2025_2 - COMPILADORES - METATURMA

AV15 - Alocação de registros para código linear

Data de entrega: segunda, 1 Dez 2025, 23:59
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py, Optimizer.py (Baixar)
Tipo de trabalho: Trabalho individual

O objetivo deste trabalho prático é implementar um alocador de registradores para código linear. Código linear é código que não possui desvios, sejam condicionais ou não. No campo de compiladores, esses programas são chamados de "blocos básicos". Assim, este trabalho começa com o código usado no VPL 11 (geração de código para expressões lógicas e aritméticas). Naquele exercício, você gerou código que alocava valores em variáveis. Sempre era possível criar um nome novo de variável. Desta vez, suas instruções têm somente um número fixo de registradores que podem ser usados. Caso a quantidade de valores exceda a quantidade de registradores disponíveis, você precisa alocar esses valores excedentes em memória. Seu programa possui agora os seguintes registradores:

- `x0`: registrador que contém sempre o valor zero. Você não consegue escrever sobre ele.
- `sp`: registrador que é inicializado com o tamanho da memória. Ele é usado para implementar uma pilha. A pilha não será necessária neste exercício, mas convém, ainda assim, não escrever neste registrador. Você pode usá-lo no futuro.
- `ra`: registrador usado para guardar o endereço de retorno de funções. Novamente, neste exercício não temos funções. Mas ainda assim, convém não escrever em `ra`, pois no futuro podemos querer incorporar funções ao nosso compilador.
- `a0`: primeiro registrador de propósito geral. Use-o com abandono selvagem!
- `a1`: segundo registrador de propósito geral. Use-o como quiser.
- `a2`: terceiro registrador de propósito geral. Novamente, use-o do jeito que preferir.
- `a3`: quarto registrador de propósito geral. Abandono selvagem (novamente)!

Suas instruções podem somente usar estes registradores. Para mapear valores em memória, use loads and stores:

- `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.

Essas instruções lhe dão acesso à *memória* do programa em execução. 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. Exemplos de acesso à memória são mostrados abaixo:

- `Sw("sp", -1, "a")`: mem[val(sp) - 1] := val(a)
- `Lw("sp", 0, "b")`: val(b) := mem[val(sp)]
- `Sw("x0", 7, "a")`: mem[7] := val(a)
- `Lw("x0", 7, "b")`: val(b) := mem[7]

O que deve ser feito

Para resolver este exercício, você deverá reusar as implementações de `RenameVisitor` e de `GenVisitor` (ambas em `Visitor.py`) vistas no VPL 12, além de implementar uma nova classe: `RegAllocator`, que está disponível no arquivo `Optimizer.py`. Você deverá implementar três métodos de RegAllocator, a saber:
`__init__(self, prog)`: assumindo-se que você queira inicializar dados relacionados à alocação de registradores.
`optimize()`: o método que substitui as instruções em prog por novas instruções, que usam somente endereços de memória ou registradores.
`get_val(var)`: o método que retorna o valor de "var". Como "prog" já não usa instruções que fazem referência ao nome "var", você precisará salvar no otimizar como "var" está mapeada. Este nome pode estar associado a algum registrador, ou a algum endereço de memória. As três figuras abaixo ilustram como o valor associado a uma variável pode ser recuperado após a alocação de registradores.

Figura 1: Representação do programa “2 - 3 - 4” antes da alocação de registradores. Note que estamos usando nomes de variáveis, como v1 e v3, que não são registradores:

```
000: v1 = addi x0 2
001: v2 = addi x0 3
002: v3 = sub v1 v2
003: v4 = addi x0 4
004: v5 = sub v3 v4
```

reg	vars
a0	v1
a1	v2
a0	v3
a1	v4
a1	v5

Figura 2: Representação do programa “2 - 3 - 4” após a alocação de registradores. Neste caso, o valor da expressão agora está armazenado no registrador a1.

```
000: a0 = addi x0 2
001: a1 = addi x0 3
002: a0 = sub a0 a1
003: a1 = addi x0 4
004: a1 = sub a0 a1
```

Para obter o valor da variável “v5”, você pode fazer assim: `prog.get_val(“a1”)`. Mas você precisa, dentro do otimizador, lembrar que “v1” agora está mapeada em “a1”

