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:

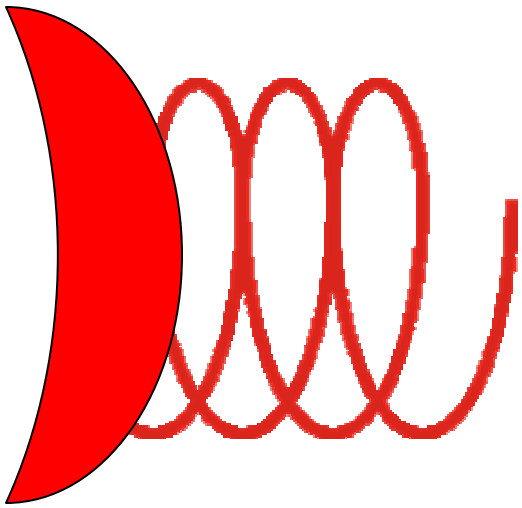
```
long factorial(int n){
  if (n<=0) return 1;
  return n*factorial(n-1);
}
```

Como JVM executa `factorial(5)`?

Algoritmos Recursivos e Pilhas

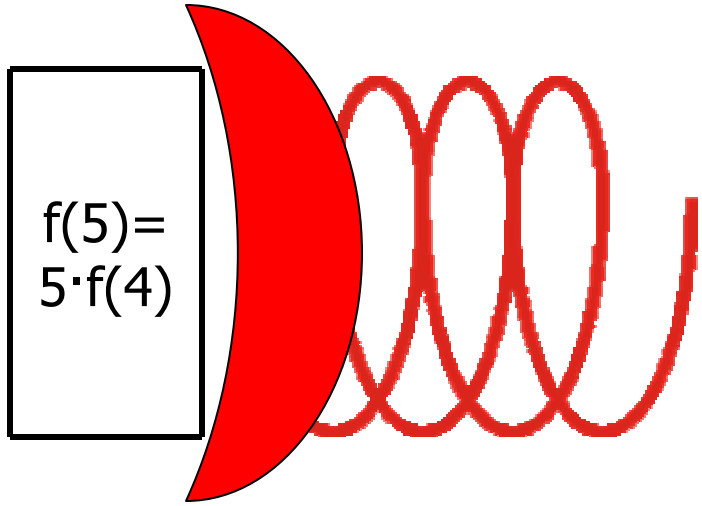
```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```

Compute 5!



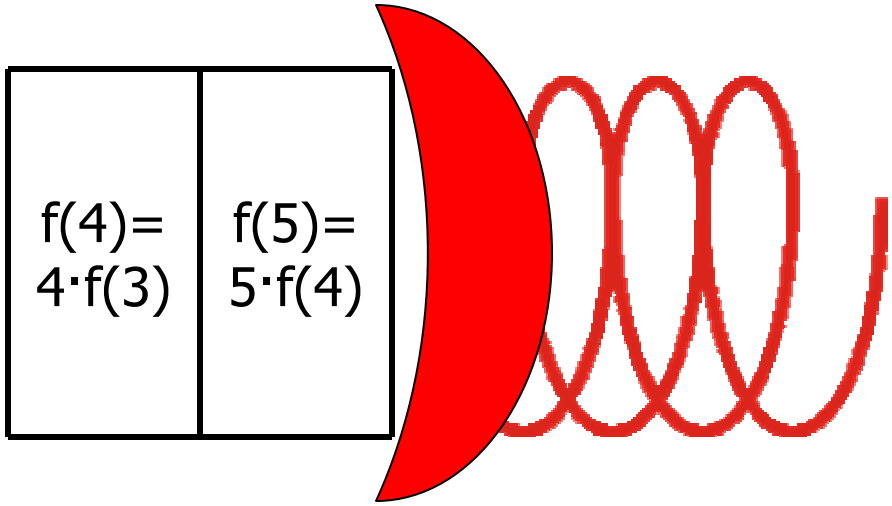
Algoritmos Recursivos e Pilhas

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



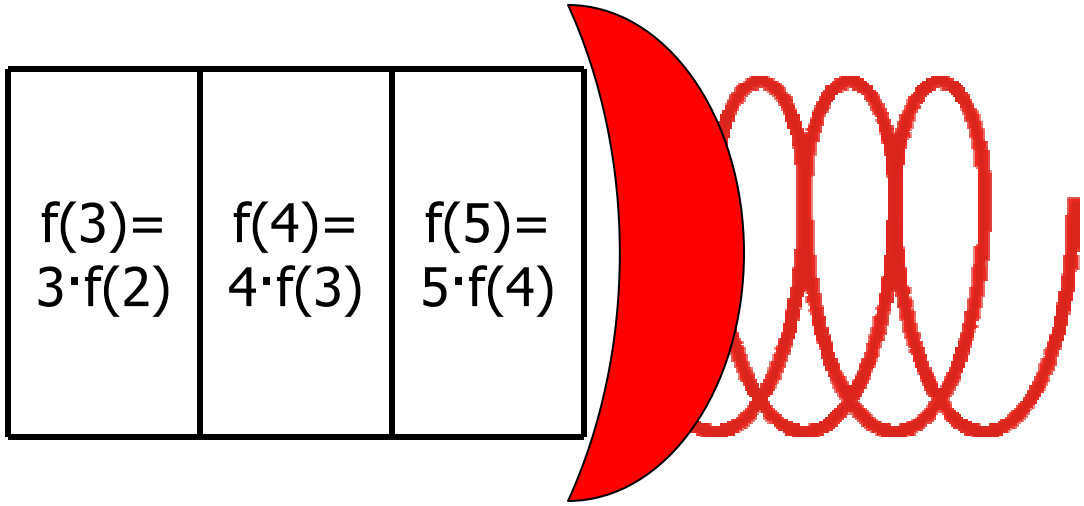
Algoritmos Recursivos e Pilhas

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



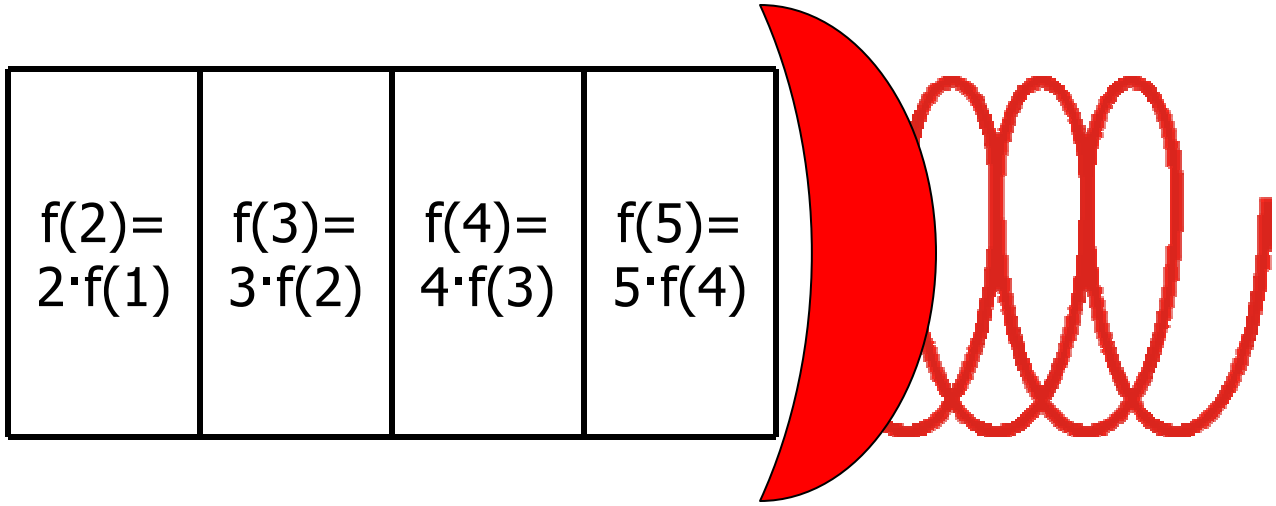
Algoritmos Recursivos e Pilhas

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



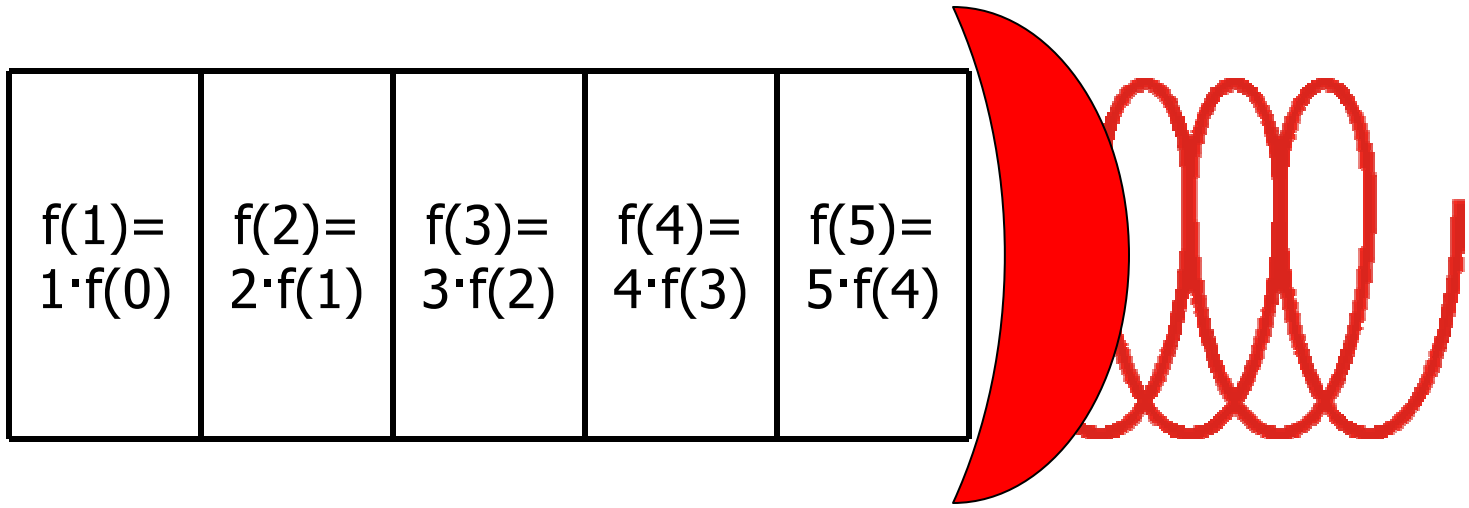
Algoritmos Recursivos e Pilhas

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



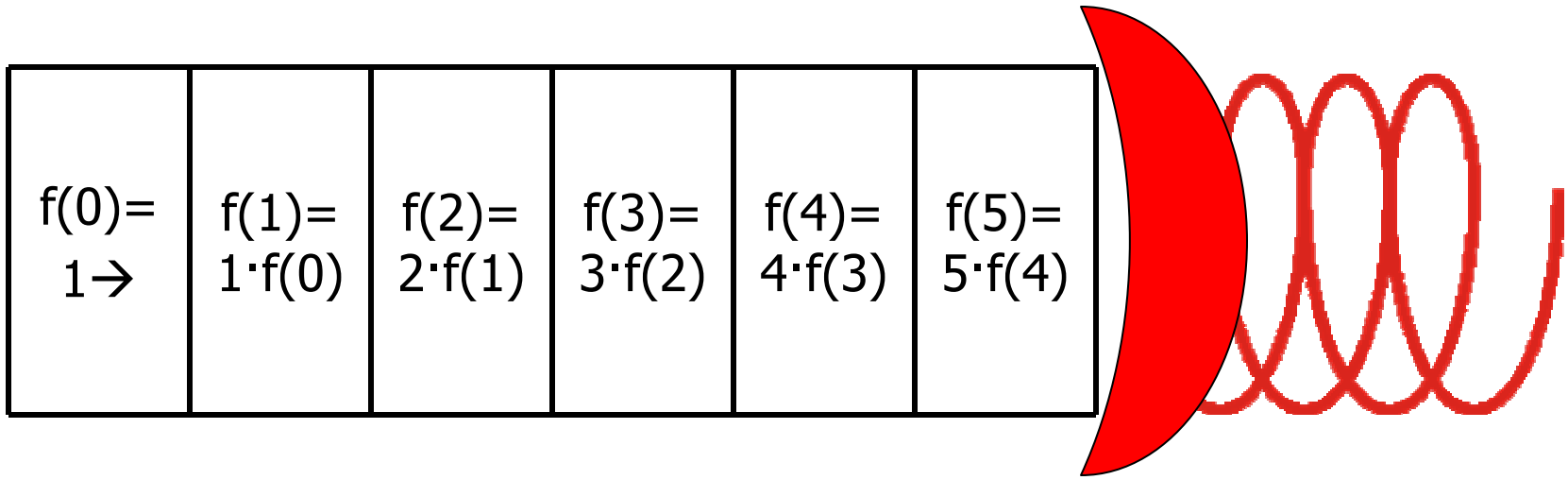
Algoritmos Recursivos e Pilhas

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



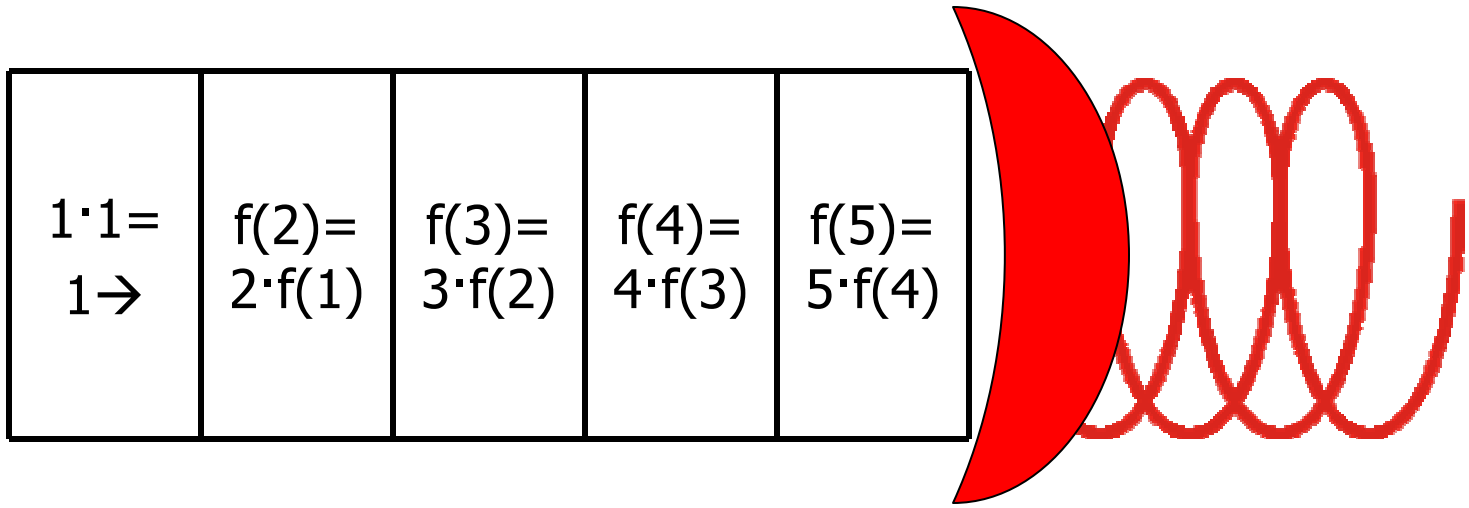
Algoritmos Recursivos e Pilhas

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



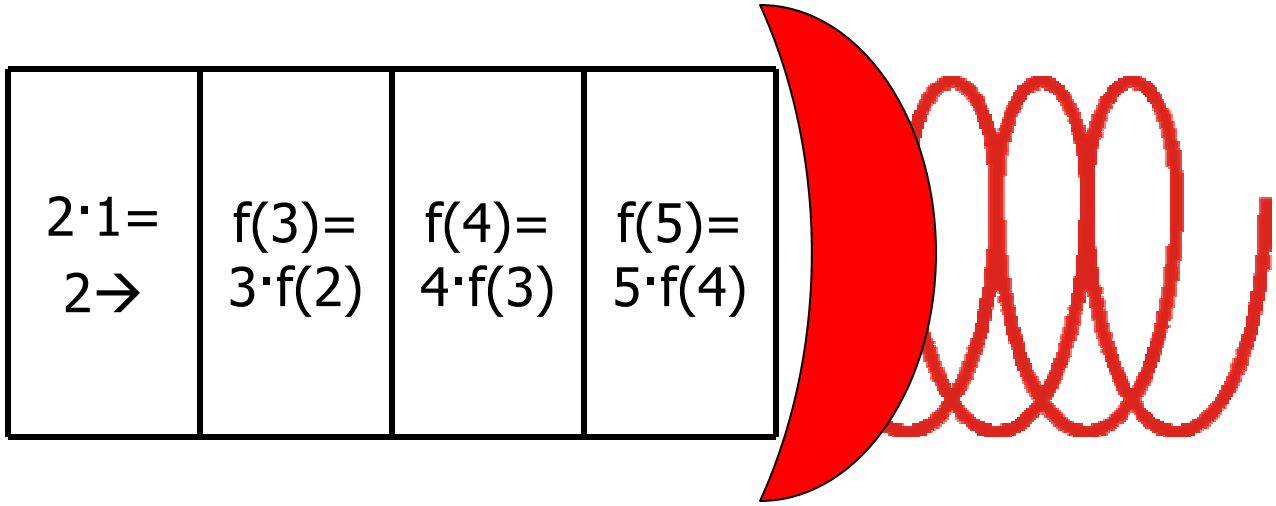
Algoritmos Recursivos e Pilhas

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



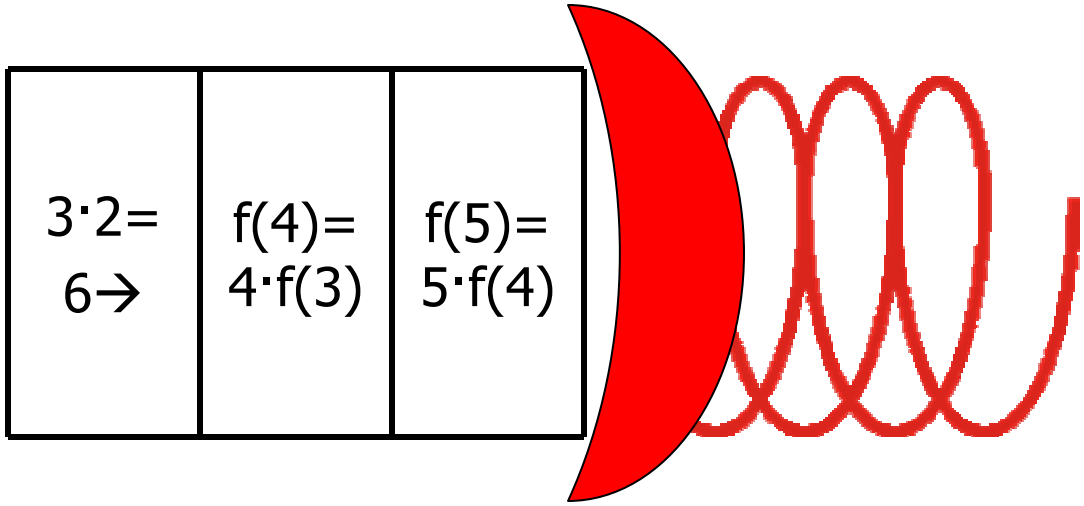
Algoritmos Recursivos e Pilhas

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



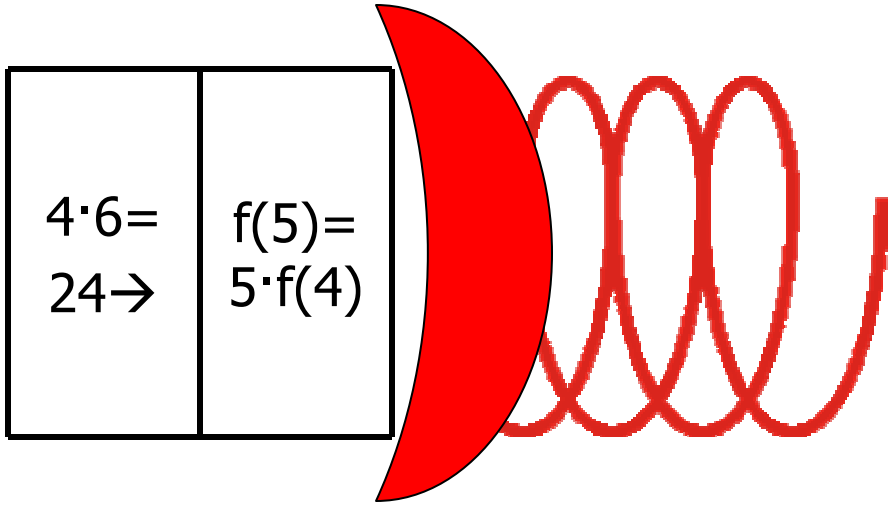
Algoritmos Recursivos e Pilhas

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



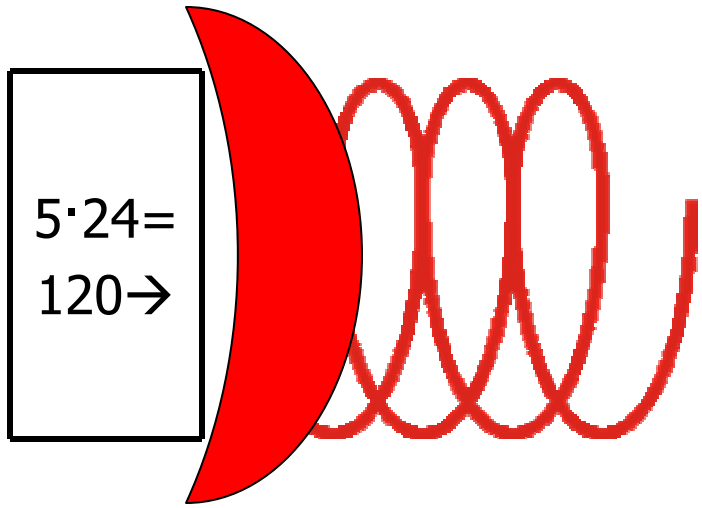
Algoritmos Recursivos e Pilhas

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



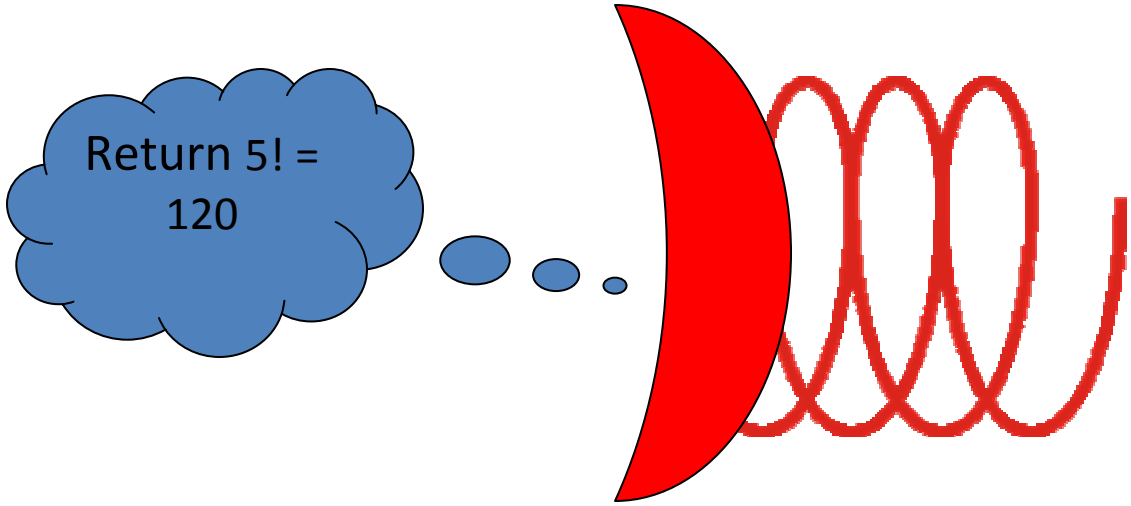
Algoritmos Recursivos e Pilhas

