Teoria dos Autômatos e Linguagens Formais

aula 19: Recursão no autômato de pilha

1 Introdução

Na aula passada, nós descobrimos que as linguagens reconhecidas por autômatos de pilha possuem uma coisa as linguagens regulares não tem: os *padrões recursivos*.

Ora, mas se a diferença é apenas essa, então nós podemos adicionar a capacidade de fazer recursão aos autômatos finitos.

E daí, eles poder fazer tudo o que um autômato de pilha é capaz de fazer.

Vamos ver como a coisa funciona.

2 Autômatos recursivos

Lembre que o procedimento de composição recursiva consiste em introduzir um bloco no interior de um outro bloco.

Por exemplo,

```
a....b
  ab
```

Logo, para reconhecer esse tipo de coisa, nós precisamos ter a capacidade de

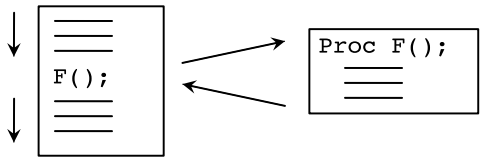
- iniciar a leitura de um bloco no meio da leitura de outro bloco
- e depois voltar para terminar a leitura do bloco que foi interrompida

De maneira mais geral, isso corresponde à capacidade de

- iniciar uma computação no meio de uma outra computação
- e depois voltar para terminar a computação que foi interrompida

E isso corresponde a uma *chamada de função*.

Quer dizer, quando o nosso programa faz uma chamada de função



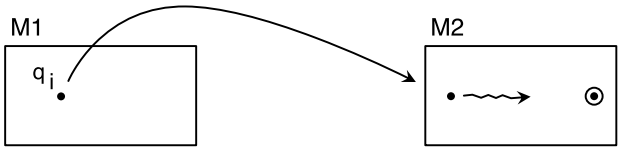
o fluxo de execução que vinha sendo realizado é interrompido por um momento, daí o código da função é executado e, quando ele termina, o fluxo de execução interrompido é retomado outra vez.

Mas, como é que isso funciona no mundo dos autômatos?

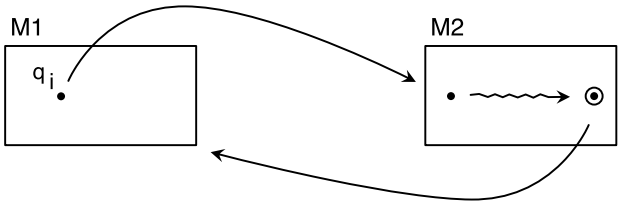
Bom, nós já vimos uma ideia semelhante na aula 02, quando nós construímos autômatos que tinham outros autômatos como componentes.

Quer dizer, a ideia é que nós podemos ter dois autômatos M_1 e M_2

e que, em um certo ponto da computação de M_1, ao invés do autômato ler um símbolo da palavra de entrada, ele dá início a uma computação de M_2



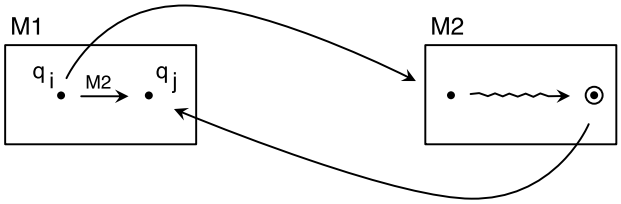
Daí, quando a computação de M_2 acaba, o controle volta para o autômato M_1



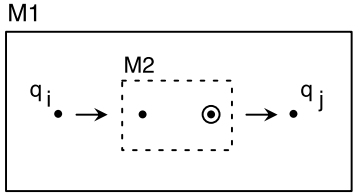
A maneira natural de operacionalizar essa ideia consiste em definir um novo tipo de transição que corresponde à ideia de chamada de função

$$q_i \xrightarrow{M_2} q_j$$

A ideia é que, no momento em que o controle volta retorna para o autômato M_1, a computação continua a partir do estado q_j



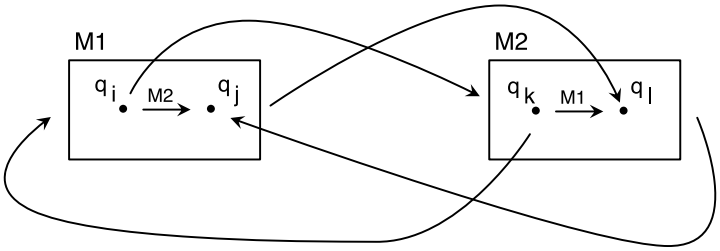
À primeira vista, isso não parece adicionar poder computacional aos autômatos, pois nós sempre podemos reorganizar a coisa da seguinte maneira



