

## 2025\_2 - COMPILADORES - METATURMA

**PAINEL** > **MINHAS TURMAS** > **2025\_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**  
 > **AV13 - FLUXO DE CONTROLE**

[Descrição](#)

[Visualizar envios](#)

### AV13 - Fluxo de controle

**Data de entrega:** quarta, 12 Nov 2025, 23:59

**Arquivos requeridos:** driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py ( [Baixar](#))

**Tipo de trabalho:** Trabalho individual

O objetivo deste trabalho prático é gerar código para expressões que envolvem [fluxo de controle variável](#). O fluxo de controle variável é caracterizado pela presença de "[branches](#)" no programa: instruções que podem alterar o contador de programa da arquitetura. Neste exercício, usaremos um subconjunto de instruções RISC-V que inclui todas as instruções vistas no laboratório anterior, além de duas instruções de desvio:

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

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

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

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

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

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

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

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

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

### Submetendo e Testando

Este VPL deve ser construído sobre o VPL 12. Para completar este VPL, você deverá entregar seis arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Visitor.py`, `Asm.py` e `driver.py`. Você não deverá alterar `Asm.py`, `driver.py` ou `Expression.py`. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
$> python3 driver.py
true and false # CTRL+D
0
```

A implementação dos diferentes arquivos possui vários comentários `doctest`, que testam sua implementação. Caso queira testar seu código, simplesmente faça:

```
python3 -m doctest xx.py
```

No exemplo acima, substituta `xx.py` por algum dos arquivos que você queira testar (experimente com `Visitor.py`, por exemplo). Caso você não gere mensagens de erro, então seu trabalho está (quase) completo!

## Arquivos requeridos

### `driver.py`

```
1 import sys
2 from Expression import *
3 from Visitor import *
4 from Lexer import Lexer
5 from Parser import Parser
6 import Asm as AsmModule
7
8 def rename_variables(exp):
9     """
10     Esta função invoca o renomeador de variáveis. Ela deve ser usada antes do
11     inicio da fase de geração de código.
12     """
13     ren = RenameVisitor()
14     exp.accept(ren, {})
15     return exp
16
17 if __name__ == "__main__":
18     """
19     Este arquivo não deve ser alterado, mas deve ser enviado para resolver o
20     VPL. O arquivo contém o código que testa a implementação do parser.
21     """
22     text = sys.stdin.read()
23     lexer = Lexer(text)
24     parser = Parser(lexer.tokens())
25     exp = rename_variables(parser.parse())
26     prog = AsmModule.Program({}, [])
27     gen = GenVisitor()
28     var_answer = exp.accept(gen, prog)
29     prog.eval()
30     print(f"Answer: {prog.get_val(var_answer)}")
```

### `Lexer.py`

```

1 import sys
2 import enum
3
4
5 class Token:
6     """
7         This class contains the definition of Tokens. A token has two fields: its
8         text and its kind. The "kind" of a token is a constant that identifies it
9         uniquely. See the TokenType to know the possible identifiers (if you want).
10        You don't need to change this class.
11    """
12    def __init__(self, tokenText, tokenKind):
13        # The token's actual text. Used for identifiers, strings, and numbers.
14        self.text = tokenText
15        # The TokenType that this token is classified as.
16        self.kind = tokenKind
17
18
19 class TokenType(enum.Enum):
20     """
21         These are the possible tokens. You don't need to change this class at all.
22     """
23     EOF = -1 # End of file
24     NLN = 0 # New line
25     WSP = 1 # White Space
26     COM = 2 # Comment
27     NUM = 3 # Number (integers)
28     STR = 4 # Strings
29     TRU = 5 # The constant true
30     FLS = 6 # The constant false
31     VAR = 7 # An identifier
32     LET = 8 # The 'let' of the let expression
33     INX = 9 # The 'in' of the let expression
34     END = 10 # The 'end' of the let expression
35     EQL = 201 # x = y
36     ADD = 202 # x + y
37     SUB = 203 # x - y
38     MUL = 204 # x * y
39     DIV = 205 # x / y
40     LEQ = 206 # x <= y
41     LTH = 207 # x < y
42     NEG = 208 # ~x
43     NOT = 209 # not x
44     LPR = 210 # (
45     RPR = 211 # )
46     ASN = 212 # The assignment '<->' operator
47     ORX = 213 # x or y
48     AND = 214 # x and y
49     IFX = 215 # The 'if' of a conditional expression
50     THN = 216 # The 'then' of a conditional expression
51     ELS = 217 # The 'else' of a conditional expression
52
53
54 class Lexer:
55
56     def __init__(self, source):
57         """
58             The constructor of the lexer. It receives the string that shall be
59             scanned.
60             TODO: You will need to implement this method.
61         """
62         pass
63
64     def tokens(self):
65         """
66             This method is a token generator: it converts the string encapsulated
67             into this object into a sequence of Tokens. Examples:
68
69             >>> l = Lexer("1 + 3")
70             >>> [tk.kind for tk in l.tokens()]
71             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
72
73             >>> l = Lexer('1 * 2 -- 3\n')
74             >>> [tk.kind for tk in l.tokens()]
75             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
76
77             >>> l = Lexer("1 + var")
78             >>> [tk.kind for tk in l.tokens()]
79             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
80
81             >>> l = Lexer("let v <- 2 in v end")
82             >>> [tk.kind.name for tk in l.tokens()]
83             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
84         """
85         token = self.getToken()
86         while token.kind != TokenType.EOF:
87             if (
88                 token.kind != TokenType.WSP
89                 and token.kind != TokenType.COM
90                 and token.kind != TokenType.NLN
91             ):
92                 yield token
93             token = self.getToken()
94
95     def getToken(self):
96         """
97             Return the next token.
98             TODO: Implement this method (you can reuse Lab 5: Visitors)!
99         """
100            token = None
101            return token

```

Parser.py

