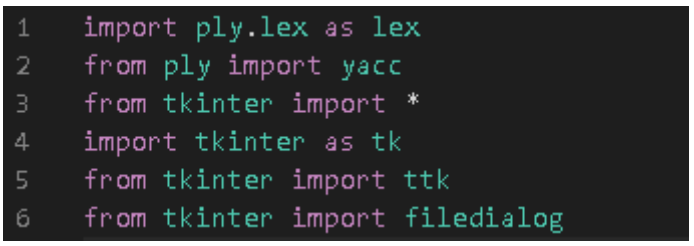
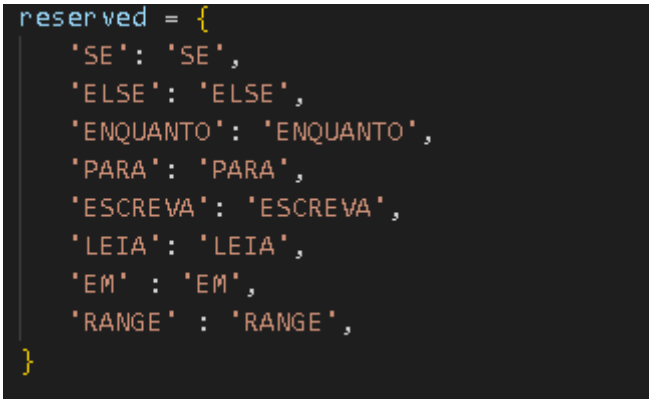


```
import ply.lex as lex
from ply import yacc
from tkinter import *
import tkinter as tk
from tkinter import ttk
from tkinter import filedialog
```

Importações:

- **ply.lex as lex:** Importa o módulo lex da biblioteca ply para análise léxica.
- **from ply import yacc:** Importa a função yacc do módulo ply para análise sintática.
- **from tkinter import ***: Importa todos os símbolos do módulo tkinter para criar uma interface gráfica.
- **import tkinter as tk:** Importa o módulo tkinter e o renomeia para tk.
- **from tkinter import ttk**: Importa widgets adicionais do módulo tkinter.
- **from tkinter import filedialog:** Importa o widget filedialog do módulo tkinter para lidar com arquivos.



```
reserved = {
    'SE': 'SE',
    'ELSE': 'ELSE',
    'ENQUANTO': 'ENQUANTO',
    'PARA': 'PARA',
    'ESCREVA': 'ESCREVA',
    'LEIA': 'LEIA',
    'EM' : 'EM',
    'RANGE' : 'RANGE',
}
```

Reserved:

É um dicionário que mapeia palavras-chave reservadas para os seus respectivos tokens. Isso permite identificar essas palavras-chave durante a análise léxica.

```
tokens = [
    'INTEIRO',
    'DOUBLE',
    'STRING',
    'INT',
    'VARIAVEL',
    'OP_MAT_ADICAO',
    'OP_MAT_SUB',
    'OP_MAT_MULT',
    'OP_MAT_DIV',
    'OP_EXEC_VIRGULA',
    'OP_ATRIB_IGUAL',
    'OP_ATRIB_MAIS_IGUAL',
    'OP_REL_DUPLO_IGUAL',
    'OP_REL_MENOR',
    'OP_REL_MAIOR',
    'OP_FINAL_LINHA_PONTO_VIRGULA',
    'OP_PRIO_ABRE_PARENTESES',
    'OP_PRIO_FECHA_PARENTESES',
    #'OP_PRIO_ABRE_COLCHETES',
    #'OP_PRIO_FECHA_COLCHETES',
    'OP_PRIO_ABRE_CHAVES',
    'OP_PRIO_FECHA_CHAVES',
```

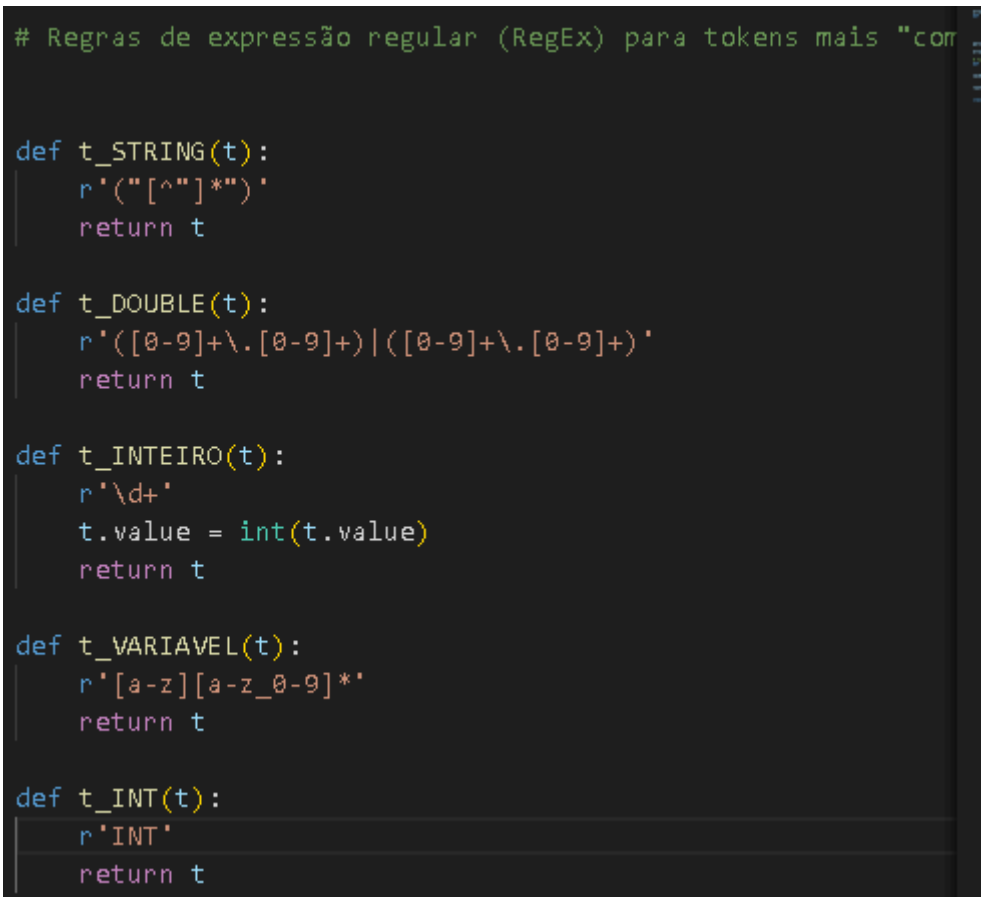
Tokens:

É uma lista que contém os nomes de todos os tokens que o analisador léxico deve reconhecer. Além dos tokens simples, como números e operadores, essa lista também inclui as palavras-chave reservadas.

```
# Regras de expressão regular (RegEx) para tokens simples
t_ENQUANTO = r'ENQUANTO'
t_SE = r'SE'
t_ELSE = r'ELSE'
t_PARA = r'PARA'
t_ESCREVA = r'ESCREVA'
t_LEIA = r'LEIA'
t_EM = r'EM'
t_RANGE = r'RANGE'
t_OP_MAT_ADICAO = r'\+'
t_OP_MAT_SUB = r'-'
t_OP_MAT_MULT = r'\*'
t_OP_MAT_DIV = r'/'
t_OP_FINAL_LINHA_PONTO_VIRGULA = r'\;'
t_OP_EXEC_VIRGULA = r'\,'
t_OP_ATRIB_IGUAL = r'\='
t_OP_ATRIB_MAIS_IGUAL = r'\+\='
t_OP_REL_DUPLO_IGUAL = r'\=\='
t_OP_REL_MENOR = r'\<'
t_OP_REL_MAIOR = r'\>'
t_OP_PRIO_ABRE_PARENTESES = r'\('
t_OP_PRIO_FECHA_PARENTESES = r'\)'
#t_OP_PRIO_ABRE_COLCHETES = r'\['
#t_OP_PRIO_FECHA_COLCHETES = r'\]'
t_OP_PRIO_ABRE_CHAVES = r'\{'
t_OP_PRIO_FECHA_CHAVES = r'\}'

t_ignore = ' \t'  # Ignora espaço e tabulação
```

Regras de expressão regular para tokens simples (t_<TOKEN>):

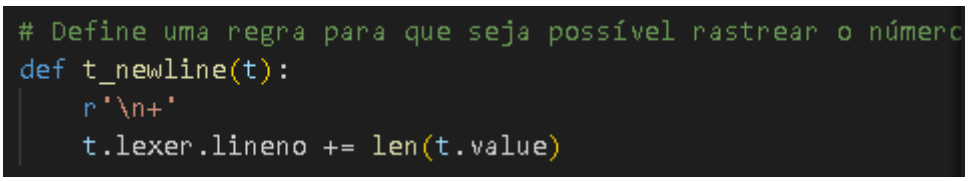
Cada regra define um padrão para um token específico usando expressões regulares. Por exemplo, **t_OP_MAT_ADICAO** define o padrão para o operador de adição **(+)**. Essas regras correspondem a tokens simples que podem ser identificados facilmente pela sua representação literal.



```
# Regras de expressão regular (RegEx) para tokens mais "com


def t_STRING(t):
    r'("[^"]*")'
    return t

def t_DOUBLE(t):
    r'([0-9]+\.[0-9]+)|([0-9]+\.[0-9]+)'
    return t

def t_INTEIRO(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_VARIAVEL(t):
    r'[a-z][a-z_0-9]*'
    return t

def t_INT(t):
    r'INT'
    return t
```

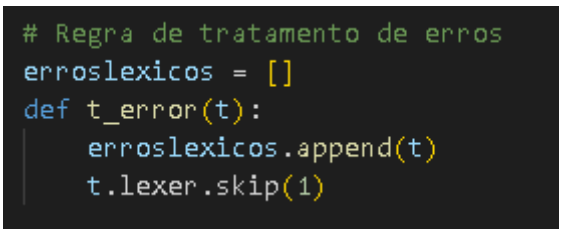
Regras de expressão regular para tokens mais complexos:

Algumas regras definem padrões para tokens mais complexos, como números de ponto flutuante, strings e identificadores de variáveis. Aqui, as expressões regulares são mais elaboradas para capturar esses padrões.



t_newline:

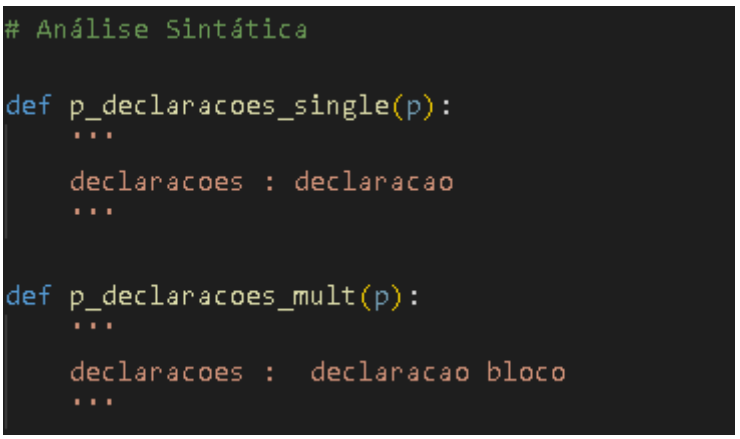
Esta regra é usada para rastrear números de linha. Ela conta o número de ocorrências de caracteres de nova linha (**\n**) para atualizar o contador de linha do analisador léxico.



t_error:

Esta regra define como lidar com erros léxicos. Se o analisador encontrar um caractere que não corresponda a nenhum token definido, esta função será chamada. Ela geralmente pula o caractere e continua a análise.

Os trechos de código a seguir, definem as regras de produção para a gramática de um analisador sintático (parser).



```
# Análise Sintática

def p_declaracoes_single(p):
    '''
    declaracoes : declaracao
    '''

def p_declaracoes_mult(p):
    '''
    declaracoes :  declaracao bloco
    '''
```

def p_declaracoes_single(p)::

Esta função define uma regra de produção chamada **p_declaracoes_single** para a gramática do parser. Essa regra especifica como uma lista de declarações (**declaracoes**) é derivada quando consiste em uma única declaração (**declaracao**).

Na gramática BNF (Backus-Naur Form), a produção é definida como:

declaracoes —----> declaracao

def p_declaracoes_mult(p)::

Esta função define outra regra de produção chamada **p_declaracoes_mult** para a gramática do parser. Essa regra específica como uma lista de declarações (**declaracoes**) é derivada quando consiste em uma declaração (**declaracao**) seguida de um bloco (bloco).

Na gramática BNF, a produção é definida como:

declaracoes —---> declaracao bloco

```
def p_bloco(p):
    '''
    bloco : OP_PRIO_ABRE_CHAVES declaracoes OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES declaracao bloco OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES impressao OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES escrita impressao OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES escrita escrita impressao OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES escrita escrita expr impressao OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES impressao param_cond OP_FINAL_LINHA_PONTO_VIRGULA OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES param_cond OP_FINAL_LINHA_PONTO_VIRGULA impressao OP_PRIO_FECHA_CHAVES
          | OP_PRIO_ABRE_CHAVES impressao expr OP_PRIO_FECHA_CHAVES

    '''
```

Este trecho do código define as regras para um bloco dentro da gramática da linguagem, permitindo várias combinações de declarações, expressões e estruturas de controle delimitadas por chaves.

Vamos analisar cada linha da definição da regra bloco:

1. **bloco : OP_PRIO_ABRE_CHAVES** declaracoes **OP_PRIO_FECHA_CHAVES**: Esta linha define que um bloco pode ser composto por um conjunto de declarações (**declaracoes**) entre chaves **(OP_PRIO_ABRE_CHAVES e OP_PRIO_FECHA_CHAVES**).
2. **bloco : OP_PRIO_ABRE_CHAVES** declaracao bloco **OP_PRIO_FECHA_CHAVES**: Aqui, um bloco pode ser composto por uma declaração (declaracao) seguida por outro bloco (bloco), também delimitado por chaves.
3. **bloco : OP_PRIO_ABRE_CHAVES** impressao **OP_PRIO_FECHA_CHAVES**: Esta linha define que um bloco pode conter uma expressão de impressão (impressao) entre chaves.
4. **bloco : OP_PRIO_ABRE_CHAVES** escrita impressao **OP_PRIO_FECHA_CHAVES**: Aqui, um bloco pode conter uma sequência de uma declaração de escrita (**escrita**) seguida por uma expressão de impressão (**impressao**), ambas entre chaves.

