

2025_2 - COMPILADORES - METATURMA

PAINEL > **MINHAS TURMAS** > **2025_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**
 > **AV11 - CÓDIGO PARA EXPRESSÕES ARITMÉTICAS**

Descrição

Visualizar envios

AV11 - Código para expressões aritméticas

Data de entrega: quarta, 29 Out 2025, 23:59

Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py (Baixar)

Tipo de trabalho: Trabalho individual

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

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

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

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

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

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

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

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

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

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

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

Cada método visit retorna o nome da variável que irá armazenar o valor da expressão. Pense neste nome como um "endereço de memória". Em um próximo lab iremos implementar endereços realmente. Por hora, podemos assumir a memória é endereçável pelo nome de variáveis.

```
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
```

Figura 3: Implementação de geração de código para variáveis.

A classe GenVisitor possui um método para criar novos nomes de variáveis: GenVisitor.next_var_name(). Você pode usá-lo sempre que precisar de armazenar um novo valor.

```
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
    """
    l_name = exp.left.accept(self, prog)
    r_name = exp.right.accept(self, prog)
    v_name = self.next_var_name()
    prog.add_inst(AsmModule.Add(v_name, l_name, r_name))
    return v_name
```

Figura 4: Implementação de geração de código para adição.

Observação sobre o nome de variáveis em blocos let

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

Submetendo e Testando

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

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

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

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

No exemplo acima, substitua `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 if __name__ == "__main__":
9 """
10 Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
11 VPL. O arquivo contem o codigo que testa a implementacao do parser.
12 """
13 text = sys.stdin.read()
14 lexer = Lexer(text)
15 parser = Parser(lexer.tokens())
16 exp = parser.parse()
17 prog = AsmModule.Program({}, [])
18 gen = GenVisitor()
19 var_answer = exp.accept(gen, prog)
20 prog.eval()
21 print(f"{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
24     EOF = -1 # End of file
25     NLN = 0 # New line
26     WSP = 1 # White Space
27     COM = 2 # Comment
28     NUM = 3 # Number (integers)
29     STR = 4 # Strings
30     TRU = 5 # The constant true
31     FLS = 6 # The constant false
32     VAR = 7 # An identifier
33     LET = 8 # The 'let' of the let expression
34     INX = 9 # The 'in' of the let expression
35     END = 10 # The 'end' of the let expression
36     EQL = 201
37     ADD = 202
38     SUB = 203
39     MUL = 204
40     DIV = 205
41     LEQ = 206
42     LTH = 207
43     NEG = 208
44     NOT = 209
45     LPR = 210
46     RPR = 211
47     ASN = 212 # The assignment '<->' operator
48
49
50 class Lexer:
51
52     def __init__(self, source):
53
54         """
55             The constructor of the lexer. It receives the string that shall be
56             scanned.
57             TODO: You will need to implement this method.
58             """
59
60         pass
61
62     def tokens(self):
63
64         """
65             This method is a token generator: it converts the string encapsulated
66             into this object into a sequence of Tokens. Examples:
67
68             >>> l = Lexer("1 + 3")
69             >>> [tk.kind for tk in l.tokens()]
70             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
71
72             >>> l = Lexer('1 * 2 -- 3\n')
73             >>> [tk.kind for tk in l.tokens()]
74             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
75
76             >>> l = Lexer("1 + var")
77             >>> [tk.kind for tk in l.tokens()]
78             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
79
80             >>> l = Lexer("let v <- 2 in v end")
81             >>> [tk.kind.name for tk in l.tokens()]
82             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
83
84         token = self.getToken()
85         while token.kind != TokenType.EOF:
86             if (
87                 token.kind != TokenType.WSP
88                 and token.kind != TokenType.COM
89                 and token.kind != TokenType.NLN
90             ):
91                 yield token
92             token = self.getToken()
93
94     def getToken(self):
95
96         """
97             Return the next token.
98             TODO: Implement this method (you can reuse Lab 5: Visitors)!
99             """
100
101         token = None
102         return token

```

Parser.py

