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.

```
se quantidade == 1 então
    count := coun+1
senão
    TorreHanoi(origem, auxiliar, destino, quantidade-1)
    TorreHanoi(origem, destino, auxiliar, 1)
    TorreHanoi(auxiliar, destino, origem, quantidade-1)
fim
```

Figura 1. Código em T++ utilizando "se", "então", "senão".

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, como <, >, =, $\leq, \geq, <>, \&\&$ e $||$.

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

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

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.

```
inteiro: principal()
  inteiro: tamanho := 5
  inteiro: vetor[tamanho]

  preeenche_vetor(vetor, tamanho)

  inteiro: resultado := processa_vetor(vetor, tamanho)

  escreva(resultado)

  retorna(0)
fim
```

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

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, 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. Este módulo é empregue com o objetivo, através de expressões regulares, 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 5), representando a expressão regular (Figura 4), como um autômato.

```
t_E_LOGICO = r'&&'
```

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

O autômato tem como objetivo identificar um string, que neste caso é "", representando um "e"lógico na linguagem tpp. Para que o autômato desenvolvida 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 carácter "", passando então do estado s1 para o s2. E o mesmo se aplica para o estado s2, ao identificar o carácter "", transitará do estado s2 para o estado final s3. Ao chegar ao estado final podemos concluir que a string de carácteres lida pode ser identificada como o token E_LÓGICO.

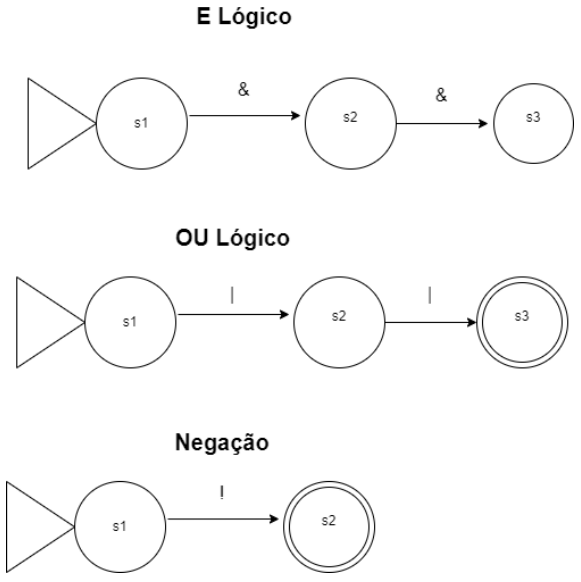

Figura 5. Autômatos que definem a identificação das expressões lógicas.