```

1 import sys
2
3 from Expression import *
4 from Lexer import Token, TokenType
5
6 """
7 Precedence table:
8   1: not ~ ()
9   2: * /
10  3: + -
11  4: < <= >= >
12  5: =
13  6: and
14  7: or
15  8: if-then-else
16
17 Notice that not 2 < 3 must be a type error, as we are trying to apply a boolean
18 operation (not) onto a number. However, in assembly code this program works,
19 because not 2 is 0. The bottom line is: don't worry about programs like this
20 one: the would have been ruled out by type verification anyway.
21
22 References:
23     see https://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm#classic
24 """
25
26 class Parser:
27     def __init__(self, tokens):
28         """
29             Initializes the parser. The parser keeps track of the list of tokens
30             and the current token. For instance:
31         """
32         self.tokens = list(tokens)
33         self.cur_token_idx = 0 # This is just a suggestion!
34         # You can (and probably should!) modify this method.
35
36     def parse(self):
37         """
38             Returns the expression associated with the stream of tokens.
39
40             Examples:
41             >>> parser = Parser([Token('123', TokenType.NUM)])
42             >>> exp = parser.parse()
43             >>> ev = EvalVisitor()
44             >>> exp.accept(ev, None)
45             123
46
47             >>> parser = Parser([Token('True', TokenType.TRU)])
48             >>> exp = parser.parse()
49             >>> ev = EvalVisitor()
50             >>> exp.accept(ev, None)
51             True
52
53             >>> parser = Parser([Token('False', TokenType.FLS)])
54             >>> exp = parser.parse()
55             >>> ev = EvalVisitor()
56             >>> exp.accept(ev, None)
57             False
58
59             >>> tk0 = Token('~', TokenType.NEG)
60             >>> tk1 = Token('123', TokenType.NUM)
61             >>> parser = Parser([tk0, tk1])
62             >>> exp = parser.parse()
63             >>> ev = EvalVisitor()
64             >>> exp.accept(ev, None)
65             -123
66
67             >>> tk0 = Token('3', TokenType.NUM)
68             >>> tk1 = Token('*', TokenType.MUL)
69             >>> tk2 = Token('4', TokenType.NUM)
70             >>> parser = Parser([tk0, tk1, tk2])
71             >>> exp = parser.parse()
72             >>> ev = EvalVisitor()
73             >>> exp.accept(ev, None)
74             12
75
76             >>> tk0 = Token('3', TokenType.NUM)
77             >>> tk1 = Token('*', TokenType.MUL)
78             >>> tk2 = Token('~', TokenType.NEG)
79             >>> tk3 = Token('4', TokenType.NUM)
80             >>> parser = Parser([tk0, tk1, tk2, tk3])
81             >>> exp = parser.parse()
82             >>> ev = EvalVisitor()
83             >>> exp.accept(ev, None)
84             -12
85
86             >>> tk0 = Token('30', TokenType.NUM)
87             >>> tk1 = Token('/', TokenType.DIV)
88             >>> tk2 = Token('4', TokenType.NUM)
89             >>> parser = Parser([tk0, tk1, tk2])
90             >>> exp = parser.parse()
91             >>> ev = EvalVisitor()
92             >>> exp.accept(ev, None)
93             7
94
95             >>> tk0 = Token('3', TokenType.NUM)
96             >>> tk1 = Token('+', TokenType.ADD)
97             >>> tk2 = Token('4', TokenType.NUM)
98             >>> parser = Parser([tk0, tk1, tk2])
99             >>> exp = parser.parse()
100            >>> ev = EvalVisitor()
101            >>> exp.accept(ev, None)
102            7
103

```

```

104     >>> tk0 = Token('30', TokenType.NUM)
105     >>> tk1 = Token('-', TokenType.SUB)
106     >>> tk2 = Token('4', TokenType.NUM)
107     >>> parser = Parser([tk0, tk1, tk2])
108     >>> exp = parser.parse()
109     >>> ev = EvalVisitor()
110     >>> exp.accept(ev, None)
111     26
112
113     >>> tk0 = Token('2', TokenType.NUM)
114     >>> tk1 = Token('*', TokenType.MUL)
115     >>> tk2 = Token('(', TokenType.LPR)
116     >>> tk3 = Token('3', TokenType.NUM)
117     >>> tk4 = Token('+', TokenType.ADD)
118     >>> tk5 = Token('4', TokenType.NUM)
119     >>> tk6 = Token(')', TokenType.RPR)
120     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
121     >>> exp = parser.parse()
122     >>> ev = EvalVisitor()
123     >>> exp.accept(ev, None)
124     14
125
126     >>> tk0 = Token('4', TokenType.NUM)
127     >>> tk1 = Token('==', TokenType.EQL)
128     >>> tk2 = Token('4', TokenType.NUM)
129     >>> parser = Parser([tk0, tk1, tk2])
130     >>> exp = parser.parse()
131     >>> ev = EvalVisitor()
132     >>> exp.accept(ev, None)
133     True
134
135     >>> tk0 = Token('4', TokenType.NUM)
136     >>> tk1 = Token('<=', TokenType.LEQ)
137     >>> tk2 = Token('4', TokenType.NUM)
138     >>> parser = Parser([tk0, tk1, tk2])
139     >>> exp = parser.parse()
140     >>> ev = EvalVisitor()
141     >>> exp.accept(ev, None)
142     True
143
144     >>> tk0 = Token('4', TokenType.NUM)
145     >>> tk1 = Token('<', TokenType.LTH)
146     >>> tk2 = Token('4', TokenType.NUM)
147     >>> parser = Parser([tk0, tk1, tk2])
148     >>> exp = parser.parse()
149     >>> ev = EvalVisitor()
150     >>> exp.accept(ev, None)
151     False
152
153     >>> tk0 = Token('not', TokenType.NOT)
154     >>> tk1 = Token('(', TokenType.LPR)
155     >>> tk2 = Token('4', TokenType.NUM)
156     >>> tk3 = Token('<', TokenType.LTH)
157     >>> tk4 = Token('4', TokenType.NUM)
158     >>> tk5 = Token(')', TokenType.RPR)
159     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
160     >>> exp = parser.parse()
161     >>> ev = EvalVisitor()
162     >>> exp.accept(ev, None)
163     True
164
165     >>> tk0 = Token('true', TokenType.TRU)
166     >>> tk1 = Token('or', TokenType.ORX)
167     >>> tk2 = Token('false', TokenType.FLS)
168     >>> parser = Parser([tk0, tk1, tk2])
169     >>> exp = parser.parse()
170     >>> ev = EvalVisitor()
171     >>> exp.accept(ev, None)
172     True
173
174     >>> tk0 = Token('true', TokenType.TRU)
175     >>> tk1 = Token('and', TokenType.AND)
176     >>> tk2 = Token('false', TokenType.FLS)
177     >>> parser = Parser([tk0, tk1, tk2])
178     >>> exp = parser.parse()
179     >>> ev = EvalVisitor()
180     >>> exp.accept(ev, None)
181     False
182
183     >>> tk0 = Token('let', TokenType.LET)
184     >>> tk1 = Token('v', TokenType.VAR)
185     >>> tk2 = Token('<-', TokenType.ASN)
186     >>> tk3 = Token('42', TokenType.NUM)
187     >>> tk4 = Token('in', TokenType.INX)
188     >>> tk5 = Token('v', TokenType.VAR)
189     >>> tk6 = Token('end', TokenType.END)
190     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
191     >>> exp = parser.parse()
192     >>> ev = EvalVisitor()
193     >>> exp.accept(ev, {})
194     42
195
196     >>> tk0 = Token('let', TokenType.LET)
197     >>> tk1 = Token('v', TokenType.VAR)
198     >>> tk2 = Token('<-', TokenType.ASN)
199     >>> tk3 = Token('21', TokenType.NUM)
200     >>> tk4 = Token('in', TokenType.INX)
201     >>> tk5 = Token('v', TokenType.VAR)
202     >>> tk6 = Token('+', TokenType.ADD)
203     >>> tk7 = Token('v', TokenType.VAR)
204     >>> tk8 = Token('end', TokenType.END)
205     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
206     >>> exp = parser.parse()
207     >>> _..._

```

