Relatório do Trabalho Prático 01

Fundamentos da Teoria da Computação

Thiago Martin Poppe

09/02/2021

1 Uma breve descrição

1.1 Problema proposto

Automatizar o processo de conversão de um autômato finito (AFD, AFN ou AFNλ) para uma expressão regular, não necessariamente mínima, através do algoritmo de remoção de estados, lecionado durante as aulas assíncronas.

1.2 Observações

- A automatização do algoritmo foi feita utilizando a linguagem Python3 e suas bibliotecas padrões, em especial o módulo **sys** para podermos ler dados da linha de comando. O código principal pode ser encontrado no arquivo **main.py**.

- Por convenção foi utilizado o símbolo λ para representar a palavra vazia e transições-λ em AFNλ. Sendo assim, é recomendado o uso da codificação UTF-8 para não ter problemas com a saída do programa.

- Não foi feita nenhuma otimização sobre o algoritmo, i.e não temos a garantia de encontrar a expressão regular mínima equivalente ao autômato finito. Também não foram feitas otimizações bruscas quanto ao uso dos parênteses; com isso, a solução final pode ser um pouco difícil de ler, mas há garantia de não gerar expressões ambíguas. Sendo assim, em alguns casos, os parênteses podem ser removidos, como por exemplo: $((0+1))^* \equiv (0+1)^*$ e $(0+1)+2 \equiv 0+1+2$, porém note que $(0+1)2 \neq 0+12$.

- Uma observação importante é que o arquivo de entrada **NÃO** deve conter estados que possuem vírgula no nome, visto que o próprio arquivo é separado por vírgulas, podendo gerar erros durante a execução ou uma expressão regular incorreta. Uma outra restrição é **NÃO** utilizar os nomes **RegEx_initial_state** e **RegEx_final_state** como nome de estados, visto que os mesmos foram utilizados para representar o novo estado inicial e final do diagrama ER.

2 Algoritmo de Eliminação de Estados

2.1 Ideia geral do algoritmo

Inicialmente, teremos que converter o autômato finito para um diagrama ER. Para tal, iremos criar um novo estado inicial, conectando este com todos os demais estados iniciais através de transições λ; criaremos também um novo estado final, aplicando a mesma ideia dita anteriormente, porém no sentido inverso. Além disso, em transições envolvendo mais de um símbolo, iremos "agregar" os mesmos através do operador +, por exemplo: a transição **0,1** passa a ser **(0+1)**.

Com o diagrama ER em mãos, podemos iniciar o processo de remoção dos estados que se dará apenas para os estados diferentes do novo estado inicial e final.

Iremos computar caminhos da forma (e_1, e, e_2), sendo e o estado que queremos remover, e_1 um estado anterior ao e (estado pai) e e_2 um estado posterior ao e (estado sucessor). Note que podemos ter o caso onde $e_1 = e_2$, mas nunca podemos ter $e_1 = e$ ou $e_2 = e$. Com isso, teremos dois casos gerais possíveis, onde r_i são expressões regulares:

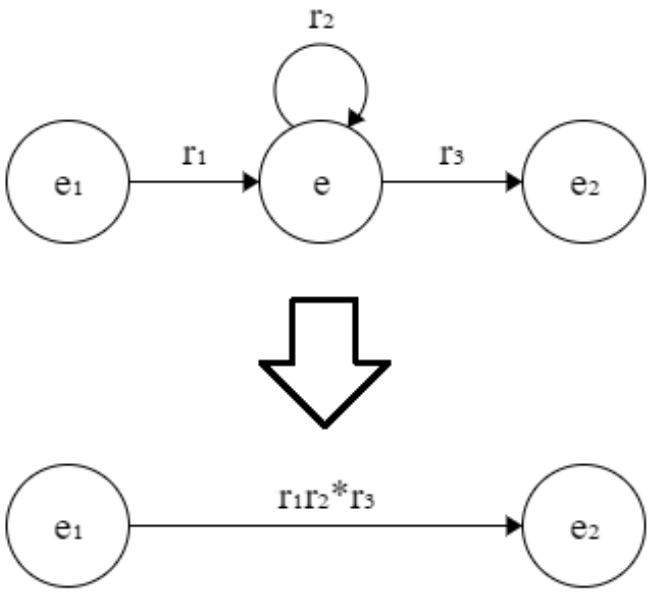

Figure 1: Caso onde $e_1 \neq e_2$

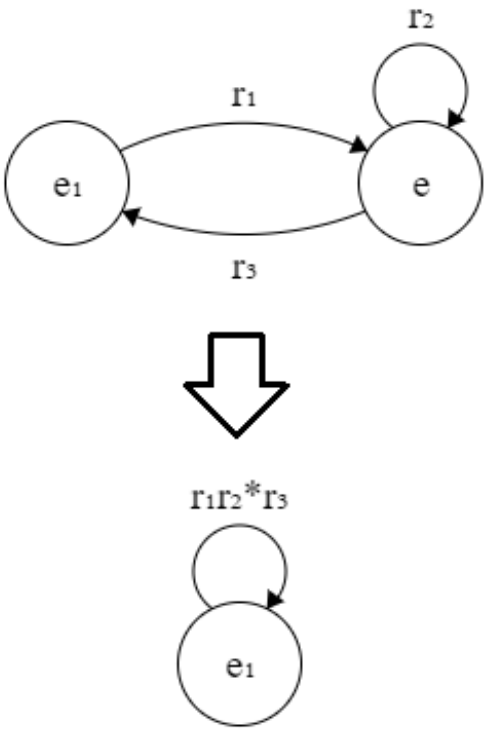

Figure 2: Caso onde $e_1 = e_2$

Em ambos os casos, podemos ter o caso que não temos transição de $e \to e$, dessa forma, não teremos o termo r_2^* na nossa expressão regular. Podemos também ter uma transição r_0 de $e_1 \to e_2$; nesse caso, basta unirmos as duas expressões com o operador $+$, resultando em algo similar a $r_0 + r_1 r_2^* r_3$.

Após remover todos os estados, teremos apenas o estado inicial e final do nosso diagrama ER. Com isso, a expressão regular equivalente será justamente a expressão que realiza a transição entre esses dois estados. Vale ressaltar que a ordem de remoção é irrelevante, podendo assim remover os estados em qualquer ordem. No caso da automatização proposta, por conveniência, a ordem de eliminação será a mesma ordem em que os estados aparecem no arquivo de entrada.

2.2 Implementação e estruturas utilizadas

A implementação do autômato finito e, consequentemente, do diagrama ER foi feito através de estruturas de dados nativas do Python, sendo elas **list**, e **dict**.

A estrutura **list** foi utilizada para armazenar os estados, símbolos e estados iniciais/finais do autômato finito; já a estrutura **dict** foi usada para modelar a função de transição do autômato, representada pelo símbolo δ, e para modelar uma estrutura auxiliar que armazena os estados pais de cada estado a fim de facilitar o algoritmo.

Uma mudança perceptível foi que a função de transição não é $state \times symbol \rightarrow state$, mas sim algo como $state \times state \rightarrow symbols$. Em outras palavras, a função mapeia o produto cartesiano dos estados para uma lista de símbolos que indica quais símbolos são usados durante a transição entre dois estados. Essa mudança foi feita para facilitar e tornar a automatização do algoritmo mais otimizada e simples, visto que dado um estado S, conseguimos saber de forma rápida quais são os seus sucessores e também conseguimos modificar de forma bem simples a forma como nosso autômato irá transitar sobre os estados. Mais especificamente, conseguimos facilmente converter a lista $[0, 1]$ para a expressão regular $(0 + 1)$, e futuramente a concatenação e união de outras expressões regulares, por exemplo.

3 Exemplos utilizados

Foram usados 10 exemplos para verificar a corretude da automatização, dentre eles alguns vistos ao longo de listas de exercícios e da prova 01. A seguir, demonstrarei o passo a passo de apenas 1 exemplo e somente as respostas para os demais, comentando alguns pontos importantes.

