Relatório do Analisador Sintático

Brendow Paolillo Castro Isidoro[1]

[1]Dacom - Universidade Técnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

brendowisidoro@alunos.utfpr.edu.br

Resumo. *Este documento descreve a implementação de um analisador sintático para a disciplina de compiladores do curso de ciência da computação. O código tem como objetivo verificar se os tokens retornados do analisador léxico se encontram no padrão de uso da linguagem. A análise utiliza gramáticas livres de contexto para reconhecer os tokens e gerar a árvore do programa.*

1. Introdução

Podemos definir um compilador como um tradutor de código fonte em alto nível para o código de máquina em baixo nível. Uma grande parte das linguagens disponíveis para os desenvolvimento de softwares e aplicativos utilizam compiladores, facilitando o desenvolvimento de novas ferramentas por abstrair conceitos complexos em funções simplificadas determinadas na documentação de cada linguagem.

A questão então é, como interpretar essas linguagens de alto nível, para algo compreensível pro computador? A implementação de um compilador é divida em várias camadas que realizam uma varredura do código e interpretam o contexto dele, dividindo-se em quatro partes principais, a (*i*) análise léxica, (*ii*) sintática, (*iii*) semântica e a (*iv*) geração de código intermediário.

Na Seção 2 está descrito como foi feita a implementação do analisador sintático. A Seção 3 demonstra os resultados desta parte, além de falar um pouco sobre a próxima etapa do processo de compilação. Por fim, na Seção 4 o artigo é concluıdo com algumas observações.

A seguir a seção 2 descreve como foi realizada a implementação do analisador sintático, descrevendo as suas gramáticas. A seção 3 demonstra o resultado gerado pela execução do código. Por último, a seção 4 apresenta as considerações finais sobre o trabalho.

2. Implementação

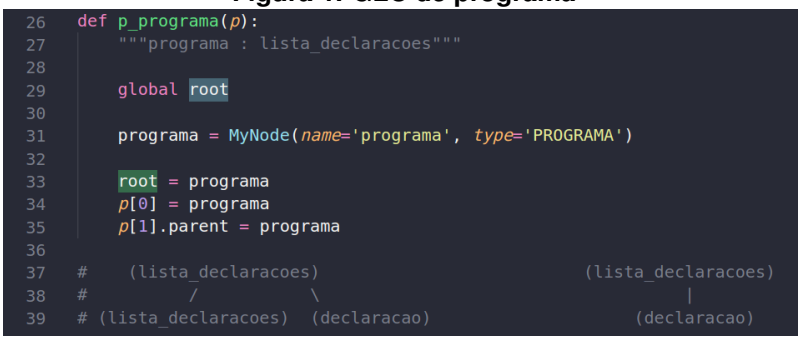
Para a implementação do analisador sintático, foi utilizado o PLY [Beazley], uma biblioteca na qual implementa o Lex e Yacc para Python e auxilia na criação dos analisadores léxico e sintático.

2.1. Verificação das gramáticas

O analisador sintático utiliza a lista de *tokens* encontrados pelo analisador léxico e verifica se ordem deles corresponde com os padrões da linguagem. Esses padrões são definidos por meio de GLC, no qual reconhecem símbolos terminais e não-terminais, e também possibilita a implementação de regras recursivas para verificação.

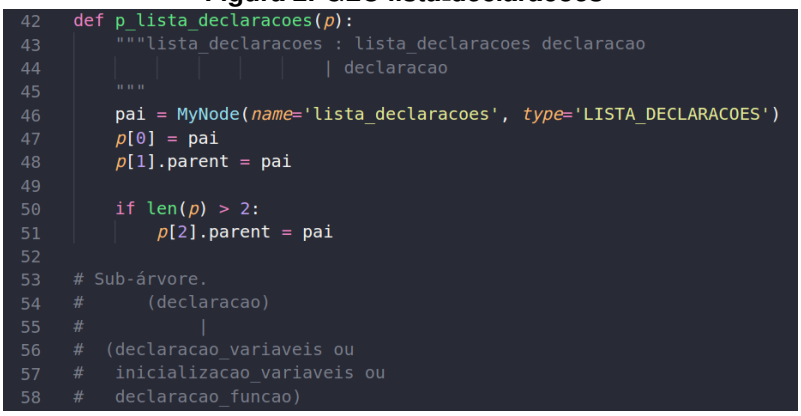
As imagens, figura 1 e figura 2, demonstram um exemplo de recursividade das GLCs. Perceba que a linha 27 do código descreve uma GLC, na qual um *token* de "programa" pode gerar nós de "lista_declaracoes", esta na qual pode gerar a si própria mais a regra da declaração, ou apenas a "declaracao". A regra de declaração gera outras regras e apenas termina quando a gramática encontra um símbolo terminal, no qual reconhece um caractere ou termo utilizado pela linguagem do TPP.