```

1 import sys
2
3 from Expression import *
4 from Lexer import Token, TokenType
5
6 """
7 This file implements the parser of arithmetic expressions. The same rules of
8 precedence and associativity from Lab 5: Visitors, apply.
9
10 References:
11     see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#classic
12 """
13
14 class Parser:
15     def __init__(self, tokens):
16         """
17             Initializes the parser. The parser keeps track of the list of tokens
18             and the current token. For instance:
19         """
20         self.tokens = list(tokens)
21         self.cur_token_idx = 0 # This is just a suggestion!
22         # You can (and probably should!) modify this method.
23
24     def parse(self):
25         """
26             Returns the expression associated with the stream of tokens.
27
28 Examples:
29 >>> parser = Parser([Token('123', TokenType.NUM)])
30 >>> g = GenVisitor()
31 >>> p = AsmModule.Program({}, [])
32 >>> exp = parser.parse()
33 >>> v = exp.accept(g, p)
34 >>> p.eval()
35 >>> p.get_val(v)
36 123
37
38 >>> parser = Parser([Token('True', TokenType.TRU)])
39 >>> g = GenVisitor()
40 >>> p = AsmModule.Program({}, [])
41 >>> exp = parser.parse()
42 >>> v = exp.accept(g, p)
43 >>> p.eval()
44 >>> p.get_val(v)
45 1
46
47 >>> parser = Parser([Token('False', TokenType.FLS)])
48 >>> g = GenVisitor()
49 >>> p = AsmModule.Program({}, [])
50 >>> exp = parser.parse()
51 >>> v = exp.accept(g, p)
52 >>> p.eval()
53 >>> p.get_val(v)
54 0
55
56 >>> tk0 = Token('~', TokenType.NEG)
57 >>> tk1 = Token('123', TokenType.NUM)
58 >>> parser = Parser([tk0, tk1])
59 >>> g = GenVisitor()
60 >>> p = AsmModule.Program({}, [])
61 >>> exp = parser.parse()
62 >>> v = exp.accept(g, p)
63 >>> p.eval()
64 >>> p.get_val(v)
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 >>> g = GenVisitor()
72 >>> p = AsmModule.Program({}, [])
73 >>> exp = parser.parse()
74 >>> v = exp.accept(g, p)
75 >>> p.eval()
76 >>> p.get_val(v)
77 12
78
79 >>> tk0 = Token('3', TokenType.NUM)
80 >>> tk1 = Token('*', TokenType.MUL)
81 >>> tk2 = Token('~', TokenType.NEG)
82 >>> tk3 = Token('4', TokenType.NUM)
83 >>> parser = Parser([tk0, tk1, tk2, tk3])
84 >>> g = GenVisitor()
85 >>> p = AsmModule.Program({}, [])
86 >>> exp = parser.parse()
87 >>> v = exp.accept(g, p)
88 >>> p.eval()
89 >>> p.get_val(v)
90 -12
91
92 >>> tk0 = Token('30', TokenType.NUM)
93 >>> tk1 = Token('/', TokenType.DIV)
94 >>> tk2 = Token('4', TokenType.NUM)
95 >>> parser = Parser([tk0, tk1, tk2])
96 >>> g = GenVisitor()
97 >>> p = AsmModule.Program({}, [])
98 >>> exp = parser.parse()
99 >>> v = exp.accept(g, p)
100 >>> p.eval()
101 >>> p.get_val(v)
102 7
103

```

```

104      >>> tk0 = Token('3', TokenType.NUM)
105      >>> tk1 = Token('+', TokenType.ADD)
106      >>> tk2 = Token('4', TokenType.NUM)
107      >>> parser = Parser([tk0, tk1, tk2])
108      >>> g = GenVisitor()
109      >>> p = AsmModule.Program({}, [])
110      >>> exp = parser.parse()
111      >>> v = exp.accept(g, p)
112      >>> p.eval()
113      >>> p.get_val(v)
114      7
115
116      >>> tk0 = Token('30', TokenType.NUM)
117      >>> tk1 = Token('-', TokenType.SUB)
118      >>> tk2 = Token('4', TokenType.NUM)
119      >>> parser = Parser([tk0, tk1, tk2])
120      >>> g = GenVisitor()
121      >>> p = AsmModule.Program({}, [])
122      >>> exp = parser.parse()
123      >>> v = exp.accept(g, p)
124      >>> p.eval()
125      >>> p.get_val(v)
126      26
127
128      >>> tk0 = Token('2', TokenType.NUM)
129      >>> tk1 = Token('*', TokenType.MUL)
130      >>> tk2 = Token('(', TokenType.LPR)
131      >>> tk3 = Token('3', TokenType.NUM)
132      >>> tk4 = Token('+', TokenType.ADD)
133      >>> tk5 = Token('4', TokenType.NUM)
134      >>> tk6 = Token(')', TokenType.RPR)
135      >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
136      >>> g = GenVisitor()
137      >>> p = AsmModule.Program({}, [])
138      >>> exp = parser.parse()
139      >>> v = exp.accept(g, p)
140      >>> p.eval()
141      >>> p.get_val(v)
142      14
143
144      >>> tk0 = Token('4', TokenType.NUM)
145      >>> tk1 = Token('==', TokenType.EQL)
146      >>> tk2 = Token('4', TokenType.NUM)
147      >>> parser = Parser([tk0, tk1, tk2])
148      >>> g = GenVisitor()
149      >>> p = AsmModule.Program({}, [])
150      >>> exp = parser.parse()
151      >>> v = exp.accept(g, p)
152      >>> p.eval()
153      >>> p.get_val(v)
154      1
155
156      >>> tk0 = Token('4', TokenType.NUM)
157      >>> tk1 = Token('<=', TokenType.LEQ)
158      >>> tk2 = Token('4', TokenType.NUM)
159      >>> parser = Parser([tk0, tk1, tk2])
160      >>> g = GenVisitor()
161      >>> p = AsmModule.Program({}, [])
162      >>> exp = parser.parse()
163      >>> v = exp.accept(g, p)
164      >>> p.eval()
165      >>> p.get_val(v)
166      1
167
168      >>> tk0 = Token('4', TokenType.NUM)
169      >>> tk1 = Token('<', TokenType.LTH)
170      >>> tk2 = Token('4', TokenType.NUM)
171      >>> parser = Parser([tk0, tk1, tk2])
172      >>> g = GenVisitor()
173      >>> p = AsmModule.Program({}, [])
174      >>> exp = parser.parse()
175      >>> v = exp.accept(g, p)
176      >>> p.eval()
177      >>> p.get_val(v)
178      0
179
180      >>> tk0 = Token('not', TokenType.NOT)
181      >>> tk1 = Token('4', TokenType.NUM)
182      >>> tk2 = Token('<', TokenType.LTH)
183      >>> tk3 = Token('4', TokenType.NUM)
184      >>> parser = Parser([tk0, tk1, tk2, tk3])
185      >>> g = GenVisitor()
186      >>> p = AsmModule.Program({}, [])
187      >>> exp = parser.parse()
188      >>> v = exp.accept(g, p)
189      >>> p.eval()
190      >>> p.get_val(v)
191      1
192
193      >>> tk0 = Token('let', TokenType.LET)
194      >>> tk1 = Token('v', TokenType.VAR)
195      >>> tk2 = Token('<-', TokenType.ASN)
196      >>> tk3 = Token('42', TokenType.NUM)
197      >>> tk4 = Token('in', TokenType.INX)
198      >>> tk5 = Token('v', TokenType.VAR)
199      >>> tk6 = Token('end', TokenType.END)
200      >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
201      >>> g = GenVisitor()
202      >>> p = AsmModule.Program({}, [])
203      >>> exp = parser.parse()
204      >>> v = exp.accept(g, p)
205      >>> p.eval()
206      >>> p.get_val(v)
207      1

```

