Projeto e Implementação de uma Ferramenta de Compilação para a Linguagem TPP

Yagho Junior Petini - 2380366[1]
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This the article template for the journal "Constructions". Its goal is to make it as easy as possible for authors to typeset their papers in LaTeX. For users with limited LaTeX experience, we recommend using Overleaf (https://www.overleaf.com). To start working on your paper in Overleaf, just make a copy of this template and replace the text of this tutorial by the text of your paper.

Resumo

This the article template for the journal "Constructions". Its goal is to make it as easy as possible for authors to typeset their papers in LaTeX. For users with limited LaTeX experience, we recommend using Overleaf (https://www.overleaf.com). To start working on your paper in Overleaf, just make a copy of this template and replace the text of this tutorial by the text of your paper.

1 Instruções

2 Introdução

Os compiladores desempenham um papel fundamental no desenvolvimento de software, são responsáveis por traduzir o código fonte que foi escrito em alguma linguagem de programação de alto nível para um formato em que o computador possa entender e gerar um executável desse código. Segundo LOUDEN (2004), o processo de compilação é dividido em partes, podendo citar como sendo essenciais as fases de Análise Léxica, Análise Sintática, Análise Semântica e Geração de Código. Cada uma desses estágios servem para certificar que o código seja corretamente traduzido e executado pelo computador.

O estudo dos compiladores tornou-se extremamente importante para que as linguagens de programação possam entregar um desempenho superior a medida que a sua evolução acontece, aumentando assim a praticidade e a versatilidade dessas linguagens. Nesse artigo mostrará como foi desenvolvido um Compilador para a linguagem TPP. O objetivo desse Compilador é proporcionar uma experiência acadêmica, sendo assim, uma aplicação sem um nível alto de complexidade buscando fins educacionais.

Portanto, o presente artigo mostrará como foi implementado o Compilador responsável por interpretar os arquivos com a extensão da linguagem **TPP**, essa extensão será a *.tpp* e somente os arquivos de código fonte com essa extensão poderão ser executados.

Corresponding author(s): Random Author, University of Somewhere, random.author@uni-somewhere.edu

Palavras Reservadas	Símbolos
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, virgula
retorna	:= atribuição
até	< menor
leia	> maior
escreva	<= menor-igual
inteiro	>= maior-igual
	(abre-par
	) fecha-par
	: dois-pontos
	[abre-col
	] fecha-col
	&& e-logico
	|| ou-logico
	! negação

Tabela 1: Tokens da linguagem **TPP**.

3 Análise Léxica

A Análise Léxica é a etapa onde um código-fonte de uma linguagem será lido como um arquivo de caracteres e separado em um conjunto de marcas (**tokens**). Podendo, cada **token** ser uma palavra reservada ou uma palavra chave especifica da linguagem em questão. Dessa forma, podemos citar alguns dos **tokens** da linguagem TPP, sendo eles o ”se”, ”então”, ”senão”. Essas palavras chaves e reservadas que posteriormente serão utilizadas como referência para identificar variáveis, funções e etc.

O reconhecimento dessas marcas e a identificação de padrões podem ser realizados de duas maneiras: utilizando *expressões regulares* ou implementando o analisador com *autômatos finitos*. Para desenvolver o sistema de varredura (Análisador Léxico) da linguagem TPP, é essencial considerar as classes de tokens apresentadas na Tabela 1.

```
MAIS, MENOS, VEZES, DIVIDE,
DOIS_PONTOS, VIRGULA, MENOR, MAIOR,
IGUAL, DIFERENTE, MENOR_IGUAL, MAIOR_IGUAL,
E, OU, NAO, ABRE_PARENTESE, FECHA_PARENTESE,
ABRE_COLCHETE, FECHA_COLCHETE, SE,
ENTAO, SENAO, FIM, REPITA, ATE, ATRIBUICAO,
LEIA, ESCREVA, RETORNA, INTEIRO, FLUTUANTE,
NUM_INTEIRO, NUM_PONTO_FLUTUANTE,
NUM_NOTACAO_CIENTIFICA, ID
```

Código 1: Tokens da Linguagem **TPP**.