```
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?

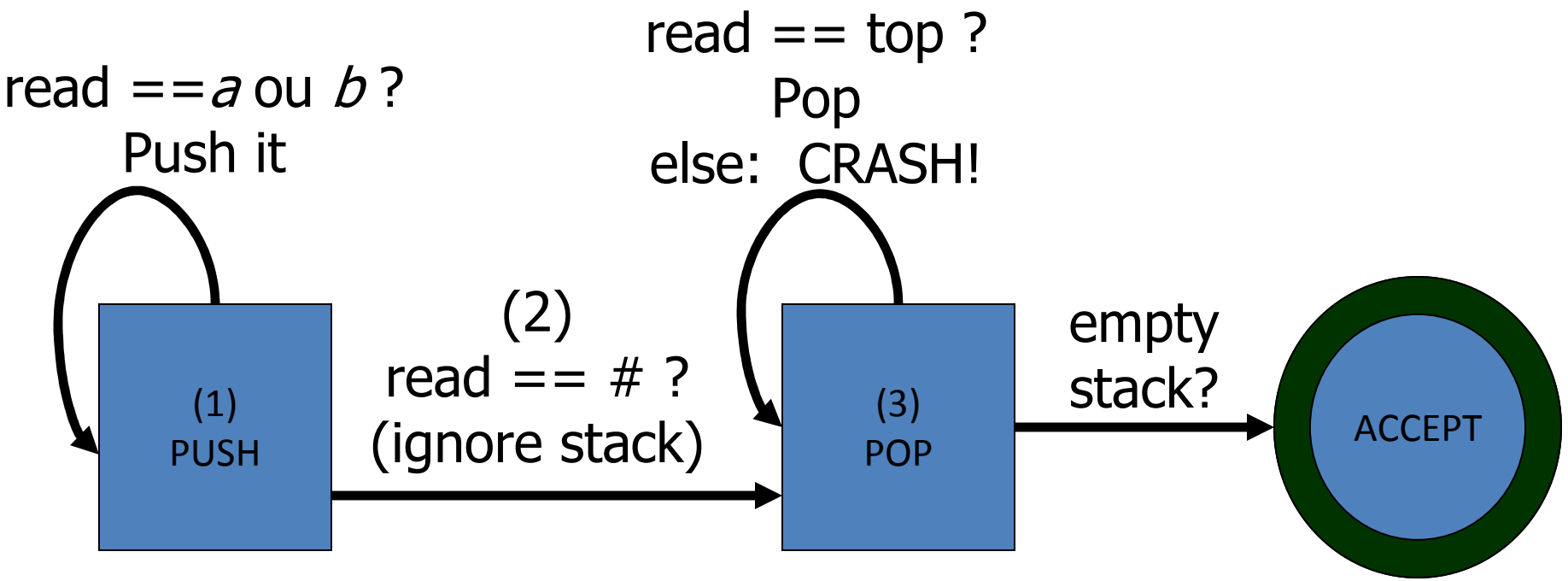
De CFG's para Máquinas de Pilha

A: Usamos um processo com três fases:

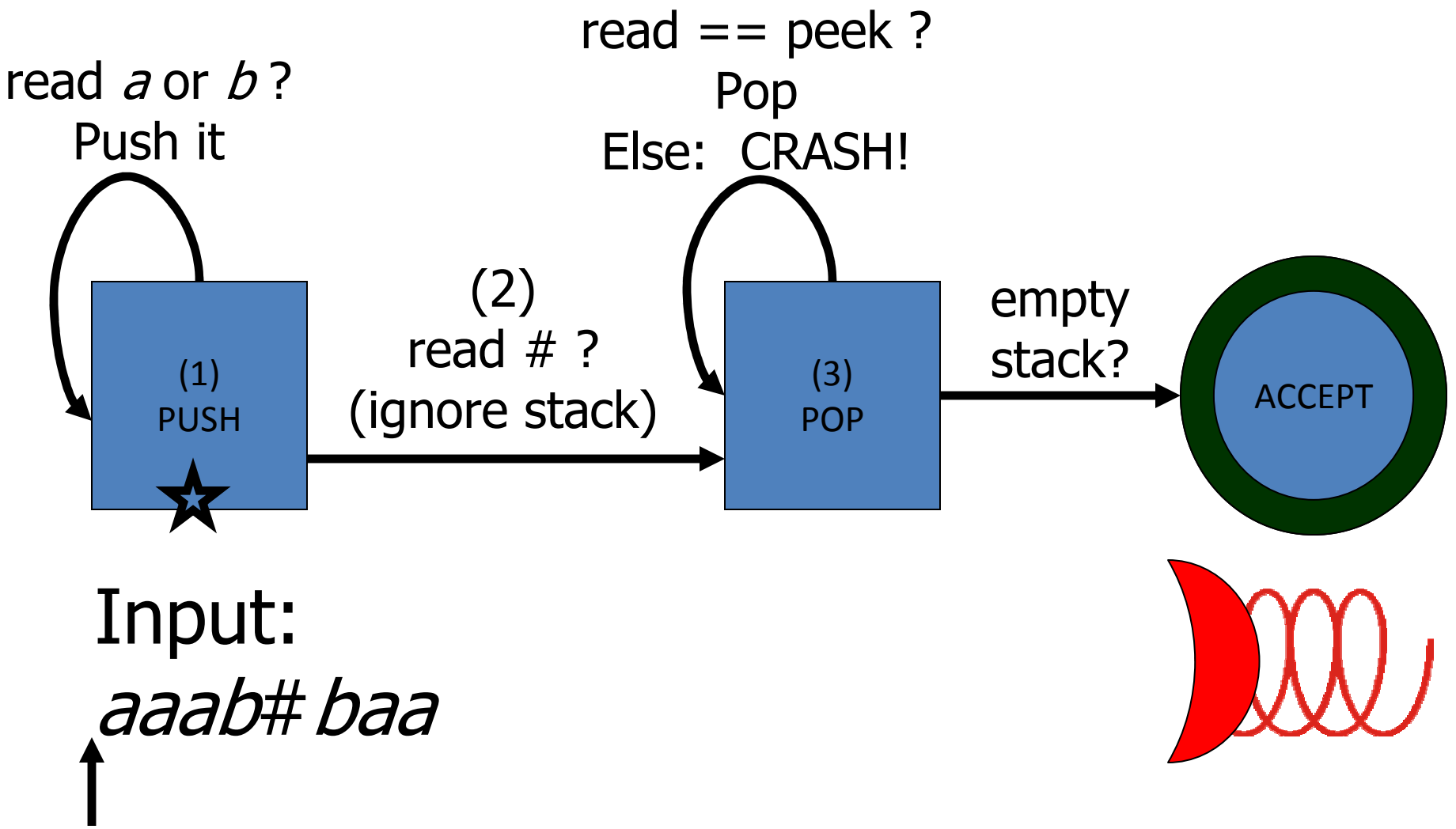
1. **modo Push :** Antes de ler "#", empilhe qualquer símbolo lido.
2. Ao ler "#" troque de modo de operação.
3. **modo Pop :** Leia os símbolos restantes garantindo que cada novo símbolo lido é idêntico ao símbolo desempilhado.

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

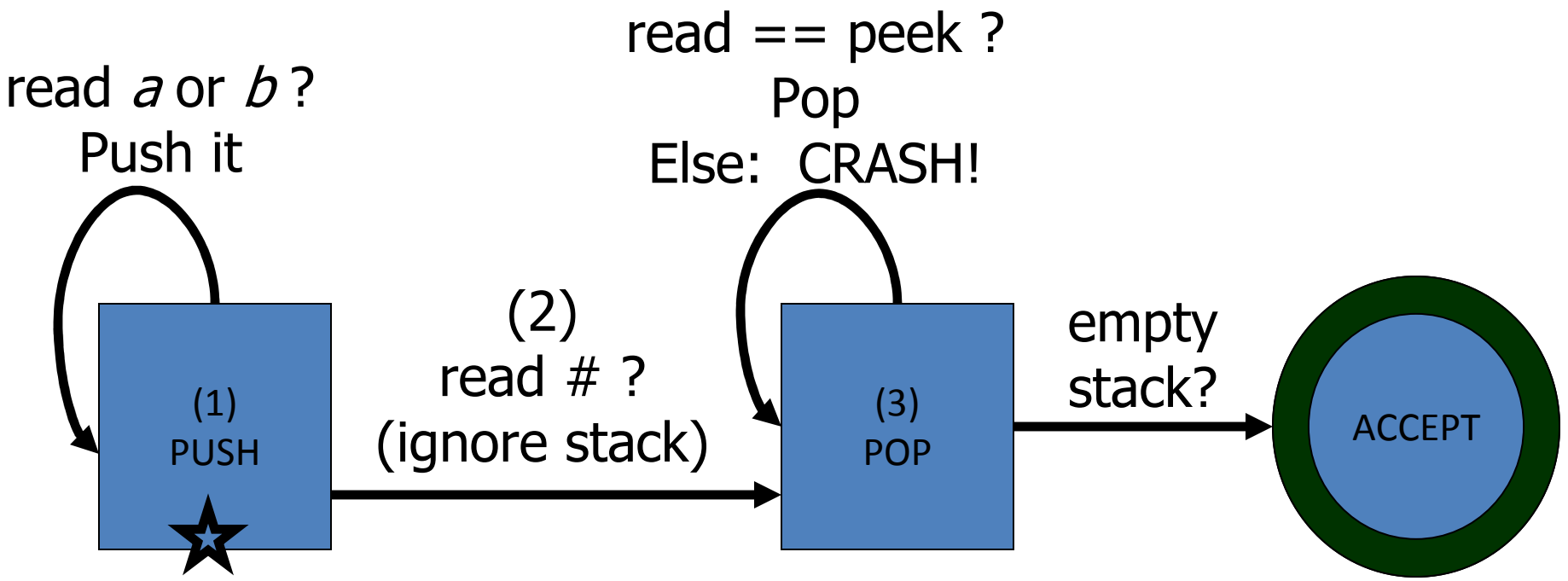
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

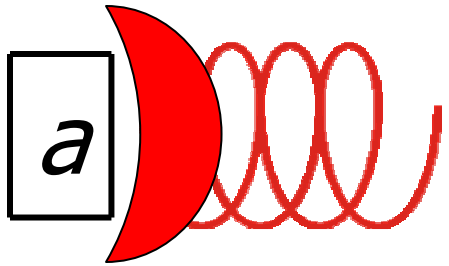


De CFG's para Máquinas de Pilha

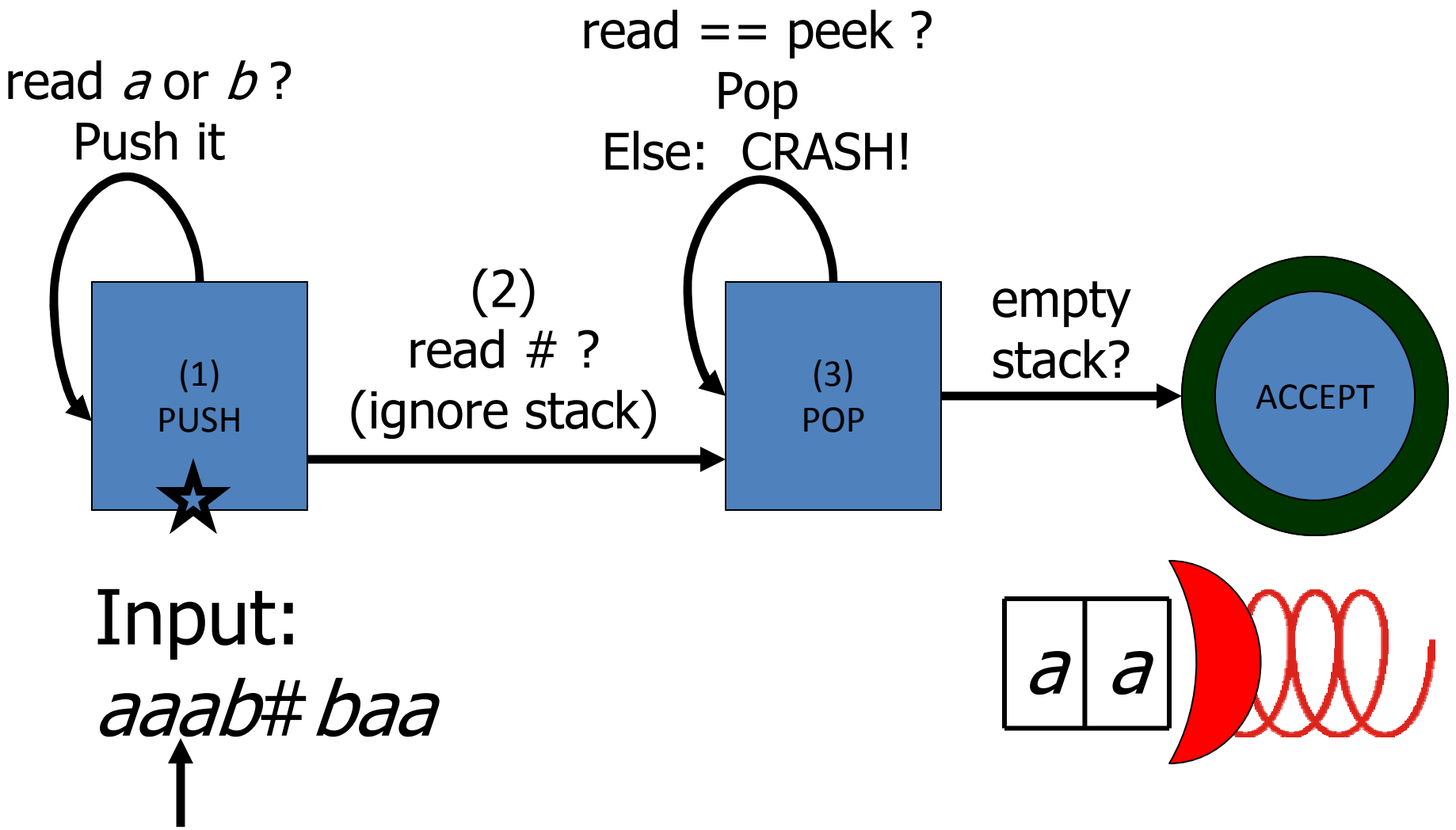


Input:

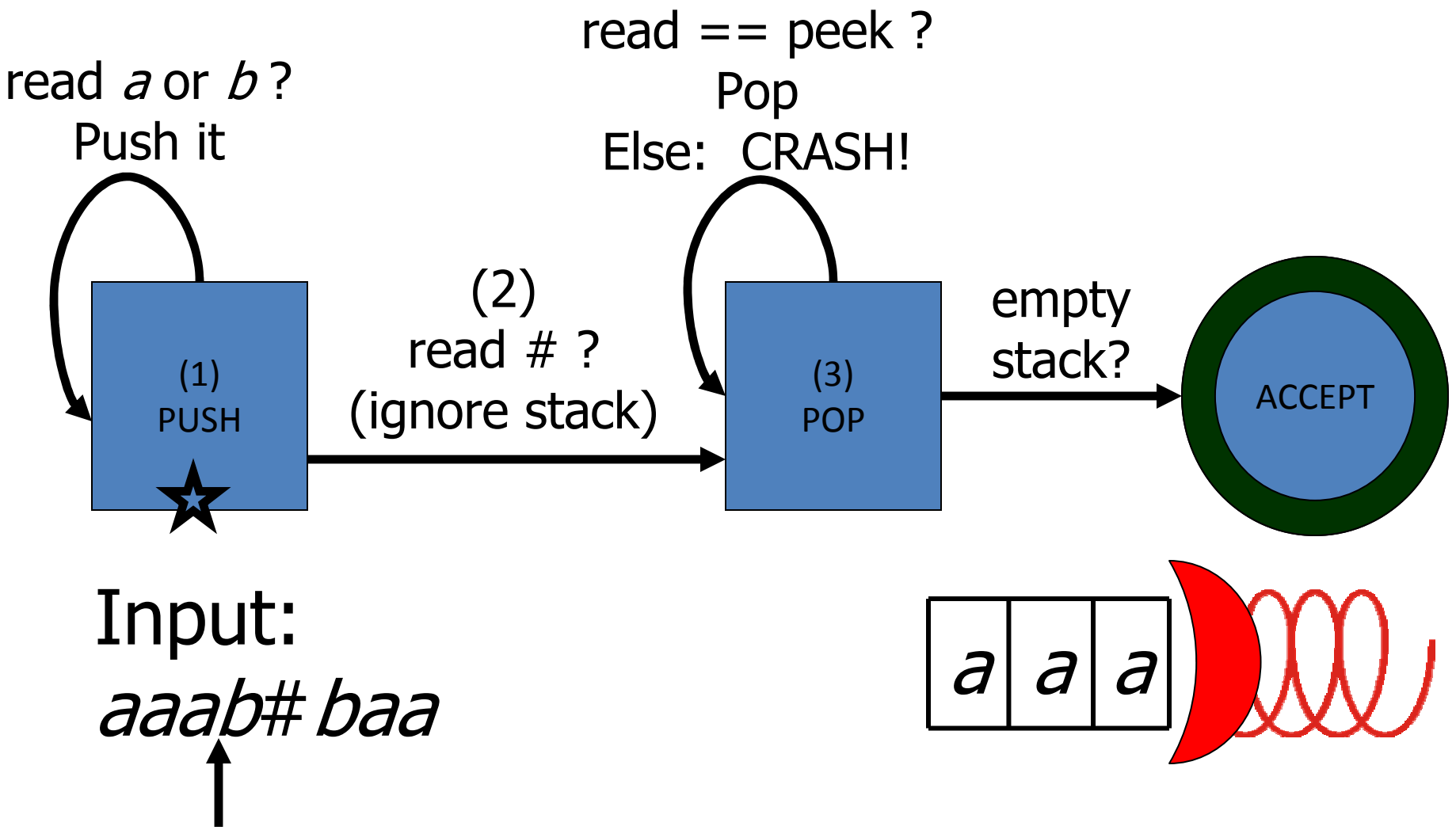
aaab# baa



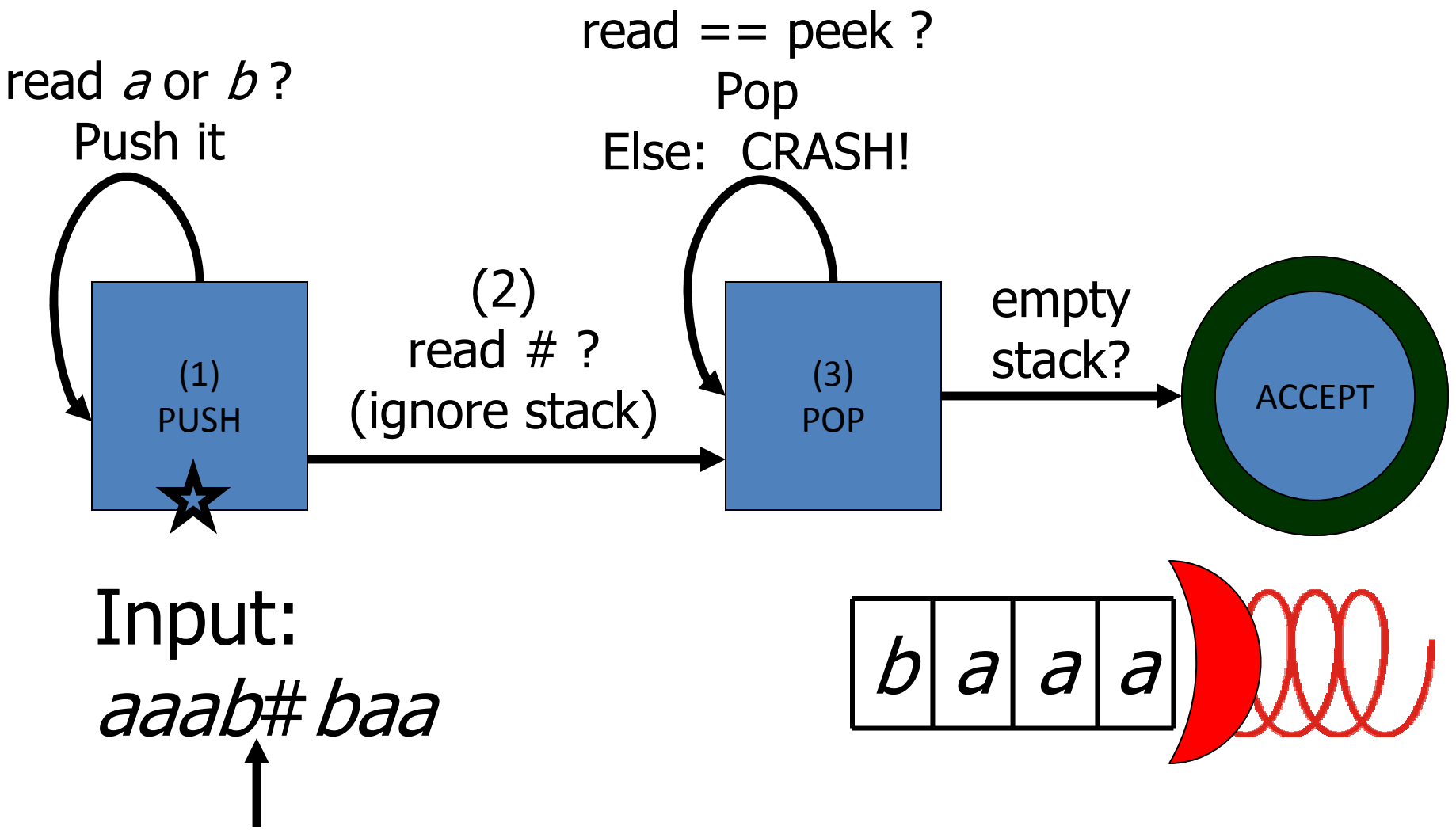
De CFG's para Máquinas de Pilha



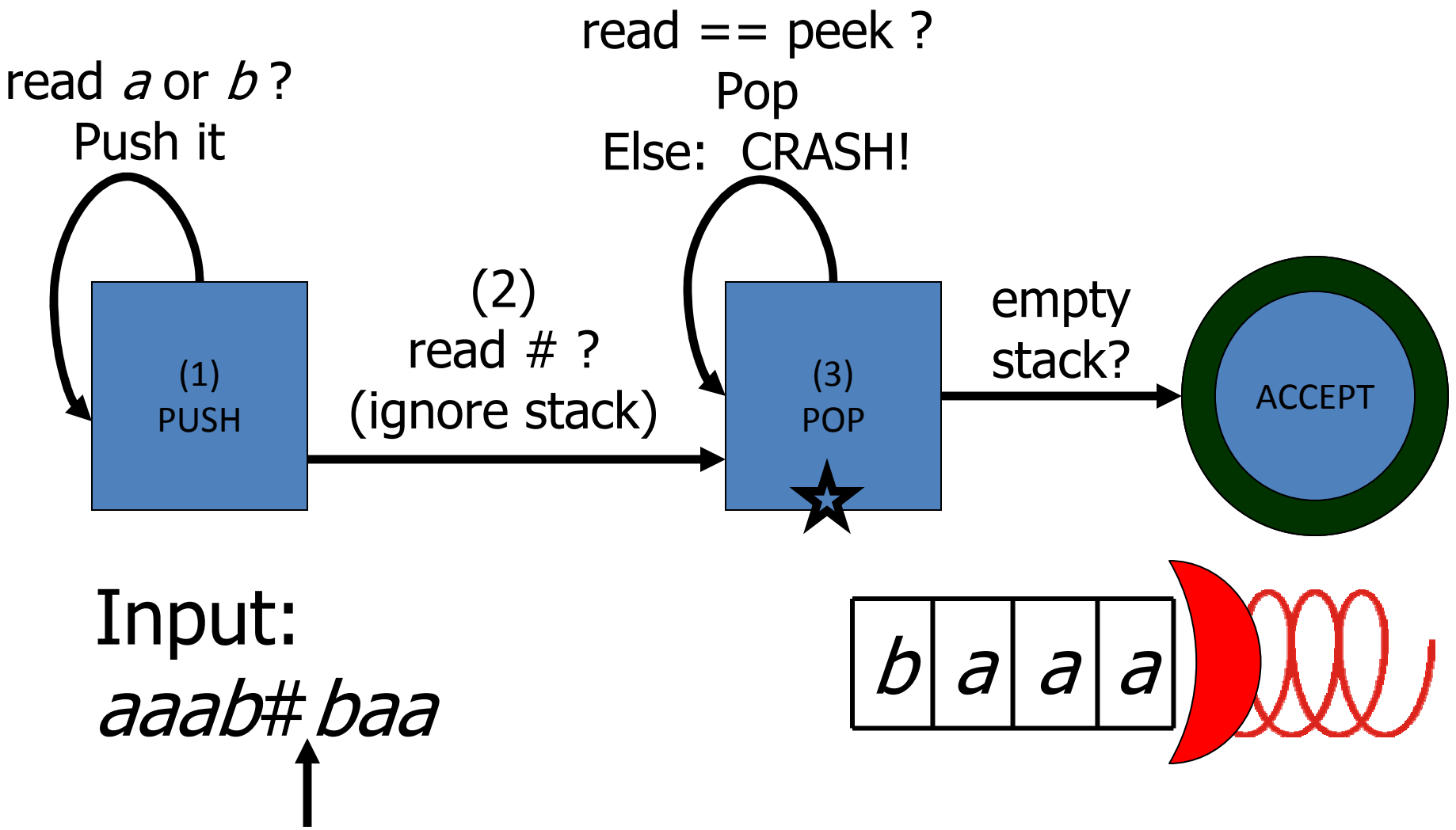
De CFG's para Máquinas de Pilha



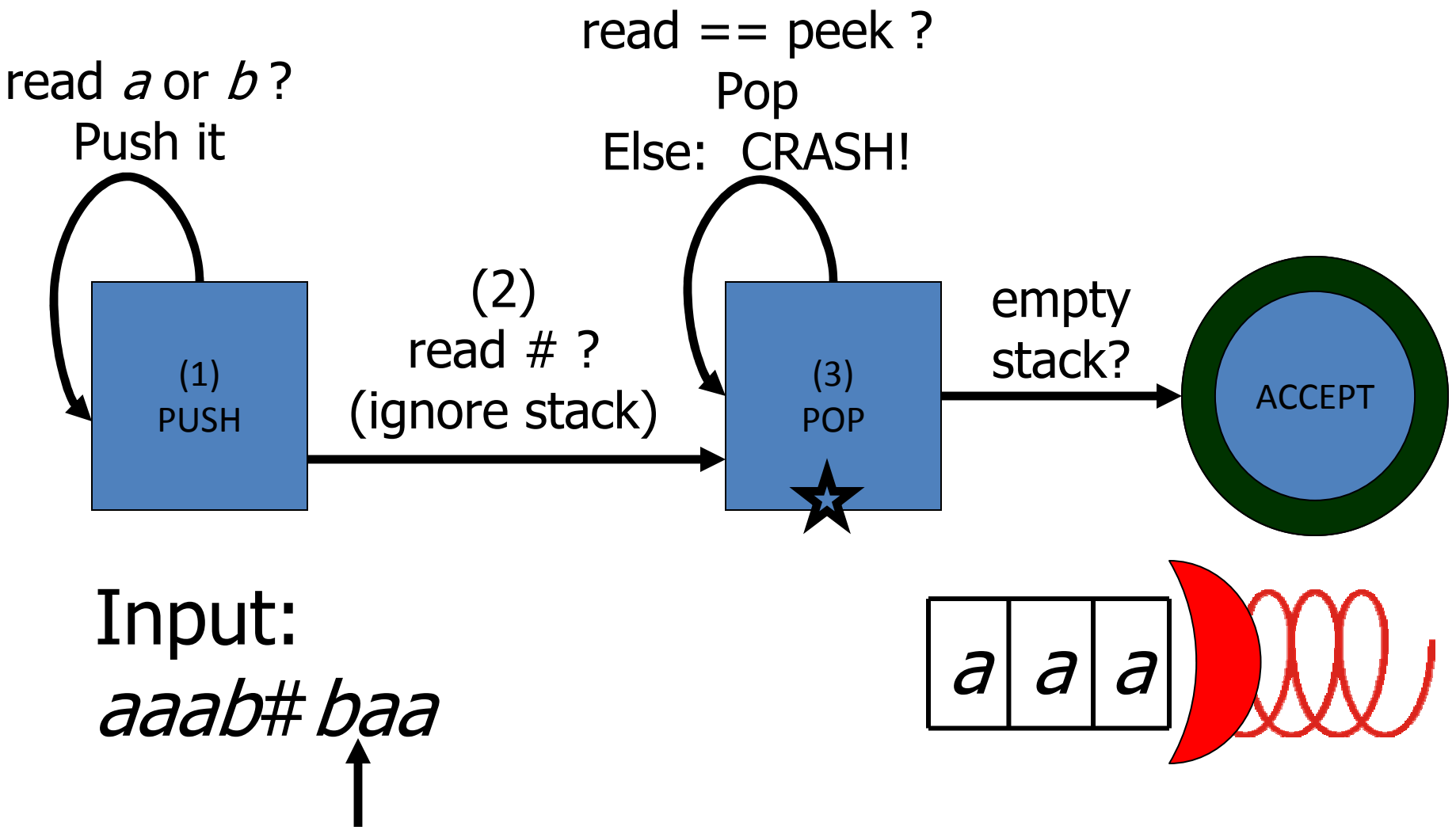
De CFG's para Máquinas de Pilha



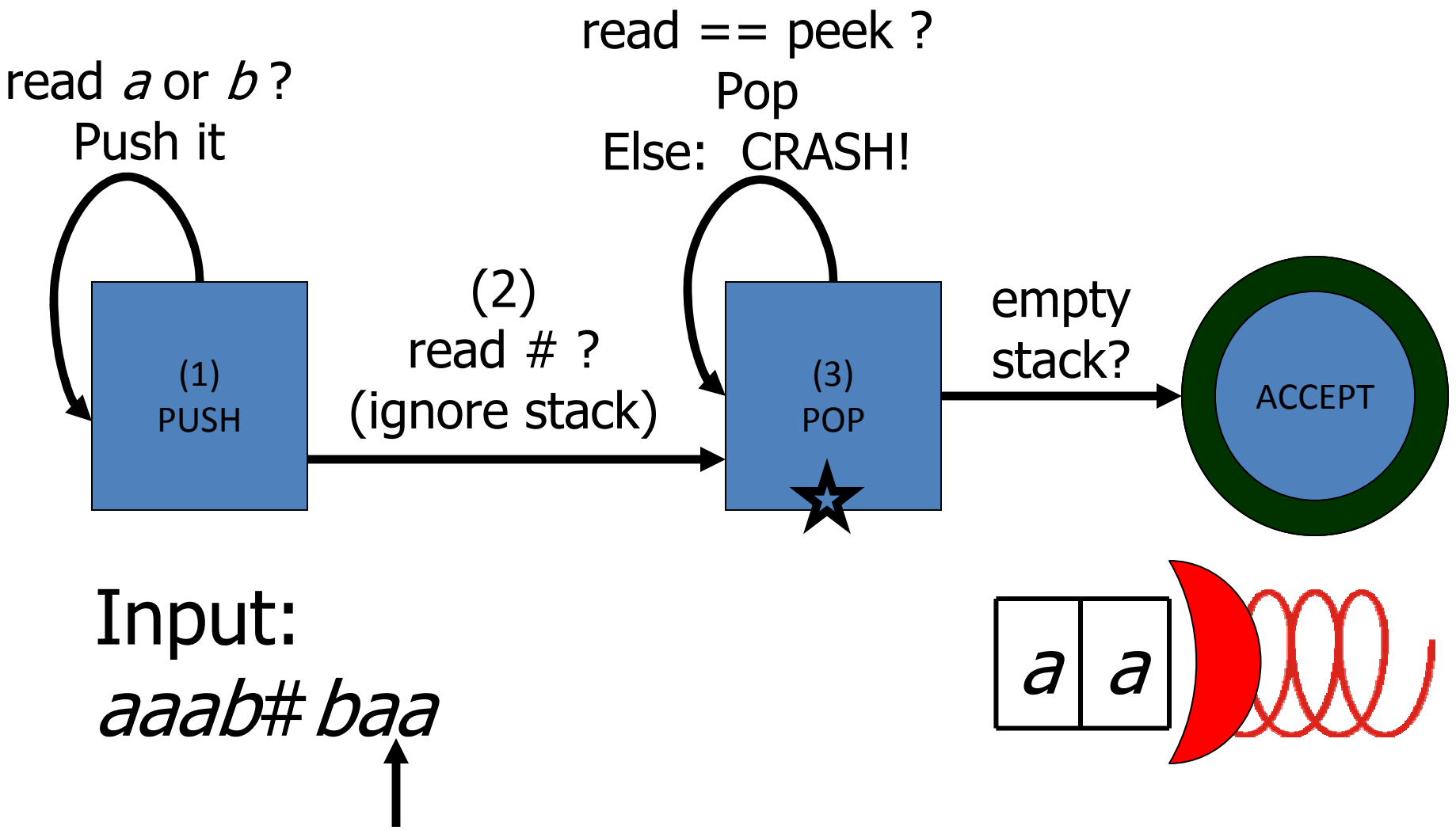
