

2025_2 - COMPILADORES - METATURMA

[PAÍNEL](#) > [MINHAS TURMAS](#) > [2025_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)
 > [AV8 - FUNÇÕES ANÔNIMAS](#)

Descrição

Visualizar envios

AV8 - Funções anônimas

Data de entrega: quarta, 15 Out 2025, 23:59

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

Tipo de trabalho: Trabalho individual

O objetivo deste trabalho é adicionar funções anônimas à nossa linguagem de programação. Uma [função anônima](#) é uma definição de função que não está associada a qualquer nome de variável. Esta modificação da linguagem vai aumentar sua expressividade de maneira notável. Em particular, passamos a ter acesso a "[closures](#)", por exemplo. As figuras abaixo ilustram alguns dos programas que podem ser escritos usando funções anônimas:

Figura 1: uma função é um valor

```
(fn x => x + 1)
Fn(x)
```

Figure 2: a sintaxe para aplicação de função é justaposição

```
(fn x => x + 1) 2
3
```

Figure 3: o corpo de uma função se estende o máximo possível para a direita. Neste caso, 'x 1' não pode ser tratado como uma aplicação de função, pois x é um inteiro. Para corrigir o programa da esquerda, seria necessário usar parênteses, como no programa da direita:

```
fn x => x * x * x * 1
Type error
```

```
(fn x => x * x * x * x) 1
1
```

Figure 4: a aplicação de funções tem a maior precedência possível. O programa da esquerda está errado, pois ele indica que estamos aplicando a função cube sobre o operador de inversão de inteiros: "cube ~". O programa da direita está correto devido ao uso de parênteses:

```
let
  cube <- fn x => x * x * x
in
  cube ~4
end
Parse error
```

```
let
  cube <- fn x => x * x * x
in
  cube (~4)
end
-64
```

Figure 5: a aplicação de funções tem a maior precedência possível. Assim, o programa da esquerda é equivalente a "(cube 4) + 1". Para forçar outra ordem de avaliação, como no programa da direita, parênteses são necessários:

```
let
  cube <- fn x => x * x * x
in
  cube 4 + 1
end
65
```

```
let
  cube <- fn x => x * x * x
in
  cube (4 + 1)
end
125
```

Figure 6: a aplicação de funções é associativa à esquerda; ou seja, "f0 f1 f2" é equivalente a "(f0 f1) f2". Para forçar uma associatividade diferente, parênteses podem ser usados:

```
(fn x => fn y => x (x y)) (fn a => a * a) 4
256

(fn a => a + a) ((fn a => a * a) 4)
32
```

Figure 7: funções podem retornar closures. Neste caso, o programa da esquerda retorna a função "fn y => 1 + y". Um closure pode ser aplicado normalmente, como é visto no programa da direita:

```
let
  add <- fn x => fn y => x + y
in
  add 1
end
Fn(y)
```

```
let
  add <- fn x => fn y => x + y
in
  let inc <- add 1 in inc 3 end
end
4
```

A adição de funções anônimas à linguagem de programação implica em algumas mudanças em nossa implementação. Duas novas categorias sintáticas deverão ser adicionadas à linguagem: funções e aplicações de funções. Essas novas categorias sintáticas levarão à modificações no analisador léxico e sintático, e necessitarão de novas construções no interpretador. Abaixo salientamos as mudanças.

Expressões

Dois novos tipos de expressões devem ser adicionados ao arquivo `Expression.py`. O primeiro tipo, `Fn`, denota funções anônimas. O segundo tipo de expressões, `App`, denota aplicações de funções.

Interpretador

A definição do visitor (Classe `EvalVisitor` em `Visitor.py`) que implementa o interpretador deverá ser alterada, para interpretar funções anônimas e aplicações. As regras semânticas são dadas na Figura 9 e na Figura 10.

Análise Léxica

O analisador léxico (`Lexer.py`) deverá ser alterado para incluir os tokens que participam da declaração de funções: a palavra reservada `fn` e a seta (`=>`). Neste caso, o método `getToken` será alterado para retornar esses novos tokens, quando eles forem encontrados. Veja que uma função como `fn a => a + 1` vai retornar seis tokens: FNX, VAR, ARW, VAR, ADD e NUM.

