10

BRAINFUCK

Brainfuck, or BF as we'll call it, is more or less the grandfather of all esolangs. It's one of the earliest and probably the most extended, modified, discussed, and parodied esolang in existence. In this chapter, we'll see what all the fuss is about it–there's more to it than just the name!

WTF is BF

BF is the brainchild of Urban Müller, who loosed it upon an unsuspecting world in 1993. His goal was to create a tiny language leading to a tiny compiler for the Amiga computer. His compiler was 296 bytes long. Later in the chapter, we'll encounter a BF compiler that's only 166 bytes long.

How can BF compilers be so tiny? Because BF itself is tiny (see Table 10-1, which describes all *eight* commands). As a certain pig might say, "Th-th-that's all, folks!"

Table 10-1: BF in all its glory

Command	Action
>	increment memory pointer
<	decrement memory pointer
.	print memory as a character
,	input a character to memory
+	increment memory
-	decrement memory
[	begin loop if memory not zero
]	continue loop if memory not zero

The machine BF expects is quite similar to a Turing machine. The BF machine is a vector of cells, each of which holds a single value. Originally, there were 30,000 cells, each capable of holding a single byte $[0, 255]$. The interpreters we'll use in this chapter have 32-bit cells. BF is analogous to what's called a *Harvard architecture*, a hardware architecture in which the program space and memory are distinct. No self-modifying code here, though I wouldn't be surprised if someone hasn't created a Von Neumann version of BF just to explore what self-modifying BF code might be able to accomplish. Recall, a Von Neumann architecture combines program space and memory. Modern PCs are Von Neumann machines.

A Turing machine has a tape head that moves along a tape to read and write symbols. BF does the same, but in this case we'll call the "tape head" a memory or cell pointer. The > and < instructions move the memory pointer from cell to cell and the remaining BF instructions operate on the current memory cell. When a BF program starts, it assumes each memory cell has a value of zero and the pointer is looking at cell zero.

We now know what two of BF's eight instructions do. The , (comma) and . (period) are input and output, respectively. The + increments the cell and - decrements it.

What makes BF interesting as a language are the [and] loop instructions. Loops begin with [and end with], but both are commands. When BF executes the [instruction, it looks at the current cell and asks: "is the value zero?" If the answer is "yes," BF skips ahead to the] instruction and continues with the instruction after it. If the answer is "no," BF moves to the next instruction to begin the loop.

When BF encounters a] instruction, it doesn't automatically jump to the corresponding [. Instead, it examines the *current* memory cell and jumps back if that cell *isn't* zero. Otherwise, the loop ends and BF continues with the next instruction.

Think about this for a bit. The [command is a gatekeeper to decide whether a loop begins, but it doesn't check anything after that. The decision about continuing a loop falls to the matching] instruction. Also, the cell that initiated the loop need not be the cell that decides whether the loop continues. BF is very flexible when it comes to looping, as we might expect from such a provocatively named language—it's messing with our brains. BF loops are neither top-tested nor bottom-tested, but rather are a mix of both.

There is a top test to decide whether the loop even begins, but from then on, the loop is bottom-tested. In C, we can accomplish something similar by nesting loops,

```
int cond1=1, cond2=1;
while (cond1) {
    cond1 = 0;
    do {

    } while(cond2);
}
```

To begin the loop, `cond1` must be true; however, after `cond1` begins the loop, it isn't examined again, the loop runs until `cond2` is no longer true. To get the desired effect in C, `cond1` must be made false (`cond1 = 0`) so the entire structure exits when `cond2` becomes false.

BF code isn't particularly easy to write, but it is complete enough to implement interesting, nontrivial programs. BF is more than ABC. We need an interpreter to see that's the case; we'll leave compiler design for another day.

The Two Implementations

BF implementations abound. Let's investigate two in this section. The first is a slightly updated version of Urban Müller's original 1993 C code for the Amiga computer. The second we'll build from scratch in SNOBOL because an unusual, minimalist esoteric language deserves an equally unusual implementation. If you skipped Chapter 5 on SNOBOL, now's a good time to go back and read it.

The Original

The original Amiga LHA archive with the first version of BF is in the file *brainf-2.lha*. Müller's implementation is in plain C. To work with the code on a modern Linux system, I took the liberty of updating it to compile without warnings, changed the cell size from 8-bits (`unsigned char`) to 32-bits (`int`), and increased the program space to 70,000 cells. Using 32-bit cells matches the SNOBOL implementation we'll develop in the next section.

Listing 10-1 shows the interpreter in its entirety.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXPROG 70000
#define MAXMEM 30000
int  p, r, q;
int a[MAXMEM];
char f[MAXPROG], b, o, *s=f;

void interpret(char *c) {
    char *d;
```

```
    r++;
    while( *c ) {
        switch(o=1,*c++) {
        case '<': p--;        break;
        case '>': p++;        break;
        case '+': a[p]++;     break;
        case '-': a[p]--;     break;
        case '.': putchar(a[p]); fflush(stdout); break;
        case ',': a[p]=getchar();fflush(stdout); break;
        case '[':
            for( b=1,d=c; b && *c; c++ )
                b+=*c=='[', b-=*c==']';
            if(!b) {
                c[-1]=0;
                while( a[p] )
❶                   interpret(d);
                c[-1]=']';
                break;
            }
        case ']':
            puts("UNBALANCED BRACKETS"), exit(0);
        case '#':
            if(q>2)
                printf("%2d %2d %2d %2d %2d %2d %2d %2d %2d %2d\n%*s\n",
                       *a,a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9],3*p+2,"^");
            break;
        default: o=0;
        }
        if( p<0 || p>(MAXMEM-1))
            puts("RANGE ERROR"), exit(0);
    }
    r--;
}

int main(int argc,char *argv[]) {
    FILE *z;
    q=argc;
    if(z=fopen(argv[1],"r")) {
        while( (b=getc(z))>0 )
            *s++=b;
        *s=0;
        interpret(f);
    }
}
```

Listing 10-1: Urban Müller's original BF interpreter (updated)

This implementation is quite compact and handles loops via recursion (notice the recursive call to `interpret` ❶). Our SNOBOL implementation will process loops without recursion. Also, notice that `#` is a supported command. It prints basic debugging information if the interpreter is called with a second command line argument. The `#` command was dropped from later versions of BF. My modifications introduce `MAXMEM` and `MAXPROG` and the addition of `int` before `main` to avoid a `gcc` warning.

The interpreter processes the input BF program loaded into `f`. The `interpret` function loops over the characters in `f`, or the characters of the nested loop enclosed in brackets via the recursive call. If the character is a BF command, the command is performed; otherwise, it is ignored.

Building the interpreter is straightforward,

