Alunos: Danielle Dias Vieira & Henrique Augusto Rodrigues

3)
1. Palavra-chave bool: `bool`
2. Identificador ID: `[a-zA-Z_][a-zA-Z0-9_]*`
   - Esta expressão regular corresponde a um identificador que começa com uma letra ou sublinhado, seguido por zero ou mais letras, dígitos ou sublinhados.
3. Valor hexadecimal: `0[xX][0-9a-fA-F]+`
   - Esta expressão regular corresponde a um valor hexadecimal que começa com "0x" ou "0X", seguido por um ou mais dígitos hexadecimais (0-9, A-F, a-f).
4. Valores floats:
   - `{3.1416}`: `[-+]?[0-9]*\.?[0-9]+`
   - `(-3e4)`: `[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?`
   - `{+1.0e-5}`: `[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?`
   - `(.567e+8)`: `[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?`
5. Valores em reais:
   - `{R$**2.345.67}`: `R\$[*]*[0-9]{1,3}(\.[0-9]{3})*(\,[0-9]+)?`
   - `{R$12.452.183.16}`: `R\$[*]*[0-9]{1,3}(\.[0-9]{3})*(\,[0-9]+)?`
   - `{R$****12}`: `R\$[*]*[0-9]{1,3}(\,[0-9]+)?`
   - `(RS**0)`: `R\$[*]*[0-9]{1,3}(\,[0-9]+)?`

4)
Estados:
1. q_0 - estado inicial
2. q_1 - estado final (aceitação)
3. q_2 - estado de rejeição

Alfabeto:
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F\}$

Transições:
1. q_0 --0-F--> q_1
2. q_0 --a-f, A-F--> q_1
3. q_1 --0-F--> q_1
4. q_1 --a-f, A-F--> q_1

Explicação:
- Começamos em q_0 e, se ler um dígito hexadecimal ou uma letra de 'a' a 'f' (maiúscula ou minúscula), vamos para q_1, onde podemos continuar a ler dígitos hexadecimais ou letras.
- q_1 é um estado final, o que significa que aceitamos a sequência se terminar nesse estado.
- Se encontrarmos qualquer outro caractere que não seja um dígito hexadecimal ou uma letra de 'a' a 'f', vamos para q_2, um estado de rejeição, onde rejeitamos a sequência.

Esse autômato aceitará qualquer constante hexadecimal e rejeitará todas as outras strings.

5)

```
<INT, PALAVRA-CHAVE>
<IDENTIFICADOR, updateScore>
<PARENTESES_ESQ, DELIMITADOR>
<IDENTIFICADOR, score>
<PARENTESES_DIR, DELIMITADOR>
<CHAVES_ESQ, DELIMITADOR>
<SE, PALAVRA-CHAVE>
<PARENTESES_ESQ, DELIMITADOR>
<IDENTIFICADOR, score>
<OP_MENOR, OPERADOR_RELACIONAL>
<INT_CONST, CONSTANTE_INTEIRA>
<PARENTESES_DIR, DELIMITADOR>
<CHAVES_DIR, DELIMITADOR>
<WHILE, PALAVRA-CHAVE>
<PARENTESES_ESQ, DELIMITADOR>
<IDENTIFICADOR, score>
<MOD, OPERADOR_ARITMETICO>
<IDENTIFICADOR, n>
<INT_CONST, CONSTANTE_INTEIRA>
<PARENTESES_DIR, DELIMITADOR>
<CHAVES_ESQ, DELIMITADOR>
<IDENTIFICADOR, score>
<INCREMENTO, OPERADOR_ARITMETICO>
<IDENTIFICADOR, score>
<IDENTIFICADOR, updateUppperBound>
<PARENTESES_ESQ, DELIMITADOR>
<IDENTIFICADOR, score>
<INCREMENTO, OPERADOR_ARITMETICO>
<REAL_CONST, CONSTANTE_REAL>
<PARENTESES_DIR, DELIMITADOR>
<OP_RETURN, PALAVRA-CHAVE>
<INT_CONST, CONSTANTE_INTEIRA>
<PONTO_VIRGULA, DELIMITADOR>
<OP_RETURN, PALAVRA-CHAVE>
<INT_CONST, CONSTANTE_INTEIRA>
<PONTO_VIRGULA, DELIMITADOR>
```

Classes de tokens utilizadas:
- PALAVRA-CHAVE: int, if, while, return
- IDENTIFICADOR: updateScore, score, n, updateUppperBound

- DELIMITADOR: (,), {, }, ;
- OPERADOR_RELACIONAL: <
- CONSTANTE_INTEIRA: 100, 8, 1, 0
- OPERADOR_ARITMETICO: %, ++
- CONSTANTE_REAL: 3.5

6)
A linguagem {(*)' (i >= 0)} representa todas as strings que começam com zero ou mais asteriscos seguidos por zero ou mais letras 'i'. Esta linguagem não pode ser reconhecida por um autômato finito, porque exige uma contagem arbitrariamente grande de símbolos 'i', o que não pode ser mantido em um número finito de estados.

A razão é que um autômato finito tem um número finito de estados e, portanto, só pode lembrar um número finito de ocorrências de um símbolo 'i'. Se a linguagem permitisse um número finito de ocorrências de 'i', então poderíamos construir um autômato finito. No entanto, a linguagem permite um número arbitrariamente grande de ocorrências de 'i', o que requer um número infinito de estados para lembrar.

Portanto, não é possível desenhar um autômato finito para reconhecer todas as strings pertencentes à linguagem {(*)' (i >= 0)}.

7)
1. No autômato descrito anteriormente, uma string que seria reconhecida, mas não configuraria uma estrutura de repetição válida, poderia ser "aaabb". Essa string seria reconhecida pelo autômato, pois começa com "aaa", contém "bb", mas não configura uma estrutura de repetição válida como as listadas no exemplo.

2. O analisador léxico da forma como estudamos (utilizando expressões regulares convertidas em um Autômato Finito Determinístico - DFA) por si só não é capaz de reconhecer programas válidos porque ele lida apenas com a análise de tokens individuais, sem considerar a estrutura sintática do programa como um todo. Enquanto o analisador léxico pode identificar padrões léxicos individuais (como identificadores, palavras-chave, números, etc.), ele não tem conhecimento da gramática da linguagem de programação para verificar se esses tokens estão dispostos de acordo com a sintaxe correta da linguagem.

Por exemplo, o analisador léxico pode identificar corretamente palavras-chave, identificadores e números em um programa, mas não seria capaz de determinar se a ordem e o uso desses tokens estão corretos de acordo com a sintaxe da linguagem de programação. Isso requer uma análise sintática mais avançada, que envolve a construção de uma árvore sintática ou a utilização de uma gramática formal para verificar a estrutura gramatical do programa.