

2025_2 - COMPILADORES - METATURMA

PAINEL > **MINHAS TURMAS** > **2025_2 - COMPILADORES - METATURMA** > **LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL**
> **AV15 - ALOCAÇÃO DE REGISTROS PARA CÓDIGO LINEAR**

 Descrição

 Enviar

 Editar

 Visualizar envios

AV15 - Alocação de registros para código linear

 **Data de entrega:** segunda, 1 Dez 2025, 23:59

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

Tipo de trabalho:  Trabalho individual

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

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

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

- `sw reg, rs1(offset)`: salva no endereço de memória `rs1 + offset` o valor que estava no registrador `reg`.
- `ld reg, rs1(offset)`: carrega em `reg` o valor que estava na posição de memória `rs1 + offset`.

Essas instruções lhe dão acesso à *memória* do programa em execução. Pense na memória como um arranjo em que são armazenados inteiros de 32 bits. Esses valores podem ser lidos via instruções `sw`, e podem ser escritos via instruções `ld`. Programas agora também possuem um registrador especial, `sp`, que é inicializado com o tamanho da memória disponível. Exemplos de acesso à memória são mostrados abaixo:

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

O que deve ser feito

Para resolver este exercício, você deverá reusar as implementações de `RenameVisitor` e de `GenVisitor` (ambas em `Visitor.py`) vistas no VPL 12, além de implementar uma nova classe: `RegAllocator`, que está disponível no arquivo `Optimizer.py`. Você deverá implementar três métodos de `RegAllocator`, a saber:

`__init__(self, prog)`: assumindo-se que você queira inicializar dados relacionados à alocação de registradores.

`optimize()`: o método que substitui as instruções em `prog` por novas instruções, que usam somente endereços de memória ou registradores.

`get_val(var)`: o método que retorna o valor de `var`. Como `prog` já não usa instruções que fazem referência ao nome `var`, você precisará salvar no otimizar como `var` está mapeada. Este nome pode estar associado a algum registrador, ou a algum endereço de memória. As três figuras abaixo ilustram como o valor associado a uma variável pode ser recuperado após a alocação de registradores.

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

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

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

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

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

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

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

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

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

Submetendo e Testando

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

```
$> python3 driver.py
1 + 2 # CTRL+D
3
```

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

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

No exemplo acima, 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 from Optimizer import *
8
9
10 def rename_variables(exp):
11     """
12     Esta funcao invoca o renomeador de variaveis. Ela deve ser usada antes do
13     inicio da fase de geracao de codigo.
14     """
15     ren = RenameVisitor()
16     exp.accept(ren, {})
17     return exp
18
19
20 def perform_register_allocation(prog, dump=False):
21     """
22     Esta funcao invoca o alocador de registradores sobre o programa. Caso queira
23     depurar sua alocao, fique a vontade para usar dump == True.
24     """
25     o = RegAllocator(prog)
26     if dump:
27         print("Before RA: -----")
28         prog.print_insts()
29     o.optimize()
30     if dump:
31         print("After RA: -----")
32         prog.print_insts()
33     return o
34
35
36 if __name__ == "__main__":
37     """
38     Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
39     VPL. O arquivo contem o codigo que testa a implementacao do parser.
40     """
41     text = sys.stdin.read()
42     lexer = Lexer(text)
43     parser = Parser(lexer.tokens())
44     exp = rename_variables(parser.parse())
45     prog = AsmModule.Program(1000, {}, [])
46     gen = GenVisitor()
47     var_answer = exp.accept(gen, prog)
48     opt = perform_register_allocation(prog, False)
49     prog.reset_env()
50     prog.eval()
51     print(f"Answer: {opt.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         The constructor of the lexer. It receives the string that shall be
55         scanned.
56         TODO: You will need to implement this method.
57         """
58         pass
59
60     def tokens(self):
61         """
62         This method is a token generator: it converts the string encapsulated
63         into this object into a sequence of Tokens. Examples:
64
65         >>> l = Lexer("1 + 3")
66         >>> [tk.kind for tk in l.tokens()]
67         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
68
69         >>> l = Lexer('1 * 2 -- 3\\n')
70         >>> [tk.kind for tk in l.tokens()]
71         [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
72
73         >>> l = Lexer("1 + var")
74         >>> [tk.kind for tk in l.tokens()]
75         [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
76
77         >>> l = Lexer("let v <- 2 in v end")
78         >>> [tk.kind.name for tk in l.tokens()]
79         ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
80         """
81         token = self.getToken()
82         while token.kind != TokenType.EOF:
83             if (
84                 token.kind != TokenType.WSP
85                 and token.kind != TokenType.COM
86                 and token.kind != TokenType.NLN
87             ):
88                 yield token
89             token = self.getToken()
90
91     def getToken(self):
92         """
93         Return the next token.
94         TODO: Implement this method (you can reuse Lab 5: Visitors)!
95         """
96         token = None
97         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         >>> exp = parser.parse()
31         >>> exp.eval(None)
32         123
33
34         >>> parser = Parser([Token('True', TokenType.TRU)])
35         >>> exp = parser.parse()
36         >>> exp.eval(None)
37         True
38
39         >>> parser = Parser([Token('False', TokenType.FLS)])
40         >>> exp = parser.parse()
41         >>> exp.eval(None)
42         False
43
44         >>> tk0 = Token('~', TokenType.NEG)
45         >>> tk1 = Token('123', TokenType.NUM)
46         >>> parser = Parser([tk0, tk1])
47         >>> exp = parser.parse()
48         >>> exp.eval(None)
49         -123
50
51         >>> tk0 = Token('3', TokenType.NUM)
52         >>> tk1 = Token('*', TokenType.MUL)
53         >>> tk2 = Token('4', TokenType.NUM)
54         >>> parser = Parser([tk0, tk1, tk2])
55         >>> exp = parser.parse()
56         >>> exp.eval(None)
57         12
58
59         >>> tk0 = Token('3', TokenType.NUM)
60         >>> tk1 = Token('*', TokenType.MUL)
61         >>> tk2 = Token('~', TokenType.NEG)
62         >>> tk3 = Token('4', TokenType.NUM)
63         >>> parser = Parser([tk0, tk1, tk2, tk3])
64         >>> exp = parser.parse()
65         >>> exp.eval(None)
66         -12
67
68         >>> tk0 = Token('30', TokenType.NUM)
69         >>> tk1 = Token('/', TokenType.DIV)
70         >>> tk2 = Token('4', TokenType.NUM)
71         >>> parser = Parser([tk0, tk1, tk2])
72         >>> exp = parser.parse()
73         >>> exp.eval(None)
74         7
75
76         >>> tk0 = Token('3', TokenType.NUM)
77         >>> tk1 = Token('+', TokenType.ADD)
78         >>> tk2 = Token('4', TokenType.NUM)
79         >>> parser = Parser([tk0, tk1, tk2])
80         >>> exp = parser.parse()
81         >>> exp.eval(None)
82         7
83
84         >>> tk0 = Token('30', TokenType.NUM)
85         >>> tk1 = Token('-', TokenType.SUB)
86         >>> tk2 = Token('4', TokenType.NUM)
87         >>> parser = Parser([tk0, tk1, tk2])
88         >>> exp = parser.parse()
89         >>> exp.eval(None)
90         26
91
92         >>> tk0 = Token('2', TokenType.NUM)
93         >>> tk1 = Token('*', TokenType.MUL)
94         >>> tk2 = Token('(', TokenType.LPR)
95         >>> tk3 = Token('3', TokenType.NUM)
96         >>> tk4 = Token('+', TokenType.ADD)
97         >>> tk5 = Token('4', TokenType.NUM)
98         >>> tk6 = Token(')', TokenType.RPR)
99         >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
100        >>> exp = parser.parse()
101        >>> exp.eval(None)
102        14
103

```

