2025_2 - COMPILADORES - METATURMA

AV13 - Fluxo de controle

Data de entrega: quarta, 12 Nov 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 prático é gerar código para expressões que envolvem fluxo de controle variável. O fluxo de controle variável é caracterizado pela presença de "branches" no programa: instruções que podem alterar o contador de programa da arquitetura. Neste exercício, usaremos um subconjunto de instruções RISC-V que inclui todas as instruções vistas no laboratório anterior, além de duas instruções de desvio:

- `beq rs1 rs2 lab`: altera PC para lab caso rs1 == rs2; doutro modo, PC passa a ser PC + 1 .
- `jal rd lab`: salva o valor de PC+1 no registrador rd, e altera PC para lab. Caso rd seja `x0`, então esta instrução é equivalente a um "*goto*" (desvio incondicional), pois escritas em `x0` não têm efeito.

As instruções de desvio têm a semântica de alterar o valor do contador de programas. Note que adotamos uma semântica um pouco diferente da semântica das instruções originais `beq` e `jal` de RISC-V, para simplificar o trabalho. Em RISC-V, essas instruções fazem o desvio relativo ao PC. Em outras palavras, a instrução `beq`, por exemplo, soma `lab` a pc, caso `rs1 = rs2`. Neste trabalho estamos considerando somente desvios absolutos. Assim, `beq` simplesmente substitui o valor de pc por `lab`, caso a condição de desvio seja verdade.

Este VPL conta com uma nova implementação de `Asm.py`, que incorpora as duas instruções acima. Esta implementação também contém novos métodos na definição da classe Program, a saber:

- `get_number_of_instructions`: informa o número de instruções armazenadas dentro de um programa. Este método pode ser usado para calcular o rótulo de branches.
- `get_pc`: informa o valor atual do contador de programas.
- `set_pc`: define um novo valor para o contador de programas.

Dentre estes métodos os dois últimos são usados pela implementação de `beq` e `jal`. Você não precisa se preocupar com eles. O primeiro método, contudo, `get_number_of_instructions`, será útil para determinar o alvo de branches. Por exemplo, para implementar um desvio condicional para a frente, você pode usar o seguinte padrão:

```
# Avalia uma expressão e coloca o valor em v
  v = exp.accept(gen, prog)
# Se v for zero, então pula (alvo a determinar)
  beq = AsmModule.Beq(v, "x0")
  prog.add_inst(beq)
# Adicione mais algumas instruções ao programa:
  prog.add_inst(AsmModule.sub(...))
  prog.add_inst(AsmModule.add(...))
  ...
  prog.add_inst(AsmModule.mul(...))
  prog.add_inst(AsmModule.etc(...))
# Vamos pular aqui! Encontre o alvo:
  beq.set_target(prog.get_number_of_instructions())
```

Para resolver este exercício, você deverá aumentar as implementações de `RenameVisitor` e de `GenVisitor` (ambas em `Visitor.py`) para que elas lidem com três novas expressões:

`exp1 and exp2`: Caso exp1 seja false, então false, senão exp2
`exp1 or exp2`: Caso exp1 seja true, então true, senão exp2
`if exp0 then exp1 else exp2`: Caso exp0 seja true, então exp1, senão exp2.

Note que as três expressões envolvem fluxo de controle variável. As duas primeiras expressões, `and` e `or`, implementam a semântica de [curto-circuito](). Assim, uma expressão como `(false and ((1 / 0) < 0))` é `false` (no caso, zero). A divisão por zero nunca será avaliada. Veja que os valores booleanos, ao nível de código de montagem, são representados como os inteiros zero (para `false`) e um (para `true`). Você não deve assumir que cada variável possui um nome único no programa. Assim, o *driver* deste exercício usa `RenameVisitor` para renomear variáveis, a fim de que o programa esteja em formato de [atribuição estática única]().

