Wstęp do programowania w Malbolge

Krzysztof "Palaiologos" Szewczyk
2018

Język Malbolge to z pewnością jeden z nieco trudniejszych języków programowania dla człowieka. Języki takie jak BF zaliczę będą do najprostszych języków ezoterycznych. Liczba instrukcji wirtualnej maszyny Malbolge jest równa ilości poleceń w wspomnianym wcześniej języku (którego założeniem było Turing completness i jak najmniejszy zestaw instrukcji). Każda instrukcja jednak jest specjalnie zakodowana, i znaczenie w rozszyfrowaniu jej może zależeć nawet od pozycji instrukcji w kodzie źródłowym!

Malbolge to nie język Turing complete, ponieważ nie zezwala na dostęp do nieskończonej ilości pamięci (w założeniu; Malbolge spełnia jednak praktyczną definicję Turing completness, ponieważ może obliczyć rozwiązanie każdego problemu mając dany nieskończony dostęp do pamięci którego jednak ciężko doszukać się i tak pośród dzisiejszych maszyn).

Ekosystem Malbolge nie jest skomplikowany - omawiany język jest kodem maszynowym dla trynarnej maszyny wirtualnej. Istnieją asemblery do języka (pozwalające ominąć bóle związane z zaszyfrowywaniem instrukcji). Specyfikacja języka i jego oficjalna implementacja lekko się różnią. Tą różnicą jest działanie po napotkaniu znaku poza zasięgiem (33-126). To na początku zostało uznane błędem, ale sam twórca języka przyznał, że błąd leżał faktycznie po stronie specyfikacji, a nie jego implementacji.

Malbolge posiada trzy rejestry (to aż o 3 więcej niż BF); A, C, i D. Kiedy program jest uruchamiany, wszystkie rejestry mają początkową wartość 0. A to akumulator, ustawiony na wartość zapisaną przez wszystkie operacje zapisu na pamięci, używany do wejścia i wyjścia. C, wskaźnik kodu to specjalny rejestr wskazujący aktualną instrukcję. D to wskaźnik danych który jest inkrementowany automatycznie po każdej instrukcji, ale lokacja którą wskazuje jest używana do manipulacji danymi. D zawiera adres, notacja [D] to wartość zawarta na tym adresie, podobnie z [C].

Maszyna wirtualna może zaadresować do 54 049 bajtów pamięci (3^10), z których każda może zawierać jedną liczbę trynarną długą na 10 trytów. Każda lokacja w pamięci zawiera adres od 0 do 59048 lub wartość w takim samym zakresie. Inkrementacja lub dekrementacja poza limitem pozostawi komórkę w pamięci na wartości zerowej. Język używa tej samej pamięci dla danych i kodu. Tą samą technikę można zauważyć w procesorach x86 Intela. Przed rozpoczęciem działania programu obraz pamięci jest ładowany. Wszystkie spacje, tabulacje, entery, itd. są ignorowane i aby sprawić że programowanie jest trudniejsze, program musi się zaczynać jedną z instrukcji omówionych później. Reszta pamięci jest wypełniona tzw. szaloną operacją na dwóch wcześniejszych adresach: **[C] = SZA[M-2],[M-1]**. Tak wypełniona pamięć będzie się powtarzać co 12 adresów (indywidualne liczby trynarne będą się powtarzać co trzy lub cztery adresy, więc grupa cyfr trynarnych z pewnością będzie powtarzać się co dwanaście miejsc).

Malbolge ma osiem instrukcji. Aby sprawdzić która instrukcja zostanie wykonana, dodaje się do

[C], wartość C, biorąc resztę z dzielenia tej sumy przez 94. Tak więc mając dane osiem instrukcji po wykonaniu **([C] + C) % 94** dane są następujące liczby:

Kod	Mnemonika	Opis
4	`jmp [D]`	[D] to punkt do którego wirtualna maszyna skoczy i zacznie wykonywać instrukcje po napotkaniu instrukcji.
5	`out A`	Wyświetla zawartość rejestru A jako znak ASCII (patrz Dodatek B)
23	`in A`	Wczytuje znak jako kod ASCII do A, nowa linia to 10, EOF to 59048
39	`rot [D]` `mov A, [D]`	Odwraca wartość [d] o jeden tryt (np. 00212 -> 20021). Nową wartość umieszcza w [D] i A.
40	`mov D, [D]`	Kopiuje [D] do D
62	`rot [D], A` `mov A, [D]`	Wykonuje szaloną operację z wartością [D] i A. Umieszcza wynik i w [D], i w A.
68	`nop`	Nie robi nic
81	`hlt`	Wyłącza wirtualną maszynę
Inne	`nop`	Patrz 68; Nie robi nic.

Po inkrementacji C, wykonywana (winna) instrukcja zostaje zaszyfrowana (patrz niżej), więc nie zrobi tego samego, chyba że wykonywany był skok. Tuż po skoku, niewykonana (niewinna) instrukcja zostanie zaszyfrowana (tj., ta wcześniejsza od tej, do której skakano). Wtedy wartości C i D są zwiększane o jeden, i następna instrukcja jest wykonywana.

