Teoria dos Autômatos e Linguagens Formais

aula 17: Linguagens (legais) para autômatos de pilha

1 Introdução

Os autômatos finitos não fazem nada de interessante.

Quer dizer, verificar se a palavra de entrada tem uma quantidade ímpar de `b`'s não é uma coisa interessante.

Mas, verificar se a palavra de entrada tem a mesma quantidade de `a`'s e `b`'s também não é uma coisa interessante.

Então, agora está na hora de ver as coisas interessantes que o autômato de pilha pode fazer.

2 Linguagens legais

Imagine que alguém apaga os números e operações de uma expressão aritmética

$$\begin{array}{lcl} \texttt{4} + \big(\texttt{7} * \texttt{5}\big) & \Rightarrow & (\ ) \\ \big(\texttt{5} + \texttt{1}\big) * \big(\texttt{4} + \texttt{7}\big) & \Rightarrow & (\ )(\ ) \\ \big(\big(\texttt{4} + \texttt{7}\big) * \big(\texttt{5} + \texttt{2}\big)\big) + \texttt{1} & \Rightarrow & (\ (\ )\ (\ )\ ) \end{array}$$

Daí, tudo o que sobra é

→ *uma sequência de* `abre-parênteses` *e* `fecha-parênteses`

onde todo `abre-parênteses` *corresponde a uma* `fecha-parênteses` *que aparece mais adiante*

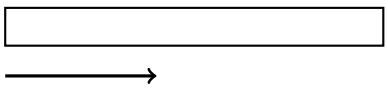
e todo `fecha-parênteses` *corresponde a uma* `abre-parênteses` *que apareceu antes*

Essa é a linguagem PARBAL — (*a linguagem dos parênteses balanceados*)

E a nossa tarefa é construir um autômato de pilha para essa linguagem.

Mas, como é que a gente faz isso?

Bom, o autômato de pilha vai ler a palavra de entrada da esquerda para a direita



Daí,

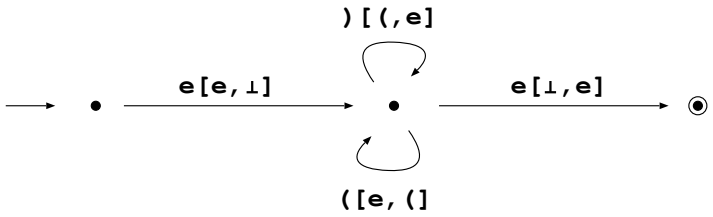
- para cada `abre-parênteses`, ele deve encontrar um `fecha-parênteses` mais tarde
- e para cada `fecha-parênteses`, ele deve ter encontrado um `abre-parênteses` antes

Então a ideia é

1. colocar os `abre-parênteses` na pilha — (*a medida que eles são encontrados*)
2. remover os `abre-parênteses` da pilha — (*quando os* `fecha-parênteses` *são encontrados*)

No final, a palavra de entrada é aceita se a pilha está vazia — (*e se a gente conseguiu ler a palavra inteira*)

A coisa fica assim



A seguir, nós vamos ver outros exemplos.

Exemplos

a. A linguagem PAROP

Agora imagine que as expressões estão completamente parentizadas — (*i.e., cada operação está dentro do seu próprio par de parênteses*)

Por exemplo,

$$\Big(\big((\,1 + 3\,) * 4 \big) + \big(2 * 5 \big) \Big)$$

E imagine que dessa vez nós removemos apenas os números da expressão — (*e deixamos os parênteses e as operações*)

$$\begin{array}{lcl} \big(4 + (7 * 5) \big) & \Rightarrow & (+(*)) \\ ((9 + 6) * (10 + 2)) & \Rightarrow & ((+) * (+)) \\ \Big(\big(\big((1 + 2) * 3 \big) + 4 \big) * 5 \Big) & \Rightarrow & ((((+)*)+)*) \end{array}$$

Essa é a linguagem PAROP.

E a tarefa agora é construir um autômato de pilha para essa linguagem.

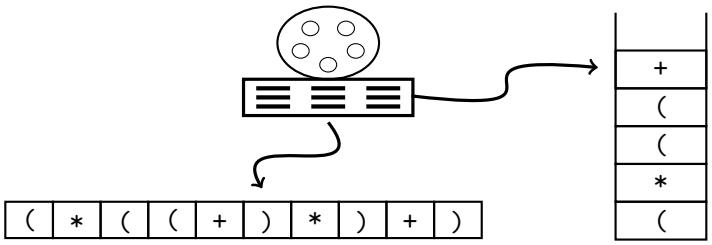
Só que isso não é difícil.

