

# Compiladores - Análise Léxica

Alisson da S. Vieira

<sup>1</sup>Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)  
Caixa Postal 15.064 – 91.501-970 – Campo Mourão – PR – Brasil

alissonv@alunos.utfpr.edu.br

**Abstract.** *For this work, a start code for Lexical Analysis was provided by the teacher. The Analysis is the first step of the compilation process, where the language tokens are extracted. The language chosen was T++, and the work describes a little about the characteristics of the language. It also discusses which language and library was used to perform the analysis, as well as showing output examples at the end.*

**Resumo.** *Para este trabalho, foi disponibilizado pelo professor, um código inicial para a Análise Léxica. Esta que é a primeira etapa do processo de compilação, onde basicamente se extrai os tokens da linguagem. A linguagem escolhida foi o T++, e o trabalho descreve um pouco sobre as características da linguagem. Ele também aborda sobre qual linguagem e biblioteca foi utilizada para realizar a análise, além de ao final mostrar exemplos de saída.*

## 1. Introdução

A análise léxica, é um processo de compilação que, faz uma varredura no código fonte, com o objetivo de converter os caracteres presentes no código em um conjunto de marcas (tokens). Esses tokens representam caracteres do código, como: laços de repetições (for, while, foreach, etc), comparações (if, else, etc), símbolos especiais (+, -, \*, /, etc). A saída desse processo de análise léxica é uma lista desses tokens, que serão utilizados nas próximas etapas do processo de compilação.

## 2. Objetivo

Estaremos realizando na disciplina um compilador para a linguagem de programação fictícia T++, esta que foi desenvolvida com o intuito de auxiliar com o aprendizado dos alunos na disciplina. Neste trabalho especificamente, estaremos realizando a análise léxica a primeira parte do processo de compilação, e para trabalho foi utilizado a linguagem de programação Python com a biblioteca PLY. A decisão de se utilizar Python, se deve ao fato de que é uma linguagem voltada para a produtividade, possuindo diversas ferramentas nativas que auxiliam o desenvolvedor a apenas focar em produzir. Já a escolha de usar a biblioteca PLY, se deve ao fato de que ela fornece a maioria dos recursos `lex/yacc` padrões, incluindo suporte para: produções vazias, regras de precedência, recuperação de erros e suporte para gramáticas ambíguas, além de ser relativamente simples de se usar.

### 3. Linguagem de Programação T++

Como já foi dito, a linguagem T++ foi desenvolvida para auxiliar no processo de aprendizagem dos alunos, e ela é totalmente em português. Quanto aos tipos básicos suportados pela linguagem são tipos inteiro e flutuante, além de contar com suporte a arranjos unidimensionais e bidimensionais. Nessa linguagem quando o retorno de uma função é omitido ela automaticamente vira um procedimento, caso contrário vira uma função. Nas subseções abaixo iremos ver mais detalhes sobre as características dessa linguagem.

#### 3.1. Tipos de Variáveis

Os tipos de variáveis suportadas pela linguagem são:

- Números inteiros
- Números reais
- Números com notação científica

#### 3.2. Operações

Os tipos de operações suportadas pela linguagem são apresentados na Tabela 1.

<i><b>OPERAÇÃO</b></i>	<i><b>SÍMBOLO</b></i>
<b>Soma</b>	+
<b>Subtração</b>	-
<b>Multiplicação</b>	*
<b>Divisão</b>	/
<b>OR</b>	
<b>AND</b>	&&
<b>NOT</b>	!

Tabela 1. Operações suportadas na linguagem T++.

#### 3.3. Comparações

Os tipos de comparações suportadas pela linguagem se encontram na Tabela 2.

<i><b>OPERAÇÃO</b></i>	<i><b>SÍMBOLO</b></i>
<b>Maior</b>	>
<b>Menor</b>	<
<b>Maior igual</b>	>=
<b>Menor igual</b>	<=
<b>Diferença</b>	<>

Tabela 2. Comparações suportadas pela linguagem T++.

