The Minicomputer-Oriented System Description Language SL/M2

Keiji MAKINO, Koji TOCHINAI and Kuniichi NAGATA

(Received June 30, 1975)

Abstract

SL/M2 is a minicomputer-oriented high level language designed for the production of system and utility programs. SL/M2 is extension of SL/M, which is a minicomputer-oriented language designed to realize the basic functions for system program description and is a preliminary step to SL/M2. The main differences between SL/M2 and SL/M are

1) prohibiting the assembly language embedding,
2) providing a number of system subroutines for stack manipulation etc., and
3) introducing the specifier WHILE.

This paper presents a formal description of SL/M2 grammar. Under this specification, SL/M2 compiler is implemented for a TACC 1200 minicomputer. And, on its implementation, the bootstrapping method is used in succession to SL/M compiler production.

I. Introduction

In recent years, the use of minicomputers has become extensive. But, the minicomputer users encounter considerable difficulties in processing their work because an efficient high level system description language for minicomputer utilization had not been developed. Accordingly, SL/M[1)2)] were planed out and implemented.

The SL/M is a minicomputer-oriented high level language for the production of system and utility programs, and can easily used to program the object expressed by a state transition diagram model. It has good writability, readability, and learnability owing mainly to its simple structure. The basic concept is that, in this language, there is only one executable statement form which corresponds to the transition from one state to another on a state transition diagram.

In this paper, SL/M2 is reported which is a higher version of the SL/M and is extended by the bootstrapping method. SL/M2 differs from SL/M mainly in the following points:

Department of Electronic Engineering, Hokkaido University, Sapporo, Japan.

1) The means of connecting with an outer assembly language routine is provided in the system, although the assembly language embedding is removed.
2) A number of system subroutines are made ready for stack manipulation and packing, etc.
3) The specifier WHILE is allowed for the representation of the repetition loop.

Section II of this paper describes briefly the main features of the language. Section III is the formal description of the grammar using the AN notation[3] for the syntax. Section IV describes the implementation of SL/M2 compiler placing emphasis on the differences with SL/M compiler. Section V is the conclusion.

II. The main features of SL/M2

SL/M2 is constructed in such a way that the beginner, who is unfamilier with computers or programming, can easily use a small minicomputer (i.e. word-length at most 16 bits, memory 4–8 kwords, teletypewriter, paper-tape reader). Therefore it is desirable that the language structure is simple and yet powerful in the system description. SL/M2 has its main features as follows.

SL/M2 has only one basic type of statement to be executed, namely the executable statement. The executable statement is fundamentally composed of two parts. One part, called executive condition, represents the executing manner of its statement, corresponding to the condition for a transition from state to state on the state transition diagram, and the other part is a sequence of so-called basic executable statements, corresponding to actions performed with the transition. All things with a transition, thus, can be represented by one executable statement. And, its representation form is designed by considering the compactness and also the extendability of the language.

The data type is the one-word-length bit-pattern type only, but its value on a program is described by using the octal notation or character string expression. The bit-pattern is specified by the user (or the programmer) for the purpose of the program described. Accordingly, it may be a character code of any type, an integer (signed or unsigned), a bit string, or purely bit pattern, etc.

In the SL/M2, two usual types of conventional operators are conveniently made use of i.e. the arithmetic type, and the logical type. The arithmetic type operators interpret the bit-pattern of the both operands as a unsigned integer, and the logical type operators interpret the bit-pattern of the both operands as a bit string.

The function of the assembly language embedding is removed which is permitted in SL/M. For the assembly language embedding is convenient but has

a possibility of spreading assembly code over the entire program. This means the program using its function is not machine independent and therefore is inconsistent with the original purpose of high level language development. Therefore, on SL/M2, the prohibition of the assembly language embedding is attempted. Instead of it, SL/M2 has a means of connection with another language routine defined outside. It is .SYS routine which is one of the subroutines provided by SL/M2 system. The .SYS routine may have some input parameters and/or some output parameters. Only by these parameters, the SL/M2 program communicates with the outer routines.

