2025_2 - COMPILADORES - METATURMA

AV11 - Código para expressões aritméticas

Data de entrega: quarta, 29 Out 2025, 23:59
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py (Baixar)
Tipo de trabalho: Trabalho individual

O objetivo deste trabalho é compilar expressões lógicas e aritméticas para um subconjunto de instruções RISC-V. Por hora, iremos considerar uma máquina com somente nove instruções, a saber:

- `add rd, rs1, rs2`: rd = rs1 + rs2
- `addi rd, rs1, imm`: rd = rs1 + imm
- `mul rd, rs1, rs2`: rd = rs1 * rs2
- `sub rd, rs1, rs2`: rd = rs1 - rs2
- `xor rd, rs1, rs2`: rd = rs1 ^ rs2
- `xori rd, rs1, imm`: rd = rs1 ^ imm
- `div rd, rs1, rs2`: rd = rs1 // rs2 (signed integer division)
- `slt rd, rs1, rs2`: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
- `slti rd, rs1, imm`: rd = (rs1 < imm) ? 1 : 0

Este trabalho possui um emulador para estas instruções, implementado no arquivo `Asm.py`. Você não precisa alterar este arquivo para resolver este trabalho. Por outro lado, pode ser instrutivo ler o código do emulador. No arquivo há duas funções, `max` e `distance_with_acceleration` que mostram como criar novas instruções. Essas funções podem ser vistas nas Figuras 1 e 2 abaixo. Note que em RISC-V existe um registrador especial, chamado "`x0`", que sempre possui o valor zero. Você pode usar este registrador para "montar" instruções complexas a partir de instruções simples. Por exemplo, para definir a variável "`a`" com o valor 2, você pode usar a instrução de soma com valor imediato: `addi("a", "x0", 2)`.

```
def max(a, b):
    """
    This function computes the maximum between a and b.
        >>> max(2, 3)
        3
    """
    p = Program({}, [])
    p.set_val("rs1", a)
    p.set_val("rs2", b)
    p.add_inst(Slt("t0", "rs2", "rs1"))
    p.add_inst(Slt("t1", "rs1", "rs2"))
    p.add_inst(Mul("t0", "t0", "rs1"))
    p.add_inst(Mul("t1", "t1", "rs2"))
    p.add_inst(Add("rd", "t0", "t1"))
    p.eval()
    return p.get_val("rd")
```

Figura 1: Uma função que calcula o máximo valor entre dois números

```
def distance_with_acceleration(V, A, T):
    """
    This function computes the position of an accelerated object.
        >>> distance_with_acceleration(3, 4, 5)
        65
    """
    p = Program({"rs1": V, "rs2": A, "rs3": T, "x0": 0}, [])
    p.add_inst(Addi("two", "x0", 2))
    p.add_inst(Mul("t0", "rs1", "rs3"))
    p.add_inst(Mul("t1", "rs3", "rs3"))
    p.add_inst(Mul("t2", "rs2", "t1"))
    p.add_inst(Div("t2", "t2", "two"))
    p.add_inst(Add("rd", "t0", "t2"))
    p.eval()
    return p.get_val("rd")
```

Figura 2: Uma função que calcula a distância percorrida por um objeto sujeito a aceleração (A) constante, com velocidade inicial (V), dado um tempo (T).

Neste trabalho, você deverá gerar código para as seguintes expressões, que fazem parte do Laboratório 5 (*Visitors*):

1. Variáveis, como Var("x").
2. Booleanos, como Bln(True).
3. Números como Num(2).
4. Igualdade, como Eql(Exp0, Exp1).
5. Adição, como Add(Exp0, Exp1).
6. Multiplicação, como Mul(Exp0, Exp1).
7. Subtração, como Sub(Exp0, Exp1).
8. Divisão, como Div(Exp0, Exp1).
9. Comparações de menor ou igual, como Leq(Exp0, Exp1).

10. Comparações de menor, como Lth(Exp0, Exp1).
11. Inversão de sinal, como Neg(Exp).
12. Negação de booleano, como Not(Exp).
13. Criação de variáveis, como Let("x", Exp0, Exp1). **Importante**: você pode assumir que todas as variáveis definitas em um programa são diferentes.

