Annals of the University of Craiova, Mathematics and Computer Science Series
Volume 41(2), 2014, Pages 271–279
ISSN: 1223-6934

Socket extensions for esoteric languages

Radu Dragoş and Diana Haliţă

Abstract. In this paper we have advanced the first interpreter for the brainfuck (BF) esoteric programming language, entirely written in awk, a fast text processing programming language. However, the main objective remained introducing socket communication to brainfuck programming language. In order to achieve that goal, we have improved the brainfuck language with a byte long instruction through which it is allowed socket communication. For that, we have advanced a series of procedures in order to test the compatibility of our brainfuck interpreter.

Moreover, we have maintained brainfucks minimalism, which is one of the special characteristics of this programming language. In the end, we succeeded to map a minimalistic programming language to the client-server paradigm.

2010 Mathematics Subject Classification. **68N15** - Programming languages, **68N20** - Compilers and interpreters.
Key words and phrases. brainfuck, socket, interpreter, esoteric languages.

1. Introduction

This paper describes a simple interpreter for the brainfuck (BF) esoteric programming language, written in AWK. It is compatible with almost any version of AWK that is supplied within UNIX systems. The interpreter illustrates the use of AWK in implementing small programs using BF language, mainly for proof of concept reasons.

1.1. Motivation. Even if BF is known for its extreme minimalism and it is designed to challenge programmers, it is not usually suitable for practical use. However, the motivation of developing such an application comes from the usage of a simple and easy syntax which can help programmers to deeply understand other programming languages and besides that, other programming paradigms.

The idea of improving BF language came from the necessity of implementing one of the most fundamental technologies of computer networking: socket communication.

Most computer programming languages implement socket communication. Including functional programming languages such as Lisp, Haskell or Erlang and logical programming languages such as Prolog ([1], [2], [3], [4]).

1.2. Objectives. The main objective is to introduce socket communication to BF programming language. In order to maintain BF's minimalistic language properties, we will define a single one byte new BF instruction.

In order to add socket support and to accommodate the new instruction, a BF interpreter needed to be modified. However, we decided not to modify an existing version, but to implement a new one using AWK programming language. Therefore,

Received August 13, 2014.

we will show how to map a minimalistic programming language to the client-server paradigm.

2. Related Work

2.1. Esoteric languages. An esoteric programming language is a programming language best suited for testing the boundaries of computer programming language design, as a proof of concept, as software art, or as a joke. Even if their goal is not the usability of programming, the main purpose is to remove or replace conventional language features while still maintaining a language that is Turing-complete.

A few examples of esoteric programming languages are: Intercal, Befunge, P", Brainfuck [5].

P" is a primitive computer programming language created in 1964 to describe a family of Turing machines. This language is formally defined as a set of words on the four-instruction alphabet R, λ, (,).

P" was the first "GOTO-less" imperative structured programming language to be proven Turing-complete. The BF language (apart from its I/O commands) is a minor informal variation of P".

2.2. Brainfuck. BF esoteric programming language is, as we previously stated, a minimalistic language, which consists in eight (one byte long) simple commands. This programming language was created as a Turing-complete language [6], meaning that, theoretically speaking, having access to an unlimited amount of memory, BF is capable of computing any function or simulating any mathematical model. All BF commands are sequentially executed and generally all other characters are ignored. The execution of the program ends at the end of the instructions set. Bellow there are few examples of code written in BF:

2.2.1. *Hello World - version 1.* This is the original "Hello World!" program where the ASCII codes for each printable character in the text are generated by running the code.

Listing 1. Hello World - version 1

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

2.2.2. *Hello World - version 2.* This is a more lisible code example that will output "hello" if you input letter "h". It will use the ASCII code of character "h" to obtain the ASCII code of "e", "l" and "o" by decrementing or incrementing the value at the current memory location.

