

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CENTRO TECNOLÓGICO  
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA  
CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

**RELATÓRIO DO ANALISADOR LÉXICO**

**Alunos: João Felliipe Uller  
Leonardo Kreuch  
Uriel Kindermann Caminha**

**Disciplina: Construção de Compiladores  
Professor: Álvaro Franco**

**FLORIANÓPOLIS, 26 DE FEVEREIRO DE 2021**

## **1 - IDENTIFICAÇÃO DOS TOKENS**

### **1.1 Elementos da Linguagem**

Ident  
Int\_constant  
Float\_constant  
String\_constant  
Whitespace (ignorado)

### **1.2 Palavras Reservadas**

Def  
Int  
Float  
String  
Print  
Read  
Return  
Break  
If  
Else  
For  
New  
Null

### **1.3 Sinais Gráficos**

Lparen  
Rparen  
Lbrace  
Rbrace  
Lbracket  
Rbracket  
Semicolon  
Comma  
Assign  
Lesser  
Greater  
Lesserequal  
Greaterequal  
Equal  
Different  
Plus  
Minus  
Multiply  
Divide  
Module

## 2 - PRODUÇÃO DAS DEFINIÇÕES REGULARES PARA CADA TOKEN

### 2.1 Elementos da Linguagem

Ident	- >	([a-z]   [A-Z]) ([a-z]   [A-Z]   [0-9])*
Int_constant	- >	([0-9])+
Float_constant	- >	([0-9])+ ('.' ([0-9])+)?
String_constant	- >	"(Σ)*?"
Whitespace (ignorado)	- >	(' '   '\n'   '\t'   '\r')+

### 2.2 Palavras Reservadas

Def	- >	def
Int	- >	int
Float	- >	float
String	- >	string
Print	- >	print
Read	- >	read
Return	- >	return
Break	- >	break
If	- >	if
Else	- >	else
For	- >	for
New	- >	new
Null	- >	null

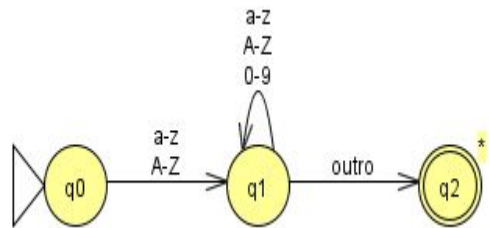
### 2.3 Sinais gráficos

Lparen	- >	(
Rparen	- >	)
Lbrace	- >	{
Rbrace	- >	}
Lbracket	- >	[
Rbracket	- >	]
Semicolon	- >	;
Comma	- >	,
Assign	- >	=
Lesser	- >	<
Greater	- >	>
Lessequal	- >	<=
Greaterequal	- >	>=
Equal	- >	==
Different	- >	!=
Plus	- >	+
Minus	- >	-
Multiply	- >	*
Divide	- >	/
Module	- >	%

