Compiladores - Análise Sintática

Alisson da S. Vieira

[1]Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brasil

alissonv@alunos.utfpr.edu.br

Abstract. *For this work, we will implement Syntactic Analysis, which is the analysis after the Lexical Analysis, done together with the tokens generated from the lexical analysis. With these tokens, we will implement the methods to build the syntax tree of the code. We will discuss development processes throughout the article.*

Resumo. *Para esse trabalho, iremos implementar a Análise Sintática, esta que é a análise posterior a Análise Léxica, feita juntamente com os tokens gerados a partir da análise léxica. Com esses tokens, iremos implementar os métodos para construir a árvore sintática do código. E vamos discutir sobre os processos de desenvolvimento com o decorrer do artigo.*

1. Introdução

A análise sintática é um processo que visa, a partir do conjunto de tokens da análise léxica, gerar uma árvore sintática que vai conter toda a estrutura do código de entrada. Cada nó dessa arvore será representado através das expressões regulares, junto dos tokens e por fim, os respectivos valores de cada um no arquivo de entrada. Essa árvore sintática será utilizada nas próximas etapas do processo de compilação.

2. Objetivo

Estaremos realizando na disciplina um compilador para a linguagem de programação fictícia T++, esta que foi desenvolvida com o intuito de auxiliar com o aprendizado dos alunos na disciplina. Neste trabalho especificamente, estaremos realizando a análise sintática a segunda parte do processo de compilação, e para trabalho foi utilizado a linguagem de programação `Python` com as bibliotecas: `PLY` e `Anytree`.

A decisão de se utilizar `Python`, se deve ao fato de que é uma linguagem voltada para a produtividade, possuindo diversas ferramentas nativas que auxiliam o desenvolvedor a apenas focar em produzir. Já a escolha de usar a biblioteca `PLY`, se deve ao fato de que ela fornece a maioria dos recursos `lex`/`yacc` padrões, incluindo suporte para: produções vazias, regras de precedência, recuperação de erros e suporte para gramáticas ambíguas, além de ser relativamente simples de se usar. Quanto a biblioteca `Anytree`, ela auxilia no processo de criação da árvore sintatica.

No final dessa análise, espera-se uma árvore sintática, esta que será utilizada em etapas futuras no processo de compilação.

3. Descrição da gramática

A gramatica da linguagem T++ possui várias regras definidas pelo padrão BNF (Backus-Naur Form), este padrão é uma forma de representação textual para a descrição das linguagens livres de contexto. Abaixo pode ser visto os diagramas sintáticos que foram gerados a partir da gramática da linguagem.

Figura 1. REGRA: programa.

A Figura 1 mostra o diagrama sintático da regra "programa", que é: programa ::= lista_declaracoes.

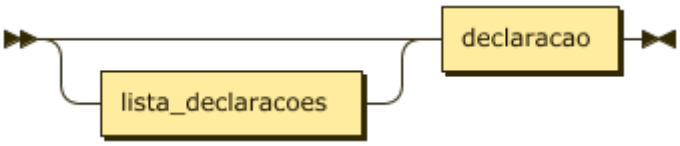

Figura 2. REGRA: lista_declaracoes.

A Figura 2 mostra o diagrama sintático da regra "lista_declaracoes", que é: lista_declaracoes ::= lista_declaracoes declaracao | declaracao.

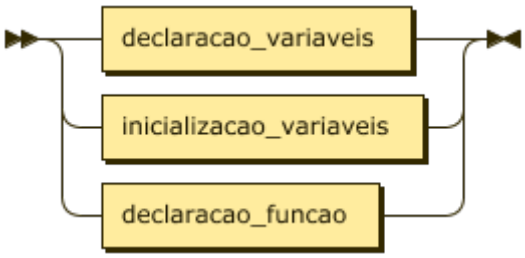

Figura 3. REGRA: declaracao.

A Figura 3 mostra o diagrama sintático da regra "declaracao", que é: declaracao ::= declaracao_variaveis | inicializacao_variaveis | declaracao_funcao.

Figura 4. REGRA: declaração_variaveis.

A Figura 4 mostra o diagrama sintático da regra "declaracao_variaveis", que é: declaracao_variaveis ::= tipo ":"lista_variaveis.

Figura 5. REGRA: inicializacao_variaveis.

A Figura 5 mostra o diagrama sintático da regra “inicializacao_variaveis”, que é: inicializacao_variaveis ::= atribuicao.

