Análise Léxica

Juan Felipe S. Rangel[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 135 – 87.301-899 – Campo Mourão – PR – Brasil

juanrangel15@gmail.com

Abstract. *The purpose of this report is to describe the lexical analysis process of a T++ source code, identifying all tokens contained in the test files. In the code provided by the professor, the* `PLY` *[Beazley] library was used to implement the scanning process, generating as output a list of tokens, as can be seen at the end of this document. Finally tests were performed to ensure that all tokens had been correctly identified, given a source code as input.*

Resumo. *O objetivo deste relatório é descrever o processo de análise léxica de um código-fonte em T++, identificando todos tokens contidos nos arquivos de teste. No código disponibilizado pelo professor, a biblioteca* `PLY`*[Beazley] foi utilizada para implementação do processo de varredura, gerando como saída uma lista de tokens, como poderá ser observado ao fim deste documento. Por fim, testes foram realizados para garantir que todos os tokens haviam sido identificados corretamente, dados um código fonte como entrada.*

1. Informações Gerais sobre implementação do Analisador Léxico

Neste artigo será utilizado a linguagem `T++`, a qual foi desenvolvida com propósito educacional, com o objetivo de contribuir para o aprendizado do processo de Análise Léxica. Para a construção de um analisador léxico será visto que um autômato finito é utilizado, especificando através de expressões regulares os tokens contidos em uma linguagem, assim como o que pode haver entre esse tokens, como espaços, que não serão considerados, construindo então o **sistema de varredura**.

2. Linguagem T++

A linguagem `T++` apresenta um estrutura simplificada, utilizando estruturas descritivas, em português. Primeiramente serão vistas as estruturas condicionais contidas nesta linguagem, sendo representadas por `"se"`, `"então"` e `"senão"`. Abaixo é possível encontrar um exemplo de sua utilização (Listing 1).

Listing 1. Código em T++ utilizando "se"

```
# para poder contar as quebras de linha dentro dos comentarios
def t_COMENTARIO(token):
    r"(\{((.|\n)*?)\})"
    token.lexer.lineno += token.value.count("\n")

# Expressao regular para identificacao da quebra de uma linha
def t_newline(token):
    r"\n+"
    token.lexer.lineno += len(token.value)
```

Outras estruturas que podem ser encontradas na linguagem T++ são as de repetição, neste caso temos o "repita-até", sendo possível também encontrar um exemplo de sua utilização a seguir. É importante citar que em ambas estruturas, repetição ou condicionais é possível utilizar operadores relacionais ou lógicos (Listing 2), como <, >, =, <=, >= ,< >, && e ||.

Listing 2. Utilização de operadores relacionais e lógicos em T++.

```
repita
    vet[j+1] := vet[j]
    j := j-1
at  j>=0 && vet[j] > chave
```

Por fim é possível citar que nesta linguagem existem tipos "inteiro" e "flutuante", utilizados ao se declarar uma variável, e já que entramos no assunto de declarações de variáveis, é necessário apontar que a linguagem T++ possibilita declarar vetor e funções, assim como leitura de dados do terminal (Listing 7).

Listing 3. Utilização do função "escreva"em T++.

```
# Conta as quebras de linha dentro dos comentarios
def t_COMENTARIO(token):
    r"(\{((.|\n)*?)\})"
    token.lexer.lineno += token.value.count("\n")

# Express o regular para identificacao da quebra
# de uma linha
def t_newline(token):
    r"\n+"
    token.lexer.lineno += len(token.value)
```

3. Palavra Reservada

Alguns desses tokens na linguagem T++ são identificados como palavras reservadas, o que implica que não é possível utilizá-las como identificadores, já que tem uma função específica na gramática da linguagem. As palavras reservadas contidas na linguagem são representadas na tabela abaixo.

