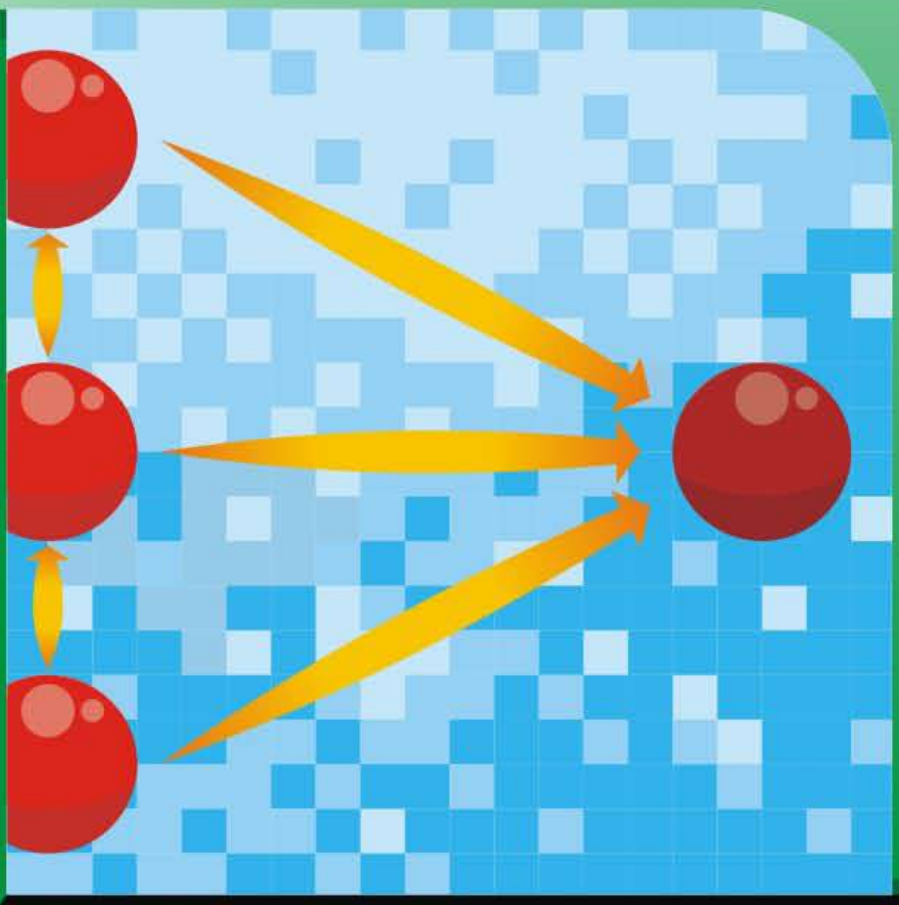
Introdução a Teoria da Computação

Wilson Galindo
Wilson Rosa
Pablo Azevedo Sampaio



FASCÍCULO 1

Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade
Vice-Reitor: Prof. Reginaldo Barros
Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho
Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire
Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira
Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena
Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial
Capa e Editoração: Alesanco Andrade Azevedo, Allyson Vila Nova, Italo Amorim e Rafael Lira
Revisão Ortográfica: Ivanda Martins
Ilustrações: Allyson Vila Nova
Coordenação de Produção: Marizete Silva Santos

Sumário

Plano da Disciplina

Carga horária: 60h

Ementa da Disciplina

- Autômatos: Finitos, a Pilha e Máquina de Turing (linearmente limitada). Linguagens Formais: Regular, Livre e Sensível ao Contexto, Estrutura de Frases. Hierarquia de Chomsky. Aplicações em compiladores. Computabilidade: modelos computacionais (funções recursivas, linguagens de programação), funções não computáveis, problema da parada, decidibilidade.

Objetivos

Gerais

- Tem como objetivo dar aos cursistas noção formal de algoritmo, computabilidade e do problema de decisão, de modo a deixá-lo consciente das limitações da Ciência da Computação. Aparelhá-los com as ferramentas de modo a habilitá-lo a melhor enfrentar a solução de problemas com o auxílio do computador. Dar subsídios para os cursistas poderem definir linguagens de programação, isto é, sua sintaxe e semântica, através do estudo das gramáticas formais.

Específicos

- Habilitar o cursista aos conhecimentos básicos de autômatos e máquina de Turing, além de um estudo geral sobre a hierarquia dos principais modelos computacionais existentes.

Cronograma de Atividades

1º Módulo Fascículo I Autômatos I	**Aula 1** • Motivação ao estudo da disciplina • Conceitos centrais da teoria de autômatos • Introdução ao estudo de autômatos • Autômatos finitos determinísticos (AFD) Atividade virtual: entrega da 1ª lista de exercícios
2º Módulo Fascículo I Autômatos II	**Aula 2** • Autômatos finitos não determinísticos (AFND) • Autômatos finitos com epsílon-transições **Aula 3** • Equivalência entre AFD e AFND Atividade virtual: entrega da 2ª lista de exercícios
3º Módulo Fascículo II Expressões Regulares	**Aula 4** • Expressões regulares **Aula 5** • Autômatos finitos e expressões regulares • Leis algébricas para expressões regulares Avaliação presencial: Prova escrita
4º Módulo Fascículo II Linguagens Regulares	**Aula 6** • Propriedades das linguagens regulares • Lema do bombeamento • Homomorfismo **Aula 7** • Equivalência e minimização de autômatos Atividade virtual: entrega da 3ª lista de exercícios

5º Módulo Fascículo III Gramáticas e Autômatos a Pilha	**Aula 8** • Gramáticas livres de contexto • Ambigüidade em gramáticas e linguagens **Aula 9** • Autômatos à Pilha Atividade virtual: entrega da 4ª lista de exercícios
6º Módulo Fascículo III Máquina de Turing	**Aula 10** • Breve histórico sobre teoria da computação • A Máquina de Turing **Aula 11** • Indecidibilidade • Problema da Parada • Problemas indecidíveis Atividade presencial 2ª VA: Prova Escrita Atividade presencial 3ª VA: Prova Escrita Atividade presencial Final: Prova Escrita

Obs.: A nota será obtida pela regra: Prova presencial: Peso 6,0. Prova virtual: 4,0.

Capítulo 1 – Autômatos Finitos Determinísticos

Olá cursista, você já conhece os conceitos básicos de programação como comandos condicionais (se, senão), contadores (para) e loop (enquanto). Nesta disciplina, você vai conhecer os fundamentos da computação, e entender melhor como funcionam máquinas simples que resultaram nos computadores atuais. O plano inicial de como funcionam esses computadores foi inicialmente projetado no papel e só posteriormente foram criados sistemas físicos dos modelos propostos.

Nessa primeira parte estudaremos um modelo simples que serve como base para outros modelos mostrados posteriormente. Lembre-se sempre que esta disciplina está ligada à resolução de problemas e a algoritmos, por isso é essencial para o curso.

O computador, antes de existir fisicamente, tinha limitações que já eram conhecidas através de modelos matemáticos que descreviam o seu funcionamento. Os físicos, então, só adequaram a tecnologia existente na época para tornar possível sua construção. Estudaremos nesse curso vários modelos para entender mais a fundo os conceitos da computação.

1.1 Conceitos Iniciais

Os computadores atuais ficam mais rápidos e poderosos a cada dia. Entretanto, essas máquinas são criadas a partir de modelos matemáticos, que foram criados no objetivo de estudar a complexidade dos algoritmos para resolver determinados tipos de problemas. Um desses modelos são os autômatos finitos. Trata-se de um reconhecedor de palavras ou *cadeia de caracteres.*

Existem vários modelos de máquinas, mas estudamos os autômatos pela sua simplicidade. Entender seu funcionamento deve ajudar bastante no aprendizado de modelos mais complexos.

Para entender o funcionamento de um autômato, tome um procedimento que você faz com certeza em casa: acender uma luz. Para isso, você normalmente deve pressionar o interruptor existente do ambiente. O problema de acender a luz pode ser modelado

por autômatos finitos e é um dos problemas mais fáceis de serem modelados.

Podemos tomar um autômato finito na forma de um *grafo* (conjunto de pontos ligados ou não por setas) para representar o problema. O interruptor armazena o estado que pode ser “ligado”, ou pressionando novamente mudamos para “desligado”. A forma gráfica é dada a seguir:

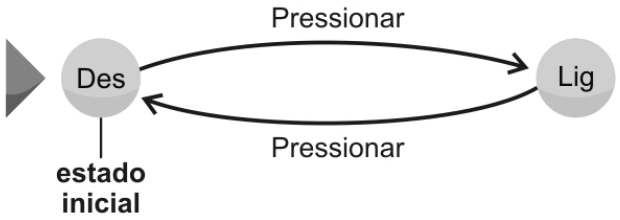


Figura 1 - Sistema de um botão liga/desliga

Todo autômato finito tem um estado inicial indicado por um triângulo no estado em questão. Podemos também usar a notação de uma seta com o nome início. Suponhamos que inicialmente o interruptor está desligado, dizemos o estado inicial é desligado. Ao pressionar novamente o botão do interruptor, o mesmo passa para o estado de ligado. E permanece até que o interruptor seja pressionado novamente o que o faz voltar para o estado desligado. Note que a ação do usuário no autômato é dada pelos arcos.

Vejamos agora um exemplo também bastante simples, tome um sistema que reconheça a palavra “amor”. Nosso sistema deve ter a capacidade de responder quando a palavra digitada for exatamente como ele estava programado para aceitar. Em outras palavras, qualquer palavra que não seja a palavra amor deverá ser rejeitada. Cada ação será a leitura de uma letra e se a letra for escolhida o sistema avança, caso contrário, o sistema pára no estado atual. Olhemos a representação gráfica do autômato:

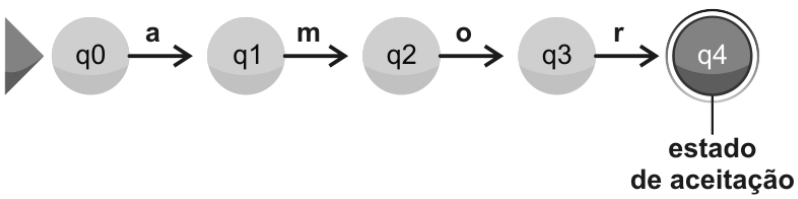
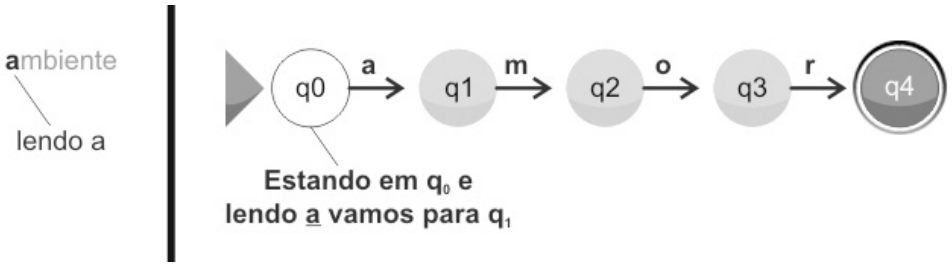


Figura 2 – Autômato para reconhecer a palavra amor

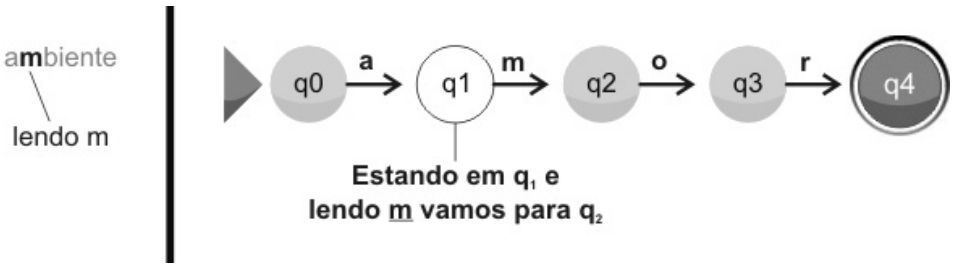
Note que basicamente a única novidade foi a presença de um círculo duplo no estado q_4, esta notação indica o estado de aceitação

ou estado final. Para ilustrar o funcionamento do autômato, vamos analisar alguns exemplos. Queremos saber como o autômato vai se comportar lendo as seguintes palavras: ambiente, amora e amor.

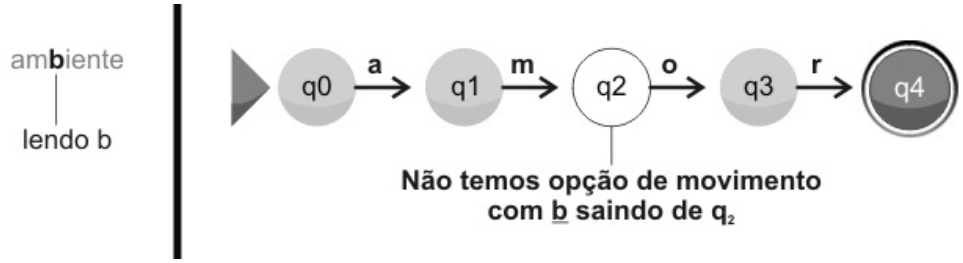
Começando com a palavra ambiente, o autômato começa do estado q_0, e só muda de estado quando lê a, que é justamente a primeira letra de nossa palavra, então temos:



Partimos para a segunda letra:

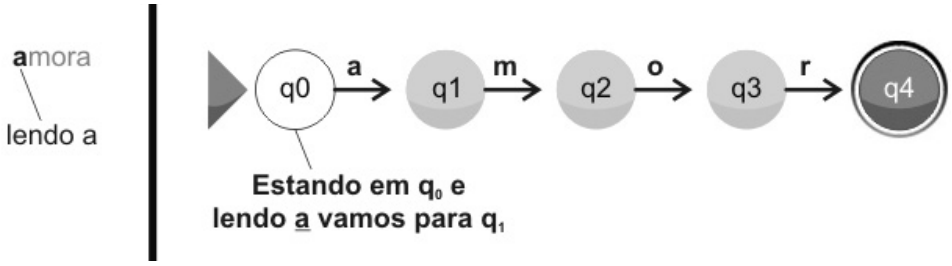


Para a letra b:

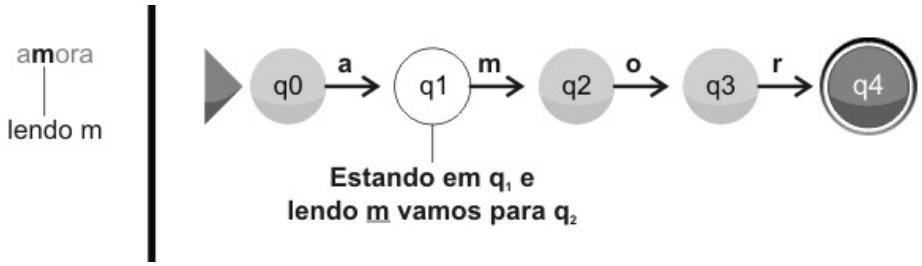


Neste caso, o autômato pára por não haver mais possibilidades de sair de q_2. Dizemos que o autômato *não reconheceu* a palavra.

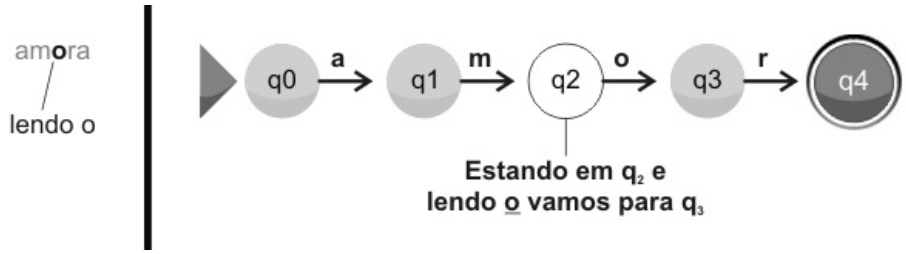
Vejamos outro exemplo com a palavra amora:



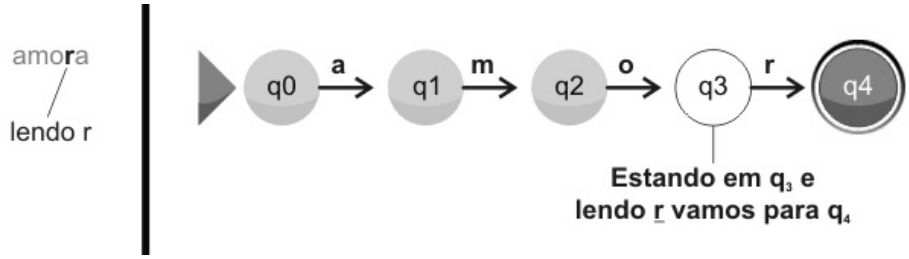
Agora lendo m:



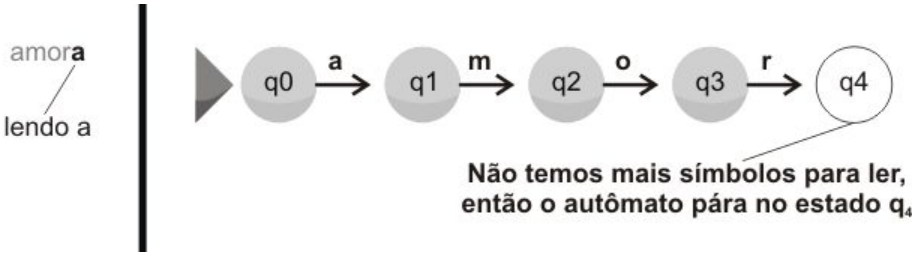
Lendo o:



Lendo r:



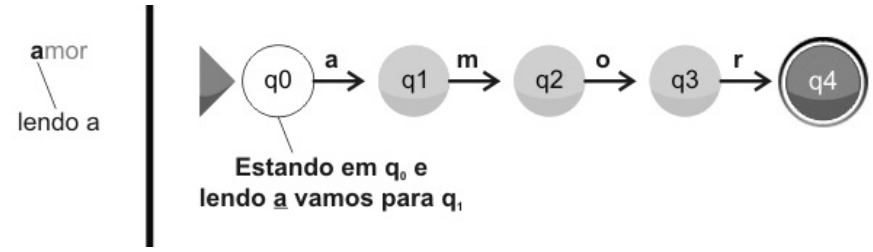
Lendo a:



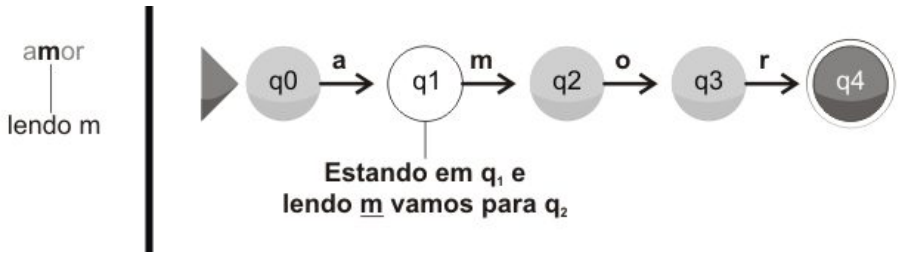
O autômato pára novamente, só que neste caso é muito interessante notar que estamos no estado de aceitação, mas mesmo assim o autômato não aceita a palavra, pois precisaríamos percorrer toda a palavra para a aceitação.

Tomemos o exemplo da palavra amor. Já temos a certeza que o autômato vai reconhecer esta palavra como foi dito a pouco, mas devemos entender exatamente porque isto acontece. Analisemos os passos a seguir:

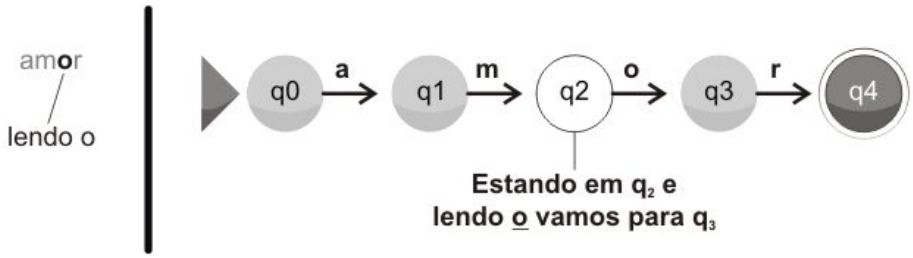
Lendo a:



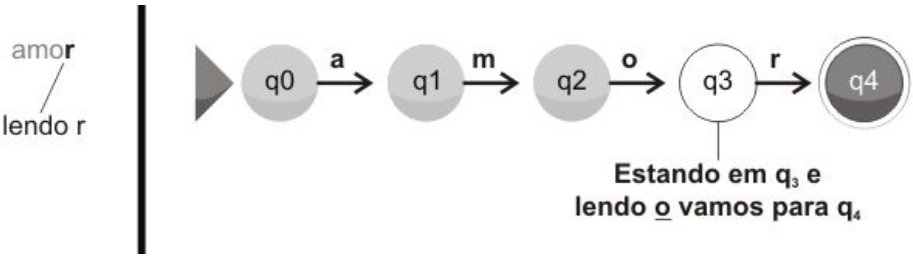
Agora lendo m:



Lendo o:



Lendo r:



Terminamos a leitura da palavra amor, percorremos todos os caracteres da palavra a agora devemos observar o autômato. Note que estamos no estado q_4, que é justamente o estado de aceitação. Se as duas situações ocorrerem: terminamos de ler a cadeia e estamos no estado de aceitação, então reconhecemos a palavra.

Você já pode aceitar que realmente o autômato finito que estamos trabalhando neste momento aceita apenas a palavra amor. E já deve ter concluído que a palavra só é reconhecida se ao término da leitura da cadeia, o autômato estiver no estado final.

Em nossos estudos trabalharemos diversos alfabetos. O último autômato utiliza o conjunto das letras do nosso alfabeto. Podemos usar como alfabeto conjuntos de maneira geral, por exemplo, o conjunto dos números inteiros ou apenas os números {0,1}.

A *linguagem* utilizada é o conjunto de cadeias que levam o autômato finito ao estado de aceitação. A linguagem do autômato anterior é o conjunto unitário {amor}.

Talvez você esteja pensando nos autômatos como simples e pouco poderosos reconhecedores de palavras, pois, no exemplo dado até agora, precisamos de um novo estado para cada letra da palavra. Pensamos logo em autômatos para reconhecer várias palavras. Mas afinal existe um número máximo de palavras que podem compor uma linguagem?

A resposta é não, acima mostramos um autômato onde a linguagem é finita. Veja abaixo um autômato capaz de reconhecer um número infinito de palavras. Curioso(a)? Veja como é simples:

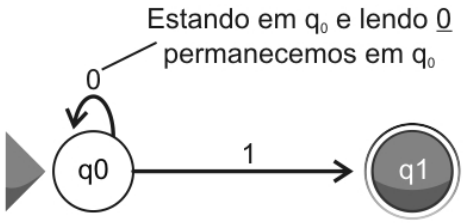


Figura 3 Autômato Finito que reconhece linguagem infinita

Mas qual a linguagem aceita por esse autômato?

Analisando com calma o autômato anterior notamos que ler qualquer quantidade de zeros repetidamente não fará com que o

autômato mude de estado. Quando lermos o primeiro 1 vamos para o estado final, se este for o último caractere da cadeia, então aceitamos a cadeia.

Tente identificar qual é a linguagem aceita por este autômato.

Podemos expressar a linguagem da seguinte forma: {todas as cadeias de nenhum ou mais zeros seguidos de um}. Exemplos de cadeias que são aceitas por esse autômato: 1, 01, 00001.

Até agora vimos alguns componentes principais dos autômatos, como estado inicial e final, o alfabeto. Também vimos que dependendo da entrada e do estado atual o autômato pode tomar uma ação. Esta ação é chamada de transição.

1.2 Definição Formal

Podemos definir um autômato finito como uma 5-tupla (uma 5-tupla guarda cinco elementos de maneira ordenada).

$$A = (Q, \sum, \delta, s_0, F), \text{ onde:}$$

- Q é o conjunto finito de estados.
- $\sum$ é o alfabeto de entrada.
- δ é a função de transição. É do tipo: $Q \times \sum \rightarrow Q$. Significa dizer que permanecendo em um estado e lendo um símbolo do alfabeto faz o autômato passar para outro estado ou mesmo ficar no mesmo.
- $s_0 \in Q$ é o estado inicial.
- $F \subseteq Q$; os elementos de F são os estados finais ou estados de aceitação. É isso mesmo que você leu, podemos ter mais de um estado final. Veremos adiante autômatos com essa característica. Note também que não há restrições em o estado inicial ser um dos estados finais.

Já estamos acostumados a trabalhar com autômatos na forma gráfica, mas vamos ver agora como passar da forma gráfica para a forma de 5-tupla e vice-versa.

Suponha o autômato para reconhecer se há um número ímpar de zeros em cadeias de zeros e uns. A forma gráfica é dada a seguir:

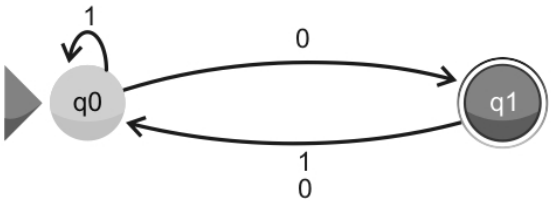


Figura 4 – Autômato que reconhece números ímpares de zeros

É difícil neste momento para você aceitar que esse autômato reconhece em qualquer cadeia, independente do tamanho, se há uma quantidade ímpar de zeros, mas veremos como fazer esse tipo de testes. Devemos preencher os cinco elementos da 5-tupla: A = (Q, ∑, δ, s_0, F). O primeiro é o Q, formado pelos estados {q_0, q_1}. Um passo importante para escrita na forma de 5-tupla é analisar a descrição informal da linguagem para identificar o alfabeto. Veja que a descrição que demos acima diz que só vão ser dados como entrada cadeias de zeros e uns, logo o alfabeto é o conjunto {0,1}. O estado inicial é q_0 e o estado de aceitação é q_1, devidamente identificados acima. Já podemos escrever a seguinte 5-tupla:

$$A = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

Vale salientar, neste momento, o motivo de q_0 não estar entre chaves: o motivo disto é que o estado inicial é sempre único.

Mas você deve estar se perguntando: quem é o delta (δ)? O delta representa a função de transição. Há uma maneira simples de representá-lo, faça uma tabela como mostra a seguir:

δ	**0**	**1**
→ **q_0**		
* **q_1**		

As linhas devem ser representadas pelos estados e as colunas pelo alfabeto. O estado inicial é marcado com uma seta e os estados finais com asteriscos.

Para preencher a tabela devemos acompanhar o autômato da seguinte forma: estado em q_0 e lendo 0 qual a ação devemos tomar? Olhando para o autômato vemos que a ação é mudar para o estado q_1. Então preenchemos com q_1 na interseção entre q_0 e 0. Depois lendo 1 estando em q_0 o autômato permanece no mesmo estado. Marcamos então q_0 na tabela. Fazemos o mesmo para o estado q_1 e obtemos o

seguinte diagrama:

δ	0	1
→ q_0	q_1	q_0
* q_1	q_0	q_0

As transições do autômato podem ser escritas de outras formas. Uma delas, bastante importante, está a seguir representando o diagrama de transição acima.

$\delta (q_0, 0) = q_1$ (lendo 0 estando em q_0, vamos para o estado q_1)

$\delta (q_0, 1) = q_0$

$\delta (q_1, 0) = q_0$

$\delta (q_1, 1) = q_0$

Bem, como você já tem afinidade suficiente com autômatos, usaremos uma regra usada em autômatos finitos determinísticos:

> Cada estado de um autômato finito determinístico (AFD) deve ter transição com todos os símbolos do alfabeto. Em outras palavras, o AFD não pode parar por falta de caminhos, e sim porque não chegou ao estado de aceitação.

Mais à frente, nos exercícios resolvidos, vamos entender como essa condição funciona na prática.

Aprenda Praticando

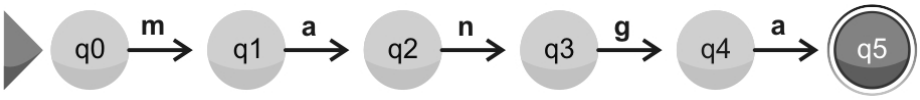
Este é o momento de colocarmos em prática nossos conhecimentos, acompanhe os exercícios.

1) Crie os autômatos finitos determinísticos referentes às seguintes linguagens:

a. L = {manga}

Resposta:

Em princípio, é trivial criar um AFD para reconhecer linguagens finitas, basta criar um estado para cada caractere lido. Então obtemos facilmente o autômato:



Mas a forma gráfica do autômato não é uma representação completa do autômato, por isso normalmente nós utilizamos a definição formal do autômato que é dada na forma de uma 5-tupla.

Sabemos que um autômato finito é representado por:

A = (Q, ∑, δ, s_0, F), onde:

Q é facilmente identificável, pois basta citar os estados presentes no autômato acima: {q_0, q_1, q_2, q_3, q_4, q_5}.

Quanto ao alfabeto (∑), está implícito que é o alfabeto que usamos {a, b, c, ..., z}, mas podemos simplesmente utilizar a forma reduzida: {a, g, m, n} pois são as únicas letras utilizadas pelo autômato. A diferença é que no primeiro caso permitíamos, por exemplo, que q_0 lesse a letra j, mas no segundo caso não, pois a linguagem tem apenas as quatro letras dadas. A escolha do alfabeto varia, mas normalmente o alfabeto é dado.

O estado inicial (s_0) está devidamente identificado: q_0

O estado final (F) também está identificado: {q_5}

Quando ao delta (δ) é obtido pela construção da tabela como ensinado anteriormente.

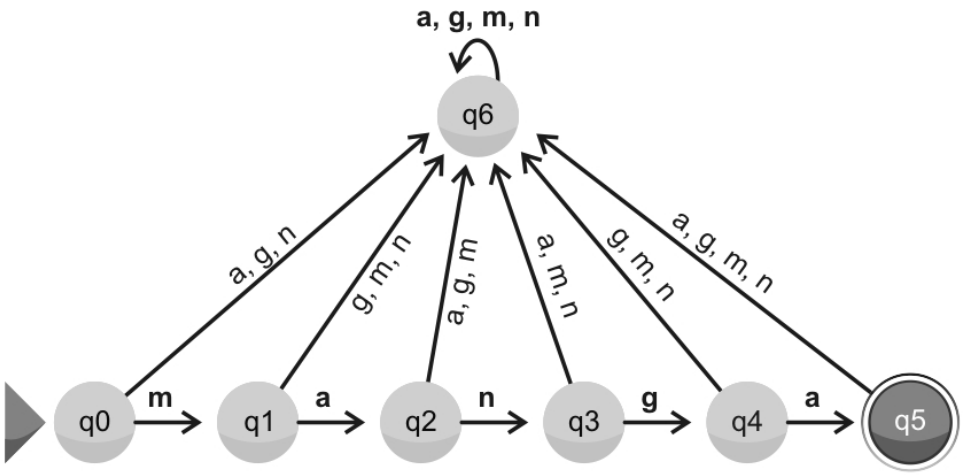
Já temos todos os componentes necessários para montar nosso autômato:

A = ({q_0, q_1, q_2, q_3, q_4, q_5}, {a, g, m, n}, δ, q_0, {q_5}), onde δ é dado por:

δ	a	g	m	n
q_0	-	-	q_1	-
q_1	q_2	-	-	-
q_2	-	-	-	q_3
q_3	-	q_4	-	-
q_4	q_5	-	-	-
q_5	-	-	-	-

Mas anteriormente vimos que o autômato finito determinístico

deve ter transição para todo símbolo do alfabeto, por exemplo, q_0 não tem transição para {g, m, n}. Neste caso existe uma estratégia: a adição de um estado especial, chamado de **estado sugadouro**. É um estado de não-aceitação e recebe as transições vazias. A forma gráfica é mostrada abaixo, mas lembre-se que não é obrigatória essa representação.