Análise Sintática

O analisador sintático (`Parser.py`) deverá ser alterado para incluir declarações de funções e aplicações de funções. A declaração de funções terá a menor precedência possível, mas a aplicação de funções terá a maior precedência possível. Assim, uma construção como `fn x => x + 1` deve ser entendida como `(fn x => x + 1)` e uma construção como `f g + 1` deve ser entendida como `((f g) + 1)`. Além disso, a associatividade de aplicação de funções é à esquerda. Assim, a construção `f0 f1 f2` é equivalente a `((f0 f1) f2)`. Finalmente, note que não existe diferença de precedência entre aplicação de funções e operações unárias. Então, uma construção como `f ~1` é um erro sintático, enquanto a construção `f (~1)` é válida. A figura 8 contém uma sugestão para a construção da gramática.

```

fn_exp ::= fn <var> => fn_exp
         | if_exp
if_exp ::= <if> if_exp <then> fn_exp <else> fn_exp
         | or_exp
or_exp ::= and_exp (or and_exp)*
and_exp ::= eq_exp (and eq_exp)*
eq_exp ::= cmp_exp (= cmp_exp)*
cmp_exp ::= add_exp ([<|=|>] add_exp)*
add_exp ::= mul_exp ([+|-] mul_exp)*
mul_exp ::= unary_exp ([*/] unary_exp)*
unary_exp ::= <not> unary_exp
             | ~ unary_exp
             | let_exp
let_exp ::= <let> <var> <- fn_exp <in> fn_exp <end>
           | val_exp
val_exp ::= val_tk (val_tk)*
val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>

Figura 8: Sugestão de gramática com as corretas regras de precedência e
associatividade

```

`eval(env, fn x => Body) → Function(x, Body, env)`

Figura 9: A avaliação de uma função é um valor que denota uma função. Este valor é uma instância da classe `Function` (em `Visitor.py`). Esta classe guarda o parâmetro formal da função, seu corpo, e o ambiente no qual a função foi declarada.

`eval(env, E1) → Function(x, Body, func_env) eval(env, E2) → v1 eval(env U {x:v1}, Body) → v2`
`eval(env, E1 E2) → v2`

Figura 10: A avaliação de uma aplicação (`E1 E2`) ocorre em múltiplos passos. Primeiro, `E1` é avaliado. Caso o resultado não seja uma função, um erro de tipo ocorre. Outro modo, o parâmetro real `E2` é avaliado, produzindo-se um valor `v1`. Então, o corpo `Body` da função é avaliado, no ambiente no qual a função foi declarada, acrescido do binding `x:v1`, onde `x` é o parâmetro formal da função.

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`. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
python3 driver.py
(fn x => x * x) (4-1) # aperte Ctrl+D
9
```

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
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     FNX = 218 # The 'fn' that declares an anonymous function
53     ARW = 219 # The '>=' that separates the parameter from the body of function
54
55
56 class Lexer:
57
58     def __init__(self, source):
59         """
60             The constructor of the lexer. It receives the string that shall be
61             scanned.
62             TODO: You will need to implement this method.
63         """
64         pass
65
66     def tokens(self):
67         """
68             This method is a token generator: it converts the string encapsulated
69             into this object into a sequence of Tokens. Notice that this method
70             filters out three kinds of tokens: white-spaces, comments and new lines.
71
72             Examples:
73
74             >>> l = Lexer("1 + 3")
75             >>> [tk.kind for tk in l.tokens()]
76             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
77
78             >>> l = Lexer('1 * 2\n')
79             >>> [tk.kind for tk in l.tokens()]
80             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
81
82             >>> l = Lexer('1 * 2 -- 3\n')
83             >>> [tk.kind for tk in l.tokens()]
84             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
85
86             >>> l = Lexer("1 + var")
87             >>> [tk.kind for tk in l.tokens()]
88             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
89
90             >>> l = Lexer("let v <- 2 in v end")
91             >>> [tk.kind.name for tk in l.tokens()]
92             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
93             """
94
95             token = self.getToken()
96             while token.kind != TokenType.EOF:
97                 if token.kind != TokenType.WSP and token.kind != TokenType.COM \
98                     and token.kind != TokenType.NLN:
99                     yield token
100                token = self.getToken()
101
102    def getToken(self):
103        """
104            Return the next token.

