2025_2 - COMPILADORES - METATURMA

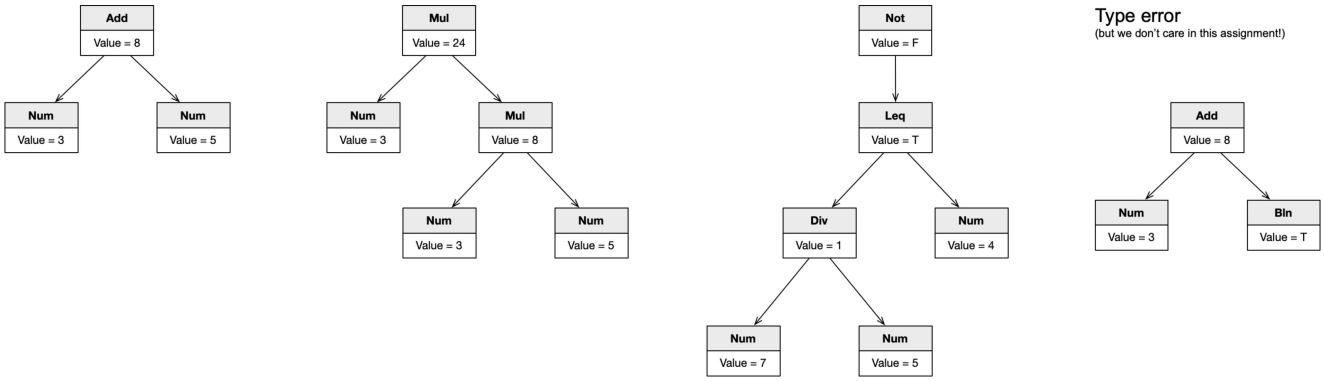
Descrição Visualizar envios

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

```python
import sys
from Expression import *

if __name__ == "__main__":
    """
    Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
    VPL. O arquivo contem o codigo que testa a implementacao do analisador
    lexico.
    """
    e = eval(sys.stdin.read())
    print(f"Value is {e.eval()}")
```

Expression.py

```python
from abc import ABC, abstractmethod

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

class Bln(Expression):
    """
    This class represents expressions that are boolean values. There are only
    two boolean values: true and false. The evaluation of such an expression is
    the boolean itself.
    """
    def __init__(self, bln):
        self.bln = bln
    def eval(self):
        """
        Example:
        >>> e = Bln(True)
        >>> e.eval()
        True
        """
        # TODO: Implement this method!
        return None

class Num(Expression):
    """
    This class represents expressions that are numbers. The evaluation of such
    an expression is the number itself.
    """
    def __init__(self, num):
        self.num = num
    def eval(self):
        """
        Example:
        >>> e = Num(3)
        >>> e.eval()
        3
        """
        # TODO: Implement this method!
        return None

class BinaryExpression(Expression):
    """
    This class represents binary expressions. A binary expression has two
    sub-expressions: the left operand and the right operand.
    """
    def __init__(self, left, right):
        self.left = left
        self.right = right

    @abstractmethod
    def eval(self):
        raise NotImplementedError

class Eql(BinaryExpression):
    """
    This class represents the equality between two expressions. The evaluation
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Eql(n1, n2)
        >>> e.eval()
        False

        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Eql(n1, n2)
        >>> e.eval()
        True
        """
        # TODO: Implement this method!
        return None

class Add(BinaryExpression):
    """
    This class represents addition of two expressions. The evaluation of such
    an expression is the addition of the two subexpression's values.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> e.eval()
        7
        """
        # TODO: Implement this method!
        return None

class Sub(BinaryExpression):
    """
    This class represents subtraction of two expressions. The evaluation of such
    an expression is the subtraction of the two subexpression's values.
    """
    def eval(self):
        """
```

```
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> e.eval()
        -1
        """
        # TODO: Implement this method!
        return None

class Mul(BinaryExpression):
    """
    This class represents multiplication of two expressions. The evaluation of
    such an expression is the product of the two subexpression's values.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> e.eval()
        12
        """
        # TODO: Implement this method!
        return None

class Div(BinaryExpression):
    """
    This class represents the integer division of two expressions. The
    evaluation of such an expression is the integer quocient of the two
    subexpression's values.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> e.eval()
        7
        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> e.eval()
        5
        """
        # TODO: Implement this method!
        return None

class Leq(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than-or-equal comparator. The evaluation of such an expression is a
    boolean value that is true if the left operand is less than or equal the
    right operand. It is false otherwise.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Leq(n1, n2)
        >>> e.eval()
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> e.eval()
        True
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> e.eval()
        False
        """
        # TODO: Implement this method!
        return None

class Lth(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than comparison operator. The evaluation of such an expression is a
    boolean value that is true if the left operand is less than the right
    operand. It is false otherwise.
    """
    def eval(self):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Lth(n1, n2)
        >>> e.eval()
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> e.eval()
        False
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> e.eval()
```

```
207         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