Turing completness of Mersenne Twister

Krzysztof Szewczyk

2019

1 Introduction

In this paper, I will prove Mersenne Twister turing complentess. Mersenne Twister is by far the most popular PRNG[1] in use as of 2019. Mersenne Twister will be used as component of Seed esoteric programming language.

2 Seed programming language

Seed is a language based on random seeds. Actually, programs only contain two instructions: length and random seed, separated by a space. To execute a Seed program, the seed is fed into a Mersenne Twister random number generator, and the randomness obtained is converted into a string of length bytes, which will be executed by a Befunge-98 interpreter (or compiler).[2].

An example Seed program looks like this, and will generate 780 bytes long valid Befunge-98 code.

```
780 983247832
```

Since standard Befunge is considered to be a finite state machine, it is, strictly speaking, not Turing-complete; thus, Seed cannot be Turing-complete either - states Esolang Wiki. Yet Befunge-98 is compiliant to Funge-98 standard. Wikipedia says:

The Befunge-93 specification restricts each valid program to a grid of 80 instructions horizontally by 25 instructions vertically. Program execution which exceeds these limits "wraps around" to a corresponding point on the other side of the grid; a Befunge program is in this manner topologically equivalent to a torus. Since a Befunge-93 program can only have a single stack and its storage array is bounded, the Befunge-93 language is not Turing-complete (however, it has been shown that Befunge-93 is Turing Complete with unbounded stack word size).[4] The later Funge-98 specification provides Turing completeness by removing the size restrictions on the program; rather than wrapping

[1] PseudoRandom Number Generator
[2] https://esolangs.org/wiki/Seed - accessed 16.06.2019

around at a fixed limit, the movement of a Funge-98 instruction pointer follows a model dubbed "Lahey-space" after its originator, Chris Lahey. In this model, the grid behaves like a torus of finite size with respect to wrapping, while still allowing itself to be extended indefinitely.[3]

This means, Befunge-98 **is** Turing complete, but Seed may not be, because we aren't able to generate **all** possible befunge programs, because Mersenne Twister has a period of $2^{19937} - 1$, but, if Mersenne Twister can generate befunge program being another Turing complete language interpreter, Seed is turing complete too.

3 ByteByteJump

ByteByteJump is an extremely simple One Instruction Set Computer (OISC). Its single instruction copies 1 byte from a memory location to another, and then performs an unconditional jump. There are two possible ways to prove turing completness of ByteByteJump.

ByteByteJump is actually a variation of BitBitJump, that is operating on bits, not bytes, therefore proving BitBitJump is Turing complete can prove Turing completness of ByteByteJump. Simulation is one way to prove Turing completeness - *If an interpreter for A can be implemented in B, then B can solve at least as many problems as A can.* Using BitBitJump assembler and standard library[4], it's possible to create brainfuck interpreter:

```
# BitBitJump brainfuck (DBFI) interpreter by O. Mazonka
Z0:0 Z1:0 start
.include lib.bbj
:mem:0 0 0
mem mem
ip_start:mem ip:mem ip_end:0
mp_start:0 mp:0 mp_end:0
x:0 y:0 m1:-1
SPACE:32 MINUS:45 PLUS:43
TICK:39 EOL:10 Excl:33
LEFT:60 RIGHT:62 LB:91
RB:93 DOT:46 COMMA:44
start: .copy ZERO x
.in x
.ifeq x ZERO initgl chkex
chkex: .ifeq x Excl initgl storei
storei: .toref x ip
.add ip BASE ip
0 0 start
initgl: .copy ip ip_end
.copy ip mp_start
.copy ip mp
```