Figura 3: Representação do programa “2 - 3 - 4” após a alocação de registradores. Neste caso, todas as variáveis foram mapeadas em memória. Para recuperar o valor da variável “v5”, agora você precisa acessá-la em memória, por exemplo, fazendo `prog.get_mem(4)`.

```
000: a1 = addi x0 2
001: sw a1, 0(x0)
002: a1 = addi x0 3
003: sw a1, 1(x0)
004: lw a2, 0(x0)
006: a1 = sub a2 a1
007: sw a1, 2(x0)
008: a1 = addi x0 4
009: sw a1, 3(x0)
010: lw a2, 2(x0)
012: a1 = sub a2 a1
013: sw a1, 4(x0)
014: END
```

mem	vars	values
0:	v1	2
1:	v2	3
2:	v3	-1
3:	v4	4
4:	v5	-5

Submetendo e Testando

Este VPL deve ser construído sobre o VPL 12. Para completar este VPL, você deverá entregar sete arquivos: `Expression.py`, `Optimizer.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
1 + 2 # CTRL+D
3
```

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
from Optimizer import *


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


def perform_register_allocation(prog, dump=False):
    """
    Esta funcao invoca o alocador de registradores sobre o programa. Caso queira
    depurar sua alocacao, fique a vontade para usar dump == True.
    """
    o = RegAllocator(prog)
    if dump:
        print("Before RA: ---------------")
        prog.print_insts()
    o.optimize()
    if dump:
        print("After RA: ---------------")
        prog.print_insts()
    return o


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(1000, {}, [])
    gen = GenVisitor()
    var_answer = exp.accept(gen, prog)
    opt = perform_register_allocation(prog, False)
    prog.reset_env()
    prog.eval()
    print(f"Answer: {opt.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
    ADD = 202
    SUB = 203
    MUL = 204
    DIV = 205
    LEQ = 206
    LTH = 207
    NEG = 208
    NOT = 209
    LPR = 210
    RPR = 211
    ASN = 212  # The assignment '<-' operator


class Lexer:

    def __init__(self, source):
        """
        The constructor of the lexer. It receives the string that shall be
        scanned.
        TODO: You will need to implement this method.
        """
        pass

    def tokens(self):
        """
        This method is a token generator: it converts the string encapsulated
        into this object into a sequence of Tokens. Examples:

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

        >>> l = Lexer('1 * 2 -- 3\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]

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

        >>> l = Lexer("let v <- 2 in v end")
        >>> [tk.kind.name for tk in l.tokens()]
        ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
        """
        token = self.getToken()
        while token.kind != TokenType.EOF:
            if (
                token.kind != TokenType.WSP
                and token.kind != TokenType.COM
                and token.kind != TokenType.NLN
            ):
                yield token
            token = self.getToken()

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

Parser.py

```python
import sys

from Expression import *
from Lexer import Token, TokenType

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

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

class Parser:
    def __init__(self, tokens):
        """
        Initializes the parser. The parser keeps track of the list of tokens
        and the current token. For instance:
        """
        self.tokens = list(tokens)
        self.cur_token_idx = 0 # This is just a suggestion!
        # You can (and probably should!) modify this method.

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

        Examples:
        >>> parser = Parser([Token('123', TokenType.NUM)])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        123

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

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

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

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

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('~', TokenType.NEG)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        -12

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

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

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('-', TokenType.SUB)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        26

        >>> tk0 = Token('2', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('(', TokenType.LPR)
        >>> tk3 = Token('3', TokenType.NUM)
        >>> tk4 = Token('+', TokenType.ADD)
        >>> tk5 = Token('4', TokenType.NUM)
        >>> tk6 = Token(')', TokenType.RPR)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        14

```

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

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

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

        >>> tk0 = Token('not', TokenType.NOT)
        >>> tk1 = Token('4', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        True

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

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

Expression.py

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

from abc import ABC, abstractmethod
from Visitor import *


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


class Var(Expression):
    """
    This class represents expressions that are identifiers. The value of an
    indentifier is the value associated with it in the environment table.
    """

    def __init__(self, identifier):
        self.identifier = identifier

    def accept(self, visitor, arg):
        """
        Variables don't need to be implemented for this exercise.
        """
        return visitor.visit_var(self, arg)


class Bln(Expression):
    """
    This class represents expressions that are boolean values. There are only
    two boolean values: true and false. The acceptuation of such an expression
    is the boolean itself.
    """

    def __init__(self, bln):
        self.bln = bln

    def accept(self, visitor, arg):
        """
        booleans don't need to be implemented for this exercise.
        """
        return visitor.visit_bln(self, arg)


class Num(Expression):
    """
    This class represents expressions that are numbers. The acceptuation of such
    an expression is the number itself.
    """

    def __init__(self, num):
        self.num = num

    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Num(3)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        3
        """
        return visitor.visit_num(self, arg)


class BinaryExpression(Expression):
    """
    This class represents binary expressions. A binary expression has two
    sub-expressions: the left operand and the right operand.
    """

    def __init__(self, left, right):
        self.left = left
        self.right = right

    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Eql(BinaryExpression):
    """
    This class represents the equality between two expressions. The acceptuation
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """

    def accept(self, visitor, arg):
        """
        Equality doesn't need to be implemented for this exercise.
        """
        return visitor.visit_eql(self, arg)


class Add(BinaryExpression):
    """
    This class represents addition of two expressions. The acceptuation of such
    an expression is the addition of the two subexpression's values.
    """

