

## 2025\_2 - COMPILADORES - METATURMA

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

> **AV4 - EXPRESSÕES ARITMÉTICAS COM VARIÁVEIS**

Descrição

Visualizar envios

### AV4 - Expressões aritméticas com variáveis

**Data de entrega:** sexta, 12 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 adicionar variáveis às expressões aritméticas de nossa linguagem. Para tanto, iremos adicionar uma construção chamada "blocos let" à linguagem. Um bloco let tem a seguinte sintaxe:

```
let var <- exp0 in exp1 end
```

O valor de um bloco let é o valor de exp1, mas em um ambiente em que o nome "var" possui o valor de exp0. A título de exemplo, as seguintes igualdades são todas verdadeiras:

```
let x <- 1 in x end = 1
```

```
let x <- 2 in x * x end = 4
```

```
let x <- 3 in let y <- 4 in let z <- 6 in (x + y) * z end end end = 42
```

A adição de blocos let à nossa linguagem modifica a gramática de expressões aritméticas de duas formas. Essas duas modificações aparecem destacadas na Figura 1.

```
exp ::= exp + exp
      | exp - exp
      | exp * exp
      | exp / exp
      | exp <= exp
      | exp < exp
      | exp = exp
      | not exp
      | ~ exp
      | ( exp )
      | let name <- exp in exp end
      | true
      | false
      | num
      | var
```

Figura 1: Gramática de expressões aritméticas

Nível de precedência	Operador lógico/aritmético
1	~ ( ) let-in-end
2	* /
3	+ -
4	<= < =
5	not

Figura 2: precedência dos operadores

```
var ::= alpha(alpha|digit)*
num ::= digit*
```

Figura 3: Estrutura léxica

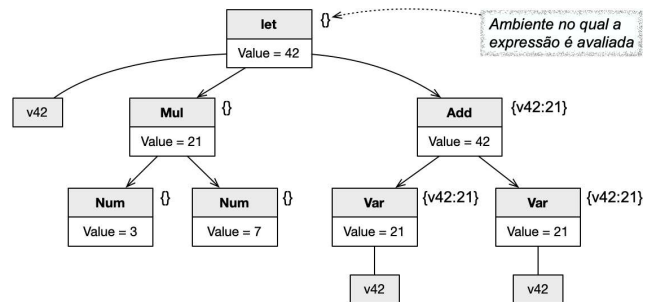


Figura 4: AST produzida para let v42 = 2 \* 7 in v42 \* v42 end no ambiente vazio {}

Note que blocos let possuem alta precedência. Em outras palavras, uma expressão como `let v <- 3 in v + v end * 2` possui o valor 12, assim como a expressão `2 * let v <- 3 in v + v end`. A Figura 2 acima mostra a precedência relativa entre os diferentes lexemas de nossa linguagem. Veja também que blocos let estão presentes na linguagem [cool](#), porém, naquela linguagem, esses blocos não possuem a palavra chave 'end' no final (conforme pode-se observar na página 16 do manual da linguagem). Em nosso caso, o end final simplifica a análise léxica da linguagem.

A adição de blocos let à linguagem vai nos forçar a modificar os três arquivos feitos no exercício anterior. Conforme pode ser visto na Figura 3 acima, agora temos de reconhecer "nomes de variáveis". Nomes de variáveis são sequências que começam com uma letra (maiúscula ou minúscula), e continuam com letras ou números. Uma modificação muito mais importante ocorre, contudo, na avaliação de expressões aritméticas. Neste caso, expressões passam a ser avaliadas em um "ambiente". O ambiente é uma tabela que associa o nome das variáveis com valores. Em outras palavras, nossa interface para Expressões Lógicas e Aritméticas passa a ser assim:

```
class Expression(ABC):
    @abstractmethod
```

```
def eval(self, env):  
    raise NotImplementedError
```

A avaliação de uma variável, em um certo ambiente `env`, é o valor associado com o nome da variável. Pode-se implementar `env` como um dicionário em Python. Assim, o valor de uma variável "x" é `env["x"]`. Note que é um erro avaliar uma variável em um ambiente que não contém uma definição de seu nome. Neste caso, seu interpretador precisa abortar o processo de avaliação, imprimindo a seguinte mensagem:

```
Variavel inexistente {nome_da_variavel}
```

Note que a palavra "Variavel" deve ser impressa sem acento! A título de exemplo, considere as chamadas abaixo:

```
>>> e = Var('var')  
>>> e.eval({'var': 42})  
42
```

```
>>> e = Var('v42')  
>>> e.eval({})  
Variavel inexistente v42
```