Submetendo e Testando

Este VPL deve ser construído sobre o VPL 12. 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
true and false # CTRL+D
0
```

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

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

def rename_variables(exp):
    """
    Esta funcao invoca o renomeador de variaveis. Ela deve ser usada antes do
    inicio da fase de geracao de codigo.
    """
    ren = RenameVisitor()
    exp.accept(ren, {})
    return exp

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 = rename_variables(parser.parse())
    prog = AsmModule.Program({}, [])
    gen = GenVisitor()
    var_answer = exp.accept(gen, prog)
    prog.eval()
    print(f"Answer: {prog.get_val(var_answer)}")
```

Lexer.py

```
import sys
import enum


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


class TokenType(enum.Enum):
    """
    These are the possible tokens. You don't need to change this class at all.
    """
    EOF = -1  # End of file
    NLN = 0   # New line
    WSP = 1   # White Space
    COM = 2   # Comment
    NUM = 3   # Number (integers)
    STR = 4   # Strings
    TRU = 5   # The constant true
    FLS = 6   # The constant false
    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 # x = y
    ADD = 202 # x + y
    SUB = 203 # x - y
    MUL = 204 # x * y
    DIV = 205 # x / y
    LEQ = 206 # x <= y
    LTH = 207 # x < y
    NEG = 208 # ~x
    NOT = 209 # not x
    LPR = 210 # (
    RPR = 211 # )
    ASN = 212 # The assignment '<-' operator
    ORX = 213 # x or y
    AND = 214 # x and y
    IFX = 215 # The 'if' of a conditional expression
    THN = 216 # The 'then' of a conditional expression
    ELS = 217 # The 'else' of a conditional expression


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

```
import sys

from Expression import *
from Lexer import Token, TokenType

"""
Precedence table:
    1: not ~ ()
    2: *   /
    3: +   -
    4: <   <=   >=   >
    5: =
    6: and
    7: or
    8: if-then-else

Notice that not 2 < 3 must be a type error, as we are trying to apply a boolean
operation (not) onto a number. However, in assembly code this program works,
because not 2 is 0. The bottom line is: don't worry about programs like this
one: the would have been ruled out by type verification anyway.

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)])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        123

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

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

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

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, 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()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        -12

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('/', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        7

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('+', TokenType.ADD)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        7

```

```
    >>> tk0 = Token('30', TokenType.NUM)
    >>> tk1 = Token('-', TokenType.SUB)
    >>> tk2 = Token('4', TokenType.NUM)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, 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()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    14

    >>> tk0 = Token('4', TokenType.NUM)
    >>> tk1 = Token('==', TokenType.EQL)
    >>> tk2 = Token('4', TokenType.NUM)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    True

    >>> tk0 = Token('4', TokenType.NUM)
    >>> tk1 = Token('<=', TokenType.LEQ)
    >>> tk2 = Token('4', TokenType.NUM)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    True

    >>> tk0 = Token('4', TokenType.NUM)
    >>> tk1 = Token('<', TokenType.LTH)
    >>> tk2 = Token('4', TokenType.NUM)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    False

    >>> tk0 = Token('not', TokenType.NOT)
    >>> tk1 = Token('(', TokenType.LPR)
    >>> tk2 = Token('4', TokenType.NUM)
    >>> tk3 = Token('<', TokenType.LTH)
    >>> tk4 = Token('4', TokenType.NUM)
    >>> tk5 = Token(')', TokenType.RPR)
    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    True

    >>> tk0 = Token('true', TokenType.TRU)
    >>> tk1 = Token('or', TokenType.ORX)
    >>> tk2 = Token('false', TokenType.FLS)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    True

    >>> tk0 = Token('true', TokenType.TRU)
    >>> tk1 = Token('and', TokenType.AND)
    >>> tk2 = Token('false', TokenType.FLS)
    >>> parser = Parser([tk0, tk1, tk2])
    >>> exp = parser.parse()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, None)
    False

    >>> 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()
    >>> ev = EvalVisitor()
    >>> exp.accept(ev, {})
    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()
```

```
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, {})
        42

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('2', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('3', TokenType.NUM)
        >>> tk4 = Token('then', TokenType.THN)
        >>> tk5 = Token('1', TokenType.NUM)
        >>> tk6 = Token('else', TokenType.ELS)
        >>> tk7 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        1

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('false', TokenType.FLS)
        >>> tk2 = Token('then', TokenType.THN)
        >>> tk3 = Token('1', TokenType.NUM)
        >>> tk4 = Token('else', TokenType.ELS)
        >>> tk5 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        2
        """
        # TODO: implement this method.
        return None
```

Expression.py