[3]https://en.wikipedia.org/wiki/Befunge - accessed 16.06.2019
[4]http://mazonka.com/bbj/bbjasm.cpp and http://mazonka.com/bbj/lib.bbj - accessed 16.06.2019

```
.copy ip mp_end
.add mp_end BASE mp_end
.copy ip_start ip
loop: 0 0
.deref ip x
.ifeq x PLUS plus chk_ms
plus: .plus
0 0 next_ip
chk_ms: .ifeq x MINUS minus chk_lt
minus: .minus
0 0 next_ip
chk_lt: .ifeq x LEFT left chk_rt
left: .left
0 0 next_ip
chk_rt: .ifeq x RIGHT right chk_dt
right: .right
0 0 next_ip
chk_dt: .ifeq x DOT dot chk_cm
dot: .deref mp x
.out x
0 0 next_ip
chk_cm: .ifeq x COMMA comma chk_lb
comma: .copy ZERO x
.in x
.toref x mp
0 0 next_ip
chk_lb: .ifeq x LB lb chk_rb
lb: .lb
0 0 next_ip
chk_rb: .ifeq x RB rb next_ip
rb: .rb
0 0 next_ip
next_ip: 0 0
.add ip BASE ip
.ifeq ip ip_end exit loop
exit: 0 0 -1
.def plus : mp x
.deref mp x
.inc x
.toref x mp
.end
.def minus : mp x
.deref mp x
.dec x
.toref x mp
.end
.def right : mp BASE mp_end x ZERO
.add mp BASE mp
```

```
.iflt mp mp_end ret incend
incend: .add mp_end BASE mp_end
.copy ZERO x
.toref x mp
ret: 0 0
.end
.def left : mp BASE
.sub mp BASE mp
.end
.def lb : ip mp x y ZERO ONE BASE LB RB
.deref mp x
.ifeq x ZERO gort ret
gort: .copy ONE y
loop: .add ip BASE ip
.deref ip cmd
.ifeq cmd LB incy chk_rb
chk_rb: .ifeq cmd RB decy loop

incy: .inc y
0 0 loop
decy: .dec y
.iflt ZERO y loop ret
cmd:0 0
ret: 0 0
.end
.def rb : ip mp x y ZERO ONE BASE LB RB
.deref mp x
.ifeq x ZERO ret golt
golt: .copy ONE y
loop: .sub ip BASE ip
.deref ip cmd
.ifeq cmd RB incy chk_lb
chk_lb: .ifeq cmd LB decy loop
incy: .inc y
0 0 loop
decy: .dec y
.iflt ZERO y loop ret

cmd:0 0
ret: 0 0
.end
.def dump : ip_start x y BASE ip_end mp_start mp_end SPACE ip mp TICK EOL
.copy ip_start y
dumpi: .deref y x
.out x
.ifeq y ip outi noi
outi: .out TICK
noi: .add y BASE y
```

```
.ifeq y ip_end dumpmst dumpi
dumpmst: 0 0
.copy mp_start y
dumpm: .deref y x
.out SPACE
.ifeq y mp outm nom
outm: .out TICK
nom: .prn x
.add y BASE y
.ifeq y mp_end ret dumpm
ret:  .out EOL
.end
```

To prove Turing completness of Brainfuck, we will use the same rule. Daniel Cristofani wrote generic Turing machine emulator written in Brainfuck:

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

This means, Brainfuck and BitBitJump are turing complete, and may possibly prove that ByteByteJump is turing compelete too. Another method to prove turing completness of ByteByteJump has been partly presented by Esolangs wiki. It's possible to simulate one SUBLEQ instruction tick in ByteByteJump. Suppose we have the following values stored in memory:

```
Address        | Value
---------------+------
000800..00087F | 01
000880..0008FF | 02
01XXYY         | XX
02XXYY         | YY
03XXYY         | XX-YY
```

Then the following ByteByteJump program (using 3-byte addresses) will take the byte value at address 100h, subtract the byte value at address 200h, store the resulting byte value at address 300h, and jump to address 400h if the result was negative (¿= 80h). Addresses which differ between the big-endian and little-endian versions are marked as bold.

```
Big-endian version           | Little-endian version
-----------------------------+-----------------------------
```

```
000000: 000100 000013 000009 | 000000: 000100 000013 000009
000009: 000200 000014 000012 | 000009: 000200 000012 000012
000012: 030000 000300 00001B | 000012: 030000 000300 00001B
00001B: 000300 000026 000024 | 00001B: 000300 000024 000024
000024: 000800 00002D 00002D | 000024: 000800 00002F 00002D
00002D: 003F36 000035 000000 | 00002D: 003F36 000033 000000
000036: 000000 000000 000400 | 000036: 000000 000000 000400
00003F: ...... ...... ...... | 00003F: ...... ...... ......
```

