2014-1-(6): Manuscript for presentation at IPSJ-SIGPRO, 20 Jun 2014.

Esoteric Programming Language Malbolge and Its Low-Level Assembler

Masahiko SAKAI[1,a)] Tatsuki KATO[1,b)]

Abstract: Malbolge is known as one of the most esoteric languages, in which programming is also very difficult. The difficulty comes from (a) its restrictive instructions, (b) an instruction-replacement after each execution and (c) a restriction on loadable data. In this presentation, we overview Malbolge language and our results to produce its programs.

Keywords: Obfuscation, Software protection.

1. Introduction

Improving readability, describability and re-usability of programs are central concerns on developing programming languages. However a completely opposite direction may also be interesting and valuable. For example, unreadable programs are strong against alternations. Suppose Alice wants to send a program (or a binary code) to Bob, who executes the program, in secure. Since an encrypted program needs to be decrypted before its execution, Bob have a chance to analyze and/or alter it. On the other hand, an unreadable program acts as encrypted one, and is executable without decryption, which may give a great benefit. Malbolge is expected to be stronger than ones by other obfuscation techniques like code swapping, unnecessary code insertion, and so on. A possible drawback is the execution traces are easily obtained as well as other obfuscated programs, since Malbolge programs are able to be executed by the interpreter.

Some of obfuscated programming languages are designed as a joke; INTERCAL defined by Don Woods and James Lyon in 1972, Brainf*ck by Urban Müller in 1993, Befunge-98 by Chris Pressey in 1998 and Malbolge by Ben Olmstead in 1998. Malbolge [5] is known as one of the most esoteric programming languages. It is even called that "A programming language come from HELL". The difficulty of Malbolge mainly comes from (a) restrictive instructions, (b) an instruction-replacement after each execution and (c) a restriction on loadable data. After Anrew Cooke's "HEllo WORld" program [7] are announced, few programs are developed; three programs that output strings by Anthony Youhas in 2000, a method to produce programs from a given string that output the string by Tomasz Wegrzanowski in 2004, and a program that copy input to output by Lou Scheffer [14]. Our group developed a program that output the lyrics "99 bottles of beer" [1], [9].

This paper overviews the language definition of Malbolge and its low-level assembler as programming techniques [8], [10], [11].

2. Malbolge

We use *characters* that are identified with numbers from 0 to 255 by the well-known ASCII table. A character c is *printable* if $33 \leqslant c \leqslant 126$; otherwise *unprintable*. A *string* is a finite sequence $s_0 s_1 \cdots s_{n-1}$ of n characters s_i $(1 \leqslant i < n)$. Here we say that each character s_i *occurs at position* i.

Ternary is the base-3 numeral system. We abbreviate ternary digits as *trits*. A ternary number is denoted by the number followed by `t`. For example, 11`t` represents a ternary number 11, which is equal to 4.

We follow the interpreter [6] for the syntax and semantics of Malbolge. We start from its syntax. Programs are strings of printable characters, where space characters that recognized by `isspace` function in C library are ignored. Each character c at position i in programs must satisfy that `xlat1`$((c - 33 + i)$ `mod` 94 is one of *instructions*, which are characters `j`, `i`, `*`, `p`, `<`, `/`, `v` and `o`. Note that `xlat1` is a permutation function $\{0, \ldots, 93\} \to \{33, \ldots, 126\}$ defined in the interpreter.

Example 1 The following "Hello World!" program is known [15]:

```
(=<`#9]~6ZY32Vx/4Rs+0No-&Jk)"Fh}
|Bcy?`=*z]Kw%oG4UUS0/@-ejc(:'8dc
```

The semantics of Malbolge is described by a ternary virtual machine. The machine has a ten-trits address area and its one-word consists of ten trits, where ten trits can represent numbers 0 to $3^{10} - 1$. Codes and data are stored in the memory area. The machine has three registers; accumulator A, code pointer C and data pointer D. The function `crz`(X, Y) is a trit-wise function induced by the trit function `crz` in Table 1.