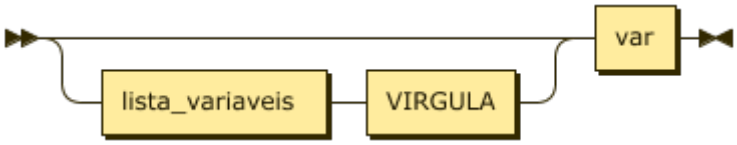

Figura 6. REGRA: lista_variaveis.

A Figura 6 mostra o diagrama sintático da regra “lista_variaveis”, que é: lista_variaveis ::= lista_variaveis “,”var | var.

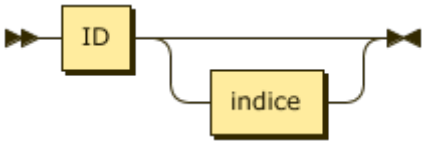

Figura 7. REGRA: var.

A Figura 7 mostra o diagrama sintático da regra “var”, que é: var ::= ID | ID indice.

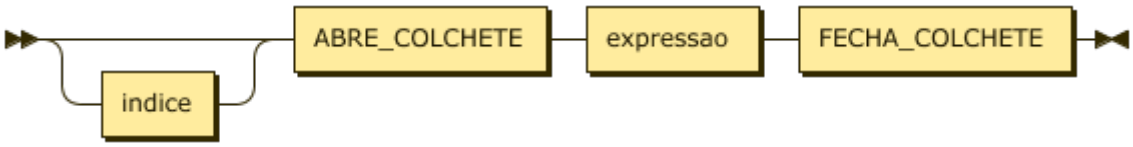

Figura 8. REGRA: indice.

A Figura 8 mostra o diagrama sintático da regra “indice”, que é: indice ::= indice “[”expressao “]”| “[”expressao “]”.

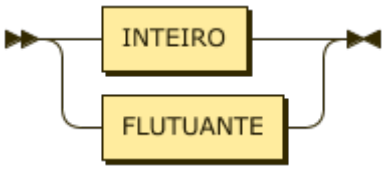

Figura 9. REGRA: tipo.

A Figura 9 mostra o diagrama sintático da regra “tipo”, que é: tipo ::= INTEIRO | FLUTUANTE.

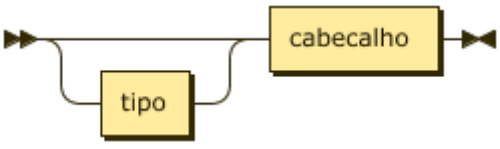

Figura 10. REGRA: declaracao_funcao.

A Figura 10 mostra o diagrama sintático da regra “declaracao_funcao”, que é: declaracao_funcao ::= tipo cabecalho | cabecalho.

Figura 11. REGRA: cabecalho.

A Figura 11 mostra o diagrama sintático da regra “cabecalho”, que é: cabecalho ::= ID “(”lista_parametros “)”corpo FIM.

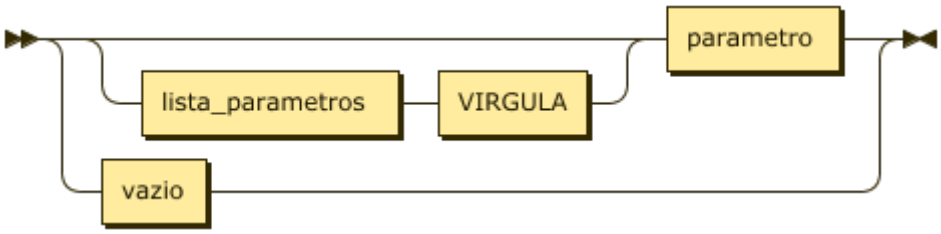

Figura 12. REGRA: lista_parametros.

A Figura 12 mostra o diagrama sintático da regra “lista_parametros”, que é: lista_parametros ::= lista_parametros “,”parametro | parametro | vazio.

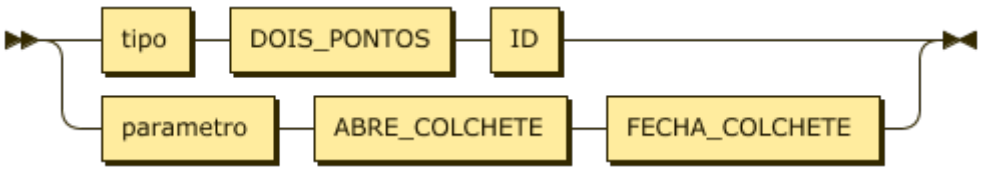

Figura 13. REGRA: parametro.

A Figura 13 mostra o diagrama sintático da regra “parametro”, que é: parametro ::= tipo “:”ID | parametro “[”“]”.