```
208      42
209      >>> tk0 = Token('let', TokenType.LET)
210      >>> tk1 = Token('v', TokenType.VAR)
211      >>> tk2 = Token('<-', TokenType.ASN)
212      >>> tk3 = Token('21', TokenType.NUM)
213      >>> tk4 = Token('in', TokenType.INX)
214      >>> tk5 = Token('v', TokenType.VAR)
215      >>> tk6 = Token('+', TokenType.ADD)
216      >>> tk7 = Token('v', TokenType.VAR)
217      >>> tk8 = Token('end', TokenType.END)
218      >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
219      >>> g = GenVisitor()
220      >>> p = AsmModule.Program({}, [])
221      >>> exp = parser.parse()
222      >>> v = exp.accept(g, p)
223      >>> p.eval()
224      >>> p.get_val(v)
225      42
226      """
227
228      # TODO: implement this method.
229      return None
```

Expression.py

```

1 """
2 This file implements the data structures that represent Expressions. You don't
3 need to modify it for this assignment.
4 """
5
6 from abc import ABC, abstractmethod
7 from Visitor import *
8
9
10 class Expression(ABC):
11     @abstractmethod
12     def accept(self, visitor, arg):
13         raise NotImplementedError
14
15
16 class Var(Expression):
17     """
18     This class represents expressions that are identifiers. The value of an
19     identifier is the value associated with it in the environment table.
20     """
21
22     def __init__(self, identifier):
23         self.identifier = identifier
24
25     def accept(self, visitor, arg):
26         """
27         Variables don't need to be implemented for this exercise.
28         """
29         return visitor.visit_var(self, arg)
30
31
32 class Bln(Expression):
33     """
34     This class represents expressions that are boolean values. There are only
35     two boolean values: true and false. The acceptuation of such an expression
36     is the boolean itself.
37     """
38
39     def __init__(self, bln):
40         self.bln = bln
41
42     def accept(self, visitor, arg):
43         """
44         booleans don't need to be implemented for this exercise.
45         """
46         return visitor.visit_bln(self, arg)
47
48
49 class Num(Expression):
50     """
51     This class represents expressions that are numbers. The acceptuation of such
52     an expression is the number itself.
53     """
54
55     def __init__(self, num):
56         self.num = num
57
58     def accept(self, visitor, arg):
59         """
60         Example:
61         >>> e = Num(3)
62         >>> ev = EvalVisitor()
63         >>> e.accept(ev, None)
64         3
65         """
66         return visitor.visit_num(self, arg)
67
68
69 class BinaryExpression(Expression):
70     """
71     This class represents binary expressions. A binary expression has two
72     sub-expressions: the left operand and the right operand.
73     """
74
75     def __init__(self, left, right):
76         self.left = left
77         self.right = right
78
79     @abstractmethod
80     def accept(self, visitor, arg):
81         raise NotImplementedError
82
83
84 class Eq1(BinaryExpression):
85     """
86     This class represents the equality between two expressions. The acceptuation
87     of such an expression is True if the subexpressions are the same, or false
88     otherwise.
89     """
90
91     def accept(self, visitor, arg):
92         """
93         Equality doesn't need to be implemented for this exercise.
94         """
95         return visitor.visit_eq1(self, arg)
96
97
98 class Add(BinaryExpression):
99     """
100    This class represents addition of two expressions. The acceptuation of such
101    an expression is the addition of the two subexpression's values.
102    """
103

```