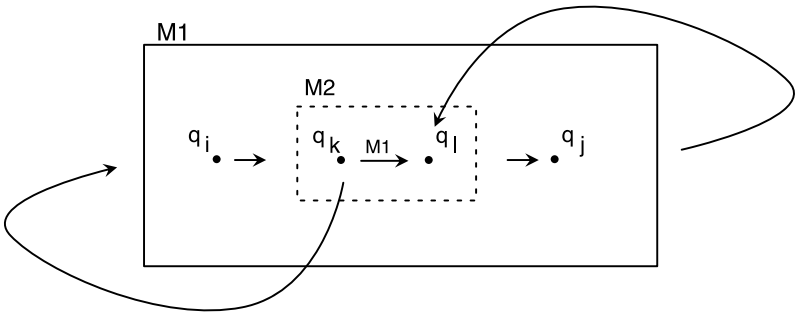
Quer dizer, nesse esquema, M_2 não é realmente um autômato separado, mas apenas uma parte do autômato M_1 que nós podemos querer desenhar em separado.

— (*Era exatamente isso o que estava acontecendo na aula 02*)

Mas, a coisa fica realmente interessante quando a gente se dá conta de que, em princípio, o autômato M_2 também pode fazer uma chamada ao autômato M_1



E daí, quando a gente embute o autômato M_2 dentro do autômato M_1 outra vez



o que a gente vê é o autômato M_1 fazendo uma chamada (recursiva) a si mesmo.

Essa é a chave para o reconhecimento de padrões recursivos.

E é isso que dá um ganho de poder computacional aos autômatos.

Vejamos um exemplo concreto.

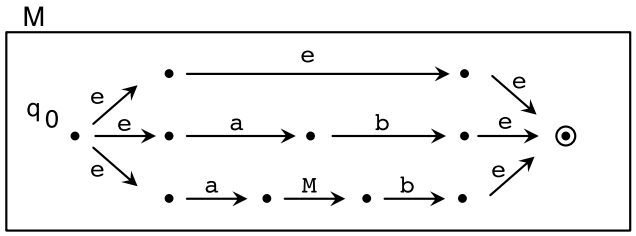
Considere a linguagem $a^n b^n$.

O esquema abaixo ilustra o padrão recursivo dessa linguagen

a . . . . . b
a . . . . b
a b

Quer dizer, logo após a leitura do símbolo a do bloco ab, nós podemos iniciar a leitura de um novo bloco.

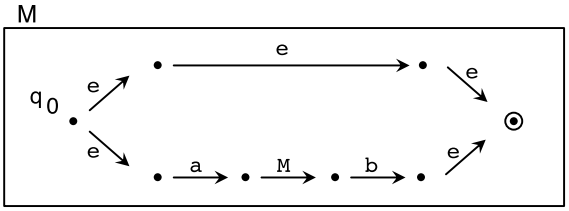
Como base nessa observação, não é difícil chegar ao seguinte autômato recursivo que reconhece a linguagem $a^n b^n$



Esse é um autômato recursivo não-determinístico que, no início da sua computação, advinha se ele deve

- apenas passar para o estado final
- ler o bloco `ab`
- ler o símbolo `a`, realizar uma chamada recursiva, e ler o símbolo `b` após o retorno da chamada

Não é difícil ver que esse autômato pode ser ligeiramente simplificado para



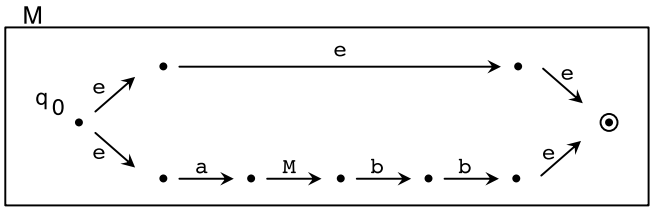
— (*Você consegue ver que esse autômato também reconhece a linguagem* $\mathtt{a}^n\mathtt{b}^n$*?*)

A seguir nós vamos ver mais exemplos de autômatos recursivos.

