seed characters must be easily
distinguishable. Since the POP-11 has characters of eight
bits and ASCII is a seven bit code, the sign bit of each
caracter was used.

In a sophisticated syntax scheme it was sometimes
necessary to follow one of two possible paths. For example

"IF B AND"

looks like a continuing logical expression such as

"IF B AND B2 THEN TYPE A3"

but it could turn out to be

"IF B AND3=14"

i.e. a variable AND3 rather than a logical operator AND.

To cope with this problem it is necessary to identify logical ends to an expression. After a variable or literal, one may have a logical end of expression or an operator. If the characters turn out not to be a continuation of the expression, a "mini recovery" must be made to the last logical end of expression and an exit taken. Note that a full recovery is neither necessary or possible. It proved to be sufficient to save and restore the two pointers L and SL.

In the example above, see 1709-1718, 1735-1737, 1984-1990.

The input characters must be recovered from INPUT and new seeds stored into OUTPUT.

It was important that the syntax check graduated from particular to general, otherwise the general syntax could accept a stream and never pass it on to the particular. Also care must be taken that another routine is available to carry on the stream. If not, a later routine may reject a character already echoed to the teleprinter; resulting in a recovery and a tight loop. For example the characters

"ACCEPT" could turn into an arithmetic or logical expression...
up until acceptance of the fifth character (variables can be four characters long). If a deviation is made from the keyword in the first five characters, a mini recovery is made from the start of the line and the string treated as an expression (4577-4578, 4723-4729, 4970-4972). In the case of an arithmetic or string expression no operator is more than one character long, and it is not necessary to ever perform a mini recovery as for a logical operator such as "AND". However care must be taken not to reject a character if a logical end of expression has just occurred, but rather pass the character back to the more specialized syntax routine which may be calling the expression. For example 64C would have the first two characters accepted as arithmetic expression but the last (the C) passed over to the more specialized syntax routine for judgement (1178-1186, 1288-1293). If the calling routine were the arithmetic routine looking for an operand for a function, the C would be rejected; any extra character must be a closing bracket (1341-1342). But if it is a TYPE statement looking for an operand it would be accepted (4680-4683). Note that even the comma separating the operands in a TYPE statement is inspected by an expression routine, found not to conform to the syntax for that expression and passed back to the TYPE statement. In many cases a general expression is required (operand for TYPE, expression statement) as opposed to a particular
type of expression (subscript value, input during execution of an ACCEPT statement). A special routine (1815-1561) is needed to determine the type of expression, to accept the expression from the keyboard and to inform the calling syntax routine. The only problem here is that either the arithmetic or string expressions may become logical expressions by use of a relational operator. For example

\[ 14*(A1+3) \]

is an arithmetic expression, but

\[ 14*(A1+3) \text{ EQ } 6 \]

is a logical expression. The mini recovery technique is well suited to this task. If the initial character does not indicate a logical or string expression, an arithmetic expression is taken from the keyboard (1917). If the characters after the arithmetic expression (1919-1948) are a relational operator, a mini recovery is made to the start of the arithmetic expression and a logical expression is sought (1949-1961).

\[ \text{SET OF ALLOWABLE NEXT CHARACTERS} \]

All the techniques described above were well established when the concept of the set of allowable next characters was introduced (see Topic 6, page A-44, Appendix A). This was born largely out of a concern to check all possible characters coming in from the keyboard in any situation. By working systematically through the keyboard
one could classify characters into not acceptable if they signaled an error or acceptable if they were echoed. In order to continue with the next test character, the particular situation must be regained. If the trial character was rejected, recovery was automatic. If the character was accepted, recovery had to be forced by means of a backspace.

In some cases this was meaningless. Suppose the trial character was a colon (treated as a carriage return) which terminated a line which in turn deleted a stored statement. Here it is not possible to reverse this action by a backspace! This was solved by restricting all the syntax routines to merely gather information. If the character was being treated as a test character, this information could be ignored (5369-5375), or if it was being treated in the normal way, it could be acted upon (5057-5065). With this modification, the automation of this "thrashing" of the system was practical. The sixty four ASCII characters (excluding control characters, lower case characters and '"', '"' and "\") were tested. Each character in turn was placed in the "next character" position and followed by several special characters (5508-5514) in case of expansion. The number of internal characters was set to a very large number (5515), typing was suppressed (5516) and a recovery initiated. A jump to the error routine will be made, either because the test character was invalid or because the
special character was detected (3764-3765). A simple test may be made (5523-5525) to determine this.

A string of allowable characters is built up internally (5482-5484, 5499-5502). If this string is empty (i.e. no characters were valid) something is wrong with the XB system. If more than one character is valid, a fresh line is taken, the set of characters typed, and a recovery made with the original line retyped.

If only one character is valid, it is echoed in the users line, and a fresh scan made for the set of allowable characters in the next position. If this new scan results in only one character, this character is also typed on the users line and the process repeated. If more than one valid character is found, control is returned to the user to continue keying in characters. Note that when the system is following a keyword (the only real situation repeatedly returning one valid character) sixty four test characters (and 64 recoveries) are made for each character typed for the user.

A simple extension of this technique is used to print out a two dimensional array of valid characters for the next two positions.

Lastly, an option exists so that the system will print the set of allowable next characters whenever the user makes an error. This prompts the user with the set of characters from which he must choose. If he was typing a keyword, where
there is only one valid next character, the system will complete the keyword. That is, if the user were typing an UNLESS keyword and had typed the U, then keyed an M instead of an N, the system would type an N instead of the M and continue with L, E, S, S, and the closing space.

An important point is that this facility (a syntax exorciser) is completely independent of the syntax routines. No changes are needed to the exorciser if the syntax is changed.

OTHER ERRORS

Other errors may be checked in the same fashion as the syntax errors. The single subscript of an array must be in the range 0-65535. Hence in the string

A(70000)

the subscript is clearly out of range. Other examples are one of two subscripts (0-255), string subscripts (1-31 at worst), references to sequence or statement numbers (0-999; GO TO, CALL, etc) and the argument of the SCF function (not negative).

In the example above, the closing bracket should be rejected. It is tempting to reject the last zero which changes the subscript from 7000 to 70000, but this would not allow the valid expression

A(70000-X)

It is only when the user indicates the subscript is
completed that any decision can be made about rejecting the terminating character.

Clearly, it is necessary to evaluate the arithmetic expression. This is not possible if reference is made to a variable. There are two cases:—

a) The statement is to be stored. The variable may assume any value when the program is run.

b) The statement is an immediate statement. At first one is tempted to use the current value of variables, but there are two difficulties:—

1) $X=-1$:
   
   $A(X)+X=1$:

2) $X=-1$:
   
   $X=1; A(X)$:

In the first case, $X$ will have a value of 1 during the "real" evaluation. Any decision based on the -1 value ignores the possibility of $X$ being redefined before the subscript is evaluated. The second case is similar.

The checking outlined above has been implemented in the XB system as an option (See Topic 4.13, page A-20, Appendix A). In the case of

$A(70000$

a pointer to the left hand end of the subscript is saved. When an attempt is made to end the subscript, the expression is copied into an alternate area. The subroutine to
reference the symbol table is conditioned so that reference to a variable causes the evaluation to be aborted. After verifying the syntax of the terminating character, an attempt is made to evaluate the subscript. If the attempt is aborted, the incoming character must be accepted. If the evaluation can be completed, a check on the range of the subscript may be made and if it is outside the range, the character may be rejected.

The time taken to evaluate the arithmetic expression can be a problem. In practice a subscript like

\[ A(1+2+3+4+5-4*4) \]

can be evaluated and the closing bracket rejected without noticeable delay. However a subscript like

\[ A(\text{SQRT}(25)-\text{SQRT}(24)) \]

takes time to evaluate (particularly with decimal floating point software!) and a considerable delay occurs before the closing bracket is accepted. The problem would be reduced if hardware floating point were available. If a real time clock were available, the attempt to evaluate the expression could be aborted (similar to the symbol table reference) after (say) 40 ms.

If the string

\[ A(\text{SQRT}(25)-\text{SQRT}(24))++ \]

were entered, the subscript would be evaluated once to determine whether the closing bracket should be accepted and again during recovery after rejecting the second plus
character. This second evaluation is unnecessary and, once
the problem is recognised, can be avoided. However, care has
to be taken that the process of determining the set of
allowable next characters is not upset (the two cases are
very similar).

In the case of an immediate GO TO with an argument
which can be evaluated, a check is made that the sequence or
statement number is defined. In the case of a statement
number, a check is made that it is unique.

The same idea may be used to monitor the FOR statement.
In the statement

FOR X=1,3,-.5 A(X)=0:

the space after the -.5 could be rejected. Here all three
parameters of the FOR loop may be evaluated and the endless
loop recognized.

This exercise of rejecting out-of-range subscripts etc,
was at first thought to be too difficult to implement.
However refinements of any technique depends on the changes
needed to incorporate more facilities into a method. (A
question of difficult problems vs. inadequate solutions.)

A check on the value for a SQRT function may be made
with the double set of allowable next characters feature.
For

```plaintext
SQR(2E4-1E+>
$0123456789$
SQR(2E4-1E+>)
```
the only numbers which may follow the exponent plus character and which may themselves be followed by a closing bracket are 0 through 4. The numbers 5 through 9 yield negative values for the argument and hence the closing bracket is not allowed.

A forerunner to rejecting nonexistent sequence numbers was the implementation of an immediate statement to delete sequence numbers and change statement numbers (Topics 5.4 and 5.5, pages A-23 to A-26, Appendix A). A rigid syntax structure dictated that the sequence number should be in cols 1, 2 and 3. When the carriage return is given, a check is made that a statement has been stored with that sequence number.
Another possibility (not implemented at the time of writing) is to check on the uniqueness of the statement number of a statement being entered for storage. This check could be made when the second character of the statement number is entered. One complication is that the current statement may be replacing one with the same statement number. If the uniqueness check were to fail, this possibility would have to be explored (by comparing sequence numbers as well). Another consideration is that the user may be well aware that he is duplicating a statement number and his next move would have been to delete the old statement. This places an unnatural restriction on his order of making this change.

Strings longer than 31 characters could be detected at input. That is, in the string

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ123456
```
the 6 is an error because it is the character which makes the string exceed the maximum length. In the situation above the only character which is valid is a closing quote. Similarly the string

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ123456
```
is too long because of the 6. In this case the system would have to recognise that two string literals, with known lengths, were being concatenated. The string

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ8H18123456
```
is also too long because of the 6. Here, although the value
of H1 is unknown at the time of receiving the 6, the string is too long even if H1 is a null string (with a length of zero).

The magnitude of numbers may not exceed \(0.999999E+39\). If a number

\[1.E+64\]

were entered the system could continuously attempt to convert the number into internal form. In this case the 4 could be rejected.

The above procedure is too simple. One must also allow

\[99999999999999999999999999999999999E-20\]

which is a valid number when the exponent is included, but to a simple minded scheme the fortieth digit is an error and would be rejected. It is only when the user indicates that the number is complete that a check could be made. In the example of \(1.E+64\) the 4 is the last character since the syntax will not allow more than two exponent characters. In

\[99999999999999999999999999999999999*\]

the * indicates the end of the number and if the maximum magnitude is exceeded, the * could be rejected.

Another problem appeared in the sample program of Topic 5.19, page A-41 of Appendix A. The condition of validity is that

\[N \geq 1 \text{ and } LE 9 \text{ and } Po(N) = Q \text{ and } Mt\]

should be true. That is, the position must be on the board and not taken. In practice, this does not achieve the
desired effect and the condition must be broken in two to be

\[
\text{IF } N \text{ LT } 1 \text{ OR } \text{GT } 9 \text{ GO ERR}
\]

\[
\text{IF } P0(N) \text{ NE MT } \text{GO ERR}
\]

The reason for this is that if \( A \) has a value of 21, say, then \( P0(21) \) is out of range, an error exit taken, an error message given and the statement aborted. But the expression is identical to

\[
\text{FALSE AND } B
\]

Here, the value will be FALSE irrespective of \( B \) being TRUE, FALSE or undefined. The error status is really indicating that \( P0(N) \) \( \text{NE MT} \) is undefined. In this case, it makes no difference to the final result.

Similarly,

\[
\text{TRUE OR } B
\]

could be overcome by allowing a third value for logical (i.e., undefined). If an undefined were generated during an evaluation, an error message should be stored. If the final value was defined, no harm would be done; if undefined, then the first error message should be given. Note that

\[
\text{TRUE AND UNDEFINED}
\]

should merely yield a partial result of UNDEFINED. There is no need to generate an error message, since clearly the error is due to some earlier condition and an error message exists.

This extra value (UNDEFINED) for a logical variable is
similar to the floating point values of undefined, plus infinity and minus infinity on the CDC 6600 computer.

The identity

FALSE AND B is always FALSE

is used in optimising code in the I.B.M. FORTRAN compiler for the 360 computer. There it is used to avoid calculations rather than avoid error messages. This rationalization has not been implemented at the time of writing.

FACILITIES FROM INTERNAL CHECKING

Such a large and closely-woven system as the XB system on the PDP-11 requires extensive checking to make sure all possibilities are considered and to assist in isolating errors. Many of these checks and facilities have provided features for the user.

The set of allowable next characters started as a method of checking that the syntax analysis was reasonable and there were no tight loops. In the final system this takes on an advisory role; giving the user an indication of the characters which may be used in a particular situation, either on request or when an error is made.

Because the syntax analysis, if working correctly, guarantees the validity of the statements it is tempting to have no checks in the evaluation stage of the interpreter. However any error not detected in the syntax would go undetected and result in a mysterious error far removed from
its cause. For this reason many checks have been included, although far less than would be needed if these checks were the first line of defense against syntax errors. These checks have been included as assembly options (49,4911).

The fine trace feature, giving an operation by operation analysis of the evaluation of an expression (Topic 4.4, page A-12, Appendix A) is another example. This was introduced to check the evaluation of expressions. As the features within expressions became more sophisticated (such as the elided operand in a relational) the fine trace feature followed. When the possibility of invoking the step-by-step analysis when an error occurred was realized the feature was retained in the system, rather than discarded like many other implementation aids.

SYMBOL TABLE

A very simple symbol table is used for the XB system. Two PDP-11 words are used to hold variables (4 characters for simple variables, 2 characters and 16 bits for singly subscripted arrays, and 2 characters and two by 16 bits for doubly subscripted arrays) and two words for data (32 bits).

A sequential search is used to reference variables. The table is not sorted, but rather entries added or altered as assignments are made.

A single area is devoted to storing statements and strings and the symbol table. The variable length data
starts at the low end of the area and increases upwards, while the symbol table (with fixed length data) starts at the high end and increases downwards. This allows optimal use of the area.

A device used in looking up variable length information is of interest. Special characters are used to indicate the start of each item. If the total length of variable length data were \( n \), then \( n \) searches need to be made if the data is stored as true variable length with dense packing. A worthwhile improvement in speed may be achieved by padding out the variable length data so that the special characters are always a multiple of \( m \) characters from the start of the table. This results in a \( m \)-fold increase of speed at the cost of \( (m-1)/2 \) characters per item. A multiple of 4 is used in the XB system resulting in a four fold increase in speed for a 7.5\% increase in space assuming twenty characters per item on average. (Actually a 370\% increase in speed because of the increase in size.)

In order to repeat statements which end in error (Topic 4.4, page A-12, Appendix A) it is necessary to reverse the effect of any assignments made during the statement. An attempt was made to do exactly this, but showed all the signs of being unwieldy. A simpler statement of the problem was to reset the symbol table to its state at the start of the statement. A solution to this equivalent problem proved to be both simple and elegant. The symbol table was
partitioned into a permanent symbol table and an incremental symbol table (initially empty). The permanent symbol table remained unchanged and assignments were made into the incremental symbol table. Any reference was tried first in the incremental table and then, if unresolved, in the permanent table. At the successful conclusion of a statement (or statements) the incremental table was consolidated into the permanent table. If the statement ended in error, the incremental table was simply abandoned.

SYNTAX

The syntax of expressions and statements in the XB system is not complex. The main concern has been exploiting the full duplex typewriter characteristics. Sample flowcharts for arithmetic expressions are given in Appendix F.

There are three types of data which must be recognised. It is desirable to be able to determine the type by the first character to avoid complexities in the recovery problem. Literals identify themselves in this way naturally (e.g. `quote` for string, `space` for logical and `number` or `decimal point` for real). The first character of variables identifies their type. (An excellent precedent exists in FORTRAN for integer variables!) Initially it was planned to commence logical variables with an `L`, but the function LOG would cause difficulties. So instead the letter `R` (for
Boolean) was chosen. Similarly string variables were to start with S but because of SQR and SIN became H (for Hollerith). When strings were introduced, the subroutine depth counter, then SRET, was changed to ZRET, but not changed back when string variables started with H rather than S.

One construction of interest is the elided relational. Here

\[ A \text{ EQ } 2 \text{ OR } A \text{ EQ } 3 \]

is simplified to

\[ A \text{ EQ } 2 \text{ OR } 3 \]

and

\[ N \text{ LT } 1 \text{ OR } N \text{ GT } 99 \]

to

\[ N \text{ LT } 1 \text{ OR } GT 99 \]

This simplification is only possible where a full relational has been given. For example

\[ A \text{ EQ } 2 \text{ OR } 3 \]

is allowed, but not

\[ 2 \text{ OR } 3 \text{ EQ } A \]

A knowledge of past relationals is maintained. It is set (1758) when a full relational is completed and reset whenever a logical variable is used or an assignment made. Also the simplification may not carry from real to string. For example

\[ A \text{ EQ } 2 \text{ OR } H \text{ EQ } ^{\sim}2^\text{ OR } A \text{ EQ } 3 \]
may not be simplified to

\[ \text{A EO 2 OR H EO "2" OR 3} \]
even though the syntax analysis and expression evaluation are well defined. These restrictions are quite arbitrary and designed to prevent the user from confusing situations.

There are two possibilities when recognizing the use of this simplification. In the first, where just the left-hand operand is elided, a relational operator follows a logical operator (1655-1659, 1687-1693). In the second a relational operator does not follow the left-hand arithmetic operand of what at first sight appears to be a complete relational (1667-1678, 1679-1686).

I am indebted to Mr. Peter Movlan of Newcastle University for bringing a similar construction to my notice. The expression

\[ \text{A1 GT A2 AND A2 GT A3} \]

may be simplified to

\[ \text{A1 GT A2 GT A3} \]

This simplification is attractive (especially its implementation) but I have resisted the temptation of including it in the present X8 version.

Another construction of interest was that of selective assignment. The construction

\[ \text{IF B THEN A=X ELSE C=X} \]

may be rewritten as

\[ \text{IF B THEN A= ELSE C=X} \]
Note that

\[
\text{IF } B \text{ THEN } A \text{ ELSE } C=X
\]
does not mean the same as the previous expressions because in the XB system assignments like \( C=X \) may be treated as an operand. This was considered as being of insufficient benefit for the simple, yet tedious problems involved in implementation.

EDIT FACILITY

The edit facility (Topic 4.7, page A-16, Appendix A) enables the user to change characters already entered in the current line or stored previously.

The change sequence involves nominating a substring in the current line. A subroutine is used to count the number of times the substring exists within the current line. If no match is found, then the change is impossible. The last character entered is either the first character of a substring or a subsequent one (to get this far, the substring must previously have existed within the current line). The last character must then have been in error and should be rejected.

If the substring exists within the current line the character may be echoed. This different criterion to echo caused the decision to give the warning bell character after a second call for an input character rather than a syntax error.
If the substring exists once only (i.e. is unique) within the current line there is a unique next character. This character may be nominated by the user via the \textasciitilde character. Once the user has given sufficient of the substring for it to be unique, the rest of the string may be easily nominated by this means. Note that the system cannot advance automatically as with the set of allowable next characters facility; the user must have complete control over the length of the substring he is nominating.

It would be possible to check the syntax of the characters making up the second substring. For example, in the following case

\begin{verbatim}
2+C\times4<E
E>/C/C)
\end{verbatim}

the \textasciitilde could be rejected just as it would be in the sequence

\begin{verbatim}
2+C)
\end{verbatim}

However to change

\begin{verbatim}
2+C\times4
\end{verbatim}

into

\begin{verbatim}
(2+C)\times4
\end{verbatim}

would mean inserting the opening bracket first and then the closing bracket. The approach finally taken was to allow the user full rein to change the current line and to perform a syntax check when the user indicates that all chances have been made. The current line is presented to the syntax as an
internal stream, and a test made to see if an error would be indicated before the end of the string. If all is well the line is retyped for the user while the system makes a complete recovery.

If a syntax error is indicated, a second scan is made exactly like the first, but with the typing indicator (INPüt) set to type the valid characters as they are accepted. A dollar character is given to make any space character at the end of the line visible. The user is returned to the edit mode with the complete line available.

In this way the user is able to make changes in the line and the system verify that the new line still has valid syntax.

INTERACTION WITH PROGRAM

The XB system has been written to provide the maximum interaction of a user with his program. Because of this it is essential that interplay between interpretation of stored statements and the user should be as flexible as possible.

When interpreting a stored program, a record must be kept to indicate the current statement and the next. In the XB system it is possible to stop a running program, add extra stored statements and continue. Because this may involve moving statements in storage the only practical means of recording a statement is by sequence or statement number.
INTERACTION WITH PROGRAM

Where one statement follows sequentially from another, the address of the next statement is held between statements in exactly this form; e.g. the statement after (sav) 120. Suppose at the time this next statement is 130 and at this point the user regains control (Topic 5.17, page A-38, Appendix A) and inserts a statement with sequence number 124. If the sequence number of the next statement is calculated too early an incorrect return would be made to 130 rather than 124.

Similarly when a CALL or GO TO is made, the target must be described exactly as such. Suppose in a stored program a statement with a sequence number of 160 had a statement number of 23 and the program had just interpreted a GO TO 23 when the user regained control. The user must be free to reassign the statement number to another statement; sav 220. If a translation from statement number to sequence number had been made too early, a return to 160 would have been made instead of the correct 220.

A breakpoint facility (Topic 5.8, page A-27, Appendix A) has been provided for the user to regain control at a specified point in his program. It is necessary to allow the breakpoint to act either before or after the statement to cover the case of a GO TO or CALL.

The user may regain control from the automatic mode (Topic 5.17, page A-38, Appendix A) with provision to return control either to the next statement at full speed with ":".
to the next statement for one statement only with ";" or to any point in the program with a GO TO.

The user may break into the running program and execute a single nominated statement (Topic 5.18, page A-39, Appendix A). Provision has been made to allow this nominated statement to be a CALL statement.

A trace feature is available for variable assignments (Topic 5.9, page A-29, Appendix A) and for listing statements as they are interpreted. Another trace feature (not yet implemented at the time of writing) will be an assignment trace for a particular variable covering all statements rather than the present assignment trace which covers all variables within a particular statement.

CALL and RETURN statements

The CALL statement and its companion, the RETURN statement, provide the means of using other statements in a program and then continuing interpretation with the statement after the CALL statement (Topic 5.14, page A-36, Appendix A). Information must be saved during the execution of a CALL statement so that the RETURN statement may send control back to the correct point. Once this information has been stored, the CALL statement may be, and is, treated as a GO TO statement. This information is held in a relative form. Suppose the CALL were made from sequence number 120 when the next sequence number was 130. If the user regained
CALL and RETURN statements

control between the CALL and the RETURN and introduced a statement with sequence number 124, the RETURN will be made to 124; if the RETURN address were stored as 130 the system would not be able to follow this change in the program by the user.

In the XB system this information is saved in the symbol table. An array, ZR, holds a stack of return locations and a variable, ZRET, counts the depth of the stack. In the case of a stored CALL a single entry is added to ZR, and ZRET increased by one. For an immediate CALL three entries are needed. The first two are the variables holding the sequence numbers of the previous and next statements. The third is an indicator so that the RETURN can interpret the stack values correctly. A CALL may also be made by an ASYNCH statement, (Topic 5.18, page A-39, Appendix A) and three stack entries are also needed.

The use of the symbol table allows recursive CALLs to be made. The values are held in floating point form under specific variable names so that the user may have access to them. For example, in a recursive subroutine a stack of variables may be referenced using the depth pointer as a subscript as in A(ZRET). The user is not permitted to redefine these variables as this could lead to mysterious errors. However, the exact state of the program has to be stored on paper tape via the SUSPEND statement and be used to reconstruct the program (Topic 5.11, page A-32, Appendix A).
A). This problem was solved by allowing an assignment character after ZR(N) or ZRET only if a non-printing password (control S, control Y, control S) is given (1248-1272). This allows the value of ZRET to be given with the password included and to look just like any other immediate statements to restore the value of a variable.

THE PRESENT SYSTEM AND THE FUTURE

The present system has been written to explore the possibilities of exploiting the full duplex behaviour of a typewriter in a conversational computing situation. The standard mathematical functions such as SIN, COS etc. have not been implemented to leave space for enhancements on the main theme. The PDP-11 used to implement the XB system has no floating point hardware so a software floating point system was written. With a system allowing a user such ease of communication, constant conversion errors became a problem with a binary internal representation so a decimal representation was used instead. However it would seem sensible to use any hardware floating point and to have an equivalent software system so that they may use the same mathematical functions.

The present XB system has been written to handle only one typewriter. Because of the architecture of the PDP-11 the present program can fairly easily be upgraded to handle several terminals.
Other mediums than paper tape are now available such as cassette and disc. It would be useful to be able to store and retrieve programs from these mediums at much higher transfer rates.

CONCLUSIONS

The concept of dynamic syntax checking via a full duplex typewriter has been developed and demonstrated. The priority given to attending characters entered from the keyboard has been reversed so that checking may be performed quickly enough that it is transparent to the user. This means that any expression or statement entered into the system has valid syntax.

A very powerful, but simple and flexible, method has been evolved to handle a complex language structure. The concept of a set of allowable next characters follows simply and easily from this. A broader view of the validity of expressions has been presented.

Flexible means have been demonstrated in the XB system for a user to interact with his program. A method of saving sequencing information which can cope with dynamic program changes has been used. A facility to repeat exactly any statement ending in error has been included.

The two earlier systems, ACTIV-8 and ACL-NOVA, have been well accepted. The ACL-NOVA system is in use in seven research and university centres. The XB system, which is of
wider scope, should enjoy a similar usage.

ACKNOWLEDGEMENTS

I wish to acknowledge the use of language concepts from BASIC, FORTRAN, ALGOL and even COBOL. The language is a vehicle to demonstrate the concepts and techniques which have been developed. The edit facility owes a great deal to the change command in the EDIT subsystem of FASBAC. In FASBAC this command is not even recognized until end-of-line is given; in the XB system incoming characters are checked character-by-character for validity.

I wish to acknowledge the support and encouragement of many people over the period of the project.

The atmosphere and facilities provided by the Australian Atomic Energy Commission during the development of the first two programs is appreciated and also the encouragement of Dr.'s J.L. Symonds and D.J. Richardson during this period. Dr. P.L. Sanger was responsible for implementing the ACL system on the NOVA computer.

UCCompunet have provided the facilities needed to develop and implement the XB system. I am grateful for the continuing support of the management; in particular, of Mr.'s Alex Smith, Theo Boehme and Fred Johnson. The encouragement of Mr. Ian G. Nichols has been very much appreciated.

The consistent encouragement of Prof. Justin Keane has
ACKNOWLEDGEMENTS

been a major factor in the completion and recording of this project.

REFERENCES

1. 1108 Processing Concepts - UCC 1030.40
   UCCompunet Ltd., 23 Cleg St., Artarmon 2064
2. FASBAC Users Guide - UCC 320.50
   UCCompunet Ltd., 23 Cleg St., Artarmon 2064
Appendix A

A Guide for Users of the XB system
## Appendix A

### CONTENTS

1. Introduction A-1
2. Conventions for input A-2
3. Expressions A-3
   3.1 Assignments A-3
   3.2 Brackets A-4
   3.3 Conditional clauses A-5
   3.4 Logical expressions A-6
   3.5 Real expressions A-7
   3.6 String expressions A-8
   3.7 Relational expressions A-9
4. Facilities and statements in the user mode A-10
   4.1 Expression statements A-10
   4.2 TYPE statement A-11
   4.3 Compound statements A-12
   4.4 Statements which end in error A-12
   4.5 FOR statement A-14
   4.6 IF statement A-15
   4.7 Edit facility A-16
   4.8 SYMBOLS statement A-18
   4.9 CLEAR statement A-18
   4.10 Undefined variables A-19
   4.11 Statement suffix A-20
   4.12 Comments A-20
   4.13 A further validity check A-20
5. Building a stored program A-22
Appendix A

5.1 Sequence and statement numbers A-22
5.2 References to statements A-22
5.3 LIST statement A-23
5.4 DELETE statement A-23
5.5 Changing statement numbers A-25
5.6 EDIT statement A-26
5.7 RUN statement A-26
5.8 Pause Before and Pause After facility A-27
5.9 Assignment trace A-29
5.10 List trace A-32
5.11 SUSPEND statement A-32
5.12 GO TO statement A-34
5.13 ACCEPT statement A-34
5.14 CALL and RETURN statements A-36
5.15 CONTINUE statement A-37
5.16 STUP statement A-37
5.17 Regaining control from automatic mode A-38
5.18 ASYNCH statement A-39
5.19 Sample program A-41

6. Allowable next characters A-44

7. Appendix A-46

7.1 Places where characters are expanded A-46
7.2 Suppressing output A-47
7.3 Functions A-48
7.4 Properties of statements A-49
7.5 Invoking options A-50
1. Introduction

The XB system is a conversational computing system for the DEC PDP-11 computer. Just like BASIC, FCRTRAN or ALGOL, XB provides a facility to implement an algorithm.

All communication with the XB system is via a teletypewriter or similar device. A central feature of the XB system is a method of accepting only those characters keyed by the user which are syntactically correct and this applies to all dialogue between the user and the XB system. A character which is incorrect is simply not typed back to the teleprinter and an audible signal may be given to indicate the error. The user is free to continue typing after an error has been made; by this means the user is protected against trivial typing errors.

There are two modes of operation within the system. The first is a user mode where the user is able to present statements and have them executed immediately, or to enter statements to be stored to build up a program. The second is an automatic mode where the stored program is being executed. In the user mode, the user is in control in that he supplies commands to the system, the system obeys them and control returns to him. In the automatic mode, the program is in control. Statements are fetched from the stored program, the system obeys them and a new statement is fetched. There are flexible means of switching between the two modes. A program may be started, the user remain
control, perhaps inspect the values of variables, perhaps change the program and then allow the program to continue.

There are three types of data: real, logical and string. One and two dimensional arrays may be made with real and logical variables. String variables may be subscripted to access individual characters within a string.

Within the X8 system, expressions may often be used where a constant or variable is required. Subscripts for arrays may be any real expression; input to the ACCEPT statement may be an expression.

The user is able to save the state of a program on paper tape, reload the program at some later time and continue.

2. Conventions for input

Input to the X8 system is a series of lines which are terminated by a carriage return. The visible character colon, `:`, is typed in response. Lines may also be terminated by the colon character itself.

When the system is expecting the user to provide input, a prompt character (or characters) is typed in the initial columns of the line. For input to the user mode the prompt is the "greater than" character or `>".

If an error has been made (such as 1.5 instead of 1.6) characters may be deleted. These deletions take the form `\`. Delete the last character. (Continue on same
Appendix A

line.)

<: Delete the last character. (A new line is taken.)

<3: Delete the last 3 characters. (A new line is taken.)

<12 Delete the last 12 characters (carriage return is not needed, new line taken).

The following commands may also be given.

<£ Enter the EDIT mode (see Topic 4.7) with the present line as data.

<A Abandon the present line and start again.

The system may be initialized at any time by the keyboard character, control-G.

3. Expressions

There are three types of expression in the XB system. These are logical (or boolean), real (or arithmetic) and string. Topics 3.1, 3.2 and 3.3 discuss features common to all these expressions.

3.1 Assignments

An equals character is used as an assignment character in the XB system and in this users guide. A left pointing arrow (ASCII 137) may optionally be substituted at generation of the XB system. An example of a simple assignment for a real variable is

A=2:
The order for performing an assignment is to evaluate the expression on the right of the assignment operator and then define the variable on the left to have this value. In the XB system an assignment is considered to have an overall value and may be used as an operand. For example

\[ 2 + A = 4: \]

has a value of 6. In particular, the ALGOL construction

\[ A_1 = A_2 = A_3 = 0 \]

is available.

Similarly multiple assignments like

\[ A_1 = 3 \times A_2 = 1 + A_3 = 1: \]

may be made with assignments being performed from right to left. In this example A3 would be assigned a value of 1, A2 a value of 2 and A1 a value of 6.

3.2 Brackets

Brackets may be used to vary the normal order of evaluation. For example 3*2+1 yields a value of 7 whereas 3*(2+1) has a value of 9.

The part of the expression to be evaluated first is bounded by the left-most closing bracket and its matching opening bracket. This selection is continued until all brackets have been eliminated. For example the expression

\[ ((2+3) \times 4) + 3 \times (5+6) \]

would be evaluated as

\[ ((2+3) \times 4) + 3 \times (5+6) \]
In the expression  
\[ A(X) = X = X + 1: \]
for an initial value of \( X \) of 2, \( A(2) \) will be set to 3.

3.3 Conditional clauses

A conditional clause of the form

\[ \text{IF b.e. THEN expr1 ELSE expr2} \]

may be used as an operand. The b.e. is a logical expression, and expr1 and expr2 are both logical, real or string. If b.e. is true the overall value of the clause is expr1, else it is expr2. For example

\[ 2 + \text{IF B1 THEN 3 ELSE 4:} \]

is an arithmetic expression,

\[ ^{\text{A}} \& \text{IF N1 EQ 1 THEN } ^{\text{CDE}} \text{ ELSE H3:} \]

is a string expression.

There is a heirarchy within these 3 levels. Conditional clauses are considered first, then brackets and then assignments.
3.4 Logical expressions

There are two logical constants

`TRUE` and `FALSE`.

The spaces, but not the quotes, are part of the constants.

Logical variable names may be 1, 2, 3 or 4 characters long; the first character must be the letter `B` and succeeding characters may be either letters or numbers.

Logical array names may be one or two characters long; the first character must be the letter `B` and may be followed by a letter or a number. There may be one or two subscripts; each of which may be a real expression. If there is one subscript, it may range in value from zero to 65535. Each of two subscripts may range in value from zero to 255.

No declaration is needed for arrays. The single, double and non-subscripted forms of a variable are distinct and may all be used.

Four logical operators may be used;

`NOT`, `AND`, `OR` and `XOR`.

They have their usual meanings and hierarchy.

Examples

>`BT OR BF= FALSE AND BT= TRUE :`  
TRUE

>`BF OR (BT AND BF):`  
FALSE

>`BT XOR IF BT THEN BF ELSE BT:`  
TRUE
3.5 Real expressions

All arithmetic in the XB system is performed with decimal floating point numbers and there are no separate integer numbers. The floating point numbers have a precision of six decimal digits with a range from .1E-39 to .999999E+39 inclusive.

Real variables and arrays are the same as logical variables and arrays except the first character must be a letter other than "B" or "H". (The simple variable ZRET and the single dimensioned array ZR are not available for general use.)

There are five arithmetic operators. They are, in order of hierarchy:-

- power; e.g. A^3
- *, / multiply and divide
- +, - add and subtract

Unary plus and minus are available where -A is interpreted as 0-A, +A is 0+A and the above hierarchy is obeyed. The construction 2^3^4 is not permitted, nor is any disguise of this construction.

A set of arithmetic functions such as SQR are available. A list of these is given in the Appendix.

Examples

>2*3/4+5:
6.5

>SQR(SQR(4))+1:
2.41422

>2+A=IF B= TRUE THEN SQR(5) ELSE 99:

4.23607

3.6 String expressions

Any sequence of characters enclosed in quotes may be used as a string. The quote character itself may be indicated by a sequence of two quotes. Examples are 

"CAT"

"THE QUICK BROWN FOX"

"WON""T"

All string variables start with the letter "H" and may be followed by a letter or a number. There are no string arrays as such, but any string variable may be subscripted to indicate reference to individual characters within the string.

There is a single string operator "&", which indicates concatenation. Strings may be concatenated to form longer strings.

An individual character within a string may be redefined with an assignment. A string may be extended by defining its next character. If a string has not been defined, the first character may be defined. A character within a string may be eliminated by redefining it as the null character. In this case the string is reduced in length by one with the higher characters being moved down one
position.
Examples are

\[ H3 = ^{AB \& C}^\sim : \]
\[ H3(2) = ^2^\sim : \]
H3 would now be \(^{AZC}\).
\[ H3(4) = ^E^\sim : \]
H3 would now be \(^{AZCE}\).
\[ H3(3) = ^\sim : \]
H3 would now be \(^{AZE}\).

3.7 Relational expressions

The validity of a relationship between real expressions has a logical value. As such they may be used as an operand in a logical expression. For example \(1 \text{ EQ } 1\) is true, whereas \(2 \text{ GT } 3\) is false. The standard six relational operators between real expressions are available. These are

\[ ^\text{ EQ }^\sim , ^\text{ NE }^\sim , ^\text{ LT }^\sim , ^\text{ GT }^\sim , ^\text{ LE }^\sim \text{ and } ^\text{ GE }^\sim . \]

The spaces, but not the quotes, are part of the operator. The relational operators are considered in the hierarchy to be after arithmetic operators, but before the logical operators.

Similarly the relationships \(\text{ EQ}\) and \(\text{ NE}\) are meaningful between strings and may be used to yield a logical value. For example

\[ 1+2 \text{ GT } 0 \text{ AND } ^A^\sim \text{ EQ } ^B^\sim : \]
would be false.
Where a logical expression has two adjacent relationals with an identical left hand side, the left hand side of the second relational may be elided. For example

A EQ 2 OR A GT 9:

becomes

A EQ 2 OR GT 9:

Similarly, if two adjacent relationals have identical left hand sides and identical relational operators, then both the left hand side and the operator of the second relational may be elided. For example

A EQ 2 OR A EQ 3:

becomes

A EQ 2 OR 3:

The rule for interpreting the simplified expression is that the missing left hand operand and, if necessary, the missing operator, are copied from the nearest preceding full relational.

4. Facilities and statements in the user mode.

4.1 Expression statements

An expression may be used as a statement. During execution the value of the expression is calculated and unless the leftmost part of the expression consists of a variable assignment, the value of the expression is typed. For example, the value of
1+2: 
A & H1 = BC:

would be typed, while the value of 
A1 = 2 + 3: 
BT = TRUE:

would not. In FORTRAN the arithmetic expression above is known as an arithmetic statement.

4.2 TYPE statement

The TYPE statement is for displaying information on the teletypewriter. There are 3 classes of operand.

a) The value of an expression (real, logical or string) may be typed.

b) Either an absolute or a relative positioning operand may be given. The absolute positioning operand has the form arithmetic expression, "C" and indicates that the next value should be typed at a particular column. If the desired column is greater than the current column, the teletypewriter carriage will be positioned to the column on that line, otherwise a new line will be taken. The relative positioning operand has the form arithmetic expression, "X" and indicates that a number of columns should be skipped.

c) A combination is provided to print an arithmetic value so that its decimal point (or right hand side if there is no decimal point) is positioned in a nominated column.
This takes the form

\[ a.e.1,a.e.2 \]

where the value of the second arithmetic expression has its value printed with its decimal point in the column nominated by the first arithmetic expression.

These three types of operand may be used in a TYPE statement separated by commas. For example

