2025_2 - COMPILADORES - METATURMA

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

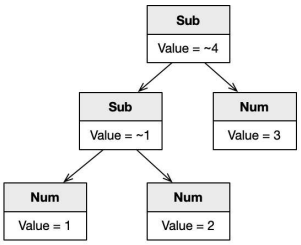

Figura 3: AST produzida para 1 - 2 - 3

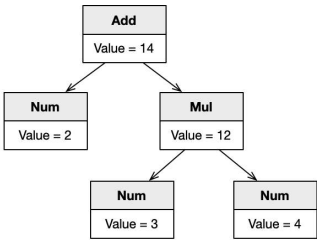

Figura 4: AST produzida para 2 + 3 * 4

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

```
import sys
from Expression import *
from Lexer import Lexer
from Parser import Parser

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 parser.
    """
    lexer = Lexer(sys.stdin.read())
    parser = Parser(lexer.tokens())
    exp = parser.parse()
    print(f"Value is {exp.eval()}")
```

Lexer.py

```
import sys
import enum


class Token:
    """
    This class contains the definition of Tokens. A token has two fields: its
    text and its kind. The "kind" of a token is a constant that identifies it
    uniquely. See the TokenType to know the possible identifiers (if you want).
    You don't need to change this class.
    """
    def __init__(self, tokenText, tokenKind):
        # The token's actual text. Used for identifiers, strings, and numbers.
        self.text = tokenText
        # The TokenType that this token is classified as.
        self.kind = tokenKind


class TokenType(enum.Enum):
    """
    These are the possible tokens. You don't need to change this class at all.
    """
    EOF = -1  # End of file
    NLN = 0   # New line
    WSP = 1   # White Space
    COM = 2   # Comment
    NUM = 3   # Number (integers)
    STR = 4   # Strings
    TRU = 5   # The constant true
    FLS = 6   # The constant false
    EQL = 201
    ADD = 202
    SUB = 203
    MUL = 204
    DIV = 205
    LEQ = 206
    LTH = 207
    NEG = 208
    NOT = 209
    LPR = 210
    RPR = 211


class Lexer:

    def __init__(self, source):
        """
        The constructor of the lexer. It receives the string that shall be
        scanned.
        TODO: You will need to implement this method.
        """
        pass

    def tokens(self):
        """
        This method is a token generator: it converts the string encapsulated
        into this object into a sequence of Tokens. Examples:

        >>> l = Lexer('1 * 2 - 3')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]

        >>> l = Lexer('1 * 2 -- 3\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
        """
        token = self.getToken()
        while token.kind != TokenType.EOF:
            if token.kind != TokenType.WSP and token.kind != TokenType.COM:
                yield token
            token = self.getToken()

    def getToken(self):
        """
        Return the next token.
        TODO: Implement this method!
        """
        token = None
        return token
```

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()
```

```
        False
        """
        # TODO: Implement this method!
        return None

class UnaryExpression(Expression):
    """
    This class represents unary expressions. A unary expression has only one
    sub-expression.
    """
    def __init__(self, exp):
        self.exp = exp

    @abstractmethod
    def eval(self):
        raise NotImplementedError

class Neg(UnaryExpression):
    """
    This expression represents the additive inverse of a number. The additive
    inverse of a number n is the number -n, so that the sum of both is zero.
    """
    def eval(self):
        """
        Example:
        >>> n = Num(3)
        >>> e = Neg(n)
        >>> e.eval()
        -3
        >>> n = Num(0)
        >>> e = Neg(n)
        >>> e.eval()
        0
        """
        # TODO: Implement this method!
        return None

class Not(UnaryExpression):
    """
    This expression represents the negation of a boolean. The negation of a
    boolean expression is the logical complement of that expression.
    """
    def eval(self):
        """
        Example:
        >>> t = Bln(True)
        >>> e = Not(t)
        >>> e.eval()
        False
        >>> t = Bln(False)
        >>> e = Not(t)
        >>> e.eval()
        True
        """
        # TODO: Implement this method!
        return None
```

Parser.py

```python
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements the parser of arithmetic expressions.

References:
    see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
"""

class Parser:
    def __init__(self, tokens):
        """
        Initializes the parser. The parser keeps track of the list of tokens
        and the current token. For instance:
        """
        self.tokens = list(tokens)
        self.cur_token_idx = 0 # This is just a suggestion!

    def parse(self):
        """
        Returns the expression associated with the stream of tokens.

        Examples:
        >>> parser = Parser([Token('123', TokenType.NUM)])
        >>> exp = parser.parse()
        >>> exp.eval()
        123

        >>> parser = Parser([Token('True', TokenType.TRU)])
        >>> exp = parser.parse()
        >>> exp.eval()
        True

        >>> parser = Parser([Token('False', TokenType.FLS)])
        >>> exp = parser.parse()
        >>> exp.eval()
        False

        >>> tk0 = Token('~', TokenType.NEG)
        >>> tk1 = Token('123', TokenType.NUM)
        >>> parser = Parser([tk0, tk1])
        >>> exp = parser.parse()
        >>> exp.eval()
        -123

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        12

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('~', TokenType.NEG)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> exp.eval()
        -12

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('/', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        7

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('+', TokenType.ADD)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        7

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('-', TokenType.SUB)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        26

        >>> tk0 = Token('2', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('(', TokenType.LPR)
        >>> tk3 = Token('3', TokenType.NUM)
        >>> tk4 = Token('+', TokenType.ADD)
        >>> tk5 = Token('4', TokenType.NUM)
        >>> tk6 = Token(')', TokenType.RPR)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
        >>> exp = parser.parse()
        >>> exp.eval()
        14

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('==', TokenType.EQL)
```

```
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        True

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('<=', TokenType.LEQ)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        True

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('<', TokenType.LTH)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval()
        False

        >>> tk0 = Token('not', TokenType.NOT)
        >>> tk1 = Token('4', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> exp.eval()
        True
        """
        return None
```

VPL