2025_2 - COMPILADORES - METATURMA

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
```

Sequência de Fibonacci

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

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

```
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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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 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.

$$\frac{\textbf{eval}(env, E1) \rightarrow \textbf{AFunction}(f, x, Body, env) \quad \textbf{eval}(env, E2) \rightarrow v1 \quad \textbf{eval}(env\ U\ \{x{:}v1\}, body) \rightarrow v2}{\textbf{eval}(env, E1\ E2) \rightarrow v2}$$

$$\frac{\begin{array}{c}\textbf{eval}(env, E1) \rightarrow \textbf{RFunction}(f, x, Body, env) \\ \textbf{eval}(env, E2) \rightarrow v1 \\ \textbf{eval}(env\ U\ \{f{:}\textbf{RFunction}(f, x, Body, env), x{:}v1\}, body) \rightarrow v2\end{array}}{\textbf{eval}(env, E1\ E2) \rightarrow v2}$$

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 `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.

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, substituta `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

```
import sys
from Expression import *
from Visitor import *
from Lexer import Lexer
from Parser import Parser

if __name__ == "__main__":
    """
    Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
    VPL. O arquivo contem o codigo que testa a implementacao do parser.
    """
    text = sys.stdin.read()
    lexer = Lexer(text)
    parser = Parser(lexer.tokens())
    exp = parser.parse()
    visitor = EvalVisitor()
    print(f"{exp.accept(visitor, {})}")
```

Lexer.py

```
import sys
import enum


class Token:
    """
    This class contains the definition of Tokens. A token has two fields: its
    text and its kind. The "kind" of a token is a constant that identifies it
    uniquely. See the TokenType to know the possible identifiers (if you want).
    You don't need to change this class.
    """
    def __init__(self, tokenText, tokenKind):
        # The token's actual text. Used for identifiers, strings, and numbers.
        self.text = tokenText
        # The TokenType that this token is classified as.
        self.kind = tokenKind


class TokenType(enum.Enum):
    """
    These are the possible tokens. You don't need to change this class at all.
    """

    EOF = -1  # End of file
    NLN = 0  # New line
    WSP = 1  # White Space
    COM = 2  # Comment
    NUM = 3  # Number (integers)
    STR = 4  # Strings
    TRU = 5  # The constant true
    FLS = 6  # The constant false
    VAR = 7  # An identifier
    LET = 8  # The 'let' of the let expression
    INX = 9  # The 'in' of the let expression
    END = 10  # The 'end' of the let expression
    EQL = 201  # x = y
    ADD = 202  # x + y
    SUB = 203  # x - y
    MUL = 204  # x * y
    DIV = 205  # x div y
    LEQ = 206  # x <= y
    LTH = 207  # x < y
    NEG = 208  # ~x
    NOT = 209  # not x
    LPR = 210  # (
    RPR = 211  # )
    VAL = 212  # The 'val' declaration
    ORX = 213  # x or y
    AND = 214  # x and y
    IFX = 215  # The 'if' of a conditional expression
    THN = 216  # The 'then' of a conditional expression
    ELS = 217  # The 'else' of a conditional expression
    FNX = 218  # The 'fn' that declares an anonymous function
    ARW = 219  # The '=>' that separates the parameter from the body of function
    FUN = 220  # The 'fun' declaration
    MOD = 221  # The 'mod' operator


class Lexer:

    def __init__(self, source):
        """
        The constructor of the lexer. It receives the string that shall be
        scanned.
        TODO: You will need to implement this method.
        """
        pass

    def tokens(self):
        """
        This method is a token generator: it converts the string encapsulated
        into this object into a sequence of Tokens. Notice that this method
        filters out three kinds of tokens: white-spaces, comments and new lines.

        Examples:

        >>> l = Lexer("1 + 3")
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]

        >>> l = Lexer('1 * 2\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]

        >>> l = Lexer('1 * 2 -- 3\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]

        >>> l = Lexer("1 + var")
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]

        >>> l = Lexer("let val v = 2 in v end")
        >>> [tk.kind.name for tk in l.tokens()]
        ['LET', 'VAL', 'VAR', 'EQL', 'NUM', 'INX', 'VAR', 'END']

        >>> l = Lexer("fn x => x + 1")
        >>> [tk.kind.name for tk in l.tokens()]
        ['FNX', 'VAR', 'ARW', 'VAR', 'ADD', 'NUM']

        >>> l = Lexer("fun inc a = a + 1")
        >>> [tk.kind.name for tk in l.tokens()]
        ['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