Note que quando o autômato chega em q_6, independente da entrada, o autômato não sai mais do estado. Agora temos um novo diagrama de transição totalmente preenchido e devemos adicionar um estado na definição formal:

A = ({q_0, q_1, q_2, q_3, q_4, q_5, q_6}, {a, g, m, n}, δ, q_0, {q_5}), onde δ é dado por:

δ	a	g	m	n
q_0	q_6	q_6	q_1	q_6
q_1	q_2	q_6	q_6	q_6
q_2	q_6	q_6	q_6	q_3
q_3	q_6	q_4	q_6	q_6
q_4	q_5	q_6	q_6	q_6
q_5	q_6	q_6	q_6	q_6
q_6	q_6	q_6	q_6	q_6

b. Sob o alfabeto {0,1} o conjunto de todas as cadeias que terminam com 11.

Resposta:

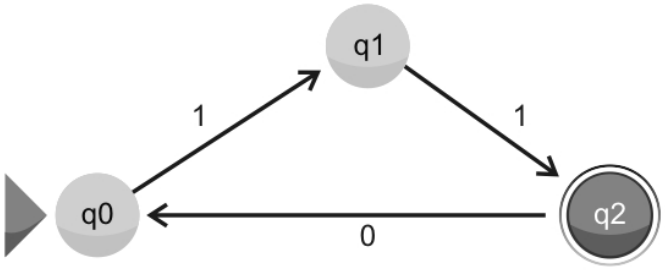
Note que nos deparamos com um caso de linguagem infinita. O que não quer dizer que seja complexo resolver esse tipo de problema.

Mostraremos de forma construtiva uma das estratégias para criar autômatos finitos, existem infinitas possibilidades de se resolver um problema de autômatos, cada pessoa resolve da maneira que acha mais fácil, mas aqui apresentarei algumas que considero básicas.

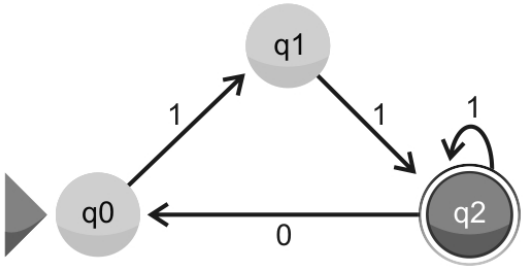
Assim como no exemplo anterior, normalmente é mais fácil criar um autômato primeiro na forma gráfica do que preencher diretamente a tabela de transição, por isso usaremos esta técnica.

Note que para ser aceito, o autômato deve ter pelo menos a cadeia 11. Então faremos o autômato simples que reconhece essa cadeia, como mostrado abaixo:

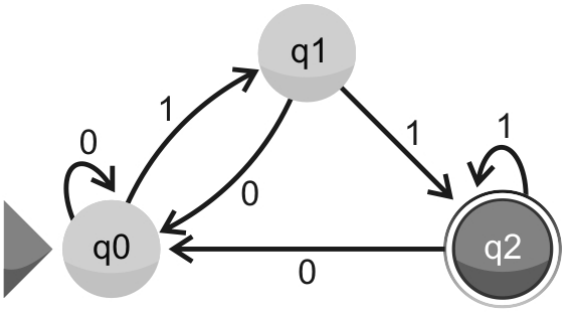
Analisemos agora a questão de 110, ou seja, se depois de ler 11 o próximo for zero. Neste caso devemos sair do estado final e ir para o estado inicial, pois para chegar no estado final, deveremos passar por 11 novamente. Observe a seguir:



Nosso autômato já computa cadeias como: 110, 1101, 11011, 11011011. E as duas últimas cadeias levam o autômato ao estado de aceitação, estando de acordo com a linguagem. Vejamos agora o caso de 111, 1111, 111111, pois todos eles devem ser aceitos. Isso quer dizer que quando o autômato está em q_2 e começa a ler uns, continua aceitando as cadeias. Obtemos então o seguinte autômato:



Note que se q_1 ler 0, devemos ir para q_0, que é o caso das cadeias {10, 1011, 11010}. Observe que já estão quase todas as transições completas, só falta a ação de q_0 quando lê zero, e facilmente notamos que o autômato deverá ficar em q_0. Temos agora o seguinte autômato:



Ainda falta responder uma pergunta essencial e intrigante: Como garantir que o autômato acima está realmente correto?

Podemos traçar um paralelo com programação. Nem sempre é simples provar que um autômato está correto, assim como também não é simples provar que um programa que você fez está correto. O que você pode fazer com os autômatos é testar para tentar encontrar uma falha, assim como você provavelmente faz com os programas que você desenvolve.

No caso do autômato, você deve testar várias palavras. Se você conseguir achar alguma palavra que ao autômato aceite quando deveria rejeitar, ou que ele rejeite quando deveria aceitar, você conseguiu provar por contra-exemplo que o autômato está errado. Porém, após fazer vários testes, se o autômato só der resultados corretos, então você pode assumir que ele está certo. Não é uma prova definitiva de que o autômato está correto, mas se você escolher bem os testes, você consegue ter segurança de que ele reconhece a linguagem que você queria.

Parece um método muito trabalhoso, mas na prática é muito útil. Quanto mais você pratica esta técnica, mais rapidamente você estará apto a identificar problemas em autômatos. Logo, treine o máximo possível para se acostumar tanto com a construção como também com esse processo de verificação dos autômatos.

Voltando para o autômato, agora só precisamos escrevê-lo como uma 5-tupla:

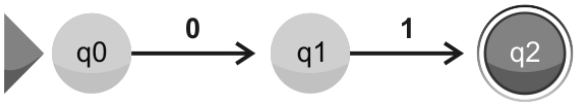
A = ($\{q_0, q_1, q_2\}$, {0, 1}, δ, q_0, $\{q_2\}$), onde delta é dado por:

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_2

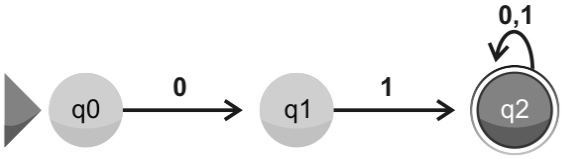
c. O conjunto de cadeias de 0's e 1's que tenham 01 como subcadeia.

Resposta:

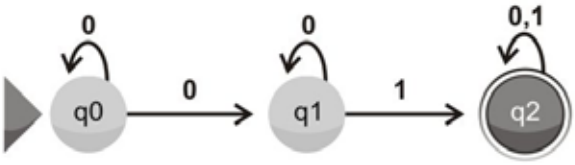
Seguindo o mesmo princípio do exercício anterior, faremos um autômato para reconhecer a cadeia 01. Você não deve ter dificuldade em entender o autômato mostrado abaixo, se tiver problemas, é melhor voltar e reler a aula e acompanhar novamente os exercícios resolvidos anteriores a este.



Este exemplo parece ser difícil, mas é facílimo. Note que uma vez que lemos 01 na cadeia, não nos interessa saber os caracteres adiante. Por exemplo: 011100110101, uma vez que o autômato começa com 01, a cadeia deve ser aceita. Isso nos leva a aceitar que uma vez que é atingido o estado de aceitação, não deveremos mais sair dele. Obtemos o seguinte:



Se estivermos em q_1 e lermos 0, o que deve acontecer? É o caso da cadeia 000001, como 01 está no final da cadeia, esta deve ser aceita, logo é como se depois que chegamos em q_1, lemos zero até que seja lido o caractere 1 ou acabar a cadeia. Então estando em q_1 e lendo 0, ficamos em q_1. Só falta completar a transição para q_0 lendo 1, neste caso não poderemos sair de q_0, pois ficaremos na esperança de aparecer um 0 para completar a cadeia 01 e aceitar a cadeia.



Está completo, só precisamos escrevê-lo como uma 5-tupla.

A = ($\{q_0, q_1, q_2\}$, {0, 1}, δ, q_0, $\{q_2\}$), onde delta é dado por:

δ	**0**	**1**
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

d. O conjunto de cadeias de 0's e 1's que não contenham 1's. Obs.: a cadeia vazia deve ser aceita.

Resposta:

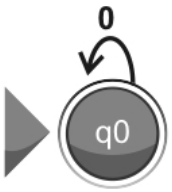
Essa questão nos faz pensar sobre duas indagações:

O autômato pode aceitar uma cadeia sem elementos?

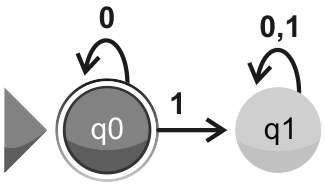
Um estado pode ser final e inicial ao mesmo tempo?

Pense nas respostas enquanto mostro o resultado dessa questão.

A linguagem é a seguinte: leia vários zeros e aceite-os. Se algum 1 for lido, o autômato sai do estado de aceitação. Seguindo a mesma técnica de construção utilizada até agora obtemos:



Você está presenciando um evento completamente normal, o estado q_0 é estado inicial e estado final ao mesmo tempo, isso significa que o caractere vazio, ou seja, uma cadeia de nenhum caractere é aceita pelo autômato. Só precisamos de uma ação agora, um estado sugadouro que uma vez lido 1, não sai mais.



Só falta agora escrever a 5-tupla:

A = ($\{q_0, q_1\}$, {0, 1}, δ, q_0, $\{q_1\}$), onde delta é dado por:

δ	0	1
q_0	q_0	q_1
q_1	q_1	q_a
q_2	q_2	q_2

Exercícios Propostos

Agora é com você, faça o possível para responder todos os exercícios pois lhe ajudará imensamente não só na disciplina, mas no decorrer do curso.

1. Forneça os AFDs que aceitam as seguintes linguagens sob o alfabeto {0,1}:

a) O conjunto de todas as cadeias que terminam em 00.

b) O conjunto de todas as cadeias com três 0's consecutivos (não necessariamente no final).

c) O conjunto de cadeias que têm 011 como subcadeia.

d) O conjunto de cadeias com número par de 1's.

e) O conjunto de cadeias com número ímpar de 0's.

f) O conjunto de cadeias tais que cada bloco de cinco símbolos consecutivos contém pelo menos dois zeros.

g) O conjunto de cadeias cujo quinto símbolo a partir da extremidade direita é 1.

h) O conjunto de cadeias que começam ou terminam (ou ambos) com 01.

i) O conjunto de cadeias tais que o número de 0's é divisível por 5.

j) O conjunto de cadeias tais que o número de 1's é divisível por 3.

Você estudou, no primeiro capítulo, que os Autômatos Finitos Determinísticos (AFD) são os modelos mais simples estudados no curso. Você conhecerá agora um modelo similar de autômatos com algumas peculiaridades que tornam sua análise interessante.

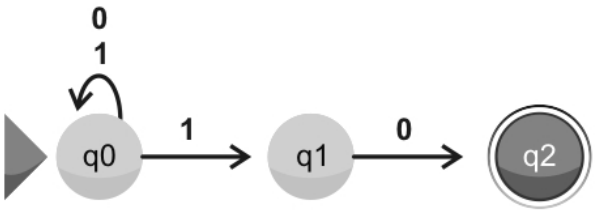
Capítulo 2 – Autômatos Finitos Não-Determinísticos

Pense agora em um autômato que pode estar em vários estados ao mesmo tempo ou ainda que não precise ler um caractere para mudar de estado. Estas são as principais características de um autômato finito não-determinístico (AFND). Em outras palavras, um autômato lendo uma letra pode ir para um estado ou vários estados ao mesmo tempo. Também pode estar em um estado e passar para outro sem necessariamente ler uma letra.

Estudaremos as características dos AFND e posteriormente compararemos com os AFD para verificar se as diferenças do AFND o tornam mais poderoso.

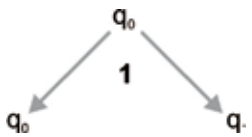
2.1 Uma Visão Informal

Vejamos agora a forma gráfica de um AFND:

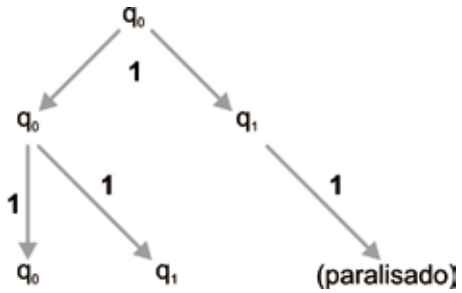


A única diferença desse autômato para o que estamos acostumados a trabalhar é que q_0 tem duas transições lendo 1. Este estado lendo um vai para o conjunto $\{q_0, q_1\}$ e fica nos dois estados ao mesmo tempo.

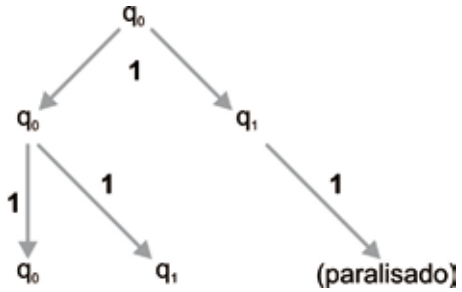
Veremos agora o comportamento do autômato com algumas entradas: Suponha a cadeia 110, começamos em q_0 lendo 1 e indo para os estados $\{q_0,q_1\}$. O diagrama a seguir mostra o resultado da ação:



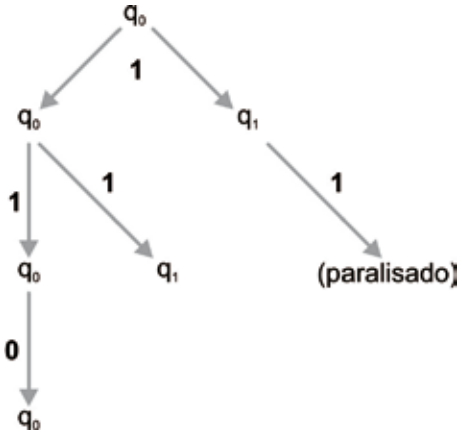
Neste momento, o autômato está no estado q_0 e q_1. A partir de agora, toda vez que o autômato ler uma entrada deve-se acompanhar o andamento de todos os estados atuais. Lendo o segundo caractere temos:



O estado q_1 não tem transição lendo 1 então simplesmente o caminho de q_1 é abandonado. Note também o estado q_0 que lendo 1 novamente bifurcou e estamos novamente em dois estados.



Vamos então para o terceiro caractere da cadeia. Neste caso, temos duas situações: calculando primeiro q_0 lendo 0 temos:

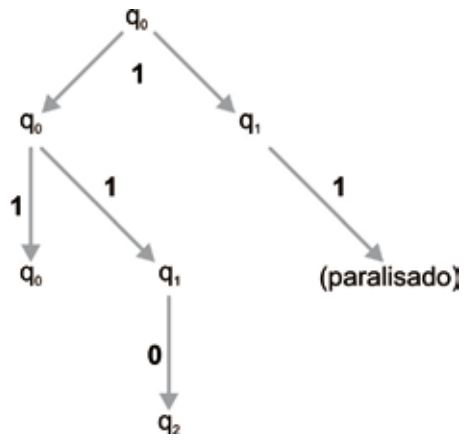


A partir de q_1, lendo 0 vamos para o estado de aceitação q_2. Igualmente ao estudado em AFD, como terminamos de ler a cadeia e estamos num estado de aceitação aceitamos esta cadeia.

Atenção!

Fique atento a um detalhe importantíssimo dos AFND dito a seguir.

Suponha que tivéssemos escolhido inicialmente q_1 para ler o caractere 0. Temos então o seguinte resultado:



Terminamos de ler a cadeia e chegamos a um estado de aceitação. É uma característica dos AFND **não** terminarem os outros caminhos quando chegam ao estado final. Em nosso exemplo não precisamos verificar o resultado de q_0 lendo 0, simplesmente aceitamos a cadeia.

2.2 Definição Formal

Um autômato finito não-determinístico (AFND) é uma 5-tupla $(Q,\sum,\delta,q_0,F)$ onde:

- Q é o conjunto finito de estados.
- $\sum$ é o alfabeto de entrada.
- δ é a função de transição. É do tipo: $Q \times \sum \rightarrow 2^Q$. Significa dizer que estando em um estado e lendo um símbolo do alfabeto faz o autômato passar para um conjunto de estados do autômato.
- $s_0 \in Q$ é o estado inicial.
- $F \subseteq Q$; os elementos de F são os estados finais ou estados de aceitação.

Você já deve ter percebido que Q, δ, q_0, e F têm o mesmo significado para os AFD. A única diferença é no δ, que agora pode acessar zero, um ou mais estados.

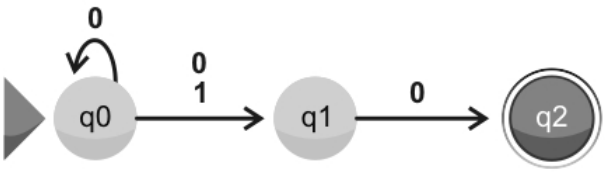
O autômato do exemplo anterior pode ser escrito assim:

$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

Onde δ é dado pela tabela:

δ	0	1
→ q_0	$\{q_0\}$	$\{q_0\ q_1\}$
q_1	$\{q_2\}$	$\varnothing$
* q_2	$\varnothing$	$\varnothing$

Na forma gráfica:



2.3 A função de transição estendida

Já sabemos que a junção de todas as cadeias que um autômato reconhece é chamada de linguagem do autômato, porém até aqui só fizemos comentários sobre ela e ainda não definimos formalmente. A notação utilizada para definir linguagens é a função de transição estendida. Supondo uma função de transição δ, sua função estendida é $\hat{\delta}$. Esta função toma como entrada um estado e um cadeia e retorna um estado, só que esse estado é especial, é usado para descrever um caminho percorrido pelo autômato a partir do estado inicial. Temos uma função que aceita como entrada uma cadeia e não só um caractere como a função δ. Veremos como funciona a função $\hat{\delta}$.

Comecemos por uma função δ dada abaixo:

δ	0	1
→ q_0	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_0\}$	$\{q_1\}$

Agora tome a cadeia 1001. Podemos escrever o comportamento da função δ lendo a cadeia dada facilmente: Primeiro, o autômato está em q_0, lê um e vai para q_1 como é descrito a seguir: $\delta(q_0,1)=\{q_1\}$, Depois, estando em q_1 lendo zero volta para o estado q_0: $\delta(q_1,0) = \{q_0\}$. Em seguida, facilmente deduzimos que $\delta(q_0,0) = \{q_0\}$ e $\delta(q_0,1) = \{q_1\}$. Preste atenção no caminho que o autômato percorreu mostrado a seguir:

$$\delta(q_0,1) = \{q_1\} \longrightarrow \delta(q_1,0) = \{q_0\} \longrightarrow \delta(q_0,0) = \{q_0\} \longrightarrow \delta(q_0,1) = \{q_1\}$$

Tomando ainda o exemplo anterior, suponha que ainda não começamos a ler nenhum caractere e, portanto, permanecemos no estado inicial q_0. Na função de transição estendida, podemos usar ε, que representa a cadeia vazia (ou seja, a cadeia que não possui nenhum símbolo) para indicar que o autômato ainda não leu nenhum símbolo e que, por isso, ainda não saiu de q_0:

$\hat{\delta}(q_0, \varepsilon) = \{q_0\}$, onde q_0 é o caminho que o autômato percorreu até agora. Quando lemos o primeiro caractere, temos:

$\hat{\delta}(q_0,1) = \delta(q_0,1)$, ou seja, o único caminho que percorremos até agora foi a transição de q_0 lendo 1, só que podemos substituir q_0 usando a expressão anterior, então obtemos:

$$\hat{\delta}(q_0,1) = \delta(q_0,1) = \{q_1\}$$

$$\hat{\delta}(q_0,1) = \delta(\hat{\delta}(q_0, \varepsilon),1) = \{q_1\}$$

Olhando para o exemplo anterior, escrevemos q_0 como o caminho que o autômato percorreu até a partir q_0 para ler o primeiro caractere, que foi o de ε.

Aparentemente parece um conceito difícil, mas não é. Basta apenas que você pratique até entender o funcionamento da estrutura. Vamos continuar acompanhando o caminho do autômato que estamos trabalhando lendo a mesma cadeia. O autômato está em q_1 e o próximo caractere é zero. Note também que estamos lendo a cadeia 10, então usando a função $\hat{\delta}$ temos:

$$\hat{\delta}(q_0,10) = \delta(q_1,0) = \{q_0\}$$

Mas $q_1 = \hat{\delta}(q_0,1)$

Então:

$$\hat{\delta}(q_0,10) = \delta(\hat{\delta}(q_0,1),0) = \{q_0\}$$

o caminho que o autômato faz para ler 10 partindo de q_0.

Estado resultante do caminho de q_0 até ler 1.

A equação anterior diz que o caminho que o autômato fez para ler 10 a partir de q_0 é igual ao resultado do estado de q_0 até o caractere anterior aplicado na função δ com o caractere atual. Para praticar mais, vamos responder a seguinte pergunta: quem é $\hat{\delta}(q_0, 1001)$? É o mesmo que perguntar para que estado o autômato termina após ler a entrada 1001? Deveremos acompanhar a seqüência de caracteres da cadeia:

Sabemos que: $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$

Começamos por: $\hat{\delta}(q_0,1) = \delta(\hat{\delta}(q_0, \varepsilon),1) = \delta(q_0,1) = \{q_1\}$

Em seguida temos: $\hat{\delta}(q_0, 10) = \delta(\hat{\delta}(q_0,1),0)$, Só que: $\hat{\delta}(q_0,1) = \{q_1\}$ pela equação anterior.

Logo, $\hat{\delta}(q_0, 10) = \delta(q_1,0) = \{q_0\}$

$\hat{\delta}(q_0, 100) = \delta(\hat{\delta}(q_0,10),0)$, mas como: $\hat{\delta}(q_0, 10) = \{q_0\}$,

$\hat{\delta}(q_0, 100) = \delta(q_0,0) = \{q_0\}$

$\hat{\delta}(q_0, 1001) = \delta(\hat{\delta}(q_0,100),1)$, da linha anterior: $\hat{\delta}(q_0, 100) = \{q_0\}$

$\hat{\delta}(q_0, 1001) = \delta(q_1,1) = \{q_1\}$

Veja que encontramos uma maneira formal de descrever o caminho de um autômato. Lembre-se que isso serve para facilitar demonstrações, que não trabalharemos neste momento.

Já podemos tentar definir formalmente a função $\hat{\delta}$. A primeira idéia seria defini-la assim:

Base: $\hat{\delta}(q, \varepsilon) = \{q\}$

Indução: $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$

Onde w = xa, ou seja, a é o último caractere de w, e x é o restante da cadeia.

Porém, a definição anterior falha na indução porque $\hat{\delta}(q,x)$ retorna um conjunto de estados enquanto δ recebe um só estado. Temos que adaptar a definição para dizer que deve ser feita a união dos resultados para cada estado do conjunto que $\hat{\delta}(q,x)$ retorna. A definição correta fica um pouco mais confusa, mas segue a mesma idéia da anterior:

Base: $\hat{\delta}(q, \varepsilon) = \{q\}$

Indução: $\hat{\delta}(q, xa) := \bigcup_{p \in S} \delta(p, a)$, onde S= $\hat{\delta}(q, x)$

A indução agora diz que, uma vez calculado o conjunto S = $\hat{\delta}(q, x)$, podemos pegar cada estado p do conjunto S e aplicar $\delta(p,a)$, fazendo a união de todos os resultados obtidos (o U grande é análogo ao Σ usado para descrever somatórios, porém ele representa várias operações de união ao invés de várias somas).

2.4 Relação entre AFD e AFND

Pelo assunto visto temos a sensação de que autômatos finitos não-determinísticos são mais poderosos que autômatos finitos determinísticos, pois os primeiros podem estar em mais de um estado ao mesmo tempo. Na verdade, dizer que um modelo é mais poderoso que outro significa afirmar a existência de alguma linguagem que os AFND reconhecem e que nenhum AFD pode reconhecer. Podemos tentar várias estratégias, mas queremos mostrar a equivalência entre os dois modelos. Para isso, criaremos um algoritmo para você transformar qualquer AFND em um AFD e ambos reconhecem a mesma linguagem. Tome como exemplo o seguinte AFND:

$A=(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

δ	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$* q_2$	$\varnothing$	$\varnothing$

Note que este autômato apresenta transição para zero, um ou mais estados, apresentando as características que distinguem os autômatos finitos não-determinísticos dos determinísticos. Cada transição, portanto, leva de 1 estado para um conjunto de estados, enquanto em um AFD cada a transição deve levar de 1 estado para 1 estado. Como fazer, então, para converter a função que leva em um conjunto para uma que leva em um único estado?

A idéia básica da conversão é tratar cada *conjunto* de estados do AFND como se fosse um *único* estado do AFD. É como se os rótulos dos estados do AFD que vamos criar fossem formados unindo os rótulos dos estados do autômato original.

Antes de analisar toda a descrição do autômato, vamos criar a nova função δ, que será a função de transição do AFD criado a partir do AFND dado. Pode parecer que a nova função de transição leva de um conjunto de estados para um conjunto de estados, mas lembre-se que, no AFD, cada conjunto será tratado como um único estado. Portanto, para o AFD, a função leva de 1 estado para 1 estado, como esperado.

Nós começamos criando a função δ do AFD quase que diretamente a partir da função do AFND, mas com as seguintes alterações:

1. adicionamos um estado vazio Ø;

2. tratamos cada estado individual como um conjunto unitário (por exemplo, q_0 virou $\{q_0\}$);

3. para cada conjunto distinto dos anteriores que aparece como destino de alguma transição, nós criamos uma nova linha na tabela (no exemplo, foi o caso do estado $\{q_0, q_1\}$ apenas).

δ	0	1
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
* $\{q_2\}$	Ø	Ø
$\{q_0, q_1\}$		

O próximo passo requer atenção e treinamento, pois você vai se questionar: Estando no estado $\{q_0, q_1\}$ ou seja q_0 e q_1 ao mesmo tempo e lendo o caractere 0, quais os caminhos seguidos pelo autômato? A partir de q_0 lendo 0 acessamos o conjunto de estados $\{q_0, q_1\}$ e a partir de q_1 acessamos Ø, então tomamos a união $\{q_0, q_1\} \cup Ø = \{q_0, q_1\}$. Preenchemos a próxima célula vazia, estando em $\{q_0, q_1\}$ lendo 1 o que acontece? Por q_0 acessa $\{q_0\}$ e por q_1 acessa $\{q_2\}$ logo a união $\{q_0\} \cup \{q_2\} = \{q_0, q_2\}$. Temos então a nova linha preenchida:

δ	0	1
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
* $\{q_2\}$	Ø	Ø
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Temos um novo problema: não sabemos o comportamento do autômato a partir de $\{q_0, q_2\}$ logo deveremos adicioná-lo como um estado e estudar o comportamento do autômato lendo os símbolos do alfabeto. Lendo 0 q_0 vai para $\{q_0, q_1\}$ e q_2 lendo o mesmo símbolo vai para Ø e lendo 1 q_0 continua no mesmo estado e lendo 1 q_2 vai para Ø logo obtemos a união $\{q_0\}$.

δ	**0**	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
* $\{q_2\}$	Ø	Ø
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Poderíamos calcular a transição para todas as outras combinações de estados, mas esses estados não são acessíveis a partir do estado inicial, por isso podem ser ignorados. O que precisamos fazer é completar a tabela da seguinte forma: como q_2 é estado de aceitação, então todos os conjuntos que contêm esse estado também são estados de aceitação. Temos, então o seguinte diagrama:

δ	**0**	**1**
Ø	Ø	Ø
→ $\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_1\}$	Ø	$\{q_2\}$
* $\{q_2\}$	Ø	Ø
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
* $\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Para facilitar nossa compreensão devemos renomear os estados para melhor visualização:

δ	**0**	**1**
A	A	A
→ B	E	B
C	A	D
* D	A	A
E	E	F
* F	E	B

Analisaremos o diagrama de transição da figura anterior: o estado C não é acessível de nenhum outro estado, ou seja, a partir do estado inicial nunca chegaremos a esse estado e vamos eliminá-lo.

δ	0	1
A	A	q_1
→ B	E	q_a
* D	A	q_2
E	E	F
* F	E	B

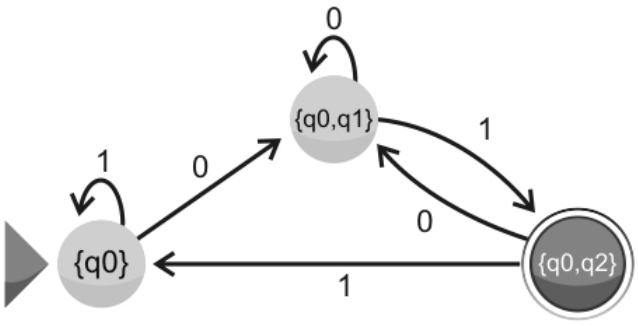
Só que o estado D não é mais acessível do estado inicial pois nenhum estado apresenta transição para o estado em questão. Eliminamos D também. Note que quando eliminarmos a linha referente ao estado D, nenhum outro estado acessará o estado A senão ele mesmo. Como não poderemos acessar A partindo do estado inicial este estado será eliminado e, finalmente temos a resposta:

δ	0	1
→ B	E	B
E	E	F
* F	E	B

Para comparar o AFND com nosso AFD devemos renomear os estados novamente:

δ	0	1
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
* $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Temos assim o modo gráfico do autômato:



Já podemos descrever completamente um AFD B que é equivalente ao AFND A descrito anteriormente:

B=({{q_0}, {q_0,q_1}, {q_0, q_2}}, {0, 1}, δ, q_0, {q_0, q_2}), onde δ é dado por:

δ	0	1
→ {q_0}	{q_0 ,q_1}	{q_0}
{q_0 ,q_1}	{q_0 ,q_1}	{q_0 ,q_2}
* {q_0 ,q_2}	{q_0 ,q_1}	{q_0 }

Vamos explicitar um algoritmo de conversão de AFND para AFD:

1. Tome o diagrama de transição do AFND e crie para cada conjunto de estados, um novo estado onde o rótulo é o próprio conjunto.

2. Complete as transições de cada novo estado acompanhando o comportamento de cada estado individualmente e tomando a união dos estados acessados.

3. Adicione como estados finais aqueles estados que tem como rótulo um estado final.

4. Elimine os estados nunca acessados a partir do estado inicial.

5. Complete o AFD da seguinte forma:

5.1. Q do AFD são os estados resultantes dos passos até este ponto.

5.2. O alfabeto não é alterado

5.3. A construção do δ já foi mencionada.

5.4. O estado inicial não se altera

5.5. Os estados finais são a união de todos os estados finais do δ resultante.

Acabamos de demonstrar de maneira simples como obter um AFD a partir de um AFND, mas para mostrar a equivalência devemos mostrar um AFND pode ser transformado em um AFD e vice-versa. Lembre-se que todo AFD é um AFND, mas um AFND não é diretamente um AFD. Mostramos como passar de AFND para AFD e a conversão de AFD para AFND é direta. Por isso, podemos afirmar que são equivalentes.

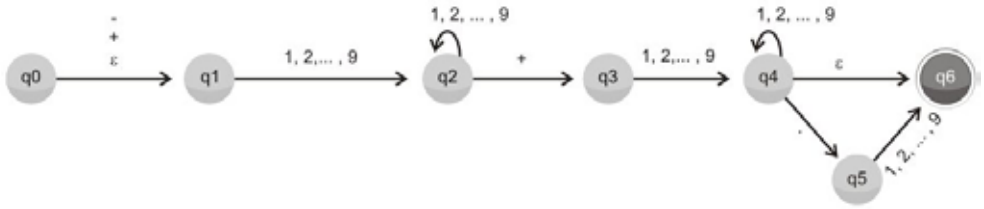
Qual a vantagem então de se trabalhar com AFND's? Em geral, o AFND pode encontrar a resposta mais rápido que o AFD e seu conjunto de estados normalmente é menor que o do AFD equivalente.