```

```
104     TODO: Implement this method!
105     """
106     token = None
107     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 class Parser:
33     def __init__(self, tokens):
34         """
35             Initializes the parser. The parser keeps track of the list of tokens
36             and the current token. For instance:
37         """
38         self.tokens = list(tokens)
39         self.cur_token_idx = 0 # This is just a suggestion!
40
41     def parse(self):
42         """
43             Returns the expression associated with the stream of tokens.
44
45             Examples:
46             >>> parser = Parser([Token('123', TokenType.NUM)])
47             >>> exp = parser.parse()
48             >>> ev = EvalVisitor()
49             >>> exp.accept(ev, None)
50             123
51
52             >>> parser = Parser([Token('True', TokenType.TRU)])
53             >>> exp = parser.parse()
54             >>> ev = EvalVisitor()
55             >>> exp.accept(ev, None)
56             True
57
58             >>> parser = Parser([Token('False', TokenType.FLS)])
59             >>> exp = parser.parse()
60             >>> ev = EvalVisitor()
61             >>> exp.accept(ev, None)
62             False
63
64             >>> tk0 = Token('~', TokenType.NEG)
65             >>> tk1 = Token('123', TokenType.NUM)
66             >>> parser = Parser([tk0, tk1])
67             >>> exp = parser.parse()
68             >>> ev = EvalVisitor()
69             >>> exp.accept(ev, None)
70             -123
71
72             >>> tk0 = Token('3', TokenType.NUM)
73             >>> tk1 = Token('*', TokenType.MUL)
74             >>> tk2 = Token('4', TokenType.NUM)
75             >>> parser = Parser([tk0, tk1, tk2])
76             >>> exp = parser.parse()
77             >>> ev = EvalVisitor()
78             >>> exp.accept(ev, None)
79             12
80
81             >>> tk0 = Token('3', TokenType.NUM)
82             >>> tk1 = Token('*', TokenType.MUL)
83             >>> tk2 = Token('~', TokenType.NEG)
84             >>> tk3 = Token('4', TokenType.NUM)
85             >>> parser = Parser([tk0, tk1, tk2, tk3])
86             >>> exp = parser.parse()
87             >>> ev = EvalVisitor()
88             >>> exp.accept(ev, None)
89             -12
90
91             >>> tk0 = Token('30', TokenType.NUM)
92             >>> tk1 = Token('/', TokenType.DIV)
93             >>> tk2 = Token('4', TokenType.NUM)
94             >>> parser = Parser([tk0, tk1, tk2])
95             >>> exp = parser.parse()
96             >>> ev = EvalVisitor()
97             >>> exp.accept(ev, None)
98             7
99
100            >>> tk0 = Token('3', TokenType.NUM)
101            >>> tk1 = Token('+', TokenType.ADD)
102            >>> tk2 = Token('4', TokenType.NUM)
103            >>> parser = Parser([tk0, tk1, tk2])

