Análise sintática

Thiago Alexsander[1]

[1]Universidade Tecnológica Federal Do Paraná (UTFPR)
R. Rosalina Maria Ferreira, 1233 - Vila Carola, Campo Mourão - PR, 87301-899

[2]DACOM –
Universidade Tecnológica Federal Do Paraná – Campo Mourão, PR – Brazil

thiago.2014@alunos.utfpr.edu.br

Abstract.

Resumo. *Este artigo tem como objetivo esclarecer e objetificar o desenvolvimento de uma análise sintática pertencente a um compilador da linguagem T++.*

1. Introdução

Após a Análise léxica o próximo passo de um compilador é a análise sintática. Seu objetivo é verificar a estrutura sintática do código (T++) utilizando-se dos *tokens* obtidos na fase anterior de análise léxica, desta forma determinando se as sentenças presentes no código são válidas para a linguagem T++.

A análise sintática foi desenvolvida utilizando a linguagem Python em conjunto com uma biblioteca especifica para o aprendizado de desenvolvimento de compiladores (PLY YACC), essa biblioteca oferece ferramentas para realizar a análise léxica e sintática.

2. T++

A linguagem T++ é uma linguagem que oferece suporte aos tipos básicos de dados, como inteiro, flutuante, notação científica e *arrays*, porém não possui suporte a caracteres de texto, como *strings* e *char*. Outra característica do T++ é que sua sintaxe é completamente em português, desse modo ajudando falantes da língua portuguesa a ter um primeiro contato com a programação.

3. Gramatica BNF

[ht] Para realizar a analise sintática é preciso definir como será construída a gramática do compilador. Assim de forma geral utiliza-se de BFN(Backus-Naur notation) para se descrever a sintaxe de uma linguagem de programação. Desta forma BNF consiste em:

- Conjunto de símbolos terminais
- Conjunto de símbolos não terminais
- Conjunto de regras de produção que deve seguir o formato:
  Left-Hand-Side ::= Right-Hand-Side

No qual a parte LHS é um simbolo não terminal e a parte RHS é uma sequencia de símbolos que podem ou não ser terminais. Neste modelo qualquer sentença produzida utilizando as regras de produção está sintaticamente correta. Também é possível criar uma arvore para determinar se uma sentença está sintaticamente correta, caso não seja possível criar a arvore então a sentença está incorreta. As figuras a seguir (1 2 3 4 5) são referentes a lista de gramática utilizada no projeto.

```
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
      | ID indice

indice ::= indice "[" expressao "]"
         | "[" expressao "]"
```

Figura 1. BNF

```
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
```

Figura 2. BNF

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

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
```

Figura 3. BNF

```
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
```

Figura 4. BNF

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

chamada_funcao ::= ID "(" lista_argumentos ")"

lista_argumentos ::= lista_argumentos "," expressao
                   | expressao
                   | vazio
```

Figura 5. BNF

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

Figura 6. Regra de gramática de forma a construir a árvore.

3.1. YACC

No PLY existe uma ferramenta chamada Yacc que é responsável por fazer a analise sintática a partir de uma lista de regras e símbolos, *tokens* da fase léxica, essa analise é feita sobre um código fornecido. É possível criar as regras de gramática de modo que já adquiramos uma árvore sintática, como monstra a Figura 6.

4. Exemplos

Com uma entrada na linguagem T++ como mostra a Figura 7

```
inteiro: n
flutuante: a[10]

inteiro fatorial(inteiro: n)
    inteiro: fat
    se n > 0 então {não calcula se n > 0}
        fat := 1
        repita
            fat := fat * n
            n := n - 1
        até n = 0
        retorna(fat) {retorna o valor do fatorial de n}
    senão
        retorna(0)
    fim
fim

inteiro principal()
    leia(n)
    escreva(fatorial(n))
    retorna(0)
fim
```

Figura 7. Exemplo entrada

O resultado da análise sintática gera uma árvore como mostra as Figuras (8,9,10,11,12).

