Trabalho prático: AFD e operações (versão: 24/5/21)

Disciplina:	Fundamentos Teóricos da Computação
Professor:	Vinícius Dias (viniciusvdias@ufop.edu.br)

Atenção: esta é uma versão preliminar e não isenta de erros da especificação com o objetivo de disponibilizar o trabalho com antecedência. Mais etapas podem ser incluídas ao longo do semestre. Tire suas dúvidas o quanto antes.

1 Introdução

O objetivo deste trabalho é possibilitar um contato de cunho prático dos alunos com formalismos teóricos de reconhecimento de linguagens, além de favorecer o reforço de algoritmos/operções importantes em se tratando de linguagens formais. Resumidamente, o aluno deverá implementar um programa em linguagem de programação C que cubra um conjunto de funcionalidades que permitam a manipulação de linguagens formais através de reconhecedores finitos (Automatos Finitos Determinísticos). AS estratégias de implementação dos algoritmos são amplamente discutidas na bibliografia da disciplina e é de responsabilidade do aluno entender, projetar e implementar tais funcionalidades. Todas as funcionalidades do seu programa devem seguir a seguinte especificação de linha de comando:

```
$ afdtool <lista-de-argumentos>
```

Isso quer dizer que nenhum tipo de interface gráfica deve ser implementada.

2 Formato de entrada

As entradas para as operações representam nomes de arquivos texto que contém a especificação de um AFD. Formalmente, um AFD A pode ser representado por uma quíntupla:

$A = (E, \Sigma, \delta, i, F)$,

em que $E = \{e_1, e_2, \ldots, e_{|E|}\}$ representa o conjunto de estados, $\Sigma = \{s_1, s_2, \ldots, s_{|\Sigma|}\}$ representa o conjunto do alfabeto de entrada, $\delta = \{\delta_1, \delta_2, \ldots, \delta_{|\delta|}\}$ formada por triplas ordenadas (onde $\delta_i = (e^a, s_b, e_c)$ indica que no estado e_a e lendo o símbolo s_b, a transição do AFD é ir para o estado e_c) representa a função de transição $E \times \Sigma \rightarrow E$, $i \in E$ representa o estado inicial e $F = \{f_1, f_2, \ldots, f_{|F|}\} \subseteq E$ representa o subconjunto de estados finais (de aceitação). Vamos definir duas três funções auxiliares $origem : (E \times \Sigma \rightarrow E) \rightarrow E$ que retorna o estado atual de uma transição, $simbolo : (E \times \Sigma \rightarrow E) \rightarrow \Sigma$ que retorna o símbolo lido durante uma transição; e $destino : (E \times \Sigma \rightarrow E) \rightarrow E$ que retorna o estado destino da transição.

Dito isso, um arquivo de entrada que representa um AFD de entrada tem o seguinte formato (com um exemplo logo ao lado):

O formato de entrada deve ser respeitado, sempre com quebras de linhas apropriadas e nos lugares corretos, o mesmo vale para os espaços em branco (únicos).

3 Funcionalidade 1: visualização

Nesta funcionalidade, o programa deve ser capaz de ler um AFD no formato de entrada e escrever o mesmo AFD em um formato de saída DOT pronto para visualização com o GraphViz[1]:

```
$ afdtool --dot afd.txt --output afd.dot
```

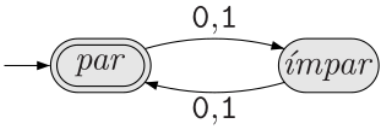
[1] <https://graphviz.org/Gallery/directed/fsm.html>

$|E|$
e_1
e_2
$\dots$
$e_{|E|}$
$|\Sigma|$
s_1
s_2
$\dots$
$s_{|\Sigma|}$
$|\delta|$
$o(\delta_1)\ s(\delta_1\ d(\delta_1)$
$o(\delta_2)\ s(\delta_2)\ d(\delta_2)$
$\dots$
$o(\delta_{|\delta|})\ s(\delta_{|\delta|})\ d(\delta_{|\delta|})$
i
$|F|$
f_1
f_2
$\dots$
$f_{|F|}$

(a) Formato genérico.

```
2
par
impar
2
0
1
4
par 0 impar
par 1 impar
impar 0 par
impar 1 par
par
1
par
```

(b) Arquivo `afd.txt`.



(c) AFD correspondente ao arquivo.