Os **tokens** da linguagem **TPP** estão definidos abaixo no Código 1, Esses **tokens** serão gerados a partir da análise léxica feita pelo Compilador. Dessa forma, nesta etapa é possível identificar se os primeiro erros que o usuário pode ter cometido, como por exemplo passar um arquivo de

tests > bubble_sort-2020-2.tpp

```
flutuante: A[20]

bubbleSort(inteiro: n)
    inteiro: i
    inteiro: j

    i := 0
    j := 0

    repita
        repita
            se A[j] > A[j+1] então
                inteiro: aux

                aux := A[j]
                A[j] := A[j+1]
                A[j+1] := aux
            fim

            j := j + 1
        até j = (n-i-1)

        i := i + 1
    até i = n
fim


inteiro principal()
    inteiro: i
    i := 20
    repita
        A[i - 1] := i
        i := i - 1
    até i = 0

    bubbleSort(20)
fim
```

Figura 2: *Bubble-Sort* na linguagem TPP.

código fonte para o compilador interpretar em que a sua extensão não seja *.tpp*, dessa forma a Análise Léxica irá interpretar essa extensão diferente como um erro e retornará para o usuário como uma mensagem de aviso.

3.1 Linguagem TPP

A ideia da Linguagem TPP é ser uma linguagem simples e de fácil entendimento, como foi mostrado anteriormente, suas palavras reservadas estão em português-br, isto para criar um aspecto intuitivo no desenvolvedor TPP. Contudo, assim como algumas linguagens de alto nível, a linguagem TPP pode ser utilizada para criar algoritmos complexos, como por exemplo na Figura 2, onde está sendo ilustrado como seria o algoritmo de ordenação *Bubble Sort* em TPP.

Uma dúvida que pode vir a surgir é como que o Compilador reconhece se uma sequência de caracteres do arquivo fonte passado pertence ou não ao conjunto de palavras reservadas da linguagem em questão. Para solucionar essa dúvida, utilizaremos as *expressões regulares* e *autômatos finitos*, estes que delimitam se uma sequência de caracteres é uma palavra chave ou reservada da linguagem especificada.

3.2 Expressões Regulares e Autômatos Finitos

Autômatos Finitos podem ser utilizados para descrever o processo de reconhecimento de padrões em cadeias de entrada, e assim podendo ser utilizados para construir um Analisador Léxico. Um autômato finito determinístico formado por entrada e estados, dessa forma, para cada entrada que ele receber, apenas uma saída será válida. Contudo, um autômato finito determinístico deve ser composto de alguns itens, sendo eles um alfabeto de entrada, um conjunto de estados possíveis, uma função de transição que informará para qual estado o autômato deve ir a partir de uma determinada entrada, um estado inicial e um estado final. Dessa maneira, após receber uma sequência de caracteres, o autômato processar individualmente cada um e verifica se para aquele estado, sendo um carácter, possui uma saída válida. Se ao final da sequência o autômato estiver em um estado final, a sequência será válida, caso contrário ela será considerada inválida.

No caso da linguagem TPP, abaixo na **Figura 3** e na **Figura 4** está ilustrado o funcionamento do reconhecimento dessas palavras por meio de um *autômato finito*. No primeiro autômato está mostrando o funcionamento da validação da palavra reservada ”INTEIRO”, podemos observar no autômato finito que se o usuário digitar um carácter errado da sequência válida, o Analisador Léxico não reconhecerá esta palavra como uma palavra reservada. Já no segundo autômato está mostrando o funcionamento da palavra reservada ”retorna”, e de forma semelhante ao anterior, caso a sequência de caracteres não seja idêntica a sequência da palavra chave, o Analisador Léxico não reconhecerá essa palavra.

Contudo, para expressões regulares o comportamento é semelhante ao autômato finito. Entretanto as expressões regulares combinam padrões de busca juntamente com caracteres especiais que servem para determinar um carácter pode aparecer em uma sequência, para determinar se uma sequência informada pertence ou não a linguagem em questão. Um exemplo de expressão regular para a linguagem TPP está ilustrado na Figura 5, expressão está que mostra o formato de um digito que a linguagem TPP aceita.

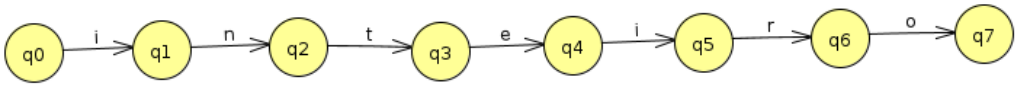

Figura 3: *Autômato Finito* da Palavra Reservada *inteiro*.