Quer dizer, vamos considerar o seguinte exemplo

$$(*(\ (+)*)+)$$

Imagine que o autômato vai empilhando tudo o que ele encontra.

E vamos examinar o conteúdo da pilha no momento em que o autômato chega na `fecha-parênteses`



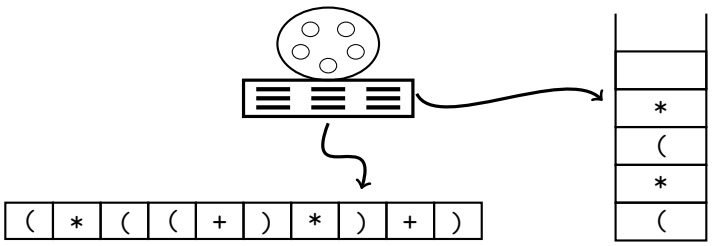
A operação associada a esse `fecha-parênteses` é o +.

E esse + se encontra no topo da pilha.

Então, o autômato pode verificar isso.

E pode remover o + e o `abre-parênteses` da pilha — (*no momento da leitura do* `fecha-parênteses`)

Daí, quando o autômato encontra o próximo `fecha-parênteses`, a pilha está assim

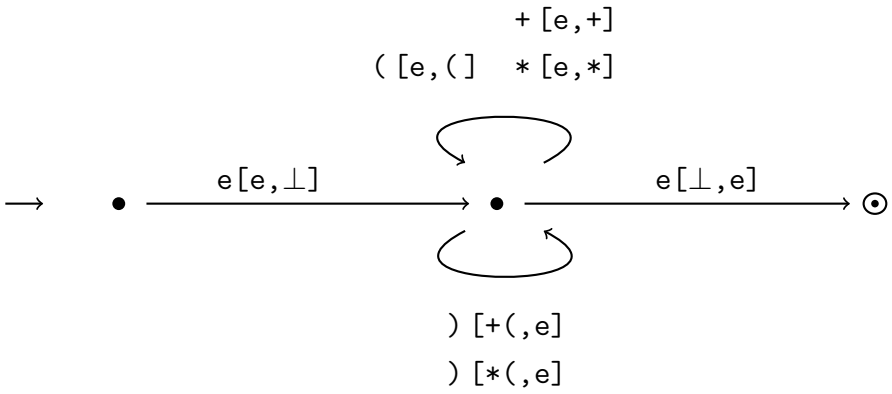


Mais uma vez, a operação associada a esse `fecha-parênteses` está no topo da pilha.

Então, é só fazer a mesma coisa outra vez.

E continuar fazendo as coisas assim.

O autômato de pilha fica assim



◇

b. A linguagem PAROP2

Agora nós podemos modificar o autômato para que ele trabalhe com palavras que não vem de expressões completamente parentizadas.

Por exemplo,

4 + 7 * 5	⇒	+ *
2 * (4 + 7 + 1) * 5	⇒	* (+ +) *
((4 + 7) * (5 + 2)) + 1	⇒	((+) * (+)) +

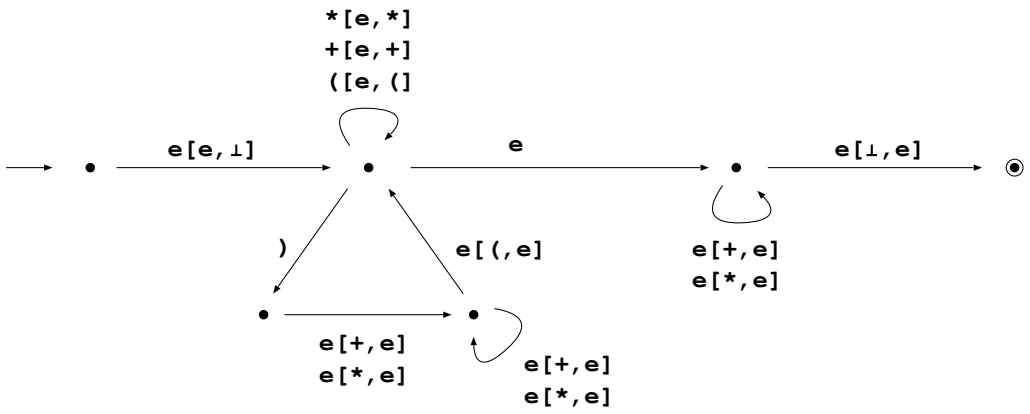
Bom, a primeira diferença é que agora podem haver várias operações dentro de um par de parênteses — (*mas precisa haver ao menos uma*)

E a outra diferença é que podem haver operações fora dos parênteses mais externos.