Para completar o VPL, você deverá modificar as implementações de três arquivos do exercício anterior: `Lexer.py`, `Expression.py` e `Parser.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 + let v <- 3 in v * v end # CTRL+D  
  
Value is 11
```

A implementação dos diferentes arquivos possui vários comentários doctest, que testam sua implementação. Caso queira testar seu código, simplesmente faça:

```
python3 -m doctest xx.py
```

No exemplo acima, substitua `xx.py` por algum dos arquivos que você precisa modificar. Caso você não gere mensagens de erro, então seu trabalho está (provavelmente) 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      VAR = 7 # An identifier
32      LET = 8 # The 'let' of the let expression
33      INX = 9 # The 'in' of the let expression
34      END = 10 # The 'end' of the let expression
35      EQL = 201
36      ADD = 202
37      SUB = 203
38      MUL = 204
39      DIV = 205
40      LEQ = 206
41      LTH = 207
42      NEG = 208
43      NOT = 209
44      LPR = 210
45      RPR = 211
46      ASN = 212 # The assignment '<-' operator
47
48
49  class Lexer:
50
51     def __init__(self, source):
52         """
53         The constructor of the lexer. It receives the string that shall be
54         scanned.
55         TODO: You will need to implement this method.
56         """
57         pass
58
59     def tokens(self):
60         """
61         This method is a token generator: it converts the string encapsulated
62         into this object into a sequence of Tokens. Examples:
63
64         >>> l = Lexer("1 + 3")
65         >>> [tk.kind for tk in l.tokens()]
66         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
67
68         >>> l = Lexer('1 * 2 -- 3\\n')
69         >>> [tk.kind for tk in l.tokens()]
70         [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
71
72         >>> l = Lexer("1 + var")
73         >>> [tk.kind for tk in l.tokens()]
74         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
75
76         >>> l = Lexer("let v <- 2 in v end")
77         >>> [tk.kind.name for tk in l.tokens()]
78         ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
79         """
80         token = self.getToken()
81         while token.kind != TokenType.EOF:
82             if token.kind != TokenType.WSP and token.kind != TokenType.COM:
83                 yield token
84             token = self.getToken()
85
86     def getToken(self):
87         """
88         Return the next token.
89         TODO: Implement this method!
90         """
91         token = None
92         return token

```

Expression.py

```

1 import sys
2 from abc import ABC, abstractmethod
3
4 class Expression(ABC):
5     @abstractmethod
6     def eval(self, env):
7         raise NotImplementedError
8
9 class Var(Expression):
10     """
11     This class represents expressions that are identifiers. The value of an
12     identifier is the value associated with it in the environment table.
13     """
14     def __init__(self, identifier):
15         self.identifier = identifier
16     def eval(self, env):
17         """
18         Example:
19         >>> e = Var('var')
20         >>> e.eval({'var': 42})
21         42
22
23         >>> e = Var('v42')
24         >>> e.eval({'v42': True, 'v31': 5})
25         True
26         """
27         # TODO: Implement this method
28         return None
29
30 class Bln(Expression):
31     """
32     This class represents expressions that are boolean values. There are only
33     two boolean values: true and false. The evaluation of such an expression is
34     the boolean itself.
35     """
36     def __init__(self, bln):
37         self.blm = bln
38     def eval(self, _):
39         """
40         Example:
41         >>> e = Bln(True)
42         >>> e.eval(None)
43         True
44         """
45         # TODO: Implement this method
46         return None
47
48 class Num(Expression):
49     """
50     This class represents expressions that are numbers. The evaluation of such
51     an expression is the number itself.
52     """
53     def __init__(self, num):
54         self.num = num
55     def eval(self, _):
56         """
57         Example:
58         >>> e = Num(3)
59         >>> e.eval(None)
60         3
61         """
62         # TODO: Implement this method
63         return None
64
65 class BinaryExpression(Expression):
66     """
67     This class represents binary expressions. A binary expression has two
68     sub-expressions: the left operand and the right operand.
69     """
70     def __init__(self, left, right):
71         self.left = left
72         self.right = right
73
74     @abstractmethod
75     def eval(self, env):
76         raise NotImplementedError
77
78 class Eql(BinaryExpression):
79     """
80     This class represents the equality between two expressions. The evaluation
81     of such an expression is True if the subexpressions are the same, or false
82     otherwise.
83     """
84     def eval(self, env):
85         """
86         Example:
87         >>> n1 = Num(3)
88         >>> n2 = Num(4)
89         >>> e = Eql(n1, n2)
90         >>> e.eval(None)
91         False
92
93         >>> n1 = Num(3)
94         >>> n2 = Num(3)
95         >>> e = Eql(n1, n2)
96         >>> e.eval(None)
97         True
98         """
99         # TODO: Implement this method
100         return None
101
102 class Add(BinaryExpression):
103     """