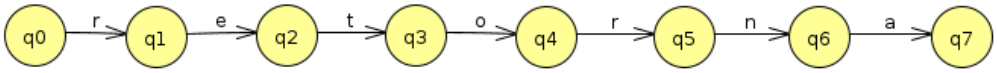

Figura 4: *Autômato Finito* da Palavra Reservada *retorna*.

```
70    digito = r"([0-9])"
71
```

Figura 5: *Expressão Regular* de um digito.

3.3 Implementação

Para a implementação do Analisador Léxico foi utilizado a linguagem de programação **Python** em conjunto com a sua biblioteca **PLY**. A escolha de Python para o desenvolvimento desse projeto foi devido a sua facilidade de uso e sua simplicidade, além de fornecer bibliotecas que facilitam o desenvolvimento de aplicações como essa de forma mais ágil. A biblioteca *PLY* que foi escolhida, particularmente é excelente para a criação de Analisadores Léxicos, dessa forma, facilitando o reconhecimento dos **tokens** na linguagem TPP. Essa implementação foi divida em partes, dessa forma individualizando cada função do Analisador Léxico.

A primeira parte da implementação foi definir os **tokens**, a partir do *PLY*, que seriam utilizados na linguagem TPP, como está sendo ilustrado na Figura 6, foram definidos todos os **tokens** que anteriormente foram citados.

```
22  tokens = [
23      "ID",  # identificador
24      # numerais
25      "NUM_NOTACAO_CIENTIFICA",  # ponto flutuante em notaçao científica
26      "NUM_PONTO_FLUTUANTE",  # ponto flutuate
27      "NUM_INTEIRO",  # inteiro
28      # operadores binarios
29      "MAIS",  # +
30      "MENOS",  # -
31      "VEZES",  # *
32      "DIVIDE",  # /
33      "E",  # &&
34      "OU",  # ||
35      "DIFERENTE",  # <>
36      "MENOR_IGUAL",  # <=
37      "MAIOR_IGUAL",  # >=
38      "MENOR",  # <
39      "MAIOR",  # >
40      "IGUAL",  # =
41      # operadores unarios
42      "NAO",  # !
43      # simbolos
44      "ABRE_PARENTESE",  # (
45      "FECHA_PARENTESE",  # )
46      "ABRE_COLCHETE",  # [
47      "FECHA_COLCHETE",  # ]
48      "VIRGULA",  # ,
49      "DOIS_PONTOS",  # :
50      "ATRIBUICAO",  # :=
51      # 'COMENTARIO', # {***}
52  ]
53
54  reserved_words = {
55      "se": "SE",
56      "então": "ENTAO",
57      "senão": "SENAO",
58      "fim": "FIM",
59      "repita": "REPITA",
60      "flutuante": "FLUTUANTE",
61      "retorna": "RETORNA",
62      "até": "ATE",
63      "leia": "LEIA",
64      "escreva": "ESCREVA",
65      "inteiro": "INTEIRO",
66  }
```

Figura 6: *TOKENS* da linguagem TPP.

A segunda parte foi definir as expressões regulares para esses *tokens*, veja na Figura 7. Para esse passo foi utilizado o *REGEX*, na Figura 7 está mostrando a definição dos *tokens* simples, já para criar as expressões regulares dos *tokens* das palavras reservadas foi utilizado a técnica ilustrada na Figura 8.

```
digito = r"([0-9])"
letra = r"([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])"
sinal = r"([\-\+]?)"

id = (
    r"(" + letra + r"(" + digito + r"+|_|" + letra + r")*)"
)
inteiro = r"\d+"

flutuante = (
    r'\d+[eE][-+]?\d+|(\.\d+|\d+\.\d*)([eE][-+]?\d+)?'
)

notacao_cientifica = (
    r"(" + sinal + r"([1-9])\." + digito + r"+[eE]" + sinal + digito + r"+)"
)  # o mesmo que '(([-\+]?)([1-9])\.([0-9])+[eE]([-\+]?)([0-9]+))'

# Símbolos.
t_MAIS = r'\+'
t_MENOS = r'-'
t_VEZES = r'\*'
t_DIVIDE = r'/'
t_ABRE_PARENTESE = r'\('
t_FECHA_PARENTESE = r'\)'
t_ABRE_COLCHETE = r'\['
t_FECHA_COLCHETE = r'\]'
t_VIRGULA = r','
t_ATRIBUICAO = r':='
t_DOIS_PONTOS = r':'

# Operadores Lógicos.
t_E = r'&&'
t_OU = r'\|\|'
t_NAO = r'!'

# Operadores Relacionais.
t_DIFERENTE = r'<>'
t_MENOR_IGUAL = r'<='
t_MAIOR_IGUAL = r'>='
t_MENOR = r'<'
t_MAIOR = r'>'
t_IGUAL = r'='
```

