Seminar Report

Malbolge

Michael Schaper
michael.schaper@student.uibk.ac.at

19 February 2011
Supervisor: Martin Avanzini

Abstract

In this work we provide informations about the esoteric programming language Malbolge. The language is unique and designed to be as difficult as possible. We will introduce the language briefly and analyze the points that make programming in Malbolge so complicated.

Contents

1 Introduction

This work is about the *esoteric* programming language Malbolge. An esoteric programming language is a language that is not designed for a "serious" or "productive" reason. Such languages are designed to experiment with weird ideas, to be hard to program or simply as a joke. Many esoteric languages use *obfuscation* to make them difficult to program and understand. In computer science obfuscation is the translation of source or binary code into equivalent code that is hard to deconstruct. This can be used for security reasons or to prevent reverse engineering.

Ben Olmstead invented Malbolge in the year 1998. The name Malbolge comes from the word "Malebolge", which is the name of the eighth circle of hell in "Dante's Inferno". Malbolge is one of the languages that are designed to be hard to program. More precisely, Malbolge is designed to be the most difficult and incomprehensible programing language. The original website of Malbolge was removed but is still available at the web archive[1].

Olmstead was inspired by other esoteric languages, such as INTERCAL and brainfuck. We will introduce them briefly as reference languages in Section 2. In Section 3, we will give an overview about the language itself and state which concepts make programming in Malbolge so difficult. Afterwards, we will mention in Section 4 three important people who tried to "crack" the language and contributed much to the Malbolge community. We conclude in Section 5.

2 Comparison to Other Languages

Malbolge is neither comparable with traditional languages like COBOL or BASIC nor with newer languages like Python or Haskell. It does not support any well-known constructs like variables, procedure definitions or even conditions. Thus, comparison to other esoteric languages is more reasonable. We will briefly describe INTERCAL and brainfuck, because Olmstead was inspired by these languages and therefore Malbolge has much in common with them. Moreover, they also provide a good introduction to esoteric languages.

2.1 The INTERCAL Programming Language

The INTERCAL programming language [3] was invented by Don R. Woods and James M. Lyon in 1972. Officially, the intention of INTERCAL was to have a compiler language that has nothing at all in common with any other major language. Considering the year of its creation, major languages are for example FORTRAN, BASIC, COBOL, ALGOL and LISP. The language is also said to satirize various aspects, the constructs and notations, provided by these languages. Inspecting the manual, one quickly discovers that the creators have fun writing this language.

[1] B. Olmstead, Malbolge: Programming from hell, `http://web.archive.org/web/20000815230017/http:/www.mines.edu/students/b/bolmstea/malbolge`

> *The full name of the compiler is 'Compiler Language With No Pronounceable Acronym', which is, for obvious reasons, abbreviated INTERCAL* [3, p. 1].

INTERCAL is well known for its obfuscated syntax and torturous statements. For example, logic operators are unary and operate on the consecutive bits of the operand. Providing the infamous `COME FROM` statement, they make their own contribution to a world without "evil" `GOTO` statements.

To get an impression of this language we want to depict the Hello World program.

```
 1  DO ,1 <- #13
 2  PLEASE DO ,1 SUB #1 <- #238
 3  DO ,1 SUB #2 <- #108
 4  DO ,1 SUB #3 <- #112
 5  DO ,1 SUB #4 <- #0
 6  DO ,1 SUB #5 <- #64
 7  DO ,1 SUB #6 <- #194
 8  DO ,1 SUB #7 <- #48
 9  PLEASE DO ,1 SUB #8 <- #22
10  DO ,1 SUB #9 <- #248
11  DO ,1 SUB #10 <- #168
12  DO ,1 SUB #11 <- #24
13  DO ,1 SUB #12 <- #16
14  DO ,1 SUB #13 <- #162
15  PLEASE READ OUT ,1
16  PLEASE GIVE UP
```

Listing 1: C–INTERCAL:"Hello, World!".

The program depicted in Listing 1 is actually written in C–INTERCAL, a dialect of INTERCAL. The original implementation of the language can only print "butchered" Roman numerals. In line number 1 an array is created and from line number 2 to 14 single characters are assigned. Line 15 prints the assigned characters on the screen. Beside the syntax, the statement `READ OUT` makes the program really difficult, because it implements character output based on the "Turing Text" model. In particular, the relative distant to the previous assigned character has to be calculated. For more details the reader may refer to [3, p. 5].

