

# An Introduction to Esoteric Programming Languages, JSFuck and Brainfuck

By James Oswald

# Workshop Overview


- 1) Introduction to Esoteric languages
  - a) About
  - b) Whitespace
  - c) Piet
  - d) Malbolge
  - e) LOLCODE
- 2) JSFuck
  - a) About
  - b) Numbers and Booleans
  - c) Letters and Strings
  - d) Execution overview
  - e) Sample programs
- 3) Brainfuck
  - a) About
  - b) Commands
  - c) Exercises
  - d) Sample Programs & Walkthrough
  - e) Interpreters



IEEE



UALBANY STUDENT  
BRANCH



# What Are Esoteric Programming Languages?

# Esoteric Programming Languages

An **Esoteric Programming Language** or “**Esolang**” is a programming language created for any reason other than practical everyday use. Some of the reasons people create the languages are for:

- Proofs of concept (Brainfuck, JSFuck, BCL)
- Jokes (Ook!, Pikachu, Whitespace, Malbolge, LOLCODE)
- Art (Piet)



# What Makes an Esolang?

Most Esolangs, like all other practical programming languages try to remain **Turing Complete** while imposing other strange rule sets around the baseline of remaining turing complete.

A programming language is turing complete if it can be used to solve any computational problem using a Universal Turing Machine, with no memory or speed guarantees.



# Some Esoteric Languages

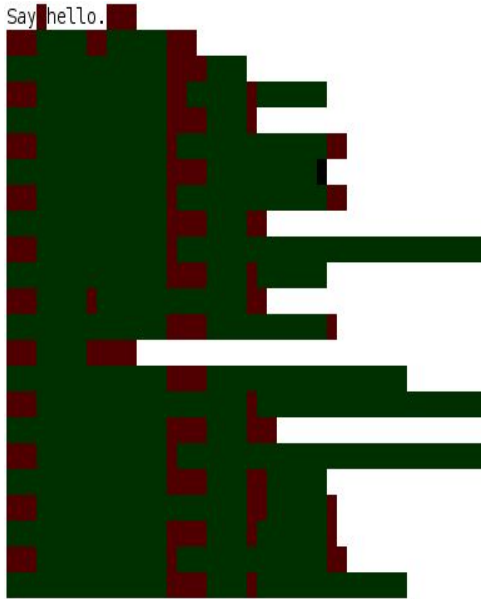
# Whitespace

Whitespace is an esolang composed entirely of whitespace characters [Space], [Tab], and [LineFeed]. Because of this white space programs are invisible in normal text editors

Different combinations of these characters indicate different “instructions” and their operands, making whitespace very assembly language esque.

[Documentation](#)

Say hello.



7,8-32 Anfang

Hello World! In Whitespace with highlighting

# Piet (Pronounced Peet)

Piet is an art programming language that uses colorful bitmaps that look like modern art as programs. The language is named after Piet Mondrian who painted the original piece that inspired the language.

The Language itself is stack based and executes commands depending on colors and arrangements of blocks.

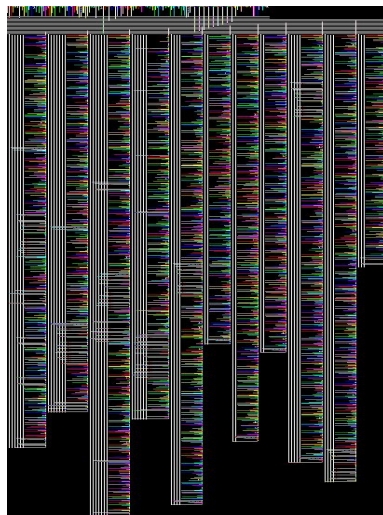
[Documentation](#), [Program Gallery](#)



Hello World



Euclid's Algorithm



Text Adventure Game



Day of the Week Calculator



# Malbolge (Pronounced mal-bulge)

Malbolge is an esolang created to be the most ungodly difficult language to program in.