Figura 7: *TOKENS* Simples da Expressão Regular das Palavras Reservadas.

```
@TOKEN(id)
def t_ID(token):
    token.type = reserved_words.get(token.value, "ID")
    # não é necessário fazer regras/regex para cada palavra reservada
    # se o token não for uma palavra reservada automaticamente é um id
    # As palavras reservadas têm precedências sobre os ids
    return token

@TOKEN(notacao_cientifica)
def t_NUM_NOTACAO_CIENTIFICA(token):
    return token


@TOKEN(flutuante)
def t_NUM_PONTO_FLUTUANTE(token):
    return token


@TOKEN(inteiro)
def t_NUM_INTEIRO(token):
    return token


t_ignore = " \t"
```

Figura 8: *TOKENS* Complexos da Expressão Regular das Palavras Reservadas.

A terceira parte foi definir um *token* para caso de erros encontrados pelo Analisador Léxico. Para isso foi utilizado a técnica demonstrada na Figura 9. Este que tem como objetivo incluir na saída deste programa uma sequência de carácter que indique que algum carácter passado pelo arquivo do código fonte não é reconhecido pela linguagem TPP.

```
158
159 def t_error(token):
160
161     # file = token.lexer.filename
162     line = token.lineno
163     column = define_column(token.lexer.lexdata, token.lexpos)
164     message = le.newError(check_key, 'ERR-LEX-INV-CHAR', token.lineno, column, valor=token.value[0])
165     # print(f"[{file}]:[{line},{column}]: {message}.")
166     print(message)
167
168     token.lexer.skip(1)
169
170     # token.lexer.has_error = True
```

Figura 9: *TOKEN* de caso de erro.

A quarta parte é onde se encontra nossa função *main*, ela que é responsável por capturar os argumentos passados pela linha de comando e definir qual será a ação do Analisador Léxico. Dessa forma, algumas validações de erros estão sendo feitas na *main*, como por exemplo se o código fonte passado não for *.tpp*, ou até mesmo se o arquivo do código fonte realmente existe. Contudo, se o arquivo passar nessas validações ele será submetido ao Analisador Léxico, este que após cada *token* encontrado gerará uma saída. Seguindo as definições já mencionadas de que será aceito somente sequências de caracteres que foram definidas na linguagem. A seguir está a Figura 10 onde ilustra a implementação dessa função.

```
172
173 def main():
174
175     global check_tpp
176     global check_key
177
178     check_tpp = False
179     check_key = False
180
181     for idx, arg in enumerate(sys.argv):
182         # print("Argument #{} is {}".format(idx, arg))
183         aux = arg.split('.')
184         if aux[-1] == 'tpp':
185             check_tpp = True
186             idx_tpp = idx
187
188         if(arg == "-k"):
189             check_key = True
190
191     #print ("No. of arguments passed is ", len(sys.argv))
192     if(len(sys.argv) <= 2):
193         raise TypeError(le.newError(check_key, 'ERR-LEX-USE'))
194
195     if not check_tpp:
196       raise IOError(le.newError(check_key, 'ERR-LEX-NOT-TPP'))
197     elif not os.path.exists(argv[idx_tpp]):
198         raise IOError(le.newError(check_key, 'ERR-LEX-FILE-NOT-EXISTS'))
199     else:
200         data = open(argv[idx_tpp])
201
202         source_file = data.read()
203         lexer.input(source_file)
204
205         # Tokenize
206         while True:
207             tok = lexer.token()
208             if not tok:
209                 break      # No more input
210             #print(tok)
211             print(tok.type)
212             #print(tok.value)
```

Figura 10: Função *Main*.