Below is the previous ByteByteJump example rewritten in ByteByte/Jump machine code. Instruction words which differ between the big-endian and little-endian versions are marked as bold.

```
Big-endian version           | Little-endian version
-----------------------------+----------------------------
000000: 000100 00000D        | 000000: 000100 00000D
000006: 000200 00000E        | 000006: 000200 00000C
00000C: 030000 800300 000017 | 00000C: 030000 800300 000015
000015: 000800 00001B        | 000015: 000800 00001D
00001B: 002724 000023        | 00001B: 002724 000021
000021: 800000               | 000021: 800000
000024: 800400               | 000024: 800400
000027: ......               | 000027: ......
```

The ByteByteJump version takes up 63 bytes, while the ByteByte/Jump version takes up 39 bytes. At address 00000C in the ByteByte/Jump program, notice the use of multiple (in this case 2) destinations for the move instruction. This proves that ByteByteJump can compute as much as SUBLEQ can, so if SUBLEQ is turing complete, ByteByteJump is too.

This SUBLEQ assembly [5] program I wrote is emulating Brainfuck interpreter:

```
top:top top sqmain

_interpret:
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; bp 0
bp; sp bp
c2 sp
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t1 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t2 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t3 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
```

[5] http://mazonka.com/subleq/sqasm.cpp - accessed 16.06.2019

```
?+6; sp ?+2; t4 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t5 0

t1; t2; bp t1; c1 t1; t1 t2
t1; t3; bp t1; c2 t1; t1 t3
?+23; ?+21; ?+24; t3 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; t2 Z; Z 0; Z

t1; t2; bp t1; c4 t1; t1 t2
?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; c3 Z; Z 0; Z
l1:
t2; t1; bp t2; c5 t2; t2 t1
t2; t3; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t3
t1; t2; bp t1; c4 t1; t1 t2
t1; t4; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t4
t2; t1; t3 t2; t4 t2; t2 t1
t2; t4; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t4
t1; t4 Z; Z t1 ?+3; Z Z ?+9; Z; t4 t1; t4 t1
Z t1 l3
t2; t4; bp t2; c5 t2; t2 t4
t2; t3; ?+11; t4 Z; Z ?+4; Z; 0 t2; t2 t3
t4; t2; bp t4; c4 t4; t4 t2
t4; t5; ?+11; t2 Z; Z ?+4; Z; 0 t4; t4 t5
t2; t4; t3 t2; t5 t2; t2 t4
t2; t5; ?+11; t4 Z; Z ?+4; Z; 0 t2; t2 t5
t4; t2; bp t4; dec t4; t4 t2
?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; t5 Z; Z 0; Z

t2; t3; bp t2; dec t2; t2 t3
t2; t5; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t5
t3; t5 Z; Z t3; Z; c14 t3 ?+3
t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
Z t3 l21
t1; t2; bp t1; c2 t1; t1 t2
t2 Z; ?+9; Z ?+5; Z; inc 0

Z Z l22
l21:
t5; t2; bp t5; dec t5; t5 t2
t5; t3; ?+11; t2 Z; Z ?+4; Z; 0 t5; t5 t3
