

# **Analizador Léxico: A primeira etapa de uma tradução**

**Diego J. Leite<sup>1</sup>, Nágila R. Honorato<sup>1</sup>**

<sup>1</sup>Engenharia de Computação – Departamento de Ciências Exatas (DEXA)

Universidade Estadual de Feira de Santana (UEFS)

Av. Transnordestina, s/n - Novo Horizonte, Feira de Santana – BA – Brasil – 44036-900

{diegojleite,nagyllarocha}@gmail.com

## **1. Introdução**

Uma linguagem de programação é um método padronizado para passar instruções para um computador. Possuindo várias formas de ser implementada, entre elas é através de um compilador. Usualmente o compilador é dividido em quatro partes principais: Analisador Léxico, Analisador Sintático, Analisador Semântico e Gerador de Código; cada uma delas com uma função específica.

O relatório abaixo traz a descrição da primeira parte do processo de composição do compilador, o analisador léxico, mostrando um pouco da sua teoria e de quais técnicas foram necessárias para a sua implementação. Além de uma explicação em alto nível da implementação e dos resultados obtidos a partir dela.

## **2. Fundamentação Teórica**

### **2.1. Compilador**

“Compiladores são programas de computador que traduzem de uma linguagem para outra” [LOUDEN 2004]. Recebendo como entrada um programa com linguagem de alto-nível, vinda de um programa fonte e produzindo um equivalente em baixo-nível (programa objeto) [AHO et al. 2008], [DELAMARO 2004], [LOUDEN 2004].

Segundo Delamaro {2004}, um compilador tradicionalmente é dividido em quatro partes principais, cada uma delas com uma função específica. São elas:

- Analisador Léxico: encarrega-se de varrer e de separar no programa fonte cada símbolo e verificar se possui algum significado para a linguagem ou não;
- Analisador Sintático: verifica se a sequência de símbolos está correta;
- Analisador Semântico: verifica a consistência semântica do programa fonte está de acordo com as definições da linguagem, ou seja, se o que é dito faz sentido;
- Gerador de Código: só é possível após passar por todos os processos acima, de forma correta. É nesta etapa em que o compilador cria de fato a linguagem objeto.

### **2.2. Analisador Léxico**

Como citado na Seção 2.1, o analisador léxico faz a varredura no programa fonte e o separa em marcas, “cada marca é uma sequência de caracteres que representa uma unidade de informação do programa fonte” [LOUDEN 2004], o lexema. Durante o processo de varredura, é necessário fazer o reconhecimento de determinados padrões. Estes padrões são reconhecidos por meio de expressões regulares e de autômatos finitos [LOUDEN 2004]. Possuindo as seguintes características:

- Expressões regulares: utilizam linguagens regulares “primitivas”, ou seja, um conjunto de símbolos individuais e combina-as por meio de alguns operadores, de modo a gerar uma nova linguagem [DELAMARO 2004], [LOUDEN 2004];
- Autômato finito: é um modelo de matemático de descrever tipos particulares de algoritmos. Sendo utilizados para descrever o processo de reconhecimento de padrões em cadeias de entrada [LOUDEN 2004].

Uma vez que são reconhecidos os lexemas, eles são classificados por seu padrão ou valor de atributo, este agrupamento é chamado de *token* [AHO et al. 2008]. Quando uma cadeia possui algum símbolo inválido na sua entrada, e por isto esta entrada não pode ser reconhecida como um *token* válido, é chamado de erro léxico [DELAMARO 2004].

### 3. Desenvolvimento

Partindo das regras estabelecidas pela Tabela 1 foi possível desenvolver o analisador léxico. Antes disso foi necessário definir qual era o conjunto de caracteres primitivos que comporia cada *token*, sendo eles: Letras (L), Dígito (D), Símbolo (S) e Espaço (E), Tabela 1. Em seguida, definir o que finalizaria cada *token* (Tabela 2).

