R D E

THE ML/I MACRO PROCESSOR

Implementing software using the LOWL language

Supplement 4: SCAN

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

Chapter 1 Extensions to LOWL

Introduction

SCAN is a simple text processing language designed for conversational use. A manual describing SCAN and its implementation under the Kent On-line System (KOS) is available separately. The logic of SCAN has been implemented using the language LOWL, so that it can easily be transferred between machines. Because SCAN performs extensive character manipulations, it has been necessary to add certain extra character operations to LOWL. Most of these should be trivial to implement.

The SCAN compiler compiles into a pseudo machine-code program which is interpreted at run-time. This program consists of a sequence of pseudo-ops. Each pseudo-op has an operation code and some of the pseudo-ops also have operands. The operation codes of pseudo-ops that possess operands are integers in the range 0-14 and the operation codes of pseudo-ops without operands are integers in the range 16-63. Operands are either addresses of SCAN variables or non-negative integers not greater than MAXINT (which is the highest permissible integer for a given SCAN implementation and is defined in the initialization code - see Chapter 2). In the former case the variables are either declared in the LOWL logic or, in the case of C variables and N variables, reserved during initialization.

The pseudo machine-code program is very close to Reverse Polish notation. For the reader who is interested full details are given in Chapter 3. However the implementor does not need to know the details of how pseudo-ops work, but he does need to choose how they are to be represented on his hardware. The following examples show some possible choices.

Example 1: byte machine

Pseudo-ops with operands: one byte
Pseudo-ops without operands: three or four bytes

Example 2: 24-bit word machine

All pseudo-ops: first six bits: operation code
last eighteen bits: operand (unused if none)

Example 3: 16-bit word machine

Pseudo-ops with operands: first four bits: operation code
last twelve bits: operand

Pseudo-ops without operands: first four bits: 1111
next six bits: unused
last six bits: operation code

This example shows a specially concise coding to give as much room as possible for the operand field. Even so, it would introduce limitations as MAXINT would need to be set to 4095 and all variables would need to be in the first 4K of storage (unless a special trick is employed - see later). If these limitations were thought to be unacceptable, pseudo-ops with operands would need to occupy two words.

Several of the extensions to LOWL needed by SCAN are concerned with creating and extracting these pseudo-ops. Pseudo-ops must occupy an integral number of storage units on the object machine, e.g. on an 8-bit byte machine they could not be made to occupy, say, 22 bits as it would be impossible to address them. (They need not, however, be aligned to word boundaries on those machines where this is applicable, as they are always addressed indirectly.)

Extensions to LOWL

The following is a complete list of the extensions to LOWL required for the MI-logic of SCAN.

Named Character constants

SCAN requires unique codes to be defined for the following three pseudo-characters: NULREP, EOSREP and SOSREP. Any codes that do not arise in normal input can be chosen to represent these.

The RCS statement

The RCS statement has the form

RCS name, OF

and is used to reserve core storage. The second argument specifies how many units to reserve. The name should be attached to the start of the area of store that has been reserved. (RCS is used to reserve space for both integer and character variables. All the integer declarations precede the character ones, so that the potential alignment problem that might arise on machines such as IBM System/360 is avoided. A glance at the logic will show this.)

Subsidiary of the OF macro

The two following additional subsidiary macros to the OF macro need to be defined:

LFN — the number of units of storage occupied by a pseudo-op with no operand.

LIN — the number of units of storage occupied by a pseudo-op with an operand.

Large integer constant

All occurrences of numerical constants have values less than 64 with the single exception of the instruction

```
LAL     1000
```

(which is used to set the initial value of L21).

Statements for manipulating the C register

The following extra statements are needed to manipulate the C register. They work in a similar way to the corresponding statements for manipulating the A register.

LCV V, {R / X}	Load C with value of V.
STC V, {P / X}	Store C in V.
STCI V, {P / X}	Store C in address pointed at by V.
CCV V	Compare C with value of V.