t2; t3 Z; Z t2; Z; c13 t2 ?+3
t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
Z t2 l19
t1; t2; bp t1; c2 t1; t1 t2
t2 Z; ?+9; Z ?+5; Z; dec 0
```

```
Z Z l20
l19:
t3; t5; bp t3; dec t3; t3 t5
t3; t2; ?+11; t5 Z; Z ?+4; Z; 0 t3; t3 t2
t5; t2 Z; Z t5; Z; c12 t5 ?+3
t5 t5 ?+9; t5 Z ?+3; Z Z ?+3; inc t5
Z t5 l17
t1; t2; bp t1; c2 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t3 Z; ?+9; Z ?+5; Z; inc 0

Z Z l18
l17:
t2; t3; bp t2; dec t2; t2 t3
t2; t5; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t5
t3; t5 Z; Z t3; Z; c11 t3 ?+3
t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
Z t3 l15
t1; t2; bp t1; c2 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t3 Z; ?+9; Z ?+5; Z; dec 0

Z Z l16
l15:
t5; t2; bp t5; dec t5; t5 t2
t5; t3; ?+11; t2 Z; Z ?+4; Z; 0 t5; t5 t3
t2; t3 Z; Z t2; Z; c10 t2 ?+3
t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
Z t2 l13
t1; t2; bp t1; c2 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t1; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t1
t1 (-1)

Z Z l14
l13:
t3; t5; bp t3; dec t3; t3 t5
t3; t2; ?+11; t5 Z; Z ?+4; Z; 0 t3; t3 t2
t5; t2 Z; Z t5; Z; c9 t5 ?+3
t5 t5 ?+9; t5 Z ?+3; Z Z ?+3; inc t5
Z t5 l11
t1; (-1) t1
t2; t3; bp t2; c2 t2; t2 t3
t2; t4; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t4
?+23; ?+21; ?+24; t4 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; t1 Z; Z 0; Z
```

```
Z Z l12
l11:
t1; t2; bp t1; dec t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t3 Z; Z t2; Z; c7 t2 ?+3
t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
t3; Z t2 l9
t1; t4; bp t1; c2 t1; t1 t4
t1; t5; ?+11; t4 Z; Z ?+4; Z; 0 t1; t1 t5
t4; t1; ?+11; t5 Z; Z ?+4; Z; 0 t4; t4 t1
t5; t1 Z; Z t5 ?+3; Z Z ?+9; Z; t1 t5; t1 t5
Z t5 l9; inc t3;
l9:
Z t3 l10
t1; t2; bp t1; c6 t1; t1 t2
?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; dec Z; Z 0; Z

l4:
t1; t2; bp t1; c6 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t3 Z; Z t2; Z; c3 t2
Z t2 l5
t1; t2; bp t1; c5 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t1; bp t2; c4 t2; t2 t1
t1 Z; ?+9; Z ?+5; Z; dec 0
t2; t4; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t4
t1; t2; t3 t1; t4 t1; t1 t2
t1; t4; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t4
t2; t1; bp t2; dec t2; t2 t1
?+23; ?+21; ?+24; t1 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; t4 Z; Z 0; Z

t2; t3; bp t2; dec t2; t2 t3
t2; t1; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t1
t3; t1 Z; Z t3; Z; c8 t3 ?+3
t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
Z t3 l7
t1; t2; bp t1; c6 t1; t1 t2
t2 Z; ?+9; Z ?+5; Z; dec 0

Z Z l8
l7:
t1; t2; bp t1; dec t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t3 Z; Z t2; Z; c7 t2 ?+3
t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
```

```
Z t2 l6
t1; t2; bp t1; c6 t1; t1 t2
t2 Z; ?+9; Z ?+5; Z; inc 0