3.1 Exemplo 01 - AFN

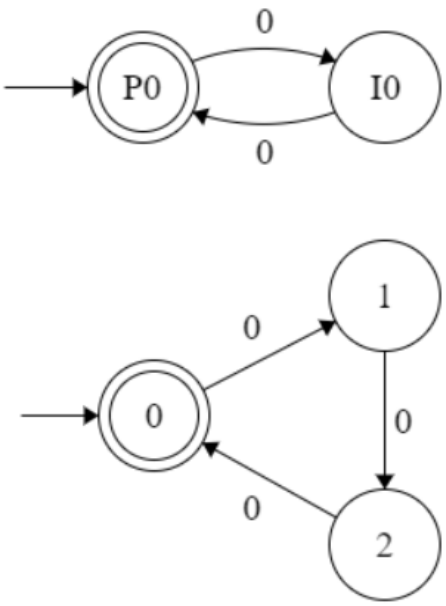

Figure 3: $L = \{w \in \{0\}^* \mid |w| \ mod \ 2 \equiv 0 \vee |w| \ mod \ 3 \equiv 0\}$

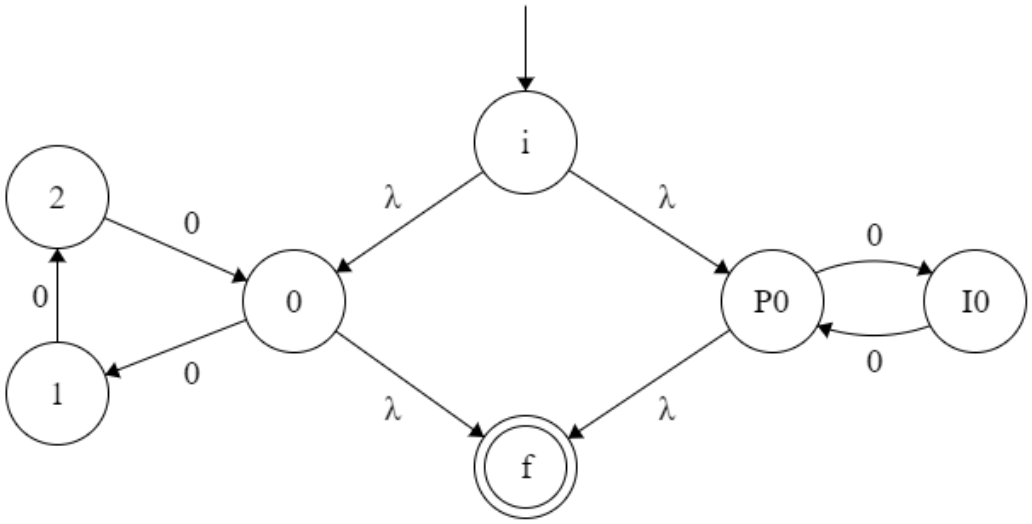

Figure 4: Construção do Diagrama ER

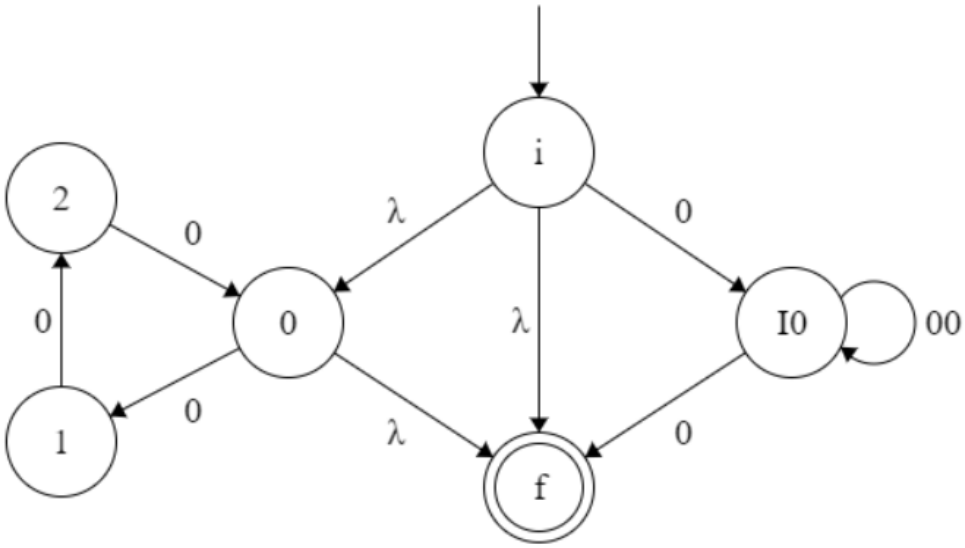

Figure 5: Remoção do estado P0

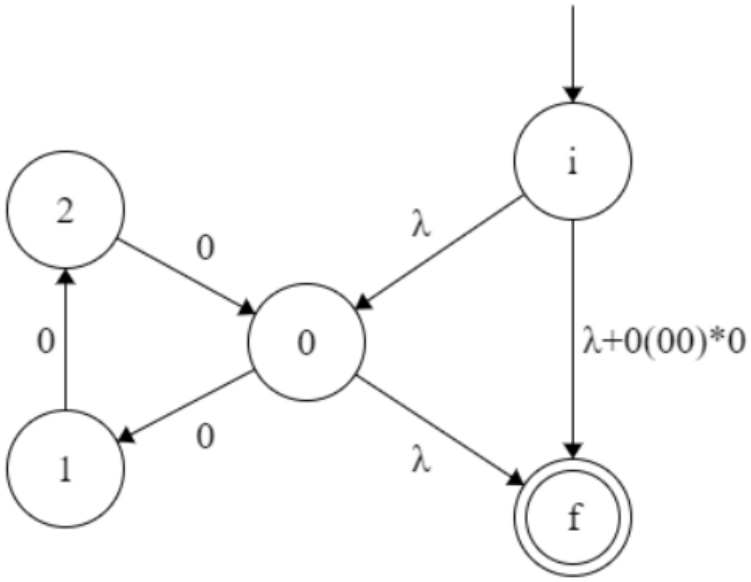

Figure 6: Remoção do estado I0

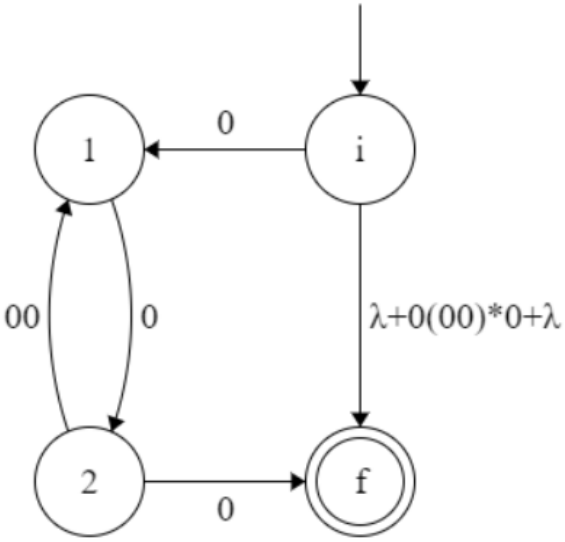

Figure 7: Remoção do estado 0

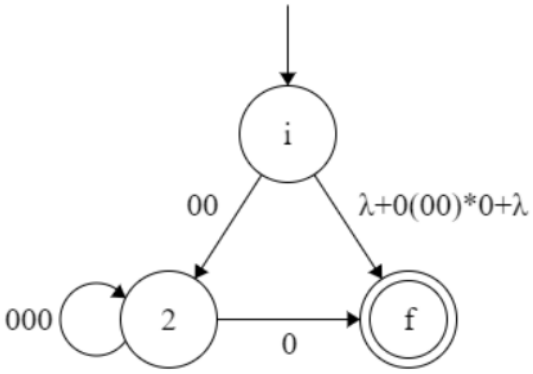

Figure 8: Remoção do estado 1

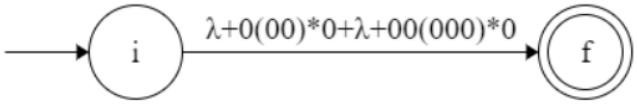

Figure 9: Remoção do estado 2

Através do passo a passo, temos que a expressão regular, não minimizada, será $\lambda + 0(00)^*0 + \lambda + 00(000)^*0$; já a saída do nosso programa será $(((\lambda + 0(00)^*0) + \lambda) + 00(000)^*0)$. Desconsiderando os parênteses desnecessários, temos a mesma expressão regular. Podemos interpretar ela como sendo a palavra vazia, uma sequência par de símbolos 0 ou uma sequência ímpar de símbolos 0, o que está coerente com a linguagem.

