Teoria da Computação
Trabalho Prático - Autômato Determinístico

Professor: Gustavo A. Fernandes
Data: 17 de Outubro de 2017

Análise Léxica

1. Descrição do trabalho

   O objetivo de um analisador léxico de um compilador é ler os caracteres do programa fonte, agrupá-los em lexemas, identificar o token correspondente ao lexema e gerar como saída uma sequência de tokens.

   - Um lexema é uma sequência de caracteres do programa fonte que casa com o padrão de algum token.
   - Um token é um par <`nome, valor`>:
     - nome: símbolo abstrato que representa um tipo de unidade léxica (por exemplo, identificador, palavra-chave, constante inteira, operador de atribuição, etc).
     - valor: o lexema representado pelo token.
   - Um padrão é uma especificação da forma que os lexemas de um token podem assumir.

   Por exemplo, vamos considerar:

   - as palavras chave `if` e `public`;
   - os operadores de comparação $\leq$ e $\geq$;
   - números inteiros;
   - strings;
   - identificadores construídos pelo seguinte padrão: "Letra seguida por letras e/ou dígitos".

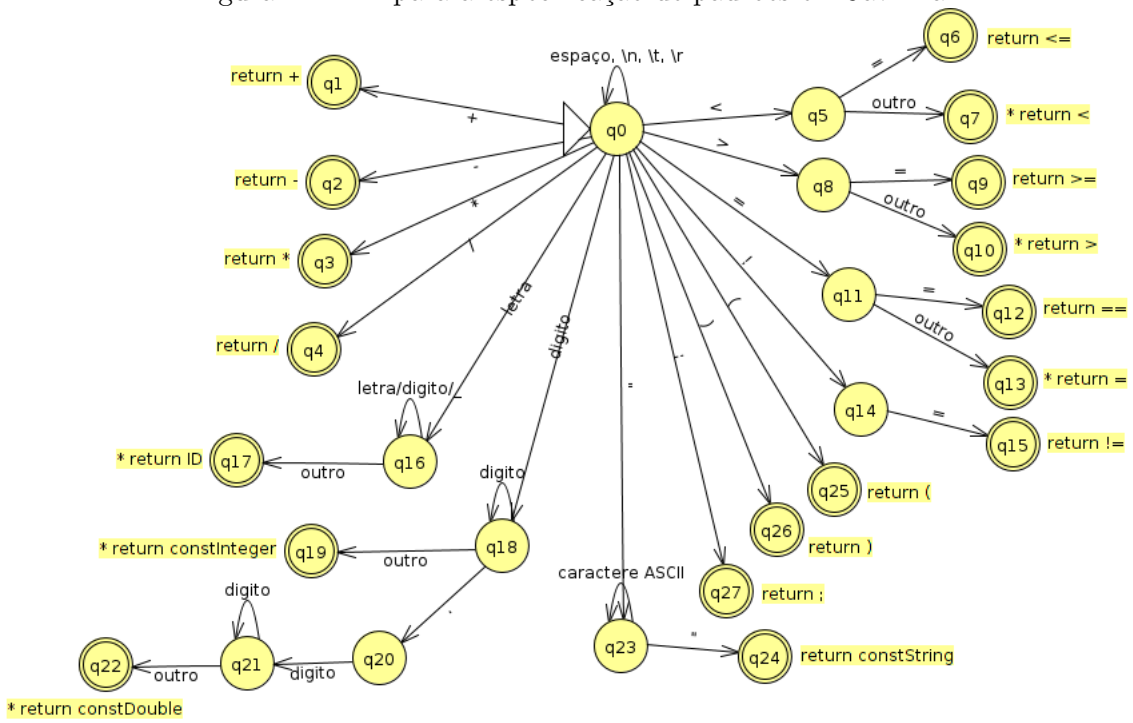
   Então, são exemplos de tokens:

   - <KW, if >
   - <KW, public >
   - <OP_MAIOR_IGUAL, $\geq$ >
   - <OP_MENOR_IGUAL, $\leq$ >
   - <NUM, 1234 >
   - <STRING, "Hello, World!">
   - <ID, minhaVariavel >

   Repare que cada token do exemplo anterior é representado pelo par <`nome, valor`>, conforme discutido anteriormente, ou seja, o token <KW, if > possui nome=KW e valor=if. Já o token, <ID, minhaVariavel > possui nome=ID e valor=minhaVariavel.

