Análise Sintática

Juan Felipe da S. Rangel[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 135 – 87.301-899 – Campo Mourão – PR – Brasil

juanrangel@alunos.utfpr.edu.br

Abstract. *The construction of a compiler is divided into four parts, lexical analysis, syntactic analysis, semantic analysis and code generation. During the course of this article, the process of building the parser, libraries used and the syntactic tree built at the end will be seen in more detail, as well as the grammar of the T++ language that will be used as a basis for the construction of the lexical analysis.*

Resumo. *A construção de um compilador é divida em quatro partes, análise léxica, análise sintática, análise semântica e geração de código. Durante o decorrer deste artigo, será visto em mais detalhes o processo da construção do analisador sintático, bibliotecas utilizadas e a árvore sintática construída ao final, assim como a gramática da linguagem T++ que será utilizada como base para construção da análise léxica.*

1. Introdução

Neste relatório foi executado o desenvolvimento de um analisador sintático para linguagem TPP, realizando apenas os tratamentos de erros, dado que um código de início foi disponibilizado pelo professor da disciplina, contendo todas as regras da gramática da linguagem. Em conjunto com o código de início foi também oferecido um conjunto de testes, para que sejam utilizados como casos de teste para certos erros. Esta implementação foi realizada utilizando a linguagem Python, além das bibliotecas Ply, em conjunto com o módulo Yacc, Tree, AnyTree e Graphviz.

2. Gramática da Linguagem T++ no padrão BNF

Uma gramática é criada com o objetivo de estabelecer regras de como uma sentença pode ser montada, verificando então dado um alfabeto,que neste caso é representado pelos tokens obtidos durante o processo de Análise Léxica, se as cadeias formadas estão sintaticamente corretas. Considerando como entrada os tokens utilizados em um código .tpp, a Análise Sintática como finalidade verificar se esses token foram utilizados na sequência correta, de acordo com a gramática criada para linguagem.
Algo importante para se lembrar é que na Análise Sintática não se obtém um significado, ou seja, não é possível verificar somente através da análise sintática se o código escrito está correto em relação a se ele executará corretamente, mas sim é possível dizer de acordo com as regras gramaticais se a estrutura das sentenças criadas está correta ou não. É possível acessar a todas as regras criadas para linguagem T++ [Gonçalves], ou observar a tabela a seguir (Tabela 1), e como poderá ser percebido, esta linguagem apenas suporta a utilização de poucas estruturas, porém estas são comumente utilizadas

em várias outras linguagens.

programa ::=	lista_declaracoes
lista_declaracoes ::=	lista_declaracoes declaracao $\|declaracao$
declaracao ::=	declaracao_variaveis $\|inicializacao\_variaveis$ $\|declaracao\_funcao$
declaracao_variaveis ::=	tipo ":"lista_variaveis
inicializacao_variaveis ::=	atribuicao
lista_variaveis ::=	lista_variaveis ","var $\|var$
var ::=	ID $\|IDindice$
indice ::=	indice "["expressao "]" $\|$"["$expressao$"]"
tipo ::=	INTEIRO $\|FLUTUANTE$
declaracao_funcao ::=	tipo cabecalho $\|cabecalho$
cabecalho ::=	ID "("lista_parametros ")"corpo FIM
lista_parametros ::=	lista_parametros ","parametro $\|parametro$ $\|vazio$
parametro ::=	tipo ":"ID $\|parametro$"[]"
corpo ::=	corpo acao $\|vazio$
acao ::=	expressao $\|declaracao\_variaveis$ $\|se$ $\|repita$ $\|leia$ $\|escreva$ $\|retorna$ $\|erro$
se ::=	SE expressao ENTAO corpo FIM $\|SEexpressaoENTAOcorpoSENAOcorpoFIM$
repita ::=	REPITA corpo ATE expressao
atribuicao ::=	var ":="expressao
leia ::=	LEIA "("var ")"
escreva ::=	ESCREVA "("expressao ")"
retorna ::=	RETORNA "("expressao ")"

