Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
DCC146 – Aspectos Teóricos da Computação

Trabalho Prático - Segunda Entrega

Beatriz Cunha Rodrigues - 201776038
João Pedro Sequeto Nascimento - 201776022
Marcus Vinícius V. A. Cunha - 201776013
Milles Joseph M e Silva - 201776026

Introdução

O objetivo dessa documentação é descrever as funcionalidades desenvolvidas, o projeto de estrutura de dados e decisões de projeto realizadas para o trabalho prático, assim como auxiliar na compilação e execução.

Equipe de Desenvolvimento

A equipe de desenvolvimento é formada por quatro alunos:

- João Pedro Sequeto
- Beatriz Cunha Rodrigues
- Marcus Vinícius V. A. Cunha
- Milles Magalhães

Linguagem De Programação

A linguagem de programação escolhida para o desenvolvimento do projeto foi o Java, utilizando a versão 8, com o uso da ferramenta Maven para gerenciamento de dependências e automação da Build.

Estruturas de Dados

No desenvolvimento do projeto foram utilizadas as seguintes estruturas de dados para as seguintes funcionalidades:

- HashMap: o HashMap armazena as informações em pares, ou seja, é possível armazenar uma chave e um valor correspondente. Utilizamos essa estrutura para armazenar a tag pela praticidade e organização para identificar e trabalhar com as mesmas. Com isso, armazenamos como “key” o nome da tag (exemplo: “EQUALS”) e armazenamos o seu “value” como o valor correspondente a tag (exemplo: “=”). Sendo assim, conseguimos manter as informações de forma correspondente usando apenas uma estrutura.
- List: a estrutura List foi necessária toda vez que um conjunto de determinado objeto precisou ser listado. Por exemplo, listas de estados e transições de autômatos, além da lista de autômatos em memória.
- Queue: a fila foi usada pela sua política FIFO nos métodos de conversão AFN λ para AFN e AFN para AFD, dado que tanto para as novas transições no primeiro quanto os estados novos criados para o AFD no segundo caso eram processados por ordem de criação.
- Stack: a pilha foi usada em dois momentos do trabalho: para busca de caminho entre estados (necessário para remover estados inúteis e inalcançáveis) e para validação de expressão regular. No primeiro

caso, considera-se que uma fila também teria funcionado, porém com uma ordem de validação diferente, pois com a pilha seriam verificados os últimos estados adicionados à pilha de processamento, enquanto com a fila teriam sido os primeiros.

Funcionamento da aplicação

Inicia-se a aplicação com o usuário digitando um input, que pode servir como comando ou como uma tag nova e sua definição. Caso seja um comando (iniciando com dois pontos a string), os que estão implementados são os a seguir:

- :d - Realiza a divisão em tags da string do arquivo informado;
- :q - Termina a aplicação;
- :c - Carrega um arquivo .lex com as definições de tags;
- :l - Lista as definições de tag válidas já processadas pela aplicação;
- :s - Salva as tags;
- :o - Especifica o caminho do arquivo de saída para a divisão em tags;
- :p - Realiza a divisão em tags da entrada informada;
- :a - Lista as definições formais dos autômatos em memória

Caso o comando seja inválido, haverá uma mensagem de erro “Comando não encontrado”. Agora, na situação de não ser um comando, a aplicação irá entender que se trata de um nova tag e irá tentar processá-la. Após a validação do input para a nova tag, haverá uma verificação da existência da mesma, pois uma tag não pode ser incluída duas vezes.

Em todas as situações de erro neste processamento (tag inválida ou já incluída), mensagens de erro personalizadas são mostradas ao usuário via console.

Para a segunda parte do trabalho, temos a validação das tags inclusas em expressões regulares, a transformação destas em autômatos

(na ordem: AFN λ -> AFN -> AFD), o casamento das tags com palavras inseridas pelo usuário e operações de leitura e escrita de tags.

Partindo do final da primeira parte do trabalho, a partir do momento que temos a tag, precisaremos validá-la enquanto expressão regular.

Para validação da expressão regular, é realizado o processamento da expressão regular pela estrutura de pilha, da seguinte forma: caso o caracter seja um operando, é realizado a criação do AFN que reconhece esse caracter e esse AFN é empilhado na pilha. Caso o caracter seja um operando (+, . ou *), os AFNs presentes na pilha são desempilhados, é realizada a operação nos AFNs de acordo com o operando e o AFN resultante é empilhado na pilha. As operações são realizadas adicionando transições Lambda (λ) para unir, concatenar ou realizar o fecho de Kleene. Ao final do processamento, é verificado se na pilha existe somente um elemento, caso seja, esse é o AFN λ válido para reconhecer palavras da linguagem denotada pela expressão regular.

