

## 2025\_2 - COMPILADORES - METATURMA

[PAINEL](#) > [MINHAS TURMAS](#) > [2025\\_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)  
 > [AV6 - VERIFICAÇÃO DINÂMICA DE TIPOS](#)

[Descrição](#)

[Visualizar envios](#)

### AV6 - Verificação dinâmica de tipos

**Data de entrega:** quarta, 1 Out 2025, 23:59

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

**Tipo de trabalho:** Trabalho individual

A linguagem que estamos criando, por enquanto, é [dinamicamente tipada](#). Exemplos de linguagens dinamicamente tipadas incluem Python, Ruby, PHP e JavaScript. Os tipos de expressões em uma linguagem dinamicamente tipada são descobertos em tempo de execução. Mas isso não quer dizer que os tipos não existam! Ainda deveria ser um erro tentar executar expressões como `not 3` ou `True < 4`.

O objetivo deste trabalho é adicionar verificação de tipos ao nosso interpretador de expressões aritméticas. A verificação de tipos vai evitar que tentemos executar operações como `"True + 3"`, ou `"not 3"`. Em vez de executar essas operações, nosso interpretador irá produzir um erro de tipos (imprimindo a mensagem `"Type error"`). Assim, para resolver esse VPL, você deverá efetuar duas alterações na implementação do VPL anterior (*Visitors*), a saber:

1. Modificar a gramática da linguagem, adicionando três novas construções: `if-then-else`, `or` e `and`.
2. Adicionar verificação de tipos à implementação de `EvalVisitor`.

#### Alterações na Gramática

A primeira parte deste exercício consiste em alterar a gramática da linguagem de expressões aritméticas com variáveis. Serão adicionadas três novas construções à linguagem:

1. Expressões condicionais, do tipo `if cond then e0 else e1`. Esta expressão avalia a condição `cond`, e caso o valor obtido seja `true`, o valor de `e0` é retornado, senão o valor de `e1` é retornado.
2. Conjunções lógicas, do tipo `e0 or e1`. Esta expressão tem o valor `true` se a avaliação de `e0` ou a avaliação de `e1` produzir o valor `true`. Doutro modo, a expressão `e0 or e1` possui o valor `false`.
3. Disjunções lógicas, do tipo `e0 and e1`. Esta expressão tem o valor `true` se a avaliação de `e0` e a avaliação de `e1` produzirem o valor `true`. Doutro modo, a expressão `e0 and e1` possui o valor `false`.

Estas três expressões possuem a semântica de [curto-circuito](#). A semântica de curto-circuito é uma otimização de código, que consiste em avaliar somente o mínimo de uma expressão lógica para obter o seu valor. Por exemplo, a expressão `if true then e0 else e1` somente avalia a expressão `e0`. De maneira similar, a expressão `if false then e0 else e1` somente avalia a expressão `e1`. Veja o que ocorreria em nosso interpretador:

```
~/Programs/DCC053/6_IfThenElse % python3 driver.py
true or 1 div 0 = 3 # Aperte CTRL+D
True
~/Programs/DCC053/6_IfThenElse % python3 driver.py
1 div 0 = 3
1D # divisao por zero!
~/Programs/DCC053/6_IfThenElse % python3 driver.py
1 div 0 = 3 or True
1D
```

A semântica de curto-circuito também ocorre nas conjunções e disjunções, conforme ilustra a Figura 1.

|   |
|---|
| Somente avalia as expressões e0 e e1:                                     |
| env $\vdash$ e0 $\Rightarrow$ true      env $\vdash$ e1 $\Rightarrow$ val |
| env $\vdash$ if e0 then e1 else e2 $\Rightarrow$ val                      |

|  |
|--|
| Somente avalia a expressão e0:           |
| env $\vdash$ e0 $\Rightarrow$ true       |
| env $\vdash$ e0 or e1 $\Rightarrow$ true |

|  |
|--|
| Somente avalia a expressão e0:             |
| env $\vdash$ e0 $\Rightarrow$ false        |
| env $\vdash$ e0 and e1 $\Rightarrow$ false |