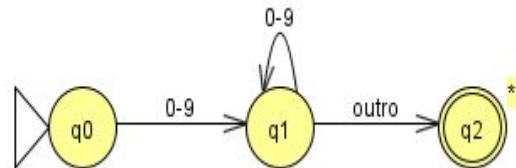
### 3 - DIAGRAMAS DE TRANSIÇÃO

#### 3.1 Elementos da Linguagem

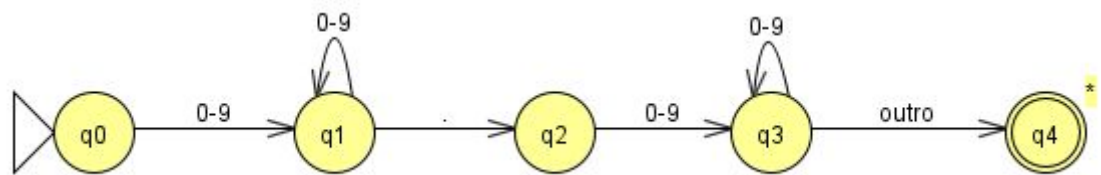
Ident



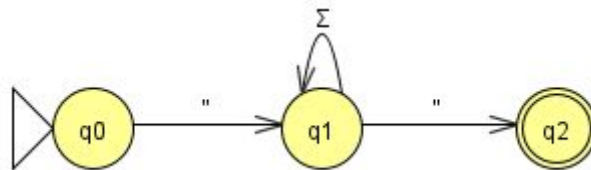
Int\_constant



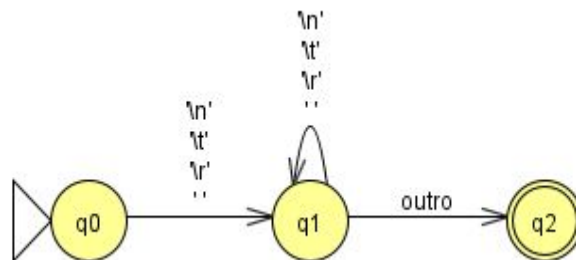
Float\_constant



String\_constant

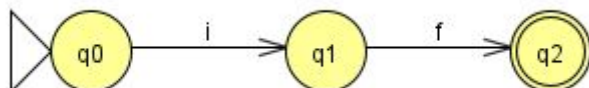


Whitespace

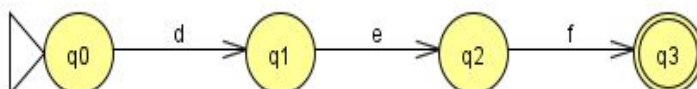


#### 3.2 Palavras Reservadas

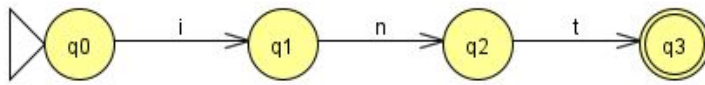
If



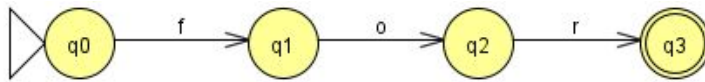
Def



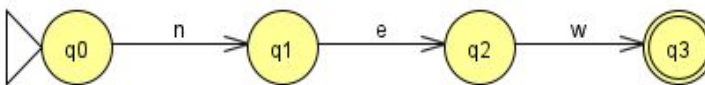
Int



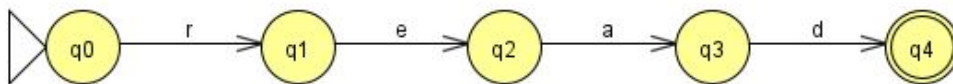
For



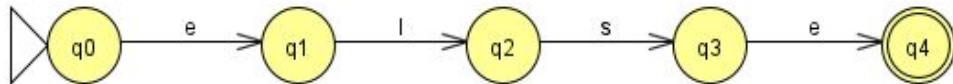
New



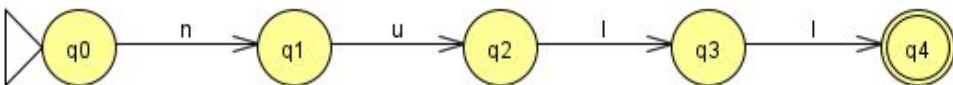
Read



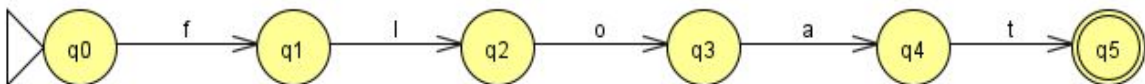
Else



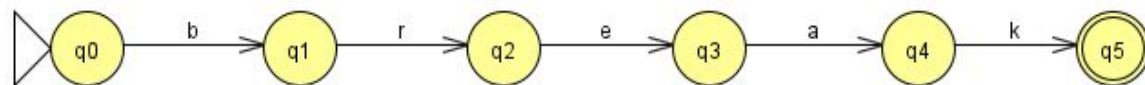
Null



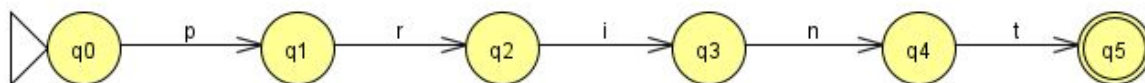
Float



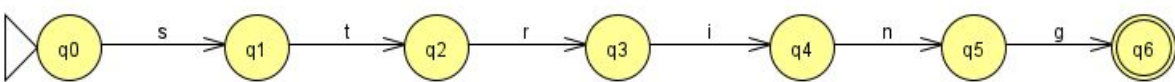
Break



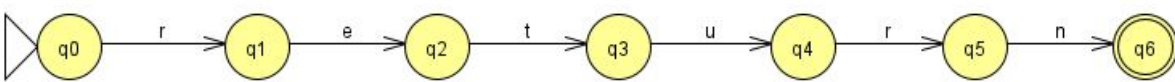
Print



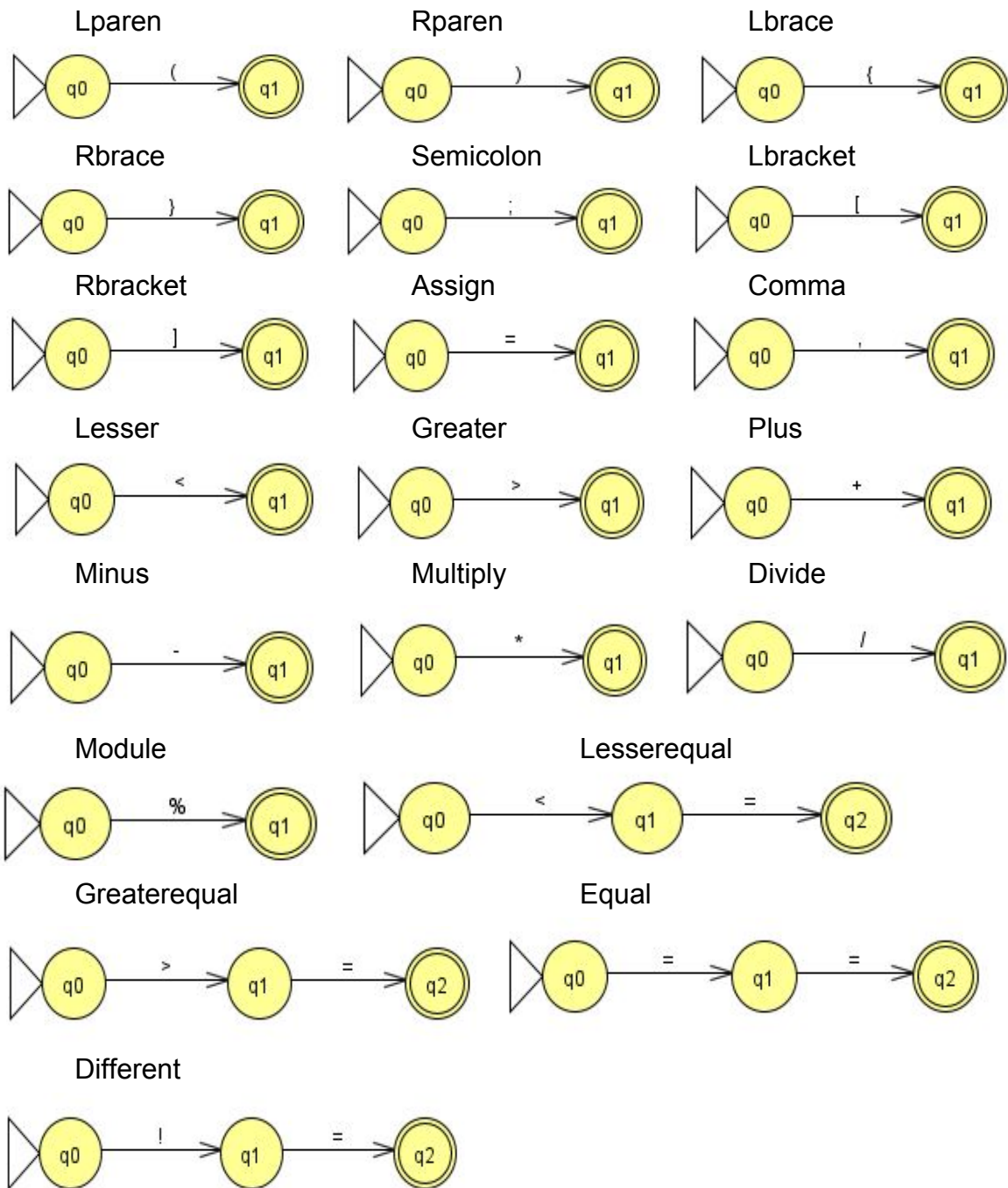
String



Return



### 3.3 Sinais Gráficos



## 4 - TABELA DE SÍMBOLOS

Para a implementação da tabela de símbolos (classe *SymbolTable*) do nosso exercício programa, a estrutura de dados utilizada como base é um *HashMap* do pacote *java.util*, distribuído nativamente entre as bibliotecas padrão da linguagem Java. Esta é uma implementação de uma *HashTable* baseada na implementação da interface *Map*, onde a ideia é que tem-se uma estrutura que associa uma chave única a um valor, que também é único em relação à sua chave, e tem tempo de operação constante para as operações básicas (*get* e *put*), assumindo uma dispersão apropriada dos elementos nos *buckets*, realizada pela função de *hashing*.