[1] Nagoya University, Furo-cho, Chikusa-ku, Nagoya 4648603, Japan
[a)] sakai@is.nagoya-u.ac.jp
[b)] tatsuki@trs.cm.is.nagoya-u.ac.jp

Table 1 Trit-wise function `crz`(x,y)

$y\backslash x$	0	1	2
0	1	0	0
1	1	0	2
2	2	2	1

We use mnemonics for instructions summarized in Table 2. Here $[n]$ is a notation for the indirect referencing that accesses the value on a memory address n. The notation $A, B := C$ is used to show that the value C is stored into both A and B. The following remarks are worth to say:

Table 2 Summary of instructions of Malbolge

instruction	mnemonic	action
`j`	`MovD`	`D := [D]`
`i`	`Jmp`	`C := [D]`
`*`	`Rot`	`A,[D] := rotr([D])`
`p`	`Opr`	`A,[D] := crz(A,[D])`
`<`	`Out`	`putchar(A)`
`/`	`In`	`A := getchar()`
`v`	`Hlt`	halt
`o`	`Nop`	no operation
other printable	`Nop`	no operation

Here, the function `rotr` circularly shifts trit value to right.

- Program is loaded on the memory from the address 0 in the increasing order. Rests of the memory unassigned by program are initialized by $[i] := \texttt{crz}([i-1], [i-2])$ like Fibonacci number.
- All registers are initialized to zero before executions. Registers C and D are both incremented by one after every step execution. Note that the instruction on the address $n+1$ is executed after execution of `Jmp` with setting an address n to C register, because of the incrementation of C.
- The character pointed by C register is replaced after an execution before the incrementation of C. The replacement is done according to a conversion table `xlat2` defined in the interpreter. We call this a *character replacement*.
  The character replacement by `xlat2` has a cycle as shown in Table 3. Thus any printable character will be recovered after some steps of executions.
- Incorrect programs are not loadable. Even if it contains a single character that does not correspond to any instructions, it is not loadable. Once loaded, a non-instruction, which caused by a computation, is executed as `Nop` if it is printable; the other (unprintable) characters cause an infinite loop in the interpreter.
  There exists a character that always works as `Nop`, i.e., is stable under the character replacement for each address. We call it `SNop`.

Table 3 The cycle of the character replacement

cycle	characters
2	F → J → F
4	* → r → } → i → *
5	) → f → ‘ → < → 3 →)
6	% → g → u → o → x → : → %
9	2 → P → B → > → L → O → C → U → I → 2
68	! → 5 → - → w → N → · · · → & → q → D → !

Example 2 The few-steps execution of a “Hello World!” program in Example 1 is shown in Table 4. The first step loads 40 into D register. The second step executes `Opr`. Since the value on the address pointed by D is $10110t$, the result $1111111111t$ of $\texttt{crz}(0, 10110t)$ is stored to both A register and the address 41. The execution continues and so on.

3. Low level assembly language

A *Low level assembly language* (L-ass, for short) is designed for preventing character-replacement and writing programs with fixed number of loops in Malbolge [8], [10], [11]. This section explains the language for the on-line assembler [13]. Let’s have a look of an example program shown in Figure 1.

Example 3 This L-ass program outputs a single ’K’ when executed. The effective Malbolge instructions executed in outputting a ’K’ are shown as follows:

```
 1  PROGRAM_START_TO ENTRY@main
 2
 3  FLAG[0/2] FF1
 4
 5  ROUTINE main {
 6  ENTRY:   ROT CON2
 7  CON2:    2222222222t
 8           REV ROT
 9           OPR K
10  K:       0000002210t
11           REV OPR
12           IF FF1
13           BRANCH ENTRY
14           OUTPUT
15           DUP
16           REV OUTPUT
17           END
18  }
```

Fig. 1 An example of low level assembly program

(1) $A, [\texttt{CON2}] := \texttt{rotr}([\texttt{CON2}])$ by the L-ass instruction `ROT CON2` in the line 6. The value `2222222222t` is loaded into A as the result.
(2) $A, [\texttt{K}] := \texttt{crz}(A, [\texttt{K}])$ by the L-ass instruction `OPR K` in the line 9. The value `1111112201t` is stored into A and the address labeled `K`.
(3) The same as (1).
(4) $A, [\texttt{K}] := \texttt{crz}(A, [\texttt{K}])$ by the L-ass instruction `OPR K` in the line 9. The value `0000002210t` is stored into A and the address labeled `K`, which is the character ‘K’.
(5) Output A by `OUTPUT` in the line 14.