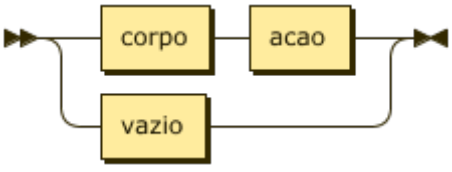

Figura 14. REGRA: corpo.

A Figura 14 mostra o diagrama sintático da regra “corpo”, que é: corpo ::= corpo acao | vazio.

Figura 15. REGRA: acao.

A Figura 15 mostra o diagrama sintático da regra “acao”, que é: acao ::= expressao | declaracao_variaveis | se | repita | leia | escreva | retorna | erro.

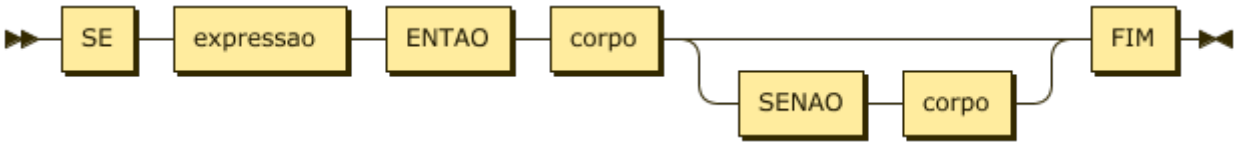

Figura 16. REGRA: se.

A Figura 16 mostra o diagrama sintático da regra “se”, que é: se ::= SE expressao ENTAO corpo FIM | SE expressao ENTAO corpo SENAO corpo FIM.

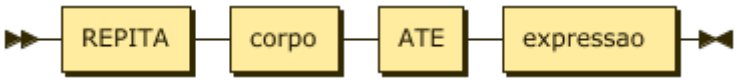

Figura 17. REGRA: repita.

A Figura 17 mostra o diagrama sintático da regra “repita”, que é: repita ::= REPITA corpo ATE expressao.

Figura 18. REGRA: atribuicao.

A Figura 18 mostra o diagrama sintático da regra “atribuicao”, que é: atribuicao ::= var “:=”expressao.

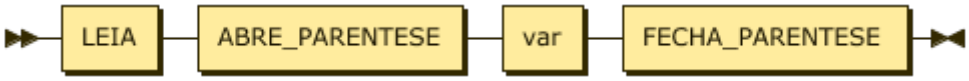

Figura 19. REGRA: leia.

A Figura 19 mostra o diagrama sintático da regra “leia”, que é: leia ::= LEIA “(”var “)”.

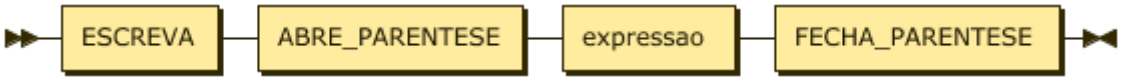

Figura 20. REGRA: escreva.

A Figura 20 mostra o diagrama sintático da regra “escreva”, que é: escreva ::= ESCREVA “(”expressao “)”.

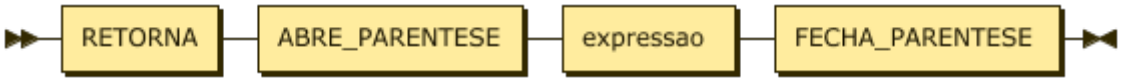

Figura 21. REGRA: retorna.

A Figura 21 mostra o diagrama sintático da regra “retorna”, que é: retorna ::= RETORNA “(”expressao “)”.

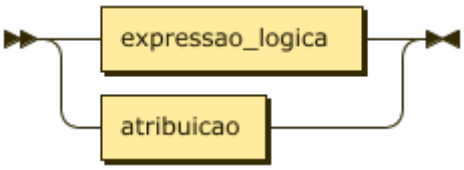

Figura 22. REGRA: expressao.

A Figura 22 mostra o diagrama sintático da regra “expressao”, que é: expressao ::= expressao_logica | atribuicao.

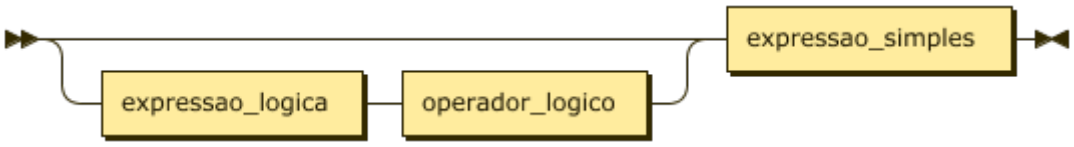

Figura 23. REGRA: expressao_logica.

