Automato com Pilha
Pushdown Automaton

Algoritmos Recursivos e Pilhas

Princípio Geral em Computação: Qualquer algoritmo recursivo pode ser transformado em um não-recursivo usando-se uma pilha e um while-loop, que termina apenas quando a pilha está vazia.

EX: JVM mantém os registros de ativação de cada chamada de método. Considere:

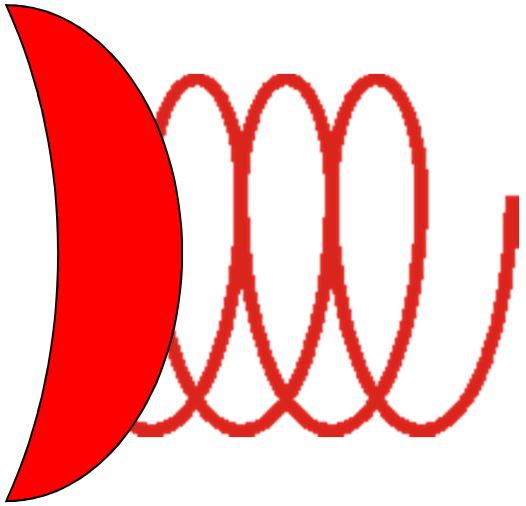
```
long factorial(int n){
  if (n<=0) return 1;
  return n*factorial(n-1);
}
```

Como JVM executa `factorial(5)`?

Algoritmos Recursivos e Pilhas

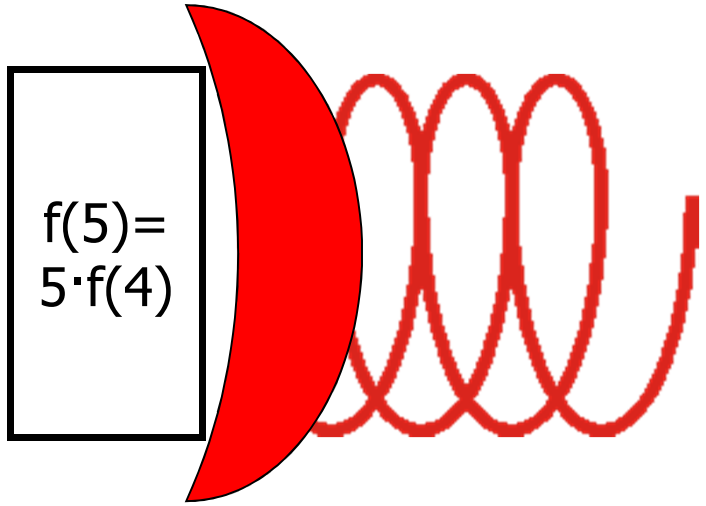
```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```

Compute 5!



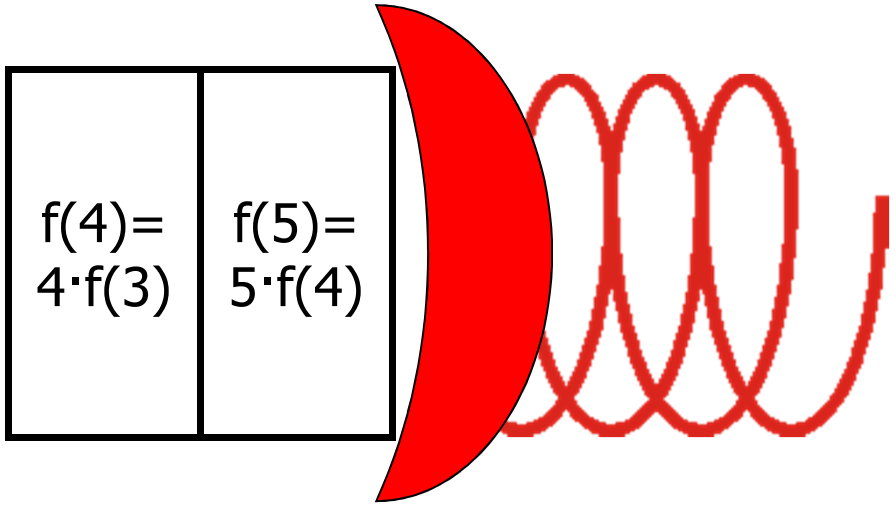
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



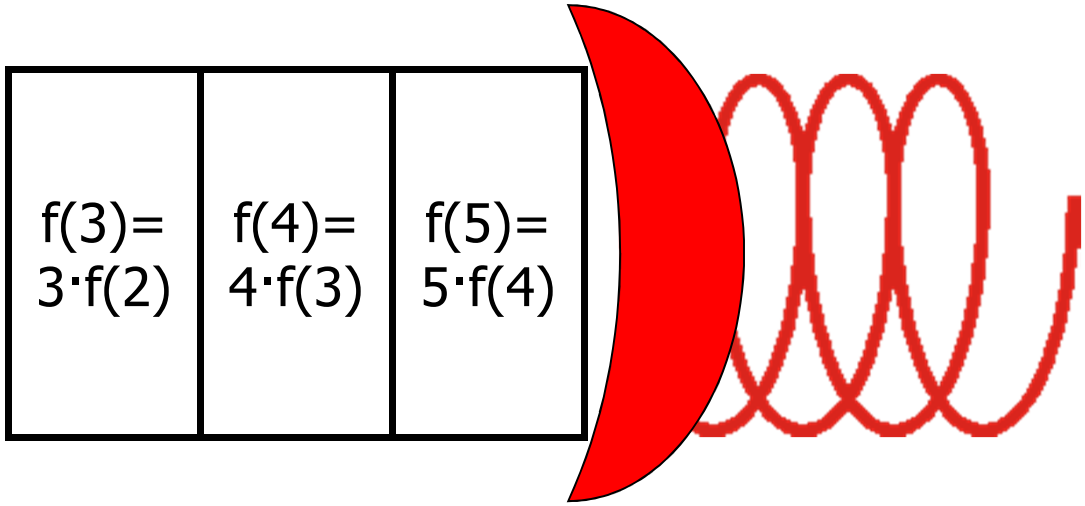
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



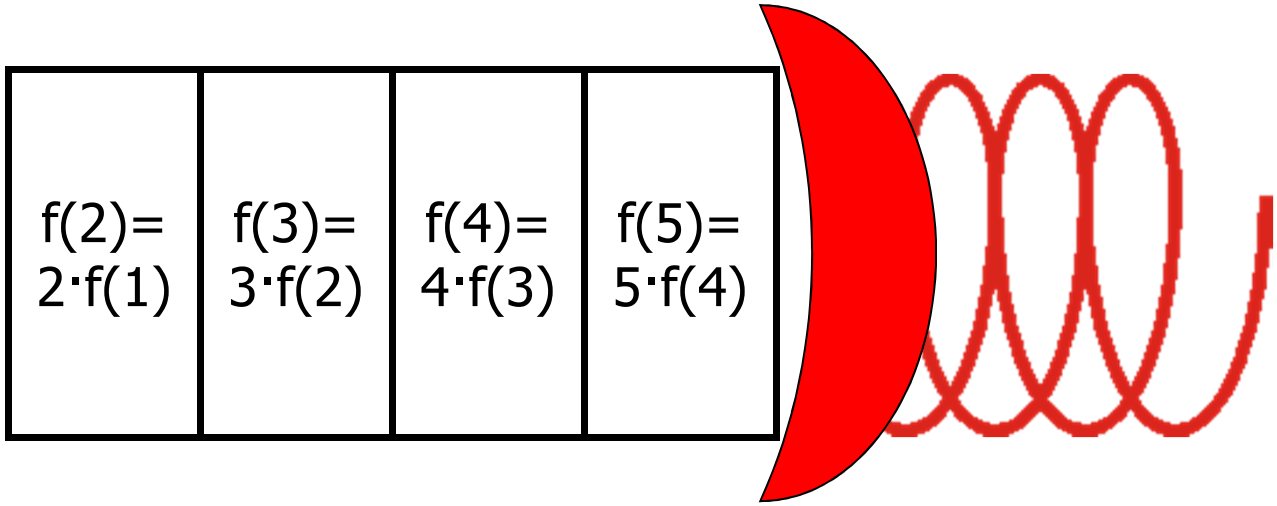
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



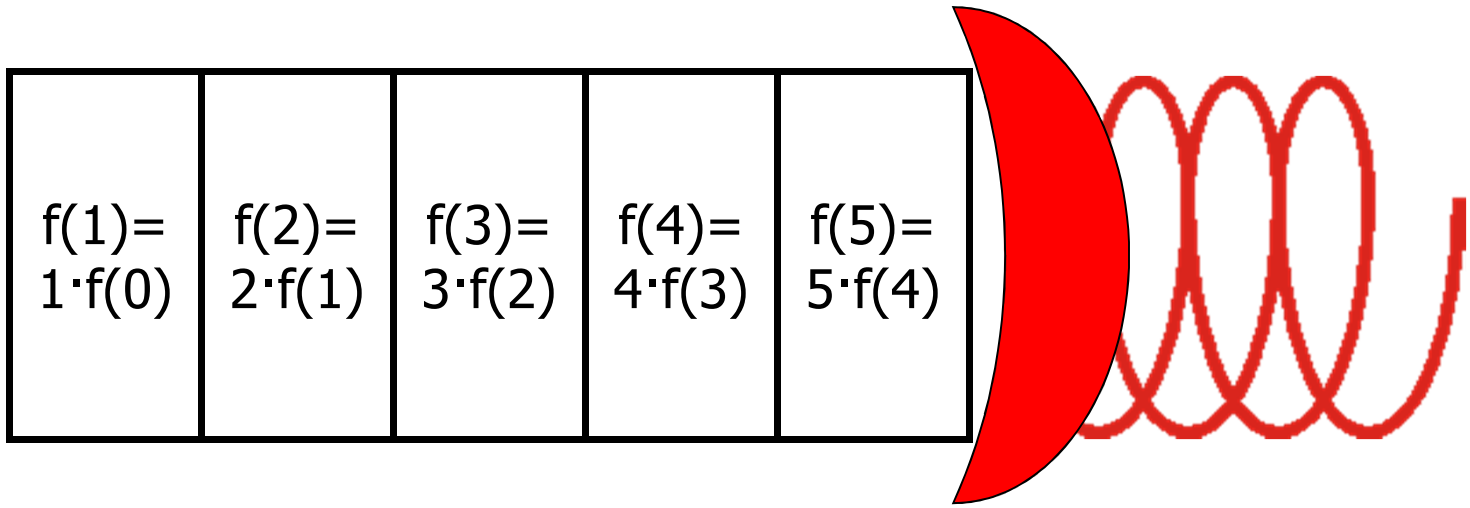
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



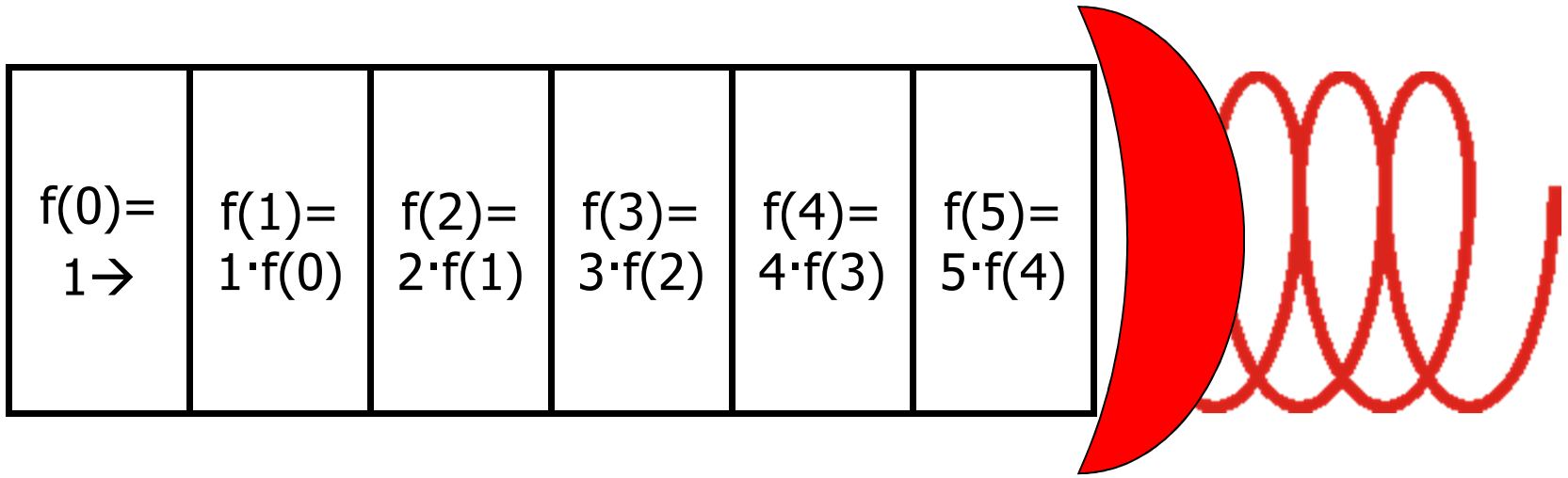
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



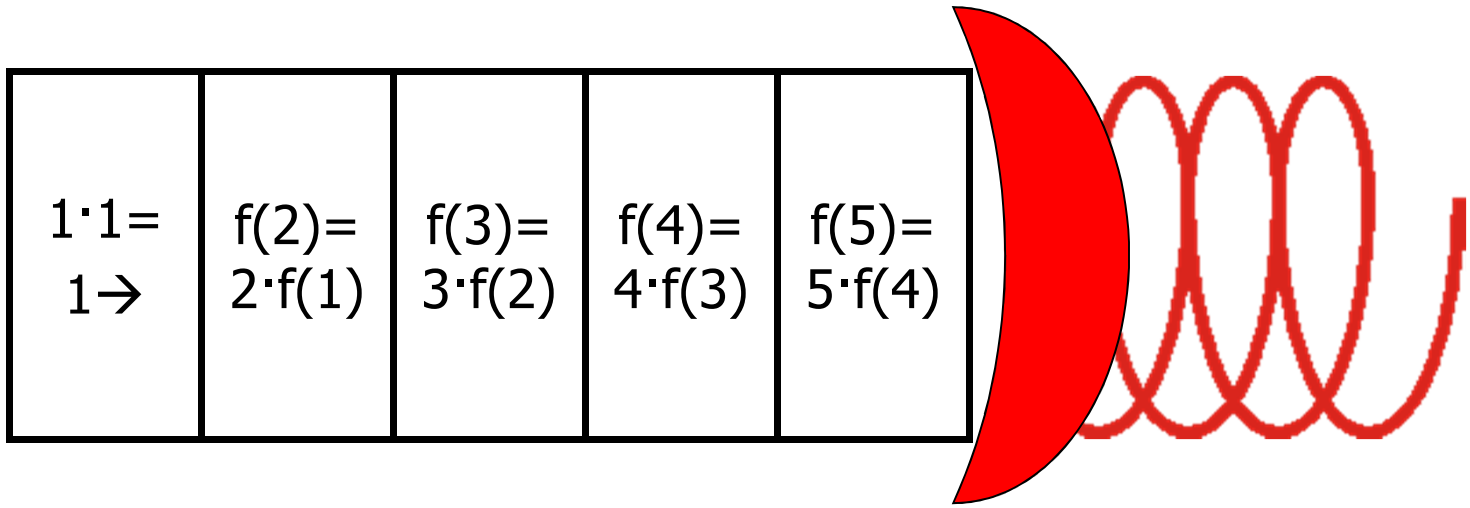
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



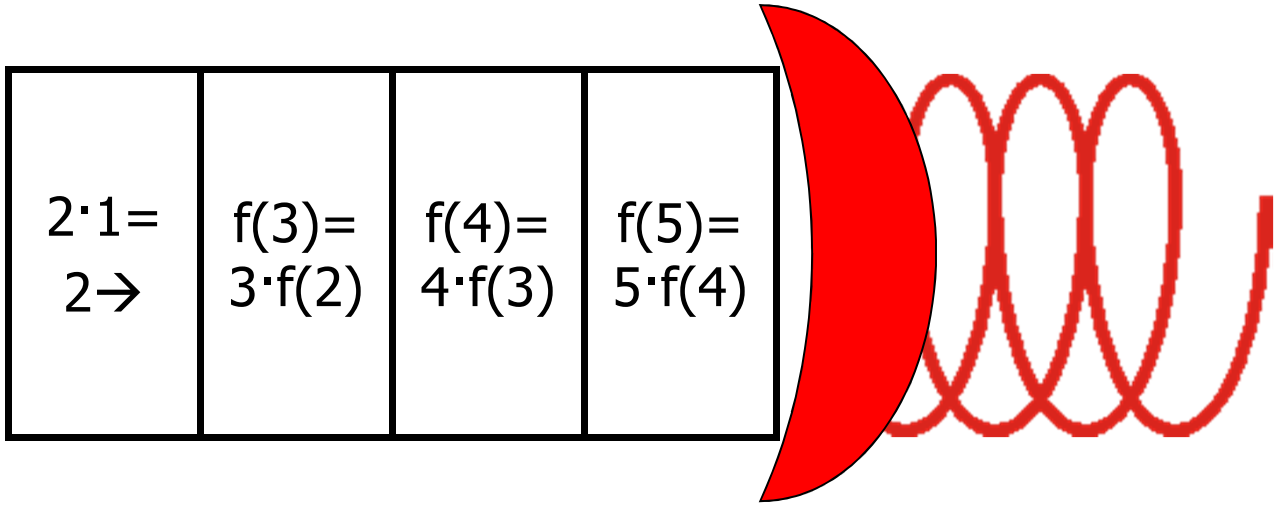
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



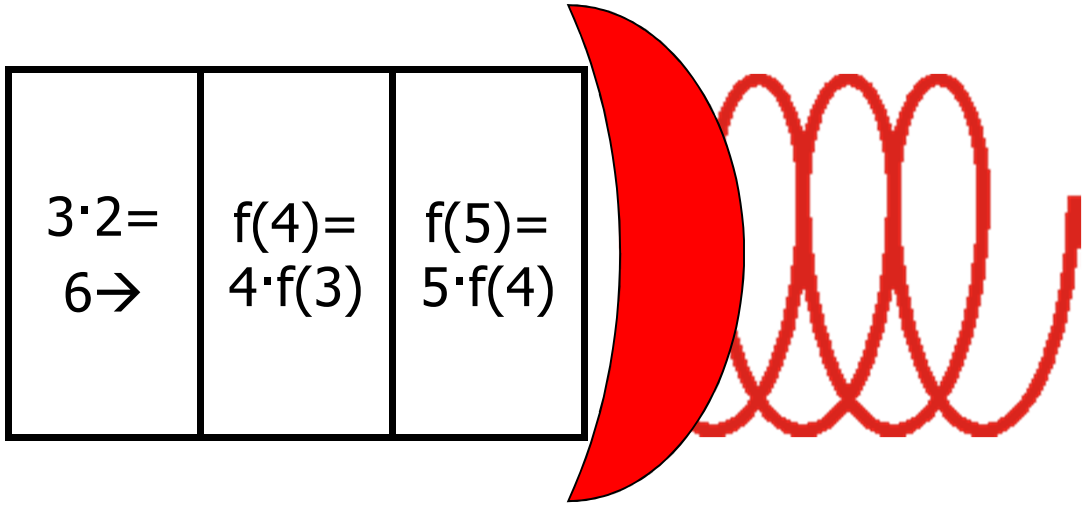
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



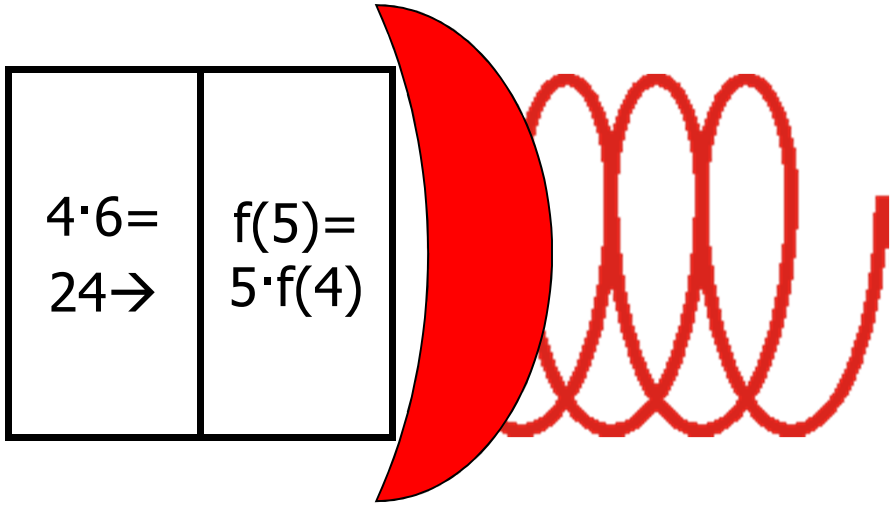
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



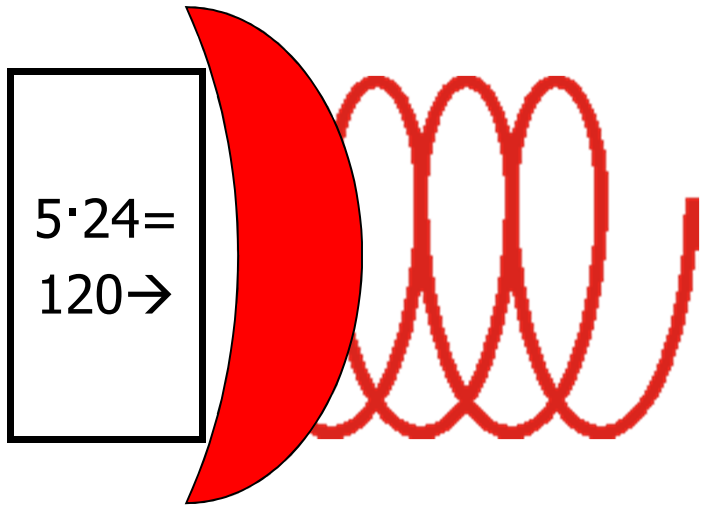