Na nossa implementação, esse *HashMap* é uma estrutura que associa uma chave que é uma *String* (lexema) a um Símbolo (classe *Symbol*), composto por um token e alguns de seus atributos, como o tipo desse token, além da linha e da coluna onde esse token aparece pela primeira vez no código fonte.

Além disso, a nossa tabela de símbolos possui métodos para a criação de uma nova entrada na tabela a partir de um *token* (*addLexeme()*), para buscar uma entrada dessa tabela a partir de uma dada chave (*getEntry()*), e para retornar a representação textual do estado atual da estrutura, em forma de tabela (*toString()*).

Na nossa implementação, apenas identificadores (*tokens* do tipo *Ident*) são armazenados na tabela de símbolos, sejam eles identificadores de funções ou variáveis. Na próxima seção teremos uma breve descrição da saída dada pelo exercício-programa, incluindo a representação textual de uma tabela de símbolos.

## 5 - DESCRIÇÃO DA SAÍDA

Na Figura 1, tem-se um exemplo para um trecho de código escrito na linguagem LCC e reconhecido pelo nosso exercício-programa, no lado esquerdo, e no lado direito um exemplo da lista de tokens gerada pela análise léxica desse trecho de código. A lista de tokens do nosso programa imprime o token identificado para cada um dos lexemas encontrados na ordem em que foram encontrados, da esquerda