expressao ::=	expressao_logica $\|atribuicao$
expressao_logica ::=	expressao_simples$\|expressao_logica$ operador_logico expressao_simples
expressao_simples ::=	expressao_aditiva $\|expressao\_simples operador\_relacional expressao\_aditiva$
expressao_aditiva ::=	expressao_multiplicativa $\|expressao\_aditiva operador\_soma expressao\_multiplicativa$
expressao_multiplicativa ::=	expressao_unaria $\|expressao\_multiplicativa operador$ _multiplicacao expressao_unaria
expressao_unaria ::=	fator $\|operador\_soma fator$ $\|operador\_negacao fator$
operador_relacional ::=	"<" $\|" > "$ $\|" = "$ $\|" < > "$ $\|" < = "$ $\|" > = "$
operador_soma ::=	"+" $\|\_"$
operador_logico ::=	"&&" $\|"\|\|"$
operador_negacao ::=	"!"
operador_multiplicacao ::=	"*" $\|"/"$
fator ::=	"("expressao ")" $\|var$ $\|chamada\_funcao$ $\|numero$
numero ::=	NUM_INTEIRO $\|NUM\_PONTO\_FLUTUANTE$ $\|NUM\_NOTACAO\_CIENTIFICA$
chamada_funcao ::=	ID "("lista_argumentos ")"
lista_argumentos ::=	lista_argumentos ","expressao $\|expressao$ $\|vazio$

Tabela 1. Tabela da Gramática T++ no padrão BNF.

3. Formato da Análise Sintática

Considerando que objetivo da aplicação do processo de Análise Sintática no nosso caso é verificar se uma cadeia de código fornecida como entrada é uma cadeia que pode ser definida pela gramática, e a construção de uma árvore sintática, é importante considerar

qual será o formato dessa análise, ou seja, qual método será utilizado para criação dessa árvore.
O método utilizado pelo YACC é o LALR(1), o qual é caracterizado como uma forma ligeiramente mais poderosa do que a análise SLR(1), e menos complexo comparado a análise LR(1), gerando também uma tabela de análise sintática menor, mas ainda necessita da computação do Automato Determinístico Finito inteiro. O motivo de conseguir criar menores tabelas está relacionado com o modo como a análise é realizada, que no caso do LALR(1), é feita da direita para esquerda, verificando a frente e realizando a combinações desses estados, resultando em menores conjuntos encontrados.

4. YACC (Yet another compiler-compiler)

O Yacc (yet another compiler-compiler) é utilizado para geração de analisadores sintáticos, que como já visto em **Formato da Análise Sintática**, utiliza o método de análise LALR(1). Ele é um módulo do ply, biblioteca a qual permite a criação das análises sintáticas e léxicas, utilizados nesta disciplina de Compiladores com o objetivo de criar um compilador para a linguagem T++.

4.1. Processo de tradução

Ao se falar do processo de tradução é importante ter em mente que na gramática podem haver símbolos terminais e não terminais, sendo os dois componentes utilizados para definição da gramática. O que chamamos de símbolo terminal pode ser representado por 'OU_LOGICO', &&', '<=', '+', '-', e todos estes símbolos são associados à um token. Já um identificador/símbolo não terminal está associado com a gramática da linguagem, como **'declaracao_variaveis'**. Nas imagens abaixo é possível observar os símbolos terminais e não terminais representados no código **'tppparser.py'**, na regra de identificação de uma ação.

Listing 1. Regra de indentificação de uma ação.

```
1
2  def p_acao(p):
3      """acao : expressao
4          | declaracao_variaveis
5          | se
6          | repita
7          | leia
8          | escreva
9          | retorna
10     """
11     pai = MyNode(name='acao', type='ACAO')
12     p[0] = pai
13     p[1].parent = pai
```

Podemos notar também que os símbolos não terminais foram representados de três formas diferentes nesses três casos, nos quais **'acao'** consiste somente de símbolos não terminais (expressao, declaracao_variaveis, se, repita, leia, escreva e retorna).