A Figura 23 mostra o diagrama sintático da regra “expressao_logica”, que é: expressao_logica ::= expressao_simples | expressao_logica operador_logico expressao_simples.

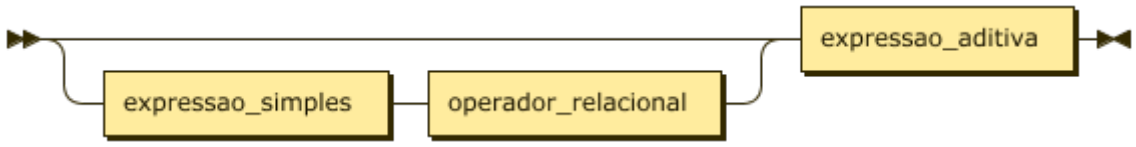

Figura 24. REGRA: expressao_simples.

A Figura 24 mostra o diagrama sintático da regra “expressao_simples”, que é: expressao_simples ::= expressao_aditiva | expressao_simples operador_relacional expressao_aditiva.

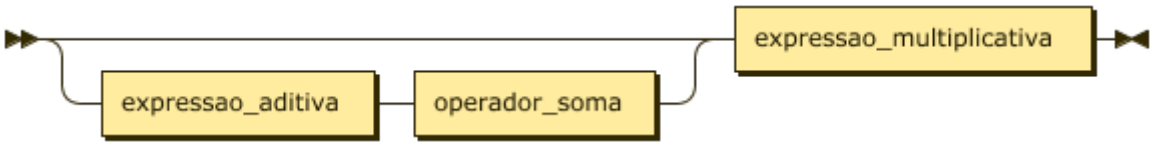

Figura 25. REGRA: expressao_aditiva.

A Figura 25 mostra o diagrama sintático da regra “expressao_aditiva”, que é: expressao_aditiva ::= expressao_multiplicativa | expressao_aditiva operador_soma expressao_multiplicativa.

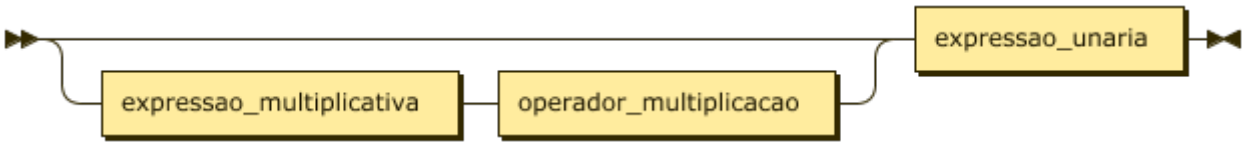

Figura 26. REGRA: expressao_multiplicativa.

A Figura 26 mostra o diagrama sintático da regra “expressao_multiplicativa”, que é: expressao_multiplicativa ::= expressao_unaria | expressao_multiplicativa operador_multiplicacao expressao_unaria.

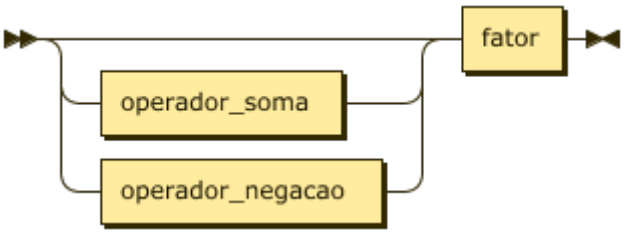

Figura 27. REGRA: expressao_unaria.

A Figura 27 mostra o diagrama sintático da regra “expressao_unaria”, que é: expressao_unaria ::= fator | operador_soma fator | operador_negacao fator.

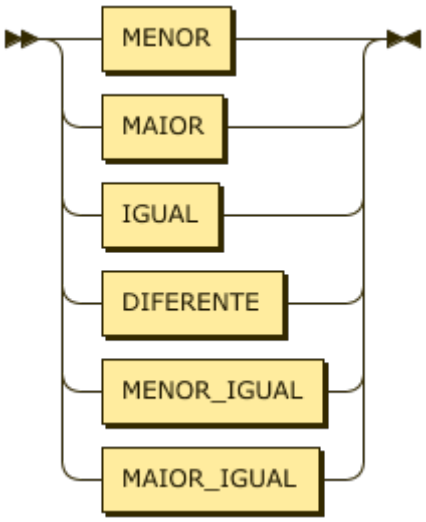

Figura 28. REGRA: operador_relacional.

