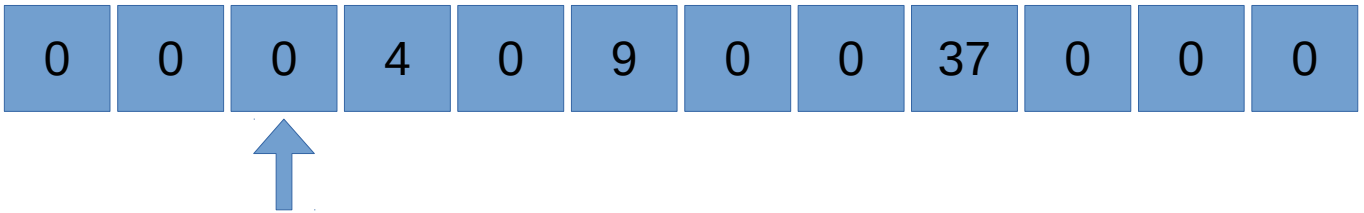
Working Of Compiler

Overview

Brainf**k is a Programming Languages with one of the simplest Instruction Sets.

Imagine there is an infinitely large array of cells, each storing a number, and a pointer which selects one of the cells.



There are only 8 instructions:
1) **>** : Move Pointer to the right
2) **<** : Move Pointer to the left
3) **+** : Increment the selected Cell by 1
4) **-** : Decrement the selected Cell by 1
5) **[** : If selected Cell is zero, then jump right to corresponding ‘]’
6) **]** : If selected Cell is non-zero, then jump left to corresponding ‘[’
7) **.** : Print to Output, the char whose ASCII value is the value in the selected Cell
8) **,** : Into the selected Cell, store the ASCII value of the next char from Input

Overview

The simplicity of the Instruction Set makes it very easy to write Interpreters or Design Hardware to run Brainf**k Code.

But its simplicity also makes in very hard for us to use this language to write programs which perform complex functions.

This was the motivation behind writing a Compiler to Brainf**k.

Project Objective

The Objective of this Project is to Design a Compiler to convert:

High Level, Easy to Read Psudo Code.

Brainf**k

Modular Programming in Brainf**k

Lets say we have 4 lines of code written in Brainf**k.

If the order of execution of these lines is known at compile time, then we could just form a string, which will be the lines concatenated in the correct order, and we have our required program.

How would we go about doing this if the order of execution depends on the on the action of the previously executed lines or the state of the system, and is not known at compile time.

Lets try to understand this using an example.

Modular Programming in Brainf**k

Example 1 :

Write a program to increment Cell 1 by 7 only if Cell 0 is 3, then increment Cell 1 by 3.

You may destroy Cell 0, and/or modify any other Cell if needed.

Soln : Naive :

```
+++                          # Start of Program, Cell 0 may or maynot be 3
>>>>->+<<<<<                 # Set Cell 4 to -1 and Cell 5 to 1
---[> +[->+]- >-<<]          # Set Cell 5 to 0 if Cell 0 was not 3
+[->+] > [-<<<<+++++++>>>>] # If Cell 5 is 1, decrement it and increment cell 1 by 7
<<<<+++>>>>                  # Increment Cell 1 by 3
```

Note : Valid chars in comments will also be run on the Interpreter, so don’t write comments. Don’t copy the comments when testing this code.

Modular Programming in Brainf**k

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

```
0   10   0   0   0   0   0   0
```