Though it is really hard to program in INTERCAL, Olmstead states different points why it is not suitable to be the most difficult programming language. First of all, it supports different data types, variables and many instructions. Although INTERCAL's constructs may be torturous, they are way too flexible, e.g. assigning a number to a variable can be done using a single statement. Further, the compiler is also quite forgiving, simply ignoring illegal statements.

2.2 The brainfuck Programming Language

The other language we want to introduce is the esoteric language brainfuck[2], which was invented by Urban Müller in 1993. He intended to provide the smallest compiler for a Turing-complete language. The language brainfuck can be compared with a Register machine. Thus, it only supports a small set of simple operations, in numbers eight, and nothing like variables or data types. This simplicity and inflexibility surely makes programming in brainfuck not easy. Nonetheless, Olmstead claims the operations to be too intuitive. For example, brainfuck uses `>` to increment the data pointer or `+` to increment the data by one byte. This operations can be compared to C's pointer operation `++ptr` and `++*ptr`, where `ptr` is of type `unsigned char*`. Listing 2 shows the Hello World program of brainfuck.

```
++++++++++[>+++++++>++++++++++>+++>+<
<<<<-]>++.>+.+++++++..+++.>++.<<++++++
++++++++.>.+++.------.--------.>+.>.
```

Listing 2: brainfuck: "Hello World!".

After executing the first ten `+` operations, the first memory segment 0 contains the value 10. Using a loop `[ ··· ]`, the memory segments from 1 to 4 are initialized with $70, 100, 30$ and 10. Afterwards, this segments are used to set the correct ASCII values and printing it out with operation `(.)`. For example, the operations `>++.` after the loop are used to set the data pointer to memory segment 1, containing the value 70, setting it to 72 and printing it out, which yields the character `H`.

3 The Malbolge Programming Language

Olmstead stated, he wanted to borrow from both languages, eliminate their weaknesses and put them together in a unique way. From the specification:

> *It was designed to be difficult to use, and so it is. It is designed to be incomprehensible, and so it is*[3].

To get an overview of the language, we will describe it briefly. Further, we try to analyze the language and express which features make Malbolge so difficult to program. Both, the specification[4] and the interpreter[5], are available on the internet. Within the spirit of Malbolge the official specification and the reference interpreter do not match completely. In particular, the operators for I/O operations are switched. Programs are written for the interpreter normally, that's why we stick with the interpreters version within this report. Furthermore, there is also a bug in the interpreter, which we ignore for now and explain later in Section 4.

[2] brainfuck, `http://www.esolangs.org/wiki/Brainfuck`
[3] The Malbolge Specification, `http://www.lscheffer.com/malbolge_spec.shtml`
[4] *Ibid.*
[5] The Malbolge Interpreter, `http://www.lscheffer.com/malbolge_interp.shtml`

OP	Description	Pseudo code
`<`	write a character	`A=PRINT(A%256)`
`/`	read a character	`A=INPUT`
`j`	set data pointer	`D=MEM[D]`
`i`	unconditional jump	`C=MEM[D]`
`*`	manipulate data	`A=MEM[D]=ROTATE_RIGHT(MEM[D])`
`p`	manipulate data	`A=MEM[D]=CRAZY_OP(A,MEM[D])`
`v`	stop execution	`STOP`
`o`	no operation	`NOP`

Table 1: The Operator Table of Malbolge.

		A		
		0	1	2
MEM[D]	0	1	0	0
	1	1	0	2
	2	2	2	1

Table 2: The `CRAZY_OP` operation.

3.1 Environment

Malbolge programs run on a virtual machine. The machine uses *trits* (ternary digits) and consists of one memory segment `MEM` and three registers `A`, `C` and `D`. The address space of the memory is fix and contains 59049 (3^{10}) words. A word is ten trits wide and can be assigned to a value ranging from 0 to 59048. Data segment and code segment share the same memory. Register `C` is the program counter and points to the instruction being executed. Register `D` points to some data and is used for data manipulation. Register `A` is the accumulator and stores a value that is used for data manipulation and I/O operations.

We denote by `MEM`$[i]$ the content of the memory at address i.

Although, most programmers are unfamiliar using ternary digits, the environment of Malbolge is one of the minor obstacles.

3.2 Instructions

Similar to brainfuck, Malbolge provides only 8 different operations. However, some of them inherit the spirit of INTERCAL and are therefore incomprehensible and difficult to use. Table 1 shows all operators of Malbolge.

