INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

Trabalho 02

Marco Furlan

Maio/2021

Instruções

- Esta **atividade** deverá ser **resolvida** em **equipe** – a mesma formada em sala de aula;
- **Responder** todas as **questões na ordem apresentada** em um arquivo Word ou LaTeX e depois **exportar** para **PDF**;
- **Identificar** claramente no **início do documento** o **nome** e **RA** dos integrantes da equipes;
- **Enviar o PDF** criado para o **link do trabalho** no **MyOpenLMS da disciplina**;
- Apenas um integrante da equipe precisa enviar;
- Esta atividade estará **disponível** a partir de **31/05/2021**;
- A solução deverá ser **enviada** até o dia **10/06/2021**.

(1) **Expressões regulares**

Java possui suporte a expressões regulares de duas formas:

- Para expressões simples, na classe `String`, com os métodos `matches()`, `split()`, `replaceFirst()` e `replaceAll()`;
- Com classes específicas de `java.util.regex`, `Pattern` e `Matcher` e seus métodos apropriados.

Estudar a lição sobre expressões regulares com Java de <https://docs.oracle.com/javase/tutorial/essential/regex/>. Depois, resolver o seguinte problema: elaborar um programa Java contendo apenas uma janela principal um rótulo, uma caixa de texto e um botão. O objetivo é testar a **força** de senhas digitadas pelo usuário. Assim, ao clicar um dos botões, o programa deverá obter o texto digitado e testá-lo com uma **expressão regular** adequada, escrevendo em um rótulo de resultado o valor "APROVADO" ou "REJEITADO", de acordo com os requisitos a seguir:

(a) O comprimento da senha deve ser maior ou igual à 8;

(b) A senha deve conter um ou mais caracteres maiúsculos;

(c) A senha deve conter um ou mais caracteres minúsculos;

(d) A senha deve conter um ou mais dígitos;

(e) A senha deve conter um ou mais caracteres especiais (`!@#$&*`).

(2) **Máquinas de estados finitos**

A tarefa de converter uma cadeia de caracteres contendo um número em um número propriamente dito pode ser vista como a tarefa de construir um pequeno interpretador de expressões. Um método simples (mas braçal) de definir tal interpretador é aquele que emprega **autômatos finitos**. A ideia é construir uma máquina de estado cujas **transições** sejam provocadas por **caracteres** lidos **da entrada** (teclado, arquivo). Dependendo do **estado atual e** do **caractere lido**, a **transição** pode **levar** a um **novo estado**, **válido**, de **erro** ou de **aceitação**, sendo este último o sinalizador que a expressão foi **aceita** (e no caso da string, convertida).

Um **diagrama de estados** que analisa se uma cadeia **contém ou não um número inteiro** está apresentado na Figura 1.

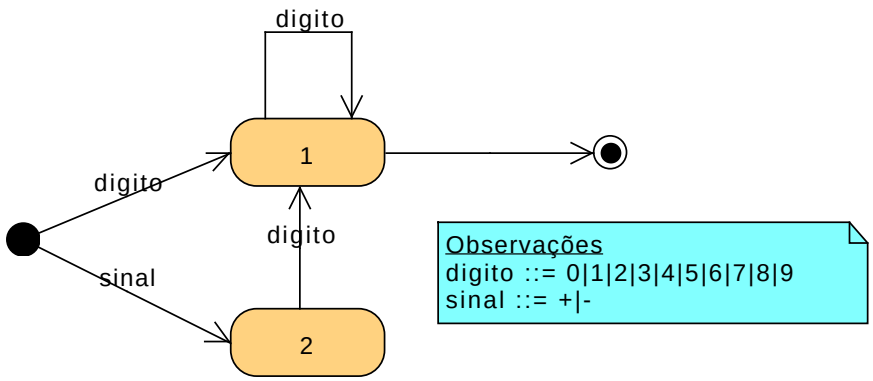


Figura 1: Máquina de estados finitos para reconhecer um número inteiro.

A notação utilizada aqui é a UML (*Unified Modeling Language*), um padrão muito utilizado atualmente. A bolinha preta representa um pseudo-estado de partida e a bolinha tipo "alvo" representa um pseudo-estado final, o estado de aceitação.

Então, partindo do pseudo-estado inicial, se for lido um dígito (`1`, `2`, …, `9`), faz-se a transição para o estado 1. O estado 1 já é um estado final, pois dele há uma transição automática para o pseudo-estado final (isto é, não é necessário ler mais nada para ir para o final – transição vazia ou com símbolo ϵ).