Palavras Reservadas	Tokens
se	SE
então	ENTAO
senão	SENAO
fim	FIM
repita	REPITA
flutuante	FLUTUANTE
retorna	RETORNA
até	ATE
leia	LEIA
escreva	ESCREVA
inteiro	INTEIRO

Tabela 1. Palavras reservadas e seus tokens

4. Expressões Regulares

Outros tokens podem ser identificados através de expressões regulares, que serão definidas no código que realizará a análise léxica. Como podemos observar na tabela abaixo (Tabela2), uma expressão regular foi criada para identificação do token `"MAIS"`, o qual consiste em um caractere "+", definida pela expressão "r'\+'".

Expressões Regulares	Tokens
r'\+'	MAIS
r'-'	MENOS
r'*'	MULTIPLICACAO
r'/'	DIVISAO
r'\('	ABRE_PARENTESE
r'\)'	FECHA_PARENTESE
r'\['	ABRE_COLCHETE
r'\]'	FECHA_COLCHETE
r','	VIRGULA
r':='	ATRIBUICAO

Expressões Regulares	Tokens
r':'	DOIS_PONTOS
r'&&'	E_LOGICO
r'||'	OU_LOGICO
r'!'	NEGACAO
r'<>'	DIFERENCA
r'<='	MENOR_IGUAL
r'>='	MAIOR_IGUAL
r'<'	MENOR
r'>'	MAIOR
r'='	IGUAL

Tabela 2. Expressões regulares e seus Tokens

Expressão Regular	Token
r"("+ r"([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])"+ r"("+ r"([0-9])"+ r"+_|"+ r"([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])"+ r")*)"	ID
r'\d+[eE][-+]?\d+|(\.\d+|\d+\.\d*)([eE][-+]?\d+)?'	PONTO_FLUTUANTE
r"("+ sinal + r"([1-9])\."+ digito + r"+[eE]"+ sinal + digito + r"+)"	NOTACAO_CIENTIFICA
r"\d+"	NUM_INTEIRO

Tabela 3. Expressões regulares complexas e seus Tokens.

5. Detalhes da Implementação

Em seguida veremos alguns detalhes da implementação, como por exemplo, os autômatos equivalentes as expressões regulares para identificação dos tokens, qual biblioteca foi utilizada, e como ela contribuiu para o desenvolvimento do analisador léxico.

5.1. PLY

Para implementação do sistema de varredura e análise léxica foi utilizado a biblioteca PLY (Python Lex-Yacc), que possui dois diferentes módulos, o `lex.py` e o `yacc.py`. O primeiro módulo é o `lex.py`, o qual foi utilizado na implementação desta atividade, cedida pelo professor. Neste módulo através de expressões regulares, é possível identificar os lexemas recebidos como entrada, através de um código `TPP`, e obter os seus respectivos tokens, retornando então como saída, uma lista de tokens.

5.2. Especificação dos Autômatos

Para definição do processo de reconhecimento são utilizados autômatos finitos, que nada mais são do que a implementação das especificações das expressões regulares que foram abordadas nas tabelas contidas na seção anterior (Tabela 2 e Tabela3). Como exemplo será utilizado o autômato finito representado na figura a seguir (Figura 1), representando a expressão regular para identificação do "E" lógico (Listing 4).

Listing 4. Autômatos que definem a identificação das expressões lógicas.

```
1 # Expressao Regular para operador logico.
2 t_E_LOGICO = r'&&'
```

O autômato tem como objetivo identificar uma string, que neste caso é "&&", representando um `"e"` lógico na linguagem `TPP`. Para que o autômato desenvolvido possa realizar essa tarefa, primeiro precisamos definir um estado inicial, neste caso, representado por s1, que será o primeiro estado que qualquer string da dados irá começar. A flecha existente entre um estado e outro representa uma transição de estados, que ocorre ao identificar o caractere `"&"`, passando então do estado s1 para o s2. E o mesmo se aplica para o estado s2, ao identificar o caractere `"&"`, transitará do estado s2 para o estado final s3. Ao chegar ao estado final podemos concluir que a string de caracteres lida pode ser identificada como o token `E_LÓGICO`.