Somente avalia as expressões e0 e e2:

|  |                                   |
|--|-----------------------------------|
| env $\vdash$ e0 $\Rightarrow$ false                  | env $\vdash$ e2 $\Rightarrow$ val |
| env $\vdash$ if e0 then e1 else e2 $\Rightarrow$ val |                                   |

|                                     |
|-------------------------------------|
| Avalia ambas as expressões:         |
| env $\vdash$ e0 $\Rightarrow$ false |
| env $\vdash$ e1 $\Rightarrow$ val   |

|                                    |
|------------------------------------|
| Avalia ambas as expressões:        |
| env $\vdash$ e0 $\Rightarrow$ true |
| env $\vdash$ e1 $\Rightarrow$ val  |

O erro de divisão por zero não ocorre:

{ }  $\vdash$  if true then 1 else 2 div 0  $\Rightarrow$  1

O erro de divisão por zero não ocorre:

{ }  $\vdash$  2 < 3 or 3 div 0 < 5  $\Rightarrow$  true

O erro de divisão por zero não ocorre:

{ }  $\vdash$  2 > 3 and 3 div 0 < 5  $\Rightarrow$  false

Figura 1: A semântica de curto-circuito, com exemplos de avaliação.

Figura 2: Regras de precedência

Além de adicionar as três novas construções gramaticais, iremos alterar as regras de precedência da gramática. As novas regras de precedência devem seguir a tabela de prioridades vista na Figura 2. Assim,  $2 < 3$  and  $3 < 4$  é equivalente a  $(2 < 3)$  and  $(3 < 4)$ , e not if true then true else false é uma expressão gramaticalmente incorreta, enquanto not (if true then true else false) possui o valor false.

## Verificação de Tipos em Tempo de Execução

Até o momento, é possível que nosso interpretador execute programas "estranhos", como `true + 1`. Caso isso aconteça, teremos uma exceção de Python (algo como "unsupported operand type(s) for +"), ou teremos o valor 2 (pois em Python, `true + 1 == 2`). A fim de evitar esse tipo de comportamento, iremos adicionar ao nosso interpretador três tipos de mensagens de erro:

1. Type error: erro que ocorre quando operações são aplicadas sobre valores de tipo inválido.
2. Def error: erro que ocorre quando tentamos acessar uma variável que não foi declarada.
3. Parse error: erro causado por sentenças que não são gramaticalmente corretas.

Para emitir uma mensagem de erro, use o comando `sys.exit(msg)`, disponível no pacote `sys` sendo `msg` a mensagem de erro. Os erros do tipo (1) e (2) acima devem ser implementados em `EvalVisitor`, no arquivo `Visitor.py`. O erro de parsing deve ser implementado em `Parser.py`. Algumas regras de tipagem podem ser vistas nas figuras 3 e 4 abaixo.

|  |                                   |
|--|-----------------------------------|
| env $\vdash$ e0 $\Rightarrow$ true                   | env $\vdash$ e1 $\Rightarrow$ val |
| env $\vdash$ if e0 then e1 else e2 $\Rightarrow$ val |                                   |

|  |                       |
|--|-----------------------|
| env $\vdash$ e0 $\Rightarrow$ val              | type(val) $\neq$ bool |
| env $\vdash$ e0 or e1 $\Rightarrow$ type error |                       |

|   |                                  |                     |
|---|----------------------------------|---------------------|
| env $\vdash$ e0 $\Rightarrow$ v0              | env $\vdash$ e1 $\Rightarrow$ v1 | type(v0) $\neq$ int |
| env $\vdash$ e0 + e1 $\Rightarrow$ type error |                                  |                     |

|  |                                   |
|--|-----------------------------------|
| env $\vdash$ e0 $\Rightarrow$ false                  | env $\vdash$ e2 $\Rightarrow$ val |
| env $\vdash$ if e0 then e1 else e2 $\Rightarrow$ val |                                   |

|  |                                   |                       |
|--|-----------------------------------|-----------------------|
| env $\vdash$ e0 $\Rightarrow$ false            | env $\vdash$ e1 $\Rightarrow$ val | type(val) $\neq$ bool |
| env $\vdash$ e0 or e1 $\Rightarrow$ type error |                                   |                       |

|   |                                  |                     |
|---|----------------------------------|---------------------|
| env $\vdash$ e0 $\Rightarrow$ v0              | env $\vdash$ e1 $\Rightarrow$ v1 | type(v1) $\neq$ int |
| env $\vdash$ e0 + e1 $\Rightarrow$ type error |                                  |                     |