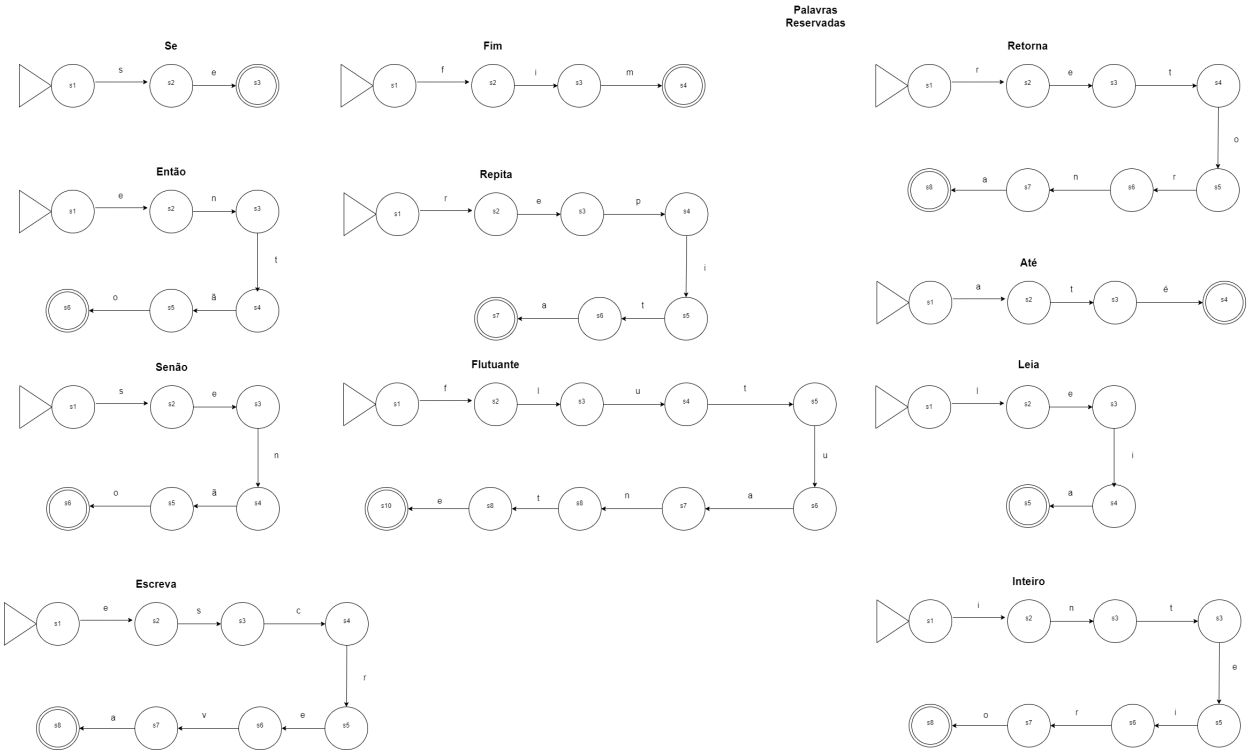

Figura 6. Autômatos que definem a identificação de palavras reservadas.

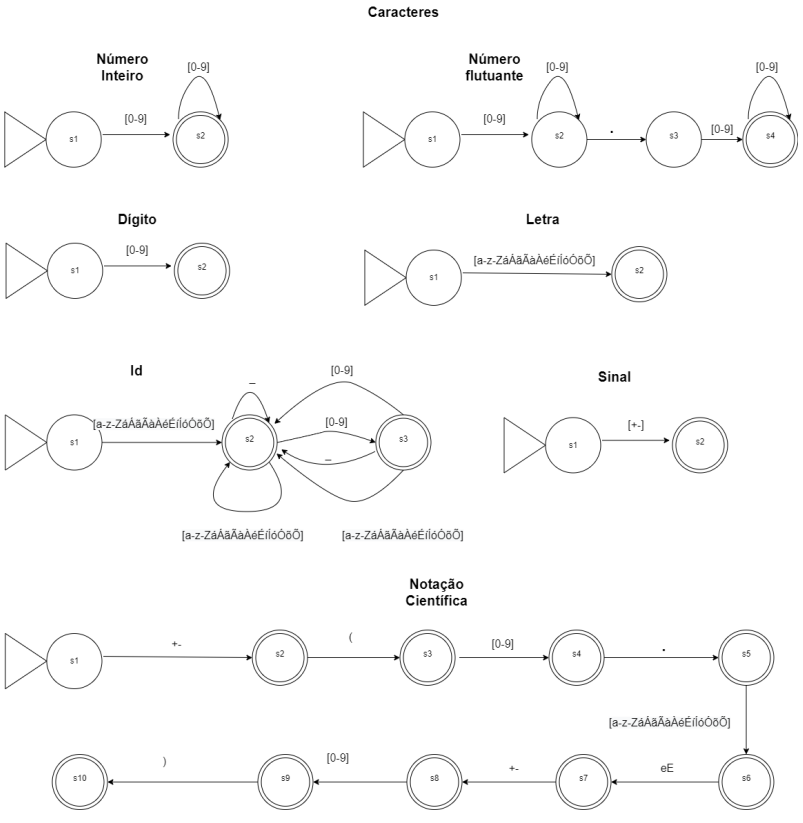

Figura 7. Autômatos que definem a identificação de um carácter.

