SL/M: A Minicomputer-Oriented System Description Language

Keiji MAKINO*, Koji TOCHINAI* and Kuniichi NAGATA*

Abstract

In this paper, a new minicomputer-oriented system description language, SL/M, and its implementation are described. There exists no efficient high level language for the description of system programs in the minicomputer field; therefore the user must use the assembly language. SL/M was planned out to solve these situations.

SL/M is a high level language designed for the system program implementation on minicomputers. It easily expresses a state transition diagram, which in an effective tool to develop a system program. SL/M is now in use on a minicomputer, and its processor was implemented by the bootstrapping method. We believe that SL/M is convenient for the system and utility program production and expands minicomputer utilization.

1. Introduction

In many cases of minicomputer applications, the minicomputer is used at the minimum construction, for which there exists no efficient high level language. And the user, who is mostly unfamiliar with computers or programming, must use the assembly language. The programming in the assembly language demands that the user have a knowledge of hardware. Moreover the small scale minicomputer has many inherent constraints, e.g., the smaller main memory capacity, the small direct addressable area, a small number of registers, and so on. They result in increased programming time, frequent errors, and difficult debugging.

The SL/M is a new minicomputer-oriented high level language for the production of system and utility programs. Owing mainly to its simple structure, it can be used even by beginners.

2. SL/M Language

2.1 Considerations in System Description

The design and production steps of a system program are analyzed as follows.

This paper first appeared in Japanese in Joho-Shori (Journal of the Information Processing Society of Japan), Vol. 17, No. 3 (1976), pp. 207~214.

* Department of Electronic Engineering, Faculty of Engineering, Hokkaido University

1) Determine the input, the output, and their relations on the entire system.

2) Comprehend the global image of the process from the system input through the system output, and decompose the entire system into subsystems.

3) Decompose each subsystem into subsystem units which realize elementary functions.

4) Decide the actual action of the elementary subsystem unit.

5) Write the program.

In the above analysis, the efficient tool is the data-oriented description method like the signal flow graph for step (2) and (3) and the state transition diagram for step (4), and the entire system is represented as the compositive graph of both. On the above concept, each subsystem is modelled as an abstract machine shown in Fig.1, and is specified by the graph representation shown in Fig.2.

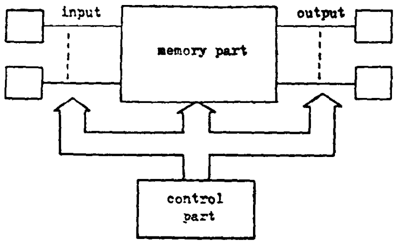

Fig.1 The structure of an abstract machine

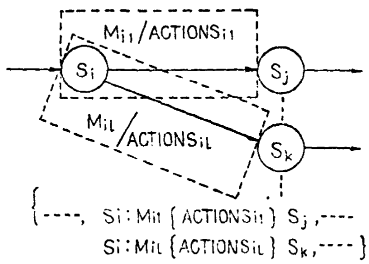

Fig.2 The state diagram representation

2.2 Structure of SL/M

SL/M is constructed from the two descriptive parts corresponding to the abstract machine. The whole syntax is given in the appendix.

o Memory part

The memory part is specified by "declaration statement". The general form is

DCL␣$v_1, v_2, \ldots, v_n$; ,where v_i are simple variables or one-dimensional arrays.

A sata type is a one-word-length bit-pattern, which is represented in the program by an octal number or a character string, and its interpretation is subject to the user.

o Control part

The control part is represented by "executable statement" and "SUB statement".

A SUB statement represents a series of transitions.

An executable statement represents the elementary transition of the abstract machine. The general form of an executable statement is as follows.

S_i: ON(condition) actions; GOTO S_j;<cr> ,where <cr> denotes a carriage-return code. The "condition" is modified by the "specifier" (e.g., ON) and, as a result of its interpretation, the "executive condition" decides the method of execution for the rest of the line. This language has a congruence with the practice of the usual procedure-oriented language, but has not the purely independent concept of control statement which is common in other languages.

For "condition", the following representation is admitted.

TO △ T1,T2, ... ,TN ,where △ is a relational operator. This is interpreted as

(TO=T1) ∨ (TO=T2) ∨ ... ∨ (TO=TN) if the relational operator △ is "=",

(TO△T1) ∧ (TO△T2) ∧ ... ∧ (TO△TN) otherwise.

They are equivalent to the following relation.

$t_0 \in \{t_1, t_2, \ldots, t_n\}$ (for =), $t_0 \notin \{t_1, t_2, \ldots, t_n\}$ (for \=).

The elementary actions (ACTION) of an abstract machine are described by elementary executable statements. These are classified on the three classes: input (to memory), output (from memory), and memory rewriting.

The assembly language may be embedded by a PROC statement. The linkage between the embedded assembly language part and the SL/M language's body is performed through the identifier or by the subroutine-call form.

