Introduction to Esoteric Language Malbolge

Masahiko Sakai

Graduate School of Information Science, Nagoya University,
furo-cho Chikusa-ku, Nagoya 4648603 Japan

1 Introduction

Improving readability, describability and re-usability of programs are one of central concerns on developing programming languages. However a completely opposite direction may also be interesting and valuable. For example, unreadable programs are strong against alternation. Suppose Alice wants to send a program (or a binary code) to Bob who executes the program. Even if it is encrypted, Bob must have authorization to decrypt it in order to execute. Thus Bob has a chance to alter it. On the other hand, an unreadable program acts as encrypted one and also executable without decryption, which is a great benefit.

Obfuscated program languages are proposed as jokes; 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 is known as one of the most esoteric programming languages. It is even called as "A programming language come from HELL". The difficulty of Malbolge mainly comes from (a) restrictive instructions, (b) instruction-replacement after execution and (c) restriction on loadable data. After Anrew Cooke's "HEllo WORld" program [4], there are few programs; 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 [9]. Our group developed a program that output the lyrics "99 bottles of beer" [1, 6].

This paper overviews the language definition [2, 3] and programming techniques [5, 7, 8].

2 Malbolge

We use *characters* corresponding to numbers from 0 to 255 by ASCII table. A character is *printable* if its corresponding number c satisfies $33 \leq c \leq 126$; otherwise *unprintable*. A *string* is a finite sequence $s_0 s_1 \cdots s_{n-1}$ of n characters $s_0, s_1, \ldots, s_{n-1}$. 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, `11t` represents a ternary number 11, which is equal to 4.

The syntax and semantics of Malbolge is defined by an interpreter [3] written in C. The simple document [2] attached with the interpreter package is not complete and also has a slight mismatch with the interpreter. Thus it is reasonable to follow the interpreter.

We start from its syntax. *Instructions* are characters `j`, `i`, `*`, `p`, `<`, `/`, `v` and `o`. Programs are strings of printable characters where space characters that recognized by `isspace` function in C library are ignored. Each character `x` at position `i` in programs must satisfy the property that `xlat1(x,i mod 94)` is one of instructions, where `xlat1` is a permutation function defined in the interpreter.

The semantics of Malbolge is described by a ternary virtual machine. The machine has ten-trits address area and its one-word consists also ten trits, where ten trits can represent numbers 0 to $3^{10} - 1 = 59048$. The code 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 derived by the following trit function `crz`:

x	0 1 2 0 1 2 0 1 2
y	0 0 0 1 1 1 2 2 2
crz(x,y)	1 0 0 1 0 2 2 2 1

.

We use mnemonics for instructions as summarized in Table 1. Here $[n]$ is a notation that represents the data in the memory address n. The following

Table 1. Summary of instructions

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.

remarks are worth to say:

- Program is loaded into `[0]`,`[1]`,... (the memories from address zero) in this order. Rest of memories unassigned by program are initialized by `[i] := crz([i-1],[i-2])` like Fibonacci number.
- The character pointed by C register is replaced after execution by the character according to a conversion table `xlat2` defined in the interpreter. We call this *character replacement*.
- All registers are initialized to zero before executions. Registers C and D are both incremented by one after step execution. Note that in order to jump

by `Jmp`, we need to set to C register one smaller address of the location of the next instruction to execute.
– Programs consist of characters corresponding to instructions and non-programs are not loadable. Every printable character corresponding to non-instructions may be produced during execution, and is interpreted as `Nop`. On the other hand, an execution of an unprintable character causes an infinite loop.

3 Overview of intermediate languages

3.1 Constructing bootstrap codes

Pseudo instructions introduced in [5] are useful to designing non-loop programs. Each pseudo instruction consists of mnemonic with an operand that represents a variable; for example `ROT X`. Converting pseudo-instruction sequence into non-loop Malbolge program is not so difficult by introducing a cyclic structure of a data module pointed by D-register, and uses `Nop` instructions.

By using the above programming structure, some operation becomes possible; zero clear, nondestructive data copy, successor function and construct some special constants. From these techniques, any values of data are possible to construct, which is very important because the only few kinds of data are loadable in Malbolge. For example nondestructive data load of a variable X into A register is represented as the pseudo-instruction sequence shown in Fig.1 (a). The corresponding Malbolge program as mnemonic sequence is shown in Fig.1 (b), where label definitions like `CON2:` denote the memory address. Note that (b) is not corresponds to a Malbolge program and we have to prepare codes that constructs data `2222222222t` and `CON2 - 1`.