```

104     def accept(self, visitor, arg):
105         """
106             Example:
107             >>> n1 = Num(3)
108             >>> n2 = Num(4)
109             >>> e = Add(n1, n2)
110             >>> ev = EvalVisitor()
111             >>> e.accept(ev, None)
112             7
113             """
114         return visitor.visit_add(self, arg)
115
116
117 class Sub(BinaryExpression):
118     """
119         This class represents subtraction of two expressions. The acceptuation of
120         such an expression is the subtraction of the two subexpression's values.
121         """
122
123     def accept(self, visitor, arg):
124         """
125             Example:
126             >>> n1 = Num(3)
127             >>> n2 = Num(4)
128             >>> e = Sub(n1, n2)
129             >>> ev = EvalVisitor()
130             >>> e.accept(ev, None)
131             -1
132             """
133         return visitor.visit_sub(self, arg)
134
135
136 class Mul(BinaryExpression):
137     """
138         This class represents multiplication of two expressions. The acceptuation of
139         such an expression is the product of the two subexpression's values.
140         """
141
142     def accept(self, visitor, arg):
143         """
144             Example:
145             >>> n1 = Num(3)
146             >>> n2 = Num(4)
147             >>> e = Mul(n1, n2)
148             >>> ev = EvalVisitor()
149             >>> e.accept(ev, None)
150             12
151             """
152         return visitor.visit_mul(self, arg)
153
154
155 class Div(BinaryExpression):
156     """
157         This class represents the integer division of two expressions. The
158         acceptuation of such an expression is the integer quotient of the two
159         subexpression's values.
160         """
161
162     def accept(self, visitor, arg):
163         """
164             Example:
165             >>> n1 = Num(28)
166             >>> n2 = Num(4)
167             >>> e = Div(n1, n2)
168             >>> ev = EvalVisitor()
169             >>> e.accept(ev, None)
170             7
171             >>> n1 = Num(22)
172             >>> n2 = Num(4)
173             >>> e = Div(n1, n2)
174             >>> ev = EvalVisitor()
175             >>> e.accept(ev, None)
176             5
177             """
178         return visitor.visit_div(self, arg)
179
180
181 class Leq(BinaryExpression):
182     """
183         This class represents comparison of two expressions using the
184         less-than-or-equal comparator. The acceptuation of such an expression is a
185         boolean value that is true if the left operand is less than or equal the
186         right operand. It is false otherwise.
187         """
188
189     def accept(self, visitor, arg):
190         """
191             Comparisons don't need to be implemented for this exercise.
192             """
193         return visitor.visit_leq(self, arg)
194
195
196 class Lth(BinaryExpression):
197     """
198         This class represents comparison of two expressions using the
199         less-than comparison operator. The acceptuation of such an expression is a
200         boolean value that is true if the left operand is less than the right
201         operand. It is false otherwise.
202         """
203
204     def accept(self, visitor, arg):
205         """
206             Comparisons don't need to be implemented for this exercise.
207             """

```

```

20/
208     return visitor.visit_lth(self, arg)
209
210
211 class UnaryExpression(Expression):
212     """
213     This class represents unary expressions. A unary expression has only one
214     sub-expression.
215     """
216
217     def __init__(self, exp):
218         self.exp = exp
219
220     @abstractmethod
221     def accept(self, visitor, arg):
222         raise NotImplementedError
223
224
225 class Neg(UnaryExpression):
226     """
227     This expression represents the additive inverse of a number. The additive
228     inverse of a number n is the number -n, so that the sum of both is zero.
229     """
230
231     def accept(self, visitor, arg):
232         """
233         Example:
234         >>> n = Num(3)
235         >>> e = Neg(n)
236         >>> ev = EvalVisitor()
237         >>> e.accept(ev, None)
238         -3
239         >>> n = Num(0)
240         >>> e = Neg(n)
241         >>> ev = EvalVisitor()
242         >>> e.accept(ev, None)
243         0
244         """
245
246         return visitor.visit_neg(self, arg)
247
248 class Not(UnaryExpression):
249     """
250     This expression represents the negation of a boolean. The negation of a
251     boolean expression is the logical complement of that expression.
252     """
253
254     def accept(self, visitor, arg):
255         """
256         No need to implement negation for this exercise, for we don't even have
257         booleans at this point.
258         """
259
260         return visitor.visit_not(self, arg)
261
262 class Let(Expression):
263     """
264     This class represents a let expression. The semantics of a let expression,
265     such as "let v <- e0 in e1" on an environment env is as follows:
266     1. Evaluate e0 in the environment env, yielding e0_val
267     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
268     """
269
270     def __init__(self, identifier, exp_def, exp_body):
271         self.identifier = identifier
272         self.exp_def = exp_def
273         self.exp_body = exp_body
274
275     def accept(self, visitor, arg):
276         """
277         We don't have bindings at this point. So, nothing to be done here, for
278         this exercise.
279         """
280
281         return visitor.visit_let(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
17 @abstractmethod
18 def visit_var(self, exp, arg):
19     pass
20
21 @abstractmethod
22 def visit_bln(self, exp, arg):
23     pass
24
25 @abstractmethod
26 def visit_num(self, exp, arg):
27     pass
28
29 @abstractmethod
30 def visit_eql(self, exp, arg):
31     pass
32
33 @abstractmethod
34 def visit_add(self, exp, arg):
35     pass
36
37 @abstractmethod
38 def visit_sub(self, exp, arg):
39     pass
40
41 @abstractmethod
42 def visit_mul(self, exp, arg):
43     pass
44
45 @abstractmethod
46 def visit_div(self, exp, arg):
47     pass
48
49 @abstractmethod
50 def visit_leq(self, exp, arg):
51     pass
52
53 @abstractmethod
54 def visit_lth(self, exp, arg):
55     pass
56
57 @abstractmethod
58 def visit_neg(self, exp, arg):
59     pass
60
61 @abstractmethod
62 def visit_not(self, exp, arg):
63     pass
64
65 @abstractmethod
66 def visit_let(self, exp, arg):
67     pass
68
69
70 class GenVisitor(Visitor):
71 """
72     The GenVisitor class compiles arithmetic expressions into a low-level
73     language.
74 """
75
76 def __init__(self):
77     self.next_var_counter = 0
78
79 def next_var_name(self):
80     self.next_var_counter += 1
81     return f"v{self.next_var_counter}"
82
83 def visit_var(self, exp, prog):
84 """
85     Usage:
86     >>> e = Var('x')
87     >>> p = AsmModule.Program({"x":1}, [])
88     >>> g = GenVisitor()
89     >>> v = e.accept(g, p)
90     >>> p.eval()
91     >>> p.get_val(v)
92     1
93 """
94     return exp.identifier
95
96 def visit_bln(self, exp, env):
97 """
98     Usage:
99     >>> e = Bln(True)
100    >>> p = AsmModule.Program({}, [])
101    >>> g = GenVisitor()
102    >>> v = e.accept(g, p)
103    >>> p.eval()

```

