GCC 130 - Analisador Léxico

Nome: *Arthur Henrique Sousa Cruz*
Matrícula: *201611484* ***Turma:*** *10A*

Nome: *Breno Gomes Cardoso*
Matrícula: *201611490* ***Turma:*** *10A*

Nome: *João Pedro Teodoro Silva*
Matrícula: *201610990* ***Turma:*** *10A*

Nome: *Pedro Silveira Lopes*
Matrícula: *201611481* ***Turma:*** *10A*

Prof. Rafael Serapilha Durelli
2018/01

Universidade Federal de Lavras
Departamento de Ciência da Computação

Conteúdo

1 Introdução

Este trabalho consiste em implementar um analisador léxico para a linguagem **j–**. O analisador léxico de uma linguagem tem como objetivo analisar se uma entrada, como um código, por exemplo, está de acordo com as regras impostas pela gramática daquela linguagem e produzir um fluxo de *tokens*.

Caso a entrada contenha algum erro léxico, o analisador deve indicar o erro, sua linha e sua coluna, o que facilita a manutenção de códigos.

2 Descrição do Trabalho e Estratégias de Solução

Para implementar o analisador léxico, foi criado um gerador de código "regex-to-code" que, a partir de uma expressão regular, consegue gerar um programa que aceita uma entrada caso ela esteja de acordo com aquela expressão e rejeita caso contrário. O programa gerado analisa de forma recursiva cada caractere de cada lexema, porém sem identificar o fim do lexema. Assim, caso fosse dado como entrada dois lexemas válidos separados por um espaço em branco, o programa não conseguiria identificá-los.

No entanto, para alcançar o objetivo do analisador, o gerador foi modificado para gerar códigos que conseguem separar os lexemas encontrados no arquivo e criar seus respectivos *tokens*. Assim, dando como entrada a expressão regular da linguagem j–, que define as palavras-chave, identificadores, operadores, separadores, etc., foi possível gerar um programa que cria o fluxo de *tokens*, o qual será utilizado pelo analisador sintático.

O programa lê um arquivo e separa suas linhas e, para cada linha, cria os lexemas identificados verificando o tipo de *token* que cada lexema pertence e, caso seja um identificador ou uma constante, o lexema é adicionado em uma tabela de símbolos. Caso o lexema não se enquadre a nenhum tipo de *token* é identificado como erro e reportado ao usuário.

Assim como o código do analisador, foi gerado também um AFD minimizado que determina as palavras aceitas pela linguagem, mostrado na Figura 1. Neste autômato é usado o termo *anything* no alfabeto para representar um caractere qualquer que esteja de acordo com a expressão regular $(ESC \mid \sim (\ \backslash \mid ' \mid LF \mid CR))$, se *anything* for precedido e sucedido por aspas simples, ou $(ESC \mid \sim (\ \backslash \mid " \mid LF \mid CR))$, se *anything* for precedido e sucedido por aspas duplas, onde ESC são os símbolos de escape \\, \", \', \n, \t, \r, \b, \f.