- It uses base 3 numbers of length 10 rather than bytes
- The operation performed is determined as  $\text{Opcode} = ([c] + c) \% 94$  where  $c$  is the instruction pointer
- After an operation is performed, the instruction encrypts itself making looping near impossible.
- Normal arithmetic is replaced with trit rotation and the “crazy operation” (don’t ask)
- Code is represented in ascii but still loaded in as base 3 numbers of length 10.
- Hello World is `(=<`#9]~6ZY32Vx/4Rs+0No-&Jk)"Fh}|Bcy?`=*z]Kw%oG4UUS0/@-ejc(:'8dc`
- It took 6 years to write a 99 bottles of beer program ([Link](#)).

[Documentation](#)



# LOLCODE

LOLCODE is an esolang that speaks in meme. It's the closest thing to a real programming language on this list and resembles COBOL if you look hard enough. It's also surprisingly easy to read.

## Hello World

```
HAI 1.2
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

## Adder

```
HAI 1.2
BTW computes the sum of 2 numbers
I HAS A x
I HAS A y
I HAS A s
GIMMEH x
GIMMEH y
x IS NOW A NUMBR
y IS NOW A NUMBR
s R SUM OF x AN Y
VISIBLE s
KTHXBYE
```

[Documentation](#), [Official Site](#)

## A few sample instructions

LOLCODE Construct	Purpose/ Usage
<b>BTW</b>	It starts a single line comment.
<b>DOWN</b> <variable>!!<times>	This corresponds to variable = variable - times. Note that "times" is a wut-only language extension.
<b>GIMMEH</b> <variable>	This represents the input statement.
<b>GTFO</b>	This is similar to <b>break</b> in other languages and provides a way to break out of a loop.
<b>HAI</b>	This corresponds to <b>main ()</b> function in other languages. It is the program entry point in LOLCODE.
<b>HEREZ</b> <label>	This is another wut-only language extension and declares a label for use with SHOO
<b>I HAS A</b> <type> <variable>	This declares a variable of said type. There are three built-in types in LOLCODE: NUMBAH (int) DECINUMBAH (double) WORDZ (std::string) Note that types are a wut-only language extension.
<b>IM IN YR LOOP</b>	This starts an infinite loop. The only way to exit the loop is using GTFO. Corresponds to <b>for(;;)</b> in other languages
<b>IZ</b> <expr1> <operator> <expr2>?: <b>structure</b> <operator> <expr2>?: <b>KTHX</b> <b>KTHXBAI</b> <b>NOWAI</b> <b>PURR</b> <expr>	This is similar to if operator in other languages. Operator is one of: BIGGER THAN, SMALLER THAN, SAEM AS. Note that the ? at the end is optional. It ends a block. Corresponds to <b>}</b> This ends a program This corresponds to else This prints argument on screen, followed by a newline. It is a wut-only language extension.
<b>RELSE</b> <b>SHOO</b>	This corresponds to <b>else (if)</b> This is another wut-only language extension, that corresponds to <b>goto</b> (the horror!)
<b>UP</b> <variable>!!<times>	This corresponds to variables = variable + times. Here



# JSFuck

# JSFuck

JSFuck is a proof of concept programming language and is simply Javascript Code that only uses the six characters: `[]()!+`

JSFuck is ALWAYS valid Javascript, it does this by taking advantage of two of Javascript's infamous design decisions, weak typing, and Javascript's nonsensical type equality system.

JSFuck is all about forcing the JS interpreter to recognize types in new ways.

Fun Fact: Any JS program can be fully converted into only JSFuck.



	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[] ]	[0]	[1]	NaN
true	==	*	IR	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*	*	IR	*
false	*	==	*	IR	*	*	*	*	IR	*	IR	*	*	*	*	IR	*	IR	IR	*	*
1	IR	*	==	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*	*	IR	*
0	*	IR	*	==	*	*	*	*	IR	*	IR	*	*	*	*	IR	*	IR	IR	*	*
-1	*	*	*	*	==	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*	*
"true"	*	*	*	*	*	==	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
"false"	*	*	*	*	*	*	==	*	*	*	*	*	*	*	*	*	*	*	*	*	*
"1"	IR	*	IR	*	*	*	*	==	*	*	*	*	*	*	*	*	*	*	*	IR	*
"0"	*	IR	*	IR	*	*	*	*	==	*	*	*	*	*	*	*	*	*	IR	*	*
"-1"	*	*	*	*	IR	*	*	*	*	==	*	*	*	*	*	*	*	*	*	*	*
""	*	IR	*	IR	*	*	*	*	*	*	==	*	*	*	*	IR	*	IR	*	*	*
null	*	*	*	*	*	*	*	*	*	*	*	==	IR	*	*	*	*	*	*	*	*
undefined	*	*	*	*	*	*	*	*	*	*	*	IR	==	*	*	*	*	*	*	*	*
Infinity	*	*	*	*	*	*	*	*	*	*	*	*	*	==	*	*	*	*	*	*	*
-Infinity	*	*	*	*	*	*	*	*	*	*	*	*	*	*	==	*	*	*	*	*	*
[]	*	IR	*	IR	*	*	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*
{}	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
[[]]	*	IR	*	IR	*	*	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*
[0]	*	IR	*	IR	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*	*	*
[1]	IR	*	IR	*	*	*	*	IR	*	*	*	*	*	*	*	*	*	*	*	*	*
NaN	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*



Not Equal



Loose equality  
often give false positives  
like "1" is true; [] is "0"



Strict equality

# Javascript's Insane Type Equality System

# Building valid JS code from six symbols (Numbers)

To start off, the **empty array** `[]`, which from the last slide is **loosely equal** to 0.

When applying the unary `+` operator to `[]` we force the interpreter to recognize `[]` as a number type. So the expression `+[[]]` forces the interpreter to recognize `[]` as 0 by way of the `+`.

From here we use `!` to force the interpreter to recognize that 0 as a boolean type meaning `!+[[]]` evaluates to `true`

We can use then use `+` to force true to evaluate to a number, `+++[[]]` evaluates to 1

From here we use a simple pattern to build any number:

`!+[[]]+!+[[]]` is broken up as `true + true`, the binary plus forces this to interpret true as a number, hence the result is 2



# Numbers and Booleans in JSF, Exercises!

Knowing that `+` forces numbers and `!` forces booleans, we discovered `+[ ] == 0`, `!+[ ] == true`, `+!+[ ] == 1`, and `!+[ ]+!+[ ] == 2`

Solve these problems:

1) What are 3 ways we could represent `false` in 5 or less characters?

`![]` , `!!+[ ]` , and `!!![]`

2) How could we represent the number 3?

`!+[ ]+!+[ ]+!+[ ]`

3) There is a way to represent 2 with only a single `+`, find it.

`!![]+!![]`



# Building valid JS code from six symbols (Strings)


We can now build any number but to write valid JS code we need letters, JSFuck uses more black magic: `[]+[]`, Binary + forces the interpreter to recognize `[]` as an empty string `""`, concatenating them as `""`.

Using binary + with `""` we can concat the name of any type value and turn it into a string getting its letters. IE `false + "" == "false"` Better yet we can get `"a"` as `(false + "")[1]` expanding this with everything else we learned:

```
(![]+([]+[]))[+!+[]]
```

JSFuck uses many tricks like this to get a valid combination of six symbols for every single available character.

Once we have every character we can make any string by using binary + on individual characters.





# Letters and Strings in JSF, Exercises!

We learned how to use the empty string `[]+[] == ""` and used this to extract a letter from false `(![]+([]+[]))[+!+[]] == (false + "")[1] == "a"`

Given that `[] [[]] == undefined` and `+[]![] == NaN` solve these:

- 1) Find a representation of the string "true", use this to find "t"

`"true" = true + "" = !![]+([]+[])` and `"t" = (true + "")[0] = (!![]+([]+[]))[+[]]`

- 2) Find a representation of the string "flu" (Hint: Use 'u' from undefined)

`"Flu" = "f"+"l"+"u" = (false + "")[0] + (false + "")[2] + (undefined + "")[0]`  
`== (![]+([]+[]))[+[]] + (![]+([]+[]))!![]+!![] + ([[]]+([]+[]))[+[]]`

- 3) Find a representation of the string "Nut"

`"Nut" = "N"+"u"+"t" = (NaN + "")[0] + (undefined + "")[0] + (true + "")[0]`  
`== (+[]![]+([]+[]))[+[]] + ([[]]+([]+[]))[+[]] + (!![]+([]+[]))[+[]]`

# Letters In JSFuck

USE\_CHAR\_CODE means the symbol is impossible to create in a cool way so instead uses the string `\uXXX` representing the unicode character and guaranteed to be creatable since `u`, `\` and any number can be created using JSFuck.