Exemplos

a. A linguagem $\mathtt{a}^n\mathtt{b}^{2n}$

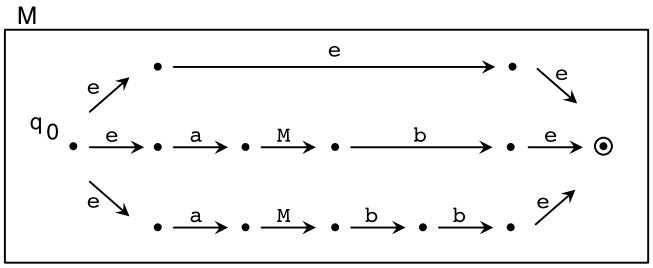
Esse exemplo é muito parecido com o anterior



◇

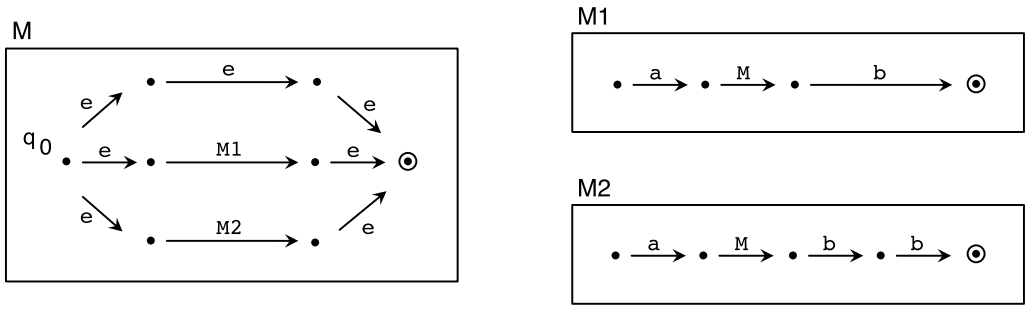
b. A linguagem $\mathtt{a}^n\mathtt{b}^m$, onde $\mathtt{n} \leq \mathtt{m} \leq 2\mathtt{n}$

Aqui nós precisamos do não-determinismo para advinhar quantos `b`'s devem ser lidos



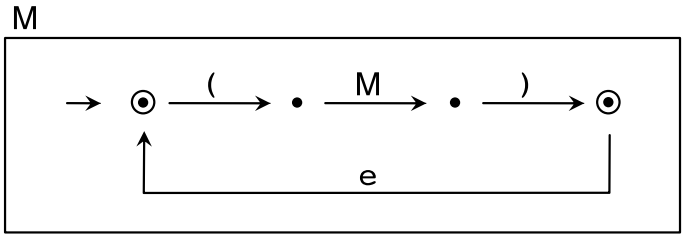
E nós também podemos decompor essa computação em 3 módulos

◇



c. PARBAL

A novidade desse exemplo é o laço que permite reconhecer padrões repetitivos



Quer dizer, no caso da palavra vazia a computação termina sem sair do estado inicial q_0.

No caso de um bloco construído recursivamente

((( )))

o autômato lê o primeiro símbolo (, realiza a chamada recursiva, e lê o último símbolo) após o retorno da chamada.

E no caso de um padrão repetitivo

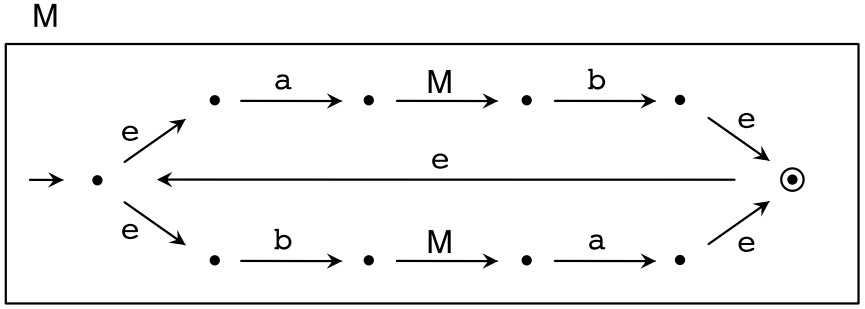
( ) ( ) ( )

ao final da leitura de cada bloco o autômato volta de q_3 para q_0 para iniciar a leitura de um novo bloco.

