THE ML/I MACRO PROCESSOR

Implementing software using the LOWL language

Supplement 1: ALGEBRA

P.J. Brown
Computing Laboratory
University of Kent at Canterbury
September 1972.

(c) 1972 P.J. Brown

Chapter 1 MD-logic and LOWL extensions

ALGEBRA is a package for students and research workers in Boolean algebra and multi-valued logics. It is described in Bulletin Inst. Maths. Applics. 7, 11 (Nov. 1971) and, in more complete detail, in user manuals for the various implementations (e.g. Brown, P.J. "The ALGEBRA system", Kent On-line System Document KUSE/ALGEBRA). This document assumes that the implementor is familiar with the principles of ALGEBRA as given in a user manual.

I/O requirements

ALGEBRA logically requires three output streams and one input stream. The output streams are the message and results streams, as described in the LOWL manual, and the questions stream, which is used to output all the questions that ALGEBRA asks of its user (e.g. "VALUES="). In a simple conversational implementation all four I/O streams may correspond to the same device, an on-line console. However, provision may be made for the results stream, which is used for output produced by the TABLE and TRY statements, to go to a line-printer or perhaps to backing storage.

In a non-conversational environment the questions stream requires special treatment as it is not possible to communicate directly with the user. One possible approach is to suppress all questions. This is trivial to implement. A second approach is to require the input lines to contain both the question and the answer. Thus an input line, a card say, might read

VALUES = TRUE FALSE

The question part is used as an error check to prevent the input getting out of phase with what ALGEBRA expects. This second approach requires significant extra work for the implementor, which in detail is as follows. Questions should be sent to an intermediate buffer, and, when an input line is read the following action would be taken:

(a) if the questions buffer is empty go to step (e).

(b) check the questions buffer character by character against the input line, ignoring spaces. If there is a mis-match before the end of the questions buffer is reached, output an error message and stop.

(c) clear the questions buffer.

(d) update pointers, etc., so that the question part is not treated as part of the input line.

(e) proceed with input line in the normal way.

Breaks

If ALGEBRA is to run in a conversational environment, it is desirable to cater for "breaks", i.e. cases where the user stops ALGEBRA by pressing some special key. The user may, for example, wish to break ALGEBRA when it is outputting a very large table so that he can make some change. The MI-logic has been designed so that it can accept such breaks at any time without losing any information (i.e. the names of values and the definitions of operators).

If an implementor decides to cater for breaks it is suggested he writes MD-logic code to perform the following action at each break:

```
MESS      '$***BREAK$'
LAV       INFFPT,X
CAL       0
GONE      NEXTLN,...      (label in MI-logic)
GOSUB     MDQUIT,X
```

This means abandon the run if no values have yet been defined, otherwise go to the code in the MI-logic which resumes operation by reading the next line of input.

Extensions to LOWL

ALGEBRA requires two extra statements in addition to the kernel of LOWL. These are the RMESS and QMESS statements, which are identical in form to the MESS statement except that the string is output, respectively, on the results and question streams. Examples are

```
QMESS   'VALUES='
RMESS   'IS  A  CONTINGENCY$'
```

The form of the MD-logic

Immediately on being entered, ALGEBRA should zeroize the variable INFFPT. This is done to tell the routine that caters for breaks, if it exists, that no values have yet been defined. Breaks can be accepted at any time after INFFPT has been cleared. The common initialization code, as given in the LOWL manual, should then be performed. The stack area need not be large - 300 words should be adequate unless use with complicated multi-valued logics is envisaged.

In addition to this, the MD-logic of ALGEBRA requires the MDQUIT subroutine, as described in the LOWL manual, and the subroutines described below.

The MDERCH, MDRCH, MDQCH subroutines

The MDERCH, MDRCH and MDQCH subroutines output the character in C on the message, results and questions stream, respectively. Each has a single exit.

The MDLINE subroutine

MDLINE is a subroutine with two exits. It reads lines of input. If input is exhausted it uses exit 1. Otherwise it reads a line of input into a buffer, sets IBFPT to point at the first character of the buffer and uses exit 2. (IBFPT should be reset on each call of MDLINE, since the MI-logic changes it.) The text in the buffer should be terminated with a newline character (as represented by NLREP). (Thus, for example, a null input line would cause the buffer to contain simply a single newline character.)

