Análise Léxica

Noemi Pereira Scherer[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

{noemischerer13}@gmail.com

Abstract. *This meta-article describes the procedures and results of the development of a lexical analyzer, which is responsible for verifying the input of character lines from a T ++ written source code, producing a synonym of lexical symbols. For the creation of this analyze, a Python language library called PLY was used.*

Resumo. *Este meta-artigo descreve os procedimentos e resultados do desenvolvimento de um analisador léxico, que é responsável por verificar as entradas de linhas de caracteres de um código fonte escrito T++, produzindo uma senquência de símbolos léxicos. Para a criação desse analisar, foi utilizado uma biblioteca da linguagem Python denominado PLY.*

1. Introdução

A análise léxica é um processo que analisa a entrada de linhas de caracteres (código fonte de um programa) e produz uma sequência de símbolos léxicos denominados *tokens*, sendo facilmente manipulado por um *parser* (leitor de saída) [Wikibooks 2018].

O analisador léxico lê cada caractere do programa fonte e, traduz em uma saída os *tokens*. Dessa forma, é possível reconhecer as palavras reservadas, constantes, identificadores e outras palavras que pertencem a linguagem de programação analisada. O analisador também pode executar outras tarefas como o tratamento de espaços, eliminação de comentários e contagem do número de linhas que o programa possui [Wikibooks 2018].

O analisador léxico funciona de duas maneiras o primeiro e segundo estado da análise. O primeiro é responsável por ler a entrada de caracteres mudando o estado em que esses se encontram. Quando o analisador encontra um caractere o qual ele não considera como correto, ele volta à última análise que foi aceita. No segundo são repassados os caracteres encontrados para produzir um valor. O tipo do léxico é combinado com seu valor constituindo um símbolo, que pode ser denominado parser.

A implementação de um analisador léxico requer uma descrição do autômato que reconhece as sentenças da gramática ou expressão regular de cada *token* que possui os seguintes procedimentos [Wikibooks 2018]:

- Estado inicial, que recebe como argumento a referência para o autômato e retorna o seu estado inicial;
- Estado final, que recebe como argumentos a referência para o autômato e a referência para o estado corrente. O procedimento retorna verdadeiro se o estado especificado é elemento do conjunto de estados finais do autômato, ou falso caso contrário; e

- Próximo estado, que recebe como argumento a referência para o autômato, para o estado corrente e para o símbolo sendo analisado. O procedimento consulta a tabela de transições e retorna o próximo estado do autômato, ou o valor nulo se não houver transição possível.

1.1. Objetivo

O objetivo desse trabalho é desenvolver em um programa capaz de realizar a análise léxica de códigos fontes escrito na Linguagem T++, retornando o conjunto de *tokens* encontrado.

2. A Linguagem T++

A linguagem T++ foi desenvolvida especialmente para ser utilizada na disciplina de compiladores. Ela é uma linguagem simples, contendo algumas palavras reservadas, símbolos e arranjos uni e bidimensionais. A Tabela 1 mostra todas as palavras reservadas e símbolos que a linguagem T++ permite.

Tabela 1. *Tokens* permitidos pela Linguagem T++

Palavras Reservadas	**Símbolos**
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, vírgula
retorna	:= atribuição
até	< menor
leia	> maior
escreve	<= menor-igual
inteiro	>= maior-igual
notação científica	() abre e fecha parênteses
	: dois pontos
	[] abre e fecha colchetes
	{} comentário
	|| ou-lógico
	&& e-lógico
	! negação

A construção do código dessa linguagem é inteiramente em Português. Os números podem ser inteiros, flutuantes (notação científica ou não), e as declarações de variáveis devem obrigatoriamente começar com letras precedente de várias letras e/ou números.

Apesar de T++ ser uma linguagem simples, ela permite a execução de algoritmos complexos, como de ordenação. A Figura 1 demonstra o algoritmo de busca BubbleSort na linguagem t++.

