

# Linguagens de Programação e Compiladores

Cap.2 - Análise Léxica (Pt.1)



Prof. MSc. Renzo P. Mesquita  
[renzo@inatel.br](mailto:renzo@inatel.br)

# Objetivos

- Apresentar de forma geral as principais partes e funções de um Analisador Léxico;
- Verificar para que serve e como funciona o reconhecimento de Tokens;
- Apresentar o que são Expressões Regulares e sua importância não só na construção de Analisadores Léxicos mas também de poderosos reconhecedores de padrões textuais;



## 2. Análise Léxica

*Parte 1*

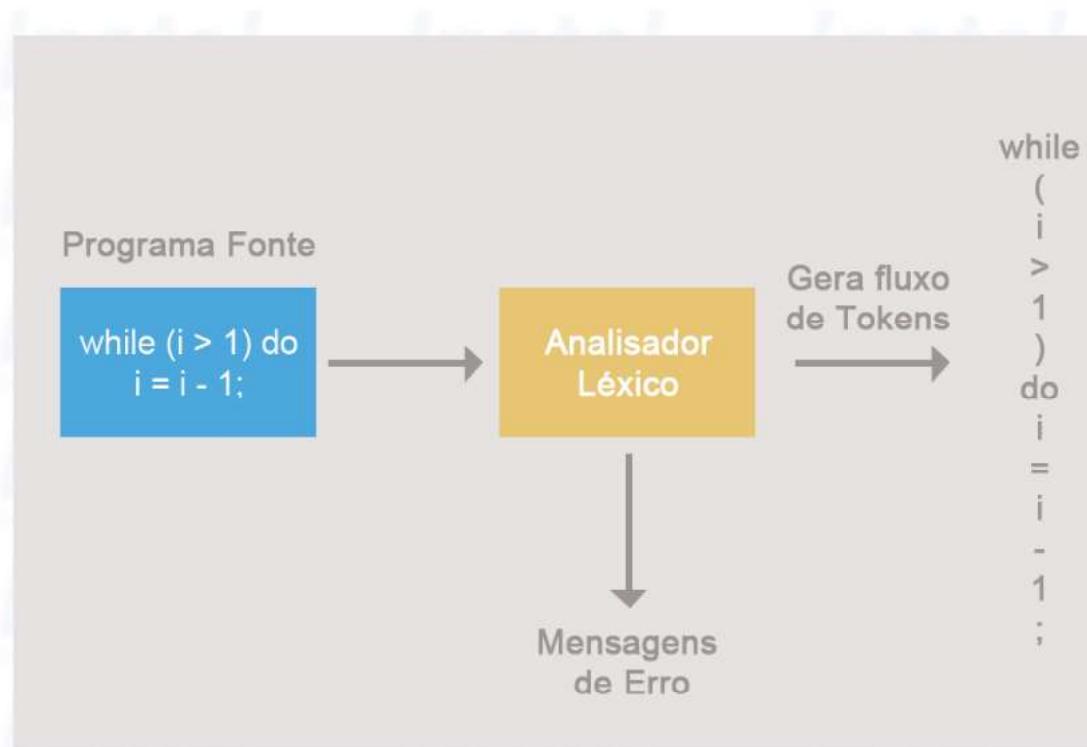
- 2.1. Introdução;*
- 2.2. Análise Léxica vs Análise Sintática;*
- 2.3. Tokens, Padrões e Lexemas;*
- 2.4. Expressões Regulares;*
- 2.5. Extensões para as Expressões Regulares.*



## 2.1. Introdução

Como já vimos, a Análise Léxica é a primeira fase de um compilador.

*A tarefa principal do Analisador Léxico é ler os caracteres de entrada do programa fonte, agrupá-los em Lexemas e produzir como saída uma sequência de Tokens para cada Lexema no programa fonte.*



## 2.1. Introdução

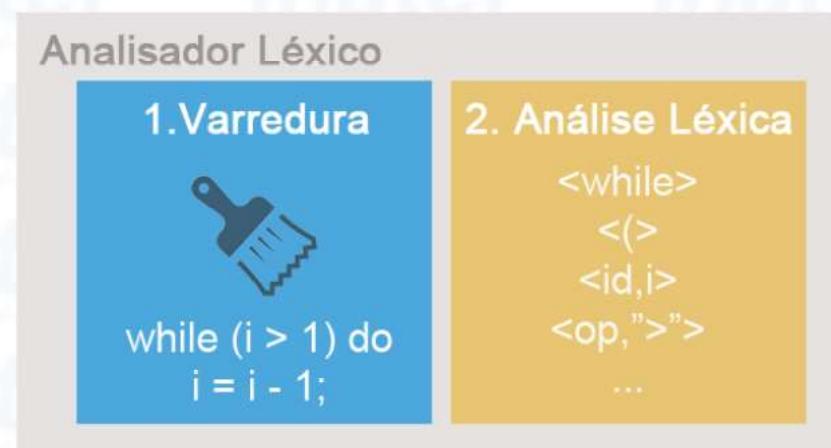
Geralmente, os Analisadores Léxicos são divididos em duas etapas:

### 1) Varredura ou Scanning

Processo de varredura utilizado, por exemplo, para **remover comentários e espaços desnecessários** (Como se fosse o Pré-processamento);

### 2) Análise Léxica

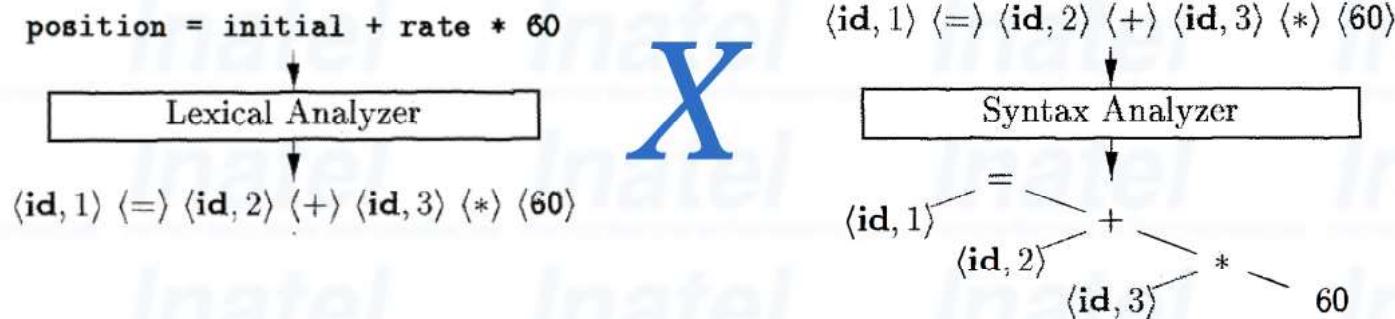
É a parte mais complexa, onde o Analisador Léxico **produz a sequência de Tokens como saída**;



## 2.2. Análise Léxica vs Análise Sintática

Existem vários motivos pelos quais as **partes de análises fundamentais de qualquer compilador** normalmente são separadas em Análise Léxica e Sintática. A principal delas é:

- Trabalhos Diferentes: a Análise Léxica tem como objetivo **encontrar padrões em um texto (lexemas) e transformá-los em Tokens (lexemas já classificados)**. Já a Análise Sintática **verifica se a interação entre um conjunto de Tokens é ou não válida** dentro da linguagem de programação, por meio de uma gramática;
- Apesar de em determinados momentos parecerem fazer tarefas parecidas, cada fase possui seu conjunto de técnicas específicas para solução de problemas computacionais;



## 2.3. Tokens, Padrões e Lexemas

Ao discutirmos Análise Léxica, usamos três termos relacionados, porém distintos, que são:

### *1. Lexema*

- Sequência de caracteres detectada no programa fonte que obedece a algum Padrão válido dentro de uma linguagem. Ex: nome de variável válido, um número inteiro, um número decimal, uma palavra reservada etc.

### *2. Token*

- É um par que consiste de <Nome\_Token, Lexema>;
- O nome do Token é uma palavra abstrata que representa um tipo de unidade lógica, ou seja, representa uma Classe de Lexemas (Ex: um nome para representar todos os nomes de variáveis válidos, um nome para representar todos números inteiros válidos, etc);

## 2.3. Tokens, Padrões e Lexemas

### 3. Padrão (Descrição)

É uma descrição da forma que os Lexemas de um Token podem assumir.

