`br++`: A MUCH NEEDED EXTENSION TO `brainfuck`

CÉDRIC HO THANH

Abstract. TODO: Write the abstract.

1. Introduction

In the year of the lord 2022, Urban Müller's celebrated `brainfuck` langage is used in numerous industries. For example, my net banking app is written in `brainfuck` so it's super dooper fast and reliable. It is no surprise then that `brainfuck` developers are highly sought after in all branches of the software development industry, with an normalized average median income of \$0.5.

Nonetheless, after almost 30 years of continued use, `brainfuck` starts to show its age. Many trendy paradigms and buzzwords are absent from the langage, such as "fullstack", "noSQL", "blockchain", and even "equal pay". In this paper, we introduce `brainfuck++` (hereafter `br++` for short), an extension that adds modern constructs to `brainfuck`. While we do not claim to address all the selling points above, we are confident this update lays some robust foundations to tackle them efficiently in ulterior works. Just like we tackle this segway to our sponsors.

Acknowledgments. Thank you to HardBoiledEgg for sponsoring this episode of UNIX Tech Tips. HardBoiledEgg retails all your favorite tech goodies, from old CPUs to faulty motherboards, and offers a premium 24/7 egg yolk-based customer support. They amazing return policy makes sure to call you out when you return a defective product, and even when you don't. So what are you waiting for? Don't let your government have all the fun and start getting egg-screwed today with HardBoiledEgg. Check them out with the link in the description below. You've been reading this entire spot with Linus's voice stop lying.

2. Vanilla extract `brainfuck`

`brainfuck` is a surprisingly simple langage. The execution environment (or BVM) consists of an infinite array of 8-bits bytes[1] A called the *tape*, and a data pointer $p \in \mathbb{N}$. The data pointer and the cells of A are initialized to 0. The langage consists of 8 keywords, echoing the bit size of bytes (8) in a heavenly synergy only fathomable by the most neurologically endowed individuals.

Date: March 2022.

Key words and phrases. Programing langage, brainfuck, concurrency, machine learning.

[1] historically, a fixed 30 000-cell array, but memory is cheap nowadays eh?

Keyword	Semantics
`>`	$p \leftarrow p + 1$
`<`	$p \leftarrow p - 1$
`+`	$A_p \leftarrow A_p + 1$
`-`	$A_p \leftarrow A_p - 1$
`.`	Outputs A_p
`,`	Inputs 1 byte into A_p
`[`	Noop woop woop
`]`	If $A_p \neq 0$, then jump to the matching `[`, skip o.w.

If at any point in the execution, the data pointer p becomes negative and < 0, then a `DataPointerUnderflowError` exception is raised. Likewise, if $p \geq \infty$, then a `DataPointerOuttaHereError` exception is raised. It is common practice to raise these twice, in case the first one is lost. For convenience, white spaces, carriage returns and tabulations are accepted by the interpreter, but have a noop woop woop semantics. Technically, non-well bracketed programs are valid, but if a `]` incurs a jump to a non-existing `[`, a generic kernel panic is initiated (twice).

Here is a simple `brainfuck` program that prompts for 2 bytes (or reads them from STDIN) and outputs their sum:

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

Theorem 2.1. *`brainfuck` is Turing-complete.*

Proof. The following program writes `Turing` on the tape so we should be good.

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + > + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + > + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + > + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + > + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + > + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + +
```

□

3. BRAINFUCK++

We now build upon section 2 and provide a complete specification of `br++`. Buckle up pleb.

3.1. Headers. A `br++` program may start with a *header*, which is just a sequence of keywords specifying various properties and runtime parameters.

- **Bigbyte.** The `B` header keyword specifies that the tape cells are bigbytes instead of traditional 8-bits bytes. A bigbyte is 9-bits big. [2]
- **Root privileges.** The `R` header keyword indicates that the program should run with root privileges. The BVM escalates using standard tools such as dirtyc0w, Pegasus, Dirty pipe, phishing the system administrator, $5 wrench, etc.
- **Archlinux support.** If `A` is specified in the header, the program will output the string `I use arch btw.` (with a newline) everytime the code pointer moves.
- **Non-horizontal semantics.** Instead of being a horizontal array, the `|` transforms the tape into a vertical one. The keywords `>` and `<` are replaced by `^` and `v` respectively.
- **Unicode support.** Classically, `brainfuck` program files are encoded in ASCII. The 8 directive informs the BVM that the file is encoded in UTF8.
- **Online assistance.** The `H` header enables the online assistance facilities. When an exception is raised, the BVM opens StackOverflow and queries the exception type and any accompanying error message.
- **True concurrency.** The `C` header enables true concurrency. See section 3.14.

3.2. **Comments.** Readability is key to write readable code. For that reason, this specification carefully specifies what comments are: they're like in Python. Aditionally, because readability is so crucial to producing high-quality code, the `br++` interpreter will reject programs that do not include at least one *helpful* comment[3], raising a `CodeUnreadableError` exception.

3.3. **Cell packing.** Even with the vast possibilities offered by the potent bigbyte construct, `br++` offers a paradigm to manipulate large data values. This is called *cell packing*. Cell packing is accomplished with the `p` keyword. When the code pointer encounters `p`, the integer value x of the current cell is read, and the following x cells are considered *packed*.

The values of a packed cell is the binary concatenation of all values of all packed cells to its left (or underneath if the tape is vertical). The following example is fairly self-explanatory:

```
+ + +       # |   3 |   0 |   0 |   0 |   0 |
            #     ^
p           # |   3 |   0 :   0 :   0 |   0 |
            #     ^  The next 3 cells are packed
> > >       # |   3 |   0 :   0 :   0 |   0 |
            #                         ^
-           # |   3 | 255 : 255 : 255 |   0 |
            #                         ^
.           # Outputs 2^24 - 1 = 16777215
<           # |   3 | 255 : 255 : 255 |   0 |
            #                   ^
.           # Outputs 2^16 - 1 = 65535
-           # |   3 | 255 : 254 : 255 |   0 |
```

[2]A bigbite in my bigmac is about 25% of the burger, meaning I can eat it in about 4 bigbites. Comment down below with your high score!

[3]The precise semantic of *helpful* is left at the discretion of the implementation.

```
      #        ^
.     # Outputs 2^16 - 2 = 65534
<     # |   3 | 255 : 254 : 255 |   0 |
      #     ^
.     # Outputs 255
```

If a 0-cell packing is attempted, a `PackingEmptyOniichanNoBakaError` exception is raised.

3.4. **Convention: floating point numbers.** In `br++`, *single precision floating point numbers* are simply sequences of 4 packed cells. For better buoyancy, when working with bigbytes, the high 4 bits of the first cell are considered as padding. For example, the *Nice constant* on tape would look like this `|  66 : 139 :  97 :  72 |`.

3.5. **String literals.** A string literal starts and ends with the keyword `"` and can contain any ASCII character verbatim. The character `"`, however, must be escaped as `\"`, and the backslash `\` by `\\`. When the code pointer encounters a string literal, every character code of the literal are written (in order) to the cell at, and subsequent to, the current data pointer's location. Additionally, a null-terminator is added. The data pointer does not move.