```
104     >>> tk0 = Token('4', TokenType.NUM)
105     >>> tk1 = Token('==', TokenType.EQL)
106     >>> tk2 = Token('4', TokenType.NUM)
107     >>> parser = Parser([tk0, tk1, tk2])
108     >>> exp = parser.parse()
109     >>> exp.eval(None)
110     True
111
112     >>> tk0 = Token('4', TokenType.NUM)
113     >>> tk1 = Token('<=', TokenType.LEQ)
114     >>> tk2 = Token('4', TokenType.NUM)
115     >>> parser = Parser([tk0, tk1, tk2])
116     >>> exp = parser.parse()
117     >>> exp.eval(None)
118     True
119
120     >>> tk0 = Token('4', TokenType.NUM)
121     >>> tk1 = Token('<', TokenType.LTH)
122     >>> tk2 = Token('4', TokenType.NUM)
123     >>> parser = Parser([tk0, tk1, tk2])
124     >>> exp = parser.parse()
125     >>> exp.eval(None)
126     False
127
128     >>> tk0 = Token('not', TokenType.NOT)
129     >>> tk1 = Token('4', TokenType.NUM)
130     >>> tk2 = Token('<', TokenType.LTH)
131     >>> tk3 = Token('4', TokenType.NUM)
132     >>> parser = Parser([tk0, tk1, tk2, tk3])
133     >>> exp = parser.parse()
134     >>> exp.eval(None)
135     True
136
137     >>> tk0 = Token('let', TokenType.LET)
138     >>> tk1 = Token('v', TokenType.VAR)
139     >>> tk2 = Token('<-', TokenType.ASN)
140     >>> tk3 = Token('42', TokenType.NUM)
141     >>> tk4 = Token('in', TokenType.INX)
142     >>> tk5 = Token('v', TokenType.VAR)
143     >>> tk6 = Token('end', TokenType.END)
144     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
145     >>> exp = parser.parse()
146     >>> exp.eval({})
147     42
148
149     >>> tk0 = Token('let', TokenType.LET)
150     >>> tk1 = Token('v', TokenType.VAR)
151     >>> tk2 = Token('<-', TokenType.ASN)
152     >>> tk3 = Token('21', TokenType.NUM)
153     >>> tk4 = Token('in', TokenType.INX)
154     >>> tk5 = Token('v', TokenType.VAR)
155     >>> tk6 = Token('+', TokenType.ADD)
156     >>> tk7 = Token('v', TokenType.VAR)
157     >>> tk8 = Token('end', TokenType.END)
158     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
159     >>> exp = parser.parse()
160     >>> exp.eval({})
161     42
162     """
163     # TODO: implement this method.
164     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 acceptance 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 acceptance 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 Eql(BinaryExpression):
85     """
86     This class represents the equality between two expressions. The acceptance
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_eql(self, arg)
96
97
98 class Add(BinaryExpression):
99     """
100     This class represents addition of two expressions. The acceptance 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 acceptance 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 acceptance 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         acceptance 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 acceptance 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 acceptance 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             """
208             return visitor.visit_lth(self, arg)
```



```
207         """
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         return visitor.visit_neg(self, arg)
246
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         return visitor.visit_not(self, arg)
260
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         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 on 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_bin(self, exp, arg):
23        pass
24
25    @abstractmethod
26    def visit_num(self, exp, arg):
27        pass
28
29    @abstractmethod
30    def visit_eq1(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 RenameVisitor(ABC):
71     """
72     This visitor traverses the AST of a program, renaming variables to ensure
73     that they all have different names.
74
75     Usage:
76     >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
77     >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
78     >>> e0.identifier == e1.identifier
79     True
80
81     >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
82     >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
83     >>> r = RenameVisitor()
84     >>> e1.accept(r, {})
85     >>> e0.identifier == e1.identifier
86     False
87
88     >>> x0 = Var('x')
89     >>> x1 = Var('x')
90     >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
91     >>> e1 = Let('x', e0, Mul(x1, Num(10)))
92     >>> x0.identifier == x1.identifier
93     True
94
95     >>> x0 = Var('x')
96     >>> x1 = Var('x')
97     >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
98     >>> e1 = Let('x', e0, Mul(x1, Num(10)))
99     >>> r = RenameVisitor()
100    >>> e1.accept(r, {})
101    >>> x0.identifier == x1.identifier
102    False
103    """

```

