A Very Short Self-Interpreter

Oleg Mazonka[1)] and *Daniel B. Cristofani*

21 November 2003

[1)]*Motorola Australia, GSG EDA*

ABSTRACT. In this paper we would like to present a very short self-interpreter, based on a simplistic Turing-complete imperative language. This interpreter explicitly processes the statements of the language, which means the interpreter constitutes a description of the language inside that same language. The paper does not require any specific knowledge; however, experience in programming and a vivid imagination are beneficial.

1. Introduction

For a few decades many people have been fascinated by quines -- self-reproducing programs [1]. This is not very surprising, because such programs possess a few attractive features. They have somewhat paradoxical self-referencing, which makes them not easy to write. They can be written in different languages. They have a definite minimal size in each language. And they do not have a reasonable practical meaning.

In this paper we explore the question one step further: if a quine contains the information about the program itself, then what would a program containing the information about the language it is written in look like? Such a program we here call a self-interpreter.

Writing a small quine involves choosing an algorithm which can use a coded version of itself to output both the coded and uncoded versions of itself, but which is not too long in either version. Writing a small self-interpreter involves choosing a language which is simple to parse and execute, but which is powerful enough to express these processes concisely.

Why an interpreter? Compilers are actually translators from one language into another. They reveal more about the details of the output language than about the behavior of the programs of the input language. So an interpreter is more like a pure description of the language -- that is, the correspondence between the text of programs and their semantics. We might note also that a compiler for a Turing-complete language can sometimes be written in a non-Turing-complete language, whereas an interpreter cannot.

The self-interpreter which we present in this paper was originally written by Daniel Cristofani in July 2002 and later improved and shortened to its current form.

2. Self-interpretation

Every language lives in its own environment. A definition of the language may be made in relation to an abstract machine, but any real implementation must take into account where that abstract machine is being realized. In other words there should be well-defined rules which describe how to instantiate the computational process and how to observe the behavior. In real computers those rules are compilers, interpreters, command shell, and inputs and outputs of programs. This kind of information is called the *pragmatics* of the language and is sometimes included in the full language description. However, it stands outside the language and is usually ignored in the formal description. Pragmatics is the third part of the language constitution; the other two are *syntax* and *semantics*.

A language specification is a description of an abstract machine, which can parse a text -- a sequence of symbols that represents a program written in that language -- and produce some observable behavior, i.e. output. A Turing-complete language refers to a Turing-complete abstract machine.

An interpreter is a realization of the abstract machine. It consumes a program and produces the corresponding behavior. The abstract machine can be realized in many different ways with different algorithms. The only thing the different realizations must have in common is their behavior -- the same valid program must produce the same behavior from different interpreters.

Since any interpreter constitutes a particular implementation of an abstract machine, one can use the interpreter's behavior to define the language's semantics while ignoring the interpreter's internal structure. A full language definition of this kind will also include a definition of the language's syntax and areas of undefined behavior. So any other interpreter will be a correct language implementation as long as it produces the same behavior for all syntactically correct programs whose behavior is defined.

Here we define *self-interpreter* as an interpreter which is written in the same language it interprets, and which also meets these three requirements:

1. the language must be Turing-complete;
2. the behavior of programs when interpreted by a self-interpreter must be the same as their behavior when interpreted by any other interpreter, i.e. the two must produce identical observable output for any legitimate input, though not necessarily at the same speed; and
3. the self-interpreter must not use language constructs which are designed to facilitate recognition and interpretation of these and other language constructs (*self-interpretation*), e.g. LISP's *eval*.

The second requirement looks like a tautology saying that a self-interpreter is an interpreter. Nevertheless this requirement rules out languages whose pragmatics are not sufficient to permit a self-interpreter. Later we show that not every Turing-complete language can have a self-interpreter in this sense. This property forms a distinction between languages: some languages are *environmentally complete*, while others are not. Thus, Turing-completeness comes from the semantics of the language and environmental completeness from its pragmatics.

