

2025_2 - COMPILADORES - METATURMA

[PAÍNEL](#) > [MINHAS TURMAS](#) > [2025_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)
 > [AV9 - FUNÇÕES RECURSIVAS](#)

[Descrição](#)

[Visualizar envios](#)

AV9 - Funções recursivas

Data de entrega: segunda, 20 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 recursivas à nossa mini-linguagem. Iremos também efetuar algumas mudanças em sua gramática, para que ela se torne um subconjunto de [Standard ML](#). Com essas modificações, seremos capazes de escrever funções mais complexas na linguagem. As figuras abaixo ilustra alguns dos programas que podem ser escritos usando funções recursivas:

Fatorial:

```
let
  fun fact n =
    if n < 2
    then 1
    else n * fact (n-1)
in
  fact 10
end
```

Outro exemplo da aplicação da função loop. Agora estamos obtendo $2^2 \cdot 2^2 \cdot 2^2$:

```
let
  fun loop n = fn f => fn a =>
    if n = 1
    then a
    else loop (n-1) f (f a)
in
  loop 10 (fn x => x * x) 2
end
```

Maior divisor comum. A chamada gcd a b encontra o maior divisor comum entre a e b:

```
let
  fun gcd a = fn b =>
    if b = 0
    then a
    else if a < b
    then gcd b a
    else gcd b (a-b)
in
  gcd 120 42
end
```

Conta a quantidade de dígitos em um número:

```
let
  fun countDigits n =
    if n = 0
    then 0
    else 1 + countDigits (n div 10)
in
  countDigits 123456
end
```

Sequência de Fibonacci

```
let
  fun fib n =
    if n < 2
    then 1
    else fib (n-1) + fib (n-2)
in
  fib 5
end
```

Soma de progressão aritmética com razão um:

```
let
  fun range n0 = fn n1 =>
    if n0 < n1
    then n0 + range (n0 + 1) n1
    else 0
in
  range 2 7
end
```

Soma os dígitos de um número. Neste exemplo, sDigs 123 = 6

```
let
  fun sDigs n =
    if n = 0
    then 0
    else (n mod 10) + sDigs (n div 10)
in
  sumOfDigits 123
end
```

Inverte os dígitos de um número. Neste exemplo, temos que reverse 18 = 81

```
let
  fun helper n = fn rev =>
    if n = 0 then rev else helper (n div 10) (rev * 10 + (n mod 10))
in
  let
    fun reverse n = helper n 0
  in
    reverse 18
  end
end
```

loop n f a aplica a função f ao argumento a n vezes. Abaixo estamos incrementando o número 2:

```
let
  fun loop n = fn f => fn a =>
    if n = 1
    then a
    else loop (n-1) f (f a)
in
  loop 10 (fn x => x + 1) 2
end
```

pow x y produz eleva x à y-ésima potência.

```
let
  fun pow a = fn b =>
    if b = 0 then 1
    else a * pow a (b-1)
in
  pow 2 7
end
```

Multiplica os dígitos de um número. Neste exemplo, pDigs 1234 = 24

```
let
  fun pDigs n =
    if n = 1
    then 1
    else (n mod 10) * pDigs (n div 10)
in
  pDigs 1234
end
```

Figura 1: Exemplos de diferentes funções recursivas em nossa mini-linguagem. Note que qualquer um desses programas deve também funcionar em SML/NJ

A adição de recursão à linguagem de programação implica em algumas mudanças na implementação da linguagem que somente possui funções anônimas. Além disso, neste trabalho iremos implementar algumas mudanças gramaticais, e adicionar divisão e módulo à linguagem (para que todos os exemplos acima funcionem). Abaixo salientamos as mudanças.

Análise Léxica

O analisador léxico (`Lexer.py`) deverá ser alterado para incluir os tokens que participam da declaração de funções, os tokens que implementam divisão e aritmética modular e os tokens que declaram variáveis. As seguintes modificações devem ser implementadas:

1. O operador de divisão anterior (/), deve ser alterado para div. Essa modificação busca deixar nossa mini-linguagem igual à SML/NJ.
2. Um operador de módulo deve ser adicionado: mod. Assim, tanto $7 \text{ div } 3 = 2$ quanto $7 \% 3 = 1$ passam a ser sentenças válidas.
3. A declaração de variáveis deve ser alterada. Agora temos dois tokens para declarar variáveis: val e fun. O segundo é usado para declarar funções.

Note que o arquivo `Lexer.py` já contem a lista de tokens que você deverá usar. Você terá que alterar a implementação da função `getToken`.

Análise Sintática

O analisador sintático (`Parser.py`) deverá ser alterado para incluir declarações de funções e aplicações de funções. A figura 2 contém uma sugestão para a construção da gramática. Note que a nova gramática modifica nossa mini-linguagem de várias formas:

1. Adição de operadores de divisão e módulo via palavras reservadas `div` e `mod`.
2. Prefixação de declaração com as palavras reservadas `fun` e `val`.
3. Substituição do operador de atribuição `<-` pelo símbolo de igual (`=`).

```

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 ::= uny_exp ([*|div|mod] uny_exp)*
uny_exp ::= <not> uny_exp
           | ~ uny_exp
           | let_exp
let_exp ::= <let> decl <in> fn_exp <end>
           | val_exp
val_exp ::= val_tk (val_tk)*
val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>
decl ::= val <var> = fn_exp
       | fun <var> <var> = fn_exp

```

Figura 2: a gramática sugerida para a linguagem com funções recursivas.