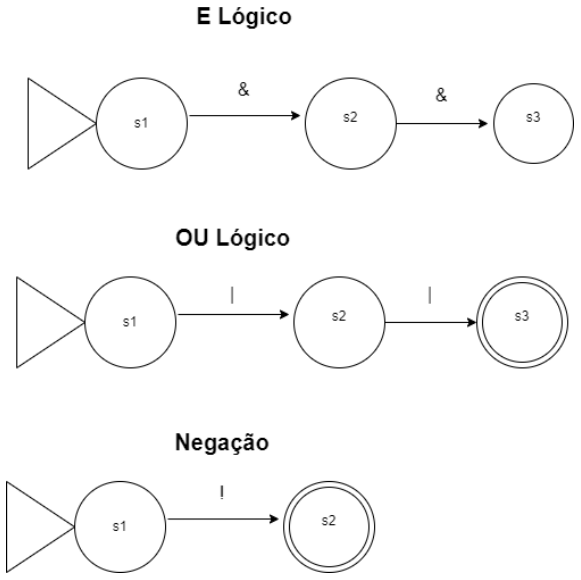

Figura 1. Autômatos que definem a identificação das expressões lógicas.

O mesmo se aplica para as palavras reservadas. Utilizando a palavra reservada "SE", o processo de identificação desse token é muito semelhante. Ele realizará as transições de estados ao identificar os caracteres "s", seguido pelo caractere "e", chegando ao estado final s3 (Figura 2). Os autômatos para identificação de um caractere (Figura 3), identificação de símbolos (Figura 4) e identificação de operadores relacionais (Figura 5) podem ser encontrados logo em seguida.

