#### Objetivos da Atividade:

- 1. Apresentar como utilizar a ferramenta PLY para realização da etapa de análise sintática.
  - a. 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.

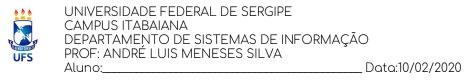
## Etapa 1 - Análise Sintática no PLY.

- 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 \rightarrow \exp + \exp \mid \exp * \exp \mid \exp \land \exp \mid call \mid assign \mid num \mid id call \rightarrow id (params) | id ( ) params \rightarrow \exp, params | \exp assign \rightarrow 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.



```
1. # -----
2. # ExpressionLanguageLex.py
3. #-----
4. import ply.lex as lex
5. tokens = ('COMMA', 'SOMA', 'ID', 'NUMBER', 'VEZES', 'POT', 'LPAREN',
   'RPAREN', 'IGUAL',)
6. t IGUAL= r'='
7. t SOMA = r' + '
8. t VEZES = r' \*'
9. t POT = r' \^'
10.t LPAREN = r' \setminus ('
11.t RPAREN = r' \setminus )'
12.t_COMMA = r','
13.t ID = r'[a-zA-Z][a-zA-Z]
14.
15.def t_NUMBER(t):
16. r'\d+'
17. t.value = int(t.value)
18. return t
19.
20. def t newline(t):
21. r'\n+'
     t.lexer.lineno += len(t.value)
23.
24.t_ignore = ' \t'
25.
26. def t error(t):
27. print("Illegal character '%s'" % t.value[0])
28. t.lexer.skip(1)
29.
30.lexer = lex.lex()
31.#
32.# # Test it out
33. data = '''
34. 3 + 4 ^ 10 + 20 *2 = chamada(a, b, 3)
35. '''
36.lexer.input(data)
```

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

```
\exp \rightarrow \exp + \exp 1 \mid \exp 1

\exp 1 \rightarrow \exp 1 * \exp 2 \mid \exp 2

\exp 2 \rightarrow \exp 3 ^ \exp 2 \mid \exp 3

\exp 3 \rightarrow \text{call} \mid \text{assign} \mid \text{num} \mid \text{id}

\text{call} \rightarrow \text{id} \text{ (params)} \mid \text{id} \text{ ()}

\text{params} \rightarrow \exp, \text{params} \mid \exp

\text{assign} \rightarrow \text{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  $exp1 \rightarrow exp1 * exp2 | exp1 \rightarrow exp2$ , temos:

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.

```
1. # -----
2. # ExpressionLanguageParser.py
3. #-----
4.
5. import ply.yacc as yacc
6. import ply.lex as lex
7. from ExpressionLanguageLex import tokens
9. def p_exp_soma(p):
10. '''exp : exp SOMA exp1
          | exp1'''
11.
12.
13. def p expl vezes(p):
     '''exp1 : exp1 VEZES exp2
14.
             | exp2'''
15.
16.
17. parser = yacc.yacc()
18.
19. result = parser.parse(debug=True)
```

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

```
\exp \rightarrow \exp + \exp 1 \mid \exp 1

\exp 1 \rightarrow \exp 1 * \exp 2 \mid \exp 2

\exp 2 \rightarrow \exp 3 * \exp 2 \mid \exp 3

\exp 3 \rightarrow \text{call } \mid \text{assign } \mid \text{num } \mid \text{id}

\operatorname{call} \rightarrow \operatorname{id} \text{ (params) } \mid \operatorname{id} \text{ ()}

\operatorname{params} \rightarrow \exp, \text{ params } \mid \exp

\operatorname{assign} \rightarrow \operatorname{id} = \exp
```

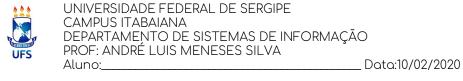
## Etapa 2 - Sintaxe Abstrata.

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

```
\exp \rightarrow \exp + \exp \mid \exp * \exp \mid \exp \land \exp \mid call \mid assign \mid num \mid id call \rightarrow id (params) | id ( ) params \rightarrow id, params | id assign \rightarrow 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)
exp, params   exp	CompoundParams (exp, Params) SingleParam (exp)
assign →	Assign(Abstrata)
id = exp	AssignAss(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	<pre>class SomaExp(Exp):     definit(self, exp1, exp2):         self.exp1 = exp1</pre>
<pre>class Exp(metaclass=ABCMeta):     @abstractmethod     def accept(self, Visitor):</pre>	<pre>self.exp2 = exp2 def accept(self, Visitor):     Visitor.visitSomaExp(self)</pre>
pass	<pre>class MulExp(Exp):     definit(self, exp1, exp2):         self.exp1 = exp1         self.exp2 = exp2     def accept(self, Visitor):         Visitor.visitMulExp(self)</pre>

## 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.

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  $\exp \rightarrow \exp$  SOMA  $\exp 1 \mid \exp 1$ . Assim, para fazermos a distinção se o analisador reconheceu a primeira ou a segunda regra, poderíamos utilizar o seguinte código:

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

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.exp1.accept(self)

def visitMulExp(self, mulExp):
        mulExp.exp1.accept(self)
        print ('*)
        mulExp.exp1.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.