

Compilador Na Linguagem TPP

Gabriel David Sacca¹

¹ Departamento de Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 87301 – Campo Mourão – PR – Brazil

{gabrielsacca}@alunos.utfpr.edu.br

Abstract. *The work was divided into 4 parts, being the lexical, syntactic, semantic analysis and the code generation part. For educational purposes the TPP language was used, and at the end of this project we will have a compiler that can understand a TPP code and generate an executable for it, presenting errors or flaws if any. the lexical part contains all the specifications of tokens and reserved words. Semantics, on the other hand, contains the full description of language grammar and discusses the easy and difficult points to be made at this stage of the project.*

Resumo. *O trabalho foi dividido em 4 partes, sendo elas as análises léxicas, sintáticas, semânticas e a parte de geração de código. Por fins educacionais a linguagem TPP foi utilizada, e no final deste projeto teremos um compilador que consegue entender um código em TPP e gerar um executável para o mesmo, apresentando erros ou falhas se houver. a parte léxica contém todas as especificações de Tokens e palavras reservadas. Já a semântica contém toda a descrição da gramática da linguagem e discute acerca dos pontos fáceis e difíceis a serem feitos nesta fase do projeto.*

1. Introdução

Este relatório está dividido em 6 grandes e principais sessões, a primeira delas da uma introdução ao tema proposto, a segunda apresenta a linguagem que será utilizada e quais são as vantagens e desvantagens de se utilizar esta linguagem em específico, a terceira sessão fala sobre a parte léxica do projeto, sua importância no todo, mostrando seu código, como ele foi criado e quais as saídas que ele irá retornar, a quarta sessão apresenta a parte sintática, também mostrando seu código e suas saídas e organização, a quinta parte apresenta a semântica e suas organizações e por último temos a geração de código realizada com a combinação de todas as outras partes. Ao fim do projeto terá também um resumo colocando as considerações finais, retratando as dificuldades e facilidades de se entender e construir um compilador.

2. Linguagem TPP

Esta linguagem foi desenvolvida com fins educacionais para a disciplina de um compilador, e nela temos algumas das estruturas mais conhecidas na área, como laços, funções, declarações de tipos de dados e afins, todas essas estruturas são mais simples e menos complexas para que facilitem no desenvolver do trabalho. Neste capítulo irá ser apresentado todos estes tipos de estruturas, todos os possíveis Tokens e seus respectivos significados para a linguagem:

- **Estrutura de Condição:** Começando pelas condições, temos nesta linguagem o Token SE, SENAO e ENTAO que respectivamente funcionam como o if, elif e else da linguagem python;
- **Laços de Repetição:** Os laços são representados nessa linguagem com os Tokens REPITA e ATE, eles funcionam como o do while da linguagem C, python e outras, e todo código dentro deste laço é executado enquanto a condição do até não seja quebrada;
- **Funções:** Já as funções nesta linguagem podem ou não possuir retorno de dados, parâmetros e todas possuem escopos separados de variáveis;
- **Tipos de dados:** Por ultimo a linguagem possui quatro tipos distintos de dados inteiro, flutuante, vetores e matrizes.

Com estas simples estruturas a linguagem é perfeita para o aprendizado da criação de um compilador, pois se fosse utilizar linguagens conhecidas como C, Python, Java e outras, a complexidade das linguagens tornaria inviável o desenvolvimento do compilador em apenas 1 semestre.

