**Objetivos da Atividade**:

1. *Apresentar como utilizar a ferramenta PLY para realização da etapa de análise sintática.*
   1. *Como mapear uma gramática livre de contexto para PLY.*
2. *Desenvolver a sintaxe abstrata e mapear para uma notação orientada a objetos.*
3. *Como instanciar objetos da sintaxe abstrata em memória, durante a execução da análise sintática.*
4. *Desenvolver a classe Visitor, cujo papel será o de visitar cada um dos nós da árvore gerada a partir de um código válido.*

**Etapa 1 - Análise Sintática no PLY.**

1. *O PLY pode gerar analisadores sintáticos a partir de uma gramática livre de contexto.*
2. *O analisador sintático gerado é LALR.*
3. *Essa etapa pode demandar retrabalho, pois a gramática livre de contexto deve ser livre de conflitos shift/reduce ou reduce/reduce.*

Para compreensão de como utilizar o PLY, considere a linguagem *expressionLanguage*, uma linguagem restrita a reconhecer expressões. Em *expressionLanguage*, uma expressão é definida pelas seguintes regras:

exp → exp + exp | exp \* exp | exp ^ exp | call | assign | num | id

call → id (params) | id ( )

params → id, params | id

assign → id = exp

Antes de prosseguirmos, são necessários dois passos. O primeiro definir o léxico dessa linguagem. E o segundo, eliminar a ambiguidade dessa gramática. A solução para o primeiro passo, é dada na página a seguir, já em PLY.

# -------------------------

# ExpressionLanguageLex.py

#----------------------

import ply.lex as lex

tokens = ('COMMA', 'SOMA', 'ID', 'NUMBER', 'VEZES', 'POT', 'LPAREN', 'RPAREN', 'IGUAL',)

t\_IGUAL= r'='

t\_SOMA = r'\+'

t\_VEZES = r'\\*'

t\_POT = r'\^'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

t\_COMMA = r','

t\_ID = r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

def t\_NUMBER(t):

r'\d+'

t.value = int(t.value)

return t

def t\_newline(t):

r'\n+'

t.lexer.lineno += len(t.value)

t\_ignore = ' \t'

def t\_error(t):

print("Illegal character '%s'" % t.value[0])

t.lexer.skip(1)

lexer = lex.lex()

#

# # Test it out

data = '''

3 + 4 ^ 10 + 20 \*2 = chamada(a, b, 3)

'''

lexer.input(data)

Para o segundo passo, temos de retirar a ambiguidade. A seguir a solução para esse problema:

exp → exp + exp1 | exp1

exp1 → exp1 \* exp2 | exp2

exp2 → exp3 ^ exp2 | exp3

exp3 → call | assign | num | id

call → id (params) | id ( )

params → exp, params | exp

assign → id = exp

Tomemos como exemplo, a regra exp → exp + exp1 | exp1. Sua tradução para PLY é feita da seguinte forma:

def p\_exp\_soma(p):

'''exp : exp SOMA exp1

| exp1'''

Da mesma forma para as regras exp1 → exp1 \* exp2 | exp1 → exp2, temos:

def p\_exp1\_vezes(p):

'''exp1 : exp1 VEZES exp2

| exp2'''

A seguir, apresentamos código inicial do parser, que já apresenta os casos citados. Ao final do código, é mostrado como construir o parser e, adicionalmente, como inicializar a análise sintaxe.

# -------------------------

# ExpressionLanguageParser.py

#----------------------

import ply.yacc as yacc

import ply.lex as lex

from ExpressionLanguageLex import tokens

def p\_exp\_soma(p):

'''exp : exp SOMA exp1

| exp1'''

def p\_exp1\_vezes(p):

'''exp1 : exp1 VEZES exp2

| exp2'''

parser = yacc.yacc()

result = parser.parse(debug=True)