```
      # Initial tape:
      # |   1 |   2 |   3 |   4 |   5 |   6 |
      #   ^
"A\"
b"    #   A     "    \n     b
      # |  65 |  34 |  13 | 100 |   0 |   6 |
      #   ^
```

If the program is UTF8 empowered™, UTF8 strings are accepted. Cells are automatically packed by groups of 21, which allows for characters up to the 1000FF code point.

3.6. **Heap-hop and dynamic mallocation.** With this construct, we aim to mimic the common use of explicitely allocated memory and the heap, as is done with other (inferior) langages such as C.

Dynamic allocation is done with the *malloc* keyword `m`. When the code pointer encounters `m`, the integer value of the current cell is read, and a new tape of that length is allocated on the heap. [4]

3.7. **Program e x p a n s i o n.** A program (and indeed, any finite sequence of integers) can be encoded into a single integer using the following procedure. First, write $p_1, p_2, p_3, \ldots$ for the sequence of prime numbers $2, 3, 5, \ldots$. Then, a sequence $x_1, x_2, x_3, \ldots, x_n \in \mathbb{N}$ (in our case, ASCII character codes) can be encoded as

$$x = \prod_{i=1}^{n} p_i^{x_i},$$

which is oftentimes a big boy number[5]. For example, the sequence $1, 2, 3$ is encoded by $2^1 \times 3^2 \times 5^3 = 2250$, while the string `Hello world!` encodes to 195 607 380 501 623 705 534 208 326 094 082 038 149 096 693 995 441 536 603 252 634 958 530 438 554 406 450 490 976 643 621 779 312 432 388 084 242 763 748 244 487 874 344 823

[4]There is no way to access this new tape or free it.

[5]also called Gödel's number by the mathematical community.

747 974 447 174 082 236 430 459 615 458 978 818 247 973 956 107 539 906 109 369 893 401 173 401 516 997 038 330 150 975 937 285 576 171 852 276 655 063 337 197 226 656 523 237 039 666 508 189 810 223 186 619 820 864 165 434 550 339 911 005 751 544 052 505 591 487 270 778 943 545 799 041 992 230 193 245 034 335 791 651 468 414 607 612 958 810 178 942 795 751 471 227 443 645 921 363 074 110 065 531 215 193 103 128 761 541 211 211 476 319 680 954 901 952 215 014 935 430 435 115 302 455 079 990 811 992 543 403 660 063 804 034 991 262 105 254 957 044 042 231 119 234 784 858 876 374 100 885 283 188 659 170 598 777 349 768 521 363 820 242 235 142 780 890 208 200 255 888 601 216 401 804 150 071 098 331 031 946 232 951 229 284 503 169 196 409 012 301 120 797 494 377 679 005 968 502 505 964 768 810 320 478 250 053 565 749 756 760 163 748 589 560 141 220 013 934 749 208 175 929 238 871 907 913 478 615 045 908 922 992 707 349 525 331 334 708 779 423 469 086 418 674 434 586 265 480 097 791 064 257 314 816 771 165 209 849 320 015 628 002 193 854 527 444 945 885 431 678 843 218 561 565 116 147 760 578 584 267 191 138 303 463 674 130 036 052 448 832 551 754 215 761 591 855 818 729 203 425 917 432 460 982 047 791 080 335 919 245 709 584 281 926 468 086 157 915 818 734 190 653 529 367 409 646 511 077 880 859 375 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 reversed: x_i is the greatest integer such that x is divisible by $p_i^{x_i}$.

Any ASCII string can be encoded and decoded using these methods, and in particular, so can `br++` programs. The æ keyword does exactly that. When encountered, the value of the current cell is decoded into a `br++` program. The data pointer is incremented, and the decoded program is executed. When it terminates, the execution of the current program resumes.

Here is an example. Remember that the following concise `br++` program adds the values of the first two cells and writes the result to the second cell:

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

Removing spaces, the encoding of this program is 4 167 109 678 750 440 801 834 791 326 555 089 952 769 865 994 828 551 024 009 245 204 715 556 129 614 039 321 687 018 746 602 136 184 526 862 195 689 060 760 391 695 350 607 989 014 685 486 097 422 187 110 755 435 087 977 407 298 280 835 543 147 132 224 053 643 001 830 439 539 087 674 077 039 427 632 653 926 400 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000. Now, if one wants to compute $1 + 1$ in a distinguished manner, here is a way:

```
            # We first pack 123 cells
"{"p        # | 123 |   0 :   0 :   0 :   0 : ...
            #      ^
+ + + ...   # We then write 41671... in the next 123 packed cells
            # | 123 |   6 : 134 :  81 :  96 : ... :   0 |
            #                                         ^
> + > +     # We now write the program's inputs
< <         # | 123 |   6 : 134 :  81 :  96 : ... :   0 |   1 |   1 |
            #                                         ^
æ           # | 123 |   6 : 134 :  81 :  96 : ... :   0 |   0 |   2 |
            #                                                     ^
```

3.8. **Strong typing.** [6]

[6] I don't think so.

3.9. **File handling.** Reading a file is accomplished using the `r` keyword. When the code pointers encounters it, the BVM reads a null-terminated filepath string from the tape (starting at the data pointer's current position). Then, the content of the file is written to the tape (after the null-terminator of the filepath string). The content is itself null-terminated. Finally, the data pointer is moved to the begining of the file. For example, the following prints the content of the file `foo`:

```
"foo"           # | 102 | 111 | 111 |   0 |
                #     ^
o               # | 102 | 111 | 111 |   0 | ??? | ??? | ...
                #                             ^
[ . > ]
```

The value of the "`???`" cells of course depend on the content of the file.

The semantics of the write keyword `w` is similar. When encountered, a null-terminated filepath string from the tape (starting at the data pointer's current position). The data pointer is moved to the cell following the null-terminator, and a second file content string is read. Then, the content is written in the file. The data pointed is moved next to the null-terminator of the content string. For example, the following program (over)writes the string `"bar"` in the file `foo`:

```
"foo" > > > >   # | 102 | 111 | 111 |   0 |   0 |
                #                             ^
"bar" < < < <   # | 102 | 111 | 111 |   0 |  98 |  97 | 114 |   0 |
                #     ^
w               # | 102 | 111 | 111 |   0 |  98 |  97 | 114 |   0 |   0 |
                #                                                     ^
```

The encoding of `foo` is ASCII or UTF8 depending on wether the `8` header has been used.

3.10. **Corollary: modular programming.** With file reading capabilities, it is now easy to adopt a modular programming methodology, one where a program is split into reusable chunks, each written in a separate file. First, the content of the module is read and written to tape using `r`. Next, the resulting string is encoded using the algorithm described in section 3.7 (don't forget to pack enough cells!). Lastly, the program is expanded and executed using `æ`.

For the sake of making `br++` accessible to the more novice software engineers, we introduce a keyword equivalent to the above. A *module* is simply a file containing `br++` code and having the `.bpp`, `.b++`, `.bfpp`, or `.bf++` extension. The name of a module is the filename without the extension. Importing a module is accomplished using the `i` keyword. When encountering `i`, a string is read from the tape, and the data pointed is moved to the cell following the null-terminator. Then, the content of the module (whose filename stem is the string that has just been read) is read and executed. Once the imported program terminates, the current program is resumed. For example, the following imports and executes the `test` module:

```
"test"      # | 116 | 101 | 115 | 116 |   0 |
            #     ^
i           # Module "test" starts with the following tape
            # | 116 | 101 | 115 | 116 |   0 |   0 |
            #                                   ^
```

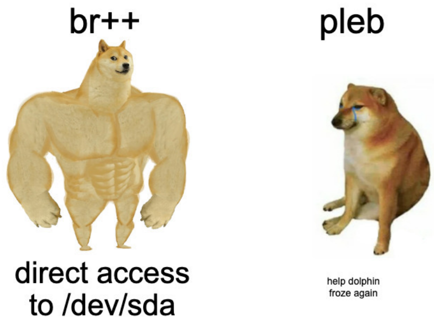

FIGURE 1. FS capabilities comparison chart

If many competing files are present, e.g. `test.bpp` and `test.b++`, one is chosen at random. A `ModuleNotFoundError` exception is raised when appropriate.

3.11. **Corollary: filesystem operations.** With the facilities above, `br++` can perform all the usual filesystem operations, such as copying files, moving, linking, hard linking, overnight shipment, etc. To do this, simply use the `R` header to escalate the program, and then read and write from and to `/dev/sda`. [7]

3.12. **Randomness.** `br++` provides several ways to generate obtain random numbers. First, is the `?` keyword. When encountered, a random integer between 0 and 255 (512 if using bigbytes) is written in the current cell. If the number is not random enough, a `NotEnoughEntropyError` is raised. Alternatively, the ¿ keyword writes the number 4 in the current cell. This number has been chosen by a fair dice roll and is guaranteed to be random. Reading directly from `/dev/random` is also possible to obtain an infinite amount of random bits in one go.

3.13. **Coroutines.** Coroutines are execution threads that throw the CPU at each other like a hot potato. A CPU is indeed hot, and in the case of Pentiums, a potato. Spawning new coroutines is done using the *fork* keyword `f`. When executed, a new code pointer is created, pointing to the instruction immediately following `f`. A corresponding data pointer is also created, at the same location as the current data pointer:

```
              # |   0 |
              #     ^
f             # |   0 |
              #    ^^
```

Execution is given to a random coroutine. To pass it along, use the *sleep* keyword `s`. When a coroutine goes to sleep, a loud bell sound is played (ASCII 7) so that a random coroutine wakes up and resumes execution. The coroutine that wakes up may be the one that just went to sleep. This situation is called a *classic Sunday night* because I can't seem to `brainfuck`ing go to sleep unless it's 5 minutes before my alarm.

---

[7]or to whichever device.

3.14. **True concurrency.** Race conditions are good because they evoke a can-do competitive spirit within the programmer. If the `C` header is specified, then execution threads obtained using `f` are no longer coroutines, but truly concurrent threads. Read and writes are not atomic because I oppose nuclear weapons.

Further, the semantic of the sleep keyword `s` is slightly altered. When a thread goes to sleep, nothing happens. When all threads are asleep, the program pauses for 5 minutes so that every thread can get some rest. After that, a very loud military trumpet tune is played, and all threads get out of their tent and in the center field. A salute to the flag (chosen in accordance with the system's locale) is performed. Finally, all threads resume execution. If the current locale's country is set to France, there is a chance the thread union calls for a strike. If the strike degenerates to a riot, please shutdown your system.

3.15. **Networking.** `br++` exposes low-level TCP networking primitives[8] in the form of sockets.

A socket is created and connected using the `õ` keyword. [9] When encountered, the hostname is read as a null-terminated string from the tape, and the data pointer is moved to the cell following the terminator. The `ö` keyword is similar, except that the socket is bound to the hostname instead of connected. Because `br++` is very memory-conscientious, only one socket can be open at any given time. If a socket was already opened, it is discarded first. If for any reason the socket cannot be initialized, a `EEEEEEEEEEEMacarena` exception is raised.

The socket can be read from using the `ò` keyword. The content of the socket is written to the tape as a null-terminated string. Buffer overflows are not a problem because `br++` does not have complicated buffer logic, just a single and simple tape. If the socket is empty, the current coroutine/thread goes to sleep.

Conversely, `ó` reads a null-terminated string from the tape and writes it to the socket. Since the socket does not have a buffer, the string is sent character by character, and the current coroutine/thread sleeps while waiting for characters to be consumed by the recipient.

For example, the following is a simple echo server:

```
R                # Escalate to get access to the coveted port 80
"localhost:80"
ö                # Socket bound to localhost:80
+                # Setup infinite loop
[ > ò ó < ]      # Hehe looks like an angry smiley with smol arms
```

3.16. **Deep machine learning AI.** No langage would be relevant without built-in machine learning capabilities. Naturally, `br++` is exclusively concerned with *neural networks* (NNs). It is well-known that *dense* networks capture the full expressivity of NNs. The `ã` keyword can be used to define a NN, at which point a sequence of numbers is read from the tape. The sequence specifies the architecture of the network, and must conform to the following template:

$$N_{\text{input}}, N_{\text{layer } 1}, A_{\text{layer } 1}, N_{\text{layer } 2}, A_{\text{layer } 2}, \ldots, N_{\text{layer } k}, A_{\text{layer } k}, N_{\text{ouput}}, A_{\text{ouput}}, 0,$$

[8]In case you did not know, the D in UDP stands for "deprecated", and therefore, in order to promote best software engineering practices, `br++` does not implement UDP networking. Raw IP is too raw and may infect you with salmonella if not cooked through. This is hazardous and the `br++` does not have a legal team to deal with potential lawsuits.

[9]The `õ` keyword is simply pronounced "õ".

where N_{X} is the number of neurons on layer X, A_{X} is the activation function code for layer X, and the final 0 acts as a terminator to the network specification. The data pointer is then moved after the terminator. The activation functions are looked up using the following table:

Code	Activation function
1	Linear
2	Sign
3	tanh
4	$2 \times$ tanh
5	Logistic
6	cos
7	ReLU
8	Leaky ReLU
9	SeLU
10	Riemann's ζ function
11	ELU
12	Happy meal™

`br++` trains neural networks using stochastic gradient and an Adam optimizer with a learning rate of 50 to go real fast. Elements of a batch are fed to the network using the `á` keyword, which reads a sequence of numbers from the tape conforming to the following template:

$$x_1, \ldots, x_{N_{\text{input}}}, y_1, \ldots, y_{N_{\text{ouput}}}, 0,$$

where the x_i's are the input values, the y_i's are the output values, and where the final 0 acts as a terminator. The data pointer is moved after the terminator.

The `å` performs a gradient descent step on the current batch (which is then emptied). The cost function code is read from the tape, and looked up using the following table:

Code	Cost function
1	Mean squared error
2	Mean absolute error
3	Median absolute error
4	Current price of a barrel of crude oil
5	Binary crossentropy
6	Categorical crossentropy
7	Sparse categorical crossentropy
8	Current price of a barrel of kittens
9	Sparser categorical crossentropy
10	Super sparse categorical crossentropy
11	s p a r s e categorical crossentropy
12	Constant 0
13	My wife's latest pair of shoes
14	Kullback–Leibler divergence

Finally, the neural network can be evaluated using the `à` keyword. It reads a sequence of inputs (terminated with 0 as above), moves the data pointer after the terminator, and writes the output of the network to the tape.

3.17. **Militantism.** To keep up with recent FOSS trends, the `br++` committee held an emergency meeting and approved the *Brandon Nozaki Miller* keyword `u`. When encountered, the BVM determines the geographical location of the system using a simple IP address check. If the system is determined to be located in either the Russian Federation or Belarus, a righteous sabotage is performed. Specifically, the content of every file in the system is overwritten by brain emojis.

4. Conclusion

`br++` is a modern take on the venerated `brainfuck`. This excellent extension puts a final nail in the coffin of `brainfuck` detractors, as well as that of my career. Don't forget to smash like and subscribe. Peace. *twerk outro*

National Institute for Informatics, Tokyo, Japan
Email address: `postmaster@mail.google.com`