```
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements a parser for SML with anonymous functions. The grammar is
as follows:

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

References:
    see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#classic
"""

class Parser:
    def __init__(self, tokens):
        """
        Initializes the parser. The parser keeps track of the list of tokens
        and the current token. For instance:
        """
        self.tokens = list(tokens)
        self.cur_token_idx = 0 # This is just a suggestion!
        # TODO: you might want to implement more stuff in the initializer :)

    def parse(self):
        """
        Returns the expression associated with the stream of tokens.

        Examples:
        >>> parser = Parser([Token('123', TokenType.NUM)])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        123

        >>> parser = Parser([Token('True', TokenType.TRU)])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        True

        >>> parser = Parser([Token('False', TokenType.FLS)])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        False

        >>> tk0 = Token('~', TokenType.NEG)
        >>> tk1 = Token('123', TokenType.NUM)
        >>> parser = Parser([tk0, tk1])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        -123

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        12

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('~', TokenType.NEG)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        -12

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('div', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        7

```

```
>>> tk0 = Token('3', TokenType.NUM)
>>> tk1 = Token('+', TokenType.ADD)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
7

>>> tk0 = Token('30', TokenType.NUM)
>>> tk1 = Token('-', TokenType.SUB)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
26

>>> tk0 = Token('2', TokenType.NUM)
>>> tk1 = Token('*', TokenType.MUL)
>>> tk2 = Token('(', TokenType.LPR)
>>> tk3 = Token('3', TokenType.NUM)
>>> tk4 = Token('+', TokenType.ADD)
>>> tk5 = Token('4', TokenType.NUM)
>>> tk6 = Token(')', TokenType.RPR)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
14

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('==', TokenType.EQL)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
True

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('<=', TokenType.LEQ)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
True

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('<', TokenType.LTH)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
False

>>> tk0 = Token('not', TokenType.NOT)
>>> tk1 = Token('(', TokenType.LPR)
>>> tk2 = Token('4', TokenType.NUM)
>>> tk3 = Token('<', TokenType.LTH)
>>> tk4 = Token('4', TokenType.NUM)
>>> tk5 = Token(')', TokenType.RPR)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
True

>>> tk0 = Token('true', TokenType.TRU)
>>> tk1 = Token('or', TokenType.ORX)
>>> tk2 = Token('false', TokenType.FLS)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
True

>>> tk0 = Token('true', TokenType.TRU)
>>> tk1 = Token('and', TokenType.AND)
>>> tk2 = Token('false', TokenType.FLS)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
False

>>> tk0 = Token('let', TokenType.LET)
>>> tk1 = Token('val', TokenType.VAL)
>>> tk2 = Token('v', TokenType.VAR)
>>> tk3 = Token('=', TokenType.EQL)
>>> tk4 = Token('42', TokenType.NUM)
>>> tk5 = Token('in', TokenType.INX)
>>> tk6 = Token('v', TokenType.VAR)
>>> tk7 = Token('end', TokenType.END)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, {})
42