De CFG's para Máquinas de Pilha



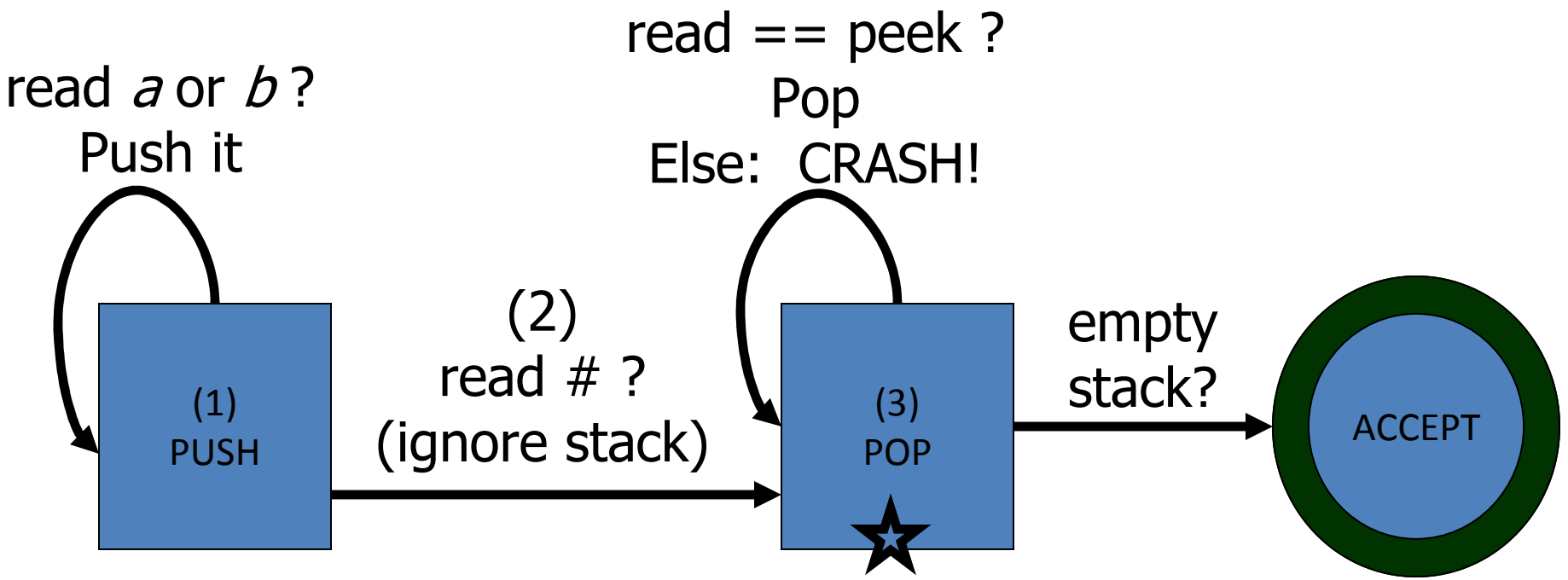
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

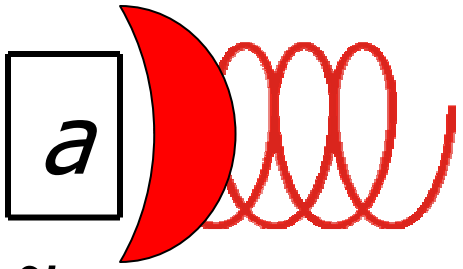


De CFG's para Máquinas de Pilha



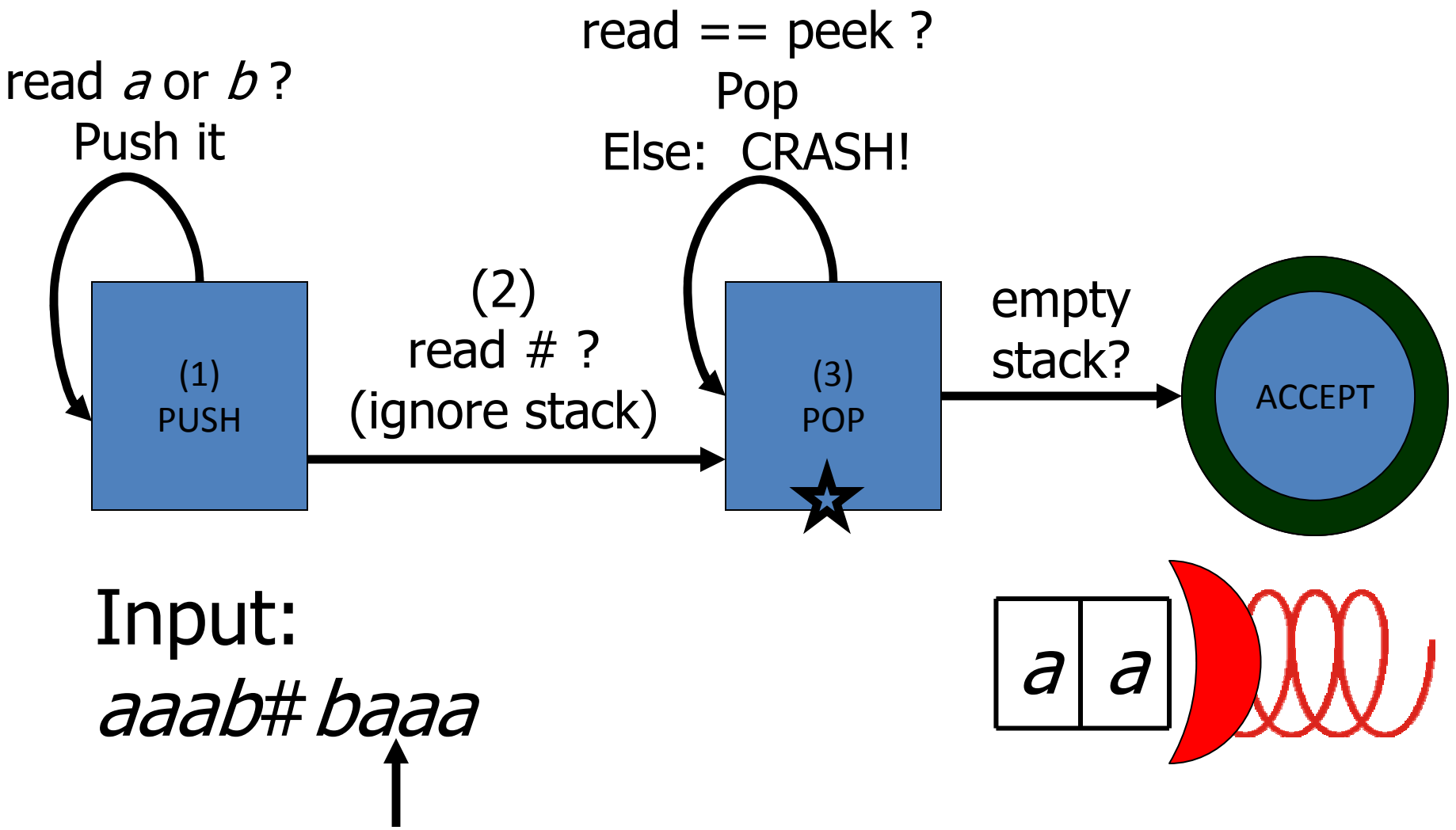
Input:

aaab# baa

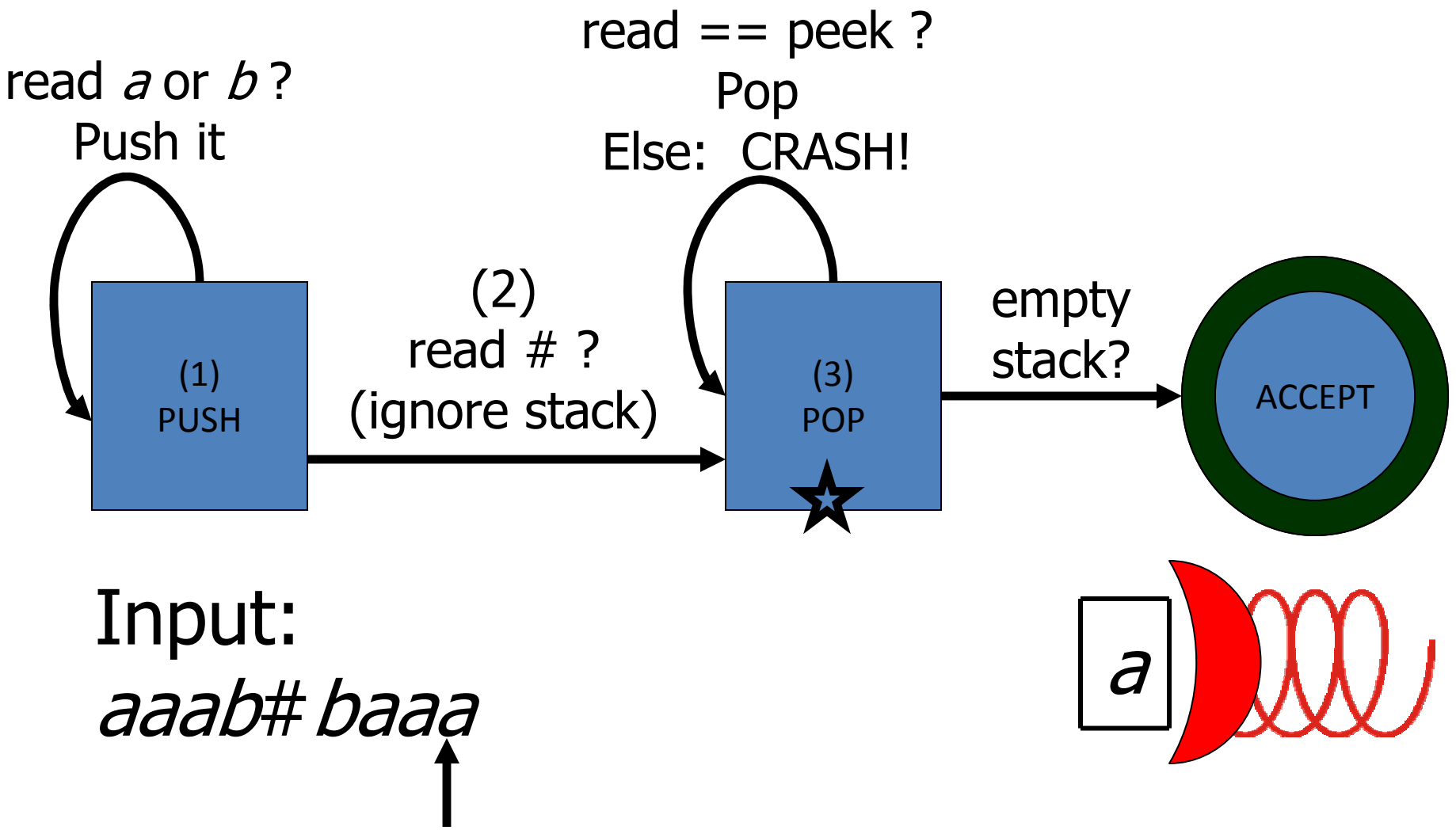


REJECT (pilha não vazia)