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

`eval(env, fun f x = Body) → RFunction(f, x, Body, env)`

Figura 3: a semântica de funções. Funções recursivas são representadas como um valor diferente daquele usado para representar funções anônimas.

$$\begin{array}{c}
\frac{\text{eval}(\text{env}, \text{E}1) \rightarrow \text{AFunction}(f, x, \text{Body}, \text{env}) \quad \text{eval}(\text{env}, \text{E}2) \rightarrow v_1 \quad \text{eval}(\text{env} \cup \{x:v_1\}, \text{body}) \rightarrow v_2}{\text{eval}(\text{env}, \text{E}1 \text{ E}2) \rightarrow v_2} \\
\\
\frac{\text{eval}(\text{env}, \text{E}1) \rightarrow \text{RFunction}(f, x, \text{Body}, \text{env}) \quad \text{eval}(\text{env}, \text{E}2) \rightarrow v_1}{\text{eval}(\text{env} \cup \{f:\text{RFunction}(f, x, \text{Body}, \text{env}), x:v_1\}, \text{body}) \rightarrow v_2} \\
\text{eval}(\text{env}, \text{E}1 \text{ E}2) \rightarrow v_2
\end{array}$$

Figura 4: a semântica de aplicação de funções. Agora temos dois casos que precisam ser considerados: aplicação de funções anônimas e aplicação de funções recursivas.

Note que assim como no exercício que envolvia somente funções anônimas, a declaração de funções continua tendo a menor precedência possível, mas a aplicação de funções ainda tem a maior precedência possível. Assim, uma construção como $\text{fn } x \Rightarrow x + 1$ deve ser entendida como $(\text{fn } x \Rightarrow x) + 1$ 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 $f \ \sim 1$ é um erro sintático, enquanto a construção $f \ (\sim 1)$ é válida.

Expressões

A linguagem passa a ter quatro tipos de valores: números (Num), booleanos (Bln), funções anônimas (Fn) e funções com nomes (Fun). As classes que denotam essas expressões já estão implementadas em `Expression.py`. A nova linguagem passa também a ter expressões do tipo Mod, que implementam a aritmética modular. Note que isso nada tem a ver com a implementação de funções. Estamos somente melhorando a expressividade da linguagem. O arquivo `Expression.py` não precisa ser modificado: você pode utilizar, sem qualquer alteração, o arquivo que está disponível neste enunciado. De todo modo, fique à vontade para alterar aquele arquivo se julgar apropriado.

Interpretador

A definição do visitor (Classe `EvalVisitor` em `Visitor.py`) que implementa o interpretador deverá ser alterada, para interpretar, além das funções anônimas, também as funções recursivas. As regras semânticas são dadas na Figura 3 e na Figura 4. Note que estamos mostrando as regras semânticas tanto para funções anônimas quanto para funções recursivas. A sua nova implementação de `EvalVisitor.visit_app` agora terá de tratar dois casos, conforme visto nas Figuras 3 e 4. Você pode separar um caso do outro testando o tipo da expressão do lado esquerdo da aplicação:

```

def visit_app(self, exp, env):
    fval = exp.function.accept(self, env)
    if isinstance(fval, Function):
        return None # do something here!
    elif isinstance(fval, RecFunction):
        return None # do something here!

```

```

else:
    sys.exit("Type 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`. Para testar sua implementação localmente, você pode usar o comando abaixo:

```

$> python3 driver.py
let
  fun helper n = fn rev =>
    if n = 0
    then rev
    else helper (n div 10) (rev * 10 + (n mod 10))
in
let
  fun reverse n = helper n 0
in
  reverse 18
end
end # Lembre-se: CTRL+D
81

```

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 nao deve ser alterado, mas deve ser enviado para resolver o
10            VPL. O arquivo contem o codigo que testa a implementacao 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
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 div 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     VAL = 212 # The 'val' declaration
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     FUN = 220 # The 'fun' declaration
56     MOD = 221 # The 'mod' operator
57
58
59 class Lexer:
60
61     def __init__(self, source):
62         """
63             The constructor of the lexer. It receives the string that shall be
64             scanned.
65             TODO: You will need to implement this method.
66         """
67         pass
68
69     def tokens(self):
70         """
71             This method is a token generator: it converts the string encapsulated
72             into this object into a sequence of Tokens. Notice that this method
73             filters out three kinds of tokens: white-spaces, comments and new lines.
74
75             Examples:
76
77             >>> l = Lexer("1 + 3")
78             >>> [tk.kind for tk in l.tokens()]
79             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
80
81             >>> l = Lexer('1 * 2\n')
82             >>> [tk.kind for tk in l.tokens()]
83             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
84
85             >>> l = Lexer('1 * 2 -- 3\n')
86             >>> [tk.kind for tk in l.tokens()]
87             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
88
89             >>> l = Lexer("1 + var")
90             >>> [tk.kind for tk in l.tokens()]
91             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
92
93             >>> l = Lexer("let val v = 2 in v end")
94             >>> [tk.kind.name for tk in l.tokens()]
95             ['LET', 'VAL', 'VAR', 'EQL', 'NUM', 'INX', 'VAR', 'END']
96
97             >>> l = Lexer("fn x => x + 1")
98             >>> [tk.kind.name for tk in l.tokens()]
99             ['FNX', 'VAR', 'ARW', 'VAR', 'ADD', 'NUM']
100
101            >>> l = Lexer("fun inc a = a + 1")
102            >>> [tk.kind.name for tk in l.tokens()]
103            ['FUN', 'VAR', 'VAR', 'EQL', 'VAR', 'ADD', 'NUM']