```
'a': '(false+""[1]','
'b': '([][["entries"]]()+"")[2]','
'c': '([][["fill"]+"")[3]','
'd': '(undefined+""[2]','
'e': '(true+""[3]','
'f': '(false+""[0]','
'g': '(false+[0]+String)[20]','
'h': '+(101))[ "to"+String["name"]](21)[1]','
'i': '([false]+undefined)[10]','
'j': '([][["entries"]]()+"")[3]','
'k': '+(20))[ "to"+String["name"]](21)','
'l': '(false+""[2]','
'm': '(Number+""[11]','
'n': '(undefined+""[1]','
'o': '(true+[][["fill"]])[10]','
'p': '+(211))[ "to"+String["name"]](31)[1]','
'q': '(""["fontcolor"]([0]+false+)"[20]','
'r': '(true+""[1]','
's': '(false+""[3]','
't': '(true+""[0]','
'u': '(undefined+""[0]','
'v': '+(31))[ "to"+String["name"]](32)','
'w': '+(32))[ "to"+String["name"]](33)','
'x': '+(101))[ "to"+String["name"]](34)[1]','
'y': '(NaN+[Infinity])[10]','
'z': '+(35))[ "to"+String["name"]](36)',
```

```
'A': '([+Array])[10]',
'B': '([+Boolean])[10]',
'C': 'Function("return escape")()(("[italics"]())[2]',
'D': 'Function("return escape")()([fill])[slice]("-
'E': '(RegExp+"))[12]',
'F': '([+Function])[10]',
'G': '(false+Function("return Date"))()[30]',
'H': USE_CHAR_CODE,
'I': '(Infinity+")[0]',
'J': USE_CHAR_CODE,
'K': USE_CHAR_CODE,
'L': USE_CHAR_CODE,
'M': '(true+Function("return Date"))()[30]',
'N': '(NaN+")[0]',
'O': '([+Object])[10]',
'P': USE_CHAR_CODE,
'Q': USE_CHAR_CODE,
'R': '([+RegExp])[10]',
'S': '([+String])[10]',
'T': '(NaN+Function("return Date"))()[30]',
'U': '(NaN+Object()("to"+String["name"])[call]())[11]',
'V': USE_CHAR_CODE,
'W': USE_CHAR_CODE,
'X': USE_CHAR_CODE,
'Y': USE_CHAR_CODE,
'Z': USE_CHAR_CODE,
```


```
'(NaN+[]["fill"])[11]',  
':': USE_CHAR_CODE,  
'"': '("")["fontcolor"]()([12]',  
'#': USE_CHAR_CODE,  
'$': USE_CHAR_CODE,  
'%': 'Function("return escape")()(()["fill"])[21]',  
& ': '("")["fontcolor"]()"([13]',  
\ : USE_CHAR_CODE,  
( : '('(["fill"]+"")([13]',  
) : '( [0]+false+[ ] ["fill"]) ([20]',  
*: USE_CHAR_CODE,  
, : '+(+ [+], +[+] [+] [+][+] [+][+] [+][+] [+][+] [+][+] [+][+] ) ([2]',  
, : '([ ][ "slice"] ("call")( false+"") +"") ([1]',  
- : '+(. +(0000001)) +"") ([2]',  
. : '+ (+[+] [+][+] + (![ ]) + [ ] [+][+] [+][+] [+][+] [+][+] + [ ] + [ ] ) + [ ] + [ ] ) ',  
/: ' (false+[0]) ["italics"] () ([10]',  
': ' (RegExp()+"" ) ([3]',  
; : '("") ["fontcolor"] ( NaN+" ") ([21]',  
< : '("") ["italics"] () ([0]',  
=: '("") ["fontcolor"] () ([11]',  
> : '("") ["italics"] () ([2]',  
? : ' (RegExp()+ "" ) ([2]',  
@ : USE_CHAR_CODE,  
[' : ' ([ ] ["entries"] () + "" ) ([0]',  
\\ : ' ( RegExp("/") + "" ) ([1]',  
] : ' ([ ] ["entries"] () + "" ) ([22]',  
^ : USE_CHAR_CODE,  
_ : USE_CHAR_CODE,  
~ : USE_CHAR_CODE,  
{ : ' ( true+[ ] ["fill"] ) ([20]',  
| : USE_CHAR_CODE,  
} : ' ([ ] ["fill"] + "" ) ["slice"] ("- -1')',  
~ : USE CHAR CODE
```

# Executing JSFuck

So far we can create JS expressions out of six symbols that evaluate to numbers and letters and strings, but in order to actually run code we need to use the **eval** function.

Eval is a built in JS function that will execute whatever string is passed as a parameter as javascript. EX. `eval("console.log('hello')")` prints hello to the console. However since we `eval(JSFuckExpressionHere)` isn't valid JSFuck in itself this would be cheating.

Thankfully there is a very creative and complex workaround to this allowing us to effectively run any string using only the symbols we have available to us.



# The Very Creative and Complex Workaround (Prototypes)

Javascript is a prototype based language, we can access any object's prototype, the prototype contains a list of properties and methods of the object's "pseudo class".

This is how JS knows if something like `d.get()` is legal without needing class information. The interpreter will look in the prototype of `d` see if `".get()"` is valid and if so will run the method from the Function object reference in the prototype.

You can access an objects prototype using `'object["protoFunctionName"]'`

**Console Demo of Prototypes**



# The Very Creative and Complex Workaround (Array Prototypes & Function Prototypes)

`[]` is an array object. Since `[]` is an Array Object we can access the Array Prototype which contains all JS's built in array methods like “push”.