Listing 2. Hello World - version2

```
,.---.+++++++..+++.
```

2.2.3. *Comparing numbers.* In this example we read from stdin 2 numbers (a and b) and the character "0". The program will output "0" if $a \leq b$ and "1" if $a > b$.

The algorithm works by setting a number of a memory locations to value 1. Then, b of them are set to 0. If there remains locations set to 1, it means that $a > b$.

TABLE 1. BF Commands

Character	Meaning
>	increment the data pointer which will point to the next cell to the right;
<	decrement the data pointer which will point to the next cell to the left;
+	increment the byte at the data pointer;
-	decrement the byte at the data pointer;
.	output the byte at the data pointer;
,	accept one byte of input and store its value in the byte at the data pointer;
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

LISTING 3. Comparing numbers

```
>>,                    #move to location 2 and read value of a
[-[>+<-]+>]            #fill a locations with 1
<[<]                   #go back to location 1
,                      #read value of b
[>[-]<-[>+<-]>]        #fill b locations with 0
                       #if location b plus 1 is 0 then a was
,>[<+.[-]>[-]]<.       #less than b and print "0"
                       #else print "1"
```

2.3. Related BF implementations. There is a large collection of BF implementations, most of them listed in [7].

The BF implementations include:

- interpreters that use other programming languages to take BF programs as input and execute commands in that specific programming language. There are even some BF interpreters written in BF;
- compilers that take BF source code and generate executable code;
- hardware implementations including emultated and real CPU-s that can run BF code directly.

Most of those implementations are pure proofs of concept just to show that BF can be implemented using some particular technology. Some of them are optimizing implementations created to run BF code as fast as possible. Some of them are notable intentions to implement support for network communications.

The BF++ project [8] aims to extend BF with file I/O and TCP network communication. Some specifications are given but the project was discontinued and no final implementation was provided.

NetFuck [9] introduces communication between two NetFuck programs over TCP. The remote host and port are given to the interpreter as command line arguments and cannot be specified within the BF code.

Another step forward toward network communication using BF was taken by another NetFuck interpreter written in C# [10] which redirects standard BF I/O commands (.,) to read/write into a TCP socket instead of standard I/O. The developer also provides an IRCbot client written in NetFuck. However, the interpreter requires the IP and remote port to be specified in the command line as arguments, and by redirecting I/O to the connected TCP socket, no further interaction with the user is possible.

Two distributed applications using BF are presented below.

In [11] is presented a BF module for a particular webserver implemented in PHP. The module allows a developer to write server-side BF code and the result will be displayed on the clients browser. The communication is performed through the web server, so no socket facilities are provided for BF.

In [12], a NodeJS webserver implementation is extended to support writting BF code as scripting language for client side web programming in web browsers.

Therefore, we choose to implement our own BF interpreter with general purpose networking support using AWK programming language since there are none known BF implementations using AWK, and because AWK has a simple to use and understand socket abstractization layer.

3. AWK

The main purpose of AWK programming language was to allow users to write short programs intended for fast text processing and generating reports. AWK code is intended to be executed for each line of each processed file after applying some input regex filters. The processing speed of various AWK implementations versus alternatives such as Perl, Python or Ruby is beyond the scope of this document. However, AWK is present as a main programming/scripting language in most UNIX-like operating systems and therefore, it is studied in computer science curriculums all around the word. While a lot of students study AWK programming and most system administrators use it on a daily basis, few of them are aware of it's networking capabilities.

Our main concern was to write an efficient program through which we can access network services. But AWK was not meant initially to be used for networking purposes and it does not introduce special functions for socket access, like other languages do. However, it treats network connections like files [13]. The special file name for network access is made up of several mandatory fields, such as:

/inet/protocol/localport/hostname/remoteport

Network communication in AWK is implemented by creating a pipeline between the main AWK process and a process over the network specified as a special filename.

So, it was necessary to introduce a new operator '|&' to make possible communication over a network.

