Brainfuck

Brainfuck is an esoteric programming language created in 1993 by Urban Müller, and is notable for its extreme minimalism.[1]

The language consists of only eight simple commands and an instruction pointer. While it is fully Turing complete, it is not intended for practical use, but to challenge and amuse programmers. Brainfuck simply requires one to break commands into microscopic steps.

The language's name is a reference to the slang term *brainfuck*, which refers to things so complicated or unusual that they exceed the limits of one's understanding.

Brainfuck

Paradigm	esoteric, imperative, structured
Designed by	Urban Müller
First appeared	September 1993
Typing discipline	typeless
Filename extensions	.b, .bf
Influenced by	
P″, FALSE	

Contents

History

In 1992, Urban Müller, a Swiss physics student, took over a small online archive for Amiga software.[2] The archive grew more popular, and was soon mirrored around the world. Today, it is the world's largest Amiga archive, known as Aminet.

Müller designed Brainfuck with the goal of implementing it with the smallest possible compiler,[3] inspired by the 1024-byte compiler for the FALSE programming language.[4] Müller's original compiler was implemented in machine language and compiled to a binary with a size of 296 bytes. He uploaded the first Brainfuck compiler to Aminet in 1993. The program came with a "Readme" file, which briefly described the language, and challenged the reader "Who can program anything useful with it? :)". Müller also included an interpreter and some quite elaborate examples. A second version of the compiler used only 240 bytes.[5]

As Aminet grew, the compiler became popular among the Amiga community, and in time it was implemented for other platforms. Several brainfuck compilers have been made smaller than 200 bytes, and one is only 100 bytes.[6]

P″: Brainfuck's formal "parent language"

Except for its two I/O commands, Brainfuck is a minor variation of the formal programming language P″ created by Corrado Böhm in 1964, which in turn is explicitly based on the Turing machine. In fact, using six symbols equivalent to the respective Brainfuck commands `+`, `-`, `<`, `>`, `[`, `]`, Böhm provided an explicit program for each of the basic functions that together serve to compute any computable function. So the first "Brainfuck" programs appear in Böhm's 1964 paper – and they were programs sufficient to prove Turing completeness.

The Infinite Abacus: Brainfuck's "grand-parent" language

A version with explicit memory addressing rather without stack and a conditional jump was introduced by Joachim Lambek in 1961 under the name of the Infinite Abacus,[7] consisting of an infinite number of cells and two instructions:

- `X+` (increment cell `X`)
- `X- else jump T` (decrement `X` if it is positive else jump to `T`)

He proves the Infinite Abacus can compute any computable recursive function by programming Kleene set of basic μ-recursive function.

His machine was simulated by Melzac's machine[8] modeling computation via arithmetic rather than logic mimicking a human operator moving pebbles on an abacus, hence the requirement that all numbers must be positive. Melzac, whose one instruction set computer is equivalent to an Infinite Abacus, gives programs for multiplication, gcd, n^t prime number, representation in base b, sorting by magnitude, and shows how to simulate an arbitrary Turing machine.

Language design

The language consists of eight commands, listed below. A brainfuck program is a sequence of these commands, possibly interspersed with other characters (which are ignored). The commands are executed sequentially, with some exceptions: an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command. The program terminates when the instruction pointer moves past the last command.

The brainfuck language uses a simple machine model consisting of the program and instruction pointer, as well as an array of at least 30,000 byte cells initialized to zero; a movable data pointer (initialized to point to the leftmost byte of the array); and two streams of bytes for input and output (most often connected to a keyboard and a monitor respectively, and using the ASCII character encoding).

Commands

The eight language commands each consist of a single character:

Character	Meaning
>	increment the data pointer (to point to the next cell to the right).
<	decrement the data pointer (to point to the next cell to the left).
+	increment (increase by one) the byte at the data pointer.
-	decrement (decrease by one) the byte at the data pointer.
.	output the byte at the data pointer.
,	accept one byte of input, storing 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.