```
[]["push"] == f push() { [native code] }
```

Pick an arbitrary array method, this method itself has a prototype, it is a Function Object. Inside the function object prototype is the function object “constructor”.

```
[]["push"]["constructor"] = f Function() { [native code] }
```



# The Very Creative and Complex Workaround (The Function Object Constructor)

We now have access to the function object constructor. The function object constructor takes a single string of JS code as a parameter and will use it as the body of a function.

```
Let f = new Function("alert('oh boy')");
```

The function Object is special in another way as well, a function object suffixed with () will be executed as a function ie:

```
new Function("alert('oh boy')")();
```




# The Very Creative and Complex Workaround (Putting it All Together)

We take the function object constructor from `array.push.constructor`, give it code as a parameter and tell it to execute our a function all in one by stating:

```
[][ "push" ][ "constructor" ] ( "alert('hello')" ) ( )
```

With this we can create the strings “push” and “constructor” and all of our code using the method to create all of our strings with JSFuck. This allows us to create and execute any Javascript program using only these 6 symbols.



```
[][ "push" ][ "constructor" ]( "alert('hello')" )() =
```

[illegible]


# A Simple JSFuck Program... Oh dear god



# But why should I care about JSFuck?

JSFuck is a great esoteric language that is made cooler by knowing every JSFuck program is valid javascript and every javascript program can be written in JSFuck. For example, looking at JQuery Skewed, it is a complete, working implementation of JQuery in JSFuck that works with any normal Javascript code. ([Link to JQ Skewed](#))

JSFuck is used as a tool for hacking, and most sites aren't clever enough to check if JSFuck is valid javascript code, making it much more likely to be successfully injected without setting off anti-scripting alarms. Ebay was known to be hit with a massive JSFuck attack that allowed users to upload custom JSFuck code to their profiles. ([Link to article](#))



The background is a solid pink color. In the top right corner, there is a geometric pattern consisting of several squares and triangles in different shades of pink and magenta, creating a stepped, architectural look.

# Brainfuck

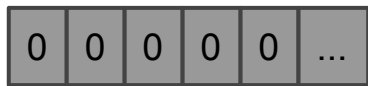
# What is Brainfuck?

**Brainfuck** is a proof of concept programming language that uses only eight symbols and is widely considered the most famous esoteric programming language, inspiring many spin-offs and copycats with its fame.

The Brainfuck machine is visualized as an infinite (or finite) tape of cells on which cells contain bytes. The machine has a pointer that points to a cell on the tape. A separate tape exists containing instructions and an instruction pointer.

A Brainfuck Abstract machine in its default state:

Memory Tape



↑ Memory Pointer

Instruction Tape (Instructions loaded by default)



↑ Instruction Pointer

# Brainfuck Instructions

All 8 of the brainfuck instructions center around manipulating cells, the cell pointer, and the instruction pointer, the instruction tape is immutable (and hence the instructions themselves are also immutable). Let's look at the instructions and their descriptions:

>	moves the cell pointer right
<	moves the cell pointer left
+	Adds 1 to the current cell pointed to by the cell pointer
-	Subtracts 1 from the current cell pointed to by the cell pointer