Assim, a cadeia `1234` é um número inteiro válido pois todos os caracteres lidos levam ao estado 1, que indica sucesso no reconhecimento. Já a cadeia `12A` é válida até a entrada do símbolo `A` (não há uma transição rotulada por letra que sai do estado 1 para qualquer outro) e a cadeia `NCC1701` não é um inteiro válido (não há uma transição rotulada por letra que sai do pseudo-estado inicial para qualquer outro). Convenciona-se aqui que, quando não há uma transição de um certo símbolo a partir de um estado qualquer, transita-se imediatamente para um **estado implícito de erro**. Por fim, números inteiros com sinal também são reconhecidos: o sinal, se existir deve estar na frente do número e, a partir daí reconhece-se o número como descrito anteriormente.

A maneira mais simples de traduzir um diagrama de estados em uma linguagem de programação qualquer é por meio de uma tabela de transição de estados, facilmente implementada por uma matriz bidimensional:

		Símbolo		
		`digito (0)`	`sinal (1)`	`outro (2)`
Estado	`0`	1	2	3
	`1`	1	3	3
	`2`	1	3	3
	`3`	3	3	3

Nesta tabela, os números de linhas representam estados atuais (o estado inicial é o `0`) e as colunas representam símbolos identificados no estado atual. As células internas indicam o próximo estado que o sistema estará dado o estado atual e o símbolo lido. Assim, a posição `[2,0]` indica que se estiver no estado 2 e ler um símbolo de `dígito` (`0`), então altere do estado atual para o estado 1, por exemplo.

O programa Java a seguir utiliza o conceito apresentado para converter strings em números inteiros:

```
//Recognizer.java
package recognizer;

public class Recognizer {
    // Relevant states
    public static final int INITIAL = 0;
    public static final int FINAL = 1;
    public static final int ERROR = 3;
    // Symbol constants
    public static final int DIGIT = 0;
    public static final int SIGNAL = 1;
    public static final int OTHER = 2;
    private int[][] table = { { 1, 2, 3 }, { 1, 3, 3 }, { 1, 3, 3 },
            { 3, 3, 3 } };
    private int currentState;

    // Constructor (empty)
```

```
    public Recognizer() {
    }

    // Recognize (or not) if the string is a number
    // returns the number or throws an NumberFormatException, otherwise
    public int recognize(String s) throws NumberFormatException {
        int pos = 0;
        int number = 0;
        int signal = 1;
        currentState = INITIAL;
        while (pos < s.length()) {
            char c = s.charAt(pos);
            int symbol = getSymbol(c);
            currentState = table[currentState][symbol];
            if (currentState == ERROR) {
                throw new NumberFormatException(s + " is not a number!");
            }
            if (symbol == DIGIT) {
                number = number * 10 + c - '0';
            }
            if (symbol == SIGNAL) {
                //Variable c carries the signal symbol
                signal = signal * (c == '+' ? 1 : -1);
            }
            pos++;
        }
        if (currentState != FINAL) {
            throw new NumberFormatException(s + " is not a number!");
        }
        number = signal * number;
        return number;
    }

    private int getSymbol(char c) {
        if (Character.isDigit(c)) {
            return DIGIT;
        } else if (c == '-' || c == '+') {
            return SIGNAL;
        } else {
            return OTHER;
        }
    }
}
```

```
//Main.java
package recognizer;

public class Main {

    public static void main(String[] args) {
        Recognizer rec = new Recognizer();
        try {
            int i = rec.recognize("123");
            System.out.println(i);
            i = rec.recognize("-123");
            System.out.println(i);
            //Uncomment the lines below to see its effect...
            //i = rec.recognize("123A");
            //i = rec.recognize("NCC1701");
```

```
            } catch (NumberFormatException ex){
                System.out.println(ex.getMessage());
            }
        }
    }
```

Baseando-se no exemplo apresentado, pede-se: **elaborar em Java** um **reconhecedor** para **números reais**. Um número real pode ser:

- Uma sequência de dígitos com ou sem sinal: 1212, +345, -4213 etc;
- Um número com ou sem sinal com ponto decimal: +3., -3.0, 3.456, 21.44 etc;
- Um número com ou sem sinal com ponto decimal na notação de engenharia: 3E5, 4e-19, -2.03E-2 etc.