The more explanation of the each L-ass instruction is explained later in Example 4.

The syntax is basically defined as a sequence of (possibly labeled) operations; `ROT`, `REV ROT`, `OPR`, `REV OPR`, `SKIP`, and so on, and flags. Some of operations has an operand that represents a variable location. The syntax is displayed in Figure 2 by an extended BNF, where non-terminals repeating one or more times are suffixed with +. ⟨name⟩ is a sequence of alpha, numeric, or _ symbols that begins from an alpha symbol. ⟨num⟩ is a decimal number or ternary number, where the latter number is followed by `t`.

Table 4 Few-steps execution of "Hello World!" program in Example 1

	initial data in memory						execution trace		
adr.	trit	dec.	chr.	inst.	mnem.	adr.	A	C	D
0	0000001111t	40	(	j	MovD	0	0000000000t	0	0
1	0000002021t	61	=	p	Opr	1	0000000000t	1	41
2	0000002020t	60	<	p	Opr	2	1111111111t	2	42
3	0000010120t	96	‘	<	Out	3	0000002200t	3	43
41	0000010110t	93	]						
42	0000002210t	75	K						
43	0000011102t	119	w						

```
⟨program⟩  ::=  ⟨prg⟩ ⟨flags⟩+ ⟨routine⟩+
⟨prg⟩      ::=  PROGRAM_START_TO ⟨label⟩ ↵
⟨flag⟩     ::=  FLAG [ ⟨num⟩ / ⟨num⟩ ] ⟨name⟩ ; ↵
⟨routine⟩  ::=  ROUTINE ⟨name⟩ { ↵ ⟨stm⟩+ } ↵
⟨stm⟩      ::=  ⟨labeldef⟩ : ↵
              | ⟨inst⟩ ↵
              | ⟨labeldef⟩ : ⟨inst⟩ ↵
⟨inst⟩     ::=  ⟨num⟩
              | DUP
              | ROT ⟨slabel⟩ | REV ROT
              | OPR ⟨slabel⟩ | REV OPR
              | JMP ⟨label⟩ | REV JMP
              | OUTPUT | REV OUTPUT
              | INPUT | REV INPUT
              | SKIP ⟨slabel⟩
              | IF ⟨name⟩ ↵ BRANCH ⟨label⟩
              | NEXT ⟨name⟩
              | END
⟨label⟩    ::=  ⟨slabel⟩
              | ⟨slabel⟩ @ ⟨name⟩
⟨slabel⟩   ::=  ⟨name⟩
```

Fig. 2 Syntax of low-level assembly language

The semantics is defined as a ternary virtual machine similarly to Malbolge. Main features are described as follows:

(1) Registers are an accumulator A and a program counter PC.
(2) The execution will start from the address specified by `PROGRAM_START_TO`.
(3) After execution of one of operators, `ROT`, `OPR`, `JMP`, `OUTPUT`, `INPUT`, which corresponds to an instruction of Malbolge, we have to execute the corresponding `REV` operator, called *restore* instructions, only once.
(4) Each label described in an operand of instructions must be placed below the operation apart at most the fixed number k. Moreover all operations between the executed operation and the corresponding variable will not be executed and just skipped.
(5) A constant is placed by ⟨num⟩ or `DUP`, where `DUP` declares the constant value unspecified.
(6) "`FLAG` $[n/m]$ ⟨name⟩" declares the named flag, which has an initial state n, where the possible states are $0, \ldots, m-1$ determined by a cycle $m \in \{2, 4, 5, 6, 9\}$. We write |flag| to denote its state. "`IF` ⟨name⟩ BRANCH ⟨label⟩" jumps to the label if the specified flag has the state 0. The state is always incremented by 1 modulo m whenever executed. "`NEXT` ⟨name⟩" increments the state of the specified flag.
(7) `INPUT` inputs a character and store to the accumulator A, and `OUTPUT` outputs A. Note that for these instructions, the program counter PC is added by 2. Normally we fill `DUP` just below the instructions.