MDLINE is used both for answers to questions and for ordinary input. In the case of questions it is desirable for the answer to be on the same line as the question. Hence when a question is output (using QMESS and/or MDQCH) it is not terminated with a newline.

If the implementor wishes to produce a gold-plated version of ALGEBRA, he should cater for switches of input stream. Users may wish to input some predefined definitions from a non-conversational device, for example from backing storage or from a paper tape reader, and then to use these at an on-line console. This has implications on the coding of MDLINE and perhaps also of the initialisation code.

Chapter 2 Testing an implementation of ALGEBRA

This Chapter contains some test data which can be used to validate an implementation of ALGEBRA. This data needs to be supplemented by tests of the machine-dependent features of the implementation. Unfortunately the nature of ALGEBRA does not allow the data to be self-checking in any way, so the implementor needs to go through the output from the test line by line to check that the output is what it should be. (Alternatively a program might be written to perform this comparison, the inputs being what the implementation should produce as output and what it actually does produce. It is very doubtful, however, whether the effort needed to produce this program and punch up the data for it would be worthwhile, unless part of a larger project.)

The test data

The test data consists of three separate tests, covering one-valued, two-valued and three-valued logics. It is reasonable to suppose that if these work then all multi-valued logics will work.

In the descriptions that follow the input lines for each test begin with three dots, the output produced on the messages stream begins with three asterisks and the remaining lines are output produced on the results stream. Where an input line is an answer to a question the question is given at the start of the input line, exactly as it might look on, say, an on-line typewriter. For example the initial input line might be specified

```
...  VALUES = T  F
```

The first test deals with a one-valued logic and is simply a path-finding test to make sure the main features of ALGEBRA are working. The test is as follows.

```
...VALUES=SINGLE
...OP +
...UNARY OR BINARY=BINARY
...PRECEDENCE=15
...SINGLE + SINGLE=SINGLE
...TRY A+B
=SINGLE

...TABLE AA+BB

AA_____      BB_____      :_VALUE
SINGLE       SINGLE       : SINGLE

...OP NOT
...UNARY OR BINARY=UNARY
```

```
...PRECEDENCE=1
...NOT SINGLE=SINGLE
...TABLE + ; ERROR LINE

***ERROR IN USE OF BINARY OPERATOR +
...TRY NOT A+B
= SINGLE

...;END OF TEST
```