Para gerar código, você deverá implementar um novo `Visitor`, chamado `GenVisitor`. Cada método visit recebe como entrada uma expressão e um "Programa", e retorna o nome da variável que irá armazenar o valor da expressão. Um Programa é uma instância da classe `Program`, que está definida no arquivo `Asm.py`. A tarefa de `GenVisitor` é inserir instruções no programa, o que pode ser feito via o método `Program.add_inst`. As Figuras 3 e 4 mostram dois exemplos de métodos `visit`. Note que algumas expressões, como `Add`, possuem instruções equivalentes em nossa máquina RISC-V reduzida. Porém, outras expressões, como `Not` e `Eql` não possuem instruções equivalentes. Você terá de resolver um pequeno quebra-cabeças, neste caso: "*Como implementar essas expressões com as instruções que estão disponíveis?*" Note que em nenhuma hipótese você deve modificar Asm.py para adicionar novas instruções. Você precisa usar somente as instruções que já estão definidas na máquina reduzida.

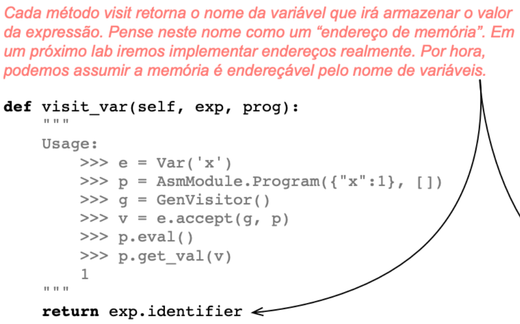

Figura 3: Implementação de geração de código para variáveis.

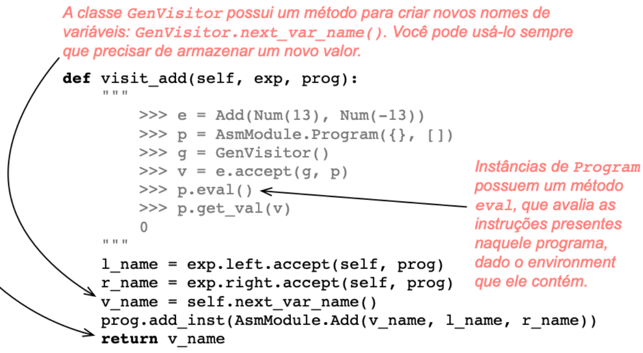

Figura 4: Implementação de geração de código para adição.

Observação sobre o nome de variáveis em blocos `let`

Conforme explicado anteriormente, neste exercício você pode assumir que blocos `let` sempre definitem variáveis com nomes diferentes. Em outras palavras, um programa como `let x <- 1 in let x <- 2 in x end + x end` não é válido. Um programa equivalente, válido, seria: `let x0 <- 1 in let x1 <- 2 in x1 end + x0 end`. Assim, você não precisa tratar situações em que variáveis são definidas com o mesmo nome. No próximo laboratório iremos implementar um `visitor` simples para fazer essa renomeação automaticamente.

Submetendo e Testando

Para completar este VPL, você deverá entregar seis arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Visitor.py`, `Asm.py` e `driver.py`. Você não deverá alterar `Asm.py`, `driver.py` ou `Expression.py`. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
python3 driver.py
2 + 3 # CTRL+D
5
```

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 `Visitor.py`, por exemplo). Caso você não gere mensagens de erro, então seu trabalho está (quase) completo!

Arquivos requeridos

driver.py

```
import sys
from Expression import *
from Visitor import *
from Lexer import Lexer
from Parser import Parser
import Asm as AsmModule

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()
    lexer = Lexer(text)
    parser = Parser(lexer.tokens())
    exp = parser.parse()
    prog = AsmModule.Program({}, [])
    gen = GenVisitor()
    var_answer = exp.accept(gen, prog)
    prog.eval()
    print(f"{prog.get_val(var_answer)}")
```

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
                and token.kind != TokenType.NLN
            ):
                yield token
            token = self.getToken()

    def getToken(self):
        """
        Return the next token.
        TODO: Implement this method (you can reuse Lab 5: Visitors)!
        """
        token = None
        return token
```

Parser.py

```python
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements the parser of arithmetic expressions. The same rules of
precedence and associativity from Lab 5: Visitors, apply.

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

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!
        # You can (and probably should!) modify this method.

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

        Examples:
        >>> parser = Parser([Token('123', TokenType.NUM)])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        123

        >>> parser = Parser([Token('True', TokenType.TRU)])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> parser = Parser([Token('False', TokenType.FLS)])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> tk0 = Token('~', TokenType.NEG)
        >>> tk1 = Token('123', TokenType.NUM)
        >>> parser = Parser([tk0, tk1])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        -123

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        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])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        -12

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('/', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        7