The first two operations `<` and `/` are simple I/O operations. The operations `j` and `i` are used to set the data pointer `D` and code pointer `C` respectively. In other words, `i` is an unconditional jump instruction. Malbolge provides two operations for data manipulation. First, the `*` operator doing a circular right shift. Second, the `p` operator performing a bitwise operation on `A` and `MEM[D]` according to Table 2. For instance, `ROTATE_RIGHT`$(1022001021) = 1102200102$ and `CRAZY_OP`$(1022001021, 1102200102) = 0101210102$. With the operations `v` and `o` Malbolge provides an operator to stop the execution and a no-operation

operator.

Writing a Hello World program requires to set the correct values in register `A`. Unfortunately, we can only use `ROTATE_RIGHT` and `CRAZY_OP`, which are quite uncomfortable to use.

Further, Malbolge only provides an unconditional jump and no operations for checking conditions. Therefore, looping becomes one of the biggest difficulties.

3.3 Initialisation

To run a Malbolge program, the interpreter first initializes the program and afterwards executes it. During initialisation the source code is verified and loaded into memory, and all registers are set to 0.

To illustrate the initialisation phase, we consider the Hello World program depicted in Listing 3.

```
('&%:9]!~}|z2Vxwv-,POqponl$Hjig%eB@@>}=<M:9w
v6WsU2T|nm-,jcL(I&%$#"'CB]V?Tx<uVtT'Rpo3NlF.
Jh++FdbCBA@?]!~|4XzyTT43Qsqq(Lnmkj"Fhg${z@>
```

Listing 3: Malbolge: "Hello World!".

Actually, the source code of a Malbolge program is a sequence of encrypted operators. A Malbolge program that has the initial encoding removed is termed *normalised*. Listing 4 shows the normalised version of the Hello World program.

```
jjjjpp<jjjj*p<jjjpp<<jjjj*p<jj*o*<i<io<</<<o
o<*o*<jvoo<<opj<*<<<<<ojjopjp<jio<ovo<<jo<p*
o<*jo<iooooo<jj*p<jji<oo<j*jp<jj**p<jjopp<i
```

Listing 4: Normalised Malbolge: "Hello World!".

During initialisation the source code is verified. That is, each character is decoded and compared against the set of operators {`</ji*pvo`}. If the verification fails, the program stops with an error. Let $c_1 \cdots c_n$ be the sequence of characters of the source code, i be a position in the source code and xlat1 be a transition table according to the specification.

```
xlat1 =
 +b(29e*j1VMEKLyC})8&m#~W>qxdRp0wkrUo[D7,XTcA\"1I
 .v%{gJh4G\\-=O@5'_3i<?Z';FNQuY]szf$!BS/|t:Pn6^Ha
```

We denote by xlat1$[j]$ the symbol returned by index j. The decoding algorithm is then given by following function.

$$\text{decode}(c_1 \cdots c_n) = d_1 \cdots d_n \text{ where } d_i = \text{xlat1}[(c_i - 33 + i) \bmod 94]$$

Example 3.1. Consider the first two symbols `('` of the Hello World program depicted in Listing 3. The ASCII value of `(` and `'` is 40 and 39. We obtain

$$\text{decode}\left((\right)' = \texttt{jj}$$

performing

$$\text{xlat1}[(40 - 33 + 0) \bmod 94] = \texttt{j} \text{ and } \text{xlat1}[(39 - 33 + 1) \bmod 94] = \texttt{j}.$$

If we reverse these two characters, then the program would terminate with an error, since `1` is not an operator:

$$\text{xlat1}[(39 - 33 + 0) \bmod 94] = * \text{ and } \text{xlat1}[(40 - 33 + 1) \bmod 94] = \texttt{1}.$$

The decoding algorithm just verifies the source code. Since the same algorithm takes place during execution, the memory itself is filled with the original characters. Let c_i be the character at position i of the source code, we obtain

$$\texttt{MEM}[i] = c_i.$$

After reading the entire source code, free memory is initialized using the `CRAZY_OP` repetitively on the previous two memory words, that is

$$\texttt{MEM}[i] = \texttt{CRAZY\_OP}(\texttt{MEM}[i-1], \texttt{MEM}[i-2])$$

for each uninitialized i. (Recall that the memory contains a fixed amount of 59049 addresses).

At last, the code register `C`, the data register `D` and the accumulator `A` are initialised with 0:

$$\texttt{C} = \texttt{D} = \texttt{A} = 0.$$

Note that the required encoding provides only a minor obstacle to the programmer. Let $c'_j = (\text{xlat1}^{-1}[c_i] + 33 - i) \bmod 94$. It is easy to see that

$$\text{decode}^{-1}(d_1 \cdots d_n) = (c_1 \cdots c_n) \text{ where } c_j = \begin{cases} c'_j & \text{if } c'_j \geq 33 \\ c'_j + 94 & \text{otherwise} \end{cases}$$

defines the inverse of the decoding function in respect to the domain. In practice this means that the programmer can write the normalised code, and using the above function, compute the input for the interpreter.

