2025_2 - COMPILADORES - METATURMA

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: Extrutura léxica

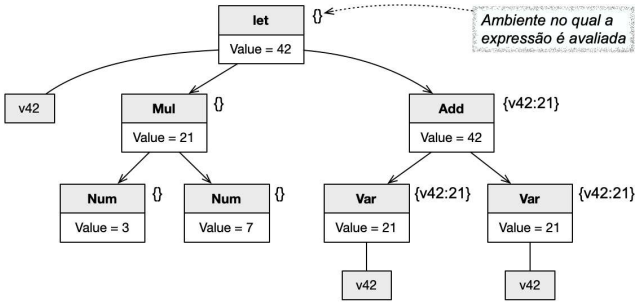

Figura 4: AST produzida para let v42 = 2 * 7 in v42 * v42 end no ambiente vazio {}

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_variável}
```

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, substituta `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

```
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

```python
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
    VAR = 7   # An identifier
    LET = 8   # The 'let' of the let expression
    INX = 9   # The 'in' of the let expression
    END = 10  # The 'end' of the let expression
    EQL = 201
    ADD = 202
    SUB = 203
    MUL = 204
    DIV = 205
    LEQ = 206
    LTH = 207
    NEG = 208
    NOT = 209
    LPR = 210
    RPR = 211
    ASN = 212 # The assignment '<-' operator


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 + 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>]

        >>> l = Lexer("1 + var")
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]

        >>> l = Lexer("let v <- 2 in v end")
        >>> [tk.kind.name for tk in l.tokens()]
        ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
        """
        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
import sys
from abc import ABC, abstractmethod

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

class Var(Expression):
    """
    This class represents expressions that are identifiers. The value of an
    indentifier is the value associated with it in the environment table.
    """
    def __init__(self, identifier):
        self.identifier = identifier
    def eval(self, env):
        """
        Example:
        >>> e = Var('var')
        >>> e.eval({'var': 42})
        42

        >>> e = Var('v42')
        >>> e.eval({'v42': True, 'v31': 5})
        True
        """
        # TODO: Implement this method
        return None

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(None)
        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(None)
        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, env):
        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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Eql(n1, n2)
        >>> e.eval(None)
        False

        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Eql(n1, n2)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> e.eval(None)
        -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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> e.eval(None)
        7
        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Leq(n1, n2)
        >>> e.eval(None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> e.eval(None)
        True
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Lth(n1, n2)
        >>> e.eval(None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> e.eval(None)
        False
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> e.eval(None)
        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, env):
        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, env):
        """
        Example:
        >>> n = Num(3)
        >>> e = Neg(n)
        >>> e.eval(None)
        -3
        >>> n = Num(0)
        >>> e = Neg(n)
        >>> e.eval(None)
        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, env):
        """
        Example:
        >>> t = Bln(True)
        >>> e = Not(t)
        >>> e.eval(None)
        False
        >>> t = Bln(False)
        >>> e = Not(t)
        >>> e.eval(None)
        True
        """
        # TODO: Implement this method
        return None

class Let(Expression):
    """
    This class represents a let expression. The semantics of a let expression,
    such as "let v <- e0 in e1" on an environment env is as follows:
    1. Evaluate e0 in the environment env, yielding e0_val
    2. Evaluate e1 in the new environment env' = env + {v:e0_val}
    """
    def __init__(self, identifier, exp_def, exp_body):
        # Notice that the identifier is a string, not an expression!
        self.identifier = identifier
        self.exp_def = exp_def
        self.exp_body = exp_body
    def eval(self, env):
        """
        Example:
        >>> e = Let('v', Num(42), Var('v'))
        >>> e.eval({})
        42

        >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
        >>> e.eval({})
        42

        >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
        >>> e.eval({})
```

```
310         >>> e.eval({})
311         1764
312         """
313         # TODO: Implement this method
314         return None
```

Parser.py

```
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(None)
        123

        >>> parser = Parser([Token('True', TokenType.TRU)])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        True

        >>> parser = Parser([Token('False', TokenType.FLS)])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        False

        >>> tk0 = Token('~', TokenType.NEG)
        >>> tk1 = Token('123', TokenType.NUM)
        >>> parser = Parser([tk0, tk1])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        -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(None)
        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(None)
        -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(None)
        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(None)
        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(None)
        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(None)
        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(None)
        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(None)
        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(None)
        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(None)
        True

        >>> tk0 = Token('let', TokenType.LET)
        >>> tk1 = Token('v', TokenType.VAR)
        >>> tk2 = Token('<-', TokenType.ASN)
        >>> tk3 = Token('42', TokenType.NUM)
        >>> tk4 = Token('in', TokenType.INX)
        >>> tk5 = Token('v', TokenType.VAR)
        >>> tk6 = Token('end', TokenType.END)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
        >>> exp = parser.parse()
        >>> exp.eval({})
        42

        >>> tk0 = Token('let', TokenType.LET)
        >>> tk1 = Token('v', TokenType.VAR)
        >>> tk2 = Token('<-', TokenType.ASN)
        >>> tk3 = Token('21', TokenType.NUM)
        >>> tk4 = Token('in', TokenType.INX)
        >>> tk5 = Token('v', TokenType.VAR)
        >>> tk6 = Token('+', TokenType.ADD)
        >>> tk7 = Token('v', TokenType.VAR)
        >>> tk8 = Token('end', TokenType.END)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
        >>> exp = parser.parse()
        >>> exp.eval({})
        42
        """
        return None
```

VPL