The meaning of the instruction is found in Table 2. From the restriction on the control stated in (4), the programming is not still easy and needs help of flags.

Example 4 We give additional comments for the program in Figure 1 explained in Example 3.

- The line 1 shows that the program will be executed from the line 6 labeled by `ENTRY` in a module `main`. Note that modules defined by `ROUTINE` is only for preventing name collisions. The low-level assembler has no ‘call-return’ functions.
- The line 3 defines a flag named `FF1` with cycle 2 and initial state 0.
- Instructions in line 6, 8, 9, 11, 12, 13, 6, 8, 9, 11, 12, 14, and 16 are executed in this order.
- `REV ROT` in the line 8 (resp. `REV OPR` in the line 11, and `REV OUTPUT` in the 16) is executed to restore `ROT` (resp. `OPR`, and `OUTPUT`) instruction.
- In the first execution of `IF FF1 BRANCH ENTRY` in the lines 12 and 13, it jumps to the line 6 because the flag `FF1` has the state 0. In the second execution, it goes to the line 14 because the state is now 1.

Suppose we want to write a code for `ROT X` and `OPR X`. The code (a) in Fig.3 does not work as expected, and `OPR X` is not executed. Instead, we should write as (b) where `ROT X` is executed, goes to `L1`, and `REV ROT`, `OPR X`, and `REV OPR` are executed in this order.

```
PROGRAM_START_TO ENTRY@main
ROUTINE main {
ENTRY:  ROT X
        REV ROT
        OPR X
X:      10110t
        REV OPR
        END
}
```

(a) An L-ass program that fails to execute `OPR X`

```
PROGRAM_START_TO ENTRY@main
FLAG    [0/2] FF1
ROUTINE main {
ENTRY:  ROT X
L1:     REV ROT
        OPR X
X:      10110t
        IF FF1
        BRANCH L1
        REV OPR
        END
}
```

(b) A correct L-ass program

Fig. 3 An example of low level assembly program

One of important notes is the absence of conditional

Table 5 Instructions of Low-level assembly language

Instructions	Action	Restriction
`ROT` label	A, $[label]$:= $\texttt{rotr}([label])$, PC:= $label + 1$	PC $< label <$ PC+k
`OPR` label	A, $[label]$:= $\texttt{crz}(A, [label])$, PC:= $label + 1$	PC $< label <$ PC+k
`JMP` label	PC:= $[label]$	
`OUTPUT`	$\texttt{putchar}(A)$, PC:= $label + 2$	
`INPUT`	A:=`getchar`, PC:= $label + 2$	
`REV` inst	Restoring instructions above, PC:= $label + 1$	
`SKIP` label	PC:= $label$	PC $< label <$ PC+k
`IF` flag `BRANCH` label	If $\|$flag$\|$ = 0 then PC:= $[label]$ else PC:=PC+2 $\|$flag$\|$:= $\|$flag$\|$ + 1	
`NEXT` flag	$\|$flag$\|$:= $\|$flag$\|$ + 1 PC:=PC+1	
`END`	Stop	

(1)	Initial data area (Addr. 36 – 83)
(2)	Code for complete (1) (Addr. 100 – 864)
(3)	Code for constructing the image (4) (Addr. 865 –)
(4)	Image of L-ass program

Fig. 4 Memory map for L-ass

branch according to the value. In order to do this, a special technique is necessary, which is omitted in this paper.

Most part of the real loop version of 99 bottles of beer program [9] was programmed in the low level assembly language.

4. Low level assembler

The low-level assembler firstly construct a memory image of Malbolge virtual machine from a given program ((4) in Figure 4), and secondly constructs a Malbolge program that produces the image ((1)–(3)), where Figure 4 demonstrates the memory map of Malbolge virtual machine.