5. **bloco : OP_PRIO_ABRE_CHAVES** escrita escrita impressao **OP_PRIO_FECHA_CHAVES**: Similar ao anterior, mas com duas declarações de

escrita (**escrita**) seguidas por uma expressão de impressão (**impressao**), todas entre chaves.

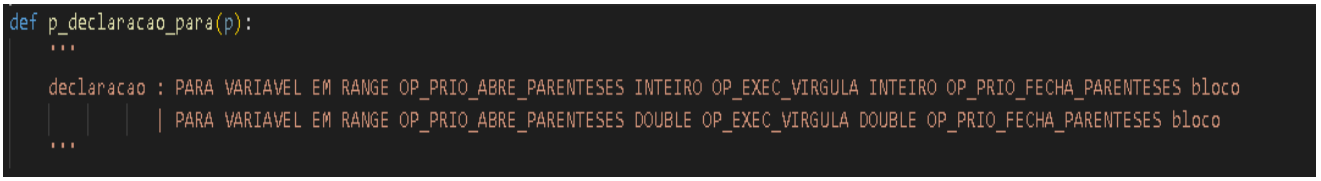
6. **bloco : OP_PRIO_ABRE_CHAVES** escrita escrita expr impressao **OP_PRIO_FECHA_CHAVES**: Nesta linha, um bloco pode conter duas declarações de escrita **(escrita**), uma expressão (**expr**) e uma expressão de impressão (**impressao**), todas entre chaves.
7. **bloco : OP_PRIO_ABRE_CHAVES impressao param_cond OP_FINAL_LINHA_PONTO_VIRGULA OP_PRIO_FECHA_CHAVES**: Aqui, um bloco pode conter uma expressão de impressão **(impressao**), seguida por uma condição (param_cond), finalizando com um ponto e vírgula.
8. **bloco : OP_PRIO_ABRE_CHAVES** param_cond **OP_FINAL_LINHA_PONTO_VIRGULA impressao OP_PRIO_FECHA_CHAVES**: Similar ao anterior, mas com a condição **(param_cond**) precedendo a expressão de impressão (**impressao**).
9. bloco : **OP_PRIO_ABRE_CHAVES** impressao expr **OP_PRIO_FECHA_CHAVES**: Aqui, um bloco pode conter uma expressão de impressão (**impressao**) seguida por uma expressão (**expr**), ambas entre chaves.

```
def p_declaracao_ENQUANTO(p):
    '''
    declaracao : ENQUANTO param_cond bloco
               | declaracao ENQUANTO param_cond bloco
    '''
```

Essa regra **p_declaracao_ENQUANTO** define a estrutura de uma declaração com a palavra-chave **ENQUANTO** (equivalente ao WHILE em muitas linguagens de programação). Ela pode ocorrer de duas formas:

1. **ENQUANTO** seguido por uma condição e um bloco: Isso significa que uma declaração começa com a palavra-chave **ENQUANTO**, seguida por uma condição (**param_cond**) e, em seguida, um bloco de código (bloco). Essa forma básica representa um loop ENQUANTO padrão.
2. Uma declaração existente seguida por **ENQUANTO**, condição e bloco: Isso significa que uma declaração já existente (**declaracao**) é seguida pela palavra-chave **ENQUANTO**, uma condição **(param_cond**) e, finalmente, um bloco de código (bloco). Essa forma permite que múltiplas declarações estejam contidas dentro de um loop ENQUANTO, ou seja, uma estrutura de loop aninhada.

Em ambas as formas, o bloco de código após a palavra-chave ENQUANTO é delimitado por chaves. Essa regra encapsula a estrutura básica de um loop ENQUANTO na gramática da linguagem;



Essa regra **p_declaracao_para** define a estrutura de uma declaração usando a palavra-chave PARA (equivalente ao FOR em muitas linguagens de programação). Ela especifica duas formas de declarações PARA, dependendo do tipo dos valores de iteração:

1. PARA loop com variáveis do tipo inteiro: Nessa forma, a declaração começa com a palavra-chave PARA, seguida pelo nome da variável que será utilizada para a iteração (VARIAVEL), a palavra-chave EM, a palavra-chave reservada RANGE, seguida por parênteses que delimitam o início e o fim do intervalo (**OP_PRIO_ABRE_PARENTESES INTEIRO OP_EXEC_VIRGULA INTEIRO OP_PRIO_FECHA_PARENTESES**), e finalmente um bloco de código (bloco). O intervalo definido pelo RANGE é composto por dois números inteiros.
2. PARA loop com variáveis do tipo ponto flutuante (double): Essa forma é semelhante à primeira, mas os valores do intervalo são números de ponto flutuante (DOUBLE), definidos após a palavra-chave RANGE.

Ambas as formas seguem a mesma estrutura básica, com a única diferença sendo o tipo de valor de iteração permitido. Essa regra encapsula a estrutura de um loop PARA na gramática da linguagem definida.

```
def p_declaracao_atribuicaoValorVariavel(p):
    '''
    declaracao : VARIAVEL OP_ATRIB_IGUAL expr OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL STRING OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL DOUBLE OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL funcao OP_FINAL_LINHA_PONTO_VIRGULA
               | param VARIAVEL OP_ATRIB_IGUAL INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_MAIS_IGUAL INTEIRO
               | VARIAVEL OP_ATRIB_MAIS_IGUAL DOUBLE
               | VARIAVEL OP_ATRIB_MAIS_IGUAL VARIAVEL
    '''
```

Essa regra **p_declaracao_atribuicaoValorVariavel** define várias formas de declarações de atribuição de valores a variáveis.

1. Atribuição de expressão a uma variável: **VARIAVEL OP_ATRIB_IGUAL expr OP_FINAL_LINHA_PONTO_VIRGULA**: Esta regra define a atribuição de uma expressão (expr) a uma variável (VARIAVEL) usando o operador de atribuição **OP_ATRIB_IGUAL** seguido por um ponto e vírgula **OP_FINAL_LINHA_PONTO_VIRGULA**. A expressão pode ser uma operação matemática, uma chamada de função ou qualquer outra expressão válida na linguagem.
2. Atribuição de uma string a uma variável: **VARIAVEL OP_ATRIB_IGUAL STRING OP_FINAL_LINHA_PONTO_VIRGULA**: Aqui, uma string é atribuída a uma variável.
3. Atribuição de uma variável a outra variável: **VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA**: Isso permite a atribuição de uma variável a outra.
4. Atribuição de um valor inteiro a uma variável: **VARIAVEL OP_ATRIB_IGUAL INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA**: Permite atribuir um valor inteiro diretamente a uma variável.
5. Atribuição de um valor de ponto flutuante (double) a uma variável: **VARIAVEL OP_ATRIB_IGUAL DOUBLE OP_FINAL_LINHA_PONTO_VIRGULA**: Similar ao anterior, mas para valores de ponto flutuante.
6. Atribuição do resultado de uma função a uma variável: **VARIAVEL OP_ATRIB_IGUAL funcao OP_FINAL_LINHA_PONTO_VIRGULA**: Permite atribuir o resultado de uma função a uma variável.
7. Atribuição de um valor inteiro a uma variável usando um parâmetro antes da variável: param **VARIAVEL OP_ATRIB_IGUAL INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA:** Similar à forma anterior, mas com um parâmetro antes da variável.
8. Atribuição aditiva de um valor inteiro a uma variável: **VARIAVEL OP_ATRIB_MAIS_IGUAL INTEIRO**: Define uma atribuição aditiva, onde um valor inteiro é adicionado ao valor atual da variável.