Figura 1. Exeplo do código BubbleSort na linguagem T++

```
inteiro: vet[10]
inteiro: tam
tam := 10

preencheVetor()
  inteiro: i
  inteiro: j
  i := 0
  j := tam
  repita
    vet[i] = j
    i := i + 1
    j := j - 1
  até i < tam
fim

bubble_sort()
  inteiro: i
  i := 0
  repita
    inteiro: j
    j := 0
    repita
      se vet[i] > v[j] então
        inteiro: temp
        temp := vet[i]
        vet[i] := vet[j]
        vet[j] := temp
      fim
      j := j + 1
    até j < i
    i := i + 1
  até i < tam
fim

inteiro principal()
  preencheVetor()
  bubble_sort()
  retorna(0)
fim
```

3. Expressões Regulares e Autômatos

Uma expressão regular é responsável por identificar cadeia de caracteres de uma determinada linguagem, como palavras ou padrões. Elas são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular [AHO 2008].

Essa expressão pode ser associada com a aritmética, entretanto ao invés dela denotar um número (como 2+2), a expressão regular denota uma linguagem regular. Por exemplo: a0+, que resultará em {"a0", "a00", "a000", ...} [AHO 2008].

As expressões regulares são utilizadas para a especificação léxica de uma determinada linguagem. Para sua demonstração são utilizados autômatos finitos determinísticos.

Um autômato finito é um modelo computacional composto de uma fita de entrada dividida em células, nas quais contém os símbolos das cadeia. A principal parte do modelo é um controle finito, o qual indica em qual estado o autômato se encontra [USP 2012].

O modelo do autômato apresenta um estado inicial e um conjunto de estados finais. O estado inicial refere-se ao início de funcionamento do modelo, dependendo do símbolo presente na fita o próximo estado será escolhido, enquanto que os estados finais indicam o término do processo com sucesso, ou seja, se a cadeia presente na fita de entrada tiver sido completamente lida e o modelo não se encontrar em um estado final, considera-se a cadeia não aceita, ou caso contrário, como aceita [USP 2012].

Para cada *token* permitido na Linguagem T++ é criado uma expressão regular, o qual tem como objetivo analisar uma cadeia de caracteres no código. As expressões regulares utilizadas nesse trabalho estão descritas na Tabela 2.

Tabela 2. Expressões regulares de cada token

Tokens	**Expressão regulares**
Soma	\+
Subtração	-
Multiplicação	*
Divisão	\/
Igual	\=
Vírgula	,
Atribuição	\:=+
Menor	\¡
Maior	\¿
Menor Igual	\¡=
Maior igual	\¿=
Abre parênteses	\(
Fecha parênteses	\)
Abre colchetes	\[
Fecha Colchetes	\]
Dois pontos	:
E lógico	&&
OU lógico	| |
Negação	!
ID	[a-zA-Zà-úÀ-Ú][_0-9a-zà-úA-ZÀ-Ú]*
Notação Científica	[0-9]+(\.[0-9]+)*(e| E)+(\+ |\-)*[0-9]+(\.[0-9])*
Flutuante	[0-9]+(\.[0-9]+)(e(\+| \-)?(d+))?
Inteiro	[0-9]+
Comentário	{ [^\{ ^\}] }
Nova Linha	\n+

A ordem de execução de cada expressão regular é representada em forma de autômatos. Os autômatos mais simples são dos operadores (soma, subtração, divisão, multiplicação, abre e fecha parênteses e colchetes, atribuição, igualdade, vírgula, dois pontos, ou e e-lógico e negação) que são representados na Figura 2.