Figura 3: Algumas regras de tipagem envolvendo a semântica de curto-circuito

Figura 4: Algumas regras de tipagem envolvendo a semântica normal

Note que para verificar tipos, você pode usar a própria linguagem Python, com chamadas às funções `type(v)`, que retorna o tipo do valor `v`, e `isinstance(v, t)`, que retorna `True` se `v` for uma instância do tipo `t`. Mas tome cuidado, pois Python possui um sistema de coerção bastante abrangente:

```
>>> type(True) == type(1)
False
>>> isinstance(True, int)
True
```

Como exemplo, uma verificação de tipos pode ser vista logo abaixo, para expressões de comparação LEQ:

```
def visit_leq(self, exp, env):
    val_left = exp.left.accept(self, env)
    val_right = exp.right.accept(self, env)
    if type(val_left) == type(1) and type(val_right) == type(1):
        return val_left <= val_right
    else:
        sys.exit("Type error")
```

Abaixo vemos alguns exemplos de diferentes erros de tipos:

```
~/Programs/DCC053/6_IfThenElse % python3 driver.py
not 1
Type error
```

| Nível de precedência | Operador lógico/aritmético |
|----------------------|----------------------------|
| 1                    | <code>~ () not let</code>  |
| 2                    | <code>* /</code>           |
| 3                    | <code>+ -</code>           |
| 4                    | <code>&lt;= &lt;</code>    |
| 5                    | <code>=</code>             |
| 6                    | <code>and</code>           |
| 7                    | <code>or</code>            |
| 8                    | <code>if-then-else</code>  |

```
~/Programs/DCC053/6_IfThenElse % python3 driver.py
not 1 < 2
Type error
~/Programs/DCC053/6_IfThenElse % python3 driver.py
if 1 then 3 else 4
Type error
```

Abaixo vemos alguns exemplos de erros de *parsing*:

```
~/Programs/DCC053/6_IfThenElse % python3 driver.py
1 + 2 +
Parse error
~/Programs/DCC053/6_IfThenElse % python3 driver.py
not if true then 1 else 2
Parse error
```

E finalmente abaixo vemos alguns exemplos de erros de variáveis indefinidas.

```
~/Programs/DCC053/6_IfThenElse % python3 driver.py
let x <- 2 in x * x end + x
Def error
~/Programs/DCC053/6_IfThenElse % python3 driver.py
x * 2
Def error
```

## Submetendo e Testando

Para completar este VPL, você deverá entregar cinco arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Visitor.py` e `driver.py`. Você não deverá alterar `driver.py`; tampouco irá precisar alterar `Expression.py`. Você pode usar, como ponto de partida, os arquivos disponíveis no exercício anterior (`Visitors`). Para testar sua implementação localmente, você pode usar o comando abaixo:

```
$> python3 driver.py
2 + let v <- 3 in v * v end # Aperte CTRL+D
Value is 11
```

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 `Expression.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
7 if __name__ == "__main__":
8     """
9         Este arquivo não deve ser alterado, mas deve ser enviado para resolver o
10        VPL. O arquivo contém o código que testa a implementação do parser.
11        """
12    text = sys.stdin.read()
13    lexer = Lexer(text)
14    parser = Parser(lexer.tokens())
15    exp = parser.parse()
16    visitor = EvalVisitor()
17    print(f'{exp.accept(visitor, {})}')
```

`Lexer.py`