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



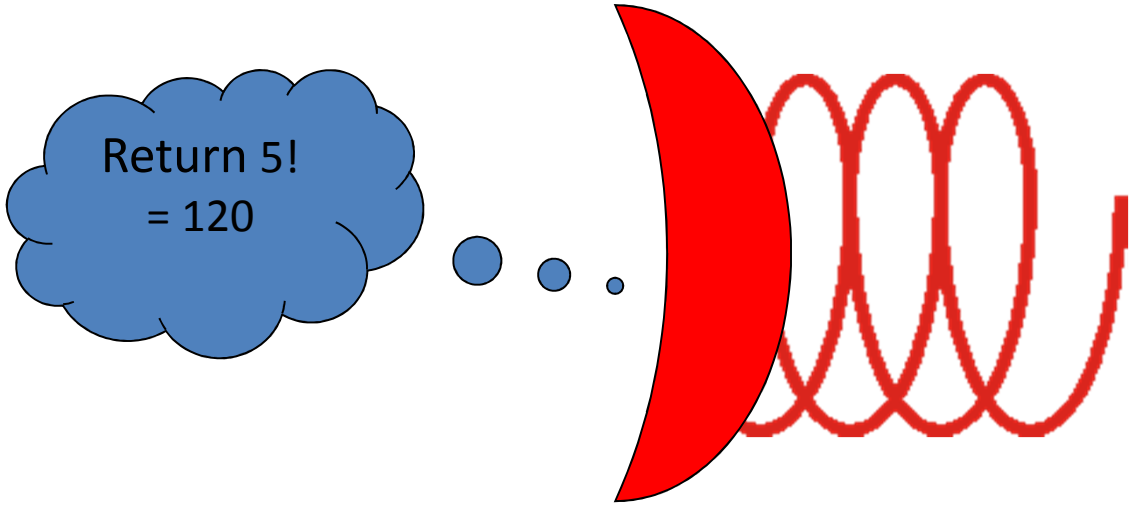
Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



Algoritmos Recursivos e Pilhas

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



De CFG's para Autômatos com Pilha

CFG's definem procedimentos recursivos:

```
boolean derives(strings x, y)
1. if (x==y) return true
2. for(all u⇒y)
       if derives(x,u) return true
3. return false
```

EX: $S \rightarrow \# \mid aSa \mid bSb$

De CFG's para Máquinas de Pilha

Por princípios gerais, *qualquer* computação recursiva pode ser executada usando-se uma pilha. Isso pode ser feito em uma versão simplificada de máquina com pilha de registro de ativação, chamada Autômato de Pilha ("**P**ush**d**own **A**utomaton"– **PDA**)

Q: Qual é a linguagem gerada por

$S \rightarrow \# \mid aSa \mid bSb$?

De CFG's para Máquinas de Pilha

R: Palíndromos em {*a,b,*#}* contendo exatamente um símbolo #.

Q: Usando uma pilha, como podemos reconhecer tais strings?

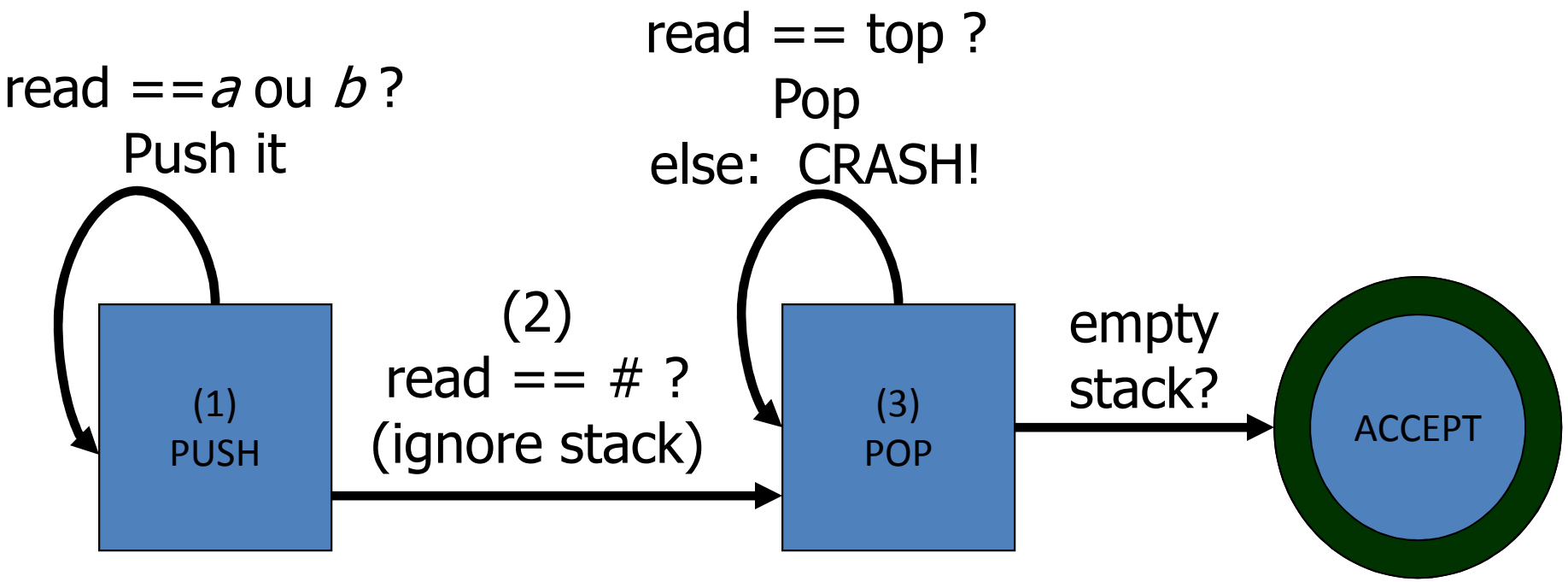
De CFG's para Máquinas de Pilha

A: Usamos um processo com três fases:

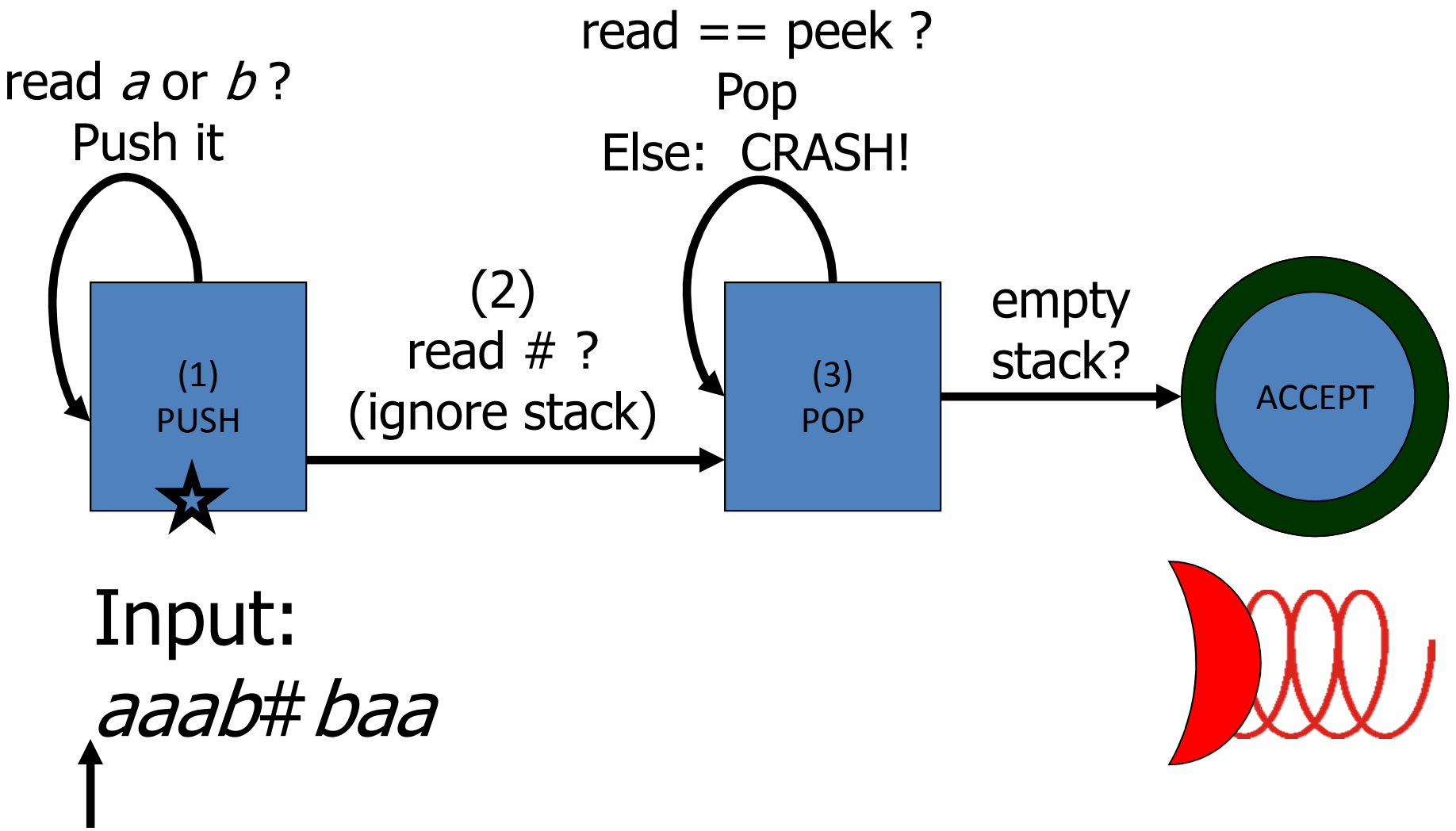
1. **modo Push :** Antes de ler "#", emplihe qualquer símbolo lido.
2. Ao ler "#" troque de modo de operação.
3. **modo Pop :** Leia os símbolos restantes garantindo que cada novo símbolo lido é idêntico ao símbolo desempilhado.

Aceite se for capaz de concluir com a pilha vazia. Caso contrário, rejeite; e rejeite se não for possível desempilhar em algum caso.