3. The Language

For the self-interpreter implementation we have chosen the BF language [2] as a basis. This language is very simple in comparison to other programming languages. At the same time computationally it has been proven to be Turing-complete [3]. We have changed it slightly to make a self-interpreter possible, but we did not adapt it to get the best result -- to get an even shorter self-interpreter -- for two reasons. First, we did not want to depart unnecessarily from the already known and popular BF language, nor from the strictest portability practices within that language [4].

Second, in this paper we would like to present general ideas leading to a short real working example. We will discuss possible shortenings in the section 4.4.

3.1 Dbfi-BF

The language realizes an abstract machine operating on an array, unbounded on the right, of memory cells, each of which has a capacity of at least a byte and is initialized to zero. There is a pointer, which initially points to the leftmost memory cell. The machine has an input stream and an output stream of characters. The computational process consists of reading a program from input, then recognizing instructions and interpreting them.

The program which is fed as input to the machine is a sequence of characters and consists of two parts: the code and the data. These parts are separated by the first '!' character.

There are 8 instructions recognized by the machine (their codes correspond to ASCII):

- `>` move the pointer one cell to the right;
- `<` move the pointer one cell to the left;
- `+` increment the memory cell under the pointer;

- `-` decrement the memory cell under the pointer;
- `,` set the memory cell under the pointer to the ASCII value of the next character obtained from the data part of the input;
- `.` output the contents of the memory cell under the pointer as an ASCII character;
- `[` if the memory cell under the pointer is zero, go to the command following the matching '`]`', otherwise to the next instruction;
- `]` if the memory cell under the pointer is not zero, go to the command following the matching '`[`', otherwise to the next instruction.

All characters in the code, except these 8 specified above and '`!`', are ignored. Instruction '`,`' consumes a character from the data part of the input. For example, the program "`,>,!ab`" will result in the array "`97 '98 0 0 ... `" (the apostrophe represents the pointer).

After evaluating the instruction the process control goes to the next instruction for '<', '>', '+', '-', '.', ',', and stops after the last instruction, which stands before '`!`'. For instructions '`]`' and '`[`' the process control goes either to the next instruction or to just after the matching '`[`' or '`]`' instruction, depending on the cell value under the pointer.

The allowed depth of nested brackets is undefined, but any implementation must be able to handle not less than 124 (a depth of 7 is proved sufficient for Turing-completeness [5]).

If the input is finished or an input error occurred, then the input instruction must not change the value of a cell if it was zero before the input instruction; otherwise, the behavior is implementation-dependent.

There are some other features of this language which are left undefined, implying that the behavior of the interpreter is undefined in some cases. Every implementation is free to choose any behavior in such situations, and will still be counted as conforming to the language. These features are: cell size, and behavior when the program moves the pointer left from the first cell, decrements a zero, increments a maximal cell value, or has unbalanced brackets.

Simple examples of the programs are: "`,+.!a`" outputs "`b`", "`a!`" does nothing, "`,[>+>+<<-]>.>.!X`" outputs "`XX`", "`>,[.>,]<[<]>[.>]!>,[.>,]<[<]>[.>]!`" is a quine.

3.2 BF and its self-interpreter

The original BF language [6] is almost the same as described above, except that it does not define the separator '`!`'. Instead, a BF program is entirely separate from the input it processes; either the program is run through a compiler and the input is given to the resulting executable, or the program and its input are given to an interpreter through separate input streams, the program typically being read from a file and its input usually from the keyboard.

A BF interpreter written in BF cannot duplicate this behavior, because its interaction with its environment is limited; only one input stream is available to it (or to any BF program), so it must receive both the program and its input through this stream. Frans Faase's original BF interpreter in BF [7] expects them to be separated with '`!`'; dbfi does the same, being made to the same specification (including strict portability).

Only a BF dialect with this addition, or some other addition or modification to do the same job, can have a self-interpreter in our sense, because of requirement (2). The (normal) BF language cannot be used to write a BF interpreter which interacts with its environment in the same way that a normal BF interpreter does; the pragmatics of the language do not allow it.

4. Dbfi

The self-interpreter which we describe in this paper is called dbfi. It is given in its entirety in section 4.2.