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

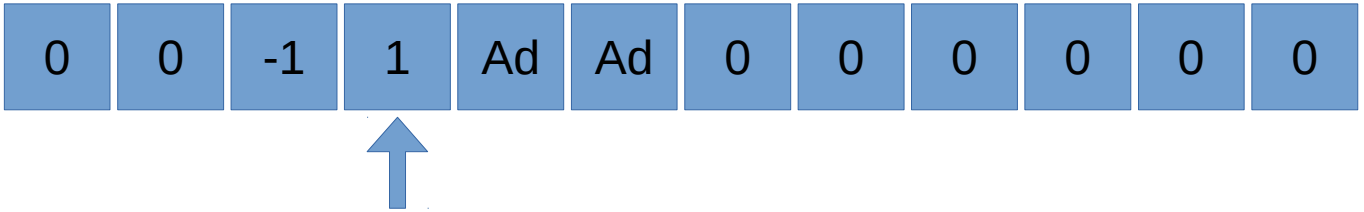
```
4   3   0   0   0   0   0   0
```

First 8 Cells can be seen on right

Modular Programming in Brainf**k

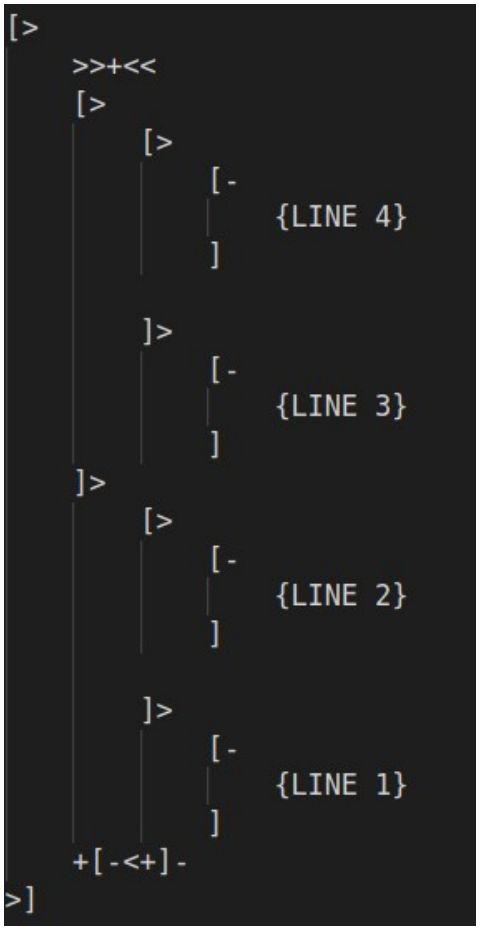
Lets look at a more modular approach.

Lets try to make a system which can have 4 lines of code, and decides which one to run based on a 2 bit Address.



The Brainf**k program shown on the next slide will execute exactly one of the 4 lines based on the Address and return to the currently selected cell. This cycle will repeat till the selected cell is made 0.

Modular Programming in Brainf**k



Now, we can just fill in the Lines correctly to get our desired function.

Starting Address is “0 0”.

Line 1 (0 0): If Cell 0 is not 3, set Address to (1 0), else (0 1).

Line 2 (0 1): Increment Cell 1 by 7, set Address to (1 0).

Line 3 (1 0): Increment Cell 1 by 3, set Address to (1 1).

Line 4 (1 1): End program by making the Cell before the Address 0.

Modular Programming in Brainf**k

```
+++
>>->+

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

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

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

Output (First 8 Cells):

```
0   10   4294967295   0   1   1   0   0
```

Note : Cells store unsigned int, hence -1 is seen as 2^(32) - 1

Modular Programming in Brainf**k

```
+++++++
>>->+

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

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

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

Output (First 8 Cells):

```
4   3   4294967295   0   1   1   0   0
```

Note : Cells store unsigned int, hence -1 is seen as 2^(32) - 1

Modular Programming in Brainf**k

Why is the longer, modular code helpful?

Because now we can add any line to a slot. Which line should be executed next can be decided by the current line.

Levels Of Program

The Program is compiled to Brainf**k through a 2 step reduction process.

The program takes 3 forms as it is converted to brainf**k :
1) User Written High Level Code
2) Sequence of Classic Operations
3) Brainf**k equivalent code of each Operation

Levels Of Program

User Written High Level Code

- An example of the highest level i.e. User Written High Level Code is shown on the right.
- The code Adds 2 numbers.

Source.pc

```
1   func MAIN
2       int X
3       int Y
4       int Sum
5
6       X set 19
7       Y set 10
8       Sum = ADD X Y
9
10      end
11  endfunc
12
13  func ADD
14      int I0
15      int I1
16
17      I0 <- I1
18
19      return I0
20  endfunc
21
```

Levels Of Program

Sequence of Classic Operations

- The High Level Code is first converted into the following Sequence of Operations

```
Function Index : 0
        ['FUNC', 'MAIN']
        ['CLEAR', 0]
        ['MOD', 0, '19']
        ['CLEAR', 1]
        ['MOD', 1, '10']
        ['CALL', 'ADD', 0, 1]
        ['CLEAR', 2]
        ['MOVE', 2, 3, 1]
        ['END']
Function Index : 1
        ['FUNC', 'ADD']
        ['MOVE', 0, 1, 1]
        ['RETURN', 0]
10 Instructions will be written.
```