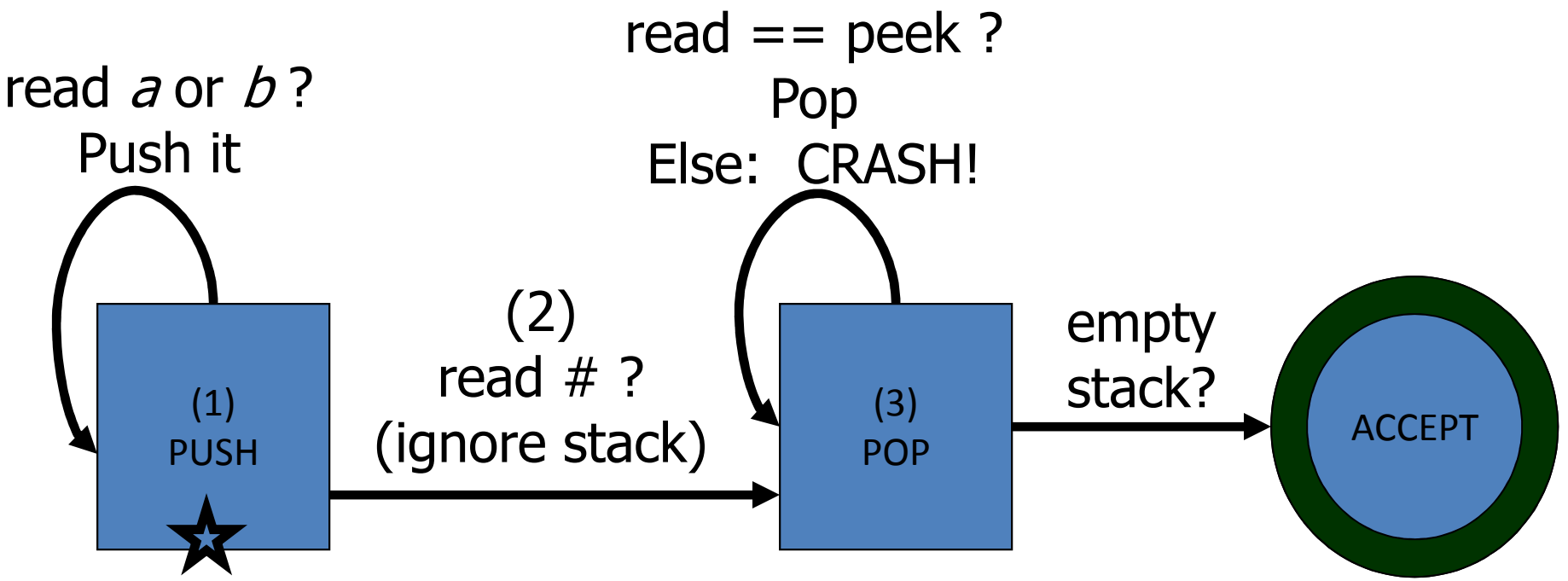
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

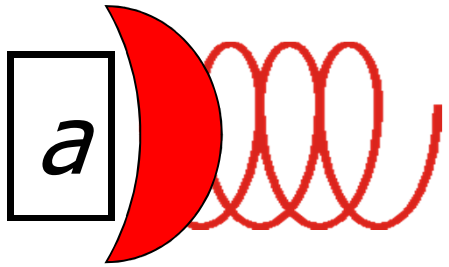


De CFG's para Máquinas de Pilha

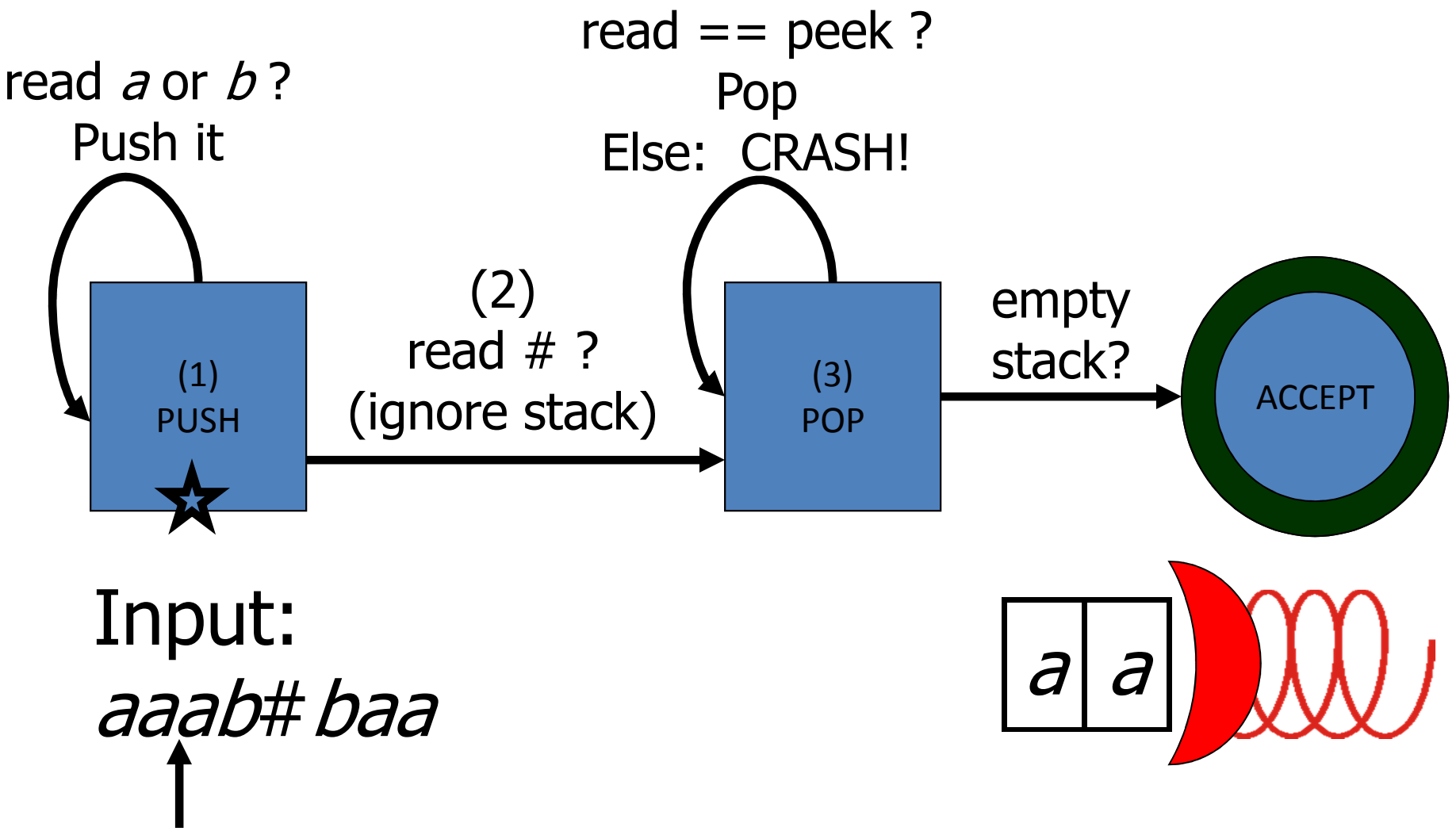


Input:

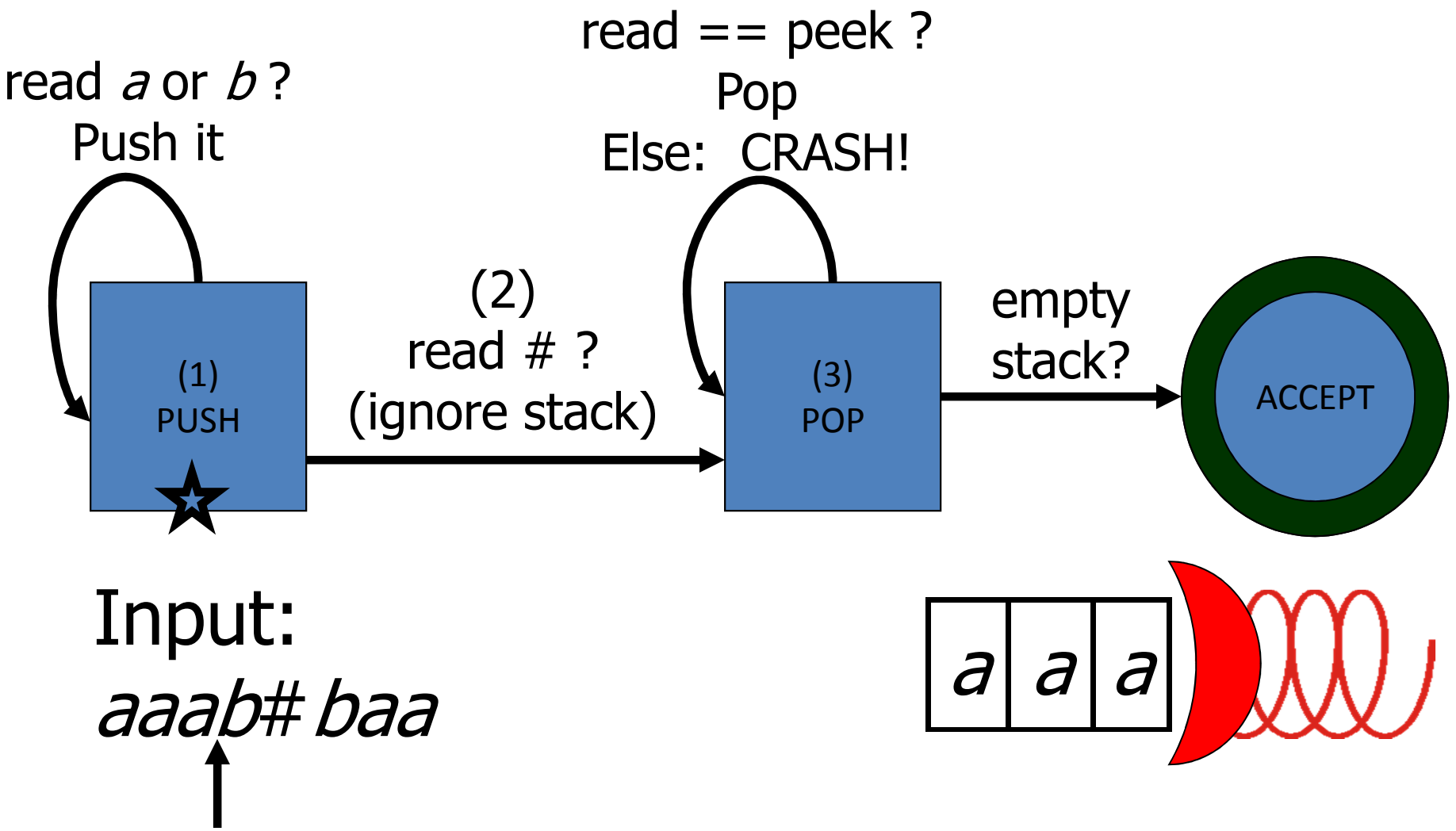
aaab# baa



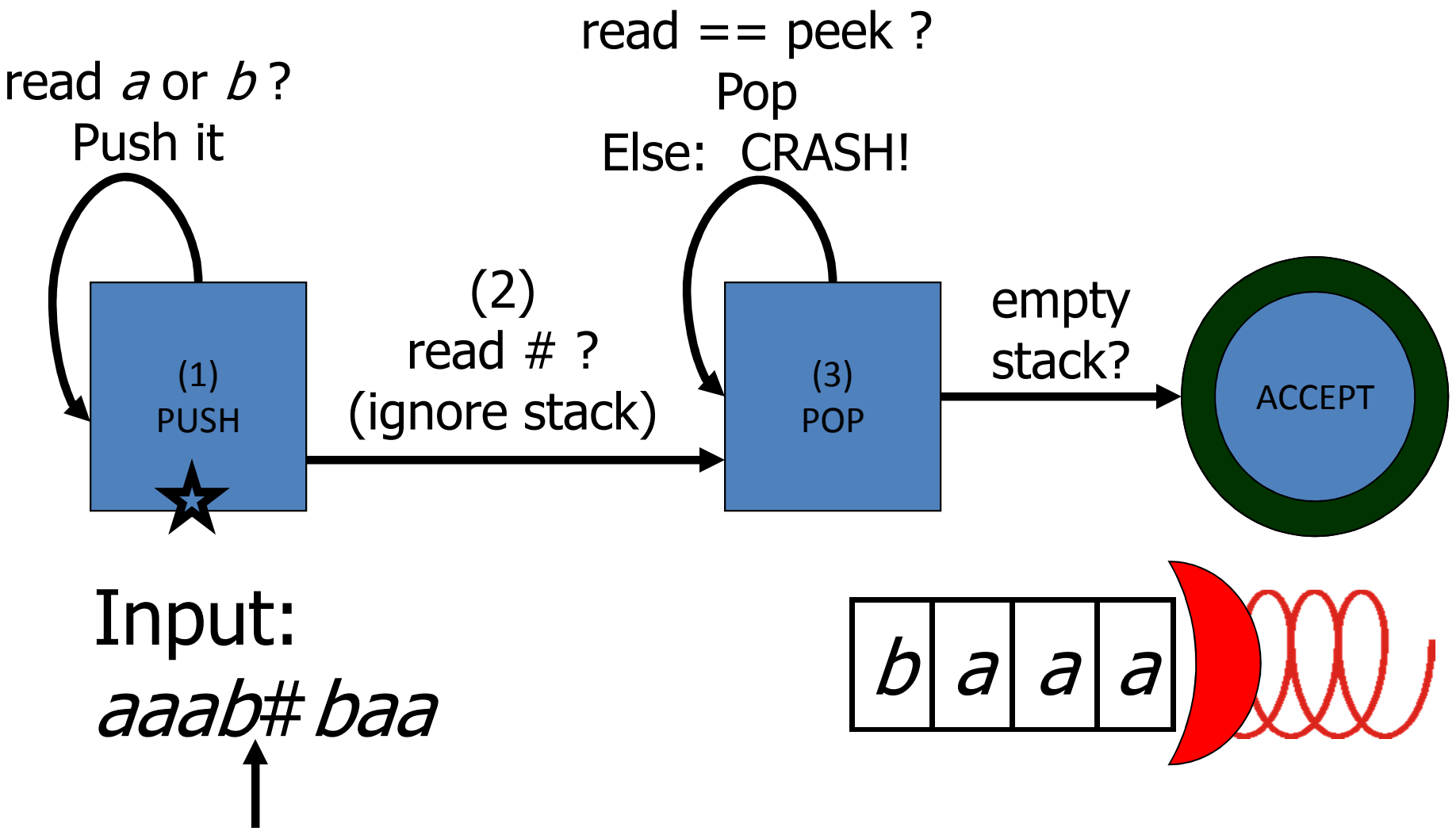
De CFG's para Máquinas de Pilha



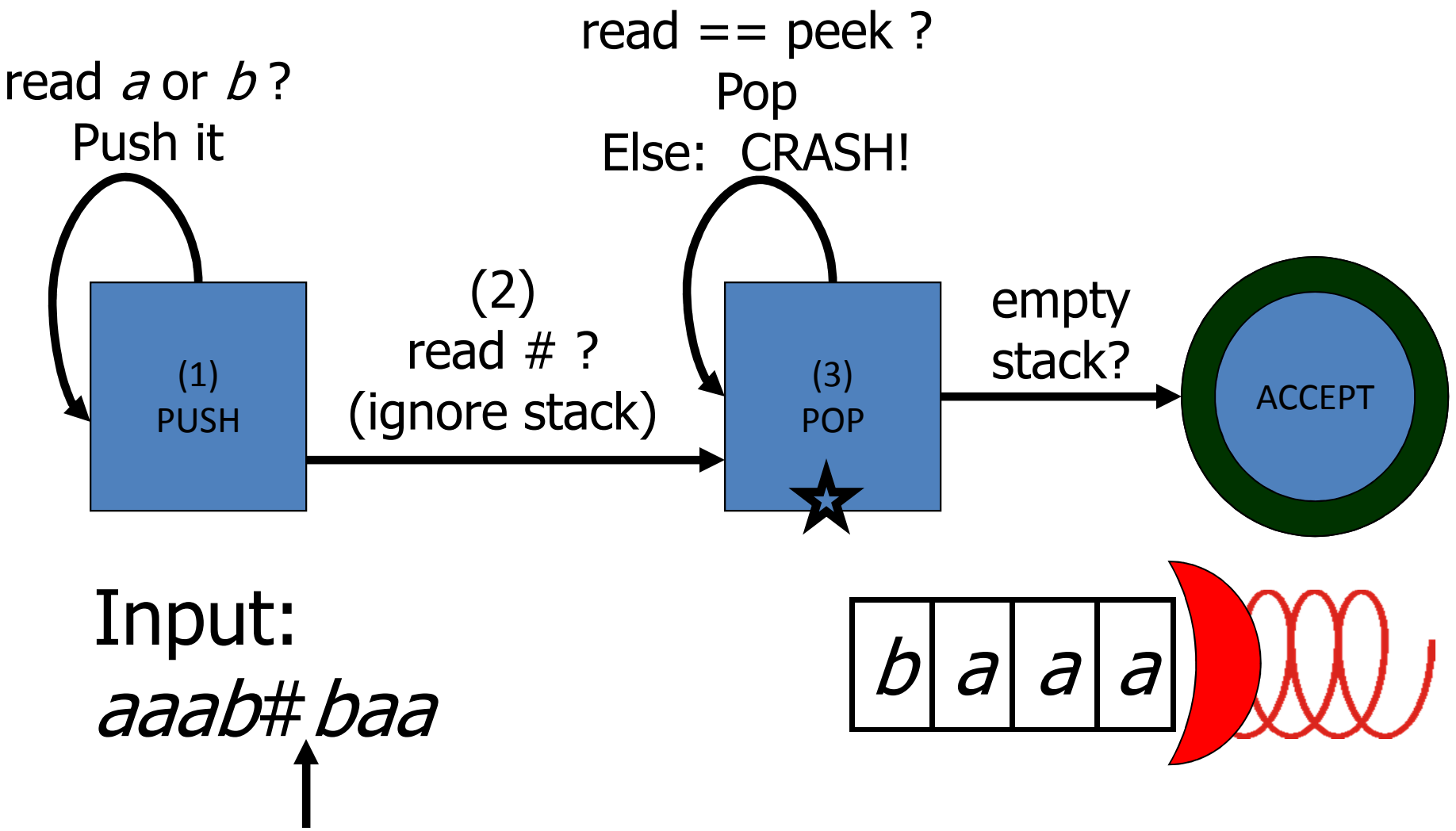
De CFG's para Máquinas de Pilha



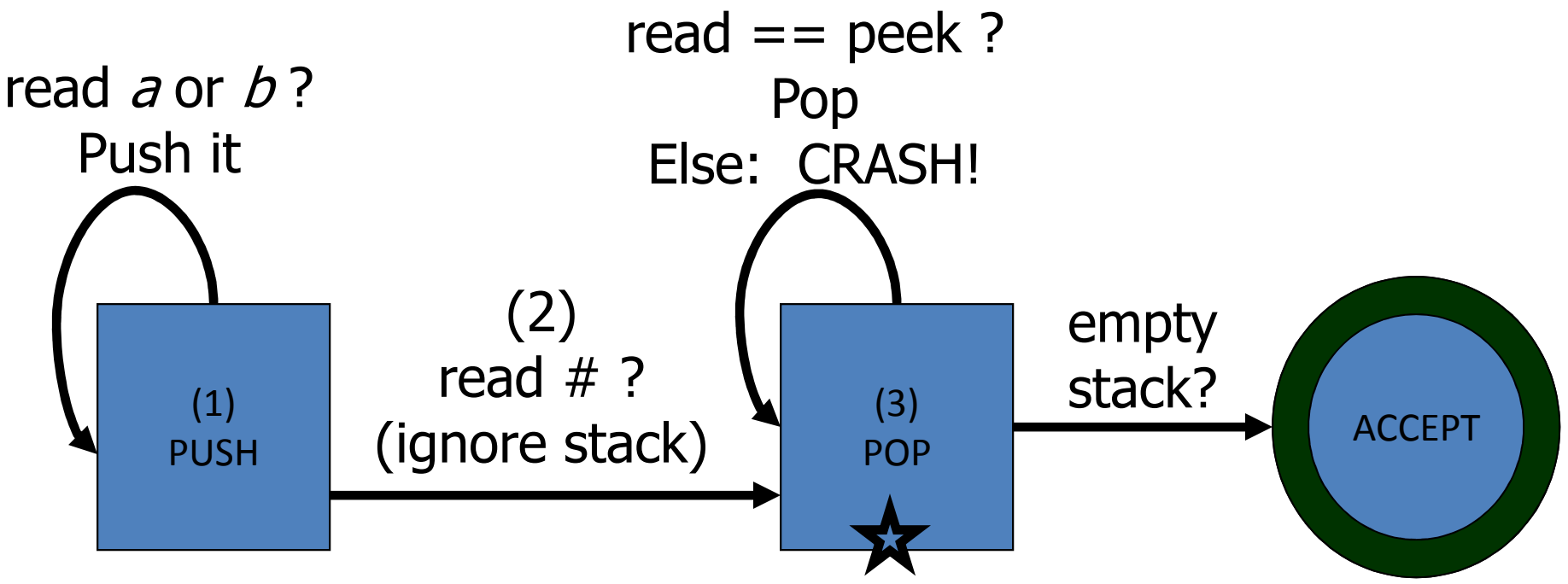
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

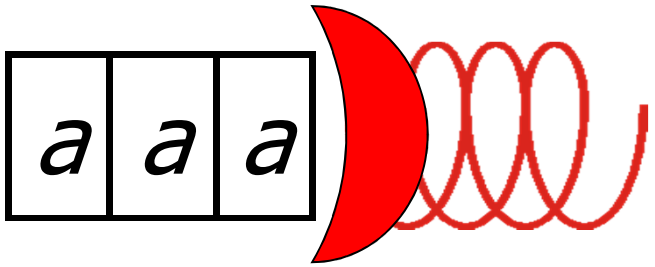


De CFG's para Máquinas de Pilha

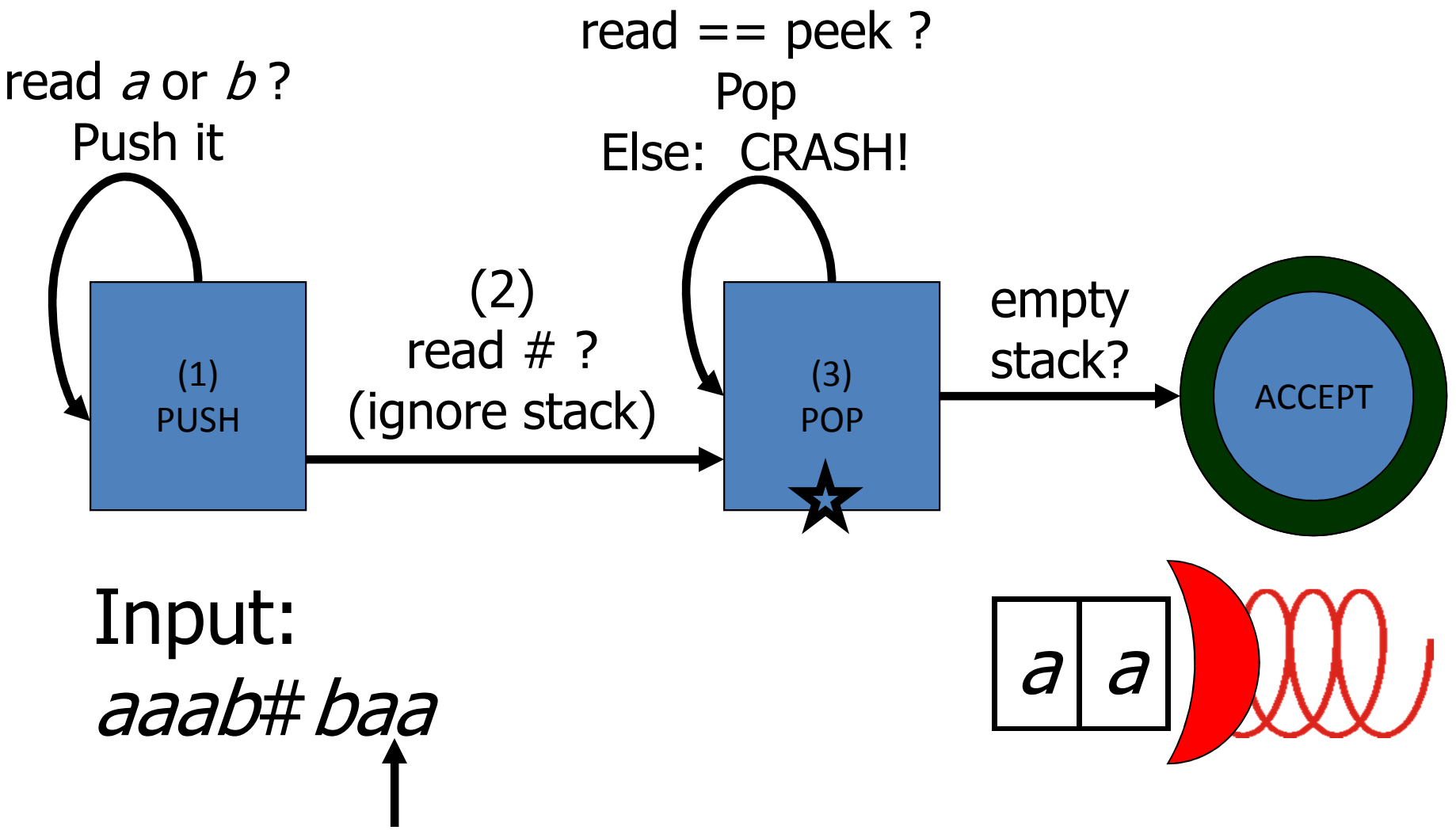