Other two groups of subroutines are also provided in the SL/M2 system. One group makes use of byte-data conveniently. .PACK routine packs each lower-byte of two words into one word. .UPU and .UPL unpack one word, and take out an upper-byte and a lower-byte, respectively. Another group helps the push-down stack manipulation. .PUSH routine pushes-down the values in its parameters into a push-down stack, and .POP routine poppes-up the value from a push-down stack into its parameters concersely.

On trial, the specifier WHILE which expresses a repetition loop is newly introduced. This is one step toward taking in various specifiers, e.g. another repetition loop from, *case*-like representation, etc.

III. SL/M2 grammar

In this section, the formal description of SL/M2 is represented.

3.1 General Rules

3.1.1 The syntax will be described with the aid of the AN notation[3].

3.1.2 The semantics will be described with the natural language.

3.1.3 Program is physically composed of lines. The line is a character string between two carriage-returns, and it is composed of 80 characters at most.

3.1.4 The representation feature is free-formatted in the line.

3.1.5 The characters used for the program description are any characters on the teletypewriter except for the control characters.

But carriage-return has the meaning of line delimiter, and line-feed (LF) is only used for the descriptive form adjustment. Moreover, null (NUL) and delete (DEL) are ignored even if they might be used for the tape form adjustment.

For other control characters, their functions are not specified at this language specification.

3.2 Basic and Lexical Concepts

3.2.1 Basic elements

<character> == <any character on teletypewriter containing space>
<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
<digit> == <octal digit> |8|9
<cr> == <carriage-return>

Although the carriage-return is a control character, it is explicitly denoted since it is used as the line delimiter.

3.2.2 Identifiers, octal numbers, character strings.

1) Syntax

<identifier> == <letter>[<letter>|<digit>]ooo
<octal number> == <octal digit> ooo
<character string> =="[<any character except for">]ooo "

2) Examples

Identifiers:	ID	NAME	L10	S1N8	BUFFER
Octal numbers:	0	123	060	7654321	0001
Character strings:	"CHARACTER STRING"			"	LDA 0, ID"

3) Semantics

Identifiers serve for the identification of simple variables, arrays, labels, and subroutines. They may be chosen freely except for key words. The first four characters are used for the identification. The scope of these identifiers covers the entire program.

Octal numbers and character strings represent bit-patterns to be converted.

3.2.3 Constants, variables, terms, and labels.

1) Syntax

<constant> == <octal number>|<character string>
<simple variable> == <identifier>
<array identifier> == <identifier>
<subscripted variable> == <array identifier> ((<simple variable>|<constant>))
<variable> == <simple variable>|<subscripted variable>
<term> == <variable>|<constant>
<label> == <identifier>

2) Examples

Simple variables:	I WK ALPHA C123 POINTER
Subscripted variables:	X(I) BUFFER(12) H("A") STACK(POINTER)
Labels:	LABEL L010 SUBNAME MAIN

3) Semantics

A constant except for the character string contained in output statement is a one-word-length fixed bit-pattern.

A variable is a designation given to a single bit-pattern.

Interpretation of each bit-pattern is subject to the user.

3.3 Executable Statements

The executable statement is the logically basic composition element for the description of computation part.

3.3.1 Executable statements

1) Syntax

```
<executable statement> == [<label>:] [<executive condition>]
   (<primitive executable statement>;|<executable statement>)ooo
<specifier> == ON | WHILE
<relational operator> == = |<|>|(\=|=\|< >|> <)|(<=|=<|\>|>\)|(>=|=>|\<|<\)
<condition> == <term> <relational operator> <term> {,}ooo
<executive condition> == <specifier>(<condition>)
<primitive executable statement> == <null statement>|<assignment statement>
   |<input statement>|<output statement>|<call statement>
   |<system-subroutine call statement>|<goto statement>|<halt statement>
```

2) Examples

```
ON(X(I) = 0,1,2) WK = X(I); I = I+1;
L: ON(I1 = " /")ON(I2 = "=") PACK(I1,I2: OP);CALL SYS1;
P: ON(A(1) = C1) B = S(J); ON(Z = 0) OUT(11, A(Z),/); GOTO Q;
```

3) Semantics

The executable statement is performed under the control of executive conditions.

In the case of specifier ON, if the condition following ON holds, the succeeding part of its executable statement is performed, otherwise the next executable statement is performed.