4.1 The model

The dbfi interpreter consists of two parts. The first reads the code of the program to be simulated from the input and stores it in memory in coded form. The second part decodes and executes the instructions sequentially.

The memory layout for dbfi is comparatively simple. The instructions of the program to be interpreted are stored as small numbers according to the table:

```
] [ > < . - , +
1 2 3 4 5 6 7 8
```

These are stored consecutively from the start of the array, except that the simulated instruction pointer is represented as a pair of zeroes just to the left of the next instruction to execute; so initially it is at the far left, before any coded instructions.

After the code for the last instruction, there are another two zeroes, followed by the simulated data array. Each simulated cell is represented by two cells: the right one holds the value of the simulated cell, and the left one is a marker cell, set to 2 for the cell where the simulated pointer is (and for cells to the left of that), and to 0 for cells right of the simulated pointer. This makes it easy to find the cell where the simulated pointer is.

For example, if dbfi executes the program "`,[.[-],]!a`" and the simulated program is about to output an "`a`" with the period instruction, the array would look like:
`7 2 0 0 5 2 6 1 7 1 0 0 2 97 0 0 0 0 0 0 ...`

Throughout the explanation, the simulated instruction pointer and the simulated data pointer are called *the simulated IP* and *the simulated pointer*; *the pointer*, with no adjective, means the real data pointer. To distinguish dbfi codes for instructions (1-8) from ASCII codes of instructions in the program, we call the former *instruction codes*.

In conjunction with the following explanation, it would be beneficial to watch dbfi as it executes a simple program, by using some implementation that can display the array.

4.2 The result

This is the main result of this paper -- the code of dbfi:

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

4.3 Code breakdown

The first part of dbfi is concerned with setting up the initial memory layout: simulated IP (two zeroes), instruction codes, another two zeroes, and the first marker cell set to 2. The basic idea is that by assigning the codes for different instructions in reverse ASCII order, we can store the differences between the ASCII codes for the different instructions in an array, then subtract them from the input character one after another, and check for an exact match each time.

Instruction	!	+	,	-	.	<	>	[	]
ASCII	33	43	44	45	46	60	62	91	93
instruction code		8	7	6	5	4	3	2	1

So.

```
>>>+[                                                                    (1)
```

One character of the program is read each time through this loop. The pointer begins one cell to the right of the place where the instruction code for that character must go, if it's an instruction and not a comment character.

```
[-]>>[-]
```

We zero the cell we began the loop at -- it was just a flag telling us there was more input to process -- and we zero another cell which might have a value left from the previous time through the loop.

```
++>+>+++++++[<++++>>++<-]++>>+>+>+++++[>++>++++++<<-]+>>>,<++[       (2)
```

Here we make the array of differences, so the configuration is
`... ? ? 0 0 0 2 29 2 14 1 1 1 10 '32 i 0 0 ... ,`
where `i` is the cell holding the input character to process. Then we enter a loop, which we go through once per difference. Symbolically we can characterize this state as
`0 r 0 d d ... 'd i 0 0 0` , where `r` represents a reversed form of the instruction code: it's incremented each time we subtract one of the `d`'s from `i`, and when we get an exact match we subtract `r` from 10 to get the instruction code. The pointer's location is indicated by an apostrophe.

```
[>[->>]<[>>]<<-]                                                       (3)
```

This has the effect of setting `i` to `i-d` or to 0, whichever is greater. It also sets `d` to 0. The way it works is that the outer loop is executed `d` times, decrementing `d` each time; the first inner loop, which decrements `i`, is executed if `i` has not been reduced to zero already, and the second inner loop is executed if it has. The purpose of the second inner loop is to resynchronize the pointer's location, since after the first inner loop it may be in either of two places.

After this subtraction, there are three possibilities.

