

2025_2 - COMPILADORES - METATURMA

[PAÍNEL](#) > [MINHAS TURMAS](#) > [2025_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)
 > [AV5 - VISITORS](#)

[Descrição](#)

[Visualizar envios](#)

AV5 - Visitors

Data de entrega: sexta, 19 Set 2025, 23:59

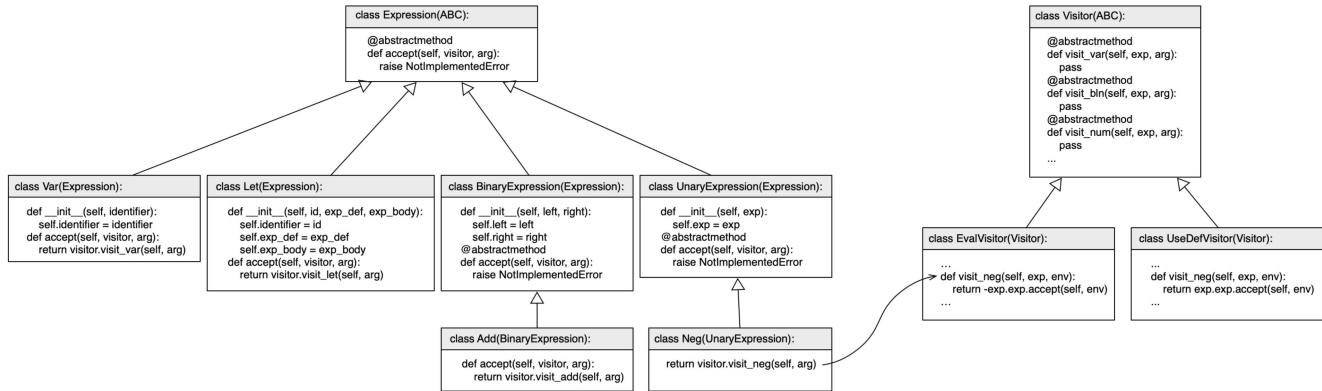
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py ([Baixar](#))

Tipo de trabalho: Trabalho individual

O objetivo deste trabalho é utilizar o padrão de projetos [Visitor](#). Para tanto, você deverá implementar dois tipos de visitors:

1. O primeiro visitor ([EvalVisitor](#)) avalia expressões lógicas e aritméticas. Esse visitor deve produzir os mesmos resultados que a função eval que havia sido feita nos exercícios anteriores.
2. O segundo visitor ([UseDefVisitor](#)) implementa uma [análise semântica](#) simples, que determina se uma expressão contém alguma variável que foi usada sem ter sido definida.

O padrão visitor nos permite remover do próprio componente (em nosso caso, instâncias de `Expression`), o código que processa esse componente (em nosso caso, a antiga função `Expression.eval()`). Assim, podemos ter diversas operações implementadas sobre instâncias de `Expression` sem precisar modificar essas instâncias. A figura abaixo ilustra como ficará nossa arquitetura de software, uma vez que o padrão esteja implementado.



O padrão visitor envolve duas partes: implementar o método `accept` sobre cada tipo de componente (instância de `Expression`), e implementar os visitors concretos. A primeira parte já está feita para você. Você não precisa modificar o arquivo `Expression.py`. Porém, seria interessante que você abrisse este arquivo para ler seu código (afinal, **o padrão visitor é assunto de prova, e será cobrado!**). Por outro lado, você deverá implementar boa parte do arquivo `Visitor.py`. Um dos visitors que se encontra declarado neste arquivo, `EvalVisitor`, já provê a assinatura dos métodos. Você deverá completar o corpo destes métodos. O outro visitor, `UseDefVisitor`, contudo, não provê nem mesmo a assinatura dos métodos. Você deverá implementar todos esses métodos. Note que os dois visitors possuem estrutura similar.