The main difficulty, caused by the initialization phase, is that the initialization of the memory is limited to 8 different values represented by the operators. There is no way to directly set the values of the characters for our HelloWorld program, nor to set the correct values for branches.

3.4 Execution

After loading the program into memory, the interpreter executes it as follows: First, the interpreter fetches the instruction pointed by code register `C`. Then, the interpreter applies a similar decoding algorithm as introduced in Section 3.3. The only difference is that the interpreter considers address i instead of source code position i. That is,

$$\text{decode}(\texttt{MEM}[i]) = j \text{ where } j = \text{xlat1}[(\texttt{MEM}[i] - 33 + i) \bmod 94].$$

If the decoded character is an operator, actions are taken according to Table 1. If it is a graphical ASCII character but not an operator, then it is treated like `o`, the no-operation operator, otherwise the program terminates with an error.

At first this procedure may seem strange, because it almost does the same as the initialisation algorithm, and the initialisation algorithm ensures that the program is effectively a sequence of operators. But during execution the content of the memory will change. If the code register then points to a value which is not a graphical character, the program terminates.

After executing the instruction, the memory is modified. Using a second translation table xlat2

```
xlat2 =
5z]&gqtyfr$(we4{WP)H–Zn,[%\\3dL+Q;>U!pJS72FhOA1C
B6v^=I_0/8|jsb9m<.TVac`uY*MK'X~xDl}REokN:#?G\"i@
```

the memory is modified as follows:

$$\texttt{MEM}[\texttt{C}] = \text{xlat2}[(\texttt{MEM}[\texttt{C}] - 33 \bmod 94)].$$

Informally, the interpreter changes the current value dereferenced by register `C`.

Example 3.2. Consider the execution of the first two symbols `('`, represented by the integer values 40 and 39, of the HelloWorld program depicted in Listing 3. We obtain,

$$\texttt{MEM}[0] = \text{xlat2}[40 - 33 \bmod 94] = t \text{ and}$$
$$\texttt{MEM}[1] = \text{xlat2}[39 - 33 \bmod 94] = q.$$

Due to the definition of xlat2, the modifying algorithm ensures that the new value is a graphical character and thus a valid instruction.

Afterwards the code and data register will be incremented and the next instruction executed.

The program terminates successfully, if the operator `v` is executed.

During execution Malbolge reveals its biggest difficulty. Each instruction is modified after executing. Thus, controlling the program flow is almost impossible. Further, the programmer has to ensure that the code register `C` never points to a value which is not a graphical ASCII character.

4 Milestones

Although, Malbolge is considered to be really difficult and not suitable for real-world application, some people have invested much time to develop a program. We want to mention the most important ones. Moreover, we can show what kind of programs are possible at all and how programming in Malbolge is like.

4.1 The HEllO WORld Program

Officially, Andrew Cooke implemented the first program in the Malbolge programming language. He wrote a `Hello World` program in the year 2000, two

years after Ben Olmstead invented the language. Following listing depicts the first version published by Cooke. The main difference to Listing 3 is that this

```
(=<‘$9]7<5YXz7wT.3,+O/o’K%$H” ’˜D|#z@b=‘{
ˆLx8%$Xmrkpohm−kNi;gsedcba‘_ˆ]\[ZYXWVUTS
RQPONMLKJIHGFEDCBA@?>=<;:9876543s+O<oLm
```

Listing 5: The 1^{st} Malbolge program.

program writes “HEllO WORld” to the standard output.

More interesting than the program itself, is the fact that Cooke was not able to write a proper version and how Cooke was able to produce it. Rather than writing it, he computed it.

First, he tried a genetic algorithm. His algorithm generated random programs, scored their output and merged programs with high scores to generate a new program. Due to the self modifying nature of Malbolge programs as explained in Section 3, Cooke was not able to divide a program into fragments that could be put together again so easily.

Instead on merging randomly chosen sections, he tried to generate the “Hello World” incrementally. That is, merging of programs takes affect only after generating the same prefix. With this strategy he was able to produce “hello wor”, however only in mixed cases. But he could not get any further, because unavoidable jumps within the program executes code fragments which were already used to print the former letters.

Finally, he was successful at adapting a Best-first search algorithm, scoring the ratio between memory access and the reward for matching a character.