#### 3.4. Palavras Reservadas

As palavras reservadas pela linguagem são mostradas na Tabela 3.

<i>se</i>
<i>então</i>
<i>senão</i>
<i>fim</i>
<i>repita</i>
<i>flutuante</i>
<i>retorna</i>
<i>ate</i>
<i>escreva</i>
<i>inteiro</i>

Tabela 3. Palavras reservadas da linguagem.

### 3.5. Exemplo de Código

No Código 1 temos um exemplo de um código em T++. Este código é um teste simples que utiliza quase tudo que foi discutido nas seções: variáveis, tipo de função, laço de repetição, comparação, igualdade, etc.

Código 1. Exemplo em T++

```

1      inteiro principal()
2          inteiro: a
3
4          a := 1
5
6          repita
7
8              se a = 10
9                  escreva(a)
10             fim
11
12             a := a + 1
13         ate a = 10
14
15         retorna(0)
16
17     fim

```

Nesse exemplo, temos apenas uma função, a função principal que retorna um valor inteiro. De início declaramos uma variável de nome *a* do tipo inteiro, e setamos o valor de 1 a ela. Logo após, temos um loop, que se repete até que essa variável *a* seja igual ao valor 10. Dentro desse loop fazemos apenas uma verificação simples, verificamos se o valor de *a* é 10, caso positivo escrevemos esse valor com a função “escreva”, caso não seja 10, aumentamos em 1 no valor da variável *a*. E ao final do código, retornamos 0.

## 4. Expressões Regulares

Como foi dito anteriormente, a análise léxica faz uma varredura no código fonte, com o objetivo de identificar tokens. Para realizar esse processo de identificação, foi utilizado expressões regulares para cada um dos símbolos, tanto simples quanto complexos. Um expressão regular é uma técnica que provê uma forma concisa e flexível de identificar

cadeias de caracteres específicas. A Tabela 4 mostra as expressões simples que foram usadas para identificação de alguns tokens.

<i>OPERAÇÃO</i>	<i>SÍMBOLO</i>
<b>Adição</b>	\ +
<b>Subtração</b>	-
<b>Multiplicação</b>	\ *
<b>Divisão</b>	\
<b>Abre parenteses</b>	\ (
<b>Fecha parenteses</b>	\ )
<b>Abre colchete</b>	[
<b>Fecha colchete</b>	]
<b>Virgula</b>	,
<b>Atribuição</b>	:=
<b>Dois pontos</b>	:
<b>E lógico</b>	&&
<b>OU lógico</b>	\ — \ —
<b>Negação</b>	!
<b>Diferença</b>	<>
<b>Menor igual</b>	<=
<b>Maior igual</b>	>=
<b>Menor</b>	<
<b>Maior</b>	>
<b>Igual</b>	=

Tabela 4. Expressões regulares simples

A Tabela 5 mostra as expressões regulares restantes, essas que são mais complexas ao se comparar com as da Tabela 4.

<i>TOKEN</i>	<i>EXPRESSÃO REGULAR</i>
<b>Dígito</b>	$([0-9])$
<b>Letra</b>	$([a-zA-Z\tilde{A}\tilde{a}\tilde{À}\tilde{à}\tilde{É}\tilde{é}\tilde{Í}\tilde{í}\tilde{Ó}\tilde{ó}\tilde{Õ}\tilde{õ}])$
<b>Sinal</b>	$([\backslash - \backslash + ]?)$
<b>ID</b>	$(([a-zA-Z\tilde{A}\tilde{a}\tilde{À}\tilde{à}\tilde{É}\tilde{é}\tilde{Í}\tilde{í}\tilde{Ó}\tilde{ó}\tilde{Õ}\tilde{õ}])(( [0-9] )+   -   ([a-zA-Z\tilde{A}\tilde{a}\tilde{À}\tilde{à}\tilde{É}\tilde{é}\tilde{Í}\tilde{í}\tilde{Ó}\tilde{ó}\tilde{Õ}\tilde{õ}])^*))$
<b>Inteiro</b>	$\backslash d+$
<b>Flutuante</b>	$\backslash d+[eE][ - + ]?\backslash d+   (\backslash . \backslash d+   nd + n.nd*) ([eE][ - + ]?\backslash d+)?$
<b>Notação Científica</b>	$([ - \backslash + ]?)([1-9])\backslash .([0-9])+[eE]([ - \backslash + ]?)([0-9]+)$
<b>Comentário</b>	$(\{((\backslash — \backslash n)^*)?\})$
<b>Nova linha</b>	$\backslash n+$

Tabela 5. Expressões regulares complexas.