Levels Of Program

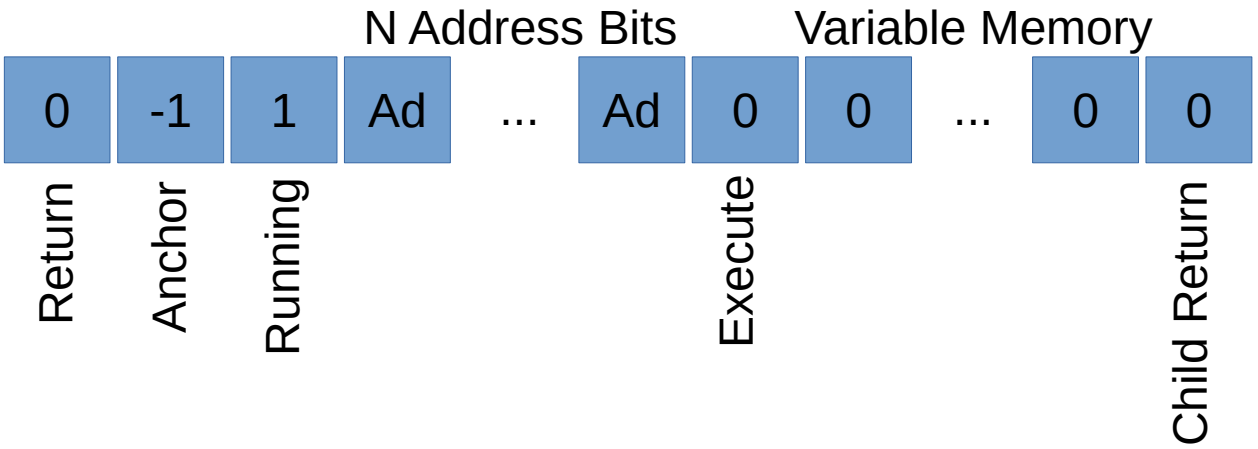
Equivalent of each Operation

- Each of the Classical Operations is converted into a line written in Brainf**k.
- These lines then occupy a slot in the Template.

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

Architecture of Compiled Binary

For every executing function, the memory will be organised as follows:



The role of each of the elements is explained in the following slides.

Architecture of Compiled Binary

- **Running :**

  Value of this Cell signifies if the programming is running or terminated. The outermost ‘[…]’ loop in the compiled source, which starts the search of the Address and navigates to a line, is started by this Cell.

- **Address Bits :**

  The next N Cells store either 0 or 1, and are used to pinpoint which line of instruction needs to be run.

- **Execute :**

  This Cell must be 1 for an insruction line to be executed. When the execution starts, this Cell is set to 0. This is to avoid unexpected execution of other lines after the selected line is run. All instruction lines are designed to leave the pointer to the right, outside the memory of the function, after its task is complete.

Architecture of Compiled Binary

- **Anchor :**

  After the execution of a line, the pointer is brought left to the Anchor using the ‘ +[-<+]- ’ command. The pointer is then moved right and the outermost loop of the source repeats.

- **Return :**

  For a function which returns a Cell, the value to return is stored in this Cell. The memory of the rest of the function is cleared, and the code returns to the parent function as ‘ +[-<+]- ’ will take it to the parent function’s anchor.

- **Variable Memory :**

  These Cells store the contents of local variables of the function.

Architecture of Compiled Binary

- **Child Return :**

  This Cell is part of the Memory of both this process as well as the child functions this function may call. When child functions are called, this cell acts as Return Cell for the child function. New Anchor, Running, Address, etc are set op for the child function and the code passes to the child function. After the child function has returned and its memory cleaned up, the value stored in Cell Child Return is copied onto one of the Cells storing a local variable.

Compilation – Part 1

This section describes the process of how the User Written High Level Code is converted to Sequence of Classic Operations.

First the source is loaded. A string array storing each line of the User Written Source is produced. The ‘\n’ characters is removed from the right ends, and ‘\t’ from the left ends.

