

## 2025\_2 - COMPILADORES - METATURMA

**PAINEL** > **MINHAS TURMAS** > **2025\_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**

> **AV3 - PARSING DE EXPRESSÕES ARITMÉTICAS**

Descrição

Visualizar envios

### AV3 - Parsing de expressões aritméticas

**Data de entrega:** sexta, 5 Set 2025, 23:59

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

**Tamanho máximo de arquivo carregado:** 128 KiB

**Tipo de trabalho:** Trabalho individual

O objetivo deste exercício é fazer o parsing de expressões aritméticas. Neste exercício usamos um subconjunto das expressões aritméticas da linguagem Cool. A Figura 1 contém uma gramática que reconhece essas expressões. Para resolver o exercício, você deverá escrever um programa que converta texto como "2 + 3 \* 4" em árvores construídas com as classes de expressões vistas no exercício anterior. Você pode usar qualquer técnica de parsing que achar mais conveniente. Contudo, seu parser deverá obedecer as regras de precedência da linguagem Cool. Essas regras estão disponíveis na página 17 do [manual](#). De todo modo, a Figura 2 resume as diferentes precedências entre os operadores.

```
exp ::= exp + exp
      | exp - exp
      | exp * exp
      | exp / exp
      | exp ≤ exp
      | exp < exp
      | exp = exp
      | not exp
      | ~ exp
      | ( exp )
      | true
      | false
      | integer
```

Figura 1: Gramática de expressões aritméticas

Nível de precedência	Operador lógico/aritmético
1	~
2	*
3	+
4	<= < =
5	not

Figura 2: precedência dos operadores

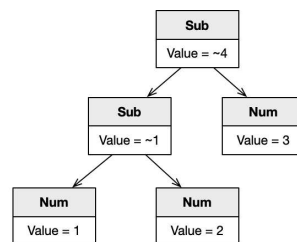


Figura 3: AST produzida para 1 - 2 - 3

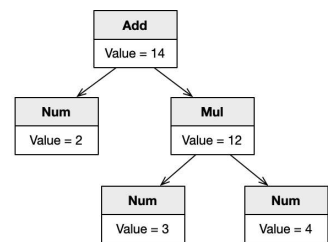


Figura 4: AST produzida para 2 + 3 \* 4

Conforme dito antes, há várias formas de escanear expressões aritméticas. Caso você opte por implementar um [analisador sintático descendente recursivo](#), veja que em [seu artigo](#), Theodore Norvell descreve três diferentes estratégias para resolver o problema da associatividade à esquerda dos operadores. Independente da forma que você escolher para implementar seu analisador sintático, ele deverá produzir uma árvore como aquela vista na Figura 3, quando alimentado com a expressão 1 - 2 - 3. Em outras palavras, o valor desta expressão deve ser -4, e não 2, caso o sinal de subtração fosse considerado associativo à direita.

Para completar o VPL, você deverá implementar o método `parse()` da classe `Parser` em `Parser.py`. Contudo, este exercício depende também dos dois exercícios anteriores. Então, você deverá também ter as implementações de `Lexer.py` e `Expression.py`. Você deverá submeter quatro arquivos: `driver.py`, `Expression.py`, `Lexer.py` e `Parser.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
2 + 3 * 4 # isso foi o que eu digitei, e então apertei CTRL+D para passar o fim de arquivo ao driver.
Value is 14
```

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

```
python3 -m doctest Parser.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 from Lexer import Lexer
4 from Parser import Parser
5
6 if __name__ == "__main__":
7     """
8     Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
9     VPL. O arquivo contem o codigo que testa a implementacao do parser.
10    """
11    lexer = Lexer(sys.stdin.read())
12    parser = Parser(lexer.tokens())
13    exp = parser.parse()
14    print(f"Value is {exp.eval()}")

```

## Lexer.py

```

1 import sys
2 import enum
3
4
5 class Token:
6     """
7     This class contains the definition of Tokens. A token has two fields: its
8     text and its kind. The "kind" of a token is a constant that identifies it
9     uniquely. See the TokenType to know the possible identifiers (if you want).
10    You don't need to change this class.
11    """
12    def __init__(self, tokenText, tokenKind):
13        # The token's actual text. Used for identifiers, strings, and numbers.
14        self.text = tokenText
15        # The TokenType that this token is classified as.
16        self.kind = tokenKind
17
18
19 class TokenType(enum.Enum):
20     """
21     These are the possible tokens. You don't need to change this class at all.
22     """
23     EOF = -1 # End of file
24     NLN = 0 # New line
25     WSP = 1 # White Space
26     COM = 2 # Comment
27     NUM = 3 # Number (integers)
28     STR = 4 # Strings
29     TRU = 5 # The constant true
30     FLS = 6 # The constant false
31     EQL = 201
32     ADD = 202
33     SUB = 203
34     MUL = 204
35     DIV = 205
36     LEQ = 206
37     LTH = 207
38     NEG = 208
39     NOT = 209
40     LPR = 210
41     RPR = 211
42
43
44 class Lexer:
45
46     def __init__(self, source):
47         """
48         The constructor of the lexer. It receives the string that shall be
49         scanned.
50         TODO: You will need to implement this method.
51         """
52         pass
53
54     def tokens(self):
55         """
56         This method is a token generator: it converts the string encapsulated
57         into this object into a sequence of Tokens. Examples:
58
59         >>> l = Lexer('1 * 2 - 3')
60         >>> [tk.kind for tk in l.tokens()]
61         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
62
63         >>> l = Lexer('1 * 2 -- 3\\n')
64         >>> [tk.kind for tk in l.tokens()]
65         [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
66         """
67         token = self.getToken()
68         while token.kind != TokenType.EOF:
69             if token.kind != TokenType.WSP and token.kind != TokenType.COM:
70                 yield token
71             token = self.getToken()
72
73     def getToken(self):
74         """
75         Return the next token.
76         TODO: Implement this method!
77         """
78         token = None
79         return token

```

## 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