```

```

104     >>> exp = parser.parse()
105     >>> ev = EvalVisitor()
106     >>> exp.accept(ev, None)
107     7
108
109     >>> tk0 = Token('30', TokenType.NUM)
110     >>> tk1 = Token('-', TokenType.SUB)
111     >>> tk2 = Token('4', TokenType.NUM)
112     >>> parser = Parser([tk0, tk1, tk2])
113     >>> exp = parser.parse()
114     >>> ev = EvalVisitor()
115     >>> exp.accept(ev, None)
116     26
117
118     >>> tk0 = Token('2', TokenType.NUM)
119     >>> tk1 = Token('*', TokenType.MUL)
120     >>> tk2 = Token('(', TokenType.LPR)
121     >>> tk3 = Token('3', TokenType.NUM)
122     >>> tk4 = Token('+', TokenType.ADD)
123     >>> tk5 = Token('4', TokenType.NUM)
124     >>> tk6 = Token(')', TokenType.RPR)
125     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
126     >>> exp = parser.parse()
127     >>> ev = EvalVisitor()
128     >>> exp.accept(ev, None)
129     14
130
131     >>> tk0 = Token('4', TokenType.NUM)
132     >>> tk1 = Token('==', TokenType.EQL)
133     >>> tk2 = Token('4', TokenType.NUM)
134     >>> parser = Parser([tk0, tk1, tk2])
135     >>> exp = parser.parse()
136     >>> ev = EvalVisitor()
137     >>> exp.accept(ev, None)
138     True
139
140     >>> tk0 = Token('4', TokenType.NUM)
141     >>> tk1 = Token('<', TokenType.LEQ)
142     >>> tk2 = Token('4', TokenType.NUM)
143     >>> parser = Parser([tk0, tk1, tk2])
144     >>> exp = parser.parse()
145     >>> ev = EvalVisitor()
146     >>> exp.accept(ev, None)
147     True
148
149     >>> tk0 = Token('4', TokenType.NUM)
150     >>> tk1 = Token('<', TokenType.LTH)
151     >>> tk2 = Token('4', TokenType.NUM)
152     >>> parser = Parser([tk0, tk1, tk2])
153     >>> exp = parser.parse()
154     >>> ev = EvalVisitor()
155     >>> exp.accept(ev, None)
156     False
157
158     >>> tk0 = Token('not', TokenType.NOT)
159     >>> tk1 = Token('(', TokenType.LPR)
160     >>> tk2 = Token('4', TokenType.NUM)
161     >>> tk3 = Token('<', TokenType.LTH)
162     >>> tk4 = Token('4', TokenType.NUM)
163     >>> tk5 = Token(')', TokenType.RPR)
164     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
165     >>> exp = parser.parse()
166     >>> ev = EvalVisitor()
167     >>> exp.accept(ev, None)
168     True
169
170     >>> tk0 = Token('true', TokenType.TRU)
171     >>> tk1 = Token('or', TokenType.ORX)
172     >>> tk2 = Token('false', TokenType.FLS)
173     >>> parser = Parser([tk0, tk1, tk2])
174     >>> exp = parser.parse()
175     >>> ev = EvalVisitor()
176     >>> exp.accept(ev, None)
177     True
178
179     >>> tk0 = Token('true', TokenType.TRU)
180     >>> tk1 = Token('and', TokenType.AND)
181     >>> tk2 = Token('false', TokenType.FLS)
182     >>> parser = Parser([tk0, tk1, tk2])
183     >>> exp = parser.parse()
184     >>> ev = EvalVisitor()
185     >>> exp.accept(ev, None)
186     False
187
188     >>> tk0 = Token('let', TokenType.LET)
189     >>> tk1 = Token('v', TokenType.VAR)
190     >>> tk2 = Token('<-', TokenType.ASN)
191     >>> tk3 = Token('42', TokenType.NUM)
192     >>> tk4 = Token('in', TokenType.INX)
193     >>> tk5 = Token('v', TokenType.VAR)
194     >>> tk6 = Token('end', TokenType.END)
195     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
196     >>> exp = parser.parse()
197     >>> ev = EvalVisitor()
198     >>> exp.accept(ev, {})
199     42
200
201     >>> tk0 = Token('let', TokenType.LET)
202     >>> tk1 = Token('v', TokenType.VAR)
203     >>> tk2 = Token('<-', TokenType.ASN)
204     >>> tk3 = Token('21', TokenType.NUM)
205     >>> tk4 = Token('in', TokenType.INX)
206     >>> tk5 = Token('v', TokenType.VAR)
207     >>> tk6 = Token('21', TokenType.NUM)
208     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
209     >>> exp = parser.parse()
210     >>> ev = EvalVisitor()
211     >>> exp.accept(ev, {})
212     21

```

```

20/
208    >>> tk6 = Token('+', TokenType.ADD)
209    >>> tk7 = Token('v', TokenType.VAR)
210    >>> tk8 = Token('end', TokenType.END)
211    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
212    >>> exp = parser.parse()
213    >>> ev = EvalVisitor()
214    >>> exp.accept(ev, {})
215
216    42
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))
256
257    Fn(v)
258
259    >>> tk0 = Token('(', TokenType.LPR)
260    >>> tk1 = Token('fn', TokenType.FNX)
261    >>> tk2 = Token('v', TokenType.VAR)
262    >>> tk3 = Token('=>', TokenType.ARW)
263    >>> tk4 = Token('v', TokenType.VAR)
264    >>> tk5 = Token('+', TokenType.ADD)
265    >>> tk6 = Token('1', TokenType.NUM)
266    >>> tk7 = Token(')', TokenType.RPR)
267    >>> tk8 = Token('2', TokenType.NUM)
268    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
269    >>> exp = parser.parse()
270    >>> ev = EvalVisitor()
271    >>> exp.accept(ev, {})
272
273    3
274
275    """
276
277    # TODO: implement this method.
278
279    return None