```

```

104     This class represents addition of two expressions. The evaluation of such
105     an expression is the addition of the two subexpression's values.
106     """
107     def eval(self, env):
108         """
109         Example:
110         >>> n1 = Num(3)
111         >>> n2 = Num(4)
112         >>> e = Add(n1, n2)
113         >>> e.eval(None)
114         7
115         """
116         # TODO: Implement this method
117         return None
118
119     class Sub(BinaryExpression):
120         """
121         This class represents subtraction of two expressions. The evaluation of such
122         an expression is the subtraction of the two subexpression's values.
123         """
124         def eval(self, env):
125             """
126             Example:
127             >>> n1 = Num(3)
128             >>> n2 = Num(4)
129             >>> e = Sub(n1, n2)
130             >>> e.eval(None)
131             -1
132             """
133             # TODO: Implement this method
134             return None
135
136         class Mul(BinaryExpression):
137             """
138             This class represents multiplication of two expressions. The evaluation of
139             such an expression is the product of the two subexpression's values.
140             """
141             def eval(self, env):
142                 """
143                 Example:
144                 >>> n1 = Num(3)
145                 >>> n2 = Num(4)
146                 >>> e = Mul(n1, n2)
147                 >>> e.eval(None)
148                 12
149                 """
150                 # TODO: Implement this method
151                 return None
152
153             class Div(BinaryExpression):
154                 """
155                 This class represents the integer division of two expressions. The
156                 evaluation of such an expression is the integer quotient of the two
157                 subexpression's values.
158                 """
159                 def eval(self, env):
160                     """
161                     Example:
162                     >>> n1 = Num(28)
163                     >>> n2 = Num(4)
164                     >>> e = Div(n1, n2)
165                     >>> e.eval(None)
166                     7
167                     >>> n1 = Num(22)
168                     >>> n2 = Num(4)
169                     >>> e = Div(n1, n2)
170                     >>> e.eval(None)
171                     5
172                     """
173                     # TODO: Implement this method
174                     return None
175
176             class Leq(BinaryExpression):
177                 """
178                 This class represents comparison of two expressions using the
179                 less-than-or-equal comparator. The evaluation of such an expression is a
180                 boolean value that is true if the left operand is less than or equal the
181                 right operand. It is false otherwise.
182                 """
183                 def eval(self, env):
184                     """
185                     Example:
186                     >>> n1 = Num(3)
187                     >>> n2 = Num(4)
188                     >>> e = Leq(n1, n2)
189                     >>> e.eval(None)
190                     True
191                     >>> n1 = Num(3)
192                     >>> n2 = Num(3)
193                     >>> e = Leq(n1, n2)
194                     >>> e.eval(None)
195                     True
196                     >>> n1 = Num(4)
197                     >>> n2 = Num(3)
198                     >>> e = Leq(n1, n2)
199                     >>> e.eval(None)
200                     False
201                     """
202                     # TODO: Implement this method
203                     return None
204
205             class Lth(BinaryExpression):
206                 """
207                 This class represents comparison of two expressions using the
208                 less-than comparator. The evaluation of such an expression is a
209                 boolean value that is true if the left operand is less than the
210                 right operand. It is false otherwise.
211                 """
212                 def eval(self, env):
213                     """
214                     Example:
215                     >>> n1 = Num(3)
216                     >>> n2 = Num(4)
217                     >>> e = Lth(n1, n2)
218                     >>> e.eval(None)
219                     True
220                     >>> n1 = Num(3)
221                     >>> n2 = Num(3)
222                     >>> e = Lth(n1, n2)
223                     >>> e.eval(None)
224                     False
225                     >>> n1 = Num(4)
226                     >>> n2 = Num(3)
227                     >>> e = Lth(n1, n2)
228                     >>> e.eval(None)
229                     False
230                     """
231                     # TODO: Implement this method
232                     return None

