Let's begin.

To code we need to create:

1. Token class: will return the token (integer, 3)
2. Interpreter class
3. interpreter class will take input, get next token
4. expr and its helper function

//Corner cases  
Whitespace error example: 4 4 + 5

Step 1

* **Creating the main function**

Now in the main function, we need to continuously ask for input like calc>  
Isme one thing is ki agar khuch input nhi kiya then again ask for input – continue  
But ab question is how to know ki we have reached finish?  
toh like python let's keep quit as our mechanism to stop the interpreter from running.

* **Next we'll send the input to our interpreter (which will be a class)**

why class? because it has to have a few functions it needs to call and it would be better to have a new object for each input given by the user. Matlab see agar normal functions banate na toh hume harr function defination ke liye arguments pass karne padte, but ek class banane se that work gets easy. As ek object ke jo variables hote voh directly use kar sakte class functions mei.

* **Deciding the structure of interpreter class**

Variables:  text, pos(determines konsa char dekh rahe hum), current token(of Token class)   
Constructor: pos = 0, text with input  
Error function: taaki koi issue ho toh we could just call this error function (rather than giving particular errors)  
Eval function: that will actually evaluate the expression  
Now we know eval function needs 1. next token function 2. eat function (basically a function which accepts a value, i.e ki uski type theek hai na matlab)

* **Now making the logic for eval function**

current token mei phele toh token read karenge (which will be by get next token function)  
will store its value int left int and then call eat to check its type, same will be done for operator and second int  
then do left+right and return the result

* **Now making the Token class**

Ab kyunki hume Tokens se deal karna, ki is the type correct hai ya nhi, ya fir next Token kaise le  
So now for the token class:  
 Variables: value(this will be of auto type as we could have int or char in this), type to store type in string.  
 We will make a constructor, which initializes the value of value and type  
 Now we need to define that if we print the Token, how will it show, like return Token karte ko kya aana chahiye  
 Auto nhi bna due toh some issue with c++, so we decided let's keep it int, it will store the ASCII values  
Also isme humne << ko overload kiya because we wanted if the next token needs to be printed, dhang se print ho jaaye voh

* **Making eat & get next token function**

Toh ab eat mei toh simply we will check the type, if current token type matches with given token type then it is right, and we can go on to have the next token.  
For get next token, phele we check the base condition ki kahi End of Line pe toh nhi aa gaye  
Agar nhi toh we check ki digit hai ya plus sign and uske according we create a Token object and return it to eval/eat function, otherwise error

**Next steps:**

* **Prev code mei changes: Value will now be string**

Kyunki ascii rakhke ke multiple digits vale number na handle ho paate toh iss liye string hi rakh rahe

toh ab value1 and value2 calculate karte hue stoi use kiya to convert from string to integer

                  But ek major challenge faced was:

                 Ki char won't directly pass to a string, so we decided to use the to\_string method, but it seems ki usme when we convert char using to\_string to error aati

Isliye fir brute force try kiya we made a new string called current\_char\_str which stores the char as string(Also isme a new thing we observed, if char needs to be assigned to a string::: declaration and defination should be different)

* **Extending the code for Minus**

This will work in the same way as + sign, so we first add MINUS as our global string (Token type basically)

Then in the eval function we will add if else statement based on the operation type and send vahi same type to the eat function to verify

                   And similarly based on operator type corresponding result return karenge

**Note:** Abhi \* / has not been added because precedence vaali cheez check karni padegi usme

* **Handling whitespace in our input**

Ab iske liye yeh hai main ki start mei space ho sakti, mid mei and end mei

                4  +5 is valid but 2  2 + 3 won't be

So ab khuch aisa hona chahiye ki jaise jaise space aaye na pos ko increment karte raho

But ab jaise jaise pos ko increment karenge toh current char change hoga and humne yeh bhi dekhn hoga ki end of input toh nhi aa gaye

So uske liye phele ek function banate advance, which will incorporate this boundary checking as well as assigning current char

Also!! isse ab hume current char Interpreter class ka variable banana padega (constructor mei it will be initialised as text[pos] i.e text[0]

* **Skip whitespace function**

Ab advance function is done, so we could have a function jisme if current char is space it just advances to next char

                    Now, with this done, how to add this in tokenization?

toh how does the functioning goes, phele hum current token mei get next token se first char leke aate, now suppose if it is a whitespace toh get next token tab tak chalna chahiye jab tak non whitespace ya end ni aata

Okay so we could put a while loop for this, isse starting and ending vaali whitespace is handled

Now what about middle whitespace, agar mid mei bhi aayi toh skip kardega yeh and check karega if the pattern needed matches or not! So haan isse whitespace sorted  
Bus remember ki current char == NULL should be checked after the whitespace loop, because of the whitespaces at the end then our current char will become NULL

* ​**Now we need to multi digit numbers**

For this we could think on similar lines as whitespace vala

      Jab tak we have a number keep on going to next char, and store  karte raho previous vaalo ko

Now, as hume pta humne string pass karni token ko ,toh  to make our work easier isse hum string rakhenge and concatenate karte rahenge char ko

So alag se function banate iske liye which will have a while loop and go on till we encounter a digit as our current char and keep on calling advance within it.

