Compiladores: Implementação e explicação sobre um compilador para a linguagem T++: Análise Sintática.

Matheus Sapia Guerra

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal: 271 Campo Mourão - PR - Brasil

guerramatheus2@gmail.com

Abstract. *This document aims to record and detail the processes of implementing a compiler. The language to be compiled is called T++ in which it contains all the attributes of a conventional programming language. The process of compilation consists of translating a high level language into an intermediary. For this, the process is divided into four parts, lexical analysis, syntactic analysis, semantic analysis and finally the generation of code.*

Resumo. *Este documento tem por objetivo registrar e detalhar os processos de implementação de um compilador. A linguagem a ser compilada é denominada de T++ na qual contém todos os atributos de uma linguagem de programação convencional. O processo de compilação consiste em traduzir uma linguagem de alto nível para uma intermediária, para isso, tal processo será dividido em quatro partes, análise léxica, análise sintática, análise semântica e por último a geração de código.*

1. Introdução

Em posse de um programa de varredura implementado e gerando os tokens, o próximo passo é realizar a análise sintática ou parsing. A análise sintática consiste em determinar a estrutura de um programa, dada geralmente pelas regras gramaticais de uma gramática livre de contexto. E por fim gerar uma árvore sintática. A biblioteca *ply* implementa tanto a parte léxica quanto a sintática, neste artigo será explicado detalhes da biblioteca ply[Beazley 2017] e detalhes de implementação de um análisador sintático.

2. Fundamentação Teórica

Nesta seção será apresentado a fundamentação teorica de elementos e conceitos utilizados em uma análise sintática.

Primeiramente é necessário ter um entendimento básico em gramáticas livre de contexto, tal como a descrição da gramática utilizada. Será abordado também conceitos utilizados pelo ply, tal como *botton-up*, LALR(1).

2.1. Grámatica Livre de Contexto

A gramática livre de contexto (GLC) é um meio mais poderoso que expressões regulares pois possuem regras recursivas, desde os anos 60, são elemento chave para a construção de compiladore. Uma GLC é formada por alguns elementos, tal como:

- Um conjunto de terminais

- Um conjunto de não-terminais
- Um não-terminal inicial (S)
- Um conjunto de produções (P)
- Linguagem denotada pela gramática G: L(G)

Uma GLC é básicamente uma regra de produção que segue a forma: $A \rightarrow \alpha$ Onde o A é um não terminal e o outro lado é a regra da cadeia. [Hopcroft and Ullman 1979]

2.2. Descrição da Gramática no Padrão BNF

Foi utilizada uma BNF[Gonçalves 2017] pré definida em sala de aula para a gramatica da linguagem TPP, a BNF é:

Código 1. Regras

```
programa : lista_declaracoes

lista_declaracoes : lista_declaracoes declaracao

lista_declaracoes : declaracao

declaracao : declaracao_variaveis
     | inicializacao_variaveis
     | declaracao_funcao

declaracao_variaveis : tipo DOISPONTOS lista_variaveis

inicializacao_variaveis : atribuicao

lista_variaveis : lista_variaveis VIRGULA var

lista_variaveis : var

var : ID

var : ID indice

indice : indice ABRECOLCH expressao FECHACOLCH

indice : ABRECOLCH expressao FECHACOLCH

tipo : INTEIRO

tipo : FLUTUANTE

declaracao_funcao : tipo cabecalho

declaracao_funcao : cabecalho

cabecalho : ID ABREPAR lista_parametros FECHAPAR corpo FIM

lista_parametros : lista_parametros VIRGULA lista_parametros
```

```
38
39 lista_parametros : parametro
40             | vazio
41
42 parametro : tipo DOISPONTOS ID
43
44 parametro : parametro ABRECOLCH FECHACOLCH
45
46 corpo : corpo acao
47
48 corpo : vazio
49
50 acao : expressao
51      | declaracao_variaveis
52      | se
53      | repita
54      | leia
55      | escreva
56      | retorna
57      | error
58
59 se : SE expressao ENTAO corpo FIM
60
61 se : SE expressao ENTAO corpo SENAO corpo FIM
62
63 repita : REPITA corpo ATE expressao
64
65 atribuicao : var ATRIBUICAO expressao
66
67 leia : LEIA ABREPAR ID FECHAPAR
68
69 escreva : ESCREVA ABREPAR expressao FECHAPAR
70
71 retorna : RETORNA ABREPAR expressao FECHAPAR
72
73 expressao : expressao_simples
74        | atribuicao
75
76 expressao_simples : expressao_aditiva
77
78 expressao_simples : expressao_simples operador_relacional
      expressao_aditiva
79
80 expressao_aditiva : expressao_multiplicativa
81
82 expressao_aditiva : expressao_aditiva operador_multiplicacao
      expressao_unaria
83
84 expressao_multiplicativa : expressao_unaria
```

```
85
86 expressao_multiplicativa : expressao_multiplicativa
       operador_multiplicacao expressao_unaria
87
88 expressao_unaria : fator
89
90 expressao_unaria : operador_soma fator
91
92 operador_relacional : MENOR
93             | MAIOR
94             | IGUAL
95             | DIFERENTE
96             | MENORIGUAL
97             | MAIORIGUAL
98             | ELOGICO
99             | OULOGICO
100
101 operador_soma : SOMA
102             | SUBR
103
104 operador_multiplicacao : VEZES
105                | DIVIDE
106
107 fator : ABREPAR expressao FECHAPAR
108
109 fator : var
110    | chamada_funcao
111    | numero
112
113 numero : INTEIRO
114    | FLUTUANTE
115
116 chamada_funcao : ID ABREPAR lista_argumentos FECHAPAR
117
118 lista_argumentos : lista_argumentos VIRGULA expressao
119
120 lista_argumentos : expressao
121           | vazio
122 vazio :
```

2.3. Formato na Análise Sintática utilizando PLY

O PLY é uma implemtação em Python de uma ferramenta para geração de compiladores. A ferramenta PLY da suporte a análise LALR(1) e tem abordagem *botton-up*. A análise do LR é uma técnica *botton up*, que tenta reconhecer o lado direita das várias regras de gramática. Sempre que um lado direito válido for encontrado na entrada, o código de ação apropriado é acinado e os simbolos de gramatica são reposto pela gramatica do lado esquerdo da regra. A análise de LR é comumente implementada mudando símbolos de gramática para uma pilha e olhando a pilha e o próximo token de entrada para padrões que

correspondem a uma das regras de gramática. A Figura exemplificao o que foi descrito.

```
Step Symbol Stack           Input Tokens            Action
---- ---------------------  ---------------------   ----------------------------
1                           3 + 5 * ( 10 - 20 )$    Shift 3
2    3                        + 5 * ( 10 - 20 )$    Reduce factor : NUMBER
3    factor                   + 5 * ( 10 - 20 )$    Reduce term   : factor
4    term                     + 5 * ( 10 - 20 )$    Reduce expr : term
5    expr                     + 5 * ( 10 - 20 )$    Shift +
6    expr +                     5 * ( 10 - 20 )$    Shift 5
7    expr + 5                     * ( 10 - 20 )$    Reduce factor : NUMBER
8    expr + factor                * ( 10 - 20 )$    Reduce term   : factor
9    expr + term                  * ( 10 - 20 )$    Shift *
10   expr + term *                  ( 10 - 20 )$    Shift (
11   expr + term * (                  10 - 20 )$    Shift 10
12   expr + term * ( 10                  - 20 )$    Reduce factor : NUMBER
13   expr + term * ( factor              - 20 )$    Reduce term : factor
14   expr + term * ( term                - 20 )$    Reduce expr : term
15   expr + term * ( expr                - 20 )$    Shift -
16   expr + term * ( expr -                20 )$    Shift 20
17   expr + term * ( expr - 20                )$    Reduce factor : NUMBER
18   expr + term * ( expr - factor            )$    Reduce term : factor
19   expr + term * ( expr - term              )$    Reduce expr : expr - term
20   expr + term * ( expr                     )$    Shift )
21   expr + term * ( expr )                    $    Reduce factor : (expr)
22   expr + term * factor                      $    Reduce term : term * factor
23   expr + term                               $    Reduce expr : expr + term
24   expr                                      $    Reduce expr
25                                             $    Success!
```

Figura 1. Exemplo de parse para 3 + 5 * (10 - 20)

3. Materiais

Foram utilizados como materiais o *Sublime text* com a função de editor de textos, Python como linguagem de programação, códigos de exemplos do PLY e uma EBNF pré definida da linguagem TPP.