>>> tk0 = Token('let', TokenType.LET)
```

```
        >>> tk1 = Token('val', TokenType.VAL)
        >>> tk2 = Token('v', TokenType.VAR)
        >>> tk3 = Token('=', TokenType.EQL)
        >>> tk4 = Token('21', TokenType.NUM)
        >>> tk5 = Token('in', TokenType.INX)
        >>> tk6 = Token('v', TokenType.VAR)
        >>> tk7 = Token('+', TokenType.ADD)
        >>> tk8 = Token('v', TokenType.VAR)
        >>> tk9 = Token('end', TokenType.END)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8, tk9])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, {})
        42

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('2', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('3', TokenType.NUM)
        >>> tk4 = Token('then', TokenType.THN)
        >>> tk5 = Token('1', TokenType.NUM)
        >>> tk6 = Token('else', TokenType.ELS)
        >>> tk7 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        1

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('false', TokenType.FLS)
        >>> tk2 = Token('then', TokenType.THN)
        >>> tk3 = Token('1', TokenType.NUM)
        >>> tk4 = Token('else', TokenType.ELS)
        >>> tk5 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        2

        >>> tk0 = Token('fn', TokenType.FNX)
        >>> tk1 = Token('v', TokenType.VAR)
        >>> tk2 = Token('=>', TokenType.ARW)
        >>> tk3 = Token('v', TokenType.VAR)
        >>> tk4 = Token('+', TokenType.ADD)
        >>> tk5 = Token('1', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> print(exp.accept(ev, None))
        Fn(v)

        >>> tk0 = Token('(', TokenType.LPR)
        >>> tk1 = Token('fn', TokenType.FNX)
        >>> tk2 = Token('v', TokenType.VAR)
        >>> tk3 = Token('=>', TokenType.ARW)
        >>> tk4 = Token('v', TokenType.VAR)
        >>> tk5 = Token('+', TokenType.ADD)
        >>> tk6 = Token('1', TokenType.NUM)
        >>> tk7 = Token(')', TokenType.RPR)
        >>> tk8 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, {})
        3

        >>> t0 = Token('let', TokenType.LET)
        >>> t1 = Token('fun', TokenType.FUN)
        >>> t2 = Token('f', TokenType.VAR)
        >>> t3 = Token('v', TokenType.VAR)
        >>> t4 = Token('=', TokenType.EQL)
        >>> t5 = Token('v', TokenType.VAR)
        >>> t6 = Token('+', TokenType.ADD)
        >>> t7 = Token('v', TokenType.VAR)
        >>> t8 = Token('in', TokenType.INX)
        >>> t9 = Token('f', TokenType.VAR)
        >>> tA = Token('2', TokenType.NUM)
        >>> tB = Token('end', TokenType.END)
        >>> parser = Parser([t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, tA, tB])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, {})
        4
        """
        # TODO: implement this method.
        return None
```

Expression.py

```
from abc import ABC, abstractmethod
from Visitor import *

"""
This file adds recursive functions to our language. You can follow the
semantics of these functions using the rules below, which were written in
Prolog:

eval(fn(Formal, Body), Env, fval(Formal, Body, Env)).

eval(fun(Name, Formal, Body), Env, rfun(Name, Formal, Body, Env)).

eval(apply(Function, Actual), Env, Value) :-
  eval(Function, Env, fval(Formal, Body, Nesting)),
  eval(Actual, Env, PValue),
  eval(Body, [(Formal, PValue)|Nesting], Value).

eval(applyrec(Function, Actual), Env, Value) :-
  eval(Function, Env, rfun(Name, Formal, Body, Nesting)),
  eval(Actual, Env, PValue),
  NEnv = [(Name, rfun(Name, Formal, Body, Nesting)), (Formal, PValue)|Nesting]
  eval(Body, NEnv, Value).
"""


class Expression(ABC):
    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Var(Expression):
    """
    This class represents expressions that are identifiers. The value of an
    indentifier is the value associated with it in the environment table.
    """

    def __init__(self, identifier):
        self.identifier = identifier

    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Var('var')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'var': 42})
        42