**Tabela 1. Estrutura Lóxica da Linguagem**

<b>Palavras Reservadas</b>	<i>class, final, if, else, for, scan, print, int, float, bool, true, false, string</i>
<b>Identificadores (Id)</b>	$L \{L \mid D \mid -\}$
<b>Número</b>	$[-] E D \{D\} [.D \{D\}]$
<b>Dígito</b>	$[0..9]$
<b>Letra</b>	$[a..z] \mid [A..Z]$
<b>Operador Aritmético</b>	$+, -, *, /, \%$
<b>Operador Relacional</b>	$!=, =, <, <=, >, >=$
<b>Operador Lógico</b>	$!, \&\&,   $
<b>Delimitador de Comentários</b>	// isso é um comentário de linha /* isso é um comentário de bloco*/
<b>Delimitadores</b>	$;, () [] \{ \}$
<b>Cadeia de Caracteres</b>	$\text{”}\{L \mid D \mid S \mid \text{”}\}$
<b>Símbolo</b>	ASCII 32 à 126 (exceto ASCII 34)
<b>Espaço</b>	$\{ASCII\ 9 \mid ASCII10 \mid ASCII13 \mid ASCII32\}$

Uma vez definido o conjunto de caracteres e o que delimita cada um, foi desenvolvendo o autômato finito correspondente para cada função, sendo eles: identificador (Id) (Figura 1), número (Num) (Figura 2), operadores aritmético (OA), relacional (OR) e lógico (OL), Figuras 4, 5, 6, respectivamente, delimitadores de comentário (Comentário) (Figura 7), cadeia de caracteres (CC) (Figura 8) e delimitadores (Del) (Figura 3).

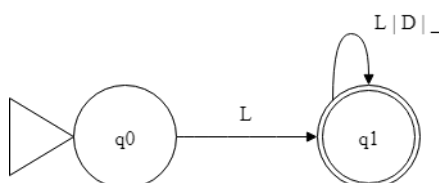
Vale ressaltar que apesar de palavra reservada (PR) (Tabela 1) possuir elementos formados por cadeias de símbolos primitivos ela não possui um autômato, devido ao fato

**Tabela 2. Tabela de separação de *tokens***

	<b>Finalizadores</b>
<b>Identificador / Palavra Reservada</b>	E, ", Del, OA, OR, OL
<b>Numero</b>	Del, OA, OR, OL, E, "
<b>Comentário de Linha</b>	ASCII 10, ASCII 13
<b>Comentário de Bloco</b>	*/
<b>Cadeia de Caracteres</b>	", ASCII 10, ASCII 13
<b>Delimitadores</b> <b>Operadores Aritméticos</b> <b>Operadores Relacionais</b> <b>Operadores Lógico</b>	são naturalmente finalizadores, além de E

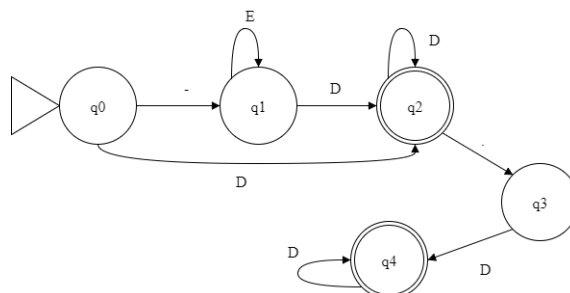
dela ser um subconjunto específico do grupo de Id, ou seja, toda PR é um Id, mas nem todo Id é uma PR. Sendo assim atendida pelo autômato da Figura 1.

No autômato de Id, Figura 1, tem-se que ele sempre inicia com uma letra, podendo receber em seguida na sua cadeia de entrada, mais L, D ou *underline* (.). Caso apareça um S na cadeia, independente de onde, será informado erro de (Identifier)\_Malformed.



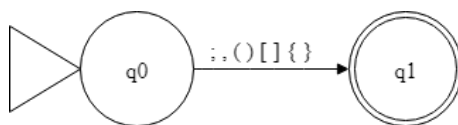
**Figura 1. Autômato finito para identificador**

Para o autômato de Num, Figura 2, pode-se iniciar com menos (-) ou não, mas o próximo caractere deve ser um pelo menos um D. Além disso, caso venha um ponto (.) o próximo caractere deve ser pelo menos um D, para o autômato poder ser finalizado corretamente. Caso o ponto (.) não seja seguido por um D, ou seja o ultimo caractere da sequência haverá o erro de (Number)\_Malformed.

