Análise Sintática para a linguagem T++

Vitor Yudi Shinohara

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

vitor.ysh@gmail.com

Abstract. *The text described in the next sections will explain one of the steps in the process of running a compiler, the syntax analysis for the T ++ language. It will clarify basic concepts and the implementation of the analyzer, in sequence results and tests performed.*

Resumo. *O texto descrito nas próximas seções irá abordar uma das etapas do processo de funcionamento de um compilador denominado análise sintática para a linguagem T++. Será esclarecido conceitos básicos e a implementação do analisador, em sequência resultados e testes realizados.*

1. Introdução

Uma etapa fundamental da construção de um compilador é a análise sintática, a qual tem principal foco em verificar se, para uma determinada gramática livre de contexto, uma cadeia qualquer pertença à linguagem da gramática, utilizando os *tokens* da análise léxica, é construido uma estrutura de árvore com base no código de entrada, o qual abordaremos com mais detalhes na seção 2, juntamente com conceitos mais detalhados para o melhor entendimento do tema.

Será abordado na seção 4, toda a implementação realizada, contendo trechos de códigos na linguagem Python, além de sua explicação. Também foi realizado testes, mostrados na seção 5, juntamente com a conclusão.

2. Fundamentação

2.1. Gramática Livre de Contexto

Para o entendimento das seções futuras, devemos entender a gramática livre de contexto, cujo papel é fundamental para o funcionamento do analisador sintático e como citado na introdução, auxilia em distinguir cadeias de caracteres válidas ou não.

Uma gramática livre de contexto é composta por um conjunto de símbolos terminais, denotado por letra minúscula, um conjunto de símbolos não terminais, denotado por letras maiúsculas, além de um não terminal inicial, um conjunto de produções e a própria linguagem denotada pela gramática.

Para gerar uma cadeia, começamos com o não-terminal inicial, em sequência, substituímos um símbolo não-terminal presente na cadeia por uma de suas regras. Este processo se repete até que obtemos somente símbolos terminais em nossa cadeia.

Gramáticas livres de contexto são utilizadas devido a diversas possibilidades de construção de cadeias já mencionadas, diferente das expressões regulares, onde existe uma certa limitação, tal como precedência de operadores, o qual é fundamental em uma linguagem de programação.

Código 1. Exemplo de gramática livre de contexto

```
1
2 SomOvelha -> béé SomOvelha
3 SomOvelha -> béé
```

Um problema que deve ser dado atenção é a ambiguidade de uma gramática livre de contexto, caso exista alguma cadeia para qual ela tem mais de uma árvore sintática, a mesma deve ser modificada, gerando assim uma inconsistência entre diferentes compiladores.

2.2. Análise Sintática

O principal objetivo da análise sintática é apontar os erros identificados através do uso de uma gramática livre de contexto, verificando se a entrada está de acordo com a linguagem de programação.

Seu funcionamento é basicamente analisar o código de entrada e criar uma estrutura de árvore sintática para facilitar a verificação de erros no código-fonte.

Existem duas formas de realizar a análise sintática tomando base na árvore gerada. O método *top-down* utiliza a raiz da árvore é derivada até as folhas serem alcançadas, já o método *bottom-up* começa das folhas e vai até a raiz.

```
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. Esquema do funcionamento do shift-reduce

Foi utilizado o método *bottom-up*, devido à *API* Yacc, tendo como consequência, o reconhecimento de mais gramáticas livres de contexto e implementações mais eficientes, utilizando o método de *shift-reduce*, o qual a operação *shift* tem função de avançar o ponteiro para o próximo símbolo de entrada, onde o mesmo é colocado no topo de uma

pilha, e o mesmo é tratado como um único nó da árvore, e a operação *reduce* (imagem 1)é realizada quando o analisador encontra uma regra gramatical e substitui o símbolo do topo da pilha por um correspondente encontrado na gramática.

A *API* emprega o método *Look ahead left-to-right* (LALR) para otimização do uso da árvore sintática, melhorando assim seu desempenho, sendo um variante do *left-to-right* (LR) expandindo os nós prioritariamente a direita e depois da esquerda.

Para a geração das estruturas, foi utilizado uma tabela EBNF especificada anteriormente, o qual basicamente é descrita pela gramática livre de contexto explicada anteriormente.

Código 2. Estrutura da árvore sintática