```
20/
208    >>> ev = EvalVisitor()
209    >>> exp.accept(ev, {})
210    42
211
212        tk0 = Token('if', TokenType.IFX)
213        tk1 = Token('2', TokenType.NUM)
214        tk2 = Token('<', TokenType.LTH)
215        tk3 = Token('3', TokenType.NUM)
216        tk4 = Token('then', TokenType.THN)
217        tk5 = Token('1', TokenType.NUM)
218        tk6 = Token('else', TokenType.ELS)
219        tk7 = Token('2', TokenType.NUM)
220
221        parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
222
223        >>> exp = parser.parse()
224        >>> ev = EvalVisitor()
225        >>> exp.accept(ev, None)
226
227        1
228
229        tk0 = Token('if', TokenType.IFX)
230        tk1 = Token('false', TokenType.FLS)
231        tk2 = Token('then', TokenType.THN)
232        tk3 = Token('1', TokenType.NUM)
233        tk4 = Token('else', TokenType.ELS)
234        tk5 = Token('2', TokenType.NUM)
235
236        parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
237
238        >>> exp = parser.parse()
239        >>> ev = EvalVisitor()
240        >>> exp.accept(ev, None)
241
242        2
243
244        """
245
246        # TODO: implement this method.
247        return None
```

Expression.py

```

1  from abc import ABC, abstractmethod
2  from Visitor import *
3
4  class Expression(ABC):
5      @abstractmethod
6      def accept(self, visitor, arg):
7          raise NotImplementedError
8
9  class Var(Expression):
10     """
11         This class represents expressions that are identifiers. The value of an
12         identifier is the value associated with it in the environment table.
13     """
14     def __init__(self, identifier):
15         self.identifier = identifier
16     def accept(self, visitor, arg):
17         return visitor.visit_var(self, arg)
18
19 class Bln(Expression):
20     """
21         This class represents expressions that are boolean values. There are only
22         two boolean values: true and false. The acceptuation of such an expression is
23         the boolean itself.
24     """
25     def __init__(self, bln):
26         self.bln = bln
27     def accept(self, visitor, arg):
28         return visitor.visit_bln(self, arg)
29
30 class Num(Expression):
31     """
32         This class represents expressions that are numbers. The acceptuation of such
33         an expression is the number itself.
34     """
35     def __init__(self, num):
36         self.num = num
37     def accept(self, visitor, arg):
38         return visitor.visit_num(self, arg)
39
40 class BinaryExpression(Expression):
41     """
42         This class represents binary expressions. A binary expression has two
43         sub-expressions: the left operand and the right operand.
44     """
45     def __init__(self, left, right):
46         self.left = left
47         self.right = right
48
49     @abstractmethod
50     def accept(self, visitor, arg):
51         raise NotImplementedError
52
53 class Eql(BinaryExpression):
54     """
55         This class represents the equality between two expressions. The acceptuation
56         of such an expression is True if the subexpressions are the same, or false
57         otherwise.
58     """
59     def accept(self, visitor, arg):
60         return visitor.visit.eql(self, arg)
61
62 class Add(BinaryExpression):
63     """
64         This class represents addition of two expressions. The acceptuation of such
65         an expression is the addition of the two subexpression's values.
66     """
67     def accept(self, visitor, arg):
68         return visitor.visit_add(self, arg)
69
70 class And(BinaryExpression):
71     """
72         This class represents the logical disjunction of two boolean expressions.
73         The evaluation of an expression of this kind is the logical AND of the two
74         subexpression's values.
75     """
76     def accept(self, visitor, arg):
77         return visitor.visit_and(self, arg)
78
79 class Or(BinaryExpression):
80     """
81         This class represents the logical conjunction of two boolean expressions.
82         The evaluation of an expression of this kind is the logical OR of the two
83         subexpression's values.
84     """
85     def accept(self, visitor, arg):
86         return visitor.visit_or(self, arg)
87
88 class Sub(BinaryExpression):
89     """
90         This class represents subtraction of two expressions. The acceptuation of such
91         an expression is the subtraction of the two subexpression's values.
92     """
93     def accept(self, visitor, arg):
94         return visitor.visit_sub(self, arg)
95
96 class Mul(BinaryExpression):
97     """
98         This class represents multiplication of two expressions. The acceptuation of
99         such an expression is the product of the two subexpression's values.
100    """
101   def accept(self, visitor, arg):
102       return visitor.visit_mul(self, arg)
103

```

```

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

```

Visitor.py

```

1 import sys
2 from abc import ABC, abstractmethod
3 from Expression import *
4 import Asm as AsmModule
5
6
7 class Visitor(ABC):
8 """
9     The visitor pattern consists of two abstract classes: the Expression and the
10    Visitor. The Expression class defines one method: 'accept(visitor, args)'.
11    This method takes in an implementation of a visitor, and the arguments that
12    are passed from expression to expression. The Visitor class defines one
13    specific method for each subclass of Expression. Each instance of such a
14    subclass will invoke the right visiting method.
15 """
16 @abstractmethod
17 def visit_var(self, exp, arg):
18     pass
19
20 @abstractmethod
21 def visit_bln(self, exp, arg):
22     pass
23
24 @abstractmethod
25 def visit_num(self, exp, arg):
26     pass
27
28 @abstractmethod
29 def visit_eql(self, exp, arg):
30     pass
31
32 @abstractmethod
33 def visit_and(self, exp, arg):
34     pass
35
36 @abstractmethod
37 def visit_or(self, exp, arg):
38     pass
39
40 @abstractmethod
41 def visit_add(self, exp, arg):
42     pass
43
44 @abstractmethod
45 def visit_sub(self, exp, arg):
46     pass
47
48 @abstractmethod
49 def visit_mul(self, exp, arg):
50     pass
51
52 @abstractmethod
53 def visit_div(self, exp, arg):
54     pass
55
56 @abstractmethod
57 def visit_leq(self, exp, arg):
58     pass
59
60 @abstractmethod
61 def visit_lth(self, exp, arg):
62     pass
63
64 @abstractmethod
65 def visit_neg(self, exp, arg):
66     pass
67
68 @abstractmethod
69 def visit_not(self, exp, arg):
70     pass
71
72 @abstractmethod
73 def visit_let(self, exp, arg):
74     pass
75
76 @abstractmethod
77 def visit_ifThenElse(self, exp, arg):
78     pass
79
80 class GenVisitor(Visitor):
81 """
82     The GenVisitor class compiles arithmetic expressions into a low-level
83     language.
84 """
85
86 def __init__(self):
87     self.next_var_counter = 0
88
89 def next_var_name(self):
90 """
91     You can use this method to get fresh variable names.
92 """
93     self.next_var_counter += 1
94     return f"v{self.next_var_counter}"
95
96 def visit_var(self, exp, prog):
97 """
98     Usage:
99         >>> e = Var('x')
100        >>> p = AsmModule.Program({"x":1}, [])
101        >>> g = GenVisitor()
102        >>> v = e.accept(g, p)
103        >>> p.eval()

```

