2025_2 - COMPILADORES - METATURMA

AV14 - Geração de código para funções

Data de entrega: sexta, 28 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 contém funções anônimas. Para tanto, além das instruções utilizadas nos exercícios anteriores, iremos usar mais três instruções:

- `jalr rd, rs1, offset`: salva o valor de PC+1 no registrador rd, e altera PC para offset + o valor em rs1. Caso rd seja `x0`, então esta instrução é equivalente a um "*goto indireto*" (desvio incondicional com valor do desvio lido a partir de registrador), pois escritas em `x0` não têm efeito.
- `sw reg, rs1(offset)`: salva no endereço de memória rs1 + offset o valor que estava no registrador reg.
- `ld reg, rs1(offset)`: carrega em reg o valor que estava na posição de memória rs1 + offset.

Note que agora nosso programa possui *memória*. Pense na memória como um arranjo em que são armazenados inteiros de 32 bits. Esses valores podem ser lidos via instruções `sw`, e podem ser escritos via instruções `ld`. Programas agora também possuem um registrador especial, `sp`, que é inicializado com o tamanho da memória disponível. Este registrador, o Stack Pointer, pode ser usado para implementar uma pilha em que são armazenados os registros de ativação de funções. Para adicionar instruções de acesso à memória ao programa, use a API abaixo:

- `prog.add_inst(Lw("rs1", offset, "reg"))`: Implementa a instrução `ld reg, rs1(offset)`.
  `prog.add_inst(Sw("rs1", offset, "reg"))`:Implementa a instrução `sw reg, rs1(offset)`.

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

- `e0 e1`: Invoca a função "e0" com o valor do parâmetro real "e1".
- `(fn v => e)`: Cria uma função com corpo "e" e parâmetro real "v"

As Figuras 1-4 abaixo ilustram diferentes aspectos da implementação de funções.

Figura 1: Funções precisam implementar escopo léxico! Assim, o valor esperado para a aplicação abaixo é 6. O valor sete, caso fosse impresso, seria *errado*!

```
let
  one <- 1
in
  let
    f <- fn x => x + one
  in
    let
      one <- 2
    in
      f 5
    end
  end
end

6
```

Figura 2: Funções podem retornar outras funções. Assim, o valor esperado da aplicação abaixo é sete.

```
let
  funToAddX <- fn x =>
                 let
                   addX <- fn y => y + x
                 in
                   addX
                 end
in
  let
    f <- funToAddX 3
  in
    f 4
  end
end

7
```

Figura 2: As regras de precedência continuam as mesmas. A aplicação de funções sempre possui a precedência mais alta:

```
let
  f <- fn a => fn b => fn c => a + b + c
in
  f 2 3 4 = (((f 2) 3) 4) and f 2 3 4 = 9
end

1
```

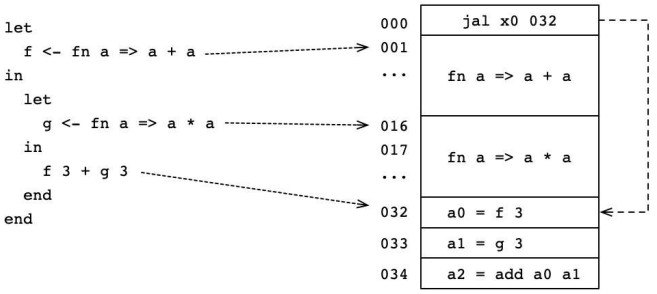
Figura 4: Funções podem ser chamadas diretamente a partir de sua definição, ou podem ser associadas a uma variável e posteriormente invocadas:

```
(fn x => x * x) 3

9
```

```
let f <- fn x => x * x in f 3 end

9
```

Uma particularidade deste novo exercício é que você precisará gerar código que encontre o correto ponto de boot do programa. Todos os programas são avaliados a partir da primeira instrução, porém, talvez a primeira instrução de seu código não seja o ponto em que o programa começa, agora que temos um programa formado por várias funções. A figura abaixo ilustra esse desafio:

Figura 5: Um dos desafios deste exercício é encontrar o ponto de partida do programa.



Nos exercícios anteriores não havia funções: todos os programas começavam a executar a partir da primeira instrução. Nosso emulador continua executando programas a partir da primeira instrução. Assim, é necessário que esta primeira instrução realmente seja o ponto de partida do programa.