```

```
        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('+', TokenType.ADD)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        7

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('-', TokenType.SUB)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        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])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        14

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('==', TokenType.EQL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('<=', TokenType.LEQ)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> tk0 = Token('4', TokenType.NUM)
        >>> tk1 = Token('<', TokenType.LTH)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> 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])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> 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])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
```

```
        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])
        >>> g = GenVisitor()
        >>> p = AsmModule.Program({}, [])
        >>> exp = parser.parse()
        >>> v = exp.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        42
        """

        # TODO: implement this method.
        return None
```

Expression.py

```python
"""
This file implements the data structures that represent Expressions. You don't
need to modify it for this assignment.
"""

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):
        """
        Variables don't need to be implemented for this exercise.
        """
        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):
        """
        booleans don't need to be implemented for this exercise.
        """
        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):
        """
        Equality doesn't need to be implemented for this exercise.
        """
        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):
        """
        Comparisons don't need to be implemented for this exercise.
        """
        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):
        """
        Comparisons don't need to be implemented for this exercise.
        """
```

```
        """
        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):
        """
        No need to implement negation for this exercise, for we don't even have
        booleans at this point.
        """
        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):
        """
        We don't have bindings at this point. So, nothing to be done here, for
        this exercise.
        """
        return visitor.visit_let(self, arg)
```

Visitor.py

```
import sys
from abc import ABC, abstractmethod
from Expression import *
import Asm as AsmModule


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 GenVisitor(Visitor):
    """
    The GenVisitor class compiles arithmetic expressions into a low-level
    language.
    """

    def __init__(self):
        self.next_var_counter = 0

    def next_var_name(self):
        self.next_var_counter += 1
        return f"v{self.next_var_counter}"

    def visit_var(self, exp, prog):
        """
        Usage:
            >>> e = Var('x')
            >>> p = AsmModule.Program({"x":1}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            1
        """
        return exp.identifier

    def visit_bln(self, exp, env):
        """
        Usage:
            >>> e = Bln(True)
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
```

```
                >>> p.get_val(v)
                1

                >>> e = Bln(False)
                >>> p = AsmModule.Program({}, [])
                >>> g = GenVisitor()
                >>> v = e.accept(g, p)
                >>> p.eval()
                >>> p.get_val(v)
                0
            """
            # TODO: Implement this method.
            raise NotImplementedError

        def visit_num(self, exp, prog):
            """
            Usage:
                >>> e = Num(13)
                >>> p = AsmModule.Program({}, [])
                >>> g = GenVisitor()
                >>> v = e.accept(g, p)
                >>> p.eval()
                >>> p.get_val(v)
                13
            """
            # TODO: Implement this method.
            raise NotImplementedError

        def visit_eql(self, exp, prog):
            """
            >>> e = Eql(Num(13), Num(13))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            1

            >>> e = Eql(Num(13), Num(10))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            0

            >>> e = Eql(Num(-1), Num(1))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            0
            """
            # TODO: Implement this method.
            raise NotImplementedError

        def visit_add(self, exp, prog):
            """
            >>> e = Add(Num(13), Num(-13))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            0

            >>> e = Add(Num(13), Num(10))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            23
            """
            # TODO: Implement this method (see the example in the lab's page).
            raise NotImplementedError

        def visit_sub(self, exp, prog):
            """
            >>> e = Sub(Num(13), Num(-13))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            26

            >>> e = Sub(Num(13), Num(10))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            3
            """
            # TODO: Implement this method.
            raise NotImplementedError

        def visit_mul(self, exp, prog):
            """
            >>> e = Mul(Num(13), Num(2))
            >>> p = AsmModule.Program({}, [])
```

```
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        26

        >>> e = Mul(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        130
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_div(self, exp, prog):
        """
        >>> e = Div(Num(13), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        6

        >>> e = Div(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_leq(self, exp, prog):
        """
        >>> e = Leq(Num(3), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Leq(Num(3), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(2), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-3), Num(-2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-3), Num(-3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-2), Num(-3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_lth(self, exp, prog):
        """
        >>> e = Lth(Num(3), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Lth(Num(3), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
```

```
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Lth(Num(2), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_neg(self, exp, prog):
        """
        >>> e = Neg(Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        -3

        >>> e = Neg(Num(0))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Neg(Num(-3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_not(self, exp, prog):
        """
        >>> e = Not(Bln(True))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Not(Bln(False))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Not(Num(0))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Not(Num(-2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Not(Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_let(self, exp, prog):
        """
        Usage:
            >>> e = Let('v', Not(Bln(False)), Var('v'))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            1

            >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
            >>> p = AsmModule.Program({}, [])
```

```
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            5

            >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
            >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e1.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            50
        """
        # TODO: Implement this method.
        raise NotImplementedError
```