Input:

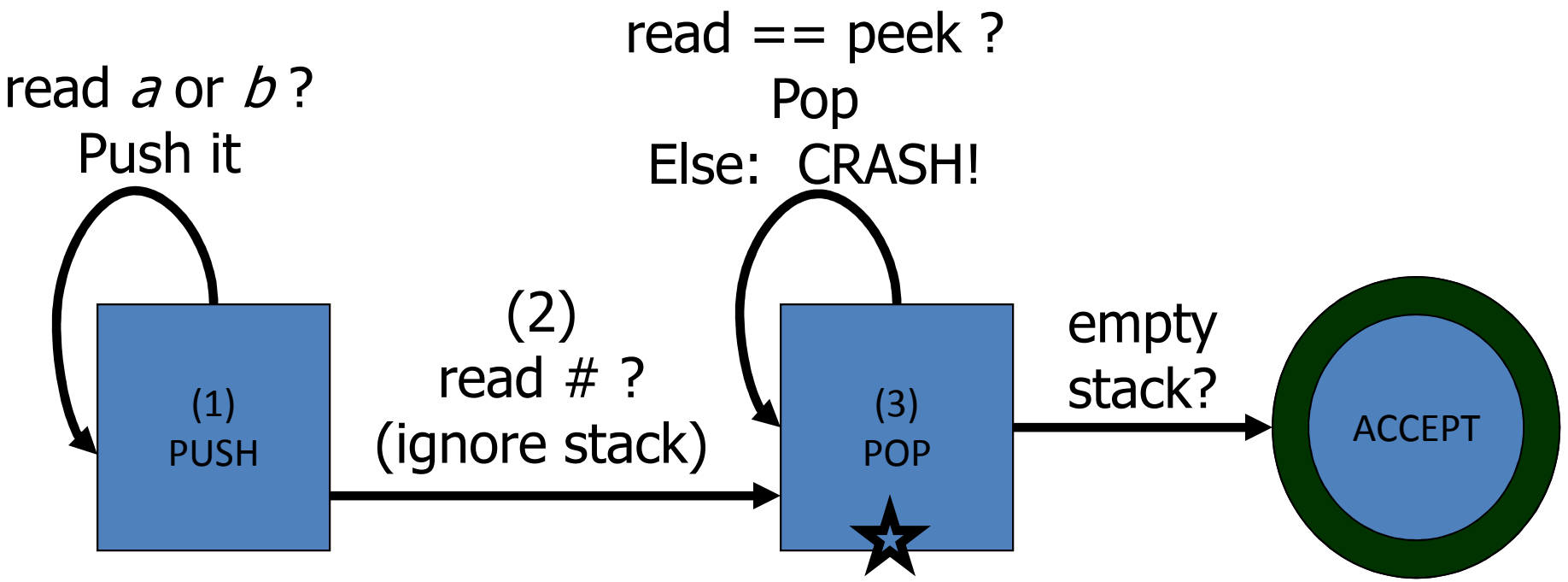
aaab# baa



De CFG's para Máquinas de Pilha

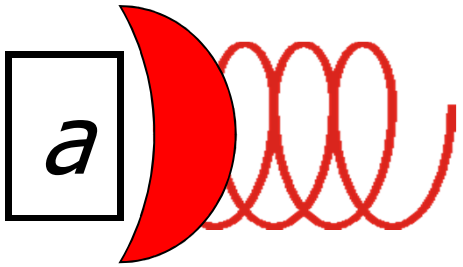


De CFG's para Máquinas de Pilha



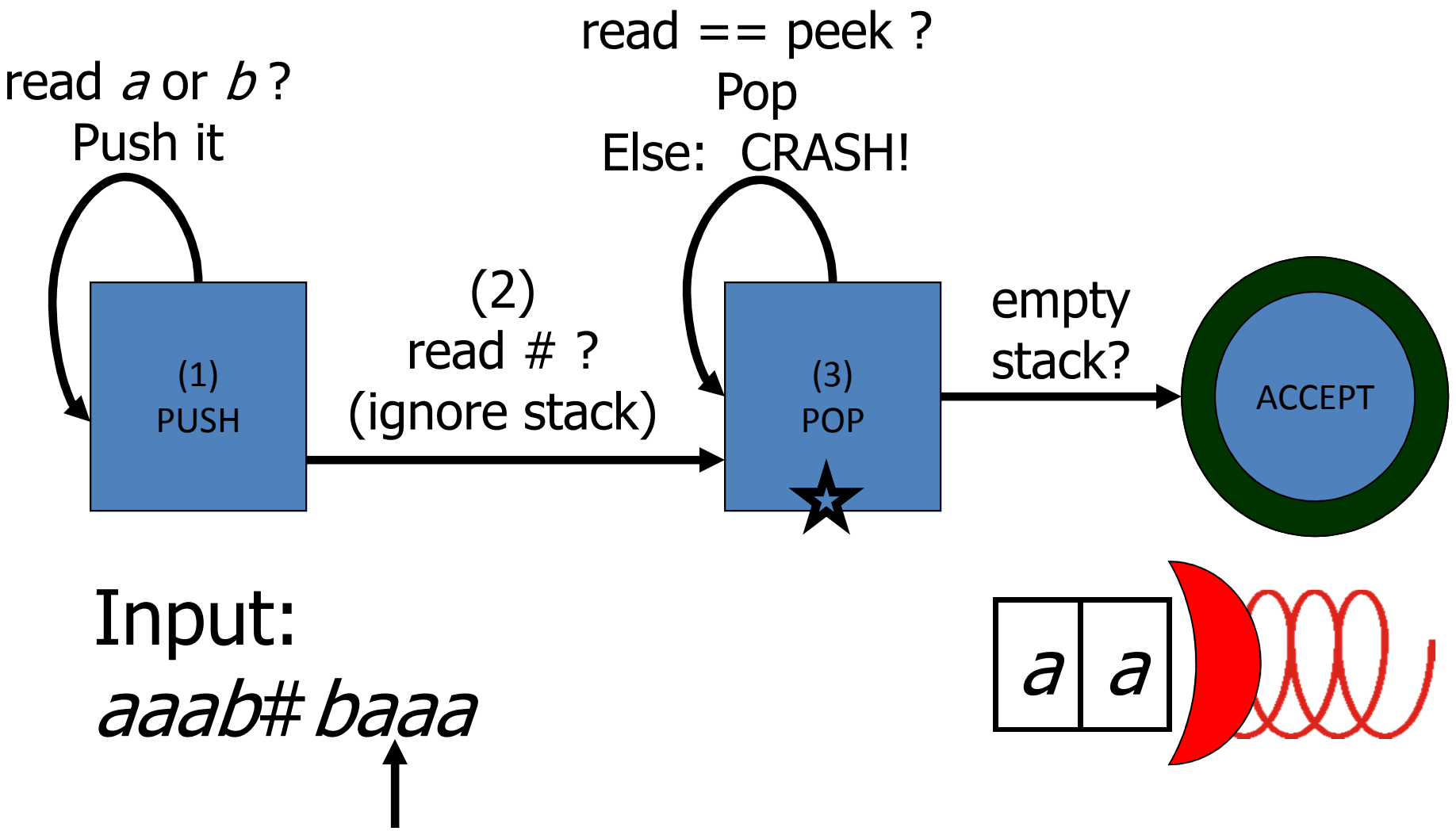
Input:

aaab# baa

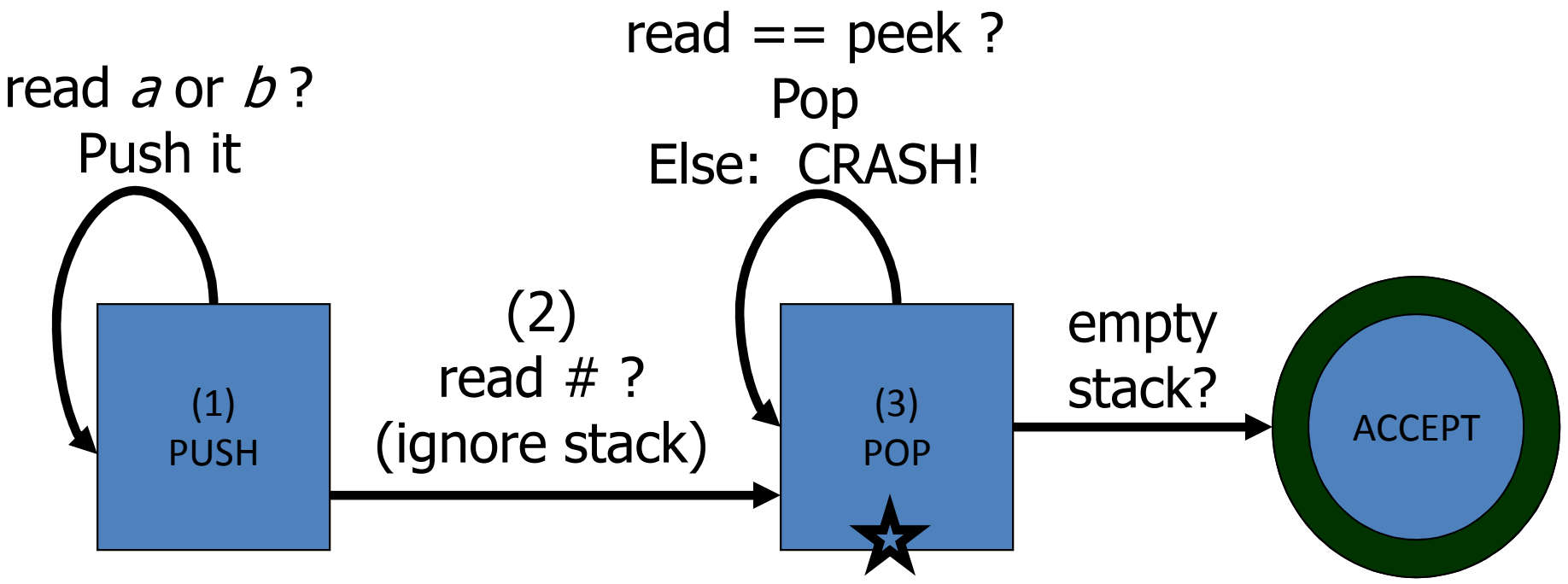


REJECT (nonempty stack)

De CFG's para Máquinas de Pilha

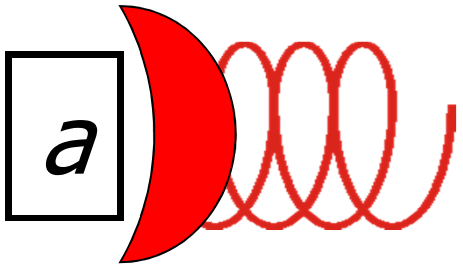


De CFG's para Máquinas de Pilha

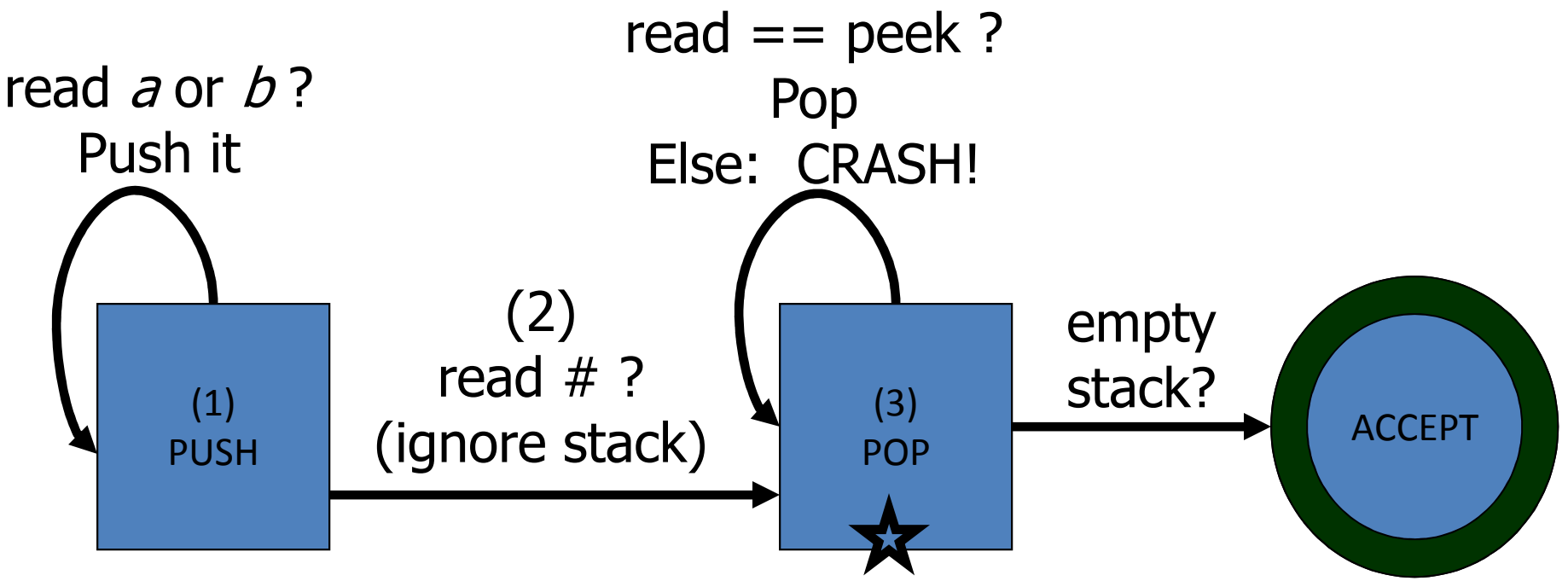


Input:

aaab# baaa



De CFG's para Máquinas de Pilha

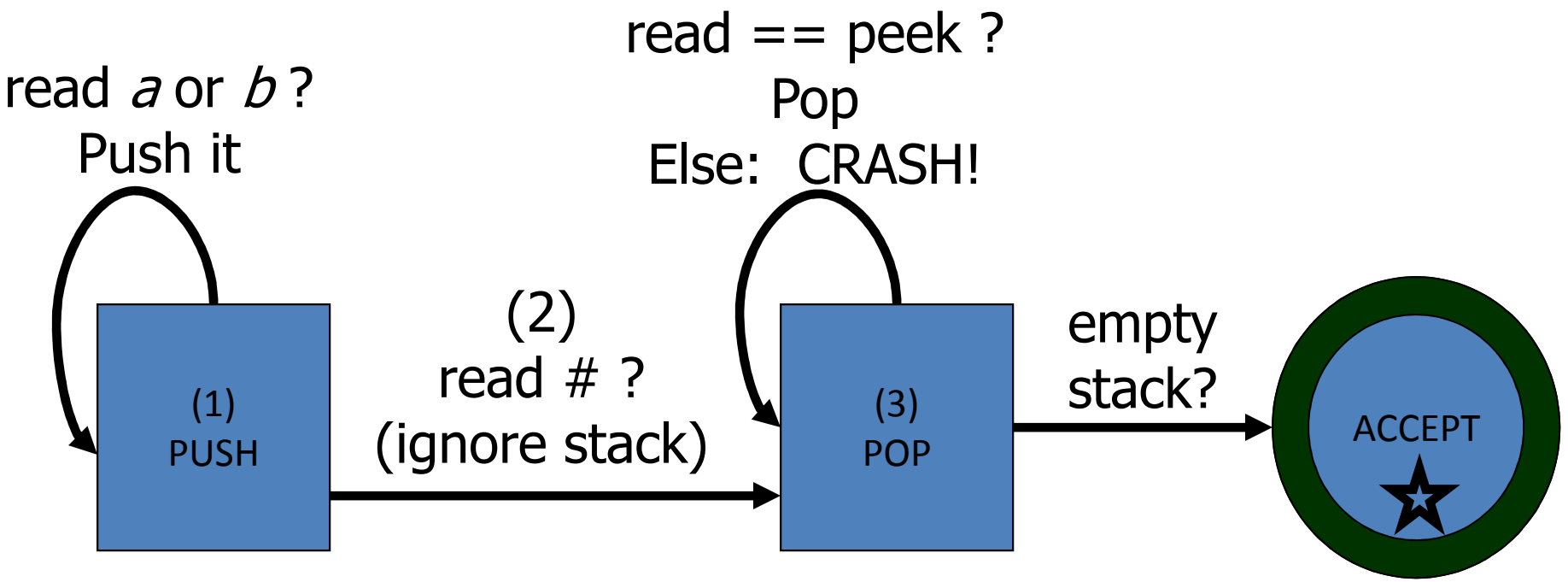


Input:

aaab# baaa

Pause input

De CFG's para Máquinas de Pilha