```

104         >>> p.get_val(v)
105         1
106
107         >>> e = Bln(False)
108         >>> p = AsmModule.Program({}, [])
109         >>> g = GenVisitor()
110         >>> v = e.accept(g, p)
111         >>> p.eval()
112         >>> p.get_val(v)
113         0
114
115         """
116         # TODO: Implement this method.
117         raise NotImplementedError
118
119     def visit_num(self, exp, prog):
120         """
121             Usage:
122             >>> e = Num(13)
123             >>> p = AsmModule.Program({}, [])
124             >>> g = GenVisitor()
125             >>> v = e.accept(g, p)
126             >>> p.eval()
127             >>> p.get_val(v)
128             13
129
130             # TODO: Implement this method.
131             raise NotImplementedError
132
133     def visit_eql(self, exp, prog):
134         """
135         >>> e = Eql(Num(13), Num(13))
136         >>> p = AsmModule.Program({}, [])
137         >>> g = GenVisitor()
138         >>> v = e.accept(g, p)
139         >>> p.eval()
140         >>> p.get_val(v)
141         1
142
143         >>> e = Eql(Num(13), Num(10))
144         >>> p = AsmModule.Program({}, [])
145         >>> g = GenVisitor()
146         >>> v = e.accept(g, p)
147         >>> p.eval()
148         >>> p.get_val(v)
149         0
150
151         >>> e = Eql(Num(-1), Num(1))
152         >>> p = AsmModule.Program({}, [])
153         >>> g = GenVisitor()
154         >>> v = e.accept(g, p)
155         >>> p.eval()
156         >>> p.get_val(v)
157         0
158
159             # TODO: Implement this method.
160             raise NotImplementedError
161
162     def visit_add(self, exp, prog):
163         """
164         >>> e = Add(Num(13), Num(-13))
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         >>> e = Add(Num(13), Num(10))
173         >>> p = AsmModule.Program({}, [])
174         >>> g = GenVisitor()
175         >>> v = e.accept(g, p)
176         >>> p.eval()
177         >>> p.get_val(v)
178         23
179
180             # TODO: Implement this method (see the example in the lab's page).
181             raise NotImplementedError
182
183     def visit_sub(self, exp, prog):
184         """
185         >>> e = Sub(Num(13), Num(-13))
186         >>> p = AsmModule.Program({}, [])
187         >>> g = GenVisitor()
188         >>> v = e.accept(g, p)
189         >>> p.eval()
190         >>> p.get_val(v)
191         26
192
193         >>> e = Sub(Num(13), Num(10))
194         >>> p = AsmModule.Program({}, [])
195         >>> g = GenVisitor()
196         >>> v = e.accept(g, p)
197         >>> p.eval()
198         >>> p.get_val(v)
199         3
200
201             # TODO: Implement this method.
202             raise NotImplementedError
203
204     def visit_mul(self, exp, prog):
205         """
206         >>> e = Mul(Num(13), Num(2))
207         >>> p = AsmModule.Program({}, [])
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1506
1507
1508
1508
1509
1510
1511
1512
1513
1514
1515
1515
1516
1517
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1525
1526
1527
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1545
1546
1547
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1556
1557
1558
1558
1559
1560
1561
1562
1563
1564
1564
1565
1566
1566
1567
1568
1568
1569
1570
1571
1572
1573
1574
1575
1575
1576
1577
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1604
1605
1606
1606
1607
1608
1608
1609
1610
1611
1612
1613
1614
1614
1615
1616
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1624
1625
1626
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1634
1635
1636
1636
1637
1638
1638
1639
1640
1641
1642
1643
1643
1644
1645
1645
1646
1647
1647
1648
1649
1649
1650
1651
1652
1653
1654
1654
1655
1656
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1664
1665
1666
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1674
1675
1676
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1684
1685
1686
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1694
1695
1696
1696
1697
1698
1698
1699
1700
1701
1702
1703
1704
1704
1705
1706
1706
1707
1708
1708
1709
1710
1711
1712
1713
1714
1714
1715
1716
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1724
1725
1726
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1734
1735
1736
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1744
1745
1746
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1754
1755
1756
1756
1757
1758
1758
1759
1760
1761
1762
1763
1764
1764
1765
1766
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1774
1775
1776
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1784
1785
1786
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1794
1795
1796
1796
1797
1798
1798
1799
1800
1801
1802
1803
1804
1804
1805
1806
1806
1807
1808
1808
1809
1810
1811
1812
1813
1814
1814
1815
1816
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1824
1825
1826
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1834
1835
1836
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1844
1845
1846
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1854
1855
1856
1856
1857
1858
1858
1859
1860
1861
1862
1863
1864
1864
1865
1866
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1874
1875
1876
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1884
1885
1886
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1894
1895
1896
1896
1897
1898
1898
1899
1900
1901
1902
1903
1904
1904
1905
1906
1906
1907
1908
1908
1909
1910
1911
1912
1913
1914
1914
1915
1916
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1924
1925
1926
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1934
1935
1936
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1944
1945
1946
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1954
1955
1956
1956
1957
1958
1958
1959
1960
1961
1962
1963
1964
1964
1965
1966
1966
1967
1968
1968
1969
1970
1971
1972
1973
1974
1974
1975
1976
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1984
1985
1986
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1994
1995
1996
1996
1997
1998
1998
1999
2000
2001
2002
2003
2004
2004
2005
2006
2006
2007
2008
2008
2009
2010
2011
2012
2013
2014
2014
2015
2016
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2024
2025
2026
2026
2027
2028
2028
2029
2030
2031
2032
2033
2034
2034
2035

```