Szalona operacja sprawia, że dla każdego trytu, dla dwóch wejść, używana jest podana tabelka (niżej) do ustalenia finalnej wartości; Wszystkie podane wartości są dla wejścia 2 zwiększającego się o 1 (tj. 0, 1, 2). Dla przykładu, SZA 012210, 210012 = 002200.

	0	1	2
0	**1**	**0**	**0**
1	**1**	**0**	**2**
2	**2**	**2**	**1**

Tuż po wykonaniu instrukcji, wartość [C] (przed wszystkimi operacjami) jest zamieniana z resztą dzielenia jej przez 94. Wtedy rezultat jest szyfrowany; Znajdź go w tabeli (dodatek A). Wszystkie powyższe techniki produkują duży i niewydajny kod (czego można było się spodziewać; aktualnie nie jest

znany mi sposób przygotowywania wydajnych i małych programów w Malbolge). Generalna strategia programowania w Malbolge jest dość prosta, jeśli uda się zmieścić cały program do pamięci, można napisać program umożliwiający arbitralny dostęp do pamięci, czego konsekwencją może istnieć konwersja z BF do Malbolge (spokojnie, pracuję już nad tym).

Biorąc pod uwagę możliwości utrudnienia, Malbolge jest proste – możliwości takie to np. usuwanie wszystkich krótkich cykli po ponownym "strawieniu" tabeli permutacji instrukcji i modyfikacja instrukcji podczas wykonywania ich (co umożliwiałoby szyfrowanie nawet skoków). Cała „trudność" języka leży w ilości rzeczy, które trzeba brać pod uwagę przy programowaniu.

Istnieje również wersja Malbolge Unshackled, która znosi limit pamięci, dzięki czemu jest szansa zaliczenia Malbolge do języków Turing complete (z definicji). Drobną wskazówką z mojej strony jest fakt, że programując w Malbolge, nie powinieneś myśleć jak programista, tylko kryptograf – jeśli przygotujesz już algorytm, implementując go nie będziesz używał dużego kawałka wiedzy programistycznej. Malbolge może być potencjalnie użyte w kryptografii (programy typu crack-me wyświetlające wynik tylko wtedy kiedy podane jest właściwe hasło; chociaż zaimplementowanie tego w BF odstraszyłoby większość domorosłych hakerów).

Dodatek A

Tabela szyfrowania. Lewa kolumna to wejście, prawa to wyjście.

0	57	**19**	108	**38**	113	**57**	91	**76**	79
1	109	**20**	125	**39**	116	**58**	37	**77**	65
2	60	**21**	82	**40**	121	**59**	92	**78**	49
3	46	**22**	69	**41**	102	**60**	51	**79**	67
4	84	**23**	111	**42**	114	**61**	100	**80**	66
5	86	**24**	107	**43**	36	**62**	76	**81**	54
6	97	**25**	78	**44**	40	**63**	43	**82**	118
7	99	**26**	58	**45**	119	**64**	81	**83**	94
8	96	**27**	35	**46**	101	**65**	59	**84**	61
9	117	**28**	63	**47**	52	**66**	62	**85**	73
10	89	**29**	71	**48**	123	**67**	85	**86**	95
11	42	**30**	34	**49**	87	**68**	33	**87**	48
12	77	**31**	105	**50**	80	**69**	112	**88**	47
13	75	**32**	64	**51**	41	**70**	74	**89**	56
14	39	**33**	53	**52**	72	**71**	83	**90**	124
15	88	**34**	122	**53**	45	**72**	55	**91**	106
16	126	**35**	93	**54**	90	**73**	50	**92**	115
17	120	**36**	38	**55**	110	**74**	70	**93**	98
18	68	**37**	103	**56**	44	**75**	104		