   A identificação dos tokens é feita por um Autômato Finito Determinístico (AFD). Você deverá implementar um AFD para o reconhecimento de tokens da linguagem **Javinha**. O AFD a ser implementado é apresentado na Figura 1. Já a linguagem Javinha é especificada pela gramática na última página, assim como um programa de exemplo gerado por essa gramática. O seu AFD deve reconhecer sentenças geradas por essa gramática, que representam programas em Javinha.

Figura 1: AFD para a especificação de padrões em Javinha



No AFD da Figura 1, quando temos `return +`, significa que devemos retornar um token com um nome para o operador "+" assim como o lexema, ou seja, a palavra "+". Como alcançamos um estado final, podemos reiniciar o AFD para o esstado inicial q_0. Já, se tivermos `* return ID`, devemos retornar um token com o nome para um identificador assim como a palavra reconhecida e, também, devemos retornar o ponteiro da leitura para lermos novamente o ("outro") símbolo visto anteriormente. Fica claro, que também devemos reiniciar o AFD para o estado q_0. Portanto preste atenção nesse AFD quando há o símbolo "*" antes do `return`.

Para facilitar a implementação, uma tabela de símbolos deverá ser usada. Essa tabela conterá, inicialmente, todas as palavras reservadas da linguagem. À medida que novos tokens do tipo "identificador" forem sendo reconhecidos, esses deverão ser consultados na tabela de símbolos antes de serem retornados. Um programa fonte, demonstrando como usar a tabela de símbolos, será disponibilizado pelo professor para guiar o desenvolvimento.

Além de reconhecer os tokens da linguagem, seu analisador léxico deverá detectar possíveis erros e reportá-los ao usuário. O programa deverá informar o erro e o local onde ocorreu (linha e coluna), lembrando que podemos ter 2 tipos de erros: caracteres desconhecidos e string não-fechada antes de quebra de linha ou fim de arquivo.

Espaços em branco, tabulações e quebras de linhas não são tokens, ou seja, devem ser descartados/ignorados pelo referido analisador.

Na gramática, os terminais de um lexema, bem como as palavras reservadas, estão entre aspas, ou seja, as aspas não fazem parte do terminal.

2. Descrição dos Padrões de Formatação

   Os padrões de formação das constantes e dos identificadores da linguagem são descritos abaixo:

   - ID: deve iniciar com uma letra seguida de 0 ou mais produções de letras e/ou dígitos e/ou "_" (undescore).
   - ConstInteger: cadeia numérica contendo 1 ou mais produções de dígitos.
   - ConstDouble cadeia numérica contendo 1 ou mais produções de dígitos, tendo em seguida um símbolo de ponto antes de 1 ou mais produções de dígitos.

- ConstString: deve iniciar e finalizar com o caractere "(aspas) contendo entre esses uma sequência de 0 ou mais produções de letras, dígitos e/ou símbolos.
- EOF é o código que representa fim de arquivo.

Os símbolos referidos na constante String (ConstString) indica qualquer caractere imprimível ASCII.

3. Cronograma e Valor

   O trabalho vale 15 pontos. Ele deverá ser entregue conforme consta na tabela abaixo.

Tabela 1: Cronograma

Data de entrega	**Valor**	**Multa por atraso**
29/11/2017	15 pontos	3pts / dia

4. O que entregar?

   Programa com todos os arquivos-fonte.

5. Regras:

   - O trabalho poderá ser realizado individualmente ou em dupla.
   - A implementação deverá ser realizada, somente, em uma das linguagens C, C++, Java, Python ou Ruby.
   - Se o programa não executar ou compilar, a nota será 0 (zero).
   - Trabalhos total ou parcialmente iguais receberão avaliação nula.
   - Ultrapassados 5 (cinco) dias, após a data definida para entrega, nenhum trabalho será recebido.