2.5 Um novo tipo de transição (ε)

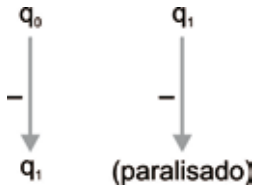
E se pudéssemos mudar de estado sem precisar ler nenhum caractere? Você não leu errado, há uma maneira onde, no momento que o autômato chega em um estado, ele passa imediatamente para outros estados sem ter lido mais nenhum caractere. Podemos fazer este tipo de transição com o uso de um identificador, o símbolo ε. Um autômato com essa propriedade é chamado de AFND com ε-transições ou ε-AFND.

O ε é utilizado para formar transições espontâneas para o autômato. Também pode ser usado λ para representar este tipo de transição. Vamos analisar um ε-AFND e entender seu funcionamento.

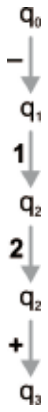
Suponha um autômato que reconhece a soma de um inteiro positivo ou negativo com um decimal positivo.



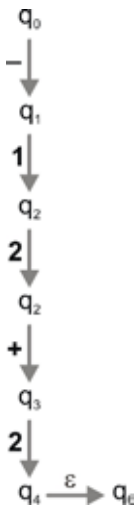
Vamos acompanhar agora o funcionamento deste autômato com a expressão “-12+2.6”. Começamos com estado q_0: o número pode ser iniciado por +, -, ou nenhum caractere.



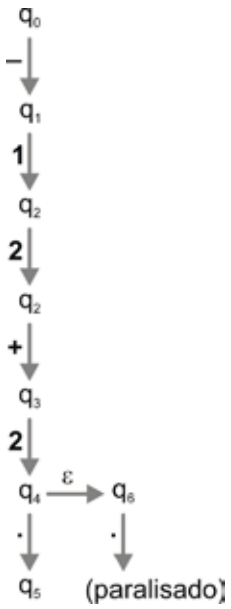
Note que, inicialmente, o autômato já está nos estados q_0 e q_1 ao mesmo tempo por causa da transição ε entre esses estados. A transição espontânea faz com que o autômato avance nas transições mesmo sem ter lido qualquer caractere da cadeia. Começamos a ler então o primeiro símbolo (-). Seguindo o caminho de q_0 lendo menos (-) faz o autômato passar para o estado q_1, mas de q_1, que é o segundo caminho, não há transição então este percurso é abandonado. Os próximos são lidos e o autômato se comporta como um simples AFD lendo 12+ e obtemos o seguinte resultado:



Estando em q_3 o autômato lê o próximo caractere dois (2) e imediatamente o autômato vai para o estado q_4 mas tratamos aqui de um autômato com ε transições. O ε mais à direita indica que chegando ao estado q_4 imediatamente o autômato passa também para o estado q_6, e por se tratar de um estado de aceitação se a cadeia terminasse neste momento seria aceita. Vejamos como fica a descrição dos caminhos do autômato até o momento:



Lembre-se que o autômato está no estado q_4 e q_6. Lendo o próximo caractere (.), o caminho de q_6 é abandonado, enquanto, no outro caminho, seguimos de q_4 para o estado q_5, como é mostrado a seguir:



Em seguida, o autômato lê o seis (6) e vai para q_6, e como é estado de aceitação e terminamos de ler a cadeia aceitamos a cadeia. Tente repetir o funcionamento do autômato com esse número e crie novos números para observar o comportamento do autômato com o uso das transições ε.

Aprenda Praticando

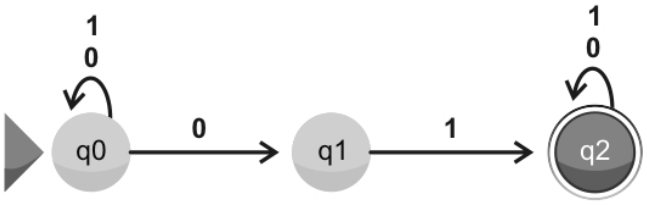
Nesta seção mostraremos passo a passo como resolver na prática os problemas aprendidos nas seções anteriores. Atente aos exercícios e se possível resolva-os novamente sem a ajuda da apostila. É essencial que você só avance aos exercícios seguintes quando entender os resolvidos. Lembre-se que você tem ajuda dos seus colegas de turma, tutores, professores, livros entre outros para esclarecer suas dúvidas.

1. Crie o autômato finito não-determinístico que reconheça a seguinte linguagem: L = O conjunto de todos os números binários {0,1} que tenham 01 como subcadeia.

Resposta:

Primeiro temos que criar um autômato que reconheça a cadeia 01. Obtido facilmente e é mostrado abaixo:

Só precisamos verificar as condições: a subcadeia pode estar após a leitura de alguns caracteres, logo precisamos adicionar transições iniciais. Também 01 pode estar antes do final da cadeia por isso precisamos adicionar transições no final. Uma vez lido 01 o autômato não sai mais do estado de aceitação. Temos como resultado:

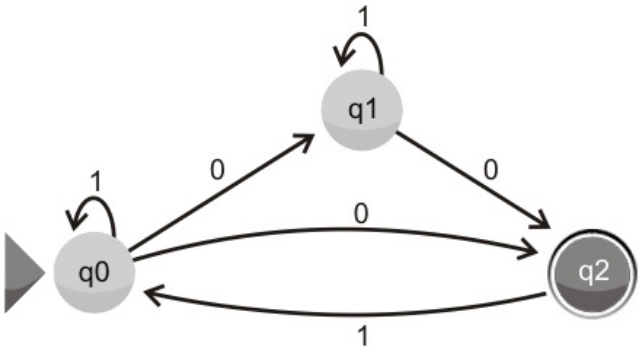


Agora só precisamos descrevê-lo de maneira formal:

A = ($\{q_0, q_1, q_2\}$, {0,1}, δ, $\{q_2\}$), onde:

δ	**0**	**1**
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	∅	$\{q_2\}$
* q_2	$\{q_2\}$	$\{q_2\}$

2. Execute utilizando a notação $\hat{\delta}$ a entrada 10010 no autômato abaixo para definir se a cadeia é aceita ou não.



Resposta:

Começamos pela base:

$\hat{\delta}(q_0, \varepsilon) = \{q_0\}$

Lendo o primeiro caractere:

$\hat{\delta}(q_0, 1) = \delta(q_0, 1) = \{q_0\}$

Note que para o segundo caractere há uma bifurcação. Logo tomamos a união dos caminhos:

$\hat{\delta}(q_0, 10) = \delta(\hat{\delta}(q_0, 1),0) = \delta(q_0,0)=\{q_1, q_2\}$

Note que estamos em dois estados, deveremos acompanhar os dois caminhos:

$\hat{\delta}(q_0, 100) = \delta(\hat{\delta}(q_0, 10),0) = \delta(q_1, 0) \cup \delta(q_2, 0) = \{q_2\} \cup \emptyset = \{q_2\}$

Sempre substituindo o que sabemos da linha anterior podemos continuar:

$\hat{\delta}(q_0, 1001) = \delta(\hat{\delta}(q_0, 100),1) = \delta(q_2, 1)=\{q_0\}$

Continuando:

$\hat{\delta}(q_0, 10010) = \delta(\hat{\delta}(q_0, 1001),0) = \delta(q_0, 0)=\{q_1, q_2\}$

Note que ao ler 10010 estamos nos estados q_1 e q_2 ao mesmo tempo. Como terminamos de ler a cadeia e um dos estados é de aceitação, a cadeia é aceita.

Atividades e Orientações para estudo

Você encontrará nesta seção uma relação de exercícios importantes. É interessante que você se esforce ao máximo para tentar resolver e entender cada questão. Resolvê-los pode ser comparado a um exercício de matemática ou de programação, não existe um método pré-definido cada um pode ver o problema de uma maneira e nos leva a várias respostas. É importante comparar com as respostas de outros colegas e tentar encontrar brechas nos autômatos deles. É um ótimo exercício para exercitar a mente na resolução de problemas.

Nunca esqueça do apoio dos integrantes desse processo com os estudos, os tutores estão sempre prontos para tirar suas dúvidas e lhe orientar nos exercícios.

1. Construa os AFND que reconheçam as seguintes linguagens sobre o alfabeto {0,1}:

a) O conjunto de cadeias que começam por 0 e terminam com 1.

b) O conjunto de cadeias que terminam com 00

c) O conjunto de cadeias que têm 01 como subcadeia.

2. Mostre um AFND que aceita o conjunto de palavras sobre o alfabeto {0,1,...,9} tal que o dígito final já tenha aparecido antes na palavra.

3. Forneça AFNDs que aceitem as linguagens a seguir:

a) O conjunto de cadeias sobre o alfabeto {0, 1, ..., 9} tal que o dígito final tenha aparecido antes.

b) O conjunto de cadeias sobre o alfabeto {0, 1, ..., 9} tal que o dígito final não tenha aparecido antes.

c) O conjunto de cadeias de 0's e 1's tais que não existam dois 0's separados por mais de três 1's.

Considerações Finais

Esperamos que você tenha gostado dos assuntos vistos neste fascículo e esteja se perguntando sobre problemas mais complexos de resolver. Bem, vamos estudar no próximo Fascículo algumas propriedades das linguagens dos autômatos e partiremos então para problemas que os autômatos finitos não são capazes de resolver. Uma dica é ler o máximo que puder da seção de links e conteúdos extras para aprofundar seus conhecimentos.

Resumo

Vimos neste fascículo nosso primeiro modelo matemático de máquinas. Os autômatos finitos são simples e por isso não apresentam um grande poder computacional comparando com os modelos que veremos adiante.

O principal modelo para descrever um autômato finito é como uma função:

$$A = (Q, \sum, \delta, s_0, F), \text{ onde:}$$

- Q é o conjunto finito de estados.
- $\sum$ é o alfabeto de entrada.

- δ é a função de transição.
- $s_0 \in Q$ é o estado inicial.
- $F \subseteq Q$; os elementos de F são os estados finais ou estados de aceitação.

Existem dois tipos de autômatos: AFD e AFND: Os primeiros só podem estar em um estado ao mesmo tempo e normalmente todas as suas transições são preenchidas. Um AFND pode estar em mais de um estado ao mesmo tempo ou mesmo em transição para nenhum estado. Embora os AFND's tenham essa peculiaridade ambos modelos apresentam o mesmo poder computacional, ou seja, o problema que pode ser resolvido por um AFND pode ser também por um AFD e vice-versa.

Saiba Mais

Muitas vezes precisamos estudar algo por diversos ângulos e de maneiras diferentes. Citamos abaixo alguns caminhos para você estudar o assunto com linguagens e notações diferentes ou até mesmo com a mesma. Sua pesquisa é importante principalmente se você não está completamente habituado com o assunto.

Internet:

http://www.ic.uff.br/%7Ecbraga/tc/2005.1 - Curso de teoria da computação.

http://www.inf.puc-rio.br/%7Einf1626 – Notas de aula de teoria da computação.

http://www.deamo.prof.ufu.br/CursoLFA.html - Curso com vários exercícios resolvidos.

http://homepages.dcc.ufmg.br/~nvieira/index.html - Notas de teoria de vários anos.

Leitura adicional:

Hopcroft, John E. e Motwani, Rajeev. e Ullman, Jeffrey D. **Introdução à Teoria de Autômatos, Linguagens e Computação**. Editora Campus, 2002.

Acióly, Benedito. e Bedregal, Benjamín R.C. e Lyra, Aarão.

Introdução à Teoria das Linguagens Formais, dos Autômatos e da Computabilidade. Edições UnP, 2002.

Sudkamp, Thomas A. **Languages and Machines: An Introduction to the Theory of Computer Science**. Addison Wesley, 1997.

Menezes, Paulo Blauth. **Linguagens Formais e Autômatos**. Editora Sagra Luzzatto, 2000.

Diverio, Tiarajú Asmuz e Menezes, Paulo Blauth. **Teoria da Computação: Máquinas Universais e Computabilidade**. Editora Sagra Luzzatto, 1999.

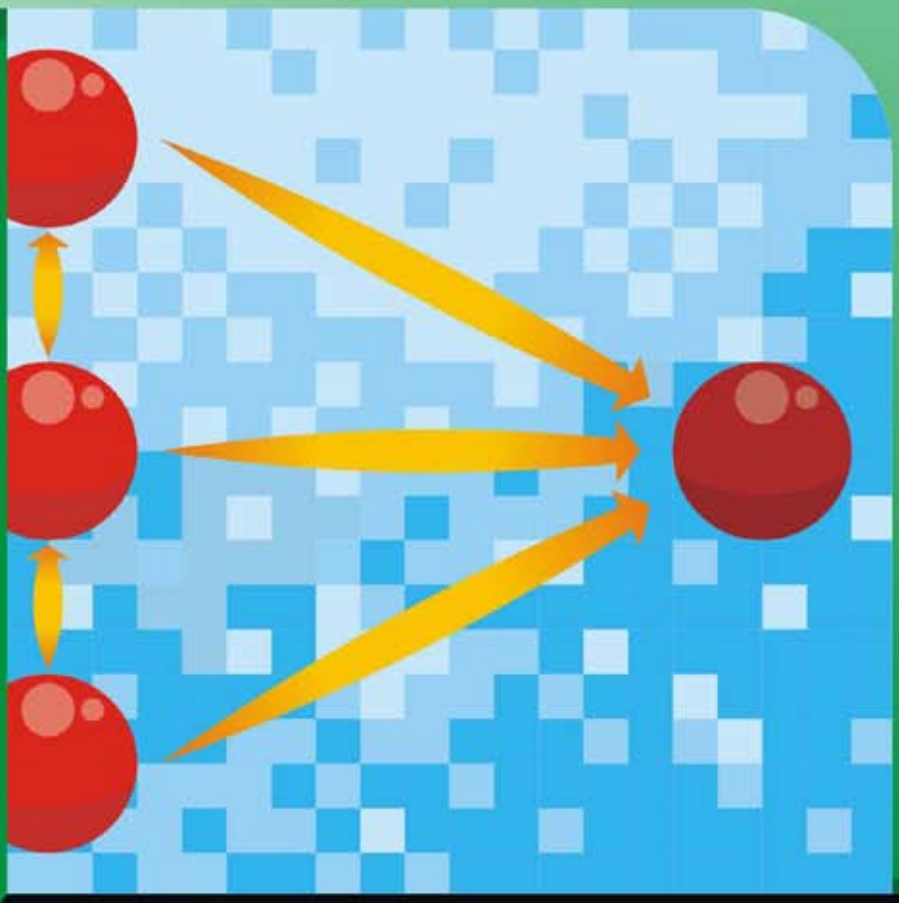
Referências

Hopcroft, John E. e Motwani, Rajeev. e Ullman, Jeffrey D. - Introdução à Teoria de Autômatos, Linguagens e Computação. Editora Campus, 2002.

Lewis, Harry R., Papadimitriou, Christos H. – Elementos de Teoria da Computação, Bookman, 2ª edição.

Introdução a Teoria da Computação

Wilson Galindo
Wilson Rosa
Pablo Azevedo Sampaio



FASCÍCULO 2

Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade
Vice-Reitor: Prof. Reginaldo Barros
Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho
Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire
Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira
Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena
Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial
Capa e Editoração: Italo Amorim, Allyson Vila Nova e Rafael Lira
Revisão Ortográfica: Marcelo Melo
Ilustrações: John Pablo e Diego Almeida
Coordenação de Produção: Marizete Silva Santos

Sumário

Plano da Disciplina

Carga horária: 60h

Ementa da Disciplina

- Autômatos: Finitos, a Pilha e Máquina de Turing (linearmente limitada). Linguagens Formais: Regular, Livre e Sensível ao Contexto, Estrutura de Frases. Hierarquia de Chomsky. Aplicações em compiladores. Computabilidade: modelos computacionais (funções recursivas, linguagens de programação), funções não computáveis, problema da parada, decidibilidade.

Objetivos

Gerais

- Tem como objetivo dar aos cursistas noção formal de algoritmo, computabilidade e do problema de decisão, de modo a deixá-lo consciente das limitações da Ciência da Computação. Aparelhá-los com as ferramentas de modo a habilitá-lo a melhor enfrentar a solução de problemas com o auxílio do computador. Dar subsídios para os cursistas poderem definir linguagens de programação, isto é, sua sintaxe e semântica, através do estudo das gramáticas formais.

Específicos

- Habilitar o cursista aos conhecimentos básicos de autômatos e máquina de Turing, além de um estudo geral sobre a hierarquia dos principais modelos computacionais existentes.

Cronograma de Atividades

1º Módulo Fascículo I Autômatos I	**Aula 1** • Motivação ao estudo da disciplina • Conceitos centrais da teoria de autômatos • Introdução ao estudo de autômatos • Autômatos finitos determinísticos (AFD) Atividade virtual: entrega da 1ª lista de exercícios
2º Módulo Fascículo I Autômatos II	**Aula 2** • Autômatos finitos não determinísticos (AFND) • Autômatos finitos com epsílon-transições **Aula 3** • Equivalência entre AFD e AFND Atividade virtual: entrega da 2ª lista de exercícios
3º Módulo Fascículo II Expressões Regulares	**Aula 4** • Expressões regulares **Aula 5** • Autômatos finitos e expressões regulares • Leis algébricas para expressões regulares Avaliação presencial: Prova escrita
4º Módulo Fascículo II Linguagens Regulares	**Aula 6** • Propriedades das linguagens regulares • Lema do bombeamento • Homomorfismo **Aula 7** • Equivalência e minimização de autômatos Atividade virtual: entrega da 3ª lista de exercícios

5º Módulo Fascículo III Gramáticas e Autômatos a Pilha	**Aula 8** • Gramáticas livres de contexto • Ambigüidade em gramáticas e linguagens **Aula 9** • Autômatos à Pilha Atividade virtual: entrega da 4ª lista de exercícios
6º Módulo Fascículo III Máquina de Turing	**Aula 10** • Breve histórico sobre teoria da computação • A Máquina de Turing **Aula 11** • Indecidibilidade • Problema da Parada • Problemas indecidíveis Atividade presencial 2ª VA: Prova Escrita Atividade presencial 3ª VA: Prova Escrita Atividade presencial Final: Prova Escrita

Obs.: A nota será obtida pela regra: Prova presencial: Peso 6,0. Prova virtual: 4,0.

Capítulo 1 – Expressões Regulares

Bem vindos à nova fase do curso onde iremos aprofundar nossos conhecimentos sobre as linguagens regulares, além de aprender um conjunto mais abrangente de linguagens, que são as linguagens livres de contexto. Este fascículo é especial, pois aprenderemos duas novas maneiras de expressar linguagens (que são conjuntos de palavras). Essas duas novas formas de representar linguagens são as expressões regulares e as gramáticas livres de contexto. A idéia nestes novos modelos não é mais a de aceitar ou rejeitar uma palavra, como antes. Siga adiante com a leitura deste fascículo e descubra, afinal, como estes novos modelos funcionam.

Neste capítulo aprenderemos uma nova forma de descrever linguagens. Uma forma simples e compacta que nos permite fazer operações com linguagens seguindo regras bem definidas – as expressões regulares.

As expressões regulares ajudam não só em problemas teóricos como os que veremos aqui, mas também têm aplicações práticas, como para localizar cadeias em um texto e para criar analisadores léxicos, que são componentes fundamentais dos compiladores. Esses assuntos ficam como sugestão para pesquisa. Antes de você se aventurar neles, no entanto, vamos aprender o que são as expressões regulares.

1.1 Conceitos Iniciais

As expressões regulares são utilizadas principalmente como descritores de linguagens, ou seja, a partir destas expressões podemos identificar uma linguagem regular e dada uma linguagem podemos escrevê-la de forma simplificada usando expressões (se a linguagem for regular).

Para exemplificar o uso de expressões regulares, tome a seguinte linguagem regular: o conjunto de cadeias de 0's e 1's tais que comece com qualquer quantidade de 1's (inclusive nenhum), seguidos necessariamente de um 0 e outra seqüencia com qualquer quantidade de 1's. Essa linguagem aparentemente complexa pode ser escrita em forma de expressão regular facilmente: **1*01***. Veja que, apesar de representar um conjunto de cadeias (ou seja, uma linguagem), as expressões regulares também parecem cadeias. Para diferenciar, vamos colocar sempre as expressões regulares em negrito.

No decorrer do capítulo trabalharemos no sentido de entender como funcionam as expressões e as regras de criação e operações que as regem.

1.2 Expressões Regulares Básicas

Precisamos mostrar a notação que usamos para representar a linguagem associada a uma expressão regular antes de definirmos as expressões: dada uma expressão regular **r** qualquer, a linguagem que ela representa é referenciada por **L(r)**, que você pode ler como "a linguagem de r".

São três os tipos de expressões regulares básicas:

- **s_i**, para todo símbolo s_i do alfabeto, e representa a linguagem { s_i }, ou seja, a linguagem formada pela palavra de um símbolo que tem apenas um símbolo s_i. Assim, podemos escrever que $L(s_i) = \{ s_i \}$.

- ε, que representa a linguagem { ε }, ou seja, a linguagem que contém apenas a palavra vazia. Assim, podemos escrever que $L(\varepsilon) = \{ \varepsilon \}$.

- **Ø**, que representa a linguagem { }, ou seja, a linguagem que não tem palavra nenhuma. Assim, podemos escrever que

L(**Ø**) = { }.

Até agora temos expressões bem simples. Os dois primeiros tipos de expressões representam uma única palavra e são idênticos a ela. Já o terceiro tipo é uma expressão que não representa palavra nenhuma. Vamos ver, a seguir, os operadores que podemos usar para combinar esses três tipos de expressões regulares para formar expressões mais complexas.

1.3 Operadores de Expressões Regulares

São três as operações que usaremos para formar novas expressões regulares. Com elas, aumentamos a capacidade de representação das expressões regulares, de maneira que poderemos descrever qualquer linguagem regular (ou seja, qualquer linguagem que pode representada com autômatos finitos).

União

Vamos primeiro apresentar a idéia de união de linguagens, que é apenas o conceito de união de conjuntos aplicado em linguagens. Tome duas linguagens L = {001, 110} e M = {ε, 11, 110}. A união dessas duas linguagens é o conjunto de cadeias que está na primeira linguagem mais as cadeias contidas na segunda, sem considerar repetições. Denotamos a união de *L* e *M* da seguinte forma: L ∪ M = {001, 110, ε, 11}.

Agora vamos mostrar como usamos esse conceito em expressões regulares. Consideremos duas expressões regulares **E** e **F**. Essas duas expressões representar certas linguagens L(**E**) e L(**F**). Para representar a união das duas linguagens usamos o operador + para combinar as duas expressões regulares assim: **E+F**.

Ou seja: L(**E+F**) = L(**E**) ∪ L(**F**).

Como exemplo, vamos trabalhar no alfabeto {a, b}. A expressão regular **a+b** representa que linguagem? Vamos pensar por partes, partindo das expressões básicas. A expressão básica **a** representa a linguagem {a} e a expressão básica **b** representa a linguagem {b}. Logo, a expressão **a+b** representa a linguagem {a, b} ! Com a prática, você irá se acostumando a entender a linguagem apenas de olhar para a expressão como um todo, sem precisar pensar em todos esses

detalhes. O operador **+** dá a idéia de “ou”, então fica fácil entender que **a+b** representa a cadeia a ou a cadeia b.

Concatenação

Novamente, vamos primeiro apresentar a idéia de concatenação de linguagens antes de falarmos de concatenação de expressões regulares. Considere as linguagens *L* = {001, 110} e *M* = {ε, 11, 110}. A concatenação de L com M é a junção de cada cadeia da primeira com os elementos da segunda cadeia. Podemos escrever *L.M* (com um ponto) ou *LM* (sem ponto), onde *LM* = {001, 00111, 001110, 110, 11011, 110110}. Note que o primeiro e o quarto elemento do conjunto LM são as próprias cadeias de L. Isso aconteceu devido à cadeia vazia ε, que quando concatenada com outra cadeia qualquer dá sempre a outra cadeia (ou seja, 001.ε = ε.001 = 001).

ATENÇÃO! Nunca esqueça que *LM* ≠ *ML*. A operação de concatenação não é comutativa. Note que enquanto *LM* = {001, 00111, 001110, 110, 11011, 110110}, temos *ML* = {001, 110, 11001, 11110, 110001, 110110}.

Assim como nos conjuntos, o operador **.** (ponto) pode ser usado também para representar a concatenação de duas expressões regulares. Também pode acontecer desse operador ser omitido. Se **E** e **F** são expressões regulares, definimos a concatenação das suas linguagens com a expressão **E.F** ou **EF**.

Assim, L(**E.F**) = L(**E**).L(**F**).

Vamos dar dois exemplos usando o alfabeto {a, b, c}. A expressão regular **a.b** ou **ab** representa que linguagem? As expressões básicas **a** e **b** representam {a} e {b}, logo concluímos que a expressão dada representa {ab}. É bem fácil de ver isso apenas olhando para a expressão, não acha? Agora vamos ver uma expressão um pouco mais difícil.

Quais palavras a expressão **(a+b)c** representa? (Considere que os parênteses na expressão servem apenas para agrupamento, como em expressões aritméticas). Na expressão dada, temos uma união de **a** e **b**, que representa {a,b}. Em seguida, concatenada a **(a+b)** temos a expressão **c** que representa {c}. O resultado da concatenação {a,b}.{c} dá a linguagem {ac, bc}, que é a resposta esperada! Você também pode olhar para a expressão **(a+b)c** e entender de maneira mais direta

como “as palavras que têm um a ou um b, seguido de um c”.

Fechamento de Kleene

Diferente das outras duas operações, esta operação é definida sobre uma única linguagem. O *fechamento de Kleene*, que também pode ser chamado de *estrela* ou de *concatenação sucessiva*, é responsável por descrever uma quantidade infinita de cadeias a partir de uma linguagem finita. A idéia é tomando uma linguagem *L*, o fechamento Kleene de *L* representado por *L** é a combinação de nenhum, um ou mais elementos da linguagem *L*. Se assumirmos *L* = {00, 11}, temos que *L** = {ε, 0011, 001100, 11110011, ...}.

Podemos definir o fechamento Kleene em termos de infinitas operações de concatenação e de união. A idéia é de que, para formar o fechamento Kleene L*, podemos usar:

- nenhuma cadeia de L, ou seja, {ε};
- ou cadeias individuais de L, que dá o próprio conjunto L;
- ou cadeias de L concatenadas aos pares, ou seja, L.L;
- ou cadeias de L concatenadas de três em três, ou seja, L.L.L;
- etc.

Essa seqüência continua infinitamente e o fechamento Kleene será a união de todos esses resultados. Portanto, se considerarmos que L^k representa L concantenada consigo mesmo *k* vezes, podemos definir o fechamento Kleene da linguagem L como a união infinita: $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup ...$

ATENÇÃO! Dissemos que o fechamento Kleene L* representa uma linguagem infinita apesar de L ser uma linguagem finita. Porém, há única exceção única. Pense sobre isso e revelaremos em breve.

Vamos tomar como exemplo a linguagem *M* = {0, 01}. O fechamento M* representa a união de todos os conjuntos descritos abaixo:

- $M^0 = \{\varepsilon\}$
- $M^1 = \{0, 01\}$
- M^2 = {00, 0101, 001, 010}, que é formado pela concatenação de pares de cadeias de L

- M^3 = {000, 0001, 01010, 010101, 0010, 00101, 0100, 01001}}, que é formado pela concatenação de pares de cadeias de L
- Continua infinitamente...

Como conseqüência da definição, se tomarmos um alfabeto T, o seu fechamento Kleene T* representará o conjunto de todas as cadeias formadas pelos símbolos do alfabeto. Por exemplo, se T = {0, 1}, então T* é o conjunto de todas as cadeias formadas de 0's 1's.

Agora que você entendeu o fechamento Kleene para linguagens e alfabetos, será fácil falar dessa operação para expressões regulares. Dada uma expressão regular **E**, representamos o fechamento Kleene como **E***, que representa o fechamento Kleene da linguagem que **E** representa.

Assim, a linguagem definida por esse operador pode ser representada como L(**E***) = (L(**E**))*.

Vamos ver exemplos com base no alfabeto {0, 1}. Que linguagem a expressão regular **0*** representa? É fácil perceber que ela representa a linguagem {ε, 0, 00, 000, 0000, ...}. Como você pode perceber neste exemplo, podemos entender o operador * como "nenhuma, uma ou mais repetições" da expressão interna (**0**, no caso).

E a expressão **(0+01)***, que linguagem representa? Veja que a expressão interna **0+01** representa {0, 01}. Assim, a expressão como um todo representa nenhuma ocorrência, ou uma ocorrência ou repetidas ocorrências dessas palavras, ou seja: {ε, 0, 01, 00, 0101, 001, 010, ... }.

Já vimos todos os casos básicos e todos os operadores de expressões regulares. Como todos os operadores podem aparecer juntos em uma mesma expressão, precisamos adotar uma convenção para facilitar o entendimento das expressões. Essa convenção é explicada na próximas seção.

1.4 Precedência de operadores em expressões regulares

Assim como em outras álgebras, já estamos acostumados a trabalhar com precedências para operadores. Por exemplo, com a expressão aritmética 1+2*3, embora o sinal de soma venha primeiro efetuamos primeiro a multiplicação, pois esta tem precedência

(prioridade) sobre a soma. Regras de precedência semelhantes existem também para expressões regulares e serão explicadas adiantes. Antes, vale ressaltar que, assim como na álgebra, sempre podemos usar parênteses para alterar as precedências de modo que (1+2)*3 é diferente de 1+2*3.

As regras de precedência para expressões regulares são explicadas abaixo:

1 – O operador de maior precedência é o estrela (*).

Ex: Suponha a expressão **1.1.0***, que de maneira simplificada escrevemos **110***. Como o operador estrela tem maior precedência, ele é aplicado apenas ao 0 e não à cadeia 110 toda. Formamos então a seguinte linguagem, onde o 0 pode aparecer nenhuma, uma ou várias vezes: {11, 110, 1100, 11000, ...}.

2 – O próximo operador que tem precedência é o operador de concatenação (.)

Ex: Como foi mostrado no caso anterior, **110*** é interpretado como **11(0*)**. Em outras palavras o * é executado inicialmente e após isso o **11** irá se concatenar à frente do resultado do **0***.

3 – O operador de última precedência é a união (+).

Ex: Dada uma expressão regular **00*0+011**, podemos entendê-la como: **(0(0*)0)+(011)** onde executamos primeiro as expressões mais internas dos parênteses e usamos o resultado para resolver as mais externas. Dessa forma, o operador + é o último a ser aplicado.

Assim como acontece com autômatos, você precisa praticar um pouco para se acostumar com o uso de expressões regulares para representar linguagens. Por isso, vamos exercitar o entendimento e a construção de expressões regulares na próxima seção.

Aprenda Praticando

1 – De acordo com a ordem de precedência estudada, reescreva as expressões regulares de forma usando parênteses deixando de forma clara as precedências de operadores.

a) 00*

Resposta: Como o estrela é prioritário, então colocamos parênteses, obtendo a expressão: **0(0*)**

b) 000+111

Resposta: Como a concatenação tem maior prioridade, então colocamos parênteses nos elementos concatenados, tendo assim a expressão: **(000)+(111)**

2 – A linguagem representada pela expressão regular **(0+1)*** é infinita, mas mostre os elementos da linguagem tenham comprimento de, no máximo, dois símbolos.

Resposta: Começamos com as cadeias de tamanho zero e já sabemos que é o ε. Agora para as cadeias de tamanho um: por causa dos parênteses precisamos aplicar primeiro a parte interna dada por 0 ou 1. São mais duas cadeias da linguagem. Agora para os elementos de tamanho 2: consistem na expressão: **(0+1)(0+1)**. São 4 possibilidades: escolher 0 no primeiro parênteses e 0 no segundo, ou 0 no primeiro e 1 no segundo, até abranger todas as combinações possíveis. Podemos então construir o conjunto dos elementos de tamanho máximo 2: {ε, 0, 1, 00, 01, 01, 10, 11}

3 – Escreva as expressões regulares de acordo com as linguagens dadas a seguir:

a) O conjunto de todas as cadeias de 0's e 1's com exatamente três símbolos.