- **Palavras Reservadas:** o padrão é apenas uma sequência de caracteres que formam a própria palavra reservada;
- **Identificadores:** o padrão é uma estrutura mais complexa, que é formada por diferentes sequências de caracteres;

*Exemplos de possíveis Classes de Tokens (mas que podem mudar de linguagem para linguagem):*

Classe do Token	Padrão (Descrição)	Exemplo de Lexema
Palavra reservada “if”	Caracteres i,f	if
Palavra reservada “else”	Caracteres e,l,s,e	else
comparação	<,>,<=,!>,>=,==	<=, !=
identificador	Letra, seguida por letra e dígitos	x,x1,x2,nome
num	Qualquer constante numérica	3.1416
literal	Qualquer caractere entre “”	“Inatel”

## 2.3. Tokens, Padrões e Lexemas

### *EXEMPLO 1*

*Para os trechos de códigos que serão apresentados a seguir, construa uma lista de Tokens para cada um deles. Utilize das seguintes nomes de Tokens sugeridos abaixo:*

- "(" ")" ";" "{" "}" ":"
- PC (Para qualquer Palavra-chave)
- ID (para qualquer sequência de letra, seguida de letra ou dígito)
- OP (Para qualquer operador)
- COMP (Para qualquer comparador)
- NUM (qualquer número válido)
- LIT (cadeia de caracteres entre "")

*Exemplos de Tokens:* <PC,while>

<(>

<ID,num>

...

*e assim por diante. Conforme os padrões vão sendo identificados, vão sendo colocados em formato de Token um debaixo do outro...*

## 2.3. Tokens, Padrões e Lexemas

### *EXEMPLO 1*

#### *A) Código JAVA*

```
if (i == j){  
    z = 0;  
}else{  
    z = 1;  
}
```



#### *B) Código C++*

```
while (x1 != 3){  
    printf("Valor de X: %d", x1);  
    x1 = x1 + 1;  
}
```



#### *C) Código Python*

```
num = float(input("Entre com um num:"))  
while num > 0:  
    num-=0.5  
    print(num)
```



## 2.4. Expressões Regulares

Também chamado de *Regular Expression (Regex)*, é um tipo de notação que fornece uma forma concisa e flexível para identificar possíveis padrões em cadeias de caracteres.

- É uma notação muito importante para especificar padrões para os Lexemas (Ex: nome de variáveis válidas em uma linguagem, números válidos em uma linguagem, etc.);
- Ou seja, oferecem um modo declarativo de expressar padrões que serão aceitos;
- Traduz um conjunto de padrões, possivelmente complicados e difíceis, numa expressão curta e relativamente fácil de interpretar.

```
+ ]]+|([":"] )*.*?\2  
+ ) ) \1(?:\ | [a-  
z0-9 ]+(:\ =\s*  
(?: [a-z0-9 ]+| )?
```

## 2.4. Expressões Regulares

Antes de entrarmos mais a fundo sobre como interpretar uma REGEX, é importante conhecermos alguns conceitos fundamentais como:

### A) ALFABETO ( $\Sigma$ )

É um **conjunto de símbolos finito e não vazio**. Os alfabetos comuns incluem por exemplo:

- Alfabeto Binário:  $\Sigma = \{0,1\}$ ;
- Conjunto de todas as letras minúsculas:  $\Sigma = \{a,b,c,\dots,z\}$ ;
- Conjunto de todos os caracteres da tabela ASCII;

*etc...*

1	2	3	#	\$	%	&	7	8	9
A	B	C	D	E	F	V	G	H	I
Q	R	S	T	U	u	v	W	X	Y
a	b	c	d	e	f	v	g	x	i
q	r	s	t	u	u	v	w	x	y
ü	é	â	ä	ö	å	á	ç	ê	ë
æ	í	ó	ñ	ñ	ñ	ñ	ñ	ñ	ñ

## 2.4. Expressões Regulares

### B) String

É uma cadeia finita de símbolos escolhidos de algum alfabeto.

- Exemplo: 010111 é uma String do alfabeto binário;
- String Vazia ( $\epsilon$ ): significa zero ocorrências de símbolos, ou seja, é uma String que pode ser escolhida de qualquer alfabeto;
- Comprimento da String  $|w|$ : indica o número de posições de uma String. Exemplo:  $|010111| = 6$ ;
- O conjunto de todas as Strings sobre um alfabeto  $\Sigma$  é denotado por  $\Sigma^*$ .



THIS IS A STRING

## 2.4. Expressões Regulares

### C) Linguagem

É um **conjunto de Strings** escolhidas a partir de algum  $\Sigma^*$ , ou seja, pode ser denotado por  $L \subseteq \Sigma^*$ .