3.4 Resultado

Para conseguir efetuar os testes e concretizar que o Analisador Léxico desenvolvido neste projeto está funcionando corretamente foi utilizado uma extensão da linguagem de programação *PYTHON* chamada *PYTEST*, a partir dessa extensão foi possível utilizar de uma maneira simplificada um arquivo composto por diversos algoritmo afim de testar a corretude atual do Compilador. Os algoritmos que foram testados foram semelhantes ao que foi mostrado anteriormente na Figura 2.

O resultado era obter todos os *tokens* dos algoritmos que serão passados para teste, dessa forma, pode ser validado que o Analisador Léxico desenvolvido até o momento está funcionando como deveria. A seguir está as Figuras 11, 12, 13 e 14 que ilustram como código fonte de testes foi feito. Primeiramente, na Figura 11 está mostrando como foi definido a função que irá de fato executar os testes, já na Figura 12, 13 e 14 está ilustrando os 37 casos de testes e quais entradas e algoritmos foram utilizados para os testes.

tpplex_test.py > test_001

```python
import tpplex
import subprocess
import shlex
import os, fnmatch

def execute_test(input_file, args):
    if(input_file != ''):
        path_file = 'tests/' + input_file
    else:
        path_file = ""

    # Por algum motivo quando passava input_file = "" o pytest passava um nome de arquivo e dava o erro:
    # b'ERR-LEX-NOT-TPP\n'
    # Expected output:
    # ERR-LEX-USE
    # process = subprocess.Popen(['python', 'tpplex.py', args, path_file], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    cmd = "python tpplex.py {0} {1}".format(args, path_file)
    process = subprocess.Popen(shlex.split(cmd), stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    stdout, stderr = process.communicate()
    stdout, stderr

    path_file = 'tests/' + input_file
    output_file = open(path_file + ".out", "r")

    #read whole file to a string
    expected_output = output_file.read()

    output_file.close()

    print("Generated output:")
    print(stdout)
    print("Expected output:")
    print(expected_output)

    return stdout.decode("utf-8") == expected_output
```

Figura 11: Função de execução dos testes.

```
41
42  def test_001():
43      assert execute_test("", "-k") == True
44
45  def test_002():
46      assert execute_test("teste.c", "-k") == True
47
48  def test_003():
49      assert execute_test("notexist.tpp", "-k") == True
50
51  def test_004():
52      assert execute_test("teste-001.tpp", "-k") == True
53
54  def test_005():
55      assert execute_test("teste-002.tpp", "-k") == True
56
57  def test_006():
58      assert execute_test("teste-003.tpp", "-k") == True
59
60  def test_007():
61      assert execute_test("teste-004.tpp", "-k") == True
62
63  def test_008():
64      assert execute_test("teste-005.tpp", "-k") == True
65
66  def test_009():
67      assert execute_test("bubble_sort_2.tpp", "-k") == True
68
69  def test_010():
70      assert execute_test("bubble_sort.tpp", "-k") == True
71
72  def test_011():
73      assert execute_test("Busca_Linear_1061992.tpp", "-k") == True
74
75  def test_012():
76      assert execute_test("buscaLinear-2020-2.tpp", "-k") == True
77
78  def test_013():
79      assert execute_test("comp.tpp", "-k") == True
80
81  def test_014():
82      assert execute_test("fatorial-2020-2.tpp", "-k") == True
83
84  def test_015():
85      assert execute_test("fatorial.tpp", "-k") == True
```

Figura 12: Entradas utilizadas para o teste do Analisador Léxico.

```
86
87  def test_016():
88      assert execute_test("fat.tpp", "-k") == True
89
90  def test_017():
91      assert execute_test("fibonacci-2020-2.tpp", "-k") == True
92
93  def test_018():
94      assert execute_test("fibonacci.tpp", "-k") == True
95
96  def test_019():
97      assert execute_test("hanoi-2020-2.tpp", "-k") == True
98
99  def test_020():
100     assert execute_test("insertionSort-2020-2.tpp", "-k") == True
101
102 def test_021():
103     assert execute_test("insertSort-2020-2.tpp", "-k") == True
104
105 def test_022():
106     assert execute_test("maiorDoVetor.tpp", "-k") == True
107
108 def test_023():
109     assert execute_test("multiplicavetor.tpp", "-k") == True
110
111 def test_024():
112     assert execute_test("operacao_vetor-2020-2.tpp", "-k") == True
113
114 def test_025():
115     assert execute_test("paraBinario-2020-2.tpp", "-k") == True
116
117 def test_026():
118     assert execute_test("primo.tpp", "-k") == True
119
120 def test_027():
121     assert execute_test("produtoEscalar.tpp", "-k") == True
122
123 def test_028():
124     assert execute_test("prog_test.tpp", "-k") == True
125
126 def test_029():
127     assert execute_test("sample.tpp", "-k") == True
128
129 def test_030():
130     assert execute_test("selectionSort-2020-2.tpp", "-k") == True
131
```