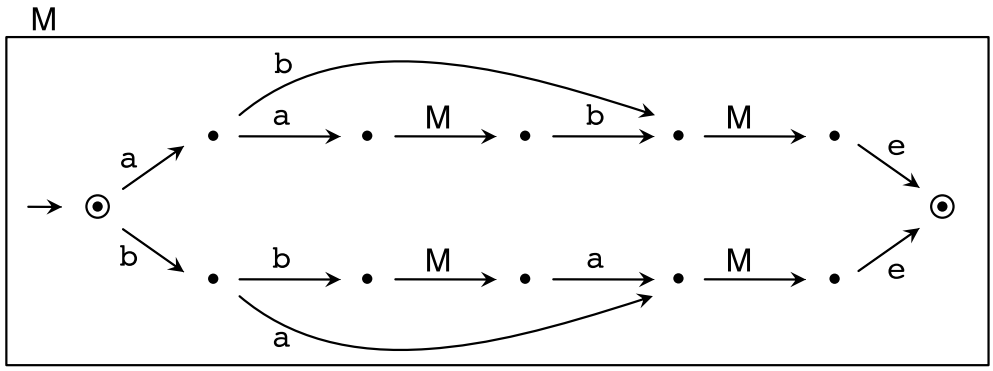
◇

d. BAL

Seguindo as ideias dos exemplos anteriores, não é difícil chegar no seguinte autômato que reconhece a linguagem BAL



Mas, nós também podemos fazer as coisas assim



Você consegue ver que esse autômato reconhece a linguagem BAL*?*

Você viu como a leitura de padrões repetitivos foi implementada nesse autômato?

Você percebeu que esse autômato é determinístico?

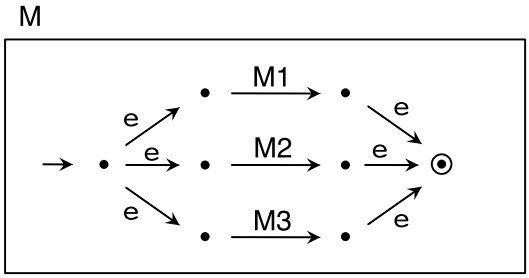
◇

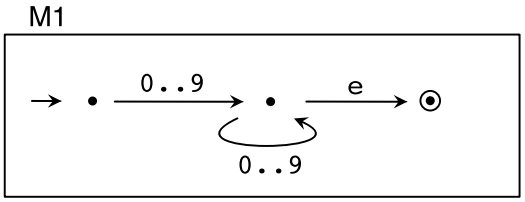
e. EXPR

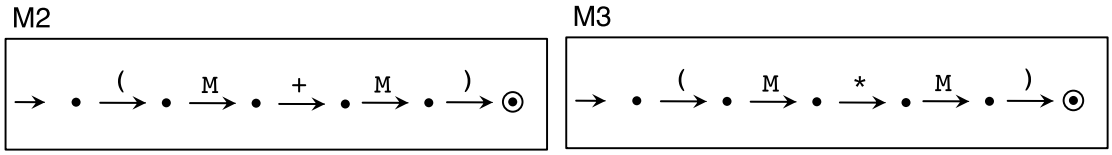
Os 3 casos da linguagem EXPR – `cp` são

- a expressão é apenas um número: $<$ `num` $>$
- a expressão é a soma de duas subexpressões: $($ $<$ `expr` $>$ $+$ $<$ `expr` $>$ $)$
- a expressão é a multiplicação de duas subexpressões: $($ $<$ `expr` $>$ $*$ $<$ `expr` $>$ $)$

A ideia é que nós podemos utilizar um módulo para cada um desses casos, e chegar à seguinte implementação







◇

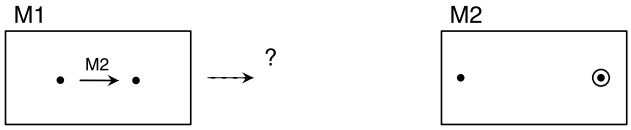
3 Recursão no autômato de pilha

A solução que nós acabamos de ver é legal, mas ela é um pouco enganadora.

Quer dizer, ela até resolve o problema.

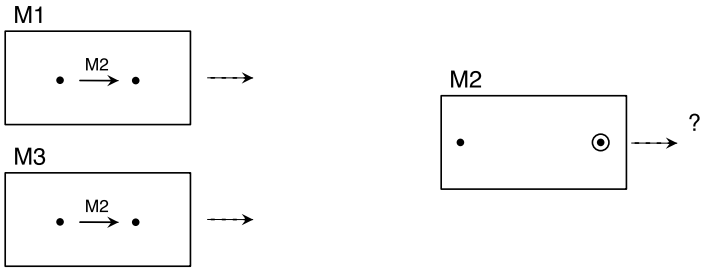
Mas ela não mostra todos os detalhes da solução.