```

```
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        """
        return visitor.visit_add(self, arg)


class Sub(BinaryExpression):
    """
    This class represents subtraction of two expressions. The acceptuation of
    such an expression is the subtraction of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -1
        """
        return visitor.visit_sub(self, arg)


class Mul(BinaryExpression):
    """
    This class represents multiplication of two expressions. The acceptuation of
    such an expression is the product of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        12
        """
        return visitor.visit_mul(self, arg)


class Div(BinaryExpression):
    """
    This class represents the integer division of two expressions. The
    acceptuation of such an expression is the integer quocient of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        5
        """
        return visitor.visit_div(self, arg)


class Leq(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than-or-equal comparator. The acceptuation of such an expression is a
    boolean value that is true if the left operand is less than or equal the
    right operand. It is false otherwise.
    """

    def accept(self, visitor, arg):
        """
        Comparisons don't need to be implemented for this exercise.
        """
        return visitor.visit_leq(self, arg)


class Lth(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than comparison operator. The acceptuation of such an expression is a
    boolean value that is true if the left operand is less than the right
    operand. It is false otherwise.
    """

    def accept(self, visitor, arg):
        """
        Comparisons don't need to be implemented for this exercise.
        """
```

```
        """
        return visitor.visit_lth(self, arg)


class UnaryExpression(Expression):
    """
    This class represents unary expressions. A unary expression has only one
    sub-expression.
    """

    def __init__(self, exp):
        self.exp = exp

    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Neg(UnaryExpression):
    """
    This expression represents the additive inverse of a number. The additive
    inverse of a number n is the number -n, so that the sum of both is zero.
    """

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


class Not(UnaryExpression):
    """
    This expression represents the negation of a boolean. The negation of a
    boolean expression is the logical complement of that expression.
    """

    def accept(self, visitor, arg):
        """
        No need to implement negation for this exercise, for we don't even have
        booleans at this point.
        """
        return visitor.visit_not(self, arg)


class Let(Expression):
    """
    This class represents a let expression. The semantics of a let expression,
    such as "let v <- e0 in e1" on an environment env is as follows:
    1. Evaluate e0 in the environment env, yielding e0_val
    2. Evaluate e1 in the new environment env' = env + {v:e0_val}
    """

    def __init__(self, identifier, exp_def, exp_body):
        self.identifier = identifier
        self.exp_def = exp_def
        self.exp_body = exp_body

    def accept(self, visitor, arg):
        """
        We don't have bindings at this point. So, nothing to be done here, for
        this exercise.
        """
        return visitor.visit_let(self, arg)
```

Visitor.py

```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_add(self, exp, arg):
        pass

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

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

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

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

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

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

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

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


class 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
    """
```

```python

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

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

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

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

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

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

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

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

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

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

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

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

    def visit_let(self, exp, arg):
        # TODO: Implement this method.
        raise NotImplementedError


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

    def __init__(self):
        self.next_var_counter = 0

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

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

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

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

    def visit_num(self, exp, prog):
        """
```

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

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

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

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

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

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

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

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

    def visit_mul(self, exp, prog):
        """
        >>> e = Mul(Num(13), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        26

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

```
        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
```

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
    * 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.registers = {"x0", "a0", "a1", "a2", "a3", "ra", "sp"}
        for reg in self.registers:
            self.__env[reg] = 0
        self.__env["sp"] = memory_size

    def reset_env(self):
        for reg in self.registers:
            self.__env[reg] = 0
        self.__env["sp"] = len(self.__mem)