```

1 import sys
2 import enum
3
4
5 class Token:
6     """
7         This class contains the definition of Tokens. A token has two fields: its
8         text and its kind. The "kind" of a token is a constant that identifies it
9         uniquely. See the TokenType to know the possible identifiers (if you want).
10        You don't need to change this class.
11    """
12    def __init__(self, tokenText, tokenKind):
13        # The token's actual text. Used for identifiers, strings, and numbers.
14        self.text = tokenText
15        # The TokenType that this token is classified as.
16        self.kind = tokenKind
17
18
19 class TokenType(enum.Enum):
20     """
21         These are the possible tokens. You don't need to change this class at all.
22     """
23     EOF = -1 # End of file
24     NLN = 0 # New line
25     WSP = 1 # White Space
26     COM = 2 # Comment
27     NUM = 3 # Number (integers)
28     STR = 4 # Strings
29     TRU = 5 # The constant true
30     FLS = 6 # The constant false
31     VAR = 7 # An identifier
32     LET = 8 # The 'let' of the let expression
33     INX = 9 # The 'in' of the let expression
34     END = 10 # The 'end' of the let expression
35     EQL = 201 # x = y
36     ADD = 202 # x + y
37     SUB = 203 # x - y
38     MUL = 204 # x * y
39     DIV = 205 # x / y
40     LEQ = 206 # x <= y
41     LTH = 207 # x < y
42     NEG = 208 # ~x
43     NOT = 209 # not x
44     LPR = 210 # (
45     RPR = 211 # )
46     ASN = 212 # The assignment '<->' operator
47     ORX = 213 # x or y
48     AND = 214 # x and y
49     IFX = 215 # The 'if' of a conditional expression
50     THN = 216 # The 'then' of a conditional expression
51     ELS = 217 # The 'else' of a conditional expression
52
53
54 class Lexer:
55
56     def __init__(self, source):
57         """
58             The constructor of the lexer. It receives the string that shall be
59             scanned.
60             TODO: You will need to implement this method.
61         """
62         pass
63
64     def tokens(self):
65         """
66             This method is a token generator: it converts the string encapsulated
67             into this object into a sequence of Tokens. Notice that this method
68             filters out three kinds of tokens: white-spaces, comments and new lines.
69
70             Examples:
71
72             >>> l = Lexer("1 + 3")
73             >>> [tk.kind for tk in l.tokens()]
74             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
75
76             >>> l = Lexer('1 * 2\n')
77             >>> [tk.kind for tk in l.tokens()]
78             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
79
80             >>> l = Lexer('1 * 2 -- 3\n')
81             >>> [tk.kind for tk in l.tokens()]
82             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
83
84             >>> l = Lexer("1 + var")
85             >>> [tk.kind for tk in l.tokens()]
86             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
87
88             >>> l = Lexer("let v <- 2 in v end")
89             >>> [tk.kind.name for tk in l.tokens()]
90             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
91
92             token = self.getToken()
93             while token.kind != TokenType.EOF:
94                 if token.kind != TokenType.WSP and token.kind != TokenType.COM \
95                     and token.kind != TokenType.NLN:
96                     yield token
97                 token = self.getToken()
98
99     def getToken(self):
100         """
101             Return the next token.
102             TODO: Implement this method!
103         """

```

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