4. Implementação

Nesta seção será abordado assuntos referente a implementação da análise sintática, considerando que a análise léxica ja esteja implementada.

4.1. Utilização da ferramenta YACC

A ferramenta YACC implementada pela biblioteca PLY foi utilizada baseando-se nos exemplos da documentação da ferramenta [Beazley 2017] e uma implementação de um calendoscópio do Prof. Dr Rodrigo Hubner[Hübner 2016].

O código se define em algumas estruturas, onde a primeira delas é uma classe *Tree*, que tem como função implementar a árvore sintática. O trecho a seguir mostra a implementação.

Código 2. class Tree

```
1 class Tree:
2
3    def __init__(self, type_node, child=[], value=None):
4       self.type = type_node
5       self.child = child
6       self.value = value
7
8    def __str__(self):
9       return self.type
```

A segunda classe dó código se refere ao parser em si, onde é chamada a análise léxica e a partir disto e das regras definidas e realiza a análise sintática. Segue um exemplo de código.

Código 3. class Parser

```
class Parser:

  def __init__(self, code):
    lex = Lexer()
    self.tokens = lex.tokens
    self.precedence = (
      ('left', 'IGUAL', 'MAIOR', 'MENOR', 'MAIORIGUAL',
         'MENORIGUAL', 'DIFERENTE'),
      ('left', 'SOMA', 'SUBR'),
      ('left', 'VEZES', 'DIVIDE'),
      ('left', 'ABREPAR', 'FECHAPAR')
    )
    parser = yacc.yacc(debug=True, module=self, optimize=False)
    self.ast = parser.parse(code)


  def p_programa(self, p):
    '''
    programa : lista_declaracoes
    '''
    p[0] = Tree('programa',[p[1]])


  def p_lista_declaracoes(self, p):
    '''
    lista_declaracoes : lista_declaracoes declaracao

    '''
    p[0] = Tree('lista_declaracoes',[p[1], p[2]])
  def p_lista_declaracoes1(self,p):
    '''
    lista_declaracoes : declaracao
    '''
    p[0] = Tree('lista_declaracoes1',[p[1]])

      .
      .
      .
```

4.2. Utilização da Árvore Sintática

Foram implementados dois métodos que imprime a árvore gerada a partir da análise sintática, um que imprime no terminal e outro que gera o dot para a visualização no graphviz, segue os códigos abaixo.

Código 4. Metodos de impressão

```
def print_arvore(node, level="-"):
  if node != None:
    if node.value != None:
      print("%s %s %s" %(level, node.type, node.value))
    else:
      print("%s %s" %(level, node.type))
    for son in node.child:
      print_arvore(son, level+"-")

def graph(node,w,i):
   if node != None:
      value = node.type + str(i)
      i = i + 1
      for son in node.child:
       w.edge(value,str(son) + str(i))
       graph(son,w,i)
```

Essa árvore será utilizada posteriormente na terceira etapa da construção do compilador, que é a análise semântica.

5. Resultados

Como resultado obtemos a árvore sintática e com a implementação de impressão do dot podemos visualizar a árvore gerada. Segue abaixo uma exemplo de árvore gerada para o código de soma de vetores em Tpp:

Código 5. Código de teste Somavet.tpp

```
inteiro: T
T:= 4

inteiro: V1[T]

inteiro somavet(inteiro: vet[], inteiro: tam)
  inteiro: result
  result := 0

  inteiro: i
  i := 0

  repita
    result := result + vet[i]
    i := i + 1
  até i = tam - 1

  retorna(result)
fim

inteiro principal ()
```

```
22   inteiro: x
23   x := somavet(V1,T)
24   retorna(0)
25 fim
```