```
> gcc bfi.c -o bfi
```

As is testing it,

```
> bfi examples/hello.b
Hello World!
```

All the BF examples in the book's GitHub repository work with this interpreter. However, not every BF example you'll find on the web does. Take a look at the *README* file in the *examples* directory as it contains attribution and license information. Credit is given to code authors where authorship is known. I'll leave working through the operation of *hello.b* as an exercise as there are explanatory comments in the file. As you might expect, it involves generating and printing the required sequence of ASCII values.

SNOBOL Meets BF

The seductive elegance of BF requires, indeed, almost demands we make our own interpreter. We'll use SNOBOL because SNOBOL provides all the facilities we need. Besides, it's fun. The full interpreter is in *bf.sno*. Let's begin with the parser:

```
            MAXPROG = 70000
            MAXMEM = 30000
            prog = array('0:' MAXPROG)
❶           mem = array('0:' MAXMEM, 0)
            jump = table()

            define('parse(name)c,n,pat')      :(eparse)
parse       pat = break('><+-.,[]')
            input('reader', 10, 'B,1', name)
parse_l0    c = reader                        :f(parse_l1)
            c pat                             :f(parse_l0)
            prog[n] = c
            n = ne(n,MAXPROG) n + 1           :s(parse_l0)
parse_l1    endfile(10)
```

```
            parse = n                           :(return)
eparse
```

Listing 10-2: Parsing the input file

Listing 10-2 presents global memory definitions and the `parse` function to read the input file and keep only actual program commands. Code is stored in the array `prog` with memory in `mem`, a second array. BF expects memory to be initialized to zero, which SNOBOL does for us via the second argument to `array` ❶. We'll discuss the `jump` table momentarily.

The `parse` function accepts the `name` of the input text file, defines a pattern to match valid program characters (`pat`), and opens the file for input, reading one character at a time.

The loop (`parse_l0`) reads a character into `c` and applies the pattern. If the pattern succeeds, then `c` contains one of the allowed command characters; therefore, `prog` is set and its index is incremented. Notice the SNOBOL idiom of embedding the `ne` predicate to test for maximum program length. If the predicate fails, the increment to `n` does not happen and execution falls through to `endfile`.

When `parse` finishes processing the input file, `prog` contains the valid commands and only the valid commands. The number of commands read is returned by assigning `n` to `parse`.

A BF program is executed sequentially until the interpreter encounters a loop. The original BF interpreter used recursion to handle loops; however, we'll take a more literal approach. Every time we see an opening bracket (`[`), we'll scan the program text forward to find the corresponding closing bracket (`]`). Similarly, for a closing bracket, we'll scan backward to find the matching opening bracket. We could do this while interpreting the code, but that's hideously inefficient—imagine a loop running tens of thousands of times.

A moment's thought makes it clear that a single pass through the code before starting the interpreter is sufficient to locate each opening bracket and its corresponding closing bracket. This is where the `jump` table comes into play. Recall, a SNOBOL table is like a Python dictionary; it's an associative array. The index into the table is the index of an opening bracket in `prog`. Closing brackets also go in `jump` as their index values are unique. With `jump` built ahead of time, a single reference to `jump` during program execution returns the proper index into `prog` for both the opening and closing brackets.

Consider Listing 10-3, which shows `buildtable` and its helper function, `closing`.

```
            define('closing(pc,plen)n,p')     :(eclosing)
closing     n = 1
            p = pc
closing_l0  p = p + 1
            eq(p,plen)                        :s(bad0)
            ident(prog[p],']')                :s(closing_l1)
            ident(prog[p],'[')                :s(closing_l2)
```

```
closing_l3  eq(n,0)                          :f(closing_l0)
            closing = p                      :(return)
closing_l1  n = n - 1                        :(closing_l3)
closing_l2  n = n + 1                        :(closing_l3)
eclosing

            define('buildtable(plen)n,m')    :(ebuildtable)
buildtable  n = 0
build_l0    ident(prog[n],'[')               :s(build_l1)
build_l2    n = n + 1
            eq(n,plen)                       :f(build_l0)s(return)
build_l1    m = closing(n,plen)
            jump[m] = n
            jump[n] = m                      :(build_l2)
ebuildtable
```

Listing 10-3: Building the jump table

Here, `buildtable` scans the program text looking for an opening bracket. When it finds one, it calls `closing` to return the index of the corresponding closing bracket. Next, `buildtable` sets the `jump` table to the opening and closing locations for rapid lookup during program execution.

The `closing` function locates the matching closing bracket by scanning forward and incrementing `n` each time a new opening bracket is found. When a closing bracket is found, `n` is decremented. When `n` is zero, the closing bracket matching the initial opening bracket has been found, so its index is returned.

BF accepts single-character input that it stores in memory as an ASCII value. SNOBOL has a `char` function to return the character associated with a given ASCII value; however, it lacks what many languages call `ord`, a function to return the ASCII value of a given character. No matter, we'll make our own,

```
      define('ord(c)v')          :(eord)
ord   &alphabet break(c) . v
      ord = size(v)              :(return)
eord
```

SNOBOL includes a special variable, `&alphabet`, which is the full range of ASCII characters, [0, 255]. The `ord` function uses pattern matching to locate all the characters of this special variable up to the given character, `c`. The pattern stores this substring in `v` and the length of the substring is the ASCII code for the character.

We're now ready to run the BF program in `prog`. Let's walk through the main portion of the interpreter. We'll add some debugging abilities to help us later. BF is hard, so we'll take all the help we can get.

The main portion of the interpreter is in Listing 10-4.