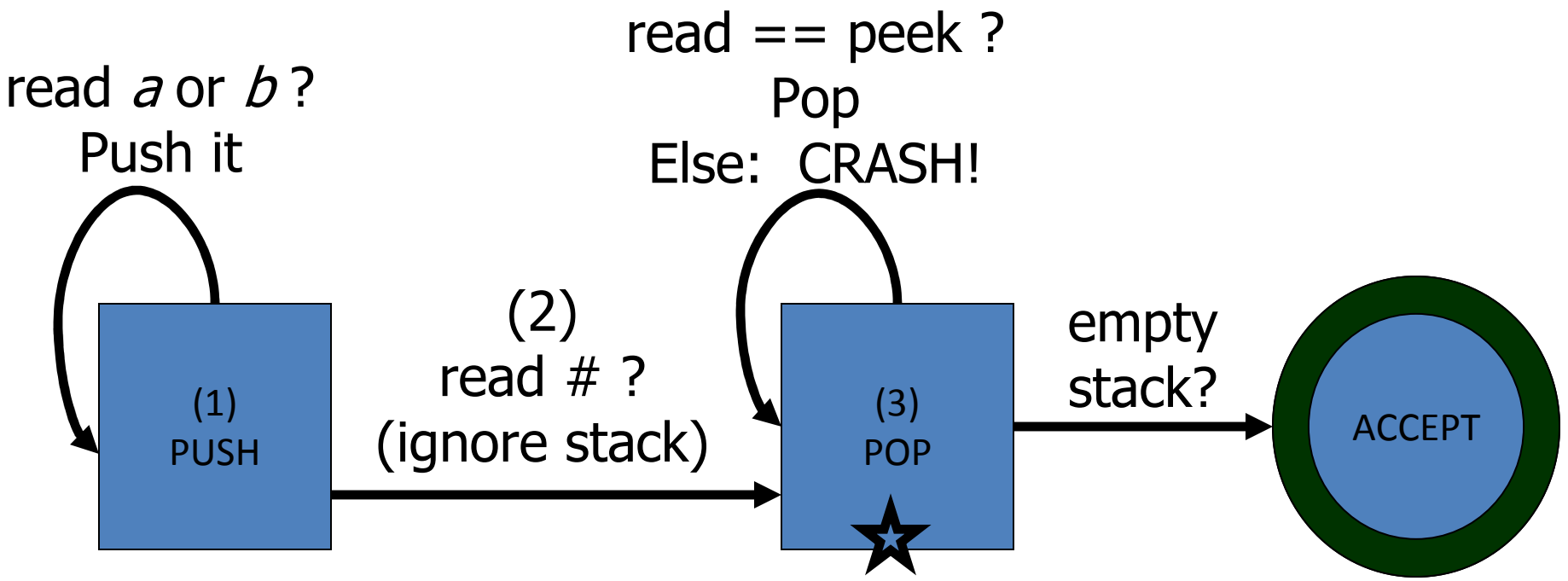
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

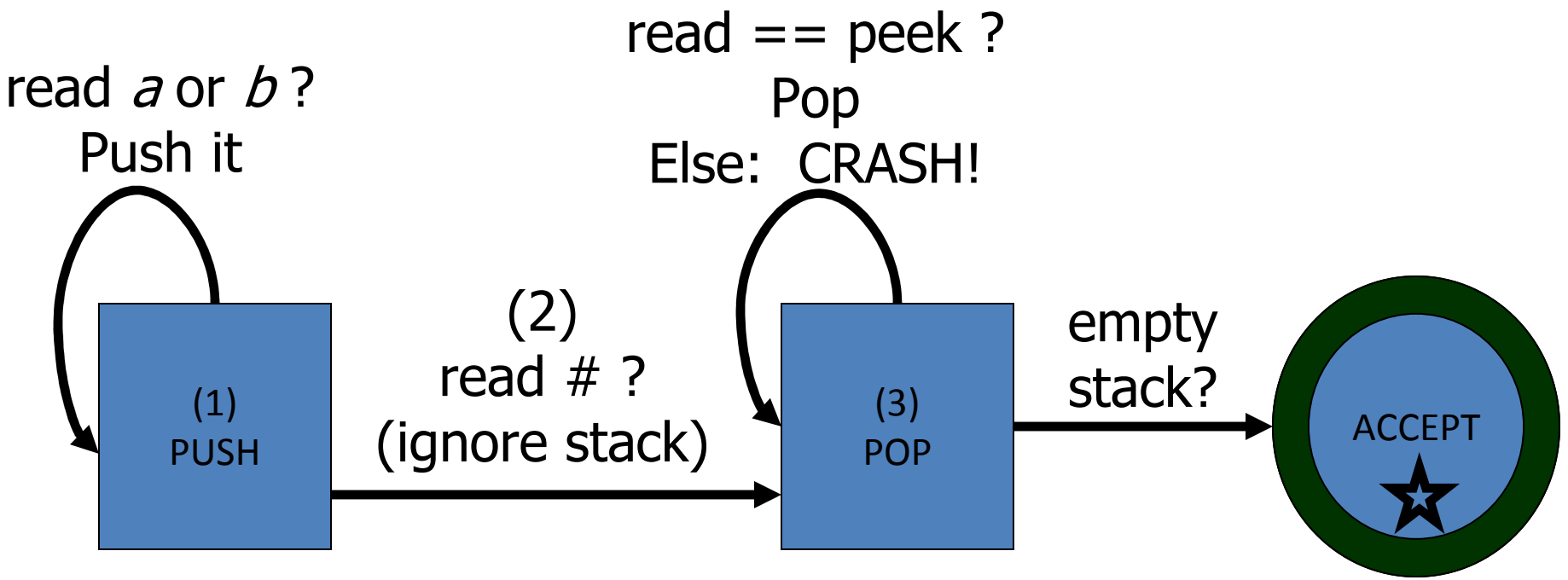


Input:

aaab# baaa

Pause input

De CFG's para Máquinas de Pilha

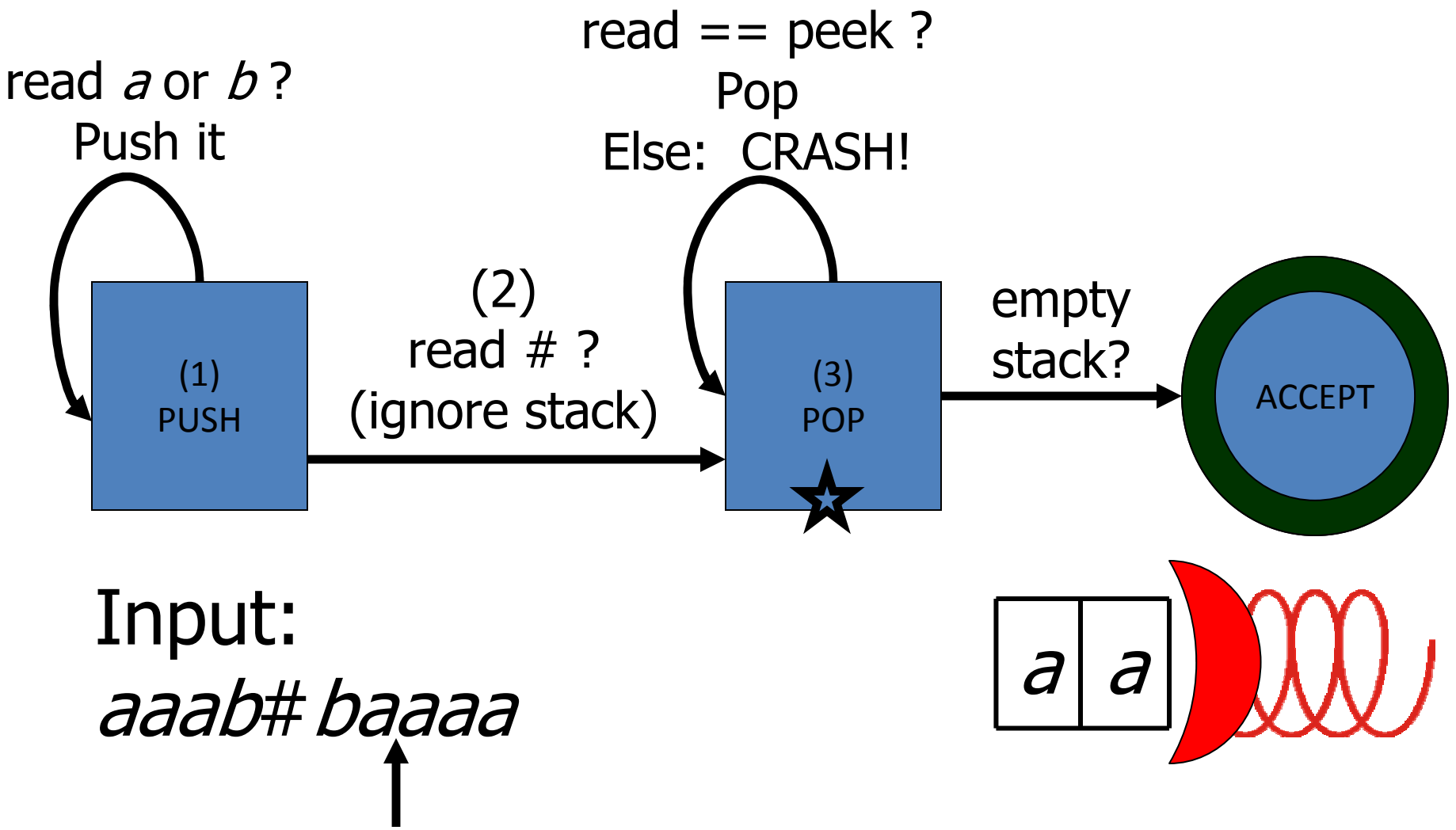


Input:

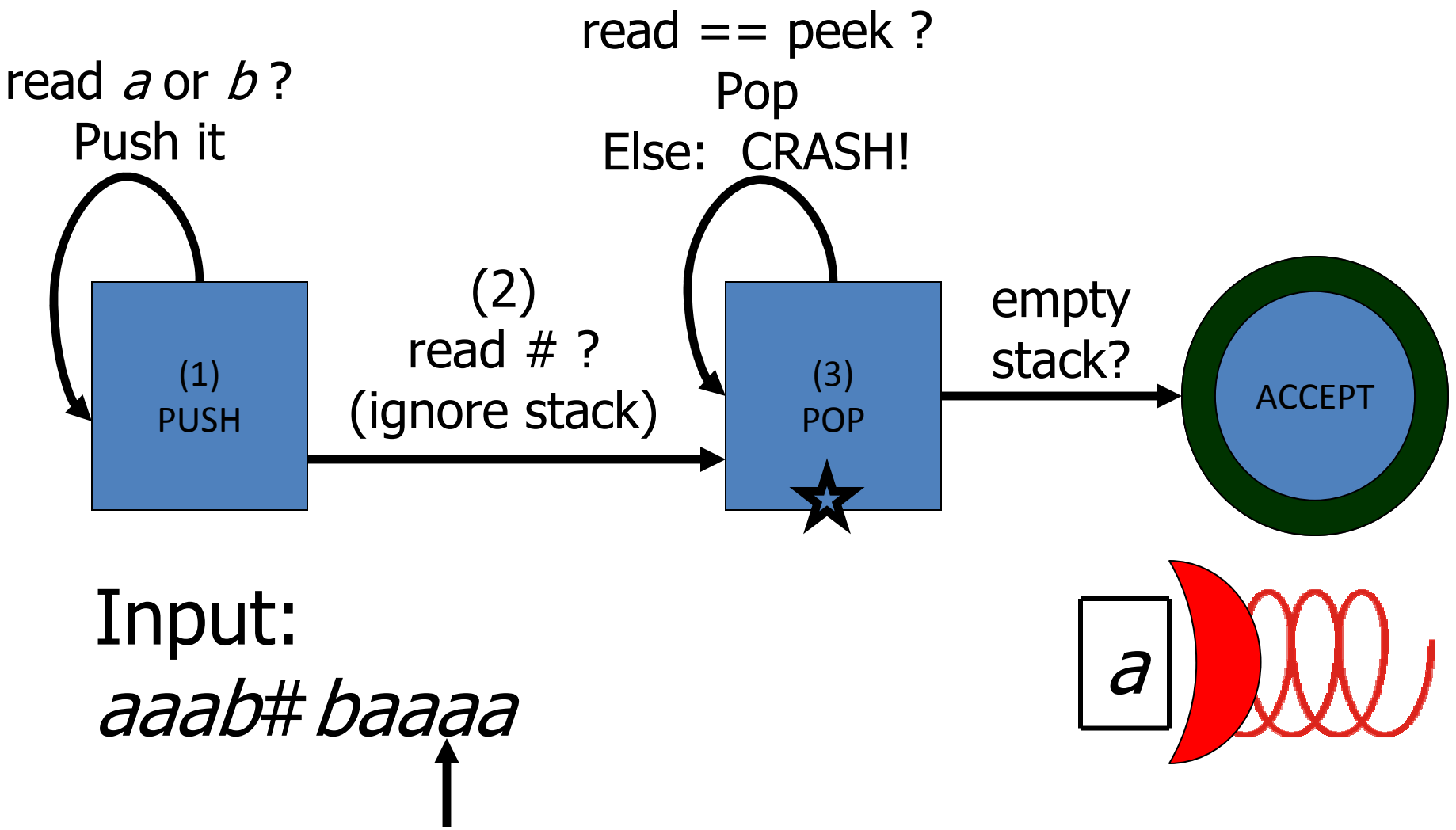
aaab# baaa

ACCEPT

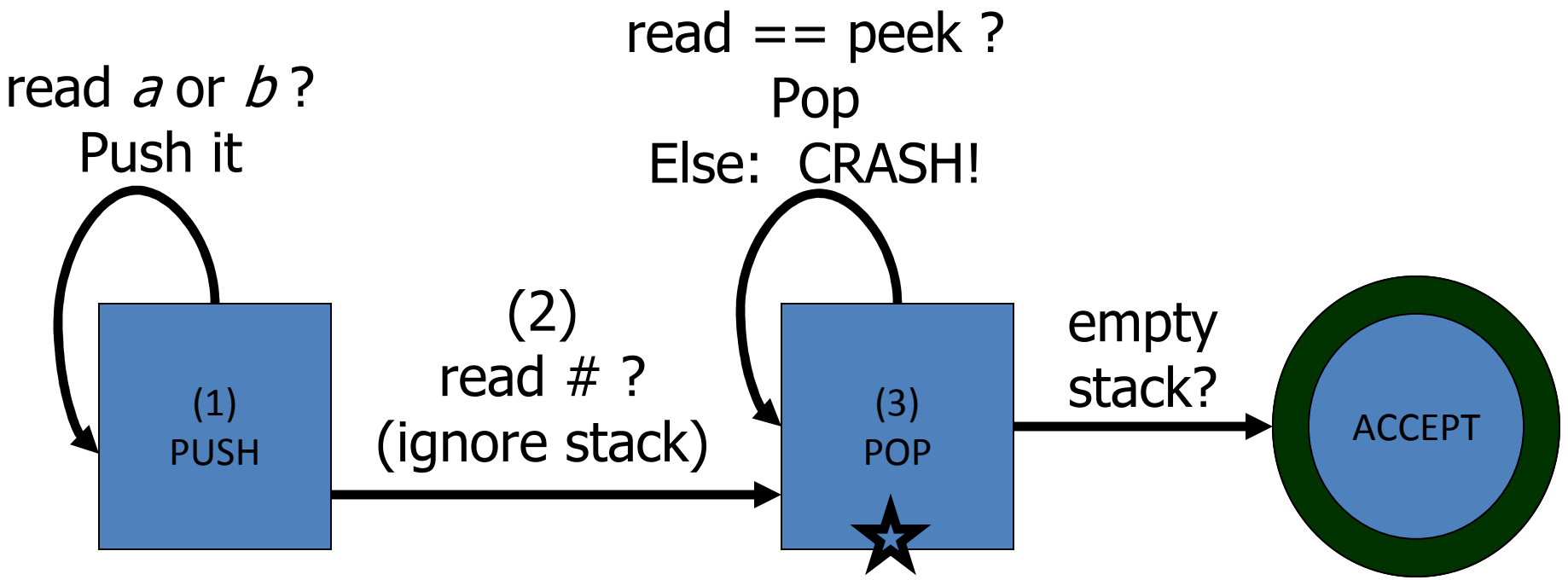
De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha



De CFG's para Máquinas de Pilha

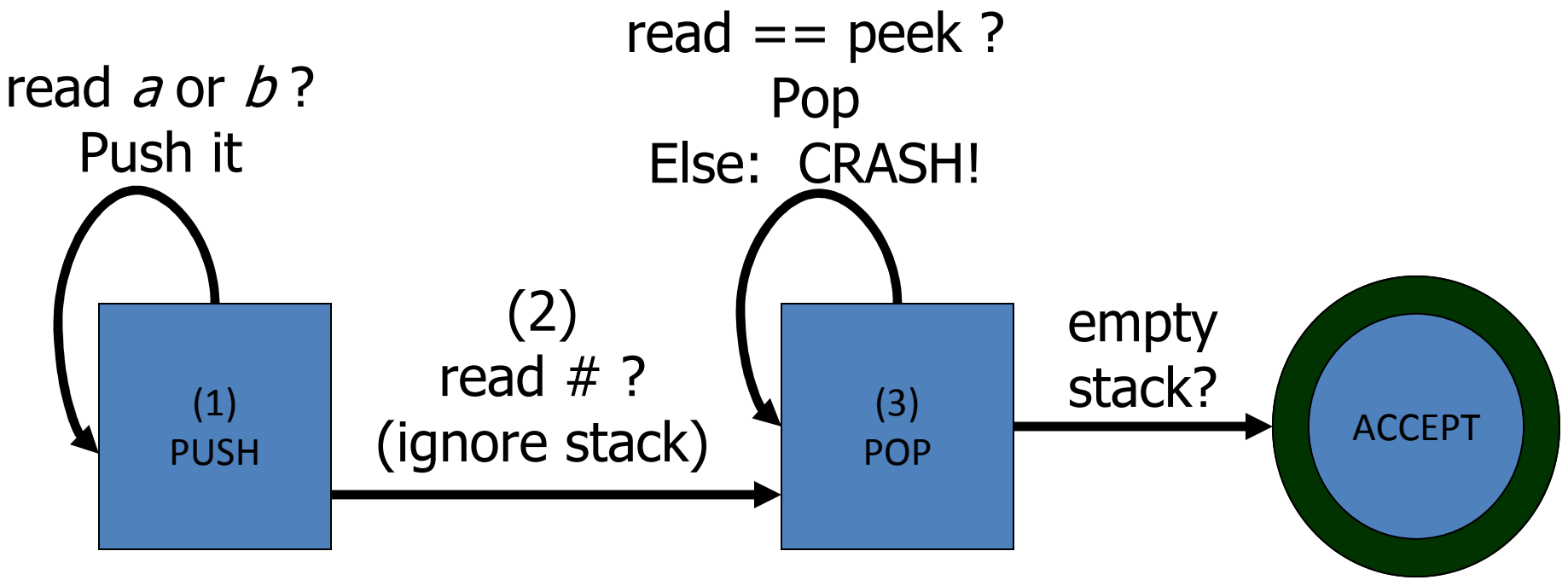


Input:

aaab# baaaa

Pause input

De CFG's para Máquinas de Pilha



Input:

aaab# baaaa

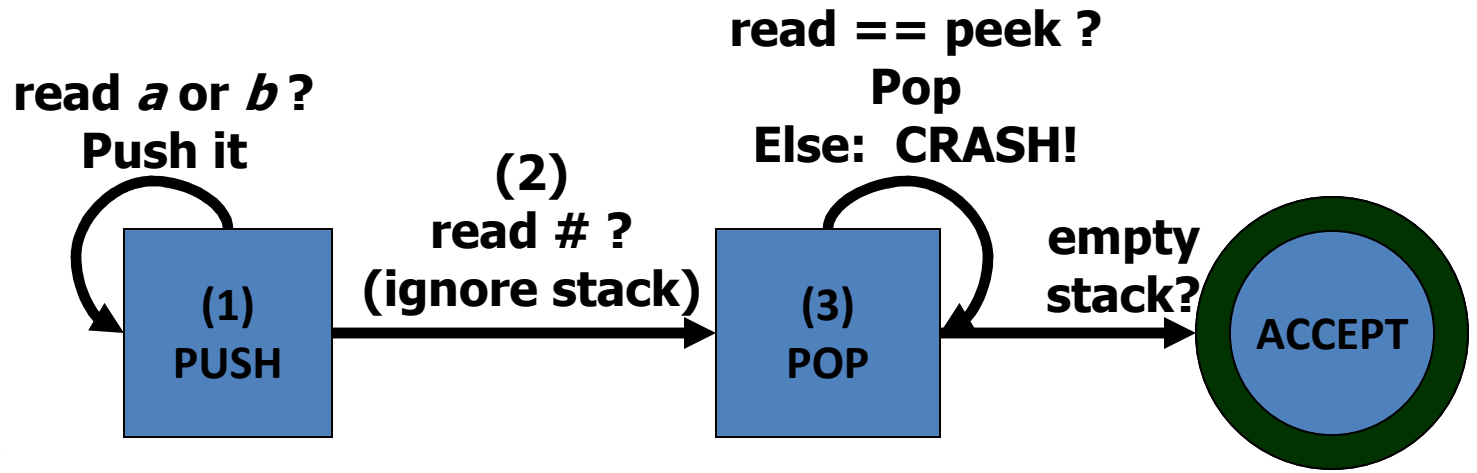
CRASH

PDA's à la Sipser

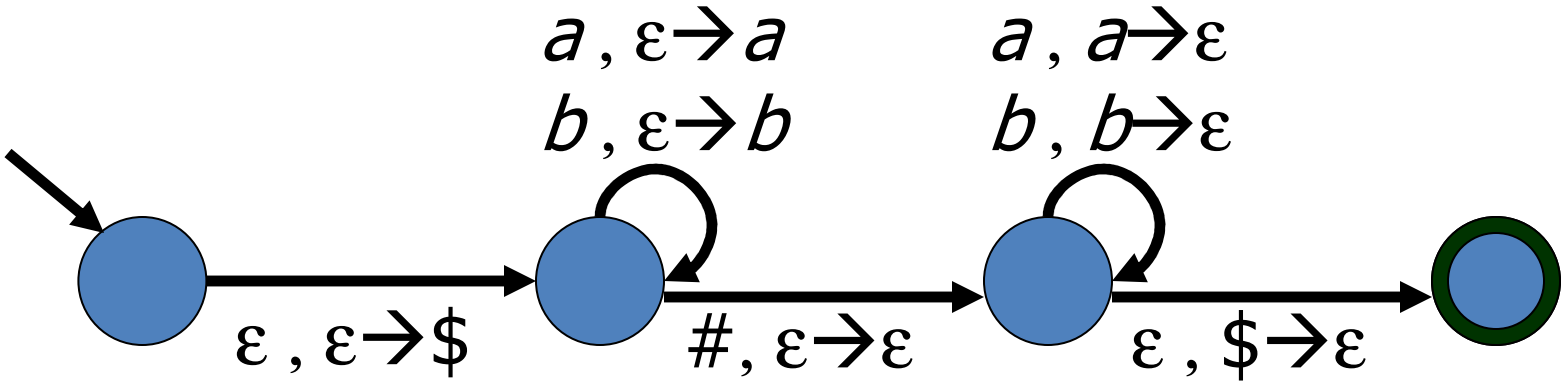
Para facilitar a análise, máquinas de pilha teóricas têm um conjunto restrito de operações. Cada livro-author tem sua própria versão. PDAs descritos em Sipser são assim:

- 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.*

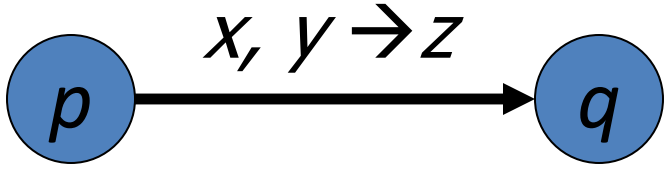
Versão Sipser's



Torna-se:



Versão Sipser's



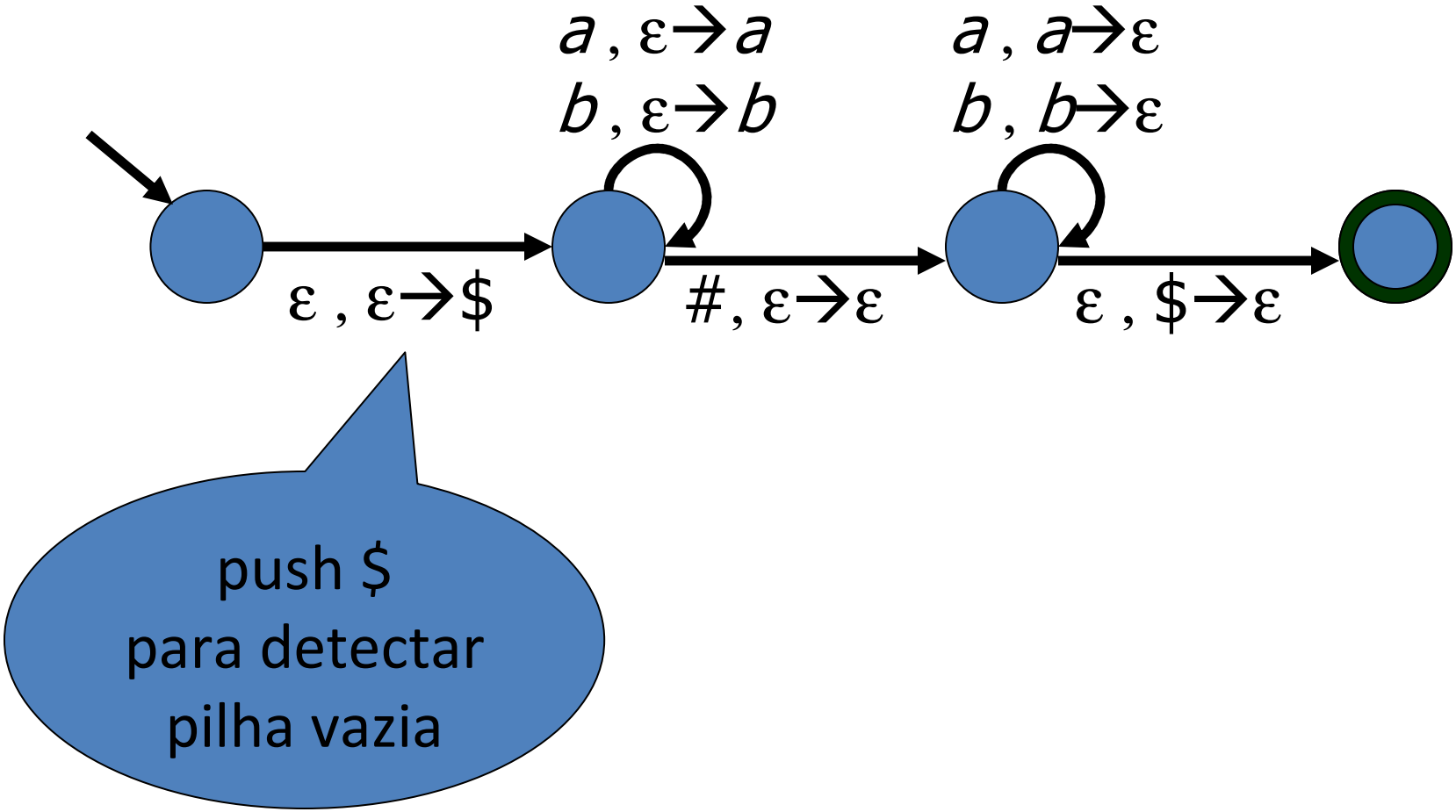
Significado da convenção do rótulo:

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

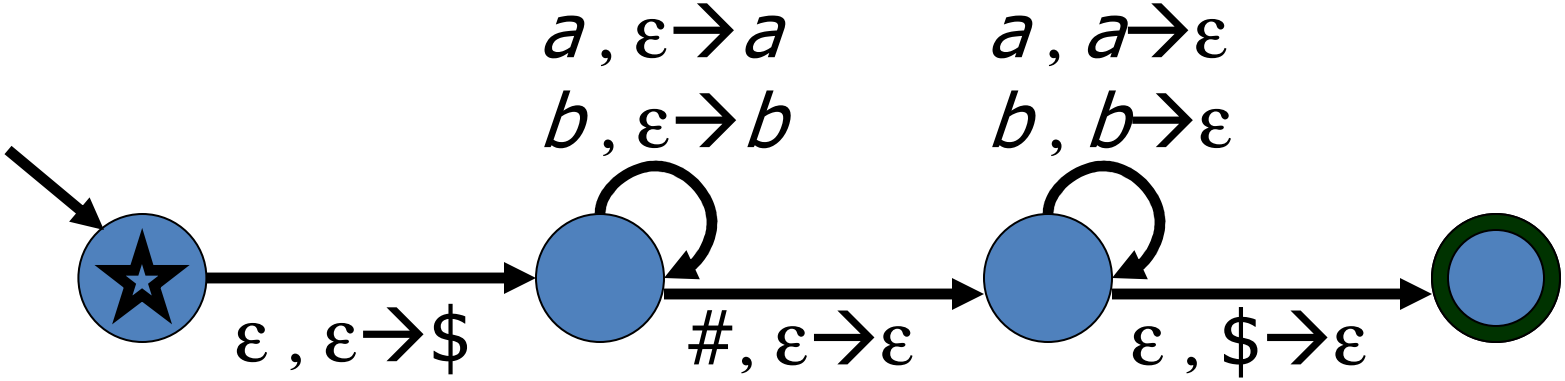
então vá para *q* **e** substitua *y* por *z* na pilha.