- If `i` is 0, it was originally less than the instruction we're now comparing it with; it was a comment, not an instruction. We then want to continue with the `d`-processing loop (2) as normal, to clear out the rest of the array of `d`'s.
- If `i` is 1, we have an exact match: `i` was originally the instruction we're checking for now. We must clear out the rest of the array of `d`'s explicitly, and process the instruction.
- If `i` is greater than 1, it was originally either a later instruction, or a comment. We continue with the `d`-processing loop as normal, but we must move `i` one cell left first.

```
<[<]<+>>[>]>
```

In any case, we first go to r and increment it.

```
[<+>-                                                                  (4)
```

If we enter this loop, `i` is at least 1. We set a flag `f` in the cell previously holding `d`, which was set to 0 in (3), and we reduce `i` by 1:
`0 r 0 d d ... d f 'i 0 0 .`

```
[[<+>-]>]                                                              (5)
```

If `i` was 2 or more, we move `i` left onto `f`, which restores the 1 we just subtracted from `i`, and we move the pointer right, to skip the next loop.

```
<[                                                                     (6)
```

If we've entered here, we are now at the flag `f` we just set; this means `i`'s original value was an exact match for the instruction we're now comparing it with.

```
[[-]<]
```

Scan left through the d's, zeroing them.

```
++<-[                                                    (7)
```

Set up "0 'r 2 0 0 ..." and subtract one from r. If r was greater than one, then i was originally not a '!' but one of the eight instructions, and we enter this loop. In that case the 2 will serve as the "process another character" flag, for the big outmost loop (1). But if r was 1, then i was a '!' and we'll skip this loop; in that case, the 2 will serve as the first marker cell.

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

Set up "9 'r 2 0 0 ..." and subtract r from 9 to give "c '0 2 0 0 ..." where c is the instruction code. Then move right two cells.

```
]>>
```

Now we either have "c 0 2 0 0 '0 ..." if the instruction was a normal instruction, or "0 0 2 '0 0 ..." if the instruction was a '!' and we skipped here from (7).

```
]
```

Now there are the same two possibilities, or "0 r 0 d d ... d i '0 0 0 ..." if we didn't find an exact match and i was moved by (5) and the loop (6) was skipped.

```
]<<
```

Now there are the same three possibilities, plus another possibility if i was zero at (4) and the loop (4) was skipped. In any case the pointer has been moved two cells left. So the possibilities now are:

1. c 0 2 '0 0 0 ... if we matched an instruction;
2. 0 '0 2 0 0 ... if we matched a '!';
3. 0 r 0 d d ... 'd i 0 0 0 ... if we didn't get a match yet, but we may later.
4. 0 r 0 d d ... 'd 0 0 0 0 ... if i is zero and we didn't get a match (character was less than the instruction we most recently compared it with, therefore a comment).

0 r '0 i 0 ... and 0 r '0 0 0 ... are special subcases of the third and fourth possibilities, if there are no d's left.

```
]<
```

If we got a match, or if there are no d's left, we're at a zero and so we exit the once-per-difference loop (2); if not, we continue processing the rest of the differences. In the third and fourth cases, each iteration of loop (2) results in reducing i by the last d, until i is zeroed; zeroing that d in the process; incrementing r; checking for a match; moving i left; all as described above. But in the fourth case, it is known that there will be no match and r will not be used; loop (4) will be skipped each time, and the whole process is just a slow way of clearing out the remaining d's.

Once we've exited the loop the possibilities are:
c 0 '2 0 0 0 ... if we matched an instruction (whose code is now in c);
'0 0 2 0 0 ... if we matched a '!';
0 'r 0 i 0 ... if we got through all the differences without matching anything. i may or may not be 0.

```
]<
```

If we matched an instruction, or didn't match anything, we execute the input-character-processing loop (1) again: in either case we're at a nonzero flag just right of where the next instruction's code should go. But if we matched a '!', we're done: we're at a 0, so we leave the input-character-processing loop (1). In that case, the 2 is in the right place to serve as the first marker cell, and we step left to the rightmost instruction code.

```
[                                                        (8)
```