        >>> e = Var('v42')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'v42': True, 'v31': 5})
        True
        """
        return visitor.visit_var(self, arg)


class Bln(Expression):
    """
    This class represents expressions that are boolean values. There are only
    two boolean values: true and false. The effect of such an expression is
    the boolean itself.
    """

    def __init__(self, bln):
        self.bln = bln

    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Bln(True)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_bln(self, arg)


class Num(Expression):
    """
    This class represents expressions that are numbers. The effect of such
    an expression is the number itself.
    """

    def __init__(self, num):
        self.num = num

    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Num(3)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        3
        """
        return visitor.visit_num(self, arg)


class BinaryExpression(Expression):
    """
    This class represents binary expressions. A binary expression has two
    sub-expressions: the left operand and the right operand.
    """

```

```
    def __init__(self, left, right):
        self.left = left
        self.right = right

    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Eql(BinaryExpression):
    """
    This class represents the equality between two expressions. The effect
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Eql(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False

        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Eql(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_eql(self, arg)


class Add(BinaryExpression):
    """
    This class represents addition of two expressions. The effect of such
    an expression is the addition of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        """
        return visitor.visit_add(self, arg)


class And(BinaryExpression):
    """
    This class represents the logical disjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical AND of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> b1 = Bln(True)
        >>> b2 = Bln(False)
        >>> e = And(b1, b2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False

        >>> b1 = Bln(True)
        >>> b2 = Bln(True)
        >>> e = And(b1, b2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_and(self, arg)


class Or(BinaryExpression):
    """
    This class represents the logical conjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical OR of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> b1 = Bln(True)
        >>> b2 = Bln(False)
        >>> e = Or(b1, b2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True

        >>> b1 = Bln(False)
        >>> b2 = Bln(False)
        >>> e = Or(b1, b2)
        >>> ev = EvalVisitor()
```

```
        >>> e.accept(ev, None)
        False
        """
        return visitor.visit_or(self, arg)


class Sub(BinaryExpression):
    """
    This class represents subtraction of two expressions. The effect of such
    an expression is the subtraction of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -1
        """
        return visitor.visit_sub(self, arg)


class Mul(BinaryExpression):
    """
    This class represents multiplication of two expressions. The effect of
    such an expression is the product of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        12
        """
        return visitor.visit_mul(self, arg)


class Div(BinaryExpression):
    """
    This class represents the integer division of two expressions. The
    effect of such an expression is the integer quocient of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7

        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        5
        """
        return visitor.visit_div(self, arg)


class Mod(BinaryExpression):
    """
    This class represents the integer modulo of two expressions. The
    effect of such an expression is the integer quocient of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Mod(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        0

        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Mod(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        2
        """
        return visitor.visit_mod(self, arg)


class Leq(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than-or-equal comparator. The effect of such an expression is a
    boolean value that is true if the left operand is less than or equal the
    right operand. It is false otherwise
```

```
        right operand. It is false otherwise.
        """

        def accept(self, visitor, arg):
            """
            Example:
            >>> n1 = Num(3)
            >>> n2 = Num(4)
            >>> e = Leq(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            True

            >>> n1 = Num(3)
            >>> n2 = Num(3)
            >>> e = Leq(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            True

            >>> n1 = Num(4)
            >>> n2 = Num(3)
            >>> e = Leq(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            False
            """
            return visitor.visit_leq(self, arg)


    class Lth(BinaryExpression):
        """
        This class represents comparison of two expressions using the
        less-than comparison operator. The effect of such an expression is a
        boolean value that is true if the left operand is less than the right
        operand. It is false otherwise.
        """

        def accept(self, visitor, arg):
            """
            Example:
            >>> n1 = Num(3)
            >>> n2 = Num(4)
            >>> e = Lth(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            True

            >>> n1 = Num(3)
            >>> n2 = Num(3)
            >>> e = Lth(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            False