Então, a solução é adicionar um loop ao autômato, para desempilhar todas as operações de um par de parênteses.

E adicionar outro loop para desempilhar as operações que estão fora de todos os parênteses.

A coisa fica assim



◇

c. A linguagem EXPR

Agora considere as expressões completamente parentizadas outra vez

$$(\,4\,+\,7\,)$$

$$(\,(\,18\,+\,5\,)\,*\,2\,)$$

$$(\,(\,5\,+\,7\,)\,*\,(\,15\,+\,2\,)\,)$$

Só que dessa vez nós não vamos remover nada.

Quer dizer, o autômato vai ler tudo.

E no final, ele precisa dizer se a palavra de entrada é uma expressão válida ou não.

Por exemplo, ele precisa rejeitar coisas como

$$(\,4\,+\,+\,7\,)$$

$$(\,(\,18\,+\,5\,)\,2\,)$$

$$(\,5\,+\,7\,)\,*\,(\,15\,+\,2\,)\,)$$

e por aí vai ...

Só que isso não é difícil.

Quer dizer, nós já temos um autômato que faz isso para as expressões onde os números foram apagados.

Então, agora é só modificar o autômato para que ele leia os números.

E aqui, nós vamos fazer uma pequena esperteza.

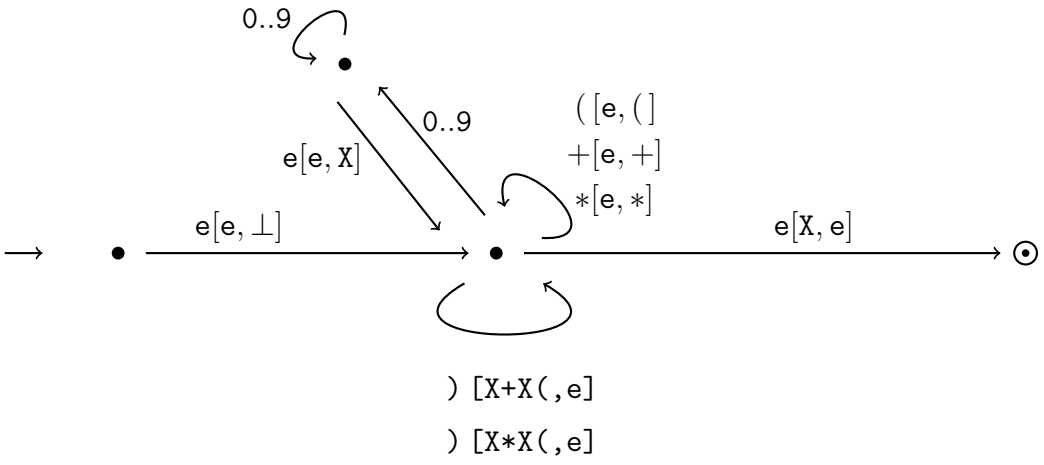
Quer dizer, a ideia é que nós não vamos empilhar os números.

Porque os números podem ter quantidades diferentes de dígitos.

Ao invés disso, a gente vai ter todos os dígitos e deixar um marcador na pilha: a letra `X` — (*para lembrar mais tarde que havia um número ali*)

E daí mais tarde, quando o autômato ler o `fecha-parênteses`, ele espera encontrar a pilha assim

A coisa fica assim



Exercício: Remova o não-determinismo desse autômato.

Exercício: A linguagem EXPR2: expressões aritméticas que não precisam estar completamente parentizadas.

◇

d. A calculadora de paridade

Agora nós vamos modificar o nosso autômato para que ele responda a seguinte pergunta

→ *O resultado da expressão aritmética é par ou ímpar ?*

Bom, a ideia é simples.