(Alternatively, the] command may instead be translated as an unconditional jump **to** the corresponding [command, or vice versa; programs will behave the same but will run more slowly, due to unnecessary double searching.)

[and] match as parentheses usually do: each [matches exactly one] and vice versa, the [comes first, and there can be no unmatched [or] between the two.

Brainfuck programs can be translated into C using the following substitutions, assuming `ptr` is of type `char*` and has been initialized to point to an array of zeroed bytes:

brainfuck command	C equivalent
(Program Start)	`char array[30000] = {0};` `char *ptr=array;`
>	`++ptr;`
<	`--ptr;`
+	`++*ptr;`
-	`--*ptr;`
.	`putchar(*ptr);`
,	`*ptr=getchar();`
[	`while (*ptr) {`
]	`}`

As the name suggests, Brainfuck programs tend to be difficult to comprehend. This is partly because any mildly complex task requires a long sequence of commands, and partly it is because the program's text gives no direct indications of the program's state. These, as well as Brainfuck's inefficiency and its limited input/output capabilities, are some of the reasons it is not used for serious programming. Nonetheless, like any Turing complete language, Brainfuck is theoretically capable of computing any computable function or simulating any other computational model, if given access to an unlimited amount of memory.[9] A variety of Brainfuck programs have been written.[10] Although Brainfuck programs, especially complicated ones, are difficult to write, it is quite trivial to write an interpreter for Brainfuck in a more typical language such as C due to its simplicity. There even exist Brainfuck interpreters written in the Brainfuck language itself.[11][12]

Brainfuck is an example of a so-called Turing tarpit: It can be used to write *any* program, but it is not practical to do so, because Brainfuck provides so little abstraction that the programs get very long and/or complicated.

Examples

Adding two values

As a first, simple example, the following code snippet will add the current cell's value to the next cell: Each time the loop is executed, the current cell is decremented, the data pointer moves to the right, that next cell is incremented, and the data pointer moves left again. This sequence is repeated until the starting cell is 0.

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

This can be incorporated into a simple addition program as follows:

```
++       Cell c0 = 2
> +++++  Cell c1 = 5

[        Start your loops with your cell pointer on the loop counter (c1 in our case)
< +      Add 1 to c0
> -      Subtract 1 from c1
]        End your loops with the cell pointer on the loop counter

At this point our program has added 5 to 2 leaving 7 in c0 and 0 in c1
but we cannot output this value to the terminal since it is not ASCII encoded!

To display the ASCII character "7" we must add 48 to the value 7
48 = 6 * 8 so let's use another loop to help us!

++++ ++++  c1 = 8 and this will be our loop counter again
[
< +++ +++  Add 6 to c0
> -        Subtract 1 from c1
]
< .        Print out c0 which has the value 55 which translates to "7"!
```

Hello World!

The following program prints "Hello World!" and a newline to the screen:

```
[ This program prints "Hello World!" and a newline to the screen, its
  length is 106 active command characters. [It is not the shortest.]

  This loop is an "initial comment loop", a simple way of adding a comment
  to a BF program such that you don't have to worry about any command
```

```
  characters. Any ".", ",", "+", "-", "<" and ">" characters are simply
  ignored, the "[" and "]" characters just have to be balanced. This
  loop and the commands it contains are ignored because the current cell
  defaults to a value of 0; the 0 value causes this loop to be skipped.
]
++++++++               Set Cell #0 to 8
[
    >++++               Add 4 to Cell #1; this will always set Cell #1 to 4
    [                   as the cell will be cleared by the loop
        >++             Add 2 to Cell #2
        >+++            Add 3 to Cell #3
        >+++            Add 3 to Cell #4
        >+              Add 1 to Cell #5
        <<<<-           Decrement the loop counter in Cell #1
    ]                   Loop till Cell #1 is zero; number of iterations is 4
    >+                  Add 1 to Cell #2
    >+                  Add 1 to Cell #3
    >-                  Subtract 1 from Cell #4
    >>+                 Add 1 to Cell #6
    [<]                 Move back to the first zero cell you find; this will
                        be Cell #1 which was cleared by the previous loop
    <-                  Decrement the loop Counter in Cell #0
]                       Loop till Cell #0 is zero; number of iterations is 8

The result of this is:
Cell No :   0   1   2   3   4   5   6
Contents:   0   0  72 104  88  32   8
Pointer :   ^

>>.                     Cell #2 has value 72 which is 'H'
>---.                   Subtract 3 from Cell #3 to get 101 which is 'e'
+++++++..+++.           Likewise for 'llo' from Cell #3
>>.                     Cell #5 is 32 for the space
<-.                     Subtract 1 from Cell #4 for 87 to give a 'W'
<.                      Cell #3 was set to 'o' from the end of 'Hello'
+++.------.--------.    Cell #3 for 'rl' and 'd'
>>+.                    Add 1 to Cell #5 gives us an exclamation point
>++.                    And finally a newline from Cell #6
```

