

2025_2 - COMPILADORES - METATURMA

[PAÍNEL](#) > [MINHAS TURMAS](#) > [2025_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)
 > [AV14 - GERAÇÃO DE CÓDIGO PARA FUNÇÕES](#)

[Descrição](#)

[Enviar](#)

[Editar](#)

[Visualizar envios](#)

AV14 - Geração de código para funções

Data de entrega: sexta, 28 Nov 2025, 23:59

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

Tipo de trabalho: Trabalho individual

O objetivo deste trabalho prático é gerar código para expressões que contém funções anônimas. Para tanto, além das instruções utilizadas nos exercícios anteriores, iremos usar mais três instruções:

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

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

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

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

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

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

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

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

6
```

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

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

1
```

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

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

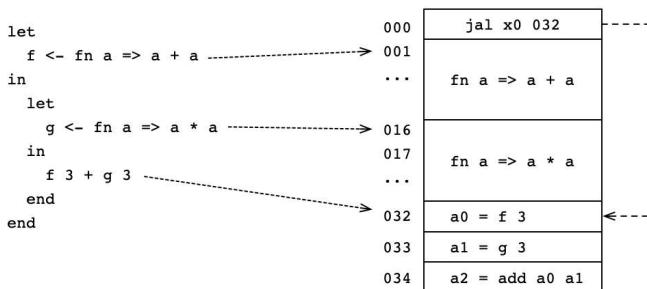
7
```

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

```
(fn x => x * x) 3           let f <- fn x => x * x in f 3 end
9                                9
```

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

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



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

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

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

Implementando funções

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

```

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

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

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

```

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

Submetendo e Testando

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

```

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

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

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

No exemplo acima, 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 def rename_variables(exp):
9     """
10    Esta função invoca o renomeador de variáveis. Ela deve ser usada antes do
11    inicio da fase de geração de código.
12    """
13    ren = RenameVisitor()
14    exp.accept(ren, {})
15    return exp
16
17 if __name__ == "__main__":
18     """
19     Este arquivo não deve ser alterado, mas deve ser enviado para resolver o
20     VPL. O arquivo contém o código que testa a implementação do parser.
21     """
22     text = sys.stdin.read()
23     lexer = Lexer(text)
24     parser = Parser(lexer.tokens())
25     exp = rename_variables(parser.parse())
26     prog = AsmModule.Program(memory_size = 1000, env = {}, insts = [])
27     gen = GenVisitor()
28     var_answer = exp.accept(gen, prog)
29     prog.eval()
30     print(f"Answer: {prog.get_val(var_answer)}")

```

`Lexer.py`