An example of reading a line from a tcp server (i.e. the ssh server on localhost listening on port 22 and presenting the ssh server welcome message) is presented below:

$echo | AWK '{"/inet/tcp/0/localhost/22" |& getline a; print a}'
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.3

Another example is how to write a message ("hello") to a UDP server using a simple AWK command:

$echo | AWK 'print "hello" |& "/inet/udp/0/localhost/2222"'

4. AWK INTERPRETER FOR BF

BF is a minimalistic esoteric programming language. Hence, as shown in the previous section, there are many interpreters and compilers for BF in most programming languages, including a BF interpreter written in BF. However, there is none written in AWK.

Our first task was to write a BF interpreter in AWK that can run any BF program. There are several procedures for testing the compatibility and the performance of BF interpreters, some of them including recursive execution of BF code using the BF interpreter written in BF [14]. It is our main goal to write a compatible AWK interpreter for BF, while the performance is not our concern for the moment.

Along with a lot of trivial BF programs that we tried out in order to test our implementation, we also run a BF program that generates an ASCII Mandelbrot set. Although it runs slower than the default Ubuntu BF interpreter, it runs successfully as it can be seen in the Figure 1.

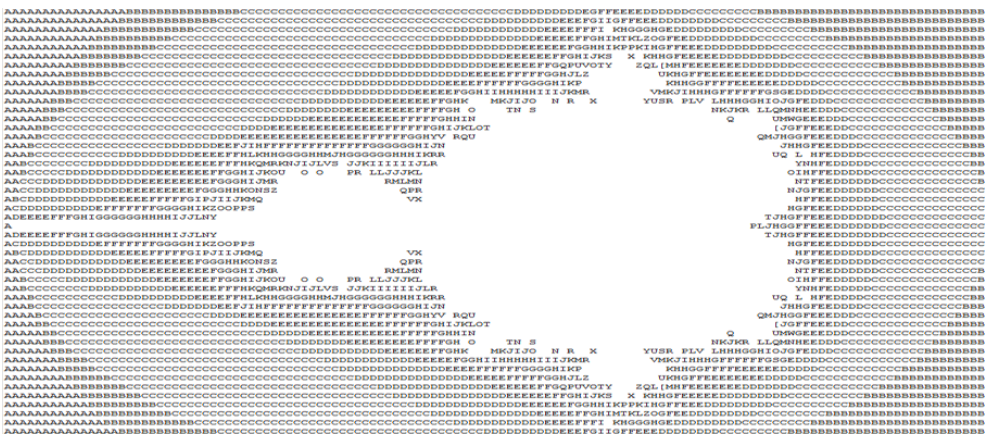

FIGURE 1. Output of the BF Mandelbrot set generator

Although AWK is a fast text processing programming language [15], we will piggyback here it's network communication abilities.

At first, the AWK interpreter reads all commands from the BF source file, then, they are parsed sequentially or repetitive for the "[","]" pair of instructions. The pseudocode is presented in Listing 4.
To maintain the minimalism of BF, we introduce only one new instruction, called @.

Depending on the interpreter which is usually used, one may consider this new instruction (i.e. our AWK interpreter) or ignore it, considering that it is a comment (as regular interpreters do). In order to make possible the client-server communication, we need to define some rules which specify the protocols that should be followed.

Listing 4. Pseudocode for interpreting network capable BF

```
// array m used for storing 30000 bytes long virtual BF tape
// c indicates current tape position
READ source code into array v
FOR i in v
    CASE v[i]
        ",":    READ a byte into m[c]
        ".":    WRITE m[c]
        "+":    Increment m[c]
        "-":    Decrement m[c]
        ">":      Increment tape pointer c
        "<":      Decrement tape pointer c
        "[":    IF m[c] not 0
                          THEN execute instructions up to
                               corresponding "]"
                          ELSE GOTO corresponding "]"
                ENDIF
        "]":    IF m[c] not 0
                          THEN GOTO corresponding "["
                ENDIF
        "@":    CASE m[c+1]
                    0:  Listen TCP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and WRITE m[c] when client READS
                    1:  Listen TCP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and READ m[c] when client WRITES
                    2:  Connect to TCP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and WRITE m[c] when server READS
                    3:  Connect to TCP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and READ m[c] when server WRITES
                    4:  //not used because UDP server cannot WRITE
                        //before READ
                    5:  Listen UDP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and READ m[c] when client WRITES
                    6:  Connect to UDP on port m[c+2]*1000+m[c+3]
                        and IP address m[c+4].m[c+5].m[c+6].m[c+7]
                        and WRITE m[c] when server READS
                    7:  //not used because UDP client cannot READ
                        //before WRITE
                ENDCASE
    ENDCASE
ENDFOR
```