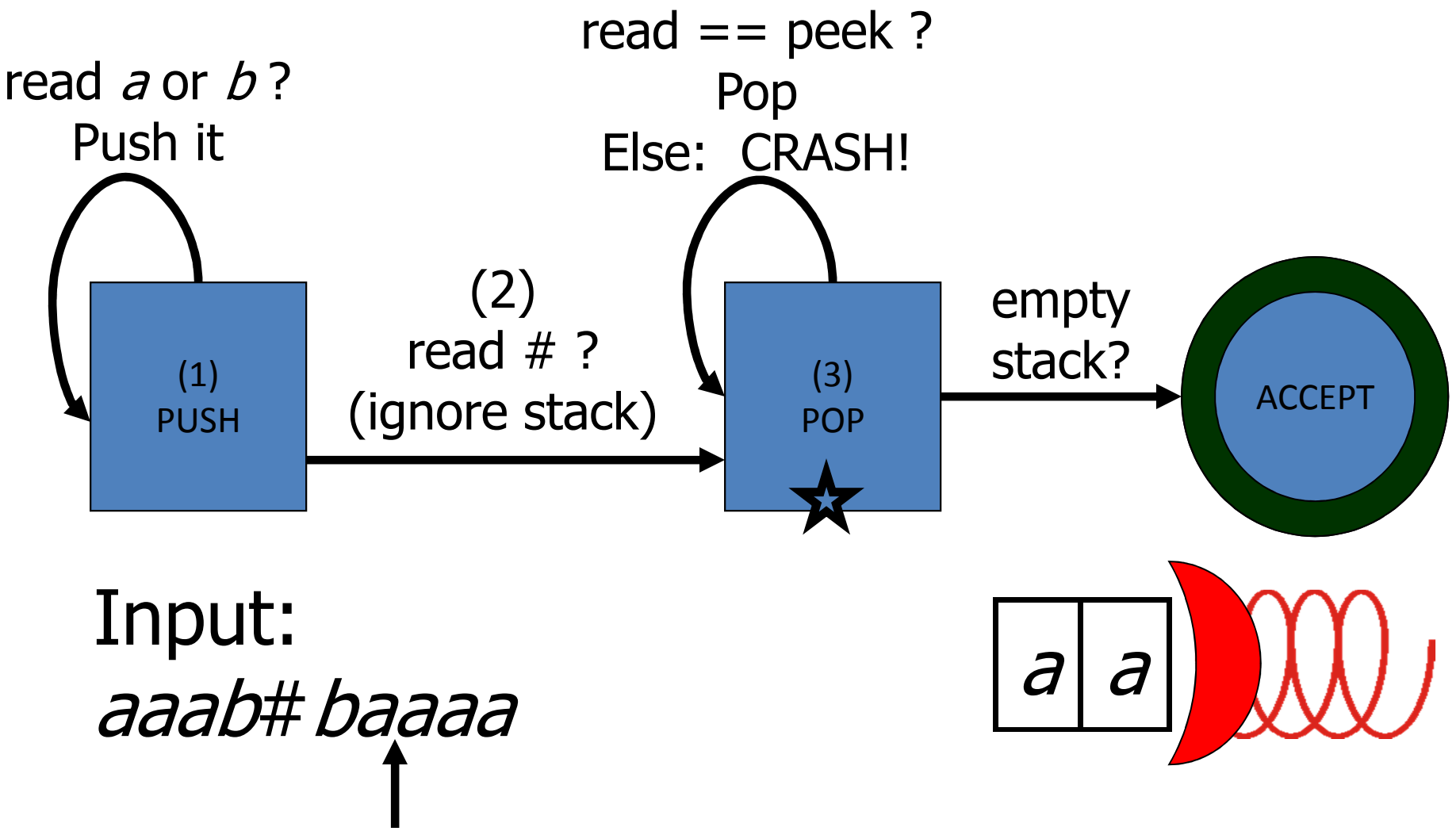


Input:

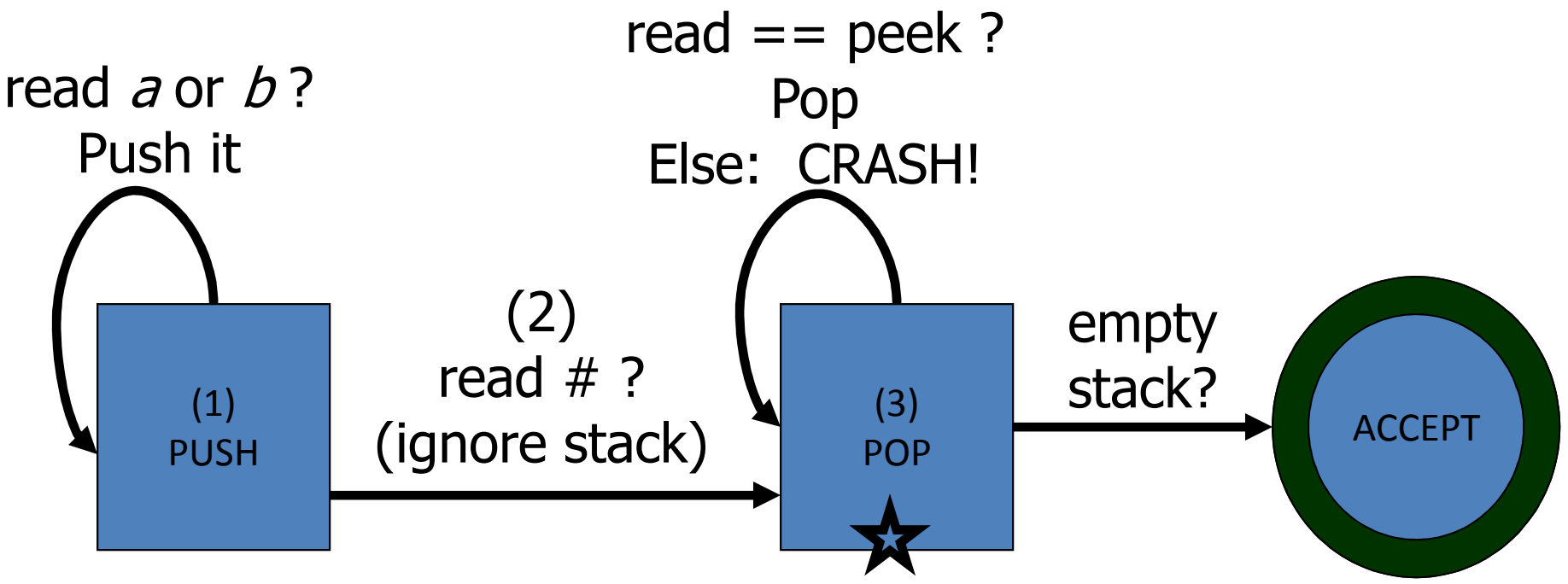
aaab# baaa

ACCEPT

De CFG's para Máquinas de Pilha

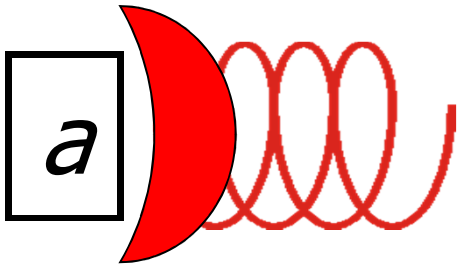


De CFG's para Máquinas de Pilha

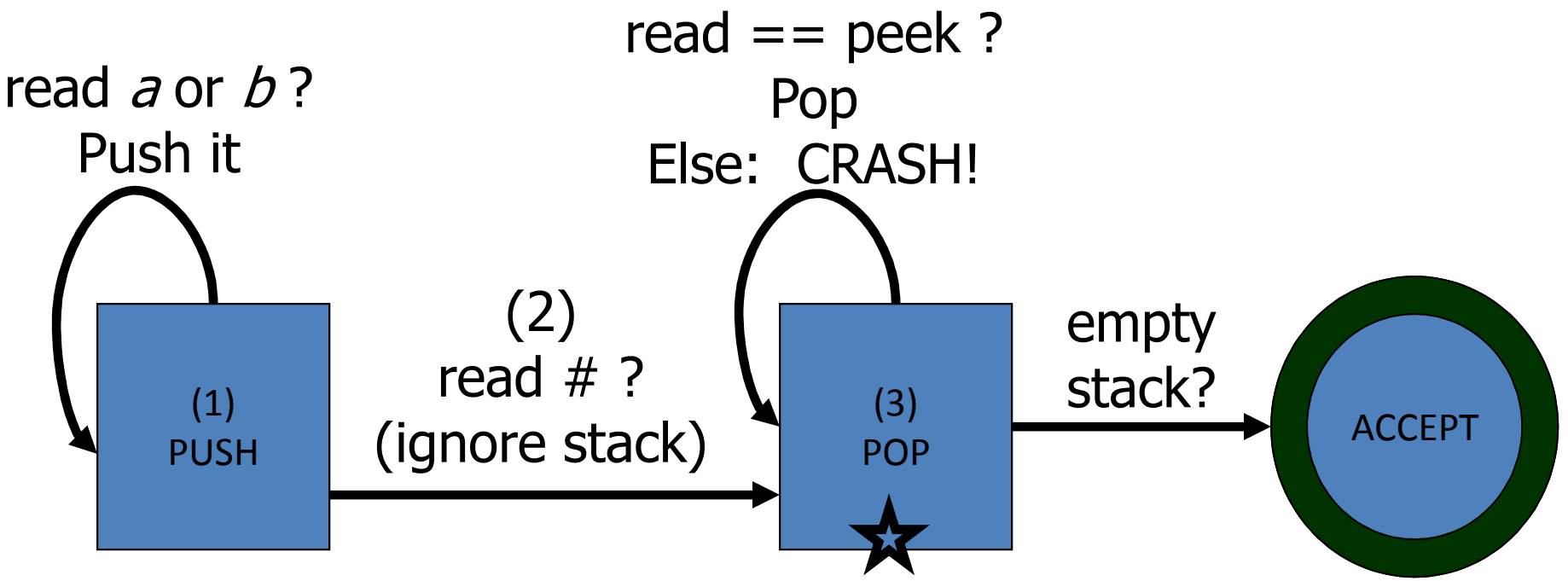


Input:

aaab# baaaa



De CFG's para Máquinas de Pilha

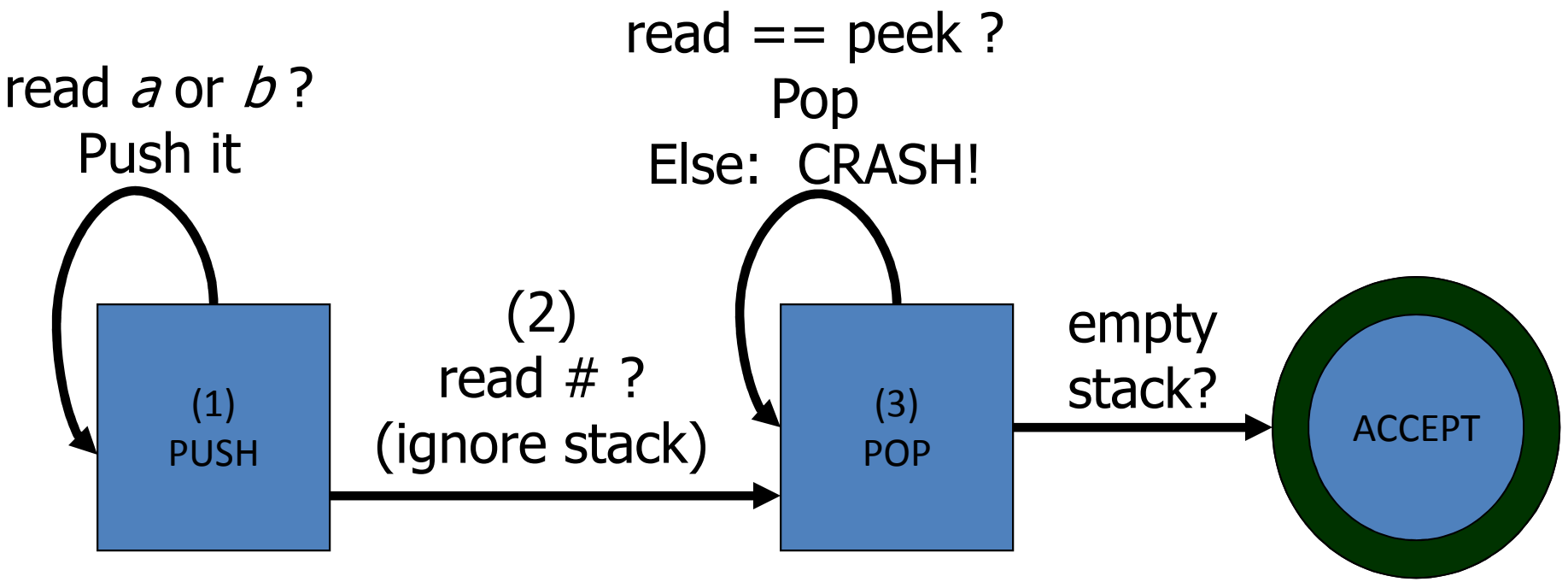


Input:

aaab# baaaa

Pause input

De CFG's para Máquinas de Pilha



Input:

aaab# baaaa

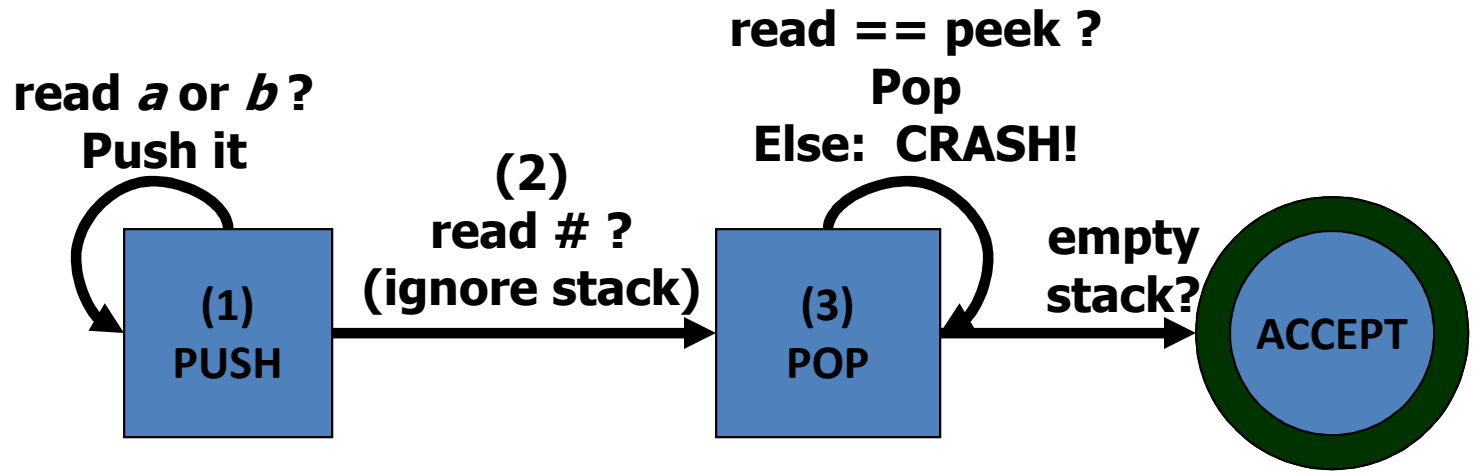
CRASH

PDA's à la Sipser

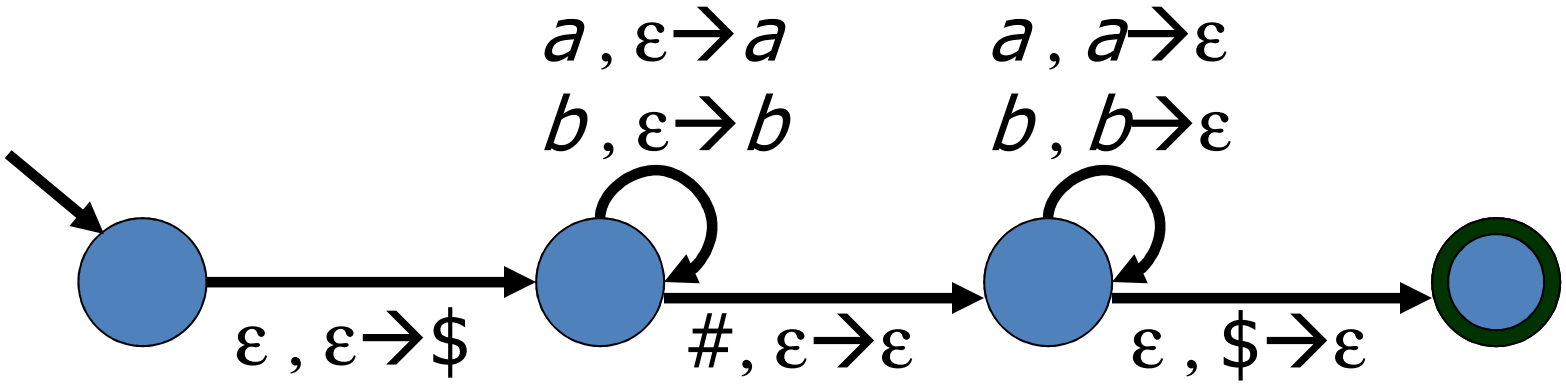
Para facilitar a análise, máquinas de pilha teóricas têm um conjunto restrito de operações. Cada livro-author tem sua própria versão. PDAs descritos em Sipser são assim:

- Push/Pop agrupados em única operação: *substituir o símbolo no topo da pilha*
- Nenhum teste intrínsico de pilha vazia
- Epsilon é usado para aumentar a funcionalidade: máquinas *não deterministas.*

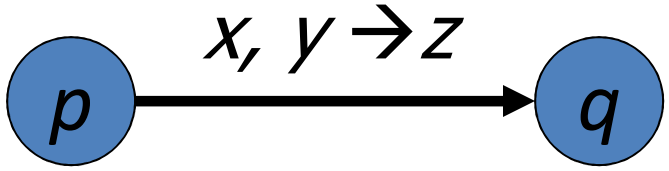
Versão Sipser's



Torna-se:



Versão Sipser's



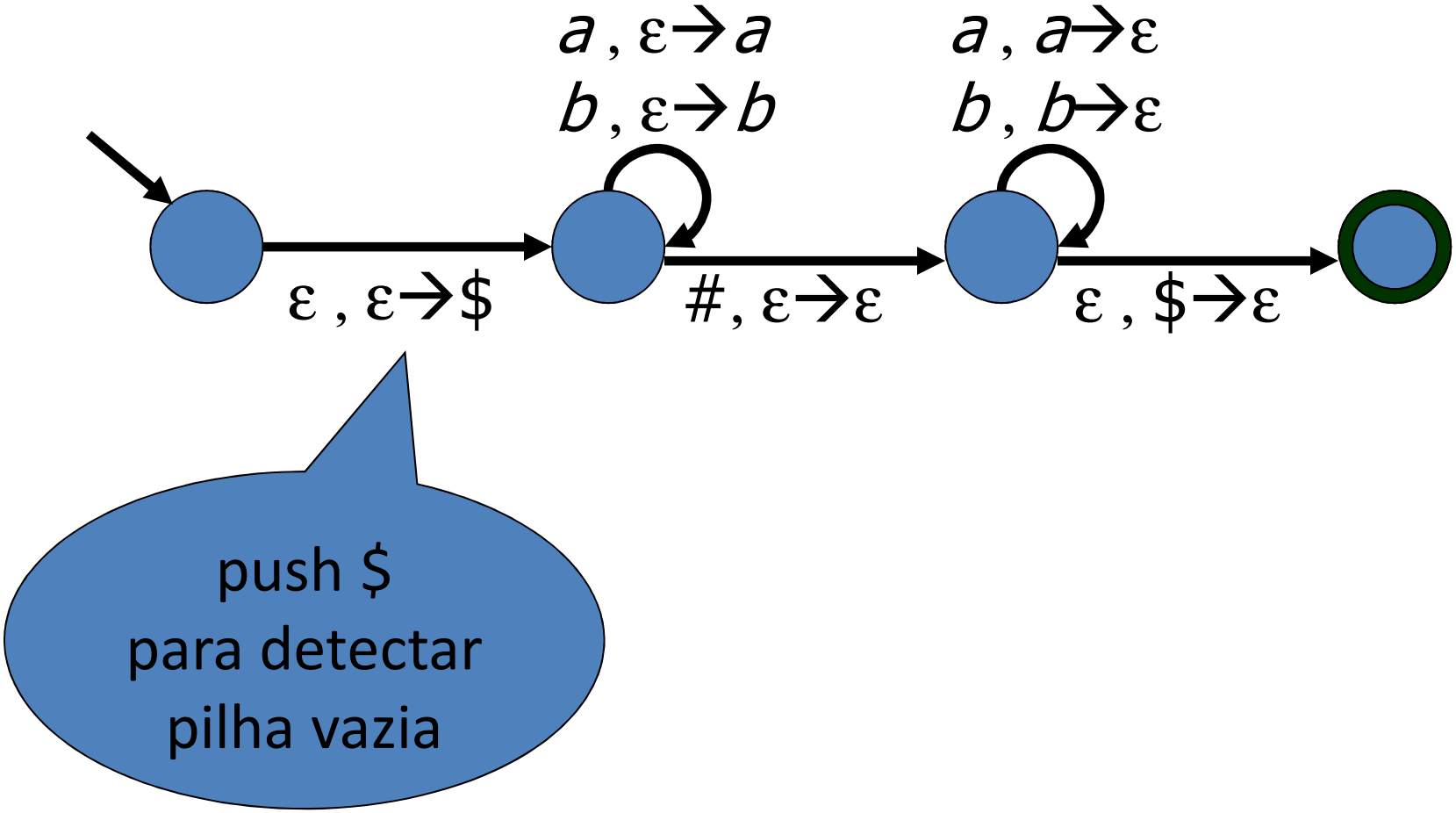
Significado da convenção do rótulo:

Se está no estado *p* **e** o próximo símbolo é *x* **e** o topo da pilha é *y*,

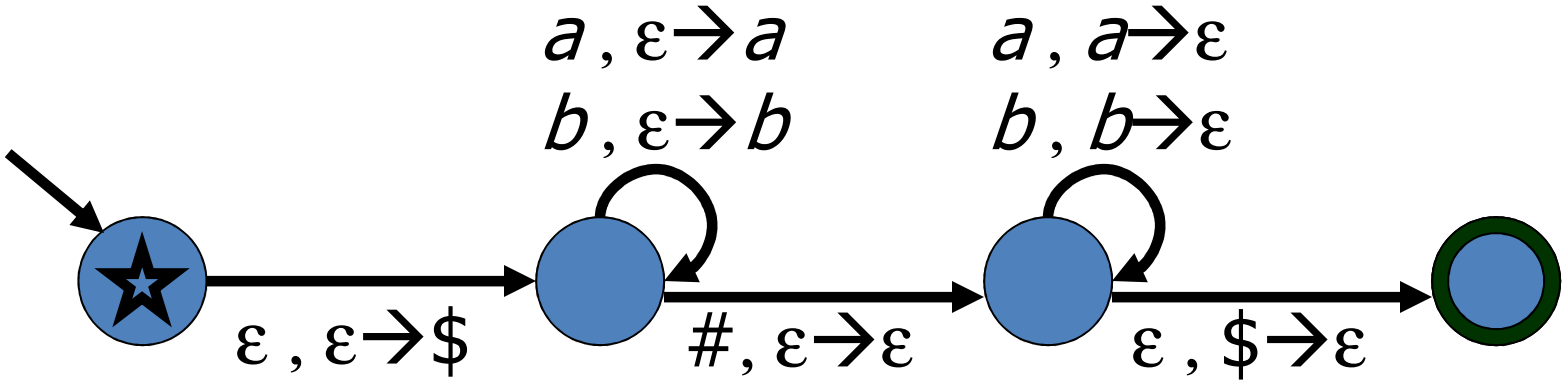
então vá para *q* **e** substitua *y* por *z* na pilha.