A Figura 28 mostra o diagrama sintático da regra “operador_relacional”, que é: operador_relacional ::= “¡”| “¿”| “=”| “¡¿”| “¡=”| “¿=”.

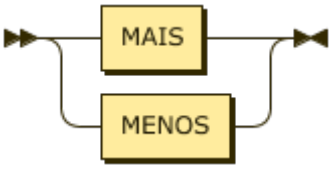

Figura 29. REGRA: operador_soma.

A Figura 29 mostra o diagrama sintático da regra “operador_soma”, que é: operador_soma ::= “+”| “-”.

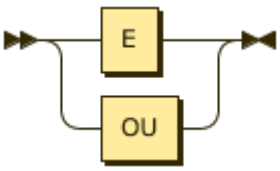

Figura 30. REGRA: operador_logico.

A Figura 30 mostra o diagrama sintático da regra “operador_logico”, que é: operador_logico ::= “”| “——”.

Figura 31. REGRA: operador_negacao.

A Figura 31 mostra o diagrama sintático da regra “operador_negacao”, que é: operador_negacao ::= “!”.

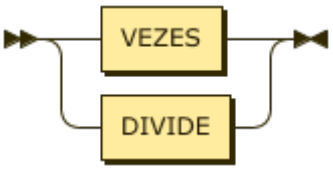

Figura 32. REGRA: operador_multiplicacao.

A Figura 32 mostra o diagrama sintático da regra “operador_multiplicacao”, que é: operador_multiplicacao ::= “*”| “/”.

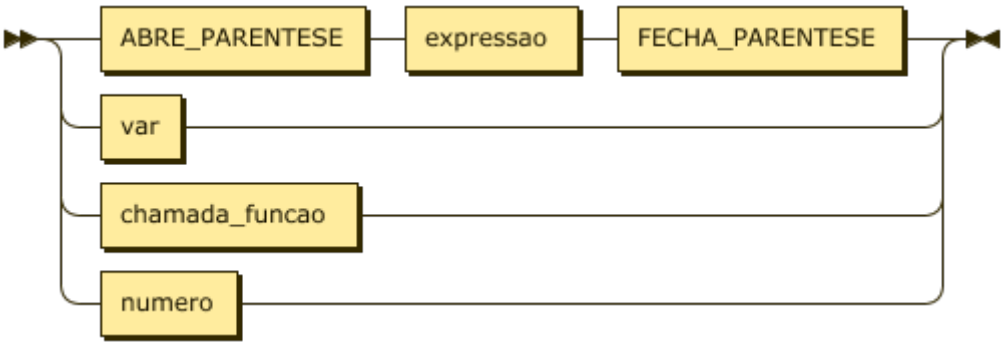

Figura 33. REGRA: fator.

A Figura 33 mostra o diagrama sintático da regra “fator”, que é: fator ::= “(”expressao “)”| var | chamada_funcao | numero.

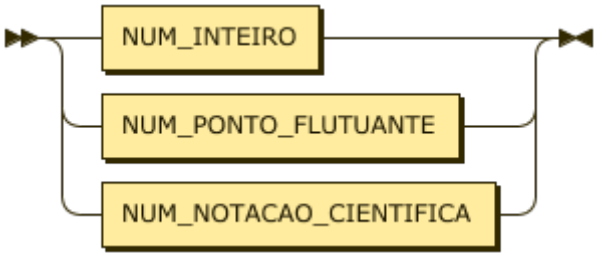

Figura 34. REGRA: numero.

A Figura 34 mostra o diagrama sintático da regra “numero”, que é: numero ::= NUM_INTEIRO | NUM_PONTO_FLUTUANTE | NUM_NOTACAO_CIENTIFICA.

Figura 35. REGRA: chamada_funcao.

A Figura 35 mostra o diagrama sintático da regra “chamada_funcao”, que é: chamada_funcao ::= ID “(”lista_argumentos “)”.

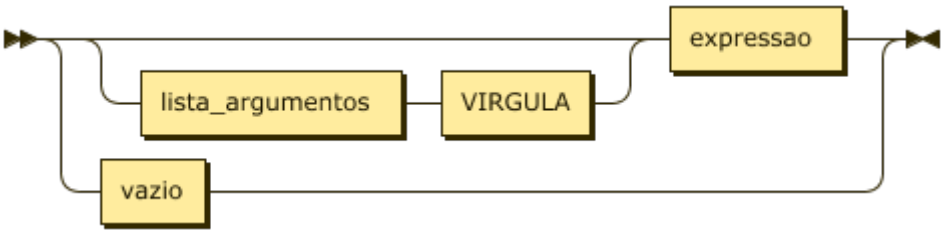

Figura 36. REGRA: lista_argumentos.