9. Atribuição aditiva de um valor de ponto flutuante (double) a uma variável: **VARIAVEL OP_ATRIB_MAIS_IGUAL DOUBLE**: Similar ao anterior, mas para valores de ponto flutuante.
10. Atribuição aditiva de uma variável a outra variável: **VARIAVEL OP_ATRIB_MAIS_IGUAL VARIAVE**L: Isso permite a atribuição aditiva de uma variável a outra.

```
def p_declaracao_condicionais(p):
    '''
    declaracao : SE param_cond bloco
               | declaracao SE param_cond bloco
               | declaracao SE param_cond bloco senao
               | SE param_cond bloco senao
    '''
```

Essa regra **p_declaracao_condicionais** define várias formas de declarações condicionais utilizando a estrutura SE (if-else) em uma linguagem de programação. Aqui está a explicação detalhada de cada uma das formas definidas:

1. Declaração condicional simples: **SE param_cond bloco:** Esta regra define uma estrutura condicional simples, onde o bloco de código é executado se a condição especificada em param_cond for verdadeira.
2. Declaração condicional encadeada: **declaracao SE param_cond bloco**: Esta regra permite a aninhamento de estruturas condicionais. O bloco de código associado é executado se a condição em **param_cond** for verdadeira, e o restante do código que segue a estrutura condicional também é avaliado.
3. Declaração condicional com cláusula **senao**: **declaracao SE param_cond** bloco **senao**: Esta regra estende a declaração condicional encadeada, adicionando uma cláusula **senao**. Isso significa que se a condição em **param_cond** não for atendida, o bloco de código após a cláusula **senao** será executado.
4. Declaração condicional simples com cláusula **senao**: **SE param_cond** bloco **senao**: Esta regra define uma estrutura condicional simples com uma cláusula **senao**. Se a condição em param_cond for verdadeira, o bloco de código após o SE será executado; caso contrário, o bloco de código após o **senao** será executado.

Essas regras permitem definir diferentes tipos de lógica condicional em um programa, desde estruturas simples **SE-SENÃO** até aninhamentos complexos de múltiplas condições.

```
def p_declaracao_funcao_invocada(p):
    '''
    declaracao : funcao OP_FINAL_LINHA_PONTO_VIRGULA
               | impressao
               | escrita
    '''
```

Essa regra **p_declaracao_funcao_invocada** define diferentes tipos de declarações que podem ocorrer em um programa.

Declaração de função invocada: **funcao OP_FINAL_LINHA_PONTO_VIRGULA**: Esta regra indica que uma função está sendo invocada, e o resultado ou efeito da função será terminado com um ponto e vírgula (;). Isso implica que a função será executada e seu resultado será tratado conforme necessário.

1. Declaração de impressão: **impressao**: Esta regra indica que uma declaração de impressão está ocorrendo. Ela geralmente envolve a exibição de algum tipo de saída na tela, como texto ou valores de variáveis.
2. Declaração de **escrita: escrita**: Esta regra indica que uma declaração de escrita está ocorrendo. Similar à impressão, a escrita pode envolver a exibição de dados, mas pode ter um propósito mais amplo, como gravar em um arquivo ou dispositivo externo.

Essas regras fornecem flexibilidade na definição de declarações em um programa, abrangendo desde a execução de funções até a saída de dados para o usuário.

```
def p_declaracao_definir_funcao(p):
    '''
    declaracao : funcao OP_PRIO_ABRE_CHAVES declaracoes OP_PRIO_FECHA_CHAVES
    '''
```

Essa regra **p_declaracao_definir_funcao** define a declaração de uma função dentro do código. Aqui está a explicação detalhada:

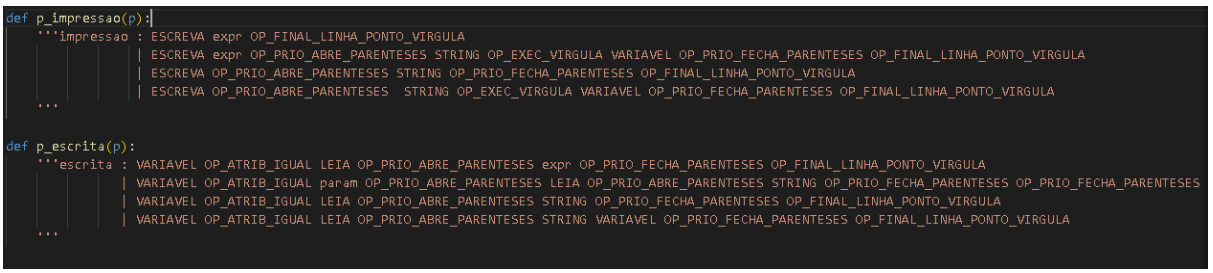
1. funcao: funcao é um não-terminal que representa a definição de uma função no código. Uma função é um bloco de código que pode ser chamado em outros lugares do programa para realizar uma tarefa específica.
2. **OP_PRIO_ABRE_CHAVES: OP_PRIO_ABRE_CHAVES** é um token que representa a abertura de chaves ({). No contexto da declaração de uma função, isso marca o início do bloco de código que compõe a função.
3. declaracoes: declaracoes é um não-terminal que representa as declarações dentro do bloco de código da função. Essas declarações podem incluir variáveis, estruturas de controle, chamadas de função, entre outras coisas.
4. **OP_PRIO_FECHA_CHAVES: OP_PRIO_FECHA_CHAVES** é um token que representa o fechamento de chaves (}). No contexto da declaração de uma função, isso marca o fim do bloco de código da função.

Essa regra indica que uma função está sendo definida, seguida por um bloco de código delimitado por chaves que compõe o corpo da função. Este corpo da função pode conter qualquer número de declarações válidas para a linguagem.

```
def p_parametro_condicional(p):
    '''
    param_cond :  VARIAVEL OP_REL_MENOR INTEIRO
                | VARIAVEL OP_REL_MENOR DOUBLE
                | VARIAVEL OP_REL_MENOR VARIAVEL
                | VARIAVEL OP_REL_MAIOR INTEIRO
                | VARIAVEL OP_REL_MAIOR DOUBLE
                | VARIAVEL OP_REL_MAIOR VARIAVEL
                | VARIAVEL OP_ATRIB_MAIS_IGUAL INTEIRO
                | VARIAVEL OP_ATRIB_MAIS_IGUAL DOUBLE
                | VARIAVEL OP_ATRIB_MAIS_IGUAL VARIAVEL
                | VARIAVEL OP_REL_DUPLO_IGUAL INTEIRO
                | VARIAVEL OP_REL_DUPLO_IGUAL DOUBLE
                | VARIAVEL OP_REL_DUPLO_IGUAL VARIAVEL

    '''
```

Essa regra **p_parametro_condicional** define os parâmetros condicionais que podem ser usados em expressões condicionais (por exemplo, em declarações **SE**).