```
      plen = parse(host(2,2))
```

```
        buildtable(plen)
        input('cin', 10, 'B,1', '-')
        output('cout', 11, 'WB,1', '-')
        pc = 0
        mp = 0
        gmp = 0
loop    ident(prog[pc],'-')             :s(dec)
        ident(prog[pc],'+')             :s(inc)
        ident(prog[pc],'<')             :s(decp)
        ident(prog[pc],'>')             :s(incp)
        ident(prog[pc],',')             :s(gchar)
        ident(prog[pc],'.')             :s(pchar)
        ident(prog[pc],'[')             :s(begin)
        ident(prog[pc],']')             :s(again)
cont    pc = pc + 1
        ne(pc,plen)                     :f(pend)s(loop)
dec     mem[mp] = mem[mp] - 1           :s(cont)f(bad1)
inc     mem[mp] = mem[mp] + 1           :s(cont)f(bad1)
decp    mp = mp - 1                     :(cont)
incp    mp = mp + 1
        gmp = gt(mp,gmp) mp             :(cont)
gchar   ch = ord(cin)                   :f(pend)
❶       eq(ch,13)                       :f(gchar0)
        ch = 10
        cout = char(ch)
gchar0  mem[mp] = ch                    :s(cont)f(bad1)
pchar   cout = char(mem[mp])            :s(cont)f(bad1)
❷ begin pc = eq(mem[mp],0) jump[pc]     :(cont)
again   pc = ne(mem[mp],0) jump[pc]     :(cont)
```

Listing 10-4: The main BF interpreter loop

Listing 10-4 consists of some preliminaries followed by a `loop` that moves through the program in `prog`. The preliminaries call `parse` to process the input file and `buildtable` to configure the `jump` table. BF expects single-character input and output with the console, which SNOBOL supports using the given `input` and `output` incantations.

The current program counter is `pc` and the memory pointer is `mp`. We'll use `gmp` to track the highest memory cell accessed by the program. Doing this simplifies dumping relevant memory when the program ends.

The `loop` executes the current instruction depending on its character. Recall that `ident` is the SNOBOL predicate to compare two strings. Executing an instruction is a jump to the relevant line. Most instructions are a single statement. For example, `>` moves the cell pointer to the right (`mp=mp+1`). When incrementing the cell pointer, there's an extra check to see if `gmp` should be updated.

SNOBOL has one quirk requiring a bit of extra code. Take a look at `gchar`, which reads a single character of input. The `cin` variable reads the

character and places its ASCII value in ch. The problem is when the user presses enter. On Unix systems, this should return the ASCII value 10; however, the SNOBOL interpreter returns ASCII value 13. So, a quick check converts ASCII 13 to ASCII 10 before assigning the character to the current memory location (gchar0) ❶. Notice that each instruction ends with a jump to cont to continue processing the next instruction.

Additionally, observe how opening and closing brackets are handled as begin and again respectively ❷. Even though there is a single statement for each, a bit of explanation is in order. For example, the code for an opening bracket is,

```
pc = eq(mem[mp],0) jump[pc]      :(cont)
```

The BF standard says to begin a loop if the currently active memory cell is not zero; otherwise, skip the loop. Here, the SNOBOL predicate eq will succeed if the current memory cell is zero. In that case, the assignment happens and pc is set to jump[pc], which is the *end* of the loop that we're currently considering. In contrast, if the memory cell is not zero, eq fails and the assignment does not happen. Therefore, the interpreter enters the loop as it should. The test in again is much the same; only the logic is reversed so we jump to the beginning of the loop if the memory cell is not zero.

Take another look at the statement to decrement the current cell,

```
dec  mem[mp] = mem[mp] - 1   :s(cont)f(bad1)
```

If the decrement succeeds, flow continues with the next instruction, s(cont). However, if mp is negative or too large, the statement fails and the interpreter jumps to bad1,

```
bad1  output = 'memory access error, mp = ' mp  :(end)
```

This prints an error message and exits. A similar error happens if an opening bracket has no matching closing bracket.

If the second command line argument is dump, the interpreter will dump the final value of all memory locations accessed by the program before exiting. The code for this is in Listing 10-5.

```
pend   ident(host(2,3),'dump')            :f(end)
       output =
       output = 'Memory: (mp = ' mp ')'
       n = 0
ploop  ascii = ''
       gt(mem[n],31)                      :f(print)
       lt(mem[n],127)                     :f(print)
       ascii = '  ' char(mem[n])
print  s = dupl(' ', 6 - size(n)) n
       s = s ':' dupl(' ', 6 - size(mem[n])) mem[n]
       output = s ascii
       n = n + 1
       gt(n,gmp)                          :f(ploop)
       output =                           :(end)
```

Listing 10-5: Dumping memory

Memory values are dumped, one per line. If the value is in the range $31 < v < 127$, the corresponding character is displayed.

The BF interpreter is now complete. Let's test it.

```
> snobol4 bf.sno examples/hello.b dump
Hello World!

Memory: (mp = 6)
     0:     0
     1:     0
     2:    72  H
     3:   100  d
     4:    87  W
     5:    33  !
     6:    10
```

The memory dump shows that cells zero through six were used at some point in the program, and that the program ended with the memory pointer looking at cell 6. Knowing which memory cell is active is critical to successful BF programming.

Our implementation appears to work. Now, let's do stuff with it.

BF in Action

Let's explore BF with worked examples. I encourage you to consider the other examples included on the Github site. However, the more complex ones like *mandelbrot.b* and *hanoi.b* are the output of programs that generate BF code. They were not written by hand.

We'll start with some basic examples and then develop more advanced examples that require a bit of thought. For example, we'll end with a program to multiply two numbers.

Baby Steps

Consider the following code.

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

It increments cell 0 five times then starts a loop: `[-]`. Incrementing five times is obvious, so let's work through the loop to see what it does. The first command is `[`. It checks to see whether the current cell is zero. In this case, the cell is five and not zero, so `[` succeeds and the loop begins.

The next instruction, `-`, decrements the value in the current memory cell (cell 0), so the value is now four. The closing bracket, `]`, asks if cell 0 is zero, which it isn't, so it jumps to the beginning of the loop. Note, the beginning of the loop isn't `[`, but the first instruction after it (`-`). Cell 0 is decre-

mented again and] runs again. When the value of cell 0 is zero,] will fail and the program will end. Therefore, the snippet of code above zeros a cell. You'll see [-] in many BF programs.

Now that we have a basic loop under our belt, let's contemplate the following bit of code,

```
,+[-.,+]
```

What do you think it might be doing? The code itself is in *cat.b*. Let's run it and see what it produces. To run it, use this command line,

```
> snobol4 bf.sno examples/cat.b <bf.sno
```

Do you see the text of *bf.sno*? The filename is a clue, of course, but this simple program acts like the Unix `cat` command to display the contents of a file. Let's add comments to the code to explain what is happening,