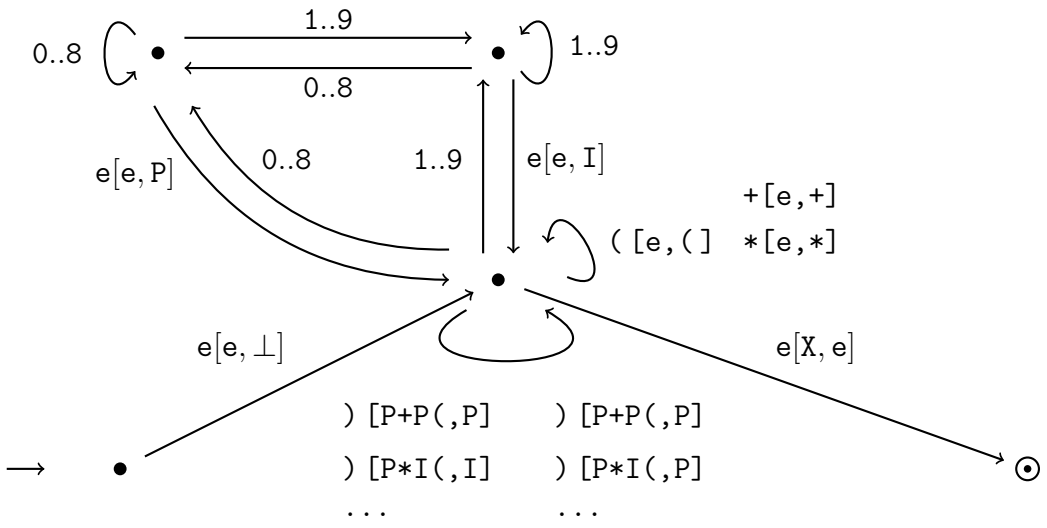
Quer dizer, ao ler os números da expressão, ao invés de colocar um `X` na pilha, o autômato vai colocar

- um `P`, quando o número é par
- um `I`, quando o número é ímpar

Daí, na hora de desempilhar, o autômato utiliza a seguinte lógica

```
P + P = P        P * P = P
P + I = I        P * I = P
I + P = I        I * P = P
I + I = P        I * I = I
```

A coisa fica assim



◇

e. A calculadora de bytes

Agora imagine que a nossa expressão aritmética só tem números menores do que 256

$$\Big(\big(\texttt{2} * (\texttt{6} + \texttt{4}) \big) + \texttt{15} \Big)$$

Então, nós podemos escrever os números em binário

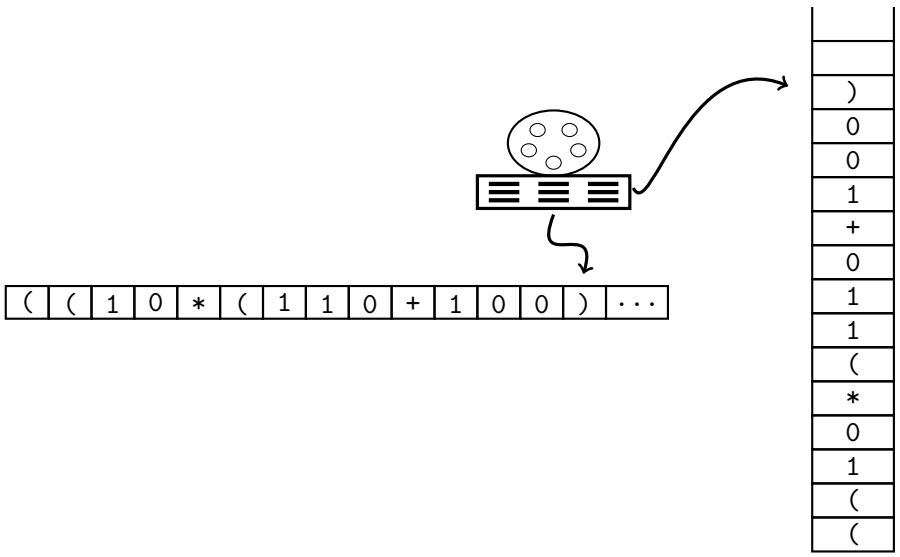
$$\Big(\big(\texttt{10} * (\texttt{110} + \texttt{100}) \big) + \texttt{11111} \Big)$$

onde nós sabemos que cada número vai ter no máximo 8 bits — (*i.e., 1 byte*)

E agora, nós queremos construir um autômato de pilha que calcula o valor dessa expressão.

Bom, a primeira observação é que dessa vez o autômato vai ter que colocar toda a informação que ele lê na pilha.