Memory pointer	Meaning
c+0	current memory location
c+1	case byte
c+2	port high byte
c+3	port low byte
c+4	IP byte 1
c+5	IP byte 2
c+6	IP byte 3
c+7	IP byte 4

TABLE 2.
Specifications for implementing @ command

Values m[c+1]
0 = (server, send, TCP)
1 = (server, receive, TCP)
2 = (client, send, TCP)
3 = (client, receive, TCP)
4 = (server, send, UDP)
5 = (server, receive, UDP)
6 = (client, send, UDP)
7 = (client, receive, UDP)

TABLE 3.
Cases for values of byte m[c+1]

4.1. Specifications for instruction @. This new instruction will allow the user to send/receive a single byte through a socket. The instruction will send/receive the byte referred by the current memory pointer (c). The type of the socket is defined by the value of c+1 as described in Table 2. The byte values at c+2 and c+3 are used to define the local or remote port foc communication, and bytes c+4 to c+7 represent the remote IPvP address (as described in Table 2).

The second byte (i.e. m[c+1]) can have only eight possible values, otherwise it will be ignored, and the communication will not be able to take place.

It is well known that two processes can be either client or server, they can either send or receive messages following one of the TCP or UDP protocols. Depending on the chosen combination from the set obtained by making the cartesian product between the above named sets: {client, server} x {send, receive} x {TCP, UDP}, the second byte should have only the values described in Table 3.

4.2. Examples of BF network communication.

4.2.1. *TCP client.* The first example is a TCP client that will read the first byte of a SMTP server welcome message running on localhost (127.0.0.1) on port 25, and outputs on the screen the value received in the current memory location:

LISTING 5. TCP client

```
>+++>                                  #value of m(c1) set to 3
                                       #for TCP client receive
+++++[>+++++<-]>>                      #setting m(c3) port to
                                       #25 multiplying 5 by 5
>>+++++[<++++[<++++++++>-]>-]<<-       #setting m(c4) first
                                       #octet of IP address
                                       #to 127=4*4*8 minus 1
>>>+                                   #setting octets m(c5) to
                                       #m(c7) to 0 and 0 and 1
                                       #move back to m(c)
<<<<<<<@.                              #execute socket command
                                       #and output the byte
                                       #read
```

The program prints on the screen the character "2" which was the first byte read from the server welcome message:

"220 athena.ubbcluj.ro ESMTP Postfix (Ubuntu)"

4.2.2. *UDP client - server.* The second example is a pair of BF programs that communicate with each other using UDP port 2000:

Listing 6. BF UDP server

```
>+++++>                                 #value of m(c1) set to 5
                                        #for UDP server receive
++>>                                    #setting m(c3) port to
                                        #2000 (high octet 2,
                                        #low octet 000)
>>++++[<++++[<++++++++>-]>-]<<- #setting m(c4) first
                                        #octet of IP address
                                        #to 127=4*4*8 minus 1
>>>+                                    #setting octets m(c5)
                                        #to m(c7) to 0 and 0
                                        #and 1
                                        #move back to m(c)
<<<<<<<@.                               #execute socket command
                                        #and output the byte
                                        #read
```

