

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
DCC045 - Teoria dos compiladores

Analizador Léxico

Grupo

André Dias Nunes — MAT: 201435031

Guilherme Barbosa — MAT: 201435031

Leonardo Reis

Relatório do primeiro trabalho da
disciplina DCC045 — Teoria dos
Compiladores. Desenvolvimento de
um Analisador Léxico.

Juiz de Fora

Maio de 2022

Sumário

1	Introdução	2
2	Especificações	3
2.1	Classes	3
2.2	JFlex	4
2.3	Tabela de Símbolos	6
3	Testes	7
4	Instruções	7

1 Introdução

O presente Relatório Técnico tem como objetivo central descrever o desenvolvimento da primeira etapa da criação de um compilador para uma linguagem **lang**. Esta etapa consiste no desenvolvimento de um analisador léxico.

Um analisador léxico tem como função transformar um input de uma sequência de caracteres em uma sequência de símbolos léxicos denominados como tokens e descartando caracteres não relevantes como espaços em brancos e comentários no processo.

Os tokens constituem classes de símbolos e podem ser representados, internamente, através do próprio símbolo ou por um par ordenado, no qual o primeiro elemento indica a classe do símbolo, e o segundo, um lexema (uma expressão regular). Eles são enviados ao analisador sintático para a verificação se o token pertence a linguagem.

2 Especificações

O programa deve receber um arquivo de texto contendo o código a ser analisado. O analisador fará a varredura caractere a caractere e se aprovado, irá imprimir a sequência dos lexemas, se não irá informar o caractere inválido.

Para realizar a varredura, foi utilizado a ferramenta **JFlex** para gerar o nosso analisador léxico. O **JFlex** recebe como entrada uma especificação com um conjunto de expressões regulares e ações correspondentes. Ele gera um programa (um lexer) que lê a entrada, compara a entrada com as expressões regulares no arquivo de especificação e executa a ação correspondente se for uma expressão regular corresponder.

2.1 Classes

- **Main**: Na main é realizado a abertura do arquivo e enviado ao analisador léxico. Em seguida se inicia o ciclo de obtenção dos tokens através da função `nextToken()` do analisador léxico. Então token a token é impresso no console até que se encerrem ou um token inválido seja encontrado.
- **TokenType**: Classe em que é definido todos os **TokenTypes** da linguagem que serão vinculados as expressões regulares.
- **Lexical**: Esta classe é gerada diretamente pelo **JFlex** através da especificação definida no arquivo `GenerateLexical.jflex`. Nela são definidos as expressões regulares e suas respectivas ações. Entraremos em mais detalhes no tópico 2.2.
- **Token**: Classe onde definimos o token e setamos seus dados. Há dois modos de criar o token, dependendo do tipo de entrada que é recebido, se recebe uma string salvamos seu lexema, se recebe um `Object` salvamos seu valor. Por fim definimos uma função de retorno para que seja exibido seus dados no console de acordo com o seu tipo. Também entraremos em mais detalhes no tópico 2.2.

2.2 JFlex

Neste segmento, demonstraremos como definimos o `GenerateLexical.jflex`.

```
/* Expressões regulares */
empty = {endLine} | [ \t\f]
endLine = \r\n|\r\n
identificador = [:letter:] + ([:letter:] | [:digit:] | "_" ) *
inteiro = [:digit:]+
decimal = [:digit:]* + "." + [:digit:]+
caracter = "\' + [:letter:]* + \" | \" [^\\] \" | \"\\\\n\" | \"\\\\t\" | \"\\\\b\" | \"\\\\r\" | \"\\\\\\\\\" | \"\\\\\\\\\\\\\"
```

Figura 1: Expressões Regulares

Na figura 1 representamos como definimos os macros para reconhecimento das expressões regulares que serão aceitas na nossa linguagem. O padrão utilizado é reconhecido internamente pelo pela classe gerada para realizar a análise.

```
<YYINITIAL>
{
    "Int" {return symbol(TokenType.IDINT); }
    "Float" {return symbol(TokenType.IDFLOAT); }
    "Char" {return symbol(TokenType.IDCHAR); }
    "Bool" {return symbol(TokenType.BOOL); }
    "true" {return symbol(TokenType.TRUE); }
    "false" {return symbol(TokenType.FALSE); }
    "null" {return symbol(TokenType.NULL); }
    "(" {return symbol(TokenType.LEFTPARENT); }
    ")" {return symbol(TokenType.RIGHTPARENT); }
    "[" {return symbol(TokenType.LEFTBRACKET); }
    "]" {return symbol(TokenType.RIGHTBRACKET); }
    "{" {return symbol(TokenType.LEFTCURLY); }
    "}" {return symbol(TokenType.RIGHTCURLY); }
    ">" {return symbol(TokenType.GREATER); }
    "<" {return symbol(TokenType.LESS); }
    "." {return symbol(TokenType.DOT); }
    "," {return symbol(TokenType.COMMA); }
    ";" {return symbol(TokenType.SEMICOLON); }
    "=" {return symbol(TokenType.ASSIGN); }
    "==" {return symbol(TokenType.EQ); }
    "!=" {return symbol(TokenType.NEQ); }
    "+" {return symbol(TokenType.PLUS); }
    "-" {return symbol(TokenType.MINUS); }
    "*" {return symbol(TokenType.MULT); }
    "/" {return symbol(TokenType.DIV); }
    "%" {return symbol(TokenType.MOD); }
    "&&" {return symbol(TokenType.AND); }
    "!" {return symbol(TokenType.NOT); }
    "if" {return symbol(TokenType.IF); }
    "else" {return symbol(TokenType.ELSE); }
    "iterate" {return symbol(TokenType.ITERATE); }
    "read" {return symbol(TokenType.READ); }
    "print" {return symbol(TokenType.PRINT); }
    "return" {return symbol(TokenType.RETURN); }
    "new" {return symbol(TokenType.NEW); }
    {inteiro} {return symbol(TokenType.INT, Integer.parseInt(yytext())); }
    {decimal} {return symbol(TokenType.FLOAT, Float.parseFloat(yytext())); }
    {caracter} {return symbol(TokenType.CHAR, yytext().substring(1,yytext().length()-1)); }
    {identificador} { return symbol(TokenType.ID, yytext()); }
    "--" { yybegin(SINGLELINECOMMENT); }
    "{-}" { yybegin(MULTILINECOMMENT); }
    {empty} { /* Não faz nada */ }
}
```