```
,    read a character; mem(0) = ch
+    inc mem(0)
[    loop if mem(0) is not zero
  -  dec mem(0)
  .  print mem(0) as a character
  ,  read another character to mem(0)
  +  inc mem(0)
]    loop if mem(0) is not zero
```

Reading a character, printing, and looping until there are no more characters to read is a good idea in this case, but what's with + and -? These extra commands handle the case where a zero character has been read. They are present to deal with how different systems process end-of-file (EOF). For example, this version of the program works nicely with our SNOBOL interpreter,

```
,[.,]
```

but hangs at EOF when using the C interpreter.

Let's look at another loop example. Honestly, all our examples will be loop examples as that's all BF has to offer that isn't quickly boring. This example is in *countdown0.b*,

```
++++++++++[-.]
```

It's only slightly more interesting than our first example. Beyond counting down, we also print the value of cell 0. However, BF's print (.) expects an ASCII character, so this example won't print anything visible, only a set of control characters. We can see this by using the Unix `xxd` command to dump binary files,

```
> snobol4 bf.sno examples/countdown0.b | xxd
00000000: 0908 0706 0504 0302 0100  ..........
```

The `xxd` command dumps binary data as hexadecimal values. Looking at the output you'll see the countdown (09, 08, 07, ..., 00). To get a countdown we must convert the current value of cell 0 to a digit. The offset between a digit value and the ASCII code for the digit is 48, so we must add 48 before printing and subtract 48 afterward.

Listing 10-6 shows us *countdown1.b*. We've included comments to explain the code.

```
++++++++++    mem(0) = 10
>             look at mem(1)
++++++++++    mem(1) = 10
[             enter loop if mem(1) not zero
 -            decrement mem(1)
 ++++++++++
 ++++++++++
 ++++++++++
 ++++++++++
 ++++++++     add 48
 .            print
 ----------
 ----------
 ----------
 ----------
 --------     sub 48
 <            look at mem(0)
 .            print it
 >            look at mem(1)
]             loop if mem(1) is not zero
```

Listing 10-6: Countdown with ASCII output

Running Listing 10-6 produces a countdown as output (9, 8, 7, ..., 0). To output newline repeatedly, it's easiest to store it somewhere, so we set cell 0 to 10. Next, `>` moves the cell pointer to look at cell 1. As you write BF code, pay very close attention to where the cell pointer is looking. Cell 1 is set to 10 as well, but in this case, it is the value to count down.

The loop begins by incrementing the value in cell 0 with 48 `+` commands. This is boring but quick to implement. The current loop count is now a valid ASCII digit, so we print it and subtract 48 to get back to the actual count. The bottom of the loop looks at cell 0, which is always 10, and prints it to get the newline character. The code then looks again at cell 1, where our count lives, and loops until zero.

Bunches O'Bits

Bit twiddling, meaning fiddling around with the bits of a byte, is the goal of this section. Here we'll implement two examples. The first calculates the *ones' complement* of a byte. The second calculates the *even parity bit*. Don't be concerned if these terms are new to you, I'll clarify as we go.

A Complimentary Complement

Internally, computers represent integers as a set number of bits, that is, as a base-2 number. One method for encoding negative numbers is to use the *ones' complement*, where each bit is the opposite of what it would be for a positive value. For example, if a number is 00001101_2 = 11, then 11110010_2 = −11 where each one is now a zero and vice versa. In this encoding, the leading bit will be one when the number should be interpreted as a negative value. Our goal is to write a BF program to calculate the ones' complement of an input byte. The byte will be entered as a string of eight characters (each zero or one).

Let's think about this task for a bit (or eight). We know we'll likely want a loop to read eight bits. After reading a bit, we need to subtract 48 to map the ASCII value read to its actual value (0 or 1). Once we have the actual value, we then output a 0 if the value is 1 or a 1 if the value is 0. In typical languages, a simple `IF` statement would do the trick. Of course, we're not working with an ordinary language, but rather in the strange world world of BF.

A loop to read a byte's worth of bits could be written as,

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

In this code, we first set cell 0 to eight and then start a loop. The loop decrements cell 0, moves to cell 1, and inputs something. It then moves the memory pointer back to cell 0 and loops if the count isn't zero. This reads eight characters and then exits. Adding a period after the comma echoes the input. Of course, we need a zero or one in memory, not the ASCII code for zero or one, so somewhere we'll have to subtract 48. We'll use a sequence of 48 - instructions.

Alright, we have the input bit, but how do we decide whether we should output a zero or a one? If the bit is one, we could enter a loop that is otherwise skipped if the bit is zero. How can we use that? Well, we might be able to set another memory location to one, read the input bit, and if it is one, decrement the preset memory location. If we do that, we'll be in business. However, before we go too far, it's a good idea to make a map of how we are using BF memory. So far, we have this setup:

```
cell   :   0    1    2
value  :   8  0|1    1
pointer:        ^
```

where our loop counter is in cell 0, the bit entered by the user is in cell 1, cell 2 holds a one, and the memory pointer is looking at cell 1.

If the user's bit is one, we want to enter a loop to decrement cell 2. If the bit is zero, the loop will be skipped and cell 2 will remain one. Then, we print cell 2 and we have it: a one is changed into a zero and a zero is changed into a one. We then move the memory pointer back to cell 0 to decrement the bit counter and repeat until we're done.

Listing 10-7 shows *ones.b*, which implements the above algorithm. Let's walk the code to see that it does what I claim.