```

```

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

```

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

Expression.py

```

1 """
2 This file implements the data structure that represents the logic and
3 arithmetic expressions in our language.
4
5 IMPORTANT: There is no need to change this file to solve this VPL!
6 """
7 from abc import ABC, abstractmethod
8 from Visitor import *
9
10 class Expression(ABC):
11     @abstractmethod
12     def accept(self, visitor, arg):
13         raise NotImplementedError
14
15 class Var(Expression):
16     """
17     This class represents expressions that are identifiers. The value of an
18     identifier is the value associated with it in the environment table.
19     """
20     def __init__(self, identifier):
21         self.identifier = identifier
22     def accept(self, visitor, arg):
23         """
24             Example:
25             >>> e = Var('var')
26             >>> ev = EvalVisitor()
27             >>> e.accept(ev, {'var': 42})
28             42
29
30             >>> e = Var('v42')
31             >>> ev = EvalVisitor()
32             >>> e.accept(ev, {'v42': True, 'v31': 5})
33             True
34         """
35         return visitor.visit_var(self, arg)
36
37 class Bln(Expression):
38     """
39     This class represents expressions that are boolean values. There are only
40     two boolean values: true and false. The acceptuation of such an expression is
41     the boolean itself.
42     """
43     def __init__(self, bln):
44         self.bln = bln
45     def accept(self, visitor, arg):
46         """
47             Example:
48             >>> e = Bln(True)
49             >>> ev = EvalVisitor()
50             >>> e.accept(ev, None)
51             True
52         """
53         return visitor.visit_bln(self, arg)
54
55 class Num(Expression):
56     """
57     This class represents expressions that are numbers. The acceptuation of such
58     an expression is the number itself.
59     """
60     def __init__(self, num):
61         self.num = num
62     def accept(self, visitor, arg):
63         """
64             Example:
65             >>> e = Num(3)
66             >>> ev = EvalVisitor()
67             >>> e.accept(ev, None)
68             3
69         """
70         return visitor.visit_num(self, arg)
71
72 class BinaryExpression(Expression):
73     """
74     This class represents binary expressions. A binary expression has two
75     sub-expressions: the left operand and the right operand.
76     """
77     def __init__(self, left, right):
78         self.left = left
79         self.right = right
80
81     @abstractmethod
82     def accept(self, visitor, arg):
83         raise NotImplementedError
84
85 class Eql(BinaryExpression):
86     """
87     This class represents the equality between two expressions. The acceptuation
88     of such an expression is True if the subexpressions are the same, or false
89     otherwise.
90     """
91     def accept(self, visitor, arg):
92         """
93             Example:
94             >>> n1 = Num(3)
95             >>> n2 = Num(4)
96             >>> e = Eql(n1, n2)
97             >>> ev = EvalVisitor()
98             >>> e.accept(ev, None)
99             False
100
101            >>> n1 = Num(3)
102            >>> n2 = Num(3)
103            >>> e = Eql(n1, n2)

```

```

104     >>> ev = EvalVisitor()
105     >>> e.accept(ev, None)
106     True
107     """
108     return visitor.visit_eql(self, arg)
109
110 class Add(BinaryExpression):
111     """
112     This class represents addition of two expressions. The acceptuation of such
113     an expression is the addition of the two subexpression's values.
114     """
115     def accept(self, visitor, arg):
116         """
117         Example:
118         >>> n1 = Num(3)
119         >>> n2 = Num(4)
120         >>> e = Add(n1, n2)
121         >>> ev = EvalVisitor()
122         >>> e.accept(ev, None)
123         7
124         """
125         return visitor.visit_add(self, arg)
126
127 class And(BinaryExpression):
128     """
129     This class represents the logical disjunction of two boolean expressions.
130     The evaluation of an expression of this kind is the logical AND of the two
131     subexpression's values.
132     """
133     def accept(self, visitor, arg):
134         """
135         Example:
136         >>> b1 = Bln(True)
137         >>> b2 = Bln(False)
138         >>> e = And(b1, b2)
139         >>> ev = EvalVisitor()
140         >>> e.accept(ev, None)
141         False
142
143         >>> b1 = Bln(True)
144         >>> b2 = Bln(True)
145         >>> e = And(b1, b2)
146         >>> ev = EvalVisitor()
147         >>> e.accept(ev, None)
148         True
149         """
150         return visitor.visit_and(self, arg)
151
152 class Or(BinaryExpression):
153     """
154     This class represents the logical conjunction of two boolean expressions.
155     The evaluation of an expression of this kind is the logical OR of the two
156     subexpression's values.
157     """
158     def accept(self, visitor, arg):
159         """
160         Example:
161         >>> b1 = Bln(True)
162         >>> b2 = Bln(False)
163         >>> e = Or(b1, b2)
164         >>> ev = EvalVisitor()
165         >>> e.accept(ev, None)
166         True
167
168         >>> b1 = Bln(False)
169         >>> b2 = Bln(False)
170         >>> e = Or(b1, b2)
171         >>> ev = EvalVisitor()
172         >>> e.accept(ev, None)
173         False
174         """
175         return visitor.visit_or(self, arg)
176
177 class Sub(BinaryExpression):
178     """
179     This class represents subtraction of two expressions. The acceptuation of such
180     an expression is the subtraction of the two subexpression's values.
181     """
182     def accept(self, visitor, arg):
183         """
184         Example:
185         >>> n1 = Num(3)
186         >>> n2 = Num(4)
187         >>> e = Sub(n1, n2)
188         >>> ev = EvalVisitor()
189         >>> e.accept(ev, None)
190         -1
191         """
192         return visitor.visit_sub(self, arg)
193
194 class Mul(BinaryExpression):
195     """
196     This class represents multiplication of two expressions. The acceptuation of
197     such an expression is the product of the two subexpression's values.
198     """
199     def accept(self, visitor, arg):
200         """
201         Example:
202         >>> n1 = Num(3)
203         >>> n2 = Num(4)
204         >>> e = Mul(n1, n2)
205         >>> ev = EvalVisitor()
206         >>> e.accept(ev, None)
207         12