```python
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):
        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):
        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):
        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):
        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):
        return visitor.visit_add(self, arg)

class And(BinaryExpression):
    """
    This class represents the logical disjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical AND of the two
    subexpression's values.
    """
    def accept(self, visitor, arg):
        return visitor.visit_and(self, arg)

class Or(BinaryExpression):
    """
    This class represents the logical conjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical OR of the two
    subexpression's values.
    """
    def accept(self, visitor, arg):
        return visitor.visit_or(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):
        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):
        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):
        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):
        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):
        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):
        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):
        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):
        return visitor.visit_let(self, arg)

class IfThenElse(Expression):
    """
    This class represents a conditional expression. The semantics an expression
    such as 'if B then E0 else E1' is as follows:
    1. Evaluate B. Call the result ValueB.
    2. If ValueB is True, then evalute E0 and return the result.
    3. If ValueB is False, then evaluate E1 and return the result.
    Notice that we only evaluate one of the two sub-expressions, not both. Thus,
    "if True then 0 else 1 div 0" will return 0 indeed.
    """
    def __init__(self, cond, e0, e1):
        self.cond = cond
        self.e0 = e0
        self.e1 = e1
    def accept(self, visitor, arg):
        return visitor.visit_ifThenElse(self, arg)
```

Visitor.py

```python
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_and(self, exp, arg):
        pass

    @abstractmethod
    def visit_or(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

    @abstractmethod
    def visit_ifThenElse(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):
        """
        You can use this method to get fresh variable names.
        """
        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
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_bln(self, exp, prog):
        """
        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_and(self, exp, prog):
        """
        >>> e = And(Bln(True), Bln(True))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = And(Bln(False), Bln(True))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = And(Bln(True), Bln(False))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = And(Bln(False), Bln(False))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
```

```
        0

        >>> e = And(Bln(False), Div(Num(3), Num(0)))
        >>> 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_or(self, exp, prog):
        """
        >>> e = Or(Bln(True), Bln(True))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Or(Bln(False), Bln(True))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Or(Bln(True), Bln(False))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

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

        >>> e = Or(Bln(True), Div(Num(3), Num(0)))
        >>> 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_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.
        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({}, [])
```

```
        >>> 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()
        >>> 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

    def visit_ifThenElse(self, exp, prog):
        """
        >>> e = IfThenElse(Bln(True), Num(3), Num(5))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3

        >>> e = IfThenElse(Bln(False), Num(3), Num(5))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        5

        >>> e = IfThenElse(And(Bln(True), Bln(True)), Num(3), Num(5))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3

        >>> e0 = Mul(Num(2), Add(Num(3), Num(4)))
        >>> e1 = IfThenElse(And(Bln(True), Bln(False)), Num(3), e0)
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e1.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        14

        >>> e0 = Div(Num(2), Num(0))
        >>> e1 = IfThenElse(Bln(True), Num(3), e0)
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e1.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3

        >>> e0 = Div(Num(2), Num(0))
        >>> e1 = IfThenElse(Bln(False), e0, Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e1.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3
        """
        # TODO: Implement this method.
        raise NotImplementedError


class RenameVisitor(ABC):
    """
    This visitor traverses the AST of a program, renaming variables to ensure
    that they all have different names.

    Usage:
        >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
        >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
        >>> e0.identifier == e1.identifier
        True

        >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
        >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
        >>> r = RenameVisitor()
        >>> e1.accept(r, {})
        >>> e0.identifier == e1.identifier
        False

        >>> x0 = Var('x')
        >>> x1 = Var('x')
        >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
        >>> e1 = Let('x', e0, Mul(x1, Num(10)))
        >>> x0.identifier == x1.identifier
        True

        >>> x0 = Var('x')
        >>> x1 = Var('x')
```

```
        >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
        >>> e1 = Let('x', e0, Mul(x1, Num(10)))
        >>> r = RenameVisitor()
        >>> e1.accept(r, {})
        >>> x0.identifier == x1.identifier
        False
    """

    def __init__(self):
        # TODO: implement something here.
        pass

    def visit_var(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_bln(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_num(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_eql(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_and(self, exp, name_map):
        """
        Example:
            >>> y0 = Var('x')
            >>> y1 = Var('x')
            >>> x0 = And(Lth(y0, Num(2)), Leq(Num(2), y1))
            >>> x1 = Var('x')
            >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
            >>> e1 = Let('x', e0, Mul(x1, Num(10)))
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> y0.identifier == y1.identifier
            True