```

104         >>> p.get_val(v)
105         1
106         """
107         # TODO: Implement this method.
108         raise NotImplementedError
109
110     def visit_bln(self, exp, prog):
111         """
112             Usage:
113             >>> e = Bln(True)
114             >>> p = AsmModule.Program({}, [])
115             >>> g = GenVisitor()
116             >>> v = e.accept(g, p)
117             >>> p.eval()
118             >>> p.get_val(v)
119             1
120
121             >>> e = Bln(False)
122             >>> p = AsmModule.Program({}, [])
123             >>> g = GenVisitor()
124             >>> v = e.accept(g, p)
125             >>> p.eval()
126             >>> p.get_val(v)
127             0
128         """
129         # TODO: Implement this method.
130         raise NotImplementedError
131
132     def visit_num(self, exp, prog):
133         """
134             Usage:
135             >>> e = Num(13)
136             >>> p = AsmModule.Program({}, [])
137             >>> g = GenVisitor()
138             >>> v = e.accept(g, p)
139             >>> p.eval()
140             >>> p.get_val(v)
141             13
142         """
143         # TODO: Implement this method.
144         raise NotImplementedError
145
146     def visit_eql(self, exp, prog):
147         """
148             >>> e = Eql(Num(13), Num(13))
149             >>> p = AsmModule.Program({}, [])
150             >>> g = GenVisitor()
151             >>> v = e.accept(g, p)
152             >>> p.eval()
153             >>> p.get_val(v)
154             1
155
156             >>> e = Eql(Num(13), Num(10))
157             >>> p = AsmModule.Program({}, [])
158             >>> g = GenVisitor()
159             >>> v = e.accept(g, p)
160             >>> p.eval()
161             >>> p.get_val(v)
162             0
163
164             >>> e = Eql(Num(-1), Num(1))
165             >>> p = AsmModule.Program({}, [])
166             >>> g = GenVisitor()
167             >>> v = e.accept(g, p)
168             >>> p.eval()
169             >>> p.get_val(v)
170             0
171         """
172         # TODO: Implement this method.
173         raise NotImplementedError
174
175     def visit_and(self, exp, prog):
176         """
177             >>> e = And(Bln(True), Bln(True))
178             >>> p = AsmModule.Program({}, [])
179             >>> g = GenVisitor()
180             >>> v = e.accept(g, p)
181             >>> p.eval()
182             >>> p.get_val(v)
183             1
184
185             >>> e = And(Bln(False), Bln(True))
186             >>> p = AsmModule.Program({}, [])
187             >>> g = GenVisitor()
188             >>> v = e.accept(g, p)
189             >>> p.eval()
190             >>> p.get_val(v)
191             0
192
193             >>> e = And(Bln(True), Bln(False))
194             >>> p = AsmModule.Program({}, [])
195             >>> g = GenVisitor()
196             >>> v = e.accept(g, p)
197             >>> p.eval()
198             >>> p.get_val(v)
199             0
200
201             >>> e = And(Bln(False), Bln(False))
202             >>> p = AsmModule.Program({}, [])
203             >>> g = GenVisitor()
204             >>> v = e.accept(g, p)
205             >>> p.eval()
206             >>> p.get_val(v)
207         """

```

```

208
209     >>> e = And(Bln(False), Div(Num(3), Num(0)))
210     >>> p = AsmModule.Program({}, [])
211     >>> g = GenVisitor()
212     >>> v = e.accept(g, p)
213     >>> p.eval()
214     >>> p.get_val(v)
215     0
216     """
217     # TODO: Implement this method.
218     raise NotImplementedError
219
220 def visit_or(self, exp, prog):
221     """
222     >>> e = Or(Bln(True), Bln(True))
223     >>> p = AsmModule.Program({}, [])
224     >>> g = GenVisitor()
225     >>> v = e.accept(g, p)
226     >>> p.eval()
227     >>> p.get_val(v)
228     1
229
230     >>> e = Or(Bln(False), Bln(True))
231     >>> p = AsmModule.Program({}, [])
232     >>> g = GenVisitor()
233     >>> v = e.accept(g, p)
234     >>> p.eval()
235     >>> p.get_val(v)
236     1
237
238     >>> e = Or(Bln(True), Bln(False))
239     >>> p = AsmModule.Program({}, [])
240     >>> g = GenVisitor()
241     >>> v = e.accept(g, p)
242     >>> p.eval()
243     >>> p.get_val(v)
244     1
245
246     >>> e = Or(Bln(False), Bln(False))
247     >>> p = AsmModule.Program({}, [])
248     >>> g = GenVisitor()
249     >>> v = e.accept(g, p)
250     >>> p.eval()
251     >>> p.get_val(v)
252     0
253
254     >>> e = Or(Bln(True), Div(Num(3), Num(0)))
255     >>> p = AsmModule.Program({}, [])
256     >>> g = GenVisitor()
257     >>> v = e.accept(g, p)
258     >>> p.eval()
259     >>> p.get_val(v)
260     1
261     """
262     # TODO: Implement this method.
263     raise NotImplementedError
264
265 def visit_add(self, exp, prog):
266     """
267     >>> e = Add(Num(13), Num(-13))
268     >>> p = AsmModule.Program({}, [])
269     >>> g = GenVisitor()
270     >>> v = e.accept(g, p)
271     >>> p.eval()
272     >>> p.get_val(v)
273     0
274
275     >>> e = Add(Num(13), Num(10))
276     >>> p = AsmModule.Program({}, [])
277     >>> g = GenVisitor()
278     >>> v = e.accept(g, p)
279     >>> p.eval()
280     >>> p.get_val(v)
281     23
282     """
283     # TODO: Implement this method.
284     raise NotImplementedError
285
286 def visit_sub(self, exp, prog):
287     """
288     >>> e = Sub(Num(13), Num(-13))
289     >>> p = AsmModule.Program({}, [])
290     >>> g = GenVisitor()
291     >>> v = e.accept(g, p)
292     >>> p.eval()
293     >>> p.get_val(v)
294     26
295
296     >>> e = Sub(Num(13), Num(10))
297     >>> p = AsmModule.Program({}, [])
298     >>> g = GenVisitor()
299     >>> v = e.accept(g, p)
300     >>> p.eval()
301     >>> p.get_val(v)
302     3
303     """
304     # TODO: Implement this method.
305     raise NotImplementedError
306
307 def visit_mul(self, exp, prog):
308     """
309     >>> e = Mul(Num(13), Num(2))
310     >>> p = AsmModule.Program([], [])
311

```