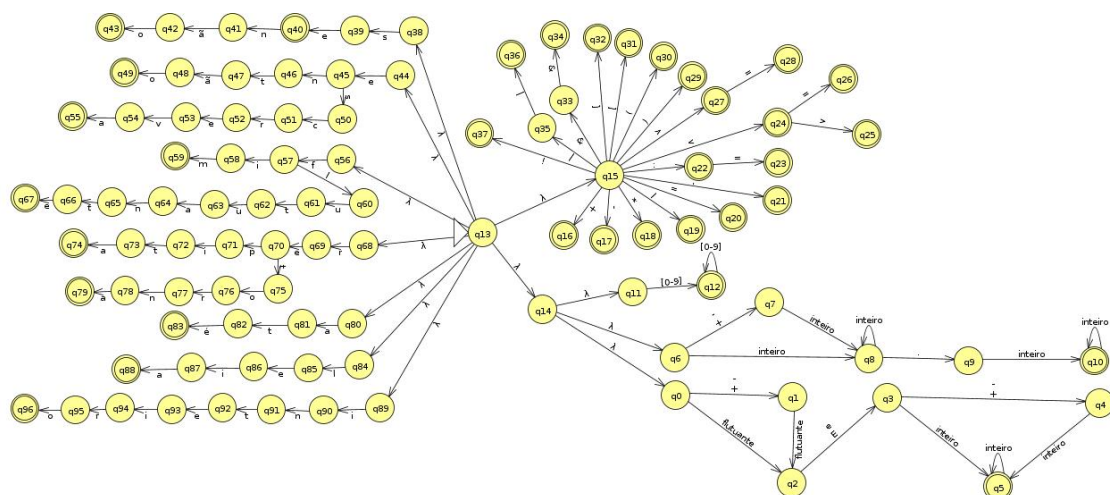


Figura 1. Imagem do Automato e todos os seus possíveis caminhos

3. Léxica

Um compilador é dividido em 4 principais etapas, como já citado na sessão 1, e nesta sessão ira ser mostrado como foi implementar a parte léxica do trabalho mostrando e exemplificando o automato necessário para gerar a expressão regular do código, a ferramenta para criação de um analisador léxico do Python denominada Python, como utilizá-lo e quais suas vantagens e desvantagens, e por ultimo alguns exemplos de entradas e saídas de código que o nosso programa léxico gera com diferentes entradas.

3.1. Especificação dos autômatos

Um automato nada mais é do que uma maquina que dada uma entrada, irá ou não te retornar uma saída com aceito ou não aceito pela linguagem. Neste relatório pode ser observado na figura 1, o automato que gera exatamente as expressões regulares que são necessários para entender todos os Tokens da linguagem. Para a criação da maquina eu

a dividi em 3 principais sessões, sendo a primeira de reconhecimento de palavras reservadas, a segunda para reconhecer Tokens de símbolos da linguagem e a terceira para reconhecer as possíveis formas de representações numéricas.

Começando pelo reconhecimento dos Tokens de símbolos eles são bem simples e intuitivos, pois cada caminho pode te levar para 1 ou 2 possíveis caracteres e a soma dos dois forma uma palavra reconhecida pela linguagem. Um exemplo de Token reconhecido pela linguagem foi , neste automato temos um estado inicial que leva para um dos e um estado final que leva para o segundo , fazendo assim com que a linguagem considere um único um carácter invalido, mas uma sequencia de dois é aceita pela mesma.

Para reconhecer os tipos de dados que na linguagem se limitam a inteiros, flutuantes e números representados com notação científica, foram necessários 3 outros sub autômatos:

- **Inteiro:** Para reconhecer inteiros foi utilizado dois estados, um que reconhece um primeiro simbolo de 0 até 9 e outro que fica repetindo um estado final enquanto houverem números de 0 até 9, se não houver nada além de números o automato irá parar e reconhecer que aquela sequencia de símbolos é um número inteiro;
- **Flutuante:** Os flutuantes são um pouco mais complexos, o primeiro estado da maquina pode levar para dois outros caminhos, um para antes de chegar nos números inteiros reconhecer sinais de positivo e negativo, e outro sem os sinais caso a pessoa não queira identifica-los, após passar por esta etapa de decisão você chega em um estado onde os inteiros podem ser repetidos e assim que encontrar um ponto o automato vai para o próximo estado, onde novamente aceita uma sequencia de inteiros até acabar a palavra, e se nenhum carácter diferente de inteiro for encontrado ela será aceita como flutuante pela linguagem;
- **Notação científica:** E por ultimo a notação científica, que é a mais complexa dos 3 reconhecedores de tipos de dados. No inicio ela trabalha de forma semelhante aos números flutuantes, pois necessita de dois caminhos, um com sinal e outro sem sinal, para reconhecer um número flutuante. Se este número for reconhecido e logo em seguida os caracteres E ou e forem detectados, o automato vai para um novo estado intermediário, e se novamente um flutuante com ou sem sinal forem encontrados, os símbolos são aceitos como notação científica pela linguagem.