```

```

20/
208     12
209     """
210     return visitor.visit_mul(self, arg)
211
211 class Div(BinaryExpression):
212     """
213     This class represents the integer division of two expressions. The
214     acceptuation of such an expression is the integer quotient of the two
215     subexpression's values.
216     """
217     def accept(self, visitor, arg):
218         """
219         Example:
220         >>> n1 = Num(28)
221         >>> n2 = Num(4)
222         >>> e = Div(n1, n2)
223         >>> ev = EvalVisitor()
224         >>> e.accept(ev, None)
225         7
226
226         >>> n1 = Num(22)
227         >>> n2 = Num(4)
228         >>> e = Div(n1, n2)
229         >>> ev = EvalVisitor()
230         >>> e.accept(ev, None)
231         5
232         """
233
233     return visitor.visit_div(self, arg)
234
235 class Leq(BinaryExpression):
236     """
237     This class represents comparison of two expressions using the
238     less-than-or-equal comparator. The acceptuation of such an expression is a
239     boolean value that is true if the left operand is less than or equal the
240     right operand. It is false otherwise.
241     """
242     def accept(self, visitor, arg):
243         """
244         Example:
245         >>> n1 = Num(3)
246         >>> n2 = Num(4)
247         >>> e = Leq(n1, n2)
248         >>> ev = EvalVisitor()
249         >>> e.accept(ev, None)
250         True
251
252         >>> n1 = Num(3)
253         >>> n2 = Num(3)
254         >>> e = Leq(n1, n2)
255         >>> ev = EvalVisitor()
256         >>> e.accept(ev, None)
257         True
258
259         >>> n1 = Num(4)
260         >>> n2 = Num(3)
261         >>> e = Leq(n1, n2)
262         >>> ev = EvalVisitor()
263         >>> e.accept(ev, None)
264         False
265         """
265
266     return visitor.visit_leq(self, arg)
266
267 class Lth(BinaryExpression):
268     """
269     This class represents comparison of two expressions using the
270     less-than comparison operator. The acceptuation of such an expression is a
271     boolean value that is true if the left operand is less than the right
272     operand. It is false otherwise.
273     """
273     def accept(self, visitor, arg):
274         """
275         Example:
276         >>> n1 = Num(3)
277         >>> n2 = Num(4)
278         >>> e = Lth(n1, n2)
279         >>> ev = EvalVisitor()
280         >>> e.accept(ev, None)
281         True
282
283         >>> n1 = Num(3)
284         >>> n2 = Num(3)
285         >>> e = Lth(n1, n2)
286         >>> ev = EvalVisitor()
287         >>> e.accept(ev, None)
288         False
289
290         >>> n1 = Num(4)
291         >>> n2 = Num(3)
292         >>> e = Lth(n1, n2)
293         >>> ev = EvalVisitor()
294         >>> e.accept(ev, None)
295         False
295         """
295
295     return visitor.visit_lth(self, arg)
296
297 class UnaryExpression(Expression):
298     """
299     This class represents unary expressions. A unary expression has only one
300     sub-expression.
301     """
302     def __init__(self, exp):
303         self.exp = exp
304
305     @abstractmethod
306     def accept(self, visitor, arg):
307         raise NotImplementedError
308
309     class Neg(UnaryExpression):
310         """