Resposta: Sempre começamos pelos casos mais simples, como, por exemplo, as menores cadeias. Então vamos pensar na expressão regular que representa cadeias de 0's e 1's com apenas um símbolo. Para cadeias de um símbolo, basta escolher entre 0 e 1, o que pode ser representado pela expressão **(0+1)**. Agora, para cadeias de tamanho 2, precisamos escolher duas vezes um símbolo, o que pode ser representado pela expressão: **(0+1)(0+1)**. De modo similar, as cadeias de tamanho 3 podem ser representadas pela expressão: **(0+1)(0+1)(0+1)**.

b) O conjunto de cadeias de 0's e 1's contendo pelo menos um símbolo 0.

Resposta: Começaremos com a primeira parte do exemplo: cadeias de 0's e 1's. Pensemos no primeiro caractere, que pode ser

0 ou 1, logo temos **0+1**. Temos aí uma cadeia de tamanho um, mas precisamos de cadeias de qualquer tamanho. Para as cadeias de tamanho dois, temos **(0+1)(0+1)** e da mesma forma as de tamanho três **(0+1)(0+1)(0+1)**. Quando o tamanho é conhecido é fácil continuar criando expressões seguindo a mesma lógica, mas como a cadeia pode ter qualquer tamanho, precisamos utilizar uma ferramenta para nos fornecer uma quantidade infinita de concatenações. Utilizaremos então o fechamento Kleene. Assim, resultando em **(0+1)***, que representa qualquer cadeia com 0's e 1's.

Devemos agora analisar a segunda parte da questão referente à obrigação de haver pelo menos um 0. Para isso, devemos evitar cadeias do tipo 1111. Tendo a cadeia pelo menos um 0, este pode estar no inicio, no meio ou no final da cadeia. Lembre-se de considerar *todos* os casos, assim como você faz com autômatos. No nosso problema são três casos:

i) Zero no início da cadeia: devemos concatenar 0 à nossa expressão: **0(0+1)***. Note que essa expressão trata de todo conjunto de 0's e 1's que começa com zero. Esta expressão envolve cadeias como: {0, 00, 01, 000, 001, 010, 011, ...}

ii) Zero no final da cadeia: consideramos neste momento qualquer cadeia de 0's e 1's mas o último caractere é necessariamente 0. Obtemos esse resultado concatenando zero no final da expressão dessa forma **(0+1)*0**. Esta expressão envolve cadeias como: {10, 000, 100, 10110, 010000, ...}

iii) Falta apenas a condição de ter um 0 no meio da cadeia. Para isto basta representar qualquer conjunto de zeros e uns, concatenar com zero e adicionar mais qualquer conjunto de zeros e uns. Obtendo assim: **(0+1)*0(0+1)***.

Como a linguagem se refere a todos os casos i), ii) e iii) citados anteriormente, tomaremos a união e formaremos uma só expressão regular: **0(0+1)* + (0+1)0* + (0+1)*0(0+1)***.

Terminado a questão é recomendado revisar sua expressão para saber se a mesma pode ser simplificada. É o nosso caso. O elemento cadeia vazia (ε) faz parte de toda expressão com fechamento. Observe atentamente a expressão anterior formada por três partes i), ii) e iii). Se substituirmos **(0+1)*** inicial de iii) por ε obtemos i). Agora se substituir o **(0+1)*** final por ε obtemos **(0+1)*0**, exatamente como

ii). Em outras palavras, podemos obter as expressões i) e ii) a partir de iii), tornando a presença das duas primeiras desnecessária. Temos então outra resposta bem mais simples: **(0+1)*0(0+1)***

Você pode "ler" essa expressão como a linguagem formada por todas as cadeias iniciadas com "qualquer cadeia de 0's ou 1's (possivelmente vazia), seguida de um 0, seguida de outra cadeia qualquer de 0's ou 1's (possivelmente vazia)".

ATENÇÃO! Não existe apenas uma expressão regular para cada problema. Quando for estudar, mostre suas expressões a seus colegas para confirmar se sua expressão está correta e analise as deles, assim você ganhará experiência essencial nessa parte do curso.

4 – Forneça uma descrição em português da expressão: **(0+1)*101(0+1)***.

Resposta: Note que a primeira e a última parte entre parênteses já são conhecidas, representam qualquer cadeia de 0's e 1's. Resta interpretar a parte intermediária. Sempre é interessante que comecemos pelo menor tamanho de cadeia possível. Substituindo as expressões **(0+1)*** por ε, temos a cadeia **101**. É a menor cadeia obtida a partir da expressão fornecida. Agora temos três possibilidades: **(0+1)*** inicial substituída por qualquer subcadeia de zeros e uns, **(0+1)*** final substituída por qualquer subcadeia de zeros e uns, ou ambas sendo substituídas. Sempre a subcadeia **101** vai estar presente. Podemos afirmar então que a expressão dada na questão representa o conjunto de todas as cadeias de zeros e uns que contém **101** como subcadeia.

Atividades e Orientações para estudo

Agora é com você. Tente resolver o máximo de questões possível e as que não conseguir reporte as dúvidas a seus tutores, pesquise, poste no fórum mas não fique sem a resposta.

1) De acordo com a ordem de precedência estudada, reescreva as expressões regulares de forma usando parênteses deixando de forma clara as precedências de operadores.

a) 000*

b) 00+00

c) 0*000*

d) 00*+0*

e) 01*+1*0

2) Escreva as linguagens das expressões regulares a seguir considerando os elementos que tenham comprimento máximo de 3 símbolos:

a) 0*01+001*

b) (0+1)*(1+0)

c) (001)*+01

d) 01*0

e) (01)*0

3) Escreva as expressões regulares de acordo com as descrições dadas a seguir:

O conjunto de cadeias sobre o alfabeto *{a, b, c}* que contém pelo menos um *a* e pelo menos um *b*.

O conjunto de cadeias de 0's e 1's cujo décimo símbolo a partir da extremidade direita é um.

O conjunto de cadeias de 0's e 1's com no máximo um par de 1's consecutivos.

4) Dada a expressão regular a seguir forneça a descrição em português das seguintes expressões:

O conjunto de cadeias sobre o alfabeto *{a, b, c}* que contém pelo menos um *a* e pelo menos um *b*.

O conjunto de cadeias de 0's e 1's cujo quinto símbolo a partir da extremidade direita é um.

O conjunto de cadeias de 0's e 1's com no máximo um par de 1's consecutivos.

O conjunto de todos os cadeias de 0's e 1's tais que todo par de 0's adjacentes aparece antes de qualquer par de 1's adjacentes.

O conjunto de cadeias de 0's e 1's cujo número de 0's é divisível por 5.

O conjunto de cadeia de 0’s e 1’s que não contém 101 como um subcadeia.

Capítulo 2 – Autômatos Finitos e Expressões Regulares

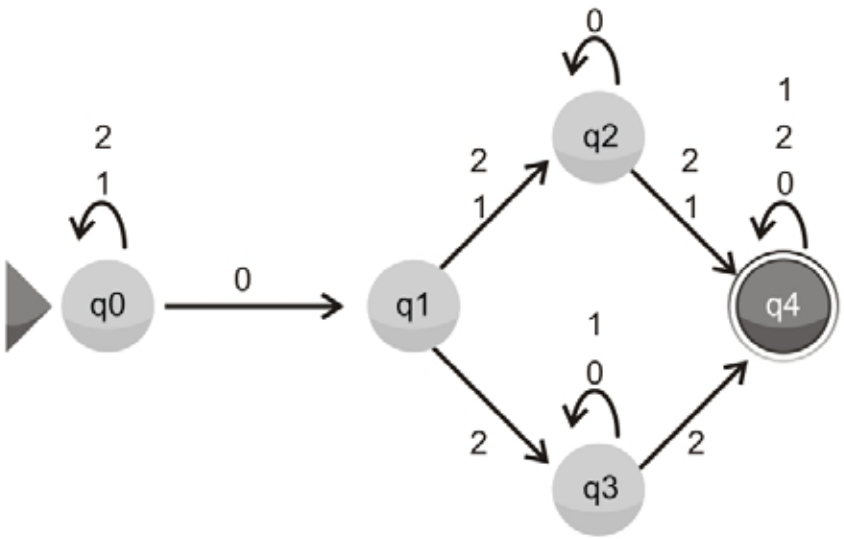
Este capítulo trata da relação entre autômatos finitos e expressões regulares. Nós sabemos que as linguagens reconhecidas pelos autômatos finitos (AF) são chamadas de linguagens regulares. Em Teoria da Computação é importante a comparação de modelos para saber se um modelo é mais poderoso que outro, ou seja, se reconhece uma classe de linguagens maior. Então, neste capítulo, vamos comparar os autômatos com as expressões regulares.

O que descobriremos aqui é que as expressões regulares também reconhecem as linguagens regulares. Para isso, vamos demonstrar a equivalência entre os autômatos finitos (AFs) e as expressões regulares (ERs). Provaremos a equivalência entre ER e AF da seguinte forma: primeiro mostraremos como obter uma ER a partir de um AF e em seguida mostraremos como obter um autômato a partir de uma ER.

2.1 De Autômatos Finitos para Expressões Regulares

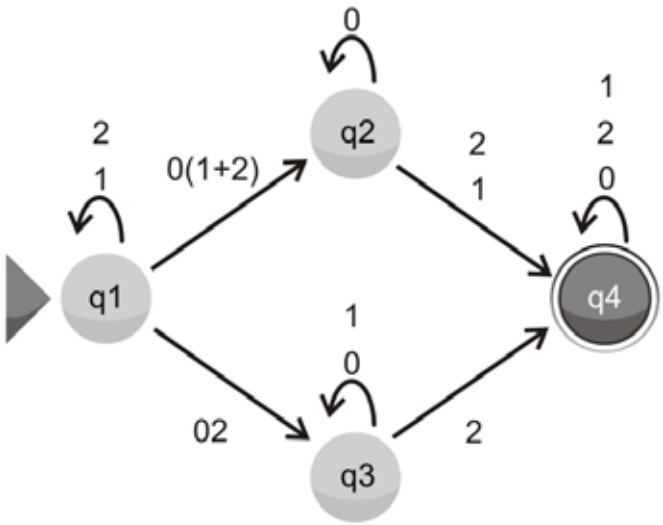
Para demonstrar essa passagem utilizaremos um método chamado Eliminação de Estados. Por simplicidade usaremos autômatos finitos determinísticos (AFD) na conversão, caso seja necessário converter um AFND basta convertê-lo em AFD como foi mostrado anteriormente. Basicamente trata de substituir as transições por expressões regulares equivalentes seguindo alguns passos básicos mostrados a seguir.

Dado um DFA sobre o alfabeto {0, 1, 2}:



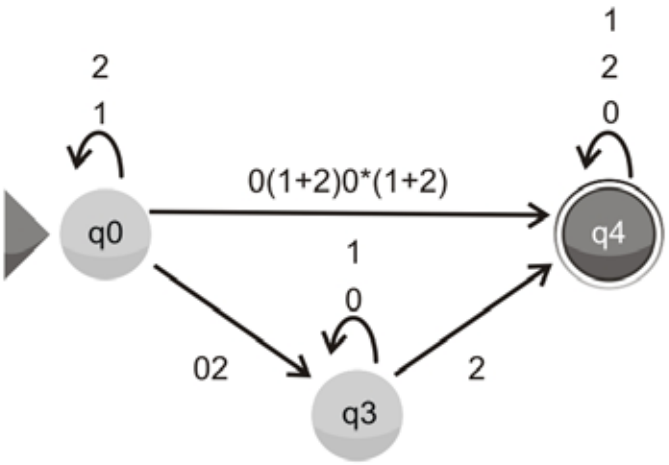
O primeiro passo é substituir todo estado exceto o inicial e os de aceitação. Queremos eliminar inicialmente o estado q_1 logo devemos começar a pensar como seria uma transição equivalente de q_0 a q_3 e q_0 a q_2 sem a necessidade do estado q_1. Começaremos com o caminho de q_0 a q_3: Estando em q_0 e lendo 0 vamos para q_1 e lendo 2 vamos para q_3. Em termo de expressão regular deduzimos **e** implica em concatenação (.) e **ou** implica em união (+) de expressões. Podemos então afirmar que lendo 0 **e** 2 q_0 vai para q_3 neste caso usamos a expressão de concatenação **02**. Analisando de q_0 a q_2: lendo 0 vamos para q_1 e lendo 2 **ou** 1 vamos para q_2. Temos então **0(2+1)**.

Reescrevendo o autômato com as novas informações:

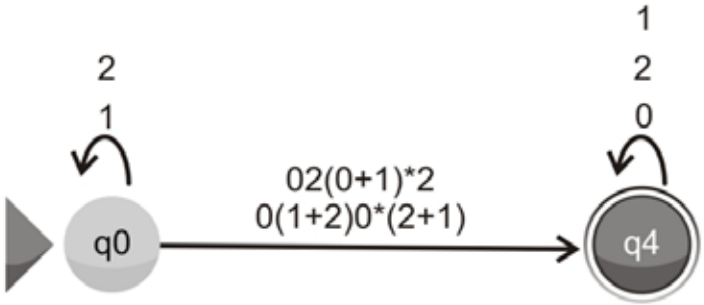


Devemos agora eliminar outro estado: Observe o caminho de q_0 a q_4 passando por q_2. Lemos a expressão **0(1+2)** mas em q_2 há uma transição 0 para ele mesmo, significa que este caractere pode

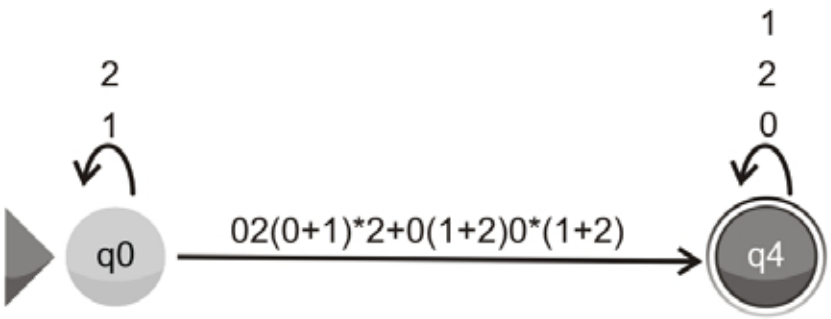
ser repetido nenhuma, uma ou várias vezes e o operador se encaixa nesse comportamento é o estrela (*) resultando em **0***. Posteriormente temos outra união resultado da transição de q_2 a q_4 com **(1+2)**. Concatenando as três expressões temos **0(1+2)0*(1+2)**. Resultando em:



Eliminando q_3 facilmente ligamos a expressão **02** com a expressão **(0+1)*** da transição em q_3 (observe que estando em q_3 e lendo qualquer seqüência de 0's e 1's não causará alteração de estado). Em seguida lemos o caractere **2**. Concatenando as expressões temos: **02(0+1)*2** e graficamente:



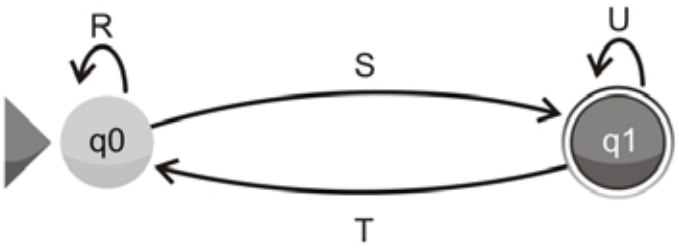
E pode ser facilmente redesenhado como:



Até agora não tivemos dificuldades em montar uma expressão regular a partir do autômato. Continuando com nossa conversão, em q_0 temos **(1+2)*** e em q_4 temos **(0+1+2)*** e concatenando todas as expressões de q_0 a q_4 obtemos a expressão:

$$\underbrace{(1+2)^*}_{q_0}\ \underbrace{02(0+1)^*2+0(1+2)0^*(1+2)}_{q_0 \text{ a } q_4}\ \underbrace{(0+1+2)^*}_{q_4}.$$

Mas você já deve ter se perguntado se esse exemplo é geral. Quando o autômato fica apenas com estado inicial e estados de aceitação podemos criar uma situação onde todas as possibilidades são utilizadas:



Os caracteres {R, S, T, U} representam expressões regulares resultantes obtidas pelo processo de eliminação de estados definido a pouco. Para representar esse autômato começamos analisando as expressões que fazem o autômato alcançar o estado de aceitação: Podemos seguir os passos:

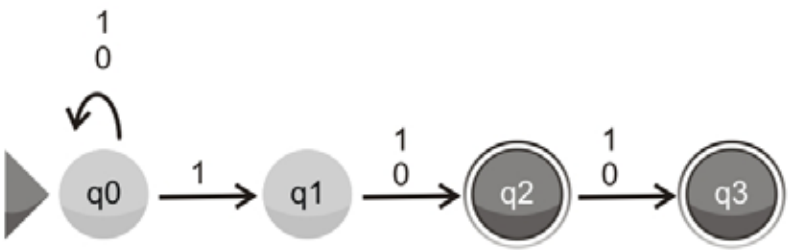
- R*SU*

- SU*TR*SU*

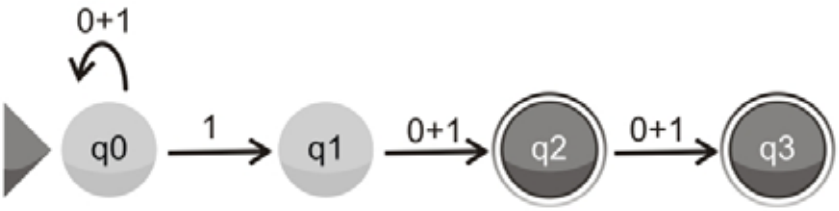
E simplificando: **(R+SU*T) SU***

Aprenda Praticando

Transforme o seguinte autômato em expressão regular:



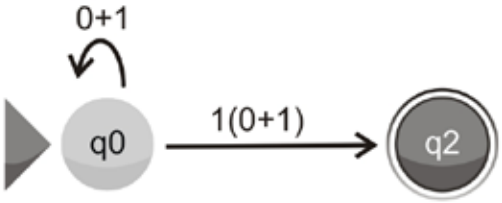
Organizando os caracteres



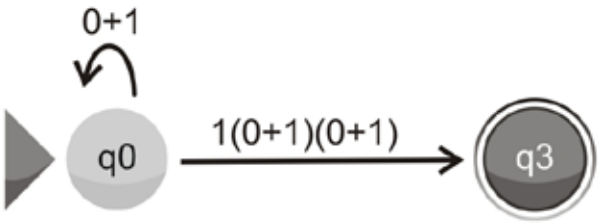
Inicialmente apenas o estado q_1 vai ser eliminado, o resultando em **1(0+1)** mostrado na figura a seguir:

Agora basta isolar os autômatos como se só existisse um estado de aceitação, assim temos:

De q_0 a q_2:



De q_0 a q_3:



Do primeiro desenho obtemos **(0+1)*1(0+1)** e o segundo **(0+1)1(0+1)(0+1)** pela simples concatenação dos rótulos dos autômatos. E agora unindo os dois caminhos: **(0+1)*1(0+1)+(0+1)1(0+1)(0+1)**.

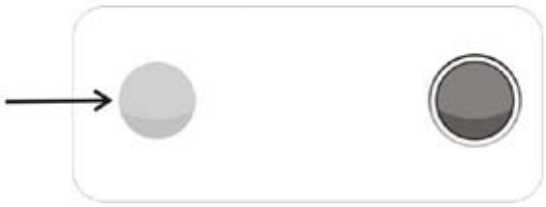
2.2 De Expressões Regulares para Autômatos Finitos

Agora concluiremos com a segunda parte da demonstração. Usaremos agora uma construção para os operadores das expressões regulares que vai resultar em autômatos finitos com ε transições (ε-AFND). Nosso objetivo é substituir os operadores por construções cujo resultado é um autômato finito equivalente.

A prova é uma indução estrutural sobre uma expressão R. Os casos base são {ε, Ø, a} uma expressão que representa {ε}, uma que representa o vazio, e uma que representa um caractere.

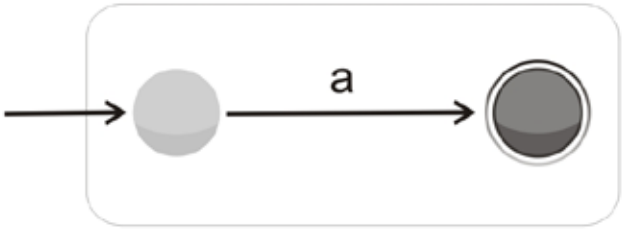
Temos abaixo uma figura que representa um autômato que aceita o elemento {ε}.

Agora mostraremos a construção para a cadeia vazia.



Note que neste caso não há caminho entre o estado inicial e o estado de aceitação, por isso o resultado deste autômato é Ø.

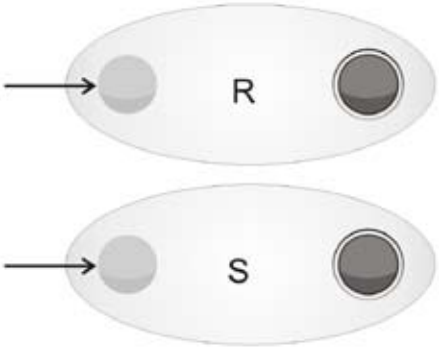
O terceiro caso trata de uma expressão regular de tamanho um que resulta no seguinte autômato:



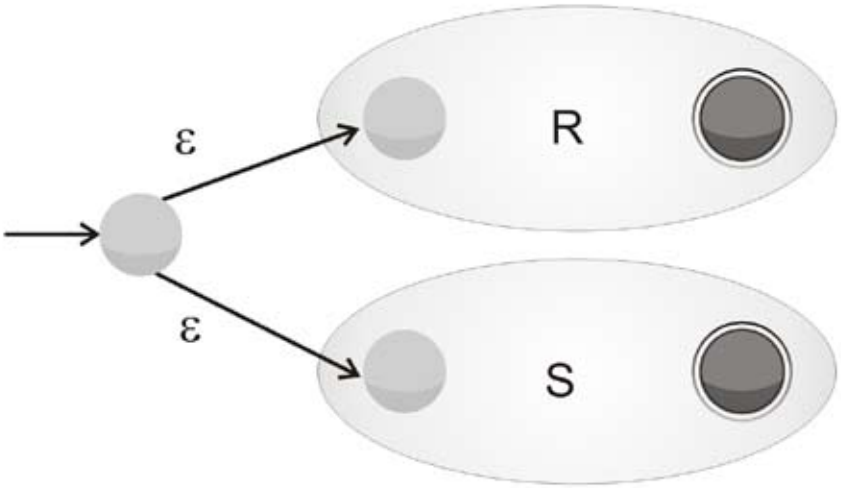
A indução é dada pela definição dos três operadores das expressões regulares em forma de autômatos.

União:

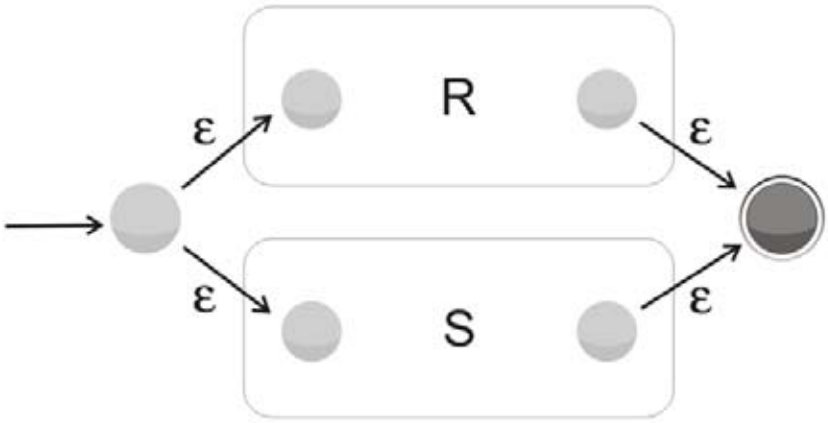
Inicialmente temos apenas dois autômatos, referentes às expressões R e S.



O primeiro passo para construir a união desses autômatos é criar um novo estado inicial e ligar com ε o novo estado inicial ao estado inicial de R e S (que não serão mais iniciais).



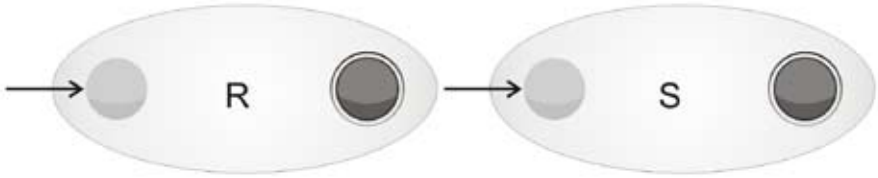
Então basta criar um novo estado de aceitação e ligá-lo aos estados de aceitação de R e S (que não serão mais de aceitação).



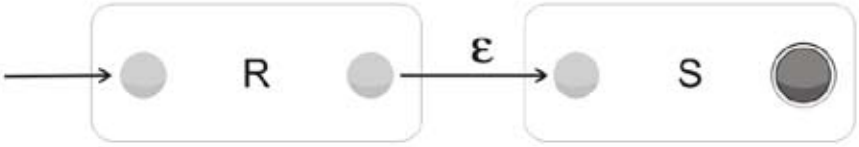
Note que tínhamos dois autômatos das expressões R e S e criamos um autômato maior referente à operação de união. Começamos no estado inicial e imediatamente seguimos nas duas direções, se algum caminho (R ou S) for aceito, então o autômato maior aceitará a cadeia.

Concatenação:

Descreveremos agora a operação de concatenação. Dados dois autômatos (das expressões R e S) queremos representar a concatenação entre as expressões R e S. Dados os autômatos abaixo:



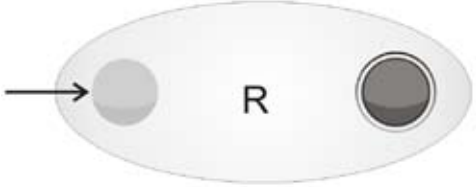
Para construir o autômato resultante da concatenação criaremos outro, onde o estado inicial é o estado inicial de R e o estado de aceitação é o de S. Como nova estrutura basta ligarmos o estado de aceitação de R (que não será mais de aceitação) e o estado inicial de S (que não será mais inicial) com ε. Resultando no seguinte autômato:



O que fizemos foi construir um autômato para aceitar somente quando R aceita e em seguida S aceita, este é o objetivo da concatenação.

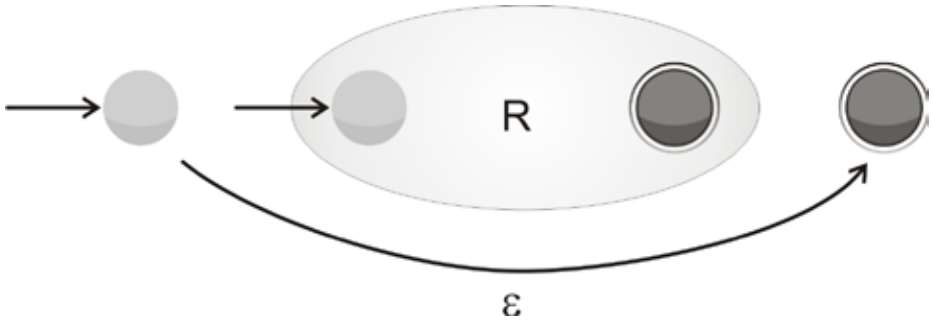
Estrela:

Mostraremos agora como construir um autômato que reconheça o operador estrela de uma expressão regular. Dado um autômato referente à expressão regular R.

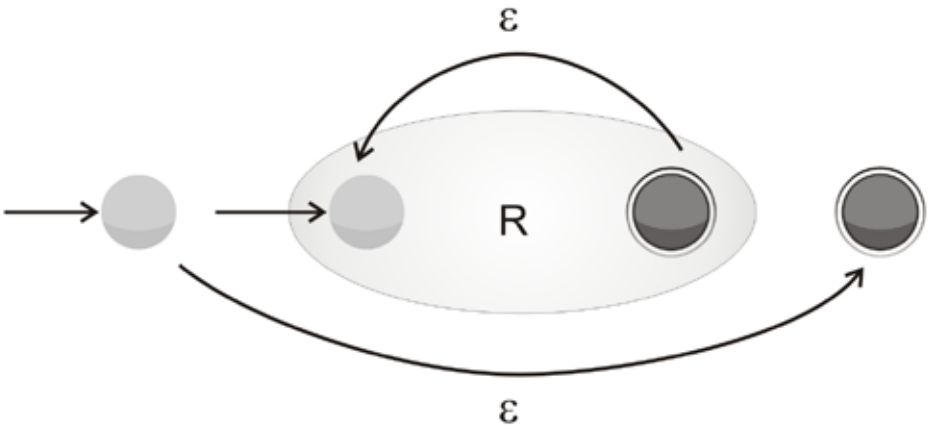


Construiremos um novo autômato que será equivalente a aplicação do operador estrela em uma expressão regular. Para isso, basta criar um novo estado inicial e um novo estado de aceitação.

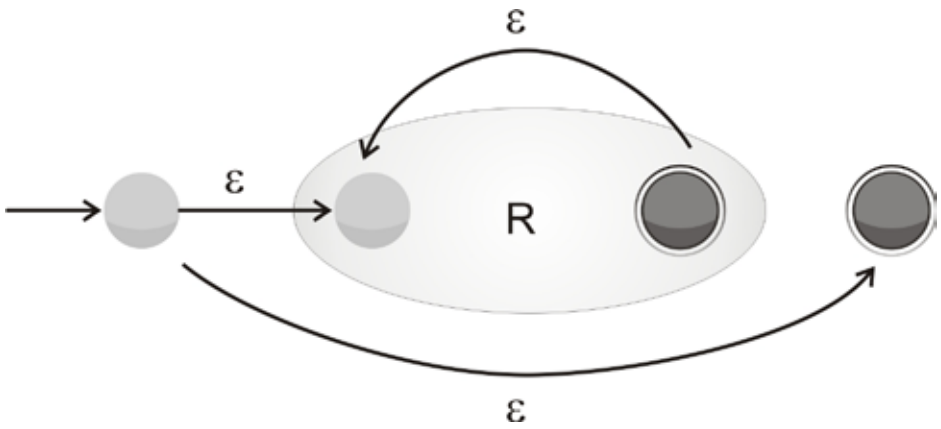
Devemos interligar diretamente com ε o novo estado inicial e o novo estado de aceitação.



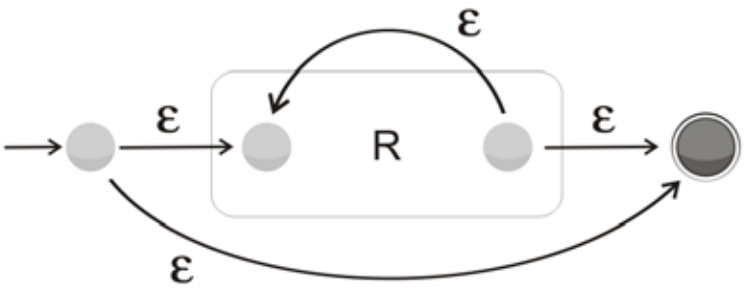
Criamos então uma ligação com ε entre o estado de aceitação de R e seu estado inicial.



Ligue o novo estado inicial e o estado inicial de R com ε;



Ligar o estado de aceitação de R e o novo estado de aceitação com ε.



Observe que R não tem mais estado inicial nem estado de aceitação. Há duas formas de se chegar ao estado de aceitação neste novo autômato, ou diretamente sem ler nenhum caractere (ε) ou reconhecendo R uma ou várias vezes (R*).

Atenção!

Não confunda! Em uma derivação, usa-se uma seta diferente da seta usada nas regras.