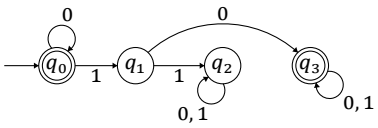
(3) **Implementar um interpretador de autômatos determinísticos em Python**. O programa deverá **ler a descrição** de um **autômato** finito determinístico **presente** em um **arquivo-texto** e então **aguardar** (até que o usuário termine o programa) a **entrada** de **cadeias** que serão **aceitas** ou **rejeitadas** pelo autômato.

Assim que uma **cadeia** for **entrada** no programa, o **programa** deverá **simular** o **funcionamento** do autômato com **esta cadeia**, **apresentando** na **tela** os **pares** (q, c) a partir de um estado inicial, onde q é um estado e c é um símbolo de entrada.

Deverá **apresentar**, no **final**, se esta cadeia foi **aceita** ou **rejeitada** e, neste caso, imprimir o **motivo**: estado de não aceitação com a cadeia vazia ou a cadeia não está vazia mas não conseguiu aplicar uma transição.

Pode-se utilizar um arquivo-texto contendo a estrutura da máquina descrita em Python. **Python permite interpretar, durante a execução, códigos Python com o auxílio da função** `eval()`.

Por exemplo, considerar a máquina M representada a seguir:



Ele aceita a linguagem $L(M) = \{0^n, 0^n10x\}$, onde x é qualquer cadeia contendo 0s e 1s. Esse **autômato** pode ser assim **codificado**, em um **arquivo-texto**, por exemplo, `m.dfa`, que contém estruturas válidas em **Python**:

```
{
    'states' : set([0,1,2,3]),
    'initial_state' : 0,
    'sigma' : set(['0','1']),
    'delta' : { (0,'0'):0, (0,'1'):1,
                (1,'0'):3, (1,'1'):2,
                (2,'0'):2, (2,'1'):2,
                (3,'0'):3, (3,'1'):3,
              },
    'final_states' : set([0, 3])
}
```

Notar que esta codificação está bem próxima dos elementos de um autômato finito determinístico: $M = (Q, \Sigma, \delta, q_0, F)$. Neste arquivo:

- A **descrição do autômato é um dicionário Python** (`dict`) cujas **chaves são os elementos** dele;
- O **conjunto de estados** e o **conjunto de estados finais** e o **alfabeto** são **representados** por um **conjunto** (`set`) de **Python**. Um `set` é uma **lista** onde a **ordem não importa** e onde **não** se **tem duplicatas**. Seguem algumas **operações** que se pode utilizar quando se tem um `set` **Python** (onde `x`, é um elemento do conjunto e `R` e `S` são `sets` quaisquer):
  - `x in S`: o mesmo que $x \in S$;
  - `x not in S`: o mesmo que $x \notin S$;
  - `R < S`: o mesmo que $R \subset S$;
  - `R <= S`: o mesmo que $R \subseteq S$.
- A **função de transição** é **representada** por **outro dicionário**: neste dicionário, a **chave** é um **par** (q, c) onde q é o **estado** e c é um **símbolo** da **entrada** e que **mapeia** para um outro **estado**;
- Notar que foram usados **números inteiros** para **estados** e **caracteres** (cadeias) para **símbolos** da **entrada**.

Como interpretar este arquivo? Primeiramente é necessário **ler** este **arquivo**. Felizmente, esta tarefa é muito **simples** em **Python**. A **função** `open()` **abre** um **arquivo** e atribui a ele um **objeto** de **arquivo** com a construção a seguir:

```
with open('m.dfa') as dfa_file:
    dfa_data = dfa_file.read()
```

A função `read()` **lê todo arquivo** e **armazena** seu **conteúdo** como uma **cadeia de caracteres na variável** `data` do exemplo. Por fim, **como transformar uma cadeia de caracteres contendo código Python** em **elementos** de um **programa** que podem ser manipulados? Isso pode ser feito com a **função** `eval()`, assim:

```
dfa = eval(dfa_data)
# Para conferir o conteúdo
print(dfa)
```

Daqui para frente, com esses elementos, é possível realizar a interpretação do autômato a partir de uma entrada qualquer.