Asm.py

```python
"""
This file contains the implementation of a simple interpreter of low-level
instructions. The interpreter takes a program, represented as an array of
instructions, plus an environment, which is a map that associates variables with
values. The following instructions are recognized:

    * add rd, rs1, rs2: rd = rs1 + rs2
    * addi rd, rs1, imm: rd = rs1 + imm
    * mul rd, rs1, rs2: rd = rs1 * rs2
    * sub rd, rs1, rs2: rd = rs1 - rs2
    * xor rd, rs1, rs2: rd = rs1 ^ rs2
    * xori rd, rs1, imm: rd = rs1 ^ imm
    * div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
    * slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
    * slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0

This file uses doctests all over. To test it, just run python 3 as follows:
"python3 -m doctest Asm.py". The program uses syntax that is excluive of
Python 3. It will not work with standard Python 2.
"""

import sys
from collections import deque
from abc import ABC, abstractmethod


class Program:
    """
    The 'Program' is a list of instructions plus an environment that associates
    names with values, plus a program counter, which marks the next instruction
    that must be executed. The environment contains a special variable x0,
    which always contains the value zero.
    """

    def __init__(self, env, insts):
        self.__env = env
        self.__insts = insts
        self.pc = 0
        self.__env["x0"] = 0

    def get_inst(self):
        if self.pc >= 0 and self.pc < len(self.__insts):
            inst = self.__insts[self.pc]
            self.pc += 1
            return inst
        else:
            return None

    def add_inst(self, inst):
        self.__insts.append(inst)

    def set_pc(self, pc):
        self.pc = pc

    def set_val(self, name, value):
        self.__env[name] = value

    def get_val(self, name):
        """
        The register x0 always contains the value zero:

        >>> p = Program({}, [])
        >>> p.get_val("x0")
        0
        """
        if name in self.__env:
            return self.__env[name]
        else:
            sys.exit("Def error")

    def print_env(self):
        for name, val in sorted(self.__env.items()):
            print(f"{name}: {val}")

    def print_insts(self):
        for inst in self.__insts:
            print(inst)

    def eval(self):
        """
        This function evaluates a program until there is no more instructions to
        evaluate.

        Example:
            >>> insts = [Add("x0", "b0", "b1"), Sub("x1", "x0", "b2")]
            >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            b2: 4
            x0: 5
            x1: 1
        """
        inst = self.get_inst()
        while inst:
            inst.eval(self)
            inst = self.get_inst()


def max(a, b):
    """
    This example computes the maximum between a and b.
```

```

    Example:
        >>> max(2, 3)
        3

        >>> max(3, 2)
        3

        >>> max(-3, -2)
        -2

        >>> max(-2, -3)
        -2
    """
    p = Program({}, [])
    p.set_val("rs1", a)
    p.set_val("rs2", b)
    p.add_inst(Slt("t0", "rs2", "rs1"))
    p.add_inst(Slt("t1", "rs1", "rs2"))
    p.add_inst(Mul("t0", "t0", "rs1"))
    p.add_inst(Mul("t1", "t1", "rs2"))
    p.add_inst(Add("rd", "t0", "t1"))
    p.eval()
    return p.get_val("rd")


def distance_with_acceleration(V, A, T):
    """
    This example computes the position of an object, given its velocity (V),
    its acceleration (A) and the time (T), assuming that it starts at position
    zero, using the formula D = V*T + (A*T^2)/2.

    Example:
        >>> distance_with_acceleration(3, 4, 5)
        65
    """
    p = Program({}, [])
    p.set_val("rs1", V)
    p.set_val("rs2", A)
    p.set_val("rs3", T)
    p.add_inst(Addi("two", "x0", 2))
    p.add_inst(Mul("t0", "rs1", "rs3"))
    p.add_inst(Mul("t1", "rs3", "rs3"))
    p.add_inst(Mul("t2", "rs2", "t1"))
    p.add_inst(Div("t2", "t2", "two"))
    p.add_inst(Add("rd", "t0", "t2"))
    p.eval()
    return p.get_val("rd")


class Inst(ABC):
    """
    The representation of instructions. Every instruction refers to a program
    during its evaluation.
    """

    def __init__(self):
        pass

    @abstractmethod
    def get_opcode(self):
        raise NotImplementedError

    @abstractmethod
    def eval(self, prog):
        raise NotImplementedError


class BinOp(Inst):
    """
    The general class of binary instructions. These instructions define a
    value, and use two values.
    """

    def __init__(self, rd, rs1, rs2):
        assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(rs2, str)
        self.rd = rd
        self.rs1 = rs1
        self.rs2 = rs2

    def __str__(self):
        op = self.get_opcode()
        return f"{self.rd} = {op} {self.rs1} {self.rs2}"


class BinOpImm(Inst):
    """
    The general class of binary instructions where the second operand is an
    integer constant. These instructions define a value, and use one variable
    and one immediate constant.
    """

    def __init__(self, rd, rs1, imm):
        assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
        self.rd = rd
        self.rs1 = rs1
        self.imm = imm

    def __str__(self):
        op = self.get_opcode()
        return f"{self.rd} = {op} {self.rs1} {self.imm}"

