Centro Federal de Educação Tecnológica de Minas Gerais – DECOM - Engenharia de Computação
Compiladores – Prof.ª Kecia Marques

Exercício 1

1) Cite as fases constituintes de um compilador, indicando seus respectivos objetivos. Escreva com as suas palavras. Não é necessário citar as fases que são opcionais.

- Para construir um compilador, precisamos saber qual é sua estrutura. O compilador tem duas funcionalidades principais: fazer análise e fazer síntese.

A fase de análise é dividida em três:

- Análise léxica: resume-se a ler o carácter do código fonte formando os lexemas para poder entregar os tokens para o analisador sintático.
- Análise sintática: resume se a reconhecer se a sequência de token é válida para a linguagem.
- Análise semântica: resume a validar as regras que a sintática não consegue ou até o momento não pode analisar.

A fase de síntese:

- Síntese: é o último processo de um compilador. Ela é responsável por gerar um código objeto.

2) No trecho de programa em C a seguir, indique a sequência de tokens que seria reconhecida pelo Analisador Léxico:

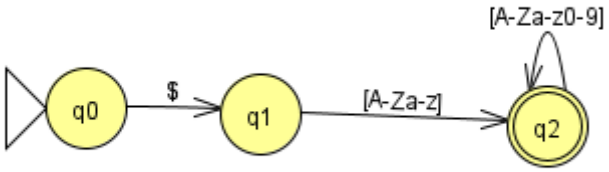
y = fatorial(n);

1. *<id,1>*
2. *<=,>*
3. *<id,2>*
4. *<(,>*
5. *<id,3>*
6. *<),>*
7. *<;,>*

3) As descrições abaixo definem o padrão de formação de tokens. Para cada uma delas, mostre uma expressão regular e/ou um AFD (autômato finito determinístico) correspondente.

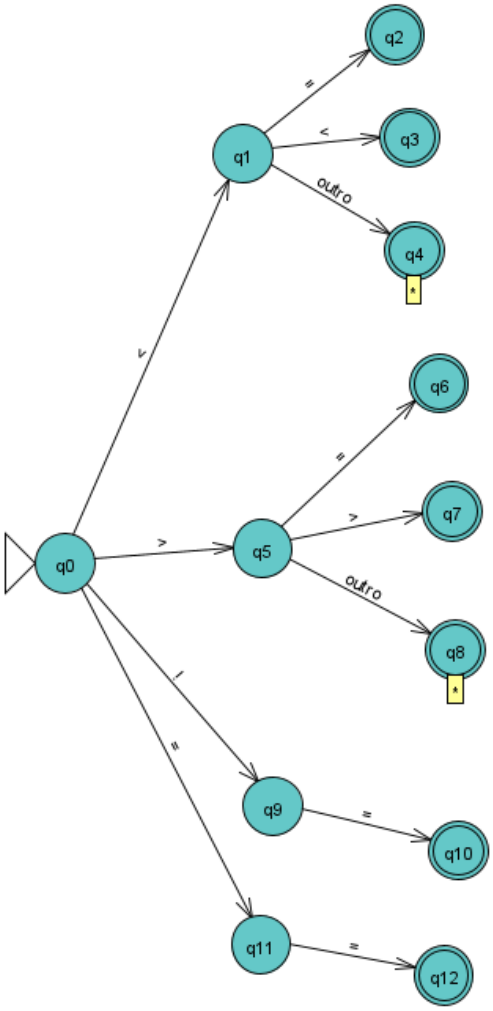
a) Identificadores devem iniciar com $, seguido de pelo menos uma letra, que pode vir seguida de uma sequência de letras e/ou dígitos.

- $[a-zA-Z]^+[a-zA-Z0-9]^*$



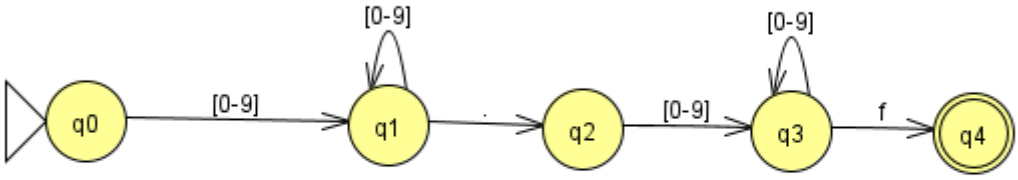
---

b) Operadores: <, <<, >, >>, >=, <=, != e ==. Neste item não é necessário mostrar a expressão regular.



c) Constantes do tipo float são sequências de pelo menos um dígito seguida por um ponto (‘.’) , seguido por pelo menos um dígito e terminadas com a letra 'f'.

- $[0\text{-}9]^{+}.[0\text{-}9]^{+}f$



4) Mostre uma implementação (em C/C++, Java ou pseudocódigo) do método *public Token scan ()* que reconheça as construções a seguir.

- Operadores ‘!, ‘!=’, ‘>’ e ‘>>’ · Considere a forma de implementação do Analisador Léxico do livro texto.

```java
public class operador{

 /*Lê o próximo caractere do arquivo*/
 private void readch() throws IOException{
    ch = (char) file.read();
    }

 /* Lê o próximo caractere do arquivo e verifica se é igual a c*/
 private boolean readch(char c) throws IOException{
 readch();
 if (ch != c) return false;
 ch = ' ';
 return true;
 }

 public Token scan() throws IOException{
     //Desconsidera delimitadores na entrada
     for (;; readch()) {
     if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\b') continue;
     else if (ch == '\n') line++; //conta linhas
     else break;
     }
     switch(ch){
     //Operadores
     case '!':
     if (readch('=')) return Word.ne;
     else return new Token('!');
     case '>':
     if (readch('>')) return Word.mgt;
     else return new Token('>');
        }

     //Caracteres não especificados
     Token t = new Token(ch);
     ch = ' ';
     return t;
    }
}
```

- Identificadores conforme descrito no item 3.a.

```java
public class identificador {

    /* Lê o próximo caractere do arquivo */
    private void readch() throws IOException {
        ch = (char) file.read();
    }

    public Token scan() throws IOException {
        // Desconsidera delimitadores na entrada
        for (;; readch()) {
            if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\b')
                continue;
            else if (ch == '\n')
                line++; // conta linhas
            else
                break;
        }

        // Identificadores
        if (Character.compare(ch, '$') == 0) {
            StringBuffer sb = new StringBuffer();
            sb.append(ch);
            readch();
            if (Character.isLetter(ch)) {
                do {
                    sb.append(ch);
                    readch();
                } while (Character.isLetterOrDigit(ch));
                String s = sb.toString();
                Word w = (Word) words.get(s);
                if (w != null)
                    return w; // palavra já existe na HashTable
                w = new Word(s, Tag.ID);
                words.put(s, w);
                return w;
            }else{
                return new Error("Token não existe")
            }
        }

        // Caracteres não especificados
        Token t = new Token(ch);
        ch = ' ';
        return t;
    }

}

```