- *x* = ε: ignore a entrada, não leia
- *y* = ε: ignore o topo da pilha e empilhe *z*
- *z* = ε: desempilhe *y*

Versão Sipser's

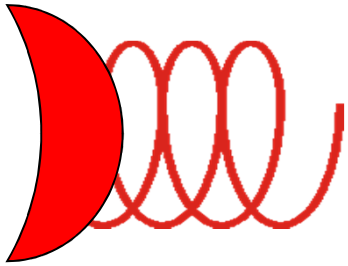


Versão Sipser's

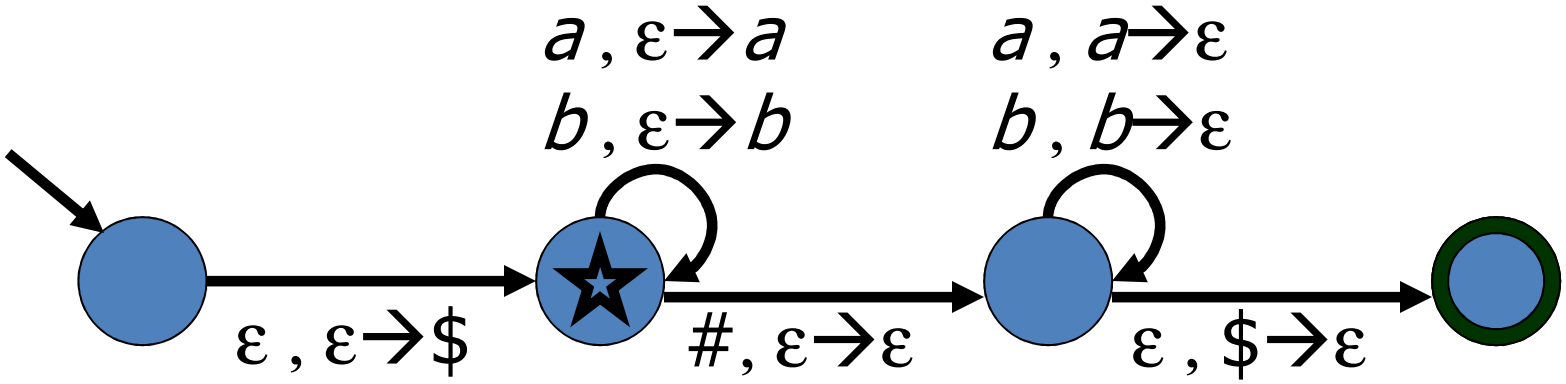


Input:

aaab# baaa

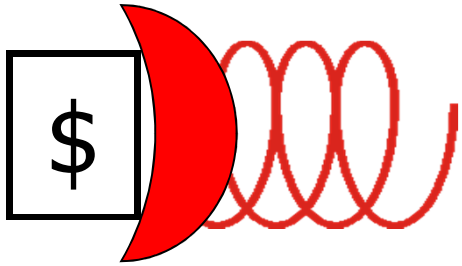


Versão Sipser's

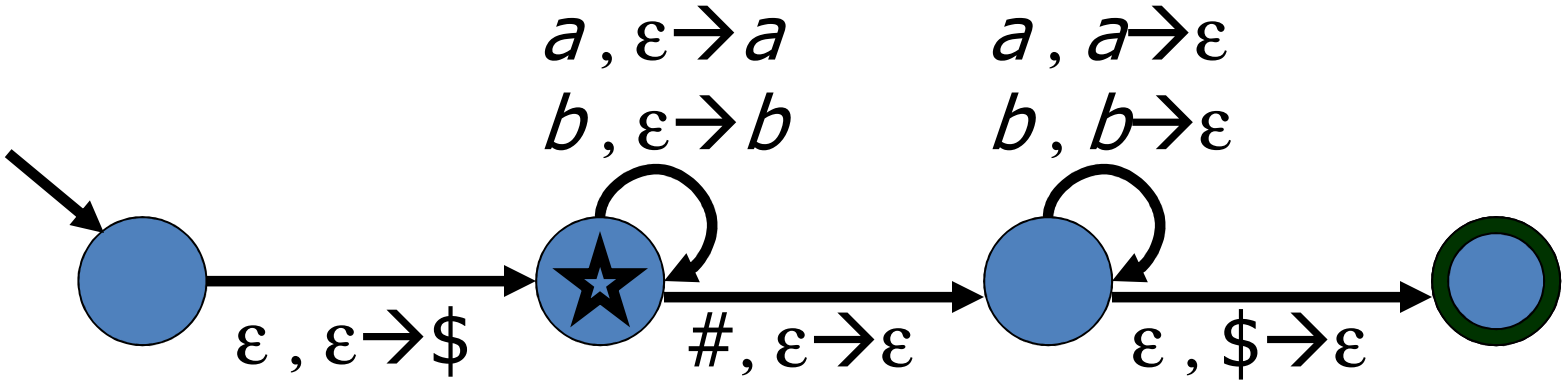


Input:

aaab# baaa

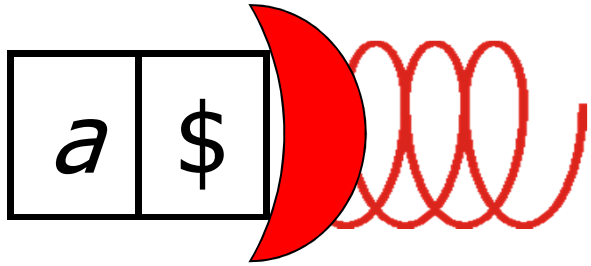


Versão Sipser's

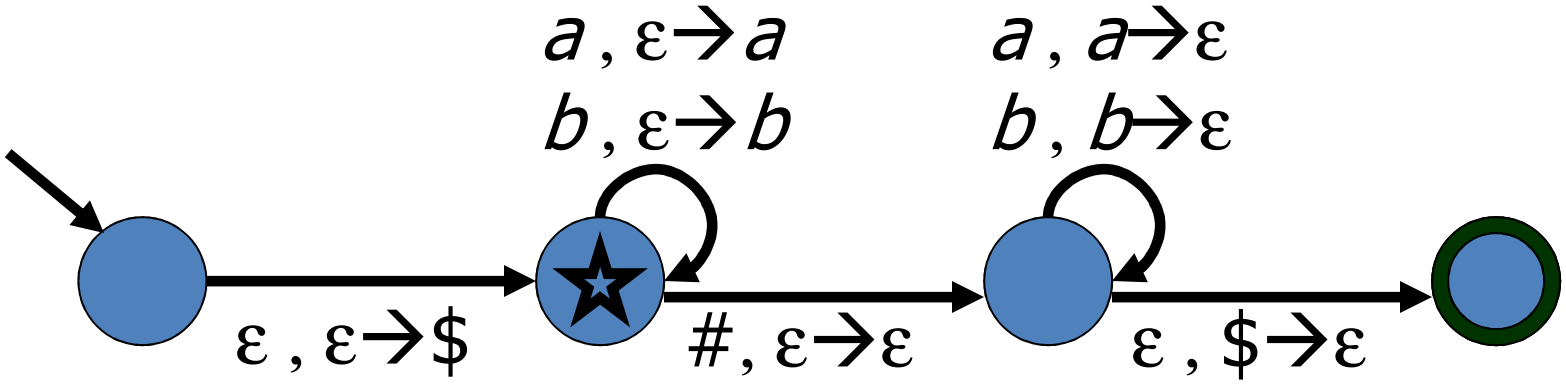


Input:

aaab# baaa

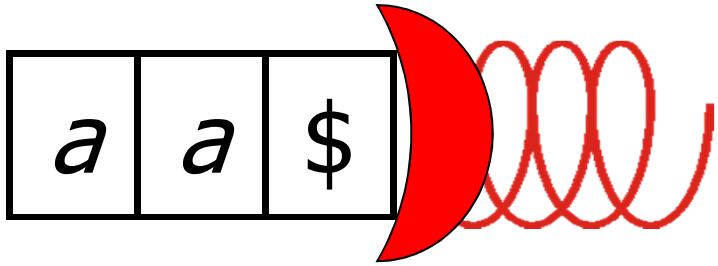


Versão Sipser's

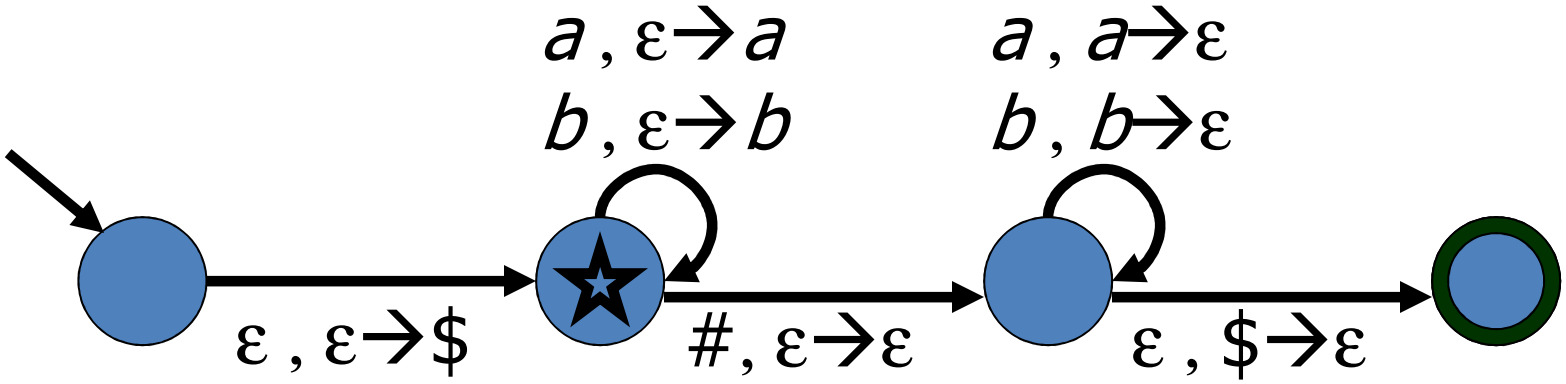


Input:

aaab# baaa

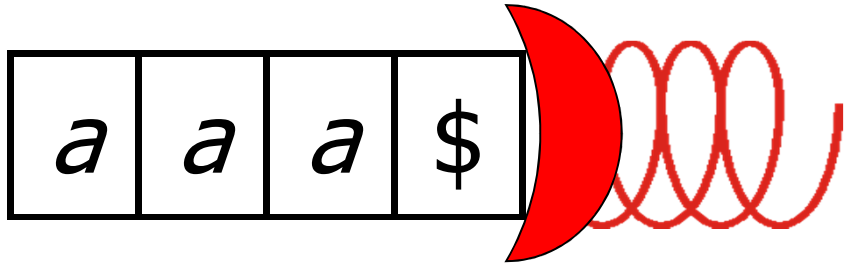


Versão Sipser's

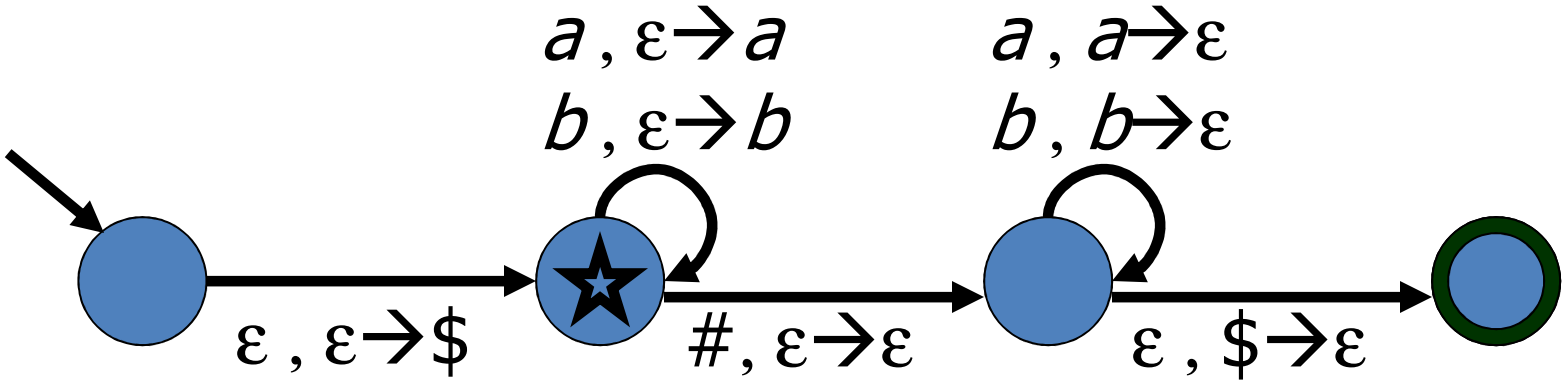


Input:

aaab# baaa

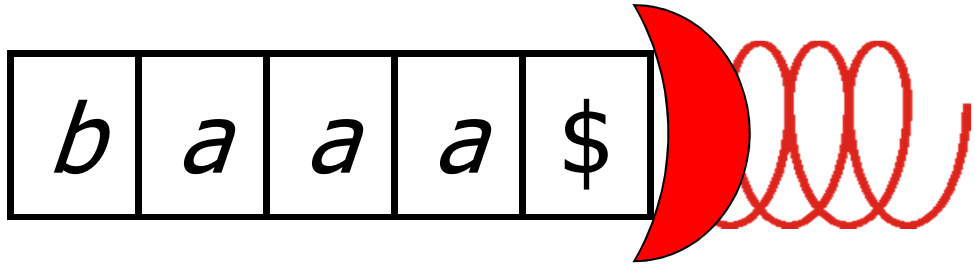


Versão Sipser's

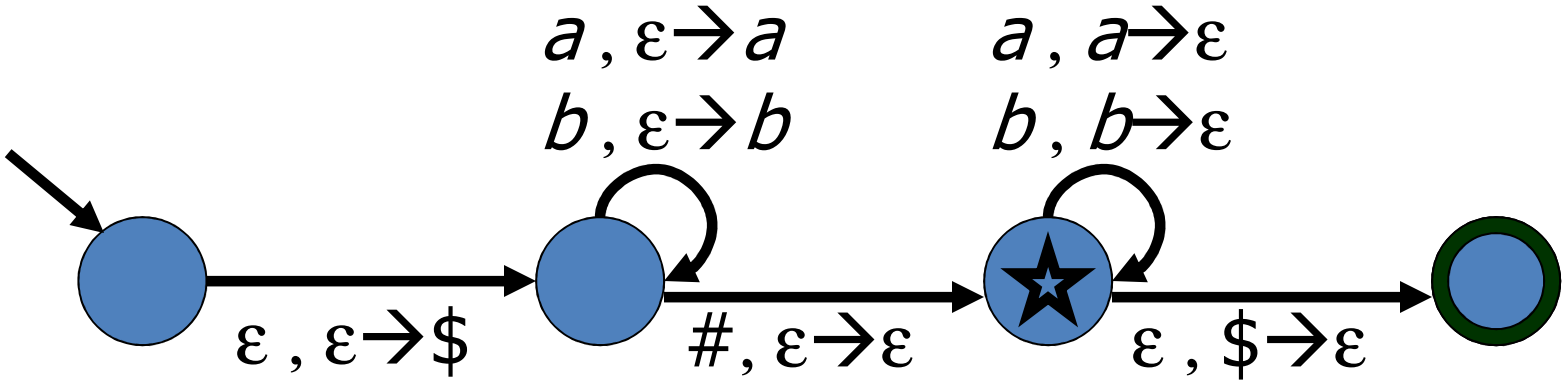


Input:

aaab# baaa

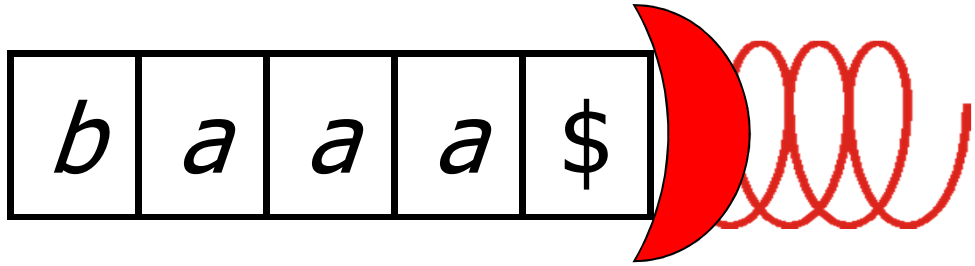


Versão Sipser's

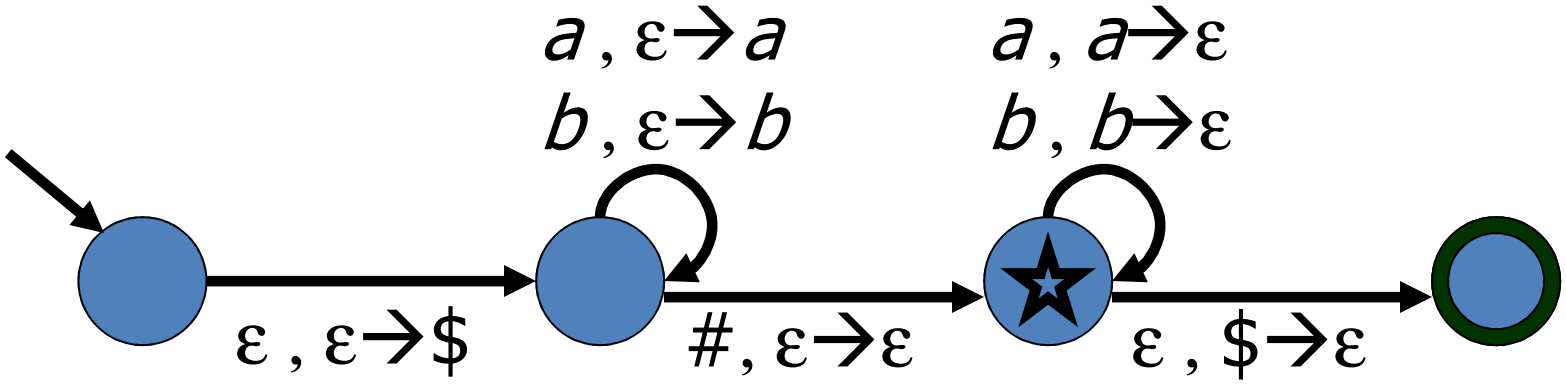


Input:

aaab# baaa

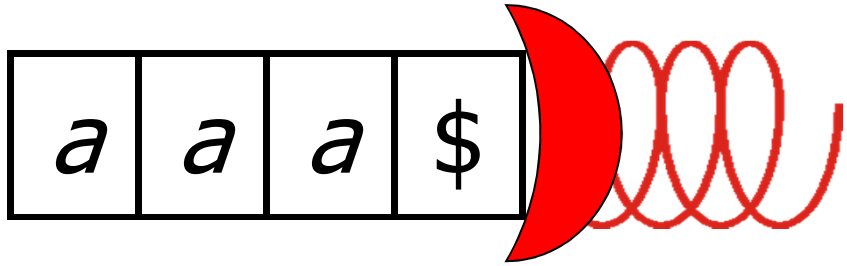


Versão Sipser's

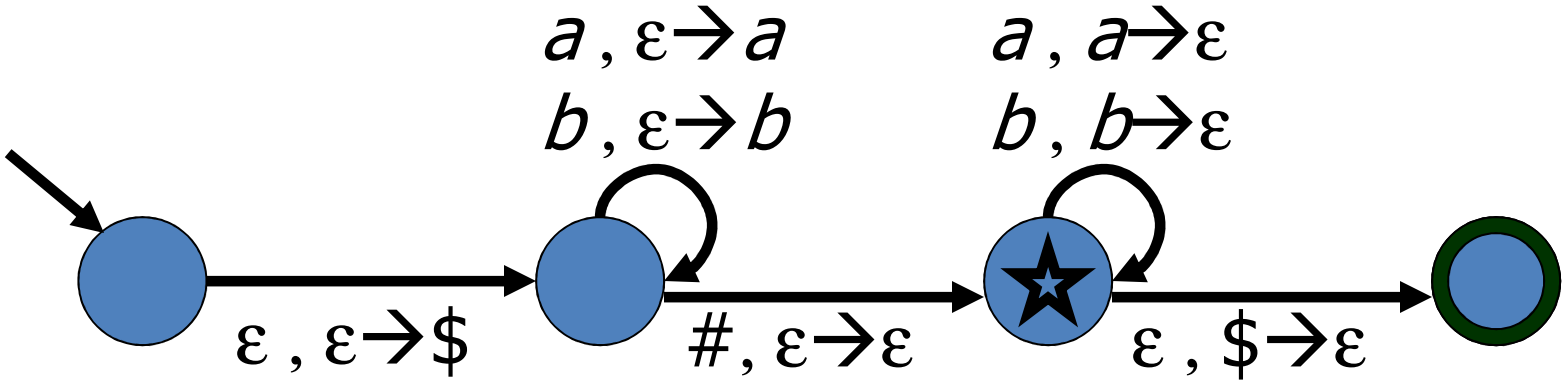


Input:

aaab# baaa

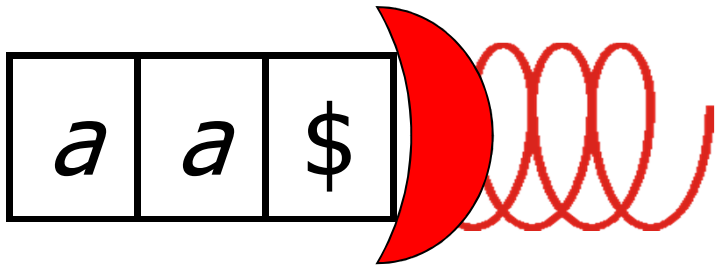


Versão Sipser's

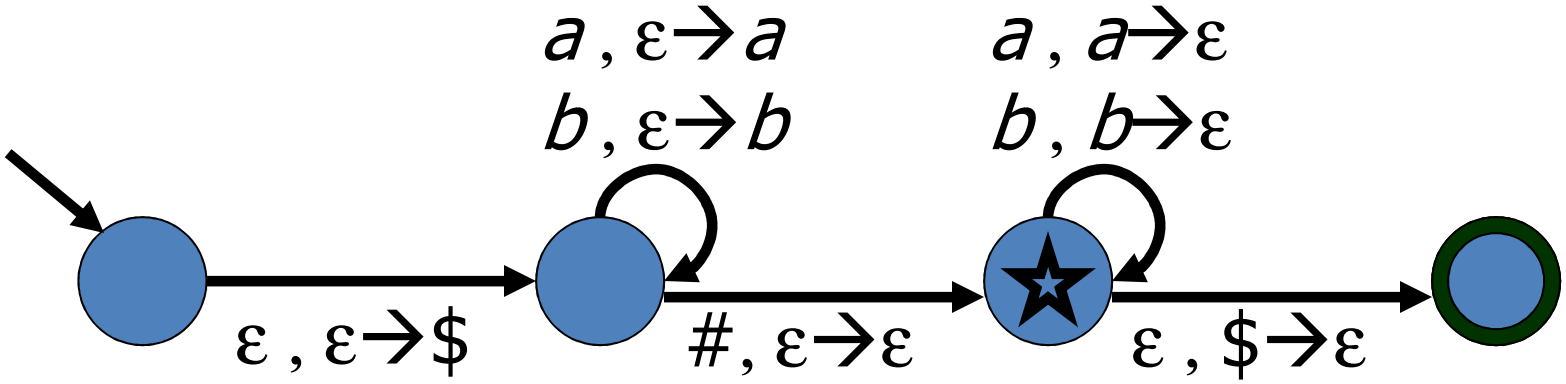


Input:

aaab# baaa

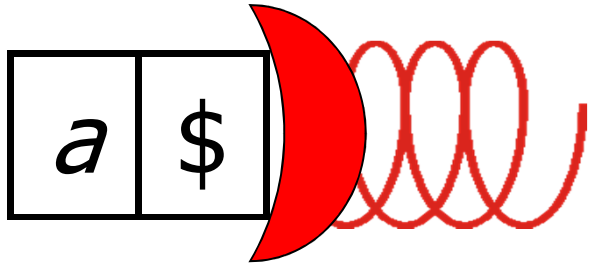


Versão Sipser's

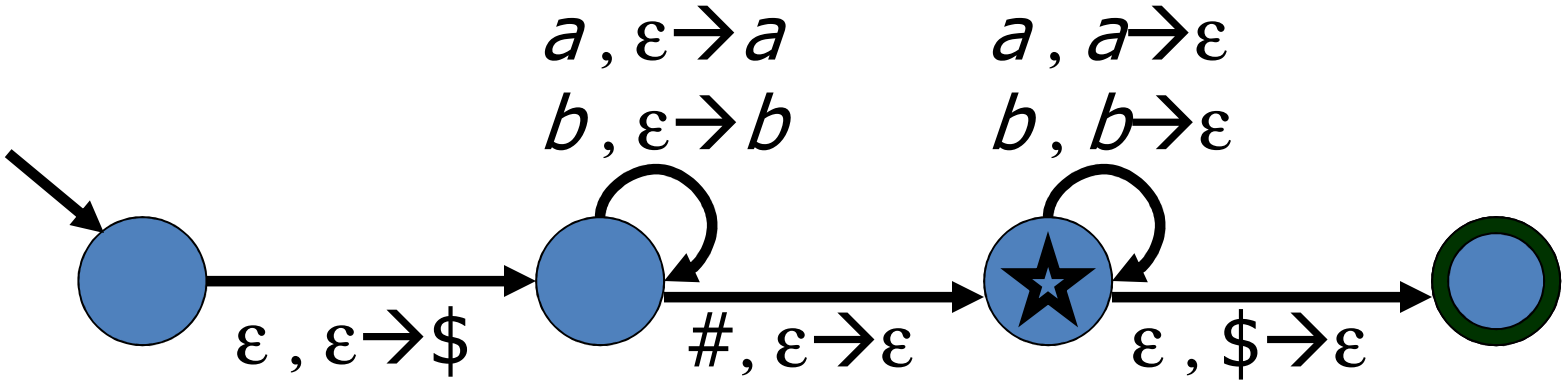


Input:

aaab# baaa

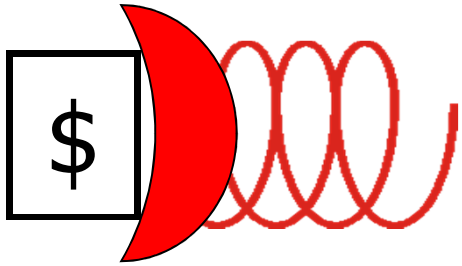


Versão Sipser's