Já para reconhecer as palavras reservadas encontradas nesta linguagem é simples, basta criar estados intermediários para as letras da palavra desejada, sendo o ultimo estado a ultima letra dela, e por consequência a palavra já será aceita pela linguagem. Todavia existem palavras que se inciam com as mesmas letras como exemplo o Token SE e o Token SENA0, estas palavras irão começar pelo mesmo caminho até o SE e depois vão se dividir em dois caminhos diferentes, um que aceita a palavra SE e outro que verifica se ainda possui mais caracteres para validar.

3.2. PLY

O PLY [Beazley] é uma ferramenta do Python para auxiliar na criação de um analisador léxico. Ele possui ferramentas que auxiliam na criação e desenvolvimento do código, e além de explicar cada ferramenta do PLY, nesta sessão ira ser mostrado também o código criado para gerar a linguagem léxica no geral.

a primeira etapa do PLY consiste em declarar todas as palavras reservadas e os Tokens como pode ser visto abaixo:

```
import sys
import ply.lex as lex

reserved = {
    'se': 'SE',
    'então': 'ENTAO',
    'senão': 'SENAO',
    'fim': 'FIM',
    'repita': 'REPITA',
    'flutuante': 'FLUTUANTE',
    'retorna': 'RETORNA',
    'até': 'ATE',
    'leia': 'LEIA',
    'escreva': 'ESCREVA',
    'inteiro': 'INTEIRO'
}

tokens = [
    'SOMA',
    'SUBTRACAO',
    'MULTIPLICACAO',
    'DIVISAO',
    'IGUALDADE',
    'VIRGULA',
    'ATRIBUICAO',
    'MENOR',
    'MAIOR',
    'MENOR_IGUAL',
    'MAIOR_IGUAL',
    'ABRE_PARENTESES',
    'FECHA_PARENTESES',
    'DOIS_PONTOS',
    'ABRE_COLCHETE',
    'FECHA_COLCHETE',
    'E_LOGICO',
    'OU_LOGICO',
    'NEGACAO',
    'DIFERENTE',
    'NOTACAO_CIENTIFICA',
    'ID',
    'NUMERO_INTEIRO',
    'NUMERO_FLUTUANTE'
] + list(reserved.values())
```

elas são divididas em 2 grandes estruturas, a primeira delas possui o nome de reserved

e nela você deve colocar todas as palavras reservadas da linguagem, sendo a primeira a palavra em si, e logo em seguida o Token que essa palavra vai gerar. A segunda estrutura com nome de Tokens possui todos os possíveis Tokens que essa linguagem pode gerar.

```
t_ignore = ' \t'
t_SOMA = r'\+'
t_SUBTRACAO = r'\-'
t_MULTIPLICACAO = r'\*'
t_DIVISAO = r'\/'
t_IGUALDADE = r'\='
t_VIRGULA = r','
t_ATRIBUICAO = r'\:='
t_MENOR = r'\<'
t_MAIOR = r'\>'
t_MENOR_IGUAL = r'\<='
t_MAIOR_IGUAL = r'\>='
t_ABRE_PARENTESES = r'\('
t_FECHA_PARENTESES = r'\)'
t_DOIS_PONTOS = r'\:'
t_ABRE_COLCHETE = r'\['
t_FECHA_COLCHETE = r'\]'
t_E_LOGICO = r'\&&'
t_OU_LOGICO = r'\||'
t_NEGACAO = r'\!'
t_DIFERENTE = r'\<>'
t_NOTACAO_CIENTIFICA =
r'((\+|-)?[\d+]+\.[\d+]*)(e|E)(\+|-)?[\d+]+\'
t_NUMERO_FLUTUANTE = r'(\+|-)?[\d+]+\.[\d+]*\'
t_NUMERO_INTEIRO = r'(-|\+)?\d+\'

def t_ID(t):
    r'[A-Za-z_][\w_]*\'
    t.type = reserved.get(t.value, 'ID')
    return t

def t_error(t):
    print('Illegal character: %s' % t.value[0])
    t.lexer.skip(1)

def t_COMMENT(t):
    r'\{((\.|\\n)*?)\}\'
    t.lexer.lineno += len(t.value.split('\\n')) - 1

def t_newline(t):
    r'\\n+\'
    t.lexer.lineno += len(t.value)
```