The most difficulty in implementing a low-level assembler lies on the data construction in the second step. This difficulty comes from the restriction that non-instructions are not allowed to load. That's why we must construct a (native) Malbolge program to produce data and expand them to right places. We leave this issue in Section 5

In the rest of this section, we explain how a low level assembly program works as Malbolge program. The register PC of L-ass machine is implemented by the register D of Malbolge machine. The key is found in introducing instruction units.

An instruction unit, called OPR unit shown in Figure 5 (a) is a sequence of Malbolge instructions. Figure 5 (b) corresponds to the line 11 to 14 of the L-ass program in Figure 1. For an `OPR` instruction, the address 85 of the `OPR` unit is placed on the address 100.

The execution of codes are shown as follows from initial register values with C = 153, D = 100 and A = `2222222222t`.

(1) The register C is set to 85 by executing `Jmp` on the address 153.

(2) After the registers C and D are both incremented, `Opr` instruction on the address 86 is executed and hence the result 1120`t` is stored to the register A and the location 101.

(3) After the registers C and D are both incremented, the register C is set to 86 by executing `Jmp` on the address 87.

(4) After the registers C and D are both incremented, the register C is set to 69 by executing `Jmp` on the address 87.

	adr	label	instruction
	82		`SNop`
	83		`SNop`
	85	`OPR`	`SNop`
	86	`REV_OPR`	`Opr`
	87	–	`Jmp`
	⋮	⋮	⋮
$C \to$	70		`Jmp`

(a) OPR unit

	adr	label	data
$D \to$	100	–	`OPR` (=85)
	101	`K`	`2210t`
	102	–	`REV_OPR` (=86)
	103	–	`FF1` (=63)
	103	–	`ENTRY` (=97)
	103	–	`OUTPUT` (=118)

(b) L-ass code

Fig. 5 An execution with the OPR unit

In this way, the low-level assembly program is executed, where the executed instructions of Malbolge are replaced with the other instructions. By the execution of `Jmp` in (1), `SNop` on the address 81 is replaced, which has no effect because `SNop` is stable under the replacement. By the execution of `Opr` in (2), `Opr` on the address 82 is replaced. By the execution of `Jmp` in (3), the resulted instruction of (2) is replaced again. If the `Opr` has a cycle 2, `Opr` is successfully restored. We write such instruction as `Opr/Nop`.

Other instruction units `ROT`, `JMP`, `OUTPUT`, and `INPUT` are similarly implemented. Next we consider a fixed number of a loop structure. The L-ass program in Figure 1 has instructions `IF` and `BRANCH`. This is implemented by a *flag* unit. A flag unit is the same as `OPR` unit except for a Malbolge instruction `Mov_d` is used instead of `Opr`.

```
FF1:        SNop
NEXT_FF1:   Mov_d/Nop
            Jmp
```

The address `FF1` in the flag unit is placed as an implementation of `IF FF1`, and the address `ENTRY` − 1 is placed

as an implementation of BRANCH ENTRY. Executing Mov_d in the flag unit, the address that corresponds to ENTRY is set to the register D, which means a jump to the address for L-ass instructions. Since the Mov_d instruction is replaced with Nop, in the second execution the jump is not effective. This is captured as the jump controlled by a flip-flop. Remark that the cycle of a Malbolge instruction like Mov_d depends on the address where the instruction is located, where several cycles $2, 4, 5, 6$, and 9 are possible as shown in Figure 3. For example, there exists a Mov_d with a cycle 5 works as Nop except for once in 5 times of executions.

We must state the execution mechanism for the case that an instruction (e.g. ROT X) is placed apart from a target location (e.g. X) as in Figure 3 (b). This is rather simple; we just put the address ROT $-$ 2 as an implementation of the L-ass instruction. The value is actually 82 for the Opr unit in Figure 5 (a). The SNop's ignore data between the instruction ROT X and the X. Note that this separation length is limited from the reason that we prepare a number of SNop's before the instruction units. This is displayed as k in Table 2.