## 5. Autômatos

Uma expressão regular define formalmente um padrão, mostrando quais os sub-padrões ou expressões regulares que a formam, com qual regularidade ou sequência cada uma delas aparece, e quais são os diferentes sub-padrões que podem ser reconhecidos em uma mesma posição do padrão. Vamos pegar por exemplo a Expressão 1, onde o “a” representa o carácter, o “\*” significa que a sequência de caracteres “[0-9]” (quais quer números de 0 a 9) podem se repetir.

$$(a * [0 - 9]) \quad (1)$$

Dado a seguinte sequência de caracteres: “a a1 a09 a123”, os caracteres retornados após a aplicação dessa expressão regular seria: “a1”, “a09” e “a123”. Qualquer expressão regular pode ser convertida em um autômato finito que ela descreve, e vice versa, ou seja, podemos converter a Expressão 1 em um automato finito, como vemos na Figura 1.

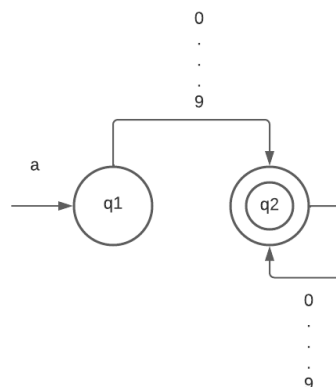


Figura 1. Autômato resultante da conversão da Expressão 1.

Assim como fizemos com a Expressão 1, podemos fazer o mesmo com todas as expressões regulares já mostradas acima, e para exemplificar iremos usar a Expressão 2 que identifica o token DIGITO.

$$([0 - 9]) \quad (2)$$

Para essa expressão, o automato resultante da conversão é o representado na Figura 2.

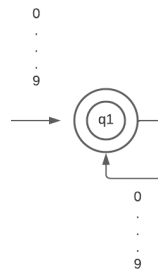


Figura 2. Autômato resultante da conversão da Expressão 2.

Também podemos observar na Figura 3, um automato para identificar a palavra reservada da linguagem “repita”.



Figura 3. Autômato que identifica a palavra reservada “repita”.

## 6. Detalhes de Implementação

Como já foi citado anteriormente, para realizar esse trabalho foi disponibilizado pelo professor um código inicial, utilizando a linguagem de programação Python, e a biblioteca Ply (Python Lex-Yacc). O Ply conta com 2 módulos separados: `lex.py` e `yacc.py`, para a análise léxica iremos estar utilizando o `lex.py`. O `lex.py` já prove uma implementação para pegar os tokens de um dado arquivo, como pode ser observado no Código 2. Para utilizar essa função, é necessário antes construir a instância do lexer, e passar o arquivo para ele realizar a leitura.

Código 2. Construção da instancia do lex

```

1      # construir a instancia do lexer
2      lexer = lex.lex(optimize=True, debug=True, debuglog=log)
3      lexer.input(source_file)
4
5      while True:
6
7          # pegar os tokens
8          tok = lexer.token()
9
10         # No more input
11         if not tok:
12             break
13
14         print(tok)
  
```

E dessa forma, para todos os tokens que definimos, podemos criar funções que irão se executar assim que o `lex` encontrar o token referenciado, como mostra o Código 3. No

caso dessa função, toda vez que o `lex` identificar o token da notação científica (expressão regular identificada na Tabela 5) ele irá apenas retornar o token, que é justamente o que queremos.