```

```
104
105     token = self.getToken()
106     while token.kind != TokenType.EOF:
107         if token.kind != TokenType.WSP and token.kind != TokenType.COM \
108             and token.kind != TokenType.NLN:
109             yield token
110         token = self.getToken()
111
112     def getToken(self):
113         """
114             Return the next token.
115             TODO: Implement this method!
116         """
117         token = None
118         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 ::= uny_exp ([*|/|mod] uny_exp)*
20 uny_exp ::= <not> uny_exp
21     | ~ uny_exp
22     | let_exp
23 let_exp ::= <let> decl <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 decl ::= val <var> = fn_exp
29     | fun <var> <var> = fn_exp
30
31 References:
32     see https://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm#classic
33 """
34
35 class Parser:
36     def __init__(self, tokens):
37         """
38             Initializes the parser. The parser keeps track of the list of tokens
39             and the current token. For instance:
40             """
41         self.tokens = list(tokens)
42         self.cur_token_idx = 0 # This is just a suggestion!
43         # TODO: you might want to implement more stuff in the initializer :)
44
45     def parse(self):
46         """
47             Returns the expression associated with the stream of tokens.
48
49             Examples:
50             >>> parser = Parser([Token('123', TokenType.NUM)])
51             >>> exp = parser.parse()
52             >>> ev = EvalVisitor()
53             >>> exp.accept(ev, None)
54             123
55
56             >>> parser = Parser([Token('True', TokenType.TRU)])
57             >>> exp = parser.parse()
58             >>> ev = EvalVisitor()
59             >>> exp.accept(ev, None)
60             True
61
62             >>> parser = Parser([Token('False', TokenType.FLS)])
63             >>> exp = parser.parse()
64             >>> ev = EvalVisitor()
65             >>> exp.accept(ev, None)
66             False
67
68             >>> tk0 = Token('~', TokenType.NEG)
69             >>> tk1 = Token('123', TokenType.NUM)
70             >>> parser = Parser([tk0, tk1])
71             >>> exp = parser.parse()
72             >>> ev = EvalVisitor()
73             >>> exp.accept(ev, None)
74             -123
75
76             >>> tk0 = Token('3', TokenType.NUM)
77             >>> tk1 = Token('*', TokenType.MUL)
78             >>> tk2 = Token('4', TokenType.NUM)
79             >>> parser = Parser([tk0, tk1, tk2])
80             >>> exp = parser.parse()
81             >>> ev = EvalVisitor()
82             >>> exp.accept(ev, None)
83             12
84
85             >>> tk0 = Token('3', TokenType.NUM)
86             >>> tk1 = Token('*', TokenType.MUL)
87             >>> tk2 = Token('~', TokenType.NEG)
88             >>> tk3 = Token('4', TokenType.NUM)
89             >>> parser = Parser([tk0, tk1, tk2, tk3])
90             >>> exp = parser.parse()
91             >>> ev = EvalVisitor()
92             >>> exp.accept(ev, None)
93             -12
94
95             >>> tk0 = Token('30', TokenType.NUM)
96             >>> tk1 = Token('div', TokenType.DIV)
97             >>> tk2 = Token('4', TokenType.NUM)
98             >>> parser = Parser([tk0, tk1, tk2])
99             >>> exp = parser.parse()
100            >>> ev = EvalVisitor()
101            >>> exp.accept(ev, None)
102            7
103