            >>> n1 = Num(4)
            >>> n2 = Num(3)
            >>> e = Lth(n1, n2)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            False
            """
            return visitor.visit_lth(self, arg)


    class UnaryExpression(Expression):
        """
        This class represents unary expressions. A unary expression has only one
        sub-expression.
        """

        def __init__(self, exp):
            self.exp = exp

        @abstractmethod
        def accept(self, visitor, arg):
            raise NotImplementedError


    class Neg(UnaryExpression):
        """
        This expression represents the additive inverse of a number. The additive
        inverse of a number n is the number -n, so that the sum of both is zero.
        """

        def accept(self, visitor, arg):
            """
            Example:
            >>> n = Num(3)
            >>> e = Neg(n)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            -3

            >>> n = Num(0)
            >>> e = Neg(n)
            >>> ev = EvalVisitor()
            >>> e.accept(ev, None)
            0
            """
            return visitor.visit_neg(self, arg)


    class Not(UnaryExpression):
```

```
    """
    This expression represents the negation of a boolean. The negation of a
    boolean expression is the logical complement of that expression.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> t = Bln(True)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False

        >>> t = Bln(False)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_not(self, arg)


class Let(Expression):
    """
    This class represents a let expression. The semantics of a let expression,
    such as "let v <- e0 in e1" on an environment env is as follows:
    1. Evaluate e0 in the environment env, yielding e0_val
    2. Evaluate e1 in the new environment env' = env + {v:e0_val}
    """

    def __init__(self, identifier, exp_def, exp_body):
        self.identifier = identifier
        self.exp_def = exp_def
        self.exp_body = exp_body

    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Let('v', Num(42), Var('v'))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

        >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

        >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        1764
        """
        return visitor.visit_let(self, arg)


class Fn(Expression):
    """
    This class represents an anonymous function.

        >>> f = Fn('v', Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> print(f.accept(ev, {}))
        Fn(v)
    """

    def __init__(self, formal, body):
        self.formal = formal
        self.body = body

    def accept(self, visitor, arg):
        return visitor.visit_fn(self, arg)


class Fun(Expression):
    """
    This class represents a named function. Named functions can be invoked
    recursively.

        >>> f = Fun('f', 'v', Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> print(f.accept(ev, {}))
        Fun f(v)
    """

    def __init__(self, name, formal, body):
        self.name = name
        self.formal = formal
        self.body = body

    def accept(self, visitor, arg):
        return visitor.visit_fun(self, arg)


class App(Expression):
    """
    This class represents a function application, such as 'e0 e1'. The semantics
    of an application is as follows: we evaluate the left side, e.g., e0. It
    must result into a function fn(p, b) denoting a function that takes in a
    parameter p and evaluates a body b. We then evaluates e1, to obtain a value
    v. Finally, we evaluate b, but in a context where p is bound to v.

```

```
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 evalute 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

```
import sys
from abc import ABC, abstractmethod
from Expression import *


class Visitor(ABC):
    """
    The visitor pattern consists of two abstract classes: the Expression and the
    Visitor. The Expression class defines on method: 'accept(visitor, args)'.
    This method takes in an implementation of a visitor, and the arguments that
    are passed from expression to expression. The Visitor class defines one
    specific method for each subclass of Expression. Each instance of such a
    subclasse will invoke the right visiting method.
    """

    @abstractmethod
    def visit_var(self, exp, arg):
        pass

    @abstractmethod
    def visit_bln(self, exp, arg):
        pass

    @abstractmethod
    def visit_num(self, exp, arg):
        pass

    @abstractmethod
    def visit_eql(self, exp, arg):
        pass

    @abstractmethod
    def visit_and(self, exp, arg):
        pass

    @abstractmethod
    def visit_or(self, exp, arg):
        pass

    @abstractmethod
    def visit_add(self, exp, arg):
        pass

    @abstractmethod
    def visit_sub(self, exp, arg):
        pass

    @abstractmethod
    def visit_mul(self, exp, arg):
        pass

    @abstractmethod
    def visit_div(self, exp, arg):
        pass

    @abstractmethod
    def visit_mod(self, exp, arg):
        pass

    @abstractmethod
    def visit_leq(self, exp, arg):
        pass

    @abstractmethod
    def visit_lth(self, exp, arg):
        pass

    @abstractmethod
    def visit_neg(self, exp, arg):
        pass

    @abstractmethod
    def visit_not(self, exp, arg):
        pass

    @abstractmethod
    def visit_let(self, exp, arg):
        pass

    @abstractmethod
    def visit_ifThenElse(self, exp, arg):
        pass

    @abstractmethod
    def visit_fn(self, exp, arg):
        pass

    @abstractmethod
    def visit_fun(self, exp, arg):
        pass

    @abstractmethod
    def visit_app(self, exp, arg):
        pass


class Function:
    """
    This is the class that represents functions. This class lets us distinguish
    the three types that now exist in the language: numbers, booleans and
    functions. Notice that the evaluation of an expression can now be a
    function. For instance:

```

```
        >>> f = Fn('v', Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> fval = f.accept(ev, {})
        >>> type(fval)
        <class 'Visitor.Function'>
    """

    def __init__(self, formal, body, env):
        self.formal = formal
        self.body = body
        self.env = env

    def __str__(self):
        return f"Fn({self.formal})"


class RecFunction(Function):
    """
    This is the class that represents named functions. The key different between
    named and anonymous functions is exactly the "name" :)

        >>> f = Fun('f', 'v', Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> fval = f.accept(ev, {})
        >>> type(fval)
        <class 'Visitor.RecFunction'>
    """

    def __init__(self, name, formal, body, env):
        super().__init__(formal, body, env)
        self.name = name

    def __str__(self):
        return f"Fun {self.name}({self.formal})"


class EvalVisitor(Visitor):
    """
    The EvalVisitor class evaluates logical and arithmetic expressions. The
    result of evaluating an expression is the value of that expression. The
    inherited attribute propagated throughout visits is the environment that
    associates the names of variables with values.

    Examples:
    >>> e0 = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
    >>> e1 = Not(Eql(e0, Num(1764)))
    >>> ev = EvalVisitor()
    >>> e1.accept(ev, {})
    False

    >>> e0 = Let('v', Add(Num(40), Num(2)), Sub(Var('v'), Num(2)))
    >>> e1 = Lth(e0, Var('x'))
    >>> ev = EvalVisitor()
    >>> e1.accept(ev, {'x': 41})
    True

    >>> e0 = Fn('v', Mul(Var('v'), Var('v')))
    >>> ev = EvalVisitor()
    >>> print(e0.accept(ev, {}))
    Fn(v)

    >>> e0 = Fn('v', Mul(Var('v'), Var('v')))
    >>> e1 = Add(Num(3), Num(4))
    >>> e2 = App(e0, e1)
    >>> ev = EvalVisitor()
    >>> print(e2.accept(ev, {}))
    49

    >>> e0 = Fun('f', 'v', Mul(Var('v'), Var('v')))
    >>> ev = EvalVisitor()
    >>> print(e0.accept(ev, {}))
    Fun f(v)

    >>> e0 = Fun('f', 'v', Mul(Var('v'), Var('v')))
    >>> e1 = Add(Num(3), Num(4))
    >>> e2 = App(e0, e1)
    >>> ev = EvalVisitor()
    >>> print(e2.accept(ev, {}))
    49
    """

    def visit_var(self, exp, env):
        if exp.identifier in env:
            return env[exp.identifier]
        else:
            sys.exit("Def error")

    def visit_bln(self, exp, env):
        return exp.bln

    def visit_num(self, exp, env):
        return exp.num

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

    def visit_and(self, exp, env):
        # TODO: Implement this method.
```

```
207             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
```

VPL