.	Prints the ascii value of the cell pointed to by the cell pointer
,	Reads an ascii value to the cell pointed to by the cell pointer
[	If cell pointed to by the cell pointer is 0 jump to matching ]
]	If cell pointed to by the cell pointer is not 0 jump to matching [



## > (Shift Memory Pointer Right)

Before:

Memory Tape



Memory Pointer

Instruction Tape



Instruction Pointer

After:

Memory Tape



Memory Pointer

Instruction Tape



Instruction Pointer

# < (Shift Memory Pointer Left)

Before:

Memory Tape



Memory Pointer

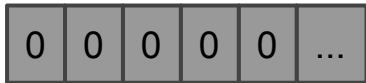
Instruction Tape



Instruction Pointer

After:

Memory Tape



Memory Pointer

Instruction Tape



Instruction Pointer

# + (Add 1 to Cell at Mem Pointer)

Before:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

After:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer



## - (Sub 1 From Cell at Mem Pointer)

Before:

Memory Tape



↑ Memory Pointer

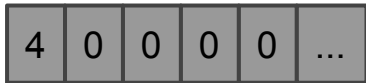
Instruction Tape



↑ Instruction Pointer

After:

Memory Tape



↑ Memory Pointer

Instruction Tape

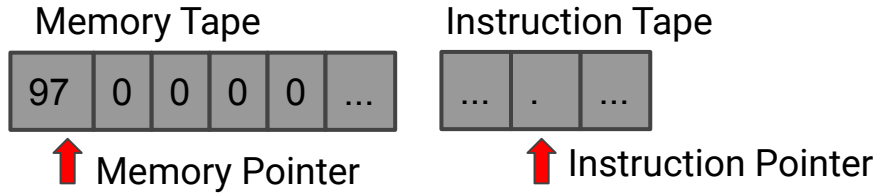


↑ Instruction Pointer



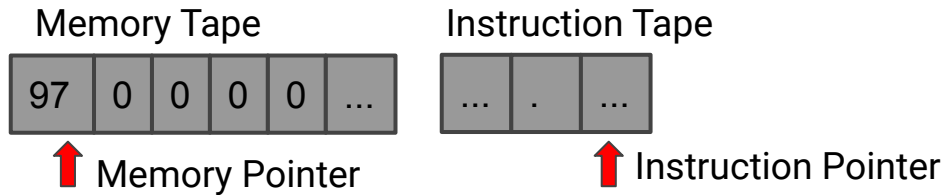
# . (Print Ascii of Cell at Mem Pointer)

Before:



! "a" printed to the console

After:



# , (Read Ascii input to Cell at Mem Pointer)

Before:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

! Prompts user to enter a char, let's say we enter "!"

After:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

# [ (If Cell at Mem Pointer == 0 jump to matching ) ]

Before Case 1:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

After:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

Before Case 2:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

After:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

# ] (If Cell at Mem Pointer $\neq 0$ jump to matching [)

Before Case 1:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

After:

Memory Tape



↑ Memory Pointer

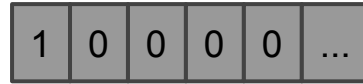
Instruction Tape



↑ Instruction Pointer

Before Case 2:

Memory Tape



↑ Memory Pointer

After:

Memory Tape



↑ Memory Pointer

Instruction Tape



↑ Instruction Pointer

Instruction Tape



↑ Instruction Pointer

# Brainfuck, Exercises!

Using the instruction list to the right solve:

>	moves the cell pointer right
<	moves the cell pointer left
+	Adds 1 to the current cell pointed to by the cell pointer
-	Subtracts 1 from the current cell pointed to by the cell pointer
.	Prints the ascii value of the cell pointed to by the cell pointer
,	Reads an ascii value to the cell pointed to by the cell pointer
[	If cell pointed to by the cell pointer is 0 jump to matching ]
]	If cell pointed to by the cell pointer is not 0 jump to matching [

1) Write a program that reads a single digit number, adds 1, then prints the number

,+.[Link](#)

2) Write a program that reads 3 characters then prints them out backwards

,>>>.<.<.[Link](#)

3) Write a program that reads 3 characters then prints them out forwards

,>>><<<.>.>.[Link](#)

4) (Challenge) Write a program that takes 2 ascii chars as bytes and adds them, and prints the resulting byte as an ascii character (Hint use a loop to add 1 to the first number and subtract 1 from the second until there is nothing left)

,>,[<+>-].

# Sample Program

Add two single digit numbers with a single digit result

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

## An explanation