```

```

104     >>> tk0 = Token('3', TokenType.NUM)
105     >>> tk1 = Token('+', TokenType.ADD)
106     >>> tk2 = Token('4', TokenType.NUM)
107     >>> parser = Parser([tk0, tk1, tk2])
108     >>> exp = parser.parse()
109     >>> ev = EvalVisitor()
110     >>> exp.accept(ev, None)
111     7
112
113     >>> tk0 = Token('30', TokenType.NUM)
114     >>> tk1 = Token('-', TokenType.SUB)
115     >>> tk2 = Token('4', TokenType.NUM)
116     >>> parser = Parser([tk0, tk1, tk2])
117     >>> exp = parser.parse()
118     >>> ev = EvalVisitor()
119     >>> exp.accept(ev, None)
120     26
121
122     >>> tk0 = Token('2', TokenType.NUM)
123     >>> tk1 = Token('*', TokenType.MUL)
124     >>> tk2 = Token('(', TokenType.LPR)
125     >>> tk3 = Token('3', TokenType.NUM)
126     >>> tk4 = Token('+', TokenType.ADD)
127     >>> tk5 = Token('4', TokenType.NUM)
128     >>> tk6 = Token(')', TokenType.RPR)
129     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
130     >>> exp = parser.parse()
131     >>> ev = EvalVisitor()
132     >>> exp.accept(ev, None)
133     14
134
135     >>> tk0 = Token('4', TokenType.NUM)
136     >>> tk1 = Token('==', TokenType.EQL)
137     >>> tk2 = Token('4', TokenType.NUM)
138     >>> parser = Parser([tk0, tk1, tk2])
139     >>> exp = parser.parse()
140     >>> ev = EvalVisitor()
141     >>> exp.accept(ev, None)
142     True
143
144     >>> tk0 = Token('4', TokenType.NUM)
145     >>> tk1 = Token('<=', TokenType.LEQ)
146     >>> tk2 = Token('4', TokenType.NUM)
147     >>> parser = Parser([tk0, tk1, tk2])
148     >>> exp = parser.parse()
149     >>> ev = EvalVisitor()
150     >>> exp.accept(ev, None)
151     True
152
153     >>> tk0 = Token('4', TokenType.NUM)
154     >>> tk1 = Token('<', TokenType.LTH)
155     >>> tk2 = Token('4', TokenType.NUM)
156     >>> parser = Parser([tk0, tk1, tk2])
157     >>> exp = parser.parse()
158     >>> ev = EvalVisitor()
159     >>> exp.accept(ev, None)
160     False
161
162     >>> tk0 = Token('not', TokenType.NOT)
163     >>> tk1 = Token('(', TokenType.LPR)
164     >>> tk2 = Token('4', TokenType.NUM)
165     >>> tk3 = Token('<', TokenType.LTH)
166     >>> tk4 = Token('4', TokenType.NUM)
167     >>> tk5 = Token(')', TokenType.RPR)
168     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
169     >>> exp = parser.parse()
170     >>> ev = EvalVisitor()
171     >>> exp.accept(ev, None)
172     True
173
174     >>> tk0 = Token('true', TokenType.TRU)
175     >>> tk1 = Token('or', TokenType.ORX)
176     >>> tk2 = Token('false', TokenType.FLS)
177     >>> parser = Parser([tk0, tk1, tk2])
178     >>> exp = parser.parse()
179     >>> ev = EvalVisitor()
180     >>> exp.accept(ev, None)
181     True
182
183     >>> tk0 = Token('true', TokenType.TRU)
184     >>> tk1 = Token('and', TokenType.AND)
185     >>> tk2 = Token('false', TokenType.FLS)
186     >>> parser = Parser([tk0, tk1, tk2])
187     >>> exp = parser.parse()
188     >>> ev = EvalVisitor()
189     >>> exp.accept(ev, None)
190     False
191
192     >>> tk0 = Token('let', TokenType.LET)
193     >>> tk1 = Token('val', TokenType.VAL)
194     >>> tk2 = Token('v', TokenType.VAR)
195     >>> tk3 = Token('=', TokenType.EQL)
196     >>> tk4 = Token('42', TokenType.NUM)
197     >>> tk5 = Token('in', TokenType.INX)
198     >>> tk6 = Token('v', TokenType.VAR)
199     >>> tk7 = Token('end', TokenType.END)
200     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
201     >>> exp = parser.parse()
202     >>> ev = EvalVisitor()
203     >>> exp.accept(ev, {})
204     42
205
206     >>> tk0 = Token('let', TokenType.LET)
...     ...     ...     ...     ...