1. **param_cond**: param_cond é um não-terminal que representa um parâmetro condicional em uma expressão condicional, como em uma declaração **SE**.
2. **VARIAVEL: VARIAVEL** é um token que representa uma variável. Variáveis são usadas para armazenar valores na memória do programa.
3. **OP_REL_MENOR: OP_REL_MENOR** é um token que representa o operador de comparação "menor que" (**<**). Ele é usado para verificar se o valor de uma variável é menor que outro valor.
4. **OP_REL_MAIOR: OP_REL_MAIOR** é um token que representa o operador de comparação "maior que" (**>**). Ele é usado para verificar se o valor de uma variável é maior que outro valor.
5. **OP_ATRIB_MAIS_IGUAL: OP_ATRIB_MAIS_IGUAL** é um token que representa o operador de atribuição e adição combinados (**+=**). Ele é usado para adicionar um valor a uma variável.
6. **OP_REL_DUPLO_IGUAL: OP_REL_DUPLO_IGUAL** é um token que representa o operador de comparação de igualdade (**==**). Ele é usado para verificar se dois valores são iguais.
7. **INTEIRO: INTEIRO** é um token que representa um valor inteiro. Inteiros são números sem parte fracionária.
8. **DOUBLE: DOUBLE** é um token que representa um valor de ponto flutuante. Doubles são números que podem conter parte fracionária.



```
def p_impressao(p):
    '''impressao : ESCREVA expr OP_FINAL_LINHA_PONTO_VIRGULA
                 | ESCREVA expr OP_PRIO_ABRE_PARENTESES STRING OP_EXEC_VIRGULA VARIAVEL OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
                 | ESCREVA OP_PRIO_ABRE_PARENTESES STRING OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
                 | ESCREVA OP_PRIO_ABRE_PARENTESES  STRING OP_EXEC_VIRGULA VARIAVEL OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
    '''

def p_escrita(p):
    '''escrita : VARIAVEL OP_ATRIB_IGUAL LEIA OP_PRIO_ABRE_PARENTESES expr OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL param OP_PRIO_ABRE_PARENTESES LEIA OP_PRIO_ABRE_PARENTESES STRING OP_PRIO_FECHA_PARENTESES OP_PRIO_FECHA_PARENTESES
               | VARIAVEL OP_ATRIB_IGUAL LEIA OP_PRIO_ABRE_PARENTESES STRING OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
               | VARIAVEL OP_ATRIB_IGUAL LEIA OP_PRIO_ABRE_PARENTESES STRING VARIAVEL OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA
    '''
```

Essas duas regras **p_impressao** e **p_escrita** definem como as instruções de impressão e escrita são estruturadas na gramática.

Regra p_impressao:

1. impressao: impressao é um não-terminal que representa uma instrução de impressão. Pode ser uma instrução simples de impressão ou uma instrução de impressão formatada.
2. **ESCREVA: ESCREVA** é um token que indica a instrução de impressão.
3. expr: expr é um não-terminal que representa uma expressão que será impressa.
4. **OP_FINAL_LINHA_PONTO_VIRGULA: OP_FINAL_LINHA_PONTO_VIRGULA** é um token que representa o final da instrução de impressão, indicado por um ponto e vírgula.
5. **OP_PRIO_ABRE_PARENTESES: OP_PRIO_ABRE_PARENTESES** é um token que representa a abertura de parênteses para agrupar argumentos opcionais na instrução de impressão formatada.
6. **STRING: STRING** é um token que representa uma string, usada em instruções de impressão formatada para fornecer um formato de saída específico.
7. **VARIAVEL: VARIAVEL** é um token que representa uma variável que será impressa.
8. **OP_PRIO_FECHA_PARENTESES: OP_PRIO_FECHA_PARENTESES** é um token que representa o fechamento de parênteses.

A regra **p_impressao** define diferentes formas de instruções de impressão, incluindo instruções simples de impressão **(ESCREVA expr OP_FINAL_LINHA_PONTO_VIRGULA)** e instruções de impressão formatada **(ESCREVA expr OP_PRIO_ABRE_PARENTESES STRING OP_EXEC_VIRGULA VARIAVEL OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA)**.

Regra p_escrita:

1. **escrita**: escrita é um não-terminal que representa uma instrução de escrita. Ela indica que um valor de entrada será atribuído a uma variável.
2. **VARIAVEL: VARIAVEL** é um token que representa a variável que receberá o valor de entrada.
3. **OP_ATRIB_IGUAL: OP_ATRIB_IGUAL** é um token que representa o operador de atribuição (=).
4. **LEIA: LEIA** é um token que indica a instrução de leitura de entrada do usuário.
5. **OP_PRIO_ABRE_PARENTESES: OP_PRIO_ABRE_PARENTESES** é um token que representa a abertura de parênteses para agrupar argumentos opcionais na instrução de escrita formatada.
6. **STRING: STRING** é um token que representa uma string usada em instruções de escrita formatada para fornecer um prompt específico para o usuário.
7. **OP_PRIO_FECHA_PARENTESES: OP_PRIO_FECHA_PARENTESES** é um token que representa o fechamento de parênteses.

A regra **p_escrita** define diferentes formas de instruções de escrita, incluindo instruções simples de escrita **(VARIAVEL OP_ATRIB_IGUAL LEIA OP_PRIO_ABRE_PARENTESES expr OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA)** e instruções de escrita formatada **(VARIAVEL OP_ATRIB_IGUAL LEIA OP_PRIO_ABRE_PARENTESES STRING OP_PRIO_FECHA_PARENTESES OP_FINAL_LINHA_PONTO_VIRGULA)**.

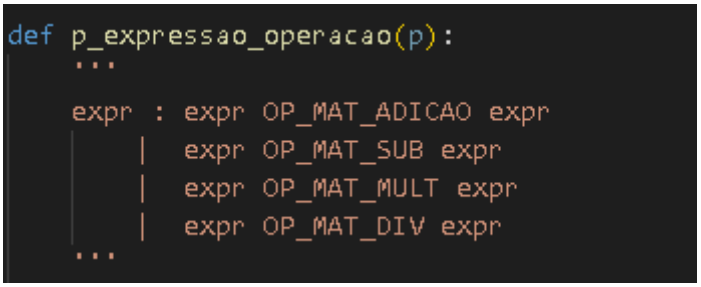
```
def p_expressao_variavel(p):
    '''
    expr :  VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_ADICAO INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_ADICAO VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_SUB INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_SUB VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_MULT INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_MULT VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_DIV INTEIRO OP_FINAL_LINHA_PONTO_VIRGULA
         |  VARIAVEL OP_ATRIB_IGUAL VARIAVEL OP_MAT_DIV VARIAVEL OP_FINAL_LINHA_PONTO_VIRGULA
    '''
```

Essa regra **p_expressao_variavel** define várias formas de expressões que envolvem variáveis. Vamos analisar cada uma delas:

1. expr: expr é um não-terminal que representa uma expressão.
2. **VARIAVEL: VARIAVEL** é um token que representa uma variável.
3. **OP_FINAL_LINHA_PONTO_VIRGULA: OP_FINAL_LINHA_PONTO_VIRGULA** é um token que indica o final da expressão, marcado por um ponto e vírgula.
4. **OP_ATRIB_IGUAL: OP_ATRIB_IGUAL** é um token que representa o operador de atribuição **(=)**.
5. **OP_MAT_ADICAO, OP_MAT_SUB, OP_MAT_MULT, OP_MAT_DIV**: Estes são tokens que representam os operadores aritméticos de adição **(+)**, subtração **(-)**, multiplicação **(*)** e divisão **(/)**, respectivamente.
6. **INTEIRO: INTEIRO** é um token que representa um valor inteiro.