```

1 import sys
2 import enum
3
4
5 class Token:
6     """
7         This class contains the definition of Tokens. A token has two fields: its
8         text and its kind. The "kind" of a token is a constant that identifies it
9         uniquely. See the TokenType to know the possible identifiers (if you want).
10        You don't need to change this class.
11        """
12    def __init__(self, tokenText, tokenKind):
13        # The token's actual text. Used for identifiers, strings, and numbers.
14        self.text = tokenText
15        # The TokenType that this token is classified as.
16        self.kind = tokenKind
17
18
19 class TokenType(enum.Enum):
20     """
21         These are the possible tokens. You don't need to change this class at all.
22         """
23
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 # x = y
37     ADD = 202 # x + y
38     SUB = 203 # x - y
39     MUL = 204 # x * y
40     DIV = 205 # x / y
41     LEQ = 206 # x <= y
42     LTH = 207 # x < y
43     NEG = 208 # ~x
44     NOT = 209 # not x
45     LPR = 210 # (
46     RPR = 211 # )
47     ASN = 212 # The assignment '<->' operator
48     ORX = 213 # x or y
49     AND = 214 # x and y
50     IFX = 215 # The 'if' of a conditional expression
51     THN = 216 # The 'then' of a conditional expression
52     ELS = 217 # The 'else' of a conditional expression
53     FNX = 218 # The 'fn' that declares an anonymous function
54     ARW = 219 # The '>=' that separates the parameter from the body of function
55
56
57    def tokens(self):
58        """
59            This method is a token generator: it converts the string encapsulated
60            into this object into a sequence of Tokens. Notice that this method
61            filters out three kinds of tokens: white-spaces, comments and new lines.
62
63        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\n')
70        >>> [tk.kind for tk in l.tokens()]
71        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
72
73        >>> l = Lexer('1 * 2 -- 3\n')
74        >>> [tk.kind for tk in l.tokens()]
75        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
76
77        >>> l = Lexer("1 + var")
78        >>> [tk.kind for tk in l.tokens()]
79        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
80
81        >>> l = Lexer("let v <- 2 in v end")
82        >>> [tk.kind.name for tk in l.tokens()]
83        ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
84        """
85
86        token = self.getToken()
87        while token.kind != TokenType.EOF:
88            if (
89                token.kind != TokenType.WSP
90                and token.kind != TokenType.COM
91                and token.kind != TokenType.NLN
92            ):
93                yield token
94            token = self.getToken()
95
96    def getToken(self):
97        """
98            Return the next token.
99            TODO: Implement this method (you can reuse Lab 5: Visitors)!
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 a parser for SML with anonymous functions. The grammar is
8 as follows:
9
10 fn_exp ::= fn <var> => fn_exp
11     | if_exp
12 if_exp ::= <if> if_exp <then> fn_exp <else> fn_exp
13     | or_exp
14 or_exp ::= and_exp (or and_exp)*
15 and_exp ::= eq_exp (and eq_exp)*
16 eq_exp ::= cmp_exp (= cmp_exp)*
17 cmp_exp ::= add_exp ([<|=|>] add_exp)*
18 add_exp ::= mul_exp ([+|-] mul_exp)*
19 mul_exp ::= unary_exp ([*//] unary_exp)*
20 unary_exp ::= <not> unary_exp
21     | ~ unary_exp
22     | let_exp
23 let_exp ::= <let> <var> <- fn_exp <in> fn_exp <end>
24     | val_exp
25 val_exp ::= val_tk (val_tk)*
26 val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>
27
28 References:
29     see https://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm#classic
30 """
31
32
33 class Parser:
34     def __init__(self, tokens):
35         """
36             Initializes the parser. The parser keeps track of the list of tokens
37             and the current token. For instance:
38         """
39         self.tokens = list(tokens)
40         self.cur_token_idx = 0 # This is just a suggestion!
41         # You can (and probably should!) modify this method.
42
43     def parse(self):
44         """
45             Returns the expression associated with the stream of tokens.
46
47             Examples:
48             >>> parser = Parser([Token('123', TokenType.NUM)])
49             >>> exp = parser.parse()
50             >>> ev = EvalVisitor()
51             >>> exp.accept(ev, None)
52             123
53
54             >>> parser = Parser([Token('True', TokenType.TRU)])
55             >>> exp = parser.parse()
56             >>> ev = EvalVisitor()
57             >>> exp.accept(ev, None)
58             True
59
60             >>> parser = Parser([Token('False', TokenType.FLS)])
61             >>> exp = parser.parse()
62             >>> ev = EvalVisitor()
63             >>> exp.accept(ev, None)
64             False
65
66             >>> tk0 = Token('~', TokenType.NEG)
67             >>> tk1 = Token('123', TokenType.NUM)
68             >>> parser = Parser([tk0, tk1])
69             >>> exp = parser.parse()
70             >>> ev = EvalVisitor()
71             >>> exp.accept(ev, None)
72             -123
73
74             >>> tk0 = Token('3', TokenType.NUM)
75             >>> tk1 = Token('*', TokenType.MUL)
76             >>> tk2 = Token('4', TokenType.NUM)
77             >>> parser = Parser([tk0, tk1, tk2])
78             >>> exp = parser.parse()
79             >>> ev = EvalVisitor()
80             >>> exp.accept(ev, None)
81             12
82
83             >>> tk0 = Token('3', TokenType.NUM)
84             >>> tk1 = Token('*', TokenType.MUL)
85             >>> tk2 = Token('~', TokenType.NEG)
86             >>> tk3 = Token('4', TokenType.NUM)
87             >>> parser = Parser([tk0, tk1, tk2, tk3])
88             >>> exp = parser.parse()
89             >>> ev = EvalVisitor()
90             >>> exp.accept(ev, None)
91             -12
92
93             >>> tk0 = Token('30', TokenType.NUM)
94             >>> tk1 = Token('/', TokenType.DIV)
95             >>> tk2 = Token('4', TokenType.NUM)
96             >>> parser = Parser([tk0, tk1, tk2])
97             >>> exp = parser.parse()
98             >>> ev = EvalVisitor()
99             >>> exp.accept(ev, None)
100            7
101
102            >>> tk0 = Token('3', TokenType.NUM)
103            >>> tk1 = Token('+', TokenType.ADD)

```