```

310
311     """ p = AsmModule.Program([], [])
312     >>> g = GenVisitor()
313     >>> v = e.accept(g, p)
314     >>> p.eval()
315     >>> p.get_val(v)
316     26
317
318     >>> e = Mul(Num(13), Num(10))
319     >>> p = AsmModule.Program({}, [])
320     >>> g = GenVisitor()
321     >>> v = e.accept(g, p)
322     >>> p.eval()
323     >>> p.get_val(v)
324     130
325     """
326
327     # TODO: Implement this method.
328     raise NotImplementedError
329
330     def visit_div(self, exp, prog):
331         """
332             >>> e = Div(Num(13), Num(2))
333             >>> p = AsmModule.Program({}, [])
334             >>> g = GenVisitor()
335             >>> v = e.accept(g, p)
336             >>> p.eval()
337             >>> p.get_val(v)
338             6
339
340             >>> e = Div(Num(13), Num(10))
341             >>> p = AsmModule.Program({}, [])
342             >>> g = GenVisitor()
343             >>> v = e.accept(g, p)
344             >>> p.eval()
345             >>> p.get_val(v)
346             1
347             """
348
349     # TODO: Implement this method.
350     raise NotImplementedError
351
352     def visit_leq(self, exp, prog):
353         """
354             >>> e = Leq(Num(3), Num(2))
355             >>> p = AsmModule.Program({}, [])
356             >>> g = GenVisitor()
357             >>> v = e.accept(g, p)
358             >>> p.eval()
359             >>> p.get_val(v)
360             0
361
362             >>> e = Leq(Num(3), Num(3))
363             >>> p = AsmModule.Program({}, [])
364             >>> g = GenVisitor()
365             >>> v = e.accept(g, p)
366             >>> p.eval()
367             >>> p.get_val(v)
368             1
369
370             >>> e = Leq(Num(2), Num(3))
371             >>> p = AsmModule.Program({}, [])
372             >>> g = GenVisitor()
373             >>> v = e.accept(g, p)
374             >>> p.eval()
375             >>> p.get_val(v)
376             1
377
378             >>> e = Leq(Num(-3), Num(-2))
379             >>> p = AsmModule.Program({}, [])
380             >>> g = GenVisitor()
381             >>> v = e.accept(g, p)
382             >>> p.eval()
383             >>> p.get_val(v)
384             1
385
386             >>> e = Leq(Num(-3), Num(-3))
387             >>> p = AsmModule.Program({}, [])
388             >>> g = GenVisitor()
389             >>> v = e.accept(g, p)
390             >>> p.eval()
391             >>> p.get_val(v)
392             0
393             """
394
395             # TODO: Implement this method.
396             raise NotImplementedError
397
398     def visit_lth(self, exp, prog):
399         """
400             >>> e = Lth(Num(3), Num(2))
401             >>> p = AsmModule.Program({}, [])
402             >>> g = GenVisitor()
403             >>> v = e.accept(g, p)
404             >>> p.eval()
405             >>> p.get_val(v)
406             0
407
408             >>> e = Lth(Num(3), Num(3))
409             >>> p = AsmModule.Program({}, [])
410             >>> g = GenVisitor()
411             >>> v = e.accept(g, p)
412             >>> p.eval()
413             >>> p.get_val(v)
414             1

```

```

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

```

```

517         >>> p = AsmModule.Program({}, [])
518         >>> g = GenVisitor()
519         >>> v = e.accept(g, p)
520         >>> p.eval()
521         >>> p.get_val(v)
522         5
523
524         >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
525         >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
526         >>> p = AsmModule.Program({}, [])
527         >>> g = GenVisitor()
528         >>> v = e1.accept(g, p)
529         >>> p.eval()
530         >>> p.get_val(v)
531         50
532
533         # TODO: Implement this method.
534         raise NotImplementedError
535
536     def visit_ifThenElse(self, exp, prog):
537         """
538             >>> e = IfThenElse(Bln(True), Num(3), Num(5))
539             >>> p = AsmModule.Program({}, [])
540             >>> g = GenVisitor()
541             >>> v = e.accept(g, p)
542             >>> p.eval()
543             >>> p.get_val(v)
544             3
545
546             >>> e = IfThenElse(Bln(False), Num(3), Num(5))
547             >>> p = AsmModule.Program({}, [])
548             >>> g = GenVisitor()
549             >>> v = e.accept(g, p)
550             >>> p.eval()
551             >>> p.get_val(v)
552             5
553
554             >>> e = IfThenElse(And(Bln(True), Bln(True)), Num(3), Num(5))
555             >>> p = AsmModule.Program({}, [])
556             >>> g = GenVisitor()
557             >>> v = e.accept(g, p)
558             >>> p.eval()
559             >>> p.get_val(v)
560             3
561
562             >>> e0 = Mul(Num(2), Add(Num(3), Num(4)))
563             >>> e1 = IfThenElse(And(Bln(True), Bln(False)), Num(3), e0)
564             >>> p = AsmModule.Program({}, [])
565             >>> g = GenVisitor()
566             >>> v = e1.accept(g, p)
567             >>> p.eval()
568             >>> p.get_val(v)
569             14
570
571             >>> e0 = Div(Num(2), Num(0))
572             >>> e1 = IfThenElse(Bln(True), Num(3), e0)
573             >>> p = AsmModule.Program({}, [])
574             >>> g = GenVisitor()
575             >>> v = e1.accept(g, p)
576             >>> p.eval()
577             >>> p.get_val(v)
578             3
579
580             >>> e0 = Div(Num(2), Num(0))
581             >>> e1 = IfThenElse(Bln(False), e0, Num(3))
582             >>> p = AsmModule.Program({}, [])
583             >>> g = GenVisitor()
584             >>> v = e1.accept(g, p)
585             >>> p.eval()
586             >>> p.get_val(v)
587             3
588
589             # TODO: Implement this method.
590             raise NotImplementedError
591
592
593     class RenameVisitor(ABC):
594         """
595             This visitor traverses the AST of a program, renaming variables to ensure
596             that they all have different names.
597
598             Usage:
599                 >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
600                 >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
601                 >>> e0.identifier == e1.identifier
602                 True
603
604                 >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
605                 >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
606                 >>> r = RenameVisitor()
607                 >>> e1.accept(r, {})
608                 >>> e0.identifier == e1.identifier
609                 False
610
611                 >>> x0 = Var('x')
612                 >>> x1 = Var('x')
613                 >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
614                 >>> e1 = Let('x', e0, Mul(x1, Num(10)))
615                 >>> x0.identifier == x1.identifier
616                 True
617
618                 >>> x0 = Var('x')
619                 >>> x1 = Var('x')

```