The instruction units are summarized as Table 6, where addresses are shown with modulo 94. For flag units, the pairs (n,c) of addresses in mod 94 and the characters are (64,F), (92,*), (32,)), (3,%), and (84,2) for cycles 2, 4, 5, 6, and 9, respectively.

Table 6 Detail of instruction units

unit	address	label	instruction	character
HALT	0	END	Halt	Q
OUTPUT	24	OUTPUT	SNop	3
	25	REV_OUTPUT	Output/Nop	J
	26		Jmp	H
SKIP	⋮		⋮	
	39	SKIP	SNop	2
	40		Jmp	:
INPUT	46	INPUT	SNop	O
	47	REV_INPUT	Input/Nop	F
	48		Jmp	2
ROT	⋮		⋮	
	58	ROT	SNop	i
	59	REV_ROT	Rot/Nop	J
	60		Jmp	&
JMP	⋮		⋮	
	63	JMP	SNop	)
	64	REV_JMP	Mov_d/Nop	F
	65		Jmp	!
OPR	⋮		⋮	
	85	OPR	SNop	:
	86	REV_OPR	Opr/Nop	F
	87		Jmp	i
flag	$n-1$	flag name	SNop	c
	n	REV_OPR	Mov_d	
	$n+1$		Jmp	

5. Constructing arbitrary data in Malbolge

This section explains how to construct a Malbolge code that constructs a necessary value and places it on a specified address. The contents of this section due to Iizawa [10].

5.1 Pseudo instructions

Pseudo instructions introduced in [8] are useful to designing programs in Malbolge. Each pseudo instruction consists of mnemonic with an operand that represents a variable; for example ROT X. Converting pseudo-instruction sequence into a straight-line Malbolge program is not so difficult by introducing a cyclic structure of data areas pointed by D-register.

For example, nondestructive load of the value of a variable X into A register is represented as a pseudo-instruction sequence shown in Fig.6 (a). The corresponding Malbolge program as a mnemonic sequence is shown in Fig.6 (b), which really works if we start with C = ENTRY and D = CON2. Note that (b) can be represented as a Malbolge program except for a special constants 2222222222t and an address CON2 - 1, which we have to prepare in advance.

```
ENTRY: Rot CON2
       Opr X
       Rot CON2
       Opr X

CON2:  2222222222t
   Y:  *
   X:  *
```

(a) Assembly code

```
ENTRY:  Rot
        Nop
        Opr
        MovD
        Rot
        Nop
        Opr
        MovD

CON2:   2222222222t
   Y:   *
   X:   *
        CON2 - 1
```

(b) Implementation

Fig. 6 nondestructive load of X

5.2 Constructing special constants

Values whose all trits are the same are convenient because they are non-destructively loadable to the accumulator by Rot. Table 7 shows pseudo instruction sequences that construct special constants 1111111111t, 0000000000t, 2222222222t and 2222222221t and that store them into locations CON1, CON0, CON2 and CON2X, respectively. Let us explain the code for 1111111111t. Assume that each trit of the value in CON1 is 0 or 2. By executing Rot CON1, the right-shifted value of CON1 is stored to both A register and the location CON1. Since $\mathtt{crz}(0,0) = 1$ and $\mathtt{crz}(2,2) = 1$, the instruction opr CON1 makes the value 1111111111t.

Actually constants 1111111111t, 0000000000t, 2222222222t and 2222222221t are possible to construct in this order by the codes. Although there are some precondition as shown in Table 7, there exists fortunately a possible initial sequence of instructions for data area in constructing these constants, where the data area is on addresses between 85 and 91 in Figure 7. Note that the address PTR is used for copying data explained later. The sequence of the instructions guarantees that C = 104, $D = 85$ in 6 steps, where the precondition for the construction of CON1 is satisfied. In order to construct constants CON0, CON1, CON2, and CON2X between addresses 85 and 90, it is enough to place codes in Table 7 in the following order