```
TYPE "TEMP. IS ",[15,T1],17C,"DEGREES F":
```

For T1 values of 1.2 and 212 this would give

```
TEMP. IS 1.2 DEGREES F
TEMP. IS 212 DEGREES F
```

4.3 Compound statements

More than one statement may be used on a single line by terminating all but the last statement with ";" (semicolon, space). The last statement must be terminated with ";:" (or carriage return) as usual.

Some statements may not be continued in this way and a list of these is given in the Appendix.

Examples

```
J=J+1; GÔ T0 4;
X=Y=0; P=Q=1:
```

4.4 Statements which end in error

Messages are given for all error conditions. The relevant statement may be repeated with a trace by
responding with a carriage return as the next line. The trace consists of a printout after each operation has been performed. For example

```
>SQR(1+2+3+4+5-4*4):
** ERROR ** SQR(-1) RANGE ERROR
>: 
SQR(1+2+3+4+5-4*4)
SQR(1+2+3+4+5-16)
SQR(3+3+4+5-16)
SQR(6+4+5-16)
SQR(10+5-16)
SQR(15-16)
SQR(-1)
** ERROR ** SQR(-1) RANGE ERROR
```

If any assignments were made during the statement(s), their effect is reversed.

Example

```
>A=-3:
>A=A+2; SQR(A): (Value of A changed from -3 to -1)
** ERROR ** SQR(-1) RANGE ERROR
>A: 
-3 (Value of A back to -3)
```
4.5 FOR statement

A FOR statement has the form

FOR J=J1,J2,J3 SI; S2:

where J is a real variable and J1, J2 and J3 are real expressions. The latter part of the statement is interpreted for J=J1, J=J1+J3, etc until J is greater than J2. A limited selection of statements may be used in the latter part of the statement. An indication of these is given in the Appendix.

A check is made that the loop is finite. The increment J3 may be elided, in which case it is chosen to be 1 or -1 as J2 is greater or less than J1.

At the normal completion of the statement, the index variable is not defined, but if a GO TO exit is taken out of the loop, the index variable retains its value.

Examples

>FOR X=1,3 TYPE [6,X],[12,SQR(X)]:

1   1
2   1.41422
3   1.73205

>FOR X=3,1 TYPE X:

3
2
1

>FOR X=250,300 A(4,X)=0:

** ERROR ** A(4,256) HAS A SUBSCRIPT OUT OF RANGE
4.6 IF statement

An IF statement has been implemented with the form

IF b.e. THEN S1; S2 ELSE S3; S4:

If the logical expression b.e. is true the compound statement S1; S2 is interpreted, else the compound statement S3; S4 is interpreted.

The ELSE part of the statement may be elided to give

IF b.e. THEN S1; S2:

In this case the "THEN " characters may also be elided to give

IF b.e. S1; S2:

Examples

IF X=X+1 LE 8 THEN GO TO 3 ELSE GO TO 1:

IF A EQ 2 OR 3 TYPE "ERROR":

An alternate IF statement of the form

IF(b.e.) S1; S2:

is also available.

The FOR statement may be used in either the THEN or the ELSE part of an IF statement, and the IF statement may be part of a FOR statement.

An arithmetic IF statement of the form

IF(r.e.)n1,n2,n3

has been implemented. The n1, n2 and n3 are real expressions representing statement numbers or sequence numbers. If the value of the r.e. is less than zero a GO TO n1 is made. Similarly if the r.e. is zero or greater than zero a
Appendix A

A TO n2 or n3 is made. Any of n1, n2 or n3 may be elided. The brackets around the r.e. may also be elided.

Examples

IF(A) 21,22,23
IF C 3,,5
IF X(I) 3,4,Y(I)

4.7 Edit facility

Lines which are being typed in by the user may be edited at any stage by entering "<E". A new line is taken with a prompt of "E>". The following responses are available:

a) The character "P". The line is typed.

b) A change sequence consisting of a special character, a string of characters in the line to be changed, the same special character to delimit the string, a string of characters to replace the first and the delimiting special character to terminate the line.

Characters in the first string are dynamically checked for validity. If the string being proposed does not exist in the line being edited, the last character entered is not echoed. If the string entered as the first string is not unique in the line being edited, the special character delimiting the first string is not echoed as a warning. At this point further characters may be entered to uniquely identify the change, or the special character may be entered
again; this time it will be echoed and the first occurrence of the string will be taken.

Once the string to be changed has been uniquely identified in the line to be edited, the character ">" may be used to nominate the next character. At any stage during the change sequence, the character "<" may be used to abandon the line.

c) A carriage return indicates that editing has finished and an exit should be taken. At this point a check is made to see that the syntax is correct up to the end of the line. If the syntax is not correct, the line is typed up to the point where an error was detected (a "$" is given at the end of the line) and the edit mode is retained. The full line is once again available. If the syntax is correct and the line terminated by a carriage return, the line is interpreted without being typed again. If the syntax is correct and the line incomplete, the line is typed again and the user may continue his input.

d) The character "A" causes the line to be abandoned just as if the characters "<A" had been used instead of "<E".

Example

>1.2345<E
E>1.2345 (P entered)
E>/23/.32/ (change 2 characters into 3)
E>1..3245 (P - note two decimal points)
4.8 SYMBOLS statement

The SYMBOLS statement may be used to print the symbol table.

4.9 CLEAR statement

The CLEAR statement may be used to clear variables from the symbol table. If no operands are used, all variables are cleared from the symbol table. If one or more variables are used as operands only those variables will be cleared from the symbol table.

Example

```
>A=1; C=2; D=3:
>SYMBOLS:
D=3
C=2
A=1
>CLEAR A,D:
>SYMBOLS:
```
4.10 Undefined variables

A reference to a variable which has not had a value assigned to it causes an error message to be printed. The user is given the option of assigning a value at that point and having the system repeat the statement.

The system will ignore any detectable errors made by the user in assigning the value and repeat the request.

Example

>SQR(A1):
A1 IS UNDEFINED, GIVE VALUE NOW
U>99:
9.9499
>SQR(ZZ):
ZZ IS UNDEFINED, GIVE VALUE NOW
U>XY:
XY UNDEFINED, TRY AGAIN FOR ZZ
U>9/0:
** ERROR ** 9/0 OVERFLOW, TRY AGAIN FOR ZZ

An exit to the user mode may be made by responding with a null line (i.e. a carriage return).
4.11 Statement suffix

Some statements may have a suffix indicating conditions under which the statement is to be performed. The forms are:

S IF b.e.
S UNLESS b.e.

Examples are:

A=200 IF C GT 0:
GO TO 4 UNLESS B:

A list of the statements which may have a suffix is given in the Appendix.

4.12 Comments

Comments may be inserted by starting any line with "C", space. All characters are valid; except that "", "<", ">" and "\" all retain their usual meanings.

4.13 A further validity check

As well as giving a means of indicating keyboard characters which are invalid for syntax reasons, the full duplex typewriter may be used to prevent some errors of numerical magnitude. Subscripts for arrays must not be negative, and not more than 65535 in the case of single subscripts and 255 in the case of double subscripts. For example, in the lines:

A1(70000)
A2(300,
A3(4,100+8*25)

the last character indicates that the subscript is complete.

An option exists (see Appendix) in the X8 system to attempt
to evaluate the subscript expression. If any reference to a
variable is made it is not possible to complete the
evaluation, the attempt is inconclusive and the system has
to accept the closing character.

If, however, the evaluation can be made, and the value
is outside the legal limits, then the terminating character
can be rejected. Note that in the case of

A2(SQR(25)-SQR(24))

a noticeable time may be taken before the closing bracket is
accepted.

Similarly, the closing bracket in

SQR(1+2+3+4+5-4*4)

would be rejected.

Another example of a numerical bounds check is with
statement and sequence numbers. For example

GO TO 1+2-5

CALL 600+600

are both invalid. The bounds check will reject the character
which signifies the end of the statement or sequence number.

A further check is made with immediate statements to
make sure that referenced sequence numbers are defined and
that referenced statement numbers have been defined, but not
duplicated.
5. Building a stored program

5.1 Sequence and Statement numbers

Every statement stored as part of a program must have a sequence number. The sequence number is a three digit number in the range 100-999 and must be followed by a space in column four. A space in column one has been interpreted as a request for the next sequence number; this will be ten more than the sequence number of the line last stored (note that this means the sequences "NOT", "TRUE" and "FALSE" cannot be used to start an immediate logical expression).

Statements to be stored may have a two digit statement number in the range 00-99. A single digit number may be used such as 9. A statement number should be followed by a space in column seven and the body of the statement in column eight. However, a non-space in column seven causes the compulsory space to be inserted and a space may be used in column five to automatically generate a null statement number.

5.2 References to statements

There are a number of statements which refer to other statements for the purpose of listing, deleting, etc. These statements have a common format for referencing other statements. There are five variations and the LIST statement is used as an example.
1) LIST:
2) LIST N:
3) LIST N1,N2:
4) LIST N1,: 
5) LIST ,N2:

The first form refers to the entire stored program.
In the other four forms, the operand may be a real expression which references a statement by statement or sequence number depending on the value of the expression being in the range 00-99 or 100-999.

The second form references a single statement. The third form refers to a group of statements from N1 through N2. The fourth form has its second operand elided and this is understood to mean from N1 to the end of the program. The last form means from the start of the program to N2.

5.3 LIST statement
A stored program may be typed by the LIST statement. The operands may assume the five forms discussed in Topic 5.2.

5.4 DELETE statement
Statements may be deleted from the stored program. The operands may assume the five forms discussed in Topic 5.2.

A statement may also be deleted by the sequence "sequence number, space, carriage return". If the sequence
number had not previously been defined the carriage return is rejected; a second carriage return causes the line to be abandoned.

Example

```
>110  X=0:
>120  X=X+1:
>130  X=X+2:
>140  X=X+3:
>150  X=X+4:
>160  X=X+5:
>170  GO TO 120:

>LIST:
  110  X=0
  120  X=X+1
  130  X=X+2
  140  X=X+3
  150  X=X+4
  160  X=X+5
  170  GO TO 120

>DELETE 120,130:

>LIST:
  110  X=0
  140  X=X+3
  150  X=X+4
  160  X=X+5
  170  GO TO 120
```
5.5 Changing statement numbers

A statement number may be changed by giving the three digit sequence number of the statement, a space, the new statement number and a carriage return. The new statement number may be either two digits, a digit or a space. Once again, if no statement has been stored with the nominated sequence number, the carriage return is rejected; a second carriage return causes the line to be abandoned.

Example

>110 99 X=1:
>LIST 110:
110 99 X=1
>110 88 :
>LIST 110:
110 88 X=1
>110 3 :
>LIST 110:
110 3 X=1
>110 :
5.6 EDIT statement

A stored statement may be edited with the EDIT statement. Its form is

EDIT N:

where N is a real expression representing either the sequence number or the statement number of the desired statement.

The actions described for the EDIT facility are now available.

5.7 RUN statement

A stored program may be run by the RUN statement. Execution begins with the lowest sequence number. The symbol table is not altered.

Example

>110   X=S=0:
>120   S=S*X*X; IF X=X+1 LE 8 GO 120:
>130   TYPE "SUM IS",S; STOP:
>RUN:
SUM IS 204
CONTROL RETURNED TO USER
5.8 Pause Before and Pause After facility

A facility is available to return control to the user before a nominated statement is executed. The statement has the general form

PB N:

The five forms discussed in Topic 5.2 are available. When a statement, which has been the subject of a PB statement, is reached in the stored program, control is given back to the user. A new line is taken; the sequence number of the statement just completed, the sequence number of the statement next to be interpreted and a prompt of "PB>" are all typed.

At this point all the facilities of the user mode are available plus the following features related to the suspended program.

a) The "?" character will cause the sequence numbers of the previous and next statements to be typed. If the next statement has been nominated by a GO TO or CALL statement, the second operand has the form "J N" where N is the statement or sequence number used in the last statement.

b) The ";" (carriage return) character results in a return to the suspended program.

c) The ";" character results in a return to the suspended program for one statement only.

Once the PB condition has been satisfied, it is removed.
The PB condition may also be removed by the statement
PB ^ N:

The same five variations are available.
A similar facility for regaining control after a statement has been interpreted is provided by a PA statement.

Example

>110   A=1:
>120   C=2:
>130   D=3:
>140   E=5:
>150   F=7:
>160   G=99:
>170   J=0:
>PB 130:
>RUN:
120 130 PB>SYMBOLES:
C=2
A=1
>?
120 130
>;
130 >?
130 140
>:
END OF PROGRAM
5.9 Assignment trace

An assignment trace may be set so that a record is typed as assignments are made to the symbol table.

There are two assignment trace conditions. A global assignment trace may be used to give an assignment trace record for all statements, for selected statements or for no statements.

The global trace may be changed from no statements to selected statements or from selected statements to all statements by

TRON:

Similarly, the global trace may be changed from all statements to selected statements or from selected statements to no statements by

TROFF:

Initially the global trace is set to selected statements. No harm will be done by promoting the trace when it is set to all statements or in demoting it when it is set to no statements. In particular the trace may be set to all statements from any state by

TRON; TRON:
or to selected statements by

TRON; TRON; TROFF:
or to no statements by

TROFF; TROFF:

A local assignment trace may be set on individual
Appendix A

statements by

IRON N:

where the operand may assume the latter four forms of Topic 5.2.

Initially the local assignment trace for each stored statement is off. To reset the local assignment the statement

TROFF N:

may be used where the operand may assume the same four forms as for TRON N.

Another way of looking at the behaviour of the assignment trace is to consider the global trace condition as a variable, say GLTR, having values of -1, 0 and 1. The action of TRON is

IF GLTR LT 1 GLTR=GLTR+1:
The action of TROFF is

IF GLTR GT -1 GLTR=GLTR-1:
The initial value of GLTR is 0.

The local trace may be considered a vector, say LT(N), having values of 0 and 1.

The statement TRON N could be considered to have the action of

LT(N)=1:
and TROFF N to be

LT(N)=0:

As each statement, N, is interpreted, a trace is
printed if

GLTR EQ 1 OR GLTR EQ 0 AND LT(N) EQ 1

Example

>110  A=1:
>120  C=2:
>130  D=3:
>140  E=4:
>150  F=5:
>160  G=99:
>170  STOP:

>TRON 130,150:
>RUN:
130  D=3
140  E=4
150  F=5
CONTROL RETURNED TO USER
>TRON:
>RUN:
110  A=1
120  C=2
130  D=3
140  E=4
150  F=5
160  G=99
CONTROL RETURNED TO USER
>TROFF; TROFF:
5.10 List trace

A trace may be set so that a statement is listed each time it is interpreted.

This trace is similar, but distinct, to the assignment trace. All statements have the keyword LTRCN rather than TRON.

5.11 SUSPEND statement

The state of the program can be saved on paper tape so that a program may be reinstated at some future time.

The full statement is SUSPEND:

Interpretation awaits a keyboard character. This gives the user an opportunity to switch on a paper tape punch in parallel with the teletypewriter. Output consists of all stored statements, the symbol table, statements to reset the state of PB, PA, TRON and LTRON conditions and a RESET statement. The RESET statement allows the program to be resumed by giving "\":" (carriage return).

If the character "P" is used to signal the start of the SUSPEND statement, output is directed to the high speed punch on the PDP-11 console. No output appears on the teleprinter.
Example

>110   A=1:
>120   C=2:
>130   D=3:
>140   E=4:
>150   F=5:
>LTRON 120,130:
>TRON 130,140:
>TROFF:
>LTRON:
>PB 120,150:
>160   STOP:
>SUSPEND:
110   A=1
120   C=2
130   D=3
140   E=4
150   F=5
160   STOP
ZRET=0
LTRON
TROFF
PB 120,150
TRON 130,140
LTRON 120,130
RESET HHHH HHHH
5.12 GO TO statement

A GO TO statement is provided so that a statement other than the next sequential statement may be selected as the next statement to be interpreted. The form is

    GO TO N:

where N is a real expression. If the value of N is in the range 00-99 it is considered to be a statement number; if it is in the range 100-999 it is considered a sequence number and a value outside these ranges is an error.

Note that the general form of a real expression for N includes the FORTRAN assigned GO TO (GO TO I) and the computed GO TO (GO TO A(I)). The statement may be abbreviated to

    GO N:

The GO TO statement may be used to commence automatic execution of a program at any statement.

5.13 ACCEPT statement

An ACCEPT statement may be used in the program to solicit the value of variables from the user. The form is

    ACCEPT N1,N2:

Two types of operand may be used. A hollerith literal may be used to type information on the teleprinter. In this regard the ACCEPT statement is identical to the TYPE statement. The second form of operand is a variable, indicating that a value should be obtained from the user and
assigned to this variable. All three data types are available and the input is required to be the same type. The input may be an expression rather than a constant.

For each variable operand of an ACCEPT statement a fresh line is taken, a prompt of 'A>' is printed and the system waits for an expression to be typed in. At this stage three responses are recognised

a) The ":" (carriage return) returns the user to the user mode with the line containing the ACCEPT statement as the next statement. The effect of any assignments is reversed.

b) The "?" character causes an identifier line to be typed consisting of the sequence number of the ACCEPT statement, the characters "ACCEPT" and the variable name.

c) An expression of the same type as the variable, followed by ":" (carriage return).

Example

>110 ACCEPT "ENTER A1,C2,D",A1,C2,D:
>120 TYPE "RESULT IS ",A1+C2/D; STOP:
>RUN:
ENTER A1,C2,D
A>? (Ask for more information)
110 ACCEPT A1
A>99/37: (Can give an expression)
A>34.5<E (Can edit input)
E>34.5
5.14 CALL and RETURN statements

A CALL statement is provided to enable other statements in the program to be used as a subroutine. The general form is

CALL N:

where N has the same meaning as in the GO TO statement. Control is passed to the new statement.

The RETURN statement is used to return control to the statement after the most recent CALL. There is only one form of the RETURN statement.

RETURN:

An arithmetic variable, ZRET, and an arithmetic array, ZR(N), are associated with the subroutine facility. The CALL statement is similar to

M ZRET=ZRET+1; ZR(ZRET)=M; GO TO N:

where M is the sequence number of the CALL statement and N is the sequence or statement number of the target statement.
The RETURN statement is similar to

\[ Z_{\text{temp}} = ZR(Z_{\text{RET}}); \quad Z_{\text{RET}} = Z_{\text{RET}} - 1; \quad \text{GO TO} \quad \text{SUC}(Z_{\text{temp}}): \]

where \( Z_{\text{temp}} \) is an internal variable and \( \text{SUC}(Z_{\text{temp}}) \) is the next statement after \( Z_{\text{temp}} \). The variable \( Z_{\text{RET}} \) and the array \( ZR \) may not be altered by the user, but may be accessed. The CALL statement may be used as an immediate statement.

Example

```plaintext
>110  X=2; CALL 9:
>120  STOP:
>130 9  TYPE "SQR IS",SQR(X); RETURN:
>RUN:
SQR IS 1.41422
CONTROL RETURNED TO USER
>X=88; CALL 9:
SQR IS 9.38085
```

**5.15 CONTINUE statement**

A CONTINUE statement provides a null operation. More than one statement number may be effectively given to a statement by means of the CONTINUE statement.

**5.16 STOP statement**

The STOP statement is identical to the TYPE statement except that a return is made to the user mode after it has been interpreted. The STOP statement may have no operands.
5.17 Regaining control from automatic mode

The mode of the system may be changed from automatic to user at any time where input is not expected by hitting any keyboard character. The character is not relevant and its only function is to cause the mode change.

The present statement is completed and control given to the user. In the particular case of the FOR statement, the statement is abandoned and the FOR statement becomes the next statement.

A new line is taken on the teleprinter and a prompt consisting of a sequence number and "->" is given. This sequence number is the sequence number of the line just completed. The full facilities of the user mode are now available plus the three features described in Topic 5.8 and reproduced here.

a) The "?" will cause the sequence numbers of the previous and next statements to be typed. If the next statement has been nominated by a GO TO or CALL statement, the second operand has the form "J N" where N is the statement or sequence number used in the last statement.

b) The ":" (carriage return) character results in a return to the suspended program.

c) The ";" character results in a return to the suspended program for one statement only.

Example

>110 X=0:
>120  X=X+1:
>130  GO TO 120:
>RUN:
130 >X:  (Get attention with space bar)
53  (Value of X is 53)
>?:  (Position?)
130 J 120  (Just done 130; Jump to 120)
;:  (Do one more)
120 >X:  (Just done 120)
54  (Value of X is 54)
>?:  (Position?)
120 130  (Done 120; sequential to 130)
>:  (Go back at full speed)

5.18 ASYNCH statement

Often control is regained from the automatic mode with a specific task in mind. Perhaps a variable will be inspected and control returned to the automatic mode. The ASYNCH statement has been provided to simplify this process. Its form is

ASYNCH $ S1; S2:

A special character, in this case "$", is associated with a compound statement $S1; S2$. During automatic mode the key "$" may be used to

a) Gain control from the automatic mode,
b) Execute the compound statement as an immediate
statement and

c) Return to the automatic mode.

The six characters from ASCII C41 to C46 (exclamation, double quote, number sign, dollar, percent and ampersand) may all be used in this fashion. The association may be cancelled by the statement

`ASYNCH $`

Example