3. Example of SL/M Program

Fig.3 represents the syntax of an assignment statement and its analysis process. And the program of it written in SL/M is shown in Fig.4.

4. Implementation[1]

The SL/M compiler is produced on a NOVA 01 minicomputer with 8-kwords memory, a teletypewriter, and a paper-tape reader by the bootstrapping method.[2][3] The compiler is a 2-step stream-data conversion type composed of an analysis phase and a code generation phase. Its delivered data is represented by a machine-independent intermediate language based on the modified Polish notation. Each phase is composed of the modules realizing the logical function unit, and has the hierarchy structure. By the above design concept, the inner structure of the compiler and the control relation among the modules are clear, and the expansion or modification of the compiler is easy.

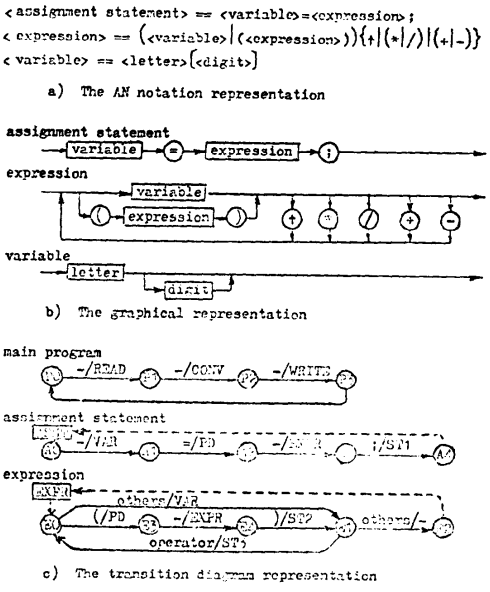

Fig.3 Representation of assignment statement

```
* SAMPLE PROGRAM (SYNTAX ANALYSIS OF ASSIGNMENT STATEMENT
* AND CONVERSION TO REVERSED POLISH NOTATION);

* INPUT BUFFER & ITS POINTER;
DCL BI(128), PI;

* OUTPUT BUFFER & ITS POINTER;
DCL BO(128), PO;

* STACK & ITS POINTER;
DCL STACK(128), PS;

* WORKING VARIABLE;
DCL I1, I2, WK;

* MAIN PROGRAM;
M01: CALL READ; CALL CONV; CALL WRITE; [illegible]

READ: SUB;
      PI=1;
 RD1: IN(10,WK); WK=WK&177; OUT(11,WK);
      ON(WK\=8,12,[illegible],177) BI(PI)=WK; PI=PI+1;
      ON(WK\=";") [illegible] TO [illegible]
      OUT(11,/);
      END;

WRITE: SUB;
      PO=1;
 WT1: ON(BO(PO)\=";") OUT(11, BO(PO)); [illegible] GOTO WT1;
      OUT(11,";",/,/,/);
      END;

ERROR: SUB;
      OUT(11,/,"ERROR",/,/);
      END;

* CONVERSION;
CONV: SUB;
      PI=1; PO=1; PS=0; CALL ASST.
      END;

* ASSIGNMENT STATEMENT;
ASST: SUB;
      CALL VAR;
      ON(BI(PI)\="=") CALL ERROR;
      CALL PD;
      CALL EXPR;
      ON(BI(PI)\=";") CALL ERROR;
      CALL ST1;
      END;

* EXPRESSION;
EXPR: SUB;
 EX1: [illegible](BI(PI)\="(") CALL VAR; GOTO [illegible]
      CALL PD; CALL EXPR;
      [illegible](BI(PI)\=")") CALL ERROR;
      CALL ST2;
 EX2: [illegible](BI(PI)="↑","*","/","+","-") CALL [illegible] GOTO [illegible]
      [illegible]

* VARIABLE;
VAR: SUB;
      ON(BI(PI)<=100) CALL ERROR;
      ON(BI(PI)>=133) CALL ERROR;
      I1=BI(PI); PI=PI+1; I2=[illegible];
      ON(BI(PI)>=60)ON(BI(PI)<=[illegible]) I2=BI(PI); PI=PI+1;
PROC;
LDA 0,I1
LDA 1,I2
MOVS 1,1
ADD 1,0
STA 0,I1
END;
      BO(PO)=I1; PO=PO+1;
      END;

* PROCESSING ON STACK (ST1, ST2, ST3);
ST1: SUB;
 S11: ON(STACK(PS)="(") CALL ERROR;
      ON(STACK(PS)\="=") [illegible] GOTO S11;
      CALL POP; BO(PO)=";";
      END;

ST2: SUB;
 S21: ON(STACK(PS)="=") [illegible]
      ON(STACK(PS)\="(") [illegible] GOTO S21;
      PI=PI+1; PS=PS-1;
      END;

ST3: SUB;
 S31: ON(BI(PI)="↑")ON(STACK([illegible]) CALL POP; GOTO S31;
      ON(BI(PI)="*","/")ON(STACK([illegible]"↑","*","/") [illegible] POP; GOTO S31;
      ON(BI(PI)="+","-")ON(STACK([illegible]"(","=") CALL POP; GOTO S31;
      CALL PD;
      END;

* PUSH-DOWN;
PD:  SUB;
      PS=PS+1; STACK(PS)=BI(PI); PI=PI+1;
      END;

* POP-UP;
POP: SUB;
      BO(PO)=STACK(PS); PS=PS-1; PO=PO+1;
      END;

* END OF SAMPLE PROGRAM;

STOP;
```