para a direita. A impressão dos tokens também respeita as quebras de linha, ou seja, lexemas encontrados numa mesma linha terão seus tokens impressos na mesma linha na saída, caso contrário em linhas distintas. No entanto, é importante notar que linhas e espaços em branco são ignorados pelo analisador léxico, e portanto não se refletem na impressão da lista de tokens.

<pre>1   2   print "Bem vindo a feira do Zeca\n"; 3   4   int opProduto; 5   string produtos[20]; 6   float soma; 7   int quantidadeProdutos; 8   int continua; 9   int i; 10   11   soma = 0; 12   quantidadeProdutos = 0;</pre>	<pre>-----LIST OF TOKENS----- Lbrace Print String constant Semicolon Int Ident Semicolon String Ident Lbracket Int_constant Rbracket Semicolon Float Ident Semicolon Int Ident Semicolon Int Ident Semicolon Int Ident Semicolon Ident Assign Int_constant Semicolon Ident Assign Int_constant Semicolon</pre>
---	--

Figura 1 - Trecho de um programa teste (esquerda) e lista de tokens gerada a partir deste (direita).

Na figura 2, tem-se um exemplo da tabela de símbolos gerada a partir da análise léxica do arquivo *feira.lcc*, disponibilizado como um dos arquivos de teste do exercício-programa, e que é o mesmo que podemos ver no trecho de código visto na Figura 1.

-----SYMBOL TABLE-----		
TOKEN NAME	TOKEN TYPE	LOCATION
continua	Ident	8:6
soma	Ident	6:8
produtos	Ident	5:9
i	Ident	9:6
quantidadeProdutos	Ident	7:6
opProduto	Ident	4:6

Figura 2 - Visualização textual de uma tabela de símbolos gerada a partir de um programa teste.