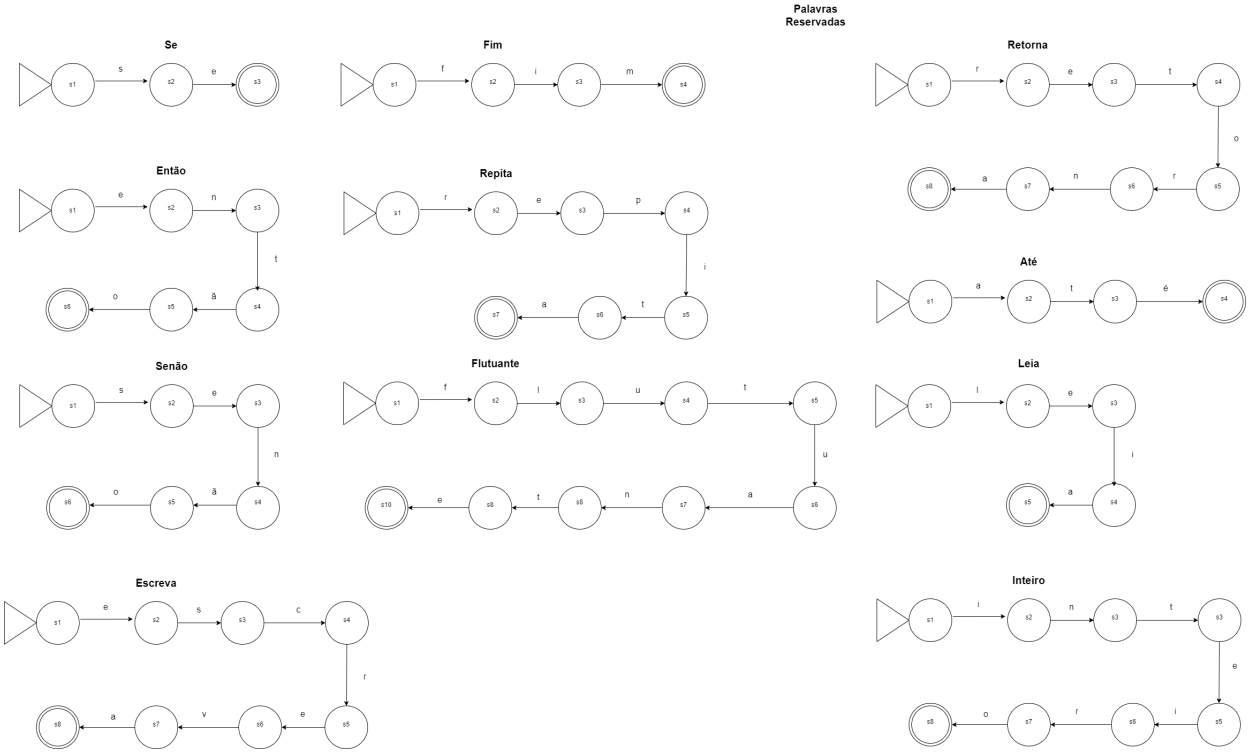

Figura 2. Autômatos que definem a identificação de palavras reservadas.

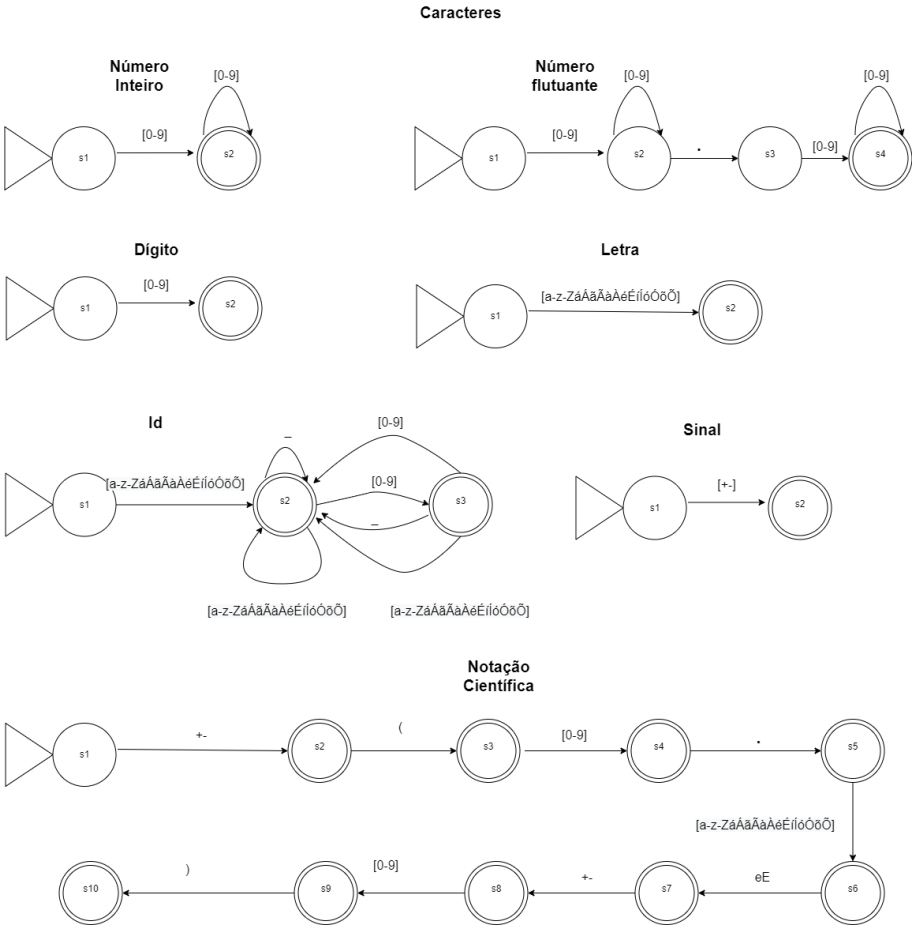

Figura 3. Autômatos que definem a identificação de um carctere.