Figura 1. Formato de entrada

Veja que o arquivo .dot também é texto, mas em um formato diferente. Após gerado, o AFD pode ser visualizado usando o GraphViz:

```
$ dot -Tpdf afd.dot > afd.pdf
```

O comando anterior deve gerar uma figura isomórfica à figura 1.

4 Funcionalidade 2: complemento

Nesta funcionalidade, o programa deve ser capaz de ler um AFD no formato de entrada e escrever um outro AFD no arquivo de saída que reconheça o complemento da linguagem do autômato de entrada. O formato de escrita é o mesmo da entrada.

```
$ afdtool --complemento afd1.txt --output afd1-complemento.txt
```

5 Funcionalidade 3: interseção e união

Nesta funcionalidade, o programa deve ser capaz de ler dois AFDs no formato de entrada e escrever um outro AFD no arquivo de saída que reconheça a interseção ou união das linguagens dos autômatos de entrada. **Importante: ambas operações devem ser implementadas usando uma mesma subrotina que calcula o produto entre dois AFDs**. O formato de escrita é o mesmo da entrada:

```
$ afdtool --intersecao afd1.txt afd2.txt --output afd1-intersecao-afd2.txt
$ afdtool --uniao afd1.txt afd2.txt --output afd1-uniao-afd2.txt
```

6 Funcionalidade 4: minimização

Nesta funcionalidade, o programa deve ser capaz de ler um AFD no formato de entrada e escrever um outro AFD que reconheça a mesma linguagem mas que seja mínimo. O formato de escrita é o mesmo da entrada:

```
$ afdtool --minimizacao afd1.txt --output afd1-minimizacao.txt
```

7 Funcionalidade 5: reconhecimento de palavra

Nesta funcionalidade, o seu programa deve ser capaz de receber uma descrição de um AFD como no formato de entrada e um arquivo contendo *n* palavras de teste (uma por linha). O objetivo é simular o AFD usando cada uma das palavras e produzir no arquivo de saída para palavra de entrada: 0 se, e somente se, a palavra não pertence à linguagem reconhecida pelo AFD; ou 1 se, e somente se, a palavra pertence à linguagem reconhecida pelo AFD.

```
$ afdtool --reconhecer afd.txt palavras.txt --output palavras-reconhecidas.txt
```

A seguir, um exemplo de entrada e saída para o AFD da figura 1:

0101
101
01011

(a) `palavras.txt`

1
0
0

(b) `palavras-reconhecidas.txt`

Observe que o formato de palavras e saída possuem sempre um item por linha e esse formato deve ser respeitado.

8 O que deve ser entregue

1. Implementação (arquivos .c e/ou .h e makefile)

- *Linguagem*. Implemente tudo na linguagem C. Você pode utilizar qualquer função da biblioteca padrão (ou bibliotecas previamente combinadas) da linguagem em sua implementação, mas não deve utilizar outras bibliotecas. Trabalhos em outras linguagens de programação serão zerados. Trabalhos que utilizem outras bibliotecas não autorizadas também.

- *Ambiente*. O seu trabalho precisa ser compatível em vários ambientes (Linux, por exemplo), então não use bibliotecas que não sejam padrão. Seu programa vai ser testado em um ambiente Linux. Portanto, garanta que seu código compila e roda corretamente nesse sistema operacional. A melhor forma de garantir que seu trabalho rode em Linux é escrever e testar o código nele. Você também pode fazer o download de uma variante de Linux como o Ubuntu[2] e instalá-lo em seu computador ou diretamente ou por meio de uma máquina virtual como o VirtualBox [3]. Há vários tutoriais sobre como instalar Linux disponíveis na web.

- *Qualidade do Código*. Seu código deve ser bem escrito: (1) dê nomes a variáveis, funções e estruturas que façam sentido; (2) divida a implementação em módulos que tenham um significado bem definido; (3) acrescente comentários sempre que julgar apropriado; (4) não é necessário parafrasear o código, mas é interessante acrescentar descrições de alto nível que ajudem outras pessoas

[2] <http://www.ubuntu.com>
[3] <https://www.virtualbox.org>

a entender como sua implementação funciona; (5) evite utilizar variáveis globais; (6) mantenha as funções concisas: seres humanos não são muito bons em manter uma grande quantidade de informações na memória ao mesmo tempo. Funções muito grandes, portanto, são mais difíceis de entender; (7) lembre-se de indentar o código: escolha uma forma de indentar (tabulações ou espaços) e mantenha-se fiel a ela, misturar duas formas de indentação pode fazer com que o código fique ilegível quando você abri-lo em um editor de texto diferente do que foi utilizado originalmente. (8) evite linhas de código muito longas. Nem todo mundo tem um monitor tão grande quanto o seu. Uma convenção comum adotada em vários projetos é não passar de 80 caracteres de largura.