```

```

20/
208    >>> tk1 = Token('val', TokenType.VAL)
209    >>> tk2 = Token('v', TokenType.VAR)
210    >>> tk3 = Token('=', TokenType.EQL)
211    >>> tk4 = Token('21', TokenType.NUM)
212    >>> tk5 = Token('in', TokenType.INX)
213    >>> tk6 = Token('v', TokenType.VAR)
214    >>> tk7 = Token('+', TokenType.ADD)
215    >>> tk8 = Token('v', TokenType.VAR)
216    >>> tk9 = Token('end', TokenType.END)
217    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8, tk9])
218    >>> exp = parser.parse()
219    >>> ev = EvalVisitor()
220    >>> exp.accept(ev, {})
221    42
222
223    >>> tk0 = Token('if', TokenType.IFX)
224    >>> tk1 = Token('2', TokenType.NUM)
225    >>> tk2 = Token('<', TokenType.LTH)
226    >>> tk3 = Token('3', TokenType.NUM)
227    >>> tk4 = Token('then', TokenType.THN)
228    >>> tk5 = Token('1', TokenType.NUM)
229    >>> tk6 = Token('else', TokenType.ELS)
230    >>> tk7 = Token('2', TokenType.NUM)
231    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
232    >>> exp = parser.parse()
233    >>> ev = EvalVisitor()
234    >>> exp.accept(ev, None)
235    1
236
237    >>> tk0 = Token('if', TokenType.IFX)
238    >>> tk1 = Token('false', TokenType.FLS)
239    >>> tk2 = Token('then', TokenType.THN)
240    >>> tk3 = Token('1', TokenType.NUM)
241    >>> tk4 = Token('else', TokenType.ELS)
242    >>> tk5 = Token('2', TokenType.NUM)
243    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
244    >>> exp = parser.parse()
245    >>> ev = EvalVisitor()
246    >>> exp.accept(ev, None)
247    2
248
249    >>> tk0 = Token('fn', TokenType.FNX)
250    >>> tk1 = Token('v', TokenType.VAR)
251    >>> tk2 = Token('>', TokenType.ARW)
252    >>> tk3 = Token('v', TokenType.VAR)
253    >>> tk4 = Token('+', TokenType.ADD)
254    >>> tk5 = Token('1', TokenType.NUM)
255    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
256    >>> exp = parser.parse()
257    >>> ev = EvalVisitor()
258    >>> print(exp.accept(ev, None))
259    Fn(v)
260
261    >>> tk0 = Token('(', TokenType.LPR)
262    >>> tk1 = Token('fn', TokenType.FNX)
263    >>> tk2 = Token('v', TokenType.VAR)
264    >>> tk3 = Token('>', TokenType.ARW)
265    >>> tk4 = Token('v', TokenType.VAR)
266    >>> tk5 = Token('+', TokenType.ADD)
267    >>> tk6 = Token('1', TokenType.NUM)
268    >>> tk7 = Token(')', TokenType.RPR)
269    >>> tk8 = Token('2', TokenType.NUM)
270    >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
271    >>> exp = parser.parse()
272    >>> ev = EvalVisitor()
273    >>> exp.accept(ev, {})
274    3
275
276    >>> t0 = Token('let', TokenType.LET)
277    >>> t1 = Token('fun', TokenType.FUN)
278    >>> t2 = Token('f', TokenType.VAR)
279    >>> t3 = Token('v', TokenType.VAR)
280    >>> t4 = Token('=', TokenType.EQL)
281    >>> t5 = Token('v', TokenType.VAR)
282    >>> t6 = Token('+', TokenType.ADD)
283    >>> t7 = Token('v', TokenType.VAR)
284    >>> t8 = Token('in', TokenType.INX)
285    >>> t9 = Token('f', TokenType.VAR)
286    >>> tA = Token('2', TokenType.NUM)
287    >>> tB = Token('end', TokenType.END)
288    >>> parser = Parser([t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, tA, tB])
289    >>> exp = parser.parse()
290    >>> ev = EvalVisitor()
291    >>> exp.accept(ev, {})
292    4
293    """
294    # TODO: implement this method.
295    return None

```

Expression.py

```

1  from abc import ABC, abstractmethod
2  from Visitor import *
3
4 """
5 This file adds recursive functions to our language. You can follow the
6 semantics of these functions using the rules below, which were written in
7 Prolog:
8
9 eval(fn(Formal, Body), Env, fval(Formal, Body, Env)).
10 eval(fun(Name, Formal, Body), Env, rfun(Name, Formal, Body, Env)).
11
12 eval(apply(Function, Actual), Env, Value) :-
13     eval(Function, Env, fval(Formal, Body, Nesting)),
14     eval(Actual, Env, PValue),
15     eval(Body, [(Formal, PValue)|Nesting], Value).
16
17 eval(applyrec(Function, Actual), Env, Value) :-
18     eval(Function, Env, rfun(Name, Formal, Body, Nesting)),
19     eval(Actual, Env, PValue),
20     NEnv = [(Name, rfun(Name, Formal, Body, Nesting)), (Formal, PValue)|Nesting]
21     eval(Body, NEnv, Value).
22 """
23
24
25
26 class Expression(ABC):
27     @abstractmethod
28     def accept(self, visitor, arg):
29         raise NotImplementedError
30
31
32 class Var(Expression):
33 """
34     This class represents expressions that are identifiers. The value of an
35     identifier is the value associated with it in the environment table.
36 """
37
38     def __init__(self, identifier):
39         self.identifier = identifier
40
41     def accept(self, visitor, arg):
42 """
43         Example:
44         >>> e = Var('var')
45         >>> ev = EvalVisitor()
46         >>> e.accept(ev, {'var': 42})
47         42
48
49         >>> e = Var('v42')
50         >>> ev = EvalVisitor()
51         >>> e.accept(ev, {'v42': True, 'v31': 5})
52         True
53 """
54     return visitor.visit_var(self, arg)
55
56
57 class Bln(Expression):
58 """
59     This class represents expressions that are boolean values. There are only
60     two boolean values: true and false. The effect of such an expression is
61     the boolean itself.
62 """
63
64     def __init__(self, bln):
65         self.bln = bln
66
67     def accept(self, visitor, arg):
68 """
69         Example:
70         >>> e = Bln(True)
71         >>> ev = EvalVisitor()
72         >>> e.accept(ev, None)
73         True
74 """
75     return visitor.visit_bln(self, arg)
76
77
78 class Num(Expression):
79 """
80     This class represents expressions that are numbers. The effect of such
81     an expression is the number itself.
82 """
83
84     def __init__(self, num):
85         self.num = num
86
87     def accept(self, visitor, arg):
88 """
89         Example:
90         >>> e = Num(3)
91         >>> ev = EvalVisitor()
92         >>> e.accept(ev, None)
93         3
94 """
95     return visitor.visit_num(self, arg)
96
97
98 class BinaryExpression(Expression):
99 """
100    This class represents binary expressions. A binary expression has two
101    sub-expressions: the left operand and the right operand.
102 """