3.2 Exemplo 02 - AFN

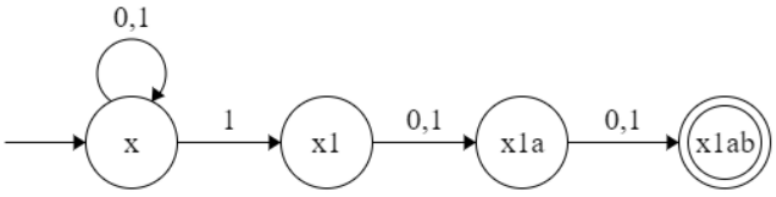

Figure 10: $L = \{w \in \{0,1\}^* \mid$ o antepenúltimo símbolo de w é $1\}$

A saída do programa será: $((0 + 1))^*1(0 + 1)(0 + 1)$, que está coerente com a linguagem.

3.3 Exemplo 03 - AFN

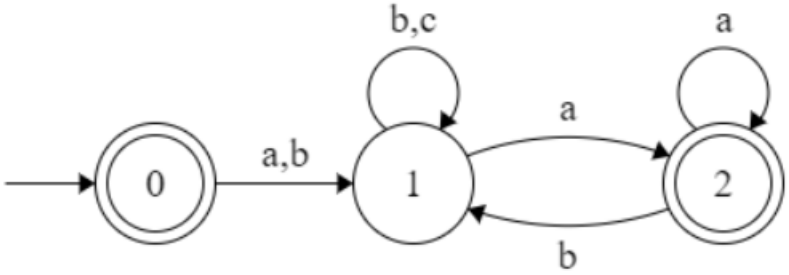

Figure 11: AFN qualquer construído para corretude do algoritmo

A saída do programa será: $(\lambda + (a + b)((b + c))^*a((a + b((b + c))^*a))^*)$, que está coerente com o autômato apresentado. Através da expressão temos que λ é uma palavra válida, $(a+b)(b+c)^*a$ também é válida e essa palavra concatenada com $(a+b(b+c)^*a)^*$ também é válida, ou seja, estando no estado final, podemos ler uma quantidade arbitrária do símbolo a ou voltar para o estado 1 e transitar para o 2 quantas vezes quisermos.

3.4 Exemplo 04 - AFN

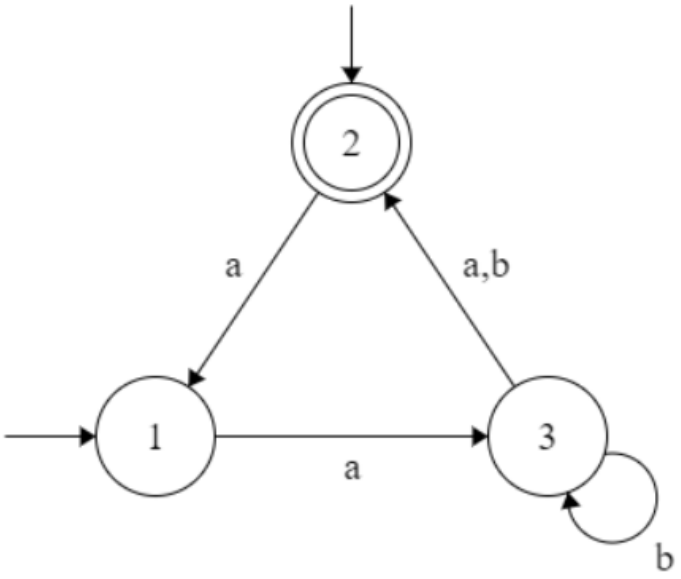

Figure 12: AFN da última questão da prova 01

A saída do programa será: $(\lambda + (a + aa)((b + (a + b)aa))^*(a + b))$, que está coerente com o autômato apresentado.