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

CON2:  2222222222t
   X:  *
```

(a) Pseudo instructions

```
ENTRY:  Rot
        Opr
        MovD
        Rot
        Opr
        MovD

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

(b) Malbolge implementation

Fig. 1. nondestructive load of `X`

3.2 Low level assembly language

Low level assembly language is designed for preventing charactor-replacement and writing loop programs in Malbolge [5, 7, 8]. The syntax is similar to pseudo

instruction sequences and defined as a labelled sequence of operations; `U_JMP`, `U_ROT`, `R_ROT`, `U_MOV_PC`, `R_MOV_PC`, `U_INPUT` and so on, and flags. Some operations have an operand representing a variable like pseudo instructions.

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

(1) Registers are accumulator A and program counter PC.
(2) After execution of an operator with `U_`, which corresponds to an instruction of Malbolge, we have to execute the corresponding operator with `R_`.
(3) Each variable described in an operand must be placed below and near the operation. Moreover all operations between the executed operation and the corresponding variable are skipped.
(4) Flags act as flip-flopped `U_MOV_PC`.

From the restriction on the control stated in (3), the programming is not still easy and needs help of flags. For example, an implementation of the following pseudo instruction sequence is shown in Fig.2.

```
ENTRY:  Rot X
        Opr X

X:      30527
```

(a) Pseudo instructions

```
L1:     R_ROT
        U_OPR X
ENTRY:  U_ROT X
X:      30537
        FLAG1
              L1
        R_OPR
        END
```

(b) Low level assembly implementation

Fig. 2. Example of low level assembly program

The basic idea of transformation of low level assembly programs into Malbolge is as follows:

- The character-replacement after execution has a cycle whose period is 2 for some characters that depends on locations [9].
- Construct instruction units and use register D as a program counter PC. Each operation (e.g. `U_OPR`) is the address of the corresponding unit minus k, where k is the location distance between the operation and its operand variable.
- Flags are the same as `U_MOV_PC`; an unit for `MovD` mnemonic.

There are some more observations, which are valuable for designing units.

- Instruction having mnemonic `Jmp` is never replaced by execution, but instead the instruction at one earlier location of the new location is replaced.
- For every location, there exists a character-replacement sequence, each of which instruction is always `Nop`, which we write as `Nop/Nop`.

Since their constructions are very similar, we only show the `Opr`-unit

```
          Nop/Nop
U_OPR:    Opr/Nop
R_OPR:    Jmp
```

written in Malbolge mnemonic, where `Opr/Nop` represents `Opr` instruction with cycle two. Similarly, `Nop/Opr` represents `Nop` obtained by a execution of `Opr/Nop`. Other units are constructed similarly to `Opr`-unit.

The real loop version of 99 bottles of beer program [6] was programmed in the low level assembly language.

3.3 High level assembly language

High level assembly language is designed for relaxing restrictions of the low level one [7, 8]. The operations are `INC X`, `DEC X`, `MOV X,Y`, `BRANCH X,Y`, `INPUT X`, `OUTPUT X` and `STOP`, where `X` and `Y` are labels of variables or operations. For example the meaning of `BRANCH X,Y` is that jump to the address labeled by `Y` if the value of variable `X` is equal to 0; go the next operation otherwise.

The idea for translating the high level one into the low level one is as follows:

- Each operation is transformed into a sequence of addresses, that is a sequence of labels of low level one.
- We prepare a module like instruction units for each operation. Each module is a low level assembly program, in which it copies the address of `X` (and `Y`) into internal variable before individual processing.

Constructing modules are not easy but this mechanism makes it possible to augment new operations by preparing modules.

References

1. 99 bottles of beer, `http://99-bottles-of-beer.net/`.
2. Ben Olmstead: Malbolge, `http://www.antwon.com/other/malbolge/malbolge.txt`, 1998.
3. Ben Olmstead: Source program of Malbolge interpreter, `http://www.lscheffer.com/malbolge_interp.html`, 1998.
4. Andrew Cooke: Malbolge: hello world, `http://www.acooke.org/andrew/writing/malbolge.html`.
5. Hisashi Iizawa, Toshiki Sakabe, Masahiko Sakai, Keiichirou Kusakari, Naoki Nishida: Programming Method in Obfuscated Language Malbolge, IEICE Technical Report、Vol.105, No.129, SS2005-22, pp.25–30, 2005 (in Japanese).
6. Hisashi Iizawa, Real Loop Version of 99 Bottles of Beer in Malbolge, `http://99-bottles-of-beer.net/language-malbolge-995.html`, 2005.
7. Hisashi Iizawa: Study on program obfuscation based on esoteric language Malbolge, Master thesis, Grad. School of Information Science, Nagoya University, 2006 (in Japanese).
8. Satoshi Nagasaka, Masahiko Sakai, Toshiki Sakabe, Keiichirou Kusakari, 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).
9. Lou Scheffer: Introduction to Malbolge, `http://www.lscheffer.com/malbolge.html`.