#### Código 3. Exemplo função `lex`.

```
1 @TOKEN(notacao_cientifica)
2 def t_NUM_NOTACAO_CIENTIFICA(token):
3
4     return token
```

Enquanto no Código 3, apenas retornamos o token, temos casos como o do Código 4 em que temos que realizar um tratamento, neste caso em específico, temos que pular as linhas.

#### Código 4. Exemplo função nova linha do `lex`.

```
1 @TOKEN(nova_linha)
2 def t_NEWLINE(token):
3
4     # para pular as linhas referentes ao valor
5     token.lexer.lineno += len(token.value)
6
7     # nao a necessidade de retornar o token
```

Foram feitas algumas melhorias em relação a saída do programa, foi implementado um argumento adicional 'd', onde na hora de execução do programa conseguimos observar uma saída mais detalhada da execução como vemos no Código 5.

#### Código 5. Saída mais detalhada.

```
1 while True:
2
3     # pegar os tokens
4     tok = lexer.token()
5
6     # no more input
7     if not tok:
8         break
9
10    if detailed:
11        print(tok)
12        arq.write(str(tok) + '\n')
13    else:
14        print(tok.type)
15        arq.write(str(tok.type) + '\n')
```

Também foi implementado uma mensagem de erro mais detalhada também como mostra o Código 6.

Código 6. Função erro com adição de uma saída mais detalhada.

```
1 # tratamento de erros
2 def t_error(token):
3
4     # variaveis para controle
5     global error, detailed, arq
6
7     error = True
8
9     if detailed:
10         print(
11             'Foi encontrado um caracter invalido: "{}", na linha: {} e na coluna {}'
12             .format(token.value, token.lineno, token.lexpos)
13         )
14
15         arq.write(
16             'Foi encontrado um caracter invalido: "{}", na linha: {} e na coluna {}'
17             .format(token.value, token.lineno, token.lexpos)
18         )
19
20     else:
21         print(
22             "Caracter invalido '{}'"
23             .format(token.value[0])
24         )
25
26         arq.write(
27             "Caracter invalido '{}'"
28             .format(token.value[0])
29         )
30
31     # pulo o erro
32     token.lexer.skip(1)
```

Juntamente com o argumento adicional, também foi feito uma melhoria da saída da execução do código, onde toda vez que o mesmo for executado, seu resultado estará disponível em um arquivo chamado “saida.txt” na pasta do programa.

## 7. Execução do Código

Para testar nosso código, foi usado como código de entrada, um exemplo em T++ que implementa o algoritmo de busca linear, como mostra o Código 7.



### Código 7. Exemplo da busca linear em T++

```
1      inteiro: A[20]
2
3      inteiro busca(inteiro: e)
4
5          inteiro: retorno
6          inteiro: i
7
8          retorno := 0
9          i := 0
10
11         repita
12
13             se A[i] = 0
14                 retorno := 1
15             fim
16
17             i := i + 1
18
19         ate i = 20
20
21         retorna(retorno)
22
23     fim
24
25     inteiro principal()
26
27         inteiro: e
28         inteiro: i
29
30         i := 0
31
32         repita
33             A[i] := i
34             i := i + 1
35         ate i = 20
36
37         leia(e)
38         escreva(busca(e))
39         retorna(0)
40     fim
```

E para esse exemplo, temos a saída completa sem o parâmetro “d” na hora da execução na Figura 4.