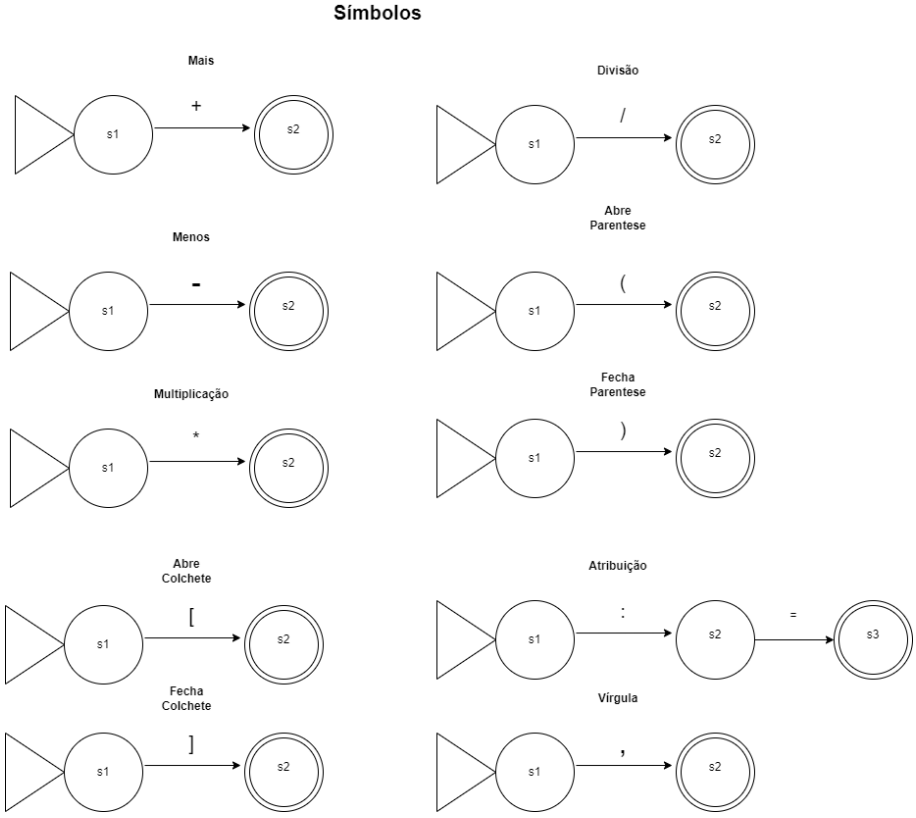

Figura 8. Autômatos que definem a identificação de símbolos.

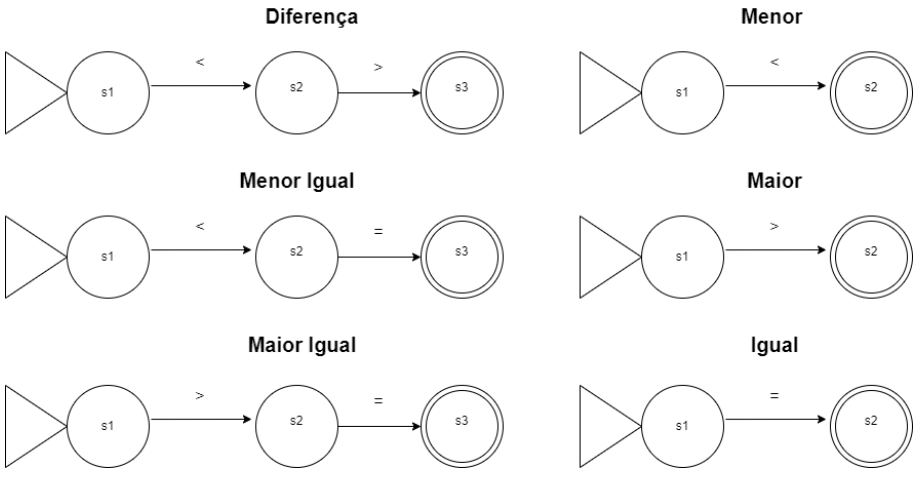

Figura 9. Autômatos que definem a identificação operadores relacionais.

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 uma lista destes tokens encontrados. Para definição dos tokens é utilizada a estrutura de uma lista, como é possível notar na imagem a seguir.

Figura 10. Lista de tokens.

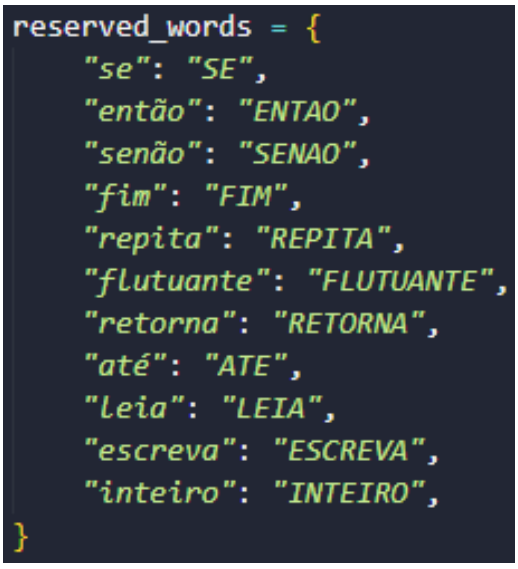

Figura 11. Lista de tokens.

Para que possamos utilizar o lexer, é preciso termos um input, antes de chamar a função token(). Abaixo podemos observar como isso é feito.

```
def main():
    aux = argv[1].split('.')
    if aux[-1] != 'tpp':
      raise IOError("Not a .tpp file!")
    data = open(argv[1])

    source_file = data.read()
    lexer.input(source_file)

    # Tokenize
    while True:
      tok = lexer.token()
      if not tok:
        break      # No more input

    # Retorno os tokens relacionados ao lexemas encontrados
      print(tok.type)
```

Figura 12. Realizando input e começando tokenização.

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.

Para que os lexemas do código de entrada seja identificado, utilizaremos as expressões regulares para definir como identificar os tokens da linguagem TPP, 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, ou utilizar o decorator @token definindo o token a partir da expressão regular (Figura 13), ou criando uma regra dentro do método, contendo a expressão regular (Figura 14)

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

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

Figura 13. Regras para identificação do token utilizando decorator @TOKEN.

```python
@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
```

Figura 14. Regras para identificação do token somente definindo a expressão regular.