Daí, a gente nota que na pilha os números ficam invertidos — (*i.e., o bit menos significativo fica em cima*)



De fato, isso não é um problema — (*até ajuda na hora de fazer a conta*)

O problema é que a conta precisa ser feita com o número que está no topo da pilha e o número que está abaixo dele na pilha.

Só que, nós não podemos acessar o número que está mais abaixo na pilha — (*porque a pilha é uma pilha ...*)

E agora?

Bom, agora é a hora para uma esperteza.

Quer dizer, a ideia é utilizar uma segunda pilha.

Só que, um autômato com duas pilhas não é um autômato de pilha — (*porque?*)

Mas, a segunda pilha não precisa ser uma pilha de verdade.

Quer dizer, uma pilha de verdade tem capacidade ilimitada — (*a gente pode empilhar quantos símbolos a gente quiser ...*)

E a pilha que nós vamos utilizar é uma pilha de 1 byte — (*i.e., 8 bits*)

Ora, uma pilha de 1 byt3e é um mecanismo de memória finito.

Um mecanismo de memória finito é basicamente um autômato finito.

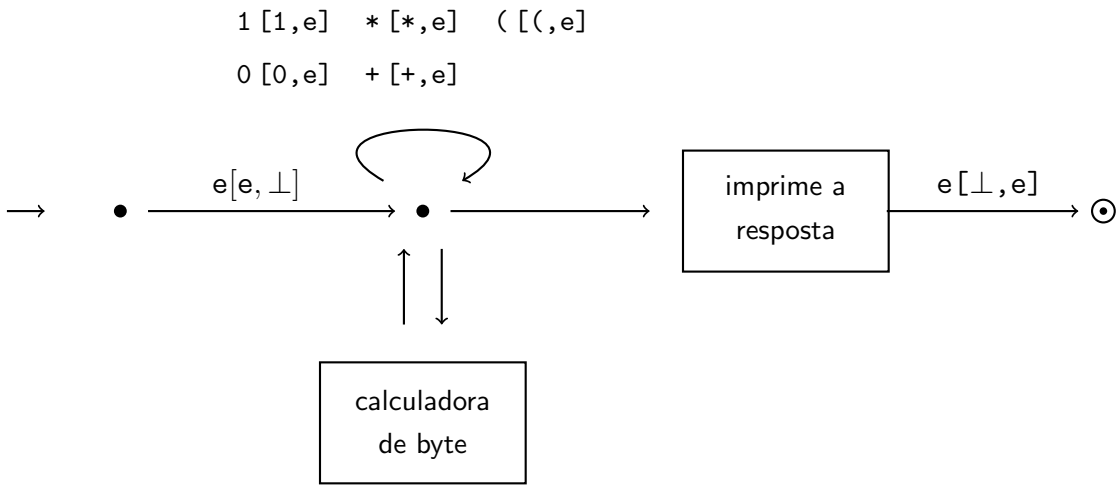
E quando nós adicionamos um autômato finito a um autômato de pilha, nós ainda temos um autômato de pilha.

Legal.

Então a lógica da computação fica assim

1. o autômato de pilha lê e empilha todos os símbolos
2. e quando ele encontra o `fecha-parênteses`,

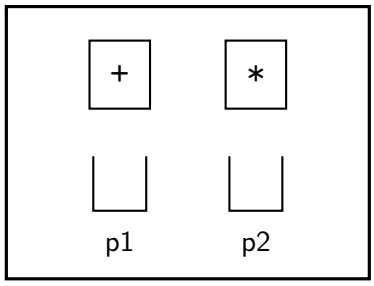
   a computação segue para um autômato finito que faz a conta dentro dos parênteses



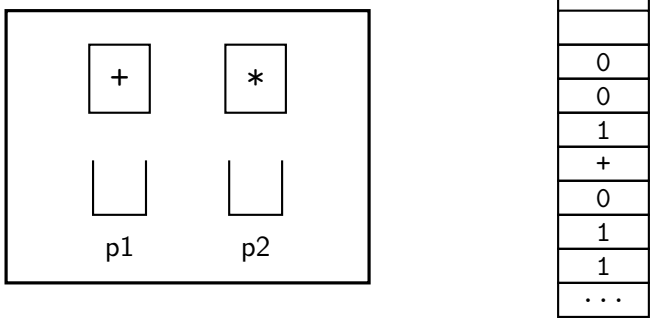
A calculadora de byte é um autômato finito — (*i.e., um mecanismo computacional com memória finita*)

Mas, ao invés de apresentar a sua lógica como um diagrama de estados, nós vamos descrever a sua funcionalidade em alto nível.