With the exception of STCI, V is a character variable that has been previously declared using

```
RCS     V, OF(LCH)
```

The MULT, DIV and DEBUMP statements

The MULT and DIV statements have the forms

MULT V	Multiply A by value of V.
DIV V	Divide A by value of V.

Overflow can be ignored. The MI-logic detects division by zero as an error and does not call DIV in this case.

The DEBUMP statement has the form

DEBUMP V,N-OF	Decrease value of V by literal value N-OF. (This may clobber A.)

The RMESS statement

The RMESS statement is exactly similar to the MESS statement in LOWL except that it uses the results stream rather than the messages stream.

The CTOA and ATOC statements

The CTOA statement copies the value of the C register into the A register and the ATOC statement does the opposite. Neither statement has an argument. If the A and C registers are implemented as physically separate registers it may be that the C register is smaller than the A register. In this case the high order bits of the A register should be ignored. ATOC is used before an assignment to a character variable, e.g.

```
10  LET  C1 =-1
```

The User's Manual for an implementation should make it clear how the result may be truncated in such an assignment.

Statements for manipulating pseudo-ops

The following are the statements for manipulating pseudo-ops.

LAFUN V Load A with the operation code of the pseudo-op pointed at by V.

Hence A should have a value in the range 0 - 63 after the execution of this statement. Note that the pseudo-op may or may not have an operand. If pseudo-ops are encoded in a "clever" way as illustrated by Example 3 earlier, then the LAFUN statement itself will need to be correspondingly clever to find the correct field, shift it, etc.

LARAND V Load A with operand of the pseudo-op pointed at by V.

The pseudo-op will always have an operand if LARAND is used.

INST N Stack on the forwards stack a pseudo-op whose operation code is given by N and whose operand is given by the value of A.

FUNCT N Similar to INST but for pseudo-ops with no operand.

FUNCTV V Stack on the forwards stack a pseudo-op whose operation code is given by the value of V. The pseudo-op has no operand.

Being stacking operations, INST, FUNCT and FUNCTV must update FFPT after having assembled the pseudo-op at the top of the stack. The code to do this is as follows (c.f. the FSTACK statement)

```
LAV     FFPT,X
AAL     0F(LFN) or, for INST, 0F(LIN)
STV     FFPT,P
CAV     LFPT,A
GOGE    ERLSO,.....
```

Two final points should be made about packing pseudo-ops into the minimum possible space. Firstly, if, say, operands are being stored in 12 bits, (thus allowing operands in the range 0 - 4095) and all variables are stored between, say, address 5000 and address 6000, then the INST statement could subtract 5000 from the operands of the pseudo-ops that have variables as operands in order to get them into the range 0 - 4095. (The pseudo-ops with variables as operands have the operation codes 2 to 5 inclusive.) LARAND could perform the reverse mapping for these pseudo-ops.

Secondly, AT elements are joined together on chains in the compiled program. If MAXINT is small (e.g. 4095) and a program huge, it may be that the offset in a chain will exceed MAXINT and not fit into an operand field. The MI-logic tests for this situation and gives the error message "TOO MUCH" if it occurs. If this error occurs frequently and is a serious inconvenience to users, the implementor must change his encoding of pseudo-ops and increase MAXINT. The situation is, however, highly unlikely.

Chapter 2 The MD-logic and documentation

Introduction

The MD-logic of SCAN consists of some initialization code together with a set of subroutines. It has two main functions:

(a) to provide I/O.

(b) to provide a command structure and operating system interface.

The SCAN processor has three possible states:

(C) compiling (either the original program or edits).

(1) running.

(2) listing.

(The variable STATE in the MI-logic is set to 0, 1 or 2 to show the current state.) The MD-logic must provide the mechanism for letting the user specify which state he wishes to enter and which I/O devices he wishes to use. The KOS implementation, for example, provides an overall entry command ENTER SCAN which enters SCAN and goes into state 0, and supplementary commands PROG, RUN and TEXT. If non-default I/O devices are required these are specified by a suffix to the command. The RUN command might, for example, have the various forms

```
RUN                              (console  I/O)

RUN  TO  PRINTER                 (console input, printer output)

RUN  FROM  disc file   TO  disc file          (disc I/O)
```

Clearly, each operating system has its own conventions and the KOS method certainly need not be followed in other implementations.