```

104     >>> tk2 = Token('4', TokenType.NUM)
105     >>> parser = Parser([tk0, tk1, tk2])
106     >>> exp = parser.parse()
107     >>> ev = EvalVisitor()
108     >>> exp.accept(ev, None)
109     7
110
111     >>> tk0 = Token('30', TokenType.NUM)
112     >>> tk1 = Token('-', TokenType.SUB)
113     >>> tk2 = Token('4', TokenType.NUM)
114     >>> parser = Parser([tk0, tk1, tk2])
115     >>> exp = parser.parse()
116     >>> ev = EvalVisitor()
117     >>> exp.accept(ev, None)
118     26
119
120     >>> tk0 = Token('2', TokenType.NUM)
121     >>> tk1 = Token('*', TokenType.MUL)
122     >>> tk2 = Token('(', TokenType.LPR)
123     >>> tk3 = Token('3', TokenType.NUM)
124     >>> tk4 = Token('+', TokenType.ADD)
125     >>> tk5 = Token('4', TokenType.NUM)
126     >>> tk6 = Token(')', TokenType.RPR)
127     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
128     >>> exp = parser.parse()
129     >>> ev = EvalVisitor()
130     >>> exp.accept(ev, None)
131     14
132
133     >>> tk0 = Token('4', TokenType.NUM)
134     >>> tk1 = Token('==', TokenType.EQL)
135     >>> tk2 = Token('4', TokenType.NUM)
136     >>> parser = Parser([tk0, tk1, tk2])
137     >>> exp = parser.parse()
138     >>> ev = EvalVisitor()
139     >>> exp.accept(ev, None)
140     True
141
142     >>> tk0 = Token('4', TokenType.NUM)
143     >>> tk1 = Token('<', TokenType.LEQ)
144     >>> tk2 = Token('4', TokenType.NUM)
145     >>> parser = Parser([tk0, tk1, tk2])
146     >>> exp = parser.parse()
147     >>> ev = EvalVisitor()
148     >>> exp.accept(ev, None)
149     True
150
151     >>> tk0 = Token('4', TokenType.NUM)
152     >>> tk1 = Token('<', TokenType.LTH)
153     >>> tk2 = Token('4', TokenType.NUM)
154     >>> parser = Parser([tk0, tk1, tk2])
155     >>> exp = parser.parse()
156     >>> ev = EvalVisitor()
157     >>> exp.accept(ev, None)
158     False
159
160     >>> tk0 = Token('not', TokenType.NOT)
161     >>> tk1 = Token('(', TokenType.LPR)
162     >>> tk2 = Token('4', TokenType.NUM)
163     >>> tk3 = Token('<', TokenType.LTH)
164     >>> tk4 = Token('4', TokenType.NUM)
165     >>> tk5 = Token(')', TokenType.RPR)
166     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
167     >>> exp = parser.parse()
168     >>> ev = EvalVisitor()
169     >>> exp.accept(ev, None)
170     True
171
172     >>> tk0 = Token('true', TokenType.TRU)
173     >>> tk1 = Token('or', TokenType.ORX)
174     >>> tk2 = Token('false', TokenType.FLS)
175     >>> parser = Parser([tk0, tk1, tk2])
176     >>> exp = parser.parse()
177     >>> ev = EvalVisitor()
178     >>> exp.accept(ev, None)
179     True
180
181     >>> tk0 = Token('true', TokenType.TRU)
182     >>> tk1 = Token('and', TokenType.AND)
183     >>> tk2 = Token('false', TokenType.FLS)
184     >>> parser = Parser([tk0, tk1, tk2])
185     >>> exp = parser.parse()
186     >>> ev = EvalVisitor()
187     >>> exp.accept(ev, None)
188     False
189
190     >>> tk0 = Token('let', TokenType.LET)
191     >>> tk1 = Token('v', TokenType.VAR)
192     >>> tk2 = Token('<', TokenType.ASN)
193     >>> tk3 = Token('42', TokenType.NUM)
194     >>> tk4 = Token('in', TokenType.INX)
195     >>> tk5 = Token('v', TokenType.VAR)
196     >>> tk6 = Token('end', TokenType.END)
197     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
198     >>> exp = parser.parse()
199     >>> ev = EvalVisitor()
200     >>> exp.accept(ev, {})
201     42
202
203     >>> tk0 = Token('let', TokenType.LET)
204     >>> tk1 = Token('v', TokenType.VAR)
205     >>> tk2 = Token('<', TokenType.ASN)
206     >>> tk3 = Token('21', TokenType.NUM)
207     >>> tk4 = Token('in', TokenType.INX)

```