Listing 2. Função para identificação da regra SE.

```
1  def p_se(p):
```

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

    pai = MyNode(name='se', type='SE')
    p[0] = pai

    filho1 = MyNode(name='SE', type='SE', parent=pai)
    filho_se = MyNode(name=p[1], type='SE', parent=filho1)
    p[1] = filho1

    p[2].parent = pai

    filho3 = MyNode(name='ENTAO', type='ENTAO', parent=pai)
    filho_entao = MyNode(name=p[3], type='ENTAO', parent=filho3)
    p[3] = filho3

    p[4].parent = pai

    if len(p) == 8:
        filho5 = MyNode(name='SENAO', type='SENAO', parent=pai)
        filho_senao = MyNode(name=p[5], type='SENAO', parent=filho5)
        p[5] = filho5

        p[6].parent = pai

        filho7 = MyNode(name='FIM', type='FIM', parent=pai)
        filho_fim = MyNode(name=p[7], type='FIM', parent=filho7)
        p[7] = filho7
    else:
        filho5 = MyNode(name='fim', type='FIM', parent=pai)
        filho_fim = MyNode(name=p[5], type='FIM', parent=filho5)
        p[5] = filho5
```

Já o símbolo não terminal **'se'** é composto pelos símbolos terminais (SE, ENTAO, SENAO e FIM) e não terminais (expressao e corpo).

Listing 3. Lista de tokens para palavras reservadas.

```
def p_tipo(p):
    """tipo : INTEIRO
        | FLUTUANTE
    """

    pai = MyNode(name='tipo', type='TIPO')
    p[0] = pai
    # p[1] = MyNode(name=p[1], type=p[1].upper(), parent=pai)

    if p[1] == "inteiro":
        filho1 = MyNode(name='INTEIRO', type='INTEIRO', parent=pai)
        filho_sym = MyNode(name=p[1], type=p[1].upper(), parent=filho1)
        p[1] = filho1
    else:
        filho1 = MyNode(name='FLUTUANTE', type='FLUTUANTE', parent=pai)
        filho_sym = MyNode(name=p[1], type=p[1].upper(), parent=filho1)
```

E por fim temos o símbolo não terminal **'tipo'**, o qual consiste unicamente de símbolos terminais (INTEIRO e FLUTUANTE).

Os símbolos não terminais nos códigos acima são: **'acao'** (Listing 1), **'se'** (Listing 2) e **'tipo'** (Listing 3), como já apresentado anteriormente. Isso se dá pois, a partir deles ainda é possível realizar uma derivação para outros símbolos.

A técnica de tradução direcionada à sintaxe é frequentemente utilizada para definir o comportamento de uma linguagem, na qual cada atributo é associada com os símbolos considerando uma dada regra gramatical, e cada regra gramatical pode ser associada a uma ação. Um método será chamada assim que uma regra gramatical for reconhecida, executando então aquele método em seguida.

Listing 4. Identificação da regra vazia.

```
1
2  def p_vazio(p):
3      """vazio : """
4
5      pai = MyNode(name='vazio', type='VAZIO')
6      p[0] = pai
```

Caso alguma exceção ao realizar a análise sintática aconteça, o YACC consegue tratar, criando regras para identificação de erros e regras vazias (Listing 4 e 5).
Como é possível observar abaixo, cada regra da linguagem T++ está associada a um método, e o parâmetro 'p' recebido em todos esses métodos representa os valores associados aos símbolos terminais ou não terminais definidos na gramática.

Listing 5. Identificação da regra vazia.

```
1  def p_error(p):
2
3      if p:
4          token = p
5          print("Erro:[{line},{column}]: Erro proximo ao token '{token}'".format(
6              line=token.lineno, column=token.lineno, token=token.value))
```

5. Implementação da Árvore Sintática

Para implementação da construção da Árvore Sintática foram utilizadas as bibliotecas anytree, Tree e graphviz, em conjunto com o código providenciado pelo professor. A imagem a seguir representa o arquivo "mytree.py", que tem como função criar os nós da árvore sintática.

Listing 6. Código que cria os nós da árvore.

