Trabalho Prático 1 - Linguagens regulares

DCC129 - FUNDAMENTOS DA TEORIA DA COMPUTAÇÃO

Aluno: Danilo Pimentel de Carvalho Costa
Matrícula: 2016058077

Introdução

Este trabalho tem como objetivo praticar os conceitos aprendidos sobre Autômatos Finitos e Expressões Regulares. O trabalho a ser desenvolvido é a implementação de um programa que converte um autômato finito não determinístico com transições lambda (AFN Lambda) em uma Expressão Regular. O autômato é recebido em um arquivo de entrada, e a expressão deve ser impressa na saída padrão.

Implementação

A especificação do trabalho permite a escolha entre as linguagens Python e C para a implementação do programa. Foi escolhida a linguagem Python. Ainda é dada a opção de uso da biblioteca numpy, para manipulação de dados em vetor. A implementação não utilizou a biblioteca numpy, mas sim somente a biblioteca padrão do Python versão 3.7.7.

O programa desenvolvido recebe o nome arquivo de entrada como argumento na linha de comando. O arquivo é lido, e as informações de estados, alfabeto, estados iniciais, estados finais e transições são extraídos. O programa não utiliza o alfabeto, porém pode-se notar que ele poderia ser utilizado para validar o arquivo de entrada procurando por erros na construção do autômato. Assim, a utilização do alfabeto para este fim fica como sugestão de melhoria futura.

Foram criadas 2 classes para representação de estado e transição: State e Transition. Em State, além do nome do estado, são guardadas as características de ser final e ser inicial. Em Transition, são guardadas as informações de estado fonte, estado destino, e o símbolo para transição. Objetos são criados para os dados recebidos como entrada e para outros criados durante a execução. A comparação entre objetos foi útil durante o desenvolvimento para determinar igualdade entre estados e transições.

A classe AfnLambda foi criada para auxiliar a representação e acesso aos dados do autômato de entrada, e somente um objeto desse tipo é instanciado durante toda a execução. Este objeto guarda os estados (State) e transições (Transition) criados a partir das informações de entrada, além de ter métodos úteis para a recuperar somente estados iniciais e finais, e adicionar novas transições.

Uma vez feitas as transformações iniciais para representar melhor as informações de entrada, o programa segue para a lógica que de fato realiza a extração da expressão regular. O algoritmo é o mesmo visto em sala:

1. Novos estados inicial e final:

    a. É adicionado um novo estado inicial I, com transições lambda deste para os estados iniciais do AFN Lambda. Os estados iniciais anteriores passam a ser somente estados simples.
    b. É adicionado um novo estado final F, com transições lambda dos estados finais do AFN Lambda para este novo estado. Os estados finais anteriores passam a ser somente estados simples.
2. A partir do AFN Lambda, cria-se um Diagrama ER. Transições neste diagrama são expressões regulares. A transformação consiste em juntar transições que têm a mesma fonte e destino em uma só transição. A expressão desta nova transição é uma união dos símbolos das transições anteriores.
3. São feitas todas as eliminações possíveis de estados diferentes de I e F. As regras para eliminação se baseiam no conteúdo visto em sala, mas outros detalhes foram descobertos durante a implementação, e são descritos abaixo.

A implementação deste algoritmo começa na classe Der (sigla para diagrama ER). que realiza as operações dos passos 1 e 2. No passo 1, um novo objeto de State é criado com a propriedade inicial colocada em verdadeiro. Um objeto de Transition é criado para cada estado inicial anterior, com a fonte (src) sendo o novo estado inicial criado, o símbolo (symb) sendo lambda, e o destino sendo o estado inicial anterior em questão. Este estado inicial anterior é transformado em estado simples facilmente ao assinalar o atributo inicial como falso. O passo para a transformação dos estados finais segue a mesma lógica.

A representação de lambda escolhida foi o caractere λ. Inicialmente, foi utilizada somente a string vazia de entrada. Porém, isso causou bugs na extração da expressão regular, uma vez que a saída para string vazia não era percebida. A impressão final da expressão regular não era fiel ao resultado. Assim, foi escolhido este caractere para evitar esse problema.

