UFPE – CENTRO DE INFORMÁTICA
ESTRUTURAS DE DADOS ORIENTADAS A OBJETOS
PROJETO

(Gustavo Carvalho – ghpc@cin.ufpe.br)

Desenvolva, individualmente e sem uso de IA generativa, um programa em C++ que leia expressões (aritméticas e booleanas) da entrada padrão e imprima na saída padrão o resultado da avaliação da expressão lida.

Requisitos Gerais

O seu código deve seguir um paradigma de programação orientado a objetos (OO). Em linhas gerais, não devem existir variáveis globais, nem funções (exceto a `main`), mas somente classes e métodos. Também se deve evitar o uso de `friend`. Desvios às essas recomendações devem ser documentados e justificados no código.

Estruture o seu código em arquivos `.h` e `.cpp` conforme boas práticas. Explore a maior quantidade possível de conceitos de OO suportados em C++, por exemplo, mas não limitado a: classes, métodos, modificadores de acesso (`public`, `private` e `protected`), namespaces, ponteiros para objetos, referências para objetos, construtores, construtores cópia, destrutores, métodos inline, métodos de acesso, passagem por valor, passagem por referência, atributos e métodos estáticos, sobrecarga de operadores (aritméticos, booleanos, de atribuição, etc.) , alocação dinâmica de objetos, herança, sobrescrita de métodos, conversão estática e dinâmica de tipos entre classes, métodos virtuais, classes abstratas, tratamento de exceção e tipos parametrizados (`template`).

Entrada

A entrada consiste de vários casos. A primeira linha contém c ($1 \leq c \leq 100.000$), o número de casos. A seguir, tem-se c casos, um por linha. Cada caso consiste em uma expressão aritmética ou booleana. Todas as expressões serão sintaticamente corretas.

```
7
1
2 + 3 * 2
( 2 - - -3 ) * 2
3 / 2
true || false == false
( true || false ) == false
true + 3
```

Formato e Processamento de Expressões

O formato de cada expressão segue a especificação dada pela Gramática Livre de Contexto (GLC) abaixo, onde o símbolo inicial é o não-terminal `exp`. O uso de colchetes `[ A ]` denota

que o símbolo `A` é opcional. As aspas duplas destacam os terminais da linguagem que estarão sempre delimitados por espaço. O símbolo `<literal>` representa literais inteiros (e.g., `2`, `-3`, etc.), assim como literais booleanos (i.e., `true` e `false`).

```
<exp>         ::= <or_exp>
<or_exp>      ::= <and_exp> [ "||" <and_exp> ]
<and_exp>     ::= <eq_exp> [ "&&" <eq_exp> ]
<eq_exp>      ::= <rel_exp> [ "==" <rel_exp> | "!=" <rel_exp> ]
<rel_exp>      ::= <add_exp> [ "<" <add_exp> | ">" <add_exp> | "<=" <add_exp> | ">=" <add_exp>
]
<add_exp>     ::= <mul_exp> [ "+" <mul_exp> | "-" <mul_exp> ]
<mul_exp>     ::= <unary_exp> [ "*" <unary_exp> | "/" <unary_exp> ]
<unary_exp>   ::= "-" <primary_exp> | <primary_exp>
<primary_exp> ::= <literal> | "(" <exp> ")"
<literal>     ::= integer literals | boolean literals
```

O processamento de expressões de acordo com a gramática acima pode ser realizado a partir do algoritmo *recursive descent parser*. Nesse algoritmo, cria-se uma operação para cada não-terminal, cujo corpo reflete diretamente a regra correspondente na gramática. Por exemplo, veja o pseudocódigo abaixo associado ao processamento dos símbolos `<exp>` e `<or_exp>`.

```
Expression parse_exp() {
	return parse_or_exp();
}

Expression parse_or_exp() {
	Expression e1;
	e1 = parse_and_exp();
	if (current_token == "||") {
		Operand op = new Operand(current_token);
		read_next_token();
		Expression e2;
		e2 = parse_and_exp();
		return new BinaryExpression(e1, op, e2);
	} else {
		return e1;
	}
}
```

Saída

Imprima o resultado da avaliação da expressão. Se um erro for encontrado durante a avaliação da expressão, imprima "`error`" (sem as aspas duplas).

```
1
8
-2
1
true
false
error
```

Entregável

Cada estudante deve enviar pelo Classroom: (1) todo o código produzido (arquivo `.zip`), além de (2) uma descrição clara de como o projeto é compilado e executado.