```

Expression.py

```

1  from abc import ABC, abstractmethod
2  from Visitor import *
3
4  class Expression(ABC):
5      @abstractmethod
6      def accept(self, visitor, arg):
7          raise NotImplementedError
8
9  class Var(Expression):
10     """
11     This class represents expressions that are identifiers. The value of an
12     identifier is the value associated with it in the environment table.
13     """
14     def __init__(self, identifier):
15         self.identifier = identifier
16     def accept(self, visitor, arg):
17         """
18         Example:
19         >>> e = Var('var')
20         >>> ev = EvalVisitor()
21         >>> e.accept(ev, {'var': 42})
22         42
23
24         >>> e = Var('v42')
25         >>> ev = EvalVisitor()
26         >>> e.accept(ev, {'v42': True, 'v31': 5})
27         True
28         """
29     return visitor.visit_var(self, arg)
30
31 class Bln(Expression):
32     """
33     This class represents expressions that are boolean values. There are only
34     two boolean values: true and false. The acceptuation of such an expression is
35     the boolean itself.
36     """
37     def __init__(self, bln):
38         self.bln = bln
39     def accept(self, visitor, arg):
40         """
41         Example:
42         >>> e = Bln(True)
43         >>> ev = EvalVisitor()
44         >>> e.accept(ev, None)
45         True
46         """
47     return visitor.visit_bln(self, arg)
48
49 class Num(Expression):
50     """
51     This class represents expressions that are numbers. The acceptuation of such
52     an expression is the number itself.
53     """
54     def __init__(self, num):
55         self.num = num
56     def accept(self, visitor, arg):
57         """
58         Example:
59         >>> e = Num(3)
60         >>> ev = EvalVisitor()
61         >>> e.accept(ev, None)
62         3
63         """
64     return visitor.visit_num(self, arg)
65
66 class BinaryExpression(Expression):
67     """
68     This class represents binary expressions. A binary expression has two
69     sub-expressions: the left operand and the right operand.
70     """
71     def __init__(self, left, right):
72         self.left = left
73         self.right = right
74
75     @abstractmethod
76     def accept(self, visitor, arg):
77         raise NotImplementedError
78
79 class Eql(BinaryExpression):
80     """
81     This class represents the equality between two expressions. The acceptuation
82     of such an expression is True if the subexpressions are the same, or false
83     otherwise.
84     """
85     def accept(self, visitor, arg):
86         """
87         Example:
88         >>> n1 = Num(3)
89         >>> n2 = Num(4)
90         >>> e = Eql(n1, n2)
91         >>> ev = EvalVisitor()
92         >>> e.accept(ev, None)
93         False
94
95         >>> n1 = Num(3)
96         >>> n2 = Num(3)
97         >>> e = Eql(n1, n2)
98         >>> ev = EvalVisitor()
99         >>> e.accept(ev, None)
100        True
101        """
102    return visitor.visit_eql(self, arg)
103

