Relatório EP4 MAC0323

Matheus Sanches Jurgensen (12542199)
15/07/2022

1. ENTREGA

Este documento é o relatório do EP4 de MAC0323. Ele integra o arquivo de tipo .zip (*EP4_MAC0323.zip*) que contém todos os arquivos a serem entregues, necessários para o correto funcionamento do arquivo principal, nomeado *RegExp.java*. Ele deve ser compilado através do comando *"javac RegExp.java"* e executado por meio de *"java RegExp"*. Após a execução, será solicitado ao usuário que digite o endereço do arquivo de entrada, o qual deve seguir o formato especificado no enunciado.

2. DETALHES DE IMPLEMENTAÇÃO

2.1 OPERAÇÕES

Como é sabido, o objetivo deste projeto é fazer um programa que, recebendo uma expressão regular, identifique quais palavras de uma lista são reconhecidas por ela.

Para que isso seja possível, é criado um grafo a partir das operações contidas na expressão regular. Seguem-se os padrões definidos em aula para concatenação, parênteses, ou lógico ("|") e fecho ("*"). Enquanto isso, para o coringa ("."), o seu vértice inclui uma E-transição para frente, enquanto o fecho com pelo menos uma cópia ("+") é idêntico ao fecho comum, porém sem haver uma E-transição do abre parênteses - ou do caractere anterior ao fecho - até o "+".

Agora, para implementar as operações de conjunto, intervalo e complemento, elas são transformadas nas outras operações, cuja implementação já se sabe. Por exemplo, o conjunto [abcd], equivalente ao intervalo [a-d], é representado como (((a|b)|c)|d). Essa mesma lógica é utilizada para intervalos - incluindo-se todos os elementos cujo valor ASCII está entre os valores das duas extremidades do intervalo, as quais também são inseridas - e complementos. Para estes últimos, é trivial perceber que funcionam de maneira similar aos conjuntos, e eles serão melhor explorados em seguida.

Além disso, com relação às expressões regulares aceitas pelo programa, algumas observações são necessárias:

1. **Ou (|):** seus elementos devem estar envolvidos por um par de parênteses, além de não haver espaços. Exemplo: o correto é (a|b), e não a|b ou (a | b).
2. **Conjunto**: deve conter apenas letras e números, além de não ser vazio.
3. **Intervalo**: deve conter apenas letras e números.

4. **Complemento**: aceita todas as letras e números contidos no array *alfabeto[]* que não estão no conjunto.

Nesse contexto, explica-se, ainda, a importância do array *alfabeto[]*, declarado como uma variável estática do arquivo *RegExp.java*. Ele contém todos os caracteres (letras maiúsculas e minúsculas e números) a serem considerados na hora de se interpretar a operação do complemento. Ou seja, dada uma operação dessa, a expressão regular aceita todos os caracteres do array *alfabeto[]*, exceto aqueles contidos no complemento.

Ademais, outro array estático a ser destacado é o array *eh_caracter[]*, o qual, para cada caractere da expressão regular, indica se ele deve ser lido como caractere ou operação.

2.2 CLASSES E BIBLIOTECAS AUXILIARES

Para o correto funcionamento do programa, algumas classes e bibliotecas externas são utilizadas.

Primeiramente, utilizam-se as classes *Pilha.java* e *Grafo.java*, implementadas em projetos anteriores e fundamentais para este. O grafo é justamente a estrutura que permite averiguar se uma expressão regular reconhece determinada palavra, enquanto a pilha é fundamental para a montagem desse grafo.

Além disso, são utilizadas algumas classes do pacote *java.util*. Em primeiro lugar, utiliza-se a classe *ArrayList*, que implementa um array de tamanho dinâmico. Esse é o caso do array *eh_caracter[]*, o qual deve ser de tamanho variável por não se saber, inicialmente, qual o tamanho que a expressão regular terá no final (afinal, são feitas modificações nela para que sejam adaptados intervalos, complementos e conjuntos).

Em segundo lugar, utiliza-se a classe *StringBuilder*, cujo objeto é um *String* mutável, podendo-se concatenar e adicionar caracteres em quaisquer posições dele. Isso é importante para que uma nova expressão regular seja construída, a partir da original, adaptando-se as operações necessárias.

Por fim, utiliza-se a classe *Scanner*, que é quem possibilita a interação com o usuário: permite o recebimento de dados da entrada padrão e de arquivos de entrada.

2.3 FUNÇÕES E MÉTODOS

Agora, faz-se uma breve descrição de como atuam as funções e os métodos principais implementados, bem como quaisquer observações sobre ele necessárias.