In the case of WHILE, while the condition following WHILE holds, the executable statement is repeatedly performed. Whenever the condition turns not to hold, the next executable statement is performed. In this version, it should be taken note that the specifier WHILE is permitted at most one time in one executable statement.

The meanings of relational operators are as follows.

relational operator	*meaning*
=	equal to
<	less than
>	greater than
<= (=<, \>, >\)	less than or equal to
>= (=>, \<, <\)	greater than or equal to
\= (=\, ><, < >)	not equal to

The condition containing a term list on the right hand of the relational operator is interpreted according to the following.

$$TO = T1, T2, \ldots, TN == (T0 = T1) \vee (T0 = T2) \vee \ldots \vee (T0 = TN)$$

$$TO \triangle T1, T2, \ldots, TN == (T0 \triangle T1) \wedge (T0 \triangle T2) \wedge \ldots \wedge (T0 \triangle TN)$$

(where "△" is any relational operator except "=", and "∨" and "∧" are disjunction and conjunction in logic, respectively.)

3.3.2 Primitive executable statements

3.3.2.1 Null statement

1) Syntax

<null statement> ==

2) Example

L: ; null statement with label

3) Semantics

A null statement executes no operation. It may serve to place a label.

3.3.2.2 Assignment statements

1) Syntax

<assignment statement> == <variable>=<term> {<arithmetic-logical operator>}...
<arithmetic-logical operator> == +|−|*|/|&|!

2) Examples

I = 1 A(I)=T1+T2 WK = M&7400 + N B(I) = WK&177 ! 60

3) Semantics

An assignment statement is a rule for converting a bit-pattern. But operations used in the assignment statement are defined by the conventional usage with the interpretation of each bit-pattern as an unsigned integer or a bit string.

The operators are binary operators, and have the following meanings.

+	addition
−	subtraction
*	multiplication
/	division

These operators interpret the bit-pattern of the both operands as unsigned integers, and ignore the overflow on the results. Also, subtraction can be regarded as 2's complement's addition.

&	AND
!	Exclusive OR

These operators interpret the bit-pattern of the both operands as a bit string, and their operations are performed bitwise.

No precedence of operators exists, and the sequence of operations within an assignment statement is always from left to right. Moreover, no use of parentheses is permitted.

The right hand of the assignment symbol (=) is evaluated in this way, and subsequently the bit-pattern of its result is assigned to the left hand variable of the assignment symbol.

3.3.2.3 Input statements

1) Syntax

<input statement> == IN ((<simple variable>|<octal number>), <variable>{,} ...)

2) Exmples

IN(12, BUFFER(I)) IN(UNIT, WK1, WK2, WK3) IN(10, P,BI(P))

3) Semantics

An input statement serves to read the data into variables from the input device. By the first parameter an input device is specified, and the data are read into succeeding parameters one by one.

3.3.2.4 Output statements

1) Syntax

<output statement> == OUT((<simple variable>|<octal number>),
(<term>|/){,} ...)

2) Examples

OUT(11, BUFFER(J)) OUT(UNIT, " *** OUTPUT *** ", /,/)

3) Semantics

An output statement serves for transferring the data to an output device. By the first parameter an output device is specified, and the data are output from succeeding parameters one by one. If any of those parameters is an octal number or term, the output is its bit-pattern interpreted as a code. If it is a character string, the output is an unchangeable character contained between double-quotation marks ("). In the case of solidus (/), a carriage-return occurs on the teletypewriter.

3.3.2.5 Call statements

1) Syntax

<call statement> == CALL <label>

2) Examples

CALL SUBROUTINE CALL READ CALL M2D6

3) Semantics

A call statement serves in such a way that the user subroutine with a label should be called which is defined by the subroutine statement in another part of this program. The call statement has no parameter because the scope of identifiers is over the entire program.

3.3.2.6 System-subroutine call statements

1) Syntax

<system-subroutine call statement> == .PACK (<term>,<term>:<variable>)
|(.UPU|.UPL)(<term>:<variable>)|.PUSH(<term>{,} ...)|.POP (<variable>{,} ...)
|.SYS [(<term>{,} ...)/(:<variable>{,} ...)]