```

20/
208    >>> g = Genvisitor()
209    >>> v = e.accept(g, p)
210    >>> p.eval()
211    >>> p.get_val(v)
212    26
213
214    >>> e = Mul(Num(13), Num(10))
215    >>> p = AsmModule.Program({}, [])
216    >>> g = GenVisitor()
217    >>> v = e.accept(g, p)
218    >>> p.eval()
219    >>> p.get_val(v)
220    130
221
222    # TODO: Implement this method.
223    raise NotImplementedError
224
225    def visit_div(self, exp, prog):
226        """
227            >>> e = Div(Num(13), Num(2))
228            >>> p = AsmModule.Program({}, [])
229            >>> g = GenVisitor()
230            >>> v = e.accept(g, p)
231            >>> p.eval()
232            >>> p.get_val(v)
233            6
234
235            >>> e = Div(Num(13), Num(10))
236            >>> p = AsmModule.Program({}, [])
237            >>> g = GenVisitor()
238            >>> v = e.accept(g, p)
239            >>> p.eval()
240            >>> p.get_val(v)
241            1
242
243            # TODO: Implement this method.
244            raise NotImplementedError
245
246            def visit_leq(self, exp, prog):
247                """
248                    >>> e = Leq(Num(3), Num(2))
249                    >>> p = AsmModule.Program({}, [])
250                    >>> g = GenVisitor()
251                    >>> v = e.accept(g, p)
252                    >>> p.eval()
253                    >>> p.get_val(v)
254                    0
255
256                    >>> e = Leq(Num(3), Num(3))
257                    >>> p = AsmModule.Program({}, [])
258                    >>> g = GenVisitor()
259                    >>> v = e.accept(g, p)
260                    >>> p.eval()
261                    >>> p.get_val(v)
262                    1
263
264                    >>> e = Leq(Num(2), Num(3))
265                    >>> p = AsmModule.Program({}, [])
266                    >>> g = GenVisitor()
267                    >>> v = e.accept(g, p)
268                    >>> p.eval()
269                    >>> p.get_val(v)
270                    1
271
272                    >>> e = Leq(Num(-3), Num(-2))
273                    >>> p = AsmModule.Program({}, [])
274                    >>> g = GenVisitor()
275                    >>> v = e.accept(g, p)
276                    >>> p.eval()
277                    >>> p.get_val(v)
278                    1
279
280                    >>> e = Leq(Num(-3), Num(-3))
281                    >>> p = AsmModule.Program({}, [])
282                    >>> g = GenVisitor()
283                    >>> v = e.accept(g, p)
284                    >>> p.eval()
285                    >>> p.get_val(v)
286                    1
287
288                    >>> e = Leq(Num(-2), Num(-3))
289                    >>> p = AsmModule.Program({}, [])
290                    >>> g = GenVisitor()
291                    >>> v = e.accept(g, p)
292                    >>> p.eval()
293                    >>> p.get_val(v)
294                    0
295
296                    # TODO: Implement this method.
297                    raise NotImplementedError
298
299                    def visit_lth(self, exp, prog):
300                        """
301                            >>> e = Lth(Num(3), Num(2))
302                            >>> p = AsmModule.Program({}, [])
303                            >>> g = GenVisitor()
304                            >>> v = e.accept(g, p)
305                            >>> p.eval()
306                            >>> p.get_val(v)
307                            0
308
309                            >>> e = Lth(Num(3), Num(3))
310                            >>> p = AsmModule.Program({}, [])
311                            ```` g = GenVisitor()

```

```

311     """  

312     >>> v = e.accept(g, p)  

313     >>> p.eval()  

314     >>> p.get_val(v)  

315     0  

316  

317     >>> e = Lth(Num(2), Num(3))  

318     >>> p = AsmModule.Program({}, [])  

319     >>> g = GenVisitor()  

320     >>> v = e.accept(g, p)  

321     >>> p.eval()  

322     >>> p.get_val(v)  

323     1  

324     """  

325     # TODO: Implement this method.  

326     raise NotImplementedError  

327  

328     def visit_neg(self, exp, prog):  

329         """  

330             >>> e = Neg(Num(3))  

331             >>> p = AsmModule.Program({}, [])  

332             >>> g = GenVisitor()  

333             >>> v = e.accept(g, p)  

334             >>> p.eval()  

335             >>> p.get_val(v)  

336             -3  

337  

338             >>> e = Neg(Num(0))  

339             >>> p = AsmModule.Program({}, [])  

340             >>> g = GenVisitor()  

341             >>> v = e.accept(g, p)  

342             >>> p.eval()  

343             >>> p.get_val(v)  

344             0  

345  

346             >>> e = Neg(Num(-3))  

347             >>> p = AsmModule.Program({}, [])  

348             >>> g = GenVisitor()  

349             >>> v = e.accept(g, p)  

350             >>> p.eval()  

351             >>> p.get_val(v)  

352             3  

353             """  

354             # TODO: Implement this method.  