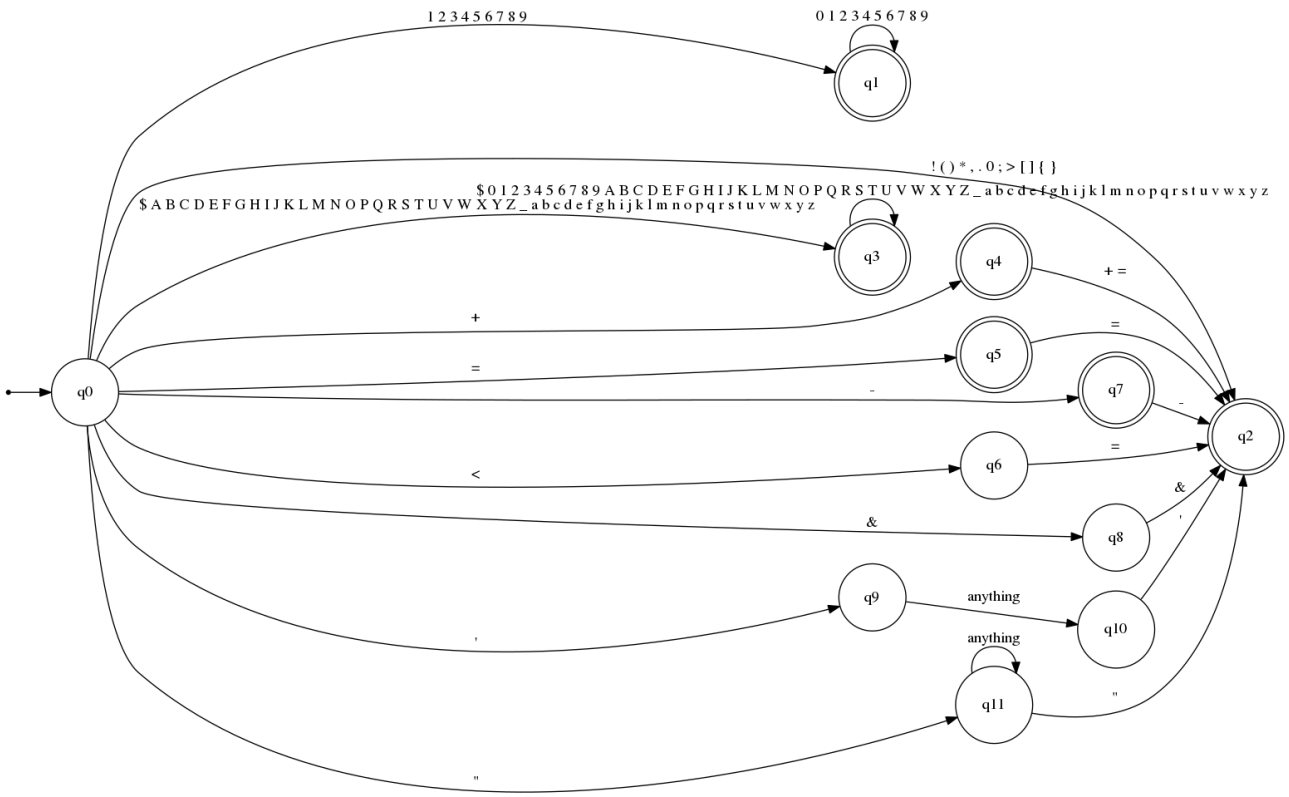


Figura 1: Grafo da linguagem J– gerada pelo *regex-to-code*

3 Testes e Resultados

Para verificar a corretude do analisador, foram criados dois testes: Um que gera todos os tipos de *tokens* e outro com 5 erros diferentes.

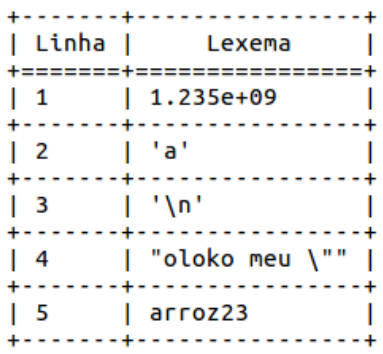
O teste com todos os tipos de *tokens* e sem erros é mostrado abaixo:

```
= == > ++ && <= ! - -- + += * //operadores

, . [] {} () ; //separadores

abstract extends int protected this boolean false //palavras chave
new public true char import null return void
class if package static while else
instanceof private super

1234567890 //int literal
'a' '\n' // char literal
"oloko meu \""

arroz23 //identificador
```