2) Examples

```
.PACK (W1, W2 : W1)
.UPU (B(I) : WK)
.UPL (300101 : CH)
.PUSH (0, J, A(J), "Z", 4)
```

.POP (T1(K), T2(K), PP)
.SYS (3, B(0), O1, O2, O3: I1, I2, I3, I4)

3) Semantics

A system-subroutine call statement serves to call the routine provided by the SL/M2 system. Each system-subroutine has the following meanings.

.PACK routine performs packing of each lower-byte of the first and the second parameters into the third parameter from lower to upper.

.UPU routine takes out the upper-byte of the first parameter and puts its byte into the second parameter.

.UPL routine takes out the lower-byte of the first parameter and puts its byte into the second parameter.

.PUSH routine performs pushing-down the value of each parameter into a stack in sequence.

.POP routine performs popping-up of one word from the stack and puts its value into each parameter in sequence.

.SYS routine is a universal connector with outer assembly language routines. Each parameter prior to a colon (:) is the output parameter, and the value of the parameter is given to outer routines through the inherent locations in the system. On the other hand, each parameter succeeding the colon is the input parameter, and the value from the outer routine is given to the parameters. The input parameters, the output parameters or both may be omitted.

3.3.2.7 Goto statements

1) Syntax

<goto statement> == GOTO <label>

2) Examples

GOTO LABEL GOTO L123 GOTO DC45

3) Semantics

A goto statement serves to indicate that further processing should continue in another part of the program text, namely at the place of the label.

3.3.2.8 Halt statement

1) Syntax

<halt statement> == HALT

2) Example

HALT

3) Semantics

A halt statement serves in such a way that the user program should halt at the place. By indication from the console (e.g. turning on the continue switch), the user program continues the subsequent execution.

3.4 Comment Statements

1) Syntax

<comment statement> ==* [<character>]ooo

2) Example

```
* COMMENT ... CONTAINING ANY CHARACTER
```

3) Semantics

A comment statement serves to give a convenient means for the user. The comment statement has no effect on the program execution.

3.5 Declaration Statements

1) Syntax

<upper bound> == <octal number>
<array> == <array identifier>(<upper bound>)
<declaration statement> == DCL (<simple variable>[:<constant>] | <array>){,}ooo;

1) Examples

```
DCL I, C01:1, A(100);
DCL B(120), P, CA: "A";
```

3) Semantics

A declaration statement serves to define that an identifier represents a simple variable or an array. The array identifier must be followed by an octal number surrounded with a pair of parentheses which specifys the upper boundary of the subscript. The lower boundary of the subscript for the array is assumed to be zero. The simple variable may assume the initial value by a constant following the colon (:).

The declaration for variables must precede the use of the variable. And also, the scope of the variables is over the entire program because the scope of identifiers is over all.

3.6 Subroutine Statements

1) Syntax

<subroutine statement> == <label>: SUB;<cr>[(<declaration statement>|<executable statement>|<subroutine statem>|<comment statement>)<cr>]ooo END;

2) Example

```
L: SUB;
   A=X+Y; B=X-Y; M=X*Y; D=X/Y;
   END;
```

3) Semantics

A subroutine statement serves to define the user subroutine. Each subroutine statement must always have a label (i.e. subroutine name) to be invoked by the call statement, and must not have any parameters because the scope of variables is over the entire program. Whenever the execution flow returns to the place called for, the flow must be through END.

3.7 Programs

1) Syntax

<stop statement> == STOP[<label>] ;
<program> == [(<declaration statement>|<executable statement>|<subroutine statement>|<comment statement>)<cr>]ooo<stop statement><cr>

2) Example

```
* SAMPLE PROGRAM;
DCL X, Y, Z;
L: IN(10, X, Y); Z=X+Y; OUT(11, Z, /);
   HALT; GOTO L;
STOP L;
```

3) Semantics

A program is the execution unit, and a stop statement must be placed as the last statement. The execution order is in the descriptive order except for encountering the statement changing the execution flow.

The label following STOP specifies the starting point of the program execution. If the label is missing, the starting point of the program execution is assumed to be the first executable statement.