```

```

311 This expression represents the additive inverse of a number. The additive
312 inverse of a number n is the number -n, so that the sum of both is zero.
313 """
314 def accept(self, visitor, arg):
315     """
316     Example:
317     >>> n = Num(3)
318     >>> e = Neg(n)
319     >>> ev = EvalVisitor()
320     >>> e.accept(ev, None)
321     -3
322     >>> n = Num(0)
323     >>> e = Neg(n)
324     >>> ev = EvalVisitor()
325     >>> e.accept(ev, None)
326     0
327     """
328     return visitor.visit_neg(self, arg)
329
330 class Not(UnaryExpression):
331     """
332     This expression represents the negation of a boolean. The negation of a
333     boolean expression is the logical complement of that expression.
334     """
335     def accept(self, visitor, arg):
336         """
337         Example:
338         >>> t = Bln(True)
339         >>> e = Not(t)
340         >>> ev = EvalVisitor()
341         >>> e.accept(ev, None)
342         False
343         >>> t = Bln(False)
344         >>> e = Not(t)
345         >>> ev = EvalVisitor()
346         >>> e.accept(ev, None)
347         True
348         """
349         return visitor.visit_not(self, arg)
350
351 class Let(Expression):
352     """
353     This class represents a let expression. The semantics of a let expression,
354     such as "let v <- e0 in e1" on an environment env is as follows:
355     1. Evaluate e0 in the environment env, yielding e0_val
356     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
357     """
358     def __init__(self, identifier, exp_def, exp_body):
359         self.identifier = identifier
360         self.exp_def = exp_def
361         self.exp_body = exp_body
362     def accept(self, visitor, arg):
363         """
364         Example:
365         >>> e = Let('v', Num(42), Var('v'))
366         >>> ev = EvalVisitor()
367         >>> e.accept(ev, {})
368         42
369
370         >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
371         >>> ev = EvalVisitor()
372         >>> e.accept(ev, {})
373         42
374
375         >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
376         >>> ev = EvalVisitor()
377         >>> e.accept(ev, {})
378         1764
379         """
380         return visitor.visit_let(self, arg)
381
382 class IfThenElse(Expression):
383     """
384     This class represents a conditional expression. The semantics an expression
385     such as 'if B then E0 else E1' is as follows:
386     1. Evaluate B. Call the result ValueB.
387     2. If ValueB is True, then evaluate E0 and return the result.
388     3. If ValueB is False, then evaluate E1 and return the result.
389     Notice that we only evaluate one of the two sub-expressions, not both. Thus,
390     "if True then 0 else 1 div 0" will return 0 indeed.
391     """
392     def __init__(self, cond, e0, e1):
393         self.cond = cond
394         self.e0 = e0
395         self.e1 = e1
396     def accept(self, visitor, arg):
397         """
398         Example:
399         >>> e = IfThenElse(Bln(True), Num(42), Num(30))
400         >>> ev = EvalVisitor()
401         >>> e.accept(ev, {})
402         42
403
404         >>> e = IfThenElse(Bln(False), Num(42), Num(30))
405         >>> ev = EvalVisitor()
406         >>> e.accept(ev, {})
407         30
408         """
409         return visitor.visit_ifThenElse(self, arg)