Cujos resultados foram os seguintes:

```
Lexema: =                       Linha:   1   Coluna:   1   Tipo do Token: OpAtribuicao
Lexema: ==                      Linha:   1   Coluna:   3   Tipo do Token: OpIgualdade
Lexema: >                       Linha:   1   Coluna:   6   Tipo do Token: OpMaior
Lexema: ++                      Linha:   1   Coluna:   8   Tipo do Token: OpIncremento
Lexema: &&                      Linha:   1   Coluna:  11   Tipo do Token: OpAnd
Lexema: <=                      Linha:   1   Coluna:  14   Tipo do Token: OpMenorIgual
Lexema: !                       Linha:   1   Coluna:  17   Tipo do Token: OpNot
Lexema: -                       Linha:   1   Coluna:  19   Tipo do Token: OpMenos
Lexema: --                      Linha:   1   Coluna:  21   Tipo do Token: OpDecremento
Lexema: +                       Linha:   1   Coluna:  24   Tipo do Token: OpSoma
Lexema: +=                      Linha:   1   Coluna:  26   Tipo do Token: OpSomaAtribuicao
Lexema: *                       Linha:   1   Coluna:  29   Tipo do Token: OpMultiplicacao
Lexema: ,                       Linha:   3   Coluna:   1   Tipo do Token: SepVirgula
Lexema: .                       Linha:   3   Coluna:   3   Tipo do Token: SepPonto
Lexema: [                       Linha:   3   Coluna:   5   Tipo do Token: SepAbreColchetes
Lexema: ]                       Linha:   3   Coluna:   6   Tipo do Token: SepFechaColchetes
Lexema: {                       Linha:   3   Coluna:   8   Tipo do Token: SepAbreChaves
Lexema: }                       Linha:   3   Coluna:   9   Tipo do Token: SepFechaChaves
Lexema: (                       Linha:   3   Coluna:  11   Tipo do Token: SepAbreParenteses
Lexema: )                       Linha:   3   Coluna:  12   Tipo do Token: SepFechaParenteses
Lexema: ;                       Linha:   3   Coluna:  14   Tipo do Token: SepPontoVirgula
Lexema: abstract                Linha:   5   Coluna:   1   Tipo do Token: PCAbstract
Lexema: extends                 Linha:   5   Coluna:  10   Tipo do Token: PCExtends
Lexema: int                     Linha:   5   Coluna:  18   Tipo do Token: PCInt
Lexema: protected               Linha:   5   Coluna:  22   Tipo do Token: PCProtected
Lexema: this                    Linha:   5   Coluna:  32   Tipo do Token: PCThis
Lexema: boolean                 Linha:   5   Coluna:  37   Tipo do Token: PCBoolean
Lexema: false                   Linha:   5   Coluna:  45   Tipo do Token: PCFalse
Lexema: new                     Linha:   6   Coluna:   1   Tipo do Token: PCNew
Lexema: public                  Linha:   6   Coluna:   5   Tipo do Token: PCPublic
Lexema: true                    Linha:   6   Coluna:  12   Tipo do Token: PCTrue
Lexema: char                    Linha:   6   Coluna:  17   Tipo do Token: PCChar
Lexema: import                  Linha:   6   Coluna:  22   Tipo do Token: PCImport
Lexema: null                    Linha:   6   Coluna:  29   Tipo do Token: PCNull
Lexema: return                  Linha:   6   Coluna:  34   Tipo do Token: PCReturn
Lexema: void                    Linha:   6   Coluna:  41   Tipo do Token: PCVoid
Lexema: class                   Linha:   7   Coluna:   1   Tipo do Token: PCClass
Lexema: if                      Linha:   7   Coluna:   7   Tipo do Token: PCIf
Lexema: package                 Linha:   7   Coluna:  10   Tipo do Token: PCPackage
Lexema: static                  Linha:   7   Coluna:  18   Tipo do Token: PCStatic
Lexema: while                   Linha:   7   Coluna:  25   Tipo do Token: PCWhile
Lexema: else                    Linha:   7   Coluna:  31   Tipo do Token: PCElse
Lexema: instanceof              Linha:   8   Coluna:   1   Tipo do Token: PCInstanceOf
Lexema: private                 Linha:   8   Coluna:  12   Tipo do Token: PCPrivate
Lexema: super                   Linha:   8   Coluna:  20   Tipo do Token: PCSuper
Lexema: 1234567890              Linha:  10   Coluna:   1   Tipo do Token: IntLiteral
Lexema: 'a'                     Linha:  11   Coluna:   1   Tipo do Token: CharLiteral
Lexema: '\n'                    Linha:  11   Coluna:   5   Tipo do Token: CharLiteral
Lexema: "oloko meu \""          Linha:  12   Coluna:   1   Tipo do Token: StringLiteral
Lexema: arroz23                 Linha:  14   Coluna:   1   Tipo do Token: Identificador
```