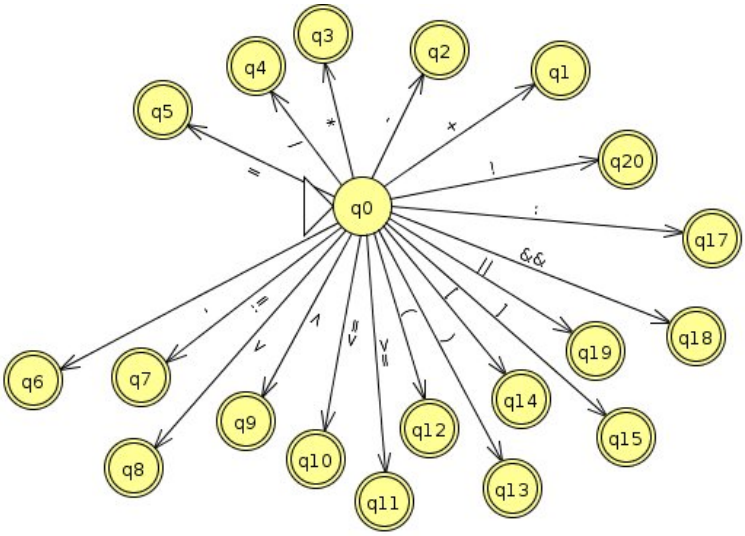

Figura 2. Autômatos dos operadores.

Os autômatos da expressão regular do ID (identificador) estão representados de duas formas, uma resumida (Figura 3), e outro detalhada (Figura 4). O seu objetivo é identificar qualquer palavra ou letra presente no código, podendo ser letras maiúsculas, minúsculas, com ou sem assento.

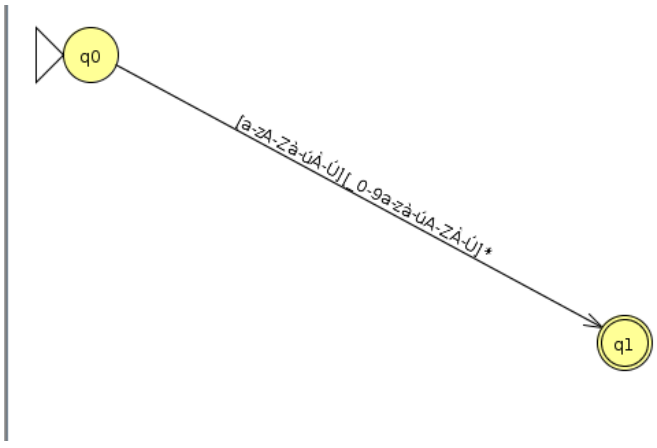

Figura 3. Autômato resumido da expressão regular do identificador (ID).

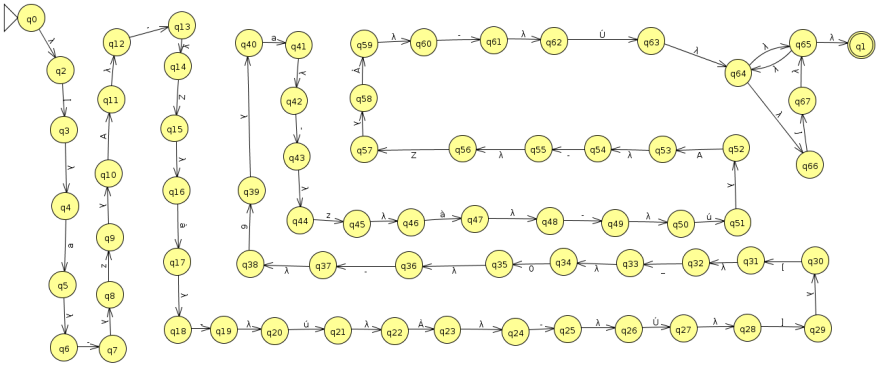

Figura 4. Autômato completo da expressão regular do identificador (ID).

O autômato da expressão regular dos números inteiros recebe qualquer número de zero a nove uma ou várias vezes até chegar em seu estado final. Ele está representado na Figura 5.

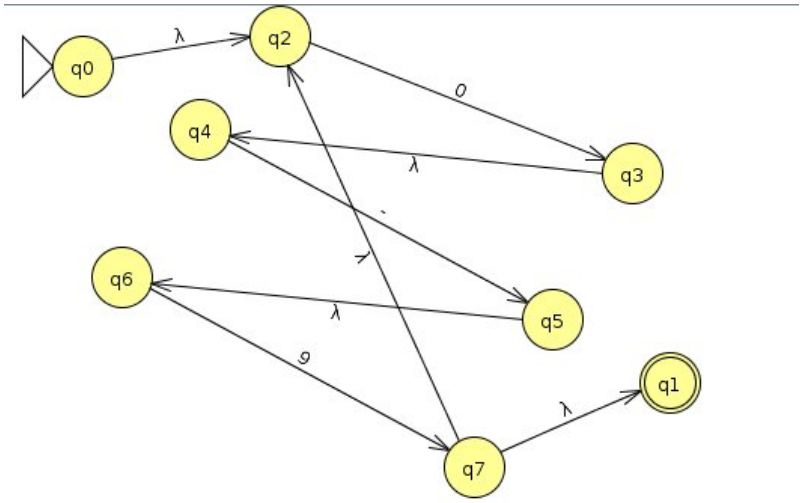

Figura 5. Autômato dos números inteiros.

