Relatório Sintática

Introdução

Para a verificação da gramática a ser utilizada pela linguagem T**++** foi-se implementado o analisador sintático, no qual verifica a ordem (estrutura) dos tokens disponíveis, resultando uma Árvore Sintática Abstrata (ASA) e verificando erros de implementação do arquivo .tpp

Descrição BNF

A gramática BNF foi disponibilizada pelo professor que ministra a disciplina, Rogério A. Gonçalves, e pode ser visualizada em:

ebnf-tpp-symbols.odt

À esquerda da lista temos a regra e a esquerda os nós terminais ou não terminais que são gerado e necessários para se compreender as regras gramaticais da linguagem.

Yacc Ply

A implementação da gramática utilizou uma biblioteca disponibilizada no gerenciador de pacotes do Python (pip), no qual simplifica o reconhecimento das regras léxicas e sintáticas, por estabelecer uma classe de objetos que possui funções prontas para a implementação do compilador.

A regras gramaticais são definidas por uma string organizadas como no arquivo sobre as regras da BNF.

A biblioteca então divide cada token e ID reconhecido como um elemento do vetor, iniciando no índice **0**.

```
47    def p_programa(p):
48        "programa : lista_declaracoes"
49        global root
50        p[0] = MyNode("programa", children = [p[1]])
51        root = p[0]
52
53
54    def p_lista_declaracoes(p):
55        """lista_declaracoes : lista_declaracoes declaracao
56                             | declaracao
57        """
58        if len(p) > 2:
59            p[0] = MyNode ('lista_declaracoes', children = [p[1], p[2]]
60        else:
61            p[0] = MyNode('lista_declaracoes', children = [p[1]])
62
```

Implementação

Para a implementação da ASA foi criada uma classe no qual gera o nó de cada token reconhecido.

Classe de nó

Derivada da classe Node da biblioteca AnyTree, herdando assim o nome do nó e o pai (parent) da classe. Para facilitar a implementação do código, foi adicionado um parâmetro a mais, o children (filho), e um campo de id caso seja necessário durante o desenvolvimento do compilador.

```
17  # Classe de nó
18  class MyNode(Node):
19      def __init__(self, typeN,  parent = None, children = None):
20          super(Node, self).__init__()
21          global id
22          self.name = str(typeN)
23          self.id = id
24          self.parent = parent
25          if children:
26              self.children = children
27          # self.value = value
28
29
30          id += 1
31      def __str__(self):
32          return (self.name)
33
```

Relacionando os filhos

Após definir o nó e adicionar as strings da BNF, é necessário relacionar os nós conforme são reconhecidos pelo analisador sintático e atribuir os valores nos nós folhas.

Para compreender melhor, veja o exemplo da regra abaixo:

```
72  # Sub-árvore.
73  #       (declaracao_variaveis)
74  #       / p[1]    |          \
75  # (tipo)    (DOIS_PONTOS)    (lista_variaveis)
76  #                 |
77  #               (:)
78
79  def p_declaracao_variaveis(p):
80      "declaracao_variaveis : tipo DOIS_PONTOS lista_variaveis"
81      dp = MyNode("DOIS_PONTOS", children=[MyNode(p[2])])
82      p[0] = MyNode('declaracao_variaveis', children=[p[1], dp, p[3]])
83
```

A estrutura da árvore deve ser como descrito nos comentários entre as linhas **72** e **77.** Porém o DOIS_PONTOS gerado, é um símbolo terminal, assim então é necessário criar um valor com o Token DOIS_PONTOS e adicionar um nó folha a ele com o caractere que representa os dois pontos (":").

Já o nó pai de todos (declaracao_variaveis) deve ser relacionado aos seus filhos.

Saída

A biblioteca do AnyTree, possui funções de exportação da árvore para um arquivo de imagem ou um Dotfile. O arquivos finais se chamam tree.png e tree.dot.

A ASA resultante após a execução do código de teste "teste-1.tpp" é a seguinte:

 .png)

Execução

Para a execução do analisador sintático, acesse a pasta /implementacao/Sintatica e execute o arquivo parser.py da seguinte forma:

```
python3 parser.py ./sintatica-testes/nomedoarquivodeteste.tpp
```