```

620      >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
621      >>> e1 = Let('x', e0, Mul(x1, Num(10)))
622      >>> r = RenameVisitor()
623      >>> e1.accept(r, {})
624      >>> x0.identifier == x1.identifier
625      False
626
627
628  def __init__(self):
629      # TODO: implement something here.
630      pass
631
632  def visit_var(self, exp, name_map):
633      # TODO: Implement this method.
634      raise NotImplementedError
635
636  def visit_bln(self, exp, name_map):
637      # TODO: Implement this method.
638      raise NotImplementedError
639
640  def visit_num(self, exp, name_map):
641      # TODO: Implement this method.
642      raise NotImplementedError
643
644  def visit_eql(self, exp, name_map):
645      # TODO: Implement this method.
646      raise NotImplementedError
647
648  def visit_and(self, exp, name_map):
649      """
650      Example:
651      >>> y0 = Var('x')
652      >>> y1 = Var('x')
653      >>> x0 = And(Lth(y0, Num(2)), Leq(Num(2), y1))
654      >>> x1 = Var('x')
655      >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
656      >>> e1 = Let('x', e0, Mul(x1, Num(10)))
657      >>> r = RenameVisitor()
658      >>> e1.accept(r, {})
659      >>> y0.identifier == y1.identifier
660      True
661
662      >>> y0 = Var('x')
663      >>> y1 = Var('x')
664      >>> x0 = And(Lth(y0, Num(2)), Leq(Num(2), y1))
665      >>> x1 = Var('x')
666      >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
667      >>> e1 = Let('x', e0, Mul(x1, Num(10)))
668      >>> r = RenameVisitor()
669      >>> e1.accept(r, {})
670      >>> y0.identifier == x1.identifier
671      False
672
673      # TODO: Implement this method.
674      raise NotImplementedError
675
676  def visit_or(self, exp, name_map):
677      """
678      Example:
679      >>> y0 = Var('x')
680      >>> y1 = Var('x')
681      >>> x0 = Or(Lth(y0, Num(2)), Leq(Num(2), y1))
682      >>> x1 = Var('x')
683      >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
684      >>> e1 = Let('x', e0, Mul(x1, Num(10)))
685      >>> r = RenameVisitor()
686      >>> e1.accept(r, {})
687      >>> y0.identifier == y1.identifier
688      True
689
690      >>> y0 = Var('x')
691      >>> y1 = Var('x')
692      >>> x0 = Or(Lth(y0, Num(2)), Leq(Num(2), y1))
693      >>> x1 = Var('x')
694      >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
695      >>> e1 = Let('x', e0, Mul(x1, Num(10)))
696      >>> r = RenameVisitor()
697      >>> e1.accept(r, {})
698      >>> y0.identifier == x1.identifier
699      False
700
701      # TODO: Implement this method.
702      raise NotImplementedError
703
704  def visit_add(self, exp, name_map):
705      # TODO: Implement this method.
706      raise NotImplementedError
707
708  def visit_sub(self, exp, name_map):
709      # TODO: Implement this method.
710      raise NotImplementedError
711
712  def visit_mul(self, exp, name_map):
713      # TODO: Implement this method.
714      raise NotImplementedError
715
716  def visit_div(self, exp, name_map):
717      # TODO: Implement this method.
718      raise NotImplementedError
719
720  def visit_leq(self, exp, name_map):
721      # TODO: Implement this method.
722      raise NotImplementedError
723

```

```

724     def visit_lth(self, exp, name_map):
725         # TODO: Implement this method.
726         raise NotImplementedError
727
728     def visit_neg(self, exp, name_map):
729         # TODO: Implement this method.
730         raise NotImplementedError
731
732     def visit_not(self, exp, name_map):
733         # TODO: Implement this method.
734         raise NotImplementedError
735
736     def visit_ifThenElse(self, exp, name_map):
737         """
738             Example:
739             >>> x0 = Var('x')
740             >>> x1 = Var('x')
741             >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
742             >>> e1 = Let('x', Num(3), e0)
743             >>> r = RenameVisitor()
744             >>> e1.accept(r, {})
745             >>> x0.identifier == x1.identifier
746             True
747
748             >>> x0 = Var('x')
749             >>> x1 = Var('x')
750             >>> e0 = IfThenElse(Lth(x0, x1), Num(1), Num(2))
751             >>> e1 = Let('x', Num(3), e0)
752             >>> e2 = Let('x', e1, Num(3))
753             >>> r = RenameVisitor()
754             >>> e1.accept(r, {})
755             >>> e2.identifier != x1.identifier == e1.identifier
756             True
757         """
758         # TODO: Implement this method.
759         raise NotImplementedError
760
761     def visit_let(self, exp, name_map):
762         # TODO: Implement this method.
763         raise NotImplementedError

```

Asm.py