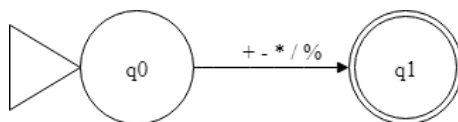


**Figura 2. Autômato finito para Numero**

Os autômatos de Del, Figura 3 e OA, Figura 4 são semelhantes em sua estrutura, diferindo apenas na sua cadeia de entrada. Sendo a cadeia validada, pela presença de qualquer símbolo do seu conjunto (Tabela 1).

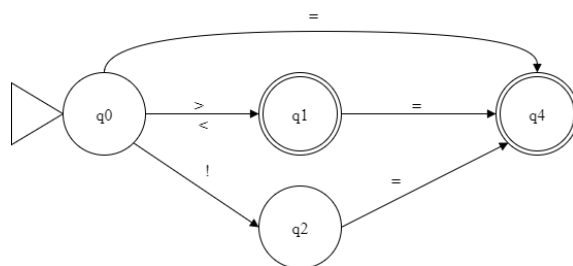


**Figura 3. Autômato finito para Delimitadores**



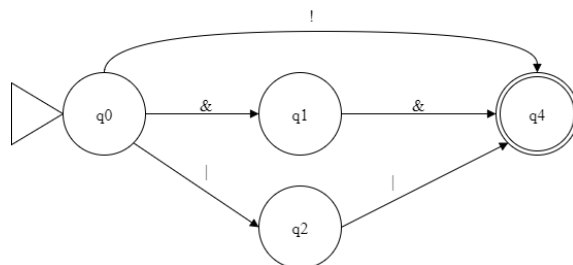
**Figura 4. Autômato finito para Operador Aritmético**

No OR, Figura 5, o autômato possui três decisões possíveis de acordo com a entrada. Em caso de igualdade (=) ele naturalmente está encerrado. Em caso de maior que (>) e menor que (<), pode-se encerrar neste caractere ou então, o próximo caractere deve ser uma igualdade (=). Por último temos a negação (!) ela deve vir obrigatoriamente seguida de uma igualdade (=), formando o símbolo de diferente, caso contrário, se vier seguida por qualquer outro símbolo, a negação (!) será considerada um OL.



**Figura 5. Autômato finito para Operador Relacional**

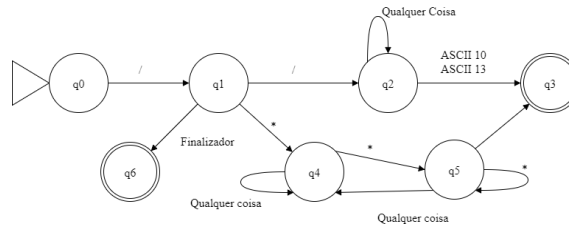
O autômato de OL, Figura 6, semelhante ao de OR também possui três entradas, a negação (!), o colorampersand (&) e o pipe (|), que devem vir sempre aos pares. Caso venha apenas um, será considerado um erro de (Logic Operator)\_Malformed.



**Figura 6. Autômato finito para Operador Lógico**

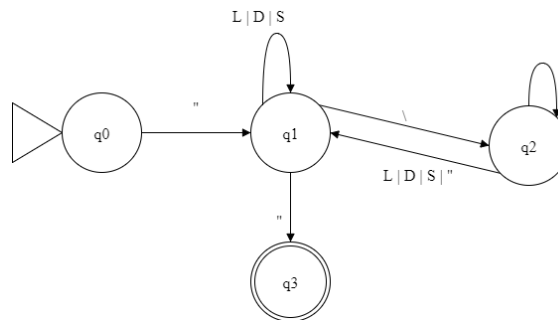
Na Figura 7, temos o autômato de comentário. Caso a barra (/) venha acompanhada de qualquer outro caractere ela será um OA. Caso ela seja seguida por outra barra (/) ou um asterisco (\*), se configurar como um comentário. Para comentário de linha (//) qualquer caracter que vier depois até a quebra de linha (ASCII 10 e 13) será considerado

parte do comentário. No comentário em bloco (/\*) tudo é considerado comentário até que se receba um \*/. Caso o comentário em bloco não seja fechado, ou seja, sem o \*/ , ele irá ler até o fim do arquivo indicando ao fim (Block Comment)\_Malformed.