Para o passo 2, transições novas são criadas. A implementação é bem simples: para cada estado são recuperadas as transições que partem dele. Outro laço é feito novamente para cada estado, procurando por transições que têm o mesmo estado fonte e destino. Para as transições "gêmeas" encontradas, seus símbolos são juntados com o caractere + que representa a união. Aqui há duas decisões de implementação:

- Por simplicidade foi utilizada a mesma classe Transition, pois o atributo symb é uma string e pode armazenar uma expressão regular.
- Na junção dos símbolos, sempre são adicionados parênteses em volta da expressão se ela tem mais de 1 símbolo. Isso evita problemas de precedência quando esta expressão é combinada com outras em passos seguintes.

O passo 3 foi o mais interessante. Uma nova classe RegExp foi criada para agrupar a sua implementação. A construção do objeto somente envolve a cópia dos estados e transições do diagrama recebido. O método build implementa a lógica de redução do diagrama, e o método display retorna a expressão resultante.

Na redução do diagrama, o método build na verdade remove transições, mas nunca remove estados. Remover os estados não era necessário na execução, e por isso sempre ficam armazenados no vetor de estados. O critério para seleção das transições a serem removidas segue o algoritmo descrito em sala: procura-se 3 estados e1, e, e2. A estratégia é procurar por uma transição entre e1-e, chamada de t1. Após achar esta transição, é encontrada a

transição entre e-e2, chamada de t3. Se encontradas estas transições, foi encontrado o estado e que pode ser removido do diagrama. Para a remoção deste estado, são também procuradas as transições t0 e t2, que representam e1-e2 e e-e. Se estas transições não existem, o estado "e" ainda é um candidato para ser removido.

Encontradas as transições t0, t1, t2, t3, é criada uma nova transição t. A expressão dessa transição segue a regra vista em aula: sejam r0, r1, r2, r3 as expressões correspondentes, a expressão resultante é r0+r1r2*r3. Os devidos cuidados com os parênteses foram tomados para não prejudicar a precedência no resultado final. Se t0 não existe, r0 não é adicionado. O mesmo para t2 e r2. A nova transição é adicionada, e t0. t1, t2, t3 são removidas. Aqui foram encontrados dois problemas:

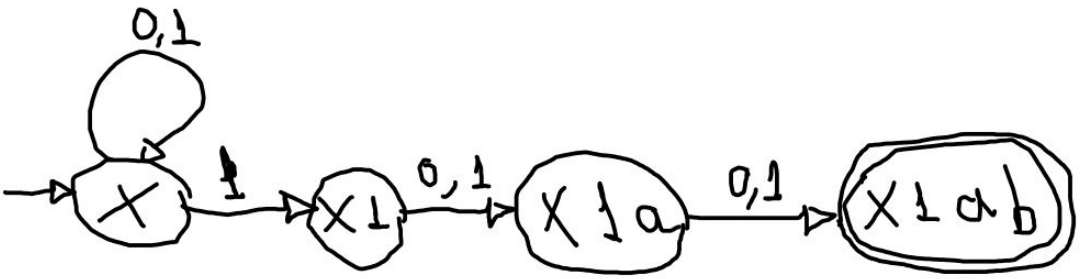
- Com a adição de t, o diagrama pode ter de ser alterado para juntar t a outras transições com os mesmos estados fonte e destino. A mesma estratégia adotada na transformação inicial foi utilizada aqui.
- Pode ser que haja outras transições com destino em "e". Neste caso, precisam ser adicionadas outras transições. A criação destas transições segue a mesma lógica descrita acima. Basta encarar uma transição deste tipo como a transição t1, e a construção da expressão na nova transição se dá da mesma forma.

Quando não são encontrados mais estados candidatos, a construção da expressão terminou. O método display simplesmente recupera a transição entre o estado inicial I e o estado final F. É impressa a expressão contida nesta transição. Se a transição não existe, significa que a expressão é vazia. Foi escolhido o caractere ∅ para representar este caso.