Ao final deste primeiro processo, teremos um AFN λ válido. Este AFN será convertido em um AFN comum e todos os estados inúteis e inacessíveis serão removidos. Por fim, haverá uma conversão do AFN sem transições λ para AFD, cujo algoritmo segue a mesma lógica da montagem da tabela de conversão, montando novos estados cujo rótulo será a concatenação dos estados que o compõem. Com este processo completado, a tag é salva, assim como o AFD gerado.

Para todos os efeitos, seguem-se as seguintes regras no programa:

1) Na criação de estados, há um contador de estados já criados e o rótulo de um estado novo de AFN sempre será o valor do contador+1. AFDs não seguem este processo caso sejam originados de um AFN, mas podem seguir caso o AFD seja gerado diretamente via código.
2) É definido que um estado é igual ao outro caso tenham o mesmo rótulo.
3) É definido que uma transição é igual a outra caso estado de origem, estado de destino e caracter consumido sejam iguais em ambas as transições.

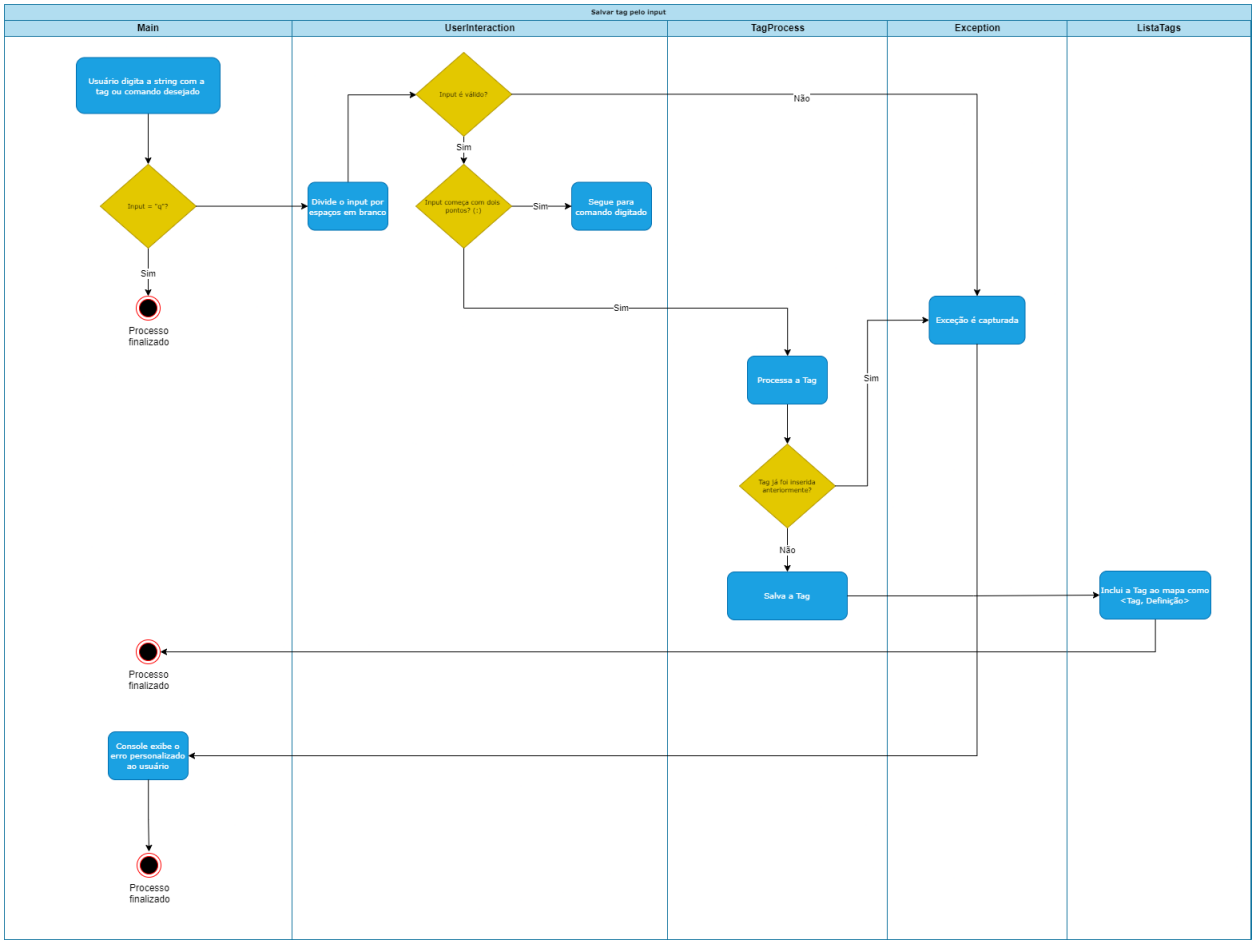
4) Um estado é dito inacessível se, a partir de todos estados iniciais, nenhum consegue alcançá-lo.
5) Um estado é dito inútil caso um estado não consiga alcançar nenhum dos estados finais do autômato.