O autômato da expressão regular dos números flutuantes está representado na Figura 6. Seu objetivo é identificar qualquer número que esteja separado por ponto no código, como 22.5 ou 0.22.

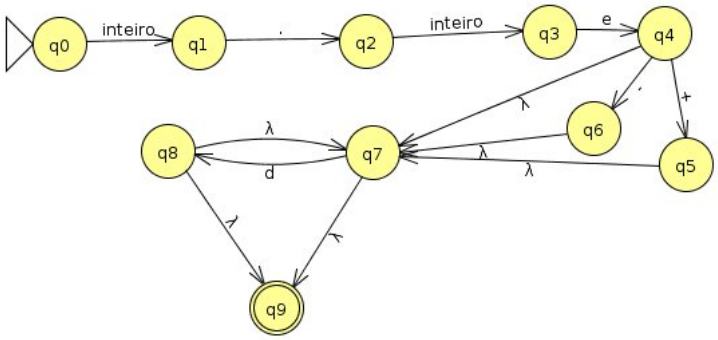

Figura 6. Autômato dos números flutuantes.

O autômato da expressão regular dos números flutuantes em notação científica está representado na Figura 7. Seu objetivo é identificar valor que possuem expoente no código, como 10E20 ou 10e-20.

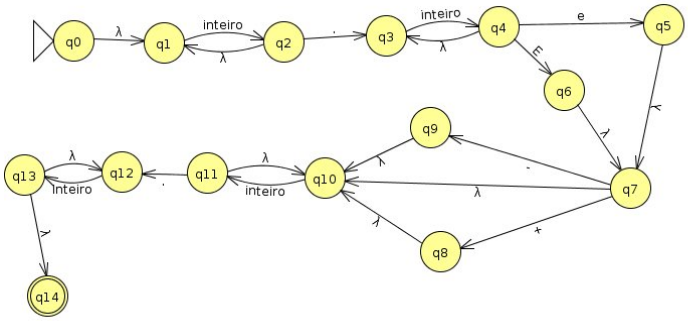

Figura 7. Autômato dos números flutuantes em notação científica.

Os autômatos da expressão regular dos comentários estão representados de duas formas, uma resumida (Figura 8) e outra detalhada (Figura 9). Os comentários no na linguagem T++ são representadas por qualquer palavra que esteja entre '{}', como exemplo: {Esse é um comentário }.

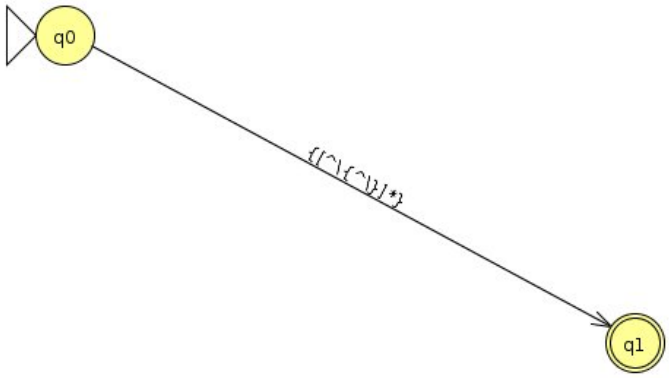

Figura 8. Autômato resumido do comentário.