```
1  node_sequence = 0
2
3  class MyNode(NodeMixin):  # Add Node feature
4
5    def __init__(self, name, parent=None, id=None, type=None, label=None, children=None):
6      super(MyNode, self).__init__()
7      global node_sequence
8
9      if (id):
10       self.id = id
11     else:
12       self.id = str(node_sequence) + ':' + str(name)
13
14     self.label = name
15     # self.name = name + '_' + str(node_sequence)
16     self.name = name
17     node_sequence = node_sequence + 1
18     self.type = type
19     self.parent = parent
20     if children:
21       self.children = children
22
23   def nodenamefunc(node):
24     return '%s' % (node.name)
```

```
25
26    def nodeattrfunc(node):
27      return '%s' % (node.name)
28
29    def edgeattrfunc(node, child):
30      # return 'label="%s:%s"' % (node.name, child.name)
31      return ''
32
33    def edgetypefunc(node, child):
34      return '--'
```

Ao detectar uma regra gramatical, é criado um nó pai para o símbolo não terminal que representa a regra da linguagem, já para sua derivação caso o símbolo reconhecido for terminal, ou seja, não é possível aplicar mais nenhuma derivação, um nó filho será criado, já caso encontre um símbolo não terminal, o nó pai dessa regra, virará o pai desse símbolo, que será reconhecido em outra regra, ou seja, o processo de derivação continuará acontecendo, repetindo então o processo para criação da árvore até que seus símbolos não terminais sejam encontrados.

6. Exemplos de entrada e saída

Como exemplo de código de entrada será utilizado o código abaixo (Listing 7), e como é possível observar, não há nenhum erro gramatical, ou seja, deve ser possível criar a árvore sintática sem nenhum problema.

Listing 7. Lista de tokens para palavras reservadas.