**Exercício 1: Mapeie as demais regras da gramática citada. As regras que devem ser mapeadas estão em negrito.**

exp → exp + exp1 | exp1

exp1 → exp1 \* exp2 | exp2

**exp2 → exp3 ^ exp2 | exp3**

**exp3 → call | assign | num | id**

**call → id (params) | id ( )**

**params → id, params | id**

**assign → id = exp**

**Etapa 2 - Sintaxe Abstrata.**

**A sintaxe abstrata da gramática livre de contexto adotada é a seguinte:**

exp → exp + exp | exp \* exp | exp ^ exp | call | assign | num | id

call → id (params) | id ( )

params → id, params | id

assign → id = exp

Dessa forma, temos a seguinte sintaxe abstrata orientado a objetos:

|  |  |
| --- | --- |
| exp → | **Exp (Abstrata)** |
| exp + exp |  exp \* exp |  exp ^ exp |  call |  assign |  num |  id | **SomaExp (Exp, Exp)**  **MulExp (Exp, Exp)**  **PotExp (Exp, Exp**  **CallExp(Call)**  **AssignExp(Assign)**  **NumExp (num)**  **IDExp(id)** |
| call → | **Call (Abstrata)** |
| id (params) |  id ( ) | **ParamsCall (id, Params)**  **SimpleCall (id)** |
| params → | **Params(Abstrata)** |
| id, params |  id | **CompoundParams (id, Params)**  **SingleParam (id)** |
| assign → | **Assign(Abstrata)** |
| id = exp | **AssignExp(id, Exp)** |

**Exercício 2. Crie em Python as classes concretas e abstratas da sintaxe abstrata acima. Tome como base o exemplo descrito a seguir. Coloque todas as classes em um único arquivo denominado SintaxeAbstrata.py.**

|  |  |
| --- | --- |
| **Classe Abstrata** | **Classe Concreta** |
| from abc import abstractmethod  from abc import ABCMeta  class Exp(metaclass=ABCMeta):  @abstractmethod  def accept(self, Visitor):  pass | class SomaExp(Exp):  def \_\_init\_\_(self, exp1, exp2):  self.exp1 = exp1  self.exp2 = exp2  def accept(self, Visitor):  Visitor.visitSomaExp(self)  class MulExp(Exp):  def \_\_init\_\_(self, exp1, exp2):  self.exp1 = exp1  self.exp2 = exp2  def accept(self, Visitor):  Visitor.visitMulExp(self) |

**Etapa 3 - Instanciando Sintaxe Abstrata em Memória.**

A instanciação da Sintaxe Abstrata em memória consiste na modificação dos métodos associados a cada regra da gramática.

Vamos utilizar como exemplo o método p\_exp\_soma, definido anteriormente.

def p\_exp\_soma(p):

'''exp : exp SOMA exp1

| exp1'''

Os elementos presentes nas regras reconhecidas por p\_exp\_soma são armazenados em p. Dessa forma, caso tenhamos recebido como entrada a expressão 3 + 4, o campo p[0] representa a variável exp. O campo p[1] representa o valor 3, p[2] o símbolo + e p[3] o valor 4.

Atentem ao fato que p\_exp\_soma reconhece duas regras exp → exp SOMA exp1 | exp1. Assim, para fazermos a distinção se o analisador reconheceu a primeira ou a segunda regra, poderíamos utilizar o seguinte código:

def p\_exp\_soma(p):

'''exp : exp SOMA exp1

| exp1'''

if (p[2] = '+'): #Primeira regra

...

else: #Segunda regra

...

A implementação do método p\_exp\_soma(p), para geração da sintaxe abstrata, é feita da seguinte maneira:

def p\_exp\_soma(p):

'''exp : exp SOMA exp1

| exp1'''

if (p[2] = '+'): #Primeira regra

p[0] = SomaExp(p[1], p[3])

else: #Segunda regra

p[0] = p[1]

**Exercício 3. Modifique os métodos introduzidos no exercício 1, de forma que seja gerada a sintaxe abstrata.**

**Etapa 4 - Visitando a Sintaxe Abstrata em Memória.**

**A seguir, é apresentado o esboço inicial do Visitor de nossa atividade. Nessa atividade, o Visitor pretende reconstituir o código que foi passado como entrada.**

class Visitor():

def visitSomaExp(self, somaExp):

somaExp.exp1.accept(self)

print ('+')

somaExp.exp2.accept(self)

def visitMulExp(self, mulExp):

mulExp.exp1.accept(self)

print ('\*')

mulExp.exp2.accept(self)

def visitAssignExp(self, assignExp):

assignExp.assign.accept(self)

**Exercício 4. Implemente os demais métodos visit da classe Visitor. Avalie se a análise sintática foi implementada da forma correta.**