Surely, we do not want the reader to miss this quote by Andrew Cooke:

> *Incidentally, I've come to hate Malbolge. no doubt that was the author's intention (I'm not sure if it was also their intention to write an interpreter whose results can be unpredictable because of memory access errors, but it adds to the general flavour - grrr.....)*[6].

4.2 The cat Program

Lou Scheffer published an article “Malbolge as a cryptosystem” on his website[7]. He used a different approach than Cooke. His idea is to conceive Malbolge as a cryptosystem, that is, the programming language is a complex algorithm transforming input to output. To get the desired result, “weaknesses” in the algorithm have to be exploited.

We mention the most important “weaknesses”. Recall that Malbolge code is self modifying in the sense that after an instruction is executed, the value pointed by code register `C` is altered. Scheffer discovered that there exist fixed permutation cycles depending on the position modulo 94. For example the permutation cycle of an instruction located at position 20 modulo 94 is:

$$\texttt{CRAZY\_OP} \rightarrow \texttt{LOAD} \rightarrow \texttt{NOP} \rightarrow \texttt{NOP} \rightarrow \texttt{CRAZY\_OP} \rightarrow \texttt{LOAD} \rightarrow \ldots$$

[6] A. Cooke, `http://www.acooke.org/malbolge.html`

[7] L. Scheffer, Malbolge as Cryptosystem, `http://www.lscheffer.com/malbolge.shtml`

There also exist cycles of length 2 for every instruction, where one operation is the no-operation `NOP`. This means that every second time the intended instruction is executed.

Another weakness is that the jump instruction does not modify itself. According to the specification, the actual jump instruction is executed before the modifying procedure. This means, a jump instructions keeps to be a jump instruction as long as it is not changed with the data manipulation operators.

Further, there is also a bug in the interpreter. Due to this bug, the interpreter writes non-graphical ASCII characters, except whitespace, directly into memory. This can be used to set the right branch target address in the right spot in memory, for example.

With this discoveries, Scheffer was able to write the `cat` program, which reds from the standard input and writes it back to the standard output. More information can be found at Scheffer's website[8].

4.3 The 99 bottles of beer Program

The `99 bottles of beer` program[9] has been implemented by Hisashi Iizawa in 2005. We want to emphasize that this version was written using a real loop. At that time nobody was sure, how conditional jumps or loops can be implemented, or if they can be implemented at all.

Some additional facts about the 99 bottles of beer program: The program is 22,561 bytes long. That is about 283 lines per 80 characters. Comparing to the memory size, it is more than one third of the space. Overall, 13,802,606 instructions have to be executed.

Iizawa et al. published "Programming Method in Obfuscated Malbolge" [2]. Unfortunately the paper is written in Japanese. However, there is also an abstract in English available [1]. The paper describes a method for program obfuscation using the Malbolge language.

Referring to the abstract, the main difficulties are given by following points:

(1) Instructions are limited and unusual for value operations and flow control.

(2) Repetition is difficult because the code is self modifying.

(3) Restrictions to initialize values.

For (1), they implemented a special data structure which provides basic functionalities like incrementation using the operations of Malbolge. For (2), they use short cycles for repetition, similar to Scheffer's program. For (3), they developed a program to generate code which initializes the memory area of the program. Using this achievements, they were able to compile C-like code into Malbolge.

[8] *Ibid.*

[9] H. Iizawa, 99 bottles of beer, `http://99-bottles-of-beer.net/language-malbolge-995.html`

5 Conclusion

In tis work we introduced the reader to the Malbolge programming language that is assigned to be the most difficult and incomprehensible programming language. The kind of programs developed and the amount of work invested in them shows that Olmstead is quite successful with his goal. During the research, questions about Malbolge's computational expressiveness showed up several times. The fact that the size of memory is fixed, violates the requirements for being a Turing complete language. Moreover, the 99 bottles of beer program shows that Malbolge programs are memory expensive. There also exist a Malbolge dialect, called Malbolge Unshackled[10], that removes the memory limitation in an attempt to create a language that is Turing complete. Though, there is no proof for it.

[10] Malbolge Unshackled, `http://www.esolangs.org/wiki/Malbolge_Unshackled`

References

[1] H. Iizawa et al. Study on program obfuscation based on esoteric language Malbolge.

[2] H. Iizawa et al. Programming method in obfuscated language Malbolge. Technical report, Institute of Electronics, Information and Communication Engineers, 2005.

[3] D. Woods and J. Lyon. *The INTERCAL Programming Language Revised Reference Manual*, 1973.