Em primeiro lugar, destaca-se a função *alteraExpReg()*, a qual, recebendo um *String* com a expressão regular inicial, realiza as modificações e adaptações necessárias (conforme já explicado anteriormente, para conjuntos, intervalos e complementos) e retorna a expressão regular final, a qual será utilizada para reconhecer as palavras.

Em seguida, há a função *constroiGrafo()*. Essa função recebe um *String* com a expressão regular, bem como um array de caracteres. Esse array, nomeado *letras[]*, tem o tamanho equivalente ao tamanho da expressão regular aumentado em 1, que é, também, a quantidade de vértices que deve ter o grafo a ser construído. Esse array contém o caractere *"¢"* para vértices do grafo que não representam caracteres, mas sim operações (ou parênteses). Qualquer outro caractere em uma posição do array indica, justamente, a presença desse caractere no vértice equivalente do grafo. Com isso, essa função constrói o grafo que representa a expressão regular e o retorna, além de preencher corretamente o array *letras[]*.

É importante destacar, também, que o coringa é representado, no array *letras[]*, como o caractere de valor ASCII 7, visto que ele é pouco utilizado e não contém uma representação visual.

Por fim, a função *reconhece()*, recebendo o grafo, o array *letras[]* e uma palavra, retorna *true* caso a expressão regular reconheça a palavra, e *false*, caso contrário, utilizando o procedimento de reconhecimento padrão, estudado e implementado em aula.

3. TESTES

Para a testagem das funções e métodos implementados, são realizados testes baseados naqueles contidos no enunciado, além de alguns extras.

Inicialmente, replica-se o primeiro teste do enunciado:

```
(([a-z])*|([0-9])*)*@(([a-z])+\.)+br
2
cef1999@ime.usp.br
thilio@bbb.com
```

```
Qual o arquivo de input? test1.txt
S
N
```

Em seguida, realiza-se um segundo teste similar a esse primeiro, porém aceitando que a palavra termine em *"com"*.

```
(([a-z])*|([0-9])*)*@(([a-z])+\.)+(br|com)
2
cef1999@ime.usp.br
thilio@bbb.com
```

```
Qual o arquivo de input? test1.txt
S
S
```

Agora, realiza-se o segundo teste do enunciado:

```
(.)*A(.)*
4
AAAAAAAAA
BCA
AAAAABBBBBB
BBB
```

```
Qual o arquivo de input? test2.txt
S
S
S
N
```

Repete-se esse teste, porém, substituindo o *"A"*, primeiro, por *"B"* e, em seguida, por *"C"*.

```
(.)*B(.)*
4
AAAAAAAAA
BCA
AAAAABBBBBB
BBB
```

```
Qual o arquivo de input? test2.txt
N
S
S
S
```

```
(.)*C(.)*
4
AAAAAAAAA
BCA
AAAAABBBBBB
BBB
```

```
Qual o arquivo de input? test2.txt
N
S
N
N
```

Dessa vez, replica-se o terceiro teste do enunciado:

```
((A*CG|A*TA)|AAG*T)*
4
AACGTAAATA
CAAGA
ACGTA
AAAGT
```

```
Qual o arquivo de input? test3.txt
S
N
S
N
```

Agora, realizam-se dois testes similares ao último do enunciado, sendo o primeiro igual, e o segundo uma versão adaptada.

```
[^AEIOU][AEIOU][^AEIOU][AEIOU]
5
GATO
FINO
OLHO
BELO
RUSSO
```

```
Qual o arquivo de input? test4.txt
S
S
N
S
N
```

```
([^AEIOU]|[AEIOU])*
5
GATO
FINO
OLHO
BELO
RUSSO
```

```
Qual o arquivo de input? test4.txt
S
S
S
S
S
```

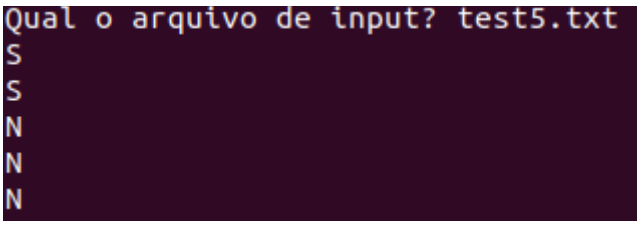
Testa-se, por sua vez, uma expressão regular que reconhece todos os número binários pares:

```
(0|1)*0
6
11010
11011
11200
11201
1
0
```

```
Qual o arquivo de input? test5.txt
S
N
N
N
N
S
```

Por fim, realizam-se testes com uma expressão regular utilizada para reconhecer números de telefone.

```
\([0-9]+\)9*[0-9]*\-[0-9]*
5
(19)99834-5509
(19)9834-5509
(19)98345509
()99834-5509
-
```



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