```

```

104 class Add(BinaryExpression):
105     """
106         This class represents addition of two expressions. The acceptuation of such
107         an expression is the addition of the two subexpression's values.
108     """
109     def accept(self, visitor, arg):
110         """
111             Example:
112             >>> n1 = Num(3)
113             >>> n2 = Num(4)
114             >>> e = Add(n1, n2)
115             >>> ev = EvalVisitor()
116             >>> e.accept(ev, None)
117             7
118         """
119         return visitor.visit_add(self, arg)
120
121 class And(BinaryExpression):
122     """
123         This class represents the logical disjunction of two boolean expressions.
124         The evaluation of an expression of this kind is the logical AND of the two
125         subexpression's values.
126     """
127     def accept(self, visitor, arg):
128         """
129             Example:
130             >>> b1 = Bln(True)
131             >>> b2 = Bln(False)
132             >>> e = And(b1, b2)
133             >>> ev = EvalVisitor()
134             >>> e.accept(ev, None)
135             False
136
137             >>> b1 = Bln(True)
138             >>> b2 = Bln(True)
139             >>> e = And(b1, b2)
140             >>> ev = EvalVisitor()
141             >>> e.accept(ev, None)
142             True
143         """
144         return visitor.visit_and(self, arg)
145
146 class Or(BinaryExpression):
147     """
148         This class represents the logical conjunction of two boolean expressions.
149         The evaluation of an expression of this kind is the logical OR of the two
150         subexpression's values.
151     """
152     def accept(self, visitor, arg):
153         """
154             Example:
155             >>> b1 = Bln(True)
156             >>> b2 = Bln(False)
157             >>> e = Or(b1, b2)
158             >>> ev = EvalVisitor()
159             >>> e.accept(ev, None)
160             True
161
162             >>> b1 = Bln(False)
163             >>> b2 = Bln(False)
164             >>> e = Or(b1, b2)
165             >>> ev = EvalVisitor()
166             >>> e.accept(ev, None)
167             False
168         """
169         return visitor.visit_or(self, arg)
170
171 class Sub(BinaryExpression):
172     """
173         This class represents subtraction of two expressions. The acceptuation of such
174         an expression is the subtraction of the two subexpression's values.
175     """
176     def accept(self, visitor, arg):
177         """
178             Example:
179             >>> n1 = Num(3)
180             >>> n2 = Num(4)
181             >>> e = Sub(n1, n2)
182             >>> ev = EvalVisitor()
183             >>> e.accept(ev, None)
184             -1
185         """
186         return visitor.visit_sub(self, arg)
187
188 class Mul(BinaryExpression):
189     """
190         This class represents multiplication of two expressions. The acceptuation of
191         such an expression is the product of the two subexpression's values.
192     """
193     def accept(self, visitor, arg):
194         """
195             Example:
196             >>> n1 = Num(3)
197             >>> n2 = Num(4)
198             >>> e = Mul(n1, n2)
199             >>> ev = EvalVisitor()
200             >>> e.accept(ev, None)
201             12
202         """
203         return visitor.visit_mul(self, arg)
204
205 class Div(BinaryExpression):
206     """
207         This class represents division of two expressions. The acceptuation of such
208         an expression is the quotient of the two subexpression's values.
209     """
210     def accept(self, visitor, arg):
211         """
212             Example:
213             >>> n1 = Num(3)
214             >>> n2 = Num(4)
215             >>> e = Div(n1, n2)
216             >>> ev = EvalVisitor()
217             >>> e.accept(ev, None)
218             0.75
219         """
220         return visitor.visit_div(self, arg)

```

```

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

```

```

310
311     Example:
312     >>> n = Num(3)
313     >>> e = Neg(n)
314     >>> ev = EvalVisitor()
315     >>> e.accept(ev, None)
316     -3
317
318     >>> n = Num(0)
319     >>> e = Neg(n)
320     >>> ev = EvalVisitor()
321     >>> e.accept(ev, None)
322     0
323     """
324
325 class Not(UnaryExpression):
326     """
327         This expression represents the negation of a boolean. The negation of a
328         boolean expression is the logical complement of that expression.
329     """
330
331     def accept(self, visitor, arg):
332         """
333             Example:
334             >>> t = Bln(True)
335             >>> e = Not(t)
336             >>> ev = EvalVisitor()
337             >>> e.accept(ev, None)
338             False
339             >>> t = Bln(False)
340             >>> e = Not(t)
341             >>> ev = EvalVisitor()
342             >>> e.accept(ev, None)
343             True
344             """
345
346     return visitor.visit_not(self, arg)
347
348 class Let(Expression):
349     """
350         This class represents a let expression. The semantics of a let expression,
351         such as "let v <- e0 in e1" on an environment env is as follows:
352         1. Evaluate e0 in the environment env, yielding e0_val
353         2. Evaluate e1 in the new environment env' = env + {v:e0_val}
354     """
355
356     def __init__(self, identifier, exp_def, exp_body):
357         self.identifier = identifier
358         self.exp_def = exp_def
359         self.exp_body = exp_body
360
361     def accept(self, visitor, arg):
362         """
363             Example:
364             >>> e = Let('v', Num(42), Var('v'))
365             >>> ev = EvalVisitor()
366             >>> e.accept(ev, {})
367             42
368
369             >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
370             >>> ev = EvalVisitor()
371             >>> e.accept(ev, {})
372             1764
373             """
374
375     return visitor.visit_let(self, arg)
376
377 class Fn(Expression):
378     """
379         This class represents an anonymous function.
380
381         >>> f = Fn('v', Mul(Var('v'), Var('v')))
382         >>> ev = EvalVisitor()
383         >>> print(f.accept(ev, {}))
384         Fn(v)
385     """
386
387     def __init__(self, formal, body):
388         self.formal = formal
389         self.body = body
390
391     def accept(self, visitor, arg):
392         return visitor.visit_fn(self, arg)
393
394 class App(Expression):
395     """
396         This class represents a function application, such as 'e0 e1'. The semantics
397         of an application is as follows: we evaluate the left side, e.g., e0. It
398         must result into a function fn(p, b) denoting a function that takes in a
399         parameter p and evaluates a body b. We then evaluates e1, to obtain a value
400         v. Finally, we evaluate b, but in a context where p is bound to v.
401
402         Examples:
403             >>> f = Fn('v', Mul(Var('v'), Var('v')))
404             >>> e = App(f, Add(Num(40), Num(2)))
405             >>> ev = EvalVisitor()
406             >>> e.accept(ev, {})
407             1764
408
409             >>> f = Fn('v', Mul(Var('v'), Var('w')))
410             >>> e = Let('w', Num(3), App(f, Num(2)))
411             >>> ev = EvalVisitor()
412             >>> e.accept(ev, {})
413             6
414
415             >>> e = Let('f', Fn('x', Add(Var('x'), Num(1))), App(Var('f'), Num(1)))
416             >>> ev = EvalVisitor()