```
1  inteiro: x
2  inteiro: y[10]
3
4  inteiro principal()
5      leia(x)
6  fim
```

Ao fim da execução da análise sintática vamos obter como resultado 3 árvores. uma delas conterá todos os valores únicos para cada ramificação, podendo ser observado na Imagem 1, e as outras duas consistem em todos os valores únicos porém não haverá repetições nos galhos da árvore, representadas nas Imagem 2 e Imagem 3. As árvores geradas abaixo são correspondentes ao código a seguir (listing 7).

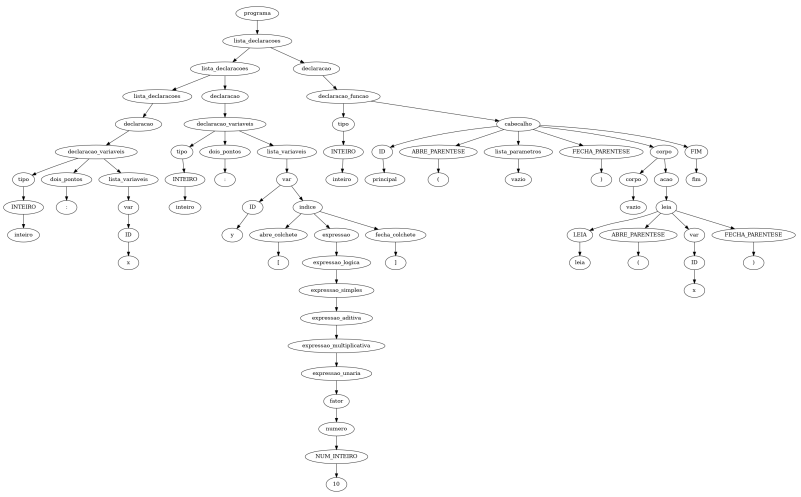

Figura 1. Árvore com todos os valores únicos para cada ramificação.

A primeira dela contem todos os valores únicos para cada ramificação, podendo ser observado acima na Imagem 1

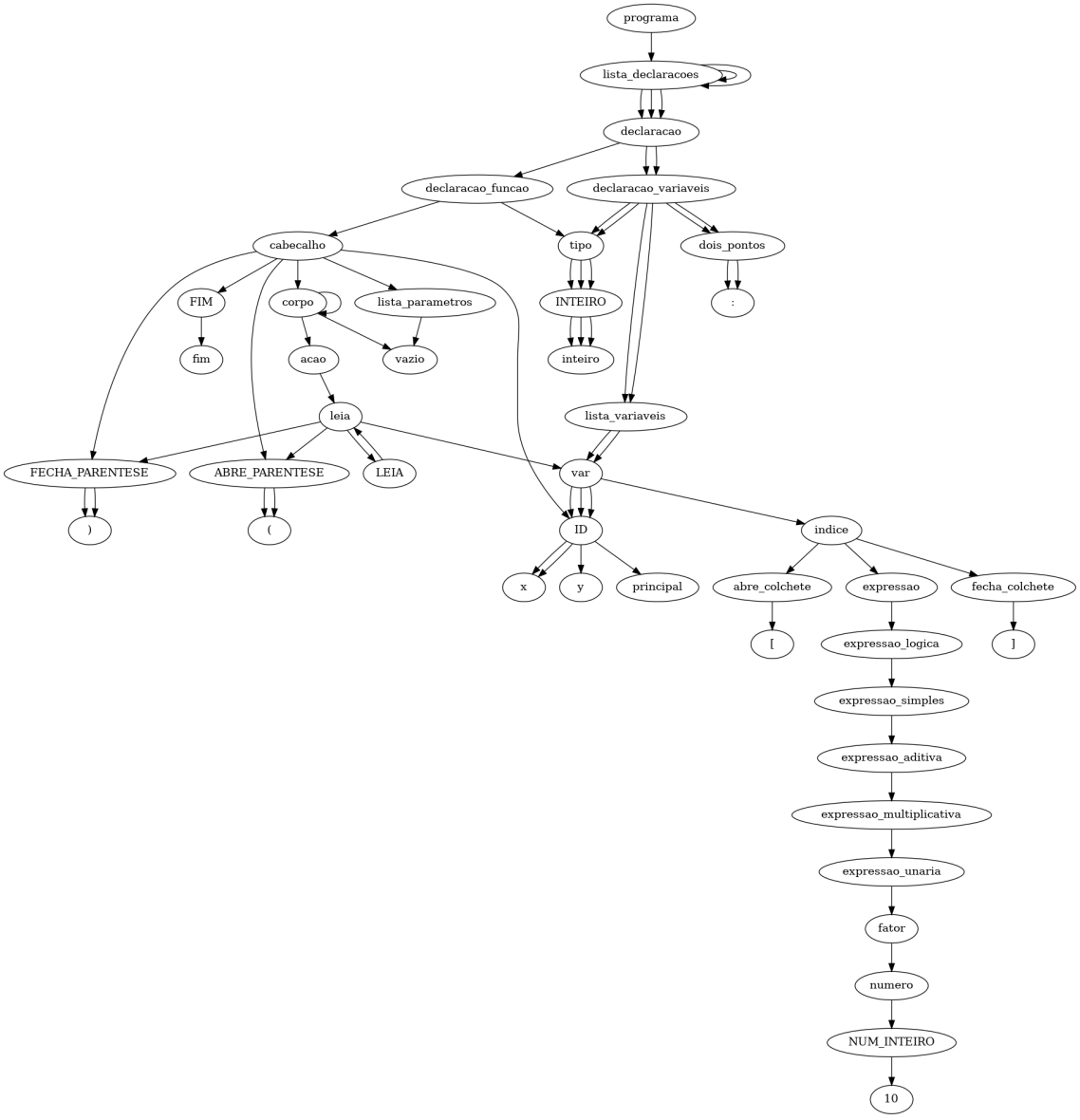

Figura 2. Árvore com valores únicos mas sem repetições nas ramificações.