Now we're done with the whole program-reading loop, and ready to execute the simulated program. On each

pass through the main loop dbfi executes one simulated instruction. This loop tests the cell that initially contains the last instruction code -- the code for the instruction preceding the '!' in the program. When that instruction has been executed and moved to the left, past the simulated IP, the test will fail and the loop will terminate.

```
[<]>[[>]>>[>>]+[<<]<[<]<+>>-]                                    (9)
```

The way dbfi decodes an instruction code and executes the correct instruction is slightly indirect. First, we go to the simulated IP. We want to advance the simulated IP past the next instruction code -- e.g. to change "`2 0 0 4 1`" into "`2 4 0 0 1`" and since the simulated IP is made of zeroes, all we have to do is move the instruction code two spaces left. But simultaneously, we want to make a copy of the instruction code we're moving in an altered form: as a string of ones, in the marker cells to the right of the simulated pointer. (The reasons for this will become apparent soon.) Each time through the long loop just above, we scan through the simulated code and data "`[>]>>[>>]`" past the simulated pointer and set another marker cell to 1, then scan back "`[<<]<[<]`" to the simulated IP and move another unit of the instruction code two cells to the left; this continues until the entire instruction code has been moved two cells left, past the simulated IP; at that time the length of the string of ones in the marker cells equals the instruction code that was moved.

For example, if the simulated array has "`97 '98 0 0 0 ...`" and the interpreter is about to execute '<' instruction in the piece of code "`[<]`", then the real array may look like:

```
2 0 0 4 '1 0 0 2 97 2 98 0 0 0 0 0 0 0 0 0 0 ... before and
2 4 0 '0 1 0 0 2 97 2 98 1 0 1 0 1 0 1 0 0 0 ... after (9).
```

```
>[>]+[->>]
```

Second, we go to the gap between simulated code and simulated data, then scan right through the marker cells of the simulated data array; those up to the simulated pointer are changed from two to one, those in the coded instruction string are changed from one to zero, and we stop at the first marker cell to the right of that string. Continuing the previous example, this move changes

```
2 4 0 '0 1 0 0 2 97 2 98 1 0 1 0 1 0 1 0 0 0 ... into
2 4 0 0 1 0 0 1 97 1 98 0 0 0 0 0 0 0 0 '0 0 ...
```

Now the pointer is *n* +1 marker cells to the right of the simulated pointer (the rightmost nonzero marker cell), where *n* is the instruction code to execute. We can move the pointer left one marker cell at a time, checking at each step to see whether we've hit the simulated pointer; how soon we find it will tell us what instruction to execute. This whole construction serves essentially the same purpose as a case statement. We do the tests with eight case loops, each containing the code for executing a particular instruction. Each case also leaves the pointer where it would have been had the instruction code been 9 -- that is, just far enough right to skip the subsequent cases -- except three cases that transfer control to later cases as a shortcut. These three will be explained individually.

```
<<<<[
```

If we enter this case loop, it's because the instruction code to execute was 1, i.e. ']', and moving the pointer 2 marker cells left was just enough to bring it to the simulated pointer -- the rightmost marker cell which is not zero.

The basic method for moving the simulated IP to the matching bracket is the same for '[' and ']'. We move instruction codes past the simulated IP one at a time, while counting the depth of the brackets (keeping the count in one cell of the simulated IP). When that depth reaches zero, we've just passed the matching bracket. We have to start just inside the brackets, with a depth of 1.

```
[<<]<[<]+<<
```

We go to the simulated IP. The code for the ']' we're executing has already been moved left, in the normal process of advancing the simulated IP; so the local configuration is "`1 0 '0`". The configuration we want is "`'1 0 1`", where the first 1 represents the depth (stored in the left cell of the simulated IP) and the second represents the current ']' instruction, moved back to the right to put the simulated IP inside the loop. The code to achieve this is just "+<<" but it also involves a little mental shift; the ']' at the left is now reused as the depth counter, without actually changing the cell's value.

```
[                                                                    (10)
```

This is the main instruction-moving loop, which continues until the depth counter becomes 0. Instruction codes are moved two cells right; the depth counter is incremented for each ']' and decremented for each '['. We use nested loops to adjust the depth based on the instruction code.

```
+>+<<-[                                                              (11)
```

The depth counter is incremented, which is appropriate if the instruction is ']'. One unit of the instruction code is moved right. If that zeroes the instruction code, it was indeed ']', so we skip past the rest of the processing.

```
>-->+<<-[                                                            (12)
```

The depth counter is decremented twice -- once to undo the previous increment, and once to decrement it further as is appropriate if the instruction is '['. A second unit of the instruction code is moved right. If that zeroes the instruction code, it was a '[', so we skip past the rest of the processing.