Neste exemplo, não faz sentido começar a executar o programa a partir do código gerado para representar "`fn a => a + a`", pois este código somente define a função, mas não a invoca. O ponto de partida do programa é a invocação que acontece via a chamada "`f 3 + g 3`". Neste caso, deveríamos começar a execução do programa com a invocação "`f 3`".

Assim, de alguma forma, você deverá pensar em uma estratégia de geração de código que garanta a correta ordem de execução das instruções, lembrando-se que o emulador sempre começa a execução a partir da instrução zero. Neste exemplo, adicionamos um desvio incondicional para a primeira instrução que deve ser executada, localizada em PC = 032.

Implementando funções

Neste exercício será necessário implementar uma pilha para gerenciar a chamada de funções. Note que não temos funções recursivas. Porém, nossas funções não têm nome, sendo sempre chamadas de forma indireta. Isso é, para invocar uma função, precisamos colocar o endereço dela em um registrador e usa `jarl` para desviar o fluxo para o código da função. Para implementar funções em RISC-V, assumindo que cada função recebe um único parâmetro e retorna sempre um valor, podemos seguir a convenção de chamada (calling convention) padrão de RISC-V, onde o parâmetro é passado no registrador "`a0`" e o valor de retorno também é colocado em "`a0`". Abaixo, vou descrever como implementar a geração de código para uma função:

```
def visit_fn(self, exp, prog):
    # 1. Prologue: Alocar espaço na pilha e salvar o registrador
    # de retorno (ra) neste espaço.
    # 2. Passar o valor de a0 para o nome do parâmetro formal
    # (assumindo-se que o valor do parâmetro estava em "a0"):
    # 3. Gerar o código para o corpo da função e obter a variável de retorno:
    # 4. Epilogue: Restaurar o valor do registrador "ra" e ajustar a pilha
    # 5. Retornar ao chamador via um jump jalr, que usa o valor salvo em "ra"
```

Para controlar a pilha, você pode usar o registrador "`sp`". Este registrador é sempre inicializado com o tamanho da memória disponível. Assim, `sp - 1` é o último endereço de memória (que você pode usar como o início de sua pilha!). Sempre que for empilhar algo, decremente `sp` e use stores (`sw`) para guardar dados usando `sp` como endereço base. Para restaurar a pilha, incremente `sp` novamente. Isso permite usar sempre as mesmas variáveis para passar parâmetros e ler retorno: se uma nova função for chamada, guarde essas variáveis na pilha!

Essa abordagem garante que cada função em RISC-V seja auto-suficiente em termos de gerenciamento da pilha e dos registradores de retorno. Seguindo essa convenção, você mantém a integridade da pilha e facilita a chamada de funções dentro do seu programa, respeitando as convenções padrão de RISC-V. Para implementar a chamada de função, use sempre um registrador pré-definido para passar parâmetros e ler valor de retorno. Uma convenção comum é usar o registrador "a0" para passar parâmetros e ler valor de retorno. E podemos sempre salvar o endereço de retorno em um registrador "ra". Por exemplo:

```
def visit_app(self, exp, prog):
    # 1. Generate the instructions to find out the target of the call:
    # 2. Generate code to compute the parameter of the call:
    # 3. Jump to the function. Remember that Jarl saves
    # the current address into some register.
    # 4. Get the return value of the function. It's meant to be on a0:
```