Figura 13: Entradas utilizadas para o teste do Analisador Léxico.

```python

def test_024():
    assert execute_test("operacao_vetor-2020-2.tpp", "-k") == True

def test_025():
    assert execute_test("paraBinario-2020-2.tpp", "-k") == True

def test_026():
    assert execute_test("primo.tpp", "-k") == True

def test_027():
    assert execute_test("produtoEscalar.tpp", "-k") == True

def test_028():
    assert execute_test("prog_test.tpp", "-k") == True

def test_029():
    assert execute_test("sample.tpp", "-k") == True

def test_030():
    assert execute_test("selectionSort-2020-2.tpp", "-k") == True

def test_031():
    assert execute_test("selectionsort.tpp", "-k") == True

def test_032():
    assert execute_test("soma_maior_que_3.tpp", "-k") == True

def test_033():
    assert execute_test("somavet.tpp", "-k") == True

def test_034():
    assert execute_test("subtraiVetores.tpp", "-k") == True

def test_035():
    assert execute_test("verifica_valor_10.tpp", "-k") == True

def test_036():
    assert execute_test("verif_num_negativo.tpp", "-k") == True

def test_037():
    assert execute_test("bubble_sort-2020-2.tpp", "-k") == True

```

Figura 14: Entradas utilizadas para o teste do Analisador Léxico.

A partir dessa função de teste, o resultado obtido após os 37 casos de teste mostraram está ilustrado na Figura 15.

```
============================================= test session starts ==============================================
platform linux -- Python 3.8.10, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sidinei/yagho/Compiladores/compilador_python/compiler_python
collected 37 items

tpplex_test.py::test_001 PASSED                                                                      [  2%]
tpplex_test.py::test_002 PASSED                                                                      [  5%]
tpplex_test.py::test_003 PASSED                                                                      [  8%]
tpplex_test.py::test_004 PASSED                                                                      [ 10%]
tpplex_test.py::test_005 PASSED                                                                      [ 13%]
tpplex_test.py::test_006 PASSED                                                                      [ 16%]
tpplex_test.py::test_007 PASSED                                                                      [ 18%]
tpplex_test.py::test_008 PASSED                                                                      [ 21%]
tpplex_test.py::test_009 PASSED                                                                      [ 24%]
tpplex_test.py::test_010 PASSED                                                                      [ 27%]
tpplex_test.py::test_011 PASSED                                                                      [ 29%]
tpplex_test.py::test_012 PASSED                                                                      [ 32%]
tpplex_test.py::test_013 PASSED                                                                      [ 35%]
tpplex_test.py::test_014 PASSED                                                                      [ 37%]
tpplex_test.py::test_015 PASSED                                                                      [ 40%]
tpplex_test.py::test_016 PASSED                                                                      [ 43%]
tpplex_test.py::test_017 PASSED                                                                      [ 45%]
tpplex_test.py::test_018 PASSED                                                                      [ 48%]
tpplex_test.py::test_019 PASSED                                                                      [ 51%]
tpplex_test.py::test_020 PASSED                                                                      [ 54%]
tpplex_test.py::test_021 PASSED                                                                      [ 56%]
tpplex_test.py::test_022 PASSED                                                                      [ 59%]
tpplex_test.py::test_023 PASSED                                                                      [ 62%]
tpplex_test.py::test_024 PASSED                                                                      [ 64%]
tpplex_test.py::test_025 PASSED                                                                      [ 67%]
tpplex_test.py::test_026 PASSED                                                                      [ 70%]
tpplex_test.py::test_027 PASSED                                                                      [ 72%]
tpplex_test.py::test_028 PASSED                                                                      [ 75%]
tpplex_test.py::test_029 PASSED                                                                      [ 78%]
tpplex_test.py::test_030 PASSED                                                                      [ 81%]
tpplex_test.py::test_031 PASSED                                                                      [ 83%]
tpplex_test.py::test_032 PASSED                                                                      [ 86%]
tpplex_test.py::test_033 PASSED                                                                      [ 89%]
tpplex_test.py::test_034 PASSED                                                                      [ 91%]
tpplex_test.py::test_035 PASSED                                                                      [ 94%]
tpplex_test.py::test_036 PASSED                                                                      [ 97%]
tpplex_test.py::test_037 PASSED                                                                      [100%]

============================================== 37 passed in 0.90s ==============================================
```

Figura 15: Saídas do teste do Analisador Léxico.