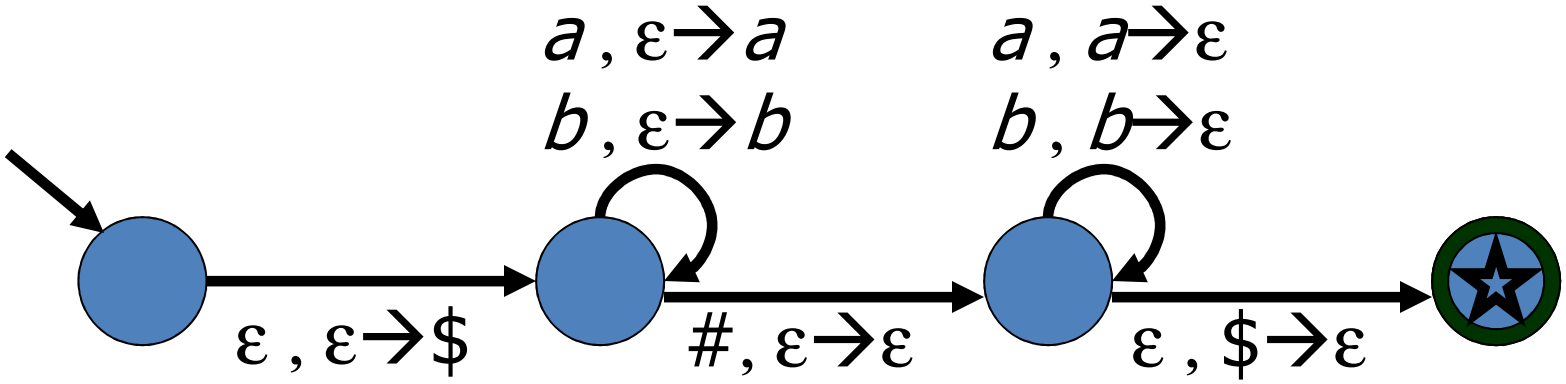


Input:

aaab# baaa



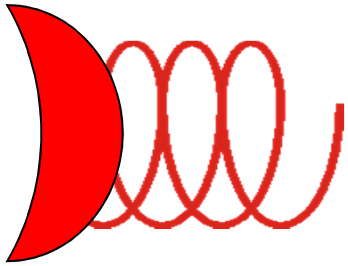
Versão Sipser's



Input:

aaab# baaa

ACCEPT!



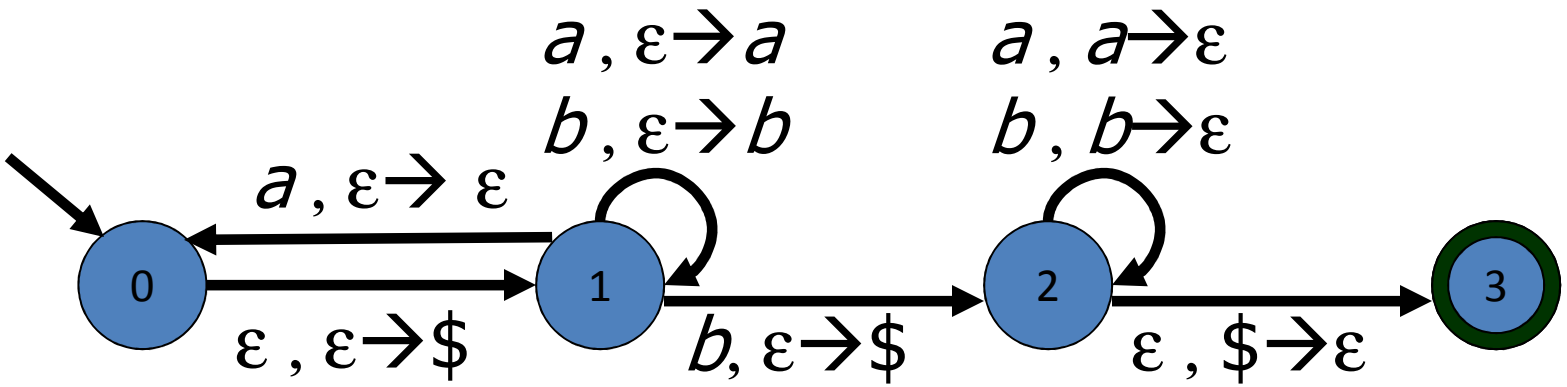
PDA
Definição Formal

DEF: Um ***autômato de pilha*** (PDA) é uma 6-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ onde Q, Σ e q_0, são como para um AF. Γ é o ***alpfabeto da pilha***. δ é uma função:

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$$

Portanto, dado um estado *p,* uma símbolo lido *x* e um símbolo de pilha *y*, $\delta(p,x,y)$ retorna todo (q,z) tal que *q* é um estado alvo e *z* é um símbolo que deve substituir *y*.

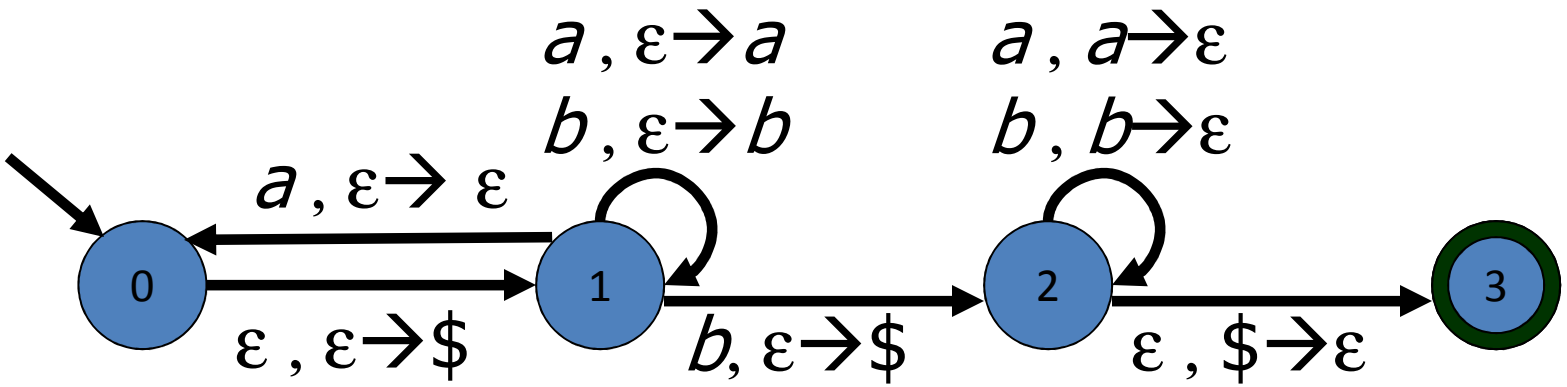
PDA
Definição Formal



Q: O que é $\delta(p,x,y)$ em cada caso?

1. $\delta(0,a,b)$
2. $\delta(0,\varepsilon,\varepsilon)$
3. $\delta(1,a,\varepsilon)$
4. $\delta(3,\varepsilon,\varepsilon)$

PDA
Definição Formal



R:

1. $\delta(0,a,b) = \varnothing$
2. $\delta(0,\varepsilon,\varepsilon) = \{(1,\$)\}$
3. $\delta(1,a,\varepsilon) = \{(0,\varepsilon),(1,a)\}$
4. $\delta(3,\varepsilon,\varepsilon) = \varnothing$

PDA - Exercício

- Forneça PDA´s para reconhecer as seguintes linguagens:
  - $\{a^n b^n \mid n \in \aleph\}$
  - $\{a^n b^{2n} \mid n \in \aleph\}$
  - $\{a^{2n} b^n \mid n \in \aleph\}$
  - $\{a^i b^j c^k \mid i = j \text{ ou } i=k\}$
  - $\{w\, w^R \mid w \in \{0,1\}^*\}$
  - $\{w \in \{0,1\}^* \mid$ o número de 0´s é igual ao de 1´s $\}$