O **algoritmo** é bem **simples** e está descrito a seguir em **pseudocódigo**:

```
function simular_dfa(dfa, entrada)
begin
    estado = obter o estado inicial do autômato
    aceitar = falso
    while comprimento(entrada) > 0
    begin
```

```
            c = remover o símbolo mais à esquerda da entrada
            if c não está no alfabeto then
            begin
                erro('O símbolo', c, 'não pertence ao alfabeto do autômato!')
                recolocar c no início da entrada
                break
            end
            if estado não está no conjunto de estados then
            begin
                erro('O estado', estado,
                    'não pertence ao conjunto de estados do autômato!')
                break
            end
            estado = obter o próximo estado a partir de estado e c
            if não for possível realizar a transição then
            begin
                erro('Não foi possível realizar a transição do estado',
                    estado, 'com entrada', c)
                break
            end
        end
        if estado estiver no conjunto de estados finais e a entrada estiver vazia then
        begin
            aceitar = verdadeiro
        end
        if aceitar for verdadeiro then
        begin
            exibir('A cadeia', entrada, 'foi aceita pelo autômato!')
        end
        else
        begin
            exibir('A cadeia', entrada, 'foi rejeitada pelo autômato!')
        end
    end
```

Lista de **funcionalidades** que o **programa** deverá apresentar:

(1) **Ler a especificação** de um autômato finito determinístico **de** um **arquivo-texto**;

(2) **Apresentar** os **estados transitados** durante uma simulação;

(3) **Sinalizar**:

   a. **Quando** uma **cadeia** é **reconhecida** com sucesso;

   b. **Erro quando** um **estado** selecionado **não está** no **conjunto** de **estados**;

   c. **Erro** quando um **símbolo** lido da **entrada não está** no **alfabeto**;

   d. **Erro** quando **não** for **possível** aplicar uma **transição**..

(4) O **programa** deve **interagir** com o **usuário** – ele poderá testar quantas cadeias quiser até interromper;

(5) A **forma** como se **entrará** o **nome** do **arquivo** do **autômato** no programa é **livre** (pode usar `input()` ou argumentos de linha de comando).

Como **exemplo de uso**, seguem alguns **testes positivos**:

```
marco@JUPITER:~$ python3 /home/marco/Projects/Python/DFASimulator/dfasim.py m3.dfa
Digite a cadeia:
A cadeia  foi aceita pelo autômato!
```

```
Digite a cadeia: 0000
(0, '0') -> 0
(0, '0') -> 0
(0, '0') -> 0
(0, '0') -> 0
A cadeia 0000 foi aceita pelo autômato!
Digite a cadeia: 00010101010
(0, '0') -> 0
(0, '0') -> 0
(0, '0') -> 0
(0, '1') -> 1
(1, '0') -> 3
(3, '1') -> 3
(3, '0') -> 3
(3, '1') -> 3
(3, '0') -> 3
(3, '1') -> 3
(3, '0') -> 3
A cadeia 00010101010 foi aceita pelo autômato!
Digite a cadeia: 10
(0, '1') -> 1
(1, '0') -> 3
A cadeia 10 foi aceita pelo autômato!
Digite a cadeia: ^D
Programa finalizado pelo usuário!
```

E alguns **testes negativos**:

```
marco@JUPITER:~$ python3 /home/marco/Projects/Python/DFASimulator/dfasim.py m3.dfa
Digite a cadeia: 1110
(0, '1') -> 1
(1, '1') -> 2
(2, '1') -> 2
(2, '0') -> 2
A cadeia 1110 foi rejeitada pelo autômato!
Digite a cadeia: 10a1
(0, '1') -> 1
(1, '0') -> 3
O símbolo a não pertence ao alfabeto do autômato!
A cadeia 10a1 foi rejeitada pelo autômato!
Digite a cadeia: ^D
Programa finalizado pelo usuário!
```

Dicas: :

- Para **ler a entrada** que está em uma **cadeia de caracteres**, **transforme** a cadeia entrada **em** uma **lista** de caracteres assim: `s = list(s)`. Desse modo, será possível **utilizar** a **função** `pop()` para **extrair** (e reduzir a cadeia) **assim**: `c = s.pop(0)`;
- Se **quiser ler** o nome do arquivo pela **linha** de **comando**: **importar** de `sys` a **variável** `args`, que contém a **lista** de **argumentos** do script (na posição 0 é o nome do script, na posição 1 tem o primeiro argumento etc.);

- Se quiser **interromper** "graciosamente" o **programa** teclando `CTRL+C` ou `CTRL+D` (fim de arquivo em Linux) ou `CTRL+Z` (fin de arquivo em Windows): adicionar um bloco `try/except` na linha que lê a entrada. Tratar respectivamente as exceções `KeyboardInterrupt` e `EOFError`;
- Se quiser **tratar exceção** de **chave não encontrada** no dicionário: proteger a instrução que usa a chave no dicionário e tratar a exceção `KeyError`.