**Ex:**

- 1) Linguagem de todas as Strings que consistem de N 0's (zeros) seguidos por N 1's, para  $N \geq 0$ :  $\{\epsilon, 01, 0011, 000111, \dots\}$ ;
- 2) Conjunto de Strings formadas por 0's e 1's com um número igual de cada um deles:  $\{\epsilon, 01, 10, 0011, 0101, 1001, \dots\}$ ;
- 3)  $\{\epsilon\}$ , a Linguagem consiste apenas na String vazia. É uma Linguagem sobre qualquer alfabeto.

*entre infinitas outras possibilidades...*

## 2.4. Expressões Regulares

No REGEX, geralmente utilizamos 3 tipos de operadores fundamentais para definirmos uma Linguagem, são elas:

*A) UNIÃO B) CONCATENAÇÃO e C) FECHAMENTO*

*A) UNIÃO*

Represeta a **união de duas Linguagens**, por exemplo, L e M, denotada por  $L \cup M$ . É o conjunto de Strings que estão em L ou M, ou em ambas.

*Ex:*

$L = \{001, 10, 111\}$  e  $M = \{\epsilon, 001\}$ , então  $L \cup M$  é:

$$L \cup M = \{\epsilon, 10, 001, 111\}$$

## 2.4. Expressões Regulares

### B) CONCATENAÇÃO

A concatenação de duas linguagens L e M é o conjunto de Strings que podem ser formadas **tomando-se qualquer String em L e concatenando-se essa String com qualquer String em M**. A concatenação de linguagens é denotada por um **(.) ponto ou sem nenhum operador**.

*Ex:*

$L = \{001, 10, 111\}$  e  $M = \{\epsilon, 001\}$ , então  $L.M$  (ou  $LM$ ) é:

$LM = \{001, 10, 111, 001001, 10001, 111001\}$



## 2.4. Expressões Regulares

### C) FECHAMENTO DE KLEENE (Repetição)

O Fechamento de uma linguagem  $L$  é denotado por  $L^*$  e representa o conjunto das Strings que podem ser formadas tomando-se qualquer número de Strings de  $L$ , possivelmente com repetições e concatenação.

*Ex:*

A) Se  $L = \{0,1\}$ , então  $L^*$  consiste no conjunto de quaisquer Strings formadas por 0's e 1's;  
Por exemplo:  $\epsilon, 0, 1, 01, 00, 11, 0101\dots$

B) Se  $L = \{0,11\}$ , então  $L^*$  consiste em quaisquer pares de 0's e 1's tais que os símbolos 1 quando apresentados são em pares.  
Por exemplo:  $\epsilon, 0, 11, 011, 00, 1111, 01111\dots$

## 2.4. Expressões Regulares

Depois de visto os conceitos básicos, agora suponha que quiséssemos descrever o conjunto de **nomes de variáveis válidas (identificadores)** da linguagem C utilizando finalmente a notação de REGEX.

Se estabelecermos que "letra" significa qualquer letra ou o símbolo underline, e que "dígito" significa qualquer dígito, então podemos descrever os identificadores da linguagem C por:

*letra (letra | dígito)\**

Observações:

- A barra vertical "|" significa união (ou alternância).
- Os parênteses são usados para agrupar subexpressões;
- O \* significa "zero ou mais ocorrências de", ou seja, é o fechamento;
- E a justaposição do primeiro letra com o restante da expressão significa concatenação;

## 2.4. Expressões Regulares

As Expressões Regulares normalmente contêm pares de parênteses desnecessários. Podemos retirar certos pares de parênteses se adotarmos as convenções que:

- O operador unário \* possui precedência mais alta e é associativo à esquerda;
- A concatenação tem a segunda maior precedência, e é associativa a esquerda;
- | tem a precedência mais baixa e também é associativa à esquerda;

Com essas convenções, por exemplo, podemos substituir a expressão regular:

*(a) | ((b)\*c)) por a | b\*c*

As duas expressões denotam "O conjunto de cadeias que são um único 'a' ou zero ou mais 'b's seguidos por um c".

## 2.4. Expressões Regulares

Outros operadores importantes para o REGEX:

*Dado o alfabeto  $\Sigma = \{a,b\}$*

- **Qualquer caracter sozinho (.)**: O Operador . (ponto) reconhece um único caracter qualquer do alfabeto;

*Ex:*  $ab = \{aba, abb\}$

$a.b. = \{aaba, aabb, abba, abbb\}$

- **Uma ou Mais Instâncias (+)**: representa o fechamento positivo de algo. Indica que deve existir pelo menos uma instância e também é associativo à esquerda;

*Ex:*  $a^+ = \{a, aa, aaa...\}$

$ba^+ = \{ba, baa, baaa...\}$

- **Zero ou Uma Instância (?)**: O Operador ? indica zero ou uma ocorrência de um caracter e também é associativo à esquerda;

*Ex:*  $ab? = \{a, ab\}$

$a?b? = \{\epsilon, a, b, ab\}$

## 2.4. Expressões Regulares

### EXEMPLO 2

Considere o alfabeto  $\Sigma = \{a,b,c\}$ . Qual a linguagem expressada pelas seguintes Expressões Regulares?