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

```
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(memory_size = 1000, env = {}, insts = [])
    gen = GenVisitor()
    var_answer = exp.accept(gen, prog)
    prog.eval()
    print(f"Answer: {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  # 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
    FNX = 218  # The 'fn' that declares an anonymous function
    ARW = 219  # The '=>' that separates the parameter from the body of function


    def tokens(self):
        """
        This method is a token generator: it converts the string encapsulated
        into this object into a sequence of Tokens. Notice that this method
        filters out three kinds of tokens: white-spaces, comments and new lines.

        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\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <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

"""
This file implements a parser for SML with anonymous functions. The grammar is
as follows:

fn_exp  ::= fn <var> => fn_exp
          | if_exp
if_exp  ::= <if> if_exp <then> fn_exp <else> fn_exp
          | or_exp
or_exp  ::= and_exp (or and_exp)*
and_exp ::= eq_exp (and eq_exp)*
eq_exp  ::= cmp_exp (= cmp_exp)*
cmp_exp ::= add_exp ([<=|<] add_exp)*
add_exp ::= mul_exp ([+|-] mul_exp)*
mul_exp ::= unary_exp ([*|/] unary_exp)*
unary_exp ::= <not> unary_exp
             | ~ unary_exp
             | let_exp
let_exp ::= <let> <var> <- fn_exp <in> fn_exp <end>
          | val_exp
val_exp ::= val_tk (val_tk)*
val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>

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

        >>> tk0 = Token('fn', TokenType.FNX)
        >>> tk1 = Token('v', TokenType.VAR)
        >>> tk2 = Token('=>', TokenType.ARW)
        >>> tk3 = Token('v', TokenType.VAR)
        >>> tk4 = Token('+', TokenType.ADD)
        >>> tk5 = Token('1', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> print(exp.accept(ev, None))
        Fn(v)

        >>> tk0 = Token('(', TokenType.LPR)
        >>> tk1 = Token('fn', TokenType.FNX)
        >>> tk2 = Token('v', TokenType.VAR)
        >>> tk3 = Token('=>', TokenType.ARW)
        >>> tk4 = Token('v', TokenType.VAR)
        >>> tk5 = Token('+', TokenType.ADD)
        >>> tk6 = Token('1', TokenType.NUM)
        >>> tk7 = Token(')', TokenType.RPR)
        >>> tk8 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, {})
        3
        """
        # 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)
```

```
104
105
106 class Mul(BinaryExpression):
107     """
108     This class represents multiplication of two expressions. The acceptuation of
109     such an expression is the product of the two subexpression's values.
110     """
111     def accept(self, visitor, arg):
112         return visitor.visit_mul(self, arg)
113
114
115 class Div(BinaryExpression):
116     """
117     This class represents the integer division of two expressions. The
118     acceptuation of such an expression is the integer quocient of the two
119     subexpression's values.
120     """
121     def accept(self, visitor, arg):
122         return visitor.visit_div(self, arg)
123
124
125 class Leq(BinaryExpression):
126     """
127     This class represents comparison of two expressions using the
128     less-than-or-equal comparator. The acceptuation of such an expression is a
129     boolean value that is true if the left operand is less than or equal the
130     right operand. It is false otherwise.
131     """
132     def accept(self, visitor, arg):
133         return visitor.visit_leq(self, arg)
134
135
136 class Lth(BinaryExpression):
137     """
138     This class represents comparison of two expressions using the
139     less-than comparison operator. The acceptuation of such an expression is a
140     boolean value that is true if the left operand is less than the right
141     operand. It is false otherwise.
142     """
143     def accept(self, visitor, arg):
144         return visitor.visit_lth(self, arg)
145
146
147 class UnaryExpression(Expression):
148     """
149     This class represents unary expressions. A unary expression has only one
150     sub-expression.
151     """
152     def __init__(self, exp):
153         self.exp = exp
154
155     @abstractmethod
156     def accept(self, visitor, arg):
157         raise NotImplementedError
158
159
160 class Neg(UnaryExpression):
161     """
162     This expression represents the additive inverse of a number. The additive
163     inverse of a number n is the number -n, so that the sum of both is zero.
164     """
165     def accept(self, visitor, arg):
166         return visitor.visit_neg(self, arg)
167
168
169 class Not(UnaryExpression):
170     """
171     This expression represents the negation of a boolean. The negation of a
172     boolean expression is the logical complement of that expression.
173     """
174     def accept(self, visitor, arg):
175         return visitor.visit_not(self, arg)
176
177
178 class Let(Expression):
179     """
180     This class represents a let expression. The semantics of a let expression,
181     such as "let v <- e0 in e1" on an environment env is as follows:
182     1. Evaluate e0 in the environment env, yielding e0_val
183     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
184     """