```

104
105 def visit_var(self, exp, arg):
106     # TODO: Implement this method.
107     raise NotImplementedError
108
109 def visit_bln(self, exp, arg):
110     # TODO: Implement this method.
111     raise NotImplementedError
112
113 def visit_num(self, exp, arg):
114     # TODO: Implement this method.
115     raise NotImplementedError
116
117 def visit_eq1(self, exp, arg):
118     # TODO: Implement this method.
119     raise NotImplementedError
120
121 def visit_add(self, exp, arg):
122     # TODO: Implement this method.
123     raise NotImplementedError
124
125 def visit_sub(self, exp, arg):
126     # TODO: Implement this method.
127     raise NotImplementedError
128
129 def visit_mul(self, exp, arg):
130     # TODO: Implement this method.
131     raise NotImplementedError
132
133 def visit_div(self, exp, arg):
134     # TODO: Implement this method.
135     raise NotImplementedError
136
137 def visit_leq(self, exp, arg):
138     # TODO: Implement this method.
139     raise NotImplementedError
140
141 def visit_lth(self, exp, arg):
142     # TODO: Implement this method.
143     raise NotImplementedError
144
145 def visit_neg(self, exp, arg):
146     # TODO: Implement this method.
147     raise NotImplementedError
148
149 def visit_not(self, exp, arg):
150     # TODO: Implement this method.
151     raise NotImplementedError
152
153 def visit_let(self, exp, arg):
154     # TODO: Implement this method.
155     raise NotImplementedError
156
157
158 class GenVisitor(Visitor):
159     """
160     The GenVisitor class compiles arithmetic expressions into a low-level
161     language.
162     """
163
164     def __init__(self):
165         self.next_var_counter = 0
166
167     def next_var_name(self):
168         self.next_var_counter += 1
169         return f"v{self.next_var_counter}"
170
171     def visit_var(self, exp, prog):
172         """
173         Usage:
174         >>> e = Var('x')
175         >>> p = AsmModule.Program({"x":1}, [])
176         >>> g = GenVisitor()
177         >>> v = e.accept(g, p)
178         >>> p.eval()
179         >>> p.get_val(v)
180         1
181         """
182         return exp.identifier
183
184     def visit_bln(self, exp, env):
185         """
186         Usage:
187         >>> e = Bln(True)
188         >>> p = AsmModule.Program({}, [])
189         >>> g = GenVisitor()
190         >>> v = e.accept(g, p)
191         >>> p.eval()
192         >>> p.get_val(v)
193         1
194
195         >>> e = Bln(False)
196         >>> p = AsmModule.Program({}, [])
197         >>> g = GenVisitor()
198         >>> v = e.accept(g, p)
199         >>> p.eval()
200         >>> p.get_val(v)
201         0
202         """
203         # TODO: Implement this method.
204         raise NotImplementedError
205
206     def visit_num(self, exp, prog):
207         """

```

```

207         """
208         Usage:
209             >>> e = Num(13)
210             >>> p = AsmModule.Program({}, [])
211             >>> g = GenVisitor()
212             >>> v = e.accept(g, p)
213             >>> p.eval()
214             >>> p.get_val(v)
215             13
216         """
217         # TODO: Implement this method.
218         raise NotImplementedError
219
220     def visit_eq1(self, exp, prog):
221         """
222             >>> e = Eq1(Num(13), Num(13))
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 = Eq1(Num(13), Num(10))
231             >>> p = AsmModule.Program({}, [])
232             >>> g = GenVisitor()
233             >>> v = e.accept(g, p)
234             >>> p.eval()
235             >>> p.get_val(v)
236             0
237
238             >>> e = Eq1(Num(-1), Num(1))
239             >>> p = AsmModule.Program({}, [])
240             >>> g = GenVisitor()
241             >>> v = e.accept(g, p)
242             >>> p.eval()
243             >>> p.get_val(v)
244             0
245         """
246         # TODO: Implement this method.
247         raise NotImplementedError
248
249     def visit_add(self, exp, prog):
250         """
251             >>> e = Add(Num(13), Num(-13))
252             >>> p = AsmModule.Program({}, [])
253             >>> g = GenVisitor()
254             >>> v = e.accept(g, p)
255             >>> p.eval()
256             >>> p.get_val(v)
257             0
258
259             >>> e = Add(Num(13), Num(10))
260             >>> p = AsmModule.Program({}, [])
261             >>> g = GenVisitor()
262             >>> v = e.accept(g, p)
263             >>> p.eval()
264             >>> p.get_val(v)
265             23
266         """
267         # TODO: Implement this method (see the example in the lab's page).
268         raise NotImplementedError
269
270     def visit_sub(self, exp, prog):
271         """
272             >>> e = Sub(Num(13), Num(-13))
273             >>> p = AsmModule.Program({}, [])
274             >>> g = GenVisitor()
275             >>> v = e.accept(g, p)
276             >>> p.eval()
277             >>> p.get_val(v)
278             26
279
280             >>> e = Sub(Num(13), Num(10))
281             >>> p = AsmModule.Program({}, [])
282             >>> g = GenVisitor()
283             >>> v = e.accept(g, p)
284             >>> p.eval()
285             >>> p.get_val(v)
286             3
287         """
288         # TODO: Implement this method.
289         raise NotImplementedError
290
291     def visit_mul(self, exp, prog):
292         """
293             >>> e = Mul(Num(13), Num(2))
294             >>> p = AsmModule.Program({}, [])
295             >>> g = GenVisitor()
296             >>> v = e.accept(g, p)
297             >>> p.eval()
298             >>> p.get_val(v)
299             26
300
301             >>> e = Mul(Num(13), Num(10))
302             >>> p = AsmModule.Program({}, [])
303             >>> g = GenVisitor()
304             >>> v = e.accept(g, p)
305             >>> p.eval()
306             >>> p.get_val(v)
307             130
308         """
309         # TODO: Implement this method.
310         raise NotImplementedError

```

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