Cada produção na regra **p_expressao_variavel** descreve uma expressão diferente envolvendo variáveis e operadores aritméticos:

- A primeira produção simplesmente atribui o valor de uma variável a outra variável.
- As produções subsequentes envolvem operações aritméticas entre variáveis e valores inteiros ou entre variáveis entre si. Cada operação aritmética é seguida pelo operador de atribuição **(=)** e pelo ponto e vírgula **(;)** para indicar o final da instrução.



```
def p_expressao_operacao(p):
    '''
    expr : expr OP_MAT_ADICAO expr
         | expr OP_MAT_SUB expr
         | expr OP_MAT_MULT expr
         | expr OP_MAT_DIV expr
    '''
```

Essa regra p_expressao_operacao define as operações aritméticas básicas entre expressões.

1. expr: expr é um não-terminal que representa uma expressão.
2. **OP_MAT_ADICAO, OP_MAT_SUB, OP_MAT_MULT, OP_MAT_DIV**: Estes são tokens que representam os operadores aritméticos de adição **(+)**, subtração **(-)**, multiplicação **(*)** e divisão **(/)**, respectivamente.

As produções nesta regra definem expressões compostas por duas expressões, separadas por um operador aritmético. Por exemplo:

- **expr OP_MAT_ADICAO expr**: Esta produção define uma expressão que é a soma de duas outras expressões.
- **expr OP_MAT_SUB expr**: Esta produção define uma expressão que é a subtração de uma expressão por outra.
- **expr OP_MAT_MULT expr**: Esta produção define uma expressão que é o produto de duas outras expressões.
- **expr OP_MAT_DIV expr**: Esta produção define uma expressão que é o resultado da divisão de uma expressão pela outra.

Essas produções permitem construir expressões aritméticas mais complexas, onde as operações podem ser aninhadas e a precedência dos operadores é mantida de acordo com as regras matemáticas padrão.

```
def p_parametro_vazio(p):
    '''
    param_vazio :
    '''
```

Essa regra **p_parametro_vazio** define um parâmetro vazio. Vamos analisar:

1. **param_vazio**: **param_vazio** é um não-terminal que representa um parâmetro vazio, ou seja, um parâmetro que não possui nenhum valor ou conteúdo.
2. Produção vazia: A regra consiste em uma produção vazia, indicada pela ausência de qualquer símbolo após os dois pontos (:). Isso significa que não há nenhum token ou não-terminal associado à regra.

Essa regra é útil em situações onde você precisa definir um parâmetro que pode estar presente ou ausente, e no caso de estar ausente, não requer nenhum processamento adicional.

```
def p_parametro(p):
    '''
    param : INTEIRO
          | INT
          | DOUBLE
          | STRING
          | VARIAVEL
    '''
```

Essa regra **p_parametro** define os diferentes tipos de parâmetros que podem ser utilizados em uma expressão. Vamos entender:

1. param: param é um não-terminal que representa um parâmetro em uma expressão.
2. Produções alternativas: A regra possui múltiplas produções alternativas, separadas pelo caractere de barra vertical (|). Isso significa que um parâmetro pode ser qualquer um dos tipos definidos nas produções.
    - **INTEIRO**: Representa um parâmetro do tipo **inteiro**.
    - **INT**: Uma abreviação para "inteiro", pode ser uma convenção ou um tipo específico usado na gramática.
    - **DOUBLE**: Representa um parâmetro do tipo ponto flutuante (**double**).
    - **STRING**: Representa um parâmetro do tipo **string**.
    - **VARIAVEL**: Representa um parâmetro que é uma **variável**.

Essa regra captura os diferentes tipos de dados que podem ser utilizados como parâmetros em uma expressão na linguagem que está sendo definida pela gramática.

```
def p_regra_funcao(p):
    '''
    funcao :  OP_PRIO_ABRE_PARENTESES param_vazio OP_PRIO_FECHA_PARENTESES
        |  OP_PRIO_ABRE_PARENTESES param OP_PRIO_FECHA_PARENTESES
    '''
```

Essa regra **p_regra_funcao** define a estrutura de uma função na gramática.

1. funcao: funcao é um não-terminal que representa uma função na gramática.
2. Produções alternativas: A regra possui duas produções alternativas separadas pelo caractere de barra vertical (|), o que significa que uma função pode ser definida de duas maneiras diferentes:
    - Sem parâmetros: **OP_PRIO_ABRE_PARENTESES** param_vazio **OP_PRIO_FECHA_PARENTESES**: Nesse caso, a função é definida sem parâmetros. Isso significa que não há nada entre os parênteses.
    - Com parâmetros: **OP_PRIO_ABRE_PARENTESES** param **OP_PRIO_FECHA_PARENTESES**: Aqui, a função é definida com pelo menos um parâmetro. A definição do parâmetro é especificada na regra param.

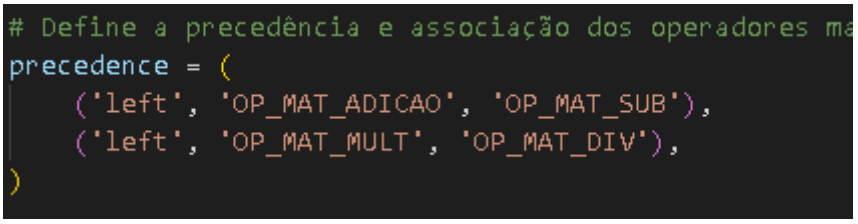
Portanto, essa regra permite definir funções tanto com quanto sem parâmetros na linguagem definida pela gramática.

```
def p_senao_se(p):
    '''
    senao : ELSE bloco
    '''
```

Essa regra **p_senao_se** define a estrutura de um bloco de código que deve ser executado caso a condição de um bloco SE não seja satisfeita. **senao: senao** é um não-terminal que representa um bloco de código que deve ser executado caso a condição de um bloco SE não seja satisfeita.

1. Produção: A regra possui apenas uma produção, que começa com a palavra-chave **ELSE** seguida pelo não-terminal bloco, indicando que este bloco de código será executado caso a condição do bloco **SE** associado não seja verdadeira.

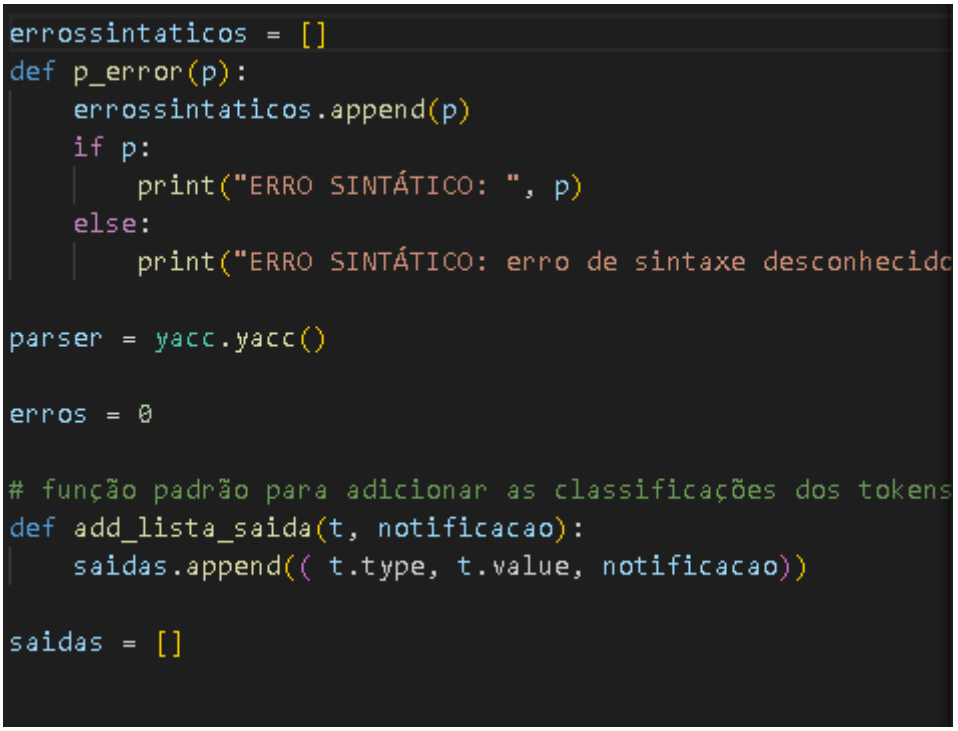
Portanto, essa regra permite a definição de um bloco de código a ser executado quando a condição de um bloco SE não é atendida, ou seja, quando a condição avalia como falsa.