355             raise NotImplementedError  

356  

357     def visit_not(self, exp, prog):  

358         """  

359             >>> e = Not(Bln(True))  

360             >>> p = AsmModule.Program({}, [])  

361             >>> g = GenVisitor()  

362             >>> v = e.accept(g, p)  

363             >>> p.eval()  

364             >>> p.get_val(v)  

365             0  

366  

367             >>> e = Not(Bln(False))  

368             >>> p = AsmModule.Program({}, [])  

369             >>> g = GenVisitor()  

370             >>> v = e.accept(g, p)  

371             >>> p.eval()  

372             >>> p.get_val(v)  

373             1  

374  

375             >>> e = Not(Num(0))  

376             >>> p = AsmModule.Program({}, [])  

377             >>> g = GenVisitor()  

378             >>> v = e.accept(g, p)  

379             >>> p.eval()  

380             >>> p.get_val(v)  

381             1  

382  

383             >>> e = Not(Num(-2))  

384             >>> p = AsmModule.Program({}, [])  

385             >>> g = GenVisitor()  

386             >>> v = e.accept(g, p)  

387             >>> p.eval()  

388             >>> p.get_val(v)  

389             0  

390  

391             >>> e = Not(Num(2))  

392             >>> p = AsmModule.Program({}, [])  

393             >>> g = GenVisitor()  

394             >>> v = e.accept(g, p)  

395             >>> p.eval()  

396             >>> p.get_val(v)  

397             0  

398             """  

399             # TODO: Implement this method.  

400             raise NotImplementedError  

401  

402     def visit_let(self, exp, prog):  

403         """  

404             Usage:  

405                 >>> e = Let('v', Not(Bln(False)), Var('v'))  

406                 >>> p = AsmModule.Program({}, [])  

407                 >>> g = GenVisitor()  

408                 >>> v = e.accept(g, p)  

409                 >>> p.eval()  

410                 >>> p.get_val(v)  

411                 1  

412  

413                 >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))  

414                 >>> n = AsmModule.Program({}.  

415

```

```
414         g = GenVisitor()
415         v = e.accept(g, p)
416         p.eval()
417         p.get_val(v)
418         5
419
420         e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
421         e1 = Let('y', e0, Mul(Var('y'), Num(10)))
422         p = AsmModule.Program({}, [])
423         g = GenVisitor()
424         v = e1.accept(g, p)
425         p.eval()
426         p.get_val(v)
427         50
428     """
429     # TODO: Implement this method.
430     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
17 This file uses doctests all over. To test it, just run python 3 as follows:
18 "python3 -m doctest Asm.py". The program uses syntax that is exclusive of
19 Python 3. It will not work with standard Python 2.
20 """
21
22 import sys
23 from collections import deque
24 from abc import ABC, abstractmethod
25
26
27 class Program:
28     """
29         The 'Program' is a list of instructions plus an environment that associates
30         names with values, plus a program counter, which marks the next instruction
31         that must be executed. The environment contains a special variable x0,
32         which always contains the value zero.
33     """
34
35     def __init__(self, env, insts):
36         self.__env = env
37         self.__insts = insts
38         self.pc = 0
39         self.__env["x0"] = 0
40
41     def get_inst(self):
42         if self.pc >= 0 and self.pc < len(self.__insts):
43             inst = self.__insts[self.pc]
44             self.pc += 1
45             return inst
46         else:
47             return None
48
49     def add_inst(self, inst):
50         self.__insts.append(inst)
51
52     def set_pc(self, pc):
53         self.pc = pc
54
55     def set_val(self, name, value):
56         self.__env[name] = value
57
58     def get_val(self, name):
59         """
60             The register x0 always contains the value zero.
61
62             >>> p = Program({}, [])
63             >>> p.get_val("x0")
64             0
65             """
66             if name in self.__env:
67                 return self.__env[name]
68             else:
69                 sys.exit("Def error")
70
71     def print_env(self):
72         for name, val in sorted(self.__env.items()):
73             print(f"{name}: {val}")
74
75     def print_insts(self):
76         for inst in self.__insts:
77             print(inst)
78
79     def eval(self):
80         """
81             This function evaluates a program until there is no more instructions to
82             evaluate.
83
84             Example:
85             >>> insts = [Add("x0", "b0", "b1"), Sub("x1", "x0", "b2")]
86             >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
87             >>> p.eval()
88             >>> p.print_env()
89             b0: 2
90             b1: 3
91             b2: 4
92             x0: 5
93             x1: 1
94             """
95             inst = self.get_inst()
96             while inst:
97                 inst.eval(self)
98                 inst = self.get_inst()
99
100            def max(a, b):
101                """
102                    This example computes the maximum between a and b.

```