```text
>110 X=0:
>120 X=X+1:
>130 GO TO 120:
>ASYNCH $ TYPE "VALUE OF X IS",X:
>RUN:
VALUE OF X IS 117
VALUE OF X IS 202
VALUE OF X IS 265
VALUE OF X IS 316
```

The special character associated with the compound statement may be used to generate the compound statement from the keyboard. This is effective in the user mode in column one, or in column 8 after giving the preamble for storing a statement. A carriage return is needed to complete the line.
5.19 Sample program

A program to play noughts and crosses is shown below.

The game is played on a 3 by 3 layout numbered as

```
1 2 3
4 5 6
7 8 9
```

Internally, a vector PO holds the state of the game. This is initialized to all squares empty (110).

A move from the user is made via the ACCEPT statement at 120. Checks for validity of range (123) and eligibility (126) are made. If all is valid, the position is assigned and a typeout of the game made (130, 320-352).

Now a scan is made over the eight possible diagonals (140-192). The number of spaces (S(J)), the location of spaces (Y(J)), the number of X's (MG(J)) and the number of O's (YG(J)) in all diagonals are counted.

A scan is made to see if the user had just made a winning move (200). A scan is made to see if the program is in a winning position (210) and similarly a scan is made to see if the user is in a winning position (220). If none of these situations apply a move is made in the first available free space, mapped into considering the most favourable positions first by the vector A.

If no positions are left, the game is a draw (250).

The program:-

```
>110 1 FOR X=1,9 PO(X)=MT;
>120 2 ACCEPT "YOUR GO",N:
>123  IF N LT 1 OR GT 9 TYPE "NOT VALID"; GO 2:
>126  IF PO(N) NE MT TYPE "NOT VALID"; GO 2:
```
Appendix A

>130  PO(N)=YOU; CALL 9:
>140  J=1:
>150  3 S(J)=YG(J)=MG(J)=0; X=D(J,1):
>160  99 IF PO(X) EQ MT Y(J)=X; S(J)=S(J)+1:
>170  IF PO(X) EQ YOU YG(J)=YG(J)+1:
>180  IF PO(X) EQ ME MG(J)=MG(J)+1:
>190  IF X=X+D(J,3) LE D(J,2) GO 99:
>192  IF J=J+1 LE 8 GO 3:
>200  FOR J=1,8 IF YG(J) EQ 3 GO 4:
>210  FOR J=1,8 IF MG(J) EQ 2 AND S(J) EQ 1 Z=Y(J); GO 5:
>220  FOR J=1,8 IF YG(J) EQ 2 AND S(J) EQ 1 Z=Y(J); GO 11:
>240  FOR X=1,9 IF PO(A(X)) EQ MT Z=A(X); GO 11:
>250  "DRAWN GAME"; GO 1:
>260  11 TYPE "MY GO IS",Z; PO(Z)=ME; CALL 9:
>270  GO 2:
>290  4 "CONGRATULATIONS ON YOUR WIN"; GO 1:
>300  5 TYPE "MY GO IS",Z; PO(Z)=ME; CALL 9:
>310  "I WIN"; GO 1:
>320  9 H1=HA(P0(1))&"!"&HA(P0(2))&"!"&HA(P0(3)):
>330  H2=HA(P0(4))&"!"&HA(P0(5))&"!"&HA(P0(6)):
>340  H3=HA(P0(7))&"!"&HA(P0(8))&"!"&HA(P0(9)):
>350  TYPE 2C,H1,2C,"--------",2C,H2,2C,"--------",2C,H3:
>352  RETURN:

YOU=3:
ME=2:
MT=1:
HA=" X0":
A(8)=6:
A(2)=1:
A(6)=2:
A(3)=3:
A(7)=4:
A(1)=5:
A(5)=7:
A(9)=8:
A(4)=9:
D(1,1)=1:
D(1,2)=3:
D(1,3)=1:
D(2,1)=4:
D(2,2)=6:
D(2,3)=1:
D(3,1)=7:
D(3,2)=9:
D(3,3)=1:
D(4,1)=1:
D(4,2)=7:
D(4,3)=3:
D(5,1)=2:
D(5,2)=8:
D(5,3)=3:
D(6,1)=3:
Appendix A

D(6,2)=9:
D(6,3)=3:
D(7,1)=1:
D(7,2)=9:
D(7,3)=4:
D(8,1)=3:
D(8,2)=7:
D(8,3)=2:
ZkET=0:
RESET IHHI HIIH:

Execution:-

>RUN:
  YOUR GO
A>9:
    !!!
  -----  
    !!!
  -----  
    !!!0
  MY GO IS 5
    !!!
  -----  
    !X!
  -----  
    !!!0
  YOUR GO
A>1:
  0! !
  -----  
    !X!
  -----  
    !!!0
  MY GO IS 3
  0! !X
  -----  
    !X!
  -----  
    !!!0
  YOUR GO
A>7:
  0! !X
  -----  
    !X!
  -----  
  0! !0
6. Allowable next characters

The set of allowable next characters may be solicited by entering the character "->". In the general case, where there is more than one character which could be given next, the "->" is echoed. A fresh line is typed consisting of all the characters (except "<", "->" and "\") which are allowable as the next character. A dollar character is given as the first and last characters of the line to allow the space character to be easily identified. The original line is retyped and the user may continue with his input.

Examples

1.2>

$ *+-/0123456789:;E^$?

1.2+ (>)

$(*+-/0123456789ACDEFGIJKLMNOPQRSTUVWXYZ)E^$

1.2+(9/>)

$*)((+-/0123456789E^$?

1.2+(9/>)
$ *+-/:;^S$

1.2+\((9/7)\):

2.48571

If there is only one character allowable, this character is typed instead of the "$ > "$ character. Characters will continue to be typed while there is only one character allowable in the next position. Thus one or more characters may be echoed in response to the "$ > "$ character. For example, the CONTINUE statement is uniquely identified by the first five characters, CONTI. If "$ > "$ is entered after "$ CONTI "$ the system will respond with N, U and E.

Similarly, the set of allowable next characters for two positions ahead may be typed by entering the character "$ > "$ a second time.

Example

$3+4 >$

$SEGILNU$

3+4 >

E Q

G ET

I F

L ET

N E

U N

3+4 GT 5:

TRUE
In the example above the characters allowed as the n+1th character are given down the page as the first character of each line. The characters from column three are the set of characters allowed as the n+2nd character. For example, G is allowed as the n+1th character and E or T as the n+2nd character to form GE or GT.

A mode is available where the set of allowable next characters is given whenever the user makes an error. In the general case, the offending character results in the audible bell character, the ">" character, a new line giving the set of allowable next characters and a line recovering the users line so he may continue.

In the particular case where only one character is correct the system will give the audible bell character and then the correct character. If the user were typing the UNLESS keyword and had typed the U and the N, but had made a mistake typing the L, the system would type the L, E, S, S and the space at the end of the keyword. The effect is that the system will "take over" typing these keywords if the user makes a mistake.

7. Appendix

7.1 Places where characters are expanded

a) A space in column one gives a 3 digit sequence number and a space.
Appendix A

b) A space in column 5 after a sequence number and a space gives 3 spaces for a null statement number.

c) A non-space in column 6 after a sequence number and a single digit statement number gives two spaces and the non-space is presented to column 8.

d) A non-space in column 7 after a sequence number and a statement number gives a space and the non-space is presented to column 8.

e) A non-space after "TRUE" or "FALSE" generates a space followed by the non-space.

f) A special character associated with a compound statement via an ASYNCH statement may be used to generate the compound statement.

g) A non-space after a ";" separating statements generates a space followed by the non-space.

7.2 Suppressing output

In some circumstances a large amount of output may be given by the system. The user may suppress output while it is being typed by striking any keyboard character for the following situations.

All error messages.

Any fine trace (i.e. as each operation is being performed).

The output from "->".

LIST, SYMBOLS and SUSPEND output.

LTRON and TRON (in the automatic mode the keyboard
character will also cause the user mode to be entered).

7.3 Functions

The following functions are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQR</td>
<td>real exp.</td>
<td>square root</td>
</tr>
<tr>
<td>ABS</td>
<td>real exp.</td>
<td>absolute value</td>
</tr>
<tr>
<td>INT</td>
<td>real exp.</td>
<td>integer part</td>
</tr>
<tr>
<td>RND</td>
<td>real exp.</td>
<td>rounded integer part</td>
</tr>
<tr>
<td>LNG</td>
<td>string exp.</td>
<td>length of string</td>
</tr>
<tr>
<td>HXT</td>
<td>any exp.</td>
<td>output string</td>
</tr>
</tbody>
</table>

The operation, ^, has not yet been implemented fully. For any arguments its result is one.
### 7.4 Properties of statements

<table>
<thead>
<tr>
<th>immediate</th>
<th>stored</th>
<th>cont</th>
<th>FOR</th>
<th>suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>ASYNCH</td>
<td>YES</td>
<td>NO</td>
<td>IROL</td>
<td>NO</td>
</tr>
<tr>
<td>(omment)</td>
<td>YES</td>
<td>YES</td>
<td>IROL</td>
<td>NO</td>
</tr>
<tr>
<td>CALL</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>CLEAR</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>DELETE</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>EDIT</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>FOR</td>
<td>YES</td>
<td>YES</td>
<td>IROL</td>
<td>NO</td>
</tr>
<tr>
<td>GO TO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>IF</td>
<td>YES</td>
<td>YES</td>
<td>IROL</td>
<td>YES</td>
</tr>
<tr>
<td>LIST</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>LTRON,TRUN</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>PA,PB</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>RESET</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>RETURN</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>RUN</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>STOP</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>SYMBOLS</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>TYPE</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>expressions</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Notes
a) ASYNCH is an immediate statement but is stored.
b) RESET is not available to the user.
c) IROL means Includes Rest Of Line.

7.5 Invoking options
Several special modes may be invoked, with control characters. These control characters may be used at any time the system is expecting input. They are:

Control-F Prompter on - Topic 4.13
Control-Q Prompter off

Control-C Numeric check on - Topic 6
Control-X Numeric check off

Control-U Fine trace on
Control-V Fine trace off

The fine trace mode exhibits the state of expressions as each operation is performed. The mode is also entered to repeat calculations which ended in error (Topic 4.4).

All these modes are reset at initialization of the system or may be reset individually.
Appendix B

The Development of the ACL system
The Development of the ACL Language and its Implementation ACL-NOVA

By N.W. Bennett† and P.L. Sanger*

The development of a one-terminal system ACTIV-8 for the PDP-8 computer is first described. The concepts of the dynamic syntax checking of keyboard input from the terminal and the interruption of stored program execution via the communicate statement are presented.

The development of the new conversational language ACL and its implementation as a multi-user conversational interpreter ACL-NOVA is then described. The ACL-NOVA system runs as a stand-alone system and provides a powerful conversational computing facility for many users in a time-sharing environment.

1. INTRODUCTION

The first conversational computing facility used at the AAEC Research Establishment was a one-terminal interpretive system called ACTIV-8 (Bennett 1968) developed for a PDP-8 computer. The system was designed to give individual scientists and engineers an easy, direct way of solving small numerical problems and at the same time to provide results more quickly then by use of a large batch processing computer.

ACTIV-8 was implemented on a 4K PDP-8 computer, and proved very popular with those users for whom it was designed. To expand the facilities provided by the ACTIV-8 system a multi-user conversational interpreter was written for a NOVA computer. This interpreter was based on a new conversational language, ACL, developed from the experience gained using ACTIV-8. The implementation of the language is referred to as ACL-NOVA and was carried out on a 12K NOVA computer with five teletypewriter terminals (Sanger 1971).

The development and implementation of the ACTIV-8 language is discussed below, leading to a discussion of the ACL language and its implementation. These discussions are limited to the most important features of ACTIV-8 and ACL. For a more detailed description of the systems the reader is referred to the reports (Bennett 1968 and Sanger 1971).

2. THE ACTIV-8 LANGUAGE

2.1 General Discussion

The design of the ACTIV-8 language was influenced by the fact that the people who would be using the system were already familiar with FORTRAN. The initial ACTIV-8 language was fairly simple. The main statements were an arithmetic statement, a GO TO statement, an arithmetic IF statement, a TYPE statement and an ACCEPT statement. Variable names were restricted to a letter followed by a number, and the only mode of arithmetic was real.

Statements were entered from the teletypewriter terminal in the form