já no exemplo acima podemos observar as expressões regulares que serão ne-

cessários para reconhecer os Tokens previamente listados, então cada um dos Tokens possui uma expressão regular para si, que deve conter informações de como é o comportamento padrão do mesmo na linguagem escolhida. Além disso temos também algumas expressões padrões do PLY, a primeira delas é o **t_ignore**, e todos os caracteres que estiverem dentro de sua expressão serão ignorados para a análise léxica da linguagem, neste código os espaços em branco, tabulações e quebras de linha foram ignorados como pode ser analisado na linha 45. já a partir da linha 75 temos as funções para exceções como o **t_error**, que é chamado caso algum caractere não seja reconhecido pela linguagem e retorna uma mensagem para o usuário, e a ultima função é o **_COMMENT** que encontra e ignora qualquer comentário.

e a última parte do código são as saídas, você pode escolher vários tipos de saídas possíveis, e neste projeto eu escolhi dois tipos, saída apenas com os Tokens encontrados e saídas com Tokens e os valores destes. Para uma saída com Tokens e valores é necessário digitar C como ultimo argumento de entrada, e sem valores digite S.

```
arquivo = open(sys.argv[1])
lex.lex()

lex.input(arquivo.read())
while True:
    tok = lex.token()
    if not tok: break
    if(len(sys.argv) >= 3):
        if(sys.argv[2] == "C"):
            print(tok.type, ",", tok.value)
        elif(sys.argv[2] == "S"):
            print(tok.type)
    else:
        print(tok.type, ",", tok.value)
```

3.3. Exemplos de entrada e saída

Para testar se tudo esta funcionando perfeitamente alguns exemplos de código em TPP foram passados para teste, um destes exemplos foi o fat, um código que calcula o fatorial de um número que pode ser observado abaixo:

```
inteiro: n

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
```