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

If we get here, the instruction code was more than 2, so it was a non-bracket instruction. We increment the depth counter to undo the previous decrement, then move the rest of the instruction code right.

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

This is where we skipped to from (11) or (12), if the instruction was '[' or ']'. The depth counter is now correct in any case. We move it to the left, and test it to see whether it's zero yet. If not, we continue the instruction-moving loop (10).

```
++>>-->[>]>>[>>]]
```

Now the simulated IP is just left of the matching '[' instruction, and our job is done. We could just go back to the right, and leave the pointer eight marker cells to the right of the simulated pointer, so the next seven case loops would be skipped. But instead we'll do something more fun: we'll leave the pointer one marker cell to the right of the simulated pointer, so we'll enter the next case loop and execute the '[' instruction immediately, without having to go through the whole instruction-fetch process another time. Naturally, we move the '[' instruction to the left of the IP first, since that's where the normal instruction-decoding process would have left it.

```
<<[
```

This is the case for the '[' instruction.

```
>>+
```

We're going to do a similar trick here: after executing this '[', we're going to leave the pointer two marker cells right of the simulated pointer, instead of seven, so control is transferred to the '<' case. To counteract that case's effects, we'll have to perform the equivalent of a '>' instruction before we leave this case, i.e. we'll have to move the simulated pointer right by setting an extra marker cell to 1; and as it happens, the easiest time to do so is right now.

```
<[[<]<]>[                                                            (13)
```

Now we go to the value cell of the simulated cell at the simulated pointer's previous location. If the value cell holds zero, then the first loop is skipped, we go to the new marker cell, and we enter the big loop in which we move the simulated IP. But if the value cell is nonzero, we go to the first of the two zeros that separate the simulated code from the simulated data; then we go to the second of those zeros, and skip the big loop.

```
[<<]<[<]+
```

We go to the simulated IP; we're inside the '[' already, so we just set the depth to 1.

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

This instruction-moving loop is much like the one above, in the ']' case. When it finishes, the pointer is now in the second zero cell of the simulated IP, and the simulated IP is just right of the matching ']', as it should be.

```
>[>]>]
```

Now we go to the second of the zeros separating the simulated code from the simulated data, and exit this big IP-moving loop; so the pointer is there whether this loop was executed, or skipped from (13).

```
>[>>]>>]
```

We leave the pointer two marker cells right of the new (temporarily) nonzero marker cell, so we'll enter the case for '<', which will zero that marker cell again.

```
<<[>>+>>+>>]
```

The other six cases, for the other six instructions, are very short. This one is '>', so it should set another marker cell to 1. But it takes a shortcut similar to those mentioned before: it sets two marker cells to 1, then passes control to the '<' case which zeroes one of them.

```
<<[->>>>>>>>>]
```

The case for '<' zeroes the last marker cell and then bypasses the next four cases. The '[' and '>' cases also shortcut to here, and the ']' case shortcuts to '[' which shortcuts to here.

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

Each of these four cases operates directly on the value cell of the simulated cell corresponding with the simulated pointer, then skips subsequent cases.

```
<<[+<<]<]
```

This is where all the paths converge. Whatever happened above, the pointer will be brought back to the simulated pointer just in time to enter this loop, and scan left, changing the marker cells leading up to the simulated pointer from ones back to twos; then the pointer is moved back to the initial location of the rightmost instruction code, for the test of the main instruction-executing loop (8).