```
++++++++[
    ->          decrement mem(0); look at mem(1)
    >+<,        look at mem(2); inc; mem(1); input
    ----------
    ----------
    ----------
    ----------
    --------    sub 48
    [->-<]      if one then dec mem(2)
    >           look at mem(2)
    ++++++++++
    ++++++++++
    ++++++++++
    ++++++++++
    ++++++++.   add 48; print
    [-]<        zero mem(2); look at mem(1)
    <           look at mem(0)
]
++++++++++.     mem(0) = 10; print newline
```

Listing 10-7: Ones' complement

First, the outermost loop uses cell 0 to count down from eight. Inside this loop, after decrementing the count, the memory pointer is moved to cell 2, which is then incremented. We know that cell 2 was initially zero, so it must be one now. We then move back to cell 1 to get the user's input, which we'll assume is a zero or a one. This much is `->>+<,` if the comments are removed. Look at the commands until you are sure you follow what's going on.

The next block of code is an uninspired sequence of 48 `-` commands to change the user's input into either a zero or a one. Recall that we're looking at cell 1. At this point, we have memory as we want it: cell 1 is zero or one and cell 2 is one. The next set of commands are key to the entire program: `[->-<]>`. The small loop (`[->-<]`) executes if the user's bit is 1 because we are looking at cell 1 and it isn't zero; therefore, `[` enters the loop. Cell 1 is immediately decremented to make it zero because we only want the code in the loop to execute once. Next, `>` looks at cell 2 and `-` decrements it to change it from one to zero. Lastly, `<` looks again at cell 1, which is now zero, so the loop exits, and the final `>` executes to look at cell 2. If the user's bit is a zero, then `[` fails, skipping the entire loop and moving directly to the final `>` to also look at cell 2. At this point, cell 2 is the focus, and it contains a one if the input bit was a zero or a zero if the input bit was a one. The next block of 48 `+` commands increments the value in cell 2 to get the corresponding ASCII code and `.` prints it.

What does `[-]` do? As we saw above, `[-]` is the BF idiom to zero a memory location. This is necessary to make sure cell 2 is zero when the outer loop comes around for the next input bit. Right now, cell 2 is either 48 or 49. The final two `<<` instructions move focus back to cell 0, the loop counter.

The outer] then loops if cell 0 isn't zero. When it is, the final line, `++++++++++.` outputs ASCII 10, a newline, and the program exits.

Whew! Let's see Listing 10-7 in action. Run *ones.b* like so,

```
> echo 00001101 | snobol4 bf.sno examples/ones.b dump
11110010

Memory: (mp = 0)
     0:    10
     1:     0
     2:     0
```

The `echo` command is a convenient way to send input to a program without typing it directly. Notice that the input is 11 as we saw it above, 00001101_2. The output is 11110010_2, which is -11 in ones' complement, as we wanted. The memory dump tells us we end the program looking at cell 0, which contains 10 for the final newline. The other two cells used by the program are both zero.

One note before moving on. Listing 10-7 excludes a comment block at the top of *ones.b*. The BF interpreter ignores non-command characters; however, the comments must not include any command characters. That gets a bit annoying at times. The comments at the top of *ones.b* are enclosed in brackets ([and]). This means the entire comment block (at least the characters that are valid BF commands) is a loop. But this doesn't matter. The comment block is the first loop in the program, and we know cell 0 is always zero, so the loop will never execute and we are free to enter whatever text we want in the comments. This was not my idea, but it is another illustration of the creativity present in the esolang community.

Achieving Parity

Serial communication protocols sometimes use a *parity bit*, an extra bit transmitted with the data that makes it easier to detect transmission errors. For example, if the data fits in seven bits, as standard ASCII characters do, then an eighth bit can be added to make the number of one bits (i.e., bits with a value of 1) in the 8-bit byte even. This is known as even parity. If the received byte does not have an even number of one bits, the receiver immediately knows there is an error and can request the byte again. A single parity bit can capture a single-bit error, which is sufficient in most cases.

Our mission is to write a BF program to accept seven input bits and output the proper even parity bit. We'll input bits as a sequence of seven ASCII characters as before and then output either ASCII 0 or ASCII 1 to make the number of one bits even. The following are some examples of bytes with parity bits:

```
0000000  →  00000000
0000010  →  00000101
0011001  →  00110011
1111111  →  11111111
```

The bold output bit ensures that every byte has an even number of ones.

How should we go about getting BF to do this for us? There are likely multiple approaches, but the approach we'll use here is first to tally the number of ones present in the seven inputs. Then we'll decide which bit to output based on this tally. As with *ones.b* above, we need an outer loop to read the ASCII bits and subtract 48. To tally the one bits, we'll increment a memory cell each time the bit is a one.

Listing 10-8 presents a loop to read seven bits and tally the number of one bits.

```
+++++++[       mem(0) = 7
    >,.        mem(1) = input; echo
    --------------------
    --------------------
    --------   sub 48
    [          inner loop if bit is one
        -      subtract the bit from mem(1)
        >+     look at mem(2); increment mem(2)
        <      look at mem(1)
    ]          exit loop because mem(1) is zero
    <-         look at mem(0); decrement mem(0)
]              loop if mem(0) not zero
```

Listing 10-8: Adding the input bits

As always, tracking memory use is essential. In this case, cell 0 holds the bits read counter, cell 1 is the input bit, and cell 2 the tally of one bits. The first part of the loop is `>,.`, which moves to cell 1, reads the input bit, and echoes it. Next comes a block of 48 `-` commands to turn the ASCII character code into a 0 or 1.

If the bit is a one, `[` begins the inner loop. The loop body, `->+<`, decrements cell 1, looks at cell 2 and increments it, and looks again at cell 1. Because cell 1 is now zero, `]` fails and the loop ends. If the input bit is zero, `[` skips ahead to `<-`. In both cases, the memory pointer is looking at cell 1, so `<` looks at cell 0, which `-` then decrements. The final `]` fires to repeat the loop six more times. When the loop ends, cell 2 contains a tally of the number of one bits read and the memory pointer is looking at cell 0. It's important to note that cell 0 and cell 1 are both zero when the outer loop exits.

Cell 2 contains the number of one bits in the input. If this number is odd, the output bit should be one. Otherwise, it should be zero. How do we tell if cell 2 is even or odd? Here's where things get a bit tricky. Our solution is in Listing 10-9, but we must walk through it to understand it.