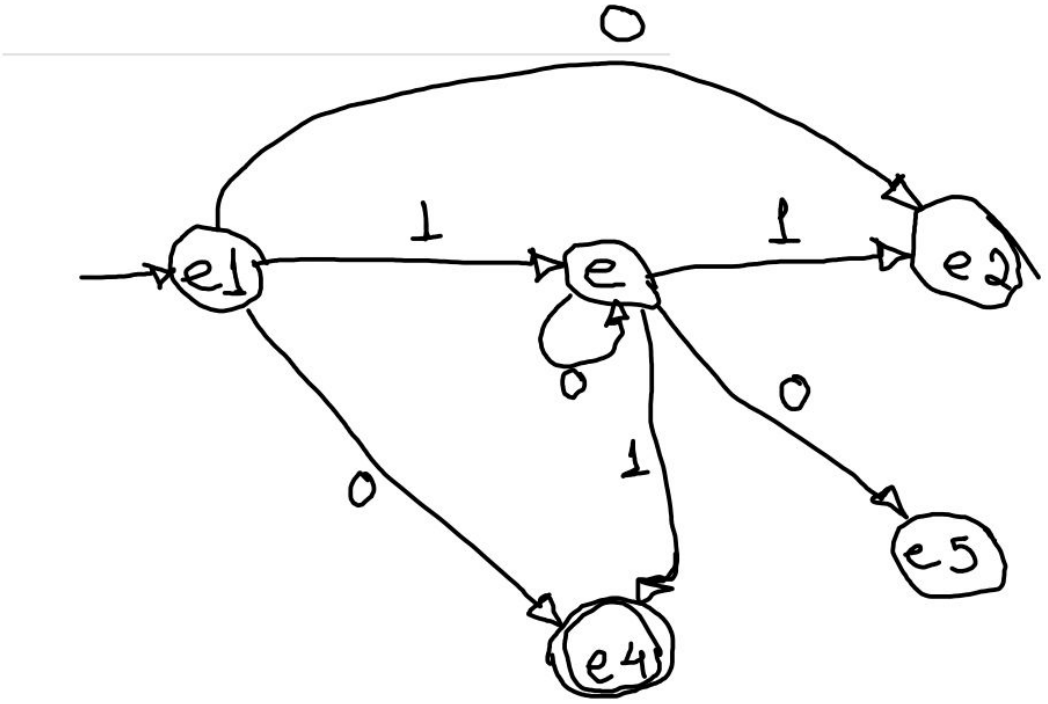
Exemplos

Como solicitado na especificação, foram construídos 10 exemplos não triviais. A trivialidade do exemplo ficou a critério do aluno. Neste trabalho, foram considerados autômatos que exploram os casos base: autômatos que podem ser combinados para a construção de qualquer expressão regular. Além disso, foram criados exemplos que exploram os casos de borda, como autômatos com componentes desconectadas, minimizações que não terminam somente com 1 transição final, união de linguagens, expressões vazias, e a própria questão da prova 1. A seguir se encontram os autômatos escolhidos com as respectivas expressões geradas pelo programa:

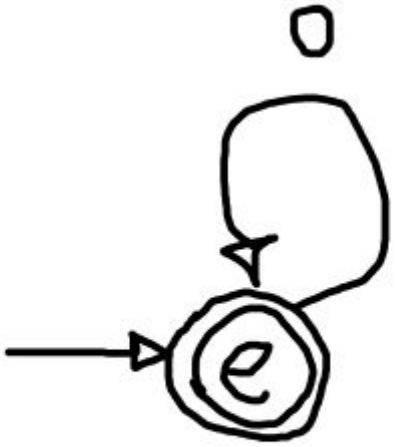
1. λ((0 + 1))*1(0 + 1)(0 + 1)λ



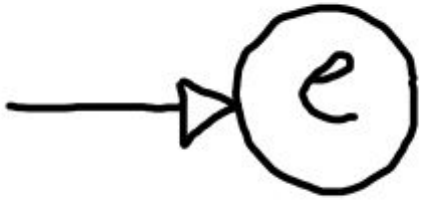
2. λ(0 + 1(0)*1)λ



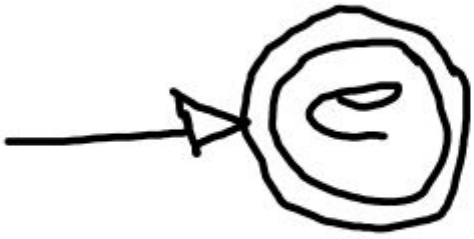
3. λ(0)*λ



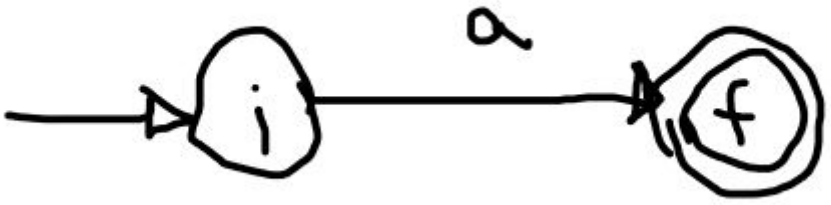
4. ⌀



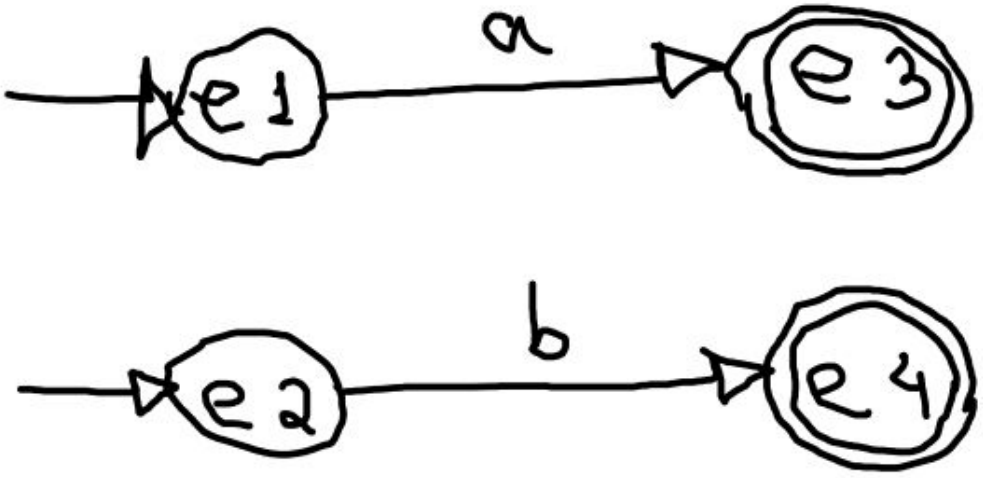
5. λλ



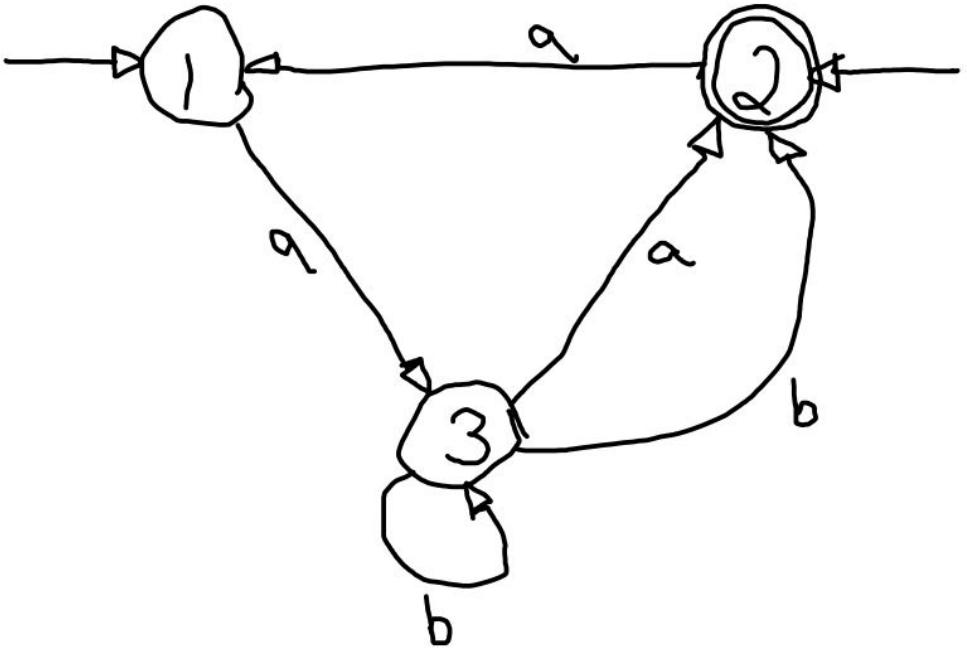
6. λaλ



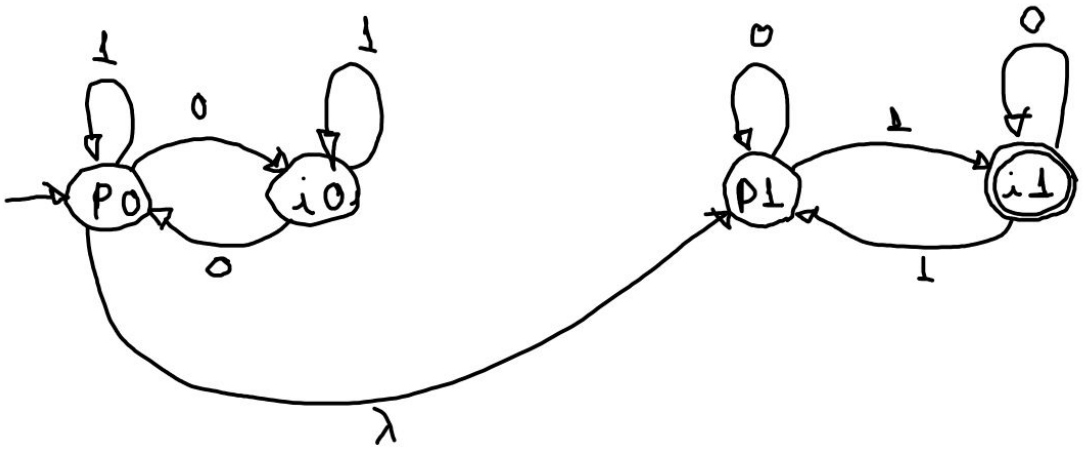
7. (λaλ + λbλ)



8. (λaa(b)*(a + b) + λ)(aa(b)*(a + b))*λ



9. λ((1 + 0(1)*0))*λ(0)*1((1λ(0)*1 + 0))*λ



10. ∅

Foram verificadas todas as respostas, e não foram encontrados erros nas expressões, que denotam a linguagem desejada. Vale notar que o resultado não representa a expressão regular mínima para o autômato, pois nenhuma técnica de minimização de expressões foi implementada. Um script auxiliar foi escrito para checar automaticamente se os casos de teste passam, e foi utilizado durante o desenvolvimento para facilitar a verificação de regressões e aderência a requerimentos para as alterações feitas.

Conclusão

Neste trabalho foi possível exercitar os conceitos aprendidos sobre autômatos finitos e expressões regulares. O programa desenvolvido atende aos requisitos básicos, mas é possível que existam casos não cobertos por ele. Pela verificação da lógica do programa e produção de casos de teste complexos, acredita-se que a implementação é robusta o suficiente. De qualquer forma, o trabalho foi relevante para o aprendizado da matéria. O tema foi complexo o suficiente para proporcionar o aprendizado, e simples o bastante para a finalização no prazo estipulado.