

2025_2 - COMPILADORES - METATURMA

PAINEL > **MINHAS TURMAS** > **2025_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**
> **AV2 - REPRESENTAÇÃO DE EXPRESSÕES ARITMÉTICAS**

Descrição

Visualizar envios

AV2 - Representação de Expressões Aritméticas

Data de entrega: domingo, 31 Ago 2025, 23:59

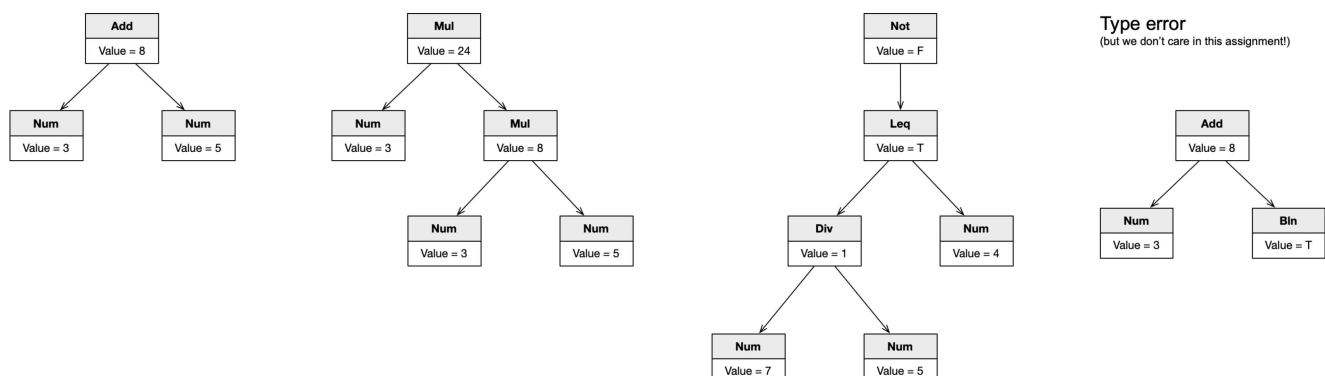
Arquivos requeridos: driver.py, Expression.py ([Baixar](#))

Tamanho máximo de arquivo carregado: 448 KiB

Tipo de trabalho: Trabalho individual

A fim de ser resolvida, uma expressão aritmética como "2 + 3 * 4" precisa ser representada como uma estrutura de dados. A maneira mais natural de fazer isso é via uma árvore. A árvore nos permite capturar a natureza recursiva de uma expressão. Por natureza recursiva, dizemos que uma expressão pode ser composta por sub-expressões. Assim, alguns nós desta árvore são unários, outros binários. Nós unários representam expressões que possuem somente uma subexpressão, como por exemplo, -2 ou Not(True). Nós binários, por sua vez, representam expressões formadas por duas subexpressões, como 2 + 3, ou 4 * (5 - 2).

Esta representação nos permite interpretar uma expressão. Em uma linguagem de expressões aritméticas, a interpretação de uma expressão é o valor dela. Por exemplo, se uma expressão for um número, então o valor da expressão é este número. Se uma expressão for uma soma, então o valor dela é a soma do valor de cada sub-expressão. A figura abaixo ilustra algumas expressões e seus valores.



Neste exercício você deverá implementar a representação de expressões aritméticas. Estaremos seguindo um padrão de projetos chamado "[composite](#)". Segundo este padrão, qualquer nós que representa expressões possui a mesma interface. Neste caso, a interface é um método `eval()`, que retorna o valor da expressão. Em outras palavras, expressões são instâncias de subclasses da seguinte classe abstrata:

```
class Expression(ABC):
    @abstractmethod
    def eval(self):
        raise NotImplementedError
```

O exercício define uma hierarquia de expressões para você. Neste caso, estamos seguindo a especificação da linguagem de programação [Cool](#). Iremos trabalhar com as seguintes expressões daquela linguagem:

```
expr ::= expr + expr
      | expr - expr
      | expr * expr
      | expr / expr
      | expr < expr
      | expr <= expr
      | expr = expr
      | not expr
      | (expr)
```

```
| integer  
| true  
| false
```