            >>> y0 = Var('x')
            >>> y1 = Var('x')
            >>> x0 = And(Lth(y0, Num(2)), Leq(Num(2), y1))
            >>> x1 = Var('x')
            >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
            >>> e1 = Let('x', e0, Mul(x1, Num(10)))
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> y0.identifier == x1.identifier
            False
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_or(self, exp, name_map):
        """
        Example:
            >>> y0 = Var('x')
            >>> y1 = Var('x')
            >>> x0 = Or(Lth(y0, Num(2)), Leq(Num(2), y1))
            >>> x1 = Var('x')
            >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
            >>> e1 = Let('x', e0, Mul(x1, Num(10)))
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> y0.identifier == y1.identifier
            True

            >>> y0 = Var('x')
            >>> y1 = Var('x')
            >>> x0 = Or(Lth(y0, Num(2)), Leq(Num(2), y1))
            >>> x1 = Var('x')
            >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
            >>> e1 = Let('x', e0, Mul(x1, Num(10)))
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> y0.identifier == x1.identifier
            False
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_add(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_sub(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_mul(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_div(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_leq(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError
```

```
    def visit_lth(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_neg(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_not(self, exp, name_map):
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_ifThenElse(self, exp, name_map):
        """
        Example:
            >>> x0 = Var('x')
            >>> x1 = Var('x')
            >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
            >>> e1 = Let('x', Num(3), e0)
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> x0.identifier == x1.identifier
            True

            >>> x0 = Var('x')
            >>> x1 = Var('x')
            >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
            >>> e1 = Let('x', Num(3), e0)
            >>> e2 = Let('x', e1, Num(3))
            >>> r = RenameVisitor()
            >>> e1.accept(r, {})
            >>> e2.identifier != x1.identifier == e1.identifier
            True
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_let(self, exp, name_map):
        # 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
    * beq rs1, rs2, lab: pc = lab if rs1 == rs2 else pc + 1
    * jal rd, lab: rd = pc + 1 and pc = lab

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 get_number_of_instructions(self):
        return len(self.__insts)

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

    def get_pc(self):
        return self.pc

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

    def set_val(self, name, value):
        if name != "x0": # Can't change x0, which is always zero.
            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):
        counter = 0
        for inst in self.__insts:
            print("%03d: %s" % (counter, str(inst)))
            counter += 1
        print("%03d: %s" % (counter, "END"))

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

        Example:
            >>> insts = [Add("t0", "b0", "b1"), Sub("x1", "t0", "b2")]
            >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            b2: 4
```

```
            t0: 5
            x0: 0
            x1: 1

       Notice that it is not possible to change 'x0':
            >>> insts = [Add("x0", "b0", "b1")]
            >>> p = Program({"b0":2, "b1":3}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            x0: 0
        """
        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 BranchOp(Inst):
    """
    The general class of branching instructions. These instructions can change
    the control flow of a program. Normally, the next instruction is given by
    pc + 1. A branch might change pc to point out to a different label..
    """

    def set_target(self, lab):
        assert(isinstance(lab, int))
        self.lab = lab


class Beq(BranchOp):
    """
```

```
    """
    beq rs1, rs2, lab:
    Jumps to label lab if the value in rs1 is equal to the value in rs2.
    """

    def __init__(self, rs1, rs2, lab=None):
        assert isinstance(rs1, str) and isinstance(rs2, str)
        self.rs1 = rs1
        self.rs2 = rs2
        if lab != None:
            assert isinstance(lab, int)
        self.lab = lab

    def get_opcode(self):
        return "beq"

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

    def eval(self, prog):
        if prog.get_val(self.rs1) == prog.get_val(self.rs2):
            prog.set_pc(self.lab)


class Jal(BranchOp):
    """
    jal rd lab:
    Stores the return address (PC+1) on register rd, then jumps to label lab.
    If rd is x0, then it does not write on the register. In this case, notice
    that `jal x0 lab` is equivalent to an unconditional jump to `lab`.
    """

    def __init__(self, rd, lab=None):
        assert isinstance(rd, str)
        self.rd = rd
        if lab != None:
            assert isinstance(lab, int)
        self.lab = lab

    def get_opcode(self):
        return "jal"

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

    def eval(self, prog):
        if self.rd != "x0":
            self.rd = prog.get_pc + 1
        prog.set_pc(self.lab)


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