Figura 8. Árvore resultante

Figura 9. Árvore resultante

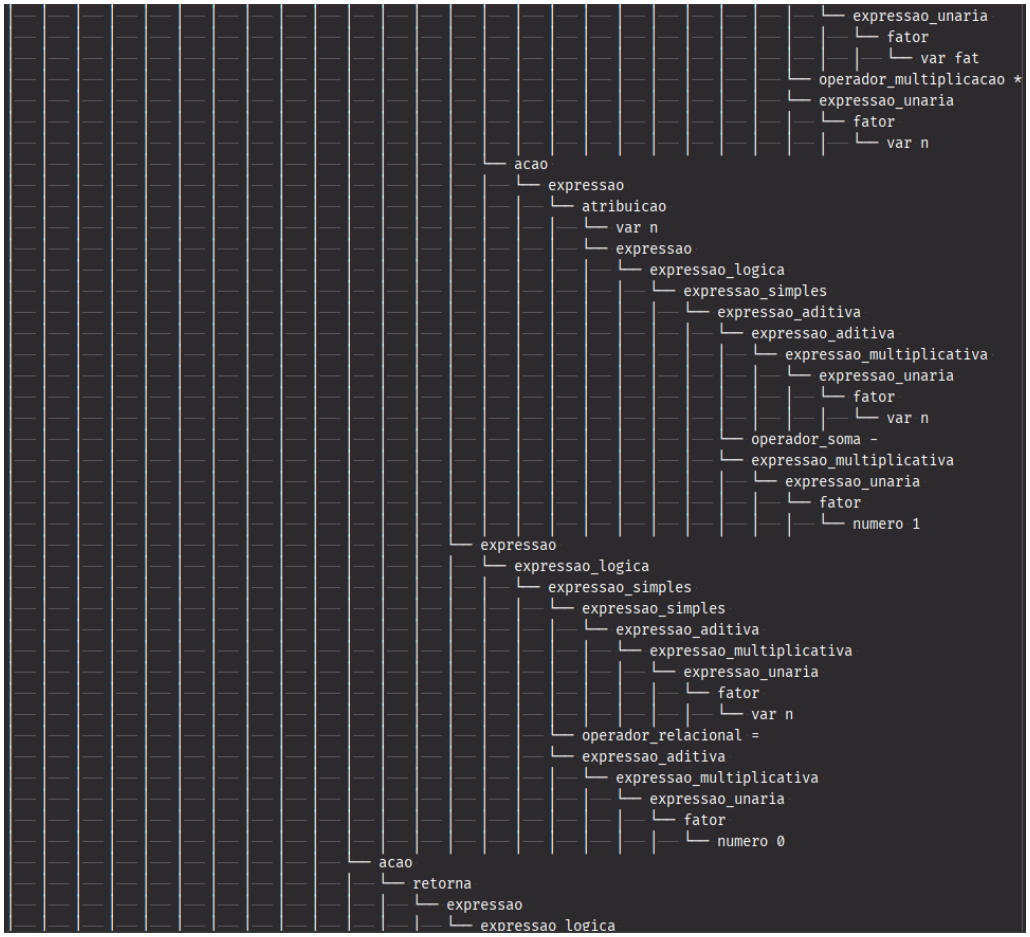

Figura 10. Árvore resultante

Figura 11. Árvore resultante

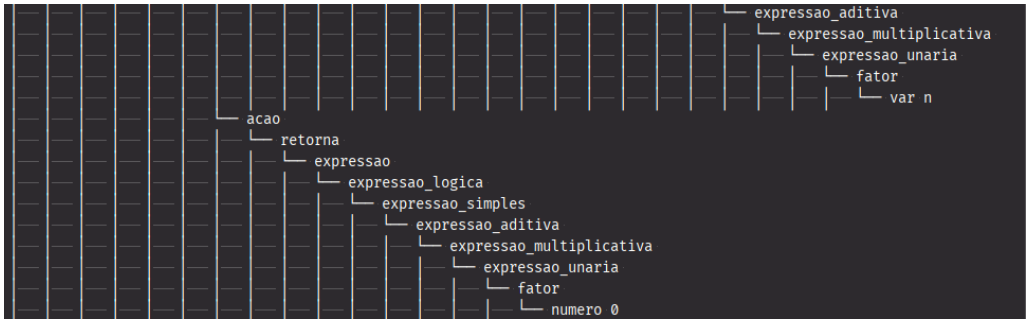

Figura 12. Árvore resultante

4.1. Parser

O YACC também cria outros dois arquivos, ”parser.out” e ”parsertab.py”, no qual demonstra todas as regras e gramáticas, símbolos terminais e não terminais e onde eles foram utilizadas, além de um detalhado percorrimento de cada estado como mostra a Figura 13.

```
Rule 0     S' -> programa
Rule 1     programa -> lista_declaracoes
Rule 2     lista_declaracoes -> lista_declaracoes declaracao
Rule 3     lista_declaracoes -> declaracao
Rule 4     declaracao -> declaracao_variaveis
Rule 5     declaracao -> inicializacao_variaveis
Rule 6     declaracao -> declaracao_funcao
Rule 7     declaracao_variaveis -> tipo DOISPONTOS lista_variaveis
Rule 8     inicializacao_variaveis -> ATRIBUICAO
Rule 9     lista_variaveis -> lista_variaveis VIRGULA var
Rule 10    lista_variaveis -> var
Rule 11    var -> ID
Rule 12    var -> ID indice
Rule 13    indice -> indice ACOLCHETE expressao FCOLCHETE
Rule 14    indice -> ACOLCHETE expressao FCOLCHETE
Rule 15    tipo -> INTEIRO
Rule 16    tipo -> FLUTUANTE
Rule 17    declaracao_funcao -> tipo cabecalho
Rule 18    declaracao_funcao -> cabecalho
Rule 19    cabecalho -> ID APAREN lista_parametros FPAREN corpo FIM
Rule 20    lista_parametros -> lista_parametros VIRGULA parametro
Rule 21    lista_parametros -> parametro
Rule 22    lista_parametros -> vazio
Rule 23    parametro -> tipo DOISPONTOS ID
Rule 24    parametro -> ID
Rule 25    parametro -> parametro ACOLCHETE FCOLCHETE
Rule 26    corpo -> corpo acao
Rule 27    corpo -> vazio
Rule 28    acao -> expressao
Rule 29    acao -> declaracao_variaveis
Rule 30    acao -> se
Rule 31    acao -> repita
Rule 32    acao -> leia
```

Figura 13. Exemplo regras do parser

5. Referencias

PLY YACC : https://www.dabeaz.com/ply/

Definição de BNF : http://www.cs.umsl.edu/ janikow/cs4280/bnf.pdf