6. Resutados

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 (Figura 15)

```
inteiro principal()

  inteiro: t1
  inteiro: t2
  inteiro: t3

  t1 := 0
  t2 := 1
  t3 := 1

  escreva(t1)

  repita
    escreva(t3)
    t3 := t1 + t2
    t1 := t2
    t2 := t3
  até t3 >= 100
fim
```

Figura 15. Código "fibonacci-2020-2.tpp"utilizado como entrada.

Para executar o analisador léxico, é preciso utilizar o comando abaixo (Figura 16), passando como parâmetro o arquivo de teste.

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

Figura 16. 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 17).

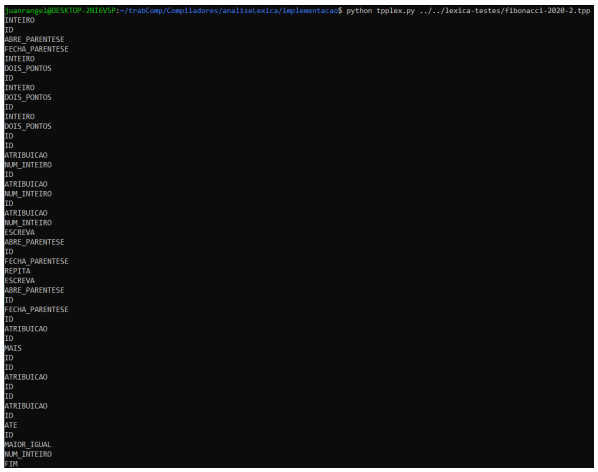

Figura 17. Lista de tokens retornados na saída da execução.

Por fim realizaremos mais um teste utilizando o arquivo "fat.tpp"(Figura 18) fornecido pelo professor.

```
1   inteiro: n
2   flutuante: a[10]
3
4   inteiro fatorial(inteiro: n)
5       inteiro: fat
6       se n > 0 então {não calcula se n > 0}
7           fat := 1
8           repita
9               fat := fat * n
10              n := n - 1
11          até n = 0
12          retorna(fat) {retorna o valor do fatorial de n}
13      senão
14          retorna(0)
15      fim
16  fim
17
18  inteiro principal()
19      leia(n)
20      escreva(fatorial(n))
21      retorna(0)
22  fim
23
24
```

Figura 18. Código de entrada "fat.tpp".

Neste caso utilizaremos basicamente o mesmo comando executado anteriormente (Figura 16), porém será passado um parâmetro diferente. Como desejo testar o arquivo "fat.tpp", passarei ele como parâmetro, como é possível observar a seguir (Figura 19).

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

Figura 19. Comando para execução da análise léxica para o arquivo "fat.tpp".

Como saída, obtivemos a lista de token a seguir (Figura 18).

```
juan_rangel@LAPTOP-5H6E554J:~/Documentos/Compiladores/3.AnaliseLexica$ python3 lex.py testes-lexica/fat.tpp
INTEIRO
DOIS_PONTOS
ID
FLUTUANTE
DOIS_PONTOS
ID
ABRE_COLCHETE
NUM_INTEIRO
FECHA_COLCHETE
INTEIRO
ID
ABRE_PARENTESE
INTEIRO
DOIS_PONTOS
ID
FECHA_PARENTESE
INTEIRO
DOIS_PONTOS
ID
SE
ID
MAIOR
NUM_INTEIRO
ENTAO
ID
ATRIBUICAO
NUM_INTEIRO
REPITA
ID
ATRIBUICAO
ID
MULTIPLICACAO
ID
ID
ATRIBUICAO
ID
MENOS
NUM_INTEIRO
ATE
ID
IGUAL
NUM_INTEIRO
RETORNA
ABRE_PARENTESE
ID
FECHA_PARENTESE
SENAO
RETORNA
ABRE_PARENTESE
NUM_INTEIRO
FECHA_PARENTESE
FIM
FIM
INTEIRO
ID
ABRE_PARENTESE
FECHA_PARENTESE
LEIA
ABRE_PARENTESE
ID
FECHA_PARENTESE
ESCREVA
ABRE_PARENTESE
ID
ABRE_PARENTESE
ID
FECHA_PARENTESE
FECHA_PARENTESE
RETORNA
ABRE_PARENTESE
NUM_INTEIRO
FECHA_PARENTESE
FIM
```

Figura 20. 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$_r esource/content/10/aula - 02 - introducao - analise - lexica - especificacao - linguagem.md.slides.pdf$.