```

```

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

```

```

20/
208     >>> e.accept(ev, None)
209     False
210
211     return visitor.visit_or(self, arg)
212
213 class Sub(BinaryExpression):
214     """
215     This class represents subtraction of two expressions. The effect of such
216     an expression is the subtraction of the two subexpression's values.
217     """
218
219     def accept(self, visitor, arg):
220         """
221             Example:
222             >>> n1 = Num(3)
223             >>> n2 = Num(4)
224             >>> e = Sub(n1, n2)
225             >>> ev = EvalVisitor()
226             >>> e.accept(ev, None)
227             -1
228             """
229
230         return visitor.visit_sub(self, arg)
231
232 class Mul(BinaryExpression):
233     """
234     This class represents multiplication of two expressions. The effect of
235     such an expression is the product of the two subexpression's values.
236     """
237
238     def accept(self, visitor, arg):
239         """
240             Example:
241             >>> n1 = Num(3)
242             >>> n2 = Num(4)
243             >>> e = Mul(n1, n2)
244             >>> ev = EvalVisitor()
245             >>> e.accept(ev, None)
246             12
247             """
248
249         return visitor.visit_mul(self, arg)
250
251 class Div(BinaryExpression):
252     """
253     This class represents the integer division of two expressions. The
254     effect of such an expression is the integer quotient of the two
255     subexpression's values.
256     """
257
258     def accept(self, visitor, arg):
259         """
260             Example:
261             >>> n1 = Num(28)
262             >>> n2 = Num(4)
263             >>> e = Div(n1, n2)
264             >>> ev = EvalVisitor()
265             >>> e.accept(ev, None)
266             7
267
268             >>> n1 = Num(22)
269             >>> n2 = Num(4)
270             >>> e = Div(n1, n2)
271             >>> ev = EvalVisitor()
272             >>> e.accept(ev, None)
273             5
274             """
275
276         return visitor.visit_div(self, arg)
277
278 class Mod(BinaryExpression):
279     """
280     This class represents the integer modulo of two expressions. The
281     effect of such an expression is the integer quotient of the two
282     subexpression's values.
283     """
284
285     def accept(self, visitor, arg):
286         """
287             Example:
288             >>> n1 = Num(28)
289             >>> n2 = Num(4)
290             >>> e = Mod(n1, n2)
291             >>> ev = EvalVisitor()
292             >>> e.accept(ev, None)
293             0
294
295             >>> n1 = Num(22)
296             >>> n2 = Num(4)
297             >>> e = Mod(n1, n2)
298             >>> ev = EvalVisitor()
299             >>> e.accept(ev, None)
300             2
301             """
302
303         return visitor.visit_mod(self, arg)
304
305 class Leq(BinaryExpression):
306     """
307     This class represents comparison of two expressions using the
308     less-than-or-equal comparator. The effect of such an expression is a
309     boolean value that is true if the left operand is less than or equal the
310     right operand. It is false otherwise.

```

```

310     right_operand, it is raise otherwise.
311
312
313     def accept(self, visitor, arg):
314         """
315             Example:
316             >>> n1 = Num(3)
317             >>> n2 = Num(4)
318             >>> e = Leq(n1, n2)
319             >>> ev = EvalVisitor()
320             >>> e.accept(ev, None)
321             True
322
323             >>> n1 = Num(3)
324             >>> n2 = Num(3)
325             >>> e = Leq(n1, n2)
326             >>> ev = EvalVisitor()
327             >>> e.accept(ev, None)
328             True
329
330             >>> n1 = Num(4)
331             >>> n2 = Num(3)
332             >>> e = Leq(n1, n2)
333             >>> ev = EvalVisitor()
334             >>> e.accept(ev, None)
335             False
336         """
337
338         return visitor.visit_leq(self, arg)
339
340     class Lth(BinaryExpression):
341         """
342             This class represents comparison of two expressions using the
343             less-than comparison operator. The effect of such an expression is a
344             boolean value that is true if the left operand is less than the right
345             operand. It is false otherwise.
346         """
347
348         def accept(self, visitor, arg):
349             """
350                 Example:
351                 >>> n1 = Num(3)
352                 >>> n2 = Num(4)
353                 >>> e = Lth(n1, n2)
354                 >>> ev = EvalVisitor()
355                 >>> e.accept(ev, None)
356                 True
357
358                 >>> n1 = Num(3)
359                 >>> n2 = Num(3)
360                 >>> e = Lth(n1, n2)
361                 >>> ev = EvalVisitor()
362                 >>> e.accept(ev, None)
363                 False
364
365                 >>> n1 = Num(4)
366                 >>> n2 = Num(3)
367                 >>> e = Lth(n1, n2)
368                 >>> ev = EvalVisitor()
369                 >>> e.accept(ev, None)
370                 False
371             """
372
373         return visitor.visit_lth(self, arg)
374
375     class UnaryExpression(Expression):
376         """
377             This class represents unary expressions. A unary expression has only one
378             sub-expression.
379         """
380
381         def __init__(self, exp):
382             self.exp = exp
383
384         @abstractmethod
385         def accept(self, visitor, arg):
386             raise NotImplementedError
387
388
389     class Neg(UnaryExpression):
390         """
391             This expression represents the additive inverse of a number. The additive
392             inverse of a number n is the number -n, so that the sum of both is zero.
393         """
394
395         def accept(self, visitor, arg):
396             """
397                 Example:
398                 >>> n = Num(3)
399                 >>> e = Neg(n)
400                 >>> ev = EvalVisitor()
401                 >>> e.accept(ev, None)
402                 -3
403
404                 >>> n = Num(0)
405                 >>> e = Neg(n)
406                 >>> ev = EvalVisitor()
407                 >>> e.accept(ev, None)
408                 0
409             """
410
411         return visitor.visit_neg(self, arg)
412
413     class Not(UnaryExpression):