```
>>             look at mem(2)
[              loop if mem(2) not zero
    [          if mem(2) not zero
        -      subtract one
        >      look at mem(3)
        +      increment it
        >      look at mem(4); which is zero
```

```
    ]           do not loop
❶   <<          look back to mem(2)
    [           if mem(2) not zero
        -       subtract one
        >       look at mem(3)
        -       decrement
        >       look at mem(4); which is zero
    ]           do not loop
    <<          look at mem(2) or mem(0) if sum exhausted
]               loop if not zero
```

Listing 10-9: Using the ones tally to decide the parity bit

In essence, when the outer loop of Listing 10-9 ends, the memory pointer will be looking at cell 0 if the output bit should be one or cell 2 if the output bit should be zero. Additionally, cell 3 will be one if we end at cell 0 and cell 5 will be its default value of zero.

The code before the main loop of Listing 10-9 is simple enough. Move the memory pointer twice to look at cell 2, which has the one bits tally. If this tally is zero, the loop is skipped by [and we move to the final bit of code with the memory pointer looking at cell 2. We'll get to the final bit of code below.

If the tally in cell 2 isn't zero, we enter the main loop of Listing 10-9. The body of this loop has two inner loops, one after the other. The body of the first inner loop is ->+>. It subtracts one from cell 2, looks at and increments cell 3, and then looks at cell 4, which is always zero. Because cell 4 is zero,] exits the loop, meaning the loop never actually loops. Notice that when the loop exits, the memory pointer is looking at cell 4 and cell 3 is one.

If we assume that cell 2 was initially one, ❶ cell 2 is now zero, cell 3 is one, and we are looking at cell 4, which is also zero. The << between the inner loops moves back to cell 2, which, as it is zero, skips the second inner loop and hits the final << to move back from cell 2 to cell 0. Because cell 0 is zero, the outer loop exits, meaning we are looking at cell 0 and cell 3 is still one.

This situation happens every time cell 2 contains an odd value. What if cell 2 contains two? After the first inner loop of Listing 10-9, cell 2 contains one, cell 3 contains one, and we are looking at cell 2. Therefore, the second inner loop fires to decrement cell 2 and cell 3, making them both zero. The loop then moves to cell 4, which is always zero, exits, and moves back to cell 2, which is now also zero. The outer loop then exits, and we are looking at cell 2 this time, not cell 0. Whenever cell 2 is initially even, both inner loops will repeatedly fire to make cell 2 zero. Cell 3 is also decremented by the second inner loop to make sure it only ever contains a one.

We're almost done. The code in Listing 10-9 ends, leaving BF in one of two states. If the tally in cell 2 was even, we're looking at cell 2. If the tally is odd, we're looking at cell 0 and cell 3 is one. To output the proper bit, we need the code in Listing 10-10.

```
>>>                  look at mem(3) or mem(5)
++++++++++++++++++++
++++++++++++++++++++
++++++++.            add 48 and print
>                    look at the next location which is zero
++++++++++.          set to 10 and print the newline
```

Listing 10-10: Printing the proper parity bit

We use >>> to move to either cell 3 or cell 5. Cell 3 would be one if we ended at cell 0 and that's the value we want to output. If we ended at cell 2, we move to cell 5, which is initialized to be zero and is also the value we want. All that remains is to add 48 to convert the value to the ASCII character code for either 1 or 0, print it, and then move to either cell 4 or cell 6, both of which are initially zero, to output the final newline character.

Let's try *parity.b* with the example inputs above,

```
> echo 0000000 | bfi examples/parity.b
00000000
> echo 0000010 | bfi examples/parity.b
00000101
> echo 0011001 | bfi examples/parity.b
00110011
> echo 1111111 | bfi examples/parity.b
11111111
```

The output is as expected, meaning that *parity.b* works. You'll get the same results if you use the SNOBOL interpreter as well.

Now, let's work on our final BF example, multiplication.

Multiplicative Multiplicity

Multiplication is repeated addition. Let's use that fact to write a BF program to accept two single-digit numbers and compute their product. We'll write two versions. The first version implements multiplication but leaves the product in memory. This is unsatisfying, so the second version uses freely available code from *https://esolangs.org/* to output the product as ASCII characters. The full source code for both examples is in *mult.b* and *mult2.b*, respectively.

We require two inputs which we'll store cells 0 and 1. We'll use cells 2 and 3 while multiplying and place the final product in cell 3. Reading the input characters is straightforward; see Listing 10-11.

```
,--------------------
--------------------     mem(0) = input
--------                 sub 48
>+++++++++++++++         mem(1) = 32 (space)
++++++++++++++++.        print
++++++++++.              add 10; print *
```

```
----------.              sub 10; print space
,--------------------    mem(1) = input
--------------------
--------                 sub 48
```

Listing 10-11: Reading and printing the inputs

The first character is read and converted to its numeric value. Then cell 1 is used to output * before reading the second digit.

To multiply, we must increment a memory location as many times as the value in cell 1 dictates (that is, increment it cell 1 times), and repeat until cell 0 is zero. For example, if cell 0 is five and cell 1 is four, the algorithm is to calculate,

$$5 \times 4 = 4 + 4 + 4 + 4 + 4 = 20$$

which we might write in a language like Python as,

```
ans = 0
for i in range(5):
    for j in range(4):
        ans += 1
```

Let's duplicate this code in BF. However, we have a minor issue. We need two loops, an outer loop running until cell 0 is zero, and an inner loop to increment cell 3 cell 1 times. Recall that BF loops are destructive. For example, if we write ++++[.-] we'll print the current value of cell 0 four times, from four down to one. When the loop exits, cell 0 is zero, meaning its original value has been lost. Thus, we must preserve the value of cell 1 to use it again on the next pass.

Listing 10-12 shows us the multiplication algorithm. Let's see how it preserves the inner loop counter.

```
<[-                 dec mem(0)
    >[-             look at mem(1); dec
        >+>+<<      inc mem(2); inc mem(3); look at mem(1)
    ]               continue until mem(1) is zero
    >[-<+>]         look at mem(2); copy back to mem(1)
    <<              look at mem(0)
]                   loop until mem(0) is zero
```

Listing 10-12: Multiplying the two digits

The first < moves us back to cell 0 as the code in Listing 10-11 ends with the memory pointer looking at cell 1. Cell 0 isn't zero, generally, so the outer loop begins, and cell 0 is immediately decremented. Next, > moves to cell 1, and the first inner loop begins if cell 1 isn't zero. Cell 1 is also immediately decremented.

The body of the first inner loop is >+>+<<. The >+ instructions move to cell 2 and increment it. The following >+ does the same to cell 3. Lastly, << moves back to cell 1 so] can decide whether to continue the loop or not.

When the loop exits, memory looks like this assuming the user entered 5 and 4,