Each line is processedone at a time.

The line is split into smaller strings seperated by spaces.

Classical Operations are decided based on the newly formed list of strings.

Compilation – Part 1

The compiler uses 3 python ‘dict’ objects, to create and store the Classic Operations.

FnMap maps function names to index of the declared function.

FnData maps the indices of functions to a list which stores all the Classic Operations generated till now.

VarMap maps the indices of functions to another dict, which maps the names of local variables to the location at which they are stored.

We will use a code sample to see clearly how the Classic Operations are generated.

Compilation – Part 1

This is the sample code we will analyse.

When run we expect 19, 10 and 29 to be stored in consecutive Cells.

The compilation of individual lines are explained in the following slides.

Source.pc

```
1   func MAIN
2       int X
3       int Y
4       int Sum
5
6       X set 19
7       Y set 10
8       Sum = ADD X Y
9
10      end
11  endfunc
12
13  func ADD
14      int I0
15      int I1
16
17      I0 <- I1
18
19      return I0
20  endfunc
21
```

Compilation – Part 1

- “func MAIN”

  The ‘func’ keyword identifies this as a new function decleration. This inserts pair “MAIN” : 0 into FnMap. A new entry is added to FnData and VarMap. 0 : {} is inserted into VarMap. 0 : [[‘FUNC’, ‘MAIN’]].

  [‘FUNC’, ‘MAIN’] is the first Classic Operation generated.

- “int X”, “int Y” and “int Sum”

  These don’t produce Classic Operations but modify VarMap.

  VarMap[0] becomes {‘X’ : 0, ‘Y’ : 1, ‘Sum’ : 2}

- “X set 19”

  This indicates that we wish the variable X to store the value 19.

  This generates 2 Classic Operations, [‘CLEAR’, 0] and [‘MOD’, 0, ‘19’].

Compilation – Part 1

- “Y set 10”

  This indicates that we wish the variable Y to store the value 10.

  This generates 2 Classic Operations, [‘CLEAR’, 1] and [‘MOD’, 1, ‘10’].

- “Sum = ADD X Y”

  This is a function call. We wish to call func ADD with X and Y as inputs and store its return value in Sum.

  This generates the following Classic Operations:

  [‘CALL’, ‘ADD’, 0, 1], [‘CLEAR’, 2], [‘MOVE’, 2, 3, 1]

- “end”

  This signifies that the program must terminate, generating [‘END’].

- “endfunc”

  This just resets a few internal variables, nothing generated.

Compilation – Part 1

- “func ADD”

  This is another function decleration. Pair “ADD” : 0 is inserted into FnMap. VarMap has a new pair inserted, 1 : {}. FnData has a new pair as well 1 : [[‘FUNC’, ‘ADD’]], generating the first Classic Operation for this function.

- “int I0” and “int I1”

  No Classic Operations produced. Inserts pair ‘I0’ : 0 and ‘I1’ : 1 to VarMap[1]

- “I0 <- I1”

  This is a fundamental operator, which decrements I1 and increments I0 till I1 becomes 0. This generates [‘MOVE’, 0, 1, 1].

- “return I0”

  This indicates that this function must send the value in I0 to parent function and return.

Compilation – Part 2

You can see the generated Classic Operations by printing the FnData variable in Compiler.py file before line 531 : “Ps = Assemble()” and after line 530 : “BreakDown()”.

The final step in producing code in pure Brainf**k is to convert each of these lines to an equivalent line in Brainf**k. Then, as discussed earlier, we can place these lines into slots as shown in Slide 9.

```
Function Index : 0
        ['FUNC', 'MAIN']
        ['CLEAR', 0]
        ['MOD', 0, '19']
        ['CLEAR', 1]
        ['MOD', 1, '10']
        ['CALL', 'ADD', 0, 1]
        ['CLEAR', 2]
        ['MOVE', 2, 3, 1]
        ['END']
Function Index : 1
        ['FUNC', 'ADD']
        ['MOVE', 0, 1, 1]
        ['RETURN', 0]
10 Instructions will be written.
```

In the following slides we will see how each of the Classic Operations generated in the previous operations is converted to Brainf**k.