IV. Implementation

The SL/M2 compiler is produced on a basic TACC 1200 minicomputer (i.e. word-length 16 bits/word, memory 8 kwords, teletypewriter, paper-tape reader).

The structure of SL/M2 is similar to that of SL/M compiler[1)2)], and the method of the production is the bootstrapping method in succession to SL/M compiler production. Therefore the SL/M2 compiler has a hierarchical modular structure and uses an intermediate language based on the modified Polish notation, too.

By using a specific register in a base-register-like way, Page Zero (i.e. direct addressable area) is preserved for other programs, and the number of usable identifiers increases. But, the small number of registers makes the generated code pattern inefficient, and the goto and call statement must be converted to

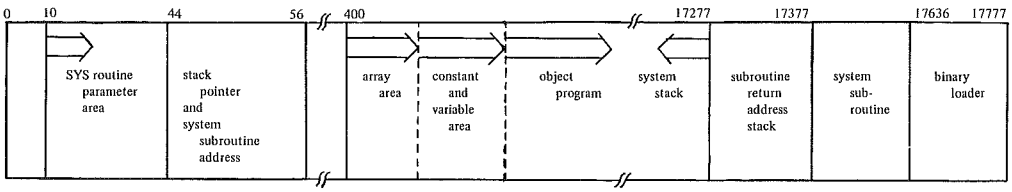

Fig. 1 Memory assignment on object program by SL/M2 compiler.

the indirect jump instruction which takes two words. The situation of the memory assignment on the object program is shown in Figure 1.

With introducing system-subroutines, internal codes for their system-subroutines' name are added[4)5)]. Thus there are seven kinds of the internal code classes: delimiter, constant, simple variable, array identifier, label, system-subroutine name, and another system-function name. Also the key-words (i.e. the names belonging to the latter two classes) are reserved in this system.

V. Conclusion

SL/M2 is based on SL/M, and is an improved version of SL/M. It has been previously described that the assembly language embedding is excluded, and a number of system-subroutines and the specifier WHILE are included. And also, it may be added that the input or output device specification can be performed at the execution of the input/output statement by permitting of simple variables for its first parameter.

But the function included on SL/M2 is not very wealthy. Certainly more addition of the function may be expected. However the required memory for each phase of present SL/M2 compiler is about 4 kwords, in both program body and its working area, and therefore it reaches the memory limitation on a 4-kwords system. On a 8-kwords system, it is possible to go over the limitation by exposing the intermediate language on a paper tape. But, in this case, the efficiency and the conveniency are sacrificed. So the author feels that the present system stands on the balanced point for functions and conveniences.

At present, considering the minicomputer-orientation, it seems that more refinement within the bounds of the already realized functions is required since almost every basic function has been realized on SL/M2.

Reference

1) Makino, K., Tochinai, K., and Nagata, K., "Minicomputer-Oriented System Description Language and its Implementation", Bulletin of the Faculty of Engineering, Hokkaido University, No. 75 (1975), pp.59–70.
2) Makino, K.: Studies of Minicomputer-Oriented System Description Language and its Processing System, doctoral thesis, Faculty of Engineering, Hokkaido University, (1975).
3) Wada, E., "ALGOL N (2-1)", J.IPSJ, Vol. 12 (1971), No. 91 pp.556–567.
4) Makino, K., "SL/M User's Manual", (1974), private document.
5) Makino, K., "SL/M2 User's Manual", (1975), private document.

Appendix: A sample program described by SL/M2 and its translated form.

The sample program converts an assignment statement which is expressed by the following syntax to the reversed Polish notation.