Na prática, nós utilizamos duas pilhas e um módulo para cada operação



Quando a calculadora é acionada, o topo da pilha (principal) contém dois números e uma operação



E daí, a coisa funciona assim

1. o primeiro número é copiado para p1 e depois para p2
   — (*para que o dígito menos significativo fique em cima*)
2. a seguir, o autômato verifica qual é a operação que deve ser realizada,
   e seque para o módulo correspondente
3. a operação é feita lendo a pilha p1 e a pilha principal,
   e o resultado é colocado na pilha p2 — (*no caso da operação de soma*)
4. o resultado em p2 tem o dígito mais significativo no topo

   então, ao copiar o resultado p/ a pilha principal, o dígito menos significativo está no topo
5. agora, a computação pode continuarl normalmente

◇

f. A linguagem REG

Essa é a linguagem das expressões regulares

$$(\mathtt{ab} \cup \mathtt{ba})^*, \qquad \mathtt{b}(\mathtt{a} \cup \mathtt{ba})^*(\mathtt{ab})^*, \qquad \big(\mathtt{a}\,(\mathtt{aaa}^*\mathtt{b})^*\big)^*, \qquad \ldots$$

E a nossa tarefa é construir um autômato de pilha que verifica se a palavra de entrada é uma expressão regular válida ou não.

Mas, isso é muito parecido com a tarefa de reconhecer expressões aritméticas.

A primeira diferença é que ao invés da letra X (que representa um número), nós vamos utilizar a

letra `E` para representar uma expressão regular.

Mais precisamente, o `a` e `b` são expressões regulares

$$\texttt{a}, \texttt{b} \quad \longrightarrow \quad \texttt{E}$$

E se a gente acrescenta um `a` ou `b` a uma expressão regular, ela ainda é uma expressão regular

$$\texttt{Ea}, \texttt{Eb} \quad \longrightarrow \quad \texttt{E}$$

Daí, a operação de união ($\cup$) funciona como as operações aritméticas

$$(\texttt{E} \cup \texttt{E}) \quad \longrightarrow \quad \texttt{E}$$

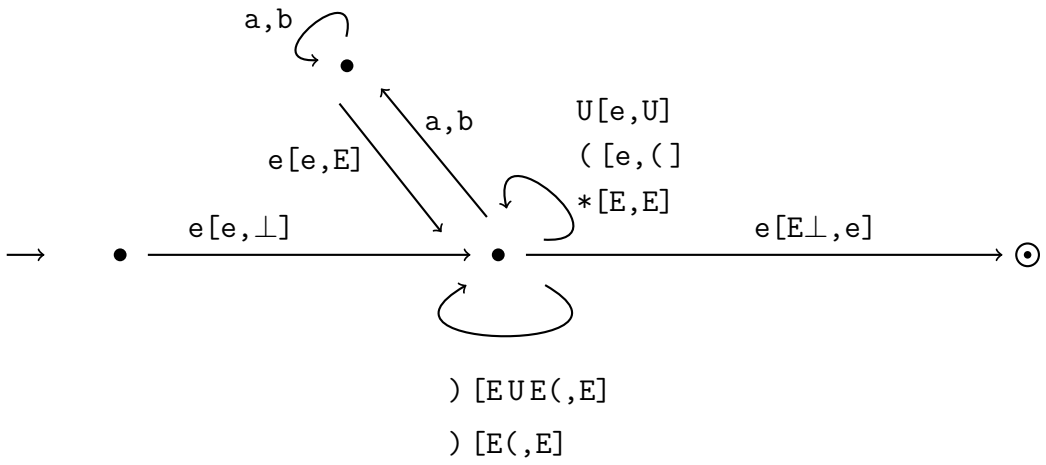
Finalmente, a operação estrela ($\star$) só pode aparecer depois de uma expressão regular (`E`) — (*i.e., ela não pode ser o 1o símbolo, e não pode vir depois de* `abre-parênteses` *ou a união* ($\cup$))

Mas, é fácil verificar isso.