```

```
class Add(BinOp):
    """
    add rd, rs1, rs2: rd = rs1 + rs2

    Example:
        >>> i = Add("a", "b0", "b1")
        >>> str(i)
        'a = add b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        5
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 + rs2)

    def get_opcode(self):
        return "add"


class Addi(BinOpImm):
    """
    addi rd, rs1, imm: rd = rs1 + imm

    Example:
        >>> i = Addi("a", "b0", 1)
        >>> str(i)
        'a = addi b0 1'

        >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        5
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        prog.set_val(self.rd, rs1 + self.imm)

    def get_opcode(self):
        return "addi"


class Mul(BinOp):
    """
    mul rd, rs1, rs2: rd = rs1 * rs2

    Example:
        >>> i = Mul("a", "b0", "b1")
        >>> str(i)
        'a = mul b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        6
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 * rs2)

    def get_opcode(self):
        return "mul"


class Sub(BinOp):
    """
    sub rd, rs1, rs2: rd = rs1 - rs2

    Example:
        >>> i = Sub("a", "b0", "b1")
        >>> str(i)
        'a = sub b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        -1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 - rs2)

    def get_opcode(self):
        return "sub"


class Xor(BinOp):
    """
    xor rd, rs1, rs2: rd = rs1 ^ rs2

    Example:
        >>> i = Xor("a", "b0", "b1")
        >>> str(i)
        'a = xor b0 b1'
```

```
        >>> p = Program(env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 ^ rs2)

    def get_opcode(self):
        return "xor"


class Xori(BinOpImm):
    """
    xori rd, rs1, imm: rd = rs1 ^ imm

    Example:
        >>> i = Xori("a", "b0", 10)
        >>> str(i)
        'a = xori b0 10'

        >>> p = Program(env={"b0":2}, insts=[Xori("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        prog.set_val(self.rd, rs1 ^ self.imm)

    def get_opcode(self):
        return "xori"


class Div(BinOp):
    """
    div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
    Notice that RISC-V does not have an instruction exactly like this one.
    The div operator works on floating-point numbers; not on integers.

    Example:
        >>> i = Div("a", "b0", "b1")
        >>> str(i)
        'a = div b0 b1'

        >>> p = Program(env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        2
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 // rs2)

    def get_opcode(self):
        return "div"


class Slt(BinOp):
    """
    slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)

    Example:
        >>> i = Slt("a", "b0", "b1")
        >>> str(i)
        'a = slt b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        1

        >>> p = Program(env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        0

        >>> p = Program(env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        0
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, 1 if rs1 < rs2 else 0)

    def get_opcode(self):
        return "slt"


class Slti(BinOpImm):
    """
    slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
    (signed comparison with immediate)

```

```
    Example:
        >>> i = Slti("a", "b0", 0)
        >>> str(i)
        'a = slti b0 0'

        >>> p = Program(env={"b0":2}, insts=[Slti("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        1

        >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        0

        >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 2)])
        >>> p.eval()
        >>> p.get_val("a")
        0
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        prog.set_val(self.rd, 1 if rs1 < self.imm else 0)

    def get_opcode(self):
        return "slti"
```

VPL