```

1 """
2 This file contains the implementation of a simple interpreter of low-level
3 instructions. The interpreter takes a program, represented as an array of
4 instructions, plus an environment, which is a map that associates variables with
5 values. The following instructions are recognized:
6
7     * add rd, rs1, rs2: rd = rs1 + rs2
8     * addi rd, rs1, imm: rd = rs1 + imm
9     * mul rd, rs1, rs2: rd = rs1 * rs2
10    * sub rd, rs1, rs2: rd = rs1 - rs2
11    * xor rd, rs1, rs2: rd = rs1 ^ rs2
12    * xori rd, rs1, imm: rd = rs1 ^ imm
13    * div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
14    * slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
15    * slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
16    * beq rs1, rs2, lab: pc = lab if rs1 == rs2 else pc + 1
17    * jal rd, lab: rd = pc + 1 and pc = lab
18
19 This file uses doctests all over. To test it, just run python 3 as follows:
20 "python3 -m doctest Asm.py". The program uses syntax that is exclusive of
21 Python 3. It will not work with standard Python 2.
22 """
23
24 import sys
25 from collections import deque
26 from abc import ABC, abstractmethod
27
28
29 class Program:
30     """
31     The 'Program' is a list of instructions plus an environment that associates
32     names with values, plus a program counter, which marks the next instruction
33     that must be executed. The environment contains a special variable x0,
34     which always contains the value zero.
35     """
36
37     def __init__(self, env, insts):
38         self.__env = env
39         self.__insts = insts
40         self.pc = 0
41         self.__env["x0"] = 0
42
43     def get_inst(self):
44         if self.pc >= 0 and self.pc < len(self.__insts):
45             inst = self.__insts[self.pc]
46             self.pc += 1
47             return inst
48         else:
49             return None
50
51     def get_number_of_instructions(self):
52         return len(self.__insts)
53
54     def add_inst(self, inst):
55         self.__insts.append(inst)
56
57     def get_pc(self):
58         return self.pc
59
60     def set_pc(self, pc):
61         self.pc = pc
62
63     def set_val(self, name, value):
64         if name != "x0": # Can't change x0, which is always zero.
65             self.__env[name] = value
66
67     def get_val(self, name):
68         """
69             The register x0 always contains the value zero.
70
71             >>> p = Program({}, [])
72             >>> p.get_val("x0")
73             0
74             """
75         if name in self.__env:
76             return self.__env[name]
77         else:
78             sys.exit("Def error")
79
80     def print_env(self):
81         for name, val in sorted(self.__env.items()):
82             print(f"{name}: {val}")
83
84     def print_insts(self):
85         counter = 0
86         for inst in self.__insts:
87             print("%03d: %s" % (counter, str(inst)))
88             counter += 1
89         print("%03d: %s" % (counter, "END"))
90
91     def eval(self):
92         """
93             This function evaluates a program until there is no more instructions to
94             evaluate.
95
96             Example:
97                 >>> insts = [Add("t0", "b0", "b1"), Sub("x1", "t0", "b2")]
98                 >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
99                 >>> p.eval()
100                >>> p.print_env()
101                b0: 2
102                b1: 3
103                b2: 4

```

```

104     t0: 5
105     x0: 0
106     x1: 1
107
108     Notice that it is not possible to change 'x0':
109     >>> insts = [Add("x0", "b0", "b1")]
110     >>> p = Program({"b0":2, "b1":3}, insts)
111     >>> p.eval()
112     >>> p.print_env()
113     b0: 2
114     b1: 3
115     x0: 0
116
117     """
118     inst = self.get_inst()
119     while inst:
120         inst.eval(self)
121         inst = self.get_inst()
122
123 def max(a, b):
124     """
125     This example computes the maximum between a and b.
126
127     Example:
128     >>> max(2, 3)
129     3
130
131     >>> max(3, 2)
132     3
133
134     >>> max(-3, -2)
135     -2
136
137     >>> max(-2, -3)
138     -2
139
140     p = Program({}, [])
141     p.set_val("rs1", a)
142     p.set_val("rs2", b)
143     p.add_inst(Slt("t0", "rs2", "rs1"))
144     p.add_inst(Slt("t1", "rs1", "rs2"))
145     p.add_inst(Mul("t0", "t0", "rs1"))
146     p.add_inst(Mul("t1", "t1", "rs2"))
147     p.add_inst(Add("rd", "t0", "t1"))
148     p.eval()
149     return p.get_val("rd")
150
151
152 def distance_with_acceleration(V, A, T):
153     """
154     This example computes the position of an object, given its velocity (V),
155     its acceleration (A) and the time (T), assuming that it starts at position
156     zero, using the formula D = V*T + (A*T^2)/2.
157
158     Example:
159     >>> distance_with_acceleration(3, 4, 5)
160     65
161
162     p = Program({}, [])
163     p.set_val("rs1", V)
164     p.set_val("rs2", A)
165     p.set_val("rs3", T)
166     p.add_inst(Addi("two", "x0", 2))
167     p.add_inst(Mul("t0", "rs1", "rs3"))
168     p.add_inst(Mul("t1", "rs3", "rs3"))
169     p.add_inst(Mul("t2", "rs2", "t1"))
170     p.add_inst(Div("t2", "t2", "two"))
171     p.add_inst(Add("rd", "t0", "t2"))
172     p.eval()
173     return p.get_val("rd")
174
175
176 class Inst(ABC):
177     """
178     The representation of instructions. Every instruction refers to a program
179     during its evaluation.
180
181
182     def __init__(self):
183         pass
184
185     @abstractmethod
186     def get_opcode(self):
187         raise NotImplementedError
188
189     @abstractmethod
190     def eval(self, prog):
191         raise NotImplementedError
192
193
194 class BranchOp(Inst):
195     """
196     The general class of branching instructions. These instructions can change
197     the control flow of a program. Normally, the next instruction is given by
198     pc + 1. A branch might change pc to point out to a different label..
199
200
201     def set_target(self, lab):
202         assert(isinstance(lab, int))
203         self.lab = lab
204
205
206     class Beq(BranchOp):
207         """

```

```

208     beq rs1, rs2, lab:
209     Jumps to label lab if the value in rs1 is equal to the value in rs2.
210
211
212     def __init__(self, rs1, rs2, lab=None):
213         assert isinstance(rs1, str) and isinstance(rs2, str)
214         self.rs1 = rs1
215         self.rs2 = rs2
216         if lab != None:
217             assert isinstance(lab, int)
218             self.lab = lab
219
220     def get_opcode(self):
221         return "beq"
222
223     def __str__(self):
224         op = self.get_opcode()
225         return f"{op} {self.rs1} {self.rs2} {self.lab}"
226
227     def eval(self, prog):
228         if prog.get_val(self.rs1) == prog.get_val(self.rs2):
229             prog.set_pc(self.lab)
230
231
232 class Jal(BranchOp):
233     """
234     jal rd lab:
235     Stores the return address (PC+1) on register rd, then jumps to label lab.
236     If rd is x0, then it does not write on the register. In this case, notice
237     that `jal x0 lab` is equivalent to an unconditional jump to `lab`.
238     """
239
240     def __init__(self, rd, lab=None):
241         assert isinstance(rd, str)
242         self.rd = rd
243         if lab != None:
244             assert isinstance(lab, int)
245             self.lab = lab
246
247     def get_opcode(self):
248         return "jal"
249
250     def __str__(self):
251         op = self.get_opcode()
252         return f"{op} {self.rd} {self.lab}"
253
254     def eval(self, prog):
255         if self.rd != "x0":
256             self.rd = prog.get_pc + 1
257             prog.set_pc(self.lab)
258
259
260 class BinOp(Inst):
261     """
262     The general class of binary instructions. These instructions define a
263     value, and use two values.
264     """
265
266     def __init__(self, rd, rs1, rs2):
267         assert isinstance(rd, str) and isinstance(rs1, str) and \
268             isinstance(rs2, str)
269         self.rd = rd
270         self.rs1 = rs1
271         self.rs2 = rs2
272
273     def __str__(self):
274         op = self.get_opcode()
275         return f"{self.rd} = {op} {self.rs1} {self.rs2}"
276
277
278 class BinOpImm(Inst):
279     """
280     The general class of binary instructions where the second operand is an
281     integer constant. These instructions define a value, and use one variable
282     and one immediate constant.
283     """
284
285     def __init__(self, rd, rs1, imm):
286         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
287         self.rd = rd
288         self.rs1 = rs1
289         self.imm = imm
290
291     def __str__(self):
292         op = self.get_opcode()
293         return f"{self.rd} = {op} {self.rs1} {self.imm}"
294
295
296 class Add(BinOp):
297     """
298     add rd, rs1, rs2: rd = rs1 + rs2
299
300     Example:
301         >>> i = Add("a", "b0", "b1")
302         >>> str(i)
303         'a = add b0 b1'
304
305         >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
306         >>> p.eval()
307         >>> p.get_val("a")
308         5
309     """

```