The MI-logic distinguishes two kinds of output: the messages stream and the results stream. The messages stream is used for system messages and output produced by the WARN statement; the results stream is used for the normal output produced during listing and running.

In addition the MI-logic requires an input stream to exist during compiling and running.

Breaks

If SCAN is to be run conversationally, it is desirable to allow the user to "break" at any time, i.e. to press some key that stops the current activity and returns to command status. A break must not destroy the current program.

The MI-logic has been designed so that breaks are acceptable throughout any activity except compiling. During compiling, breaks can only be accepted while within the MDLINE routine (q.v.) . as at other times the program may be in an unstable state (e.g. a chain may be being updated). This is no inconvenience for the user as compiling of lines should be almost instantaneous, so it will appear to him as if breaks are always acceptable.

It is suggested that the action for an acceptable break should be to go to the MDCOM routine (q.v.) and for an unacceptable break to go to the MDQUIT routine (q.v.).

If breaks are to be catered for, it is suggested that the variable STATE be used to record the status. The following actions should be added to the MD-logic:

(a) initialization should set STATE to zero before any possible break can occur.

(b) the MDLINE routine should, on entry, preserve the old value of STATE and re-set STATE to 3, and on exit should restore STATE to its previous value.

Otherwise the value of STATE would be as set by the MI-logic. With this scheme, breaks are acceptable when STATE is non-zero.

The MD-logic is defined in detail in the following paragraphs.

Initialization code

SCAN uses its stack area mainly to store the user's program and the current sentence. Obviously the stacks should be as large as practicable, but experience has shown that useful SCAN programs can run using an area of less than one thousand words.

The numbers of C variables and N variables that are to exist is defined by the SCAN user at the start of the session. The way this is done will vary between implementations, and it is the duty of the initialization code for each implementation to reserve storage for these variables.

In detail, the complete list of duties of the initialization code is as follows:

(a) if necessary, set STATE to zero before any break can occur.

(b) reserve the C and N variables and set

NUMC = number of C variables.

ADDRC = address of first C variable (C1) minus OF(LCH).

NUMN = number of N variables.

ADDRN = address of first N variable (N1) minus OF(LNM).

(ADDRC and ADDRN gives the address of the hypothetical elements CØ and NØ.)

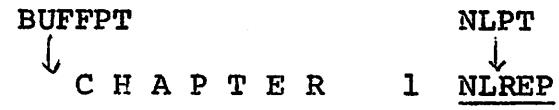
(c) set MAXINT as the maximum allowable integer in the program.

(d) perform the common initialisation code described in the LOWL manual.

The MDLINE subroutine

MDLINE is a subroutine that reads lines of input, either during compilation or at run-time. It has two exits, exit 1 being used if data is exhausted. Otherwise it reads a line of input into a buffer, sets BUFFPT to point one character position before the start of the buffer and uses exit 2. (BUFFPT should be reset on each call of MDLINE, since the MI-logic changes it.)

MDLINE is responsible for reserving space for the buffer. (Note that the buffer must not be in read-only storage as a SCAN program can change the buffer.) The text in the buffer should be terminated by the newline character, as represented by NLREP, and NLPT should be set to point at this character. Thus, for example, if an input line reads "CHAPTER 1" this should be set up as follows



Note that the MI-logic works in units of lines, each terminated with a newline. This may necessitate artificially adding a newline at the end of the very last line if the input stream has an incomplete line at the end.

The MDERCH subroutine

The MDERCH subroutine, which has one exit, outputs on the messages stream the character in C. If this character is SOSREP, EOSREP or NULREP it should be ignored. There is no guarantee that the value of C will be a legal character code. For example the program might read

```
10  LET  C1 = 134

20  PRINT  C1
```

This would result in MDERCH being called with the number 134 in C. The action to be taken for illegal codes is up to the implementor. The user's manual for an implementation should specify what this action is.