Essa parte do código define a precedência e a associação dos operadores matemáticos na expressão que está sendo analisada pelo parser.

1. **precedence**: Este é um objeto que define a precedência dos operadores. É uma sequência de tuplas, onde cada tupla representa um nível de precedência. No exemplo dado, há dois níveis de precedência definidos.
2. (**'left', 'OP_MAT_ADICAO', 'OP_MAT_SUB**'): Esta tupla define o primeiro nível de precedência. Aqui, **OP_MAT_ADICAO** e OP_MAT_SUB têm a mesma precedência e associatividade à esquerda. Isso significa que, quando esses operadores aparecem em sequência na expressão, eles são avaliados da esquerda para a direita.
3. (**'left', 'OP_MAT_MULT', 'OP_MAT_DIV**'): Esta segunda tupla define o segundo nível de precedência. Aqui, **OP_MAT_MULT** e **OP_MAT_DIV** têm a mesma precedência e associatividade à esquerda. Da mesma forma que no primeiro nível, isso significa que, quando esses operadores aparecem em sequência na expressão, eles são avaliados da esquerda para a direita.

Essa definição de precedência e associação é crucial para garantir que as expressões matemáticas sejam avaliadas corretamente pelo parser, seguindo as regras usuais de precedência de operadores matemáticos.



```
errossintaticos = []
def p_error(p):
    errossintaticos.append(p)
    if p:
        print("ERRO SINTÁTICO: ", p)
    else:
        print("ERRO SINTÁTICO: erro de sintaxe desconhecidc

parser = yacc.yacc()

erros = 0

# função padrão para adicionar as classificações dos tokens
def add_lista_saida(t, notificacao):
    saidas.append(( t.type, t.value, notificacao))

saidas = []
```

Este trecho de código adiciona uma função para lidar com erros sintáticos durante o processo de parsing e cria um objeto parser usando o módulo yacc.

1. **errossintaticos = []**: É uma lista que será usada para armazenar os erros sintáticos encontrados durante o parsing.
2. **def p_error(p)**: ...: Esta é uma função que será chamada quando ocorrer um erro sintático durante o parsing. Ela recebe o token **p** que causou o erro. Dentro dessa função:
    - O token p é adicionado à lista errossintaticos.
    - Se o token **p existir** (não for None), imprime uma mensagem de erro indicando o tipo de erro sintático.
    - Se o **token p for** None, significa que ocorreu um erro de sintaxe desconhecido.
3. **parser = yacc.yacc()**: Aqui, um objeto parser é criado usando o módulo yacc. Este parser será responsável por analisar a gramática definida anteriormente e gerar a árvore de análise sintática correspondente.
4. **erros = 0:** Esta variável é inicializada com zero. Provavelmente será usada para rastrear o número total de erros encontrados durante o parsing.
5. **def add_lista_saida(t, notificacao)**: ...: Esta parece ser uma função genérica que adiciona classificações de tokens para serem impressas pelo compilador. No entanto, seu uso não é mostrado neste trecho de código.
6. **saidas = []**: É uma lista que provavelmente será usada para armazenar saídas classificadas pelos tokens durante o processo de parsing.

Essas regras são essenciais para definir a estrutura da gramática e orientar o parser sobre como interpretar sequências de tokens durante a análise sintática. Eles descrevem a estrutura hierárquica das declarações em um programa, indicando como uma lista de declarações pode ser construída a partir de suas partes constituintes.

```
def chama_analisador(self):
    columns = ( 'token', 'lexema', 'notificacao')
    self.saida = ttk.Treeview(self.frame_2, height=5, columns=columns, show='he
    self.saida.heading("token", text='Token')
    self.saida.heading("lexema", text='Lexema')
    self.saida.heading("notificacao", text='Notificacao')

    data = self.codigo_entry.get(1.0, "end-1c")
    lexer = lex.lex()
    lexer.input(data)

    print("Análise Léxica:")
    # Tokenizar a entrada para passar para o analisador léxico
    for tok in lexer:
        print("Token:", tok.type)
        print("Valor:", tok.value)
        global erros
        if tok.type == "INTEIRO":
            max = (len(str(tok.value)))
            if max > 15:
                erros += 1
                add_lista_saida(tok, f"entrada maior que a suportada")
            else:
                add_lista_saida(tok, f" ")
        elif tok.type == "SE" or tok.type == "ELSE" or tok.type == "ENQUANTO" o
            max = len(tok.value)
            if max < 20:
                if tok.value in reserved:
                    tok.type = reserved[tok.value]
                    add_lista_saida(tok, f"palavra reservada")
                else:
                    add_lista_saida(tok, f" ")
        else:
            add_lista_saida(tok, f" ")

    for tok in erroslexicos:
        add_lista_saida(tok, f"Caracter Inválido na linguagem")

    tamerroslex = len(erroslexicos)
    if tamerroslex == 0 and erros == 0:
        print("Análise Léxica Concluída sem Erros")
        parser.parse(data)
        tamerrosin = len(errossintaticos)
        if tamerrosin == 0:
            print("Análise Sintática Concluída sem Erros")
        else:
            print("Erro Sintático")
    else:
        print("Erro Léxico")

    for retorno in saidas:
        self.saida.insert('', tk.END, values=retorno)
```

Esse método **chama_analisador** é responsável por realizar a análise léxica e sintática de um código fonte fornecido como entrada.

1. Definição da estrutura de saída: O método cria uma tabela de árvore ttk para exibir os resultados da análise. Esta tabela terá três colunas: 'token', 'lexema' e 'notificação'.
2. Obtenção do código-fonte: O método extrai o código-fonte a ser analisado do widget **codigo_entry**.
3. Análise Léxica:
    - Um objeto lexer é criado usando **lex.lex()**.
    - O código-fonte é fornecido ao **lexer** através de **lexer.input(data)**.
    - Cada token é iterado usando um loop **for tok in lexer**:.
    - Para cada token, verifica-se se é um token numérico (**INTEIRO**) e se seu tamanho excede um limite. Se exceder, é registrado um erro e adicionado à lista de saída com uma notificação apropriada.
    - Se o token for uma palavra reservada, ela é identificada e seu tipo é ajustado de acordo. O token é então adicionado à lista de saída com uma notificação apropriada.
    - Se não for um token numérico nem uma palavra reservada, é adicionado à lista de saída com uma notificação vazia.
4. Tratamento de erros léxicos: Os tokens inválidos são iterados a partir da lista **erroslexicos** e cada um é adicionado à lista de saída com uma notificação indicando um erro léxico.
5. Verificação de erros léxicos e sintáticos: O número de erros léxicos e sintáticos é verificado. Se não houver erros de nenhum tipo, a análise sintática é realizada chamando **parser.parse(data)**. Em seguida, é verificado se houve erros sintáticos. Dependendo do resultado, são exibidas mensagens apropriadas.