Além disso, você deverá implementar uma função `safe_eval(exp)` que faz a avaliação de expressões lógicas e aritméticas, mas somente se essas expressões não contêm variáveis não definidas. Essa função combina os dois visitors que você desenvolveu neste trabalho. Primeiro, ela usa `UseDefVisitor` para verificar se não existem variáveis indefinidas. Caso existam variáveis não definidas, então a função imprime a mensagem abaixo:

Error: expression contains undefined variables.

De outro modo, ou seja, se a expressão for válida, então sua função imprime a mensagem abaixo, sendo XX o valor da expressão lógica e aritmética `exp` avaliada:

```
Value is XX
```

Para completar este VPL, você deverá entregar cinco arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Visitor.py` e `driver.py`. Porém, somente `Visitor.py` constitui parte nova, que você deverá implementar. Você não deverá alterar `driver.py`. E os outros arquivos, exceto `Expression.py`, você pode reutilizar do exercício anterior. `Expression.py` já está feito para você. Você não deve modificar este arquivo. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
$> python3 driver.py
eval # esta palavra define a opção que será testada
2 + let v <- 3 in v * v end # Aperte 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 queira testar (experimente com `Expression.py`, por exemplo). 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 Visitor import *
4 from Lexer import Lexer
5 from Parser import Parser
6
7 if __name__ == "__main__":
8     """
9     Este arquivo não deve ser alterado, mas deve ser enviado para resolver o
10    VPL. O arquivo contém o código que testa a implementação do parser.
11    """
12    text = sys.stdin.read()
13    (option, rest) = text.split(maxsplit=1)
14    lexer = Lexer(rest)
15    parser = Parser(lexer.tokens())
16    exp = parser.parse()
17    if option == 'eval':
18        visitor = EvalVisitor()
19        print(f"Value is {exp.accept(visitor, {})}")
20    elif option == 'usedef':
21        visitor = UseDefVisitor()
22        print(f"Are there undefs? {len(exp.accept(visitor, set())) > 0}")
23    elif option == 'safe_eval':
24        safe_eval(exp)
25    else:
26        sys.exit(f"Invalid option = {option}")
```

`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
81         token = self.getToken()
82         while token.kind != TokenType.EOF:
83             if token.kind != TokenType.WSP and token.kind != TokenType.COM:
84                 yield token
85             token = self.getToken()
86
87     def getToken(self):
88         """
89             Return the next token.
90             TODO: Implement this method!
91             """
92         token = None
93         return token