```

```

414     >>> e.accept(ev, {})
415     2
416
417     >>> e0 = Let('w', Num(3), App(Var('f'), Num(1)))
418     >>> e1 = Let('f', Fn('v', Add(Var('v'), Var('w'))), e0)
419     >>> e2 = Let('w', Num(2), e1)
420     >>> ev = EvalVisitor()
421     >>> e2.accept(ev, {})
422     3
423
424     """
425     def __init__(self, function, actual):
426         self.function = function
427         self.actual = actual
428     def accept(self, visitor, arg):
429         return visitor.visit_app(self, arg)
429
430 class IfThenElse(Expression):
431     """
432     This class represents a conditional expression. The semantics an expression
433     such as 'if B then E0 else E1' is as follows:
434     1. Evaluate B. Call the result ValueB.
435     2. If ValueB is True, then evaluate E0 and return the result.
436     3. If ValueB is False, then evaluate E1 and return the result.
437     Notice that we only evaluate one of the two sub-expressions, not both. Thus,
438     "if True then 0 else 1 div 0" will return 0 indeed.
439     """
440     def __init__(self, cond, e0, e1):
441         self.cond = cond
442         self.e0 = e0
443         self.e1 = e1
444     def accept(self, visitor, arg):
445         """
446         Example:
447         >>> e = IfThenElse(Bln(True), Num(42), Num(30))
448         >>> ev = EvalVisitor()
449         >>> e.accept(ev, {})
450         42
451
452         >>> e = IfThenElse(Bln(False), Num(42), Num(30))
453         >>> ev = EvalVisitor()
454         >>> e.accept(ev, {})
455         30
456         """
457         return visitor.visit_ifThenElse(self, arg)

```

Visitor.py

```

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    @abstractmethod
79    def visit_fn(self, exp, arg):
80        pass
81
82    @abstractmethod
83    def visit_app(self, exp, arg):
84        pass
85
86
87 class Function():
88     """
89         This is the class that represents functions. This class lets us distinguish
90         the three types that now exist in the language: numbers, booleans and
91         functions. Notice that the evaluation of an expression can now be a
92         function. For instance:
93
94         >>> f = Fn('v', Mul(Var('v'), Var('v')))
95         >>> ev = Evalvisitor()
96         >>> fval = f.accept(ev, {})
97         >>> type(fval)
98         <class 'Visitor.Function'>
99     """
100    def __init__(self, formal, body, env):
101        self.formal = formal
102        self.body = body
103        self.env = env

```



```
207      ot code! You must implement the evaluation of a function application.  
208      """  
209      # TODO: Implement this method.  
210      raise NotImplementedException
```

[VPL](#)