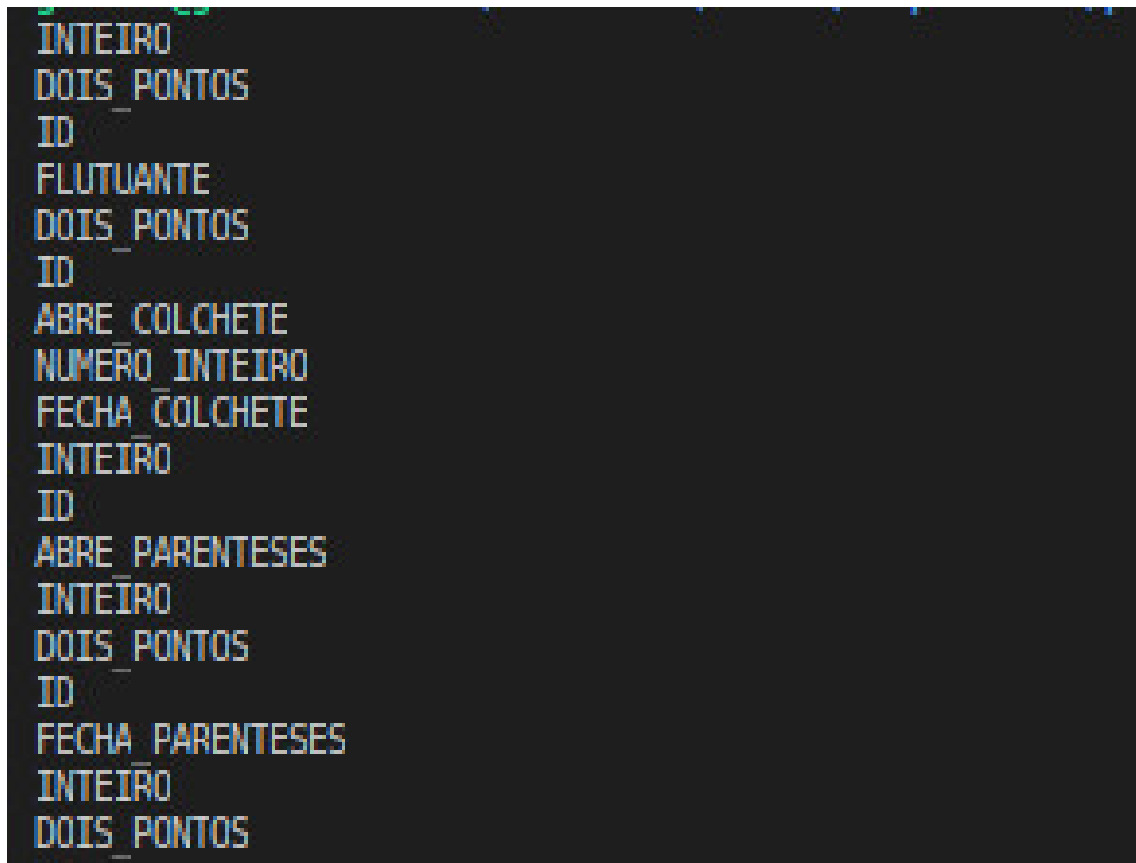
```

        retorna(0)
    fim
fim

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

```

Se a parte léxica estiver funcionando corretamente, então uma lista de Tokens correspondentes aos apresentados deve ser impressa no terminal como os da figura 2, que mostra um trecho inicial da saída esperada, com os Tokens correspondentes.



```

INTEIRO
DOIS_PONTOS
ID
FLUTUANTE
DOIS_PONTOS
ID
ABRE COLCHETE
NUMERO INTEIRO
FECHA COLCHETE
INTEIRO
ID
ABRE PARENTESES
INTEIRO
DOIS_PONTOS
ID
FECHA PARENTESES
INTEIRO
DOIS_PONTOS

```

Figura 2. Exemplo de código em TPP

4. Sintática

Esta sessão relatar todos os pontos para a criação da parte sintática do compilador, mostrando o que é a gramática BNF, como ela funciona, como o Yacc ajuda na criação da sintática e mostrando os resultados dessa etapa.

4.1. Gramatica BNF

Para que o programa consiga captar as regras da linguagem e separar os significados das palavras na mesma, além da separação dos Tokens, também se faz necessário uma

separação lógica dentro da própria linguagem, e para que essa separação seja possível uma gramática livre de contexto, ou BNF, foi utilizada.