A)  $a | b$

F)  $(a | b)^+c$

B)  $(a | b)(a | b)$

G)  $(a | b)ab$

C)  $a^*$

H)  $abc?$

D)  $(a | b)^*$

I)  $a?bc.$

E)  $a | a^*b$

J)  $(a | bc)^*$

Lembretes:

- \* (zero mais ocorrências)
- + (uma ou mais ocorrências)
- ? (zero ou uma ocorrência)
- . (qualquer caractere)
- | (alternância)



## 2.5. Extensões para as Expressões Regulares

Com o passar dos anos o REGEX passou por alguns "upgrades" com o objetivo de ficar mais enxuto, mas ainda assim oferecer alta capacidade de reconhecimento.

Como exemplo de novas extensões, pode-se citar a [Classe de Caracteres](#).

**Ex:**

O Operador [] (colchete) busca uma ocorrência de um dos caracteres que se encontram no seu interior.

- [ab] -> busca por uma ocorrência de “a” ou “b”
- [a-h] -> busca por uma ocorrência de “a” ou “b” ou “c” ... ou “h”
- [a-z] -> qualquer letra minúscula
- [a-zA-Z] -> qualquer letra
- [0-9] -> qualquer número entre 0 e 9
- [.] -> O literal “.”
- [a-zA-Z][.] -> qualquer letra seguida do literal “.”
- [a-zA-Z] -> qualquer letra minúscula ou o literal “-”
- [^a-zA-Z] -> tudo que não é uma letra minúscula
- [a-zA-Z^] -> qualquer letra minúscula ou o literal “^”

• [RegEx]\*

## 2.5. Extensões para as Expressões Regulares

*Ex:*

Com o operador {m,n} podemos determinar a quantidade de caracteres mínimos e máximos das Strings pertencentes a uma Linguagem.

- [a-z]{2,4} -> podemos produzir uma String que possua de 2 até 4 ocorrências de qualquer letra minúscula
- [a-z]{1,} -> no mínimo uma ocorrência de uma letra
- [0-9]{3} -> 3 números
- [a-zA-Z0-9]{,5} -> qualquer String formada por até 5 caracteres que podem ser letras minúsculas ou maiúsculas ou números

Para prática e outras extensões presentes em interpretadores REGEX, uma dica é usar o <https://pythex.org/>

pythex

Your regular expression:

Hello\*

## 2.5. Extensões para as Expressões Regulares

### *EXEMPLO 3*

*Utilizando das Extensões das Expressões Regulares, escreva expressões que busquem identificar o seguinte:*

- A) Números inteiros negativos (Ex: -77);
- B) Palavras formadas apenas por vogais (Ex: uai);
- C) A palavra "Inatel" em qualquer combinação de maiúsculas/minúsculas (Ex: InaTeL);
- D) Nome de usuário válido com no mínimo 3 e no máximo 15 caracteres (Ex: JoTC100);
- E) O CEP de uma cidade Brasileira (Ex: 37540-000);
- F) Endereços de email do Gmail ou Yahoo (Ex: jotc@gmail.com ou jotc@yahoo.com);
- G) Horário no formato de 24 horas (Ex: 19:30);



## 2.5. Extensões para as Expressões Regulares

### EXEMPLO 4

*Números sem sinal (inteiros ou ponto flutuante), são cadeias como 5777, 0.099, 3.14E4 ou 1.5E-4. Como ficaria uma REGEX capaz de identificar todos estes perfis juntos?*

(Faça cada parte separadamente, para depois juntar todas e resolver o problema por completo. Esse é o "Jump of the Cat!" para resolver um REGEX mais complexo ;))

#### Dicas:

- A) Digito -> apenas um número qualquer;
- B) Digitos -> qualquer número inteiro positivo;
- C) numFrac -> apenas a parte fracionária;
- D) numExp -> apenas a parte do expoente;
- E) num -> (aqui já será a junção de todas as partes acima)



**Fim  
do  
Capítulo 2  
(Parte 1)**



**EXERCÍCIOS**