185     def __init__(self, identifier, exp_def, exp_body):
186         self.identifier = identifier
187         self.exp_def = exp_def
188         self.exp_body = exp_body
189     def accept(self, visitor, arg):
190         return visitor.visit_let(self, arg)
191
192
193 class IfThenElse(Expression):
194     """
195     This class represents a conditional expression. The semantics an expression
196     such as 'if B then E0 else E1' is as follows:
197     1. Evaluate B. Call the result ValueB.
198     2. If ValueB is True, then evalute E0 and return the result.
199     3. If ValueB is False, then evaluate E1 and return the result.
200     Notice that we only evaluate one of the two sub-expressions, not both. Thus,
201     "if True then 0 else 1 div 0" will return 0 indeed.
202     """
203     def __init__(self, cond, e0, e1):
204         self.cond = cond
205         self.e0 = e0
206         self.e1 = e1
```

```
207         def accept(self, visitor, arg):
208             return visitor.visit_ifThenElse(self, arg)
209
210
211     class Fn(Expression):
212         """
213         This class represents an anonymous function.
214         """
215
216         def __init__(self, formal, body):
217             self.formal = formal
218             self.body = body
219
220         def accept(self, visitor, arg):
221             return visitor.visit_fn(self, arg)
222
223
224     class App(Expression):
225         """
226         This class represents a function application, such as 'e0 e1'. The semantics
227         of an application is as follows: we evaluate the left side, e.g., e0. It
228         must result into a function fn(p, b) denoting a function that takes in a
229         parameter p and evaluates a body b. We then evaluates e1, to obtain a value
230         v. Finally, we evaluate b, but in a context where p is bound to v.
231         """
232
233         def __init__(self, function, actual):
234             self.function = function
235             self.actual = actual
236
237         def accept(self, visitor, arg):
238             return visitor.visit_app(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

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

    @abstractmethod
    def visit_app(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
        """
        # 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.
```

```
        # 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)
```

```
414          >>> p.eval()
415          >>> p.get_val(v)
416          0
417
418          >>> e = Lth(Num(3), Num(3))
419          >>> p = AsmModule.Program({}, [])
420          >>> g = GenVisitor()
421          >>> v = e.accept(g, p)
422          >>> p.eval()
423          >>> p.get_val(v)
424          0
425
426          >>> e = Lth(Num(2), Num(3))
427          >>> p = AsmModule.Program({}, [])
428          >>> g = GenVisitor()
429          >>> v = e.accept(g, p)
430          >>> p.eval()
431          >>> p.get_val(v)
432          1
433          """
434          # TODO: Implement this method.
435          raise NotImplementedError
436
437      def visit_neg(self, exp, prog):
438          """
439          >>> e = Neg(Num(3))
440          >>> p = AsmModule.Program({}, [])
441          >>> g = GenVisitor()
442          >>> v = e.accept(g, p)
443          >>> p.eval()
444          >>> p.get_val(v)
445          -3
446
447          >>> e = Neg(Num(0))
448          >>> p = AsmModule.Program({}, [])
449          >>> g = GenVisitor()
450          >>> v = e.accept(g, p)
451          >>> p.eval()
452          >>> p.get_val(v)
453          0
454
455          >>> e = Neg(Num(-3))
456          >>> p = AsmModule.Program({}, [])
457          >>> g = GenVisitor()
458          >>> v = e.accept(g, p)
459          >>> p.eval()
460          >>> p.get_val(v)
461          3
462          """
463          # TODO: Implement this method.
464          raise NotImplementedError
465
466      def visit_not(self, exp, prog):
467          """
468          >>> e = Not(Bln(True))
469          >>> p = AsmModule.Program({}, [])
470          >>> g = GenVisitor()
471          >>> v = e.accept(g, p)
472          >>> p.eval()
473          >>> p.get_val(v)
474          0
475
476          >>> e = Not(Bln(False))
477          >>> p = AsmModule.Program({}, [])
478          >>> g = GenVisitor()
479          >>> v = e.accept(g, p)
480          >>> p.eval()
481          >>> p.get_val(v)
482          1
483
484          >>> e = Not(Num(0))
485          >>> p = AsmModule.Program({}, [])
486          >>> g = GenVisitor()
487          >>> v = e.accept(g, p)
488          >>> p.eval()
489          >>> p.get_val(v)
490          1
491
492          >>> e = Not(Num(-2))
493          >>> p = AsmModule.Program({}, [])
494          >>> g = GenVisitor()
495          >>> v = e.accept(g, p)
496          >>> p.eval()
497          >>> p.get_val(v)
498          0
499
500          >>> e = Not(Num(2))
501          >>> p = AsmModule.Program({}, [])
502          >>> g = GenVisitor()
503          >>> v = e.accept(g, p)
504          >>> p.eval()
505          >>> p.get_val(v)
506          0
507          """
508          # TODO: Implement this method.
509          raise NotImplementedError
510
511      def visit_let(self, exp, prog):
512          """
513          Usage:
514              >>> e = Let('v', Not(Bln(False)), Var('v'))
515              >>> p = AsmModule.Program({}, [])