6. Pontuação Extra:

   Para aqueles que quiserem +3 pontos extras, incrementem seu analisador léxico para comentários no padrão Java. Ou seja, será permitido fazer comentário no padrão Java, (//) para comentário de uma linha ou (/* */) para comentários em várias linhas. Nesse caso o comentário não é um token e deverá ser ignorado. Atenção com comentários não fechado antes de fim de arquivo.

Gramática Javinha:

```
Programa        →   Classe EOF
Classe          →   "public" "class" ID ListaDeclaraVar ListaCmd "end"
ListaDeclaraVar →   TipoPrimitivo ID ";" ListaDeclaraVar | ε
TipoPrimitivo   →   "integer" | "double" | "string"
ListaCmd        →   CmdDispln ListaCmd | CmdAtrib ListaCmd | ε
CmdDispln       →   "SystemOutDispln" "(" Expressao ")" ";"
CmdAtrib        →   ID "=" Expressao ";"
Expressao       →   Expressao1 Expressao'
Expressao'      →   ">" Expressao1 Expressao' | "<" Expressao1 Expressao' |
                     ">=" Expressao1 Expressao' | "<=" Expressao1 Expressao' |
                     "==" Expressao1 Expressao' | "!=" Expressao1 Expressao' | ε
Expressao1      →   Expressao2 Expressao1'
Expressao1'     →   "+" Expressao2 Expressao1' | "-" Expressao2 Expressao1' | ε
Expressao2      →   Expressao3 Expressao2'
Expressao2'     →   "*" Expressao3 Expressao2' | "/" Expressao3 Expressao2' | ε
Expressao3      →   ConstNumInt | ConstNumDouble | ConstString | ID
```

Exemplo de programa em Javinha:

```
public class HelloJavinha
   integer data_entrega;
   string msgm;

   data_entrega = 29 * 11 * 17;
   msgm = "Caros alunos, O TP1 ja esta disponivel. Bom trabalho!";

   SystemOutDispln(msgm + " " + data_entrega);
end
```

Tokens reconhecidos desse programa:

```
Token: <KW, "public"> Linha: 1  Coluna: 7
Token: <KW, "class"> Linha: 1  Coluna: 13
Token: <ID, "HelloJavinha"> Linha: 1  Coluna: 26
Token: <KW, "integer"> Linha: 2  Coluna: 11
Token: <ID, "data_entrega"> Linha: 2  Coluna: 24
Token: <SMB_SEMICOLON, ";"> Linha: 2  Coluna: 25
Token: <KW, "string"> Linha: 3  Coluna: 10
Token: <ID, "msgm"> Linha: 3  Coluna: 15
Token: <SMB_SEMICOLON, ";"> Linha: 3  Coluna: 16
Token: <ID, "data_entrega"> Linha: 5  Coluna: 16
Token: <RELOP_ASSIGN, "="> Linha: 5  Coluna: 18
Token: <INTEGER, "29"> Linha: 5  Coluna: 21
Token: <RELOP_MULT, "*"> Linha: 5  Coluna: 23
Token: <INTEGER, "11"> Linha: 5  Coluna: 26
Token: <RELOP_MULT, "*"> Linha: 5  Coluna: 28
Token: <INTEGER, "17"> Linha: 5  Coluna: 31
Token: <SMB_SEMICOLON, ";"> Linha: 5  Coluna: 32
Token: <ID, "msgm"> Linha: 6  Coluna: 8
Token: <RELOP_ASSIGN, "="> Linha: 6  Coluna: 10
Token: <STRING, "Bom trabalho!"> Linha: 6  Coluna: 26
Token: <SMB_SEMICOLON, ";"> Linha: 6  Coluna: 27
```

```
Token: <KW, "SystemOutDispln"> Linha: 8  Coluna: 19
Token: <SMB_OP, "("> Linha: 8  Coluna: 20
Token: <STRING, "Caros alunos, O TP ja esta disponivel."> Linha: 8  Coluna: 60
Token: <SMB_CP, ")"> Linha: 8  Coluna: 61
Token: <SMB_SEMICOLON, ";"> Linha: 8  Coluna: 62
Token: <KW, "end"> Linha: 9  Coluna: 4
Token: <EOF, "EOF"> Linha: 10  Coluna: 1
```