Esta gramática consiste em um conjunto de regras que devem ser atendidas para que dado programa seja ou não aceito em uma linguagem, e que além de ser aceito também possa gerar uma árvore de recorrência para este programa que será muito útil nos próximos passos para se criar um compilador. Então para gerar todos os possíveis caminhos dessa linguagem, as seguintes regras foram utilizadas:

```
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 DOIS_PONTOS 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 ABRE_COLCHETE expressao FECHA_COLCHETE
Rule 14     indice -> ABRE_COLCHETE expressao FECHA_COLCHETE
Rule 15     tipo -> INTEIRO
Rule 16     tipo -> FLUTUANTE
Rule 17     declaracao_funcao -> tipo cabecalho
Rule 18     declaracao_funcao -> cabecalho
Rule 19     cabecalho -> ID ABRE_PARENTESES lista_parametros
FECHA_PARENTESES corpo FIM
Rule 20     lista_parametros -> lista_parametros VIRGULA parametro
Rule 21     lista_parametros -> parametro
Rule 22     lista_parametros -> vazio
Rule 23     parametro -> tipo DOIS_PONTOS ID
Rule 24     parametro -> parametro ABRE_COLCHETE FECHA_COLCHETE
Rule 25     corpo -> corpo acao
Rule 26     corpo -> vazio
Rule 27     acao -> expressao
Rule 28     acao -> declaracao_variaveis
Rule 29     acao -> se
Rule 30     acao -> repita
Rule 31     acao -> leia
Rule 32     acao -> escreva
Rule 33     acao -> retorna
Rule 34     se -> SE expressao ENTAO corpo FIM
Rule 35     se -> SE expressao ENTAO corpo SENAO corpo FIM
Rule 36     repita -> REPITA corpo ATE expressao
Rule 37     atribuicao -> var ATRIBUICAO expressao
```



```
Rule 38      leia -> LEIA ABRE_PARENTESES var FECHA_PARENTESES
Rule 39      escreva -> ESCREVA ABRE_PARENTESES expressao
FECHA_PARENTESES
Rule 40      retorna -> RETORNA ABRE_PARENTESES expressao
FECHA_PARENTESES
Rule 41      expressao -> expressao_logica
Rule 42      expressao -> atribuicao
Rule 43      expressao_logica -> expressao_simples
Rule 44      expressao_logica -> expressao_logica operador_logico
expressao_simples
Rule 45      expressao_simples -> expressao_aditiva
Rule 46      expressao_simples -> expressao_simples operador_relacional
expressao_aditiva
Rule 47      expressao_aditiva -> expressao_multiplicativa
Rule 48      expressao_aditiva -> expressao_aditiva operador_soma
expressao_multiplicativa
Rule 49      expressao_multiplicativa -> expressao_unaria
Rule 50      expressao_multiplicativa -> expressao_multiplicativa
operador_multiplicacao expressao_unaria
Rule 51      expressao_unaria -> fator
Rule 52      expressao_unaria -> operador_soma fator
Rule 53      expressao_unaria -> operador_negacao fator
Rule 54      operador_relacional -> MENOR
Rule 55      operador_relacional -> MAIOR
Rule 56      operador_relacional -> IGUALDADE
Rule 57      operador_relacional -> DIFERENTE
Rule 58      operador_relacional -> MENOR_IGUAL
Rule 59      operador_relacional -> MAIOR_IGUAL
Rule 60      operador_soma -> SOMA
Rule 61      operador_soma -> SUBTRACAO
Rule 62      operador_logico -> E_LOGICO
Rule 63      operador_logico -> OU_LOGICO
Rule 64      operador_negacao -> NEGACAO
Rule 65      operador_multiplicacao -> MULTIPLICACAO
Rule 66      operador_multiplicacao -> DIVISAO
Rule 67      fator -> ABRE_PARENTESES expressao FECHA_PARENTESES
Rule 68      fator -> var
Rule 69      fator -> chamada_funcao
Rule 70      fator -> numero
Rule 71      numero -> NUMERO_INTEIRO
Rule 72      numero -> NUMERO_FLUTUANTE
Rule 73      numero -> NOTACAO_CIENTIFICA
Rule 74      chamada_funcao -> ID ABRE_PARENTESES lista_argumentos
FECHA_PARENTESES
Rule 75      lista_argumentos -> lista_argumentos VIRGULA expressao
Rule 76      lista_argumentos -> expressao
```

```
Rule 77      lista_argumentos -> vazio
Rule 78      vazio -> <empty>
```

Estas regras partem de um S' que será sempre o nó raiz de uma árvore que será gerada, a partir dele um automato de pilha vai buscando novos possíveis caminhos de escolha pra que o programa no final seja aceito, se o mesmo for aceito a árvore é gerada através de uma imagem, todavia se for rejeito, um erro deve ser impresso mostrando a linha e coluna que ele ocorreu.

4.2. Analise Sintatica LALR

Este programa trabalha com o analisador sintático do tipo LALR (look-ahead left-right), ou seja, ele analisa as informações provenientes da gramática BNF da esquerda para a direita e de baixo para cima, tentando deduzir as análises sintáticas e gerando uma tabela de ações do próximo estado.