Fig.4 An example of SL/M program

5. Conclusion

In SL/M, anything associated with a single transition on a transition diagram can be easily described by one "executable statement" in compact, understandable form. The program is directly converted from a graph representation given by the analysis of the objective without going through the flow chart representation. Therefore, by SL/M, even the beginner, who is unfamiliar with computers or programming, can easily make necessary software on a small scale minicomputer.

The language structure is simple, the function included on SL/N is not very wealthy,[4] and certainly addition of other functions might be suggested. However, considering the minicomputer-orientation, this language is fairly powerful in the system description, and the present system balances in functions, convenience, and hardware limitations (e.g., memory capacity).

The size of the compiler is shown in Table 1. In the bootstrapping processes using SL/M subset, the degree of errors was 1 error/100 lines on the average.

Table 1 Size of SL/M compiler

	analysis phase	code generation phase
source program (nearly all states)	about 280 lines	about 230 lines
object program's instruction part	about 2100 steps	about 1300 steps
explicit states (labels)	76	50

Reference

1) Special Edition: ETL's System Description Language, Bulletin of the Electrotechnical Laboratory, Vol.34, No.5-6, 1970 (in Japanese).

2) Kokubo & Saya: A Compiler Writing Language BPL, J.IPSJ, Vol.11, No.6, pp.342-349, 1970 (in Japanese).

3) Earley & Sturgis: A Formalism for Translator Interactions, Comm.ACM, Vol.13, No.10, pp.607-619, 1970.

4) Yamada et al.: Studies of writing a FORTRAN compiler using FORTRAN language, Bulletin of the Faculty of Engineering, Hokkaido University, Japan, Vol.73, pp.71-83, 1974 (in Japanese).

5) Wada: ALGOL N, J.IPSJ, Vol.12, No.9, pp.556-567, 1971 (in Japanese).

Appendix: SL/M syntax (by AN notation[5)])

```
<program> == [(<declaration statement>|
      <executable statement>|<SUB statement>|
      <comment statement>)<cr> ... STOP[_<label>] ;<cr>
<SUB statement> == <label>:SUB;<cr>
      [(<declaration statement>|<executable statement>|
      <SUB statement>|<comment statement>)<cr>]... END;
<comment statement> == *[<character>]...
<declaration statement> == DCL_(<identifier>
      [:<constant>|(<octal number>)]){,}...
<executable statement> == [<label>:]
      [<executive condition>]...
      [<primitive executable statement>;]...
      [<GOTO statement>;]
<executive condition> == <specifier>(<condition>)
<specifier> == ON
<condition> == <term><relational operator><term>{,}...
<relational operator> == =|<|>|(<=|=<)|(>=|=>)|
      (\=|=\|><|<>)
<primitive executable statement> == <null statement>|
      <assignment statement>|<CALL statement>|
      <PROC statement>|<HALT statement>|
      <IN statement>|<OUT statement>
<null statement> ==
<assignment statement> == <variable>=<term>{+|-|*|&}...
<CALL statement>* == CALL_<label>[(<term>{,}... )]
<PROC statement>** == [<label>:]PROC;<cr>
      [[<character>]... <cr>]... END
<HALT statement> == HALT
<IN statement> == IN(<octal number>,<variable>{,}... )
<OUT statement> == OUT(<octal number>,(<term>|/){,}... )
<GOTO statement> == GOTO_<label>
<label> == <identifier>
<term> == <variable>|<constant>
<variable> == <identifier>[((<identifier>|<constant>))]
<constant> == <octal number>|<character string>
<identifier>*** == <letter>[<alpha-numeric>]...
<octal number> == <octal digit>...
<character string> == "[<any character
      except for " >]... "
<letter> == A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
      U|V|W|X|Y|Z
<octal digit> == 0|1|2|3|4|5|6|7
<alpha-numeric> == <letter>|<octal digit>|8|9
<character> == <any character on teletypewriter
      containing space>
<cr> == <carriage-return>
```

* On the CALL statement, parameters are permitted only if the called subroutine is described by a PROC statement.

** In each line of PROC statement's body, the first syllable must be except for "END".

*** The scope of identifiers covers the entire program, and the first four characters are used for the identification. Key words are the reserved word.