A Figura 36 mostra o diagrama sintático da regra 'lista_argumentos', que é: lista_argumentos ::= lista_argumentos ","expressao — expressao — vazio.

4. Formato da Análise Sintática

O formato da Árvore Sintática gerada pela análise é o LALR(1), ou Look-Ahead LR parser, que é o implementado pela biblioteca `yacc`. Este formato consegue diminuir os estados da tabela final, porém gera menos estados que outros formatos. Mas levando em conta como a grande maioria dos compiladores possuem suporte para esse formato, ele foi o escolhido.

5. Implementação e Yacc

Quanto ao desenvolvimento, foi utilizado o modulo `yacc.py` (Yet Another Compiler Complier), do `ply`, biblioteca da linguagem de programação `Python`, que contem ferramentas para tanto a análise léxica como a sintática. O `yacc.py` fornece muitos recursos que já estão disponíveis no `UNIX yacc` e alguns recursos extras que dão a ele algumas vantagens sobre o `yacc` tradicional.

Para a implementação, cada regra gramatical é definida por uma função Python em que contém um cabeçalho com a especificação gramatical apropriada. As instruções que compõem o corpo da função implementam as ações semânticas da regra. Cada função aceita um único argumento p que é uma sequência contendo os valores de cada símbolo gramatical na regra correspondente. No Código 1 vemos um exemplo utilizando a regra: programa.

Código 1. Exemplo da função com o cabeçalho da regra: programa

```
1     def p_programa(p):
2         """programa : lista_declaracoes"""
3
4         global root
5
6         programa = MyNode(name='programa', type='PROGRAMA')
7
8         root = programa
9         p[0] = programa
10        p[1].parent = programa
```

No caso do Código 1, temos a declaração da regra que será o nó raiz da árvore, essa é a regra: programa, isso pois logo após ter declarado a função, definimos dentro de um comentário a regra que, se encontrada, executa essa função. Para isso, temos que criar uma instância do `MyNode`, que é uma classe criada unicamente para representar os nós

da árvore, e como forma de identifica-los passamos uma label, representada pelo atributo name, que nese caso é programa, e o tipo, representado pelo atributo type, que nesse caso é PROGRAMA. Logo após a definição, passamos esse valor para a instância `root` que é a nossa árvore, dessa forma definimos a regra programa como o nó raiz.

Código 2. Exemplo do tratamento de erro para a regra: indice

```
1  def p_indice_error(p):
2  """indice : error ABRE_COLCHETE expressao FECHA_COLCHETE
3                | ABRE_COLCHETE error FECHA_COLCHETE
4                | indice ABRE_COLCHETE error FECHA_COLCHETE
5  """
6
7      global linha, coluna, erroMessage
8
9      print('\n[Linha: {}, Coluna: {}] {}: Erro na definicao do indice (expressao
10     ou indice).\n'.format(linha, coluna, erroMessage))
11     mostrarErro(p)
12
13 def mostrarErro(p):
14     global linha
15
16     if detailedLogs:
17         for i in range(len(p)):
18             print("p[{}]:{}".format(i, p[i]))
19         print(end='\n')
20
21     father = MyNode(name='ERROR::{}'.format(error_line), type='ERROR')
22     logging.error('Syntax error parsing index rule at line {}'.format(linha))
23     parser.errok()
24
25     p[0] = father
```

Já, o Código 2 mostra o tratamento de quando é identificado um erro. Para isso, novamente, é necessário definir as regras que serão o gatilho para essa função executar. Após isso, uma mensagem de erro é mostrada na tela, e é chamado a função `mostrarErro()`. Essa função mostra um erro formatado, caso o usuário tenha passado por linha comando o seguinte parametro: 'd' na hora da execução do programa. Ele também cria o nó de erro, de maneira idêntica ao nó criado no Código 1.

No código disponibilizado pelo professor, as funções já estavam criadas para cada regra da gramática, então foi implementado para elas as funções de erro com seus cabeçalhos correspondente.

6. Resultados

A árvore sintática é gerada ao final da análise, e ela será utilizada nas próximas etapas do processo de compilação. Para testar a análise, foi usado como arquivo de entrada o Código 3.

Código 3. Arquivo de entrada

```
1 inteiro principal()
2     retorna(0)
3 fim
```

Como o Código 3 não possui nenhum erro sintático, a árvore sintática é gerada ao final da análise, como podemos ver na Figura 37.

