

INFO3220 Tutorial 10/Week 11

Compulsory lab work

Your task is to read in a line from standard input that consists of integers and basic mathematical operators (plus, minus, times, divide), and evaluate that line.

Sample input

3 + 40 * -2 / 5 - 16 / 4

Sample output

-17

Details

You will need to:

- write a recursive descent parser; and
- write a tokeniser that splits the input into integers and operators, and remembers the position where the parser has processed up until; and
- design a hierarchy of node classes in the composite design pattern that will be used by your parser.

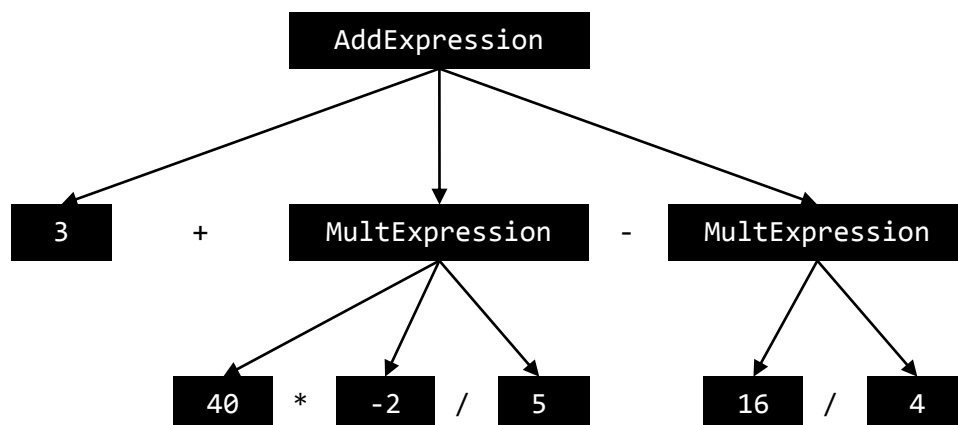
The recursive descent parser should follow the following grammar (so that it respects the mathematical order of operations):

`<AddExpression> := <MultExpression> {(+|-) <MultExpression>}`

`<MultExpression> := <Integer> {(*|/) <Integer>}`

`<Integer> := an integer`

For example, the sample input should parse into:



Your parser class *might* look like this:

```
class Parser
{
private:
    // returns a node for the AddExpression at tok's current position;
    // afterwards, tok is at one after the end of the AddExpression
    Node* get_add_expression(Tokenizer& tok);

    // returns a node for the MultExpression at tok's current position
    Node* get_mult_expression(Tokenizer& tok);

    // returns a node for the Integer at tok's current position
    Node* get_integer(Tokenizer& tok);

public:
    // returns the root of the parse tree
    Node* parse(Tokenizer& tok);
};
```

Recall that with a recursive descent parser, each rule in the grammar is usually implemented with a separate function. For example, `get_mult_expression` corresponds to the `<MultExpression>` rule. If the tokeniser is up to the third token in the sample input above (the integer 40), `get_mult_expression` will return a node that represents $40 * -2 / 5$, and afterwards, the tokeniser will be up to the minus sign after the 5.

After the parse tree has been constructed, you should then be able to compute the value of the input line.

You may assume that there are no errors in the input line and that there is at least one whitespace character between each integer and operator.

Your program must manage memory correctly; that is, it must have no memory leaks. (Consider using the `boost` library pointer classes.)

Optional extension

Note that the grammar above is not actually recursive. Design a grammar that can handle parentheses:

$3 + 40 * (3 + (36 / 6) - 1)$

Modify the parser and create new node classes (if necessary) to handle your new grammar.

Also, ensure that your program handles divide by zero errors and integer overflow and underflow elegantly.