```

```

414
415     This expression represents the negation of a boolean. The negation of a
416     boolean expression is the logical complement of that expression.
417     """
418
419     def accept(self, visitor, arg):
420         """
421             Example:
422             >>> t = Bln(True)
423             >>> e = Not(t)
424             >>> ev = EvalVisitor()
425             >>> e.accept(ev, None)
426             False
427
428             >>> t = Bln(False)
429             >>> e = Not(t)
430             >>> ev = EvalVisitor()
431             >>> e.accept(ev, None)
432             True
433             """
434
435     return visitor.visit_not(self, arg)
436
437 class Let(Expression):
438     """
439         This class represents a let expression. The semantics of a let expression,
440         such as "let v <- e0 in e1" on an environment env is as follows:
441         1. Evaluate e0 in the environment env, yielding e0_val
442         2. Evaluate e1 in the new environment env' = env + {v:e0_val}
443         """
444
445     def __init__(self, identifier, exp_def, exp_body):
446         self.identifier = identifier
447         self.exp_def = exp_def
448         self.exp_body = exp_body
449
450     def accept(self, visitor, arg):
451         """
452             Example:
453             >>> e = Let('v', Num(42), Var('v'))
454             >>> ev = EvalVisitor()
455             >>> e.accept(ev, {})
456             42
457
458             >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
459             >>> ev = EvalVisitor()
460             >>> e.accept(ev, {})
461             42
462
463             >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
464             >>> ev = EvalVisitor()
465             >>> e.accept(ev, {})
466             1764
467             """
468
469     return visitor.visit_let(self, arg)
470
471 class Fn(Expression):
472     """
473         This class represents an anonymous function.
474
475             >>> f = Fn('v', Mul(Var('v'), Var('v')))
476             >>> ev = EvalVisitor()
477             >>> print(f.accept(ev, {}))
478             Fn(v)
479             """
480
481     def __init__(self, formal, body):
482         self.formal = formal
483         self.body = body
484
485     def accept(self, visitor, arg):
486         return visitor.visit_fn(self, arg)
487
488
489 class Fun(Expression):
490     """
491         This class represents a named function. Named functions can be invoked
492         recursively.
493
494             >>> f = Fun('f', 'v', Mul(Var('v'), Var('v')))
495             >>> ev = EvalVisitor()
496             >>> print(f.accept(ev, {}))
497             Fun f(v)
498             """
499
500     def __init__(self, name, formal, body):
501         self.name = name
502         self.formal = formal
503         self.body = body
504
505     def accept(self, visitor, arg):
506         return visitor.visit_fun(self, arg)
507
508
509 class App(Expression):
510     """
511         This class represents a function application, such as 'e0 e1'. The semantics
512         of an application is as follows: we evaluate the left side, e.g., e0. It
513         must result into a function fn(p, b) denoting a function that takes in a
514         parameter p and evaluates a body b. We then evaluates e1, to obtain a value
515         v. Finally, we evaluate b, but in a context where p is bound to v.
516

```

```

517 Examples:
518     >>> f = Fn('v', Mul(Var('v'), Var('v')))
519     >>> e = App(f, Add(Num(40), Num(2)))
520     >>> ev = EvalVisitor()
521     >>> e.accept(ev, {})
522     1764
523
524     >>> f = Fn('v', Mul(Var('v'), Var('w')))
525     >>> e = Let('w', Num(3), App(f, Num(2)))
526     >>> ev = EvalVisitor()
527     >>> e.accept(ev, {})
528     6
529
530     >>> e = Let('f', Fn('x', Add(Var('x'), Num(1))), App(Var('f'), Num(1)))
531     >>> ev = EvalVisitor()
532     >>> e.accept(ev, {})
533     2
534
535     >>> e0 = Let('w', Num(3), App(Var('f'), Num(1)))
536     >>> e1 = Let('f', Fn('v', Add(Var('v'), Var('w'))), e0)
537     >>> e2 = Let('w', Num(2), e1)
538     >>> ev = EvalVisitor()
539     >>> e2.accept(ev, {})
540     3
541
542     >>> e0 = Fun('f', 'v', Mul(Var('v'), Var('v')))
543     >>> e1 = Add(Num(3), Num(4))
544     >>> e2 = App(e0, e1)
545     >>> ev = EvalVisitor()
546     >>> print(e2.accept(ev, {}))
547     49
548     """
549
550     def __init__(self, function, actual):
551         self.function = function
552         self.actual = actual
553
554     def accept(self, visitor, arg):
555         return visitor.visit_app(self, arg)
556
557
558 class IfThenElse(Expression):
559     """
560     This class represents a conditional expression. The semantics an expression
561     such as 'if B then E0 else E1' is as follows:
562     1. Evaluate B. Call the result ValueB.
563     2. If ValueB is True, then evaluate E0 and return the result.
564     3. If ValueB is False, then evaluate E1 and return the result.
565     Notice that we only evaluate one of the two sub-expressions, not both. Thus,
566     "if True then 0 else 1 div 0" will return 0 indeed.
567     """
568
569     def __init__(self, cond, e0, e1):
570         self.cond = cond
571         self.e0 = e0
572         self.e1 = e1
573
574     def accept(self, visitor, arg):
575         """
576         Example:
577         >>> e = IfThenElse(Bln(True), Num(42), Num(30))
578         >>> ev = EvalVisitor()
579         >>> e.accept(ev, {})
580         42
581
582         >>> e = IfThenElse(Bln(False), Num(42), Num(30))
583         >>> ev = EvalVisitor()
584         >>> e.accept(ev, {})
585         30
586         """
587         return visitor.visit_ifThenElse(self, arg)