## 6 - ANTLR

Para a realização do trabalho utilizamos a ferramenta ANTLR (ANother Tool for Language Recognition), versão 4.9.1, na linguagem Java. Este é um gerador de análise poderoso para ler, processar, executar ou traduzir texto estruturado ou arquivos binários. É amplamente usado para construir linguagens, ferramentas e estruturas.

A Figura 3 mostra uma visão geral a respeito do funcionamento do ANTLR. Como pode-se ver, a ferramenta recebe como entrada uma gramática com extensão **.g4**, e a partir das definições gera um arquivo **.tokens**, que descreve os símbolos terminais da gramática, um **Lexer** com extensão **.java**, capaz de ler um código-fonte em busca dos tokens descritos na gramática, e um **Parser** também com extensão **.java**, capaz de construir e percorrer uma árvore sintática de acordo com as regras de produção descritas na gramática, além de alguns arquivos auxiliares da ferramenta.

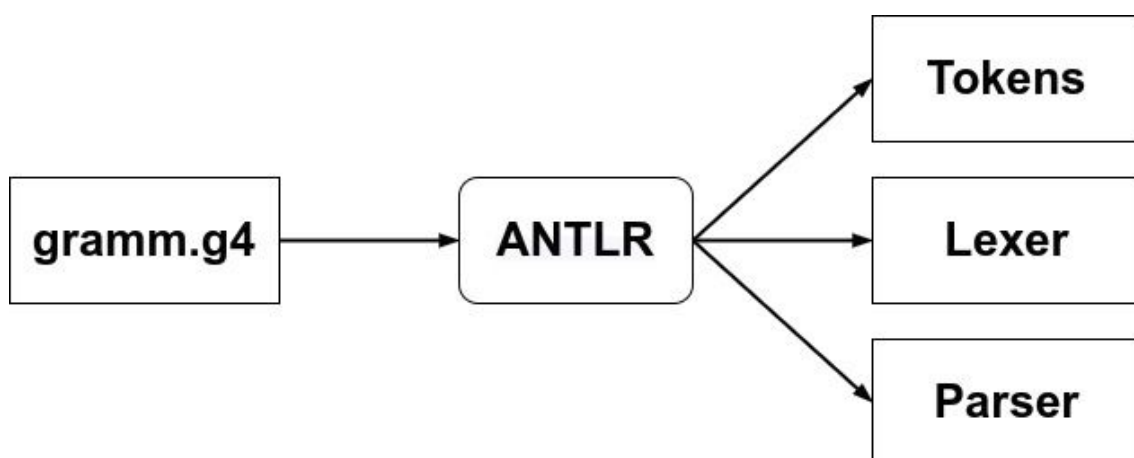


Figura 3 - Visão geral ANTLR.

Na Figura 4, tem-se um exemplo de uma gramática **.g4** que define uma linguagem para a definição de expressões matemáticas simples envolvendo as



quatro operações elementares, além de permitir esses cálculos com números inteiros e variáveis. Mais detalhes a respeito da definição de gramáticas para o ANTLR podem ser encontrados em: <https://github.com/antlr/antlr4/tree/master/doc>. Nesse formato de gramática, os tokens, ou símbolos terminais, são definidos tendo a primeira letra como maiúscula, enquanto as regras de produção são escritas todas em minúsculas.

```
1 grammar MyGrammar;  
2  
3 expr  : (Ident | T_num) (relop (Ident | T_num))+ T_EOF;  
4  
5 relop : T_mais | T_menos | T_vezes | T_divi;  
6  
7 T_EOF : ' ';  
8  
9 T_mais : '+';  
10  
11 T_vezes : '*';  
12  
13 T_menos : '-';  
14  
15 T_divi : '/';  
16  
17 T_num  : DIGIT+;  
18  
19 Ident  : LETTER (LETTER | DIGIT)*;
```

Figura 4 - Exemplo de uma gramática que reconhece expressões matemáticas simples.

```
≡ MyGrammar.interp  
≡ MyGrammar.tokens  
● MyGrammarBaseListener.java  
≡ MyGrammarLexer.interp  
● MyGrammarLexer.java  
≡ MyGrammarLexer.tokens  
● MyGrammarListener.java  
● MyGrammarParser.java
```

Figura 5 - Lista de arquivos gerados pelo ANTLR.