```

20/
208    >>> tk4 = token('in', TokenType.IN)
209    >>> tk5 = Token('v', TokenType.VAR)
210    >>> tk6 = Token('+', TokenType.ADD)
211    >>> tk7 = Token('v', TokenType.VAR)
212    >>> tk8 = Token('end', TokenType.END)
213    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
214    >>> exp = parser.parse()
215    >>> ev = EvalVisitor()
216    >>> exp.accept(ev, {})
217
218    >>> tk0 = Token('if', TokenType.IFX)
219    >>> tk1 = Token('2', TokenType.NUM)
220    >>> tk2 = Token('<', TokenType.LTH)
221    >>> tk3 = Token('3', TokenType.NUM)
222    >>> tk4 = Token('then', TokenType.THN)
223    >>> tk5 = Token('1', TokenType.NUM)
224    >>> tk6 = Token('else', TokenType.ELS)
225    >>> tk7 = Token('2', TokenType.NUM)
226    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
227    >>> exp = parser.parse()
228    >>> ev = EvalVisitor()
229    >>> exp.accept(ev, None)
230
231    1
232
233    >>> tk0 = Token('if', TokenType.IFX)
234    >>> tk1 = Token('false', TokenType.FLS)
235    >>> tk2 = Token('then', TokenType.THN)
236    >>> tk3 = Token('1', TokenType.NUM)
237    >>> tk4 = Token('else', TokenType.ELS)
238    >>> tk5 = Token('2', TokenType.NUM)
239    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
240    >>> exp = parser.parse()
241    >>> ev = EvalVisitor()
242    >>> exp.accept(ev, None)
243
244    2
245
246    >>> tk0 = Token('fn', TokenType.FNX)
247    >>> tk1 = Token('v', TokenType.VAR)
248    >>> tk2 = Token('=>', TokenType.ARW)
249    >>> tk3 = Token('v', TokenType.VAR)
250    >>> tk4 = Token('+', TokenType.ADD)
251    >>> tk5 = Token('1', TokenType.NUM)
252    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
253    >>> exp = parser.parse()
254    >>> ev = EvalVisitor()
255    >>> print(exp.accept(ev, None))
Fn(v)
256
257    >>> tk0 = Token('(', TokenType.LPR)
258    >>> tk1 = Token('fn', TokenType.FNX)
259    >>> tk2 = Token('v', TokenType.VAR)
260    >>> tk3 = Token('=>', TokenType.ARW)
261    >>> tk4 = Token('v', TokenType.VAR)
262    >>> tk5 = Token('+', TokenType.ADD)
263    >>> tk6 = Token('1', TokenType.NUM)
264    >>> tk7 = Token(')', TokenType.RPR)
265    >>> tk8 = Token('2', TokenType.NUM)
266    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
267    >>> exp = parser.parse()
268    >>> ev = EvalVisitor()
269    >>> exp.accept(ev, {})
270
271    3
272
273    # TODO: implement this method.
274    return None

```

Expression.py

```

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

```

```

104
105
106 class Mul(BinaryExpression):
107     """
108     This class represents multiplication of two expressions. The acceptuation of
109     such an expression is the product of the two subexpression's values.
110     """
111     def accept(self, visitor, arg):
112         return visitor.visit_mul(self, arg)
113
114
115 class Div(BinaryExpression):
116     """
117     This class represents the integer division of two expressions. The
118     acceptuation of such an expression is the integer quotient of the two
119     subexpression's values.
120     """
121     def accept(self, visitor, arg):
122         return visitor.visit_div(self, arg)
123
124
125 class Leq(BinaryExpression):
126     """
127     This class represents comparison of two expressions using the
128     less-than-or-equal comparator. The acceptuation of such an expression is a
129     boolean value that is true if the left operand is less than or equal the
130     right operand. It is false otherwise.
131     """
132     def accept(self, visitor, arg):
133         return visitor.visit_leq(self, arg)
134
135
136 class Lth(BinaryExpression):
137     """
138     This class represents comparison of two expressions using the
139     less-than comparison operator. The acceptuation of such an expression is a
140     boolean value that is true if the left operand is less than the right
141     operand. It is false otherwise.
142     """
143     def accept(self, visitor, arg):
144         return visitor.visit_lth(self, arg)
145
146
147 class UnaryExpression(Expression):
148     """
149     This class represents unary expressions. A unary expression has only one
150     sub-expression.
151     """
152     def __init__(self, exp):
153         self.exp = exp
154
155     @abstractmethod
156     def accept(self, visitor, arg):
157         raise NotImplementedError
158
159
160 class Neg(UnaryExpression):
161     """
162     This expression represents the additive inverse of a number. The additive
163     inverse of a number n is the number -n, so that the sum of both is zero.
164     """
165     def accept(self, visitor, arg):
166         return visitor.visit_neg(self, arg)
167
168
169 class Not(UnaryExpression):
170     """
171     This expression represents the negation of a boolean. The negation of a
172     boolean expression is the logical complement of that expression.
173     """
174     def accept(self, visitor, arg):
175         return visitor.visit_not(self, arg)
176
177
178 class Let(Expression):
179     """
180     This class represents a let expression. The semantics of a let expression,
181     such as "let v <- e0 in e1" on an environment env is as follows:
182     1. Evaluate e0 in the environment env, yielding e0_val
183     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
184     """
185     def __init__(self, identifier, exp_def, exp_body):
186         self.identifier = identifier
187         self.exp_def = exp_def
188         self.exp_body = exp_body
189     def accept(self, visitor, arg):
190         return visitor.visit_let(self, arg)
191
192
193 class IfThenElse(Expression):
194     """
195     This class represents a conditional expression. The semantics an expression
196     such as 'if B then E0 else E1' is as follows:
197     1. Evaluate B. Call the result ValueB.
198     2. If ValueB is True, then evaluate E0 and return the result.
199     3. If ValueB is False, then evaluate E1 and return the result.
200     Notice that we only evaluate one of the two sub-expressions, not both. Thus,
201     "if True then 0 else 1 div 0" will return 0 indeed.
202     """
203     def __init__(self, cond, e0, e1):
204         self.cond = cond
205         self.e0 = e0
206         self.e1 = e1
207         ...
208

```

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

Visitor.py