```

Visitor.py

```

1 import sys
2 from abc import ABC, abstractmethod
3 from Expression import *
4
5
6 class Visitor(ABC):
7     """
8         The visitor pattern consists of two abstract classes: the Expression and the
9             Visitor. The Expression class defines on method: 'accept(visitor, args)'.
10            This method takes in an implementation of a visitor, and the arguments that
11                are passed from expression to expression. The Visitor class defines one
12                    specific method for each subclass of Expression. Each instance of such a
13                        subclass will invoke the right visiting method.
14                """
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_mod(self, exp, arg):
58     pass
59
60 @abstractmethod
61 def visit_leq(self, exp, arg):
62     pass
63
64 @abstractmethod
65 def visit_lth(self, exp, arg):
66     pass
67
68 @abstractmethod
69 def visit_neg(self, exp, arg):
70     pass
71
72 @abstractmethod
73 def visit_not(self, exp, arg):
74     pass
75
76 @abstractmethod
77 def visit_let(self, exp, arg):
78     pass
79
80 @abstractmethod
81 def visit_ifThenElse(self, exp, arg):
82     pass
83
84 @abstractmethod
85 def visit_fn(self, exp, arg):
86     pass
87
88 @abstractmethod
89 def visit_fun(self, exp, arg):
90     pass
91
92 @abstractmethod
93 def visit_app(self, exp, arg):
94     pass
95
96
97 class Function:
98     """
99         This is the class that represents functions. This class lets us distinguish
100            the three types that now exist in the language: numbers, booleans and
101                functions. Notice that the evaluation of an expression can now be a
102                    function. For instance:
103

```



```

20/      raise NotImplementedException
208
209 def visit_or(self, exp, env):
210     # TODO: Implement this method.
211     raise NotImplementedException
212
213 def visit_add(self, exp, env):
214     # TODO: Implement this method.
215     raise NotImplementedException
216
217 def visit_sub(self, exp, env):
218     # TODO: Implement this method.
219     raise NotImplementedException
220
221 def visit_mul(self, exp, env):
222     # TODO: Implement this method.
223     raise NotImplementedException
224
225 def visit_div(self, exp, env):
226     # TODO: Implement this method.
227     raise NotImplementedException
228
229 def visit_mod(self, exp, env):
230     # TODO: Implement this method.
231     raise NotImplementedException
232
233 def visit_leq(self, exp, env):
234     # TODO: Implement this method.
235     raise NotImplementedException
236
237 def visit_lth(self, exp, env):
238     # TODO: Implement this method.
239     raise NotImplementedException
240
241 def visit_neg(self, exp, env):
242     # TODO: Implement this method.
243     raise NotImplementedException
244
245 def visit_not(self, exp, env):
246     # TODO: Implement this method.
247     raise NotImplementedException
248
249 def visit_let(self, exp, env):
250     # TODO: Implement this method.
251     raise NotImplementedException
252
253 def visit_ifThenElse(self, exp, env):
254     # TODO: Implement this method.
255     raise NotImplementedException
256
257 def visit_fn(self, exp, env):
258     """
259         The evaluation of a function is the function itself. Remember: in our
260         language, functions are values as well. So, now we have four kinds of
261         values: numbers, booleans, anonymous functions and named functions.
262     """
263     return Function(exp.formal, exp.body, env)
264
265 def visit_fun(self, exp, env):
266     """
267         The evaluation of a named function returns a value that is the function
268         itself. However, we use a different type of value: RecFunction. In this
269         way, we have access to the name of the named function (and that's why
270         they are called named functions :).
271     """
272     return RecFunction(exp.name, exp.formal, exp.body, env)
273
274 def visit_app(self, exp, env):
275     """
276         The application of function to actual parameter must contain two cases:
277         1. An anonymous function is applied: (fn x => x + 1) 2
278         2. A named function is applied: f 2, where f is fun f a = a + a
279         The only difference between these two cases is that in the second we
280         must augment the environment with the name of the named function.
281
282         Example:
283         >>> f = Fun('f', 'v', Mul(Var('v'), Var('v')))
284         >>> e0 = Let('f', f, App(Var('f'), Num(2)))
285         >>> ev = EvalVisitor()
286         >>> e0.accept(ev, {})
287         4
288     """
289     # TODO: Implement this method.
290     raise NotImplementedException

```