Listing 7. BF UDP client

```
,                                       #read a byte from
                                        #standard input
>++++++>                                #value of m(c1) set to 6
                                        #for UDP client send
++>>                                    #setting m(c3) port to
                                        #2000 (high octet 2,
                                        #low octet 000)
>>++++[<++++[<++++++++>-]>-]<<- #setting m(c4) first
                                        #octet of IP address
                                        #to 127=4*4*8 minus 1
>>>+                                    #setting octets m(c5) to
                                        #m(c7) to 0 and 0 and 1
<<<<<<<@                                #move back to m(c)
                                        #execute socket command
```

4.3. Limitations. Although, a lot can be accomplished by using an application layer protocol that transmits only byte sized payload packets, it is not an efficient use of resources. This implementation can only transmit/receive maximum one byte of information for every @ instruction used. This was our choice for this proof of concept implementation in order to keep the BF extension minimalistic.

The interpreter could be easily changed to send/receive variable sized data packets by using another byte (or two) in the memory to specify packet sizes up to 265 (65535) bytes. But this would greatly increase the already high complexity and readability of BF code.

5. CONCLUSIONS AND FUTURE WORK

The goal of writing a BF interpreter and extending BF for client-server communication has been realised in the most minimalistic possible way.

We hope that what we realised was indeed a proof of concept and that our ideas will inspire others to learn how to program in AWK.

As future directions for research and development we are considering implementing (and testing) variable packet size network communication in BF in such a way so to

maintain the minimalistic and esoteric principles of BF. Also, based on the research done for this paper we could also extend BF for running code in parallel so it could be used as a didactic tool for understanding the non-sequential programming paradigm.

References

[1] Peter Seibel, *Practical Common Lisp*, Apress, 2005.

[2] J. Armstrong, Making reliable distributed systems in the presence of software errors, A Dissertation submitted to the Royal Institute of Technology in partial fulfilment of the requirements for the degree of Doctor of Technology The Royal Institute of Technology Stockholm, Sweden (2003).

[3] G. Hutton, *Programming in Haskell*, Cambridge University Press, 2007.

[4] J. Wielemaker, Z. Huang and L. van der Meij, SWI-Prolog and the Web, *Theory and Practice of Logic Programming* **8** (2008), no. 3, 363–392.

[5] M. Mateas and N. Montfort, A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics, *Proceedings of the 6 th Digital Arts and Culture Conference*, IT University of Copenhagen, 1-3 Dec 2005, 144–153.

[6] C. Bohm and G. Jacopini, Flow diagrams, turing machines and languages with only two formation rules, *Communications of the ACM* **9** (1966), no. 5, 366–371.

[7] `http://esolangs.org/wiki/Brainfuck_implementations`, last accessed on: 15.12.2014

[8] `http://www.jitunleashed.com/bf/index.html`, last accessed on: 15.12.2014

[9] `https://bitbucket.org/shadwick/netfuck/wiki/Home`, last accessed on: 16.12.2014

[10] `https://github.com/SirCmpwn/bf-irc-bot/blob/master/irc-bot.bf`, last accessed on: 16.12.2014

[11] `http://nanoweb.si.kz/manual/mod_bsp.html`, last accessed on: 16.12.2014

[12] `https://github.com/masylum/node-brainfuck`, last accessed on: 16.12.2014

[13] J. Kahrs, Network Administration with AWK, *Linux J.* **1999** (1999), no. 60, art. 5.

[14] `http://www.nada.kth.se/kurser/kth/2D1464/awib.pdf`, last accessed on: 15.12.2014

[15] IEEE Software Staff, Improving Productivity and Quality with Awk, *IEEE Software* **7** (1990), no. 2, 94–95.

(Radu Dragoş, Diana Haliţă) Babeş-Bolyai University, Faculty of Mathematics and Computer Science, 1 M.Kogălniceanu st., 400084 Cluj-Napoca, România
E-mail address: `radu.dragos@cs.ubbcluj.ro, diana.halita@ubbcluj.ro`