Figura 1. GLC de programa



```
26  def p_programa(p):
27      """programa : lista_declaracoes"""
28
29      global root
30
31      programa = MyNode(name='programa', type='PROGRAMA')
32
33      root = programa
34      p[0] = programa
35      p[1].parent = programa
36
37  #    (lista_declaracoes)                        (lista_declaracoes)
38  #          /          \                                 |
39  # (lista_declaracoes)  (declaracao)                  (declaracao)
```

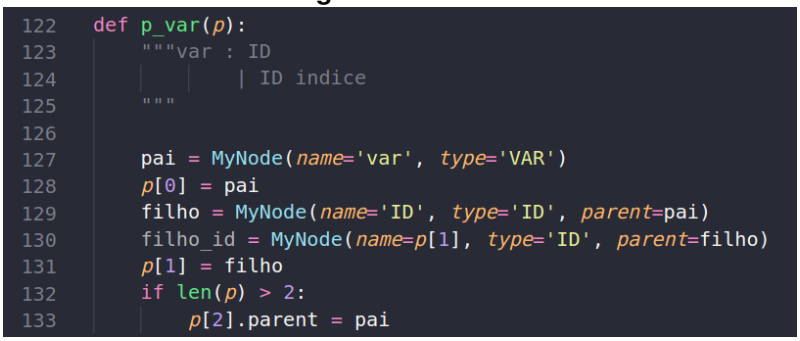
Figura 2. GLC lista_declaracoes



```
42  def p_lista_declaracoes(p):
43      """lista_declaracoes : lista_declaracoes declaracao
44                           | declaracao
45      """
46      pai = MyNode(name='lista_declaracoes', type='LISTA_DECLARACOES')
47      p[0] = pai
48      p[1].parent = pai
49
50      if len(p) > 2:
51          p[2].parent = pai
52
53  # Sub-árvore.
54  #      (declaracao)
55  #           |
56  #  (declaracao_variaveis ou
57  #   inicializacao_variaveis ou
58  #   declaracao_funcao)
```

A figura 3 demonstra o reconhecimento de um símbolo terminal, no qual este é o ID de uma variável, o ID pode ser reconhecido como uma cadeia de caracteres aceitas pelo analisador léxico.

Figura 3. GLC var



```
122  def p_var(p):
123      """var : ID
124             | ID indice
125      """
126
127      pai = MyNode(name='var', type='VAR')
128      p[0] = pai
129      filho = MyNode(name='ID', type='ID', parent=pai)
130      filho_id = MyNode(name=p[1], type='ID', parent=filho)
131      p[1] = filho
132      if len(p) > 2:
133          p[2].parent = pai
```

Após definir os conceitos para o reconhecimento dos *tokens* a partir de gramáticas livres de contexto, podemos verificar as regras que compõe o analisador sintático no qual servirão para gerar a árvore sintática do programa. A seguir é possível verificar as regras gramaticais implementadas:

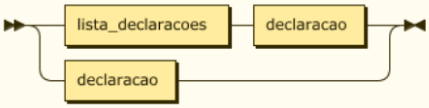
Figura 4. Gramáticas reconhecidas

programa:


```
programa ::= lista_declaracoes
```

no references

lista_declaracoes:

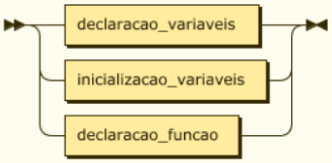



```
lista_declaracoes
         ::= lista_declaracoes declaracao
           | declaracao
```

referenced by:

- lista_declaracoes
- programa

declaracao:




```
declaracao
         ::= declaracao_variaveis
           | inicializacao_variaveis
           | declaracao_funcao
```

referenced by:

- lista_declaracoes

declaracao_variaveis:


```
declaracao_variaveis
         ::= tipo DOIS_PONTOS lista_variaveis
```

referenced by:

- acao
- declaracao

inicializacao_variaveis:

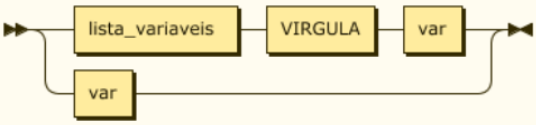

```
inicializacao_variaveis
         ::= atribuicao
```

referenced by:

- declaracao

Figura 5. Gramáticas reconhecidas

lista_variaveis:

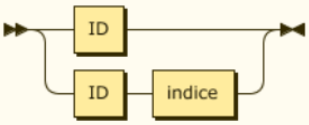



```
lista_variaveis
         ::= lista_variaveis VIRGULA var
           | var
```

referenced by:

- declaracao_variaveis
- lista_variaveis

var:

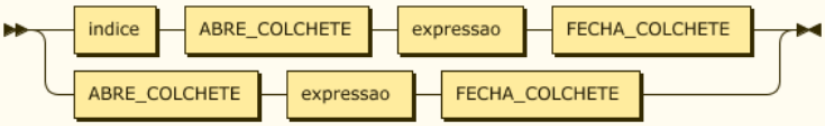



```
var      ::= ID
           | ID indice
```

referenced by:

- atribuicao
- fator
- leia
- lista_variaveis

indice:

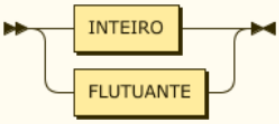



```
indice   ::= indice ABRE_COLCHETE expressao FECHA_COLCHETE
           | ABRE_COLCHETE expressao FECHA_COLCHETE
```

referenced by:

- indice
- var

tipo:



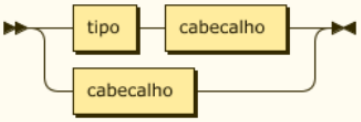

```
tipo     ::= INTEIRO
           | FLUTUANTE
```

referenced by:

- declaracao_funcao
- declaracao_variaveis
- parametro

Figura 6. Gramáticas reconhecidas

declaracao_funcao:




```
declaracao_funcao
         ::= tipo cabecalho
           | cabecalho
```

referenced by:

- declaracao

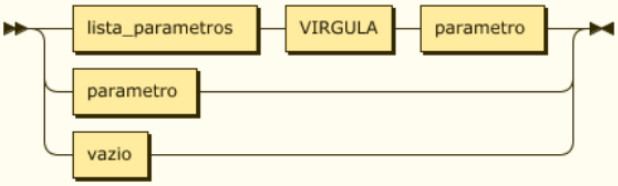
cabecalho:


```
cabecalho
         ::= ID ABRE_PARENTESE lista_parametros FECHA_PARENTESE corpo FIM
```

referenced by:

- declaracao_funcao

lista_parametros:

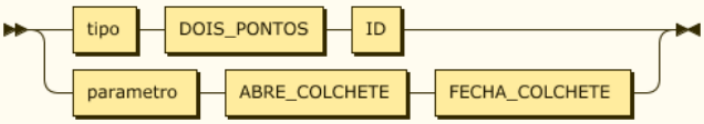



```
lista_parametros
         ::= lista_parametros VIRGULA parametro
           | parametro
           | vazio
```

referenced by:

- cabecalho
- lista_parametros

parametro:



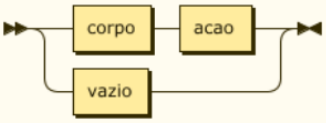

```
parametro
         ::= tipo DOIS_PONTOS ID
           | parametro ABRE_COLCHETE FECHA_COLCHETE
```

referenced by:

- lista_parametros
- parametro

Figura 7. Gramáticas reconhecidas

corpo:




```
corpo    ::= corpo acao
           | vazio
```

referenced by:

- cabecalho
- corpo
- repita
- se

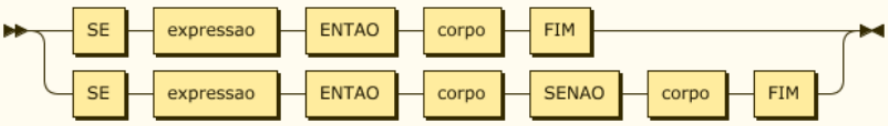
acao:


```
acao     ::= expressao
           | declaracao_variaveis
           | se
           | repita
           | leia
           | escreva
           | retorna
           | erro
```

referenced by:

- corpo

se:



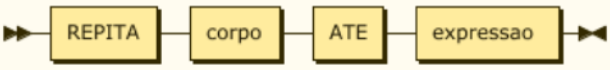

```
se       ::= SE expressao ENTAO corpo FIM
           | SE expressao ENTAO corpo SENAO corpo FIM
```

referenced by:

- acao

Figura 8. Gramáticas reconhecidas

repita:




```
repita   ::= REPITA corpo ATE expressao
```

referenced by:

- acao

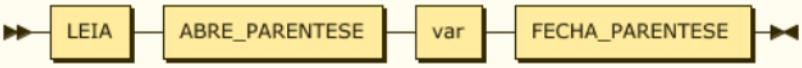
atribuicao:


```
atribuicao
         ::= var ATRIBUICAO expressao
```

referenced by:

- expressao
- inicializacao_variaveis

leia:

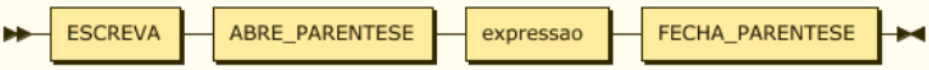



```
leia     ::= LEIA ABRE_PARENTESE var FECHA_PARENTESE
```

referenced by:

- acao

escreva:

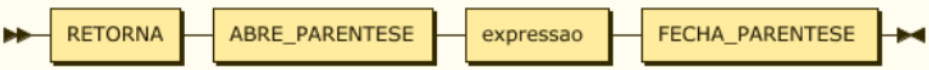



```
escreva  ::= ESCREVA ABRE_PARENTESE expressao FECHA_PARENTESE
```

referenced by:

- acao

retorna:



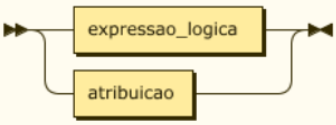

```
retorna  ::= RETORNA ABRE_PARENTESE expressao FECHA_PARENTESE
```

referenced by:

- acao

Figura 9. Gramáticas reconhecidas

expressao:

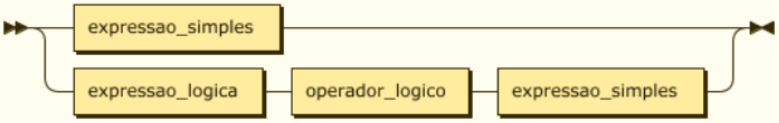



```
expressao
         ::= expressao_logica
           | atribuicao
```

referenced by:

- acao
- atribuicao
- escreva
- fator
- indice
- lista_argumentos
- repita
- retorna
- se

expressao_logica:

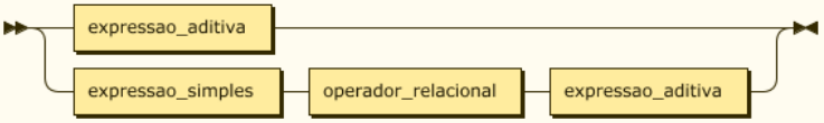



```
expressao_logica
         ::= expressao_simples
           | expressao_logica operador_logico expressao_simples
```

referenced by:

- expressao
- expressao_logica

expressao_simples:

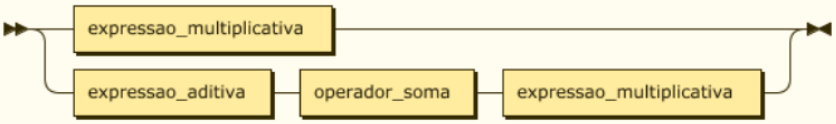



```
expressao_simples
         ::= expressao_aditiva
           | expressao_simples operador_relacional expressao_aditiva
```

referenced by:

- expressao_logica
- expressao_simples

expressao_aditiva:



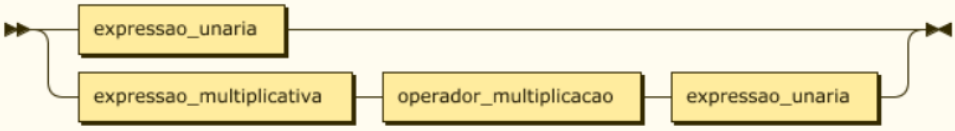

```
expressao_aditiva
         ::= expressao_multiplicativa
           | expressao_aditiva operador_soma expressao_multiplicativa
```

referenced by:

- expressao_aditiva
- expressao_simples

Figura 10. Gramáticas reconhecidas

expressao_multiplicativa:

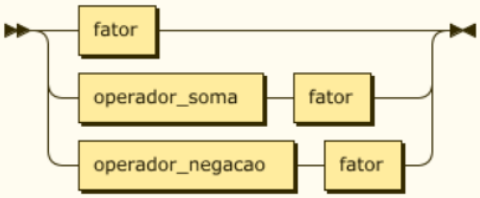



```
expressao_multiplicativa
         ::= expressao_unaria
           | expressao_multiplicativa operador_multiplicacao expressao_unaria
```

referenced by:

- expressao_aditiva
- expressao_multiplicativa

expressao_unaria:

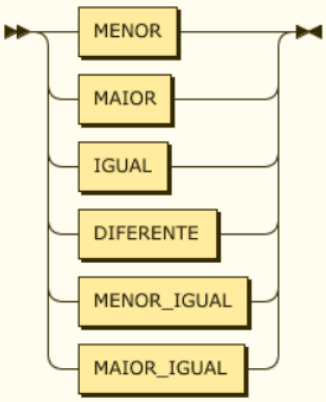



```
expressao_unaria
         ::= fator
           | operador_soma fator
           | operador_negacao fator
```

referenced by:

- expressao_multiplicativa

operador_relacional:



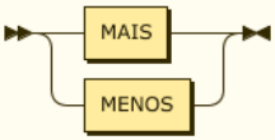

```
operador_relacional
         ::= MENOR
           | MAIOR
           | IGUAL
           | DIFERENTE
           | MENOR_IGUAL
           | MAIOR_IGUAL
```

referenced by:

- expressao_simples

Figura 11. Gramáticas reconhecidas

operador_soma:

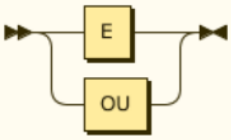



```
operador_soma
         ::= MAIS
           | MENOS
```

referenced by:

- expressao_aditiva
- expressao_unaria

operador_logico:




```
operador_logico
         ::= E
           | OU
```

referenced by:

- expressao_logica

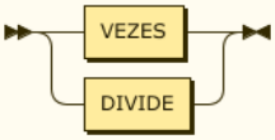
operador_negacao:


```
operador_negacao
         ::= NAO
```

referenced by:

- expressao_unaria

operador_multiplicacao:



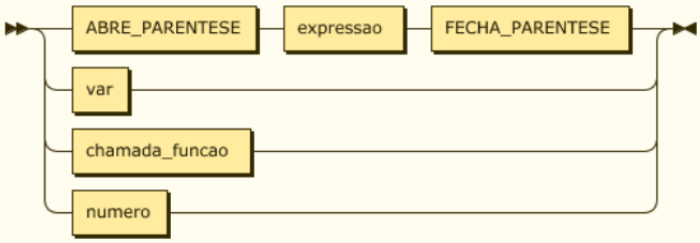

```
operador_multiplicacao
         ::= VEZES
           | DIVIDE
```

referenced by:

- expressao_multiplicativa

Figura 12. Gramáticas reconhecidas

fator:

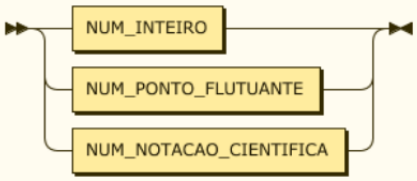



```
fator    ::= ABRE_PARENTESE expressao FECHA_PARENTESE
           | var
           | chamada_funcao
           | numero
```

referenced by:

- expressao_unaria

numero:




```
numero   ::= NUM_INTEIRO
           | NUM_PONTO_FLUTUANTE
           | NUM_NOTACAO_CIENTIFICA
```

referenced by:

- fator

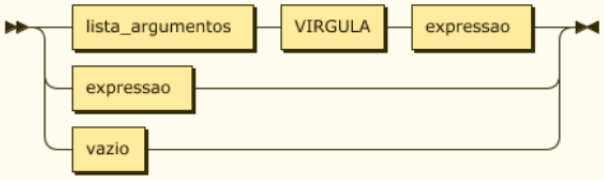
chamada_funcao:


```
chamada_funcao
         ::= ID ABRE_PARENTESE lista_argumentos FECHA_PARENTESE
```

referenced by:

- fator

lista_argumentos:




```
lista_argumentos
         ::= lista_argumentos VIRGULA expressao
           | expressao
           | vazio
```

referenced by:

- chamada_funcao
- lista_argumentos

2.2. Verificação de erros

Ao se realizar a análise sintática, podemos verificar se há algum erro na utilização dos *tokens* e retorná-lo para o usuário. Os erros também são reconhecidos por GLCs, nas quais avaliam se há incidências de erros nos símbolos terminais e sinalizam a linha que contém o erro.