Table 7 Pseudo instruction sequences to construct some constants

Constant	Code	Precondition	Postcondition
CON1	Rot CON1 Opr CON1	• [CON1] has no 1s	• A = [CON1] = 1111111111t
CON0	Opr CON0 Opr CON0 Opr CON0	• A = 1111111111t	• A = [CON0] = 0000000000t
CON2	Rot CON1 Opr CON2 Opr CON2 Rot TMP, Opr CON2, Opt CON2X } 10 times	• [CON1] = 1111111111t. • [TMP] = $\alpha 20 \beta$t, where $\alpha, \beta \in \{0,1\}^*$	• [CON1] = 1111111111t, and A = [CON2] = [CON2X] = 2222222222t • [TMP] is preserved by the execution
CON2X	Opr CON2X	• [CON2X] = 2222222222t • $A = \alpha 2$t where $\alpha \in \{0,1\}^*$	• A = [CON2X] = 2222222221t

```
adr label mnem.  data
  0        Jmp    98
  1        Hlt    80

 85 TMP:   In     0000011200t
 86 PTR:   Nop
 87 CON0:  Nop
 88 CON1:  Nop    0000002202t
 89 CON2:  Nop
 90 CON2X: Nop
 91        Hlt    84

 99        MovD
100        Nop
101        Nop
102        Nop
103        Nop
```

Fig. 7 Construction of constants

from the address 104.
(1) CON1.
(2) CON0, and then CON1.
(3) CON2.
(4) CON2X after rotating TMP twice by

```
Rot TMP
Rot TMP
```

The area between 85 and 90 is the initial data area (1) in Figure 4. The code above that complete the data area is located on (2) in Figure 4.

5.3 Constructing arbitrary value

Now we are ready to write a code that generates an arbitrary value, where the codes (3) in Figure 4 is produced by the techniques in this subsection.

In the sequel, we assume that the locations CON0, CON1 and CON2 have the values 0000000000t, 1111111111t and 2222222222t, respectively. We also assume that the location CON2X has the value either 2222222220t or 2222222221t.

Fact 5 Let the rightmost trit of X be 1. Then after execution of the following sequence:

```
Opr X
Rot CON2
Opr X
```

the rightmost trit of X is set without changing the other trits of X as follows

0 if A = 2222222221t
1 if A = 2222222222t
2 if A = 2222222220t

Proof. The lemma is easily proved from the following trit function $\mathtt{crz}(2, \mathtt{crz}(a, x))$:

$x \backslash a$	0	1	2
0	2	0	0
1	2	0	1
2	1	1	2

□

Fact 6 Arbitrary value can be generated

Proof. The value 2222222220t is loadable to register A by

```
Rot CON1
Opr CON2X
```

The value 2222222221t is also loadable to register A by

```
Rot CON0
Opr CON2X
```

where we assume the location CON2X has the value either 2222222220t or 2222222221t. Note that the assumption is preserved by the execution of the codes. From these results, we can generate an arbitrary value as follows:

(1) Store the constant 1111111111t into TMP by

```
Rot CON1
Opr TMP
Opr TMP
```

(2) Load one of the constants 2222222221t, 2222222222t and 2222222220t into A and apply the code in Fact 5.
(3) Execute rot TMP.
(4) Repeat the above (1–3) in 10 times. □

The procedure in the proof of Fact 5 can be implemented by using data area from 85 to 90 in Figure 7.

Suppose we generated a value on location TMP. In order to copy the value on TMP into an intended location L, we use the code for "nondestructive copy" in Table 8. However there is a problem because the location L has to be in the same data loop according to the method for repeated access of memory in Subsection 5.1.

If there exists a location M ($> L$) having a value N (< 85) then the following is a solution of the above problem, which construct such a loop (Figure 8):

(1) Generate a value L' ($\leqslant L$) on location PTR.

Table 8 Nondestructive copy

Name	Code	Explanation

Nondestructive data copy	`Rot CON1`
	`Opr Z`
	`Opr Z`
	`Opr Y`
	`Opr Y`
	`Rot CON2`
	`Opr TMP`
	`Rot CON2`
	`Opr TMP`
	`Opr Z`
	`Opr Y`
		• `Z` is used as temporal area
		• `[Y] := [TMP]`

(2) Execute `MovD` at $D =$ `PTR`, $D = M$ and $D = 91$ in this order.