Note that MDERCH needs to cater for tab and newline characters.

The MDERNM subroutine

The MDERNM subroutine, which has one exit, outputs the number in A on the messages stream. This number should be converted to a decimal representation. It may be negative, in which case a minus sign should be output in front of it. Redundant leading zeroes should be suppressed.

For example the MI-logic code

```
MESS     'ABC'
LCN      NLREP
GOSUB    MDERCH,X
LAL       0
SAL      123
GOSUB    MDERNM,X
MESS     'XYZ$'
```

should generate the two lines

```
ABC
-123XYZ
```

The MDOUCH and MDOUNM subroutines

The MDOUCH and MDOUNM subroutines are identical to the MDERCH and MDERNM subroutines, respectively, except that they use the results stream rather than the messages stream.

The MDTLIS subroutine

The MDTLIS subroutine has two exits. It is called when a compile-time error message is being output, and its purpose is to decide whether the offending statement should be listed. In general, it should be listed only if the device used for messages is different from the input device, i.e. if the usage of SCAN is not conversational.

Exit 2 should be used if a listing is required, exit 1 if it is not.

The MDCOM subroutine

The MDCOM subroutine is entered at the end of a compilation, run or listing. It should close the I/O - there may be incomplete lines left on either the results stream or the messages stream at the end of a run. It should then communicate with the user to find out what to do next, reset I/O devices if necessary, and re-enter the MI-logic at one of the labels PROG, RUN or TEXT, depending on which activity is to be performed next.

The MDQUIT subroutine

The MDQUIT subroutine is called after a fatal error. It should close all I/O and terminate.

Documentation

When an implementation of SCAN is working, the implementor needs to provide documentation for the user. This can, if desired, be a copy of the KOS SCAN Manual with suitable alterations to cover changes in operating system interface, use of I/O devices, character set, size restrictions, action at breaks, etc.

Chapter 3 The compiled code

The following paragraphs describe the compiled form of a SCAN program. This information may be of use to the implementor if bugs arise. The following notation is used.

(a) Pseudo-ops with operands are written operation code/operand.

(b) Each field is followed by an indication of its type and size. This is given by the name, in parentheses, of the appropriate subsidiary macro to the OF macro. For instance the pseudo-op 23 (which has no operand) would be written

23 (LFN)

(c) STR is used to mean the equivalent encoding to the STR statement in LOWL.

A complete table of operation codes appears at the end of this Chapter.

Within the compiled program, expressions are represented in a Reverse Polish form. (See Software Practice and Experience, July 1972 for a discussion of this.) This consists of pseudo-ops to stack operands and pseudo-ops to operate on the operands at the top of the stack. (The result, if any, is placed at the top of the stack.) In addition each statement has two fields of header information. For example the statement

```
10  LET  N1 = L7+3
```

would be compiled as

7/10	(LIN)	Set statement number as 10
length of statement	(LNM)	
4/ address of N1	(LIN)	Stack address of N1
2/ address of L7	(LIN)	Stack value of L7
0/3	(LIN)	Stack literal value 3
22	(LFN)	Add
31	(LFN)	Assign

The program is stored with statements end-to-end in numerical order. The program is terminated by an end marker of form

7/0 (LIN)

Statements themselves are encoded in the following way

7/statement number (LIN)

total size of encoded statement (LNM)

statement body

Present only if there is a comment:

43 (LFN)

STR 'comment' (...* LCH)

NLREP (LCH)

The statement bodies of the various types of statement are as follows.