```

```

1 import sys
2 from abc import ABC, abstractmethod
3 from Expression import *
4
5 class Visitor(ABC):
6     """
7         The visitor pattern consists of two abstract classes: the Expression and the
8         Visitor. The Expression class defines one method: 'accept(visitor, args)'.
9         This method takes in an implementation of a visitor, and the arguments that
10        are passed from expression to expression. The Visitor class defines one
11        specific method for each subclass of Expression. Each instance of such a
12        subclass will invoke the right visiting method.
13    """
14    @abstractmethod
15    def visit_var(self, exp, arg):
16        pass
17
18    @abstractmethod
19    def visit_bln(self, exp, arg):
20        pass
21
22    @abstractmethod
23    def visit_num(self, exp, arg):
24        pass
25
26    @abstractmethod
27    def visit_eql(self, exp, arg):
28        pass
29
30    @abstractmethod
31    def visit_and(self, exp, arg):
32        pass
33
34    @abstractmethod
35    def visit_or(self, exp, arg):
36        pass
37
38    @abstractmethod
39    def visit_add(self, exp, arg):
40        pass
41
42    @abstractmethod
43    def visit_sub(self, exp, arg):
44        pass
45
46    @abstractmethod
47    def visit_mul(self, exp, arg):
48        pass
49
50    @abstractmethod
51    def visit_div(self, exp, arg):
52        pass
53
54    @abstractmethod
55    def visit_leq(self, exp, arg):
56        pass
57
58    @abstractmethod
59    def visit_lth(self, exp, arg):
60        pass
61
62    @abstractmethod
63    def visit_neg(self, exp, arg):
64        pass
65
66    @abstractmethod
67    def visit_not(self, exp, arg):
68        pass
69
70    @abstractmethod
71    def visit_let(self, exp, arg):
72        pass
73
74    @abstractmethod
75    def visit_ifThenElse(self, exp, arg):
76        pass
77
78 class EvalVisitor(Visitor):
79     """
80         The EvalVisitor class evaluates logical and arithmetic expressions. The
81         result of evaluating an expression is the value of that expression. The
82         inherited attribute propagated throughout visits is the environment that
83         associates the names of variables with values.
84
85         Notice that this implementation must perform type verification. If some
86         verification fail, then it invokes sys.exit with the correct error
87         message. We expect two different messages:
88
89         1. sys.exit("Type error")
90         2. sys.exit("Def error")
91
92         Examples:
93         >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
94         >>> e1 = Not(Eql(e0, Num(1764)))
95         >>> ev = EvalVisitor()
96         >>> e1.accept(ev, {})
97         False
98
99         >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
100        >>> e1 = Lth(e0, Var('x'))
101        >>> ev = EvalVisitor()
102        >>> e1.accept(ev, {'x': 41})
103        True

```

```
104     """
105     def visit_var(self, exp, env):
106         # TODO: Implement this method!
107         raise NotImplementedError
108
109     def visit_bln(self, exp, env):
110         # TODO: Implement this method!
111         raise NotImplementedError
112
113     def visit_num(self, exp, env):
114         # TODO: Implement this method!
115         raise NotImplementedError
116
117     def visit_eql(self, exp, env):
118         # TODO: Implement this method!
119         raise NotImplementedError
120
121     def visit_and(self, exp, env):
122         # TODO: Implement this method!
123         raise NotImplementedError
124
125     def visit_or(self, exp, env):
126         # TODO: Implement this method!
127         raise NotImplementedError
128
129     def visit_add(self, exp, env):
130         # TODO: Implement this method!
131         raise NotImplementedError
132
133     def visit_sub(self, exp, env):
134         # TODO: Implement this method!
135         raise NotImplementedError
136
137     def visit_mul(self, exp, env):
138         # TODO: Implement this method!
139         raise NotImplementedError
140
141     def visit_div(self, exp, env):
142         # TODO: Implement this method!
143         raise NotImplementedError
144
145     def visit_leq(self, exp, env):
146         # TODO: Implement this method!
147         raise NotImplementedError
148
149     def visit_lth(self, exp, env):
150         # TODO: Implement this method!
151         raise NotImplementedError
152
153     def visit_neg(self, exp, env):
154         # TODO: Implement this method!
155         raise NotImplementedError
156
157     def visit_not(self, exp, env):
158         # TODO: Implement this method!
159         raise NotImplementedError
160
161     def visit_let(self, exp, env):
162         # TODO: Implement this method!
163         raise NotImplementedError
164
165     def visit_ifThenElse(self, exp, env):
166         # TODO: Implement this method!
167         raise NotImplementedError
```