    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 in self.registers and name != "x0":
            self.__env[name] = value
        else:
            sys.exit(f"Undefined register: {name}")

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

    def get_mem(self, addr):
        if addr < 0 or addr >= len(self.__mem):
            sys.exit(f"Invalid memory address: {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 get_insts(self):
        return self.__insts.copy()

    def set_insts(self, insts):
        self.__insts = insts

    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
```

```python
    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")
        (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):
        rs_val = prog.get_val(self.rs)
        if self.rd != "x0":
            prog.set_val(self.rd, prog.get_pc())
        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)

    def get_opcode(self):
        return "lw"


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(0, 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(0, 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(0, 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(0, 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(0, 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(0, 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(0, 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(0, env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        1

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

        >>> p = Program(0, 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(0, env={"b0":2}, insts=[Slti("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        1

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

        >>> p = Program(0, 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"
```

Optimizer.py

```python
from abc import ABC, abstractmethod
from Asm import *


class Optimizer(ABC):
    """
    This class implements an "Optimization Pass". The pass receives a sequence
    of instructions stored in a program, and produces a new sequence of
    instructions.
    """

    def __init__(self, prog):
        self.prog = prog

    @abstractmethod
    def optimize(self):
        pass


class RegAllocator(Optimizer):
    """This file implements the register allocation pass."""

    def __init__(self, prog):
        # TODO: you might want to save/initialize some stuff in the ctor.
        super().__init__(prog)

    def get_val(self, var):
        """
        Informs the value that is associated with the variable var within
        the program prog.
        """
        # TODO: Implement this method.
        raise NotImplementedError


    def optimize(self):
        """
        This function perform register allocation. It maps variables into
        memory, and changes instructions, so that they use one of the following
        registers:
        * x0: always the value zero. Can't change.
        * sp: the stack pointer. Starts with the memory size.
        * ra: the return address.
        * a0: function argument 0 (or return address)
        * a1: function argument 1
        * a2: function argument 2
        * a3: function argument 3

        Notice that next to each register we have suggested a usage. You can,
        of course, write on them and use them in other ways. But, at least x0
        and sp you should not overwrite. The first register you can't overwrite,
        actually. And sp is initialized with the number of memory addresses.
        It's good to use it to control the function stack.

        Examples:
        >>> insts = [Addi("a", "x0", 3)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        3

        >>> insts = [Addi("a", "x0", 1), Slti("b", "a", 2)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        1

        >>> insts = [Addi("a", "x0", 3), Slti("b", "a", 2), Xori("c", "b", 5)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        5

        >>> insts = [Addi("sp", "sp", -1),Addi("a", "x0", 7),Sw("sp", 0, "a")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_mem(p.get_val("sp"))
        7

        >>> insts = [Addi("sp", "sp", -1),Addi("a", "x0", 7),Sw("sp", 0, "a")]
        >>> insts += [Lw("sp", 0, "b"), Addi("c", "b", 6)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        13

        >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Add("c", "a", "b")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        7
```

```
        >>> insts = [Addi("a", "x0", 28),Addi("b", "x0", 4),Div("c", "a", "b")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        7

        >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Mul("c", "a", "b")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        12

        >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Xor("c", "a", "b")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        7

        >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Slt("c", "a", "b")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        1

        >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Slt("c", "b", "a")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_val("a1")
        0

        If you want, you can allocate Jal/Jalr/Beq instructions, but that's not
        necessary for this exercise.

        >>> insts = [Jal("a", 30)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> (p.get_pc(), p.get_val("a1") > 0)
        (30, True)

        >>> insts = [Addi("a", "x0", 30), Jalr("b", "a")]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> (p.get_pc(), p.get_val("a1") > 0)
        (30, True)

        >>> insts = [Addi("a", "x0", 3), Addi("b", "a", 0), Beq("a", "b", 30)]
        >>> p = Program(1000, env={}, insts=insts)
        >>> o = RegAllocator(p)
        >>> o.optimize()
        >>> p.eval()
        >>> p.get_pc()
        30
        """
        # TODO: Implement this method.
        raise NotImplementedError
        # Hints:
        # new_insts = []
        # for inst in self.prog.get_insts():
        #     action = self.alloc_action[inst.get_opcode()]
        #     last_insts = action(inst, self)
        #     new_insts += last_insts
        # self.prog.set_insts(new_insts)
```

VPL