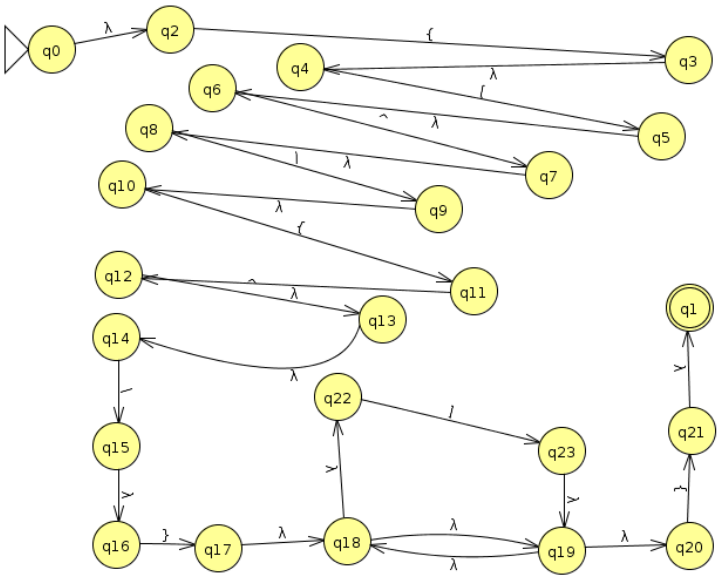

Figura 9. Autômato completo do comentário.

Por fim, o autômato da expressão regular da nova linha está representado representado na Figura 10. Seu objetivo é encontrar qualquer linha no código denotado por \n.

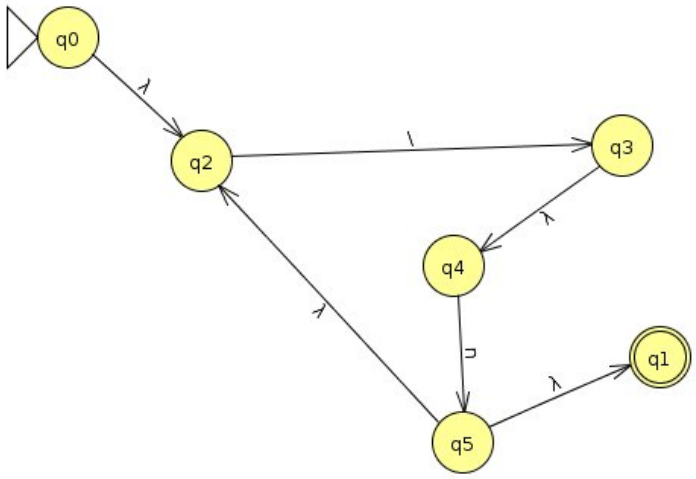

Figura 10. Autômato da nova linha.

4. PLY - Python Lex Yacc e o Código

O PLY é uma ferramenta da linguagem Python para construção de compiladores populares lex e yacc. O principal objetivo do PLY é permanecer fiel ao modo como as ferramentas lex/yacc tradicionais funcionam. Isso inclui fornecer validação extensiva de entradas, relatórios de erros e diagnósticos [David M. Beazley 2011].

Como o PLY foi desenvolvido principalmente como uma ferramenta de instrução, é possível notar a exigência em sua especificação de regras de *token* e gramática. Em parte, isso acontece para detectar erros comuns de programação feitos por usuários iniciantes.

O PLY consiste em dois módulos separados; lex.py e yacc.py, ambos encontrados em um pacote do Python chamado ply. O módulo lex.py é usado para dividir o texto de entrada em uma coleção de *tokens* especificados por uma coleção de regras de expressão regular. O yacc.py é usado para reconhecer a sintaxe da linguagem que foi especificada na forma de uma gramática livre de contexto [David M. Beazley 2011].

As duas ferramentas são destinadas a trabalhar juntas. O lex.py fornece uma interface externa na forma de uma função `token()` que retorna o próximo *token* válido no fluxo de entrada. Já a yacc.py chama isso repetidamente para recuperar *tokens* e invocar regras gramaticais. A saída de yacc.py é geralmente uma *Abstract Syntax Tree* (AST) [David M. Beazley 2011].

Para o desenvolvimento da análise sintática, foi utilizado apenas o módulo lex.py. Para isso, o primeiro passo foi instalar o ply no ambiente linux utilizando o seguinte comando:

```
pipe3 install ply
```

É importante ressaltar que a versão do python deve ser acima de 3, para aceitar caracteres com acento, pois na linguagem T++ existem palavras reservadas que obrigatoriamente possuem acentos, como então e senão.