The second test, which is a comprehensive one, covers a two-valued logic. The test precedes as follows. Firstly a set of operators is defined, a large number of errors being made, and corrected, during this process. There follows a series of statements, all of which contain errors. The TRY statement is then tested, firstly with a series of expressions all of which should be TRUE, then with a series of FALSE expressions and finally with some contingencies. This is followed by a series of tables, which need to be carefully checked by the tester. The test concludes by redefining an existing operator, and checking the new definition. In detail, the test is as follows.

```
...VALUES=(
***ERROR -- CLASHING USE OF SYMBOL
...VALUES=X+
***ERROR -- ILLEGAL SYMBOL
...VALUES=
***ERROR -- INCOMPLETE LINE
...VALUES=TRUE FALSE
...
...;*********FIRST DEFINE SOME OPERATORS*********
...OP     +; EXCLUSIVE OR
...UNARY OR BINARY=
***EH
...UNARY OR BINARY=UN X
***EH
...UNARY OR BINARY=UXARY
***EH
...UNARY OR BINARY=BINARY
...PRECEDENCE=
***EH
...PRECEDENCE=5X
***EH
...PRECEDENCE=-1
***EH
...PRECEDENCE=1 X
***EH
...PRECEDENCE=1001
```

```
***EH
...PRECEDENCE=1
...TRUE + TRUE=
***EH
...TRUE + TRUE=TRUE X
***EH
...TRUE + TRUE=+
***EH
...TRUE + TRUE= FALSE
...TRUE + FALSE=TRUE
...FALSE + TRUE=TRUE
...FALSE + FALSE=FALSE
...;
...OPERATOR-
...UNARY OR BINARY=UNARY
...PRECEDENCE=999
...- TRUE=FALS
***EH
...- TRUE=FALSE
...- FALSE=
***EH
...- FALSE=+
***EH
...- FALSE=TRUE
...OPERATOR NOT ; LOW PRECEDENCE NEGATION
...UNARY OR BINARY=UNARY
...PRECEDENCE=0
...NOT TRUE=FALSE
...NOT FALSE=TRUE
...OPER  X
...UNARY OR BINARY=BIN
...PRECEDENCE=2
...TRUE X TRUE=TRUE
...TRUE X FALSE=FALSE
...FALSE X TRUE=FALSE
...FALSE X FALSE=FALSE
...OPERATOR >   ; IMPLIES OPERATOR
...UNARY OR BINARY=BIN
...PRECEDENCE=3
...TRUE > TRUE=TRUE
...TRUE > FALSE=FALSE
...FALSE > TRUE=TRUE
...FALSE > FALSE=TRUE
...
...;*********THE FOLLOWING STATEMENTS ALL CONTAIN ERRORS*********
...BBB
***ERROR -- UNRECOGNISED STATEMENT
...XYZ+Q
```

```
***ERROR -- UNRECOGNISED STATEMENT
...OP TRUE
***ERROR -- CLASHING USE OF SYMBOL
...OP
***ERROR -- INCOMPLETE LINE
...OP(
***ERROR -- CLASHING USE OF SYMBOL
...TABLE
***ERROR -- INCOMPLETE LINE
...TABLE A+
***ERROR -- INCOMPLETE LINE
...TABLE TRUE+FALSE
***ERROR -- NO VARIABLES
...TABLE AX B
***ERROR IN USE OF VARIABLE B
...TABLE AX FALSE
***ERROR IN USE OF VALUE FALSE
...TABLE A(B+A)
***ERROR IN USE OF PARENTHESES
...TABLE A+B)
***ERROR IN USE OF PARENTHESES
...TABLE)
***ERROR IN USE OF PARENTHESES
...TABLE A NOR
***ERROR IN USE OF VARIABLE NOR
...TABLE A NOT B
***ERROR IN USE OF UNARY OPERATOR NOT
...TABLE +B
***ERROR IN USE OF BINARY OPERATOR +
...TABLE F+(+B)
***ERROR IN USE OF BINARY OPERATOR +
...TABLE (A
***ERROR IN USE OF PARENTHESES
...TABLE ((NOT A) > (B)
***ERROR IN USE OF PARENTHESES
...TRY
***ERROR -- INCOMPLETE LINE
...TRY ----------------------------------------------------------X
***ERROR -- EXPRESSION TOO COMPLICATED
...
...*********THE FOLLOWING SHOULD ALL BE TRUE*********
...TRY TRUE
= TRUE

...TR NOT FALSE
=TRUE

...TRYFORMEPLEASE NOT-TRUE
= TRUE
```

```
...TRY TRUE X TRUE
= TRUE

...TRY,,TRUE, ,X,NOT,,,FALSE,,;
= TRUE

...TRY NOT--FALSE X TRUE
=TRUE

...TRY FALSE>(TRUE+TRUE)
= TRUE

...TRY(TRUE+FALSE)>TRUE
= TRUE

...TRY A+-A+XX+-XX+C+(-(C) )
= TRUE

...TRY TRUE+A X -A
= TRUE

...TRY NOT FALSE X FALSE
= TRUE

...TRY((((TRUE)))X TRUE)
= TRUE

...
...;*********THE FOLLOWING SHOULD ALL BE FALSE*********
...TRY FALSE>TRUE+TRUE; > IS DONE FIRST
= FALSE

...TRY TRUE+FALSE>TRUE
= FALSE

...TRY A X-A
= FALSE

...   TRY A X-A
= FALSE

...,,,TRY A+A
= FALSE

...TRY -A X A
= FALSE

...
...;*********THE FOLLOWING SHOULD ALL BE CONTINGENCIES*********
...TRY PIG
IS A CONTINGENCY
```

```
...TRY NOT A X A
IS A CONTINGENCY

...TRY A+B+C+D+E+F+G
IS A CONTINGENCY

...
...;*********TEST THE TABLE STATEMENT*********
...TA XYZ
```

