2025_2 - COMPILADORES - METATURMA

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

> **AV5 - VISITORS**

Descrição Visualizar envios

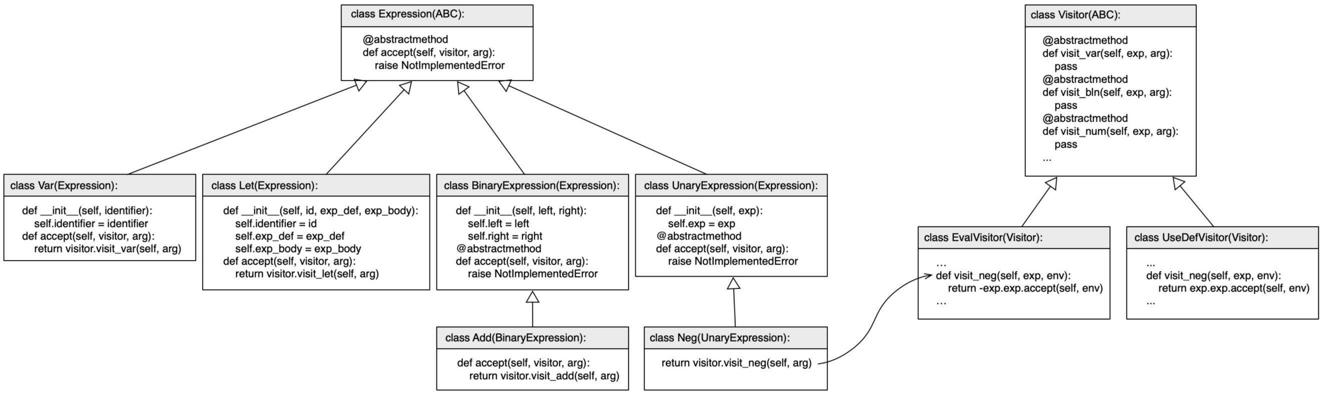
AV5 - Visitors

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 instânticas. A figura abaixo ilustra como ficará nossa arquitetura de software, uma vez que o padrão esteja implementado.



```
class Expression(ABC):
    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError
```

```
class Var(Expression):
    def __init__(self, identifier):
        self.identifier = identifier
    def accept(self, visitor, arg):
        return visitor.visit_var(self, arg)
```

```
class Let(Expression):
    def __init__(self, id, exp_def, exp_body):
        self.identifier = id
        self.exp_def = exp_def
        self.exp_body = exp_body
    def accept(self, visitor, arg):
        return visitor.visit_let(self, arg)
```

```
class BinaryExpression(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right
    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError
```

```
class UnaryExpression(Expression):
    def __init__(self, exp):
        self.exp = exp
    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError
```

```
class Add(BinaryExpression):
    def accept(self, visitor, arg):
        return visitor.visit_add(self, arg)
```

```
class Neg(UnaryExpression):
    return visitor.visit_neg(self, arg)
```

```
class Visitor(ABC):
    @abstractmethod
    def visit_var(self, exp, arg):
        pass
    @abstractmethod
    def visit_bln(self, exp, arg):
        pass
    @abstractmethod
    def visit_num(self, exp, arg):
        pass
    ...
```

```
class EvalVisitor(Visitor):
    ...
    def visit_neg(self, exp, env):
        return -exp.exp.accept(self, env)
    ...
```

```
class UseDefVisitor(Visitor):
    ...
    def visit_neg(self, exp, env):
        return exp.exp.accept(self, env)
    ...
```

O padrão visitor envolve duas partes: implementar o método `accept` sobre casa 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 mensage 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, substituta `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

```python
import sys
from Expression import *
from Visitor 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.
    """
    text = sys.stdin.read()
    (option, rest) = text.split(maxsplit=1)
    lexer = Lexer(rest)
    parser = Parser(lexer.tokens())
    exp = parser.parse()
    if option == 'eval':
        visitor = EvalVisitor()
        print(f"Value is {exp.accept(visitor, {})}")
    elif option == 'usedef':
        visitor = UseDefVisitor()
        print(f"Are there undefs? {len(exp.accept(visitor, set())) > 0}")
    elif option == 'safe_eval':
        safe_eval(exp)
    else:
        sys.exit(f"Invalid option = {option}")
```

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
```

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
```

Expression.py

```
from abc import ABC, abstractmethod
from Visitor import *

class Expression(ABC):
    @abstractmethod
    def accept(self, visitor, arg):
        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 accept(self, visitor, arg):
        """
        Example:
        >>> e = Var('var')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'var': 42})
        42

        >>> e = Var('v42')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'v42': True, 'v31': 5})
        True
        """
        return visitor.visit_var(self, arg)

class Bln(Expression):
    """
    This class represents expressions that are boolean values. There are only
    two boolean values: true and false. The acceptuation of such an expression is
    the boolean itself.
    """
    def __init__(self, bln):
        self.bln = bln
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Bln(True)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_bln(self, arg)

class Num(Expression):
    """
    This class represents expressions that are numbers. The acceptuation of such
    an expression is the number itself.
    """
    def __init__(self, num):
        self.num = num
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Num(3)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        3
        """
        return visitor.visit_num(self, arg)

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 accept(self, visitor, arg):
        raise NotImplementedError

class Eql(BinaryExpression):
    """
    This class represents the equality between two expressions. The acceptuation
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Eql(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False

        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Eql(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_eql(self, arg)