```

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

```

Expression.py

```

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

```

```

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

```

```

20/
208     ....
209     return visitor.visit_leq(self, arg)
210
210 class Lth(BinaryExpression):
211     """
212         This class represents comparison of two expressions using the
213         less-than comparison operator. The acceptation of such an expression is a
214         boolean value that is true if the left operand is less than the right
215         operand. It is false otherwise.
216     """
217     def accept(self, visitor, arg):
218         """
219             Example:
220             >>> n1 = Num(3)
221             >>> n2 = Num(4)
222             >>> e = Lth(n1, n2)
223             >>> ev = EvalVisitor()
224             >>> e.accept(ev, None)
225             True
226             >>> n1 = Num(3)
227             >>> n2 = Num(3)
228             >>> e = Lth(n1, n2)
229             >>> ev = EvalVisitor()
230             >>> e.accept(ev, None)
231             False
232             >>> n1 = Num(4)
233             >>> n2 = Num(3)
234             >>> e = Lth(n1, n2)
235             >>> ev = EvalVisitor()
236             >>> e.accept(ev, None)
237             False
238         """
239     return visitor.visit_lth(self, arg)
240
241 class UnaryExpression(Expression):
242     """
243         This class represents unary expressions. A unary expression has only one
244         sub-expression.
245     """
246     def __init__(self, exp):
247         self.exp = exp
248
249     @abstractmethod
250     def accept(self, visitor, arg):
251         raise NotImplementedError
252
253 class Neg(UnaryExpression):
254     """
255         This expression represents the additive inverse of a number. The additive
256         inverse of a number n is the number -n, so that the sum of both is zero.
257     """
258     def accept(self, visitor, arg):
259         """
260             Example:
261             >>> n = Num(3)
262             >>> e = Neg(n)
263             >>> ev = EvalVisitor()
264             >>> e.accept(ev, None)
265             -3
266             >>> n = Num(0)
267             >>> e = Neg(n)
268             >>> ev = EvalVisitor()
269             >>> e.accept(ev, None)
270             0
271         """
272     return visitor.visit_neg(self, arg)
273
274 class Not(UnaryExpression):
275     """
276         This expression represents the negation of a boolean. The negation of a
277         boolean expression is the logical complement of that expression.
278     """
279     def accept(self, visitor, arg):
280         """
281             Example:
282             >>> t = Bln(True)
283             >>> e = Not(t)
284             >>> ev = EvalVisitor()
285             >>> e.accept(ev, None)
286             False
287             >>> t = Bln(False)
288             >>> e = Not(t)
289             >>> ev = EvalVisitor()
290             >>> e.accept(ev, None)
291             True
292         """
293     return visitor.visit_not(self, arg)
294
295 class Let(Expression):
296     """
297         This class represents a let expression. The semantics of a let expression,
298         such as "let v <- e0 in e1" on an environment env is as follows:
299         1. Evaluate e0 in the environment env, yielding e0_val
300         2. Evaluate e1 in the new environment env' = env + {v:e0_val}
301     """
302     def __init__(self, identifier, exp_def, exp_body):
303         self.identifier = identifier
304         self.exp_def = exp_def
305         self.exp_body = exp_body
306     def accept(self, visitor, arg):
307         """
308             Example:
309             >>> e = Let('v', Num(42), Var('v'))
310             >>> ev = EvalVisitor()

```

```
310
311     >>> ev = EvalVisitor()
312     >>> e.accept(ev, {})
313     42
314
315     >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
316     >>> ev = EvalVisitor()
317     >>> e.accept(ev, {})
318     42
319
320     >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
321     >>> ev = EvalVisitor()
322     >>> e.accept(ev, {})
323     1764
324     """
325
326     return visitor.visit let(self, arg)
```

Visitor.py

```

1 import sys
2 from abc import ABC, abstractmethod
3 from Expression import *
4
5 class Visitor(ABC):
6     """
7         The visitor pattern consists of two abstract classes: the Expression and the
8         Visitor. The Expression class defines one method: 'accept(visitor, args)'.
9         This method takes in an implementation of a visitor, and the arguments that
10        are passed from expression to expression. The Visitor class defines one
11        specific method for each subclass of Expression. Each instance of such a
12        subclass will invoke the right visiting method.
13    """
14    @abstractmethod
15    def visit_var(self, exp, arg):
16        pass
17
18    @abstractmethod
19    def visit_bln(self, exp, arg):
20        pass
21
22    @abstractmethod
23    def visit_num(self, exp, arg):
24        pass
25
26    @abstractmethod
27    def visit_eql(self, exp, arg):
28        pass
29
30    @abstractmethod
31    def visit_add(self, exp, arg):
32        pass
33
34    @abstractmethod
35    def visit_sub(self, exp, arg):
36        pass
37
38    @abstractmethod
39    def visit_mul(self, exp, arg):
40        pass
41
42    @abstractmethod
43    def visit_div(self, exp, arg):
44        pass
45
46    @abstractmethod
47    def visit_leq(self, exp, arg):
48        pass
49
50    @abstractmethod
51    def visit_lth(self, exp, arg):
52        pass
53
54    @abstractmethod
55    def visit_neg(self, exp, arg):
56        pass
57
58    @abstractmethod
59    def visit_not(self, exp, arg):
60        pass
61
62    @abstractmethod
63    def visit_let(self, exp, arg):
64        pass
65
66 class EvalVisitor(Visitor):
67     """
68         The EvalVisitor class evaluates logical and arithmetic expressions. The
69         result of evaluating an expression is the value of that expression. The
70         inherited attribute propagated throughout visits is the environment that
71         associates the names of variables with values.
72     """
73     Examples:
74     >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
75     >>> e1 = Not(Eql(e0, Num(1764)))
76     >>> ev = EvalVisitor()
77     >>> e1.accept(ev, {})
78     False
79
80     >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
81     >>> e1 = Lth(e0, Var('x'))
82     >>> ev = EvalVisitor()
83     >>> e1.accept(ev, {'x': 41})
84     True
85     """
86     def visit_var(self, exp, env):
87         # TODO: Implement this method!
88         raise NotImplementedError
89
90     def visit_bln(self, exp, env):
91         # TODO: Implement this method!
92         raise NotImplementedError
93
94     def visit_num(self, exp, env):
95         # TODO: Implement this method!
96         raise NotImplementedError
97
98     def visit_eql(self, exp, env):
99         # TODO: Implement this method!
100        raise NotImplementedError
101
102    def visit_add(self, exp, env):
103        # TODO: Implement this method!

```