```

311     def eval(self, prog):
312         rs1 = prog.get_val(self.rs1)
313         rs2 = prog.get_val(self.rs2)
314         prog.set_val(self.rd, rs1 + rs2)
315
316     def get_opcode(self):
317         return "add"
318
319
320 class Addi(BinOpImm):
321     """
322     addi rd, rs1, imm: rd = rs1 + imm
323
324     Example:
325         >>> i = Addi("a", "b0", 1)
326         >>> str(i)
327         'a = addi b0 1'
328
329         >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
330         >>> p.eval()
331         >>> p.get_val("a")
332         5
333     """
334
335     def eval(self, prog):
336         rs1 = prog.get_val(self.rs1)
337         prog.set_val(self.rd, rs1 + self.imm)
338
339     def get_opcode(self):
340         return "addi"
341
342
343 class Mul(BinOp):
344     """
345     mul rd, rs1, rs2: rd = rs1 * rs2
346
347     Example:
348         >>> i = Mul("a", "b0", "b1")
349         >>> str(i)
350         'a = mul b0 b1'
351
352         >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
353         >>> p.eval()
354         >>> p.get_val("a")
355         6
356     """
357
358     def eval(self, prog):
359         rs1 = prog.get_val(self.rs1)
360         rs2 = prog.get_val(self.rs2)
361         prog.set_val(self.rd, rs1 * rs2)
362
363     def get_opcode(self):
364         return "mul"
365
366
367 class Sub(BinOp):
368     """
369     sub rd, rs1, rs2: rd = rs1 - rs2
370
371     Example:
372         >>> i = Sub("a", "b0", "b1")
373         >>> str(i)
374         'a = sub b0 b1'
375
376         >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
377         >>> p.eval()
378         >>> p.get_val("a")
379         -1
380     """
381
382     def eval(self, prog):
383         rs1 = prog.get_val(self.rs1)
384         rs2 = prog.get_val(self.rs2)
385         prog.set_val(self.rd, rs1 - rs2)
386
387     def get_opcode(self):
388         return "sub"
389
390
391 class Xor(BinOp):
392     """
393     xor rd, rs1, rs2: rd = rs1 ^ rs2
394
395     Example:
396         >>> i = Xor("a", "b0", "b1")
397         >>> str(i)
398         'a = xor b0 b1'
399
400         >>> p = Program(env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
401         >>> p.eval()
402         >>> p.get_val("a")
403         1
404     """
405
406     def eval(self, prog):
407         rs1 = prog.get_val(self.rs1)
408         rs2 = prog.get_val(self.rs2)
409         prog.set_val(self.rd, rs1 ^ rs2)
410
411     def get_opcode(self):
412         return "xor"
413

```

```

414
415     class Xori(BinOpImm):
416         """
417             xori rd, rs1, imm: rd = rs1 ^ imm
418
419         Example:
420             >>> i = Xori("a", "b0", 10)
421             >>> str(i)
422             'a = xori b0 10'
423
424             >>> p = Program(env={"b0":2}, insts=[Xori("a", "b0", 3)])
425             >>> p.eval()
426             >>> p.get_val("a")
427             1
428
429
430         def eval(self, prog):
431             rs1 = prog.get_val(self.rs1)
432             prog.set_val(self.rd, rs1 ^ self.imm)
433
434         def get_opcode(self):
435             return "xori"
436
437
438     class Div(BinOp):
439         """
440             div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
441             Notice that RISC-V does not have an instruction exactly like this one.
442             The div operator works on floating-point numbers; not on integers.
443
444         Example:
445             >>> i = Div("a", "b0", "b1")
446             >>> str(i)
447             'a = div b0 b1'
448
449             >>> p = Program(env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
450             >>> p.eval()
451             >>> p.get_val("a")
452             2
453
454
455         def eval(self, prog):
456             rs1 = prog.get_val(self.rs1)
457             rs2 = prog.get_val(self.rs2)
458             prog.set_val(self.rd, rs1 // rs2)
459
460         def get_opcode(self):
461             return "div"
462
463
464     class Slt(BinOp):
465         """
466             slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
467
468         Example:
469             >>> i = Slt("a", "b0", "b1")
470             >>> str(i)
471             'a = slt b0 b1'
472
473             >>> p = Program(env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
474             >>> p.eval()
475             >>> p.get_val("a")
476             1
477
478             >>> p = Program(env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
479             >>> p.eval()
480             >>> p.get_val("a")
481             0
482
483             >>> p = Program(env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
484             >>> p.eval()
485             >>> p.get_val("a")
486             0
487
488
489         def eval(self, prog):
490             rs1 = prog.get_val(self.rs1)
491             rs2 = prog.get_val(self.rs2)
492             prog.set_val(self.rd, 1 if rs1 < rs2 else 0)
493
494         def get_opcode(self):
495             return "slt"
496
497
498     class Slti(BinOpImm):
499         """
500             slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
501             (signed comparison with immediate)
502
503         Example:
504             >>> i = Slti("a", "b0", 0)
505             >>> str(i)
506             'a = slti b0 0'
507
508             >>> p = Program(env={"b0":2}, insts=[Slti("a", "b0", 3)])
509             >>> p.eval()
510             >>> p.get_val("a")
511             1
512
513             >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 3)])
514             >>> p.eval()
515             >>> p.get_val("a")
516             0

```

```
517      518      >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 2)])
519      520      >>> p.eval()
521      522      >>> p.get_val("a")
523      524      0
525      526      """
527      528      def eval(self, prog):
529      529          rs1 = prog.get_val(self.rs1)
530          prog.set_val(self.rd, 1 if rs1 < self.imm else 0)
531
532      533      def get_opcode(self):
534          return "slti"
```

[VPL](#)