```

414
415     def visit_neg(self, exp, prog):
416         """
417         >>> e = Neg(Num(3))
418         >>> p = AsmModule.Program({}, [])
419         >>> g = GenVisitor()
420         >>> v = e.accept(g, p)
421         >>> p.eval()
422         >>> p.get_val(v)
423         -3
424
425         >>> e = Neg(Num(0))
426         >>> p = AsmModule.Program({}, [])
427         >>> g = GenVisitor()
428         >>> v = e.accept(g, p)
429         >>> p.eval()
430         >>> p.get_val(v)
431         0
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         # TODO: Implement this method.
442         raise NotImplementedError
443
444     def visit_not(self, exp, prog):
445         """
446         >>> e = Not(Bln(True))
447         >>> p = AsmModule.Program({}, [])
448         >>> g = GenVisitor()
449         >>> v = e.accept(g, p)
450         >>> p.eval()
451         >>> p.get_val(v)
452         0
453
454         >>> e = Not(Bln(False))
455         >>> p = AsmModule.Program({}, [])
456         >>> g = GenVisitor()
457         >>> v = e.accept(g, p)
458         >>> p.eval()
459         >>> p.get_val(v)
460         1
461
462         >>> e = Not(Num(0))
463         >>> p = AsmModule.Program({}, [])
464         >>> g = GenVisitor()
465         >>> v = e.accept(g, p)
466         >>> p.eval()
467         >>> p.get_val(v)
468         1
469
470         >>> e = Not(Num(-2))
471         >>> p = AsmModule.Program({}, [])
472         >>> g = GenVisitor()
473         >>> v = e.accept(g, p)
474         >>> p.eval()
475         >>> p.get_val(v)
476         0
477
478         >>> e = Not(Num(2))
479         >>> p = AsmModule.Program({}, [])
480         >>> g = GenVisitor()
481         >>> v = e.accept(g, p)
482         >>> p.eval()
483         >>> p.get_val(v)
484         0
485         """
486         # TODO: Implement this method.
487         raise NotImplementedError
488
489     def visit_let(self, exp, prog):
490         """
491         Usage:
492         >>> e = Let('v', Not(Bln(False)), Var('v'))
493         >>> p = AsmModule.Program({}, [])
494         >>> g = GenVisitor()
495         >>> v = e.accept(g, p)
496         >>> p.eval()
497         >>> p.get_val(v)
498         1
499
500         >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
501         >>> p = AsmModule.Program({}, [])
502         >>> g = GenVisitor()
503         >>> v = e.accept(g, p)
504         >>> p.eval()
505         >>> p.get_val(v)
506         5
507
508         >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
509         >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
510         >>> p = AsmModule.Program({}, [])
511         >>> g = GenVisitor()
512         >>> v = e1.accept(g, p)
513         >>> p.eval()
514         >>> p.get_val(v)
515         50
516         """

```

```
517 # TODO: Implement this method.  
518 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     * jalr rd, rs1, offset: rd = pc + 1 and pc = rs1 + offset
19     * sw reg, offset(rs1): mem[offset+rs1] = reg
20     * lw reg, offset(rs1): reg = mem[offset+rs1]
21
22 This file uses doctests all over. To test it, just run python 3 as follows:
23 "python3 -m doctest Asm.py". The program uses syntax that is exclusive of
24 Python 3. It will not work with standard Python 2.
25 """
26
27 import sys
28 from collections import deque
29 from abc import ABC, abstractmethod
30
31
32 class Program:
33     """
34     The 'Program' is a list of instructions plus an environment that associates
35     names with values, plus a program counter, which marks the next instruction
36     that must be executed. The environment contains a special variable x0,
37     which always contains the value zero.
38     """
39
40     def __init__(self, memory_size, env, insts):
41         self.__mem = memory_size * [0]
42         self.__env = env
43         self.__insts = insts
44         self.pc = 0
45         self.registers = {"x0", "a0", "a1", "a2", "a3", "ra", "sp"}
46         for reg in self.registers:
47             self.__env[reg] = 0
48         self.__env["sp"] = memory_size
49
50     def reset_env(self):
51         for reg in self.registers:
52             self.__env[reg] = 0
53         self.__env["sp"] = len(self.__mem)
54
55     def get_inst(self):
56         if self.pc >= 0 and self.pc < len(self.__insts):
57             inst = self.__insts[self.pc]
58             self.pc += 1
59             return inst
60         else:
61             return None
62
63     def get_number_of_instructions(self):
64         return len(self.__insts)
65
66     def add_inst(self, inst):
67         self.__insts.append(inst)
68
69     def get_pc(self):
70         return self.pc
71
72     def set_pc(self, pc):
73         self.pc = pc
74
75     def set_val(self, name, value):
76         if name in self.registers and name != "x0":
77             self.__env[name] = value
78         else:
79             sys.exit(f"Undefined register: {name}")
80
81     def set_mem(self, addr, value):
82         self.__mem[addr] = value
83
84     def get_mem(self, addr):
85         if addr < 0 or addr >= len(self.__mem):
86             sys.exit(f"Invalid memory address: {addr}")
87         return self.__mem[addr]
88
89     def get_val(self, name):
90         """
91         The register x0 always contains the value zero:
92
93         >>> p = Program(0, {}, [])
94         >>> p.get_val("x0")
95         0
96         """
97         if name in self.__env:
98             return self.__env[name]
99         else:
100             sys.exit(f"Undefined register: {name}")
101
102     def print_env(self):
103         for name, val in sorted(self.__env.items()):

```