```

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    @abstractmethod
17    def visit_var(self, exp, arg):
18        pass
19
20    @abstractmethod
21    def visit_bln(self, exp, arg):
22        pass
23
24    @abstractmethod
25    def visit_num(self, exp, arg):
26        pass
27
28    @abstractmethod
29    def visit_eql(self, exp, arg):
30        pass
31
32    @abstractmethod
33    def visit_and(self, exp, arg):
34        pass
35
36    @abstractmethod
37    def visit_or(self, exp, arg):
38        pass
39
40    @abstractmethod
41    def visit_add(self, exp, arg):
42        pass
43
44    @abstractmethod
45    def visit_sub(self, exp, arg):
46        pass
47
48    @abstractmethod
49    def visit_mul(self, exp, arg):
50        pass
51
52    @abstractmethod
53    def visit_div(self, exp, arg):
54        pass
55
56    @abstractmethod
57    def visit_leq(self, exp, arg):
58        pass
59
60    @abstractmethod
61    def visit_lth(self, exp, arg):
62        pass
63
64    @abstractmethod
65    def visit_neg(self, exp, arg):
66        pass
67
68    @abstractmethod
69    def visit_not(self, exp, arg):
70        pass
71
72    @abstractmethod
73    def visit_let(self, exp, arg):
74        pass
75
76    @abstractmethod
77    def visit_ifThenElse(self, exp, arg):
78        pass
79
80    @abstractmethod
81    def visit_fn(self, exp, arg):
82        pass
83
84    @abstractmethod
85    def visit_app(self, exp, arg):
86        pass
87
88
89 class GenVisitor(Visitor):
90     """
91         The GenVisitor class compiles arithmetic expressions into a low-level
92         language.
93     """
94
95     def __init__(self):
96         self.next_var_counter = 0
97
98     def next_var_name(self):
99         self.next_var_counter += 1
100        return f"v{self.next_var_counter}"
101
102    def visit_var(self, exp, prog):
103        """

```