4.4 Possible shortenings

There are several obvious possibilities for making this self-interpreter shorter. The first part of the interpreter would be almost trivial if one used a different instruction encoding (not ASCII). It would also be possible to shorten the first part if the language were changed to forbid other characters (comments) in the code of the program. The language also leaves behavior undefined in cases like decrementing zero or incrementing the maximal cell value, so the interpreter spends some extra instructions to avoid such behavior.

In this paper we did not pursue the aim of getting the shortest possible self-interpreter by any means. Since many people know and acknowledge the BF language, we tried to avoid restrictions and modifications of this language.

5. Conclusion

In this paper we have presented a very short self-interpreter. It is not trivial or even simple, but of course no one would expect great simplicity in simulating a Turing-complete machine. We do not deny the possibility of an even shorter self-interpreter. One way to get a shorter self-interpreter is to find more shortcut tricks or a better algorithm. Another way is to invent a different language allowing for the construction of a shorter self-interpreter. The interesting point here is that such a language must be simple enough to keep its (self-)

description short, and at the same time it must be rich enough to express its description in a concise manner. In other words, the self-interpreter of a complex language will be long because of the large number of language features, and the self-interpreter of a very simple language may be long because of poor expressiveness. In this paper we have also shown that not every Turing-complete language can have a self-interpreter.

Appendix

This interpreter written in C++ is an example of a dbfi-BF interpreter. It is not optimized for speed or memory. Its specifications are the following. The memory cell is equal to a byte. Decrementing zero or incrementing a byte's maximal value is wrapped. The behavior is undefined for input with unbalanced brackets. Bad input does not change the cell value (as C++ function istream::get). And there is no hardcoded memory limit on the data array.

```
#include <iostream>
#include <string>
#include <map>

int main() {

     std::string c;
     std::map<int,char> m;

     std::getline(std::cin,c,'!');

     int p = 0, l;

     for( int i = 0; i < c.size(); i++ ){
       l=1;
       if( c[i] == ']' && m[p] ) while(l){ i--;l+=(c[i]==']')-(c[i]=='['); }
       if( c[i] == '[' && !m[p]) while(l){ i++;l-=(c[i]==']')-(c[i]=='['); }
       if( c[i] == '+' ) m[p]++;
       if( c[i] == '-' ) m[p]--;
       if( c[i] == '.' ) std::cout << m[p];
       if( c[i] == ',' ) std::cin.get(m[p]);
       if( c[i] == '>' ) p++;
       if( c[i] == '<' ) p--;
     }
}
```


Acknowledgements

This work was funded and supported by pure enthusiasm of the authors and their love and devotion to the beauty of intellectual art.

References

1. See quines on internet, for example, http://www.nyx.net/~gthompso/quine.htm

2. The BF language was invented by Urban Mueller in 1993 and its original name is brainfuck. Probably its name condemned the language to be spread on internet only and never was published before. The language attracts by its extreme simplicity. It seems that it may be of practical use in some research areas, for instance, AI or Kolmogorov complexity.

3. http://home.planet.nl/~faase009/Ha_bf_Turing.html
http://www.hevanet.com/cristofd/bf/bf.html#turing

4. http://www.muppetlabs.com/~breadbox/bf/standards.html
http://www.hevanet.com/cristofd/bf/epistle.html
Note that dbfi follows the specification of Frans Faase's original BF interpreter in BF

(http://home.planet.nl/~faase009/Ha_bf_inter.html), which is quite portable in most respects. (Its only departure from portability is that the number of cells available to simulated programs is limited by the capacity of the cells of the underlying implementation; dbfi does not have this limit.) Faase's interpreter originated the use of '!' as a code-data separator.

5. http://www.hevanet.com/cristofd/bf/bf.html#turing

6. See, for instance:
the canonical BF distribution at http://wuarchive.wustl.edu/pub/aminet/dev/lang/brainfuck-2.lha
Frans Faase's page at http://home.wxs.nl/~faase009/Ha_BF.html
Brian Raiter's page at http://www.muppetlabs.com/~breadbox/bf/
and Daniel Cristofani's page at http://www.hevanet.com/cristofd/bf/

7. http://home.planet.nl/~faase009/Ha_bf_inter.html