1	INTEIRO	36	NUM_INTEIRO	71	ID
2	DOIS_PONTOS	37	FIM	72	ID
3	ID	38	ID	73	ATRIBUICAO
4	ABRE_COLCHETE	39	ATRIBUICAO	74	ID
5	NUM_INTEIRO	40	ID	75	MAIS
6	FECHA_COLCHETE	41	MAIS	76	NUM_INTEIRO
7	INTEIRO	42	NUM_INTEIRO	77	ATE
8	ID	43	ATE	78	ID
9	ABRE_PARENTESE	44	ID	79	IGUAL
10	INTEIRO	45	IGUAL	80	NUM_INTEIRO
11	DOIS_PONTOS	46	NUM_INTEIRO	81	LEIA
12	ID	47	RETORNA	82	ABRE_PARENTESE
13	FECHA_PARENTESE	48	ABRE_PARENTESE	83	ID
14	INTEIRO	49	ID	84	FECHA_PARENTESE
15	DOIS_PONTOS	50	FECHA_PARENTESE	85	ESCREVA
16	ID	51	FIM	86	ABRE_PARENTESE
17	INTEIRO	52	INTEIRO	87	ID
18	DOIS_PONTOS	53	ID	88	ABRE_PARENTESE
19	ID	54	ABRE_PARENTESE	89	ID
20	ID	55	FECHA_PARENTESE	90	FECHA_PARENTESE
21	ATRIBUICAO	56	INTEIRO	91	FECHA_PARENTESE
22	NUM_INTEIRO	57	DOIS_PONTOS	92	ID
23	ID	58	ID	93	ABRE_PARENTESE
24	ATRIBUICAO	59	INTEIRO	94	NUM_INTEIRO
25	NUM_INTEIRO	60	DOIS_PONTOS	95	FECHA_PARENTESE
26	REPITA	61	ID	96	FIM
27	SE	62	ID		
28	ID	63	ATRIBUICAO		
29	ABRE_COLCHETE	64	NUM_INTEIRO		
30	ID	65	REPITA		
31	FECHA_COLCHETE	66	ID		
32	IGUAL	67	ABRE_COLCHETE		
33	ID	68	ID		
34	ID	69	FECHA_COLCHETE		
35	ATRIBUICAO	70	ATRIBUICAO		

Figura 4. Saída da execução da análise léxica.

Já a Figura 5 mostra a saída com o parâmetro “d”.