```

20/
208    >>> e = And(Bln(False), Bln(False))
209    >>> p = AsmModule.Program({}, [])
210    >>> g = GenVisitor()
211    >>> v = e.accept(g, p)
212    >>> p.eval()
213    >>> p.get_val(v)
214    0
215
216    >>> e = And(Bln(False), Div(Num(3), Num(0)))
217    >>> p = AsmModule.Program({}, [])
218    >>> g = GenVisitor()
219    >>> v = e.accept(g, p)
220    >>> p.eval()
221    >>> p.get_val(v)
222    0
223
224    # TODO: Implement this method.
225    raise NotImplementedError
226
227    def visit_or(self, exp, prog):
228        """
229            >>> e = Or(Bln(True), Bln(True))
230            >>> p = AsmModule.Program({}, [])
231            >>> g = GenVisitor()
232            >>> v = e.accept(g, p)
233            >>> p.eval()
234            >>> p.get_val(v)
235            1
236
237            >>> e = Or(Bln(False), Bln(True))
238            >>> p = AsmModule.Program({}, [])
239            >>> g = GenVisitor()
240            >>> v = e.accept(g, p)
241            >>> p.eval()
242            >>> p.get_val(v)
243            1
244
245            >>> e = Or(Bln(True), Bln(False))
246            >>> p = AsmModule.Program({}, [])
247            >>> g = GenVisitor()
248            >>> v = e.accept(g, p)
249            >>> p.eval()
250            >>> p.get_val(v)
251            1
252
253            >>> e = Or(Bln(False), Bln(False))
254            >>> p = AsmModule.Program({}, [])
255            >>> g = GenVisitor()
256            >>> v = e.accept(g, p)
257            >>> p.eval()
258            >>> p.get_val(v)
259            0
260
261            >>> e = Or(Bln(True), Div(Num(3), Num(0)))
262            >>> p = AsmModule.Program({}, [])
263            >>> g = GenVisitor()
264            >>> v = e.accept(g, p)
265            >>> p.eval()
266            >>> p.get_val(v)
267            1
268
269            # TODO: Implement this method.
270            raise NotImplementedError
271
272    def visit_add(self, exp, prog):
273        """
274            >>> e = Add(Num(13), Num(-13))
275            >>> p = AsmModule.Program({}, [])
276            >>> g = GenVisitor()
277            >>> v = e.accept(g, p)
278            >>> p.eval()
279            >>> p.get_val(v)
280            0
281
282            >>> e = Add(Num(13), Num(10))
283            >>> p = AsmModule.Program({}, [])
284            >>> g = GenVisitor()
285            >>> v = e.accept(g, p)
286            >>> p.eval()
287            >>> p.get_val(v)
288            23
289
290            # TODO: Implement this method.
291            raise NotImplementedError
292
293    def visit_sub(self, exp, prog):
294        """
295            >>> e = Sub(Num(13), Num(-13))
296            >>> p = AsmModule.Program({}, [])
297            >>> g = GenVisitor()
298            >>> v = e.accept(g, p)
299            >>> p.eval()
300            >>> p.get_val(v)
301            26
302
303            >>> e = Sub(Num(13), Num(10))
304            >>> p = AsmModule.Program({}, [])
305            >>> g = GenVisitor()
306            >>> v = e.accept(g, p)
307            >>> p.eval()
308            >>> p.get_val(v)
309            3
310
311            # TODO: Implement this method.

```

```

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

```

```

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

```

```

517         >>> v = e.accept(g, p)
518         >>> p.eval()
519         >>> p.get_val(v)
520         1
521
522         >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
523         >>> p = AsmModule.Program({}, [])
524         >>> g = GenVisitor()
525         >>> v = e.accept(g, p)
526         >>> p.eval()
527         >>> p.get_val(v)
528         5
529
530         >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
531         >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
532         >>> p = AsmModule.Program({}, [])
533         >>> g = GenVisitor()
534         >>> v = e1.accept(g, p)
535         >>> p.eval()
536         >>> p.get_val(v)
537         50
538
539         # TODO: Implement this method.
540         raise NotImplementedError
541
542     def visit_ifThenElse(self, exp, prog):
543         """
544             >>> e = IfThenElse(Bln(True), Num(3), Num(5))
545             >>> p = AsmModule.Program({}, [])
546             >>> g = GenVisitor()
547             >>> v = e.accept(g, p)
548             >>> p.eval()
549             >>> p.get_val(v)
550             3
551
552             >>> e = IfThenElse(Bln(False), Num(3), Num(5))
553             >>> p = AsmModule.Program({}, [])
554             >>> g = GenVisitor()
555             >>> v = e.accept(g, p)
556             >>> p.eval()
557             >>> p.get_val(v)
558             5
559
560             >>> e = IfThenElse(And(Bln(True), Bln(True)), Num(3), Num(5))
561             >>> p = AsmModule.Program({}, [])
562             >>> g = GenVisitor()
563             >>> v = e.accept(g, p)
564             >>> p.eval()
565             >>> p.get_val(v)
566             3
567
568             >>> e0 = Mul(Num(2), Add(Num(3), Num(4)))
569             >>> e1 = IfThenElse(And(Bln(True), Bln(False)), Num(3), e0)
570             >>> p = AsmModule.Program({}, [])
571             >>> g = GenVisitor()
572             >>> v = e1.accept(g, p)
573             >>> p.eval()
574             >>> p.get_val(v)
575             14
576
577             >>> e0 = Div(Num(2), Num(0))
578             >>> e1 = IfThenElse(Bln(True), Num(3), e0)
579             >>> p = AsmModule.Program({}, [])
580             >>> g = GenVisitor()
581             >>> v = e1.accept(g, p)
582             >>> p.eval()
583             >>> p.get_val(v)
584             3
585
586             >>> e0 = Div(Num(2), Num(0))
587             >>> e1 = IfThenElse(Bln(False), e0, Num(3))
588             >>> p = AsmModule.Program({}, [])
589             >>> g = GenVisitor()
590             >>> v = e1.accept(g, p)
591             >>> p.eval()
592             >>> p.get_val(v)
593             3
594
595             # TODO: Implement this method.
596             raise NotImplementedError
597
598     def visit_fn(self, exp, prog):
599         # Here goes some hints. Feel free to disregard them if you think about
600         # any other way to implement functions. Indeed, there are many ways to
601         # do just that:
602         #
603         # 1. Allocate space on the stack:
604         # 2. Get the param value. Assumed to be on a0:
605         # 3. Generate a body for the function and save the return value in a0:
606         # 4. Restore the stack, and get back the return address:
607         # 5. Jump back to the caller:
608         #
609         # TODO: Implement this method.
610         raise NotImplementedError
611
612     def visit_app(self, exp, prog):
613         # Here goes some more hints. Again, take them if you feel like it:
614         #
615         # 1. Generate the instructions to find out the target of the call:
616         # 2. Generate code to compute the parameter of the call:
617         # 3. Jump to the function. Remember that Jarl saves the current address
618         # into ra:
619         # 4. Get the return value of the function. It's meant to be on a0:

```