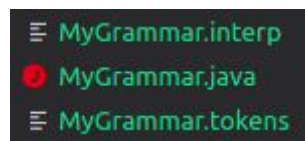
Uma vez que se tenha uma gramática bem formada, a mesma pode ser inserida no ANTLR que fará a verificação dos tokens e regras de produção, e gerará um analisador léxico (Lexer) e um analisador sintático (Parser) que reconhecem a linguagem definida pela gramática dada. Na Figura 5, têm-se a lista dos arquivos gerados pelo ANTLR como resultado da entrada da gramática especificada na figura 4. Uma vez que tanto o Lexer quanto o Parser são gerados como classes Java, pode-se, a partir daí, integrá-las no projeto, compilá-las e utilizá-las como classes regulares de qualquer outra biblioteca disponível.

No entanto, como nessa parte do trabalho estamos apenas interessados no analisador léxico, é possível definir uma gramática que possua apenas a especificação dos tokens através de definições regulares para que apenas o Lexer dessa linguagem seja gerado pelo ANTLR. Na Figura 6 pode-se ver um exemplo

desse tipo de gramática, onde apenas as definições dos símbolos terminais da nossa gramática anterior estão definidos, enquanto na Figura 7 pode-se ver que os únicos arquivos gerados pelo ANTLR foram aqueles relacionados ao Lexer da linguagem, sem a criação de um parser.

```
1  lexer grammar MyGrammar;
2
3  T_EOF: ' ';
4
5  T_mais: '+';
6
7  T_vezes: '*';
8
9  T_menos: '-';
10
11 T_divi: '/';
12
13 T_num: DIGIT+;
14
15 Ident: LETTER (LETTER | DIGIT)*;
```

Figura 6 - Gramática que contém apenas definições regulares de tokens.



MyGrammar.interp  
MyGrammar.java  
MyGrammar.tokens

Figura 7 - Arquivos gerados pelo ANTLR para Lexer Grammar da Figura 6.

É importante notar, no entanto, que além de definir apenas os símbolos terminais da gramática, deve-se informar ao ANTLR que a gramática definida conterá apenas definições regulares, o que é feito nesse código da figura 6 ao especificar que essa é uma gramática do tipo *Lexer Grammar*, na linha 1.

Dessa forma, para o presente trabalho, apenas foram especificados os símbolos terminais da gramática fonte da linguagem LCC, o que é suficiente para o ANTLR gerar um analisador léxico que reconheça os tokens dessa linguagem, o que é o principal objetivo desta atividade. A partir daí, basta fazer a integração do Lexer gerado pelo ANTLR com o código do exercício-programa para que o mesmo seja capaz de realizar as tarefas envolvidas no processo da análise léxica do código de algum código-fonte inserido pelo usuário. Na Figura 8 tem-se o exemplo da instanciação do lexer gerado pelo ANTLR, dando como parâmetro de entrada o caminho de um arquivo que contenha o código-fonte a ser analisado, enquanto na Figura 9 pode-se ver o programa principal solicitando o próximo token ao lexer, na linha 86, e extraindo as suas informações, onde esse processo se repete até que um token EOF, representando o fim do arquivo, seja lido.

```
58  /* Instantiates the ANTLR4 generated lexer. */
59  myLexer = new MyGrammar(CharStreams.fromFileName(dir+filename));
60  vocabulary = myLexer.getVocabulary();
```

Figura 8 - Instanciação do Lexer no código do programa principal.

```
85      /* Get the next token. */
86      token = myLexer.nextToken();
87      tokenTypeName = vocabulary.getSymbolicName(token.getType());
88      presentLine = token.getLine();
89
90      /* EOF? */
91      if (token.getType() == Token.EOF)
92          break;
```

Figura 9 - Processo de leitura de um token e seus atributos.