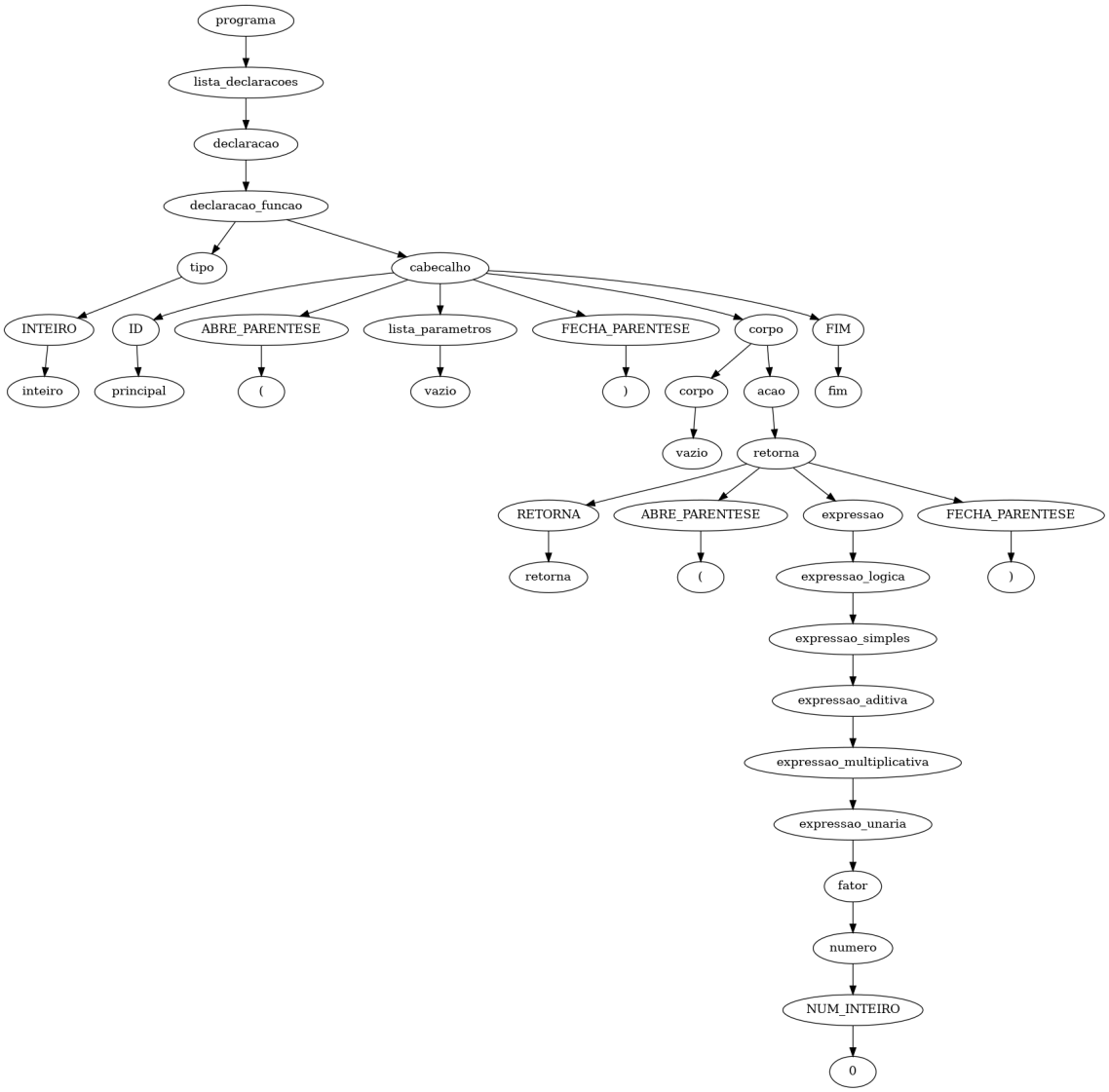

Figura 37. Arvore sintática.

No caso do Código 3, não vemos nenhum problema na sintaxe do programa, mas já no Código 4 temos um problema.

Código 4. Arquivo de entrada com erro

```
1 inteiro principal(inteiro b)
2     retorna(0)
3 fim
```

Agora a função 'principal' espera um parâmetro 'b', este que foi declarado da maneira incorreta. Para a declaração se espera o ':'. Podemos ver a árvore gerada na Figura 38.