<assignment statement> == <variable>=<expression><cr>
<expression> == (<variable>|(<expression>)){↑|(*|/)|(+|−)}...
<variable> == <letter>{<digit>}

```
* CONVERSION OF ASSIGNMENT STATEMENT TO REVERSED POLISH NOTATION;

* INPUT BUFFER & ITS POINTER;
DCL BI(120), PI;

* OUTPUT BUFFER & ITS POINTER;
DCL BO(120), PO;

* WORKING VARIABLE;
DCL I1, I2, DV, OP;

* MAIN PROGRAM;
        CALL INIT;
M01:    CALL READ;
        CALL CONV;
        CALL WRITE;
        GOTO M01;

INIT:   SUB;
 IT1:   OUT(11, /,/, " INPUT DEVICE ... TTY(T) OR PTR(P) ?   ");
        IN(10, DV);     DV=DV&177; OUT(11, DV);
        ON(DV="T")      DV=10; GOTO IT2;
        ON(DV="P")      DV=12; GOTO IT2;
        GOTO IT1;
 IT2:   OUT(11, /,/,/);
        END;

READ:   SUB;
        PI=0; BI(0)=0;
 RD1:   IN(DV, BI(PI)); BI(PI)=BI(PI)&177; OUT(11, BI(PI));
        ON(BI(PI)=0, 12, 40, 177) GOTO RD1;
        ON(BI(PI)\=15) PI=PI+1; GOTO RD1;
        OUT(11, /);
        END;

WRITE:  SUB;
        PO=0; BO(0)=0;
        WHILE(BO(PO)\=15)  PO=PO+1; OUT(11, BO(PO));
        OUT(11, /,/,/);
        END;

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

CONV:   SUB;
        PI=1; PO=1; .PUSH(0);
        CALL ASST;
        END;

ASST:   SUB;
        CALL VAR; BO(PO)=I1; PO=PO+1;
        ON(BI(PI)\="=") CALL ERROR;
        .PUSH("="); PI=PI+1;
        CALL EXPR;
        ON(BI(PI)\=15) CALL ERROR;
        CALL ST3;
        END;

EXPR:   SUB;
 EX1:   ON(BI(PI)\="(") CALL VAR; BO(PO)=I1; PO=PO+1; GOTO EX2;
        .PUSH("("); PI=PI+1; CALL EXPR;
        ON(BI(PI)\=")") CALL ERROR;
        CALL ST1; PI=PI+1;
 EX2:   ON(BI(PI)="↑", "*", "/", "+", "−") CALL ST2; GOTO EX1;
        END;

VAR:    SUB;
        ON(BI(PI)<=100) CALL ERROR;
        ON(BI(PI)>=133) CALL ERROR;
        I1=BI(PI); PI=PI+1; I2=0;
        ON(BI(PI)>=60) ON(BI(PI)<=71)I2=BI(PI); PI=PI+1;
        .PACK(I1, I2: I1);
        END;

ST1:    SUB;
 S11:   .POP(OP);
        ON(OP="=") CALL ERROR;
        ON(OP\="(") BO(PO)=OP; PO=PO+1; GOTO S11;
        END;

ST2:    SUB;
 S21:   .POP(OP);
        ON(BI(PI)="↑") ON(OP="↑") BO(PO)=OP; PO=PO+1; GOTO S21;
        ON(BI(PI)="*", "/") ON(OP="↑", "*", "/") BO(PO)=OP; PO=PO+1; GOTO S21;
        ON(BI(PI)="+", "−") ON(OP\="(", "=") BO(PO)=OP; PO=PO+1; GOTO S21;
        .PUSH(OP, BI(PI)); PI=PI+1;
        END;

ST3:    SUB;
 S31:   .POP(OP);
        ON(OP="(") CALL ERROR;
        BO(PO)=OP; PO=PO+1;
        ON(OP\="=") GOTO S31;
        BO(PO)=15;
        END;

STOP;
```

```
1111
36

OUTPUT FORM = 1



.LOC 642
; ARRAY ... 2
000400  ;  BI    :  −200
000521  ;  BO    :  −177

; CONST ... 26
000011  ;  −176
000010  ;  −175
000177  ;  −174
000124  ;  −173
000012  ;  −172
000120  ;  −171
000000  ;  −170
000040  ;  −167
000001  ;  −166