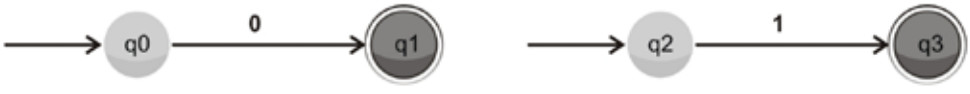
Já sabemos como agir diante de qualquer elemento de uma expressão regular para transformá-la em autômato finito, precisamos agora praticar.

Aprenda Praticando

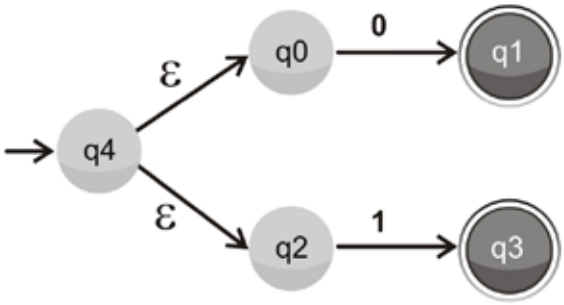
Converta as seguintes expressões regulares em autômatos:

a) (0+1)1

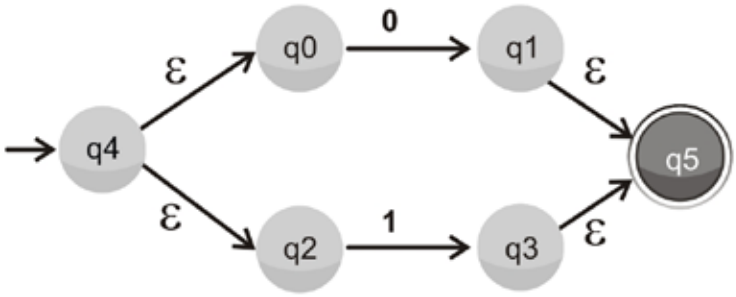
Resposta: Começamos com a parte em parênteses **(0+1)**. Devemos escrever primeiro os autômatos referentes ao 0 e ao 1.



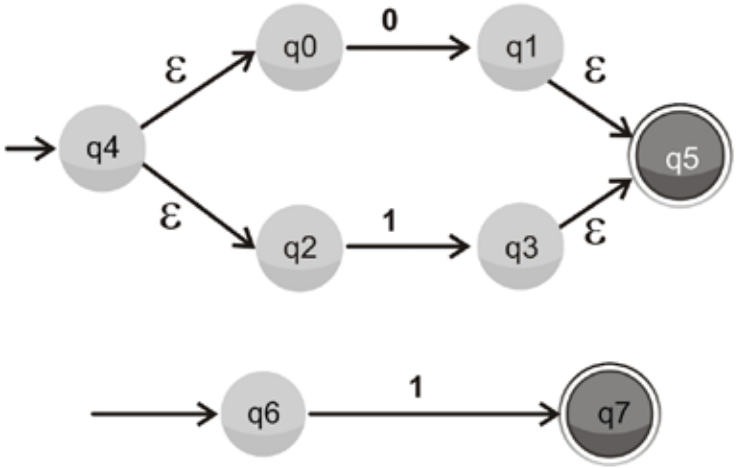
Seguindo o algoritmo dado na explicação, criaremos o novo estado inicial e ligaremos com ε aos dois estados iniciais.



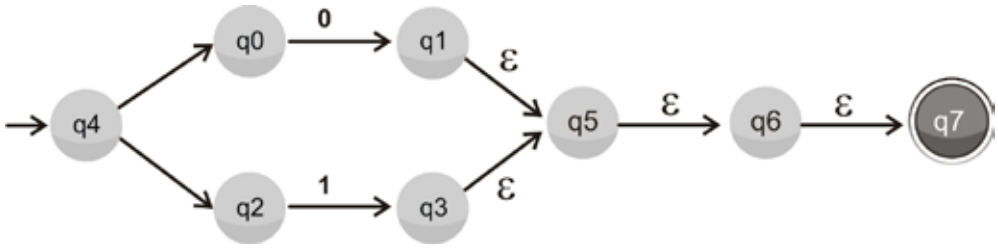
Criamos então o estado de aceitação e ligamos com ε com os estados de aceitação dos autômatos.



Só precisamos agora concatenar nosso novo autômato com o autômato que reconhece 1. Dados os autômatos a seguir:



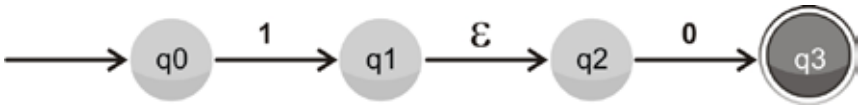
Para isto, basta ligarmos o estado de aceitação do primeiro autômato **(0+1)** com o estado inicial do segundo autômato **(1)** e obter o seguinte resultado:



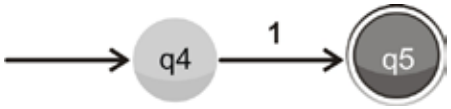
b) 101*

Resposta: Faremos inicialmente a concatenação dos dois primeiros caracteres. Dados dois autômatos:

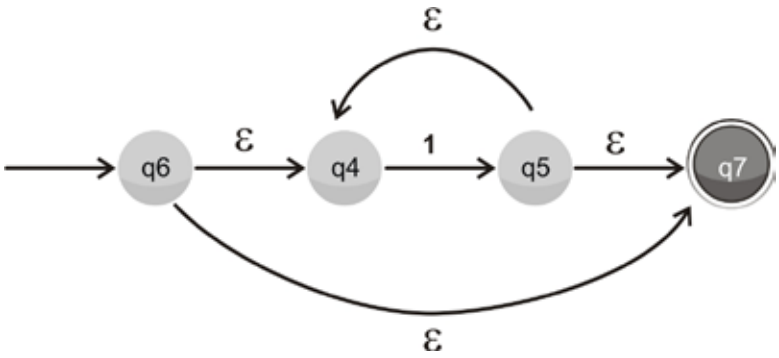
Obtemos a concatenação ligando com ε o estado de aceitação do primeiro autômato com o estado inicial do primeiro:



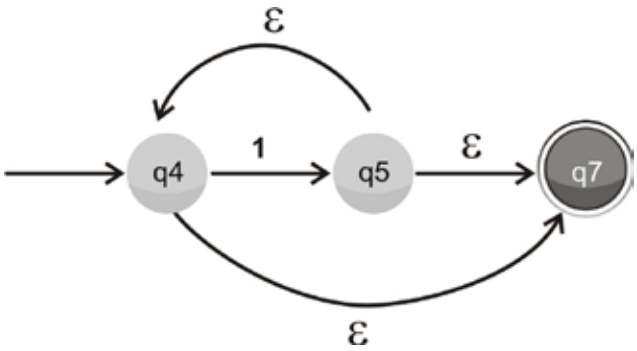
Agora trataremos da expressão **1***, dado o autômato abaixo:



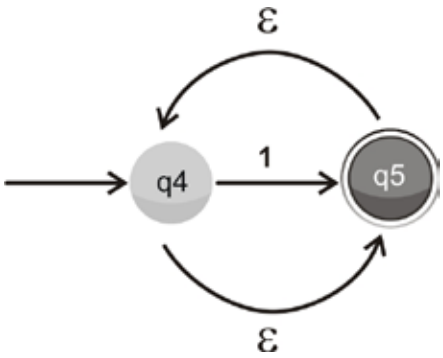
Seguindo a regra dada anteriormente, temos o seguinte autômato para **1***:



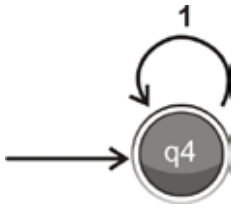
Mas não necessariamente devemos seguir exatamente essas regras, podemos diminuir o tamanho dos autômatos para tornar mais simples o entendimento. No autômato anterior, note que o autômato nunca fica em q_6 pois espontaneamente vai para q_4 e q_7. Podemos então eliminar o estado q_6 desde que q_4 seja o estado inicial e tenha uma transição ε para q_7 para manter o autômato equivalente.



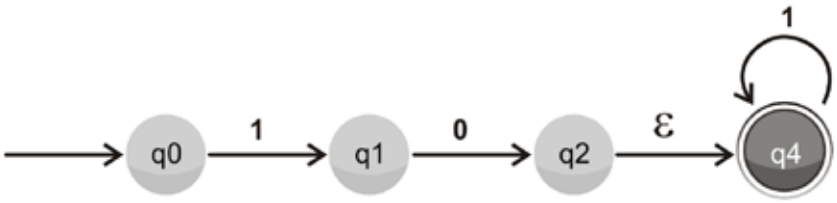
Podemos eliminar q_7 da seguinte forma: há uma transição de q_4 para o estado de aceitação que deve ser respeitada e estando em q_5 imediatamente o autômato é levado ao estado de aceitação, então tornaremos q_5 o estado de aceitação e conseqüentemente q_4 com ε apontará para q_5. Obtendo o seguinte autômato:



Podemos simplificar pois note o papel dos ε sempre fazendo o autômato oscilar entre q_4 e q_5 por isso eliminando q_5 podemos chegar no seguinte autômato equivalente:

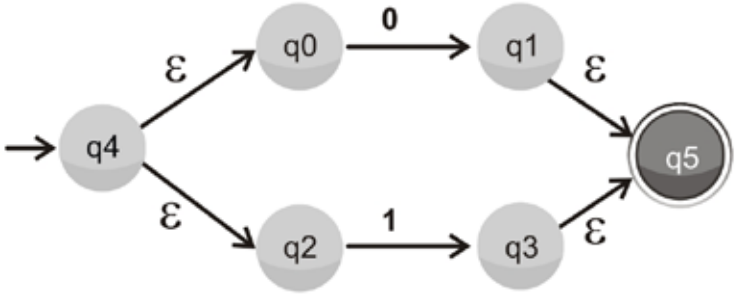


Lembre-se que você não é obrigado a simplificar os autômatos, este exemplo quis apenas mostrar a possibilidade. Continuando com a resolução da questão basta agora concatenar o autômato construído por nós (expressão **10**) com o autômato da expressão **1***. Resultando em:

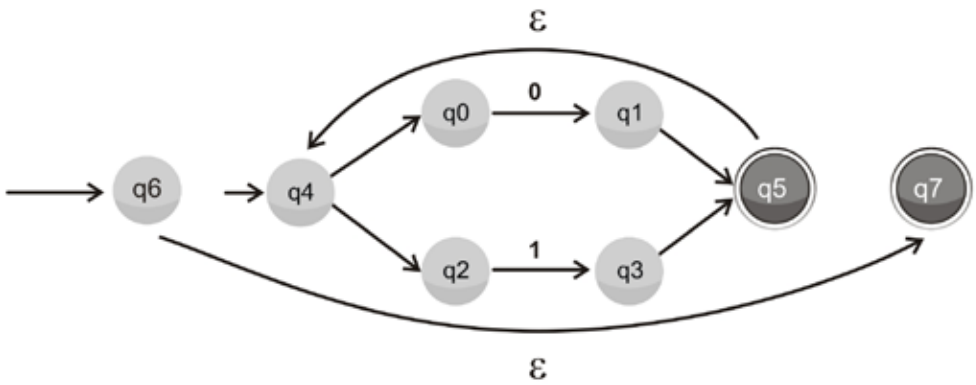


c) (0+1)*

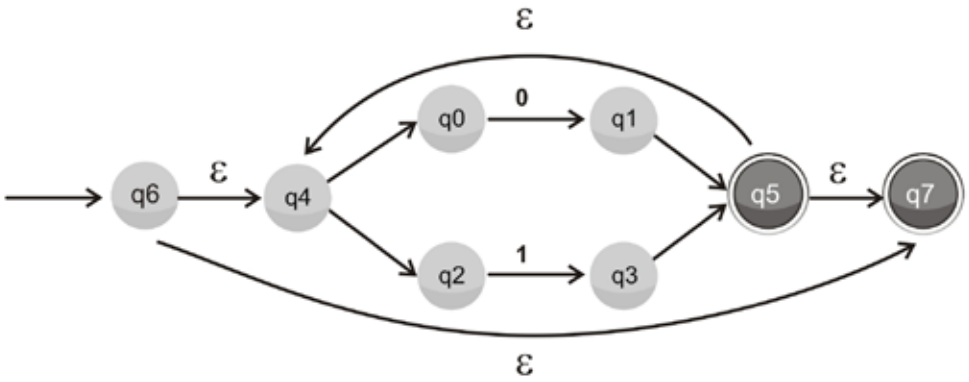
Resposta: Para responder as questões devemos sempre começar pelas expressões entre parênteses. Note que já conhecemos o autômato que reconhece **(0+1)** da letra a, como é dado a seguir:



Precisamos apenas aplicar estrela no autômato como um todo. Primeiro criaremos os dois estados e interligaremos ε entre eles. Podemos também ligar os antigos estado de aceitação e estado inicial com ε.



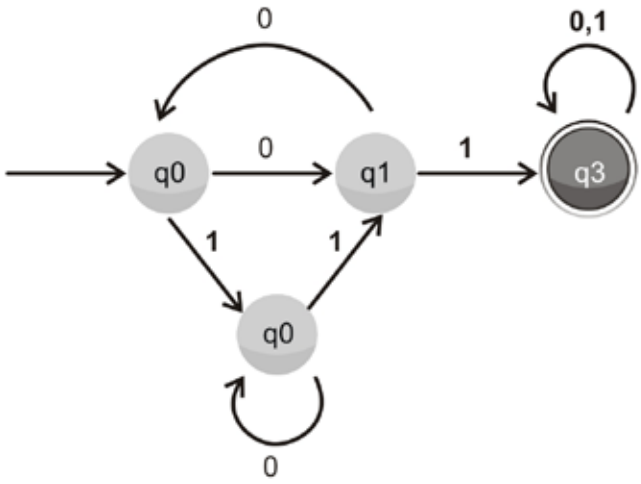
Agora só precisamos ligar o novo estado inicial e de aceitação ao autômato.



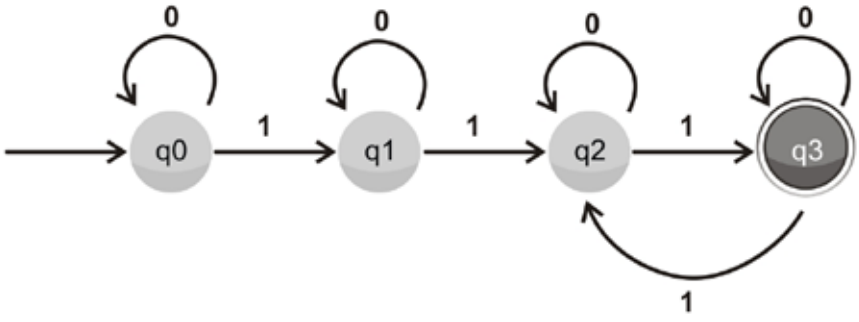
Atividades e Orientações para estudo

1) Converta os seguintes autômatos para expressões regulares:

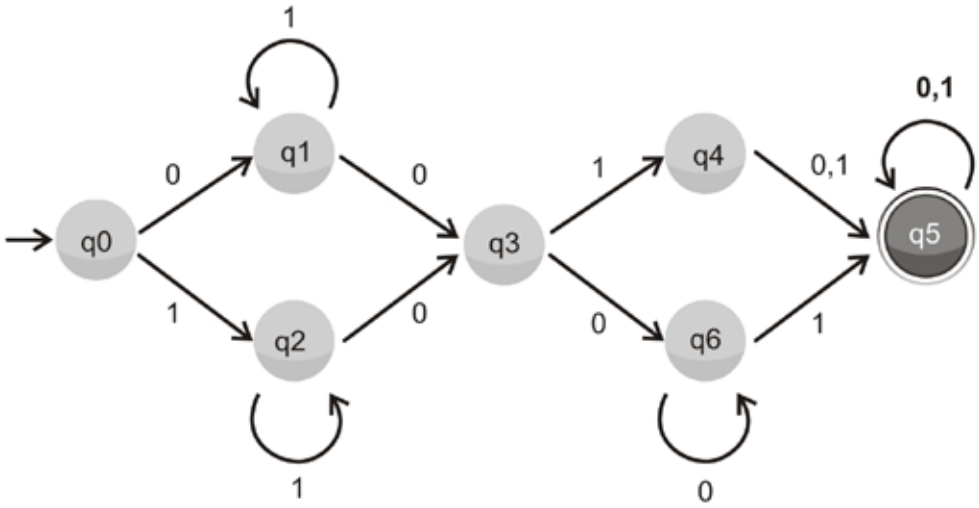
a)



b)



c)



2) Converta as seguintes expressões regulares em autômatos finitos:

a) (00)*(0+1)

b) (0+1)(0+1)

c) (01+1)*

Capítulo 3 – Gramáticas Livres de Contexto

Até este ponto do curso, vimos dois modelos matemáticos para representar linguagens: autômatos finitos (e suas variações) e expressões regulares. Vimos também que os dois modelos são equivalentes e as linguagens representadas por eles são chamadas de linguagens regulares. Neste capítulo, vamos mostrar um modelo diferente e também mais poderoso do que aqueles vistos – as gramáticas livres de contexto.

Lembre-se que, quando falarmos que um modelo é mais poderoso que outro, queremos dizer que ele consegue representar mais linguagens. No caso, as gramáticas livres de contexto reconhecem todas as linguagens regulares e mais algumas. As linguagens que as gramáticas livres de contexto podem representar são chamadas de linguagens livres de contexto. O diagrama abaixo, que é parte da chamada Hierarquia de Chomsky, representa a relação entre as duas classes de linguagens:

Vamos começar a estudar as gramáticas por um exemplo simples. Depois apresentamos a definição formal de gramáticas livres de contexto, bem como alguns conceitos importantes ligados às gramáticas.

3.1 Um Exemplo Simples

Cada modelo visto tem a sua própria maneira de representar linguagens. Os autômatos reconhecem as cadeias que estão na linguagem e rejeitam as que não estão; já as expressões regulares são um “formato” geral que todas as cadeias da linguagem obedecem. E as gramáticas livres de contexto, como funcionam?

As gramáticas livres de contexto possuem regras para formar cadeias, de modo que apenas as cadeias que são da linguagem podem ser formadas usando as regras, enquanto as cadeias que não são da linguagem não podem ser formadas de maneira nenhuma pelas regras.

Para entender melhor, vamos descrever a partir de agora a criação de uma gramática para uma linguagem que não é regular: a linguagem das cadeias na forma 0^n1^n (n ocorrências do símbolo 0 seguidas de n ocorrências do símbolo 1), para todo n > 0. Exemplos de cadeias dessa linguagem são dados abaixo:

01

0011

000111

00001111

etc.

Antes de construirmos uma gramática para a linguagem em questão, você precisa entender como funcionam as regras nas gramáticas livres de contexto. As regras usam símbolos auxiliares, chamados não-terminais, que servem de ponto de partida para gerar alguma cadeia. No caso que estamos estudando, basta usar um símbolo não-terminal X (poderia ser outra letra qualquer). Ele será o ponto de partida para gerar cadeias completas da linguagem. Os símbolos 0 e 1, por sua vez, são chamados de símbolos terminais.

Agora, vamos discutir uma “receita” para gerar cadeias da forma 0^n1^n, que usaremos para escrever as regras da gramática. Em primeiro lugar, veja que a menor cadeia da linguagem é a cadeia 01. Precisamos de uma regra que diga que essa é uma cadeia que pode ser gerada. Como X é o símbolo que escolhemos usar para gerar as cadeias da linguagem, precisamos dizer que X pode gerar 01 e isso é feito com a seguinte regra:

$$X \rightarrow 01$$

Reveja acima que, na definição da linguagem, cada cadeia precisa ter, para cada símbolo 0, um símbolo 1 correspondente. Isso nos dá uma idéia de como podemos usar uma cadeia da linguagem para gerar outra cadeia maior da linguagem: basta acrescentar um 0 no

início da cadeia e um 1 (correspondente ao zero) no fim da cadeia. A regra que representa isso seria simplesmente:

$$X \rightarrow 0X1$$

Podemos pensar essa regra também de outra maneira: para criar uma cadeia da linguagem, podemos colocar um 0 no início e um 1 no fim da cadeia e depois criar outra cadeia da linguagem entre estes dois símbolos. Para gerar essa cadeia intermediária, continuamos o processo recursivamente usando as regras de X.

Apenas com as duas regras acima conseguimos gerar todas as cadeias desejadas. Portanto, a gramática para reconhecer a linguagem das palavras na forma 0^n1^n é composta pelo conjunto de regras:

$$X \rightarrow 01$$

$$X \rightarrow 0X1$$

Vamos agora partir dessas regras para gerar uma palavra, de modo a conferirmos se palavra obedece à descrição dada inicialmente. Para gerar uma palavra, iniciamos no símbolo X e seguimos um processo em que repetidamente substituímos uma ocorrência de X pelo lado direito de alguma das regras. Os demais símbolos da cadeia que está sendo formado são mantidos inalterados. Esse processo é chamado de derivação de uma cadeia.

Na figura a seguir, mostramos uma derivação em que usamos a segunda regra e depois a primeira regra. Com isso, a palavra 0011 foi gerada pela gramática e, de fato, essa palavra era uma das que esperávamos gerar, pois tem dois 0's seguidos de dois 1's.

$$X \Rightarrow 0X1 \Rightarrow 0011$$

Para gerar outras palavras, basta aplicar regras diferentes a cada etapa. Veja como isso acontece abaixo em dois outros exemplos de derivações, para as palavras 01 e 000111. Depois, tente você mesmo fazer uma derivação para outra palavra.

$$X \Rightarrow 01$$

$$X \Rightarrow 0X1 \Rightarrow 00X11 \Rightarrow 000111$$

O fato de termos chegado às cadeias 0011, 01 e 000111 sem não-terminais (sem X, neste caso) significa que elas fazem parte da linguagem que a gramática representa. Outras cadeias, como 011, 1 e 0101 não podem ser derivadas pelas regras e, portanto, não fazem parte da linguagem. Esta é uma maneira mais sutil de representar linguagens do que com autômatos. Enquanto os autômatos são usados para testar qualquer palavra dada diretamente, nas gramáticas costumamos partir das regras para descobrir as palavras que são válidas.

Pronto, agora que vimos este primeiro exemplo de gramática e entendemos como ela funciona, vamos apresentar a definição formal de gramáticas livres de contexto em geral.

3.2 Definição Formal

Uma gramática livre de contexto G é uma 4-tupla (quatro elementos ordenados) da forma:

$$G = (V, T, P, S)$$

A seguir, explicamos os quatro elementos que formam uma gramática:

- **V** é o alfabeto que contém os símbolos **não-terminais** ou **variáveis**, que são os símbolos auxiliares no processo de geração de cadeias (como o símbolo X do exemplo). Em uma gramática bem ajustada, cada não-terminal tem regras que permitem que ele gere uma cadeia ou parte de uma cadeia da linguagem.
- **T** é o alfabeto dos símbolos **terminais**, que são os símbolos usados nas cadeias da linguagem. O conjunto T é equivalente ao alfabeto de entrada Σ dos autômatos finitos.
- **P** é o conjunto das **regras** ou **produções** da gramática, que servem para gerar as palavras da linguagem. Cada regra tem a forma $N \rightarrow a$, onde:
  - o N é chamado de **cabeça** da produção e consiste simplesmente

em um símbolo não-terminal (ou seja, $N \in V$).

- o a é chamado de **corpo** da produção e consiste em qualquer cadeia formada de símbolos terminais e/ou não-terminais, inclusive a cadeia vazia ε.

Podemos ler a produção $N \rightarrow a$ como "N gera a". Dizemos também que esta é uma produção "de N", porque N é a cabeça.

- **S** é o símbolo não-terminal de início ($S \in V$), pois é usado para iniciar as derivações de cadeias da linguagem.

Já podemos definir a gramática vista no exemplo anterior (para representar as palavras 0^n1^n) de maneira completa. Chamando aquela gramática de G_1, teríamos a seguinte definição:

$$G_1 = (\{X\}, \{0,1\}, \{X \rightarrow 01, X \rightarrow 0X1\}, X)$$

Em casos como este, em que há várias produções com uma mesma cabeça, podemos representar uma só vez a cabeça e separar os corpos pela barra vertical "|", que pode ser lida como "ou". A definição ficaria:

$$G1 = (\{X\}, \{0,1\}, \{X \rightarrow 01 \mid 0X1\}, X)$$

Vamos utilizar algumas convenções nas gramáticas de exemplo deste material, de forma a facilitar o entendimento:

- As letras maiúsculas representarão símbolos não-terminais.
- Todos os outros símbolos serão considerados terminais, automaticamente.
- A cabeça da primeira produção será o símbolo inicial.

Com estas convenções, você pode analisar apenas as produções da gramática e já perceber facilmente quem são os não-terminais, os terminais e qual o símbolo inicial. Isso permite que, nos exemplos, sejam mostradas apenas as produções, deixando o restante da definição implícita.

No entanto, queremos ressaltar que essas convenções têm o objetivo de facilitar o entendimento dos exemplos, mas elas *não* são obrigatórias. Não deixa de ser uma gramática livre de contexto se usar uma letra maiúscula para um terminal, por exemplo.

Já trabalhamos muito em cima de um mesmo exemplo de gramática, então vamos agora ver outro exemplo, seguindo as

convenções acima. Essa segunda gramática será chamada G_2. Ela representa cadeias que têm a forma de expressões aritméticas com os operadores de adição (símbolo +) e multiplicação (símbolo x). Para simplificar, vamos trabalhar apenas com números binários. Exemplos de cadeias que ela gera são dados abaixo:

10

1+100

0x1+1

Vamos começar mostrando apenas o conjunto P_2, que representa as produções da gramática G_2, depois formaremos a 4-tupla da gramática.

P_2:

$$E \rightarrow N \mid E+E \mid ExE$$

$$N \rightarrow 0 \mid 1 \mid 0N \mid 1N$$

Considerando que a gramática segue as nossas convenções, podemos identificar facilmente qual o conjunto de não-terminais da gramática: {E, N}, sendo E o símbolo de início, pois ele aparece na cabeça da primeira produção. Os símbolos terminais são os símbolos restantes, ou seja: {+, x, 0, 1}. Assim, a definição completa da gramática G_2 seria:

$G_2 = (\{E, N\}, \{+, x, 0, 1\}, P_2, E)$, onde P_2 é o conjunto dado antes

Veja que, por conta das convenções adotadas, nós conseguimos olhar apenas as produções e já inferir o restante da definição. Por esse motivo, nos exemplos dados adiante, mostraremos apenas as produções e esperamos que você consiga ter o entendimento completo da gramática (seus não-terminais, terminais e símbolo de início).

A seguir, veremos mais alguns conceitos importantes no estudo das gramáticas livres de contexto.

3.3 Conceitos

Um importante conceito relacionado às gramáticas já foi apresentado antes: o conceito de **derivação**. Porém, vamos complementar a apresentação dada antes, descrevendo de maneira mais genérica o processo pelo qual você pode derivar cadeias da linguagem:

1. Escreva o símbolo não-terminal de início. Este símbolo sozinho será a cadeia inicial com a qual você vai trabalhar.
2. Escolha uma ocorrência de um não-terminal N qualquer na cadeia e escolha alguma produção de N. Na cadeia, substitua a ocorrência de N pelo corpo da produção.
3. Repita o passo 2 até que não reste nenhuma variável.

Vamos praticar essa descrição derivando algumas cadeias da gramática G_2 definida na seção anterior e reproduzida abaixo.

G_2 (produções):

$$E \rightarrow N$$
$$\mid E+E$$
$$\mid ExE$$

$$N \rightarrow 0$$
$$\mid 1$$
$$\mid 0N$$
$$\mid 1N$$

No passo 1 do procedimento dado acima, dissemos para partir do símbolo de início, que é o símbolo E. Então, a derivação começa simplesmente assim:

E

Agora, pelo passo 2, temos que escolher um símbolo não-terminal e uma produção dele. Neste momento, só podemos escolher o próprio símbolo E na cadeia, mas quanto à produção, podemos escolher qualquer uma das três de E. Vamos escolher E → E+E. Portanto, vamos trocar a ocorrência de E (em negrito abaixo) por E+E. Colocamos o símbolo $\Rightarrow$ para indicar "deriva".

$$\begin{aligned} & E \\ \Rightarrow\ & E+E \end{aligned}$$

Bem, agora temos duas ocorrências do símbolo E. Podemos escolher qualquer uma delas para o próximo passo da derivação. Vamos escolher a segunda ocorrência de E (em negrito) e vamos aplicar a produção E → N:

$$\begin{aligned} & E+E \\ \Rightarrow\ & E+N \end{aligned}$$

Agora, podemos escolher livremente entre E ou N e depois usar quaisquer das produções destes símbolos. Vamos escolher N (em negrito) e usar a sua produção N → 0:

$$\begin{aligned} & E+N \\ \Rightarrow\ & E+0 \end{aligned}$$

Agora você já deve estar se acostumando com o processo, então vamos dizer apenas a produção que vamos aplicar. Se escolhermos aplicar E → N teremos:

$$\begin{aligned} & E+0 \\ \Rightarrow\ & N+0 \end{aligned}$$

Agora vamos aplicar N → 1.

$$\begin{aligned} & N+0 \\ \Rightarrow\ & 1+0 \end{aligned}$$

Neste ponto, só há símbolos terminais, então damos por encerrada a derivação. A derivação completa pode ser expressa assim:

$$
\begin{aligned}
& E \\
\Rightarrow\ & E+E \\
\Rightarrow\ & E+N \\
\Rightarrow\ & E+0 \\
\Rightarrow\ & N+0 \\
\Rightarrow\ & 1+0
\end{aligned}
$$

Veja que geramos a palavra **1+0**, o que é coerente com a descrição que demos da linguagem (cadeias que parecem expressões aritméticas). Uma maneira de omitir os passos intermediários da derivação e dizer que E deriva (ou gera) direta ou indiretamente a cadeia **1+0** é colocando um asterisco sobre a seta assim:

$$E \overset{*}{\Rightarrow} 1+0$$

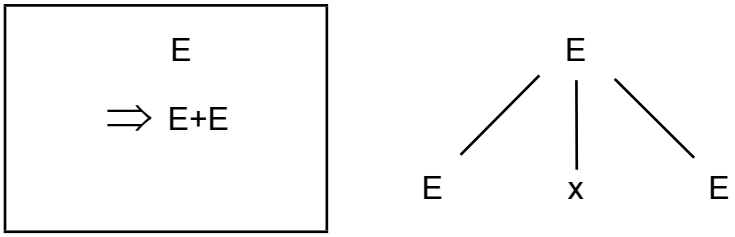
Outra maneira de chegarmos a uma palavra da linguagem é construindo uma **árvore de derivação** (ou **árvore de sintaxe concreta**). Este tipo de árvore tem as seguintes características:

1. Seus nós são símbolos terminais ou não-terminais.
2. A sua raiz (representada no topo) é o símbolo inicial.
3. Um símbolo não-terminal N é um nó que possui filhos. Se $N \rightarrow a_1 a_2 \ldots a_n$ for uma produção, então um nó N pode aparecer com os filhos a_1, a_2, ... e a_n, nesta ordem.
4. Os símbolos terminais e a cadeia ε não têm filhos (são folhas da árvore).