Para utilizar a ferramenta no código fonte deve-se importar o ply por meio do

seguinte comando:

```
import ply.lex as lex
```

No código fonte da análise léxica é necessário descrever quais são as palavras reservadas e os *tokens* da linguagem, como demostra na Figura 11.

```
 4 # Palavras reservadas
 5 reservadas = {
 6     'se':'SE',
 7     'então':'ENTAO',
 8     'senão':'SENAO',
 9     'fim':'FIM',
10     'repita':'REPITA',
11     'vazio':'VAZIO',
12     'até':'ATE',
13     'leia':'LEIA',
14     'escreva':'ESCREVA',
15     'retorna':'RETORNA',
16     'principal':'PRINCIPAL',
17     'inteiro' : 'INTEIRO',
18     'flutuante' : 'FLUTUANTE'
19     }
20
21 # Lista de tokens
22 tokens = ['SOMA', 'SUB', 'MULT', 'DIVISAO', 'IGUAL', 'VIRGULA',
   'ATRIBUICAO', 'MENOR', 'MAIOR', 'MENOR_IGUAL','MAIOR_IGUAL',
   'ABRE_PAR', 'FECHA_PAR', 'DOIS_PONTOS', 'ABRE_COL', 'FECHA_COL',
   'E_LOGICO', 'OU_LOGICO', 'NEGACAO', 'ID','NOVA_LINHA', 'COMENTARIO',
   'NOTACAO_CIENTIFICA'] + \
23     list(reservadas.values())
```

Figura 11. Palavras reservadas e *tokens* no código fonte da análise léxica.

Para o reconhecimento das palavras reservadas e *tokens*, são definidas as expressões regulares de cada uma no código fonte por meio de regras e funções (Figura 12). As expressões regulares já foram definidas na Tabela 2.

```
25 # Regras de expressões regulares
26 t_SOMA = r'\+'
27 t_SUB = r'-'
28 t_MULT= r'\*'
29 t_DIVISAO = r'\/'
30 t_IGUAL = r'\='
31 t_VIRGULA = r'\,'
32 t_ATRIBUICAO = r'\:=+'
33 t_MENOR = r'\<'
34 t_MAIOR = r'\>'
35 t_MENOR_IGUAL = r'<='
36 t_MAIOR_IGUAL = r'>='
37 t_ABRE_PAR = r'\('
38 t_FECHA_PAR = r'\)'
39 t_ABRE_COL = r'\['
40 t_FECHA_COL = r'\]'
41 t_DOIS_PONTOS = r':'
42 t_E_LOGICO = r'&&'
43 t_OU_LOGICO = r'\|\|'
44 t_NEGACAO = r'!'
45
46
47 def t_ID(t):
48   r'[a-zA-Zà-úÀ-Ú][_0-9a-zà-úA-ZÀ-Ú]*'
49   t.type  = reservadas.get(t.value, 'ID')
50   return t
51
52 def t_NOTACAO_CIENTIFICA(t):
53   r'[0-9]+(\.[0-9]+)*(e|E)+(\+|\-)?[0-9]+(\.[0-9])*'
54   t.type = "NOTACAO_CIENTIFICA"
55   return t
```

Figura 12. Expressões regulares das palavras reservadas e *tokens* no código fonte da análise léxica.

5. Resultados

Com o código fonte da análise léxica pronta, é possível executá-lo de forma que receba um arquivo contendo o código na linguagem T++, leia cada *token* do mesmo, e retorne um arquivo de saída identificando cada *token* encontrado, bem como sua linha e coluna.