For "readability", this code has been spread across many lines, and blanks and comments have been added. Brainfuck ignores all characters except the eight commands `+-<>[],.` so no special syntax for comments is needed (as long as the comments do not contain the command characters). The code could just as well have been written as:

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

ROT13

This program enciphers its input with the ROT13 cipher. To do this, it must map characters A-M (ASCII 65-77) to N-Z (78-90), and vice versa. Also it must map a-m (97-109) to n-z (110-122) and vice versa. It must map all other characters to themselves; it reads characters one at a time and outputs their enciphered equivalents until it reads an EOF (here assumed to be represented as either -1 or "no change"), at which point the program terminates.

The basic approach used is as follows. Calling the input character *x*, divide *x*-1 by 32, keeping quotient and remainder. Unless the quotient is 2 or 3, just output *x*, having kept a copy of it during the division. If the quotient is 2 or 3, divide the remainder ((*x*-1) modulo 32) by 13; if the quotient here is 0, output *x*+13; if 1, output *x*-13; if 2, output *x*.

Regarding the division algorithm, when dividing *y* by *z* to get a quotient *q* and remainder *r*, there is an outer loop which sets *q* and *r* first to the quotient and remainder of 1/*z*, then to those of 2/*z*, and so on; after it has executed *y* times, this outer loop terminates, leaving *q* and *r* set to the quotient and remainder of *y*/*z*. (The dividend *y* is used as a diminishing counter that controls how many times this loop is executed.) Within the loop, there is code to increment *r* and decrement *y*, which is usually sufficient; however, every *z*th time

through the outer loop, it is necessary to zero *r* and increment *q*. This is done with a diminishing counter set to the divisor *z*; each time through the outer loop, this counter is decremented, and when it reaches zero, it is refilled by moving the value from *r* back into it.

```
-,+[                         Read first character and start outer character reading loop
    -[                       Skip forward if character is 0
        >>++++[>++++++++<-]  Set up divisor (32) for division loop
                               (MEMORY LAYOUT: dividend copy remainder divisor quotient zero zero)
        <+<-[                Set up dividend (x minus 1) and enter division loop
            >+>+>-[>>>]      Increase copy and remainder / reduce divisor / Normal case: skip forward
            <[[>+<-]>>+>]    Special case: move remainder back to divisor and increase quotient
            <<<<<-           Decrement dividend
        ]                    End division loop
    ]>>>[-]+                 End skip loop; zero former divisor and reuse space for a flag
    >--[-[<->+++[-]]]<[         Zero that flag unless quotient was 2 or 3; zero quotient; check flag
        ++++++++++++<[       If flag then set up divisor (13) for second division loop
                               (MEMORY LAYOUT: zero copy dividend divisor remainder quotient zero
zero)
            >-[>+>>]         Reduce divisor; Normal case: increase remainder
            >[+[<+>-]>+>>]   Special case: increase remainder / move it back to divisor / increase
quotient
            <<<<<-           Decrease dividend
        ]                    End division loop
        >>[<+>-]             Add remainder back to divisor to get a useful 13
        >[                   Skip forward if quotient was 0
            -[               Decrement quotient and skip forward if quotient was 1
                -<<[-]>>     Zero quotient and divisor if quotient was 2
            ]<<[<<->>-]>>    Zero divisor and subtract 13 from copy if quotient was 1
        ]<<[<<+>>-]          Zero divisor and add 13 to copy if quotient was 0
    ]                        End outer skip loop (jump to here if ((character minus 1)/32) was not 2
or 3)
    <[-]                     Clear remainder from first division if second division was skipped
    <.[-]                    Output ROT13ed character from copy and clear it
    <-,+                     Read next character
]                            End character reading loop
```