Figura 2: Reconhecendo os TokenTypes

Na figura 2 definimos as relações das entradas com seus respectivos tokens. Caso a entrada seja um dos macros definidos anteriormente, definimos, além do TokenType, o informação que ele carrega. A função `yytext()` é uma função nativa do JFlex que obtém o lexema atual analisado. Caso o lexema for um de um comentário, apenas ignoramos e não o salvamos.

Optamos por salvar os lexemas "Int", "Float", "Char" como TokenType: "IDINT", "IDFLOAT", "IDCHAR" respectivamente, para quando for uma valor destes tipos de variáveis, podermos salvar seus valores com os TokenType "INT", "FLOAT", "CHAR". Valor recebido pelos seus respectivos macros "inteiro", "decimal", "caracter". Optamos por este caminho para facilitar a impressão dos resultados como exemplificaremos a seguir.

```
%{  
    private Token symbol (TokenType type) {  
        return new Token (type, yytext(), yyline+1, yycolumn+1);  
    }  
  
    private Token symbol (TokenType type, Object value) {  
        return new Token (type, value, yyline+1, yycolumn+1);  
    }  
%}
```

Figura 3: Token Analisado

Na figura 3 temos a função que irá retornar o token analisado. Caso o lexema seja reconhecido, é retornado o seu TokenType, o lexema, mas caso sejam dos tipos "Int", "Float", "Char", "Caracter" ou "ID", ao invés do lexema, retornamos a informação processada, para que seja feita a diferenciação e o resultado impresso seja "TokenType: Info". Por exemplo: "Int: 10", "Char: Exemplo".

2.3 Tabela de Símbolos

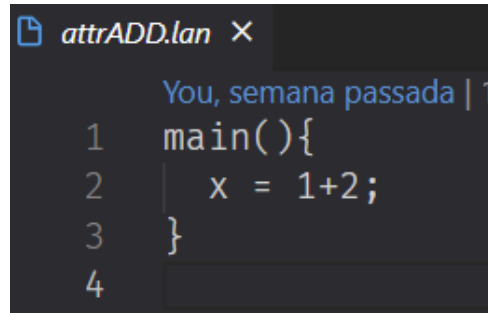
Nesta seção foram descritos os algoritmos desenvolvidos. Além de justificar a escolha da função critério utilizada e a estratégia de atualização da lista de candidatos.

Tag do Token	Lexema
ID	letter+(letter digit _)*
IDINT	Int
IDFLOAT	Float
IDCHAR	Char
INT	digit+
FLOAT	digit*+. ⁺ digit+
CHAR	letter* \\ \\ \\ n \\t \\b \\n
BOOL	Bool
TRUE	true
FALSE	false
NULL	null
LEFTPARENT	(
RIGHTPARENT)
LEFTBRACKET	[
RIGHTBRACKET]
LEFTCURLY	{
RIGHTCURLY	}
ITERATE	iterate
READ	read
PRINT	print

Tag do Token	Lexema
GREATER	>
LESS	<
SEMICOLON	;
DOT	.
COMMA	,
COLON	:
DOUBLECOLON	::
ASSIGN	=
EQ	==
NEQ	!=
PLUS	+
MINUS	-
MULT	*
DIV	/
MOD	%
AND	&&
NOT	!
IF	if
ELSE	else
RETURN	return
NEW	new

3 Testes

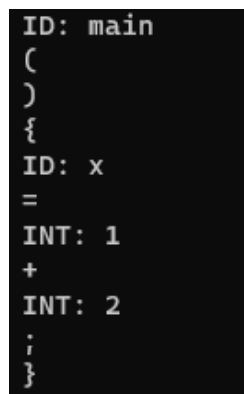
A figura 4 apresenta a entrada teste que Utilizaremos para exemplificar, o arquivo `testes/sintaxe/certo/attrADD.lan`



```
attrADD.lan X
You, semana passada | 1
1  main(){
2      x = 1+2;
3  }
4
```

Figura 4: attrADD.lan

Sua análise gera como resultado a Figura 5



```
ID: main
(
)
{
ID: x
=
INT: 1
+
INT: 2
;
}
```

Figura 5: Resultado

4 Instruções

Para compilar basta executar o comando: `make`

Para executar: `java Main arquivo.lan`

Referências

<https://www.jflex.de/>