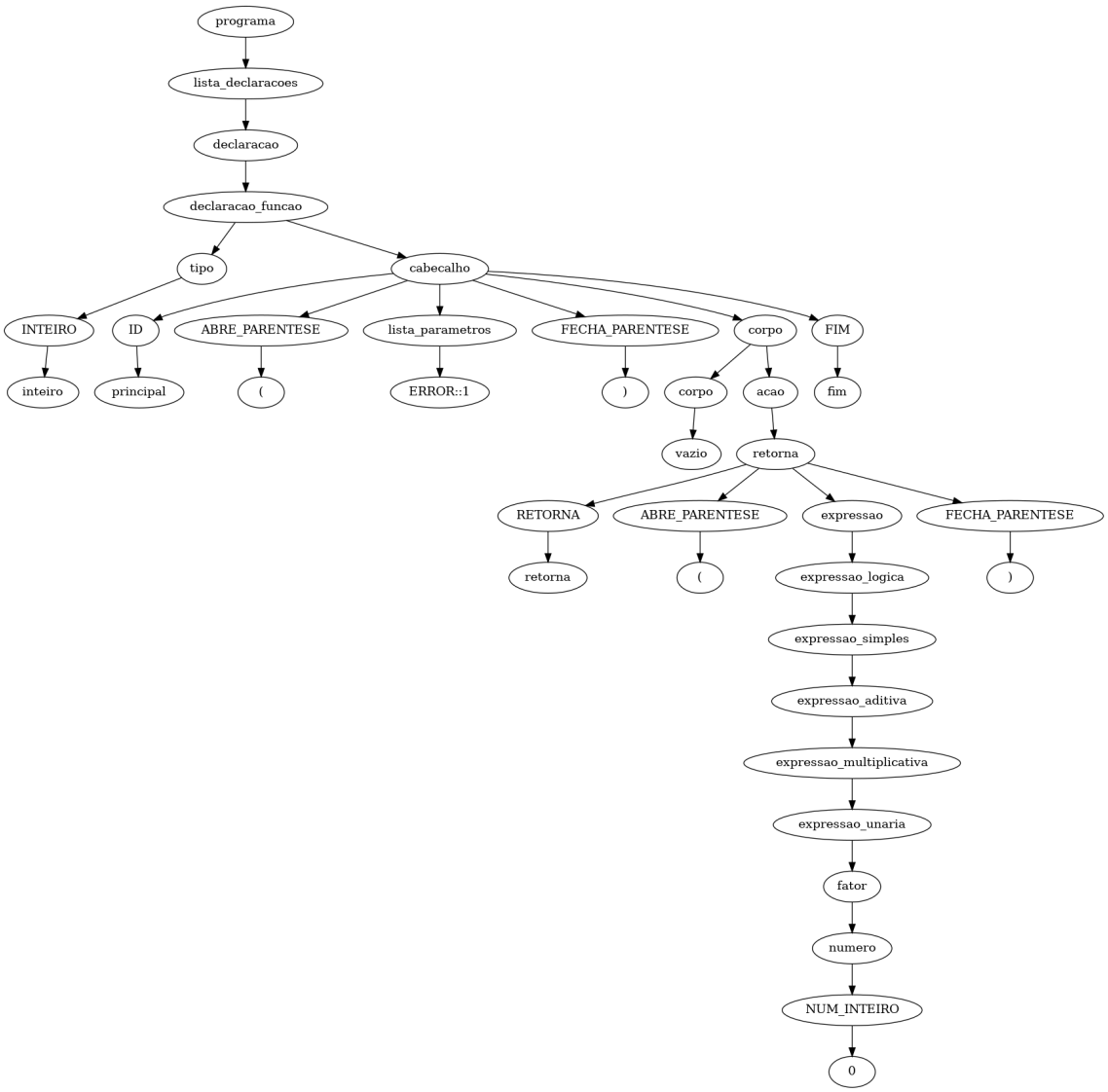

Figura 38. Arvore sintática com um nó de erro.

Referências

Beazley, D. M. Documentação ply.py. ttps://www.dabeaz.com/ply/ply.tml.

Gonçalves, R. A. Conjunto de testes. ttps://moodle.utfpr.edu.br/mod/resource/view.pp?id=136962.

Gonçalves, R. A. Slides - análise léxica. ttps://moodle.utfpr.edu.br/pluginfile.pp/241274/mod$_r$$esource/content/11/aula - 06 - analise - sintatica - glc.md.slides.pdf.

Wikipedia. Lalr parser. ttps://en.wikipedia.org/wiki/LALR_parser.\end{tebibliography