516              >>> 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

    def visit_fn(self, exp, prog):
        # Here goes some hints. Feel free to disregard them if you think about
        # any other way to implement functions. Indeed, there are many ways to
        # to just that:
        #
        # 1. Allocate space on the stack:
        # 2. Get the param value. Assumed to be on a0:
        # 3. Generate a body for the function and save the return value in a0:
        # 4. Restore the stack, and get back the return address:
        # 5. Jump back to the caller:
        #
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_app(self, exp, prog):
        # Here goes some more hints. Again, take them if you feel like it:
        #
        # 1. Generate the instructions to find out the target of the call:
        # 2. Generate code to compute the parameter of the call:
        # 3. Jump to the function. Remember that Jarl saves the current address
        # into ra:
        # 4. Get the return value of the function. It's meant to be on a0:
```

```
        #
        # 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.
    """

    def __init__(self):
        # TODO: You might want to initialize some stuff here.
        pass

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

    def visit_bln(self, exp, name_map):
        pass

    def visit_num(self, exp, name_map):
        pass

    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.
```

```
723         raise NotImplementedError
724
725     def visit_lth(self, exp, name_map):
726         # TODO: Implement this method.
727         raise NotImplementedError
728
729     def visit_neg(self, exp, name_map):
730         # TODO: Implement this method.
731         raise NotImplementedError
732
733     def visit_not(self, exp, name_map):
734         # TODO: Implement this method.
735         raise NotImplementedError
736
737     def visit_ifThenElse(self, exp, name_map):
738         """
739         Examples:
740             >>> x0 = Var('x')
741             >>> x1 = Var('x')
742             >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
743             >>> e1 = Let('x', Num(3), e0)
744             >>> r = RenameVisitor()
745             >>> e1.accept(r, {})
746             >>> x0.identifier == x1.identifier
747             True
748
749             >>> x0 = Var('x')
750             >>> x1 = Var('x')
751             >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
752             >>> e1 = Let('x', Num(3), e0)
753             >>> e2 = Let('x', e1, Num(3))
754             >>> r = RenameVisitor()
755             >>> e1.accept(r, {})
756             >>> e2.identifier != x1.identifier == e1.identifier
757             True
758         """
759         # TODO: Implement this method.
760         raise NotImplementedError
761
762     def visit_let(self, exp, name_map):
763         """
764         Examples:
765             >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
766             >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
767             >>> e0.identifier == e1.identifier
768             True
769
770             >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
771             >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
772             >>> r = RenameVisitor()
773             >>> e1.accept(r, {})
774             >>> e0.identifier == e1.identifier
775             False
776
777             >>> x0 = Var('x')
778             >>> x1 = Var('x')
779             >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
780             >>> e1 = Let('x', e0, Mul(x1, Num(10)))
781             >>> x0.identifier == x1.identifier
782             True
783
784             >>> x0 = Var('x')
785             >>> x1 = Var('x')
786             >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
787             >>> e1 = Let('x', e0, Mul(x1, Num(10)))
788             >>> r = RenameVisitor()
789             >>> e1.accept(r, {})
790             >>> x0.identifier == x1.identifier
791             False
792         """
793         # TODO: Implement this method.
794         raise NotImplementedError
795
796     def visit_fn(self, exp, name_map):
797         """
798         >>> e0 = Fn('v', Mul(Var('v'), Var('v')))
799         >>> e1 = Let('v', e0, Var('v'))
800         >>> e1.accept(RenameVisitor(), {})
801         >>> e0.formal != e1.identifier
802         True
803
804         >>> x0 = Var('v')
805         >>> x1 = Var('v')
806         >>> x2 = Var('v')
807         >>> e0 = Fn('v', Mul(x0, x2))
808         >>> e1 = Let('v', e0, x1)
809         >>> e1.accept(RenameVisitor(), {})
810         >>> x0.identifier != x1.identifier and x0.identifier == x2.identifier
811         True
812         """
813         # TODO: Implement this method.
814         raise NotImplementedError
815
816     def visit_app(self, exp, name_map):
817         """
818         >>> x0 = Var('x')
819         >>> x1 = Var('x')
820         >>> x2 = Var('x')
821         >>> e = Let('x', Fn('x', Add(x0, Num(1))), App(x1, x2))
822         >>> e.accept(RenameVisitor(), {})
823         >>> x0.identifier != x1.identifier and x1.identifier == x2.identifier
824         True
825         """
826         # TODO: Implement this method
```

```
826         # TODO: Implement this method.
827         raise NotImplementedError
```