**Figura 7. Autômato finito para Delimitador de Comentário**

Uma CC, Figura 8 é iniciada por aspas duplas (”), podendo receber L, D, S ou \\* (esse mecanismo permite que dentro do próprio conjunto possa ter aspas duplas (”) inclusas), sendo finalizado por aspas duplas(”). Caso a aspa dupla (”) não seja fechada, o erro percorrerá até a próxima quebra de linha, se caracterizando como uma (Characters Chain)\_Malformed.



**Figura 8. Autômato finito para Cadeia de Caracteres**

Após o desenvolvimento do automato foram feitas as expressões regulares, sendo elas:

- **Id:**  $L^*(L \mid D \mid \_)^*$
- **Num:**  $-?E^*D^+(\cdot D^+)?$
- **OA:**  $(+ \mid - \mid * \mid / \mid \% )^*$
- **OR:**  $(= \mid (< \mid >) \mid =? \mid != )^*$
- **OL:**  $(\&\& \mid ' \mid | \mid ! )^*$
- **Comentário:**  $/((/^{+}(L \mid D \mid OA \mid OR \mid OL \mid CC \mid S \mid ")^*(ASCII10 \mid ASCII13)) \mid (*^{+}(L \mid D \mid OA \mid OR \mid OL \mid CC \mid S \mid " \mid E)^{*} ** /))$
- **CC:**  $"((L \mid D \mid S \mid )^* \mid (\backslash^*(L \mid D \mid S \mid ")^*))"$
- **Del:**  $(; \mid , \mid ' \mid ( \mid ' \mid ' \mid [ \mid ] \mid \{ \mid \} )$

Obs.: os caracteres que aparecem entre aspas simples ('), implicam que são literais e não parte da representação (parenteses ('(' ')') e pipe ('|')).

## 4. Resultados e Discussões

### 4.1. Informações

O analisador léxico foi desenvolvido através da linguagem de programação Java 8 [jav ], valendo-se de suas bibliotecas, utilizando o Ambiente de Desenvolvimento Integrado (IDE) IntelliJ IDEA CE [int ]. Possuindo algumas características básicas:

- O arquivo de entrada análise **program.h5** encontra-se dentro do diretório **./entrada/**, localizado na raíz do projeto;
- A saída da análise pode ser encontrada no console do Java e em um arquivo de saída **program.out** dentro do diretório **./entrada/**. Nele serão encontrados os *tokens*, erros léxicos, caso existam, do contrário uma mensagem de sucesso, sendo separados por quebra de linha. Os *tokens* serão apresentados com o seguinte padrão:
  - **Token:** <lexema, tipo >;
  - **Erro léxico:** <lexema, (tipo)\_mal\_formado>;
- PR sempre devem ser escritas em minúsculo
- A gramática é formada por: Letras, Dígitos, Símbolos e Espaço. Logo, nunca um *token* virá com o tipo L, D, S ou E.

### 4.2. Etapas da análise léxica

Para este projeto a análise léxica é subdividida em 7 etapas:

1. Leitura do arquivo de análise
2. Filtragem de Cadeias de Caracteres, Comentários em Linha (CL) e Comentários em Bloco (CB);
3. Remoção de qualquer tipo de espaço do que não é CC, CL ou CB;
4. Aplicação de expressões regulares para cada tipo de categoria (PR, Del, OA, OR e OL) no programa pré-processado para iniciar a classificação dos padrões encontrados;
5. Checagem e aplicação de regras de precedência nos padrões encontrados pelas expressões regulares;
6. Classificação dos *Tokens* em: CC, CL, CB, PR, Del, OA, OR, OL, Id e Erros;
7. Impressão do arquivo processado no console do Java e criação do arquivo de saída com o resultado.

### 4.3. Descrição em Auto-nível

De modo geral o processo de análise, inicia-se com a leitura de linha por linha do arquivo **program.h5**, localizado na pasta **./entrada** na raíz do projeto, salvando toda sua informação em uma *string* (Snippet 1).