```

```
class Add(BinaryExpression):
    """
    This class represents addition of two expressions. The acceptuation of such
    an expression is the addition of the two subexpression's values.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        """
        return visitor.visit_add(self, arg)

class Sub(BinaryExpression):
    """
    This class represents subtraction of two expressions. The acceptuation of such
    an expression is the subtraction of the two subexpression's values.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -1
        """
        return visitor.visit_sub(self, arg)

class Mul(BinaryExpression):
    """
    This class represents multiplication of two expressions. The acceptuation of
    such an expression is the product of the two subexpression's values.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        12
        """
        return visitor.visit_mul(self, arg)

class Div(BinaryExpression):
    """
    This class represents the integer division of two expressions. The
    acceptuation of such an expression is the integer quocient of the two
    subexpression's values.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        5
        """
        return visitor.visit_div(self, arg)

class Leq(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than-or-equal comparator. The acceptuation 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 accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        """
```

```python
        """
        return visitor.visit_leq(self, arg)

class Lth(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than comparison operator. The acceptuation 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 accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        """
        return visitor.visit_lth(self, arg)

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 accept(self, visitor, arg):
        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 accept(self, visitor, arg):
        """
        Example:
        >>> n = Num(3)
        >>> e = Neg(n)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -3
        >>> n = Num(0)
        >>> e = Neg(n)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        0
        """
        return visitor.visit_neg(self, arg)

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 accept(self, visitor, arg):
        """
        Example:
        >>> t = Bln(True)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        >>> t = Bln(False)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_not(self, arg)

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):
        self.identifier = identifier
        self.exp_def = exp_def
        self.exp_body = exp_body
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Let('v', Num(42), Var('v'))
        >>> ev = EvalVisitor()
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

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

        >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        1764
        """
        return visitor.visit_let(self, arg)
```

Visitor.py

```python
import sys
from abc import ABC, abstractmethod
from Expression import *

class Visitor(ABC):
    """
    The visitor pattern consists of two abstract classes: the Expression and the
    Visitor. The Expression class defines on method: 'accept(visitor, args)'.
    This method takes in an implementation of a visitor, and the arguments that
    are passed from expression to expression. The Visitor class defines one
    specific method for each subclass of Expression. Each instance of such a
    subclasse will invoke the right visiting method.
    """
    @abstractmethod
    def visit_var(self, exp, arg):
        pass

    @abstractmethod
    def visit_bln(self, exp, arg):
        pass

    @abstractmethod
    def visit_num(self, exp, arg):
        pass

    @abstractmethod
    def visit_eql(self, exp, arg):
        pass

    @abstractmethod
    def visit_add(self, exp, arg):
        pass

    @abstractmethod
    def visit_sub(self, exp, arg):
        pass

    @abstractmethod
    def visit_mul(self, exp, arg):
        pass

    @abstractmethod
    def visit_div(self, exp, arg):
        pass

    @abstractmethod
    def visit_leq(self, exp, arg):
        pass

    @abstractmethod
    def visit_lth(self, exp, arg):
        pass

    @abstractmethod
    def visit_neg(self, exp, arg):
        pass

    @abstractmethod
    def visit_not(self, exp, arg):
        pass

    @abstractmethod
    def visit_let(self, exp, arg):
        pass

class EvalVisitor(Visitor):
    """
    The EvalVisitor class evaluates logical and arithmetic expressions. The
    result of evaluating an expression is the value of that expression. The
    inherited attribute propagated throughout visits is the environment that
    associates the names of variables with values.

    Examples:
    >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
    >>> e1 = Not(Eql(e0, Num(1764)))
    >>> ev = EvalVisitor()
    >>> e1.accept(ev, {})
    False

    >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
    >>> e1 = Lth(e0, Var('x'))
    >>> ev = EvalVisitor()
    >>> e1.accept(ev, {'x': 41})
    True
    """
    def visit_var(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_bln(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_num(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_eql(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_add(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_sub(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_mul(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_div(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_leq(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_lth(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_neg(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_not(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_let(self, exp, env):
        # TODO: Implement this method!
        raise NotImplementedError

class UseDefVisitor(Visitor):
    """
    The UseDefVisitor class reports the use of undefined variables. It takes
    as input an environment of defined variables, and produces, as output,
    the set of all the variables that are used without being defined.

    Examples:
    >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
    >>> e1 = Not(Eql(e0, Num(1764)))
    >>> ev = UseDefVisitor()
    >>> len(e1.accept(ev, set()))
    0

    >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
    >>> e1 = Lth(e0, Var('x'))
    >>> ev = UseDefVisitor()
    >>> len(e1.accept(ev, set()))
    1

    >>> e = Let('v', Add(Num(40), Var('v')), Sub(Var('v'), Num(2)))
    >>> ev = UseDefVisitor()
    >>> len(e.accept(ev, set()))
    1

    >>> e1 = Let('v', Add(Num(40), Var('v')), Sub(Var('v'), Num(2)))
    >>> e0 = Let('v', Num(3), e1)
    >>> ev = UseDefVisitor()
    >>> len(e0.accept(ev, set()))
    0
    """
    # TODO: Implement all the 13 methods of the visitor.

def safe_eval(exp):
    """
    This method applies one simple semantic analysis onto an expression, before
    evaluating it: it checks if the expression contains free variables, there
    is, variables used without being defined.

    Example:
    >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
    >>> e1 = Not(Eql(e0, Num(1764)))
    >>> safe_eval(e1)
    Value is False

    >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
    >>> e1 = Lth(e0, Var('x'))
    >>> safe_eval(e1)
    Error: expression contains undefined variables.
    """
    # TODO: Implement this method!
    raise NotImplementedError

```

VPL