Quer dizer, no momento em que a gente encontra a operação estrela ($\star$), basta verificar se o topo da pilha é uma expressão regular (`E`)

$$\texttt{E}* \quad \longrightarrow \quad \texttt{E}$$

Agora, é fácil construir o autômato de pilha



◇

g. LOOP

A linguagem LOOP é uma linguagem de programação muito simples que possui apenas dois tipos de instrução

```
X ← <expr aritmética>;
```

```
Loop X
    <bloco de comandos>
EndLoop
```

Isto é, um comando de atribuição e um laço de repetição.

O número de voltas do laço é determinado pelo valor da variável `X` no momento em que o programa chega ao laço.

Modificações no valor dessa variável dentro do bloco de comandos não afetam o número de voltas do laço.

Por exemplo, o programa abaixo troca os valores de `X` e `Y`, caso o primeiro seja menor do que o segundo

```
Z  <--  Y - X;
LOOP Z
   X  <--  X + 1;
   Y  <--  Y - 1;
ENDLOOP
```

Note que a coisa funciona porque na situação emq em `Z` vale zero ou um número negativo, o programa não entra no laço.

Certo.

Agora imagine que um programa em LOOP é uma palavra — (*i.e., uma sequência de caracteres*)

Daí, a gente pode definir a linguagem

ProgLOOP = palavras que correspondem a programas válidos em LOOP

Por exemplo, o programa acima é uma palavra de ProgLOOP.

Mas o seguinte programa não é

```
LOOP X
   Y  <--  X + * 41;
ENDLOOP
LOOP Y
ENDLOOP
LOOP Z
```

Quer dizer, nós temos 3 problemas aqui

- a expressão aritmética no comando de atribuição não é válida
- o segundo laço não tem um bloco de comandos
- as quantidades de termos `LOOP` e `ENDLOOP` não são iguais

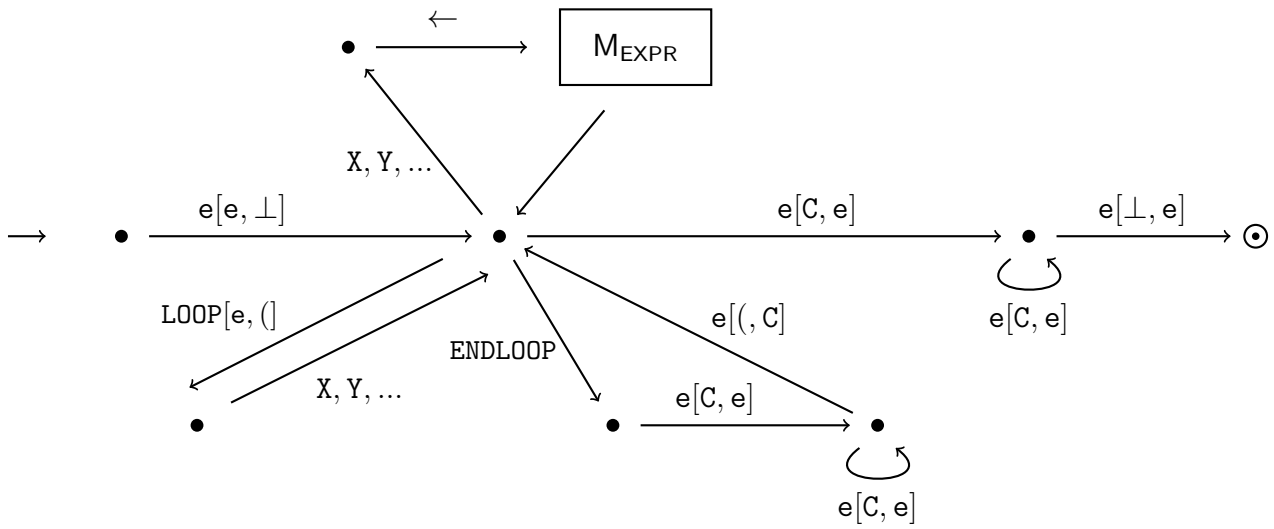
A tarefa agora é

→ *Construir um autômato de pilha que reconhece a linguagem* ProgLOOP

Bom, é bem fácil ver que os termos `LOOP` e `ENDLOOP` fazem o papel de abre-parênteses e fecha-parênteses.

E nós podemos reutilizar o autômato que reconhece EXPR para verificar se os comandos de atribuição são válidos.

Daí, a coisa fica assim


←
M_{EXPR}
X, Y, ...
e[e, ⊥]
e[C, e]
e[⊥, e]
LOOP[e, (]
ENDLOOP
e[(, C]
e[C, e]
X, Y, ...
e[C, e]
e[C, e]

◇