```
statement number — space — statement
```

and were used to build up a stored program. Statement numbers were optional and, if present, consisted of two digit numbers in the range 00-99. Statements were stored strictly in the order in which they were entered, and the stored program was executed on receipt of an END statement.

2.2 Dynamic Syntax Checking

The teletypewriter attached to the PDP-8 computer is operated in full-duplex mode. This means that characters may be transmitted between the teletypewriter and the computer in both directions simultaneously. In this case, it also means that a character pressed on the keyboard is transmitted to the computer but is not automatically printed on the teleprinter. In normal practice, a character entered from the keyboard is read by the computer and transmitted back, or "echoed", immediately to the teleprinter, which thus appears to operate like a normal office typewriter. By taking advantage of full-duplex mode operation, the computer can check if the character entered is valid in syntax and, if not, the character is simply not echoed back to the teletypewriter.

The dynamic syntax checking of input was an important part of the ACTIV-8 system and later played an important part in the design and implementation of the ACL language.

To see how this syntax checking was carried out, consider the case of an arithmetic expression. In its simplest terms, an arithmetic expression must consist of operands (i.e. variables and numbers) separated by operators (e.g. * and ÷). An expression must start with an operand (or unary + or —) and it must finish with an operand. The (simplified) structure of an arithmetic expression is illustrated by the directed-graph shown in Figure 1, and the syntax checking ensured that this pattern was obeyed. As shown in Figure 1, a "valid" Carriage Return was also required to complete a statement. Expressions containing brackets can be checked in the same way. An opening bracket may only appear at the start of an expression or after an operator, while closing brackets may only appear after an operand. An extra condition for brackets is that there may never be more closing brackets than
opening brackets, and before a carriage return is accepted there must be an equal number of opening and closing brackets.

Reference to the standard mathematical functions took the form of a three letter function name followed by an arithmetic expression enclosed in brackets (e.g. \texttt{LOG(2)}). The first indication of a function name was the combination of a letter followed by another letter instead of the number required for a variable name. An attempt was then made to verify these first two letters in a table of function names. If the second letter was not verified in this way, it was rejected. The character was not echoed to the teleprinter and the program looped to request a new character in its place. If the character was verified, the character was printed and that table entry used to check the rest of the function name.

All the statements in the ACTIV-8 language were checked in this way. For example, if the first character entered is the letter \texttt{G}, then at this stage the statement may be either a \texttt{GO TO} statement or an arithmetic statement. If the second character is the letter \texttt{O}, the user is committed to a \texttt{GO TO} statement. The only sequence of characters allowed from this point is space, \texttt{T}, \texttt{O}, space, number, number and carriage return. If the second character were a number, a commitment to an arithmetic statement would be made. Variable names were restricted to a letter followed by a number to settle this ambiguity as to statement type as soon as possible. It must be emphasised that while this scheme can detect incorrect syntax, it cannot detect the entry of incorrect information. For example, \texttt{1.6} has the same syntax as \texttt{1.5}, and there is no way for the system to detect which number the user really wanted to enter.

When a statement was completed by a "valid" carriage return, it was stored for later execution, except in the case of an END statement which caused the stored program to be executed.

Dynamic syntax checking offered both advantages and disadvantages for the ACTIV-8 system. Since all statements were checked on input there could be no syntax errors at execution time. This saved program space in the system and saved time in processing each statement. However, the sophistication of checking the syntax of incoming characters was, at that time, thought to preclude the possibility of backspacing over errors, particularly on a small machine. Once an error had been made, the entire line had to be abandoned.

2.3 Interpreter versus Compiler

Two broad methods of processing the language are available. A compiler can scan the statements and produce an object program which must be stored and can later be given control. An interpreter also scans the statements, but instead of generating actions to be stored and executed later, those actions are carried out as each statement is considered. Thus the interpretive approach does not require as much space as a compiler, but to execute a program each statement must be decoded from its source form each time control passes to it. The interpretive approach was chosen for the implementation of ACTIV-8 because of the limited space available.

An interpreter has a number of advantages over a compiler. For example, when a \texttt{GO TO} statement refers to a statement number that is not present in the source program, the process of compilation cannot be completed in the normal way. In an interpreter execution terminates only if an attempt is made to pass control to a statement with that statement number. The case where two source statements have the same statement number is very similar.

Another feature of an interpreter is that there is always a convenient link between the name of the variable and its value; that is, via the symbol table. With a compilation system, all variable names are translated into machine locations during compilation and the symbol table is not usually accessible during execution. The ACTIV-8 system takes advantage of this feature.
in the ACCEPT statement. The ACCEPT statement allows the user to input a value during execution which will be associated with a variable. Because the symbol table is available, ACTIV-8 allows the user to enter an arithmetic expression rather than just a numeric value. This expression is evaluated and its value is assigned to the variable. Note that this arithmetic expression is entered by the user during stored program execution. This proved to be very useful. For example, when the statement ACCEPT A1 is executed, the variable A1 could be increased by one by entering A1+1, A1 could be increased by twenty percent by entering A1*1.2, A1 could be given the same value as another variable by entering this variable name e.g. D4, and if A1 was expected to have a value in centimetres then 7 inches would be entered as 2.5*.7.

With a compiler it is not normally possible during execution to check whether the program is using a variable which has been set to a particular value. In an interpreter, values can only be obtained by using a variable name to access the symbol table and this allows undefined variables to be detected.

2.4 The Communicate Statement

The generality of the possible responses to the ACCEPT statement led to the design of the communicate statement. This statement consisted of the question mark character stored as part of the program and its action was determined by the response given by the user at execution time. Some of the possible responses were

(i) An arithmetic statement which is interpreted and control returns to the communicate state. This looping allows the user to give several responses.

(ii) The left hand side of an arithmetic statement, i.e. a variable name followed by an equals sign and a carriage return. The value of the variable is printed on the same line and control returns to the communicate state.

(iii) Two numbers indicating a statement number. Control passes to the statement with that statement number.

(iv) A null response (i.e. a carriage return immediately) indicating the end of the dialogue. Control passes to the statement stored after the communicate state.

The above features (and others), proved to be very useful for debugging programs, provided the user stored a communicate statement as part of the program. The user can easily determine the value of variables and influence the path of control.

The communicate statement also allows the system to act like a desk calculator, since the program?

END

lets the user enter a series of arithmetic statements which can refer to previous results. For example, a user could give the responses

A1=3
B1=4
C1=5
S1=(A1+B1+C1)/2

A2=SQR(S1*(S1-A1)*(S1-B1)*(S1-C1))

where the last line causes the value of the variable A2 to be printed.

To allow for the case where a communicate statement may not have been stored as part of a program, a facility was added so that the communicate state could be entered if the user pressed the question mark character during program execution. Processing of the current stored statement is completed and control is given to the communicate state, with the position in the program where the interruption took place being printed. The user can now interact with his program using the responses described above. This feature is also convenient for another user who wants to do a quick “desk” calculation and return control to the stored program (provided, of course, that variables in the suspended program have not been altered unintentionally).

2.5 Implementation

The ACTIV-8 language was implemented on a 4K PDP-8 computer supporting an Analex line printer, a Burroughs card reader and one teletypewriter terminal. This computer was already being used to provide a card listing facility.

The PDP-8 computer has a 12-bit word which results in a fairly limited instruction set and an addressing scheme which can directly refer to only 256 words from any particular location. The combination of word length, restricted instruction set, fixed page addressing and limited storage available in the PDP-8 computer influenced the design of the ACTIV-8 system. To simplify the task of developing and testing the ACTIV-8 system, a program was written for the IBM 360/50 computer to accept PDP-8 source language from cards, assemble the PDP-8 program into a pseudo-core storage and simulate the actions of executing the assembled program on the PDP-8 computer. This type of program is known as an assembler/simulator.

3. THE ACL LANGUAGE

3.1 General discussion

To expand the facilities provided by the ACTIV-8 system a multi-user conversational interpreter was written for a NOVA computer. This interpreter was based on a more sophisticated language, ACL, developed as a result of the experience gained with ACTIV-8 and a clearer understanding of the needs of this class of user.

5.2 Development of the Language Structure

The most useful development had its origins in the communicate statement. Here the user was able to perform arithmetic statements and GO TO statements in an immediate mode. The ACL language recognised that, in theory, there need be no barrier to any statement being used in this way; and it would be very convenient to have this mode available on first entering the system. Hence there would be two modes

1) The system would accept statements put in from the teletypewriter; execute some statements in an immediate mode and store others.
2) The system would execute previously stored statements in an automatic mode.

To allow statements to be referenced for storage, deletion and editing, a sequence number was introduced. This appeared at the start of statements which were meant to be stored, distinguishing them from statements meant for immediate execution. A requirement on the exact layout of the statements was that the dynamic syntax checking which had proved useful in ACTIV-8 should be easily implemented. In this instance it led to a space character separating the sequence number and the actual statement. The two types of statement looked like this:

sequence number — space — statement

For the syntax checking, the sequence number was indistinguishable from a number as part of an arithmetic expression (this possibility is raised later, Section 3.5). The space, which was not allowed in arithmetic expressions, served to give an indication that a sequence number was meant.

Because of the FORTRAN background of most users and because statement numbers were used in ACTIV-8, it was felt that the new language should have statement numbers. Sequence numbers can also serve as targets for transfers of control, but in some cases statement numbers are more convenient. For example, in debugging a program one may wish to change the statement number of a statement. If only the sequence number was available a change might also mean a change in the order in which statements were executed sequentially. As both sequence and statement numbers can be used for transfer of control and they were both integer numbers, an ambiguity arose. This was resolved by having statement numbers from zero to 99 and sequence numbers from 100 to 999.

Again, a space was introduced between the statement number and the statement, this time for purely aesthetic reasons. The two types of statement now had this format:

sequence number — space — statement number — space — statement

For those stored statements which did not include a statement number, two spaces could have been used. To simplify the process the user could enter a single space and the system responded with 5 spaces: two for the statement number and one for the compulsory space. Also the user could ask for a new sequence number by giving a space at the start of the line. This sequence number is 10 greater than the last sequence number specified. Thus for a statement to be stored, a new sequence number and a null statement number may be nominated by two spaces.

3.3 Statement Modification

Provision was made for deleting stored statements by entering:

sequence number — space — carriage return

Statement numbers could be altered by entering:

sequence number — space — new statement number — space — carriage return

A facility was introduced to allow the users to backspace n characters by typing <n, to be able to recon-
"arithmetic expression" and its value was printed. Thus the values of \(2+3\) and \(4-(A1\leftarrow6)\) should be printed, while the values of \(A1\leftarrow6\) and \(A23\leftarrow24+(C3\leftarrowA1)\) should not.

Finally, the extra bracket pair around assignments used as operands was dropped on the understanding that assignment should proceed from right to left, with the value of the expression on the right of the assignment arrow being assigned to the variable on the left. Thus the expression \(A23\leftarrow24+(C3\leftarrowA1)\) becomes \(A23\leftarrow24\leftarrowC3\leftarrowA1\). This allows a very general structure for multiple assignments and includes the type of multiple assignments that occur in ALGOL e.g. \(A\leftarrowB\leftarrowC\leftarrow2\) is allowed.

3.6 ACCEPT and TYPE Statements

The ACCEPT and TYPE statements were included in ACL. While the ACCEPT statement was unchanged, the generality of the TYPE statement was increased in a number of ways. Firstly, literals were allowed so that descriptive information as well as numeric values could be printed. Secondly, a variety of separators provided line control. Thus, if a comma were used as a separator, the next operand would be given on the same line. If a semi-colon were used, the next operand would be given on a new line. The form 
\(<\text{arithmetic expression}>\) was introduced to position the carriage to an absolute location. Lastly, a function called DPT was provided to allow numbers on different lines to be typed so that their decimal points were aligned vertically. The value of the DPT function is the position of the decimal point relative to the start of the number. If the number \(A3\) has a value of 89.5 then DPT(A3) would have a value of 3. To ensure that the decimal point of a number is printed in column 10 the statement
\[
\text{TYPE } <10\text{-DPT(A3)}, A3
\]
may be used.

3.7 Control Statements

The GO TO statement was expanded to allow an expression instead of an absolute statement number. If the value of the expression was in the range zero to 99, the reference is to a statement number. If the value was in the range 100-999, a sequence number was required. This form encompassed the FORTRAN ASSIGN statement (GO TO I) and also the computed GO TO (GO TO A3(I)).

A CALL statement and its associated RETURN statement were introduced to allow the user to reference other statements within the program as subroutines.

A logical IF statement was provided to give decision making capabilities. The format was
\[
\text{IF(expression.relational operator.expression)}
\]
where the relational operator could be any of the six common FORTRAN relational operators. The generality of the expressions allowed the IF statement to be used as a loop control e.g.
\[
\text{IF}(I=I+1\text{.LE.}100)\text{ GO TO 23}
\]

3.8 Supervisory Statements

A number of statements were introduced for auxiliary purposes. The EDIT statement falls into this category. Because of their usage, these statements are only available as immediate statements. A list of all the statements with an indication of these restrictions is given in Table 1.

**Table 1. List of ACL Statements**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>Accepts input from the user.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Types output to the user.</td>
</tr>
</tbody>
</table>

ACL-NOVA

A RUN statement is used to begin execution of a stored program starting at the statement with the lowest sequence number. A LIST statement is provided to give a listing of the stored statements. A companion SUSPEND statement is designed to record the state of the program as a series of immediate arithmetic statements followed by an immediate GO TO statement. These two statements may be used to prepare a paper tape so that execution may be resumed at some later time.

Trace statements allow assignments that occur in individual stored statements to be printed, or allow every stored statement that is executed to be listed.

Two conditional PAUSE statements are available. A pause before (PB) and a pause after (PA) statement will cause control to be returned to the user just before or just after a specified statement is executed. These two statements provide a valuable checkpoint facility and, together with the trace statements, form a powerful aid to program debugging.

4. DEVELOPMENT OF THE ACL-NOVA SYSTEM

4.1 General Discussion

The ACL language was implemented as a multi-user conversational interpreter on a 12K NOVA computer supporting five teletypewriter terminals.

The NOVA computer has a 16-bit word length, a powerful instruction set and four registers, two of which can be used as index registers. The addressing scheme on the NOVA allows direct reference to page zero (the first 256 locations), 256 locations centred about the program counter and 256 locations centred about each of the index registers. Each instruction occupies one NOVA word. An assembler/simulator for the NOVA computer written to run on the IBM 360/50 computer (Sanger 1970) was also used to develop and test the ACL-NOVA system.

The resident part of the ACL-NOVA system uses the first 6K of the NOVA computer. Parameters, such as the number of simultaneous users to be supported by the system, the size of each increment of work area to be allocated to a user and the total amount of work space available to the system are not assembled into the program but must be specified when the system is first loaded into the computer. Work space may be reserved for terminals or "floating", and when a user signs on at a terminal either the space that was reserved for that terminal or one increment of work space is initialised. Dynamic allocation of work space allows the "floating" work space to be allocated to users who need more space, and this is returned to the system when the user completes work at his terminal.

4.2 Arithmetic

All arithmetic is performed on 32 bit floating point numbers stored as two NOVA words. These numbers have a sign bit, a 7 bit characteristic and a 24 bit fraction. This form corresponds to the short floating point number on an IBM 360 computer and can represent numbers approximately in the range $5.4 \times 10^{-70} \leq \text{number} \leq 7.2 \times 10^{-75}$. This choice simplified the development of the floating point software because the results obtained from the NOVA simulations could easily be compared with the corresponding IBM 360 hardware floating point instructions. The only difference between the two was that the NOVA routines were set up to round the result of each arithmetic operation to produce more uniform error distributions. The times taken to perform the arithmetic operations were: addition and subtraction, 0.55 msec; multiplication, 1.72 msec; division, 2.96 msec.

4.3. The ACL-NOVA Program

The ACL-NOVA program has two main parts. One part, the Interrupt Handler, syntax checks the input from the terminals and stores it essentially as a string of source characters in a buffer area located in the appropriate user work area. This part of the program also handles any output that must be sent to the terminals.

The second part, the Background Program, controls statement execution. Requests for statement execution from each terminal are treated at the same priority level. Each terminal is serviced in a round-robin approach with the computer processing one statement at a time for each user. Thus the Background Program: (i) executes an immediate statement by interpreting the source string stored in the user buffer area and performing the indicated operations, (ii) stores a statement for later execution by moving the character string from the buffer area to another part of the user work area, and (iii) executes a stored program by fetching one statement at a time from the user work area and moving this into the buffer area for processing as in (i).

The ACL-NOVA program spends most of its time in the Background Program executing statements or checking whether there is a statement to be executed. However, the Background Program can be interrupted at any time by input or output operations at a terminal and these interrupts are serviced completely before control is returned to the Background Program.

4.4 Allocation of Space within a Work Area

The first 135 words of each user's work area is used for various pointers and buffers for the ACL-NOVA system. It includes two flag words and nineteen words of pointers used by the system as well as an input buffer, an internal code buffer and an output buffer. The remaining part of the user work area is used for storing statements and for the symbol table. To provide the most efficient use of core storage, statements are stored starting from the end of the buffer areas and continuing towards the end of the work area, while symbol table entries are stored starting at the end of the work area and continuing backwards towards the start of the work area. In this way a program with many statements but few variables, or a program with few statements but many variables can be handled. By setting up the work area in this way, the task of expanding user areas was simplified and this is discussed in Section 4.9.

4.5 Internal Code Buffer

When a statement is being entered at a terminal, it is syntax checked character by character and a copy of the "valid" characters is kept in the input buffer in input or external code form. At the same time, the
syntax checker translates these characters into an internal code to simplify statement processing and stores them in an internal code buffer. When the input statement reaches a point where the statement type can be determined, an appropriate indicator is stored at the front of the internal code (IC) buffer.

4.6 Editing Statements and Error Correction

When a stored program is listed at a terminal, each statement is translated back into external code form and stored in the output buffer from which it is sent one character at a time to the terminal. To allow statements to be modified using the EDIT statement, the required statement is listed at the terminal in the above way. The pointer to the current position in the output is reset to point to the start of the output buffer.

When the space character is entered, the character indicated by the output buffer pointer is stored in the input buffer as virtual input from the terminal and the syntax checked in the normal way. If the character is valid, the IC buffer is updated, the input buffer pointer increased by one and the output buffer pointer is increased by one; otherwise it is rejected. In the same way, new characters can be entered from the terminal during edit mode and these affect only the input buffer, the input buffer pointer and the IC buffer. When each DEL or RUB OUT character is entered, the output buffer pointer is increased by one, thus deleting one character from the statement.

A similar procedure allows input from the terminal to be modified in edit mode. The only difference is that after the carriage return characters are entered, the current contents of the input buffer are copied into the output buffer before sending a carriage return, line feed to the terminal. The original input is then edited as described above.

The correction of input from a terminal by deleting the last n characters may also be carried out simply. If there are m characters in the input buffer (m is always greater than n), then the input buffer pointer is reset to m-n and the contents of the input buffer are syntax checked in a loop that analyses 1 character, then 2 characters, then 3 characters, etc., then m-n characters so that the contents of the IC buffer are updated correctly. This multiple scan is particularly important in the case of an IF statement, and at the point in the input where the statement type may be altered by the deletion of input characters.

4.7 Symbol Table

Each symbol table entry uses four NOVA words; two words to store the variable name and two words to store its floating point value. This allows a simple variable name consisting of a letter, which may be followed by up to three letters or numbers, stored in internal code form padded with blanks where necessary as shown in Figure 2(a).

Subscripted variable names were chosen to be a letter, which may be followed by a letter or a number, stored in the first word in internal code form padded with a blank if necessary with the appropriate subscript or subscripts stored in the second word. This allows a singly subscripted variable to have a subscript in the range 0 to 65535, and each of the subscripts of a doubly subscripted variable to be in the range 0 to 255 as shown in Figure 2(b) and 2(c).

The fact that the internal codes for letters and numbers required at most 6 bits was used to distinguish between simple variables, singly subscripted variables and doubly subscripted variables. This means that the top bits of each character in the first word of a symbol table entry can be used as indicators. A singly subscripted variable is set up so that the top bit of each character in the name is set to one. Doubly subscripted variables are set up with the top bit of each of the characters in the first word of the symbol table entry set to one. This is also shown in Figure 2(b) and 2(c). This scheme has the advantage that there is no conflict when the variables A, A(1), A(1,1) are referenced by a user, as occurs for example in FORTRAN.

The symbol table entries are not ordered and a given entry is located by a sequential search of the symbol table. The search is carried out by adding an entry to the end of the symbol table for the variable required and ensures that a match is obtained from the sequential search. If the match occurs on the last entry of the symbol table, this indicates that the variable is undefined. The choice of a sequential search method simplified the structure of each work area and is adequate for this application where the symbol table is quite small for most users.

4.8 Stored Statement Processing

Execution of a stored program begins with the execution of a RUN statement or an immediate GO TO statement. This causes the sequence number of
the first statement in the stored program or the sequence number of the statement referred to in the GO TO statement to be stored as the sequence number of the next statement to be executed and the stored program to be executed flag is turned on.

When this terminal is next serviced by the round-robin background program this statement is located in the user work area and is copied into the IC buffer for processing. The sequence number of this statement is stored as the sequence number of the next stored statement (which can easily be located by using the length of the statement being processed) is stored as the sequence number of the next statement to be processed. The stored statement is now executed from the IC buffer. In this process, the IC buffer is partially overwritten by replacement operands (see Section 4.10) and this is why the statement cannot be executed directly from the user work area. At this point control is returned to the Background Program.

Sequential statement processing continues in this way until a statement that alters the path of control is executed. The simplest way to do this is to execute a GO TO statement. In this case, once the statement has been processed the sequence number referred to in the GO TO statement is stored as the sequence number of the next statement to be processed.

The CALL statement can also be used to transfer control to another group of stored statements. This statement causes (i) the sequence number of the statement referred to in the CALL statement to be stored as the sequence number of the next statement to be executed, (ii) the sequence number to be used by the RETURN statement to be stored in this CALL statement in the user area, and (iii) the sequence number of this CALL statement to be stored as the sequence number to be used by the RETURN statement. This scheme allows nested CALL statements to be used.

When a RETURN statement is executed, the sequence number to be used by the RETURN statement contains the sequence number of the last CALL statement executed. In this case, the sequence number of the statement following the last CALL statement executed is stored as the sequence number of the next statement to be executed, and the sequence number stored within this CALL statement is stored as the new sequence number to be used by the RETURN statement.

4.9 Expansion of a User's Work Area

When an arithmetic statement or expression that occurs in any statement is executed, an initial scan through the expression counts the number, \( n \), of assignment arrows. During execution of this expression, a maximum of \( 4n \) words may be added to the symbol table, and if this space is not available in the user's own area the system allocates the user an extra increment of work space from the "floating" space if this is possible. Similarly if a statement to be stored cannot fit into the user's work area, the system attempts to increase this work area.

When additional space is allocated to a user, that user's symbol table and the other user work areas that are stored above this user area must be copied into higher core locations as shown in Figure 3. The index to the start of the symbol table and the index for the next entry in the symbol table must be updated for this user while the amount of "floating" space available to the system and pointers to the other user work areas must also be updated.

After the user has completed his calculations by executing an END statement, the extra space that was obtained is returned to the system with the other user work areas stored above this user being shuffled down into lower core locations.

This dynamic allocation of work space gives the system considerable flexibility. The structure of the user work area made this scheme possible, while care was taken to ensure that the pointers in the 135 word area were used as displacements from the start of the user area rather than as absolute addresses.

4.10 Expression Evaluation

If a variable is followed by an assignment arrow, then during statement execution the expression to the right of the assignment arrow is evaluated and its value given to that variable. The variable name and its value in floating point form are stored in the symbol table. When multiple assignments occur in an expression, assuming first of all that the expression is bracket free, then the rightmost assignment arrow is located. The expression to the right of this is evaluated and the resulting value given to the variable. This process is continued until all the assignments have been carried out.

In more complicated expressions that contain a number of levels of brackets plus multiple assignments,
ACL-NOVA

The expression is evaluated by searching first for the rightmost opening bracket. The expression enclosed between the corresponding pair of opening and closing brackets (an expression at zero bracket level) is then evaluated by searching for the rightmost assignment arrow and proceeding as described above. This process is continued until the whole expression is reduced to zero bracket level and evaluated.

During the process of reducing an expression to zero bracket level, subscripted variables are replaced by subscripted variable replacement operands. A four character subscripted variable replacement operand of the form R_{p_i} was chosen (to cope with the simplest form of a subscripted variable, namely A(1)) where the character R indicates the start of a subscripted variable replacement operand, the character p is a pointer to the position of the next operator or delimiter in the expression, and the two characters i form a 16-bit index to the appropriate symbol table entry relative to the start of the user work area.

Expressions at zero bracket level (and the bracket pair surrounding the expression, if any) are replaced by simple replacement operands. A three character simple replacement operand of the form R_i was chosen (to cope with the simplest form of arithmetic expression, namely 3+2) where the character R indicates the start of a simple replacement operand, the character p is a pointer to the position of the next operator or delimiter in the expression, and the character i is an index to the floating point value of the expression. If an expression already contains simple replacement operands, then the simple replacement operand describing the value of the expression is given in the same index as the last simple replacement operand in the expression. By using the same simple replacement operand a number of times the number of NOVA words required for saving simple replacement operand values was minimised. The maximum number of simple replacement operands occurs for the expression A*A+B*B+C*C+ ... up to 71 characters and turns out to be 18. (Because of the mathematical hierarchy the multiplications must be done first.)

However, if statements are being traced by a user, then statement processing is temporarily suspended when a symbol table assignment occurs so that trace output may be printed at the terminal. During this time, statements can be processed for other terminals and consequently the replacement operand index and the values of simple replacement operands must be stored in each user area. The input buffer is not being used at this stage and it is exactly 36 words long (i.e. the space required to store the floating point values of 18 simple replacement operands). Thus the simple replacement operand index is an index to the floating point value relative to the start of the input buffer.

5. CONCLUSIONS

The one-terminal ACTIV-8 system gave individual scientists and engineers an easy, direct way of solving small numerical problems. Statements entered at a terminal were stored in the user work area strictly in the order in which they were typed and the resulting stored program could later be executed. Stored program execution could be interrupted at any time to examine the contents of the symbol table, or to evaluate arithmetic statements, or to examine the value of individual variables or to allow tracing to occur. Stored program execution could then be continued from the same point or some other point in the program.

The ACL-NGVA system allows for stored program execution or for the execution of immediate statements to perform one-time or "desk calculator" calculations, to control the execution of a stored program and to perform various editing and debugging functions. Stored program execution can again be interrupted at any time by pressing the ? character. However this time stored statements can be inserted, modified or deleted as well as any of the immediate statements being executed, before program execution is continued from the same point or some other point in the program.

The EDIT statement is the most important of the error correction facilities provided in the ACL language and it provides a novel way of allowing characters to be copied from, inserted into or deleted from an existing statement. This facility combined with other special features such as ways of suspending stored program execution and powerful tracing statements provide interaction with the users. The flexibility of the IF statement and the generality of the arithmetic statements and expressions, with the freedom of multiple assignments, also adds a great deal of power to the ACL language.

Consideration of the full duplex mode of operation of teletype/riter terminals played an extremely important part in the design of the ACTIV-8 and ACL-NOVA systems. The dynamic syntax checking of keyboard input provides interaction with the computer and protects the user from trivial typing errors. It also has the advantage that statements do not have to be checked for syntax at run time thus improving program execution times.

The implementation of the ACL language as a multi-user conversational interpreter has provided a powerful conversational computing facility for many users in a time-sharing environment. The number of terminals to be supported, the size of work area increments and the available user area are specified when the ACL-NOVA system is first loaded into the computer and work space may be reserved for each terminal or left "floating" to provide the greatest flexibility for the system. The dynamic allocation of work space in the ACL-NOVA system provides for a more efficient use of the available user area that can be obtained from a system having fixed user partitions.

6. ACKNOWLEDGEMENTS

The authors thank Dr. D. J. Richardson and Mr. R. P. Backstrom for valuable discussions throughout this work.

It is interesting to relate the work described in this paper to time, and the above projects were carried out over the following periods: ACTIV-8, January 1968 to July 1968 (NWB); ACL language development, January 1969 to March 1970 (NWB, PLS); ACL-NOVA implementation, May 1969 to May 1971 (PLS, NWB).