```

```

207     """
208     This class represents comparison of two expressions using the
209     less-than comparison operator. The evaluation of such an expression is a
210     boolean value that is true if the left operand is less than the right
211     operand. It is false otherwise.
212     """
213     def eval(self, env):
214         """
215         Example:
216         >>> n1 = Num(3)
217         >>> n2 = Num(4)
218         >>> e = Lth(n1, n2)
219         >>> e.eval(None)
220         True
221         >>> n1 = Num(3)
222         >>> n2 = Num(3)
223         >>> e = Lth(n1, n2)
224         >>> e.eval(None)
225         False
226         >>> n1 = Num(4)
227         >>> n2 = Num(3)
228         >>> e = Lth(n1, n2)
229         >>> e.eval(None)
230         False
231         """
232         # TODO: Implement this method
233         return None
234
235     class UnaryExpression(Expression):
236         """
237         This class represents unary expressions. A unary expression has only one
238         sub-expression.
239         """
240         def __init__(self, exp):
241             self.exp = exp
242
243         @abstractmethod
244         def eval(self, env):
245             raise NotImplementedError
246
247     class Neg(UnaryExpression):
248         """
249         This expression represents the additive inverse of a number. The additive
250         inverse of a number n is the number -n, so that the sum of both is zero.
251         """
252         def eval(self, env):
253             """
254             Example:
255             >>> n = Num(3)
256             >>> e = Neg(n)
257             >>> e.eval(None)
258             -3
259             >>> n = Num(0)
260             >>> e = Neg(n)
261             >>> e.eval(None)
262             0
263             """
264             # TODO: Implement this method
265             return None
266
267     class Not(UnaryExpression):
268         """
269         This expression represents the negation of a boolean. The negation of a
270         boolean expression is the logical complement of that expression.
271         """
272         def eval(self, env):
273             """
274             Example:
275             >>> t = Bln(True)
276             >>> e = Not(t)
277             >>> e.eval(None)
278             False
279             >>> t = Bln(False)
280             >>> e = Not(t)
281             >>> e.eval(None)
282             True
283             """
284             # TODO: Implement this method
285             return None
286
287     class Let(Expression):
288         """
289         This class represents a let expression. The semantics of a let expression,
290         such as "let v <- e0 in e1" on an environment env is as follows:
291         1. Evaluate e0 in the environment env, yielding e0_val
292         2. Evaluate e1 in the new environment env' = env + {v:e0_val}
293         """
294         def __init__(self, identifier, exp_def, exp_body):
295             # Notice that the identifier is a string, not an expression!
296             self.identifier = identifier
297             self.exp_def = exp_def
298             self.exp_body = exp_body
299         def eval(self, env):
300             """
301             Example:
302             >>> e = Let('v', Num(42), Var('v'))
303             >>> e.eval({})
304             42
305
306             >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
307             >>> e.eval({})
308             42
309
310             >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
311             >>> e.eval({})

```

```
310  
311     eval(\s)  
312     ""  
313     # TODO: Implement this method  
314     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(None)
30         123
31
32         >>> parser = Parser([Token('True', TokenType.TRU)])
33         >>> exp = parser.parse()
34         >>> exp.eval(None)
35         True
36
37         >>> parser = Parser([Token('False', TokenType.FLS)])
38         >>> exp = parser.parse()
39         >>> exp.eval(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
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(None)
133     True
134
135     >>> tk0 = Token('let', TokenType.LET)
136     >>> tk1 = Token('v', TokenType.VAR)
137     >>> tk2 = Token('<-', TokenType.ASN)
138     >>> tk3 = Token('42', TokenType.NUM)
139     >>> tk4 = Token('in', TokenType.INX)
140     >>> tk5 = Token('v', TokenType.VAR)
141     >>> tk6 = Token('end', TokenType.END)
142     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
143     >>> exp = parser.parse()
144     >>> exp.eval({})
145     42
146
147     >>> tk0 = Token('let', TokenType.LET)
148     >>> tk1 = Token('v', TokenType.VAR)
149     >>> tk2 = Token('<-', TokenType.ASN)
150     >>> tk3 = Token('21', TokenType.NUM)
151     >>> tk4 = Token('in', TokenType.INX)
152     >>> tk5 = Token('v', TokenType.VAR)
153     >>> tk6 = Token('+', TokenType.ADD)
154     >>> tk7 = Token('v', TokenType.VAR)
155     >>> tk8 = Token('end', TokenType.END)
156     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
157     >>> exp = parser.parse()
158     >>> exp.eval({})
159     42
160     """
161     return None
```

[VPL](#)