```

201  class Add(BinOp):
202      """
203      add rd, rs1, rs2: rd = rs1 + rs2
204
205      Example:
206          >>> i = Add("a", "b0", "b1")
207          >>> str(i)
208          'a = add b0 b1'
209
210          >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
211          >>> p.eval()
212          >>> p.get_val("a")
213          5
214
215      """
216
217      def eval(self, prog):
218          rs1 = prog.get_val(self.rs1)
219          rs2 = prog.get_val(self.rs2)
220          prog.set_val(self.rd, rs1 + rs2)
221
222      def get_opcode(self):
223          return "add"
224
225
226  class Addi(BinOpImm):
227      """
228      addi rd, rs1, imm: rd = rs1 + imm
229
230      Example:
231          >>> i = Addi("a", "b0", 1)
232          >>> str(i)
233          'a = addi b0 1'
234
235          >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
236          >>> p.eval()
237          >>> p.get_val("a")
238          5
239
240      """
241
242      def eval(self, prog):
243          rs1 = prog.get_val(self.rs1)
244          prog.set_val(self.rd, rs1 + self.imm)
245
246      def get_opcode(self):
247          return "addi"
248
249
250  class Mul(BinOp):
251      """
252      mul rd, rs1, rs2: rd = rs1 * rs2
253
254      Example:
255          >>> i = Mul("a", "b0", "b1")
256          >>> str(i)
257          'a = mul b0 b1'
258
259          >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
260          >>> p.eval()
261          >>> p.get_val("a")
262          6
263
264      """
265
266      def eval(self, prog):
267          rs1 = prog.get_val(self.rs1)
268          rs2 = prog.get_val(self.rs2)
269          prog.set_val(self.rd, rs1 * rs2)
270
271      def get_opcode(self):
272          return "mul"
273
274
275  class Sub(BinOp):
276      """
277      sub rd, rs1, rs2: rd = rs1 - rs2
278
279      Example:
280          >>> i = Sub("a", "b0", "b1")
281          >>> str(i)
282          'a = sub b0 b1'
283
284          >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
285          >>> p.eval()
286          >>> p.get_val("a")
287          -1
288
289      """
290
291      def eval(self, prog):
292          rs1 = prog.get_val(self.rs1)
293          rs2 = prog.get_val(self.rs2)
294          prog.set_val(self.rd, rs1 - rs2)
295
296      def get_opcode(self):
297          return "sub"
298
299
300
301  class Xor(BinOp):
302      """
303      xor rd, rs1, rs2: rd = rs1 ^ rs2
304
305      Example:
306          >>> i = Xor("a", "b0", "b1")
307          >>> str(i)
308          'a = xor b0 b1'
309
310      """
311

```

```

311     >>> p = Program(env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
312     >>> p.eval()
313     >>> p.get_val("a")
314     1
315
316
317     def eval(self, prog):
318         rs1 = prog.get_val(self.rs1)
319         rs2 = prog.get_val(self.rs2)
320         prog.set_val(self.rd, rs1 ^ rs2)
321
322     def get_opcode(self):
323         return "xor"
324
325
326 class Xori(BinOpImm):
327     """
328         xori rd, rs1, imm: rd = rs1 ^ imm
329
330     Example:
331         >>> i = Xori("a", "b0", 10)
332         >>> str(i)
333         'a = xori b0 10'
334
335         >>> p = Program(env={"b0":2}, insts=[Xori("a", "b0", 3)])
336         >>> p.eval()
337         >>> p.get_val("a")
338         1
339
340
341     def eval(self, prog):
342         rs1 = prog.get_val(self.rs1)
343         prog.set_val(self.rd, rs1 ^ self.imm)
344
345     def get_opcode(self):
346         return "xori"
347
348
349 class Div(BinOp):
350     """
351         div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
352     Notice that RISC-V does not have an instruction exactly like this one.
353     The div operator works on floating-point numbers; not on integers.
354
355     Example:
356         >>> i = Div("a", "b0", "b1")
357         >>> str(i)
358         'a = div b0 b1'
359
360         >>> p = Program(env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
361         >>> p.eval()
362         >>> p.get_val("a")
363         2
364
365
366     def eval(self, prog):
367         rs1 = prog.get_val(self.rs1)
368         rs2 = prog.get_val(self.rs2)
369         prog.set_val(self.rd, rs1 // rs2)
370
371     def get_opcode(self):
372         return "div"
373
374
375 class Slt(BinOp):
376     """
377         slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
378
379     Example:
380         >>> i = Slt("a", "b0", "b1")
381         >>> str(i)
382         'a = slt b0 b1'
383
384         >>> p = Program(env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
385         >>> p.eval()
386         >>> p.get_val("a")
387         1
388
389         >>> p = Program(env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
390         >>> p.eval()
391         >>> p.get_val("a")
392         0
393
394         >>> p = Program(env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
395         >>> p.eval()
396         >>> p.get_val("a")
397         0
398
399
400     def eval(self, prog):
401         rs1 = prog.get_val(self.rs1)
402         rs2 = prog.get_val(self.rs2)
403         prog.set_val(self.rd, 1 if rs1 < rs2 else 0)
404
405     def get_opcode(self):
406         return "slt"
407
408
409 class Slti(BinOpImm):
410     """
411         slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
412         (signed comparison with immediate)
413

```

```
414 Example:  
415     >>> i = Slti("a", "b0", 0)  
416     >>> str(i)  
417     'a = slti b0 0'  
418  
419     >>> p = Program(env={"b0":2}, insts=[Slti("a", "b0", 3)])  
420     >>> p.eval()  
421     >>> p.get_val("a")  
422     1  
423  
424     >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 3)])  
425     >>> p.eval()  
426     >>> p.get_val("a")  
427     0  
428  
429     >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 2)])  
430     >>> p.eval()  
431     >>> p.get_val("a")  
432     0  
433     """  
434  
435     def eval(self, prog):  
436         rs1 = prog.get_val(self.rs1)  
437         prog.set_val(self.rd, 1 if rs1 < self.imm else 0)  
438  
439     def get_opcode(self):  
440         return "slti"
```

VPL