l6:
l8:

Z Z l4
l5:

l10:
l12:
l14:
l16:
l18:
l20:
l22:

l2:
t4; t2; bp t4; c4 t4; t4 t2
t2 Z; ?+9; Z ?+5; Z; inc 0
Z Z l1
l3:

?+8; sp ?+4; t5; 0 t5; inc sp
?+8; sp ?+4; t4; 0 t4; inc sp
?+8; sp ?+4; t3; 0 t3; inc sp
?+8; sp ?+4; t2; 0 t2; inc sp
?+8; sp ?+4; t1; 0 t1; inc sp
sp; bp sp
?+8; sp ?+4; bp; 0 bp; inc sp
?+8; sp ?+4; ?+7; 0 ?+3; Z Z 0

_main:
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; bp 0
bp; sp bp
c15 sp
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t1 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t2 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t3 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t4 0
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t5 0
```

```
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; t6 0

t1; t2; bp t1; c15 t1; t1 t2
?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; dec Z; Z 0; Z

l23:
t1; t2; bp t1; c15 t1; t1 t2
t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
t2; t3 Z; Z t2 ?+3; Z Z ?+9; Z; t3 t2; t3 t2
Z t2 l25
t3; (-1) t3
t1; t4; bp t1; dec t1; t1 t4
t1; t5; bp t1; c16 t1; t1 t5
t1; t6; ?+11; t5 Z; Z ?+4; Z; 0 t1; t1 t6
t5 Z; ?+9; Z ?+5; Z; inc 0
t5; t1; t4 t5; t6 t5; t5 t1
?+23; ?+21; ?+24; t1 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; t3 Z; Z 0; Z

t2; t1; bp t2; dec t2; t2 t1
t2; t3; bp t2; c16 t2; t2 t3
t2; t4; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t4
t3; t4 Z; Z t3; Z; dec t3
t4; t2; t1 t4; t3 t4; t4 t2
t4; t3; ?+11; t2 Z; Z ?+4; Z; 0 t4; t4 t3
t2; t3 Z; Z t2; Z; c17 t2 ?+3
t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
t3; inc t3; Z t2 ?+3; Z Z l26
t4; t1; bp t4; dec t4; t4 t1
t4; t5; bp t4; c16 t4; t4 t5
t4; t6; ?+11; t5 Z; Z ?+4; Z; 0 t4; t4 t6
t5; t6 Z; Z t5; Z; dec t5
t6; t4; t1 t6; t5 t6; t6 t4
t6; t5; ?+11; t4 Z; Z ?+4; Z; 0 t6; t6 t5
t4; t5 Z; Z t4; Z; c18 t4 ?+3
t4 t4 ?+9; t4 Z ?+3; Z Z ?+3; inc t4
Z t4 ?+3; Z Z l26; t3;
l26:
Z t3 l27
t1; t2; bp t1; c15 t1; t1 t2
?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
Z ?+11; Z; 0; c3 Z; Z 0; Z
l27:

l24:
Z Z l23
```

```
l25:

t1; t2; bp t1; dec t1; t1 t2
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+9; sp ?+5; t2 Z; Z 0; Z
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; ?+2 0 _interpret; . ?;
c5 sp

?+8; sp ?+4; t6; 0 t6; inc sp
?+8; sp ?+4; t5; 0 t5; inc sp
?+8; sp ?+4; t4; 0 t4; inc sp
?+8; sp ?+4; t3; 0 t3; inc sp
?+8; sp ?+4; t2; 0 t2; inc sp
?+8; sp ?+4; t1; 0 t1; inc sp
sp; bp sp
?+8; sp ?+4; bp; 0 bp; inc sp
?+8; sp ?+4; ?+7; 0 ?+3; Z Z 0

sqmain:
dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
?+6; sp ?+2; ?+2 0 _main; . ?; inc sp

Z Z (-1)

. c5:-2 c3:0 c17:10 c18:13 c4:2 c2:20 c6:3 c1:4 c12:43 c9:44 c11:45 c10:46 c16:501
 (wrapped) c15:502 c13:60 c14:62 c8:91 c7:93

. t1:0 t2:0 t3:0 t4:0 t5:0 t6:0

. inc:-1 Z:0 dec:1 ax:0 bp:0 sp:-sp
```