1	LexToken(INTEIRO,'inteiro',2,1)	36	LexToken(NUM_INTEIRO,'1',14,136)	71	LexToken(ID,'1',29,268)
2	LexToken(DOIS_PONTOS,':',2,8)	37	LexToken(FIM,'fim',15,148)	72	LexToken(ID,'1',30,272)
3	LexToken(ID,'A',2,18)	38	LexToken(ID,'1',16,148)	73	LexToken(ATRIBUICAO,':-',30,274)
4	LexToken(ABRE_COLCHETE,['',2,11)	39	LexToken(ATRIBUICAO,':-',16,158)	74	LexToken(ID,'1',30,277)
5	LexToken(NUM_INTEIRO,'20',2,12)	40	LexToken(ID,'1',16,153)	75	LexToken(MAIS,'+',30,279)
6	LexToken(FECHA_COLCHETE,']',2,14)	41	LexToken(MAIS,'+',16,155)	76	LexToken(NUM_INTEIRO,'1',30,281)
7	LexToken(INTEIRO,'inteiro',4,17)	42	LexToken(NUM_INTEIRO,'1',16,157)	77	LexToken(ATE,'até',31,284)
8	LexToken(ID,'busca',4,25)	43	LexToken(ATE,'até',17,160)	78	LexToken(ID,'1',31,288)
9	LexToken(ABRE_PARENTESE,'(',4,38)	44	LexToken(ID,'1',17,164)	79	LexToken(IGUAL,'=',31,290)
10	LexToken(INTEIRO,'inteiro',4,31)	45	LexToken(IGUAL,'=',17,166)	80	LexToken(NUM_INTEIRO,'20',31,292)
11	LexToken(DOIS_PONTOS,':',4,38)	46	LexToken(NUM_INTEIRO,'20',17,168)	81	LexToken(LEIA,'leia',33,297)
12	LexToken(ID,'e',4,40)	47	LexToken(RETORNA,'retorna',19,173)	82	LexToken(ABRE_PARENTESE,'(',33,301)
13	LexToken(FECHA_PARENTESE,')',4,41)	48	LexToken(ABRE_PARENTESE,'(',19,180)	83	LexToken(ID,'e',33,302)
14	LexToken(INTEIRO,'inteiro',6,46)	49	LexToken(ID,'retorna',19,181)	84	LexToken(FECHA_PARENTESE,')',33,303)
15	LexToken(DOIS_PONTOS,':',6,53)	50	LexToken(FECHA_PARENTESE,')',19,188)	85	LexToken(ESCREVA,'escreva',34,306)
16	LexToken(ID,'retorno',6,55)	51	LexToken(FIM,'fim',20,190)	86	LexToken(ABRE_PARENTESE,'(',34,313)
17	LexToken(INTEIRO,'inteiro',7,64)	52	LexToken(INTEIRO,'inteiro',22,195)	87	LexToken(ID,'busca',34,314)
18	LexToken(DOIS_PONTOS,':',7,71)	53	LexToken(ID,'principal',22,203)	88	LexToken(ABRE_PARENTESE,'(',34,319)
19	LexToken(ID,'1',7,73)	54	LexToken(ABRE_PARENTESE,'(',22,212)	89	LexToken(ID,'e',34,320)
20	LexToken(ID,'retorno',9,77)	55	LexToken(FECHA_PARENTESE,')',22,213)	90	LexToken(FECHA_PARENTESE,')',34,321)
21	LexToken(ATRIBUICAO,':-',9,85)	56	LexToken(INTEIRO,'inteiro',23,216)	91	LexToken(FECHA_PARENTESE,')',34,322)
22	LexToken(NUM_INTEIRO,'0',9,88)	57	LexToken(DOIS_PONTOS,':',23,223)	92	LexToken(ID,'retorno',35,325)
23	LexToken(ID,'1',10,91)	58	LexToken(ID,'e',23,225)	93	LexToken(ABRE_PARENTESE,'(',35,332)
24	LexToken(ATRIBUICAO,':-',10,93)	59	LexToken(INTEIRO,'inteiro',24,228)	94	LexToken(NUM_INTEIRO,'0',35,333)
25	LexToken(NUM_INTEIRO,'0',10,96)	60	LexToken(DOIS_PONTOS,':',24,235)	95	LexToken(FECHA_PARENTESE,')',35,334)
26	LexToken(REPITA,'repita',12,100)	61	LexToken(ID,'1',24,237)	96	LexToken(FIM,'fim',36,336)
27	LexToken(SE,'se',13,110)	62	LexToken(ID,'1',26,241)		
28	LexToken(ID,'1',13,113)	63	LexToken(ATRIBUICAO,':-',26,243)		
29	LexToken(ABRE_COLCHETE,['',13,114)	64	LexToken(NUM_INTEIRO,'0',26,246)		
30	LexToken(ID,'1',13,115)	65	LexToken(REPITA,'repita',28,250)		
31	LexToken(FECHA_COLCHETE,']',13,116)	66	LexToken(ID,'A',29,260)		
32	LexToken(IGUAL,'=',13,118)	67	LexToken(ABRE_COLCHETE,['',29,261)		
33	LexToken(ID,'e',13,120)	68	LexToken(ID,'1',29,262)		
34	LexToken(ID,'retorno',14,125)	69	LexToken(FECHA_COLCHETE,']',29,263)		
35	LexToken(ATRIBUICAO,':-',14,133)	70	LexToken(ATRIBUICAO,':-',29,265)		

Figura 5. Saída detalhada da execução da análise léxica.

## References

Beazley, D. M. Documentação ply.py. <https://www.dabeaz.com/ply/ply.tml>.

Gonçalves, R. A. Conjunto de testes. <https://moodle.utfpr.edu.br/mod/resource/view.php?id=725874>.

Gonçalves, R. A. Slides - análise léxica. [https://moodle.utfpr.edu.br/pluginfile.php/239681/mod\\_resource/content/4/aula-05-analise-lexica-lex-ply.md.slides.pdf](https://moodle.utfpr.edu.br/pluginfile.php/239681/mod_resource/content/4/aula-05-analise-lexica-lex-ply.md.slides.pdf).