3.5 Exemplo 05 - AFN

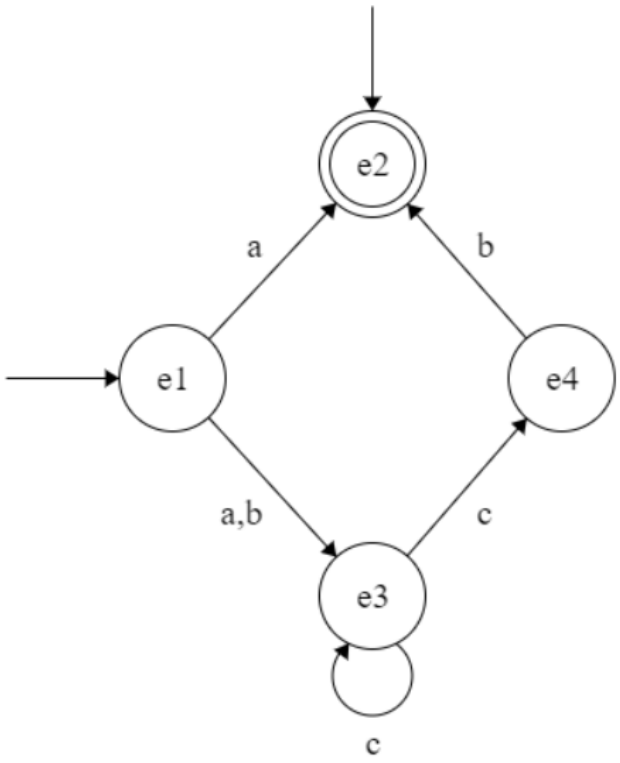

Figure 13: AFN qualquer construído para corretude do algoritmo

A saída do programa será: $((\lambda + a) + (a + b)c^*cb)$, que está coerente com o autômato apresentado, apesar dos parênteses desnecessários, sendo equivalente a $\lambda + a + (a + b)c^*cb$, representando os "3 fluxos" possíveis começando dos estados iniciais até o estado final.

3.6 Exemplo 06 - AFD

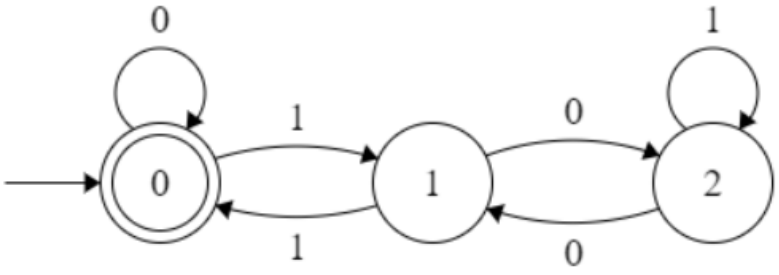

Figure 14: AFD que reconhece números binários divisíveis por 3

A saída do programa será: $((0 + 1(01^*0)^*1))^*$, que está coerente com a linguagem. Um fato interessante é que se eliminarmos os estados na ordem $0 \to 1 \to 2$, a expressão regular final é muito maior e muito mais difícil de corrigir do que essa, que foi gerada a partir da ordem de eliminação $2 \to 1 \to 0$.

3.7 Exemplo 07 - AFD

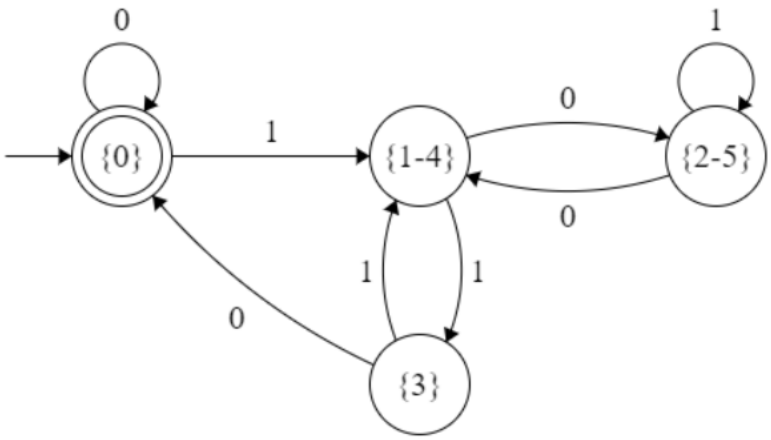

Figure 15: AFD mínimo que reconhece números binários divisíveis por 6

A saída do programa será: $((0^* + 0^*11((1 + 00^*1)1)^*00^*) + (0^*10 + 0^*11((1 + 00^*1)1)^*(1 + 00^*1)0)(((1+00)+01((1+00^*1)1)^*(1+00^*1)0))^*01((1+00^*1)1)^*00^*)$, que está coerente com a linguagem, apesar dos parênteses desnecessários e da expressão longa. Respostas longas são mais difíceis de corrigir, então seria uma boa opção tentar outra ordem de eliminação de estados a fim de minimizar a expressão.