XYZ	: VALUE
TRUE	: TRUE
FALSE	: FALSE

```
...TABLE A+-B
```

A	B	: VALUE
TRUE	TRUE	: TRUE
TRUE	FALSE	: FALSE
FALSE	TRUE	: FALSE
FALSE	FALSE	: TRUE

```
...TABLE Z>(LONGNAME+C)X(-TRUE X C)>(D+Z)
```

Z	LONGNA	C	D	: VALUE
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	TRUE	TRUE	FALSE	: FALSE
TRUE	TRUE	FALSE	TRUE	: TRUE
TRUE	TRUE	FALSE	FALSE	: TRUE
TRUE	FALSE	TRUE	TRUE	: TRUE
TRUE	FALSE	TRUE	FALSE	: TRUE
TRUE	FALSE	FALSE	TRUE	: FALSE
TRUE	FALSE	FALSE	FALSE	: FALSE
FALSE	TRUE	TRUE	TRUE	: TRUE
FALSE	TRUE	TRUE	FALSE	: TRUE
FALSE	TRUE	FALSE	TRUE	: TRUE
FALSE	TRUE	FALSE	FALSE	: TRUE
FALSE	FALSE	TRUE	TRUE	: TRUE
FALSE	FALSE	TRUE	FALSE	: TRUE
FALSE	FALSE	FALSE	TRUE	: TRUE
FALSE	FALSE	FALSE	FALSE	: TRUE

```
...
...;*********REDEFINE AN OPERATOR*********
...OP >
...UNARY OR BINARY=BI
...PRECEDENCE=10
...TRUE > TRUE=FALSE
...TRUE > FALSE=TRUE
...FALSE > TRUE=FALSE
...FALSE > FALSE=FALSE
...TRY A>A;    THIS SHOULD NOW BE FALSE
= FALSE

...;END OF TEST
```

The third and last test covers a three-valued logic. The test is short and contains little that is not covered by the previous test, and thus should cause few problems. It is as follows.

```
...VALUES=0,1,2
...OPERATOR -
...UNARY OR BINARY=BINARY
...PRECEDENCE=2
...0 - 0=0
...0 - 1=2
...0 - 2=1
...1 - 0=1
...1 - 1=0
...1 - 2=2
...2 - 0=2
...2 - 1=1
...2 - 2=0
...
...;*********THE FOLLOWING SHOULD ALL BE 0*********
...TRY 0
= 0

...TRY FISH-FISH
=0

...TRY 2-1-1
=0

...TRY 2-(1-1)-2
=0
```

```
...;*********NOW PRINT A TABLE*********
...TABLE (A-B)-(C-A)

A          B          C          : VALUE
------     ------     ------     :------
0          0          0          : 0
0          0          1          : 2
0          0          2          : 1
0          1          0          : 2
0          1          1          : 1
0          1          2          : 0
0          2          0          : 1
0          2          1          : 0
0          2          2          : 2
1          0          0          : 2
1          0          1          : 1
1          0          2          : 0
1          1          0          : 1
1          1          1          : 0
1          1          2          : 2
1          2          0          : 0
1          2          1          : 2
1          2          2          : 1
2          0          0          : 1
2          0          1          : 0
2          0          2          : 2
2          1          0          : 0
2          1          1          : 2
2          1          2          : 1
2          2          0          : 2
2          2          1          : 1
2          2          2          : 0

...;END OF TEST
```

Testing of implementation dependent features

The given test data needs to be supplemented by tests of implementation-dependent features. Areas to be tested may include:

(a) Operating system interface. Entry, exit, sudden termination (e.g. in the middle of an operator definition), a null run, a run with little or no storage for the stack area.

(b) Breaks. Breaks during input, output and in reply to a question. Breaks before values have been defined. Recovery from breaks.

(c) Input/output. I/O options, incorrectly specified I/O devices, overflow conditions (lines too long, etc.), switching of input stream (if available).