ACL-NOVA

REFERENCES

The UCC FASTEXT system
Appendix C

This document has been prepared with the use of the UCC FASTEXT system. The original source was entered via a typewriter connected to the UCC FASBAC system as an EDIT file. This enabled the candidate to type the source in free format, proof-read and correct errors.

The EDIT file was then processed by the FASTEXT system to produce a formatted version of the file. Commands may be built into the source file to specify spacing, page size and paragraphs.

The next page is the original input from which the title page and this page were prepared.
This document has been prepared with the use of the UCC FASTEXT system. The original source was entered via a typewriter connected to the UCC FASBAC system as an EDIT file. This enabled the candidate to type the source in free format, proof-read and correct errors. The EDIT file was then processed by the FASTEXT system to produce a formatted version of the file. Commands may be built into the source file to specify spacing, page size and paragraphs. The next page is the original input from which the title page and this page were prepared.
Appendix D

An Assembler/Simulator for the PDP-11
The next eight pages of computer listings are a sample run of the assembler/simulator which was used to develop the XB system on a UNIVAC 1108.

Page 1 shows the assembly of a small program. Lines 17 through 32 are a fixed point divide by ten subroutine for the PDP-11. Lines 11 through 15 are used to set up data, enter the subroutine and display results. The RESET instruction causes the simulator to re-enter the assembler.

There are five columns to the left of the assembler source language on Page 1. The first column is a decimal line number for each source line. The second column is a six-digit octal address of each word of the assembled program and the third column is the octal contents which have been assembled into that word. The fourth and fifth columns are hexadecimal equivalents of the second and third columns.

A cross-reference table showing where symbols have been defined and used is at the bottom of the page. There are several columns to the right of the symbols. The first column is always a one; this indicates the program segment within which the symbol was defined and in the sample this feature is not being used. The second column gives the line where the symbol was defined and succeeding columns give the numbers of the lines where a reference was made to the symbol.

The END statement (line 33) nominates a starting
address for simulation.

The second and third pages show the simulation of the assembled program. Normally there are 9 fields across each line. These are:

1) The hexadecimal address of each instruction.
2) The mnemonic of the instruction.
3) The contents of the N, Z, V and C indicators before the instruction.
4) The hexadecimal address (or register name) of the source field.
5) The hexadecimal contents of the source field.
6) The hexadecimal address (or register name) of the destination field.
7) The hexadecimal contents of the destination field before the instruction.
8) The hexadecimal contents of the destination field after the instruction.
9) The contents of the N, Z, V and C indicators after the instruction.

Jumps and successful branches have an indicator printed on the extreme right and a line skipped. In the case of subroutine jumps the contents of the register 6 stack are displayed after the instruction. The first line is a set of hexadecimal addresses from four words below the stack pointer to 12 words above it. The actual value of register 6 is highlighted. The second line is the contents of the
stack. For example the return address of the JSR instruction at HEX 2718 is HEX 271C. The return from subroutine instruction causes the contents of the stack before the instruction to be printed.

The instructions at HEX 271C and 271E display the contents of registers zero and one; the remainder and result of the divide.

Page 4 shows an assembly of two different instructions into the existing assembled program.

Pages 5 and 6 show the simulation of the altered program with the results of 3 and 5 shown.

Page 7 is similar to page 5 but the printout of the divide subroutine has been suppressed (line 1). This results in a much reduced output from the simulation (page 8).
ORG 10000

1 023420 2710

START MOV =800>R6

DIV10 CLR R1  DIVIDE BY TEN; ANSWER IN R1; REMAINDER IN R0

MOV =TBL>R2

L36 CLC

L15A SUB (R2),R0

L15A ADD 2(R2),R1

R1 L15A

L24 ADD (R2)==R0

L24 ADDR(R2)+R0

CMP (R2)==1

REAL TIME CLOCK INTERROGATED AT 11:42:31
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Instruction</th>
<th>Address</th>
<th>Data</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>273C</td>
<td>RTS</td>
<td>0 1 0 0</td>
<td>0007</td>
<td></td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>271C</td>
<td>MOV</td>
<td>0 1 0 0</td>
<td>REG 0</td>
<td>0001</td>
<td>REG 0 0001</td>
</tr>
<tr>
<td>271E</td>
<td>MOV</td>
<td>0 0 0 0</td>
<td>REG 1</td>
<td>0002</td>
<td>REG 1 0002</td>
</tr>
<tr>
<td>0318</td>
<td>031A</td>
<td>031C 0320*</td>
<td>0322</td>
<td>0324</td>
<td>0326 0328</td>
</tr>
<tr>
<td>032A</td>
<td>032C</td>
<td></td>
<td>0330</td>
<td>0332</td>
<td>0334 0336</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>271C 0000</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>2720</td>
<td>RESET</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*0320* indicates a hexadecimal value.
<table>
<thead>
<tr>
<th></th>
<th>Address 1</th>
<th>Address 2</th>
<th>Address 3</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>023420</td>
<td>2710</td>
<td>10000</td>
<td>ORG 10000</td>
</tr>
<tr>
<td>2</td>
<td>023420</td>
<td>012706</td>
<td>2710</td>
<td>MOV =800+R6</td>
</tr>
<tr>
<td></td>
<td>023422</td>
<td>001440</td>
<td>2712</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>023424</td>
<td>012700</td>
<td>2714</td>
<td>MOV =53+R0</td>
</tr>
<tr>
<td></td>
<td>023426</td>
<td>00000005</td>
<td>2716</td>
<td></td>
</tr>
</tbody>
</table>

END START

REAL TIME CLOCK INTERROGATED AT 11:42:33
<table>
<thead>
<tr>
<th>Address</th>
<th>Destination</th>
<th>Instruction</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0316</td>
<td>031A</td>
<td>JSR</td>
<td>031C</td>
<td>031D</td>
</tr>
<tr>
<td>0400</td>
<td>0400</td>
<td>2722</td>
<td>0410</td>
<td>0411</td>
</tr>
<tr>
<td>0500</td>
<td>0500</td>
<td>0202</td>
<td>0510</td>
<td>0511</td>
</tr>
<tr>
<td>0600</td>
<td>0600</td>
<td>0328</td>
<td>0610</td>
<td>0611</td>
</tr>
<tr>
<td>0700</td>
<td>0700</td>
<td>032A</td>
<td>0710</td>
<td>0711</td>
</tr>
<tr>
<td>0800</td>
<td>0800</td>
<td>032C</td>
<td>0810</td>
<td>0811</td>
</tr>
<tr>
<td>0900</td>
<td>0900</td>
<td>032E</td>
<td>0910</td>
<td>0911</td>
</tr>
<tr>
<td>0A00</td>
<td>0A00</td>
<td>0330</td>
<td>0A10</td>
<td>0A11</td>
</tr>
<tr>
<td>0B00</td>
<td>0B00</td>
<td>0332</td>
<td>0B10</td>
<td>0B11</td>
</tr>
<tr>
<td>0C00</td>
<td>0C00</td>
<td>0334</td>
<td>0C10</td>
<td>0C11</td>
</tr>
</tbody>
</table>

The table above represents a hexadecimal memory map with addresses in hexadecimal format and instructions and destinations in a natural format. The instructions include `JSR`, `CLR`, `MOV`, and others, with source and result fields indicating the operands and outcomes. The content suggests a technical or educational context, possibly related to computer programming or hardware.
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Register 0</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>272A</td>
<td>SUB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>272C</td>
<td>UCS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>2732</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>272A</td>
<td>SUB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>272C</td>
<td>UCS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>272E</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>2734</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>273B</td>
<td>CMP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>273A</td>
<td>UNE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0316</td>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>273C</td>
<td>RTS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>271C</td>
<td>MOV</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>271E</td>
<td>MOV</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>031B</td>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>2720</td>
<td>RESET</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
</tbody>
</table>
REAL TIME CLOCK INTERROGATED AT 11:42:35
Appendix E

Use of the XB paper tape system
Appendix E

There are two paper tapes provided with the thesis. They are marked "XB system" and "XB demonstration tape". The XB system is in standard PDP-11 binary loader format and should be loaded normally via the high speed paper tape reader.

The key combination CONTROL-G may be used at any time to initialize the system. A heading describing the version of XB will be typed each time the system is initialized.

The demonstration tape is a collection of about 50 graded examples designed to introduce the major concepts of the system. Each frame may be read by giving a CONTROL-D character. This sets a mode to read from the high speed paper tape reader. The user is free to use the system from the keyboard between frames.

Most of the frames are self-contained. However, an asynchronous interrupt from the keyboard can only be made by the user. This is needed to interrupt the automatic mode to regain control or to invoke an ASYNCH statement.
Appendix F

Sample flowchart - Arithmetic expressions
Appendix F

The next few pages illustrate the techniques used in accepting or rejecting individual characters to form an arithmetic expression. When a character is accepted, a new character is taken from the keyboard and compared with the characters or groups of characters allowable as the next character. For example, after an exponent \( E \) (see flowchart of Number) the next character must be plus, minus or a number. In other situations, such as after a number in the final flowchart, the arithmetic routine cannot judge the validity of characters by itself. If a character is not one of the five operators, an exit is taken and the validity of the character decided elsewhere. The comma in

\[
\text{TYPE 1+2,}
\]

would be accepted, but the comma in

\[
A=1+2,
\]

would not be.

As explained in the body of the thesis, the program to check expressions for validity is written in assembler for maximum speed. For arithmetic expressions, see 1116-1349, Appendix G.
Number
Arithmetic Expression
Appendix G

Partial Source of the XB system
Appendix G

The next 61 pages are the source statements of the PDP-11 program written as the major part of this project. The source statements given here are not complete; the full program consists of over 8000 statements. That portion given here is the code which implements the algorithms described in the body of the thesis.

A row of asterisks marks the places where part of the program has been left out. Reference is made in the thesis to the program by the decimal line number on the left hand side. The addresses and contents given here are not necessarily the same as in the program supplied on paper tape.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>TROFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000000</td>
<td>0000</td>
<td>K EQR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>000000</td>
<td>0000</td>
<td>RO EQR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>000001</td>
<td>0001</td>
<td>RI EQR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>000002</td>
<td>0002</td>
<td>R2 EQR 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>000003</td>
<td>0003</td>
<td>R3 EQR 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>000004</td>
<td>0004</td>
<td>R4 EQR 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>000005</td>
<td>0005</td>
<td>R5 EQR 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>000006</td>
<td>0006</td>
<td>R6 EQR 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>000007</td>
<td>0007</td>
<td>PC EQR 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>177550</td>
<td>FF68</td>
<td>PRS EQU 0*177550'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>177552</td>
<td>FF6A</td>
<td>PRB EQU 0*177552'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>177554</td>
<td>FF6C</td>
<td>PPS EQU 0*177554'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>177556</td>
<td>FF6E</td>
<td>PPR EQU 0*177556'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>177558</td>
<td>FF70</td>
<td>TKR EQU 0*177560'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>177560</td>
<td>FF72</td>
<td>TPS EQU 0*177564'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>177562</td>
<td>FF74</td>
<td>TPR EQU 0*177566'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>177564</td>
<td>FF76</td>
<td>SWR EQU 0*177570'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>177570</td>
<td>FF78</td>
<td>CHSTHI EQU <em>FFAF</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>177572</td>
<td>FF80</td>
<td>CHCODE EQU <em>FFD0</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>177574</td>
<td>FF82</td>
<td>FALSE EQU <em>FFFA</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>177576</td>
<td>FF84</td>
<td>TRUE EQU X'OED'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>177578</td>
<td>FF86</td>
<td>EQU 177552</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>17757A</td>
<td>FF88</td>
<td>EQU 177554</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>17757C</td>
<td>FF8A</td>
<td>EQU 177556</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>17757E</td>
<td>FF8C</td>
<td>EQU 177558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>177580</td>
<td>FF90</td>
<td>EQU 177560</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>177582</td>
<td>FF92</td>
<td>EQU 177562</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>177584</td>
<td>FF94</td>
<td>EQU 177564</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>177586</td>
<td>FF96</td>
<td>EQU 177566</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>177588</td>
<td>FF98</td>
<td>EQU 177568</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>17758A</td>
<td>FF9A</td>
<td>EQU 177570</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>17758C</td>
<td>FF9C</td>
<td>EQU 177572</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>17758E</td>
<td>FF9E</td>
<td>EQU 177574</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>177590</td>
<td>FFA0</td>
<td>EQU 177576</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>177592</td>
<td>FFA2</td>
<td>EQU 177578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>177594</td>
<td>FFA4</td>
<td>EQU 177580</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>177596</td>
<td>FFA6</td>
<td>EQU 177582</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>177598</td>
<td>FFA8</td>
<td>EQU 177584</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>17759A</td>
<td>FAA0</td>
<td>EQU 177586</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>17759C</td>
<td>FAA2</td>
<td>EQU 177588</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>17759E</td>
<td>FAA4</td>
<td>EQU 177590</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>1775A0</td>
<td>FAA6</td>
<td>EQU 177592</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1775A2</td>
<td>FAA8</td>
<td>EQU 177594</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1775A4</td>
<td>FAAA</td>
<td>EQU 177596</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1775A6</td>
<td>FFA0</td>
<td>EQU 177598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1775A8</td>
<td>FFA2</td>
<td>EQU 1775A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>1775AA</td>
<td>FFA4</td>
<td>EQU 1775A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1775AC</td>
<td>FFA6</td>
<td>EQU 1775A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>1775AE</td>
<td>FFA8</td>
<td>EQU 1775A6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>000000</td>
<td>0000</td>
<td>ORG 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>000001</td>
<td>0001</td>
<td>DCH TRAPZZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>000002</td>
<td>0002</td>
<td>DCH 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>000003</td>
<td>0003</td>
<td>DCH TRAPZZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>000004</td>
<td>0004</td>
<td>DCH 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>000005</td>
<td>0005</td>
<td>DCH TRAPZZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>000006</td>
<td>0006</td>
<td>DCH 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>000007</td>
<td>0007</td>
<td>DCH 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>000008</td>
<td>0008</td>
<td>DCH 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>000009</td>
<td>0009</td>
<td>DCH 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>000010</td>
<td>0010</td>
<td>DCH 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INDICATE STRING**

**INDICATE OPERATOR**

**REPRESENTATION OF FALSE**

**TRUE EQU X'OED'**

**REPRESENTATION OF TRUE**

**SPECIAL ZERO**

**INDICATE OPENING BRACKET**

**START OF ARITH FIELD**

**END OF ARITH FIELD**

**INDICATE ARRAY NAME**

**INDICATE NAME AFTER 'IF'**

**END OF LOGICAL EXPR AFTER 'IF'**

**INDICATOR AT END OF 'TRUE' PART**

**INDICATOR AT END OF 'FALSE' PART**

**INDICATOR FOR A FUNCTION**

**START OF A STORED LINE**

**STRIND EQU X'8888'**

**INDICATE STRING IN A+2**

$**S$ INCLUDE CONFIDENCE CODE
ORG 0'03C*
DCH EMTVEC
ORG 0
DCH 0
DCH TRPVEC
ORG 0
DCH 0
ORG 128
DCH 0

FFCGN DSB 0
LOOK AT PC! SHOULD BE LESS THAN OCTAL 400

* AREA FROM 256 TO 1024 FOR R6 STACK
* USE THIS FOR PATCH AREA

LOCORE DSH 0 FOR CHECKSUM

DCH TRAPZZ FOR TOO MANY RTS
TRAPZZ EMT VC0071-EMTT8L HALT ROUTINE
CHECK ON STACK

EMTVEC MOV R1»SAVR1
MOV (R6)+R1 RETURN ADDRESS
MOV 2(R6)+(R6)
MOV -(R1)fR1 GET INSTRUCTION
BIC =X'FF00»»R1
MOV EMTTBL(R1)»-<R6)
MOV EMTTBL-X'0900'+1(R1)r(R6)
MOV SAVR1»R1
RTI

TRPVEC MOV R1»SAVR1
MOV R1»R6
MOV -(R1)+R1 GET INSTRUCTION

TRAPEZ EMT VC0071-EMTT8L HALT ROUTINE

****** ALLOW FOR 1'S ARITH
MOV SAVR1+R1
RTI

****** ALLOW FOR 1'S ARITH
MOV SAVR1+R1
RTI
945 006710 051072 00CA 523A  VC0130  DCH  SH3500
946 006712 045660 00CA 48B0  VC0131  DCH  GET2ND
947 006714 025352 00DE 2AEC  VC0139  DCH  TBLUPM
948 006716 046214 0000 4C8C  VC0137  DCH  ORLARG
949 006720 050424 00D2 5114  VC0140  DCH  BBL2
950 006722 050252 00D4 50AA  VC0141  DCH  LISTD
951 006724 052270 00D6 5466  VC0142  DCH  FN2DEN
952 006726 052006 00D8 5466  VC0144  DCH  FNDZB
953 006728 052164 00DA 5474  VC0145  DCH  FNDSTN
954 006730 050242 00DF 527C  VC0041  DCH  MJSCHD
955 006732 052744 00D6 2F64  VC0043  DCH  PLUB
956 006734 047016 00D8 41CE  VC0045  DCH  LPEI1
957 006736 043700 00E2 0AF8  V00002  DCH  SLEIRR
958 006740 002122 00E4 056A  V00004  DCH  TRSBZ
959 006744 035640 00E6 38A0  V00005  DCH  STTF
960 006750 012326 00EA 194C  V00111  DCH  STSC
961 006752 013514 00EA 174C  V00114  DCH  STP
962 006756 012876 00EC 153E  V0023  DCH  R21
963 006756 025322 00DE 2752  V0023  DCH  VG2O
964 006760 012242 00F0 14A2  V0027  DCH  ALT2
965 006762 010616 00F2 118E  V0031  DCH  AREX
966 006764 051174 00F4 527C  V0042  DCH  MJUSHD
967 006766 027544 00F6 2F64  V0043  DCH  PLUB
968 006770 040716 00F8 41CE  V0045  DCH  LPEI1
969 006772 050242 00FA 50A2  V0051  DCH  INTERB
970 006774 050053 00FC 0AF8  DSH  3  SPARE

*******************************************************************************
1116 010616 118E  AREX  DSH  0
1117 010620 162706 018E  E5C6  SUBI  =2*4R
1118 010622 000002 1190 0002  CLR  (R6)  LAST  OPERATOR  NOT  UP  ARROW
1119 010624 000003 1194 0003  DEBUG  STL  SUBR
1120 010626 177772 1196 00FF  DCH  CHFAEX
1121 010630 022700 1198 2500  S1  CMP  =E'++'K  IS  IT  A  UNARY  PLUS
1122 010632 000053 119A 032A  BNE  M1
1123 010636 000003 119E 0003  M2  DEBUG  STL  SUBR
1124 010640 177740 11A0 0001  DCH  CHSPZR  STORE  A  SPECIAL  ZERO
1125 010642 000003 11A2 0003  DEBUG  STL  SUBR
1126 010644 177724 11A4 0003  DCH  CHD4EX  STORE  INDICATOR  BEFORE  OPERATOR
1127 010646 104054 11A6 802C  EMT  V0024-EMTTL
1128 010650 004003 11A8 0103  BR  S2
1129 010652 022700 11AA 2500  M1  CMP  =E'-'K  IS  IT  A  UNARY  MINUS
1130 010654 000055 11AC 002D  BNE  M2  TREAT  AS  FOR  UNARY  PLUS
1131 010656 01707 11AE 03F7  REG  M2  TREAT  AS  FOR  UNARY  PLUS
1132 010658 022700 11AJ 2500  S2  CMP  =E'++'K  IS  IT  AN  OPENING  BRACKET
1133 01065A 000003 0102 0028  BNE  M3
1134 01065C 000063 1106 0003  DEBUG  STL  SUBR  STORE  INDICATOR  TO  THE  OPENING  BRACKET
1135 010670 177740 1108 0003  DCH  CHC0EX
1136 010672 104054 110A 802C  EMT  V0024-EMTTL
1137 010674 104056 110C 802E  EMT  V0025-EMTTL
1138 010676 022700 110E 2500  CMP  =E'++'K  MUST  END  IN  A  CLOSING  BRACKET
1139 010678 000051 1110 0029  BNE  LBJ  IF  NOT  IS  AN  ERROR
1140 01067A 001110 1112 024C  BR  L315
1141 01067C 010454 1114 802C  EMT  V0024-EMTTL
1142 01067E 000436 1116 011E  BR  L315
1143 01067F 000102 1118 024E  BNE  M4
1143 010716 104054 11CE 8A2C EMT VC0024=EMTTL
1144 010720 000417 11D1 010F BR L34AZX
1145 010722 104032 11D2 M4 DSH 0 IS IT A NUMBER
1146 010722 104032 11D2 881A EMT VC0014=EMTTL
1147 010724 001002 11D4 0202 BNE M5 NO
1148 010726 001002 11D4 BR DSH 0
1149 010726 104054 11D6 882C EMT VC0024=EMTTL
1150 010730 000502 11D8 0142 BR L43
1151 010732 000502 11D8 M5 DSH 0 IS IT A LETTER OTHER THAN 'B' OR 'H'
1152 010732 004737 11D0 09DF JSR PC*LET
1153 010734 015929 11D4 1A14 O11T4 EMT VC0014=EMTTL
1154 010736 010762 11D0 E023A BNE LBJ IF NOT IS AN ERROR
1155 010740 005037 11E0 01AF CLR VARNM
1156 010742 053312 11E2 56CA CLKB VARNM+2
1157 010744 053314 11E4 M5 DSH 0 IS IT A LETTER OTHER THAN 'B' OR 'H'
1158 010744 053312 11E2 56CA EMT VC0024=EMTTL
1159 010746 050476 11E3 013E BR L53AZX
1160 010750 110037 11E8 882C IS IT A NUMBER
1161 010752 053312 11EA 009F BNE LBJ
1162 010760 104032 11F0 881A EMT VC0014=EMTTL
1163 010762 001000 11F2 0230 BNE LBJ
1164 010764 104054 11F4 882C EMT VC0024=EMTTL
1165 010766 104032 11F6 881A IS IT A NUMBER
1166 010770 001775 11F8 03FD BEQ M7
1167 010772 022700 11FA 25CO L315 CMP =E'++*K IS IT THE LETTER E
1168 010776 001002 11FE 0202 BNE L315
1169 011000 104054 1200 882C EMT VC0024=EMTTL
1170 011002 000432 1202 611A BR L322
1171 011004 022700 1204 25CO L319 CMP =E'++*K IS IT A PLUS
1172 011006 000503 1206 002B BNE B13
1173 011010 010005 1208 0205 M5 CLR (R6) THIS OPERATOR NOT AN UP ARROW
1174 011012 005016 120A 0A0E THIS OPERATOR NOT AN UP ARROW
1175 011014 000003 120C 0003 MO5B DEBUG STL SUBR
1176 011016 177724 120E FF04 OSS CH04EX STOREE INDICATOR BEFORE OPERATOR
1177 011020 104054 1210 882C EMT VC0024=EMTTL
1178 011022 000716 1212 01CE BR S2
1179 011024 022700 1214 25CO B13 CMP =E'++*K
1180 011026 000555 1216 002D BNE B13
1181 011030 001770 1218 03FD BEQ M85
1182 011032 022700 121A 25CO CMP =E'++*K IS IT AN ASTERISK
1183 011034 00002 121C 002A BEQ M85
1184 011036 01765 121E 03F5 BEQ M85
1185 011040 022700 1220 25CO CMP =E'++*K IS IT DIVIDE
1186 011042 000597 1222 002F BEQ M85
1187 011044 001762 1224 03F2 BEQ M85
1188 011046 023700 1226 27CO CMP EXPCH*K IS IT UP ARROW
1189 011050 052444 1228 5524 BEQ M85
1190 011052 001402 122A 030E JMP LEND
1191 011054 00137 122C 005F M5A TST (R6) WAS LAST OPERATOR AN UP ARROW
1192 011056 011924 122E 1314 BNE LBJ YES; IS AN ERROR
1193 011060 005716 1230 00CE INC (R6)
1194 011062 001020 1232 0210 DNE LBJ
1195 011064 005216 1234 008E INC (R6)
1196 011066 007552 1236 001E BR M5B
1197 011070 022700 1238 25CO L322 CMP =E'++*K IS IT E*
DEBUG STL SUPR
DCI CHCIEX STORAGE INDICATOR (ARRAY NAME)
EMT VC0024-EMTBL
EMT VC0025-EMTBL
CMP =E'**'K

MOVB R0)VARNM+1  SAVE 2ND CHAR OF VARIABLE

EMT VC0024-EMTBL
EMT VC0025-EMTBL

CMP =E'**'K  MUST FINISH WITH A CLOSING BRACKET
1285 011414 022700 130C 25C0 \textit{B73AZX CMP} =E'F'\textbullet K IS IT AN ARRAY NAME (TWO CHARACTERS)
1286 011416 000050 130E 0028 \textit{BNE B71}
1287 011420 001006 1310 0206 \textit{BR B6A YES}
1288 011422 000655 1312 0185 \textit{LEND DSH 0}
1289 011424 062706 1314 65C6 \textit{ADD} =2\textbullet R6
1290 011426 000002 1316 0002 \textit{STRTS DSH 0}
1291 011430 000003 1318 0003 \textit{DEBUG STL SUBR}
1292 011432 177773 131A FF00 \textit{DCH CHF0EX}
1293 011434 000207 131C 0087 \textit{RTS PC}
1294 011436 022700 131E 25C0 \textit{BNE L53} CMP =E'I'\textbullet KCH2 WAS SECOND LAST CHARACTER AN I
1295 011442 001304 1322 02C4 \textit{BNE L53}
1296 011444 022737 1324 250F CMP =E'I'\textbullet KCH2 WAS SECOND LAST CHARACTER AN I
1297 011446 000111 1326 0049 \textit{MOV =L3i5rSUBRCT}
1298 011450 053320 1328 5600 \textit{BNE LEND}
1299 011452 001364 132A 02F4 \textit{BNE LEND}
1300 011454 022737 132C 250F CMP =E'I'\textbullet KCH1 WAS LAST CHARACTER AN F
1301 011456 000166 132E 0046 \textit{LEND}
1302 011460 053322 1330 5600 \textit{LEND}
1303 011462 001360 1332 02F0 \textit{BNE LEND}
1304 011464 104054 1334 8A2C \textit{EMT VC0024=EMTTBL}
1305 011466 000003 1336 0003 \textit{DEBUG STL SUBR}
1306 011470 177603 1338 FF83 \textit{DCH CH03EX STORE INDICATOR FOR IF}
1307 011472 104066 133A 0836 \textit{EMT VC0031=EMTTBL}
1308 011474 000003 133C 0003 \textit{DCH J03EX STORE INDICATOR}
1309 011476 177626 133E FF96 \textit{DCH CH03EX STORE INDICATOR}
1310 011500 004437 1340 091F \textit{JSR CHK\textbullet R\textbullet CHECK}
1311 011502 015170 1342 1A76 \textit{DCC E' THEN : ' NEXT CHARACTERS TO BE ' THEN '}
1312 011504 000044 1344 0020 \textit{LENG}
1313 011506 000124 1346 0054 \textit{BNE LEND}
1314 011508 000110 1348 0048 \textit{BNE LEND}
1315 011510 000116 134A 004E \textit{BNE LEND}
1316 011512 000040 134C 0020 \textit{BNE LEND}
1317 011514 104056 134E 8A2E \textit{EMT VC0025=EMTTBL}
1318 011516 000003 134G 0003 \textit{DEBUG STL SUBR}
1319 011520 177646 1350 FF6A \textit{DCH CHA3EX STORE INDICATOR}
1320 011522 004437 1352 091F \textit{JSR CHK\textbullet R\textbullet CHECK}
1321 011524 015170 1354 1A78 \textit{DCC E' ELSE : ' NEXT CHARACTERS TO BE ' ELSE '}
1322 011526 000040 1356 0020 \textit{LENG}
1323 011528 000105 1358 0045 \textit{BNE LEND}
1324 011530 000114 135A 004C \textit{BNE LEND}
1325 011532 000123 135C 0053 \textit{BNE LEND}
1326 011533 000043 135E 0045 \textit{BNE LEND}
1327 011534 000072 1360 0020 \textit{BNE LEND}
1328 011535 000124 1362 0054 \textit{BNE LEND}
1329 011536 104056 1364 8A2E \textit{EMT VC0025=EMTTBL}
1330 011538 000003 1366 0003 \textit{DEBUG STL SUBR}
1331 011542 177603 1368 FF83 \textit{DCH CH03EX STORE INDICATOR TO END OF SECOND PART}
1332 011544 000727 136A 0107 \textit{DCH CH03EX STORE INDICATOR TO END OF SECOND PART}
1333 011546 053320 136C 5600 \textit{L59 DSH 0}
1334 011548 104050 136E 8A30 \textit{EMT VC0027=EMTTBL}
1335 011550 001656 136F 039E \textit{BNE B73 FOURTH CHARACTER IS NUMBER OR LETTER}
1336 011552 022700 1370 25C0 \textit{BNE L53}
1337 011554 000005 1372 0028 \textit{BNE L53 NOT A FUNCTION}
1338 011556 012800 1374 150F \textit{MOV =L315\textbullet SUBRET}
SUBRAB DSH 0
MOV =NM5,R3

LOOPC CMPB (R3),=E'=' FUNCTION: SEE IF NAME IS ON LIST

BNE AD2
CMPB (R3)+KCH2

BNE AD1
CMPB (R3)+KCH1

BNE ADO
MOV STUL(R3)+R1

JSR PC.(R1)

Very close with a bracket

DEBUG STL SUBR

DCH CHC3EX INDICATES FUNCTION

EMT VC0024-EMTBL

MOV SUBRET+PC

BR LOOPC

LBSTSC JMP STSC

STREX DSH 0

MP =E'='X K

BNE STSC

CMP =E'H'XK

BEQ STRAl

CMP =E'0'XK QUOTE

BNE STRA5

EMT VC0024-EMTTBL PICK UP STRING LITERAL

BNE STRAT

STRIF1

CMP =E'H'XK

BNE STRA4

CMP =0*047'XK QUOTE
1446 012262 000111 14B2 0049  DCC E'INT'
1447 012263 000116 14B3 004E
1448 012264 000124 14B4 0054
1449 0122a5 000000 14B5 0000  DCB 0 AREX
1450 012266 015602 14B6 1882  DCH 'INT'
1451 012270 000122 14B8 0052  DCC E'INT'
1452 012271 000116 14B9 004E
1453 012272 000104 14B0 0044  DCB 0 AREX
1454 012273 000000 14B1 0000  DCH RND
1455 012274 015306 14C0 1866  DCC E'INT'
1456 012276 000116 14C1 004C  DCC E'LNG'
1457 012277 000116 14C2 004E
1458 012300 000000 14C3 0047
1459 012301 000002 14C4 0002  DCB 2 STRING
1460 012302 012566 14C5 14F6
1461 012304 000010 14C6 0048  DCC E'INT'
1462 012305 000140 14C7 005B
1463 012306 000124 14C8 0054
1464 012310 001700 14C9 0404  DCH SXTSXT
1465 012311 000000 14CA 003A  DCC E'INT'
1466 012312 000000 14CB 0020
1467 012314 010616 14CC 110E  STBL DCH AREX
1468 012316 011700 14CD 13C0  DCH STREX
1469 012320 012470 14CE 153B  DCH LOGEX
1470 012322 014202 14CF 1682  DCH GENEX

*******************************************************************************
1500 012470 153B  LOGEX DSH 0
1501 012470 162706 153B  E5C6  SUB =2,R6
1502 012472 000002 153A  0002
1503 012474 005016 153C  0A0E  CLR (R6)  NO FULL RELATIONAL FOUND
1504 012476 024700 153E  25C0  R21 CMP =E','K IS IT A SPACE
1505 012477 001107 1542  0247  HNE R12
1506 012478 013737 1544  170F  MOV L\LOGZ  SAVE STATE
1507 012486 053324 1546  5604
1508 012490 053342 1548  5622
1509 012494 053326 154A  5606
1510 012496 053344 154C  5644
1511 012498 104054 1550  682C  EMT VC0024=EMTBL
1512 012499 022700 1552  25C0  CMP =E'N',K IS IT 'N'
1513 012504 001101 1554  0219  BNE PIA
1514 012506 010501 1556  0219  BNE PIA
1515 012508 010500 1558  0219  BNE PIA
1516 012509 010500 155A  0219
1517 012500 011017 155C  004F
1518 012501 000117 155E  0272  BNE R21REC
1519 012502 104054 1560  082C  EMT VC0024=EMTBL
1520 012503 022700 1562  25C0  CMP =E'T',K IS IT 'NOT'
1521 012504 00124 1564  0054  BNE STSKA
1522 012505 001052 1566  022A  EMT VC0024=EMTBL
1523 012506 104054 1568  082C  DEBUG STL SUBR
1524 012507 000003 156A  0003  DCH CMD4EX STORE INDICATOR
1525 012508 177724 156C  FF04  CMP =E',K
1526 012509 000040 1570  0020
1527 01250a 001404 1572  0224  BNE STSKA MUST BE 'NOT'
1528 01250b 104054 1574  082C  EMT VC0024=EMTBL
1529 01250c 022700 1576  25C0  CMP =E',K IS IT A SPACE
BNE STSKA MUST END IN CLOSING BRACKET

EKM VC0024-EMTBBL

TST LOG2 TEST TYPE OF EXPRESSION JUST PICKED UP

REQ PPP61 ARITH

SPL PSTRA LOGICAL

JMP P77AA STRING

PPP61 DSH 0

JMP P61

PSTRA DSH 0

CLR (R6) NOT FULL RELATIONAL ANY MORE; LOGICAL EXPR

TRAP VC0014-TRAPBL

A13 CMP =E'B'K IS IT A 'B' LOGICAL VARIABLES START WITH B

BEQ P3X

CMP =E'I'K IS IT AN 'I' POSSIBLY IF

BEQ P4

OR R6 MUST BE START OF ARITH OR STRING EXPRESS.

AS CMP =E'I'K IS SECOND CHARACTER AN OPENING BRACKET

BNE P34

A63 DEHHG STL SUBR

DCH CHCIF.X STORE INDICATOR (ARRAY NAME)

EKM VC0024-EMTBBL

EKM VC0025-EMTBBL

CMP =E'I'K

BNE STSKA

BEQ PPP61

BNE STSKA MUST END CLOSING BRACKET

REQ P322 JUST ONE SUBSCRIPT

CMP =E'I'K IF COMMA A SECOND SUBSCRIPT IS THERE

BNE STSKA

P322 DSH 0

EKM VC0024-EMTBBL

P34 CMP ASEQLS,K IS IT ASSIGNMENT ARROW

BNE PSTRA PICK UP OPERATOR

CLR (R6) NOT FULL RELATIONAL ANY MORE; ASSIGNMENT

DEBUG STL SUBR

DCH CHD4EX

EKM VC0024-EMTBBL

EKM VC0025-EMTBBL

CMP =E'I'K MUST END IN CLOSING BRACKET

BNE STSKA

P32 DSH 0

ACCEPT FIRST CHARACTER OF VARIABLE NAME

CLR (R6) NOT FULL RELATIONAL ANY MORE BOOLEAN VARIABLE

EKM VC0024-EMTBBL

EKM VC0027-EMTBBL

BNE A5

EKM VC0024-EMTBBL

P32 DSH 0 IS THIRD CHARACTER A NUMBER OR A LETTER

EKM VC0027-EMTBBL

BEQ P32A
CMP =E"F", K IS THIRD CHARACTER AN OPENING BRACKET

BNE P34
BR A63
P32A DSH 0
EMT VC0024-EMTTBL

BNE P32B
MOV =PSTRA+SUBRET

JMP SUBRAB
P32B DSH 0
EMT VC0027-EMTTBL
BNE P34
EMT VC0024-EMTTBL
BR P34

MOV L+LOGZ SAVE STATE OF POINTERS

MOV (R6)+LOG4 SAVE STATE OF FULL RELATIONAL

EMT VC0024-EMTTBL

BNE Z2
EMT VC0024-EMTTBL

BNE 22
EMT VC0024-EMTTBL

DEBUG STL GUBR
DCH CH63EX STORE INDICATOR AFTER ' 'IF ' •
EMT VC0031-EMTTBL
DEBUG STL SUBR
DCH CH96EX STORE INDICATOR
JSR CHK3rCHECK

DCC E' THEN ' ' NEXT CHARACTERS MUST BE ' THEN ' •

EMT VC0032-EMTTBL
MOV R1+LOGZ SAVE TYPE OF EXPRESSION
DEBUG STL SUBR
DCH CHA6EX STORE INDICATOR
JSR CHK3rCHECK

DCC E' ELSE ' ' NEXT CHARACTERS MUST BE ' ELSE ' •
CASE OF ELIDED LHOP AND REL

BR P999V

STRING
Z2AA DSH 0

EMT VC0031-EMTBTB

DEBUG STL SUBR

DCH CMDOFX INQUIRE END OF SECOND PART

BR R32 MUST ME END OF LOGICAL EXPRESSION

Z2 MOV LOG4+(R6)

Z2SS MOV LOGZ+L NOT 'IF ' RESTORE POINTERS

MOV LOG2+SL

EMT VC0033-EMTBTB

P6 TST (R6)

BEQ P999B NO FULL RELATIONAL

BMI P99CC PREVIOUS STRING RELATIONAL

JSR PC+P61REL SEE IF ONE OF 6

BR R32 MUST RE END OF LOGICAL EXPRESSION

DEBUG STL SUBR

TST (R6)

BR P5

BPL STSK2

* CASE OF ELIDED LHOP AND REL

P9999 DEBUG

DCH CMDOFX

DEBUG

DCH CMDOEX

BR P5

P999E EMT VC0073-EMTBTB

BR P5

P999D EMT VC0025-EMTBTB

P61 JSR PC+P61REL

BPL STSK2

* CASE OF ELIDED LHOP AND REL

BR P9999
CASE OF ELIDEC I.HOP FOR STRING
EMT VC0073-EMTBL

P99JJ DEBUG
DCH CHB0EX
BR P5

P996 EMT VC0025-EMTBL
BR P99WW
PICK UP ARITHMETIC OPERATOR

!MOV L.LOG2 SAVE STATE OF POINTERS

"EMT VC0025-EMTBL"

\* CHARACTERS \* XOR\" OR \* OR*
B74 MOV LOGZ2, L RESTORE POINTERS
NEQ BAXWOW

MOV LOG2, SL

R33 DSH 0

ADD =2, R6

FULL RELATIONAL SPACE

RTS PC, EXIT

P6IREL MOV L, LOGZ

MOV SL, LOG2

CMP =E', ' + K

BNE STSTXX

EMT VC0024-EMTTBL

CMP =E'. E'TK IS FIRST CHARACTER AN 'E'

BNE B14

CMP =E'. E' TK MUST FINISH WITH SPACE

BNE STSTST

EMT VC0024-EMTTBL

BNE STSTST

EMT VC0024-EMTTBL

MOV COMP, (2(R6)) TURN ON IF COMPON IS NON-ZERO

SEQ SET TO EQUAL

RTS PC, GO BACK

B14 CMP =E'. N', ' K IS FIRST CHARACTER AN 'N'

BNE B2

EMT VC0024-EMTTBL

CMP =E'. E' '+ K IS TWO CHARACTERS NE

BNE STSTST

B2 CMP =E'. L', ' K IS FIRST CHARACTER AN 'L'

BR P7

BNE B33

U254 DSH 0

B7 CMP =E'. T', ' K IS TWO CHARACTERS 'LT' OR 'GT'

BNE B7
G11BB DSH 0

IF SO GO TO G1 & PICK UP A LOGICAL EXP.

JMP G1

G11BB DSH 0

CMP =E* 'K IS FIRST CHARACTER A 'B'

BEQ RGG1 IF SO GO TO G1

MOV L(R6) SAVE STATE OF POINTERS

MOV SL(R6)

G11BB DSH 0

CMP =E* 'K IS FIRST CHAR AN 'H'

BEQ GEWSAA

CMP =O*047* 'K IS FIRST CHARACTER A QUOTE

BEQ GEWSAA

CMP =E* 'K SEE IF OPENING BRACKET

BNE A14

DEBUG STL SUBR

DCH CHCDEX STORE INDICATOR

EMT VC0024-EMTBL

EMT VC0032-EMTBL

MOV RI+GENZ SAVE TYPE OF EXPRESSION: 0 ARITH, 1 LOGICAL

CMP =E* 'K

BNE STSCXX MUST BE CLOSING BRACKET

EMT VC0024-EMTBL

TS T GENZ TEST TYPE

BEQ G2 ARITH

BMI GEWSAB STRING

JMP A64 LOGICAL

BNE STSCXX DSH 0

TRAP VB0011-TRPTBL

GENSAA EMT VC0073-EMTBL

GENSAB DSH 0

CMP =E* 'K

BNE STRG9 MUST BE STRING

MOV L+GEN6

MOV SL+GEN8

EMT VC0024-EMTBL
<table>
<thead>
<tr>
<th>Year</th>
<th>Code1</th>
<th>Code2</th>
<th>Code3</th>
<th>Code4</th>
<th>Code5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1929</td>
<td>014704</td>
<td>001413</td>
<td>19C4</td>
<td>030B</td>
<td>BEQ A33</td>
</tr>
<tr>
<td>1930</td>
<td>014706</td>
<td>022700</td>
<td>19C6</td>
<td>25C0</td>
<td>CMP =E<em>N</em>K IS IT 'N'</td>
</tr>
<tr>
<td>1931</td>
<td>014710</td>
<td>006116</td>
<td>19C8</td>
<td>004E</td>
<td></td>
</tr>
<tr>
<td>1932</td>
<td>014714</td>
<td>014356</td>
<td>19CA</td>
<td>031E</td>
<td>BEQ A4</td>
</tr>
<tr>
<td>1933</td>
<td>014714</td>
<td>013737</td>
<td>19CC</td>
<td>17OF</td>
<td>MOV GEN6*L</td>
</tr>
<tr>
<td>1934</td>
<td>014720</td>
<td>053320</td>
<td>19CE</td>
<td>56F0</td>
<td></td>
</tr>
<tr>
<td>1935</td>
<td>014722</td>
<td>013737</td>
<td>19C0</td>
<td>5604</td>
<td></td>
</tr>
<tr>
<td>1936</td>
<td>014724</td>
<td>053320</td>
<td>19DD</td>
<td>56F2</td>
<td></td>
</tr>
<tr>
<td>1937</td>
<td>014726</td>
<td>053326</td>
<td>19D0</td>
<td>5606</td>
<td>MOV GEN8*SL</td>
</tr>
<tr>
<td>1938</td>
<td>014730</td>
<td>104072</td>
<td>19D4</td>
<td>883A</td>
<td>EMT VC0033-EMTBL</td>
</tr>
<tr>
<td>1939</td>
<td>014732</td>
<td>004430</td>
<td>19FA</td>
<td>0118</td>
<td>BR G3 SET ARITH AND EXIT</td>
</tr>
<tr>
<td>1940</td>
<td>014734</td>
<td>014736</td>
<td>19UC</td>
<td>0119</td>
<td>A33 DSH 0</td>
</tr>
<tr>
<td>1941</td>
<td>014738</td>
<td>104054</td>
<td>19UC</td>
<td>882C</td>
<td>EMT VC0024-EMTBL</td>
</tr>
<tr>
<td>1942</td>
<td>014740</td>
<td>000121</td>
<td>19LE</td>
<td>882C</td>
<td>CMP =E'O*K IS IT 'EQ'</td>
</tr>
<tr>
<td>1943</td>
<td>014742</td>
<td>001364</td>
<td>19L2</td>
<td>0240</td>
<td>BNE A49</td>
</tr>
<tr>
<td>1944</td>
<td>014744</td>
<td>004477</td>
<td>19L4</td>
<td>0107</td>
<td>BR A64</td>
</tr>
<tr>
<td>1945</td>
<td>014746</td>
<td>104054</td>
<td>19L6</td>
<td>024C</td>
<td>G21 DSH 0</td>
</tr>
<tr>
<td>1946</td>
<td>014748</td>
<td>011354</td>
<td>19L8</td>
<td>139F</td>
<td></td>
</tr>
<tr>
<td>1947</td>
<td>014750</td>
<td>022700</td>
<td>19LE</td>
<td>0504</td>
<td>MOV (R6)*L YES RELATIONAL OP: RESTORE TO START OF A.E.</td>
</tr>
<tr>
<td>1948</td>
<td>014750</td>
<td>053324</td>
<td>19FA</td>
<td>5604</td>
<td></td>
</tr>
<tr>
<td>1949</td>
<td>014752</td>
<td>000124</td>
<td>19LA</td>
<td>5606</td>
<td>MOV 2(R6)*SL</td>
</tr>
<tr>
<td>1950</td>
<td>014754</td>
<td>001403</td>
<td>19LC</td>
<td>0303</td>
<td>BEQ A54</td>
</tr>
<tr>
<td>1951</td>
<td>014756</td>
<td>022700</td>
<td>19LF</td>
<td>883A</td>
<td>EMT VC0033-EMTBL</td>
</tr>
<tr>
<td>1952</td>
<td>014758</td>
<td>000105</td>
<td>19EF</td>
<td>0045</td>
<td>G1 DSH 0</td>
</tr>
<tr>
<td>1953</td>
<td>014760</td>
<td>001354</td>
<td>19EF</td>
<td>02EC</td>
<td>G21 DSH 0</td>
</tr>
<tr>
<td>1954</td>
<td>014762</td>
<td>011637</td>
<td>19EF</td>
<td>139F</td>
<td></td>
</tr>
<tr>
<td>1955</td>
<td>014764</td>
<td>053324</td>
<td>19FA</td>
<td>5604</td>
<td></td>
</tr>
<tr>
<td>1956</td>
<td>014766</td>
<td>000002</td>
<td>19FA</td>
<td>0002</td>
<td>MOV 2(R6)*SL</td>
</tr>
<tr>
<td>1957</td>
<td>014768</td>
<td>053326</td>
<td>19FC</td>
<td>5606</td>
<td></td>
</tr>
<tr>
<td>1958</td>
<td>014770</td>
<td>104072</td>
<td>19FE</td>
<td>883A</td>
<td>EMT VC0033-EMTBL</td>
</tr>
<tr>
<td>1959</td>
<td>014772</td>
<td>000002</td>
<td>19FA</td>
<td>0002</td>
<td>PICK UP LOGICAL EXPRESSION</td>
</tr>
<tr>
<td>1960</td>
<td>014774</td>
<td>053328</td>
<td>19FA</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>1961</td>
<td>014776</td>
<td>104072</td>
<td>19FC</td>
<td>5606</td>
<td></td>
</tr>
<tr>
<td>1962</td>
<td>015000</td>
<td>104066</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1963</td>
<td>015002</td>
<td>102701</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1964</td>
<td>015004</td>
<td>000001</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1965</td>
<td>015006</td>
<td>000003</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>015008</td>
<td>000004</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1967</td>
<td>015010</td>
<td>000005</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1968</td>
<td>015012</td>
<td>000006</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1969</td>
<td>015014</td>
<td>000007</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>015016</td>
<td>000008</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1971</td>
<td>015018</td>
<td>000009</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1972</td>
<td>015020</td>
<td>000010</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1973</td>
<td>015022</td>
<td>000011</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1974</td>
<td>015024</td>
<td>000012</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1975</td>
<td>015026</td>
<td>000013</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1976</td>
<td>015028</td>
<td>000014</td>
<td>19FC</td>
<td>083E</td>
<td></td>
</tr>
<tr>
<td>1977</td>
<td>015030</td>
<td>001424</td>
<td>1A18</td>
<td>0314</td>
<td>BEQ NO</td>
</tr>
<tr>
<td>1978</td>
<td>015032</td>
<td>022700</td>
<td>1A1A</td>
<td>0314</td>