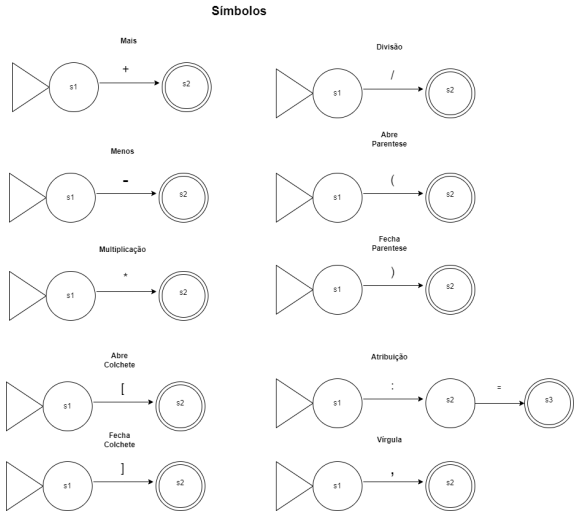

Figura 4. Autômatos que definem a identificação de símbolos.

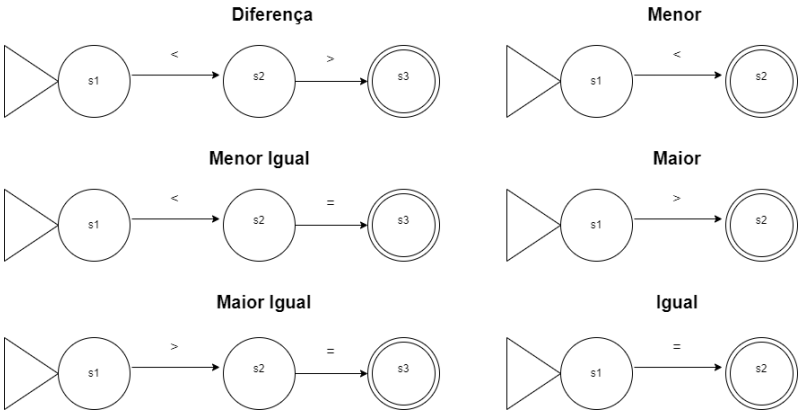

Figura 5. Autômatos que definem a identificação operadores relacionais.

5.3. Detalhes da Implementação da varredura

Como citado anteriormente é necessário definir os tokens anteriormente a varredura, para que o `PLY` possa ao fim de sua execução retornar todos tokens encontrados, os quais são guardados dentro de uma lista, como é possível notar na imagem a seguir (Listing 5).

Listing 5. Lista de tokens.

```
tokens = [
    # identificador
    "ID",
    # numerais
    "NUM_NOTACAO_CIENTIFICA",    # notacao cient fica
    "NUM_PONTO_FLUTUANTE",       # ponto flutuate
    "NUM_INTEIRO",               # inteiro
    # operadores binarios
    "MAIS",                      # +
```

```
    "MENOS",                    # -
    "MULTIPLICACAO",            # *
    "DIVISAO",                  # /
    "E_LOGICO",                 # &&
    "OU_LOGICO",                # ||
    "DIFERENCA",                # <>
    "MENOR_IGUAL",              # <=
    "MAIOR_IGUAL",              # >=
    "MENOR",                    # <
    "MAIOR",                    # >
    "IGUAL",                    # =
    # operadores unarios
    "NEGACAO",                  # !
    # simbolos
    "ABRE_PARENTESE",           # (
    "FECHA_PARENTESE",          # )
    "ABRE_COLCHETE",            # [
    "FECHA_COLCHETE",           # ]
    "VIRGULA",                  # ,
    "DOIS_PONTOS",              # :
    "ATRIBUICAO",               # :=
]
```

No caso acima são definidos a lista de tokens para identificador, numerais, operadores binários, operadores unários e símbolos (Listing 5), porém abaixo, é possível observar a lista de tokens relacionados às palavras reservadas da linguagem `TPP` (Listing 6).