Para executar o código pelo terminal utiliza-se o seguinte comando:

```
python3 lexer.py nomeDoArquivo.tpp
```

Considere o seguinte código feito em T++ responsável por calcular o fatorial de um número:

```
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
```

Quando executado o código da análise léxica para o exemplo anterior, tem-se a seguinte saída:

```
LexToken(INTEIRO,'inteiro',1,0)
LexToken(DOIS_PONTOS,':',1,7)
LexToken(ID,'n',1,9)
LexToken(NOVA_LINHA,'\n\n',1,10)
LexToken(INTEIRO,'inteiro',3,12)
LexToken(ID,'fatorial',3,20)
LexToken(ABRE_PAR,'(',3,28)
LexToken(INTEIRO,'inteiro',3,29)
LexToken(DOIS_PONTOS,':',3,36)
LexToken(ID,'n',3,38)
LexToken(FECHA_PAR,')',3,39)
LexToken(NOVA_LINHA,'\n',3,40)
LexToken(INTEIRO,'inteiro',4,45)
LexToken(DOIS_PONTOS,':',4,52)
LexToken(ID,'fat',4,54)
LexToken(NOVA_LINHA,'\n',4,57)
LexToken(SE,'se',5,62)
LexToken(ID,'n',5,65)
LexToken(MAIOR,'>',5,67)
LexToken(INTEIRO,'0',5,69)
LexToken(ENTAO,'então',5,71)
LexToken(COMENTARIO,'{não calcula se n > 0}',5,77)
LexToken(NOVA_LINHA,'\n',5,99)
LexToken(ID,'fat',6,108)
LexToken(ATRIBUICAO,':=',6,112)
LexToken(INTEIRO,'1',6,115)
LexToken(NOVA_LINHA,'\n',6,116)
LexToken(REPITA,'repita',7,125)
LexToken(NOVA_LINHA,'\n',7,131)
LexToken(ID,'fat',8,144)
LexToken(ATRIBUICAO,':=',8,148)
```

```
LexToken(ID,'fat',8,151)
LexToken(MULT,'*',8,155)
LexToken(ID,'n',8,157)
LexToken(NOVA_LINHA,'\n',8,158)
LexToken(ID,'n',9,171)
LexToken(ATRIBUICAO,':=',9,173)
LexToken(ID,'n',9,176)
LexToken(SUB,'-',9,178)
LexToken(INTEIRO,'1',9,180)
LexToken(NOVA_LINHA,'\n',9,181)
LexToken(ATE,'até',10,190)
LexToken(ID,'n',10,194)
LexToken(IGUAL,'=',10,196)
LexToken(INTEIRO,'0',10,198)
LexToken(NOVA_LINHA,'\n',10,199)
LexToken(RETORNA,'retorna',11,208)
LexToken(ABRE_PAR,'(',11,215)
LexToken(ID,'fat',11,216)
LexToken(FECHA_PAR,')',11,219)
LexToken(COMENTARIO,'{retorna ...}',11,221)
LexToken(NOVA_LINHA,'\n',11,255)
LexToken(SENAO,'senão',12,260)
LexToken(NOVA_LINHA,'\n',12,265)
LexToken(RETORNA,'retorna',13,274)
LexToken(ABRE_PAR,'(',13,281)
LexToken(INTEIRO,'0',13,282)
LexToken(FECHA_PAR,')',13,283)
LexToken(NOVA_LINHA,'\n',13,284)
LexToken(FIM,'fim',14,289)
```

É possível notar que o código da análise léxica avaliou cada palavra do código em T++, o classificou e retornou seu valor em um arquivo de saída. Isso é exatamente o objetivo da análise léxica, retornar todos os *tokens* que um determinado código possui.

Referências

AHO, A. V. (2008). *Compuladores: Princípios, técnicas e ferramentas*. SP: Pearson Addison-Wesley, 2nd edition.

David M. Beazley (2011). Ply (python lex-yacc). `http://www.eng.utah.edu/~cs3100/lectures/l14/ply-3.4/doc/ply.html`. Acessado em 26-03-2018.

USP (2012). Autômatos finito determinísticos. `https://edisciplinas.usp.br/pluginfile.php/4159708/mod_resource/content/1/De_representacao_de_linguagens_ate_Automato_Finito_-_2012.pdf`. Acessado em 27-03-2018.

Wikibooks (2018). Construção de compiladores: Análise léxica. https://pt.wikibooks.org/wiki/Constru%C3%A7%C3%A3o_de_compiladores/An%C3%A1lise_l%C3%A9xica. Acessado em 25-03-2018.