```
,          ; read character and store it in cell 1
>          ; move pointer to cell 2
,          ; read character and store it in cell 2
[          ; enter loop
    <      ; move to cell 1
    +      ; increment cell 1
    >      ; move to cell 2
    -      ; decrement cell 2
]          ; we exit the loop when the last cell is empty
<          ; go back to cell 1
----- ; subtract 48 (ie ASCII char code of '0')
.          ; print cell 1
```

EX.

Say we input 3 and 4, the program will read these as ascii values into cells 1 and 2 as their ascii representations 51 and 52.

We then enter the loop which performs the addition and get,  $51 + 52 = 103$ , in cell 1.

Cell 1 then has 48 subtracted from it,  $103 - 48 = 55$ . Which is then printed as the ascii character 7.

Let's Step Through this on an interpreter ([Click Here](#))

# Hello World! In Brainfuck

# Minimized Hello World! in Brainfuck

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

## An explanation

```

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

```

Visualize This ([Click Here](#))

# Brainfuck Interpreters

Brainfuck is an easy language to make an interpreter for, here are a few:

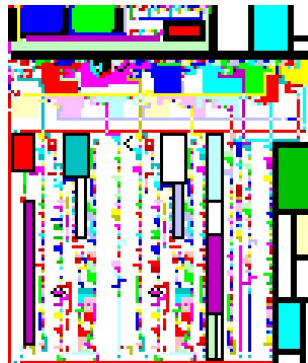
C

```
#include<stdio.h>
int d[9999],a[9999],p,l,i,c;
main(int j,char **k){
    FILE *f=fopen(k[1],"r");
    for(;;(c=fgetc(f))!=EOF;i++){d[i]=c;
        for(i=0;d[i]>0;i++){
            if(d[i]==43)a[p]++;
            if(d[i]==45)a[p]--;
            if(d[i]==62)p++;
            if(d[i]==60)p--;
            if(d[i]==46)putchar(a[p]);
            if(d[i]==44)a[p]=getchar();
            if(d[i]==91&&!a[p])
                for(l=0;i++&&d[i]!=93||l;){
                    if(d[i]==91)l++;
                    if(d[i]==93)l--;
                }
            if(d[i]==93&&a[p])
                for(l=0;i--&&d[i]!=91||l;){
                    if(d[i]==93)l++;
                    if(d[i]==91)l--;
                }
        }
    }
}
```

# Brainfuck

[illegible]

# Piet





# Brainfuck spinoffs

Due to brainfucks popularity, It has gained a number of copycat esoteric languages. They all just replace the 8 symbols with new symbols or words. This family of esolang are called TrivialBrainfuckSubstitution languages ([see here for more](#))

## Ook!

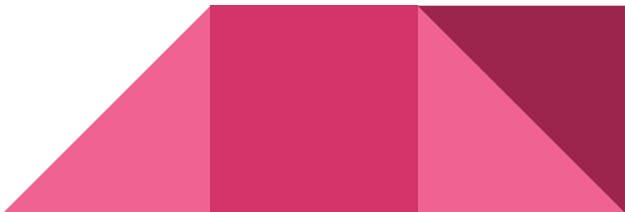
Brainfuck	Ook!
>	Ook. Ook?
<	Ook? Ook.
+	Ook. Ook.
-	Ook! Ook!
.	Ook! Ook.
,	Ook. Ook!
[	Ook! Ook?
]	Ook? Ook!

## Pikalang

Brainfuck	Pikalang
>	pipi
<	pichu
+	pi
-	ka
.	pikachu
,	pikapi
[	pika
]	chu

## Alphuck

Brainfuck	Alphuck
>	a
<	c
+	e
-	i
.	j
,	o
[	p
]	s





Thanks For Coming!  
Stay Around for Noor