```
def __init__(self, type_node, child=[], value=''):
    self.type = type_node
    self.child = child
    self.value = value
```

3. Materiais

Foi utilizado como linguagem de programação, o *Python*, por conter certas bibliotecas cujo a implementação seria menos complexa tal como exemplo, o *Ply*, o qual foi utilizado para realizar a própria análise sintática e geração da árvore, além de outra *API* cujo objetivo é representar a árvore de uma forma gráfica, denominada *Graphviz*.

4. Implementação

4.1. Árvore Sintática

Primeiramente foi definido uma estrutura para a árvore sintática, o qual armazenaria o tipo do nó, sendo definido com base na gramática da linguagem, um vetor de filhos e por final um valor, tomando base para o funcionamento do algoritmo.

Código 3. Estrutura da árvore sintática

```
def __init__(self, type_node, child=[], value=''):
    self.type = type_node
    self.child = child
    self.value = value
```

4.2. Analisador Sintático

Para construir a árvore sintática, foi implementado diversas funções obedecendo a tabela EBNF da linguagem T++ (consultar EBNF.txt) obedecendo o padrão determinado pelo Ply, como mostra o algoritmo 4.

Código 4. Função programa

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

A estrutura das funções as quais ditam a regra para popular segue a forma de conter uma ou mais produções comentadas e em seguida a operação cuja adiciona dados à arvore.

Temos também códigos com mais de uma produção, sendo assim devemos distinguir através da quantidade de símbolos produzidos, e adicionando na maioria dos casos, dados diferentes a estrutura especificada anteriormente.

Código 5. Função cuja gramática tem mais de uma produção

```
def p_lista_variaveis(self, p):
    '''
    lista_variaveis : lista_variaveis VIRGULA var
              | var
    '''
    if (len(p) == 4):
      p[0] = Tree('lista_variaveis',[p[1], p[3]])

    elif(len(p) == 2):
      p[0] = Tree('lista_variaveis',[p[1]])
```

Foi necessário em algumas funções, definirmos o valor do 'IDENTIFICADOR', fazendo o uso do terceiro parâmetro da árvore, como mostra o código 6

Código 6. Função populando o campo valor da árvore

```
def p_var(self, p):
    '''
    var : IDENTIFICADOR
      | IDENTIFICADOR indice
    '''

    if (len(p) == 2):
      p[0] = Tree('var', [], p[1])

    elif(len(p) == 3):
      p[0] = Tree('var', [p[2]], p[1])
```

5. Resultados e Conclusão

Por fim, foram realizados testes com diferentes arquivos contendo diferentes erros sintáticos, tais quais, comentários incompletos, escopos de funções e inicialização e declaração de variáveis feitas de forma errada sendo representados a seguir, resultando em uma saída a qual indicava o erro.

Código 7. Declaração de variável incorreta

```
inteiro: a
inteiro: b
flutuante: e[1024][
```

Código 8. Declaração de variável incorreta

```
inteiro: a
inteiro: b
inteiro: c[]
flutuante: d[10][]
flutuante: e[1024][1024]
flutuante: f[1024][1024][1024]

inteiro principal()
   leia(a)
   escreva(b)
fim
```

Código 9. Declaração de comentário incorreto

```
{

inteiro principal()

fim
```

Testes com arquivos sem erros sintáticos foram realizados, obtendo como saída uma visualização gráfica da árvore gerada, e também um arquivo o qual continha o código *dot* da mesma.

Foram realizados também, testes com algoritmos funcionais, tais quais cálculo de fatorial, busca linear, números primos, entre outros. Será ilustrado a seguir, a árvore sintática seguido do código do algoritmo denominado expressão, com objetivo de testar estruturas da linguagem.

Código 10. Algoritmo Expressão