Asm.py

```
"""
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
    * jalr rd, rs1, offset: rd = pc + 1 and pc = rs1 + offset
    * sw reg, offset(rs1): mem[offset+rs1] = reg
    * lw reg, offset(rs1): reg = mem[offset+rs1]

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, memory_size, env, insts):
        self.__mem = memory_size * [0]
        self.__env = env
        self.__insts = insts
        self.pc = 0
        self.__env["x0"] = 0
        self.__env["sp"] = memory_size

    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 set_mem(self, addr, value):
        self.__mem[addr] = value

    def get_mem(self, addr):
        return self.__mem[addr]

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

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

    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(0, {"b0":2, "b1":3, "b2": 4}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            b2: 4
            sp: 0
            t0: 5
            x0: 0
            x1: 1

        Notice that it is not possible to change 'x0':
            >>> insts = [Add("x0", "b0", "b1")]
            >>> p = Program(0, {"b0":2, "b1":3}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            sp: 0
            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(0, {}, [])
    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(0, {}, [])
    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`.

    Example:
        >>> i = Jal("a", 20)
        >>> str(i)
        'jal a 20'

        >>> p = Program(10, env={}, insts=[Jal("a", 20)])
        >>> p.eval()
        >>> p.get_pc(), p.get_val("a")
        (20, 2)

        >>> p = Program(10, env={}, insts=[Jal("x0", 20)])
        >>> p.eval()
        >>> p.get_pc(), p.get_val("x0")
        (20, 0)
    """

    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":
            # Notice that Jal and Jalr set pc to pc + 1. However, when we fetch
            # an instruction, we already increment the PC. Therefore, by using
            # get_pc, we are indeed, reading pc + 1.
            prog.set_val(self.rd, prog.get_pc())
        prog.set_pc(self.lab)


class Jalr(BranchOp):
    """
    jalr rd, rs, offset
    The jalr rd, rs, offset instruction performs an indirect jump to the
    address computed by adding the value in rs to the immediate offset, and
    stores the address of the instruction following the jump into rd.

    Example:
        >>> i = Jalr("a", "b", 20)
        >>> str(i)
        'jalr a b 20'

        >>> p = Program(10, env={"b":30}, insts=[Jalr("a", "b", 20)])
        >>> p.eval()
        >>> p.get_pc(), p.get_val("a")
        (50, 2)

        >>> p = Program(10, env={"b":30}, insts=[Jalr("x0", "b", 20)])
        >>> p.eval()
        >>> p.get_pc(), p.get_val("x0")
```

```
        >>> p.get_pc(), p.get_val("x0")
        (50, 0)
    """

    def __init__(self, rd, rs, offset=0):
        assert isinstance(rd, str) and isinstance(rs, str)
        self.rd = rd
        self.rs = rs
        if offset != None:
            assert isinstance(offset, int)
        self.offset = offset

    def get_opcode(self):
        return "jalr"

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

    def eval(self, prog):
        if self.rd != "x0":
            prog.set_val(self.rd, prog.get_pc())
        rs_val = prog.get_val(self.rs)
        prog.set_pc(rs_val + self.offset)


class MemOp(Inst):
    """
    The general class of instructions that access memory. These instructions
    include loads and stores.
    """

    def __init__(self, rs1, offset, reg):
        assert isinstance(rs1, str) and isinstance(reg, str) and isinstance(offset, int)
        self.rs1 = rs1
        self.offset = offset
        self.reg = reg

    def __str__(self):
        op = self.get_opcode()
        return f"{op} {self.reg}, {self.offset}({self.rs1})"


class Sw(MemOp):
    """
    sw reg, offset(rs1)
    *(rs1 + offset) = reg

    * reg: The source register containing the data to be stored.
    * rs1: The base register containing the memory address.
    * offset: A 12-bit signed immediate that is added to rs1 to form the
      effective address.

    Example:
        >>> i = Sw("a", 0, "b")
        >>> str(i)
        'sw b, 0(a)'

        >>> p = Program(10, env={"b":2, "a":3}, insts=[Sw("a", 0, "b")])
        >>> p.eval()
        >>> p.get_mem(3)
        2
    """

    def eval(self, prog):
        val = prog.get_val(self.reg)
        addr = prog.get_val(self.rs1) + self.offset
        prog.set_mem(addr, val)

    def get_opcode(self):
        return "sw"


class Lw(MemOp):
    """
    lw reg, offset(rs1)
    reg = *(rs1 + offset)

    * reg: The destination register that will be overwritten.
    * rs1: The base register containing the memory address.
    * offset: A 12-bit signed immediate that is added to rs1 to form the
      effective address.

    Example:
        >>> i = Lw("a", 0, "b")
        >>> str(i)
        'lw b, 0(a)'

        >>> p = Program(10, env={"a":2}, insts=[Lw("a", 0, "b")])
        >>> p.eval()
        >>> p.get_val("b")
        0

        >>> insts = [Sw("a", 0, "b"), Lw("a", 0, "c")]
        >>> p = Program(10, env={"a":2, "b":5}, insts=insts)
        >>> p.eval()
        >>> p.get_val("c")
        5
    """

    def eval(self, prog):
        addr = prog.get_val(self.rs1) + self.offset
        val = prog.get_mem(addr)
        prog.set_val(self.reg, val)
```

```
414
415     def get_opcode(self):
416         return "lw"
417
418
419 class BinOp(Inst):
420     """
421     The general class of binary instructions. These instructions define a
422     value, and use two values.
423     """
424
425     def __init__(self, rd, rs1, rs2):
426         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(rs2, str)
427         self.rd = rd
428         self.rs1 = rs1
429         self.rs2 = rs2
430
431     def __str__(self):
432         op = self.get_opcode()
433         return f"{self.rd} = {op} {self.rs1} {self.rs2}"
434
435
436 class BinOpImm(Inst):
437     """
438     The general class of binary instructions where the second operand is an
439     integer constant. These instructions define a value, and use one variable
440     and one immediate constant.
441     """
442
443     def __init__(self, rd, rs1, imm):
444         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
445         self.rd = rd
446         self.rs1 = rs1
447         self.imm = imm
448
449     def __str__(self):
450         op = self.get_opcode()
451         return f"{self.rd} = {op} {self.rs1} {self.imm}"
452
453
454 class Add(BinOp):
455     """
456     add rd, rs1, rs2: rd = rs1 + rs2
457
458     Example:
459         >>> i = Add("a", "b0", "b1")
460         >>> str(i)
461         'a = add b0 b1'
462
463         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
464         >>> p.eval()
465         >>> p.get_val("a")
466         5
467     """
468
469     def eval(self, prog):
470         rs1 = prog.get_val(self.rs1)
471         rs2 = prog.get_val(self.rs2)
472         prog.set_val(self.rd, rs1 + rs2)
473
474     def get_opcode(self):
475         return "add"
476
477
478 class Addi(BinOpImm):
479     """
480     addi rd, rs1, imm: rd = rs1 + imm
481
482     Example:
483         >>> i = Addi("a", "b0", 1)
484         >>> str(i)
485         'a = addi b0 1'
486
487         >>> p = Program(0, env={"b0":2}, insts=[Addi("a", "b0", 3)])
488         >>> p.eval()
489         >>> p.get_val("a")
490         5
491     """
492
493     def eval(self, prog):
494         rs1 = prog.get_val(self.rs1)
495         prog.set_val(self.rd, rs1 + self.imm)
496
497     def get_opcode(self):
498         return "addi"
499
500
501 class Mul(BinOp):
502     """
503     mul rd, rs1, rs2: rd = rs1 * rs2
504
505     Example:
506         >>> i = Mul("a", "b0", "b1")
507         >>> str(i)
508         'a = mul b0 b1'
509
510         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
511         >>> p.eval()
512         >>> p.get_val("a")
513         6
514     """
515
516     def eval(self, prog):
```

```
517         rs1 = prog.get_val(self.rs1)
518         rs2 = prog.get_val(self.rs2)
519         prog.set_val(self.rd, rs1 * rs2)
520
521     def get_opcode(self):
522         return "mul"
523
524
525 class Sub(BinOp):
526     """
527     sub rd, rs1, rs2: rd = rs1 - rs2
528
529     Example:
530         >>> i = Sub("a", "b0", "b1")
531         >>> str(i)
532         'a = sub b0 b1'
533
534         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
535         >>> p.eval()
536         >>> p.get_val("a")
537         -1
538     """
539
540     def eval(self, prog):
541         rs1 = prog.get_val(self.rs1)
542         rs2 = prog.get_val(self.rs2)
543         prog.set_val(self.rd, rs1 - rs2)
544
545     def get_opcode(self):
546         return "sub"
547
548
549 class Xor(BinOp):
550     """
551     xor rd, rs1, rs2: rd = rs1 ^ rs2
552
553     Example:
554         >>> i = Xor("a", "b0", "b1")
555         >>> str(i)
556         'a = xor b0 b1'
557
558         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
559         >>> p.eval()
560         >>> p.get_val("a")
561         1
562     """
563
564     def eval(self, prog):
565         rs1 = prog.get_val(self.rs1)
566         rs2 = prog.get_val(self.rs2)
567         prog.set_val(self.rd, rs1 ^ rs2)
568
569     def get_opcode(self):
570         return "xor"
571
572
573 class Xori(BinOpImm):
574     """
575     xori rd, rs1, imm: rd = rs1 ^ imm
576
577     Example:
578         >>> i = Xori("a", "b0", 10)
579         >>> str(i)
580         'a = xori b0 10'
581
582         >>> p = Program(0, env={"b0":2}, insts=[Xori("a", "b0", 3)])
583         >>> p.eval()
584         >>> p.get_val("a")
585         1
586     """
587
588     def eval(self, prog):
589         rs1 = prog.get_val(self.rs1)
590         prog.set_val(self.rd, rs1 ^ self.imm)
591
592     def get_opcode(self):
593         return "xori"
594
595
596 class Div(BinOp):
597     """
598     div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
599     Notice that RISC-V does not have an instruction exactly like this one.
600     The div operator works on floating-point numbers; not on integers.
601
602     Example:
603         >>> i = Div("a", "b0", "b1")
604         >>> str(i)
605         'a = div b0 b1'
606
607         >>> p = Program(0, env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
608         >>> p.eval()
609         >>> p.get_val("a")
610         2
611     """
612
613     def eval(self, prog):
614         rs1 = prog.get_val(self.rs1)
615         rs2 = prog.get_val(self.rs2)
616         prog.set_val(self.rd, rs1 // rs2)
617
618     def get_opcode(self):
619         return "div"
```

```
620
621
622 class Slt(BinOp):
623     """
624     slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
625
626     Example:
627         >>> i = Slt("a", "b0", "b1")
628         >>> str(i)
629         'a = slt b0 b1'
630
631         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
632         >>> p.eval()
633         >>> p.get_val("a")
634         1
635
636         >>> p = Program(0, env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
637         >>> p.eval()
638         >>> p.get_val("a")
639         0
640
641         >>> p = Program(0, env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
642         >>> p.eval()
643         >>> p.get_val("a")
644         0
645     """
646
647     def eval(self, prog):
648         rs1 = prog.get_val(self.rs1)
649         rs2 = prog.get_val(self.rs2)
650         prog.set_val(self.rd, 1 if rs1 < rs2 else 0)
651
652     def get_opcode(self):
653         return "slt"
654
655
656 class Slti(BinOpImm):
657     """
658     slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
659     (signed comparison with immediate)
660
661     Example:
662         >>> i = Slti("a", "b0", 0)
663         >>> str(i)
664         'a = slti b0 0'
665
666         >>> p = Program(0, env={"b0":2}, insts=[Slti("a", "b0", 3)])
667         >>> p.eval()
668         >>> p.get_val("a")
669         1
670
671         >>> p = Program(0, env={"b0":3}, insts=[Slti("a", "b0", 3)])
672         >>> p.eval()
673         >>> p.get_val("a")
674         0
675
676         >>> p = Program(0, env={"b0":3}, insts=[Slti("a", "b0", 2)])
677         >>> p.eval()
678         >>> p.get_val("a")
679         0
680     """
681
682     def eval(self, prog):
683         rs1 = prog.get_val(self.rs1)
684         prog.set_val(self.rd, 1 if rs1 < self.imm else 0)
685
686     def get_opcode(self):
687         return "slti"
```

VPL