---

```
int x = 10;
float y = 3.14;

print("Result: ");
print(x * y);
```

---

**Snippet 1. Arquivo de entrada**

A filtragem do arquivo de entrada (Snippet 1) é feita através da busca dos identificadores CC, CL e CB. O que não fizer parte desse grupo de identificadores terá os espaços removidos (Snippet 2). Além disso, estes *tokens* serão ignorados até que sejam classificados.

---

```
int x=10; float y=3.14; print (
-----
"Result: "
-----
); print (x*y);
```

---

#### **Snippet 2. Arquivo filtrado. O separador — para indicar o *split* da *string***

Em seguida uma função de parsing (Snippet 3) é aplicada de forma recursiva para cada categoria de delimitador existente, fazendo o *split* dos padrões encontrados e segmentando ainda mais a lista encadeada de entrada.

---

```
public LinkedList<String> parser(LinkedList<String> list,
    String[] category) {
    String[] parts;
    String pattern;

    for(int i = 0; i < category.length; i++) {
        for(int j = 0; j < list.size(); j++) {
            pattern = "(" + category[i] + " | (" + category[i] + " )";
            parts = list.get(j).split(pattern);

            if(parts.length > 1) {
                list.remove(j);

                for(int k = 0; k < parts.length; k++) {
                    list.add(j + k, parts[k]);
                }
            }
        }
    }
    return list;
}
```

---

#### **Snippet 3. Função de *parsing* utilizada para segmentar a lista de entrada**

Uma vez que a lista encadeada é separada pelos finalizadores (Figura 9 - Antes), faz-se a checagem de regras de precedência em toda a lista para corrigir possíveis divisões incorretas, gerando uma nova lista (Figura 9 - Depois) pronta para ter seus *tokens* classificados.

Antes	Depois
int	int
x	x
=	=
10	10
;	;
float	float
y	y
*	*
3.14	3.14
;	;
pr	print
int	{
{	"Result: "
"Result: "	}
}	;
;	print
pr	{
int	x
{	*
x	y
*	
y	;
}	
;	

**Figura 9. Lista encadeada separada por finalizadores (Antes) e lista encadeada separada por finalizadores com regras de precedência aplicadas (Depois).**

Para finalizar, todos os itens (*tokens*) da lista encadeada são identificados e classificados (Snippet 4), gerando o arquivo de saída **program.out** na pasta **./entrada**, localizada na raiz do projeto.

---

```

<int, Reserved Word>
<x, Identifier>
<=, Relational Operator>
<10, Number>
<;, Delimiter>
<float, Reserved Word>
<y, Identifier>

```



```
<=, Relational Operator>
<3.14, Number>
<;, Delimiter>
<print, Reserved Word>
<(, Delimiter>
<"Result: ", Characters Chain>
<), Delimiter>
<;, Delimiter>
<print, Reserved Word>
<(, Delimiter>
<x, Identifier>
<*, Arithmetic Operator>
<y, Identifier>
<), Delimiter>
<;, Delimiter>
```

---

#### Snippet 4. Resultado da análise léxica

## 5. Conclusão

Todos os requisitos solicitados pelo problema foram implementados e estão funcionando corretamente. A detecção de *tokens* e erros está sendo realizada pelo analisador léxico e sendo apresentada tanto no console da aplicação quanto no arquivo de saída do projeto.

Como futuras melhorias podem ser realizados estudos sobre análise de complexidade dos algoritmos utilizados e das expressões regulares aplicadas, com o objetivo de reduzir a complexidade do código e conseguir ganhos de performance deste projeto.

## Referências

- IntelliJ IDEA jet brains. <https://www.jetbrains.com/idea/>. Acesso em: 26 set. 2017.
- Java SE oracle. <http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html>. Acesso em: 10 out. 2017.
- AHO, A. V., LAM, M. S., SETHI, R., and ULLAM, J. D. (2008). *Compiladores: princípios, técnicas e ferramentas*. São Paulo: Pearson Addison-Wesley, 2ª edição.
- DELAMARO, M. E. (2004). *Como construir um Compilador: utilizando ferramentas Java*. São Paulo: Novatec Editora Ltda., 1ª edição.
- LOUDEN, K. C. (2004). *Compiladores: princípios e práticas*. São Paulo: Pioneira Thomson Learning, 1ª edição.