```

104         print(f"{name}: {val}")
105
106     def print_insts(self):
107         counter = 0
108         for inst in self.__insts:
109             print("%03d: %s" % (counter, str(inst)))
110             counter += 1
111         print("%03d: %s" % (counter, "END"))
112
113     def get_insts(self):
114         return self.__insts.copy()
115
116     def set_insts(self, insts):
117         self.__insts = insts
118
119     def eval(self):
120         """
121         This function evaluates a program until there is no more instructions
122         to evaluate.
123
124         Example:
125         >>> insts = [Add("t0", "b0", "b1"), Sub("x1", "t0", "b2")]
126         >>> p = Program(0, {"b0":2, "b1":3, "b2": 4}, insts)
127         >>> p.eval()
128         >>> p.print_env()
129         b0: 2
130         b1: 3
131         b2: 4
132         sp: 0
133         t0: 5
134         x0: 0
135         x1: 1
136
137         Notice that it is not possible to change 'x0':
138         >>> insts = [Add("x0", "b0", "b1")]
139         >>> p = Program(0, {"b0":2, "b1":3}, insts)
140         >>> p.eval()
141         >>> p.print_env()
142         b0: 2
143         b1: 3
144         sp: 0
145         x0: 0
146
147         """
148         inst = self.get_inst()
149         while inst:
150             inst.eval(self)
151             inst = self.get_inst()
152
153     def max(a, b):
154         """
155         This example computes the maximum between a and b.
156
157         Example:
158         >>> max(2, 3)
159         3
160
161         >>> max(3, 2)
162         3
163
164         >>> max(-3, -2)
165         -2
166
167         >>> max(-2, -3)
168         -2
169
170         """
171         p = Program(0, {}, [])
172         p.set_val("rs1", a)
173         p.set_val("rs2", b)
174         p.add_inst(Slt("t0", "rs2", "rs1"))
175         p.add_inst(Slt("t1", "rs1", "rs2"))
176         p.add_inst(Mul("t0", "t0", "rs1"))
177         p.add_inst(Mul("t1", "t1", "rs2"))
178         p.add_inst(Add("rd", "t0", "t1"))
179         p.eval()
180         return p.get_val("rd")
181
182     def distance_with_acceleration(V, A, T):
183         """
184         This example computes the position of an object, given its velocity (V),
185         its acceleration (A) and the time (T), assuming that it starts at position
186         zero, using the formula  $D = V*T + (A*T^2)/2$ .
187
188         Example:
189         >>> distance_with_acceleration(3, 4, 5)
190         65
191
192         """
193         p = Program(0, {}, [])
194         p.set_val("rs1", V)
195         p.set_val("rs2", A)
196         p.set_val("rs3", T)
197         p.add_inst(Addi("two", "x0", 2))
198         p.add_inst(Mul("t0", "rs1", "rs3"))
199         p.add_inst(Mul("t1", "rs3", "rs3"))
200         p.add_inst(Mul("t2", "rs2", "t1"))
201         p.add_inst(Div("t2", "t2", "two"))
202         p.add_inst(Add("rd", "t0", "t2"))
203         p.eval()
204         return p.get_val("rd")
205
206     class Inst(ABC):
207         """

```

```

207     """
208     The representation of instructions. Every instruction refers to a program
209     during its evaluation.
210     """
211
212     def __init__(self):
213         pass
214
215     @abstractmethod
216     def get_opcode(self):
217         raise NotImplementedError
218
219     @abstractmethod
220     def eval(self, prog):
221         raise NotImplementedError
222
223
224     class BranchOp(Inst):
225         """
226         The general class of branching instructions. These instructions can change
227         the control flow of a program. Normally, the next instruction is given by
228         pc + 1. A branch might change pc to point out to a different label..
229         """
230
231         def set_target(self, lab):
232             assert isinstance(lab, int)
233             self.lab = lab
234
235
236         class Beq(BranchOp):
237             """
238             beq rs1, rs2, lab:
239             Jumps to label lab if the value in rs1 is equal to the value in rs2.
240             """
241
242             def __init__(self, rs1, rs2, lab=None):
243                 assert isinstance(rs1, str) and isinstance(rs2, str)
244                 self.rs1 = rs1
245                 self.rs2 = rs2
246                 if lab != None:
247                     assert isinstance(lab, int)
248                     self.lab = lab
249
250             def get_opcode(self):
251                 return "beq"
252
253             def __str__(self):
254                 op = self.get_opcode()
255                 return f"{op} {self.rs1} {self.rs2} {self.lab}"
256
257             def eval(self, prog):
258                 if prog.get_val(self.rs1) == prog.get_val(self.rs2):
259                     prog.set_pc(self.lab)
260
261
262         class Jal(BranchOp):
263             """
264             jal rd lab:
265             Stores the return address (PC+1) on register rd, then jumps to label lab.
266             If rd is x0, then it does not write on the register. In this case, notice
267             that 'jal x0 lab' is equivalent to an unconditional jump to 'lab'.
268
269             Example:
270             >>> i = Jal("a", 20)
271             >>> str(i)
272             'jal a 20'
273
274             >>> p = Program(10, env={}, insts=[Jal("a", 20)])
275             >>> p.eval()
276             >>> p.get_pc(), p.get_val("a")
277             (20, 2)
278
279             >>> p = Program(10, env={}, insts=[Jal("x0", 20)])
280             >>> p.eval()
281             >>> p.get_pc(), p.get_val("x0")
282             (20, 0)
283             """
284
285             def __init__(self, rd, lab=None):
286                 assert isinstance(rd, str)
287                 self.rd = rd
288                 if lab != None:
289                     assert isinstance(lab, int)
290                     self.lab = lab
291
292             def get_opcode(self):
293                 return "jal"
294
295             def __str__(self):
296                 op = self.get_opcode()
297                 return f"{op} {self.rd} {self.lab}"
298
299             def eval(self, prog):
300                 if self.rd != "x0":
301                     # Notice that Jal and Jalr set pc to pc + 1. However, when we fetch
302                     # an instruction, we already increment the PC. Therefore, by using
303                     # get_pc, we are indeed, reading pc + 1.
304                     prog.set_val(self.rd, prog.get_pc())
305                     prog.set_pc(self.lab)
306
307
308         class Jalr(BranchOp):
309             """
310             jalr rd, rs, offset:

```