The interpreter was build from following C code:

```
void interpret(char * input) {
    char current_char;
    int i, loop;
    unsigned char tape[16] = {0};
    unsigned char * ptr = tape;

    for (i = 0; input[i] != 0; i++) {
        current_char = input[i];
        if (current_char == '>') {
            ++ptr;
        } else if (current_char == '<') {
```

```
            --ptr;
        } else if (current_char == '+') {
            ++*ptr;
        } else if (current_char == '-') {
            --*ptr;
        } else if (current_char == '.' ) {
            putchar(*ptr);
        } else if (current_char == ',') {
            *ptr = getchar();
        } else if (current_char == ']' && *ptr) {
            loop = 1;
            while (loop > 0) {
                current_char = input[--i];
                if (current_char == '[') {
                    loop--;
                } else if (current_char == ']') {
                    loop++;
                }
            }
        }
    }
}


int main(void) {
    char buf[500];
    char c, r = 1;
    for(;r;) {
        buf[c++] = getchar();
        if(buf[c-1] == 10 || buf[c-1] == 13)
            r = 0;
    }
    interpret(buf);
}
```

It's able to hold up to 500 program bytes (just enough to simulate Cristofani's Turing machine emulator) and 16 memory cells. That may not be enough, but the size can be flexibly changed. The resulting subleq code is pretty big[6], it's 16 kilobytes big, but it's doing his job. Turing completness of Brainfuck was proven before, therefore to prove Turing completness of Seed, it's required to create ByteByteJump interpreter in Befunge-98, and then transform it to Seed.

[6]https://pastebin.com/TEgwuLsb - accessed 16.06.2019

4 Proof

The following code ByteByteJump being a reference implementation written in C.

```
a[99], b, c;

main() {
    for (; b<99; b++)
        scanf("%d",&a[b]);

    for (; !c<0; c++) {
        if (a[c]<0)
            a[c]=getchar();
        if (a[c+1]<0)
            putchar(a[c+1]);

        a[c+1]=a[c];
        c=a[c+2]
    }
}
```

If elements were listed in counter-order they are pushed, A is equal to `*(pc)` and is first, B is second and is equal to `*(pc+1)` and C is third and is equal to `*(pc+2)`, following Befunge code will perform ByteByteJump instruction tick[7]:

```
v@0~$<
>:0‘!|
v‘0:\<
>!v  >’$\@
 v_$:^
 >:,0@
```

The problem here is, the registers are immutable, so the place marked with apostrophe needs to implement some kind of storing register values once again. This code is able to perform single ByteByteJump tick. I couldn’t find exact code to be generated using Mersenne Twister, but this is close enough:

```
9$v@0~$<
  >:0‘!|
  v‘0:\<
  >!v  >’$\@
```