Compilation – Part 2

- [‘FUNC’, ‘MAIN’]

  Till now, zero brainf**k lines have been produced. This Operation does not get a line for itself, but it just causes the compiler to note that the code for a function called ‘MAIN’ begins at line 0.

- [‘CLEAR’, 0]

  This indicates we must empty the variable indexed as 0.

  So we first navigate to the target memory location (>), then empty it ([-]), and return to the Execution Cell (<). Then we change the Address to 1 (<+<<<<<<<<<>>>>>>>>>). Then we move outside the memory of the function (>>>>>), this finction has 3 variables and 1 location reserved for Child Return.

  Line 0 generated is :

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

Compilation – Part 2

- [‘MOD’, 0, ‘19’]

  This increments the variable indexed as 0 by 19.

  So navigate to memory where variable 0 is present (>), increment it by 19 (++++++++++++++++++++), then return to Execution Cell (<). Now change Address to 2 (<-<+<<<<<<<>>>>>>>>), and move outside the memory (>>>>>).

  Line 1 generated is :

  >++++++++++++++++++++<

  <-<+<<<<<<<>>>>>>>>>>>>>

- [‘CLEAR’, 1]

  This indicated that variable indexed as 1 must be emptied. The only difference here with [‘CLEAR’, 0] is that now, to access the target memory, we must move 2 setps right (>>).

Compilation – Part 2

Line 2 generated is :

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

- [‘MOD’, 1, ‘10’]

  Very similar to the previous MOD command, except the increment and memory access is different.

  Line 3 generated is :

  >>++++++++++<<<-<-<+<<<<<<<>>>>>>>>>>>>>>>

- [‘CALL’, ‘ADD’, 0, 1]

  This creats a Child Function ADD. The new Anchor is set up. Address will be set up once ADD is declared. The Address of the Parent is also changed to 5. Here variables indexed 0 and 1 of parent must be passed to Child. These values are copied to index 0 and 1 of Child. The pointer is left outside the Child’s memory.

Compilation – Part 2

Line 4 generated is :

```
 >>>>>->+<<>>>>>>>>>>>>$

<<<<<<<<<<<<<<<<+<<<<<<<>>>>>>>>>

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

<[->+<]>>

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

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

Note : The ‘$’ is where Address of ADD will be placed once it is declared.

Note : Function parameters can be copied from any index of parent function. But they are always copied to index starting from 0 of child function.

Compilation – Part 2

- [‘CLEAR’, 2]

  Just like the previous CLEAR Operations.

  Line 5 generated is :

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

- [‘MOVE’, 2, 3, 1]

  Variable indexed as 3 here is actually the return value of the child function. This is moved to variable indexed as 2. This Operation decrements index 3 by 1 and increments index 2 by 1 till index 3 becomes zero. The last parameter 1, in the Operation indicates that index 2 is incremented once for every decrement of index 3.

  Line 6 generated is:

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

  <<<<<+<<<<<<>>>>>>>>>>>>>

Compilation – Part 2

- [‘END’]

  This just sets the Running Cell to 0, stopping the program.

  Line 7 generated is :

  <<<<<<<<<->>>>>>>>>>>>>>>

- [‘FUNC’, ‘ADD’]

  Marks that function ADD begins at line 8. Also the missing Address of ADD in line 4 is filled.

- [‘MOVE’, 0, 1, 1]

  Just like the previous move Operation.

  Line 8 generated is :

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

Compilation – Part 2

- ['RETURN', 0]

  Moves the content of variable indexed as 0 to the Return Cell. Then Cleans up the memory of the current function.

  Line 9 generated is :

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

  <<<<<<<<<<<<<>+

  >[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]

- This completes the conversion of Operations to equivalent Brainf**k lines. The file GenTemplate.py produces the main backbone into which these lines are inserted.

Compilation

After compilation the intermediate Brainf**k lines are written to file Compiled.txt.

The compiler calls ‘python WriteIns.py’, which loads the template and inserts the generated lines into their slots.

The newly written file Instruction.txt is the compiled Brainf**k code.

This code can be directly run on Brainf**k interpreter.

Notes

There are many other features the User Written Code can have which have not been discussed such as if ... else statements and arrays.

Their usage instructions can be found on the README.md uploaded on the git repository.

Please look at the README.md file to see the rules which must be followed during writing of the User Written Code.