```
cell   :   0    1    2    3
value  :   4    0    4    4
pointer:        ^
```

Recall that cell 3 holds our product. It's currently 4 because the first inner loop ran four times. We now must restore cell 1. That's why the loop incremented both cell 2 and cell 3. We use cell 3 for the product and can use cell 2 to restore the inner loop counter. That's what >[-<+>] does, it decrements cell 2 while incrementing cell 1. The final << ensures that the outer loop's] instruction is looking at cell 0.

Each pass through the outer loop adds cell 1 to cell 3, using cell 2 to restore cell 1 for the next pass. When cell 0 is finally zero, cell 3 holds the product. Note that we entered single digits, but this multiplication routine is generic and will work for any two values. Also, the trick of double incrementing memory to have a place to restore from is another BF idiom. We saw similar code in Chapter 8 when we implemented multiplication in FRACTRAN.

Let's take *mult.b* out for a test drive. We'll use our interpreter's ability to dump memory to see if it is working. For example,

```
> snobol4 bf.sno examples/mult.b dump
3 * 5

Memory: (mp = 0)
     0:    10
     1:     5
     2:     0
     3:    15

> snobol4 bf.sno examples/mult.b dump
9 * 8

Memory: (mp = 0)
     0:    10
     1:     8
     2:     0
     3:    72  H
```

In both cases, we see that cell 3 contains the correct product.

To print the product as a number, we add the code in Listing 10-13 to the end of the multiplication routine in Listing 10-12. See *mult2.b*.

```
>>>                                 move to mem(3)
>[-]>[-]+>[-]+<                     set n and d to one to start loop
[                                   loop on 'n'
    >[-<-                           on the first loop
```

```
    <<[->+>+<<]                  copy V into N (and Z)
    >[-<+>]>>                    restore V from Z
  ]
  ++++++++++>[-]+>[-]>[-]>[-]<<<<<  init for the division by 10
  [->-[>+>>]>[[-<+>]+>+>>]<<<<<]    full division
  >>-[-<<+>>]                    store remainder into n
  <[-]++++++++[-<++++++>]        make it an ASCII digit; clear d
  >>[-<<+>>]                     move quotient into d
  <<                             shuffle; new n is where d was and
                                   old n is a digit
  ]                              end loop when n is zero
<[.[-]<]                         Move to were Z should be and
                                 output the digits till we find Z
<                                back to V
<++++++++++.                     newline
```

Listing 10-13: The print routine

As mentioned, this routine comes from *https://esolangs.org/*. What is particularly nice about this routine is that it works with any memory location, so we move from cell 0 to cell 3 prior to running it. The provided comments give some indication of what the routine is doing. Notice that the second line of Listing 10-13 uses the "clear a cell" idiom three times to initialize memory. We won't walk through Listing 10-13 in any detail as it is quite challenging. Motivated readers will find the code, with some additional details, at *https://esolangs.org/wiki/Brainfuck_algorithms/* under the heading beginning with "Print value of cell x as number."

Let's review a couple examples to see that the routine works as advertised.