<td>CMP =E'H*K</td>
</tr>
<tr>
<td>1979</td>
<td>015034</td>
<td>001426</td>
<td>1A18</td>
<td>0314</td>
<td></td>
</tr>
<tr>
<td>1980</td>
<td>015036</td>
<td>022700</td>
<td>1A1A</td>
<td>0314</td>
<td></td>
</tr>
<tr>
<td>1981</td>
<td>015038</td>
<td>001426</td>
<td>1A18</td>
<td>0314</td>
<td></td>
</tr>
<tr>
<td>1982</td>
<td>015040</td>
<td>020027</td>
<td>1A20</td>
<td>0171</td>
<td>LETL CMP K=Z'A</td>
</tr>
<tr>
<td>1983</td>
<td>015042</td>
<td>000110</td>
<td>1A22</td>
<td>0041</td>
<td></td>
</tr>
<tr>
<td>1984</td>
<td>015044</td>
<td>104916</td>
<td>1A24</td>
<td>001E</td>
<td>BMI NO</td>
</tr>
<tr>
<td>1985</td>
<td>015046</td>
<td>022700</td>
<td>1A26</td>
<td>25C0</td>
<td>CMP =E'Z*K</td>
</tr>
<tr>
<td>1986</td>
<td>015048</td>
<td>000112</td>
<td>1A28</td>
<td>005A</td>
<td></td>
</tr>
<tr>
<td>1987</td>
<td>015050</td>
<td>104913</td>
<td>1A2A</td>
<td>610B</td>
<td>BMI NO</td>
</tr>
<tr>
<td>1988</td>
<td>015052</td>
<td>000410</td>
<td>1A2C</td>
<td>010E</td>
<td>BR YES</td>
</tr>
<tr>
<td>1989</td>
<td>015054</td>
<td>000412</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>015056</td>
<td>000414</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>015058</td>
<td>000416</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1992</td>
<td>015060</td>
<td>000418</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>015062</td>
<td>000420</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>015064</td>
<td>000422</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>015066</td>
<td>000424</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1996</td>
<td>015068</td>
<td>000426</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>015070</td>
<td>000428</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>015072</td>
<td>000430</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>015074</td>
<td>000432</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>015076</td>
<td>000434</td>
<td>1A2C</td>
<td>010E</td>
<td></td>
</tr>
</tbody>
</table>
1972 015056 104074 1A2E 883C  EMT VC0034-EMTTL
1973 015060 001447 1A30 0307  BEQ RETPC
1974  * JSUBR PC+NUMB  * RTS PC
1976 015062 020027 1A32 2017  NUMB CMP K,*E*0' NUMBER TEST
1977 015064 000000 1A44 0030  BMI NO
1978 015070 002070 1A48 25C0  CMP E*9*,K
1979 015072 000071 1A5A 0039  BMI NO
1980 015076 000244 1A6E 0004  YES SEZ
1981 015100 000207 1A40 0087  RETPC RTS PC
1982 015102 000244 1A42 0044  NO CLZ
1983 015104 000207 1A44 0087  RTS PC
1984 015106 013701 1A46 17C1  GETK MV L*R1 RESTORE BUFFER AND CHARACTER IN REG ZERO
1985 015112 162701 1A6A E5C1  SUB E=+R1
1986 015114 000000 1A4C 0004  MOVV (R1)+KCH3
1987 015120 005316 1A60 56CE  MOVV (R1)+KCH2
1988 015122 112137 1A62 94F8  MOVV (R1)+KCH1
1989 015126 112137 1A64 56D0  MOVV (R1)+K
1990 015126 000207 1A4C 0007  RTS PC
1991 015126 011127 1A5E 10BE  STL MOV R1,SAVR1
1992 015140 005376 1A60 56FE  MOV SL*R2
1993 015142 013702 1A62 17C2  MOV (R6)+R1
1994 015144 005326 1A64 56D6  MOV (R6)+R1
1995 015146 111061 1A66 1361  MOV (R1)+(R2)+
1996 015152 102327 1A6A 109F  MOV R2*SL
1997 015166 005201 1A6E 0041  INC R1
1998 015166 011160 1A70 17C4  MOV R1+(R6)
1999 015166 013701 1A72 17C1  MOV SAVR1*R1
2000 015166 005376 1A74 56FE  RTI
2001 015170 112401 1A7B 9501  CHECK MOVV (CHKR)+R1 CHECK A SEQUENCE OF CHARACTERS
2002 015172 122701 1A7A A5C1  CMPB =E**+R1
2003 015174 000072 1A7C 003A  BEQ CHECK1
2004 015176 001144 1A7E 0304  BEQ CHECK1
2005 015172 020001 1A80 02E0  CMP K,R1
2006 015172 001006 1A82 02E0  INC CHECK2
2007 015172 104054 1A84 09C2  EMT VC0024=EMTTL
2008 015172 000770 1A86 01F8  ED CHECK
2009 015172 013701 1A88 0000  CHECK1 DSH 0
2010 015172 005204 1A8A 00A4  INC CHKR
2011 015172 042704 1A8A 45C4  EIC =X*0001',CHKR
2012 015172 0000C1 1A8C 0001  CHECK3 RTS CHKR
2013 015172 000204 1A8E 0004  CHECK3 RTS CHKR
2014 015172 012400 1A90 0000  TRAP V00011-TRPTBL
2015 015172 174400 1A90 0900  STK MOV SL+R1 LOAD POINTER TO VECTOR
GNCl MOV KCH2*KCH3 GET NEXT CHARACTER

MOV L*R1

DEC R1

CMP R1*INPB1

BDI STKERR

ONE STK1

MOVB (R1) R0

MOV R1*INPB1

ONE STK1

MOV L*R1
STORE POINTER

GNC MOV KCH2*KCH3 GET NEXT CHARACTER

MOV L*R1 POINTER TO LAST + 1 CHARACTER TAKEN FROM STRING

MOVB (R1) K*KCH1

GET NEXT CHAR. FROM STRING! INC THE POINTER

CLR STRCHR

BEQ F2222

CMP =E'<' RO

GNC1 IF EQ MEANS THAT THE CHARACTERS HAVE ALL BEEN GIVEN TO THE PROG.

* WILL HAVE TO GET ONE FROM THE KEYBOARD

MOVB (R1) K*KCH1

GET NEXT CHAR. FROM STRING! INC THE POINTER

CLR STRCHR

BEQ RUBOUT

CMP =E'<' RO

TRAP VBO011-TRPTBL

GET CHARACTER FROM KEYBOARD! MUST NOT Clobber R1

MOVB LASTCH+LASTCH+1

MOVB R2*INPB

MOV L*R1
STORE POINTER

BEQ F2222

CMP =E'<' RO

REG RUBOUT

CMP =E'<' RO

3743 027142 010137 2E62 105F MOV R1*SL STORE POINTER

3744 027144 053326 2E94 5606 MOV L*R1

3745 027152 005301 2E6A 0AC1 DEC R1

3746 027154 020137 2E6C 203F CMP R1*INPB1

3747 027156 053906 2E6E 5706 BHI STKERR

3748 027160 101363 2E70 42F3 ONE STK1

3749 027162 010144 2E72 02FU MOVB (R1) R0

3750 027164 010137 2E76 105F MOV R1*INPB1

3751 027170 005306 2E76 5706 ONE STK1

3752 027172 104014 2E7A 080C EMT VC0007-EMTBL

3753 027174 013737 2EC7 170F STK1 DSH 0

3754 027202 013737 2E92 170F MOV KCH1*KCH2

3755 027204 053322 2E94 5602 MOV KCH1*KCH2

3756 027206 053320 2E96 5600 MOV KCH1*KCH2

3757 027210 010037 2E98 101F MOV L*R1

3758 027212 053322 2E9A 5602 MOV L*R1

3759 027214 013701 2E9C 17C1 MOV L*R1

3760 027216 053324 2E9E 5604 MOV L*R1

3761 027220 013702 2EA0 17C2 MOV L*R1

3762 027222 053304 2EA2 5704 MOV L*R1

3763 027224 020142 2EA4 2042 MOV L*R1

3764 027226 001412 2EA6 030A MOV L*R1

3765 027230 112100 2E98 9440 MOV L*R1

3766 027232 005037 2E9A 0A1F MOV L*R1

3767 027234 055002 2E9C 5A02 MOV L*R1

3768 027236 010137 2EA6 105F MOV L*R1

3769 027240 053324 2EA8 5604 MOV L*R1

3770 027242 022700 2EAA 2500 MOV L*R1

3771 027244 177777 2EAC FFFF MOV L*R1

3772 027246 001401 2EAG 0301 MOV L*R1

3773 027250 000207 2EAB 0087 MOV L*R1

3774 027252 022700 2EAC 2500 MOV L*R1

3775 027254 104152 2EAE 86A MOV L*R1

3776 027256 012757 2EAE 150F MOV L*R1

3777 027260 000077 2E00 000F MOV L*R1

3778 027262 055002 2E02 5A02 MOV L*R1

3779 027264 010237 2E04 109F MOV L*R1

3780 027266 055322 2E06 5ADA MOV L*R1

3781 027270 113737 2E08 97DF MOV L*R1

3782 027272 054644 2E0A 59A4 MOV L*R1

3783 027274 054645 2E0C 59A5 MOV L*R1

3784 027276 110037 2E0E 901F MOV L*R1

3785 027280 034944 2E10 59A4 MOV L*R1

3786 027282 023727 2E12 2707 MOV L*R1

3787 027284 054644 2E14 59A4 MOV L*R1

3788 027290 037076 2E16 3E3E MOV L*R1

3789 027292 001414 2E18 030C MOV L*R1

3790 027294 022700 2E1A 2500 MOV L*R1

3791 027296 000074 2E1C 003C MOV L*R1

3792 027298 001415 2E1E 030D MOV L*R1

3793 02729A 022700 2E20 2500 MOV L*R1
MOV INPZZ,R0

SUU RUBAAA+RO

MOV R0,INPA1

CMP R0,=INPUT+1

BPL RUB9

ALT3 MOV ALT3A+PC

PRINT ALL OVER
| 3961 | 030346 001062 30E6 0202 | BNE NBS5 |
| 3962 | 030350 012700 30EA 15CD | MOV =E';,R0 |
| 3963 | 030354 000072 30CA 003A | |
| 3964 | 030354 032700 30EC 35C0 | NBS DSH 0 |
| 3965 | 030356 000140 30LE 0060 | CMP =0'140',R0 |
| 3966 | 030360 001064 30F0 0204 | SEE IF CONTROL CHARACTER |
| 3967 | 030362 05037 30F2 0A1F | BNE NRS123 |
| 3968 | 030364 053914 30F4 560C | CLR HNECHO |
| 3969 | 030366 005037 30F6 0A1F | NEVER DING A CONTROL CHARACTER |
| 3970 | 030370 052514 30F8 554C | CLR MODE2A |
| 3971 | 030372 002700 30FA 25C0 | NBS123 DSH 0 |
| 3972 | 030374 000003 30FC 0003 | CONTROL C1 TURN ON TABLE TRACE |
| 3973 | 030376 001065 30FE 2003 | |
| 3974 | 030380 012737 3100 150F | BNE TBLRS1 |
| 3975 | 030402 000077 3102 003F | MOV =63',TBLTRG |
| 3976 | 030404 052410 3104 550B | |
| 3977 | 030406 006117 3106 005F | TBLRS3 JMP KEY8CH |
| 3978 | 030410 027506 3108 2F6E | TBLRS1 CMP =0'030',R0 |
| 3979 | 030412 022700 310A 25C0 | CONTROL XI TURN OFF TABLE TRACE |
| 3980 | 030414 000030 310C 0018 | |
| 3981 | 030416 01003 310E 0203 | RNE TBLRS2 |
| 3982 | 030420 005037 3110 0A1F | CLR TBLTRG |
| 3983 | 030422 052410 3112 550B | |
| 3984 | 030424 000070 3114 01F8 | BR TBLRS3 |
| 3985 | 030426 3116 | TBLRS2 DSH 0 |
| 3986 | 030428 022700 3116 25C0 | CMP =0'004',R0 |
| 3987 | 030430 000004 3118 0004 | CONTROL D1 TURN ON PAPER TAPE READER |
| 3988 | 030432 010004 311A 0204 | RNE COMO |
| 3989 | 030434 012737 311C 150F | MOV =63',PARIND |
| 3990 | 030436 000077 311E 003F | |
| 3991 | 030440 052402 3120 5502 | BR TBLRS3 |
| 3992 | 030442 000761 3122 01F1 | COMO DSH 0 |
| 3993 | 030444 3124 | CMP =0'005',R0 |
| 3994 | 030446 000005 3126 0005 | CONTROL E1 TURN OFF PAPER TAPE READER |
| 3995 | 030450 01010 3128 0208 | BNE COME |
| 3996 | 030452 005037 312A 0A1F | CLR PARIND |
| 3997 | 030454 052402 312C 5502 | |
| 3998 | 030456 005737 312E 00DF | TST SIMULA |
| 3999 | 030460 056376 3130 54FE | |
| 4000 | 030462 01791 3132 03E9 | BEQ TBLRS3 |
| 4001 | 030464 005337 3134 0A1F | CLR PRS |
| 4002 | 030466 177550 3136 FF68 | |
| 4003 | 020470 000746 3138 01E6 | BR TBLRS3 |
| 4004 | 020472 313A | COME DSH 0 |
| 4005 | 020474 022700 313A 25C0 | CMP =0'101',R0 |
| 4006 | 030476 00010 313C 0008 | CONTROL H1 TURN ON PUNCH; ECHO INPUT |
| 4007 | 030478 012737 3140 150F | RNE COMH |
| 4008 | 030502 000077 3142 003F | MOV =63',HSPDAT |
| 4009 | 030504 00004 3144 5500 | |
| 4010 | 030506 00537 3146 010F | BR TBLRS3 |
| 4011 | 030510 3148 | COMH DSH 0 |
| 4012 | 030512 00010 314A 0009 | CMP =0'011',R0 |
| 4013 | 030514 01003 314C 0203 | CONTROL I1 TURN OFF PUNCH; ECHO INPUT |
| 4014 | 030516 00002 314E A1F | RNE COMI |
| 4015 | 030520 052400 3150 5500 | CLR HSPDAT |
```assembly
3999 030522 000731 3152 01D9 KEYJCC BR TBLRS3
4000 030524 3154 COMI DSH 0
4001 030524 022700 3154 25C0 CMP =0*017*+R0 CONTROL 01 TURN ON PUNCH: ECHO OUTPUT
4002 030520 000017 3156 000F BNE COMO
4003 030532 012757 315A 15DF MOV =63*FASTXT
4004 030534 000077 315C 003F
4005 030536 052404 315E 5504
4006 030542 022700 3162 25C0 CMP =0*020*+R0 CONTROL P: TURN OFF PUNCH: ECHO OUTPUT
4007 030546 001003 3166 0203 BNE COMP
4008 030550 005037 3168 0A1F CLR FASTXT
4009 030552 052404 316A 5504
4010 030554 000072 316C 01F2 BR KEYJCC
4011 030556 022700 316E 25C0 CMP =0*025*+R0 CONTROL U: TURN ON PROUT8
4012 030562 001004 3172 0204 BNE COMU
4013 030564 012737 3174 15DF MOV =63*PROUT8
4014 030566 000077 3176 003F
4015 030568 052406 3178 5506
4016 030572 000753 317A 01EB BR KEYJCC
4017 030574 010003 317C COMU DSH 0
4018 030574 022700 317C 25C0 CMP =0*026*+R0 CONTROL V: TURN OFF PROUT8
4019 030580 000026 317E 0316
4020 030600 001003 3180 0203 BNE COMV
4021 030602 005037 3182 0A1F CLR PROUT8
4022 030604 052406 3184 5506
4023 030606 000745 3186 01EB BR KEYJCC
4024 030610 3188 COMV DSH 0
4025 030610 022700 318A 25C0 CMP =0*001*+K
4026 030612 000001 318C 0001
4027 030614 001007 318E 0207 BNE MODLB
4028 030616 012700 3190 E5C0 MOV K*MODE2B
4029 030620 000001 3192 0701
402A 030622 000002 3194 01F2
402B 030624 032420 3196 5516
402C 030626 005037 3198 001F CLR MODE2A
402D 030630 052514 319a 554C
402E 030632 010733 319c 01DU BR KEYJCC
402F 030634 022700 319e 25C0 MODLB CMP =0*002*+K
4030 030636 000002 319f 0002
4031 030640 012700 31A0 03F6 BEu MODLA
4032 030642 022700 31A2 25C0 CMP =0*022*+R0 CONTROL R
4033 030644 000022 31A4 0012 BNE ECHI
4034 030646 01102 31A6 0202 JMP CLRBLLL
4035 030650 001537 31A8 005F
4036 030652 052370 31AA 54FB
4037 030654 31AC ECHI DSH 0
4038 030656 022700 31AE 25C0 CMP =0*007*+R0 CONTROL G
4039 030658 000007 31AF 0007 * CONTROL G SHOULD WORK UNDER INTERRUPT
403A 030660 001150 31B0 0258 BNE TYPQA2
403B 030662 000157 31B2 005F JMP SSTRT
403C 030664 035464 31B4 3534
403D 030666 31B6 TYPECH DSH 0
403E 030668 057373 31B8 08DF
403F 030670 052376 31BA 54FE
```
BEQ TYPECZ
TST OPTION

BEQ TYPECZ
TST BKS

DPL TYPECZ

MOV =1, ASYCHR

HAS BEEN ECHOED

DONT TYPE

BEQ TYPAL2

DONT TYPE

BEQ TYPAL3

TYPAL3 DSH 0

TRON
4121 031222 001431 3292 0319  "BEQ HSPBB
4122 031224 022700 3294 25C0  "CMP =0'020''R0
4123 031226 000020 3296 0010  "CONTROL P
4124 031228 001426 3298 0316  "BEQ HSPBB
4125 031230 014231 329A 25C0  "CMP =0'004''R0
4126 031232 000004 329C 0004  "CONTROL D
4127 031234 001420 329E 0310  "BEQ HSPBB
4128 031236 022700 329G 25C0  "CMP =0'003''R0
4129 031238 001415 329I 0300  "CONTROL C
4130 03123A 022700 329K 25C0  "BEQ HSPBB
4131 03123C 000030 329M 0010  "CONTROL X
4132 03123E 005737 329O 0003  "BEQ HSPBB
4133 031240 052400 329A 5500  "HSPAAA TST HSPDAT
4134 031242 001407 329C 0307  "BEQ HSPBB
4135 031244 137554 329E 0003  "BIT =X'8080'PPS
4136 031246 017771 329G 03F9  "REQ HSPAAA
4137 031248 017771 329I 03F9  "ERROR
4138 03124A 110037 329K 901F  "ECHO THE INPUT
4139 03124C 177554 329O FF6C  "LAD22A LADOUT
4140 03124E 001407 329Q 0007  "RTS PC
4141 031250 001407 329R 0007  "LAD22A LADOUT
4142 031252 001407 329S 0007  "CLR FORPER
4143 031254 001407 329T 0007  "LA022A INC R4
4144 031256 001407 329U 0007  "MOV R4(R6)
COULD COUNT UP TO 31: THEN WOULD NOT NEED CHECK IN LHOP

* OPTIONS DSH 0

EMT VC0024-EMTTL

BIT =0'140' CONTROL CHARACTER

BEG ORSTA

CMP =0'140' CONTROL CHARACTER

BEG ORSTA
********** IF ! TO & SEE IF CAN EXPAND INTO INPUT

CMP K:=O'41'

BMI ZZA001

CMP =O'46',K

BMI ZZA001

TSTB ASYINO-O'41'(R0)

DEQ ASYNIX

MOV =SEG3+r2

MOVB =E'0',*(R2)+

MOVB =E'9',*(R2)+

ADD =O'20',R0

MOVB KG,(R2)

ENT VC0144-EMTTLB

MOV INPA1,R4

DEC H4

ADD =19,R1

MOVB (R1)+R0

ZZA002 MOV3 (R1)+R3

RMI ZZA002

CMP =E'3',R3.

MOV R3,(R4)+

BR ZZA002

BNE CIA. EDIT CANNOT BE STORED

MOV SL,R2

RNC CIA. EDIT CANNOT BE STORED

MOV SL,R2

TSTB MADMOD

IS IT TO BE STORED

MOV SL,R2

ZZA001 DSH 0

IS IT TO BE STORED

MOV R2+SL

ZZA003 DSH 0
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4399</td>
<td>032516</td>
<td>104420</td>
<td>354E</td>
<td>8910</td>
<td>* JSUBR PC.=AREX</td>
</tr>
<tr>
<td>4400</td>
<td>032520</td>
<td>10456</td>
<td>3550</td>
<td>886E</td>
<td>* RTS PC</td>
</tr>
<tr>
<td>4401</td>
<td>032524</td>
<td>053340</td>
<td>3554</td>
<td>56EU</td>
<td>TRAP VB0031-TRMTBL</td>
</tr>
<tr>
<td>4402</td>
<td>032528</td>
<td>013702</td>
<td>355A</td>
<td>17C2</td>
<td>CIA DSH 0</td>
</tr>
<tr>
<td>4403</td>
<td>032532</td>
<td>055326</td>
<td>355A</td>
<td>5606</td>
<td>GO BACK</td>
</tr>
<tr>
<td>4404</td>
<td>032534</td>
<td>112722</td>
<td>355C</td>
<td>9502</td>
<td>EMT VC0076-EMTBL</td>
</tr>
<tr>
<td>4405</td>
<td>032536</td>
<td>005356</td>
<td>355E</td>
<td>000E</td>
<td>* LOOK FOR ASYNCH</td>
</tr>
<tr>
<td>4406</td>
<td>032540</td>
<td>010237</td>
<td>3500</td>
<td>109F</td>
<td>TST MAJMOD</td>
</tr>
<tr>
<td>4407</td>
<td>032542</td>
<td>038289</td>
<td>3592</td>
<td>5606</td>
<td>MUST BE IMMEDIATE BUT WILL BE STORED</td>
</tr>
<tr>
<td>4408</td>
<td>032546</td>
<td>000101</td>
<td>3500</td>
<td>041</td>
<td>MOVB =X'EE',(R2)+</td>
</tr>
<tr>
<td>4409</td>
<td>032550</td>
<td>000123</td>
<td>3503</td>
<td>0034</td>
<td>MOVE R2,SL</td>
</tr>
<tr>
<td>4410</td>
<td>032551</td>
<td>000004</td>
<td>3509</td>
<td>0034</td>
<td>IOT</td>
</tr>
<tr>
<td>4411</td>
<td>032552</td>
<td>002111</td>
<td>350A</td>
<td>0034</td>
<td>DCB E'A',POASY1-LADTAB</td>
</tr>
<tr>
<td>4412</td>
<td>032553</td>
<td>000064</td>
<td>350B</td>
<td>0034</td>
<td>DCB E'S',POASY1-LADTAB</td>
</tr>
<tr>
<td>4413</td>
<td>032554</td>
<td>000016</td>
<td>350C</td>
<td>0034</td>
<td>DCB E'Y',POASY1-LADTAB</td>
</tr>
<tr>
<td>4414</td>
<td>032555</td>
<td>000064</td>
<td>350D</td>
<td>0034</td>
<td>DCB E'N',POASY1-LADTAB</td>
</tr>
<tr>
<td>4415</td>
<td>032556</td>
<td>000183</td>
<td>350E</td>
<td>0034</td>
<td>DCB E'C',POASY1-LADTAB</td>
</tr>
<tr>
<td>4416</td>
<td>032557</td>
<td>000004</td>
<td>350F</td>
<td>0034</td>
<td>DCB E'H',POSTSC-LADTAB</td>
</tr>
<tr>
<td>4417</td>
<td>032559</td>
<td>000060</td>
<td>3510</td>
<td>010</td>
<td>DCB E',POSTSC-LADTAB</td>
</tr>
</tbody>
</table>
| 4418 | 032559 | 000072 | 3574 | 003A | DCH E';
| 4419 | 032560 | 005237 | 3576 | 009F | INC ASYDIG |
| 4420 | 032560 | 055350 | 3578 | 5618 | IT IS AN ASYNCH STATEMENT |
| 4421 | 032562 | 020027 | 357A | 2017 | CMP x=0'41' |
| 4422 | 032564 | 000441 | 357C | 0021 | BMI ASYNIX |
| 4423 | 032600 | 027270 | 3560 | 25C0 | CMP =0'46',K |
| 4424 | 032604 | 000840 | 3592 | 0226 | BMI ASYNIX |
| 4425 | 032604 | 010912 | 3504 | 61A | EMT VC0024-EMTBL |
| 4426 | 032606 | 027270 | 3566 | 882C | CMP =E'44',K |
| 4427 | 032610 | 000772 | 3583 | 003A | RESTORE POINTERS |
| 4428 | 032614 | 001005 | 358C | 0205 | REG ASYDEL |
| 4429 | 032616 | 027300 | 359E | 25C0 | CMP =E'44',K |
| 4430 | 032620 | 000040 | 359D | 0020 | BNE ASYNIX |
| 4431 | 032622 | 040003 | 592 | 0203 | EMT VC0024-EMTBL |
| 4432 | 032624 | 104904 | 3594 | 882C | BR NEWSUB |
| 4433 | 032626 | 000004 | 3596 | 0104 | ASYDEL RTS PC |
| 4434 | 032632 | 009S57 | 359A | 0005F | ASYNIX JMP STSC |
| 4435 | 032634 | 012236 | 359C | 149E | NO GO, POGO |
| 4436 | 032636 | 104156 | 359E | 886E | ASY1 EMT VC0076-EMTBL |
| 4437 | 032640 | 005037 | 35A0 | 0A1F | * JSUBR PC.=NEWSUB |
| 4438 | 032642 | 053330 | 35A2 | 560A | * RTS PC |
| 4439 | 032644 | 005037 | 35A4 | 0A1F | NEWSUB DSH 0 |
| 4440 | 032646 | 055322 | 35A6 | 560A | CLP FORINO |
| 4441 | 032650 | 005037 | 35A8 | 0A1F | CLR FORPER |
| 4442 | 032640 | 35A0 | 0A1F | CLR ELSFOR |
CMP =E' */0
ONE C1B
C1A2 D5M
E3T VC0024-EMTTL
CMP =E' */0
BNE C1A2
MOV =X'0'0'*/STYP
C1B D5M
E3T VC0033-EMTTL
INC FORPER
DCB E'G'*/PDA11-LADTAB
DCB E'O'*/PCEXX-LADTAB
DCB E' */POEXX-LADTAB
DCB E' */POEXX-LADTAB
DCB E'O'*/PCEXX-LADTAB
DCB E' */POEXX-LADTAB
DCB E'O'*/PDSHGO-LADTAB
DCB E'O'*/PDSHGO-LADTAB
DCB E'O'*/PDSHGO-LADTAB
DCB E'O'*/PDSHGO-LADTAB
DCB E'O'*/PDSHGO-LADTAB
CLOSED SQUARE BRACKET
INDICATE LOGICAL VARIABLE ACCEPT OP.

OPENING SQUARE BRACKET
INDICATE ARITH VARIABLE
STORE THE 'H'
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4714</td>
<td>034176</td>
<td>0104056 387E 082E</td>
<td>EMT VC0025-EMTTBL AREX</td>
</tr>
<tr>
<td>4715</td>
<td>034200</td>
<td>002774 38B0 25CD</td>
<td>CMP =E'11',K</td>
</tr>
<tr>
<td>4716</td>
<td>034202</td>
<td>000005 38d2 0052</td>
<td>BNE STSKNB</td>
</tr>
<tr>
<td>4717</td>
<td>034206</td>
<td>0104054 38b6 882C</td>
<td>EMT VC0024-EMTTBL</td>
</tr>
<tr>
<td>4718</td>
<td>034210</td>
<td>000745 38b8 01E5</td>
<td>BR HII2D2</td>
</tr>
<tr>
<td>4719</td>
<td>034212</td>
<td>005737 38da 08DF</td>
<td>DI1 DSH 0</td>
</tr>
<tr>
<td>4720</td>
<td>034214</td>
<td>005330 38dc 56E0</td>
<td>TST MAVMD</td>
</tr>
<tr>
<td>4721</td>
<td>034216</td>
<td>001415 38e1 030D</td>
<td>BNE C11</td>
</tr>
<tr>
<td>4722</td>
<td>034220</td>
<td>000004 38f9 0084</td>
<td>IOT</td>
</tr>
<tr>
<td>4723</td>
<td>034222</td>
<td>000101 3902 0041</td>
<td>DCB E'A'*POC11-LADTAB</td>
</tr>
<tr>
<td>4724</td>
<td>034223</td>
<td>000024 3953 0014</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4725</td>
<td>034224</td>
<td>001013 39f4 0043</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4726</td>
<td>034225</td>
<td>000016 39e5 000C</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4727</td>
<td>034226</td>
<td>000103 39f5 0043</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4728</td>
<td>034227</td>
<td>000104 39f7 000C</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4729</td>
<td>034228</td>
<td>000105 39f6 0043</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4730</td>
<td>034229</td>
<td>000014 39f6 0043</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4731</td>
<td>034230</td>
<td>000124 39f3 000C</td>
<td>DCB E'C'*POEXX-LADTAB</td>
</tr>
<tr>
<td>4732</td>
<td>034231</td>
<td>000020 4c0e 0020</td>
<td>BCE E 'S',PDD11-LADTAB</td>
</tr>
<tr>
<td>4733</td>
<td>034232</td>
<td>000016 4c0f 0016</td>
<td>IF NOT 'S' GO TO D11</td>
</tr>
<tr>
<td>4734</td>
<td>034233</td>
<td>000014 4c0a 0016</td>
<td>DCB E'T',PDE11-LADTAB</td>
</tr>
<tr>
<td>4735</td>
<td>034234</td>
<td>000004 4c0b 0016</td>
<td>IF NOT 'ST' GO TO E11</td>
</tr>
<tr>
<td>4736</td>
<td>034235</td>
<td>000026 4c0c 0016</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4737</td>
<td>034236</td>
<td>000004 4c0d 0016</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4738</td>
<td>034237</td>
<td>000004 4c0e 0016</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4739</td>
<td>034238</td>
<td>000014 4c0f 0016</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4740</td>
<td>034239</td>
<td>000020 4c10 0020</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4741</td>
<td>034240</td>
<td>000068 4c11 0016</td>
<td>DCB E'O',PDEXX-LADTAB</td>
</tr>
<tr>
<td>4742</td>
<td>034241</td>
<td>000062 4c12 0016</td>
<td>MOV =X'E3',STYPE</td>
</tr>
<tr>
<td>4743</td>
<td>034242</td>
<td>000062 4c13 0016</td>
<td>MOV =X'E3',STYPE</td>
</tr>
<tr>
<td>4744</td>
<td>034243</td>
<td>000004 4c14 0016</td>
<td>CMP =E' H',K</td>
</tr>
<tr>
<td>4745</td>
<td>034244</td>
<td>000004 4c15 0016</td>
<td>IS IT A SPACE</td>
</tr>
<tr>
<td>4746</td>
<td>034245</td>
<td>000004 4c16 0016</td>
<td>BNE E13 CANNOT BE 'S'</td>
</tr>
<tr>
<td>4747</td>
<td>034246</td>
<td>000020 4c17 0020</td>
<td>EMT VC0024-EMTTBL</td>
</tr>
<tr>
<td>4748</td>
<td>034247</td>
<td>000004 4c18 0020</td>
<td>INC STOPER</td>
</tr>
<tr>
<td>4749</td>
<td>034248</td>
<td>000014 4c19 0020</td>
<td>BR PIGGY</td>
</tr>
<tr>
<td>4750</td>
<td>034249</td>
<td>000014 4c1a 0020</td>
<td>E11 IOT</td>
</tr>
<tr>
<td>4751</td>
<td>034250</td>
<td>000014 4c1b 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4752</td>
<td>034251</td>
<td>000004 4c1c 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4753</td>
<td>034252</td>
<td>000004 4c1d 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4754</td>
<td>034253</td>
<td>000004 4c1e 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4755</td>
<td>034254</td>
<td>000004 4c1f 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4756</td>
<td>034255</td>
<td>000004 4c20 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
<tr>
<td>4757</td>
<td>034256</td>
<td>000004 4c21 0020</td>
<td>DCB E'Y',PDSUS2-LADTAB</td>
</tr>
</tbody>
</table>
• COULD DO AS IMMEDIATE; STORE RETURN ADDRESS AS ZERO

* IOT

DCB E'\text{C}++PDH11-LADTAB

DCB E'\text{A}++PDCLR-LADTAB

DCB E'\text{L}++PDEXX-LADTAB

DCB E' \text{A}++PDEXX-LADTAB

JMP SH2JKL

DCB E\text{'A'}+PDHSL-LADTAB

DCB E' \text{E'}++PDEXX-LADTAB

H11EXX DSH 0

JMP EXX

H11 IOT

DCB E'\text{P}++PDH11H-LADTAB

DCB E'\text{A}++PDH51-LADTAB

DCB E' \text{E'}++H13 MOV =X'\text{E'}++\text{STYPE}

EMT VC0103-EMTBL

CMP =E'++R0

REQ J13

CMPP =E'++R0

DNE HSTSC

EMT VC0024-EMTBL

CMP PANEG\text{K} UP ARROW

DNE HH74C

EMT VC0024-EMTBL

CMP =E'++K

REQ HH74B

CMPP =E'++K

BEQ J13

HSTSC DSH 0

HSTSC DSH 0

TRAP VB0011-TRPTBL

HH74B DSH 0

EMT VC0024-EMTBL

BH HH74C

CLR TST MAJMOD
GET NAME OF VARIABLE
SSTRT  DSH  0
SIMXXX  DSH  0

MOV=15FR0  NUMBER  OF  WORDS
MOV=CRUD»R1  SET  UP  ADDRESS

MOV=0'007»»R0
BEO SIMDIM
TST SIMULA

MOV =0'005»»R0
BNE STKTF
MOV=80»SAVE

MOV =0'010»»STSEQ

MOV =X'7FF8'»SYMHI

MOV =X'0FF8'»SYMHI

MOV =X'0FF8'»SYMHI
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5014</td>
<td>035576 001375 3B7E 02FD</td>
<td>BNE SSLOOP</td>
</tr>
<tr>
<td>5015</td>
<td>* CLR (R1)+</td>
<td>CKUD</td>
</tr>
<tr>
<td>5016</td>
<td>* CLR (R1)+</td>
<td>PAROP</td>
</tr>
<tr>
<td>5017</td>
<td>* CLR (R1)+</td>
<td>TRACE</td>
</tr>
<tr>
<td>5018</td>
<td>* CLR (R1)+</td>
<td>GLLTR</td>
</tr>
<tr>
<td>5019</td>
<td>* CLR (R1)+</td>
<td>GLTR</td>
</tr>
<tr>
<td>5020</td>
<td>* CLR (R1)+</td>
<td>LSTERR</td>
</tr>
<tr>
<td>5021</td>
<td>* CLR (R1)+</td>
<td>PKEVAD</td>
</tr>
<tr>
<td>5022</td>
<td>* CLR (R1)+</td>
<td>SPECIF</td>
</tr>
<tr>
<td>5023</td>
<td>* CLR (R1)+</td>
<td>PREVCH</td>
</tr>
<tr>
<td>5024</td>
<td>* CLR (R1)+</td>
<td>OPTION</td>
</tr>
<tr>
<td>5025</td>
<td>* CLR (R1)+</td>
<td>WUNMOR</td>
</tr>
<tr>
<td>5026</td>
<td>* CLR (R1)+</td>
<td>ASYIND</td>
</tr>
<tr>
<td>5027</td>
<td>* CLR (R1)+</td>
<td>ASYIND+2</td>
</tr>
<tr>
<td>5028</td>
<td>* CLR (R1)+</td>
<td>ASYIND+4</td>
</tr>
<tr>
<td>5029</td>
<td>* CLR (R1)+</td>
<td>MODE2A</td>
</tr>
<tr>
<td>5030</td>
<td>035560 104016 3D60 881E</td>
<td>EMT VC0016-EMTTL</td>
</tr>
<tr>
<td>5031</td>
<td>035562 012701 3D42 1561</td>
<td>MOV =HEADY&gt;R1</td>
</tr>
<tr>
<td>5032</td>
<td>035564 007324 3D84 0ED4</td>
<td>EMT VC0010-EMTTL</td>
</tr>
<tr>
<td>5033</td>
<td>035566 104016 3D60 881E</td>
<td>EMT VC0000-EMTTL</td>
</tr>
<tr>
<td>5034</td>
<td>035568 013737 3D66 176F</td>
<td>MOV STORLO&gt;STPTR</td>
</tr>
<tr>
<td>5035</td>
<td>03556A 052546 3D6E 5514</td>
<td>MOV STORLO&gt;SYMLO</td>
</tr>
<tr>
<td>5036</td>
<td>03556C 005037 3D96 0A1F</td>
<td>DROP CLR RETADR</td>
</tr>
<tr>
<td>5037</td>
<td>03556E 052554 3D98 556C</td>
<td>NO RETURN ADDRESS</td>
</tr>
<tr>
<td>5038</td>
<td>035570 012737 3D9A 150F</td>
<td>STRTF DSH 0</td>
</tr>
<tr>
<td>5039</td>
<td>035572 000001 3D4C 0031</td>
<td>MOV =E'1'&gt;STATE</td>
</tr>
<tr>
<td>5040</td>
<td>035574 052550 3D9E 5568</td>
<td>STRTF CLR OPTION</td>
</tr>
<tr>
<td>5041</td>
<td>035576 013737 3D9C 176C</td>
<td>MOV STKTOP&gt;R6</td>
</tr>
<tr>
<td>5042</td>
<td>035578 052562 3D9E 5542</td>
<td>MOV STKTOP&gt;R6</td>
</tr>
<tr>
<td>5043</td>
<td>03557A 013766 3D9C 5518</td>
<td>EMT VC0016-EMTTL</td>
</tr>
<tr>
<td>5044</td>
<td>03557C 013766 3D9C 5518</td>
<td>MOV =E'2'&gt;R0</td>
</tr>
<tr>
<td>5045</td>
<td>03557E 012737 3D9A 150F</td>
<td>STRTF2 MOV STKTOP&gt;R6</td>
</tr>
<tr>
<td>5046</td>
<td>035580 052546 3D9E 5568</td>
<td>MOV =INPUT&gt;INPA1</td>
</tr>
<tr>
<td>5047</td>
<td>035582 005037 3D60 0A1F</td>
<td>SET POINTER TO BUFFER; NO STORED STRING</td>
</tr>
<tr>
<td>5048</td>
<td>035584 052550 3D9E 5568</td>
<td>MOV =INPUT&gt;INPB1</td>
</tr>
<tr>
<td>5049</td>
<td>035586 012737 3D9A 150F</td>
<td>ALL CHARACTERS TO BE TYPED</td>
</tr>
<tr>
<td>5050</td>
<td>035588 000001 3D4C 0031</td>
<td>MOV =RESTRZ&gt;ALT2A</td>
</tr>
<tr>
<td>5051</td>
<td>03558A 052546 3D9E 5568</td>
<td>STSC RESTART</td>
</tr>
<tr>
<td>5052</td>
<td>03558C 013766 3D9C 5518</td>
<td>MOV =RESTRZ&gt;ALT3A</td>
</tr>
<tr>
<td>5053</td>
<td>03558E 013766 3D9C 5518</td>
<td>ABORT RESTART</td>
</tr>
<tr>
<td>5054</td>
<td>035590 012737 3D9A 150F</td>
<td>RESTZ MOV STKTOP&gt;R6</td>
</tr>
<tr>
<td>5055</td>
<td>035592 005037 3D60 594C</td>
<td>MOV R6&gt;TYSMR6</td>
</tr>
<tr>
<td>5056</td>
<td>035594 052546 3D9E 5568</td>
<td>SAVE REG 6</td>
</tr>
<tr>
<td>5057</td>
<td>035596 012737 3D9A 150F</td>
<td>MOV =LOOKLN&gt;TYSAM</td>
</tr>
</tbody>
</table>
BEQ MAXINE

MOV =63,MAJMOD  ASSUME WILL BE STORED

GET NEXT SEQ NUMBER AND A SPACE

NOT A NUMBER?  ABORT BACK TO COL 1

ACCEPT COL 1  'N'

DELETE STORED STATEMENT

DON'T STORE THE CONTROL Z
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0007H</td>
<td>MOV R0,(R4)+</td>
<td>Storing statement</td>
</tr>
<tr>
<td>0008H</td>
<td>MOV R0,(R4)+</td>
<td>Storing statement</td>
</tr>
<tr>
<td>0009H</td>
<td>MOV R0,(R4)+</td>
<td>Storing statement</td>
</tr>
<tr>
<td>000A0H</td>
<td>BR COL010</td>
<td>Branch to COL010</td>
</tr>
<tr>
<td>000A1H</td>
<td>EMT VC0014-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000A2H</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000A3H</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000A4H</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000A5H</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000A6H</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000A7H</td>
<td>COL7 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000A8H</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000A9H</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000AAH</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000ABH</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000AC9</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000ADH</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000AEH</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000AF9</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000B0H</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000B1H</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000B2H</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000B3H</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000B4H</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000B5H</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000B6H</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000B7H</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000B8H</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000B9H</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000BAH</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000BBH</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000BC9</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000BCD</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000BD9</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000BE9</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000BF9</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000C09</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000C19</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000C29</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000C39</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000C49</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000C59</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000C69</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000C79</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000C89</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000C99</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000CA9</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000CB9</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000CC9</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000CD9</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000CE9</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000CF9</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000D09</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000D19</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000D29</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000D39</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000D49</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000D59</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000D69</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000D79</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000D89</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
<tr>
<td>000D99</td>
<td>EMT VC0024-EMTTBL</td>
<td>External memory transfer</td>
</tr>
<tr>
<td>000DA9</td>
<td>NOW COMMITED TO STORING STATEMENT</td>
<td></td>
</tr>
<tr>
<td>000DB9</td>
<td>COL04 DSH 0</td>
<td>Column 5 checkout</td>
</tr>
<tr>
<td>000DC9</td>
<td>ACCEPT COL 5 'NNN N' OR 'NNN '</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000DD9</td>
<td>COL06 DSH 0</td>
<td>Column 6 checkout</td>
</tr>
<tr>
<td>000DE9</td>
<td>ACCEPT COL 6 'NNN NN' OR 'NNN N'</td>
<td>Acceptance check</td>
</tr>
<tr>
<td>000DF9</td>
<td>COL07 CMP 'E': 'R0</td>
<td>Comparing column 7</td>
</tr>
<tr>
<td>000E09</td>
<td>CHANGE STATEMENT NUMBER 'NNN NN':</td>
<td>Changing statement number</td>
</tr>
</tbody>
</table>
FERR  MOV =FERSTKfRO  DEC OPTION  MOV E=F'VfRO
FERSUB MOV =FERCHRfRO  EMT VC007-EMTBL  DEC OPTION  MOV RO=FERP
FERR3MOV FERCHR(R2) +  DEC OPTION  DEC OPTION  DEC OPTION  DEC OPTION  DEC OPTION
FERRTM0V =E'VfRO  HALT ROUTINE  TRAP VU027-TRPLBL  ALT2  MOV E'=VfRO
FERR3MOV =E'VfRO  MOV E=F'VfRO  MOV E=F'VfRO  MOV E=F'VfRO  MOV E=F'VfRO
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>040602</td>
<td>012700 4182 1350</td>
<td>F23FIN MOV =E*3,R0</td>
</tr>
<tr>
<td>040606</td>
<td>000044 4104 0024</td>
<td>EMT VC0007-EMTTBL</td>
</tr>
<tr>
<td>040670</td>
<td>000674 4106 01BC</td>
<td>BR F22C</td>
</tr>
<tr>
<td>040672</td>
<td>005337 410A OADF</td>
<td>F22FIN DEC OPTION</td>
</tr>
<tr>
<td>040674</td>
<td>052502 41dC 5542</td>
<td></td>
</tr>
<tr>
<td>040676</td>
<td>013737 41BE 17DF</td>
<td>MOV INPZ1,INPA1.</td>
</tr>
<tr>
<td>040700</td>
<td>055332 41C0 5A0A</td>
<td></td>
</tr>
<tr>
<td>040702</td>
<td>053404 41C2 5704</td>
<td></td>
</tr>
<tr>
<td>040704</td>
<td>104424 41C4 6914</td>
<td>TRAP VB0043-TRPTBL</td>
</tr>
</tbody>
</table>
Supplementary

Historical

Perspective
The XB system is an experimental system designed to demonstrate the practicability of detecting syntax errors by echo-checking individual characters as they are presented via a teletypewriter.

The state of the art of computing, upon which the XB system builds, can be broken down into three main areas of interest, each with their own history. These areas are:

a) The Development of Electronic Computers.

b) The Development of Programming Languages.

c) The Development of On-line Computing.


The first electronic computer was a machine called ENIAC (Electronic Numerical Integrator And Calculator) built by Eckert and Mauchly at the University of Pennsylvania in 1946.

This machine was based on eighteen thousand vacuum tubes and programs to be run had to be installed by engineers who changed the wiring among its various components. The technology used in computers has grown from vacuum tubes via the transistor, invented in 1948, to large-scale integrated circuits. Three generations of computers based on these technologies correspond to the years 1946-1959, 1959-1965 and 1965 to present.

Electronic computers and the programs which drive them have each had a large effect on the evolution of the other. Programs are naturally written to most effectively use the hardware which is available and hardware is built with the needs of the programs
The RC-11, apart from being the electronic computer used to implement the XE system, is typical of the state of the present art of electronic computers. It is one of a family of computers, each having a similar instruction set which enables programs developed on a small, slow member of the family to be run on a larger, faster member. Other examples of families are the IBM 360 series and the CDC Cyber series. The evolution of these families has been brought about by a recognition of the enormous task of rewriting large programs to take advantage of faster cheaper computers, a clearer understanding of instruction sets and the passing of computers from the experimental stage to being a tool.

The internal organization of electronic computers has evolved in the direction of large instruction sets (including floating point instructions), priority interrupt systems, multiple registers and stack techniques (which allow recursive subroutine calls, used heavily by the XE system).

High speed memory systems have developed from electostatic storage systems to solid-state memories with cycle speeds of a few hundred nanoseconds. Early machines were considered adequate with a thousand words of storage while the IBM 360 series has been designed to address four million words of storage.

An excellent historical survey of the development of Electronic Computers has been written by ROSEN-68. See also ALI-72, SPRAGUE-72, MILNES-68 and BELL-71 (Chap. 3 esp.).
2. Development of Programming Languages

At the level where instructions are electronically interpreted in a computer the language used is normally expressed in binary code. In the first computers it was necessary for the programmer to write a program as a set of binary instructions. For example, on the IBM 704 the instruction

011011 000000 000000 000000 000000 110000

would mean: place the contents of storage location 48 in the accumulator.

A first step in relieving the problem of dealing totally in binary was to introduce mnemonic codes for instructions and the instruction above became

CLA 000000 000000 000000 000000 110000

Another development was to condense 3 bits into an octal number

CLA 00 00 00 00 60

or the address field on the right could be expressed in decimal

CLA 00 00 00 00 48

If it was found necessary to later insert a new instruction in a program this could cause references to some instructions to be incorrect. A partial solution was provided by a concept of regional addressing where a program was divided into sections and a modification to one section did not upset the entire program.

The concept of symbolic addresses was developed, so that the sample instruction became

CLA TEMP
where TEMP was a storage location defined elsewhere by the programmer. The translation of the mnemonic and the allocation of storage was made via the machine itself by a program called an assembler. One of the first assemblers was for the IBM 704 and known as SAP (Symbolic Assembly Program).

While such an assembler made life a lot easier for the programmer, he still had to write a sequence such as

CLA C
MPY C
ADD B
STO A

when he really wanted to form A equal to B+C*D. This led to the development of higher-level languages, the most widely used being FORTRAN (FORmula TRANslation). The advantages of higher-level languages (vs assemblers) has been given by SAMMETT-68 (pp 14-17). These are

1. Ease of learning.
2. Ease of coding and understanding.
3. Ease of debugging.
4. Ease of maintenance and documenting.
5. Ease of conversion.
6. Reduced elapsed time for problem solving.

In general, higher level languages allow the programmer to interface more directly with the problem he is trying to solve.
SAMMETT-68 (pp 17-19) also discusses these disadvantages:

1. Time required for compiling.
2. Inefficient object code.
3. Difficulties in debugging without learning machine language.
4. Inability of the language to express all needed operations.

Despite these objections, the use of higher-level languages is usually preferred where the problem may be expressed in a higher-level language. Much work has been put into reducing the disadvantages listed above.

2.1 FORTRAN

A landmark quoted by SAMMETT-68 (p 143) is a document marked "PRELIMINARY REPORT, Specifications for the IBM Mathematical FORMula TRANslating system, FORTRAN" dated 10th November, 1954 issued by the programming Research Group, Applied Science Division of IBM for the 704 computer.

The 704 FORTRAN system was issued early in 1957 and a new FORTRAN compiler FORTRAN II was issued in June 1958 (thus causing the earlier FORTRAN to become FORTRAN I). These developments initiated a flood of FORTRAN compilers (709 & 650 in late 1958; 1620 & 7070 in 1960 and 7030 (Stretch) in 1962) by IBM and other companies. By 1963 virtually all manufacturers had FORTRAN compilers. OSWALD-64 cites the existence of 43 FORTRAN compilers.

Incompatibilities crept in because of this flood of FORTRAN
compilers. Methods of implementation differed not only between manufacturers but even within the same manufacturer for different machines. In an effort to standardise and also introduce new features in a controlled fashion, the SHARE FORTRAN Committee in March, 1961 went on record as favouring a new, improved FORTRAN language. In May, 1962 a Committee working for the American Standards Association was formed and a publication "American Standard FORTRAN" was approved dated March 7th, 1966. At the same time a proper subset known as Basic FORTRAN was also defined.

The development of FORTRAN has had a very significant impact on computing. Because it was the first such language and because such enormous effort has been spent on education, compiler development and programs, it is difficult for FORTRAN to take advantage of research into programming languages.

2.2 ALGOL