```

310     jalr rd, rs, offset
311     The jalr rd, rs, offset instruction performs an indirect jump to the
312     address computed by adding the value in rs to the immediate offset, and
313     stores the address of the instruction following the jump into rd.
314
315     Example:
316     >>> i = Jalr("a", "b", 20)
317     >>> str(i)
318     'jalr a b 20'
319
320     >>> p = Program(10, env={"b":30}, insts=[Jalr("a", "b", 20)])
321     >>> p.eval()
322     >>> p.get_pc(), p.get_val("a")
323     (50, 2)
324
325     >>> p = Program(10, env={"b":30}, insts=[Jalr("x0", "b", 20)])
326     >>> p.eval()
327     >>> p.get_pc(), p.get_val("x0")
328     (50, 0)
329     """
330
331     def __init__(self, rd, rs, offset=0):
332         assert isinstance(rd, str) and isinstance(rs, str)
333         self.rd = rd
334         self.rs = rs
335         if offset != None:
336             assert isinstance(offset, int)
337             self.offset = offset
338
339     def get_opcode(self):
340         return "jalr"
341
342     def __str__(self):
343         op = self.get_opcode()
344         return f"{op} {self.rd} {self.rs} {self.offset}"
345
346     def eval(self, prog):
347         rs_val = prog.get_val(self.rs)
348         if self.rd != "x0":
349             prog.set_val(self.rd, prog.get_pc())
350             prog.set_pc(rs_val + self.offset)
351
352
353     class MemOp(Inst):
354         """
355         The general class of instructions that access memory. These instructions
356         include loads and stores.
357         """
358
359         def __init__(self, rs1, offset, reg):
360             assert isinstance(rs1, str) and isinstance(reg, str) and isinstance(offset, int)
361             self.rs1 = rs1
362             self.offset = offset
363             self.reg = reg
364
365         def __str__(self):
366             op = self.get_opcode()
367             return f"{op} {self.reg}, {self.offset}({self.rs1})"
368
369     class Sw(MemOp):
370         """
371         sw reg, offset(rs1)
372         *(rs1 + offset) = reg
373
374         * reg: The source register containing the data to be stored.
375         * rs1: The base register containing the memory address.
376         * offset: A 12-bit signed immediate that is added to rs1 to form the
377           effective address.
378
379         Example:
380         >>> i = Sw("a", 0, "b")
381         >>> str(i)
382         'sw b, 0(a)'
383
384         >>> p = Program(10, env={"b":2, "a":3}, insts=[Sw("a", 0, "b")])
385         >>> p.eval()
386         >>> p.get_mem(3)
387         2
388         """
389
390         def eval(self, prog):
391             val = prog.get_val(self.reg)
392             addr = prog.get_val(self.rs1) + self.offset
393             prog.set_mem(addr, val)
394
395         def get_opcode(self):
396             return "sw"
397
398     class Lw(MemOp):
399         """
400         lw reg, offset(rs1)
401         reg = *(rs1 + offset)
402
403         * reg: The destination register that will be overwritten.
404         * rs1: The base register containing the memory address.
405         * offset: A 12-bit signed immediate that is added to rs1 to form the
406           effective address.
407
408         Example:
409         >>> i = Lw("a", 0, "b")
410         >>> str(i)
411         'lw b, 0(a)'

```

```

414
415     >>> p = Program(10, env={"a":2}, insts=[Lw("a", 0, "b")])
416     >>> p.eval()
417     >>> p.get_val("b")
418     0
419
420     >>> insts = [Sw("a", 0, "b"), Lw("a", 0, "c")]
421     >>> p = Program(10, env={"a":2, "b":5}, insts=insts)
422     >>> p.eval()
423     >>> p.get_val("c")
424     5
425
426
427     def eval(self, prog):
428         addr = prog.get_val(self.rs1) + self.offset
429         val = prog.get_mem(addr)
430         prog.set_val(self.reg, val)
431
432     def get_opcode(self):
433         return "lw"
434
435
436     class BinOp(Inst):
437         """
438         The general class of binary instructions. These instructions define a
439         value, and use two values.
440         """
441
442         def __init__(self, rd, rs1, rs2):
443             assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(rs2, str)
444             self.rd = rd
445             self.rs1 = rs1
446             self.rs2 = rs2
447
448         def __str__(self):
449             op = self.get_opcode()
450             return f"{self.rd} = {op} {self.rs1} {self.rs2}"
451
452
453     class BinOpImm(Inst):
454         """
455         The general class of binary instructions where the second operand is an
456         integer constant. These instructions define a value, and use one variable
457         and one immediate constant.
458         """
459
460         def __init__(self, rd, rs1, imm):
461             assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
462             self.rd = rd
463             self.rs1 = rs1
464             self.imm = imm
465
466         def __str__(self):
467             op = self.get_opcode()
468             return f"{self.rd} = {op} {self.rs1} {self.imm}"
469
470
471     class Add(BinOp):
472         """
473         add rd, rs1, rs2: rd = rs1 + rs2
474
475         Example:
476         >>> i = Add("a", "b0", "b1")
477         >>> str(i)
478         'a = add b0 b1'
479
480         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
481         >>> p.eval()
482         >>> p.get_val("a")
483         5
484
485
486         def eval(self, prog):
487             rs1 = prog.get_val(self.rs1)
488             rs2 = prog.get_val(self.rs2)
489             prog.set_val(self.rd, rs1 + rs2)
490
491         def get_opcode(self):
492             return "add"
493
494
495     class Addi(BinOpImm):
496         """
497         addi rd, rs1, imm: rd = rs1 + imm
498
499         Example:
500         >>> i = Addi("a", "b0", 1)
501         >>> str(i)
502         'a = addi b0 1'
503
504         >>> p = Program(0, env={"b0":2}, insts=[Addi("a", "b0", 3)])
505         >>> p.eval()
506         >>> p.get_val("a")
507         5
508
509
510         def eval(self, prog):
511             rs1 = prog.get_val(self.rs1)
512             prog.set_val(self.rd, rs1 + self.imm)
513
514         def get_opcode(self):
515             return "addi"
516

```