```
> snobol4 bf.sno examples/mult2.b
3 * 5 = 15
> snobol4 bf.sno examples/mult2.b
9 * 8 = 72
```

The examples of this section, *ones.b*, *parity.b*, *mult.b*, and *mult2.b* serve as our introduction to BF. There's much more we might say, but we covered the essentials. Let's turn now to outside resources to see additional examples, learn more about BF programming, and gain insight on how BF has influenced esolangs as a whole, to say nothing of genuine academic research involving BF.

The BF Multiverse

If Piet generated a universe, then to be fair, we must say that BF has created a multiverse. Let's briefly investigate some of those universes in this section: examples, tutorials, implementations, inspirations, and academic BF. Enjoy!

Examples

The best way to learn a language is to use it. We did that in the previous section. The next best way to learn a language is to see how others have used it. Let's take a cursory look at the BF examples included with this book. I did not write these examples. See the *README* file for attribution information.

The most impressive set of BF programs written by hand and not generated by another system producing BF code as output I've found are by Daniel B. Cristofani. You'll find them at *https://brainfuck.org/*, which alone tells you Cristofani's a serious BF coder—he registered the domain name. I suspect you'll learn much from the examples and even more from the tutorial information on his site.

The book repository contains the following, all of which run with both the C and SNOBOL interpreters:

squares.b Print n^2 for $[0, 100]$.

fib.b Generate an endless stream of Fibonacci numbers. We encountered the Fibonacci sequence in Chapter 1 and again in Chapter 7. This version does not use a single cell to hold the number, but rather handles arbitrary-sized numbers. This is a good example of how compact BF code can be while still doing something interesting.

factorial2.b Another gem. This one calculates an endless stream of factorials.

sierpinski.b The Sierpinski triangle is a common fractal, one that a straightforward algorithm can generate. This version produces ASCII output. We'll work with the Sierpinski triangle again in Chapter 14. Consider this example a preview.

random.b Implements Wolfram's Rule 30, a one-dimensional cellular automaton. This automaton, especially the center bit, passes many tests for randomness and formed the basis for Mathematica's first pseudo-random generator. To experiment more with Rule 30 and other one-dimensional automatons, see Chapter 7 of my book, *Random Numbers and Computers*.

golden.b Calculates the decimal expansion of $\phi = \frac{1+\sqrt{5}}{2}$.

e.b Calculates the decimal expansion of e, the base of the natural logarithm. The natural log can be defined via an integral, $\ln x = \int_1^x \frac{1}{t} dt$ with e the limit such that the log is one, $1 = \int_1^e \frac{1}{t} dt$.

tictactoe.b Tic-tac-toe in BF. You against the computer. Good luck.

The remaining examples, beyond *cat.b* and *hello.b* which we saw above, include:

prime.b Calculate prime numbers less than the given number. This commented example was written by hand, but I have not succeeded in identifying the author.

hanoi.b An animated Towers of Hanoi. This example is the output of Claire Wolf's BF compiler suite (see below). It's fun to watch, but run it with the C interpreter or you'll be waiting a very long time indeed.

mandelbrot.b Creates an ASCII version of the Mandelbrot set. The *README* file gives the URL of the code. It appears to be the output of Wolf's BF compiler as well. If you use the SNOBOL interpreter, you'll eventually finish, but it runs at about a hundred times slower than the C interpreter.

Tutorials

The tutorials here offer plenty of good BF programming insights, idioms, and explanations.

Daniel B Cristofani's BF Pages Mentioned above but worth mentioning again because of the helpful programming advice. You'll even find advice on how to write a "compliant" interpreter. Our SNOBOL interpreter is not compliant, but we're happy with it. (*https://brainfuck.org/*)

Frans Faase's BF Pages You'll find many good reference/tutorial pages here. Some are Faase's whereas others are links to still more information about BF. The World Wide Web is a web, after all. (*https://www.iwriteiam.nl/Ha_BF.html*)

Katie Ball's BF Tutorial Ball's tutorial is another good reference. (*https://gist.github.com/roachhd/dce54bec8ba55fb17d3a*)

Implementations

The implementations of BF are legion, which is somehow fitting. A tiny selection is referenced here, and I'm completely ignoring all the hardware implementations.

Compilers

The phrase *BF compiler* has multiple meanings. For example, a BF compiler might be a program that takes a higher-level language and produces BF code. In that case, BF is the machine code for the compiler. Alternatively, a BF compiler might be just that, a program that takes BF as input and produces executable code from it. I offer an example of each kind here.

Brian Raiter's Native BF Compiler As promised above, here's Brian Raiter's 166 byte BF compiler. It's written in assembly language (install `nasm` on Linux) and produces standalone executables. Not every example in the repository works with this compiler, but many do, and the results are significantly faster than even the C interpreter. Try *e.b*, *golden.b*, and *tictactoe.b*. There are many comments in the source code, *bf.asm*. Hopefully, your x86 assembly is much stronger than mine. (*http://www.muppetlabs.com/~breadbox/software/tiny/bf.asm.txt*)

Claire Wolf's Compiler to BF This compiler takes a higher-level macro language and produces executable BF code. It produced two of our examples: *hanoi.b* and *mandelbrot.b*. (*http://bygone.clairexen.net/bfcpu/bfcomp.html*)

Interpreters

We saw how easy it is to write a BF interpreter, even in SNOBOL. The two links here point to large lists of BF interpreters in all kinds of languages.

Esolangs.org's BF Implementations This page has a long list of BF and BF-related goods and services, er, implementations. (*https://esolangs.org/wiki/Brainfuck_implementations*)

Rosetta Code's BF Implementations BF interpreters in a plethora of languages. Neither Jefe nor I are responsible for time or bits lost due to incomplete or erroneous code. (*http://rosettacode.org/wiki/Execute_Brain****)

Inspirations

Perhaps the greatest tribute to BF is that it has inspired many other esolangs. Some are serious, genuine extensions to core BF. Others are less serious or even outright jokes. If you browse the (long) language list at *esolangs.org/wiki/Language_list*, you'll recognize many BF-related languages from nothing more than the colorful, if not sometimes offensive, names.

Academic BF

BF isn't all just fun and games. The language is elementary, yet Turing complete. This makes it attractive to researchers looking for a target or other language to use in their systems. The references here are to academic papers that use BF, either actively or as an example. What's particularly interesting is that not all of the references are from traditional computer science journals. BF's useful even in relation to more traditional human pursuits, like poetry. This list is by no means exhaustive, merely illustrative, and favoring more recent references to BF.

BF++: A Language for General-purpose Program Synthesis, Vadim Liventsev, Aki Härmä, and Milan Petkovic (2021).

Neural Program Synthesis with Priority Queue Training, Daniel A. Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le (2018).

Resisting Clarity/Highlighting Form: Comparing Vanguard Approaches in Poetry and Programming, Irina Lyubchenko (2020).

Fully Human, Fully Machine: Rhetorics of Digital Disembodiment in Programming, Brandee Easter (2020).

50,000,000,000 Instructions per Second: Design and Implementation of a 256-Core BrainFuck Computer, Sang-Woo Jun (2016).

A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics, Michael Mateas and Nick Montfort (2005).

The first two references use BF with reinforcement learning, thereby combining esolangs and deep machine learning. Advanced neural networks generate BF programs to solve problems.

Discussion

BF is Turing complete. It's imperative, has the requisite control structures (brackets), and, ignoring the self-imposed 30,000 cell memory limit, uses arbitrary memory. Additionally, and impressively, Daniel Cristofani implemented a universal Turing machine in BF, thereby directly demonstrating Turing completeness. The machine is in *utm.b* in the BF examples directory. Comments in the file explain, in detail, what the program is and what it means.

There's a certain enticing nature to BF due to its simplicity. Yes, it's challenging to work with, which might have been intentional, like a gauntlet thrown down to see who might pick it up. But I don't view BF that way. Life is built from the combinatorial mixing of a multitude of smaller components. Might it be possible to view something like BF as the DNA of programming? We already know from Chapter 3 that a Turing machine captures the essence of what an algorithm is. BF is more advanced than a Turing machine, but just barely, so it can serve the same purpose as an encapsulation of the idea of an "algorithm."

In his famous *Epigrams on Programming*, Alan Perlis wrote,

> 19. A language that doesn't affect the way you think about programming, is not worth knowing.

This is true for every language in this book, but I hope it is especially so for the esolangs, with BF chief among them. Struggling to write code in BF, especially when decades of experience make the necessary code almost instantly present itself in more familiar languages, does affect the way you think about programming. I found myself trying, with varying levels of success, to think in a new way to understand how to fit what BF offers to what I would instinctively do in a language like Python or C. Perhaps that's the most enduring effect of learning BF. It requires you to think in new ways instead of relying on what is already familiar. BF is a way out of the Python (or C or Java or ...) echo chamber, as it were.

Perlis offers more wisdom directly applicable to BF,

> 23. To understand a program you must become both the machine and the program.

For modern, high-level languages, we need not think about the machine too much. Indeed, modern languages go to great lengths to abstract themselves from the machine. With BF, as with a Turing machine, we must consider both the machine and the program if we hope to be successful.

As we're quoting Perlis, I'd be remiss not to include this epigram,

> 54. Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Turing tar-pits might be a bit like beauty—in the eye of the beholder. The following that has grown around BF and, by extension, esolangs in general, argues against Perlis in this case, at least to me. Perlis' first epigram is "One man's constant is another man's variable." I'm tempted to rephrase it: "One man's Turing tar-pit is another man's inspiration."

Chapter Summary

This chapter introduced us to the strangely attractive, if frustratingly difficult, multiverse of BF. We explored what BF is and then implemented it twice: once in C using the original implementation and again in SNOBOL. After this, we wrote a few example programs to get a feel for thinking in BF. With a basic grasp of the language in hand, we then turned our gaze upward to examine some of the brighter lights in the BF multiverse. As with every language, we closed the chapter with a brief discussion.

In Chapter 9 we painted pretty pictures with Piet, a two-dimensional language. Let's close our survey of existing esolangs by returning to the world of 2D programming, but this time we'll use text instead of pixels. Next stop, Befunge.