```

104     raise NotImplementedError
105
106     def visit_sub(self, exp, env):
107         # TODO: Implement this method!
108         raise NotImplementedError
109
110     def visit_mul(self, exp, env):
111         # TODO: Implement this method!
112         raise NotImplementedError
113
114     def visit_div(self, exp, env):
115         # TODO: Implement this method!
116         raise NotImplementedError
117
118     def visit_leq(self, exp, env):
119         # TODO: Implement this method!
120         raise NotImplementedError
121
122     def visit_lth(self, exp, env):
123         # TODO: Implement this method!
124         raise NotImplementedError
125
126     def visit_neg(self, exp, env):
127         # TODO: Implement this method!
128         raise NotImplementedError
129
130     def visit_not(self, exp, env):
131         # TODO: Implement this method!
132         raise NotImplementedError
133
134     def visit_let(self, exp, env):
135         # TODO: Implement this method!
136         raise NotImplementedError
137
138 class UseDefVisitor(Visitor):
139 """
140     The UseDefVisitor class reports the use of undefined variables. It takes
141     as input an environment of defined variables, and produces, as output,
142     the set of all the variables that are used without being defined.
143
144     Examples:
145     >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
146     >>> e1 = Not(Eql(e0, Num(1764)))
147     >>> ev = UseDefVisitor()
148     >>> len(e1.accept(ev, set()))
149     0
150
151     >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
152     >>> e1 = Lth(e0, Var('x'))
153     >>> ev = UseDefVisitor()
154     >>> len(e1.accept(ev, set()))
155     1
156
157     >>> e = Let('v', Add(Num(40), Var('v')), Sub(Var('v'), Num(2)))
158     >>> ev = UseDefVisitor()
159     >>> len(e.accept(ev, set()))
160     1
161
162     >>> e1 = Let('v', Add(Num(40), Var('v')), Sub(Var('v'), Num(2)))
163     >>> e0 = Let('v', Num(3), e1)
164     >>> ev = UseDefVisitor()
165     >>> len(e0.accept(ev, set()))
166     0
167     """
168
169     # TODO: Implement all the 13 methods of the visitor.
170
171     def safe_eval(exp):
172 """
173     This method applies one simple semantic analysis onto an expression, before
174     evaluating it: it checks if the expression contains free variables, there
175     is, variables used without being defined.
176
177     Example:
178     >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
179     >>> e1 = Not(Eql(e0, Num(1764)))
180     >>> safe_eval(e1)
181     Value is False
182
183     >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
184     >>> e1 = Lth(e0, Var('x'))
185     >>> safe_eval(e1)
186     Error: expression contains undefined variables.
187     """
188
189     # TODO: Implement this method!
190     raise NotImplementedError

```