Portanto, após a execução desses testes, podemos concluir que o Analisador Léxico está funci-

onando como deveria. A seguir está um exemplo dos *tokens* que foram obtidos na saida de um dos testes. A Figura 16 ilustra o algoritmo para o teste e a Figura 17 a saida dos *tokens*.

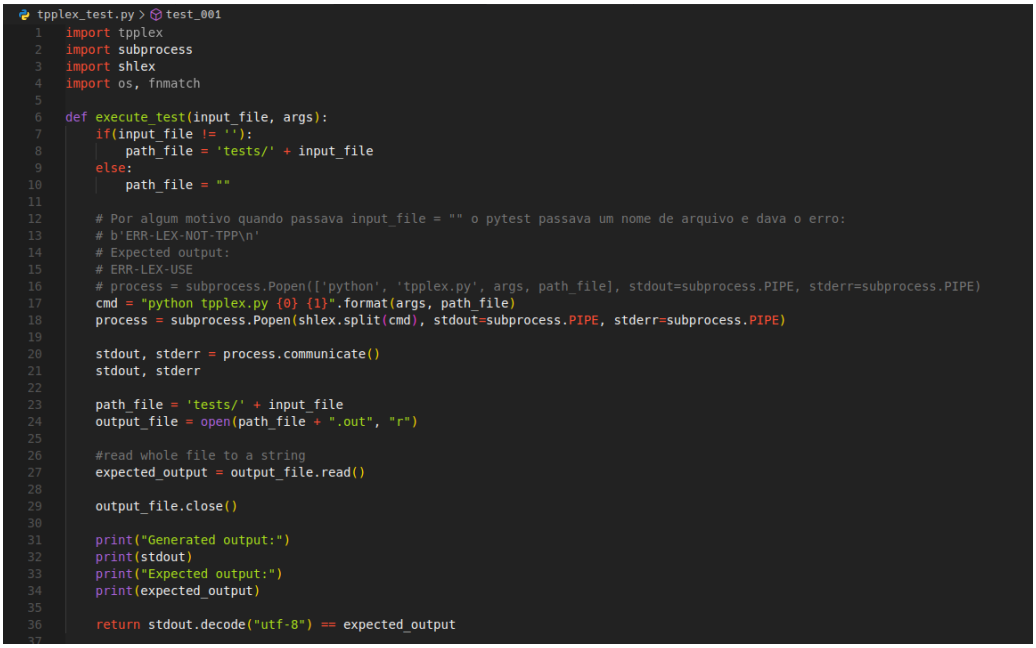

Figura 16: Algoritmo utilizado para o teste.

tests > verifica_valor_10.tpp.out

```
1  INTEIRO
2  ID
3  ABRE_PARENTESE
4  FECHA_PARENTESE
5  INTEIRO
6  DOIS_PONTOS
7  ID
8  ID
9  ATRIBUICAO
10 NUM_INTEIRO
11 REPITA
12 SE
13 ID
14 IGUAL
15 NUM_INTEIRO
16 ESCREVA
17 ABRE_PARENTESE
18 ERR-LEX-INV-CHAR
19 ID
20 NUM_INTEIRO
21 ERR-LEX-INV-CHAR
22 FECHA_PARENTESE
23 FIM
24 ID
25 MAIS
26 MAIS
27 ATE
28 ID
29 IGUAL
30 NUM_INTEIRO
31 RETORNA
32 ABRE_PARENTESE
33 NUM_INTEIRO
34 FECHA_PARENTESE
35 FIM
36
```

Figura 17: Saídas do teste do Analisador Léxico.

Portanto, pode-se concluir que a etapa da Análise Léxica está concluída e correta, pois passou em todos os testes feitos e gerou todos os *tokens* que deveria ser gerado.

4 Análise Sintática

5 Análise Semântica

6 Geração de Código

Referências

LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e práticas*. São Paulo, SP: Thomson 1st edn.