```

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

```

```

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

```

826

~~raise NotImplementedError~~

827

~~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.__env["x0"] = 0
46         self.__env["sp"] = memory_size
47
48     def get_inst(self):
49         if self.pc >= 0 and self.pc < len(self.__insts):
50             inst = self.__insts[self.pc]
51             self.pc += 1
52             return inst
53         else:
54             return None
55
56     def get_number_of_instructions(self):
57         return len(self.__insts)
58
59     def add_inst(self, inst):
60         self.__insts.append(inst)
61
62     def get_pc(self):
63         return self.pc
64
65     def set_pc(self, pc):
66         self.pc = pc
67
68     def set_val(self, name, value):
69         if name != "x0": # Can't change x0, which is always zero.
70             self.__env[name] = value
71
72     def set_mem(self, addr, value):
73         self.__mem[addr] = value
74
75     def get_mem(self, addr):
76         return self.__mem[addr]
77
78     def get_val(self, name):
79         """
80             The register x0 always contains the value zero:
81
82             >>> p = Program(0, {}, [])
83             >>> p.get_val("x0")
84             0
85             """
86         if name in self.__env:
87             return self.__env[name]
88         else:
89             sys.exit(f"Undefined register: {name}")
90
91     def print_env(self):
92         for name, val in sorted(self.__env.items()):
93             print(f"{name}: {val}")
94
95     def print_insts(self):
96         counter = 0
97         for inst in self.__insts:
98             print("%03d: %s" % (counter, str(inst)))
99             counter += 1
100            print("%03d: %s" % (counter, "END"))
101
102     def eval(self):
103         """

```

```

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

```

```

201     class BranchOp(inst):
202         """
203         The general class of branching instructions. These instructions can change
204         the control flow of a program. Normally, the next instruction is given by
205         pc + 1. A branch might change pc to point out to a different label..
206         """
207
208     def set_target(self, lab):
209         assert isinstance(lab, int)
210         self.lab = lab
211
212
213
214     class Beq(BranchOp):
215         """
216         beq rs1, rs2, lab:
217         Jumps to label lab if the value in rs1 is equal to the value in rs2.
218         """
219
220         def __init__(self, rs1, rs2, lab=None):
221             assert isinstance(rs1, str) and isinstance(rs2, str)
222             self.rs1 = rs1
223             self.rs2 = rs2
224             if lab != None:
225                 assert isinstance(lab, int)
226                 self.lab = lab
227
228         def get_opcode(self):
229             return "beq"
230
231         def __str__(self):
232             op = self.get_opcode()
233             return f"{op} {self.rs1} {self.rs2} {self.lab}"
234
235         def eval(self, prog):
236             if prog.get_val(self.rs1) == prog.get_val(self.rs2):
237                 prog.set_pc(self.lab)
238
239
240     class Jal(BranchOp):
241         """
242         jal rd lab:
243         Stores the return address (PC+1) on register rd, then jumps to label lab.
244         If rd is x0, then it does not write on the register. In this case, notice
245         that `jal x0 lab` is equivalent to an unconditional jump to `lab`.
246
247         Example:
248             >>> i = Jal("a", 20)
249             >>> str(i)
250             'jal a 20'
251
252             >>> p = Program(10, env={}, insts=[Jal("a", 20)])
253             >>> p.eval()
254             >>> p.get_pc(), p.get_val("a")
255             (20, 2)
256
257             >>> p = Program(10, env={}, insts=[Jal("x0", 20)])
258             >>> p.eval()
259             >>> p.get_pc(), p.get_val("x0")
260             (20, 0)
261
262         """
263
264         def __init__(self, rd, lab=None):
265             assert isinstance(rd, str)
266             self.rd = rd
267             if lab != None:
268                 assert isinstance(lab, int)
269                 self.lab = lab
270
271         def get_opcode(self):
272             return "jal"
273
274         def __str__(self):
275             op = self.get_opcode()
276             return f"{op} {self.rd} {self.lab}"
277
278         def eval(self, prog):
279             if self.rd != "x0":
280                 # Notice that Jal and Jalr set pc to pc + 1. However, when we fetch
281                 # an instruction, we already increment the PC. Therefore, by using
282                 # get_pc, we are indeed, reading pc + 1.
283                 prog.set_val(self.rd, prog.get_pc())
284                 prog.set_pc(self.lab)
285
286
287     class Jalr(BranchOp):
288         """
289         jalr rd, rs, offset
290         The jalr rd, rs, offset instruction performs an indirect jump to the
291         address computed by adding the value in rs to the immediate offset, and
292         stores the address of the instruction following the jump into rd.
293
294         Example:
295             >>> i = Jalr("a", "b", 20)
296             >>> str(i)
297             'jalr a b 20'
298
299             >>> p = Program(10, env={"b":30}, insts=[Jalr("a", "b", 20)])
300             >>> p.eval()
301             >>> p.get_pc(), p.get_val("a")
302             (50, 2)
303
304             >>> p = Program(10, env={"b":30}, insts=[Jalr("x0", "b", 20)])
305             >>> p.eval()
306             >>> p.get_pc(), p.get_val("b")
307             (50, 2)
308
309             >>> p = Program(10, env={"b":30}, insts=[Jalr("x0", "b", 20)])
310             >>> p.eval()
311             >>> p.get_pc(), p.get_val("va")
312