```

517
518 class Mul(BinOp):
519     """
520     mul rd, rs1, rs2: rd = rs1 * rs2
521
522     Example:
523     >>> i = Mul("a", "b0", "b1")
524     >>> str(i)
525     'a = mul b0 b1'
526
527     >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
528     >>> p.eval()
529     >>> p.get_val("a")
530     6
531     """
532
533     def eval(self, prog):
534         rs1 = prog.get_val(self.rs1)
535         rs2 = prog.get_val(self.rs2)
536         prog.set_val(self.rd, rs1 * rs2)
537
538     def get_opcode(self):
539         return "mul"
540
541
542 class Sub(BinOp):
543     """
544     sub rd, rs1, rs2: rd = rs1 - rs2
545
546     Example:
547     >>> i = Sub("a", "b0", "b1")
548     >>> str(i)
549     'a = sub b0 b1'
550
551     >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
552     >>> p.eval()
553     >>> p.get_val("a")
554     -1
555     """
556
557     def eval(self, prog):
558         rs1 = prog.get_val(self.rs1)
559         rs2 = prog.get_val(self.rs2)
560         prog.set_val(self.rd, rs1 - rs2)
561
562     def get_opcode(self):
563         return "sub"
564
565
566 class Xor(BinOp):
567     """
568     xor rd, rs1, rs2: rd = rs1 ^ rs2
569
570     Example:
571     >>> i = Xor("a", "b0", "b1")
572     >>> str(i)
573     'a = xor b0 b1'
574
575     >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
576     >>> p.eval()
577     >>> p.get_val("a")
578     1
579     """
580
581     def eval(self, prog):
582         rs1 = prog.get_val(self.rs1)
583         rs2 = prog.get_val(self.rs2)
584         prog.set_val(self.rd, rs1 ^ rs2)
585
586     def get_opcode(self):
587         return "xor"
588
589
590 class Xori(BinOpImm):
591     """
592     xori rd, rs1, imm: rd = rs1 ^ imm
593
594     Example:
595     >>> i = Xori("a", "b0", 10)
596     >>> str(i)
597     'a = xori b0 10'
598
599     >>> p = Program(0, env={"b0":2}, insts=[Xori("a", "b0", 3)])
600     >>> p.eval()
601     >>> p.get_val("a")
602     1
603     """
604
605     def eval(self, prog):
606         rs1 = prog.get_val(self.rs1)
607         prog.set_val(self.rd, rs1 ^ self.imm)
608
609     def get_opcode(self):
610         return "xori"
611
612
613 class Div(BinOp):
614     """
615     div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
616     Notice that RISC-V does not have an instruction exactly like this one.
617     The div operator works on floating-point numbers; not on integers.
618
619     Example:

```

```

620     >>> i = Div("a", "b0", "b1")
621     >>> str(i)
622     'a = div b0 b1'
623
624     >>> p = Program(0, env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
625     >>> p.eval()
626     >>> p.get_val("a")
627     2
628     """
629
630     def eval(self, prog):
631         rs1 = prog.get_val(self.rs1)
632         rs2 = prog.get_val(self.rs2)
633         prog.set_val(self.rd, rs1 // rs2)
634
635     def get_opcode(self):
636         return "div"
637
638
639     class Slt(BinOp):
640         """
641         slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
642
643         Example:
644         >>> i = Slt("a", "b0", "b1")
645         >>> str(i)
646         'a = slt b0 b1'
647
648         >>> p = Program(0, env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
649         >>> p.eval()
650         >>> p.get_val("a")
651         1
652
653         >>> p = Program(0, env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
654         >>> p.eval()
655         >>> p.get_val("a")
656         0
657
658         >>> p = Program(0, env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
659         >>> p.eval()
660         >>> p.get_val("a")
661         0
662         """
663
664     def eval(self, prog):
665         rs1 = prog.get_val(self.rs1)
666         rs2 = prog.get_val(self.rs2)
667         prog.set_val(self.rd, 1 if rs1 < rs2 else 0)
668
669     def get_opcode(self):
670         return "slt"
671
672
673     class Slti(BinOpImm):
674         """
675         slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
676         (signed comparison with immediate)
677
678         Example:
679         >>> i = Slti("a", "b0", 0)
680         >>> str(i)
681         'a = slti b0 0'
682
683         >>> p = Program(0, env={"b0":2}, insts=[Slti("a", "b0", 3)])
684         >>> p.eval()
685         >>> p.get_val("a")
686         1
687
688         >>> p = Program(0, env={"b0":3}, insts=[Slti("a", "b0", 3)])
689         >>> p.eval()
690         >>> p.get_val("a")
691         0
692
693         >>> p = Program(0, env={"b0":3}, insts=[Slti("a", "b0", 2)])
694         >>> p.eval()
695         >>> p.get_val("a")
696         0
697         """
698
699     def eval(self, prog):
700         rs1 = prog.get_val(self.rs1)
701         prog.set_val(self.rd, 1 if rs1 < self.imm else 0)
702
703     def get_opcode(self):
704         return "slti"

```

Optimizer.py