[7]If anything is wrong, please contact me!

```
v_$:^
>:,0@
```

Stray 9 is being pushed, but $ is taking it away. Finally, the Seed code looks such:

```
57 4714293885035653766802571995034228521569587547045721276692009395896559901429168221147480393002
0623371824316602132470018544327484274274429718304239791045420821013074226051910565103245066485566
9371471875981623296895528966344487863605303564523124642891799249376650587202998954406589441816666
8528473885999991247165905756500039241968936127197248544473128440498723880311521042857294510890081
9109721497838211221824262771965172871257730665240545292278230724948594704521598278479039655572661
4452660277847338727989733470914285315139381293017116330090443654887060300654900551570907468284433
5684420122066804505740714675179347042321709946714525517419824160681457835126076935957105375588881
0681819723911697300636559356323536164774341134162482205210381696815327412243428020088882495487556
2281132506425581815497956492571053416557285244276124917677841668804463094204096627196372343024559
7922118193085784182969436218465393939394001579733297845979425317611031487399422826188880122899992
9357032961855122345718242074692721280155064674315275482164006462676154258255713845265197000925337
7091434613017288430562202737079349699328184701701764350643556222991698410708395193828657701227332
2219142205431519815793667424793469949647120254427032506135201483013717824508244571725326017756004
4975718676244570705702898737127857362907737063247049618621857432080179804651084670862050213956002
7754634519868667509507825587559416906479667307470882210665992018788206224760958756017478117064413
6743072295100224221360470988706248114992855174516311004557299499184422321666362120304207529419550
0745833998452733312509339018972104231560449843526914354942016673217720037022852727360621861717719
7536243182416326967200398253738298206613661379940302492401814551109955772049230530374809932781008
1151108031426236401028185165115107295747536562912806803359755956018662587794205470438617535949995
7313993037809942014945274573180903373775605194791392426548458280061824447333395717396022224331117
3852287502254661029862749222258797175689732808771940745415324855720388642182864345388909019235559
7070508424531218444167409851565925348262126061721178655020485289565223676888685220950653552341449
9109933185767482637394783058702849451069760329660736109348084293515467235328841969935473965016883
0901784848513155341695640591168352689623204677386196191176731937343246021775587448160758760436117
5808993600773025345073337583122812710629525926172361177133446855374616073954837595004683192376550
2332934633396873279641319268293676713312232548127335481012472966440017336778132548865685958143887
6994047422939469208951998181090971962826335728497344217756804141636338689151672559295289216807775
2356058400558627679496749205182329061576759920265706082022392867890077460161690803132134681942221
6212304883453292637286835792711473250781222609148840033877981488957801787967021003926268521517888
4690864530024735262690192365649204079386396597134754136470637502698463271368572751598489549710881
1537632795997904759573044368087076491167998363882673296770584453611919293523264588504061601806995
5661086088136640840699435161518950137902407329143121025976204756085930220702569150665965556851221
9835596964453538868154736015979177273127691802171300980538950899957283803566284733509787307717119
3936867853512068970124025278864374957406200960980358401890344196818419366442665565233304000989916
8155591616539241508713639411085080660922342180388228036038525967356447426705058047766316393550224
3290490410756418271781267361399899055278596661870896239925767576324437445711891644688673328221221
0362105820596963847974398138260084810928299592303752614571116571022823779460950752835470391520113
0119189248659445456969881408263241988857807733444241186788650176173795738111278538048664089231111
6670398352032649300868407673959773579274098681550041794542231494345780439993535709534098372862775
9781472350963366324334832505812355887056277394521822664605842958915239046798098312981248204253998
8498211802160856050619604988202549753175804080595725408479149663132813751936679102863460635201335
2893022645685937197536591324480638936291421869333003035589964887012501598584627954873778545759770
```

```
1660480003548776361388635599184200284787732790576126541717800575984909992012529723504316502149033
3302308657503260801435454535982171679442853656182774566952396924923182294765433084988868697171623
7396868000201669610933377152133737152953915778781714490981300790256067330382748357193667515575295
6468151198399299824849371557419395088184496640936585965755956047107554849094056693419198684284828
0350580447774922616754738637913966135280765612298431632124788935874832435215821516763346571652777
6084688265574327558513539648807127829912281209682019673489191027817267548721467926285098163847777
8831517495139792751189543410881018023778053019410542577234354969472050680157273807502834211032467
7949232070984269750058808783128633847146395610845826509896468076116687143221265516675187449427166
0866800005032054052622503455930132768042103450714412669774691093860576193044919302840800632506088
5252231928742952214202131083884155002769196736829271921117561901608697438592835450953781667168433
9647380719451599848231917829645797443465580265747056178465323035121516890578737905623252104371288
3993389314608790219505130452519420960967732539387543479620679249464910473882224675858770123260511
0751696561596785252079342032338993444484986052730434094957972636719839031778138948702341781930488
9851391195527957491567222240785692085777124180392300222701691021904224071892394818587294230608799
7408239826438435829734668545570887492385127478715155562688191396785100834470425736498685957653744
9541383421812116403669800687358014897177349971495985444471613301076146773088954006953747537209299
4135008061806933812831787096430106597286403938313052776280540622762243137906808580288288740511577
0414054250270638005368179693628910170272391174057226547124400049054704344976091458139016126903666
6808857168372211352386285416442793139710707568224295682835332462395529097603291417737226364858288
0671024290980643261315762199707108223057113732134994731100798425950079925305294043028536184347655
322901634633654273940675293455160894832168899873740923632353288
```

And it's **6 014 bytes** big. Given all these partial arguments, the final and proven thesis is, Seed language is turing complete, and Mersenne Twister is able to generate program, that will be capable of simulating any algorithm's logic can be constructed.