In the late fifties many groups were working on programming languages. In October 1957 CAMM (German Association for applied Mathematics and Mechanics) and the ACM (the Association for Computing Machinery, the American computer society formed in 1947) joined efforts to produce a language first known as IAL (International Algorithmic Language) and later as ALGOL 58 (Algorithmic Language). Despite the interest of the ACM, ALGOL was (and still is) largely a European language. ALGOL 60 followed from an international meeting held in Paris in January, 1960 and an "Algorithms" section appeared in the Communications of the ACM
In February of that year for the dissemination of basic programming algorithms (written in ALGOL).

SAMMETT-68 (p 192) discusses these contributions to the state of the art made by ALGOL:

1) block structure and defining the scope of variables.
2) a formal language definition.
3) recursive procedures.
4) a significant embedding capability for differing subunits.
5) a general simplicity combined with power for stating computational processes.
6) concepts of separate reference, publication and hardware implementation languages.
7) a requirement for the development of better implementation techniques.
8) the spawning of a significant number of languages as outgrowths (such as NELIAC, MAD and JOVIAL).

However ALGOL was designed as a language for procedures internal to a machine and facilities for input and output of data were simply not defined and were left to individual implementations to define (inevitably, differently).

2.3 COBOL

While these developments were going on in the programming languages meant for technical purposes, a parallel development was happening for a business language.

In June 1959 a committee was formed by six manufacturers
(Lurroughs, IDM, Honeywell, RCA, Remington Rand and Sylvania Electric Products) and two government agencies (Air Material Command, USAF and David Taylor Model Basin, Department of the Navy) and known as CODASYL (Conference on Data Systems Languages) Short-range committee. COBOL (Common Business Oriented Language) specifications were designed by December of that year.

There have been three main contributions to the state of the art by the COBOL language. (See SAMMETT-68 p 375).

The first is the separation of any COBOL program into four main divisions:-

a) An IDENTIFICATION division, which does not contribute to the final running program but its requirements are an aid to documenting the program.

b) An ENVIRONMENT division which collects the hardware descriptions together. This division associates all input and output files with the hardware implementation. It is not intended that this division be independent of a particular machine.

c) A DATA division describes all the data used in the program. This is intended to be machine independent but in practice there is a conflict between describing items in a machine-independent fashion at the loss of efficiency, or choosing efficiency by describing items in a way which is well suited to the particular machine at the cost of machine dependence.

d) A PROCEDURE division where the actions to be taken on the data are described. The same conflict arises between efficiency
and independence for the PROCEDURE division.

The second contribution is the accent on handling large files and the third is the development of a language which is very natural to read (although not necessarily to write) and allows long mnemonic names.

2.4 PL/I

The language now known as PL/I arose from inadequacies of FORTRAN in two main areas; character handling and interactions with modern operating systems. (See SAMMETT-68 p 540).

A committee was formed in September 1963 by SHARE and IBM to look at ways of extending FORTRAN. As this work progressed, it was felt by this Advanced Language Committee that trying to preserve compatibility with FORTRAN would prevent very worthwhile concepts from ALGOL and COBOL being included. The FORTRAN compatibility was abandoned and after a series of reports and a change of name (NPL for New Programming Language met with objection from the National Physical Laboratory) an official manual for PL/I was issued in early 1965 by IBM and in August 1966 the first compiler for the system 360 was released.

By building on the experience gained from FORTRAN, ALGOL and COBOL and also by abandoning considerations of compatibility with an existing language, PL/I may eventually replace these languages with a single language, suitable over a wide range of applications.

See CURRIE-66 for a comparison between PL/I and ALGOL, COBOL
and FORTRAN.


All the programming languages described so far were primarily designed to specify a computing procedure in its entirety. A file of input (program and data) was presented to the machine, usually via a card reader, and a file of output was returned, usually via a printer.

In the early sixties programmers began to feel that a typewriter directly connected to a computer and, also, a file system could be a very effective substitute for a card punch and card reader.

LAWRENCE-72 (p 590) summarises some of the advantages of this approach. These are

1) Direct program entry which eliminates the keypunching cycle.

2) Syntax checking on a line-by-line basis.

3) Line by line execution.

4) An editing capability.

5) Data set handling.

6) Direct input/output.

7) Use of an error-detecting fast-compiling system for debugging and later use of an optimizing fast-executing system.

8) Immediate execution.

9) Stop, modify and continue capability.

10) Object deck storage.

See also MEADOW-70 pp 281-293.
3.1 QUICKTRAN

Work on QUICKTRAN was started in 1961 in IBM. The objective was to use standard computing equipment and to write a system which would be largely compatible with an existing language but with powerful debugging and terminal control facilities added. The equipment used were IBM 7040/44 computers and 1050 terminals and the language Basic FORTRAN. A first version of this system was running in mid 1963.

There were two modes; a COMMAND mode (where statements were executed immediately) and a PROGRAM mode where statements were stored to build a program. A number of program control statements were introduced to assist in debugging.

SAMMETT-68 (p 229) summarises QUICKTRAN like this: "QUICKTRAN was significant from several viewpoints. It was the first on-line system using standard equipment; it retained compatibility with an existing language and thus made possible for the user to debug a program on-line and then to use a regular FORTRAN compiler for batch production runs." See also MARTIN-73 pp 55-58.

3.2 JOSS

JOSS (Johnniac Open Shop System) is a system developed at the RAND Corporation in 1963-64 to run on the JOHNNIAC computer. A goal of JOSS (SAMMETT-68 p 218) was to demonstrate "the value of on-line access to a computer via an appropriate language." "It was designed to give the individual scientist or engineer an
easy, direct way of solving his small numerical problems without a large investment in learning to use an operating system, a compiler and debugging tools or in explaining his problems to a professional computer programmer and in checking the latter's results."

JOSS uses a specially-designed typewriter (BAKER-67) with upper- and lower-case letters, the 10 digits and a set of special characters including (for example) a single character for less than or equal to (a combination of < and =).

There are two modes of entry of statements; a DIRECT mode where statements are executed and an INDIRECT mode where statements are stored to be run as a program. The INDIRECT statements have a line label which consists of numbers which can have a decimal point in them to allow for insertion of lines. A step refers to a single line, e.g. 3.2, while a part refers to all lines with the designated number at the left of the decimal point.

Variables consist of a single upper- or lower-case letter which can have two subscripts separated by commas. The value of subscripts must be between 0 and 99.

Only arithmetic variables are permitted, but they can be either integer or floating point. JOSS stores all numbers internally as floating point.

Operations are +, -, \cdot (for multiply), /, and * (for exponentiation). Conditions can be quite complex as in

i: a < b < c and c = e
The assignment statement is of the form
\[ \text{Set var} = \text{expression} \]
The transfer of control is to a step as in
\[ \text{To step 1.35} \]
or to a part as in
\[ \text{To part 4.} \]
Do statements may refer to a single step or to a part; e.g.
\[ \text{Do step n for } x = a(b)c \]
or
\[ \text{Do part n for } x = a(b)c \]
where \( a \) and \( c \) are initial and terminal values and \( b \) the increment. A list is also allowed as in
\[ \text{Do step 3.1 for } x = 1, 2, 3, 100 \]
A full set of functions are available. Both SAMMETT-66 pp 217-226 and MEADOW-70 pp 293-323 have summaries of JOSS.

3.3 BASIC

BASIC (Beginners All-purpose Symbolic Instruction Code) is a system developed at Dartmouth College in 1965 by Kemeny and Kurtz for the GE 225. It was intended for students as a first language and a stepping stone towards FORTRAN or ALGOL.

As in QUICKTRAN and JOSS there are two modes of statement entry: an immediate and a program-storage mode. A statement number (between 1 and 99999) acts to indicate the sequence of the stored program and as a target for both GOTO and GOSUB statements.

Variables in BASIC are a single letter possibly followed by
a single digit. Single letter variables can have one or two subscripts.

Only arithmetic data is available, but more advanced BASIC systems have features such as string variables and matrix manipulation. (See DATA GENERAL CORPORATION-74, Hewlett-Packard-68).

Assignment statements are of the form

\[ \text{LET var = expression} \]

The transfer of control statement is of the form

\[ \text{GCTO 123} \]

A subroutine call is of the form

\[ \text{GCSUB 345} \]

with its companion

\[ \text{RETURN} \]

A simple IF statement of the form

\[ \text{IF A<B THEN 23} \]

transfers control if the condition is met.

The loop control is of the form

\[ \text{FOR I=1 TO 9} \]

and is ended by its companion

\[ \text{NEXT I} \]

SAMMETT-68 pp 229-232 discusses the original BASIC system.

3.4 AMTRAN

AMTRAN (Automatic Mathematical TRANslation) is a system developed at NASA in Huntsville, Alabama. The basic objective of the work was to provide an on-line system to facilitate the
solution of mathematical problems by non-professional programmers.

The over-all system was inspired by the work of Culler and Fried and was influenced by JOSS. (See SAMMET-68 pp 258-264).

The main aim (REINFELDS-72 p 293) of AMTRAN is to exploit the general n-dimensional rectangular array as a basic data structure using operators derived from concepts familiar to a numerical analyst. The language is designed to eliminate as much explicit array component loop writing as possible.

In Fortran, Algol or PL/I, the basic data item is a single entity: number or character or whatever. In AMTRAN the basic data item is an array of entities of the same type.

The following examples are taken from REINFELDS-73 pp 9-14. AMTRAN operands are either numbers or strings. For example

XXX = 5

W = 'A STRING'

Automatic arrays form the backbone of the AMTRAN system. Memory is allocated dynamically and arrays may be created or changed in dimension without prior declarations. Arrays may be created by concatenation,

X=13285, TYPE X

1
2
5

or by an ARRAY operator
Subscripting is performed by a SUB operator. For example

\[ W = 'A STRUNG', W \text{ SUB } 5 = 'I', \text{ TYPE } W \]

A STRING

A volume of the ACM SIGPLAN Notices was devoted to AMTRAN (REINFELDS-71).

3.5 APL

ROSEN-72 p 593 describes the history of APL. "The APL language was developed over many years by Kenneth Iverson at the IBM research Laboratories - a development which culminated in the publication of his book in 1962. (IVERSON-62). The language and the book achieved considerable notice, mostly in academic circles. It seemed useful as a publication language for describing hardware and software computing algorithms but, at least originally, was not considered important as a practical programming language. In 1966 Iverson and his colleagues at IBM designed and installed an elegant time-sharing system on an IBM 360/50 based on the use of APL as a programming language. APL differs from most so-called higher-level languages in that it attempts to emulate and exploit the conciseness and elegance of
mathematical notation in the expression of algorithms. It provides many operators on scalars and vectors and matrices, and some of the operators do complicated things. The resulting conciseness is attractive to users of slow terminals, since powerful algorithms can be expressed in a few lines. Some people find this unattractive, claiming that the language is cryptic and confusing, but some, especially sophisticated programmers, love it."

MARTIN-73 p 58 also describes this dichotomy. "The computer world became divided on APL into enthusiasts and detractors. Few persons familiar with the topic remained neutral." "Often the enthusiasm will rival that of a religious convert."


3.6 The development of the syntax-checking systems.

The three syntax-checking systems which are described in the thesis all indicate syntax errors to the user at the time an invalid character is entered. This is in contrast to compilers which were written to receive input from a card reader. These compilers were usually multipass systems and depended on the entire program being available. With on-line computing, compilers evolved which gave error messages on a line-by-line basis (See LAWRENCE-72 p 590).

The syntax checking systems exploit the full duplex behaviour of a teletype so that syntax errors can be detected on a character-by-character basis. This approach means that the user
can correct his error immediately and messages relating to syntax errors are not needed.

The first of these systems (ACTIV-8) was written in early 1968 for a D.E.C. PDP-8 computer and was a first attempt to exploit the full-duplex behaviour of the teletype. The syntax of the language implemented was kept very simple and the problem of recovery to an earlier point in the same line was avoided. ACTIV-8 had only a program entry mode and the only type of data permitted was real.

Variables were a letter followed by a number. This very simple form made a decision between a command and an assignment statement possible at the second character.

Transfer of control was a GO TO such as

GO TO 21

and conditional transfer of control could be made by an arithmetic IF such as

IF(A3-100) 18,21,28

Input to ACTIV-8 during execution allowed expressions including references to defined variables. For example, an input value equal to A3 plus 10% was A3*1.1 and 14 inches could be expressed in centimetres as 2.54*14.

The second system (ACL) was written between 1969 and 1971 with Dr. P.L. Sanger for a D.G.C. NOVA computer. This system implemented a more complex language and allowed an immediate execution mode as well as a program entry mode. Like ACTIV-8, the only type of data permitted was real; although ACL did allow IF
statements similar to the Fortran logical IF.

The third system (XB) was written in 1972 and 1973 for a D.E.C. PDP-11 computer. The language implemented is much more complex than the ACTIV-8 or ACL systems. There are three data types allowed (real, boolean and string) and these may all be combined in one expression such as

\[
\text{IF } B2 \text{ AND } R1 \text{ GT } 2 \text{ H3='12'&H2}
\]

Assignments in XB may occur at several levels.

\[A1=3*A2=1+A3=1\]

is evaluated from right to left (at assignment level). A3 is given a value of 1; A2 a value of 2 and A1 a value of 6.

Transfer of control is via a GO TO statement like

\[\text{GO TO } 12\]

or

\[\text{GO TO } I\]

where I has previously been defined or

\[\text{GO TO } X(I)\]

which gives a computed GO TO capability.

Conditionals are performed by a logical IF statement of the form

\[\text{IF logical THEN statement ELSE statement}\]

or

\[\text{IF logical THEN statement}\]

or the THEN may be elided to

\[\text{IF logical statement}\]

such as
3 S=S+X(I); IF I=I+1 LE 8 GO TO 3
There is a FOR statement such as
FOR X=150,200 A(X)=0
There is a powerful EDIT statement which allows source
language previously entered to be changed.
All these statements and commands are entered under the
watchdoc checking of the syntax. In the XB system it is not
possible to make any syntax errors; on the other hand, any
expression which is correct in syntax is permitted.
It is necessary to echo correct characters quickly enough to
retain the confidence of the user. In order to do this a method
similar to CONWAY-63 was evolved. Timing tests on the XB system
show a complete recovery time of about 3 ms for a line of 72
characters; a per character acceptance time of about 125
microseconds using a PDP 11/50.
A formal grammar approach to language analysis is presented
by HIGMAN-73, GRIES-71 and FELDMAN-68. The techniques presented
in the thesis allow the syntax checking of a language more
general than LR(1) (FELDMAN-68 pp 89-90) as shown in the example
on p 15 of the thesis. It is necessary only to determine whether
an incoming character is 'acceptable. The language string entered
may be ambiguous (as in the example on p 15 of the thesis) but
this is not important - echo checking can still proceed.
The Text Editor for the Nova (DATA GENERAL CORPORATION-73 p
1-1) also exploits the full duplex behaviour of a teletype with
its rubout character. If the user types
and he really wanted

he can delete three characters by entering three rubouts. These rubouts echo the character being deleted, so the line will look like

and the user can continue thus

and the information entered into the system will be

On the DEC SYSTEM 10, the system maintains the security of user passwords by not echoing the information back to the teletype. (DIGITAL EQUIPMENT CORPORATION - 1972, p 2-129)
4. References.
HEWLETT-PACKARD, BASIC LANGUAGE, 1968.
SPRAGUE, R. E., "A Western View of Computer History",

Demon - An Automatic System For Solving Ordinary Differential Equations

By N. W. BENNETT,
Applied Mathematics and Computing Section,
A.A.E.C. Research Establishment, Lucas Heights,
Private Mail Bag, Sutherland, N.S.W.

SUMMARY

Demon is a system for rapid solution of problems involving ordinary differential equations. It accepts differential equations in the D-notation and generates a FORTRAN program to solve the problem. A means is provided for solving boundary equation problems. New integration methods may be defined to DEMON together with the problem. These methods may range from Predictor-Corrector to Gaussian quadrature. Non self-starting methods are automatically started. The DEMON program has been written in COBOL and should be suitable for any large machine. Using the A.A.E.C.'s 32K IBM-7040 at Lucas Heights, N.S.W. the time taken from completely defining the problem to receipt of the results may be as low as three minutes.

1. Introduction

This paper describes a system called DEMON which has been written for a digital computer so that a person can solve problems involving ordinary differential equations much faster than by conventional methods.

The system allows the problem to be described in a language very similar to FORTRAN, and then produces a complete FORTRAN program to solve the given problem.

There are two reasons for the speed of the system. Firstly, problems can be specified in a language which has been orientated towards describing differential equations. The specification is usually very short since it only has to describe the problem and not specify how to solve it. Secondly, speed is gained because the problem is handled entirely by the computer from this stage.

DEMON is rather like a library information retrieval system and depends on a large skeleton program. Parts of this are selected and modified to fit the specified problem and are then moulded into a complete FORTRAN program.

2. Typical Problems and Essential Notation

The following specification is an example of a problem which may be posed to the DEMON system.

\[
\begin{align*}
T &= 0.0 \quad 1 \\
Y &= 0.0 \quad 2 \\
STEP &= 0.01 \quad 3 \\
ENTER DEMON &= 4 \\
\text{FUNCTION} &= 5 \\
\text{DY} &= 1.0 + \text{Y}^{**2} \quad 6 \\
\text{END DEMON} &= 7 \\
\text{CALL EXIT} &= 8 \\
\text{END} &= 9 \\
\end{align*}
\]

The problem described is "Integrate the differential equation (9)\)

\[
\frac{dy}{dt} = 1 + y^2,
\]

with initial values \(t = 0.0\) (1) and \(y = 0.0\) (2) using a step size of 0.01 (3). When the point \(t = 1\) (7) is reached, pass control back to the monitor (9)."

The problem has thus been described in 10 statements, or which four are directions to the system.

The eventual solution given to this problem is the numerical values of \(T\) and \(Y\) which are printed identifying format. For example, the first line of the output would be:

\[
T = 0.0 \quad Y = 0.0
\]

The total time taken to solve this problem from the specification was 204 seconds on the IBM-7040 at Lucas Heights. This was composed of:

- generation 64 seconds
- compilation and loading 66 seconds
- execution 4 seconds.

The discrepancy in the total is due to tape winding, such as tape rewinding.

The language for describing differential equations, relies heavily on two devices of notation.

The first is the D-notation which is well accepted and may be punched on a card for input to the machine. The following example illustrates the use of the D-notation in DEMON:

\[
a = 1 + y \times z^2 \quad \Delta = 1. + Y' \times DZ
\]

\[
y' = z + \sin (\pi) \quad DZ = -Y + DZ
\]

Conventional FORTRAN may be mixed with the differential equations. Suppose a differential equation changes when \(T\) is equal to \(A\). This may be described by:

\[
1/1/1
\]

\[
1 \quad \text{IF} (T = A) = 1, 2, 2
\]

\[
1 \quad \text{DY} = \text{F} (T, Y)
\]

\[
1 \quad \text{GO TO 3}
\]

\[
2 \quad \text{DY} = \text{G} (T, Y)
\]

\[
3 \quad \text{CONTINUE}
\]

Any number of differential equations of differing order will be accepted by the DEMON system.

The second device of notation is used to refer to points along the integration path. The statement

\[
\text{(T = 1)}
\]

was used in the sample problem to indicate the last point of the integration. More complex points may be described; for example

\[
X**X + Y**Y = 25.
\]

indicates the point on the integration path which is five units from the origin.

Points that occur more than once may also be described. The notation:

\[
Y = 0.0, 3
\]

describes the point where \(y\) is zero for the third time.

The notation for describing points may also be extended for describing boundary equations. The following problem illustrates this.

\[
T = 0.0 \quad 1
\]

\[
Y = 0.0 \quad 2
\]

\[
A = 1. \quad 3
\]

\[
\text{STEP} = 0.01 \quad 4
\]

\[
\text{ENTER DEMON} \quad 5
\]

\[
\text{ALTER} (A) \quad 6
\]

\[
Y (T = 1) = 1.5, .001 \quad 7
\]

\[
\text{FUNCTION} \quad 8
\]

\[
\text{DY} = 1. + A \times Y**2 \quad 9
\]

\[
\text{END DEMON} \quad 10
\]

\[
\text{CALL EXIT} \quad 11
\]

\[
\text{END} \quad 12
\]

This problem is very similar to the first example. The differential equation (9) has a slightly different form. A boundary equation states that \(A\) should be altered (6) so that \(y\) will be 1.5 when \(t\) is equal to one. An initial estimate of \(A\) has been given (3).

A variation of Newton's method is used in the generated program to find the required value of \(A\).

If the first integration does not satisfy the boundary equation to within a certain tolerance (as shown on statement 7), another integration is made with a slightly changed value of \(A\). From these two integrations, an extrapolation may be made to a better approximation for \(A\). This cycle is then repeated until the boundary condition is satisfied.

The total time taken to run this problem on the IBM-7040 at Lucas Heights was 292 seconds.

DEMON has been designed to solve any number of boundary equations, but no guarantee can be given that the process will converge for every problem.
3. Systems Language

The DEMON generator, which produces the FORTRAN program, has been written in COBOL and consists of six thousand statements.

There are several reasons for choosing the COBOL language. The most important was its ability to handle characters and strings of characters. The following statements illustrate this:

IF CN (CNPT) = ' ' ADD 1 TO BC.
MOVE 'GO TO 4108' TO CARD-AREA.

With such a large program, it was also important to have it written in a language which is independent of the machine.

FORTRAN was chosen for the generated program because it is the major scientific computing language used at Lucas Heights. The generated program is built up from only the most basic FORTRAN, so that the entire DEMON system is machine independent.

4. Implementation of DEMON

Since the program generator is much slower than the FORTRAN compiler, some interesting compromises had to be made. The generated program may contain the following statements:

GO TO 4108
4108 CONTINUE.

It turns out that it is much faster in total time to leave these redundant statements in the program rather than attempt to remove them.

These statements were created because the generated program must be correct for any combination of its basic building blocks. In a different combination there may be a block of statements between the two shown and statement 4108 may be the start of another block. However, since statement 4108 has been introduced, a redundant statement has to be used.

DEMON has been in use on the 32K IBM-7040 at Lucas Heights since April 1965. Problems using DEMON are run in one job by generating the FORTRAN program on to a scratch tape and then compiling and executing the program from this tape.

5. Other Applications of DEMON

Three more features of DEMON have proved to be very useful.

The first is that new integration methods may be defined to DEMON with the problem. The way in which \( y = f(y) \) should be integrated is described; from this the method may be applied to the particular problem.

Both self-starting and non-self-starting methods may be defined. The non-self-starting methods are automatically started by DEMON. The methods may range from simple Runge Kutta to high order Predictor-corrector methods which change their step size. An outline of the definition of integration methods is given in the Appendix.

Sometimes sets of differential equations have a common structure. These may be described to DEMON by using dimensioned differential equations in a DO-loop. For example:

\[
\text{DO } 1 \ K = 2 \text{, } 99
1 \ \text{DX}(k) = C^*X(k+1) - 2*X(k) + X(k-1))/DELF**2
\]

may be used to describe the flow of heat in a bar of metal.

The equations of objects under mutual gravitational fields may also be cast in this form.

DEMON may also be used to solve problems not really involving differential equations. The best example of this is in non-linear equations.

Consider the equations:

\[
\begin{align*}
\alpha x^2 & = .25 \\
(\alpha - 1)x^2 + y^2 & = 1.
\end{align*}
\]

The problem of solving these equations may be described to DEMON as follows:

\[
\begin{align*}
T &= 0.0 \\
X &= 1.0 \\
Y &= 1.0 \\
\text{STEP} &= 10.
\end{align*}
\]

The problem of solving these equations may be described to DEMON as follows:

\[
\begin{align*}
T &= 0.0 \\
X &= 1.0 \\
Y &= 1.0 \\
\text{STEP} &= 10.
\end{align*}
\]

The problem of solving these equations may be described to DEMON as follows:

\[
\begin{align*}
T &= 0.0 \\
X &= 1.0 \\
Y &= 1.0 \\
\text{STEP} &= 10.
\end{align*}
\]

The problem of solving these equations may be described to DEMON as follows:

\[
\begin{align*}
T &= 0.0 \\
X &= 1.0 \\
Y &= 1.0 \\
\text{STEP} &= 10.
\end{align*}
\]

6. Conclusion

Programs produced by DEMON are usually slightly slower and larger than an equivalent hand-written program. However, since the DEMON produced program is obtained very simply and invariably compiles correctly, results may be produced very much faster by using DEMON. This has been confirmed by experience in using DEMON at Lucas Heights.

7. Reference


APPENDIX

New integration methods may be described to DEMON. A simplified explanation of the process is given here. The method described treats vectors directly. In practice DO-statements and subscripted variables would be used.

The second order Runge Kutta method illustrates the basic features of the technique:

\[
\begin{align*}
\text{DY}2 &= \text{FUNCTION} (Y2) \\
\text{LOOP} \\
\text{ROTATE} (Y1, Y2, DY1, DY2) \\
\text{DP} &= \text{FUNCTION} (P) \\
Y2 &= Y1 + 5*\text{STEP}*(DY1 + DP) \\
\text{DY}2 &= \text{FUNCTION} (Y2) \\
\text{END}
\end{align*}
\]

This description shows how \( Y2 \) and \( DY2 \) may be calculated from \( Y1 \) and \( DY1 \).

In the beginning the initial values are stored in \( Y2 \). Statement 1 indicates that the derivative of \( Y2 \) should be evaluated and placed in \( DY2 \). Statements 5 and 7 have a similar indication. The \( \text{LOOP} \) statement (2) indicates the entry point for a new integration step. Statement 3 places the information \( Y2 \) and \( DY2 \) into \( Y1 \) and \( DY1 \) so that a new integration step can be made. This second order Runge Kutta is the standard method used in DEMON.

A non-self-starting method may also be used. The method illustrates this:

\[
\begin{align*}
\text{DY}2 &= \text{FUNCTION} (Y2) \\
\text{EXAMINE} \\
\text{ROTATE} (Y2, Y1, DY2, DY1, DY2) \\
\text{CALCULATE} (Y1, DY1, Y2, DY2) \\
\text{LOOP} \\
\text{ROTATE} (Y2, Y1, DY2, DY1, DY2) \\
Y2 &= 5*Y2 - 4*Y1 + 2*\text{STEP}*(DY2 + 2*DY1) \\
\text{DY}2 &= \text{FUNCTION} (Y2) \\
\text{END}
\end{align*}
\]

The \( \text{EXAMINE} \) statement (2) indicates that the values in \( Y2 \) and \( DY2 \) are complete and may be used for output and other purposes. Statement 4 is translated as a call to the self-starting method.

It is also possible to describe Gaussian quadrature methods and methods that change their step size.
ABACUS—A Fast Fortran System For The IBM/360

By N. W. Bennett*

ABACUS is a fast FORTRAN system for the IBM 360 computer. The U.S.A.S.I. standard FORTRAN has been implemented without any extensions, to provide a reference point for teaching and programme interchange. A large proportion of the system is devoted to the detection of errors and the production of diagnostics. A simple monitor ensures rapid transition from one job to the next.

Introduction

The IBM 360/50 computer system is supported by a large operating system, which includes many language compilers. With such an operating system there is an inevitable overhead in recognising a job being presented, invoking the appropriate compiler and running the compiled program. For long-running jobs this overhead is insignificant, but for small jobs overhead can be the major part of the job time. Compilation of a small program may also be a large part of the job time.

For these reasons it is desirable to have an additional compiler which can compile a small program as quickly as possible, execute it and proceed to the next job with a minimum of delay. This report describes such a compiler called ABACUS.

The easiest way to keep overhead down is to restrict a stream of jobs to the one language and to keep the compiler resident in the machine from one job to the next. With this type of compiler a single pass of the source statements is made, and the compiled program is generated into a reserved area of core storage where it can be executed. Programs in compiled form should be neither produced nor accepted because compilation would be slowed down to produce an object deck and input with an object deck would be only marginally faster than with the original source deck.

An alternative of generating a program onto disc and then reading it back into the machine was rejected because of the overhead involved and the possibility of a future expansion of the core storage.

Naturally, if a compiler is written to be as fast as possible the compiled program cannot be as efficient as that produced by a compiler without this bias. Also if the compiler is resident during execution the space available to the object program is less. Many systems written along these lines are now available, but when the ABACUS project was started in August 1966, there was no fast FORTRAN compiler available for the 360 computer.

Choice of FORTRAN

FORTRAN is the language most used by nuclear research establishments throughout the world, for both domestic use and for program interchange. It is used at the A.A.E.C.'s Research Establishment, Lucas Heights and it was obvious that any fast processor for small jobs should use the FORTRAN language.

The United States of America Standards Institute (1966) has approved a standard for FORTRAN which has been set so that a FORTRAN program may be written for one machine and then run on another, if desired. Many computer companies have implemented this standard FORTRAN for their particular machine, but in doing this, a lot of extensions have been added to the standard. Some of these extensions have filled a genuine need in the language, but others have introduced discrepancies which the standard was intended to remove. One of the worst of these is an extension made by two companies which looks the same, but has different meanings. Honeywell Inc. (1966) interpret $A^{*}*B^{*}*C$ as $(A^{*}B)^{*}C$, whereas IBM (1966) regard it as $A^{*}(B^{*}C)$.

It was decided to write a fast FORTRAN compiler for the 360 and to implement the complete standard FORTRAN as large a subset of it as possible. Because of space limitations it was not possible to implement the COMPLEX data type or the EXTERNAL statement. Also the EXTERNAL statement creates complications which the author considered outweighed its usefulness. The PAUSE statement was not implemented because it conflicts with the concept of fast job processing.

Two popular extensions to the standard were implemented, but these may be deleted by use of a compile option as suggested by the American Nuclear Society (Communications of the A.C.M. 1966). These extensions exist in the IBM FORTRAN system, so their inclusion does not preclude testing a job with ABACUS, and then running it for much longer periods using the IBM system. The extensions are

1. The 'END:' option in the READ statement which gives the address of an end of file routine.
2. The use of quotes to delimit Hollerith fields within FORMAT statements.

Description of the 360 at Lucas Heights

The IBM 360/50 at Lucas Heights has 256K bytes

---


of core storage. These bytes are eight bit characters. A single word (or one Storage unit as defined by the standard) consists of 4 bytes. A card reader, card punch and printer are attached on a multiplexor channel, with 4 discs and 2 tapes connected to two selector channels. The ABACUS system resides on a disc, but once it has been read into core storage, the discs and tapes are ignored.

The ABACUS system is organized along the lines already outlined. It consists of over 150 assembler subroutines which take up 162K bytes. The symbol table uses 12K bytes, the compiled program may use 16K bytes for variables and arrays, 8K bytes are reserved for storage of constants and addresses, and the compiled program may be up to 24K bytes long, and the rest of the machine (34K) is used by the IBM operating system.

The desirability of having a single pass over the source language forces a requirement on the order in which specification statements may be presented. This required order is:

1) DIMENSION or type statements
2) COMMON statements
3) EQUIVALENCE statements.

The DIMENSION and type statements must precede COMMON statements so that addresses in COMMON can be assigned while the COMMON statement is being scanned. Similarly, the DIMENSION and COMMON statements must precede the EQUIVALENCE statements so that a complete description of an array or variable is available for processing the equivalence relationships.

Addressing on the 360

The 360 system has been designed for a maximum core storage of 16M bytes. To reference a position in this maximum size store requires a 24 bit address and the designers considered a 24 bit address in every memory reference instruction to be wasteful of storage space, particularly when an actual store of 256K requires only an 18 bit address. Instead, memory reference instructions contain a reference to one of 16 general purpose registers and also a positive displacement of 12 bits. When referenced in this way, the general purpose register is called a base register. The address to memory is obtained by adding the displacement to the contents of the base register. With this addressing scheme an origin in a base register can only be used to address an expand of 4096 bytes. To address a location outside this space, the origin in the base register must be changed, or another base register containing a suitable origin must be used.

This addressing structure creates problems in the organization of the compiled program. Firstly, reference must be made to different points in the compiled program; to statement numbers, subroutine entries, etc. These references are made only occasionally and in ABACUS an address is loaded into a register which is used as a base. This base is then used by an instruction with a zero displacement. This approach is also needed because of the single pass structure of the system.

Reference to variables is a more serious problem. Execution would be very slow if a double reference were needed for each variable. Because of the small amount (4K words) of storage available to the compiled program, a much faster and more pleasing approach may be taken.

The base register designation in the instruction is to the left of the displacement. Therefore if an increment was made to the displacement in an instruction, any overflow would cause the numerically next higher register to be indicated. Now this overflow does not matter if the next register contains an address 4096 higher than the first. In effect, by this arrangement the displacement within the instruction can be extended. Six of the 16 registers are used in ABACUS to reference the 4K words of data and 2K words of pointers to statement numbers, subroutine entries, etc.

When extra core storage becomes available for the 360/50 at Lucas Heights, the complete standard FORTRAN will be implemented. The addressing scheme described above is valuable mainly for variables. Currently the 4K words of data contain both variables and arrays. The arrays could be taken out of this region with a small loss in performance due to double referencing these array elements which have absolute subscripts. Array elements whose subscripts have to be calculated are double referenced either way. For variables in COMMON and variables in equivalence with array elements, a double reference addressing scheme would be needed.

The Blank Character in FORTRAN

In most statements in FORTRAN (including standard FORTRAN) the blank character has no meaning and may be used freely to improve the appearance of the program. The author feels that blanks imbedded in words, or words without a blank separating them, do not improve the appearance of a program. A compiler can be more easily written to recognize individual statements in the FORTRAN language if the restrictions suggested above are made. As an example, consider the two statements:

```
DO 2 J = 1,3
DO 2 J = 1.3
```

For compilers which are tolerant to blanks the first statement is a DO statement and the second is an arithmetic statement. However, considerable analysis of one statement must be made before it is reclassified as a statement of the other type. For ABACUS the first statement is a valid DO statement and the second is an invalid DO statement. This approach has led to some conflicts. People used to writing

```
GOTO28
```

are not pleased at having the statement rejected, but why should all programs be compiled more slowly because a few users have poor punctuation?

Treatment of Errors—Compilation

One of the aims of ABACUS was to produce good error messages. During compilation an extensive check is made of the statements. For all errors de-
A full message is produced, with a pointer to a card column where needed. The parts of the compiler used to check for errors and all the possible error messages take up a large amount of storage. Further action depends on the type of error. If the error is in an executable statement, instructions are generated so that execution will be terminated if control reaches that point. If the error is in a specification statement, it will affect the entire program unit. If the error is in the mainline program, execution is deleted; if it is in a sub-program, execution will be terminated on entry. For statement numbers missing or subprograms missing, their address link is filled in so that control will pass to an error routine if they are referenced.

Treatment of Errors—Execution

The operating system must consider the invocation of the batch processor as a single job. If the operating system is allowed to follow its usual course when, say, an overflow occurs, the remaining jobs in the batch may be flushed. Because of this unsympathetic behaviour of the operating system and because so few of its features are used, ABACUS has its own small operating system to handle input-output, timing and interrupts.

There are two modes of operation on the 360. One is a supervisor state where all instructions are legal, and the second is a problem state where instructions such as input-output are illegal. A special instruction, supervisor call, is provided for a program in the problem state to request supervisor services. A special supervisor call, SVC 255, has been implemented at Lucas Heights to allow a program in problem state to enter the supervisor state. This feature is used by ABACUS to give control to its own operating system.

Errors during execution can be divided into errors which any program might make, such as an overflow, and those which violate the rules of FORTRAN, such as the subscript for an array element being out of range. The first type of error causes an interrupt and information needed to diagnose the fault is available to the system.

It is much more difficult to detect the second type of error and gather information about it. The program produced by ABACUS contains checks for this type of error and, if one should occur, provides information necessary to diagnose it. One example of this is the array subscripts already mentioned. Another is a check that none of the parameters of a DO statement is changed during its range. This is achieved by making a copy of the parameters at the start of the loop and comparing them at the end of the loop.

The Disassembler

A disassembler is part of the ABACUS system and this is a program which can provide an assembler language listing from basic machine language. The disassembler was used during development of the system to check the compiled program. This feature which takes up little space, was invaluable to the author when checking the generated program. The disassembler is used in the completed program to give an assembler language listing of the compiled program (if the user desires), and also to give a list of any statement which signals an error during execution. Several tables are available to the disassembler in order to make the listing more readable. The first of these is the symbol table, which is retained throughout the job. Addresses found in instructions can be used to find the relevant variable name in the symbol table; this name may then be included in the listing. The symbol table entry also gives the variable type. This means the current value of the variable may be given in the appropriate format. Many instructions are only generated in unique circumstances. A table of these is available to the disassembler and when they occur a relevant comment can be included in the listing. During compilation a table containing the lengths of individual statements is built; this is used later to separate the compiled program into the original statements for listing.

The Epilogue

At the end of execution an epilogue is given which is a summary of the execution giving the time taken, the number of cards read, the number of lines printed and the number of cards punched. The last card read and the last card punched are reproduced in the epilogue. The reasons for termination are given. This may merely be an indication that control reached a STOP statement or it may be much longer. Suppose the error was an overflow in an arithmetic statement. Using the address of the overflow, the disassembler is able to list the complete arithmetic statement in question. As mentioned before, this list will contain the names of the variables in the statement and their values. Every fault is handled in this way. A message gives the nature of the fault and the list of the relevant statement gives added detailed information. If the fault occurs in a subprogram, the path back to the mainline program is followed, giving a list of each call statement, including the value of all arguments.

A compile option is available to give the program's path prior to termination. The compiled program has an instruction inserted before the normal code for each statement. During execution this instruction calls a trace subroutine which records the latest 1000 return addresses and counts the number of calls. This count and these addresses are given in the epilogue. If the program had been in an endless loop, the actual loop could be determined by an analysis of the statement locations. This could also be done by the system, but lack of space prevents it: inclusion in ABACUS. Another option is available to list the value of all variables and arrays after execution. This provides a valuable aid to locating errors.

The Batch Control Card

It is usual for a system to have different options available. Among these may be the option to produce a list of the variables and statement numbers in the program, or to execute not longer than a set time. So that every job need not explicitly state all the options, a standard set of options is normally adopted. If a job does not state a particular option, the relevant
standard state can be taken. This standard set of options is usually built into a system, but with ABACUS a standard state may be set for a batch of jobs. This is done with a control card which precedes the batch and also specifies if an option may be changed by an individual job. There are two classes of options. The first contains qualitative options, such as a list of the variables and statement numbers. The control card states the standard option and also whether it may be changed. If it may not be changed, the user's attempt to do so will be fruitless. The second class contains quantitative options, such as a time limit for execution. As well as the standard length of time, the control card sets a maximum time which the user cannot exceed. Suppose the standard limit is 10 secs. and the maximum limit 60 secs. If the individual job does not set its own limit, 10 secs. will be taken. If the job specifies any time up to 60 secs. this will be allowed, but any attempt to request more than the 60 secs. will be taken as 60 secs. only.

This concept of a maximum limit also applies to all quantitative options such as the number of lines printed and the number of cards punched.

Performance

ABACUS will run each small job in approximately 2 seconds. This compares very favourably with the equivalent IBM time of 61 secs. Naturally, the execution speed of ABACUS is not as good as the longer compiling IBM system; this is also due to the time required for checking errors. For jobs which use many subscripted variables, the comparative speed may be about 35 per cent, but a carefully written program using only simple variables may execute at about 95 per cent. The size of the source program naturally varies with the complexity of the statements. If the statements were all as simple as

\[ A = B + C \]

then ABACUS could handle a program with two thousand statements. The realistic limit would be about 500 to 700 statements. It has been very worthwhile to include the check for execution errors with the production of full error messages. Users of the ABACUS system have commented that errors are well diagnosed and that this has enabled them to correct their programs more quickly.

Conclusions

Possibly the most valuable lesson learnt from writing ABACUS is that detecting errors and producing good diagnostics takes up a lot of space. Apart from the fact that parts of standard FORTRAN are not implemented because of space limitations, the design aims of the project have been realised. The time taken to process small jobs has been greatly reduced from that required by the IBM system. The time required to find and correct errors has been reduced by the provision of execution checking and good messages.

References