O analisador sintático LALR é utilizado pelo fato de existirem muitos estados diferentes que possuem o mesmo conjunto de componentes, então para resolver isto ele combina estes estados e os expressa através de uma tabela LALR que é utilizada de forma mais simples e barata computacionalmente.

Além desses benefícios de se utilizar o LALR também temos a vantagem de sua análise ser realizada de maneira Bottom-up, ou seja, as últimas regras mais a esquerda serão as primeiras a serem geradas, para que melhore a eficiência na hora de identificar uma informação. Isto ocorre pois é mais barato computacionalmente você identificar um token nas regras que podem gerar Tokens, e ir subindo elas até chegar no raiz, do que partir do nó raiz e ir buscar este Token.

4.3. Yacc

O Yacc é uma ferramenta do python para facilitar na criação e na utilização da gramática BNF para geração da árvore de um compilador, ela possui vários métodos para geração de código e trabalha em conjunto com o Lex da parte léxica o que facilita em muito a programação como um todo.

a primeira etapa que precisa ser feita para utilizar o Yacc é importar suas bibliotecas que podem ser vistas abaixo:

```
import ply.yacc as yacc
import lexica
from anytree import Node, RenderTree
from anytree.exporter import DotExporter
tokens = lexica.tokens
```

nelas, além do ply.yacc que é a biblioteca padrão para geração de código, temos também mais duas coisas que precisam ser importadas, uma biblioteca de geração de código chamada anytree além de toda a parte léxica que já foi criada e explicada nos capítulos anteriores, para que seja possível acessar os Tokens que foram gerados anteriormente.

após isso devemos criar funções que irão gerar a gramática BNF da linguagem, um exemplo dessas funções pode ser visto abaixo:

```
def p_programa(p):
    '''programa : lista_declaracoes'''
    global id
    p[0] = Node("< " + str(id) + " >" + " programa",
                children = [p[1]])
    id +=1
```

Como podemos observar, estas funções possuem o seguinte escopo, o nome da função primeiramente deve conter um `p_` acompanhado de um nome, que será o nome dado a regra que irá ser criada na gramática, e como parâmetro dessa função se passa a palavra a ser verificada, logo em seguida temos a regra que deve ser tratada em formato de string, a primeira palavra dessa regra deve ser exatamente o nome dado a regra BNF, e depois dos ":" vem a regra em si (neste exemplo o nome é `programa`, e a regra é `lista_declaracoes`), se esta regra for um conjunto final, então a frase é aceita e retorna ali, se for uma função ela continuará entrando até encontrar um final ou um erro. Além disso também temos em seguida a parte de geração da árvore, ela utiliza da biblioteca `anytree` como já especificado acima e cria nós para cada interação nesta função onde este nó possui um nome e um filho que será dado quando esta função retornar da `lista_declaracoes`.

Existem algumas funções que possuem diferenças das demais, e estas são a `p_error` e a `find_column`, e elas podem ser vistas a seguir:

```
def find_column(token):
    input = token.lexer.lexdata
    line_start = input.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1

def p_error(p):
    global error
    if p:
        print("Syntax error at token", p.type, "line: ",
              p.lineno, "column: ", find_column(p))

    else:
        print("Syntax error at EOF")
        error = True
```