Para completar o VPL, você deverá implementar o método `eval()` de cada classe concreta de expressões. Você deverá submeter dois arquivos: `Expression.py` e `driver.py`. Porém, você não deve alterar `driver.py`. Ele está disponível para que você possa testar seu exercício localmente. Para tanto, você pode usar o comando abaixo:

```
$> python3 driver.py
```

```
Add(Num(3), Num(5)) # isso foi o que eu digitei, e então apertei CTRL+D para passar o fim de arquivo  
ao driver.
```

```
Value is 8
```

A implementação de `Expression.py` possui vários comentários doctest, que testam sua implementação. Caso queira testar seu código, simplesmente faça:

```
$> python3 -m doctest Expression.py
```

Caso você não gere mensagens de erro, então seu trabalho está completo!

Arquivos requeridos

driver.py

```
1 import sys  
2 from Expression import *  
3  
4 if __name__ == "__main__":  
5     """  
6     Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o  
7     VPL. O arquivo contem o codigo que testa a implementacao do analisador  
8     lexico.  
9     """  
10    e = eval(sys.stdin.read())  
11    print(f"Value is {e.eval()}")
```

Expression.py

```
1 from abc import ABC, abstractmethod
2
3 class Expression(ABC):
4     @abstractmethod
5     def eval(self):
6         raise NotImplementedError
7
8 class Bln(Expression):
9     """
10    This class represents expressions that are boolean values. There are only
11    two boolean values: true and false. The evaluation of such an expression is
12    the boolean itself.
13    """
14    def __init__(self, bln):
15        self.bln = bln
16    def eval(self):
17        """
18        Example:
19        >>> e = Bln(True)
20        >>> e.eval()
21        True
22        """
23        # TODO: Implement this method!
24        return None
25
26 class Num(Expression):
27     """
28    This class represents expressions that are numbers. The evaluation of such
29    an expression is the number itself.
30    """
31    def __init__(self, num):
32        self.num = num
33    def eval(self):
34        """
35        Example:
36        >>> e = Num(3)
37        >>> e.eval()
38        3
39        """
40        # TODO: Implement this method!
41        return None
42
43 class BinaryExpression(Expression):
44     """
45    This class represents binary expressions. A binary expression has two
46    sub-expressions: the left operand and the right operand.
47    """
48    def __init__(self, left, right):
49        self.left = left
50        self.right = right
51
52    @abstractmethod
53    def eval(self):
54        raise NotImplementedError
55
56 class Eql(BinaryExpression):
57     """
58    This class represents the equality between two expressions. The evaluation
59    of such an expression is True if the subexpressions are the same, or false
60    otherwise.
61    """
62    def eval(self):
63        """
64        Example:
65        >>> n1 = Num(3)
66        >>> n2 = Num(4)
67        >>> e = Eql(n1, n2)
68        >>> e.eval()
69        False
70
71        >>> n1 = Num(3)
72        >>> n2 = Num(3)
73        >>> e = Eql(n1, n2)
74        >>> e.eval()
75        True
76        """
77        # TODO: Implement this method!
78        return None
79
80 class Add(BinaryExpression):
81     """
82    This class represents addition of two expressions. The evaluation of such
83    an expression is the addition of the two subexpression's values.
84    """
85    def eval(self):
86        """
87        Example:
88        >>> n1 = Num(3)
89        >>> n2 = Num(4)
90        >>> e = Add(n1, n2)
91        >>> e.eval()
92        7
93        """
94        # TODO: Implement this method!
95        return None
96
97 class Sub(BinaryExpression):
98     """
99    This class represents subtraction of two expressions. The evaluation of such
100    an expression is the subtraction of the two subexpression's values.
101    """
102    def eval(self):
103        """
```

```
104         Example:
105         >>> n1 = Num(3)
106         >>> n2 = Num(4)
107         >>> e = Sub(n1, n2)
108         >>> e.eval()
109         -1
110         """
111         # TODO: Implement this method!
112         return None
113
114     class Mul(BinaryExpression):
115         """
116         This class represents multiplication of two expressions. The evaluation of
117         such an expression is the product of the two subexpression's values.
118         """
119         def eval(self):
120             """
121             Example:
122             >>> n1 = Num(3)
123             >>> n2 = Num(4)
124             >>> e = Mul(n1, n2)
125             >>> e.eval()
126             12
127             """
128             # TODO: Implement this method!
129             return None
130
131     class Div(BinaryExpression):
132         """
133         This class represents the integer division of two expressions. The
134         evaluation of such an expression is the integer quotient of the two
135         subexpression's values.
136         """
137         def eval(self):
138             """
139             Example:
140             >>> n1 = Num(28)
141             >>> n2 = Num(4)
142             >>> e = Div(n1, n2)
143             >>> e.eval()
144             7
145             >>> n1 = Num(22)
146             >>> n2 = Num(4)
147             >>> e = Div(n1, n2)
148             >>> e.eval()
149             5
150             """
151             # TODO: Implement this method!
152             return None
153
154     class Leq(BinaryExpression):
155         """
156         This class represents comparison of two expressions using the
157         less-than-or-equal comparator. The evaluation of such an expression is a
158         boolean value that is true if the left operand is less than or equal the
159         right operand. It is false otherwise.
160         """
161         def eval(self):
162             """
163             Example:
164             >>> n1 = Num(3)
165             >>> n2 = Num(4)
166             >>> e = Leq(n1, n2)
167             >>> e.eval()
168             True
169             >>> n1 = Num(3)
170             >>> n2 = Num(3)
171             >>> e = Leq(n1, n2)
172             >>> e.eval()
173             True
174             >>> n1 = Num(4)
175             >>> n2 = Num(3)
176             >>> e = Leq(n1, n2)
177             >>> e.eval()
178             False
179             """
180             # TODO: Implement this method!
181             return None
182
183     class Lth(BinaryExpression):
184         """
185         This class represents comparison of two expressions using the
186         less-than comparison operator. The evaluation of such an expression is a
187         boolean value that is true if the left operand is less than the right
188         operand. It is false otherwise.
189         """
190         def eval(self):
191             """
192             Example:
193             >>> n1 = Num(3)
194             >>> n2 = Num(4)
195             >>> e = Lth(n1, n2)
196             >>> e.eval()
197             True
198             >>> n1 = Num(3)
199             >>> n2 = Num(3)
200             >>> e = Lth(n1, n2)
201             >>> e.eval()
202             False
203             >>> n1 = Num(4)
204             >>> n2 = Num(3)
205             >>> e = Lth(n1, n2)
206             >>> e.eval()
207             False
208             """
209             # TODO: Implement this method!
210             return None
```

```
207         return False
208     """
209     # TODO: Implement this method!
210     return None
211
212 class UnaryExpression(Expression):
213     """
214     This class represents unary expressions. A unary expression has only one
215     sub-expression.
216     """
217     def __init__(self, exp):
218         self.exp = exp
219
220     @abstractmethod
221     def eval(self):
222         raise NotImplementedError
223
224 class Neg(UnaryExpression):
225     """
226     This expression represents the additive inverse of a number. The additive
227     inverse of a number n is the number -n, so that the sum of both is zero.
228     """
229     def eval(self):
230         """
231         Example:
232         >>> n = Num(3)
233         >>> e = Neg(n)
234         >>> e.eval()
235         -3
236         >>> n = Num(0)
237         >>> e = Neg(n)
238         >>> e.eval()
239         0
240         """
241         # TODO: Implement this method!
242         return None
243
244 class Not(UnaryExpression):
245     """
246     This expression represents the negation of a boolean. The negation of a
247     boolean expression is the logical complement of that expression.
248     """
249     def eval(self):
250         """
251         Example:
252         >>> t = Bln(True)
253         >>> e = Not(t)
254         >>> e.eval()
255         False
256         >>> t = Bln(False)
257         >>> e = Not(t)
258         >>> e.eval()
259         True
260         """
261         # TODO: Implement this method!
262         return None
```

[VPL](#)