```

1 from abc import ABC, abstractmethod
2 from Asm import *
3
4
5 class Optimizer(ABC):
6     """
7     This class implements an "Optimization Pass". The pass receives a sequence
8     of instructions stored in a program, and produces a new sequence of
9     instructions.
10    """
11
12    def __init__(self, prog):
13        self.prog = prog
14
15    @abstractmethod
16    def optimize(self):
17        pass
18
19
20 class RegAllocator(Optimizer):
21     """This file implements the register allocation pass."""
22
23    def __init__(self, prog):
24        # TODO: you might want to save/initialize some stuff in the ctor.
25        super().__init__(prog)
26
27    def get_val(self, var):
28        """
29        Informs the value that is associated with the variable var within
30        the program prog.
31        """
32        # TODO: Implement this method.
33        raise NotImplementedError
34
35
36    def optimize(self):
37        """
38        This function perform register allocation. It maps variables into
39        memory, and changes instructions, so that they use one of the following
40        registers:
41        * x0: always the value zero. Can't change.
42        * sp: the stack pointer. Starts with the memory size.
43        * ra: the return address.
44        * a0: function argument 0 (or return address)
45        * a1: function argument 1
46        * a2: function argument 2
47        * a3: function argument 3
48
49        Notice that next to each register we have suggested a usage. You can,
50        of course, write on them and use them in other ways. But, at least x0
51        and sp you should not overwrite. The first register you can't overwrite,
52        actually. And sp is initialized with the number of memory addresses.
53        It's good to use it to control the function stack.
54
55        Examples:
56        >>> insts = [Addi("a", "x0", 3)]
57        >>> p = Program(1000, env={}, insts=insts)
58        >>> o = RegAllocator(p)
59        >>> o.optimize()
60        >>> p.eval()
61        >>> p.get_val("a1")
62        3
63
64        >>> insts = [Addi("a", "x0", 1), Slti("b", "a", 2)]
65        >>> p = Program(1000, env={}, insts=insts)
66        >>> o = RegAllocator(p)
67        >>> o.optimize()
68        >>> p.eval()
69        >>> p.get_val("a1")
70        1
71
72        >>> insts = [Addi("a", "x0", 3), Slti("b", "a", 2), Xori("c", "b", 5)]
73        >>> p = Program(1000, env={}, insts=insts)
74        >>> o = RegAllocator(p)
75        >>> o.optimize()
76        >>> p.eval()
77        >>> p.get_val("a1")
78        5
79
80        >>> insts = [Addi("sp", "sp", -1), Addi("a", "x0", 7), Sw("sp", 0, "a")]
81        >>> p = Program(1000, env={}, insts=insts)
82        >>> o = RegAllocator(p)
83        >>> o.optimize()
84        >>> p.eval()
85        >>> p.get_mem(p.get_val("sp"))
86        7
87
88        >>> insts = [Addi("sp", "sp", -1), Addi("a", "x0", 7), Sw("sp", 0, "a")]
89        >>> insts += [Lw("sp", 0, "b"), Addi("c", "b", 6)]
90        >>> p = Program(1000, env={}, insts=insts)
91        >>> o = RegAllocator(p)
92        >>> o.optimize()
93        >>> p.eval()
94        >>> p.get_val("a1")
95        13
96
97        >>> insts = [Addi("a", "x0", 3), Addi("b", "x0", 4), Add("c", "a", "b")]
98        >>> p = Program(1000, env={}, insts=insts)
99        >>> o = RegAllocator(p)
100        >>> o.optimize()
101        >>> p.eval()
102        >>> p.get_val("a1")
103        7

```

```

104
105 >>> insts = [Addi("a", "x0", 28),Addi("b", "x0", 4),Div("c", "a", "b")]
106 >>> p = Program(1000, env={}, insts=insts)
107 >>> o = RegAllocator(p)
108 >>> o.optimize()
109 >>> p.eval()
110 >>> p.get_val("a1")
111 7
112
113 >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Mul("c", "a", "b")]
114 >>> p = Program(1000, env={}, insts=insts)
115 >>> o = RegAllocator(p)
116 >>> o.optimize()
117 >>> p.eval()
118 >>> p.get_val("a1")
119 12
120
121 >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Xor("c", "a", "b")]
122 >>> p = Program(1000, env={}, insts=insts)
123 >>> o = RegAllocator(p)
124 >>> o.optimize()
125 >>> p.eval()
126 >>> p.get_val("a1")
127 7
128
129 >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Slt("c", "a", "b")]
130 >>> p = Program(1000, env={}, insts=insts)
131 >>> o = RegAllocator(p)
132 >>> o.optimize()
133 >>> p.eval()
134 >>> p.get_val("a1")
135 1
136
137 >>> insts = [Addi("a", "x0", 3),Addi("b", "x0", 4),Slt("c", "b", "a")]
138 >>> p = Program(1000, env={}, insts=insts)
139 >>> o = RegAllocator(p)
140 >>> o.optimize()
141 >>> p.eval()
142 >>> p.get_val("a1")
143 0
144
145 If you want, you can allocate Jal/Jalr/Beq instructions, but that's not
146 necessary for this exercise.
147
148 >>> insts = [Jal("a", 30)]
149 >>> p = Program(1000, env={}, insts=insts)
150 >>> o = RegAllocator(p)
151 >>> o.optimize()
152 >>> p.eval()
153 >>> (p.get_pc(), p.get_val("a1") > 0)
154 (30, True)
155
156 >>> insts = [Addi("a", "x0", 30), Jalr("b", "a")]
157 >>> p = Program(1000, env={}, insts=insts)
158 >>> o = RegAllocator(p)
159 >>> o.optimize()
160 >>> p.eval()
161 >>> (p.get_pc(), p.get_val("a1") > 0)
162 (30, True)
163
164 >>> insts = [Addi("a", "x0", 3), Addi("b", "a", 0), Beq("a", "b", 30)]
165 >>> p = Program(1000, env={}, insts=insts)
166 >>> o = RegAllocator(p)
167 >>> o.optimize()
168 >>> p.eval()
169 >>> p.get_pc()
170 30
171 """
172 # TODO: Implement this method.
173 raise NotImplementedError
174 # Hints:
175 # new_insts = []
176 # for inst in self.prog.get_insts():
177 #     action = self.alloc_action[inst.get_opcode()]
178 #     last_insts = action(inst, self)
179 #     new_insts += last_insts
180 # self.prog.set_insts(new_insts)

```

[VPL](#)