000015  ;  −165
000075  ;  −164
000050  ;  −163
000051  ;  −162
000136  ;  −161
000052  ;  −160
000057  ;  −157
000053  ;  −156
000055  ;  −155
000100  ;  −154
000133  ;  −153
000060  ;  −152
000071  ;  −151

; S. VAR ... 6
000000  ;  PI    :  −150
000000  ;  PO    :  −147
000000  ;  I1    :  −146
000000  ;  I2    :  −145
000000  ;  DV    :  −144
000000  ;  OP    :  −143



001042
L.0:
 LDA    2, −1, 1

 JSR    @.+1
  INIT
M01:
 JSR    @.+1
  READ
 JSR    @.+1
  CONV
 JSR    @.+1
  WRIT
 JMP    @.+1
  M01
INIT:
 STA    3, @44
 DSZ    44
IT1:
 LDA    0, −176, 2   ; 000011
 JSR    @47
   3
   3
   4
   047111  ;  IN
   052520  ;  PU
   020124  ;  T
   042504  ;  DE
   044526  ;  VI
   042503  ;  CE
   027040  ;
   027056  ;  .
   052040  ;
   054524  ;  TY
   052050  ;  (T
   020051  ;  )
   051117  ;  OR
   050040  ;
   051124  ;  TR
   050050  ;  (P
   020051  ;  )
   020077  ;  ?
   020040  ;
   000040  ;
   0
LDA  0, −175, 2   ; 000010
JSR  @46
   1
  −144     ;  DV
   0
 LDA    0, −144, 2   ;  DV
 LDA    1, −174, 2   ;  000177
 AND    1,0
 STA    0, −144, 2   ;  DV
 LDA    0, −176, 2   ;  000011
 JSR    @47
  1
 −144   ; DV
  0
 LDA    0, −144, 2   ;  DV
 LDA    1, −173, 2   ;  000124
 SUBZ# 0, 1 SZR
 JMP    L.1
 LDA    0, −175, 2   ;  000010
 STA    0, −144, 2   ;  DV
 JMP    @.+1
  IT2
L.1:
 LDA    0, −144, 2   ;  DV
 LDA    1, −171, 2   ;  000120
 SUBZ# 0, 1 SZR
 JMP    L.2
 LDA    0, −172, 2   ;  000012
 STA    0, −144, 2   ;  DV
 JMP    @.+1
  IT2
L.2:
 JMP    @.+1
  IT1
IT2:
 LDA    0, −176, 2   ;  000011
 JSR    @47
  3
  3
  3
  0
 ISZ    44
 LDA    3, @44
 JMP    1, 3
READ:
 STA    3, @44
 DSZ    44
 LDA    0, −170, 2   ;  000000
 STA    0, −150, 2   ;  PI
 LDA    0, −170, 2   ;  000000
 LDA    3, −200, 2   ;  BI
 STA    0, 0, 3      ;  000000
RD1:
 LDA    0, −144, 2   ;  DV
 JSR    @46
  2
 −200   ; BI
 −150   ; PI
  0
 LDA    3, −200, 2   ;  BI
 LDA    1, −150, 2   ;  PI
 ADD    1, 3
 LDA    0, 0, 3
 LDA    1, −174, 2   ;  000177
 AND    1, 0
 LDA    3, −200, 2   ;  BI
 LDA    1, −150, 2   ;  PI
 ADD    1, 3
 STA    0, 0, 3
 LDA    0, −176, 2   ;  000011
 JSR    @47
  2
 −200   ; BI
 −150   ; PI
  0
 LDA    3, −200, 2   ;  BI
 LDA    1, −150, 2   ;  PI
 ADD    1, 3
 LDA    0, 0, 3
 LDA    1, −170, 2   ;  000000
 SUBZ# 0, 1 SNR
 JMP    L.3
 LDA    1, −172, 2   ;  000012
 SUBZ# 0, 1 SNR
 JMP    L.3
 LDA    1, −167, 2   ;  000040
 SUBZ# 0, 1 SNR
 JMP    L.3
 LDA    1, −174, 2   ;  000177
 SUBZ# 0, 1 SNR
 JMP    L.3
 LDA    0, −150, 2   ;  PI
 INC    0, 0
 STA    0, −150, 2   ;  PI
L.3:
 LDA    3, −200, 2   ;  BI
 LDA    1, −150, 2   ;  PI
 ADD    1, 3
 LDA    0, 0, 3
 LDA    1, −165, 2   ;  000015
 SUBZ# 0, 1 SNR
 JMP    L.4
 JMP    @.+1
  RD1
L.4:
 ISZ    44
 LDA    3, @44
 JMP    1, 3
```