So, this completes possible combinations for addition!

**Note:**-We are still NOT handling float or double, just int

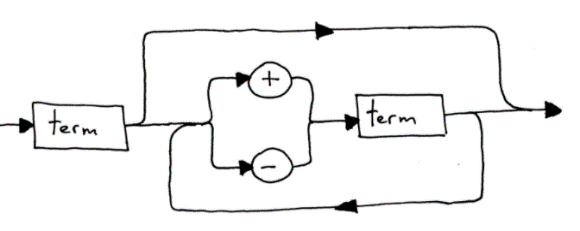
-The pattern given is still INTEGER OPERATOR INTEGER, so 5+9 -1 won't work yet.

**Next Steps:**

**New Thing learned:** Recognizing a pattern/phrase in a stream of tokens is PARSING. So till now, in our eval function we were doing both Lexical analysis (using get next token) and parsing (by giving the set pattern of INT->OP->INT)

Now, in order to extend our functionality of having multiple + or - signs, we will have to change our parsing!

For this, we will take the help of Syntax Diagram or FSA like visualization to understand what all expressions are valid.



Now, this will be accepting expressions like 3, 3+4-2 etc but not 3+ or +32

This raises an important point:: **We need to deal with Unary operators as well.**

**Grammar accepted till now:**

3

            3+4

            3-4

            3+4-5+6

So now for this, our parsing function should be something like:

                Accept Integer (a term)

                            |

                while loop ( till do not reach the end)

                            |

                check + or -

                            |

                check for an integer (a term)

                            |

                while loop ends

**Implementation:**

* **Starting with the implementation of the term function**

- We first store the current token in token object because if it matches the correct type we will have to return the value of that token (because eat will advance to the next token, so storing the previous token is important)

                        - Then we call the eat function to verify the type and if correct we return the token value(string)

* **Now we create a result variable which will have our evaluated result**

- So we create the same functionality as in the flowchart above

- Here an important thing is that while loop will terminate when current token comes to end and not current char (observed by dry running on the example 3+)

**Now, the next step would be to decide the grammar (Done in another document)**

**Also, we separate the Lexer and Parser(Interpreter class till now)**

**Next Steps:**

Now, we have to convert our first grammar into code.

                    expr:  term (( PLUS | MINUS) term)\*                  Example:

                    term: factor (( MUL | DIV) factor)\*                   17+8\*2-10/5

                    factor: INTEGER

So, following the rules for converting the grammar to the code, we convert the above grammar.

First of all, we convert the factor rule. Done.

Next, we make a function for term. and then for expr (this will replace the eval function now)

Now, after converting this grammar to code, our interpreter correctly evaluates the operators consisting of integers with multiple operators like +,-,\*,/ according to there precedence and associativity.

**Now moving on to the next grammar.**

expr:  term (( PLUS | MINUS) term)\*                  Example:

            term: factor (( MUL | DIV) factor)\*                   (17+8)\*2-10/5

            factor: INTEGER | LPAREN expr RPAREN

Now this grammar was added as a basic unit. WHY? We could have added it as another level also (following rules of precedence), but here you need to understand that () is not an operator, and precedence is defined for operators only.

Also, in maths when we consider ( ) it is because of how rules are defined and not the precedence order. Thus, these grammars rules are defined for operators and thus, this will be considered as basic unit of the grammar.

Also, in this we do not have \* for it as, (expr) (expr) won't make any sense and thus only the recursion expression will be helpful.

**Converting the above grammar to code:**

There will be 2 major changes:

            -    New token will be added in token type and functionality for the same will be added in the get next token function

            -    Factor function will be updated.

As we have OR in the factor rule, that means it will be converted into if else condition. **Also, remember one if condition will be for INTEGER while other will be just for LPAREN as any expression cannot begin with RPAREN**

**Now, we thought if we could add the exponential functionality too.** So, first of all we thought about it's grammar and as it has the highest precedence so it's production rule will be added just before the base expression.

So, the probable grammar decided was:

expr:  term (( PLUS | MINUS) term)\*

            term: factor (( MUL | DIV) factor)\*

    factor: atom (POW atom)\*

            atom: INTEGER | LPAREN expr RPAREN

So, we implemented this grammar the same way we did expr and term using the C++ inbuilt pow function. But we noticed that (4+1)^2 was giving the answer as 24. After exploration we figured out it was due to typecasting conversion from double to int (https://www.geeksforgeeks.org/power-function-cc/). So we added 0.5 to the answer as mentioned in the article and it solved the wrong answer problem.

But now as we were testing it for a few more programs we saw that it gave the wrong answer for statements like 2^3^2. It evaluated it from left to right rather than right to left. So we did a dry run for the example and observed that the grammar wasn't correct. As this needs to be evaluated from right to left, so we have to calculate the rightmost term first and that is why the grammar will be **factor: atom (POW factor)\***

And now, our exponent function works fully correct, and with this we implement the most common integer arithmetic operations.

**Now, completing this we have built RECURSIVE INTERPRETER.**