3.8 Exemplo 08 - AFD

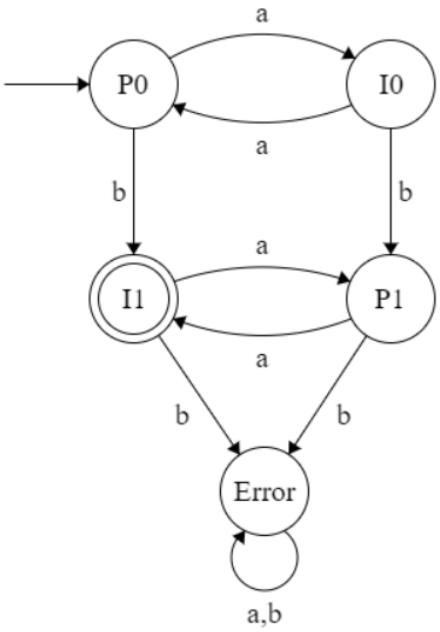

Figure 16: $L = \{w \in \{a, b\}^* \mid |w| \ mod \ 2 \neq 0 \wedge \eta_b(w) = 1\}$

A saída do programa será: $((b + a(aa)^*ab) + (a(aa)^*b + (b + a(aa)^*ab)a)(aa)^*a)$, que está coerente com a linguagem. Algo bem interessante a ser notado é que se retirarmos o estado $Error$, a expressão permanece a mesma, visto que esse estado não possui sucessores diferentes dele mesmo, ou seja, ele não é "processado" pelo algoritmo.

3.9 Exemplo 09 - AFD

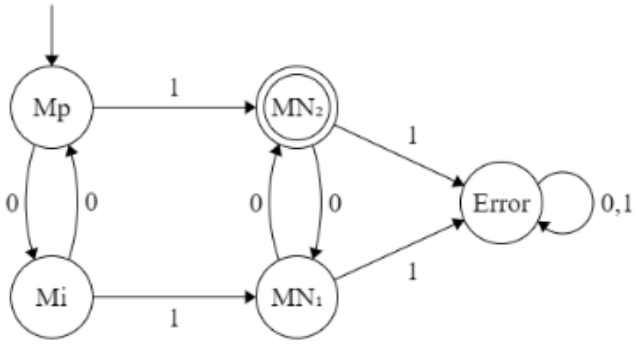

Figure 17: $L = \{0^m 10^n \mid m + n \text{ é par}\}$

A saída do programa será: $((1 + 0(00)^*01) + (0(00)^*1 + (1 + 0(00)^*01)0)(00)^*0)$, que está coerente com a linguagem. Podemos interpretar essa expressão regular como sendo palavras

no formato: 1, 0(00)*01, 10(00)*0, 0(00)*1(00)*0 e 0(00)*010(00)*0.

3.10 Exemplo 10 - AFDλ

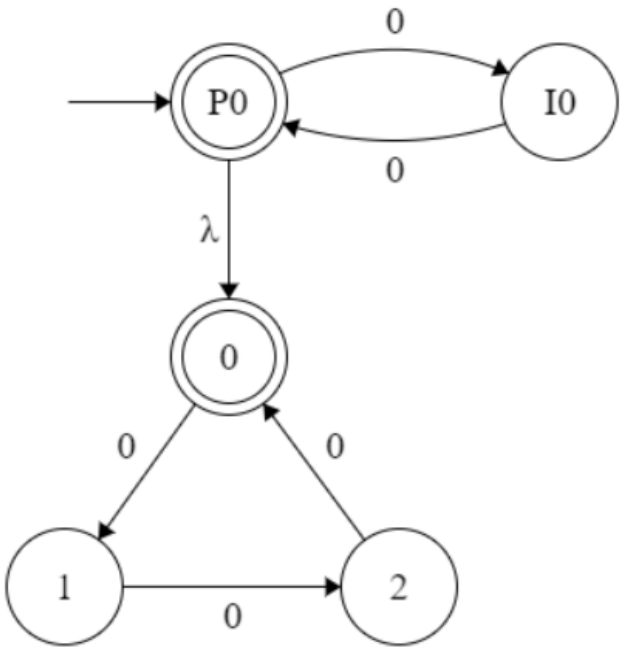

Figure 18: AFNλ similar ao exemplo 01

A saída do programa será: $(((\lambda + 0(00)^*0) + (\lambda + 0(00)^*0\lambda)) + (\lambda + 0(00)^*0\lambda)00(000)^*0)$, que está coerente com o autômato apresentado.