Listing 6. Lista de tokens para palavras reservadas.

```
# Tokens das palavras reservadas
reserved_words = {
    "se": "SE",
    "entao": "ENTAO",
    "senao": "SENAO",
    "fim": "FIM",
    "repita": "REPITA",
    "flutuante": "FLUTUANTE",
    "retorna": "RETORNA",
    "ate": "ATE",
    "leia": "LEIA",
    "escreva": "ESCREVA",
    "inteiro": "INTEIRO",
}
```

Para que possamos utilizar o `lexer`, é preciso termos um input, o qual é definido utilizando a função `input()`, antes de chamar a função `token()`, que percorrerá o arquivo e identificará os tokens dados as expressões regulares definidas. Abaixo podemos observar como isso é feito (Listing 7).

Listing 7. Método main do analisador léxico.

```
def main():
    aux = argv[1].split('.')
    global status_mode;
    status_mode = False
```

```
5
6  # Verifico se passou o parametro "debug"
7      if len(argv) > 2:
8          if argv[2] == "debug":
9              status_mode = True
10
11 # Verifica se o codigo passado como parametro e tpp
12     if aux[-1] != 'tpp':
13         error_message = "ERRO: Estamos esperando por um arquivo .tpp,
14         porem recebemos como entrada um arquivo .%s" % aux[-1]
15
16         print("\n")
17         print(error_message)
18         print("\n")
19         raise IOError("Not a .tpp file!")
20
21     # Abre o arquivo passado como parametro
22     data = open(argv[1])
23
24     # Arquivo tpp    definido como input do lexer
25     source_file = data.read()
26     lexer.input(source_file)
27
28     # Passa por todo o arquivo
29     while True:
30       tok = lexer.token()
31       if not tok:
32         break
33
34     # Retorno os tokens encontrados
35       print(tok.type)
```

Acima é possível notar que dado um arquivo `TPP`, ele será lido e passado como input do lexer, através do método `input()`, possibilitando então que o método `token()` seja chamado constantemente até o arquivo todo ser lido. Além do código de início dado, implementei algumas pequenas mudanças, como a criação de uma mensagem de aviso caso o arquivo passado como parâmetro não seja um `.tpp`, e também inclui um modo `"debug"`, que ao passá-lo como parâmetro na execução da análise léxica, retornará uma mensagem com mais detalhes, que será exibida ao um caractere inválido ser identificado, como é possível notar a seguir (Listing 8).

Listing 8. Alteração no caso padrão para tratamento de erro.

```
1  def t_error(token):
2
3      line = token.lineno
4
5      if (status_mode):
6          debug_message = "Caracter invalido '%s' encontrado na linha
7          '%s'" % (token.value[0], token.lineno)
8
9          print(debug_message)
10     else:
11         message = "Caracter invalido '%s'" % token.value[0]
12         print(message)
```

```
13
14      token.lexer.skip(1)
```

Para que os tokens do código de entrada sejam identificados, utilizaremos as expressões regulares para definir como será o lexema que será identificado, como foi visto na figuras na seção anterior (Tabela 2 e Tabela 3). Além disso, criaremos uma regra para identificação do token, que em seguida realizará uma ação, no caso abaixo, retornando o token encontrado. Em posição subsequente podemos notar que há duas opções, utilizando o decorator `@token` (Listing 9.