```

```
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
```

Parser.py

```

1  import sys
2
3  from Expression import *
4  from Lexer import Token, TokenType
5
6  """
7  This file implements the parser of arithmetic expressions.
8
9  References:
10 see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
11 """
12
13 class Parser:
14     def __init__(self, tokens):
15         """
16         Initializes the parser. The parser keeps track of the list of tokens
17         and the current token. For instance:
18         """
19         self.tokens = list(tokens)
20         self.cur_token_idx = 0 # This is just a suggestion!
21
22     def parse(self):
23         """
24         Returns the expression associated with the stream of tokens.
25
26         Examples:
27         >>> parser = Parser([Token('123', TokenType.NUM)])
28         >>> exp = parser.parse()
29         >>> exp.eval()
30         123
31
32         >>> parser = Parser([Token('True', TokenType.TRU)])
33         >>> exp = parser.parse()
34         >>> exp.eval()
35         True
36
37         >>> parser = Parser([Token('False', TokenType.FLS)])
38         >>> exp = parser.parse()
39         >>> exp.eval()
40         False
41
42         >>> tk0 = Token('~', TokenType.NEG)
43         >>> tk1 = Token('123', TokenType.NUM)
44         >>> parser = Parser([tk0, tk1])
45         >>> exp = parser.parse()
46         >>> exp.eval()
47         -123
48
49         >>> tk0 = Token('3', TokenType.NUM)
50         >>> tk1 = Token('*', TokenType.MUL)
51         >>> tk2 = Token('4', TokenType.NUM)
52         >>> parser = Parser([tk0, tk1, tk2])
53         >>> exp = parser.parse()
54         >>> exp.eval()
55         12
56
57         >>> tk0 = Token('3', TokenType.NUM)
58         >>> tk1 = Token('*', TokenType.MUL)
59         >>> tk2 = Token('~', TokenType.NEG)
60         >>> tk3 = Token('4', TokenType.NUM)
61         >>> parser = Parser([tk0, tk1, tk2, tk3])
62         >>> exp = parser.parse()
63         >>> exp.eval()
64         -12
65
66         >>> tk0 = Token('30', TokenType.NUM)
67         >>> tk1 = Token('/', TokenType.DIV)
68         >>> tk2 = Token('4', TokenType.NUM)
69         >>> parser = Parser([tk0, tk1, tk2])
70         >>> exp = parser.parse()
71         >>> exp.eval()
72         7
73
74         >>> tk0 = Token('3', TokenType.NUM)
75         >>> tk1 = Token('+', TokenType.ADD)
76         >>> tk2 = Token('4', TokenType.NUM)
77         >>> parser = Parser([tk0, tk1, tk2])
78         >>> exp = parser.parse()
79         >>> exp.eval()
80         7
81
82         >>> tk0 = Token('30', TokenType.NUM)
83         >>> tk1 = Token('-', TokenType.SUB)
84         >>> tk2 = Token('4', TokenType.NUM)
85         >>> parser = Parser([tk0, tk1, tk2])
86         >>> exp = parser.parse()
87         >>> exp.eval()
88         26
89
90         >>> tk0 = Token('2', TokenType.NUM)
91         >>> tk1 = Token('*', TokenType.MUL)
92         >>> tk2 = Token('(', TokenType.LPR)
93         >>> tk3 = Token('3', TokenType.NUM)
94         >>> tk4 = Token('+', TokenType.ADD)
95         >>> tk5 = Token('4', TokenType.NUM)
96         >>> tk6 = Token(')', TokenType.RPR)
97         >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
98         >>> exp = parser.parse()
99         >>> exp.eval()
100        14
101
102        >>> tk0 = Token('4', TokenType.NUM)
103        >>> tk1 = Token('==', TokenType.EQL)

```

```
104     >>> tk2 = Token('4', TokenType.NUM)
105     >>> parser = Parser([tk0, tk1, tk2])
106     >>> exp = parser.parse()
107     >>> exp.eval()
108     True
109
110     >>> tk0 = Token('4', TokenType.NUM)
111     >>> tk1 = Token('<=', TokenType.LEQ)
112     >>> tk2 = Token('4', TokenType.NUM)
113     >>> parser = Parser([tk0, tk1, tk2])
114     >>> exp = parser.parse()
115     >>> exp.eval()
116     True
117
118     >>> tk0 = Token('4', TokenType.NUM)
119     >>> tk1 = Token('<', TokenType.LTH)
120     >>> tk2 = Token('4', TokenType.NUM)
121     >>> parser = Parser([tk0, tk1, tk2])
122     >>> exp = parser.parse()
123     >>> exp.eval()
124     False
125
126     >>> tk0 = Token('not', TokenType.NOT)
127     >>> tk1 = Token('4', TokenType.NUM)
128     >>> tk2 = Token('<', TokenType.LTH)
129     >>> tk3 = Token('4', TokenType.NUM)
130     >>> parser = Parser([tk0, tk1, tk2, tk3])
131     >>> exp = parser.parse()
132     >>> exp.eval()
133     True
134     """
135     return None
```

[VPL](#)