Dodatek B

Tabela ASCII															
	_0	**_1**	**_2**	**_3**	**_4**	**_5**	**_6**	**_7**	**_8**	**_9**	**_A**	**_B**	**_C**	**_D**	**_E**	**_F**
0_	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Dodatek C (kod źródłowy oryginalnej VM)

```
/* Interpreter for Malbolge.                                          */
/* '98 Ben Olmstead.                                                  */
/*                                                                    */
/* Malbolge is the name of Dante's Eighth circle of Hell.  This       */
/* interpreter isn't even Copylefted; I hereby place it in the public */
/* domain.  Have fun...                                               */
/*                                                                    */
/* By the way, this code assumes that short is 16 bits.  I haven't    */
/* seen any case where it isn't, but it might happen.  If short is    */
/* longer than 16 bits, it will still work, though it will take up    */
```

```
/* considerably more memory.                                          */
/*                                                                    */
/* If you are compiling with a 16-bit Intel compiler, you will need   */
/* >64K data arrays; this means using the HUGE memory model on most   */
/* compilers, but MS C, as of 8.00, possibly earlier as well, allows  */
/* you to specify a custom memory-model; the best model to choose in  */
/* this case is /Ashd (near code, huge data), I think.                */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <malloc.h>
#include <string.h>

#ifdef __GNUC__
    static inline
#endif
void exec( unsigned short * mem );

#ifdef __GNUC__
    static inline
#endif
unsigned short op( unsigned short x, unsigned short y );

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

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

int main( int argc, char ** argv ) {
    FILE * f;
    unsigned short i = 0, j;
    int x;
    unsigned short * mem;
    if ( argc != 2 ) {
        fputs( "invalid command line\n", stderr );
        return ( 1 );
    }
    if ( ( f = fopen( argv[1], "r" ) ) == NULL ) {
        fputs( "can't open file\n", stderr );
        return ( 1 );
    }
    #ifdef _MSC_VER
    mem = (unsigned short *)_halloc( 59049, sizeof(unsigned short) );
    #else
    mem = (unsigned short *)malloc( sizeof(unsigned short) * 59049 );
    #endif
    if ( mem == NULL ) {
        fclose( f );
        fputs( "can't allocate memory\n", stderr );
        return ( 1 );
    }
    while ( ( x = getc( f ) ) != EOF ) {
        if ( isspace( x ) ) continue;
```

```
        if ( x < 127 && x > 32 ) {
            if ( strchr( "ji*p</vo", xlat1[( x - 33 + i ) % 94] ) == NULL ) {
                fputs( "invalid character in source file\n", stderr );
                free( mem );
                fclose( f );
                return ( 1 );
            }
        }
        if ( i == 59049 ) {
            fputs( "input file too long\n", stderr );
            free( mem );
            fclose( f );
            return ( 1 );
        }
        mem[i++] = x;
    }
    fclose( f );
    while ( i < 59049 ) mem[i] = op( mem[i - 1], mem[i - 2] ), i++;
    exec( mem );
    free( mem );
    return ( 0 );
}

#ifdef __GNUC__
    static inline
#endif
void exec( unsigned short * mem ) {
    unsigned short a = 0, c = 0, d = 0;
    int x;
    for (;;) {
        if ( mem[c] < 33 || mem[c] > 126 ) continue;
        switch ( xlat1[( mem[c] - 33 + c ) % 94] ) {
            case 'j': d = mem[d]; break;
            case 'i': c = mem[d]; break;
            case '*': a = mem[d] = mem[d] / 3 + mem[d] % 3 * 19683; break;
            case 'p': a = mem[d] = op( a, mem[d] ); break;
            case '<':
                #if '\n' != 10
                if ( x == 10 ) putc( '\n', stdout );
                else
                #endif
                    putc( a, stdout );
                break;
            case '/':
                x = getc( stdin );
                #if '\n' != 10
                if ( x == '\n' ) a = 10;
                else
                #endif
                    if ( x == EOF ) a = 59048;
                    else a = x;
                break;
            case 'v': return;
        }
        mem[c] = xlat2[mem[c] - 33];
        if ( c == 59048 ) c = 0;
        else c++;
```

```
        if ( d == 59048 ) d = 0;
        else d++;
    }
}

#ifdef __GNUC__
    static inline
#endif
unsigned short op( unsigned short x, unsigned short y ) {
    unsigned short i = 0, j;
    static const unsigned short p9[5] =
    { 1, 9, 81, 729, 6561 };
    static const unsigned short o[9][9] = {
        { 4, 3, 3, 1, 0, 0, 1, 0, 0 },
        { 4, 3, 5, 1, 0, 2, 1, 0, 2 },
        { 5, 5, 4, 2, 2, 1, 2, 2, 1 },
        { 4, 3, 3, 1, 0, 0, 7, 6, 6 },
        { 4, 3, 5, 1, 0, 2, 7, 6, 8 },
        { 5, 5, 4, 2, 2, 1, 8, 8, 7 },
        { 7, 6, 6, 7, 6, 6, 4, 3, 3 },
        { 7, 6, 8, 7, 6, 8, 4, 3, 5 },
        { 8, 8, 7, 8, 8, 7, 5, 5, 4 },
    };
    for ( j = 0; j < 5; j++ )
        i += o[y / p9[j] % 9][x / p9[j] % 9] * p9[j];
    return ( i );
}
```

Dodatek D (przykładowe programy)

Wyświetlanie „Hello World!”:

```
(=<`$9]7<5YXz7wT.3,+O/o'K%$H"'~D|#z@b=`{^Lx8%$Xmrkpohm-
kNi;gsedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;:9876543s+O<oLm
```

Kopiowanie wejścia do wyjścia:

```
(=BA#9"=<;:3y7x54-21q/p-,+*)"!h%B0/.
~P<
<:(8&
66#"!~}|{zyxwvu
gJ%
```

Dodatek E (linki)

Artykuł na Wikipedii: Malbolge

Interpreter online, generator tekstu: ZB3

Malbolge Unshackled: Matthias Ernst

Kod źródłowy interpretera BF w Malbolge: Github GIST

Asembler dla Malbolge: Matthias Ernst

Debugger dla Malbolge: Debugger