Portability issues

Partly because Urban Müller did not write a thorough language specification, the many subsequent brainfuck interpreters and compilers have come to use slightly different dialects of brainfuck.

Cell size

In the classic distribution, the cells are of 8-bit size (cells are bytes), and this is still the most common size. However, to read non-textual data, a brainfuck program may need to distinguish an end-of-file condition from any possible byte value; thus 16-bit cells have also been used. Some implementations have used 32-bit cells, 64-bit cells, or bignum cells with practically unlimited range, but programs that use this extra range are likely to be slow, since storing the value n into a cell requires $O(n)$ time as a cell's value may only be changed by incrementing and decrementing.

In all these variants, the `,` and `.` commands still read and write data in bytes. In most of them, the cells wrap around, i.e. incrementing a cell which holds its maximal value (with the `+` command) will bring it to its minimal value and vice versa. The exceptions are implementations which are distant from the underlying hardware, implementations that use bignums, and implementations that try to enforce portability.

It is usually easy to write brainfuck programs that do not ever cause integer wraparound or overflow, and therefore don't depend on cell size. Generally this means avoiding increment of +255 (unsigned 8-bit wraparound), or avoiding overstepping the boundaries of [-128, +127] (signed 8-bit wraparound) (since there are no comparison operators, a program cannot distinguish between a signed and unsigned two's complement fixed-bit-size cell and negativeness of numbers is a matter of interpretation). For more details on integer wraparound, see the Integer overflow article.

Array size

In the classic distribution, the array has 30,000 cells, and the pointer begins at the leftmost cell. Even more cells are needed to store things like the millionth Fibonacci number, and the easiest way to make the language Turing complete is to make the array unlimited on the right.

A few implementations[13] extend the array to the left as well; this is an uncommon feature, and therefore portable brainfuck programs do not depend on it.

When the pointer moves outside the bounds of the array, some implementations will give an error message, some will try to extend the array dynamically, some will not notice and will produce undefined behavior, and a few will move the pointer to the opposite end of the array. Some tradeoffs are involved: expanding the array dynamically to the right is the most user-friendly approach and is good for memory-hungry programs, but it carries a speed penalty. If a fixed-size array is used it is helpful to make it very large, or better yet let the user set the size. Giving an error message for bounds violations is very useful for debugging but even that carries a speed penalty unless it can be handled by the operating system's memory protections.

End-of-line code

Different operating systems (and sometimes different programming environments) use subtly different versions of ASCII. The most important difference is in the code used for the end of a line of text. MS-DOS and Microsoft Windows use a CRLF, i.e. a 13 followed by a 10, in most contexts. UNIX and its descendants (including GNU/Linux and Mac OS X) and Amigas use just 10, and older Macs use just 13. It would be difficult if brainfuck programs had to be rewritten for different operating systems. However, a unified standard was easy to create. Urban Müller's compiler and his example programs use 10, on both input and output; so do a large majority of existing brainfuck programs; and 10 is also more convenient to use than CRLF. Thus, brainfuck implementations should make sure that brainfuck programs that assume newline = 10 will run properly; many do so, but some do not.

This assumption is also consistent with most of the world's sample code for C and other languages, in that they use '\n', or 10, for their newlines. On systems that use CRLF line endings, the C standard library transparently remaps "\n" to "\r\n" on output and "\r\n" to "\n" on input for streams not opened in binary mode.