- **Makefile:** Esse arquivo é obrigatório e usado para compilar o seu trabalho. Uma referência introdutória sobre como usar *makefiles* para compilação pode ser encontrada no rodapé[4].

2. Documentação (arquivo .pdf) Você deve submeter uma documentação de até 10 páginas contendo uma descrição da solução proposta, detalhes relevantes da implementação, além de uma análise de complexidade de tempo dos algoritmos envolvidos e de complexidade de espaço das estruturas de dados utilizadas. A seguir, alguns tópicos essenciais esperados em uma boa documentação:

- *Introdução.* Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.
- *Solução do problema.* Você deve descrever a solução do problema de maneira clara e precisa. Para tal, artifícios como pseudocódigos, exemplos ou figuras podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. Não é preciso incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando os mesmos influenciem o seu algoritmo principal, o que se torna interessante.
- *Guia de como usar o seu software.* De preferência, inclua um fluxo amigável de como as funcionaliades podem ser combinadas.

3. Apresentação (arquivo .txt com link para video da apresentação) O link para um video contendo a apresentação do software desenvolvido. A apresentação precisa ter a participação de todos os integrantes do grupo. A apresentação deve conter: (1) uma breve descrição do problema; (2) uma breve descrição da interação entre os programas desenvolvidos e como o problema é solucionado; (3) uma breve demonstração da compilação e execução do trabalho. **Dica:** grave seu video usando o Google Meet e poste no youtube ou através de um link direto para a gravação no Google Drive.

9 Como deve ser entregue

O trabalho deve ser entregue pelo Moodle[5] na forma de um único arquivo zipado (formato `.zip`) contendo documentação, implementação e makefile organizadas em um diretório chamado “tp”:

```
tp/
  |- tp.pdf
  |- <arquivos .c e .h>
  |- ...
  |- makefile
  |- link-apresentacao.txt
```

[4] <https://bit.ly/2PrgIJk>
[5] <moodlepresencial.ufop.br>

Você deve compactar esse diretório e seu conteúdo em um arquivo único chamado `tp-<seu-nome>-<seu-sobrenome>.zip`. Caso o trabalho seja em grupo, use o seguinte formato `tp-grupo-<numero-grupo>.zip`. Arquivos em outros formatos (RAR, por exemplo) que não `.zip` não serão aceitos. Por exemplo, para o aluno "João da Silva": `tp-joao-silva.zip`. Ou para o grupo 4: `tp-grupo-4.zip`.

10 Avaliação

Este trabalho será avaliado de acordo com os seguintes critérios:

- O aluno seguiu a especificação do trabalho: formatos dos arquivos, nome dos arquivos, linguagem de programação, entrada e saída, argumentos, makefile, etc.;
- Qualidade da documentação: completude, texto bem escrito e formatado, análise de complexidade e experimental bem explicadas, etc.;
- Qualidade da implementação: código indentado, bem comentado, variáveis legíveis, código modularizado, correto, alocação dinâmica correta, etc.;
- Qualidade da apresentação: individual e como grupo.

11 Observações gerais

- Leia esta especificação com cuidado;
- Essa especificação não é isenta de erros e ambiguidades. Portanto, se tiverem problemas para entender o que está escrito aqui: pergunte!
- Comece o trabalho o quanto antes;
- O trabalho deve ser implementado em linguagem C. Trabalhos em outras linguagens (C++, Java, Python, etc.) não serão aceitos;
- Siga exatamente as especificações de formato de entrega, compilação e execução descritas neste documento;
- Apenas um arquivo deve ser entregue, compactado e no formato `.zip`;
- Seu trabalho deve ser compatível com ambiente Linux;
- **Seja honesto.** Você não aprende nada copiando código de terceiros nem pedindo a outra pessoa que faça o trabalho por você. Se a cópia for detectada, sua nota será zerada e o colegiado será devidamente informado para que providências possam ser tomadas.