```

```

310     """ p.get_pc(), p.get_val() )
311     (50, 0)
312 """
313
314     def __init__(self, rd, rs, offset=0):
315         assert isinstance(rd, str) and isinstance(rs, str)
316         self.rd = rd
317         self.rs = rs
318         if offset != None:
319             assert isinstance(offset, int)
320             self.offset = offset
321
322     def get_opcode(self):
323         return "jalr"
324
325     def __str__(self):
326         op = self.get_opcode()
327         return f"{op} {self.rd} {self.rs} {self.offset}"
328
329     def eval(self, prog):
330         if self.rd != "x0":
331             prog.set_val(self.rd, prog.get_pc())
332             rs_val = prog.get_val(self.rs)
333             prog.set_pc(rs_val + self.offset)
334
335
336 class MemOp(Inst):
337     """
338         The general class of instructions that access memory. These instructions
339         include loads and stores.
340     """
341
342     def __init__(self, rs1, offset, reg):
343         assert isinstance(rs1, str) and isinstance(reg, str) and isinstance(offset, int)
344         self.rs1 = rs1
345         self.offset = offset
346         self.reg = reg
347
348     def __str__(self):
349         op = self.get_opcode()
350         return f"{op} {self.reg}, {self.offset}({self.rs1})"
351
352
353 class Sw(MemOp):
354     """
355         sw reg, offset(rs1)
356         *(rs1 + offset) = reg
357
358         * reg: The source register containing the data to be stored.
359         * rs1: The base register containing the memory address.
360         * offset: A 12-bit signed immediate that is added to rs1 to form the
361             effective address.
362
363         Example:
364             >>> i = Sw("a", 0, "b")
365             >>> str(i)
366             'sw b, 0(a)'
367
368             >>> p = Program(10, env={"b":2, "a":3}, insts=[Sw("a", 0, "b")])
369             >>> p.eval()
370             >>> p.get_mem(3)
371             2
372     """
373
374     def eval(self, prog):
375         val = prog.get_val(self.reg)
376         addr = prog.get_val(self.rs1) + self.offset
377         prog.set_mem(addr, val)
378
379     def get_opcode(self):
380         return "sw"
381
382
383 class Lw(MemOp):
384     """
385         lw reg, offset(rs1)
386         reg = *(rs1 + offset)
387
388         * reg: The destination register that will be overwritten.
389         * rs1: The base register containing the memory address.
390         * offset: A 12-bit signed immediate that is added to rs1 to form the
391             effective address.
392
393         Example:
394             >>> i = Lw("a", 0, "b")
395             >>> str(i)
396             'lw b, 0(a)'
397
398             >>> p = Program(10, env={"a":2}, insts=[Lw("a", 0, "b")])
399             >>> p.eval()
400             >>> p.get_val("b")
401             0
402
403             >>> insts = [Sw("a", 0, "b"), Lw("a", 0, "c")]
404             >>> p = Program(10, env={"a":2, "b":5}, insts=insts)
405             >>> p.eval()
406             >>> p.get_val("c")
407             5
408     """
409
410     def eval(self, prog):
411         addr = prog.get_val(self.rs1) + self.offset
412         val = prog.get_mem(addr)
413         prog.set_val(self.reg, val)

```

```

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

```

```

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

```

```

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

```