End-of-file behavior

The behavior of the `,` command when an end-of-file condition has been encountered varies. Some implementations set the cell at the pointer to 0, some set it to the C constant EOF (in practice this is usually -1), some leave the cell's value unchanged. There is no real consensus; arguments for the three behaviors are as follows.

Setting the cell to 0 avoids the use of negative numbers, and makes it marginally more concise to write a loop that reads characters until EOF occurs. This is a language extension devised by Panu Kalliokoski.

Setting the cell to -1 allows EOF to be distinguished from any byte value (if the cells are larger than bytes), which is necessary for reading non-textual data; also, it is the behavior of the C translation of `,` given in Müller's readme file. However, it is not obvious that those C translations are to be taken as normative.

Leaving the cell's value unchanged is the behavior of Urban Müller's brainfuck compiler. This behavior can easily coexist with either of the others; for instance, a program that assumes EOF = 0 can set the cell to 0 before each `,` command, and will then work correctly on implementations that do either EOF = 0 or EOF =

"no change". It is so easy to accommodate the "no change" behavior that any brainfuck programmer interested in portability should do so.

Frameworks

Tyler Holewinski developed a C# .NET Framework, BrainF.NET (https://www.github.com/erwijet/brainf.net), which by default runs brainfuck, but can also be used to derive various forms of the language, as well as add new commands, or modify the behavior of existing ones. BrainF.NET thereby allows for development of programs such as an IDE.

Derivatives

Many people have created brainfuck equivalents (languages with commands that directly map to brainfuck) or brainfuck derivatives (languages that extend its behavior or map it into new semantic territory).

Some examples:

- **Pi**, which maps brainfuck into errors in individual digits of Pi.[14]
- **VerboseFuck**, which looks like a traditional programming language, only what appears as parameters or expressions are actually parts of longer commands that cannot be altered.[15]
- **DerpPlusPlus**, in which the commands are replaced with words such as 'HERP', 'DERP', 'GIGITY', etc.[16]
- **Ook!**, which maps brainfuck's eight commands to two-word combinations of "Ook.", "Ook?", and "Ook!", jokingly designed to be "writable and readable by orang-utans" according to its creator, a reference to the orang-utan Librarian in the novels of Terry Pratchett.[17][18]
- **Ternary**, similar in concept to Ook! but instead consisting of permutations of the ASCII characters 0, 1, and 2.[19]
- **BodyFuck**, a BrainFuck implementation based on a gesture-controlled system so that programmer's movements are captured by a video camera and converted into the 8 possible characters.[20]
- **OooWee**, commands are variations of OooWee (e.g. ooo,ooe,wee etc.). Inspired from the Rick and Morty character Mr. PoopyButthole.[21]

See also

- JSFuck – an esoteric JavaScript programming language with a very limited set of characters

References

1. Easter, Brandee (2020-04-02). "Fully Human, Fully Machine: Rhetorics of Digital Disembodiment in Programming" (https://dx.doi.org/10.1080/07350198.2020.1727096). *Rhetoric Review*. **39** (2): 202–215. doi:10.1080/07350198.2020.1727096 (https://doi.org/10.1080%2F07350198.2020.1727096). ISSN 0735-0198 (https://www.worldcat.org/issn/0735-0198).
2. "Aminet hits 5000 files" (http://de4.aminet.net/docs/misc/5000.txt). Urban Müller. 1993-09-24. Retrieved 2015-05-03.
3. "The Brainfuck Programming Language" (http://www.muppetlabs.com/~breadbox/bf/). Muppetlabs.com. Retrieved 2013-10-30.
4. "Wouter's Wiki : False Language" (http://strlen.com/false/index.html). Strlen.com. 2013-08-03. Retrieved 2013-10-30.