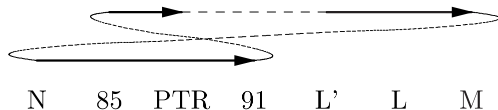

Fig. 8 Loop of register D for data copy

The assumption on M is not hard because every program code is printable character, that is between 33 and 126. If M is near with L and $L' = L$ then we can shorten the data cycle hence the length of the codes. For preparing value N in many locations, we can use the memory initialization property, where the memory area unassigned by programs are initialized like Fibonacci number stated in Section 2. Actually the low-level assembler put the character sequence `RQ` as the last two instructions at one of addresses shown in Table 9, where `R` $= 82 = 0000010001t$ and `Q` $= 81 = 0000010000t$. If the program end with one of those sequences, values in addresses after the program are set to the alternation of $29443 (= 1111101111t)$ and $81 (= 0000010000t)$.

Table 9 Last two instructions

Character	possible locations in modulo 94							
`R`	16	17	35	51	52	74	80	93
`Q`	17	18	36	52	53	75	81	94

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Number 26540027.

References

[1] 99 bottles of beer, `http://99-bottles-of-beer.net/`.

[2] Satoshi Ando, Masahiko Sakai, Toshiki Sakabe, Keiichirou Kusakari, and Naoki Nishida, *Introducing array mechanism into high-level assembly language for Malbolge*, Technical report of IEICE, Vol.112, No.23, pp.43–48 (2012, in Japanese).

[3] Satoshi Ando, Masahiko Sakai, Toshiki Sakabe, Keiichirou Kusakari, and Naoki Nishida, *A SAT encoding for finding operation sequences of Malbolge that implement trit-wise functions*, Technical report of IEICE, Vol.112, No.275, pp.7–12 (2012, in Japanese).

[4] Satoshi Ando, Masahiko Sakai, Toshiki Sakabe, Keiichirou Kusakari, and Naoki Nishida, *Using SAT solvers for solving control-instruction layout problems in low-level assembly programming for Malbolge*, Technical report of IEICE, Vol.112, No.373, pp.25–30 (2013, in Japanese).

[5] Esolang: *Malbolge*, `http://esolangs.org/wiki/Malbolge` (1998).

[6] Ben Olmstead: *Source program of Malbolge interpreter*, `http://esoteric.sange.fi/orphaned/malbolge/` (1998).

[7] Andrew Cooke: *Malbolge: hello world*, `http://www.acooke.org/andrew/writing/malbolge.html`.

[8] Hisashi Iizawa, Toshiki Sakabe, Masahiko Sakai, Keiichirou Kusakari, and Naoki Nishida, *Programming Method in Obfuscated Language Malbolge*, Technical Report of IEICE, Vol.105, No.129, SS2005-22, pp.25–30 (2005, in Japanese).

[9] Hisashi Iizawa, *Real Loop Version of 99 Bottles of Beer in Malbolge*, `http://99-bottles-of-beer.net/language-malbolge-995.html` (2005).

[10] Hisashi Iizawa: *Study on program obfuscation based on esoteric language Malbolge*, Master thesis, Grad. School of Information Science, Nagoya University (2006, in Japanese).

[11] Satoshi Nagasaka, Masahiko Sakai, Toshiki Sakabe, Keiichirou Kusakari, and Naoki Nishida, *On Turing completeness of an esoteric language, Malbolge*, Technical report of IEICE, Vol.110, No.227, SS2010-37, pp.55–60 (2010, in Japanese).

[12] Masahiko Sakai, *Introduction to esoteric language Malbolge*, Japan-Vietnam Workshop on Software Engineering 2010 (JVSE 2010), Hanoi, pp.15–19.

[13] *On-line low-level assembler of Malblge*, `http://www.trs.cm.is.nagoya-u.ac.jp/Malboge/on-lal.html`

[14] Lou Scheffer: *Introduction to Malbolge*, `http://www.lscheffer.com/malbolge.html`.

[15] Wikipedia: *Malbolge*, `http://en.wikipedia.org/wiki/Malbolge`.