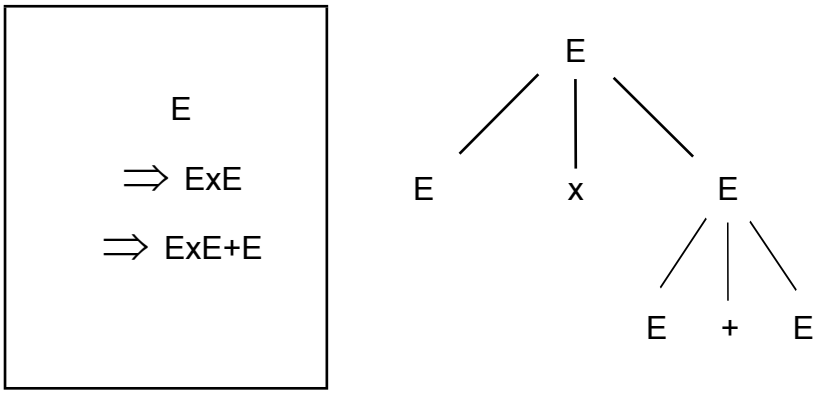
Uma árvore de derivação está intimamente ligada a uma derivação de uma palavra. Aliás, é fácil passar de uma derivação para a árvore ou da árvore para uma derivação, considerando uma mesma cadeia. A seguir, mostramos a construção em paralelo de uma derivação e sua árvore de derivação correspondente usando a gramática G_2.

No início da derivação e também da árvore, temos apenas o símbolo E. Vamos começar aplicando a produção E→ExE. Abaixo mostramos

como fica a derivação e a árvore. Veja que a ocorrência de E que foi substituída (em negrito) fica como pai do corpo da produção.

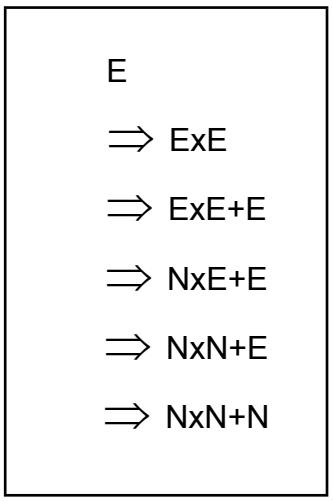


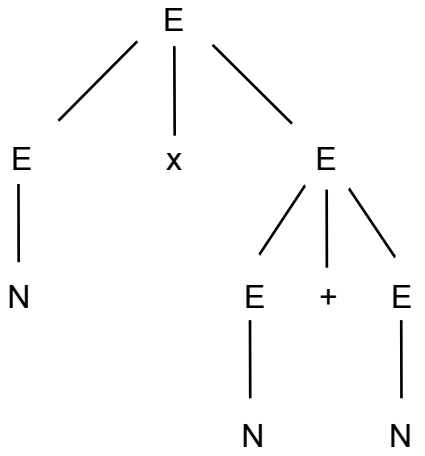
Veja que as folhas da árvore, analisadas da esquerda para a direita, representam exatamente a última cadeia da derivação, que é ExE. No próximo passo, vamos aplicar a produção E → E+E na segunda ocorrência de E dessa cadeia. O resultado é exibido abaixo:



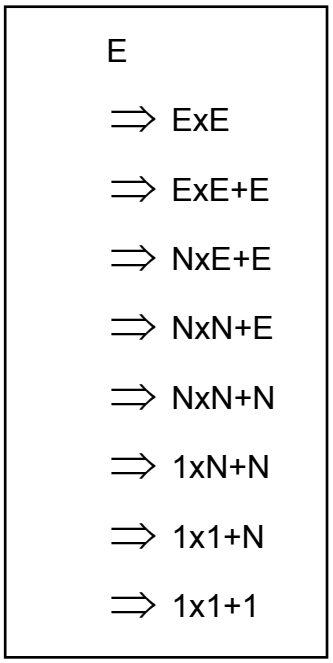
Novamente, vemos que as folhas da árvore representam mais uma vez a última cadeia da derivação (ExE+E). Porém, percebemos aqui uma diferença entre a derivação e a árvore: na derivação, sempre repetimos toda a cadeia, mesmo que uma parte dela não mude, enquanto na árvore, apenas adicionamos filhos na parte que mudou.

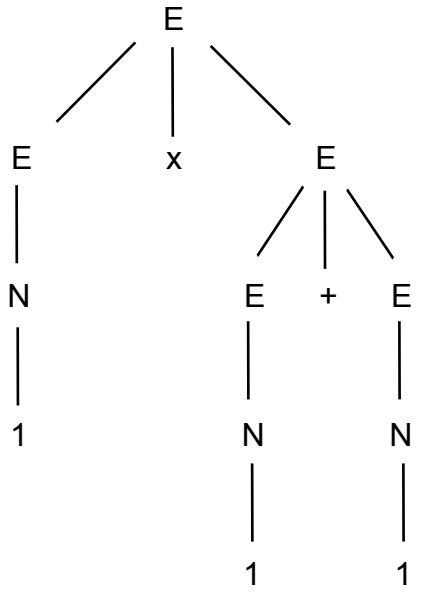
Vamos agora adiantar um pouco o exemplo, mostrando três passos da derivação em que aplicamos a produção E → N em cada ocorrência de E da cadeia. O resultado é mostrado abaixo.





Neste ponto, temos três ocorrências do símbolo N. Vamos aplicar as produções N → 1 em cada um deles:





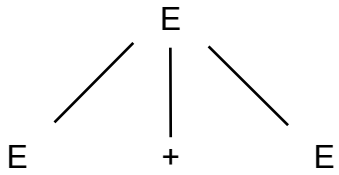
É fácil ver na derivação que chegamos à cadeia 1x1+1 da linguagem. A princípio, pode parecer um pouco mais difícil perceber, mas a árvore de derivação também mostra este mesmo resultado. Basta analisarmos as folhas da árvore da esquerda para a direita. Confira que a cadeia formada assim é realmente 1x1+1.

Se você já entendeu o conceito de árvore de derivação, podemos passar para o último conceito. Vamos falar de **gramática ambígua**, que é como chamamos uma gramática que possui duas (ou mais) árvores de derivação distintas para alguma palavra. Veja que a definição de gramática ambígua não exige que existam duas árvores de derivação para *todas* as palavras – basta existir duas árvores de derivação para *uma* palavra qualquer da linguagem.

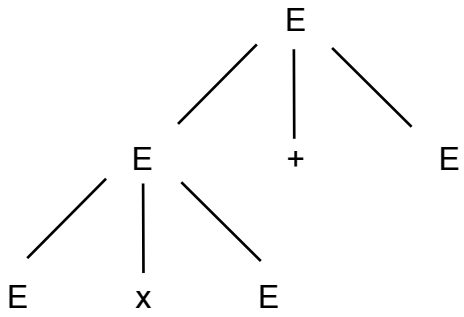
Para ilustrar, vamos usar novamente a gramática G_2: será que ela é ambígua? Ou seja, será que existe uma palavra que podemos construir com duas árvores de derivação distintas? Bem, vamos começar analisando a palavra 1x1+1. Já criamos uma árvore de derivação para ela há pouco, será que é possível criarmos outra árvore para 1x1+1?

A resposta para todas estas perguntas é sim. Vamos mostrar a justificativa a seguir, mas, se você quiser, pode parar um pouco a leitura para tentar justificar por conta própria.

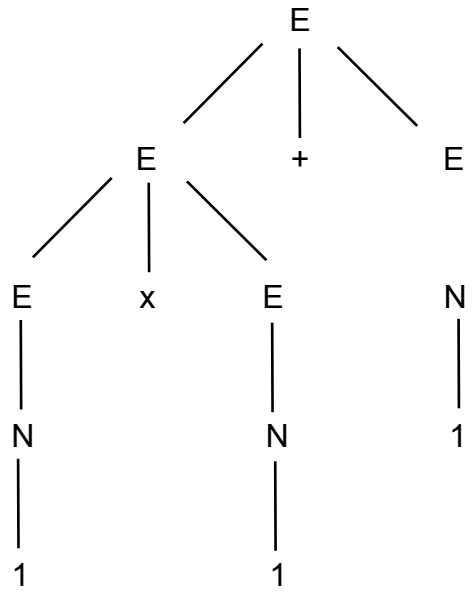
Veja que a árvore de derivação que criamos antes para a palavra 1x1+1 começa pela produção E → ExE. Vamos mostrar agora que é possível gerar a mesma palavra começando pela produção E → E+E. Ou seja, a árvore começa assim:



Perceba, com base na palavra 1x1+1, que precisamos gerar 1x1 antes do símbolo +. Naturalmente, isso será feito a partir da folha E esquerda, aplicando a produção E → ExE.



Neste ponto, falta apenas gerar os símbolos 1 a partir de cada símbolo E sem filhos. Para isso, aplicamos E → N em cada um dos três E livres (sem filhos), ficando com três N livres. Depois, aplicamos N → 1 em cada N livre. O resultado final é mostrado abaixo.



Analisando as folhas da esquerda para a direita, confira que a árvore forma a palavra 1x1+1. Também compare com a árvore dada antes para a mesma palavra e confirme que são diferentes. Assim, por causa de uma única palavra, podemos concluir que a gramática é ambígua.

Não existe um algoritmo para identificar se uma gramática livre de contexto qualquer é ambígua. Por esse motivo, identificar ambigüidades em gramáticas livres de contexto não é uma tarefa muito fácil. Você precisa estar bem acostumado com o conceito de gramática livre de contexto e precisa analisar muito bem a gramática dada. Se a gramática for ambígua, você vai acabar encontrando uma palavra que possa ser gerada por duas árvores distintas.

3.4 Importância

Para concluir, gostaríamos de falar rapidamente da importância prática das gramáticas livres de contexto. Esse modelo foi proposto para tentar representar a linguagem natural (a língua portuguesa, por exemplo), onde classes de palavras são usadas para formar certas frases, segundo certas regras, e frases menores são usadas para construir outras maiores, num processo recursivo.

Posteriormente, as gramáticas passaram a ser usadas para descrever linguagens computacionais, em especial, as linguagens de programação. No caso, as gramáticas são usadas para descrever o formato dos programas válidos nas linguagens de programação. Todas

as linguagens modernas são definidas a partir de alguma gramática livre de contexto.

Com base em uma gramática livre de contexto de uma linguagem computacional, existem várias técnicas que podem ser usadas para construir um compilador para a linguagem. Um importante requisito, no entanto, é que a gramática não seja ambígua, pois deixaria o compilador "em dúvida" na hora de montar uma árvore de derivação. A árvore, por sua vez, também é muito importante no processo de compilação.

Não nos aprofundaremos nesses assuntos, mas é importante ter consciência da relação deles com as gramáticas livres de contexto.

Atividades e Orientações para estudo

1) Com base na gramática G_1 dada antes, mostre uma árvore de derivação para a palavra **000111**.

2) Com base na gramática G_2 dada antes, mostre uma derivação e uma árvore de derivação para a palvra **000111**.

3) Considere a gramática G_3 definida abaixo. Diga se ela gera as palavras abaixo. Construa uma derivação e uma árvore de derivação para as palavras que a gramática puder gerar.

G_3:

$$S \rightarrow aSa \mid bSb \mid T$$

$$T \rightarrow a \mid b \mid \varepsilon$$

a) ε

b) ab

c) aba

d) baa

e) babab

4) Prove que a gramática G_4 definida abaixo é ambígua. (Dica: use a cadeia abc para provar).

G_4:

$$S \rightarrow XC$$

$$| AY$$

$$X \rightarrow aXb \mid \varepsilon$$

$$Y \rightarrow bYc \mid \varepsilon$$

$$C \rightarrow cC \mid \varepsilon$$

$$A \rightarrow aA \mid \varepsilon$$

Resumo

Neste capítulo, aprendemos sobre as gramáticas livres de contexto, que representam linguagens de maneira diferente dos modelos vistos em capítulos passados. As gramáticas livres de contexto também são capazes de representar mais linguagens do que os modelos vistos antes. Chamamos de linguagem livre de contexto toda linguagem representada com esse tipo de gramática.

Vimos como definir formalmente as gramáticas livres de contexto e aprendemos dois mecanismos parecidos para gerar palavras de uma gramática: derivação e árvores de derivação. Aprendemos sobre gramáticas ambíguas e, por fim, discutimos a importância das gramáticas livres de contexto.

Fascículo 3

Depois de trabalharmos com outros modelos matemáticos no fascículo 2, voltaremos a trabalhar com autômatos neste fascículo. Primeiro veremos o autômato com pilha, para linguagens livres de contexto. Depois, veremos um autômato especial conhecido como Máquina de Turing, que é o modelo de computação mais poderoso já pensado. Devido à importância desse modelo, dedicamos o capítulo final à análise dos seus limites – quais linguagens a Máquina de Turing pode e quais ela não pode calcular sem entrar em loop infinito. Veremos que os limites teóricos da Máquina de Turing são considerados limites teóricos também para os nossos mais modernos computadores.

Capítulo 1 – Autômatos com Pilha (APs)

No fascículo 1, vimos três tipos distintos de autômatos finitos que possuem o mesmo poder de representação de linguagens: AFD, AFND e AFND-ε. Os três modelos são equivalentes e as linguagens que elas representam formam a classe das linguagens regulares. Neste capítulo, veremos um tipo de autômato mais poderoso do que os autômatos finitos – o **autômato de pilha** (AP). Este modelo é capaz de reconhecer a classe das **linguagens livres de contexto**, que inclui toda a classe das linguagens regulares.

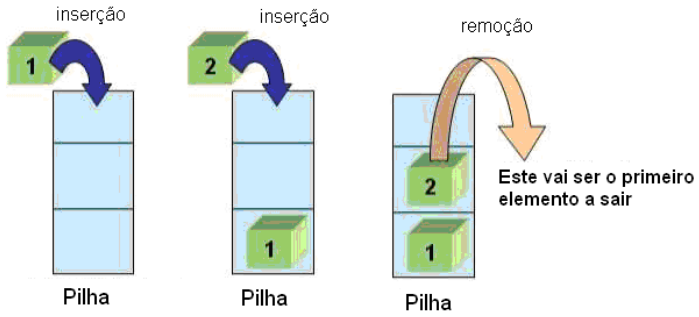
Já estudamos a classe das linguagens livres de contexto no final do fascículo 2. Você deve lembrar que foi assim que chamamos a classe das linguagens que podem ser representadas por meio de gramáticas livres de contexto. Por isso, entenda o autômato de pilha como um modelo de poder *equivalente* ao das gramáticas livres de contexto. Porém, como o nome sugere, o princípio de funcionamento do autômato de pilha é mais parecido com o dos autômatos finitos que citamos acima do que propriamente com as gramáticas.

Você entenderá melhor isso quando virmos o funcionamento dos autômatos com pilha em detalhes, na seção a seguir.

1.1 Princípio de Funcionamento

Os autômatos com pilha (APs) são como os autômatos finitos com transições vazias (AFND-ε), mas têm um componente extra chamado **pilha**. A pilha serve como uma memória adicional infinita (por isso não podemos chamar um autômato de pilha de autômato “finito”). Nela, o autômato pode escrever símbolos na pilha para ler mais tarde e isso dá ao autômato um poder adicional em relação aos AFND-ε.

Para explicar o nome “pilha”, pense no que é uma “pilha de pratos” (ou uma “pilha de livros”). Esse é o nome que damos a um conjunto de pratos postos um sobre o outro. Geralmente, se queremos colocar mais um prato na pilha, colocamos ele no topo, acima dos demais pratos. Também quando vamos remover um prato, geralmente, tiramos aquele que está sobre o topo. Como resultado, o prato que colocamos por *último* acaba sendo o *primeiro* a ser retirado. De modo geral, em Computação, chamamos de pilha toda estrutura que segue essa regra de inserção e remoção. A figura abaixo ilustra isso.



Os autômatos que vamos ver usam uma pilha para guardar símbolos especiais, que chamaremos **símbolos da pilha**. Os autômatos só podem inserir e remover esses símbolos de uma mesma extremidade (como se fosse o “topo” de uma pilha de pratos). Assim, o *último* símbolo que é escrito fica na *primeira* posição, como esperado. Vamos considerar que a pilha dos APs não inicia vazia – ela começa automaticamente com um **símbolo de início**.

Nos APs, o primeiro símbolo da pilha serve para decidir para qual estado o autômato mudará. Relembre que, nos autômatos vistos antes, as mudanças de estado (transições) dependiam apenas do próximo símbolo da palavra de entrada. Agora, no AP, as mudanças de estado vão depender de um *par* de condições: o próximo símbolo da entrada e o próximo símbolo da pilha. Semelhante aos AFND-ε, também podemos usar o símbolo especial ε para indicar que não precisa ler símbolo (da entrada ou da pilha) para mudar de estado.

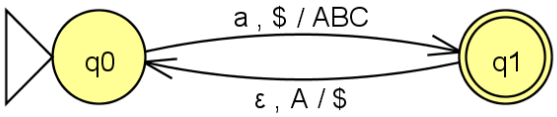
Nos autômatos finitos, o símbolo lido em uma transição era automaticamente removido da entrada e, depois disso, o autômato só podia ver o próximo símbolo da entrada. No AP é parecido: os símbolos lidos da entrada e da pilha são, ambos, automaticamente removidos quando o autômato muda de estado. Adicionalmente, depois das remoções, o autômato também insere na pilha uma cadeia de símbolos. Pode-se usar a cadeia vazia ε para indicar que nada vai ser escrito.

A representação gráfica de um AP é parecida com a dos outros autômatos vistos. O estado inicial e os estados de aceitação são representados da mesma maneira (reveja a apostila 1 caso não lembre). Porém, a seta de transição entre estados precisa conter toda a informação que apresentamos. Por isso, ela recebe rótulos na forma:

s, X / $Y_1...Y_k$, onde:

- **s** é o símbolo que será lido da palavra (ou ε para não ler nada)
- **X** é o símbolo que será lido da pilha (ou ε para não ler nada)
- **$Y_1...Y_k$** é a cadeia de símbolos que será escrita na pilha (ou ε para não escrever nada), considerando que os símbolos serão empilhados na ordem inversa (de Y_k para Y_1), deixando Y_1 no topo

A figura a seguir ilustra um AP simples que aceita cadeias de entrada formadas com o alfabeto {a,b}.



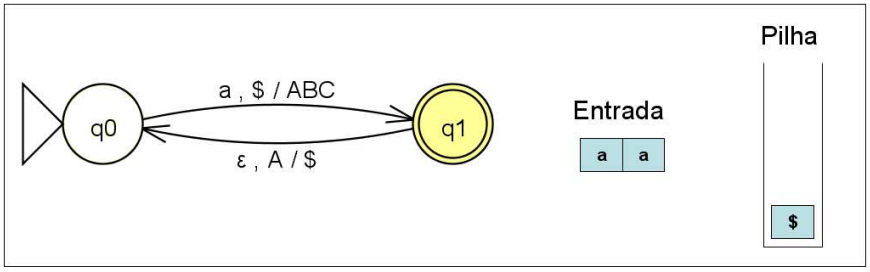
Veja que o autômato tem uma transição de q_0 para q_1 que só pode ser realizada se duas condições forem verdadeiras: o próximo símbolo da palavra de entrada é “a” e o próximo símbolo da pilha é “$”. Quando ambas forem verdadeiras no estado q_0, o autômato muda para q_1 e escreve os símbolos ABC na pilha na ordem inversa (deixando A no topo). A outra transição (de q_1

para q_0) não lê símbolo da entrada, mas exige que o próximo símbolo da pilha seja A. Se for, o autômato muda do estado q_1 para o estado q_0 e escreve $ na pilha.

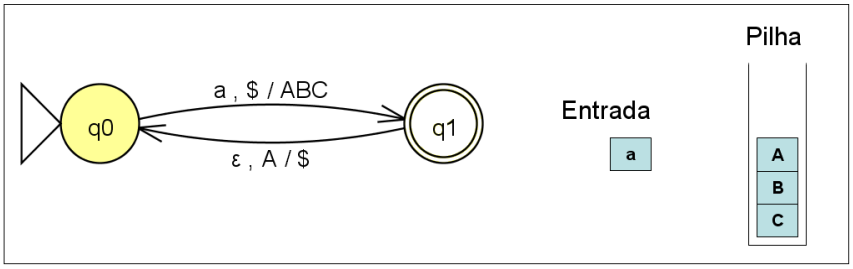
Vamos apresentar a seguir como um AP processa uma cadeia de entrada para indicar se ela faz parte da linguagem ou não.

1.2 Computação e Aceitação no Autômato com Pilha

Vamos começar analisando o comportamento do AP anterior quando ele recebe a palavra de entrada aa. Vamos considerar que o autômato tem $ como símbolo inicial da pilha. Observe também que a figura indica também que q_0 é o estado inicial. Portanto teríamos a seguinte configuração do autômato, no início:

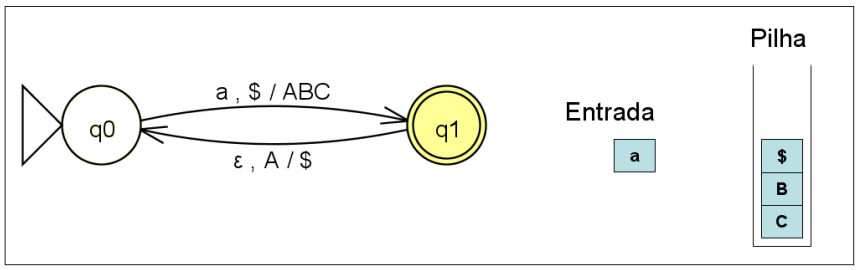


Na figura acima, o autômato está em seu estado inicial q_0, a pilha contém apenas o símbolo $ e a entrada ainda contém toda a cadeia aa. Nessa configuração, temos a como próximo símbolo da entrada e $ como próximo símbolo da pilha. Vemos, então, que é possível fazer a mudança de q_0 para q_1 resultando na seguinte configuração:

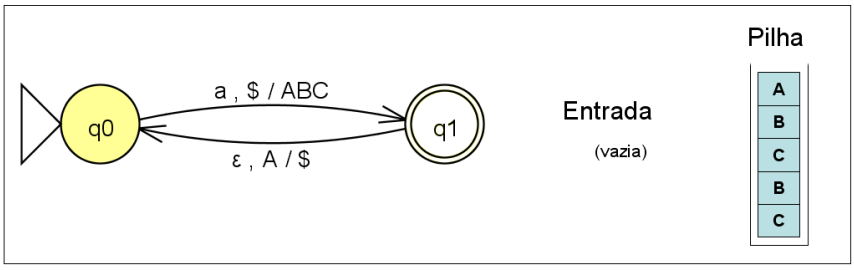


Para entender como chegamos aqui, lembre-se que os símbolos a (da esquerda) e $ foram removidos após serem lidos. Além disso, a cadeia ABC foi

inserida (de trás para frente) na pilha. Partindo da configuração representada acima, podemos fazer a transição de volta para o estado q_0 sem ler símbolo da entrada, mas lendo o A que está no topo da pilha. O resultado está representado abaixo:



Veja que a transição de q_1 para q_0 causou a escrita do símbolo $ na pilha. Assim, temos novamente um símbolo a para ser lido da entrada e um símbolo $ para ser lido da pilha e podemos fazer outra transição para q_1:



Neste ponto, a cadeia de entrada já foi completamente lida e o autômato chegou a um estado de aceitação. Como acontecia nos autômatos finitos, essas duas condições nos permitem concluir que a cadeia aa é *aceita*, ou seja, faz parte da linguagem que o autômato representa! Já o fato da pilha ainda ter símbolos *não é* levado em consideração quando vamos definir se a cadeia é ou não aceita.

Esperamos que, ao chegar neste ponto da leitura, você já tenha entendido bem a idéia do reconhecimento de uma cadeia em um AP. O que veremos agora é que aqueles mesmos passos de reconhecimento que mostramos podem ser representados de maneira mais sucinta. Para isso, representamos cada configuração do autômato simplesmente como triplas na forma (q, $s_1...s_n$, $X_1...X_p$) onde:

- q é o estado atual do autômato

- $s_1...s_n$ é o restante (ainda não lido) da cadeia de entrada
- $X_1...X_p$ é uma cadeia que reflete o conteúdo da pilha, considerando que o símbolo da esquerda X_1 representa o "topo" da pilha (como se a pilha tivesse sido "deitada" para a esquerda)

Essas triplas são chamadas de **descrições instantâneas** ou **IDs** (do inglês *instantaneous description*) do autômato. Apesar do nome especial, essas triplas são apenas uma maneira mais formal e sucinta de representar aquilo que chamamos de "configuração" do autômato em um dado instante.

Vamos considerar um AP que está na ID genérica que acabamos de mostrar $(q, s_1...s_n, X_1...X_p)$. Se houver no autômato uma transição saindo do estado q para o estado r, e na transição tiver o rótulo "$s_1, X_1 / Y_1...Y_k$ ", então o autômato pode mudar de estado e de configuração. Aconteceriam três mudanças na ID:

- trocamos o estado q por r,
- removemos o s_1 da entrada e o X_1 da pilha,
- acrescentamos $Y_1...Y_k$ na esquerda (topo) da pilha.

A nova ID seria a tripla $(r, s_2...s_n, Y_1...Y_kX_2...X_p)$. Representamos que esta é uma mudança de configuração válida assim:

$$(q, s_1...s_n, X_1...X_p) \vdash (r, s_2...s_n, Y_1...Y_kX_2...X_p)$$

(ao diagramador: "|-" é um símbolo só, como se fosse o símbolo "$\perp$" rotacionado)

Podemos também representar que de uma ID chegamos a outra depois de uma quantidade qualquer de mudanças válidas usando um asterisco:

$$(q, w, \alpha) \vdash^* (r, x, \beta)$$

Vamos agora usar as IDs para definir quando uma cadeia é aceita por um AP. Para testar se o autômato aceita uma cadeia w, iniciamos com uma ID

formada pelo estado inicial, a entrada w e a pilha contendo o símbolo de início. A partir dela, vamos sucessivamente movimentando o autômato por uma seqüência de IDs. Essa seqüência válida de IDs é chamada de **computação** no autômato com pilha.

Como os APs são não-determinísticos, pode haver mais de uma computação para uma mesma cadeia. Algumas computações podem chegar a um estado final e outras não. No entanto, a cadeia é considerada aceita se existir pelo menos *uma* computação que leve a um estado final após ler toda a cadeia. Ou seja, alguma computação deve chegar a uma ID na forma (q_x, ε, α), em que q_x é algum estado de aceitação, a entrada está vazia e α é um conteúdo qualquer da pilha. Essa é a definição chamada de **aceitação por estado final**.

Para ilustrar o que acabamos de explicar, vamos mostrar a computação que descreve aquele mesmo processo de reconhecimento da cadeia aa que representamos antes:

$(q_0, aa, \$) \vdash (q_1, a, ABC) \vdash (q_0, a, \$BC) \vdash (q_1, \varepsilon, ABCBC)$

Perceba, primeiramente, que cada ID está diretamente relacionada a cada figura mostrada do início da seção. Em segundo lugar, veja que a computação chega a uma ID cujo estado é de aceitação (q_1) e cuja entrada está vazia (ε). Concluímos que, pela definição de aceitação por estado final, a cadeia aa foi aceita!

Compare com a computação da palavra ab no mesmo autômato:

$(q_0, ab, \$) \vdash (q_1, b, ABC) \vdash (q_0, b, \$BC)$

A computação pára no estado q_0 com o símbolo b na entrada e $ na pilha, pois não existe nenhuma seta saindo de q_0 com estes dois símbolos. Como nem a entrada foi toda lida nem o estado q_0 é de aceitação, concluímos que a palavra não é aceita.

Veja que a definição de aceitação dada depende do estado onde o AP pára, mas não depende do conteúdo da pilha. O primeiro exemplo que demos ilustra

bem isso, pois a pilha não terminou vazia e mesmo assim a palavra foi aceita. Existe uma definição alternativa, chamada de **aceitação por pilha vazia**, que desconsidera o estado onde o AP pára, mas considera obrigatório que a pilha termine vazia para a palavra ser aceita. Portanto, dizemos que a palavra é aceita se existir uma computação que chegue a uma ID na forma (q, ε, ε), onde q é um estado qualquer.

Sempre que um AP for apresentado, é muito importante deixar claro se ele foi construído pensando em aceitação por estado final ou em aceitação por pilha vazia. Isso faz diferença na linguagem (conjunto de palavras) que ele representa. Em geral, assuma que os APs que daremos usam a definição de aceitação por estado final, por ser mais parecida com a definição de aceitação dos autômatos finitos do 1º fascículo (AFD, AFND e AFND-ε). Se algum AP usar aceitação por pilha vazia, vamos indicar isso claramente.

Na próxima seção, damos a definição matemática formal dos autômatos com pilha. É importante que você tenha compreendido bem o que vimos até aqui antes de passar para ela. (Se não compreendeu, tente reler as seções 1.1 e 1.2 ou entre em contato com seu tutor).

1.3 Definição Formal

Os autômatos com pilha são 7-tuplas (ou seja, estruturas contendo uma seqüência de 7 elementos) na forma:

P = (Q, Σ, Γ, δ, s, $, F) , onde:

- **Q** é conjunto finito de **estados**, como nos autômatos finitos
- **Σ** (sigma) é o conjunto dos **símbolos de entrada**, como nos autômatos finitos
- Γ (gama) é o conjunto dos **símbolos da pilha**, que são os únicos símbolos que podem ser escritos na pilha

- **δ** (delta) é a **função de transição**, parecida com a dos autômatos finitos (abaixo descreveremos melhor essa função)

- **s** é o elemento de Q que serve de **estado inicial**, como também ocorre nos autômatos finitos

- **$** é um elemento de Γ usado como **símbolo inicial da pilha**

- **F** é o conjunto de **estados de aceitação** ou estados finais, como nos autômatos finitos

Se você comparar com a definição dada para os autômatos finitos no 1º fascículo, verá que temos apenas dois novos componentes. São eles: Γ e $, ambos relacionados à pilha do autômato que estamos vendo. Como os autômatos finitos não tinham pilha, não precisam desses dois componentes.

Outro ponto de diferença é a função de transição δ. Ela também está presente nos autômatos finitos, mas é definida de maneira diferente nos APs. Aqui, ela é definida como $\boldsymbol{\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^{Q \times \Gamma^*}}$. Isso quer dizer que ela recebe como argumento uma tripla assim δ(q,a,X), onde:

- **q** é o estado de onde sai a transição

- **a** é o próximo símbolo da palavra de entrada ou ε (para não ler nada)

- **X** é o próximo símbolo a ser lido da pilha ou ε (para não ler nada)

Para cada tripla dada como argumento, a função de transição delta dá como resultado um conjunto de pares { (p_1, α_1), (p_2, α_2), ..., (p_j, α_j) }. Cada par indica o próximo estado p_x e a cadeia que será escrita na pilha α_x. Usamos conjunto porque os APs são não-determinísticos e podem ter mais de uma opção de transição para cada entrada. Isso quer dizer que o autômato pode escolher uma das opções:

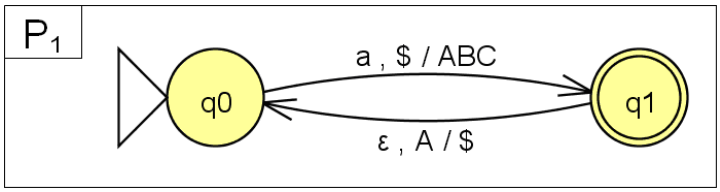
- Mover para p_1 e escrever α_1 na pilha,

- OU mover para p_2 e escrever α_2 na pilha, OU ...,

- OU mover para p_j e escrever α_j na pilha.

A representação da função de transição dos APs com tabelas é mais confusa do que a dos autômatos finitos. Por esse motivo, vamos preferir listar as opções de transição na forma $\delta(q,a,X) = \{ (p_1, \alpha_1), ..., (p_j, \alpha_j) \}$, desde que o conjunto de saída não seja vazio. Se o resultado for um conjunto vazio, simplesmente não escrevemos nada.