Listing 9. Regras para identificação do token utilizando decorator @TOKEN.

```
1  @TOKEN(notacao_cientifica)
2  def t_NUM_NOTACAO_CIENTIFICA(token):
3      return token
4
5  @TOKEN(flutuante)
6  def t_NUM_PONTO_FLUTUANTE(token):
7      return token
8
9  @TOKEN(inteiro)
10 def t_NUM_INTEIRO(token):
11     return token
```

Outra opção é criar uma regra dentro do método, contendo a expressão regular do token que será identificado, como é possível observar a seguir. (Figura 9)

Listing 10. Regras para identificação do token somente definindo a expressão regular.

```
1  # Verifica quebras de linha dentro dos comentarios
2  def t_COMENTARIO(token):
3      r"(\{((.|\n)*?)\})"
4      token.lexer.lineno += token.value.count("\n")
5
6  # Expressao regular para identificar a quebra de uma linha
7  def t_newline(token):
8      r"\n+"
9      token.lexer.lineno += len(token.value)
```

6. Resultados

Para testar o analisador léxico, será utilizado o arquivo `"fibonacci-2020-2.tpp"` como entrada. É possível observar o arquivo de teste na figura abaixo (Listing 11).

Listing 11. Código "fibonacci-2020-2.tpp"utilizado como entrada.

```
1  inteiro principal()
2
3    inteiro: t1
4    inteiro: t2
5    inteiro: t3
6
7    t1 := 0
8    t2 := 1
9    t3 := 1
```

```
10
11    escreva(t1)
12
13    repita
14      escreva(t3)
15      t3 := t1 + t2
16      t1 := t2
17      t2 := t3
18    ate t3 >= 100
19  fim
```

Para executar o analisador léxico, é preciso utilizar o comando abaixo (Figura 6), passando como parâmetro o arquivo de teste.

```
python tpplex.py ../../lexica-testes/fibonacci-2020-2.tpp
```

Figura 6. Comando para execução do teste.

Como saída obtemos uma lista de tokens identificados apartir do código de entrada. É possível visualizar esta lista de tokens a seguir (Figura 7).

```
juanrangel@DESKTOP-2NI6V5P:~/trabComp/Compiladores/analiseLexica/implementacao$ python tpplex.py ../../lexica-testes/fibonacci-2020-2.tpp
INTEIRO
ID
ABRE_PARENTESE
FECHA_PARENTESE
INTEIRO
DOIS_PONTOS
ID
INTEIRO
DOIS_PONTOS
ID
INTEIRO
DOIS_PONTOS
ID
ID
ATRIBUICAO
NUM_INTEIRO
ID
ATRIBUICAO
NUM_INTEIRO
ID
ATRIBUICAO
NUM_INTEIRO
ESCREVA
ABRE_PARENTESE
ID
FECHA_PARENTESE
REPITA
ESCREVA
ABRE_PARENTESE
ID
FECHA_PARENTESE
ID
ATRIBUICAO
ID
MAIS
ID
ID
ATRIBUICAO
ID
ID
ATRIBUICAO
ID
ATE
ID
MAIOR_IGUAL
NUM_INTEIRO
FIM
```

Figura 7. Lista de tokens retornados na saída da execução.

Por fim realizaremos mais um teste utilizando o arquivo `"fat.tpp"` (Figura 12) fornecido pelo professor.

Listing 12. Código de entrada "fat.tpp".

```
1  inteiro: n
2  flutuante: a[10]
3
4  inteiro fatorial(inteiro: n)
5      inteiro: fat
6      se n > 0 ent o {n o calcula se n > 0}
7          fat := 1
```

```
8           repita
9               fat := fat * n
10              n := n - 1
11          at   n = 0
12          retorna(fat) {retorna o valor do fatorial de n}
13      sen o
14          retorna(0)
15      fim
16  fim
17
18  inteiro principal()
19      leia(n)
20      escreva(fatorial(n))
21      retorna(0)
22  fim
```

Neste caso utilizaremos basicamente o mesmo comando executado anteriormente (Figura 6), porém será passado um parâmetro diferente. Como desejo testar o arquivo "fat.tpp", passarei ele como parâmetro, como é possível observar a seguir (Figura 8).

```
juan_rangel@LAPTOP-5H6E554J:~/Documentos/Compiladores/3.AnaliseLexica/testes-lexica$ python3 ../lex.py fat.tpp
```

Figura 8. Comando para execução da análise léxica para o arquivo "fat.tpp".