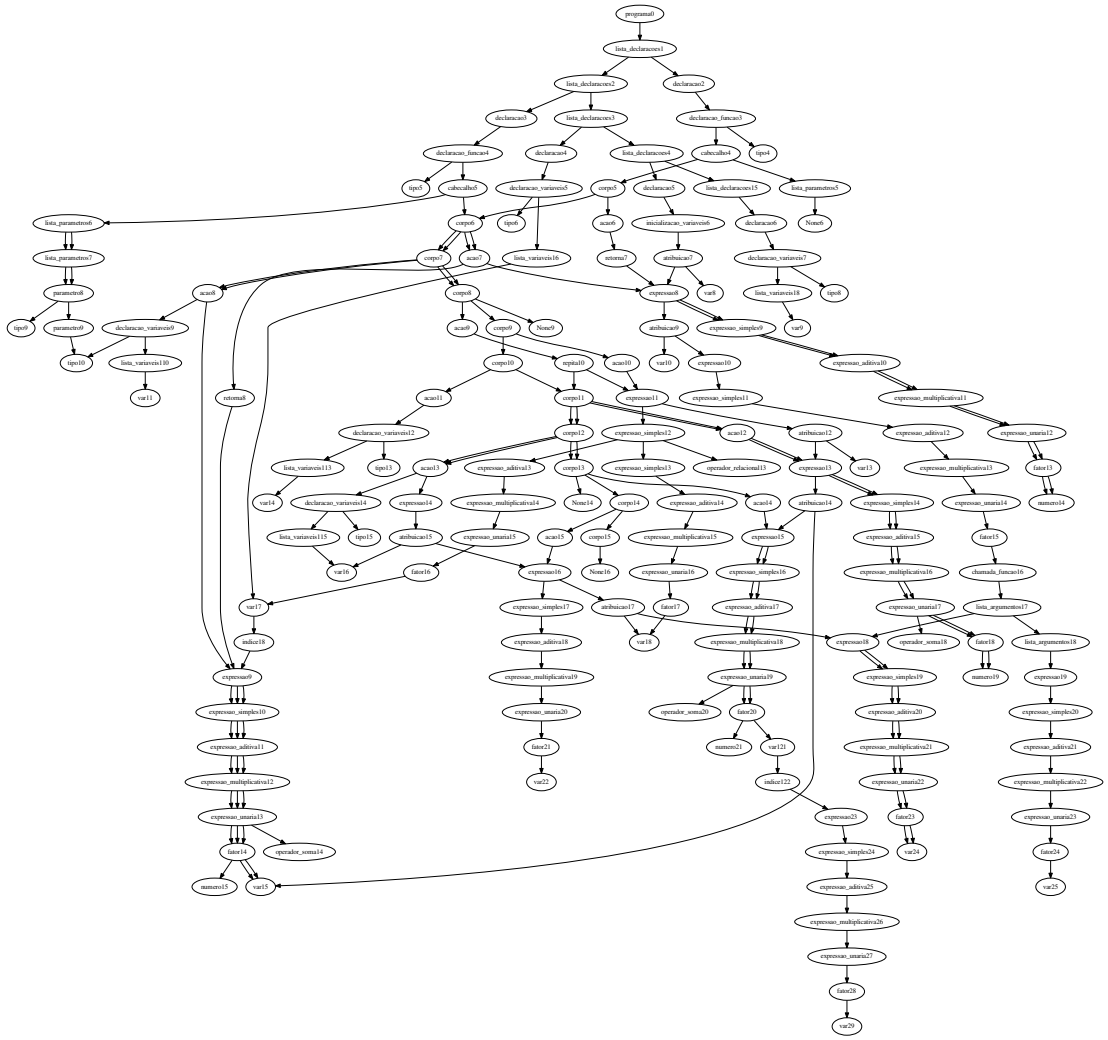

Figura 2. Arvore gerada para o código de soma de vetores

Referências

Beazley, D. M. (2017). Ply (python lex-yacc). http://www.dabeaz.com/ply/ply.html acessado em 27/10/2017.

Gonçalves, R. A. (2017). ebnf-tpp-symbols.odt. Google drive acessado em 27/10/2017.

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*, volume 1. Addison-Wesley, 3 edition. Páginas 169 -182.

Hübner, R. (2016). caleidoscopio. https://bitbucket.org/rhubner/caleidoscopio acessado em 27/10/2017.