Vamos agora fazer a representação formal do AP visto na seção 1.1 e reproduzido abaixo. Vamos chamar o autômato de P_1:



É fácil perceber que o conjunto de estados é $\{q_0, q_1\}$, sendo q_0 o estado inicial e q_1 o único estado de aceitação. É fácil também identificar os símbolos de entrada, pois, na descrição dada naquela seção, dissemos que seria o conjunto {a,b}. Já os símbolos da pilha nós podemos descobrir olhando as transições do autômato. São eles: {$, A, B, C}, onde $ o símbolo que usamos para iniciar a pilha no exemplo dado naquela seção. Assim, já podemos preencher seis componentes da definição formal:

$$P_1 = (\{q_0, q_1\}, \{a, b\}, \{\$, A, B, C\}, \delta_1, q_0, \$, \{q_1\})$$

A função de transição δ_1 nós descrevemos com base nas setas da representação gráfica. Veja que temos uma única seta saindo de q_0. Ela lê o símbolo a, da entrada, e o símbolo $, da pilha. Essa transição faz o autômato ir para q_1 e escrever ABC na pilha. Portanto, temos:

$$\delta_1(q_0, a, \$) = \{ (q_1, ABC) \}$$

A outra seta sai de q_1, não lê nada da entrada, mas lê A da pilha. Ela leva para q_0 e escreve $ na pilha. Portanto, temos:

$$\delta_1(q_1, \varepsilon, A) = \{ (q_o, \$) \}$$

Pronto, essas duas equações definem a função δ_1. Podemos entender que todas as outras entradas possíveis dão como resultado o conjunto vazio. Por exemplo: $\delta_1(q_0, b, A)$ ou $\delta_1(q_1, a, B)$ dão o resultado $\varnothing$.

Observe que o único AP dado como exemplo até agora é ainda muito simples. Ele simplesmente reconhece todas as palavras que são formadas apenas por símbolos a. Esse exemplo ainda não demonstra o poder dos autômatos de pilha porque essa linguagem nós poderíamos representar com um autômato finito sem dificuldades. A seguir, veremos a construção de um AP para uma linguagem que não pode ser representada com autômatos finitos.

Aprenda praticando

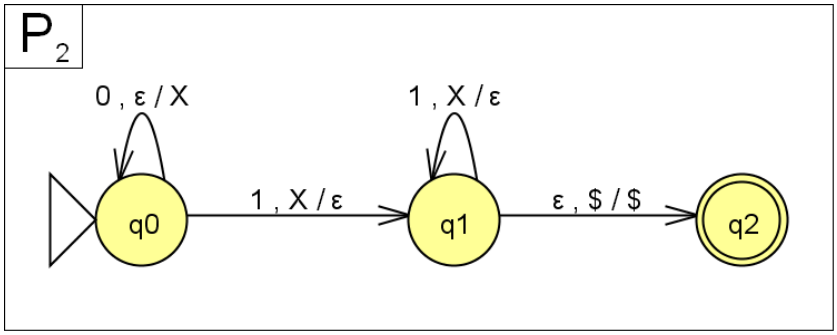
Questão 1: Vimos no capítulo sobre gramáticas livres de contexto (fascículo 2) a linguagem das palavras na forma 0^n1^n, para todo n>1. Lá, dissemos que essa linguagem não é regular, mas é livre de contexto. Isso quer dizer que há alguma gramática livre de contexto e também algum AP para representá-la. A gramática já foi apresentada no fascículo 2. Crie agora um AP de nome P_2 para representá-la.

Resolução:

Veja, pela descrição dada, que as palavras que o autômato deve aceitar começam com algum número *n* de símbolos 0, depois deve ter o mesmo número *n* de símbolos 1. O autômato vai precisar "contar" de alguma maneira os símbolos 0, para depois "conferir" se existe a mesma quantidade de 1s.

A idéia é usar a pilha para isso. A cada símbolo 0 lido da palavra de entrada, colocamos algum símbolo na pilha (escolhemos usar X para isso). Depois, a cada símbolo 1 lido da entrada, o autômato P_2 deve também ler um símbolo X na pilha. O autômato irá para o estado de aceitação se retirarmos todos os Xs da pilha nesse processo.

O diagrama do autômato P_2 é dado abaixo. Considere que $ é o símbolo de início da pilha.



Veja que a transição de q_0 para q_0 serve para ler todos os 0s da entrada, colocando os Xs correspondentes na pilha. A transição para q_1 só vai acontecer quando houver, ao mesmo tempo, um símbolo 1 na entrada e um símbolo X na pilha. A partir daí, no estado q_1, para cada símbolo 1 lido, deve haver um símbolo X na pilha. Se houver a quantidade certa de símbolos 1s, a entrada vai ficar vazia no mesmo momento em que a pilha ficar sem símbolos Xs. A pilha, então, volta ao símbolo inicial $ (que ficaria "abaixo" dos Xs) e o autômato pode mover para o estado de aceitação.

Questão 2: Represente formalmente o autômato da questão anterior.

Resolução:

Os estados são facilmente identificáveis na figura, dando o conjunto: $\{q_0,q_1,q_2\}$. Desses, temos q_0 como estado inicial e o conjunto unitário de estados de aceitação $\{q_2\}$. Pela descrição da linguagem que foi dada, podemos concluir que os símbolos de entrada são: {0,1}. Por sua vez, os símbolos da pilha você pode descobrir revendo o processo de construção do autômato (ou apenas olhando nas transições do autômato final). Confira que são apenas dois símbolos da pilha, formando o conjunto: {$,X}. Destes, temos $ como símbolo de início da pilha.

Falta apenas a função de transição do autômato, que chamaremos de δ_2. Ela é construída com base nas transições do autômato, representadas por setas. Por exemplo, temos uma transição de q_0 para q_0 que lê 0 da entrada e nada (ε) da pilha e, depois escreve X. Essa transição nos leva a concluir que

$\delta_2(q_0,0,\varepsilon) = \{ (q_0, X) \}$. Fazendo o mesmo para as outras três transições, completamos a definição:

$P_2 = (\{q_0,q_1,q_2\}, \{0,1\}, \{\$,X\}, \delta_2, q_0, \$, \{q_2\})$, onde δ_2 é dada por:

$$\delta_2(q_0,0,\varepsilon) = \{ (q_0,X) \}$$

$$\delta_2(q_0,1,X) = \{ (q_1,\varepsilon) \}$$

$$\delta_2(q_1,1,X) = \{ (q_1,\varepsilon) \}$$

$$\delta_2(q_1,\varepsilon,\$) = \{ (q_2,\$) \}$$

Questão 3: Mostre a computação do AP da questão anterior quando ele recebe a cadeia 0011. Diga se esta cadeia é aceita e justifique.

Resolução:

Começamos no estado inicial q_0, com a entrada 0011 e com a pilha com seu símbolo de início $:

$(q_0, 0011, \$)$

Podemos agora fazer a transição de q_0 para q_0 que lê 0 e escreve X na pilha, o que nos dá a ID:

$(q_0, 011, X\$)$

Podemos fazer novamente a mesma transição:

$(q_0, 11, XX\$)$

Veja que já lemos os dois 0s e, como resultado, escrevemos dois Xs na pilha. O que o autômato vai fazer a partir de agora é como se ele “conferisse” que há um símbolo 1 para cada X. Veja que, neste ponto, só podemos fazer a transição de q_0 para q_1 que lê justamente um 1 e um X juntos, dando o resultado:

$(q_1, 1, X\$)$

Agora podemos fazer uma transição de q_1 para q_1 que faz algo parecido:

$(q_1, \varepsilon, \$)$

Observe que palavra já está vazia, mas ainda não estamos em um estado de aceitação. Por sorte, ainda podemos fazer uma transição que não lê símbolo da entrada, e lê \$ da pilha. Ela leva de q_1 para q_2, resultando na ID:

$(q_2, \varepsilon, \$)$

Veja que chegamos a uma configuração com um estado de aceitação e com a entrada vazia. Portanto, podemos concluir que a palavra 0011 é aceita.

Nós detalhamos com bastante cuidado todo o processo de reconhecimento para facilitar o seu entendimento, mas nas suas respostas em exercícios ou provas, você pode ser mais objetivo e listar diretamente toda a computação, como mostrado abaixo:

$$(q_0, 0011, \$) \vdash (q_0, 011, X\$) \vdash (q_0, 11, XX\$)$$

$$\vdash (q_1, 1, X\$) \vdash (q_1, \varepsilon, \$) \vdash (q_2, \varepsilon, \$)$$

Em todo caso, é possível tirarmos a mesma conclusão: a palavra 0011 dada como entrada é aceita.

Saiba mais

A apresentação inicial dos autômatos com pilha na seção 1.1 foi baseado no livro de Sipser, que recomendamos pela sua boa didática. Porém, os principais detalhes do presente capítulo, tais como definição formal, computação e aceitação em APs, estão mais parecidos com o livro de Hopcroft, Ullman e Motwani. Por isso, este livro é a principal leitura complementar recomendada para o assunto.

Um assunto presente no livro citado que destacamos é prova da equivalência entre gramáticas livres de contexto e autômatos de pilha. Nós apenas citamos aqui que esses dois modelos são equivalentes, mas o livro vai além e descreve detalhadamente como converter de um modelo para o outro.

Atividades e Orientações para estudo

Abaixo apresentamos algumas atividades que servirão para você, cursista, fixar o assunto aqui apresentado. Gostaríamos de lembrar que atividades extras podem ser colocadas no ambiente para valer parte da nota. Em todo caso, se você fizer os exercícios aqui apresentados corretamente estará preparado para fazer também os exercícios do ambiente.

Questão 1: Considerando o autômato P_2 dado antes, mostre a computação de cada uma das cadeias abaixo e diga se cada cadeia é aceita:

a) 01

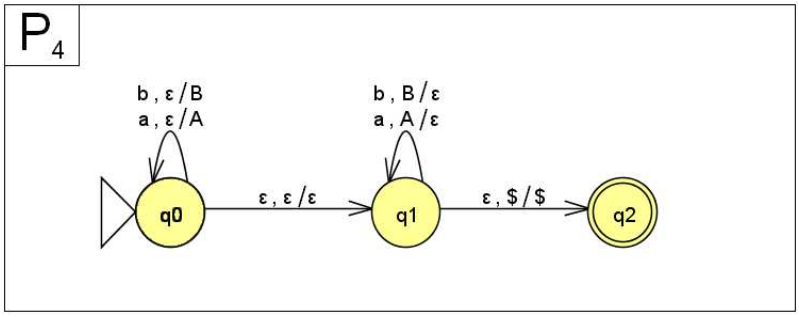
b) 001

c) 000111

d) 010

Questão 2: Crie um autômato P_3 para representar a linguagem das palavras na forma 0^n1^{2n}, para n>1. Ou seja, são cadeias que iniciam com uma número qualquer (n) de 0s e que, depois, têm um número duas vezes maior (2n) de 1s. (DICA: use uma idéia muito parecida com a que usamos para criar P_2, porém o novo autômato, para cada 0, deverá se preparar para ler *dois* 1s).

Questão 3: Vimos que os APs são não-determinísticos, mas só demos exemplos de APs determinísticos. Abaixo, mostramos um exemplo de APs não-determinístico para reconhecer palavras na forma $w.w^r$, ou seja, palavras cuja segunda metade é o inverso da primeira.



Por ser (verdadeiramente) não-determinístico, o autômato P_4 pode permitir mais de uma computação para uma palavra dada. Para cada palavra abaixo, encontre alguma computação que comprove que ela é aceita pelo autômato. (Já estamos afirmando que todas são aceitas. Falta achar uma computação que comprove).

a) aa

b) abba

c) bbbb

Questão 4: Crie as representações formais dos autômatos P_3 e P_4.

Questão 5: Um autômato que aceita por pilha vazia precisa ter estado de aceitação? Se tiver, faz diferença na linguagem que ele aceita?

Capítulo 2 – Máquinas de Turing (MTs)

Talvez você, cursista, tenha chegado neste ponto do curso ainda sem saber exatamente o que os modelos matemáticos vistos têm a ver com os computadores modernos. Se este for o seu caso, não se preocupe, pois o que veremos neste capítulo o ajudará a entender melhor a relação entre os modelos matemáticas e os computadores.

Primeiramente, veremos mais um modelo matemático para representação de linguagens – a Máquina de Turing (MT). Apesar de sua aparente simplicidade, este modelo é mais poderoso do que os modelos que vimos antes. Aliás, nenhum outro modelo matemático é capaz de representar mais linguagens do que ele.

Devido a essa grande capacidade das máquinas de Turing, elas é que são usadas como base para o estudo teórico dos limites dos computadores reais criados pelo homem. Acredita-se, por exemplo, que nunca haverá um computador capaz de resolver mais problemas do que as máquinas de Turing.

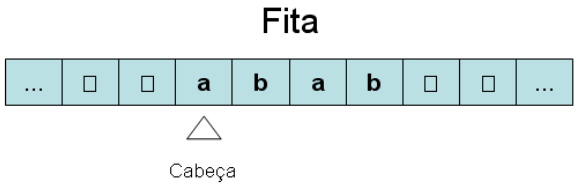
Antes de nos aprofundarmos nessa discussão, vamos aprender como funcionam as máquinas de Turing (que abreviaremos como MTs).

2.1 Princípio de Funcionamento das MTs

As máquinas de Turing são parecidas com os autômatos vistos anteriormente neste curso. Porém, elas possuem uma memória linear infinita que chamaremos de **fita**, por lembrar as fitas magnéticas usadas para

armazenar dados em computadores antigos[1]. Na máquina de Turing, cada posição da fita guarda algum símbolo. Esses símbolos são lidos em seqüência, um por vez, como faz a “cabeça” ou “cabeçote” com as fitas magnéticas reais.

A palavra de entrada da máquina de Turing é automaticamente gravada na fita quando a máquina é iniciada. Como a fita é infinita, as (infinitas) posições mais à esquerda e mais à direita da palavra são preenchidas com um símbolo especial que chamaremos de **símbolo branco**. Por exemplo, se considerarmos que a palavra de entrada é abab e que o símbolo branco é representado por □, a fita da MT seria iniciada assim:



Veja que a MT começa com a “cabeça” posicionada sobre o primeiro símbolo da palavra de entrada e, a partir daí, começa a operar. A cada etapa de sua operação, a MT faz três ações na fita: ela lê o símbolo que está sob a cabeça, escreve outro símbolo sobre ele (substituindo-o) e move a cabeça uma única posição, para a esquerda ou para a direita. Essas três ações ocorrem em conjunto, levando a máquina a caminhar na fita uma posição por vez em qualquer direção.

Para representar uma MT, podemos usar um diagrama de estados parecido com o dos autômatos finitos e dos autômatos com Pilha. Novamente, temos um estado inicial e zero ou mais estados de aceitação, representados da mesma maneira. Porém, as setas das transições recebem rótulos diferentes dos usados nos outros autômatos. Cada transição tem um rótulo:

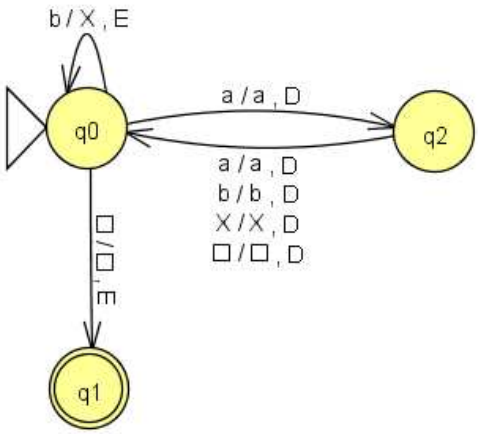
X / Y, M , onde:

- **X** é o símbolo que precisa ser lido da fita (pela cabeça)

[1] Também é usada em computadores modernos, mas de maneira muito restrita. Por exemplo: para fazer backups de servidores.

- **Y** é símbolo que será escrito na fita (substituindo X)
- **M** indica a direção para onde a cabeça se moverá na fita, podendo assumir apenas dois valores:
    - E, para “esquerda” (também pode ser usado L, do inglês *left*)
    - D, para “direita” (também pode ser usado R, do inglês *right*)

Para ilustrar, mostramos abaixo um primeiro exemplo de MT, que aceita palavras sobre o alfabeto {a,b}:



Para explicar este primeiro exemplo de MT, vamos detalhar como ela manipula a fita. A idéia usada na construção da máquina é que, toda vez que ele encontrar um b na fita, ela vai escrever X no lugar e vai voltar uma posição. Isso acontece na transição “b/X,E” em q_0. Já se a MT dada ler um a, ela vai andar duas posições para a direita. A primeira posição ela anda ao encontrar o próprio a, como você pode conferir na transição “a/a,D” de q_0 para q_1. A segunda posição ela anda qualquer que seja o símbolo, como você pode ver nas várias opções de transição de q_2 para q_0.

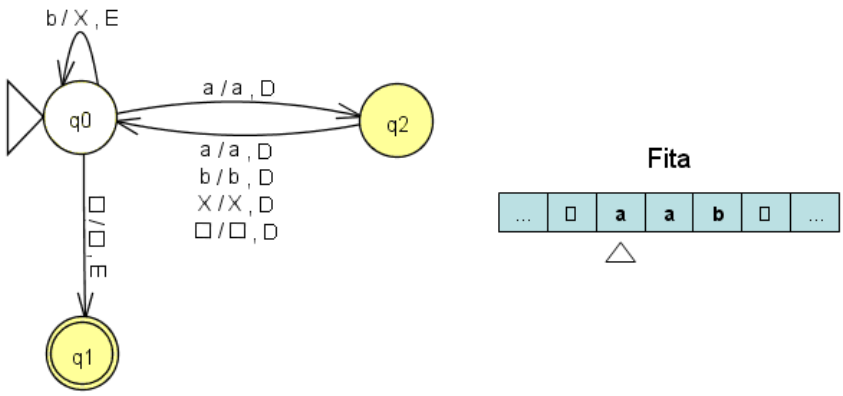
Se em algum momento, no estado q_0, a máquina encontrar o símbolo vazio, ela vai para o estado de aceitação q_1 e ali irá parar. Entenda que **uma máquina de Turing pára** em um estado quando nenhuma transição pode ser usada a partir desse estado.

Destacamos, ainda, que nas máquinas de Turing a palavra não precisa ser lida inteira para ser aceita, como acontecia nos outros tipos de autômatos. A única coisa que importa é que a MT pare em um estado de aceitação. Por isso, toda vez que a MT mostrada acima chegar ao estado q_1, ela não só irá *parar* como também irá *aceitar* a palavra (mesmo que ela não tenha sido lido inteira).

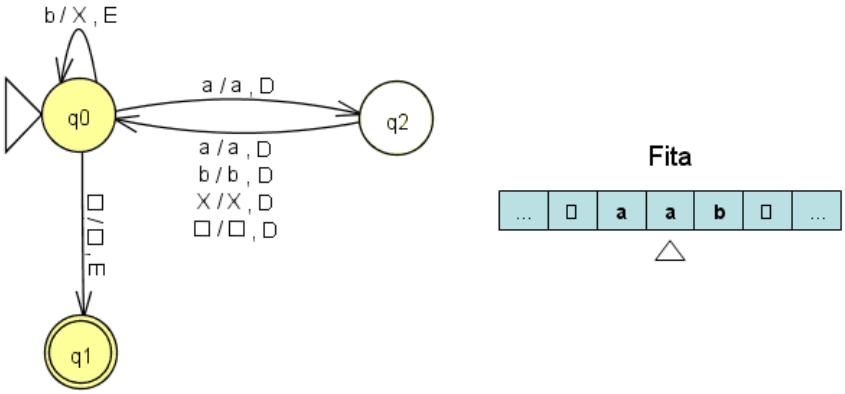
Como fizemos com os APs, vamos dedicar uma seção inteira aos detalhes do processo de reconhecimento em uma MT.

2.2 Computação e Aceitação nas Máquinas de Turing

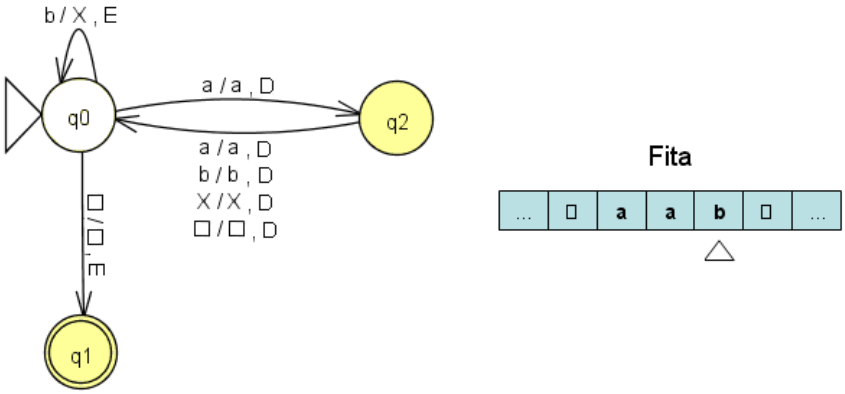
Vamos começar analisando o comportamento da MT dada quando ela recebe a palavra aab como entrada. A configuração inicial da MT é exibida graficamente abaixo:



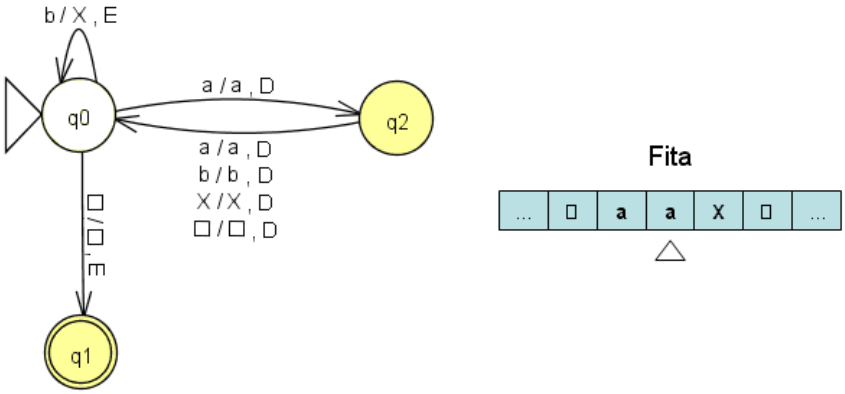
Veja que a máquina está no estado q_0 e a cabeça da fita está no primeiro símbolo da palavra. Lembramos que tanto à direita quanto à esquerda da palavra há infinitos símbolos brancos □. Veja que há uma transição saindo de q_0 para q_2 lendo o símbolo a. O que ela faz na fita é reescrever o a (ou seja, não muda o conteúdo) e mover a cabeça para a direita. A configuração resultante é:



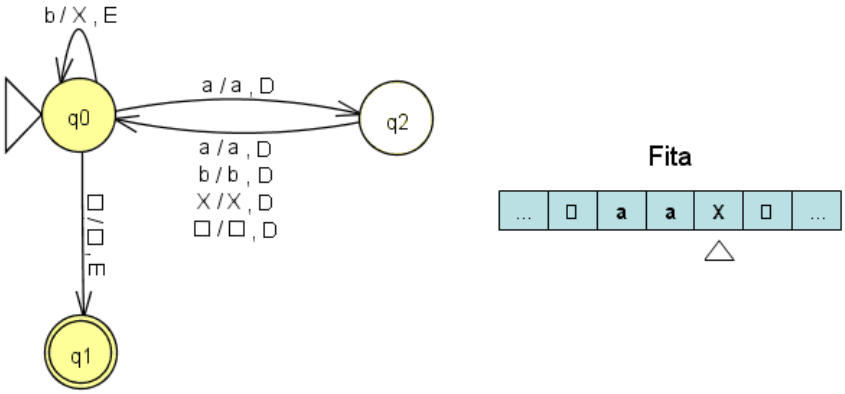
Agora, temos a cabeça de leitura apontando para outra ocorrência do símbolo a. Este é um ponto interessante para lembrar a você, cursista, que o movimento da cabeça da fita e a mudança entre os estados são coisas distintas. A mudança de estados é definida pelas setas, enquanto a direção da fita é definida pelo terceiro elemento do rótulo da seta (E ou D). Veja que, neste ponto, a MT vai fazer uma transição de volta ao estado q_0, mas movendo a cabeça para a direita. O resultado é mostrado abaixo:



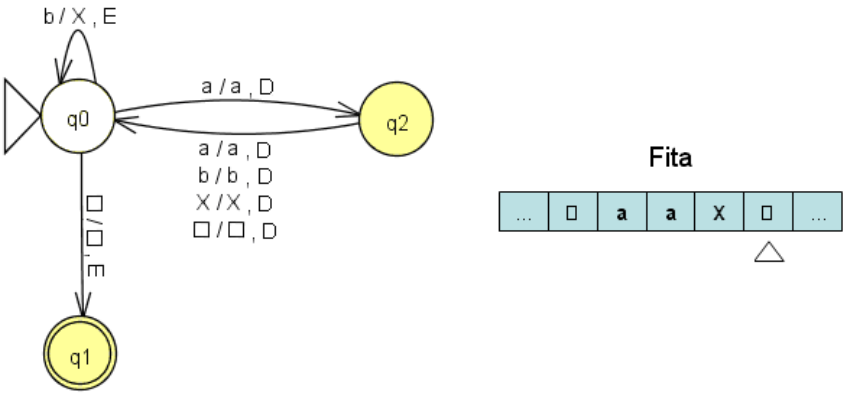
Com a MT no estado q_0 e com o símbolo b sendo lido pela cabeça da fita, a máquina vai fazer a transição de q_0 para q_0. Apesar de não mudar realmente de estado, a fita da MT vai mudar: o símbolo b vai ser trocado pelo símbolo X e a cabeça anda para esquerda. O resultado é a configuração:



Estamos novamente no estado q_0 com a cabeça apontando para um símbolo a. Você agora já deve ser capaz de perceber que a MT vai mudar para q_2 e a fita vai andar para a direita, resultando na configuração:

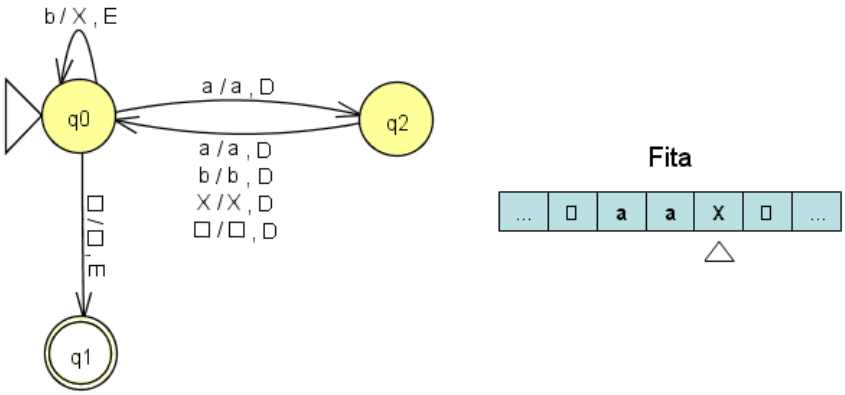


Em q_2 e com o símbolo X sendo lido pela cabeça, a máquina vai mudar para a seguinte configuração:



A máquina aqui está no estado q_0 com a cabeça da fita posicionada em um símbolo branco. Pelo diagrama de estados, vemos que a próxima configuração

será no estado q_0 com a cabeça da fita posicionada um símbolo à esquerda, assim:



Agora a máquina chegou a um estado do qual não sai nenhuma transição. Concluímos, portanto, que a MT parou no estado q_1. Como q_1 é um estado de aceitação, concluímos que a palavra aab é aceita pela máquina dada. Em outras palavras, a palavra faz parte da linguagem que a MT representa.

Se você entendeu o processo de reconhecimento descrito acima, não terá dificuldades para entender o que vamos definir a seguir. Se ainda estiver inseguro, aconselhamos você a reler cuidadosamente a partir do início da seção.

Como fizemos com os APs, vamos mostrar uma representação mais sucinta de cada configuração do autômato. Pense um pouco: quantos e quais elementos são importantes na configuração de uma MT?

Um elemento é fácil de ver: é o estado do autômato. Não precisamos mostrar todo o diagrama como fizemos acima, basta mostrarmos o estado do diagrama onde a MT está.

Outro elemento poderia ser simplesmente a fita, que poderia ser representada como uma cadeia (palavra). Contudo, tem um problema com essa idéia: como representar a “posição atual” lida pela cabeça? O que vamos fazer para resolver esse problema é dividir a fita em duas partes: uma parte contendo o que está à esquerda da posição atual e a outra parte contendo tanto a posição atual como o que está à direita dela.

Os três elementos citados vão formar uma **descrição instantânea** ou **ID** (instantaneous description) de uma máquina de Turing. Para simplificar, uma ID será representada na seguinte forma:

<fita à esquerda> <estado> <posição atual + fita à direita>

Um detalhe é que a fita de uma MT possui infinitos símbolos brancos tanto à esquerda quando à direita, que poderiam dificultar a representação. Usaremos duas regras para lidar com isso:

- Primeiramente, não representaremos seqüência infinitas de símbolos brancos que ocorrerem na extrema esquerda de <fita à esquerda> ou na extrema direita de <posição atual + fita à direita>.

- Em segundo lugar, se uma das duas partes da fita ficar vazia, colocaremos um símbolo □ (do trecho infinito que foi omitido).

Vejamos agora como representar com IDs o processamento da palavra aab que representamos antes de maneira mais gráfica. A configuração inicial da MT inclui o estado q_0. O lado esquerdo da fita é composto apenas de símbolos brancos, que representaremos com um só □ (para não ficar vazia). Já a outra parte da fita, é formada justamente pela cadeia aab. Assim a ID inicial será:

□ q_0 aab

Como fizemos com os APs, representaremos mudanças de IDs válidas nas máquinas de Turing com o símbolo |-. Assim, o processamento da palavra aab pode ser representado pela seqüência de IDs mostrada abaixo. Compare com as configurações apresentadas anteriormente com figuras.

□ q_0 aab

|- a q_2 ab

|- aa q_0 b

|- a q_0 aX

|- aa q_2 X

|- aaX q_0 □

|- aa q_1 X

Atente para o detalhe de que o primeiro símbolo depois do estado representa a posição que será lida pela cabeça para decidir a próxima configuração. Veja que na primeira ID foi lido um “a”, o que levou ao estado q_2 e causou um movimento para a direita, como pode ser visto na segunda ID.

Observe, ainda, que na primeira e na sexta IDs tivemos que mostrar um símbolo branco da seqüência infinita de alguma das extremidades lado da fita. Nas outras IDs, a MT estava em posições intermediárias da palavra e, por isso, omitimos todos os símbolos brancos. Se não usássemos essas regras para omitir os símbolos brancos, as IDs seriam representadas como no exemplo abaixo (equivalente à primeira ID da seqüência anterior):

(infinitos)...□□□□□□□ q_0 aab□□□□□□□... (infinitos)

Uma seqüência de IDs como aquela que apresentamos será chamada de **computação**, pois representa como a máquina de Turing “calcula” (ou “computa”) se a palavra faz parte da linguagem ou não. Agora, usaremos esses conceitos para definir de maneira mais precisa quando uma palavra é aceita.

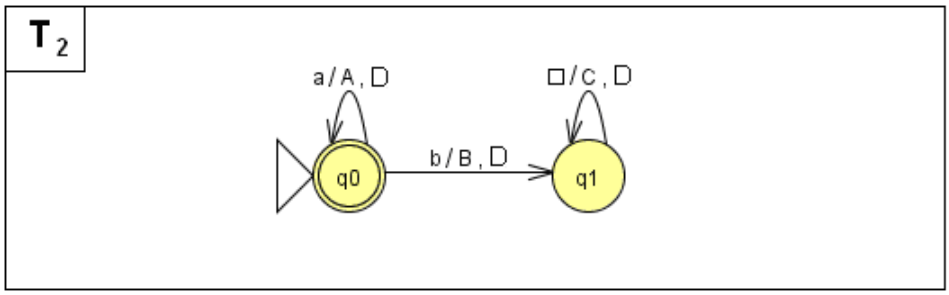
Consideramos que uma palavra *w* dada como entrada para uma MT será **aceita** se a computação dela tiver as seguintes características:

- Inicia em uma ID na forma □ q_0 w, onde q_0 é o estado inicial
- Termina em uma ID qualquer α q_x β , de maneira que a MT não possa mais sair dessa configuração.
- O estado q_x desta última ID é um estado de aceitação.