Já as outras duas consistem em todos os valores únicos, porém não haverá repetições nos galhos da árvore, representadas nas Imagem 2 e Imagem 3.

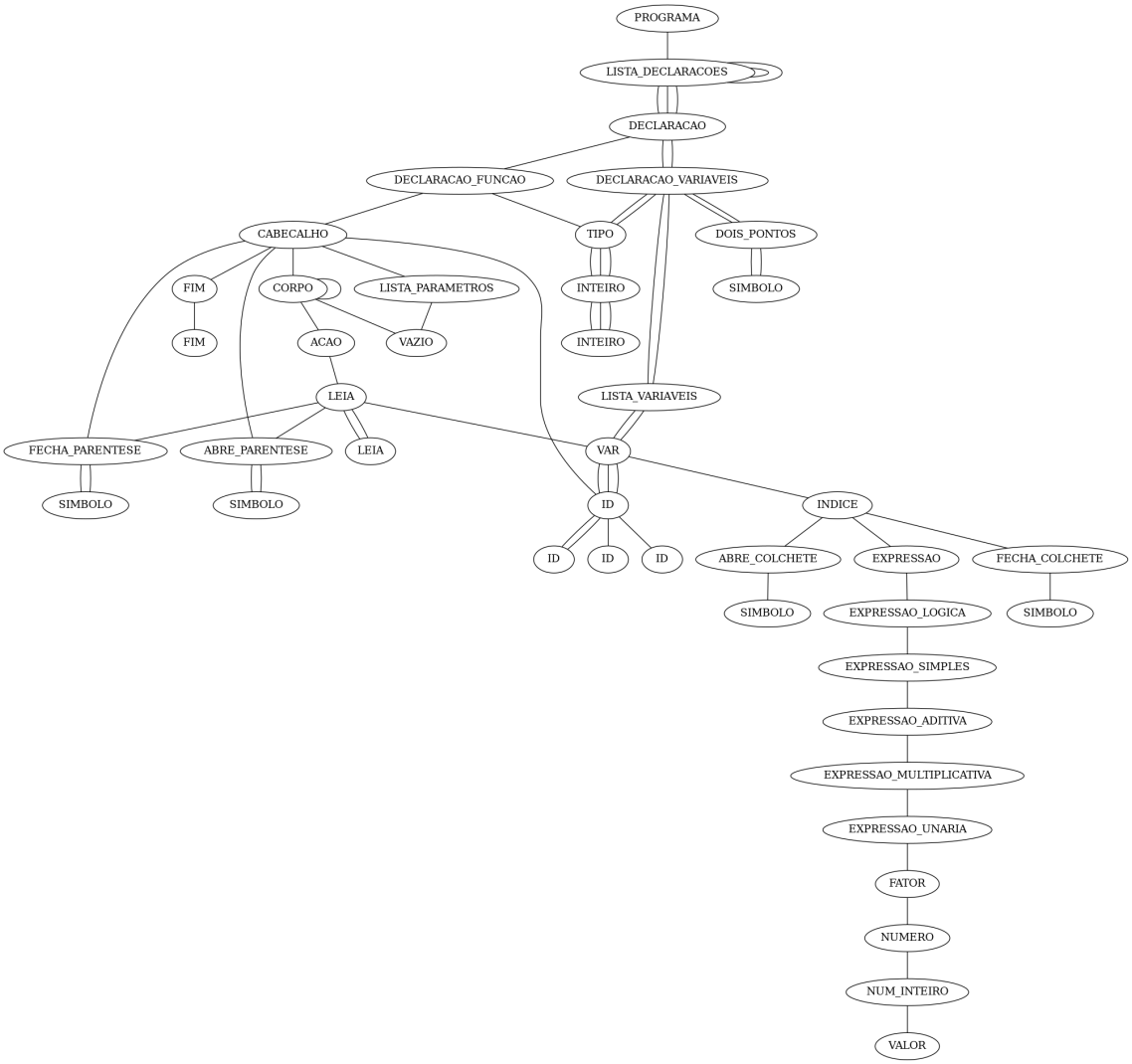

Figura 3. Segunda árvore com valores únicos mas sem repetições nas ramificações.

Referências

Beazley, D. M. Ply (python lex-yacc). ttps://www.dabeaz.com/ply/ply.tml.

Gonçalves, R. A. Gramática da linguagem t++. ttps://docs.google.com/document/d/1oYX-5ipzL_izj_O8s7axuo2OyA279YEhnAItgXzXAQ/edit.

Gonçalves, R. A. (2021). Análise sintática geral lr(1) e lalr(1). ttps://moodle.utfpr.edu.br/pluginfile.pp/260510/mod$_r esource/content/0/01021500.PDF$.

Louden, K. C. Compiladores: princípios e práticas. ttps://integrada.minabiblioteca.com.br/reader/books/9788522128532/pageid/213.