Figura 2: Resultado Teste 1



Linha	Lexema
1	1.235e+09
2	'a'
3	'\n'
4	"oloko meu \""
5	arroz23

Figura 3: Tabela de Símbolos do Teste 1

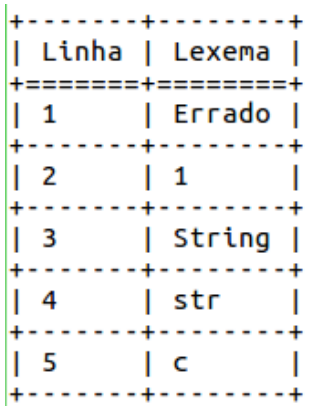
O teste com os erros léxicos é mostrado abaixo:

```
class Errado {
   int @45367 = 000;
   int 12psodk = 1;
   String str = "\";
   char c = ’aspodkapsodk’;
}
```

Cujos resultados foram os seguintes:

```
Lexema: class                   Linha:   2   Coluna:    1   Tipo do Token: PCClass
Lexema: Errado                  Linha:   2   Coluna:    7   Tipo do Token: Identificador
Lexema: {                       Linha:   2   Coluna:   14   Tipo do Token: SepAbreChaves
Lexema: int                     Linha:   3   Coluna:    2   Tipo do Token: PCInt
Lexema: =                       Linha:   3   Coluna:   13   Tipo do Token: OpAtribuicao
Lexema: ;                       Linha:   3   Coluna:   18   Tipo do Token: SepPontoVirgula
Lexema: int                     Linha:   4   Coluna:    2   Tipo do Token: PCInt
Lexema: =                       Linha:   4   Coluna:   14   Tipo do Token: OpAtribuicao
Lexema: 1                       Linha:   4   Coluna:   16   Tipo do Token: IntLiteral
Lexema: ;                       Linha:   4   Coluna:   17   Tipo do Token: SepPontoVirgula
Lexema: String                  Linha:   5   Coluna:    2   Tipo do Token: Identificador
Lexema: str                     Linha:   5   Coluna:    9   Tipo do Token: Identificador
Lexema: =                       Linha:   5   Coluna:   13   Tipo do Token: OpAtribuicao
Lexema: char                    Linha:   6   Coluna:    2   Tipo do Token: PCChar
Lexema: c                       Linha:   6   Coluna:    7   Tipo do Token: Identificador
Lexema: =                       Linha:   6   Coluna:    9   Tipo do Token: OpAtribuicao
Lexema: ;                       Linha:   6   Coluna:   24   Tipo do Token: SepPontoVirgula
Lexema: }                       Linha:   7   Coluna:    1   Tipo do Token: SepFechaChaves
Lexema: @45367                  Linha:   3   Coluna:    6   Erro: Sintaxe inválida
Lexema: 000                     Linha:   3   Coluna:   15   Erro: Sintaxe inválida
Lexema: 12psodk                 Linha:   4   Coluna:    6   Erro: Sintaxe inválida
Lexema: "\";                    Linha:   5   Coluna:   15   Erro: Sintaxe inválida
Lexema: 'aspodkapsodk'          Linha:   6   Coluna:   10   Erro: Sintaxe inválida
```

Figura 4: Resultado Teste 2



Linha	Lexema
1	Errado
2	1
3	String
4	str
5	c

Figura 5: Tabela de Símbolos do Teste 2

4 Conclusão

A implementação do analisador possibilitou uma melhor compreensão de como um compilador indica os erros léxicos presentes em um programa escrito em uma determinada linguagem e, também, como autômatos têm uma relação profunda com compiladores.

Além disso, foi possível perceber que construir um analisador léxico seria muito trabalhoso e repetitivo sem o "regex-to-code", que também facilitou a criação do fluxo de *tokens* e da tabela de símbolos, elementos importantes para as outras fases da compilação.

Referências

[1] Slides disponibilizados pelo professor no Campus Virtual

[2] AHO, Alfred V. et al. Compiladores: princípios, técnicas e ferramentas . 2. ed. São Paulo, SP: Pearson Addison Wesley, c2008. x, 634 p. ISBN 9788588639249.