De maneira parecida, a palavra será considerada **rejeitada** se a computação dela satisfizer as duas primeiras condições (de sair da configuração inicial e parar), mas não terminar em um estado de aceitação.

Você agora pode estar pensando que aceitar e rejeitar uma palavra são as duas únicas opções de uma MT, mas, infelizmente, essa não é a realidade. Existem situações em que a MT não consegue satisfazer a segunda condição dada antes: de “parar” em alguma configuração.

Para exemplificar, vamos usar a seguinte máquina de Turing T_2, que recebe entradas sobre o alfabeto {a,b}:



Se você tentar entender essa MT verá que ela aceita apenas as palavras formadas apenas por símbolos a, mas o que acontece se a palavra tiver um b? Vamos analisar o comportamento de T_2 quando ela recebe a entrada “ab” para vermos. A computação resultante é descrita abaixo:

□ q_0 ab

|- A q_0 b

|- AB q_1 □

|- ABC q_1 □

|- ABCC q_1 □

|- ABCCC q_1 □

|- ...

Perceba que, depois de entrar em q_1, a MT começa a trocar cada □ por C, seguindo para a direita. Porém, à direita ela sempre encontra outro □ (pois a fita tem infinitos deles) e o processo se repete indefinidamente.

Ou seja, essa MT nunca vai parar! Dizemos que a MT **entra em loop** para a palavra “ab”. Neste caso, a palavra definitivamente não é considerada aceita, mas também não chega ser devidamente rejeitada. De maneira informal, podemos dizer que a cadeia “deveria ser rejeitada, mas a MT não consegue parar para rejeitar”.

Esse comportamento não é desejável em uma MT. A realidade é que, em muitos casos, é possível criar uma MT que nunca entra em loop, qualquer que seja a entrada. Porém, em outros casos isso não é possível. Não entraremos em detalhes porque esse assunto será visto no próximo capítulo.

O importante é que já vimos todos os detalhes de funcionamento das MTs. Então agora podemos apresentar a definição matemática precisa desse modelo.

2.3 Definição Formal

As máquinas de Turing podem ser definidas matematicamente como 7-tuplas contendo os componentes descritos abaixo:

T = (Q, Σ, Γ, δ, s, □, F) , onde:

- **Q** é conjunto finito de **estados**
- **Σ** (sigma) é o conjunto dos **símbolos de entrada**
- Γ (gama) é o conjunto dos **símbolos de fita**, que são os símbolos que podem ser lidos e escritos na pilha, incluindo todo o conjunto **Σ**
- **δ** (delta) é a **função de transição**, que deixaremos para descrever melhor abaixo

- **s** é o elemento de Q que serve de **estado inicial**
- □ é o **símbolo branco**, que é um elemento de Γ mas não de **Σ**, ou seja, ele aparece na fita mas não pode aparecer em palavras de entrada
- **F** é o conjunto de **estados de aceitação** ou estados finais

Como você pode perceber, a definição da máquina de Turing é parecida com as definições dos autômatos finitos e do autômato de pilha, vistos antes. As definições todas variam em dois ou três dos componentes, mas o componente que é o principal responsável pelas diferentes formas de operação de cada um é a função de transição δ. Por esse motivo, falaremos dela a partir de agora.

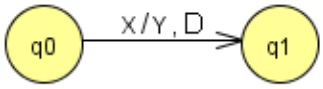
Aqui, ela é definida como **δ : Q x Γ → Q x {E,D}** . Isso quer dizer que ela recebe como argumento um par assim δ(q,X), onde:

- **q** é o estado de onde sai a transição
- **X** é o símbolo na próxima posição da fita

O valor da função, se estiver definido, é uma tripla (p,Y,M) onde:

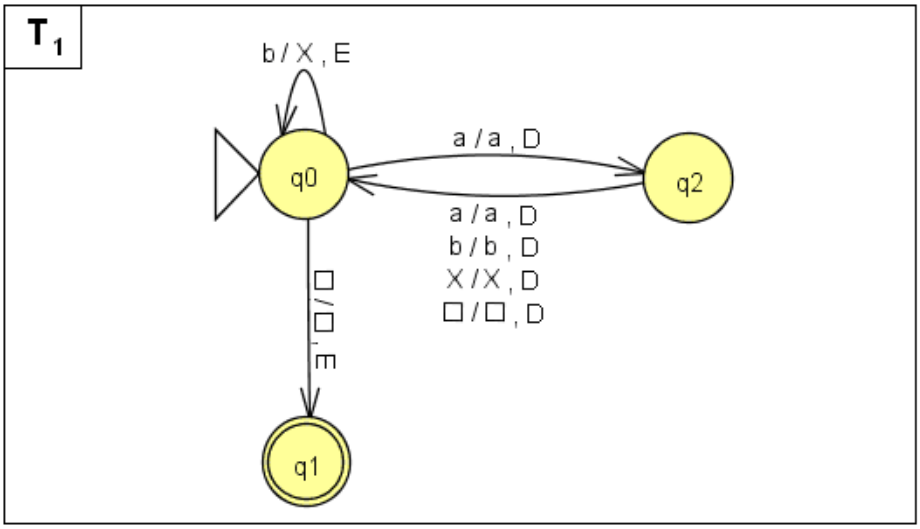
- **p** é o próximo estado
- **Y** é o símbolo que será escrito sobre o X (substituindo-o)
- **M** é a direção de movimento da cabeça, podendo assumir apenas os valores E (esquerda) ou D (direita)

Como nos outros autômatos, o que a função de transição representa é o mesmo que as setas entre os estados da representação gráfica representam. Aquilo que formalmente dizemos assim $\delta(q_0,X) = (q_1,Y,D)$, graficamente é representado assim:



O estado de onde sai a seta (q_0) e o símbolo antes da barra (X) são os dois argumentos da função, enquanto que o estado onde chega a seta (q_1), o símbolo depois da barra (Y) e a direção da fita (D, de direita), formam o resultado da função.

Vamos agora dar a representação formal da MT do primeiro exemplo, que reproduzimos abaixo. Vamos chamá-la aqui de T_1.



Os estados de T_1 são facilmente identificáveis: $\{q_0, q_1, q_2\}$, onde q_0 é estado inicial e o conjunto de estados de aceitação é o conjunto unitário $\{q_1\}$. Os símbolos de entrada, quando apresentamos esta MT na primeira seção, dissemos que seriam o conjunto {a,b}. Já os símbolos da fita incluem todos os símbolos usados: {a,b,X,□} [2], onde □ é o símbolo branco. Com isso, já podemos dar quase todos os detalhes da representação formal:

$$T_1 = (\{q_0, q_1, q_2\}, \{a,b\}, \{a,b,X,□\}, \delta_1, q_0, □, \{q_1\})$$

O que falta representar é a função de transição δ_1. Veja que as funções de transição das MTs recebem apenas duas entradas. Isso nos permite representá-las com tabelas. Como nos autômatos finitos, usaremos as linhas para os estados e as colunas para os símbolos de fita. No encontro de linha e coluna colocaremos a tripla que representa o valor da função.

---

[2] Atenção, não confunda! E e D não foram listados porque não são símbolos da fita, mas sim direções do movimento da cabeça da fita.

Dessa forma, podemos concluímos a representação de T_1 mostrando sua função δ_1 na forma de tabela:

	a	**b**	**X**	□
q_0	(q_2, a, D)	(q_0, X, E)	-	(q_1, □, E)
q_1	-	-	-	-
q_2	(q_0, a, D)	(q_0, b, D)	(q_0, X, D)	(q_0, □, D)

Podemos perceber na tabela acima que, dado um estado e um símbolo, a máquina oferece uma das duas opções: ou existe uma opção de movimento única (dada pela tripla) ou não existe opção de movimento (representada com um traço). Por isso, em um dado instante na máquina de Turing, sabemos sempre exatamente o que fazer: se houver a tripla, a MT faz o movimento; se não houver, a MT pára.

Isso significa que, da maneira que definimos, as MTs são **determinísticas**. Nelas não existem múltiplas opções de movimento para um mesmo par (estado, símbolo). Você deve lembrar que os autômatos que apresentavam ambigüidades de movimento, nós chamamos antes de *não-determinísticos*. O exemplo mais recente que vimos foram os autômatos com pilhas.

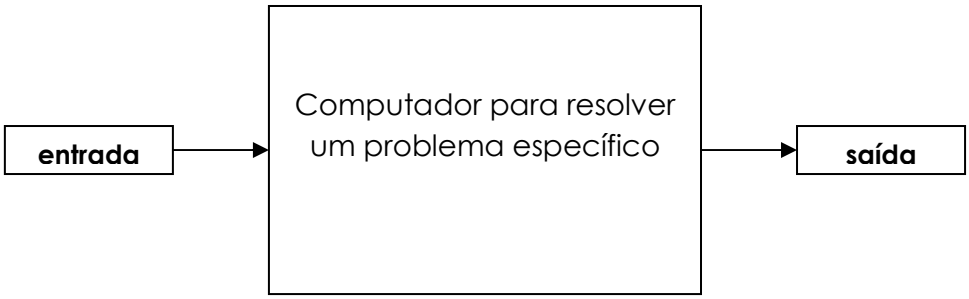
Definimos os APs daquela maneira porque, se os definíssemos como determinísticos, eles seriam *menos* poderosos. Porém, as Máquinas de Turing determinísticas e não-determinísticas têm o mesmo poder! É possível transformar uma MT de um tipo em uma MT do outro tipo. Por isso, preferimos trabalhar com o tipo mais simples que é a MT determinística.

A seguir, vamos dar uma noção introdutória da subárea da Teoria da Computação conhecida pelo nome de Teoria da Computabilidade. Veremos que importância o modelo “Máquina de Turing” tem para tal área.

2.4 Noções de Computabilidade

O que vamos apresentar agora é uma visão simplificada da área. Vamos considerar que a **Teoria da Computabilidade** estuda quais problemas matemáticos podem ser resolvidos construindo-se máquinas para eles.

Imagine tais máquinas como “computadores” específicos, criados para tratar de um único problema. Vamos considerar que um problema matemático pode ter algumas variações de parâmetros e tais parâmetros serão dados como **entrada** para o computador. Por sua vez, o equipamento retorna, em tempo finito, a **saída** que representa a solução do problema. A figura abaixo representa essa idéia:



Um problema será chamado de **computável** se for possível construir uma máquina física real para resolvê-lo (computá-lo) em tempo finito. Caso contrário, o problema é chamado de **não-computável**, dando a idéia de que nenhum computador pode ser criado para resolvê-lo adequadamente. A grande questão em Computabilidade, portanto, é provar quais problemas são computáveis e quais não são.

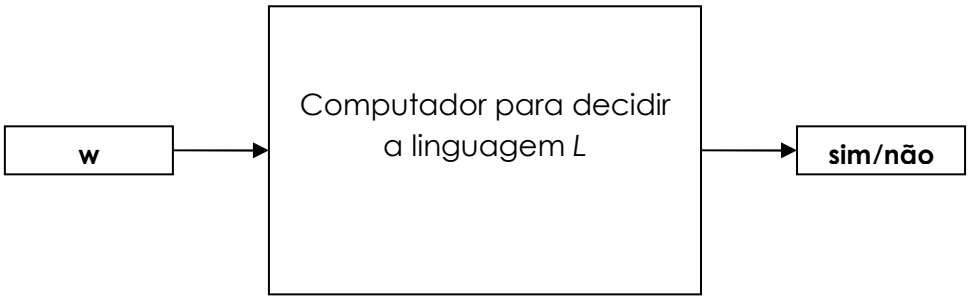
Alguns exemplos de problemas matemáticos que são computáveis:

- Somar dois números (entrada: par de números, saída: número)
- Dividir dois números inteiros com resultado inteiro (entrada: par de números, saída: número)
- Verificar se um número é primo (entrada: um número, saída: sim/não)
- Encontrar menor caminho de um ponto a outro de uma cidade (entrada: representação da cidade e das duas localizações, saída: caminho)

Um exemplo de problema matemático que é não-computável:

- Encontrar raízes inteiras de um polinômio (entrada: polinômio, saída: raízes)

A verdade é que já temos estudado a questão da Computabilidade desde o início da disciplina. Porém, estamos tratando apenas de uma classe de problemas ligados a linguagens: "descobrir se uma cadeia *w* faz parte de uma linguagem *L*". Este tipo de problema é chamado de **problema de decisão**, pois ele decide se a palavra faz parte da linguagem ou não. A entrada dos computadores, no caso, é a cadeia. Já a saída é, simplesmente, um dos valores: "sim" ou "não", como ilustra a figura abaixo:



Assim, a idéia apresentada antes de "problemas computáveis" é trocada pela idéia de **linguagens decidíveis**, que são linguagens que podem ser decididas com algum computador em tempo finito. Assim, o objetivo da Teoria da Computabilidade se transforma no problema de provar quais linguagens são decidíveis e quais não são.

Ao contrário do que possa parecer, essa abordagem não traz grandes limitações, pois muitos problemas matemáticos podem ser tratados naturalmente como problemas de decisão de linguagens. Por exemplo, o problema de verificar se um número é primo (que citamos acima) já tem respostas "sim/não". Então podemos tratá-lo como a linguagem das palavras no alfabeto {0,1,2,...,9} que representam números primos[3] e tentar criar um "computador" para decidir esta linguagem.

---

[3] Lembrando que linguagem é um conjunto de cadeias/palavras.

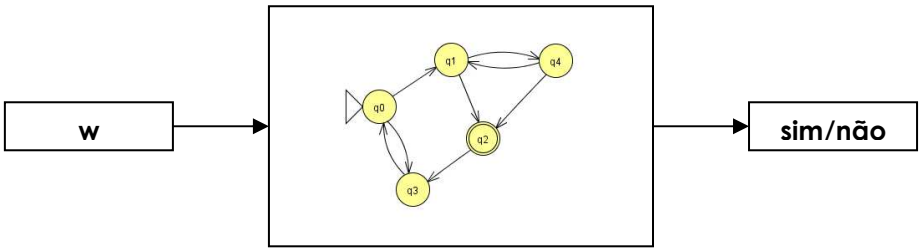
O que você talvez ainda esteja questionando é que não detalhamos a construção de nenhum “computador” real na disciplina. Realmente não vimos detalhes físicos da construção, porém, o que importa no computador é a sua lógica interna. E o que vimos nesta disciplina foram maneiras de modelar a lógica interna de um “computador”. O que usamos para isso foram os vários modelos matemáticos vistos, mas, em especial, os autômatos: AFD, AFND, AFND-ε, AP e, agora, MTs.

Vamos revisar um pouco sobre os tipos de autômatos, comentando sobre a capacidade de cada um, ou seja, sobre quais linguagens eles são capazes de decidir.

O que é comum a todos os autômatos é que eles têm um diagrama de estados, com transições entre eles. Grosseiramente falando, esse diagrama representa o que a máquina “pensa” da palavra a cada instante. Ou seja, cada estado tem algum “significado”, que você define quando projeto o autômato. Por exemplo, você pode construir um autômato de maneira que um estado dê a idéia (implícita) de “leu um número ímpar de a’s”, enquanto outro estado daria a idéia de “leu um número par de a’s”.

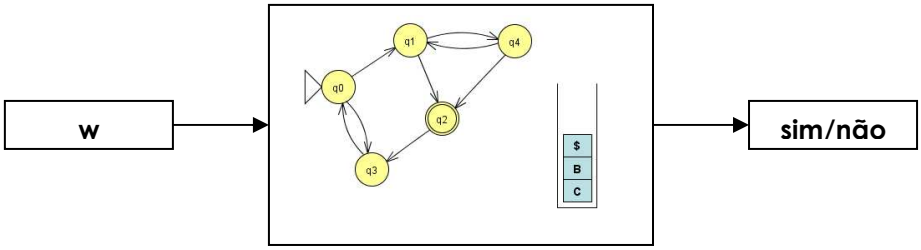
Além disso, os autômatos podem ter alguma memória auxiliar. Essa memória vai fazer diferença na classe de linguagens que os autômatos são capazes de computar.

Os autômatos finitos (AFD, AFND e AFND-ε), por exemplo, vimos que não têm nenhuma memória auxiliar. Por isso, eles são os modelos menos poderosos que estudamos. Chamamos as linguagens que eles são capazes de decidir de **linguagens regulares**.



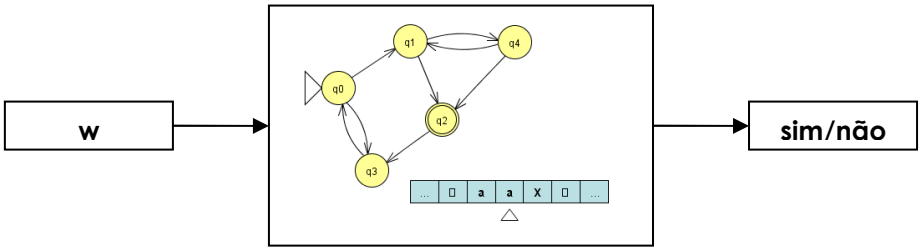
Computadores baseados em autômatos finitos

Já os Autômatos com Pilha (APs) têm uma memória infinita que funciona segundo a regra “o último que entra é o primeiro que sai”. Por isso, estes autômatos são capazes de decidir um conjunto maior de linguagens. Chamamos a essas linguagens de **linguagens livres de contexto**.



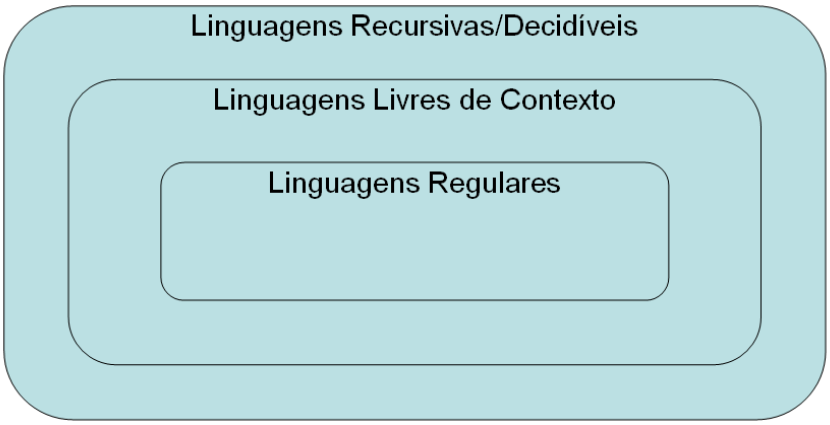
Computadores baseados em autômatos com pilha

Por fim, vimos as MTs como autômatos com uma memória unidimensional infinita que pode ser lida e escrita livremente (mesmo caminhando uma posição por vez). Como conseqüência, este é o modelo que é capaz de decidir mais linguagens do que todos os outros modelos vistos!



Para definir as linguagens que ele computa, só podemos considerar as MTs que sempre param (nunca entram em loop). Afinal, a definição de "linguagem decidível" que demos exige que a máquina execute em tempo finito e isso não acontece nas MTs que entram em loop. Pois bem, as linguagens que podem ser computadas por MTs que sempre param são chamadas de **linguagens recursivas** ou **linguagens decidíveis**.

Esse tipo de classificação das linguagens de acordo com os autômatos que as decidem forma a **Hierarquia de Chomsky**[4], representada abaixo. Já mostramos essa hierarquia no 2º fascículo, porém aqui ela aparece com um nível a mais, que é o das linguagens decidíveis.



O modelo MT, na verdade, não apenas é mais poderoso do que os outros modelos de computação que nós vimos como também é o mais poderoso que existe! Nenhum outro modelo proposto é capaz de decidir mais linguagens do que as máquinas de Turing. Por isso dissemos que ele reconhece as "linguagens decidíveis" – porque nenhuma linguagem é decidível por outro modelo se não for também decidível por uma MT.

---

[4] Na verdade, a hierarquia de Chomsky original não continha as linguagens recursivas. Mesmo assim, manteremos o nome porque a idéia que seguimos é a mesma.

Se sairmos do estudo das linguagens e pensarmos em problemas computáveis também temos resultado parecido: os problemas computáveis são todos resolvidos por alguma MT. Por esse motivo, as máquinas de Turing são vistas como o modelo de computação mais geral que existe, em termos de capacidade de resolver problemas. Existem outros modelos equivalentes, mas nenhum é mais poderoso.

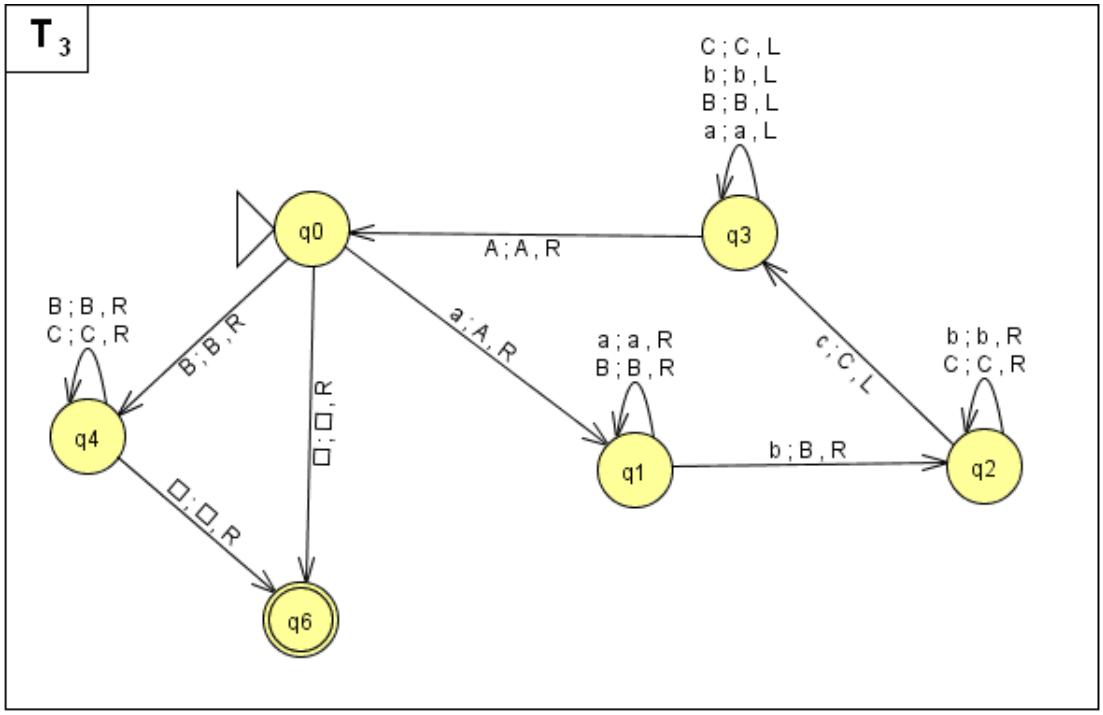
Isso quer dizer que não importa como um computador seja construído de fato. Não importa quanta memória seja colocada nem a maneira de acessá-la. O máximo que podemos conseguir é resolver os mesmos problemas que as máquinas de Turing.

Os computadores modernos que usamos, na verdade, têm poder de resolução de problemas igual ao das MTs, desde que possamos adicionar mais memória nos computadores reais sempre que precisarmos (lembrando que as MTs têm fita infinita). A verdade é que os computadores atuais evoluem na velocidade, mas não no poder de resolver problemas.

Assim, o modelo matemático Máquina de Turing, apesar da sua simplicidade, representa um marco dos limites da Computação moderna. Aquilo que você provar que ele não pode resolver, estará provando também que não pode ser resolvido por um computador real.

Aprenda praticando

Questão 1: A linguagem das palavras na forma a^nb^n, com n>1, é livre de contexto. Porém a linguagem $a^nb^nc^n$, com n>1, não é livre de contexto, mas é decidível. Abaixo mostramos uma MT para ela com entradas sobre o alfabeto {a,b,c}. Mostre uma computação para a palavra “abc” e diga se a palavra é aceita por T_3.



>> À diagramação: Nas setas, trocar: ponto-e-vírgula por barra "/". Trocar também a letra **L** (left) por **E** (esquerda) e trocar **R** (right) por **D** (direita) <<

Resolução:

Iniciamos a máquina T_3 com o estado inicial seguido da palavra. Depois, fazemos os movimentos válidos na MT até ela parar (quando não tiver mais movimento válido). A computação resultante é mostrada abaixo:

□ q_0 abc

|- A q_1 bc

|- AB q_2 c

|- A q_3 BC

|- □ q_3 ABC

|- A q_0 BC

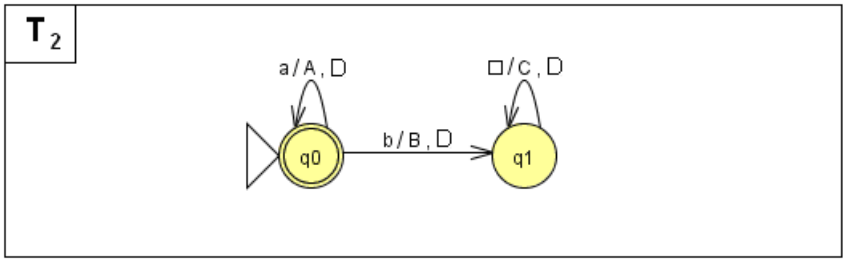
|- AB q_4 C

|- ABC q_4 □

|- ABC□ q_6 □

Veja que a computação termina no estado de aceitação q_6. Isso nos permite afirmar que a palavra abc é aceita!

Questão 2: Represente formalmente o autômato T_2 do exemplo de loop.



Resolução:

Os estados formam o conjunto $\{q_0, q_1\}$, onde q_0 é estado inicial e o único estado de aceitação. Os símbolos de entrada são {a,b}, como pode ser lido na descrição original na seção 2.2. Já o alfabeto da pilha tem a mais os símbolos A, B, C e □, onde □ é o símbolo branco.

Com isso, podemos terminar a representação formal de T_2. Ela é dada abaixo:

$T_1 = (\{q_0, q_1\}, \{a,b\}, \{a,b,A, B,C,□\}, \delta_2, q_0, □, \{q_0\})$, onde δ_2 é:

	a	**b**	**A**	**B**	**C**	**□**
q_0	(q_0,A,D)	(q_1,B,D)	-	-	-	-
q_1	-	-	-	-	-	(q_1,C,D)
q_2	-	-	-	-	-	-

Em um caso como este, que a MT não está definida para muitas configurações, teria sido mais simples representar os valores da função assim:

$$\delta_2(q_0,a) = (q_0,A,D)$$

$$\delta_2(q_0,b) = (q_1,B,D)$$

$$\delta_2(q_1, \square) = (q_1, C, D)$$

As duas maneiras são equivalentes, então fique à vontade para escolher como fazer, nos exercícios.

Saiba mais

Todos os livros das referências influenciaram na escrita deste capítulo. No entanto, a definição das máquinas de Turing que demos foi baseada no livro de Hopcroft, Ullman e Motwani e outros autores. Recomendamos ao aluno interessado procurar neste livro várias definições alternativas para a Máquina de Turing: não-determinística, com múltiplas fitas, com fita infinita apenas à esquerda, etc. Algo interessante que o livro apresenta é a demonstração da equivalência entre todas essas definições e a definição da MT determinística que usamos.

Atividades e Orientações para estudo

Abaixo apresentamos algumas atividades que servirão para você, cursista, fixar o assunto aqui apresentado. Recomendamos que você tente resolver e que, em caso de dúvidas, interaja com os tutores e com colegas no ambiente para tentar tirar as dúvidas.

Questão 1: Tomando como base a máquina de Turing T_3 dada antes, mostre a computação de cada uma das cadeias abaixo e diga se cada cadeia é aceita:

e) ab

f) aabbcc

g) aabc

Questão 2: Dê a representação formal de T_3.

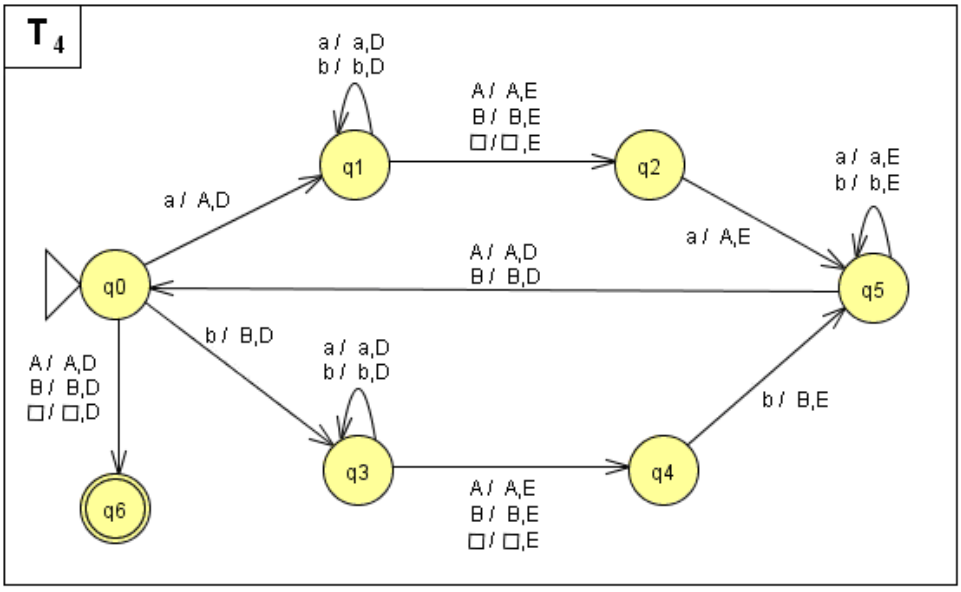
Questão 3: Tomando como base a máquina de Turing T_4 com alfabeto de entrada {a,b} dada abaixo, mostre a computação de cada uma das seguintes cadeias e diga se cada cadeia é aceita:

a) aa

b) abba

c) aab

d) bb



Questão 4: Crie uma máquina de Turing com alfabeto de entrada {0,1} que entre em loop infinito para alguma palavra de entrada. (DICA: você pode ficar apagando e reescrevendo algo na fita ou você pode fazer avançar a fita na direção dos símbolos brancos indefinidamente).

Questão 5: Explique a ligação entre os conceitos de *problemas computáveis* e *linguagens decidíveis*. Destaque a diferença entre eles.

Resumo

Neste fascículo, vimos dois novos modelos computacionais: Autômato com Pilha e Máquina de Turing. Em especial, vimos que o modelo Máquina de Turing, é o modelo computacional mais poderoso que existe. Ou seja, ele é o modelo capaz de *decidir* mais linguagens e, também, de resolver mais problemas. Até mesmo os computadores modernos que usamos são, no máximo, tão capazes quando as Máquinas de Turing.

A comparação pode parecer estranha porque programamos um mesmo computador para resolver vários problemas enquanto os exemplos de MTs que vimos são específicas para um problema. Porém, a verdade é que podemos criar uma MT Universal, que pode ser "programada" como os computadores modernos. Esse assunto não foi visto, mas fica como sugestão de leitura complementar.

Outro detalhe no qual não nos aprofundamos é que existem certas linguagens e certos problemas matemáticos que *não* podem ser decididos por nenhuma Máquina de Turing. Este é um resultado importante, que serve para entendermos os limites teóricos dos computadores (atuais e futuros). Recomendamos que você procure nas referências pelo assunto "Indecidibilidade" (ou "Decidibilidade", dependendo do livro) para saber mais.

Referências

HOPCROFT, John E.; ULLMAN, Jeffrey D.; MOTWANI, Rajeev. **Introdução à teoria de autômatos, linguagens e computação**. Rio de Janeiro: Campus, c2003. 560 p. ISBN 8535210725

MENEZES, Paulo Fernando Blauth. **Linguagens formais e autômatos**. 4. ed. Porto Alegre: Sagra Luzzatto, 2002. 165p. ISBN 8524105542

SIPSER, Michael. **Introdução à teoria da computação**. [São Paulo]: Thomson Learning, 2007. 460 p. ISBN 9788522104994.