```
inteiro: a[1024]

inteiro func(inteiro: a[])
   retorna(0)
fim

inteiro principal()
   inteiro: i,b

   i:= 1

   a[i]:= b := 0

   b := b + func(a)

   b := +b + i

   retorna(0)
```

```
19 fim
```

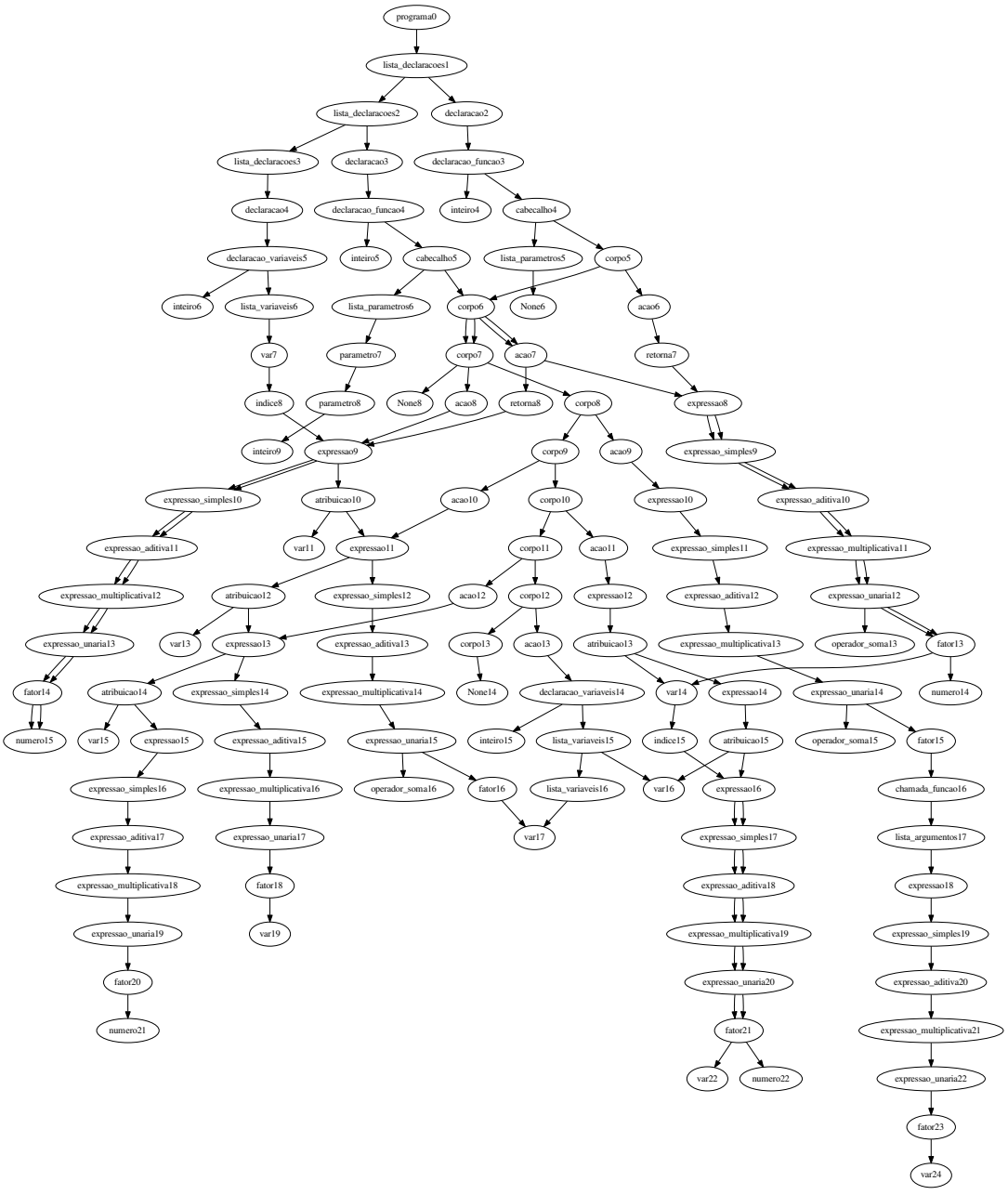

Figura 2. Árvore Sintática do algoritmo Expressão

Referências

Beazley, D. (2001). Ply (python lex-yacc). *See http://www. dabeaz. com/ply*.

Johnson, S. C. (1975). *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ.

Sethi, R., Ullman, J. D., and monica S. Lam (2008). *Compiladores: princípios, técnicas e ferramentas*. Pearson Addison Wesley.