(1) Null statement

16 (LFN)

(2) STOP statement

20 (LFN)

(3) GOTO statement

If it is GOTO 0 then

17 (LFN)

Otherwise

10/statement number (LIN)

pointer to position of designated statement (LNM)

(4) AT statement

element 1

:

element N

6/N (LIN)

The elements on all AT statements are connected together on two chains, which run right through the program. One chain contains word-patterns, the other elements that match separators. Word-patterns are compiled in their original character form. The only change that is made is that if there is a dash at the end this is replaced by SPREP; otherwise NLREP is added to terminate the word-pattern. Other AT elements are encoded exactly as if they were expressions. Each element is preceded by an operation with code 9 (for word-patterns) or 8 (otherwise) with the operand giving the relative offset of the next item of the chain.

For example

```
AT   "A", C1, "+", "PIG-"
```

would be encoded

9/OF(5*LIN+2*LCH)	(LIN)	Word-pattern chain
A	(LCH)	Word-pattern "A"
NLREP	(LCH)	Terminator
8/OF(LIN+LIN)	(LIN)	Separator chain
4/address of C1	(LIN)	Stack value of C1
8/offset of next	(LIN)	Separator chain
1/+	(LIN)	Stack the character "+"
9/offset of next	(LIN)	Word-pattern chain
P	(LCH)	Word-pattern "PIG"
I	(LCH)	
G	(LCH)	
SPREP	(LCH)	Dash at end
6/4	(LIN)	Terminator

(5) LET statements, IF clauses

These are encoded entirely in Reverse Polish.

(6) PRINT and WARN statements

18(for PRINT) or 19(for WARN) (LFN)

Reverse Polish

44 (LFN)

The main body is pure Reverse Polish except that elements that are character constants are represented as follows:

46		(LFN)
STR	'character constant'	(...* LCH)
NLREP		(LCH)

For example the statement

```
PRINT  "XY", A1 TO AN3, L1+3
```

would be encoded

18	(LFN)	Set stream as results
46	(LFN)	Long character constant
X	(LCH)	"XY"
Y	(LCH)	
NLREP	(LCH)	Terminator
4/address of A1	(LIN)	Stack of address of A1
5/address of ADDRN	(LIN)	Stack address of ADDRN ("dope vector" - see MI-logic)
2/address of N3	(LIN)	Stack value of N3
28	(LFN)	Stack address of array element
30	(LFN)	Output multiple characters
2/address of L1	(LIN)	Stack address of L1
0/3	(LIN)	Stack literal value 3
22	(LFN)	Add
41	(LFN)	Output single integer
44	(LFN)	End of output

Table of operation codes

The following table defines all the existing pseudo-ops. The column "use of stack" should be interpreted thus

+ n	means adds n items to the stack.
- m	means removes m items from the stack and uses them as operands.
—	means does not use the stack.

Note that a pseudo-op such as addition, which removes the top two items from the stack and then puts the result back, is described as -2,+1.

Code	Meaning	Use of stack
0	Stack literal integer	+1
1	Stack literal character	+1
2	Stack value of integer variable	+1
3	Stack value of character variable	+1
4	Stack address of variable	+1
5	Stack address of dope vector	+1
6	End of AT statement	-
7	Statement number	-
8	AT separator chain	-
9	AT word-pattern chain	-
10	GOTO statement	-
11-15	Not used	-
16	Null statement	-
17	GO TO 0 statement	-
18	Start of PRINT	-
19	Start of WARN	-
20	STOP statement	-
21	Subtract	-2,+1
22	Add	-2,+1
23	Divide	-2,+1
24	Multiply	-2,+1
25	Subscript value (integer)	-2,+1
26	Subscript value (character)	-2,+1
27	Subscript address (integer)	-2,+1
28	Subscript address (character)	-2,+1
29	TO (integer)	-2
30	TO (character)	-2
31	Assign (integer)	-2
32	Assign (character)	-2
33	=	-2
34	<	-2
35	LE	-2
36	>	-2
37	GE	-2

Code	Meaning	Use of stack
38	NE	-2
39	Unary minus	-1,+1
40	Left parenthesis	-
41	Output (integer)	-1
42	Output (character)	-1
43	Comment	-
44	End of output statement	-
45	Right parenthesis	-
46	Long character constant	-