Outras operações incluem: carregar as definições formais dos autômatos, carregar as tags dos arquivos e processá-las e dividir uma string em tags, podendo ser digitada pelo usuário ou carregada de um arquivo.

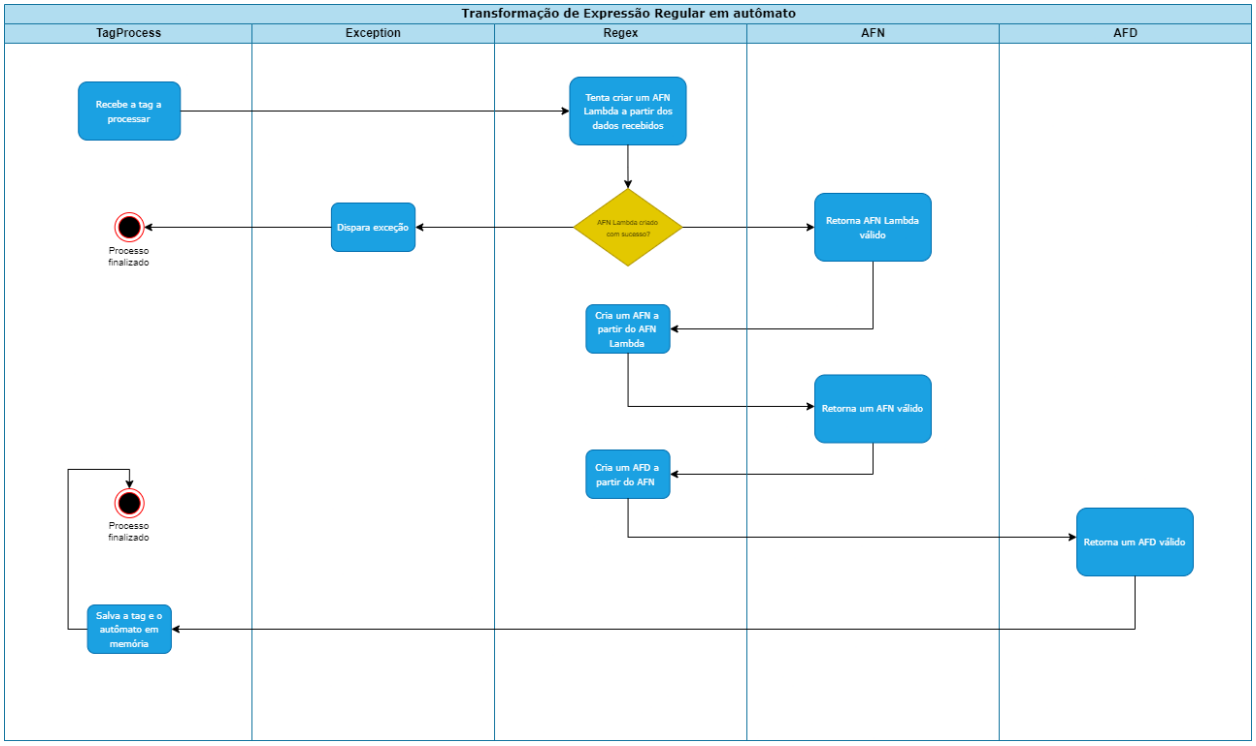
A divisão em tags ocorre passando a entrada pelos AFDs salvos e verificando se algum deles o reconhece. Caso tenha mais de um AFD que reconheça a entrada, será priorizado o primeiro AFD salvo.