Esse método é parte central do analisador, onde as etapas de análise léxica e sintática são realizadas, e os resultados são exibidos na interface do usuário.

```
def transpilar_codigo(self):
    codigo_fonte = self.codigo_entry.get(1.0, tk.END)
    codigo_transpilado = self.transpilar_para_python(codigo_fonte)
    self.mostrar_codigo_transpilado(codigo_transpilado)
```

Este método **transpilar_codigo** é responsável por transpilar o código fonte fornecido para Python e exibir o código transpilado. A

Obtenção do código-fonte: O método extrai o código-fonte a ser transpilado do widget **codigo_entry**.

1. Transpilação para Python: O código-fonte é passado para o método **transpilar_para_python**, que aparentemente é uma função ou método que converte o código-fonte para Python. O resultado da transpilação é armazenado na variável **codigo_transpilado**.
2. Exibição do código transpilado: O método **mostrar_codigo_transpilado** é chamado para exibir o código transpilado na interface do usuário.

Este método é parte de um processo de transpilação de uma linguagem de programação personalizada para Python, possibilitando a execução do código escrito na linguagem personalizada em um ambiente Python.

```python
def transpilar_para_python(self, codigo_fonte):
    # Remover o ponto e vírgula no final da linha e espaços em branco subseq
    codigo_fonte = codigo_fonte.replace(';', '').rstrip()

    # Substituir PARA por for
    codigo_fonte = codigo_fonte.replace('PARA', 'for')

    codigo_fonte = codigo_fonte.replace('ELSE', 'else')

    # Substituir LEIA por input
    codigo_fonte = codigo_fonte.replace('LEIA', 'input')

    # Substituir SE por if e ELSE por else
    codigo_fonte = codigo_fonte.replace('SE', 'if')

    # Substituir ESCREVA por print
    codigo_fonte = codigo_fonte.replace('ESCREVA', 'print')

    # Substituir chaves por indentação
    codigo_fonte = codigo_fonte.replace('{', '').replace('}', '')

    # Remover ':' após a linha 'x = 11'
    codigo_fonte = codigo_fonte.replace('x = 11\n:', 'x = 11\n')

    # Substituir acentos
    codigo_fonte = codigo_fonte.replace('eh', 'é')

    # Corrigir formatação da string dentro da função ESCREVA
    in_string = False
    temp = ''
    word = ''
    for char in codigo_fonte:
        if char == '"':
            in_string = not in_string
        if not in_string:
            if char.isalpha():
                word += char
            else:
                if word == 'RANGE':
                    temp += 'range'
                elif word == 'EM':
                    temp += 'in'
                elif word == 'INT':
                    temp += 'int'
                else:
                    temp += word
                word = ''
                temp += char
        else:
            temp += char
    codigo_fonte = temp
```

Este método **transpilar_para_python** realiza várias substituições e manipulações no código fonte fornecido para convertê-lo em código Python equivalente.

1. Remoção de ponto e vírgula: Remove os pontos e vírgulas no final de cada linha, pois o Python não usa ponto e vírgula como delimitadores de linha.
2. Substituição de palavras-chave:
    - **PARA** é substituído por **for**, para loops.
    - **ELSE** é substituído por **else**, para estruturas condicionais.
    - **LEIA** é substituído por **input**, para entrada de dados.
    - **SE** é substituído por **if**, para estruturas condicionais.
    - **ESCREVA** é substituído por **print**, para saída de dados.
3. Manipulação de chaves:
    - Remove chaves (**{}**), possivelmente para se adequar à sintaxe de indentação do Python.
    - Realiza substituições de caracteres acentuados (**'eh'** por **'é'**), possivelmente para corrigir caracteres acentuados.
4. Correção de formatação de strings:
    i. O código itera por cada caractere no código fonte fornecido.
    ii. Para cada caractere:
        1. Se o caractere for uma aspa dupla **(")**, isso indica o início ou o fim de uma string. A variável **in_string** é alternada entre **True** e **False**.
        2. Se não estiver dentro de uma string:
            a. Se o caractere for alfabético (**char.isalpha()**), ele é adicionado à palavra temporária (**word**).
            b. Caso contrário, se a palavra temporária for uma palavra-chave como '**RANGE', 'EM' ou 'INT**', ela é substituída por sua forma correta (**'range', 'in' ou 'int'**) e adicionada ao código temporário (temp).
        3. Se estiver dentro de uma string, o caractere é simplesmente adicionado ao código temporário (temp).
5. Atualização do Código Fonte:
    - Após o loop, o código fonte é atualizado para o valor armazenado em **temp**, que é o código corrigido.

```
# transpilar a estrutura ENQUANTO
codigo_fonte = codigo_fonte.replace('ENQUANTO', 'while')

# Transpilar atribuição de valor a uma variável
codigo_fonte = codigo_fonte.replace('OP_ATRIB_IGUAL', '=')

# Corrigir indentação
linhas = codigo_fonte.split('\n')
for i in range(len(linhas)):
    linha = linhas[i].strip()
    if linha.startswith(('if', 'else', 'while', 'for')):
        # Adicionar ':' se ainda não estiver presente
        if not linha.endswith(':'):
            linhas[i] = linha + ':'
    # Corrigir indentação do bloco else
    elif linha.startswith('else'):
        if i > 0 and linhas[i-1].strip().endswith(':'):
            linhas[i] = ' ' + linha
        else:
            if i+1 < len(linhas) and not linhas[i+1].strip().startswith('pr
                linhas[i+1] = '    ' + linhas[i+1]
                linhas[i] = ' ' + linha

codigo_fonte = '\n'.join(linhas)

return codigo_fonte
```

Este trecho de código é responsável por mais algumas etapas na transpilação do código fornecido para Python. Aqui está uma explicação detalhada do que está acontecendo:

1. Transpilação da Estrutura **ENQUANTO** para **while**:
    - Todas as ocorrências da palavra-chave '**ENQUANTO**' são substituídas por **'while'**, que é a estrutura de repetição equivalente em Python.
2. Transpilação da Atribuição de Valor para uma Variável:
    - Todas as ocorrências do operador '**OP_ATRIB_IGUAL**' são substituídas por **'='**, que é o operador de atribuição em Python.
3. Correção da Indentação:
    - O código é dividido em linhas usando a função **split('\n')**.
    - Para cada linha:
        - Se a linha começar com **'if'**, **'else'**, **'while'** ou **'for'**, isso indica o início de um bloco de código condicional ou de repetição. Então, verifica-se se a linha termina com **':'**. Se não, adiciona-se **' : '** para indicar o início do bloco.
        - Se a linha começar com **'else'**, verifica-se se a linha anterior termina com **':'**. Se sim, a linha é indentada com um espaço. Se não, a próxima linha é indentada com quatro espaços.
    - As linhas modificadas são unidas de volta em uma única string usando **'\n'**.

Essas etapas garantem que a estrutura do código transpilado seja válida em Python, com a formatação correta e os operadores substituídos adequadamente. Isso prepara o código para ser executado corretamente em um ambiente Python.

Essas substituições e manipulações são feitas para garantir que o código fonte na nossa linguagem seja convertido em código Python válido e funcional.