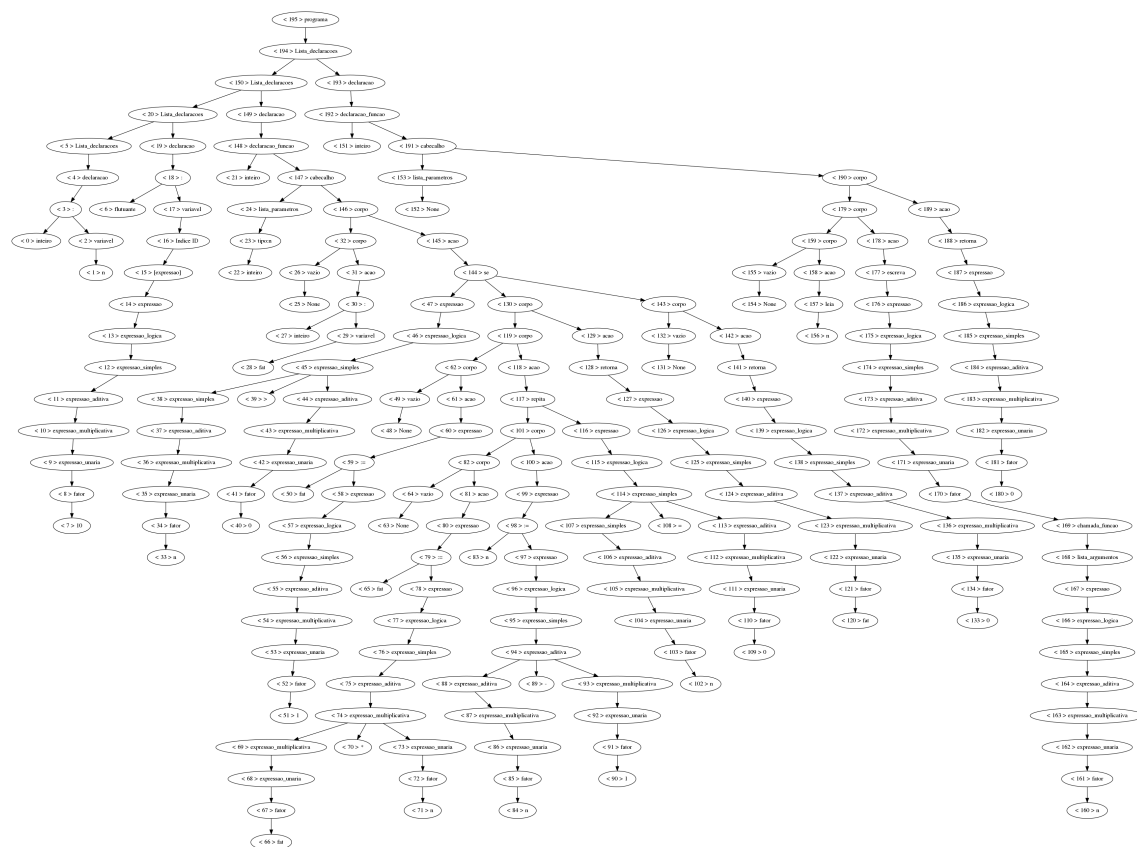
Elas são funções específicas para tratamento de erros, onde a `p_error` fica responsável por se algum erro for encontrado ele imprime o Token, a linha e a coluna deste erro, todavia para encontrar a coluna um cálculo um pouco mais complexo se faz necessário, então a `find_column` foi utilizada, onde se passa o Token e utilizando funções da `lex` e da `yacc` conseguimos a coluna do erro.

E por último temos a parte de geração da árvore que irá utilizar os nós da `anytree` e gerar uma imagem `jpg` com a seguinte função:

```
if(not error):
    DotExporter(programa).to_picture("programa.png")
```

Nela se a função de erro não tiver sido acionada nenhuma vez, ela gera uma árvore com todos os nós e seus conteúdos.

Com todos os passos realizados a árvore gerada irá ser como a da imagem 3



nela podemos visualizar os nós gerados e os seus caminhos, além também de uma prova visual do bottom-up, pois os identificadores como já citado nas sessões anteriores no nó mais a esquerda de todos, e mais a baixo e terminam no nó raiz.

Desta forma, se torna possível perceber que a parte léxica e a sintática são muito importantes para o funcionamento total de um compilador, elas geram Tokens, palavras reservadas e a árvore de decisões que serão utilizados nas demais áreas, todavia não é são as únicas, e no decorrer do projeto as próximas áreas irão utilizar o que já foram desenvolvidas nestas.

Beazley, D. M. Ply (python lex-yacc) documentation.