Como saída, obtivemos a lista de token a seguir (Figura 9).

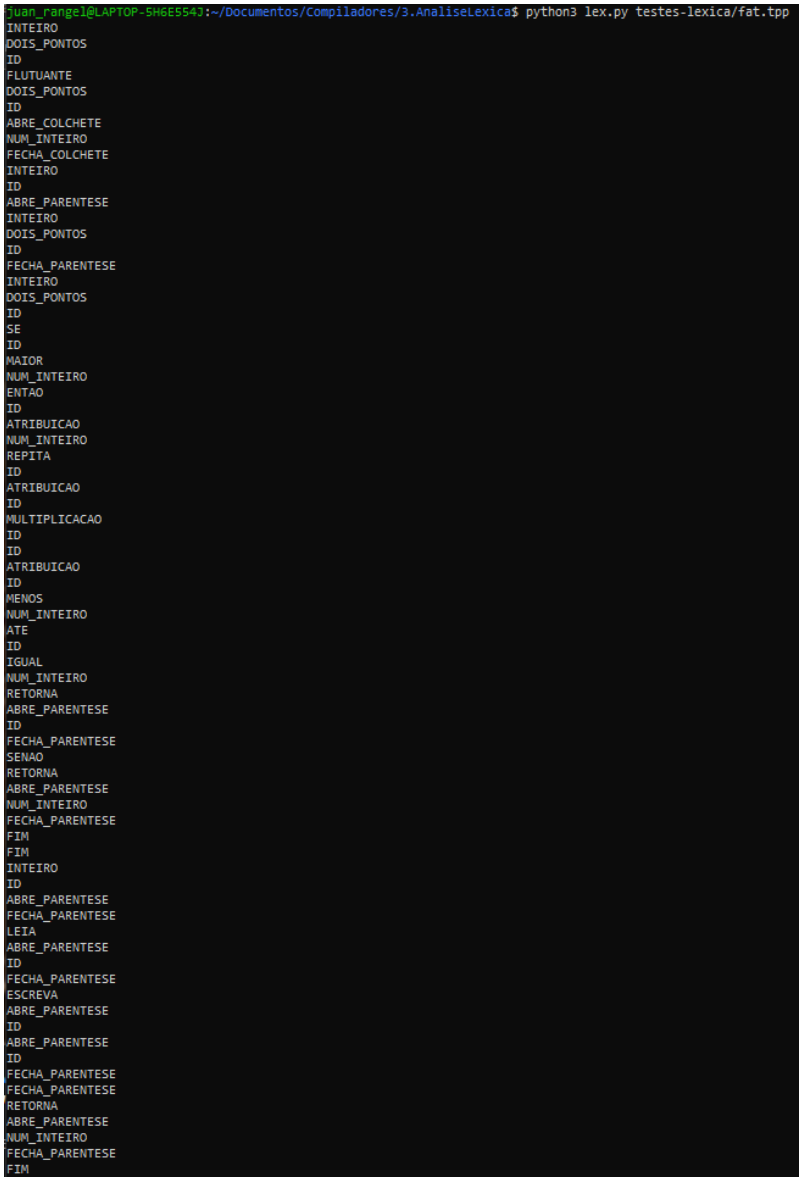

Figura 9. Código de entrada "fat.tpp".

Referências

Beazley, D. M. Ply (python lex-yacc). ttps://www.dabeaz.com/ply/ply.tml.

Gonçalves, R. A. Aula 002 – visão geral da análise léxica. ttps://moodle.utfpr.edu.br/pluginfile.pp/222642/mod$_resource/content/10/aula - 02 - introducao - analise - lexica - especificacao - linguagem.md.slides.pdf$.