A figura 13 demonstram duas regras de erros, uma sendo chamada de "p_error", sendo uma função genérica na qual identifica a ocorrência de um erro e imprime no terminal a linha, coluna e token próximo ao problema. A regra "p_parametro_error" é uma função específica, na qual reconhece algum erro durante a produção de parâmetro, imprimindo na tela uma mensagem de "Erro na definição da lista de parâmetros".

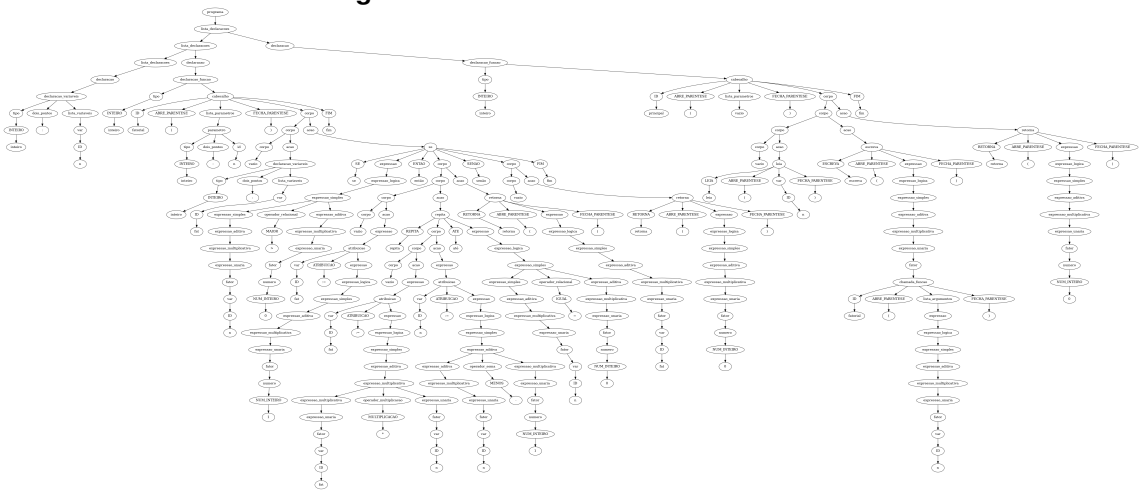
Figura 13. GLC indice_error

```
798  def p_parametro_error(p):
799      """parametro : tipo error ID"""
800      global error_code
801
802      if error_code ==  1:
803          print("Erro na definicao da lista de parametros")
804          error_code = -1
805      p.error()
806
807  def p_error(p):
808      global error_code
809
810      if error_code == 0 and p:
811          token = p
812          print("Erro:[{line},{column}]: Erro próximo ao token '{token}' ".format(
813              line=token.lineno, column=token.lexpos, token=token.value))
814          error_code = 1
```

3. Resultados

Com a implementação do analisador sintático espera que a saída dele contenha a árvore sintática, na qual preenche os nós com os símbolos não terminais que são reconhecidos pelas GLCs que resultam em símbolos terminais nas folhas da árvore. A figura 14 demonstra o resultado de uma análise sobre o código do fatorial.

Figura 14. Árvore sintática do Fatorial



4. Considerações Finais

A implementação do analisador sintático demonstra como é verificada a ordem de uso dos *tokens* definidos pelo analisador léxico, basta criar regras gramaticais que descrevem os uso deles.

A árvore sintática auxilia no reconhecimento da ordem de execução do código caso se observe os nós folhas, lendo-os da esquerda para a direita. Possibilitando a implementação do analisador semântico.

A maior dificuldade encontrada foi definir os casos de erros, pois algumas regras gramaticais não possibilitam o reconhecimento de todos os erros, sendo necessário realizar mais tratativas durante a análise semântica.

Referências

[Beazley] Beazley, D. M. Ply (python lex-yacc). `http://www.dabeaz.com/ply/ply.html`. Acessado: 2021-07-01.

[Gonçalves] Gonçalves, R. A. Bnf comentada. `https://docs.google.com/document/d/1oYX-5ipzL_izj_hO8s7axuo2OyA279YEhnAItgXzXAQ/edit`. Acessado: 2021-07-01.