Quer dizer, quando o módulo M_1 faz uma chamada para o módulo M_2



como é que o controle da computação é passado de um módulo para o outro?

E mais importante, se dois módulos M_1 e M_2 fazem chamadas para um terceiro módulo M_3



quando a computação de M_3 termina, como é que nós sabemos para onde nós devemos voltar?

Na prática, quando fazemos chamadas de função em nossos programas, é o sistema operacional quem cuida desses problemas.

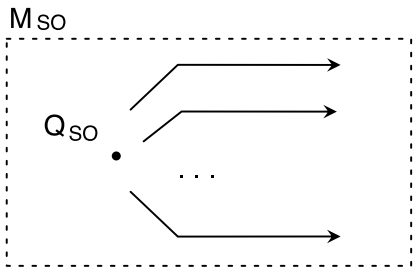
Mas, quando nós estudamos a teoria da computação, todos os detalhes devem estar à mostra para que a gente entenda como a computação realmente acontece.

A boa notícia é que tudo o que é preciso para gerenciar chamadas de função e retornos é uma pilha.

Quer dizer, agora nós vamos ver que tudo o que os autômatos recursivos são capazes de fazer (com a ajuda do sistema operacional) também pode ser feito por um autômato de pilha (sem a ajuda de ninguém).

E a ideia é bem simples: *nós vamos colocar o sistema operacional dentro do autômato de pilha.*

Mais especificamente, o papel do sistema operacional vai ser realizado por um estado Q_{SO}



A ideia é que, momento em que um estado q_i quer fazer uma chamada para o módulo M

$$q_i \xrightarrow{M} q_j$$

nós colocamos o nome do estado inicial do módulo M na pilha, e fazemos a transição para o "sistema operacional"

$$q_i \xrightarrow{e[e,Q_M]} Q_{SO}$$

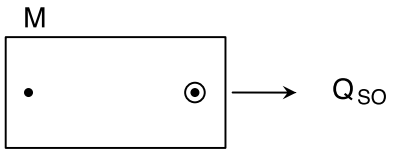
Daí, o sistema operacional remove esse nome do topo da pilha, e realiza a transição para o estado inicial do módulo M

$$Q_{SO} \xrightarrow{e[Q_M,e]} Q_M$$

Certo.

Mas, ainda é preciso voltar para o estado q_j após o termino da computação do módulo M.

Bom, a ideia é que após o termino da computação de M o controle volta para o sistema operacional



E o sistema operacional sabe para onde voltar em seguida, porque no momento da chamada nós também colocamos essa informação na pilha

$$q_i \xrightarrow{e[e,Q_M q_j]} Q_{SO}$$

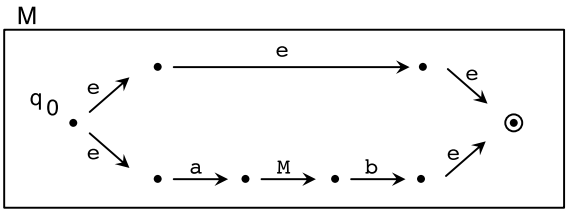
Quer dizer, após o termino da computação de M, o controle volta para o sistema operacional que realiza a seguinte transição

$$Q_{SO} \xrightarrow{e[q_j,e]} q_j$$

Pronto, é só isso!

Vejamos como a coisa funciona em um exemplo concreto.

Abaixo nós temos o autômato recursivo que reconhece a linguagem $a^n b^n$



As duas modificações que nós fazemos nesse módulo são as seguintes

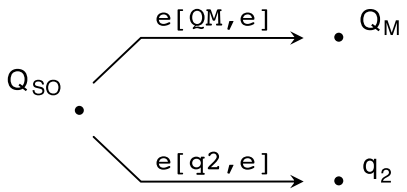
- a chamada recursiva é substituída por uma transição para o estado Q_{SO}

$$q_1 \xrightarrow{e[e,Q_M q_2]} Q_{SO}$$

- o estado final do módulo agora também realiza uma transição para o estado Q_{SO}

$$\odot \xrightarrow{e} Q_{SO}$$

O estado Q_{SO} implementa a chamada recursiva e o retorno para o local da chamada da seguinte maneira



Até aqui tudo bem.

Mas, como é que a computação começa e termina?

Bom, a ideia é ter um módulo principal M_P, onde se encontram o estado inicial e o estado final do autômato de pilha, e que realiza a primeira chamada ao módulo M

A figura abaixo apresenta a implementação completa do autômato de pilha