Diagramas de Sequência

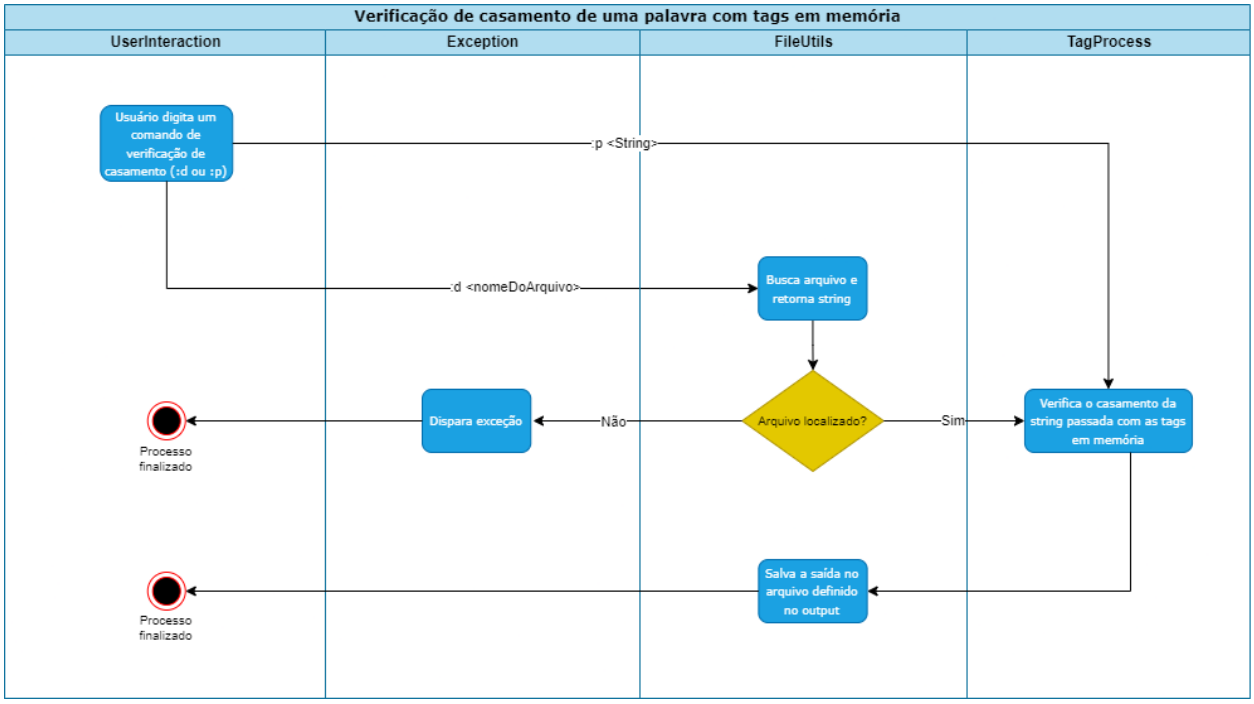
- Processamento e inclusão da tag digitada pelo usuário e sua definição.



- Processamento da tag como expressão regular e criação dos autômatos.



- Divisão de string em tags salvas em memória



Módulos e Classes

O trabalho foi implementado em módulos para facilitar a implementação e manutenção do projeto. Os módulos e classes criados são os seguintes:

- Exceptions: módulo responsável pelas classes de exceção
    - InputErrorException: disparada caso seja digitado um input incorreto na tela de comandos (ex: string vazia ou fora do padrão).
    - AutomataProcessingException: disparada na situação de erro no processamento de uma expressão regular para autômato.
- Interaction: módulo responsável pelas classes de interação de usuário;
    - UserInteraction: classe que filtra os comandos e definição de tag, caso aplicável.
- Model: módulo que contém as classes de domínio
    - ListaTags: classe que contém o mapa <Tag, Definição> e os métodos auxiliares.
    - ListaAutomatos: classe que contém a lista de autômatos em memória e os métodos auxiliares.
    - Automato: classe abstrata que contém os atributos e métodos comuns às classes AFN e AFD.
    - AFN: classe que contém atributos e métodos próprios ao processamento de um autômato finito não-determinístico.
    - AFD: classe que contém atributos e métodos próprios ao processamento de um autômato finito determinístico.
    - Estado: contém métodos e atributos para trabalhar com os estados de autômatos.
    - Transicao: contém métodos e atributos para trabalhar com as transições de autômatos.
    - Regex: classe responsável pelo tratamento de expressões regulares e conversão em AFN Lambda.

    - Word: Classe Responsável pela associação entre palavra e tag
- Resources: módulo que contém os processamentos dos elementos utilizados na aplicação
    - TagsProcess: classe que realiza o processamento da tag inserida no input para o mapa e de seu AFD para a lista de Autômatos em memória.
- Utils: módulo que contém classes auxiliares
    - FileUtils: classe que realiza leitura e escrita em arquivos;
    - IOUtils: classe para input e output em console, com mensagens de avisos, informações e erros em tela.

Dependências

- Apache Commons Text versão 1.6
- Java SE versão 14

Página do projeto

- https://github.com/sequeto/Analisador-Lexico

Compilação e Execução

Para rodar o projeto, é necessário ter o JRE 8 (Java Runtime Enviroment) instalado. Sendo assim, basta abrir o prompt na pasta principal do projeto e digitar o seguinte comando:

- **java -jar analisador-0.0.1-SNAPSHOT-jar-with-dependencies.jar**