5. "dev/lang/brainfuck-2.lha" (https://web.archive.org/web/20051106213924/http://www.aminet.net/package.php?package=dev%2Flang%2Fbrainfuck-2.lha). Aminet. Archived from the original (http://www.aminet.net/package.php?package=dev/lang/brainfuck-2.lha) on 2005-11-06. Retrieved 2013-10-30.
6. "BRAINFCK IN 100 BYTES!" (https://github.com/peterferrie/brainfuck). github.com. Retrieved 2016-03-22.
7. J. Lambek (1961). "How to program an infinite abacus" (https://web.archive.org/web/20180915122032/https://cms.math.ca/10.4153/CMB-1961-032-6). *Canadian Mathematical Bulletin*. Archived from the original (https://cms.math.ca/10.4153/CMB-1961-032-6) on 2018-09-15. Retrieved 2018-09-15.
8. Z. A. Melzak (1961). "An informal arithmetical approach to computability and computation" (https://web.archive.org/web/20180915122012/https://cms.math.ca/10.4153/CMB-1961-031-9). *Canadian Mathematical Bulletin*. Archived from the original (https://cms.math.ca/10.4153/CMB-1961-031-9) on 2018-09-15. Retrieved 2018-09-15.
9. "BF is Turing-complete" (http://www.iwriteiam.nl/Ha_bf_Turing.html). Iwriteiam.nl. Retrieved 2013-10-30.
10. "Index of /brainfuck/bf-source/prog" (http://esoteric.sange.fi/brainfuck/bf-source/prog/). Esoteric.sange.fi. 2002-01-22. Retrieved 2013-10-30.
11. "BF interpreter written in BF" (http://www.iwriteiam.nl/Ha_bf_inter.html). Iwriteiam.nl. Retrieved 2013-10-30.
12. "brainfuck interpreter" (http://www.hevanet.com/cristofd/dbfi.b). Daniel B. Cristofani.
13. Bolognani, Andrea. "Beef –" (http://kiyuko.org/software/beef). Kiyuko.org. Retrieved 2013-10-30.
14. "Pi - Esolang" (https://esolangs.org/wiki/Pi). *esolangs.org*. Retrieved 2019-03-19.
15. "VerboseFuck - Esolang" (https://esolangs.org/wiki/VerboseFuck). *esolangs.org*. Retrieved 2019-09-11.
16. "TheRaz/DerpPlusPlus" (https://github.com/TheRaz/DerpPlusPlus). Github.com. Retrieved 2015-04-08.
17. Morgan-Mar, David (2009-03-21). "Ook!" (http://www.dangermouse.net/esoteric/ook.html). *DM's Esoteric Programming Languages*. Retrieved 2014-06-28.
18. Paloque-Bergès, Camille (2009). *Poétique des codes sur le réseau informatique* (https://books.google.com/books?id=HQt00bhlSqsC&pg=PA73) (in French). Paris: Éditions des archives contemporaines. p. 73. ISBN 978-2-914610-70-4.
19. "Ternary Programming Language" (https://github.com/zerosum0x0/ternary). *Ternary Programming Language*. Retrieved 2015-06-14.
20. Hanselmann, Nik. "There is no hardware" (http://nik.works/bodyfuck/). Retrieved 2 February 2016.
21. "omkarjc27/OooWee" (https://github.com/omkarjc27/OooWee). Github.com. Retrieved 2019-01-19.

External links

- Brainfuck (https://curlie.org/Computers/Programming/Languages/Brainfuck) at Curlie
- Brainfuck (http://www.bf.doleczek.pl/) interpreter on-line in JavaScript with collection of programs
- Brainfuck IDE (https://github.com/zk-phi/bfbuilder) - A brainfuck development environment with interactive debugger
- Brainfuck generator (http://tunn.us/bf/generator.php) - A text to brainfuck generator
- Brainfuck collection (https://github.com/pablojorge/brainfuck) of Interpreters and scripts

- Brainfuck (https://www.vanheusden.com/misc/blog/2016-05-19_brainfuck_cobol_compiler.php) to COBOL, C, ASM, PL/1, ... compiler
- Brainfuck Assembler (https://gitlab.com/hilmar-ackermann/brainfuckassembler) translating x86 assembly code (reduced set) into brainfuck code

---