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

Versão Sipser's

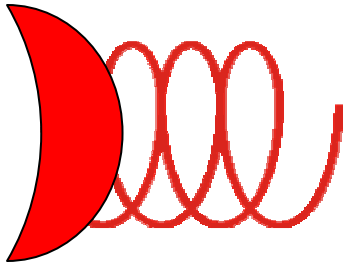


Versão Sipser's

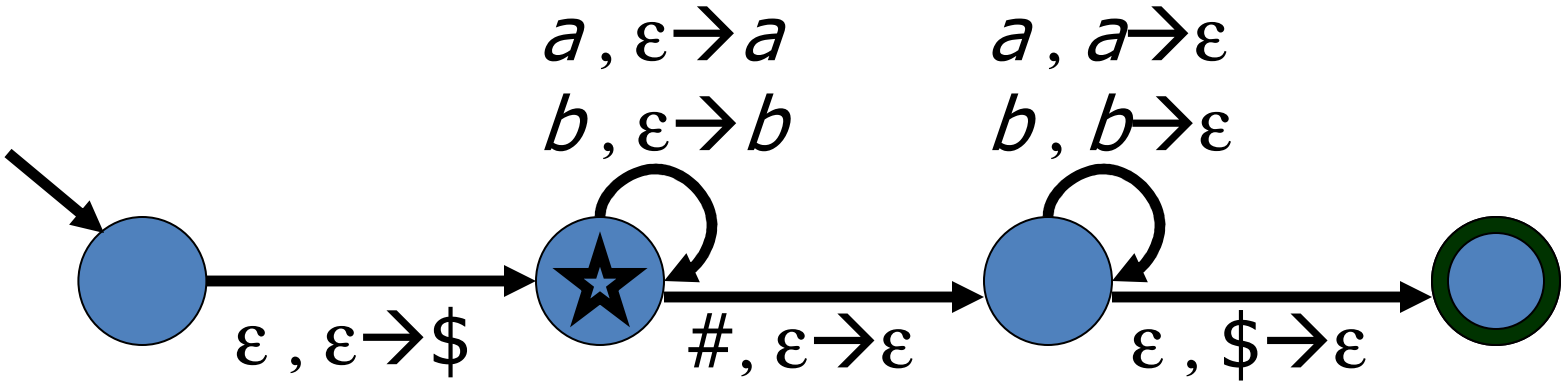


Input:

aaab# baaa

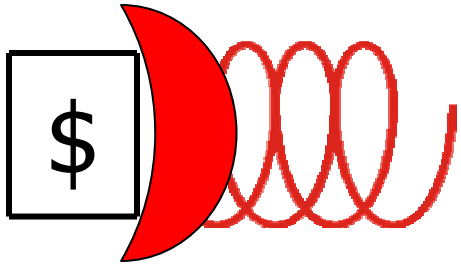


Versão Sipser's

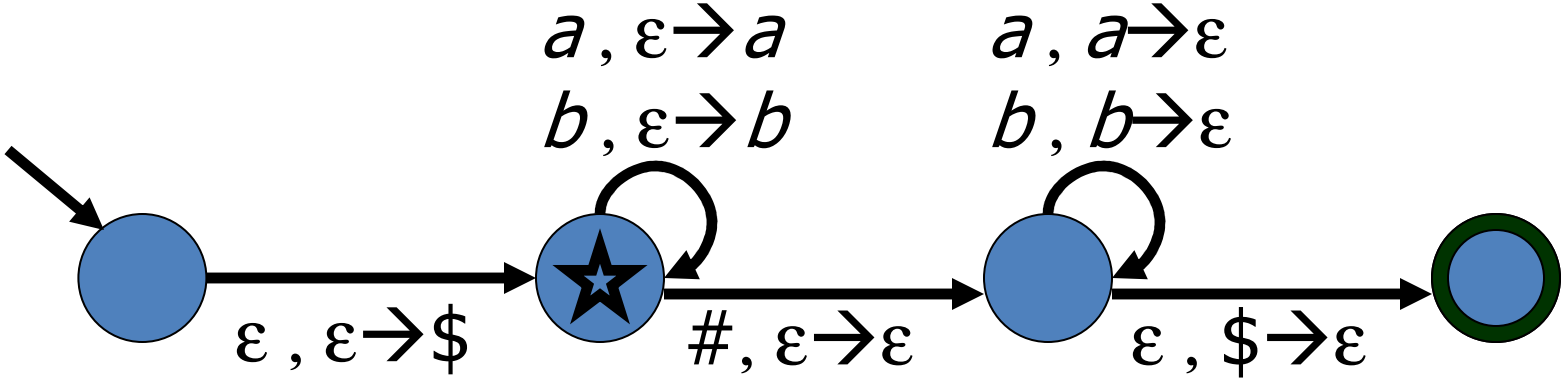


Input:

aaab# baaa

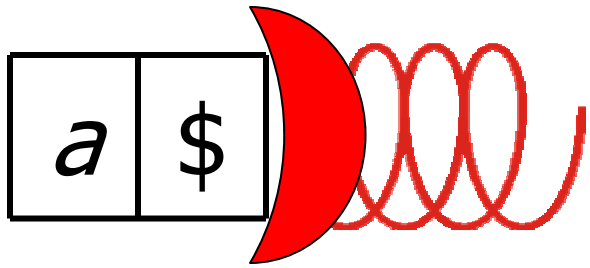


Versão Sipser's

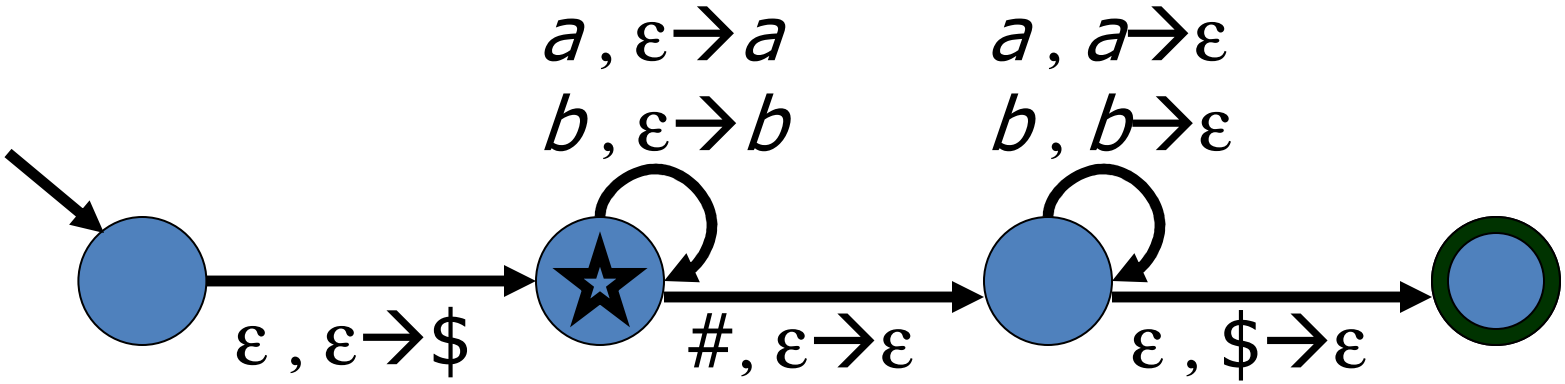


Input:

aaab# baaa

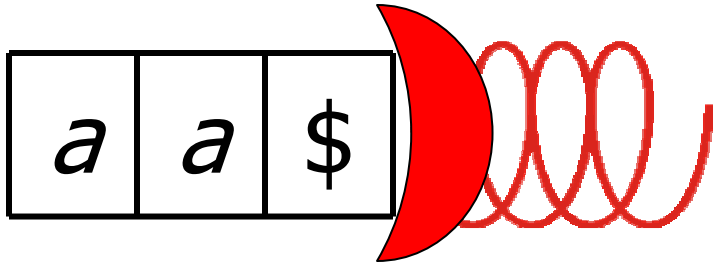


Versão Sipser's

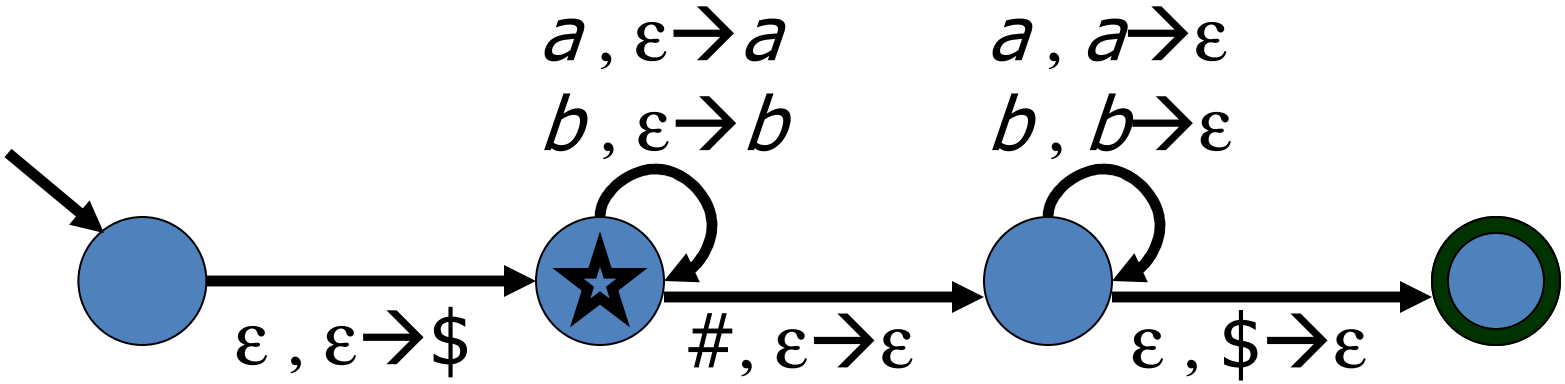


Input:

aaab# baaa

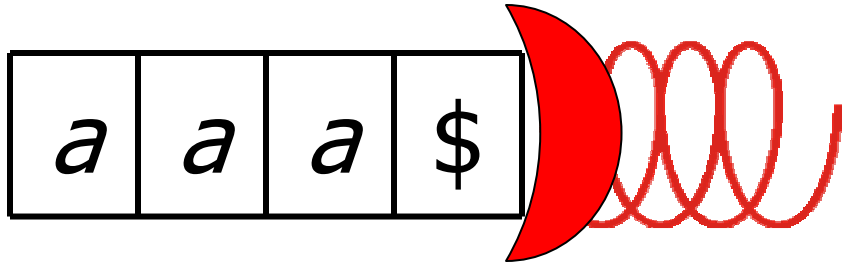


Versão Sipser's

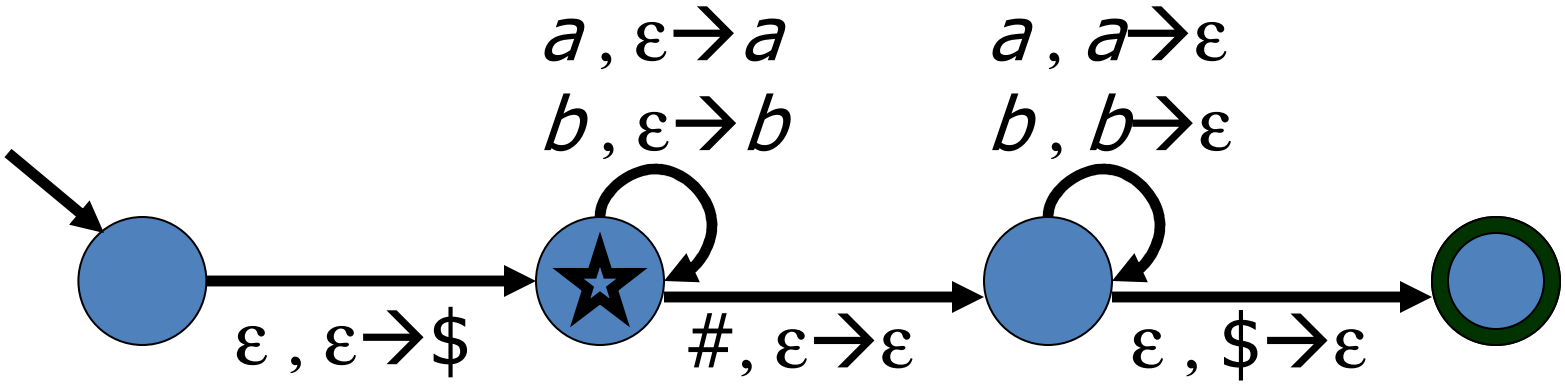


Input:

aaab# baaa

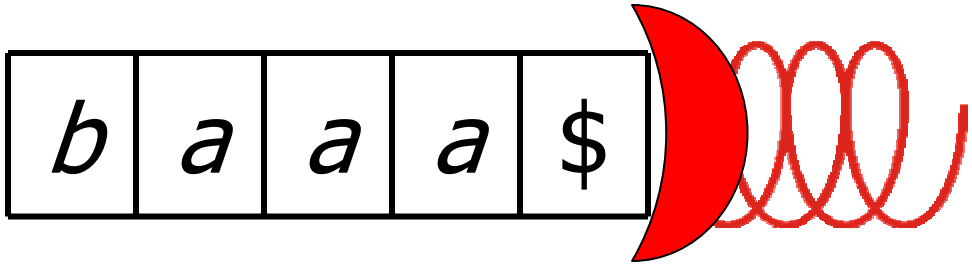


Versão Sipser's

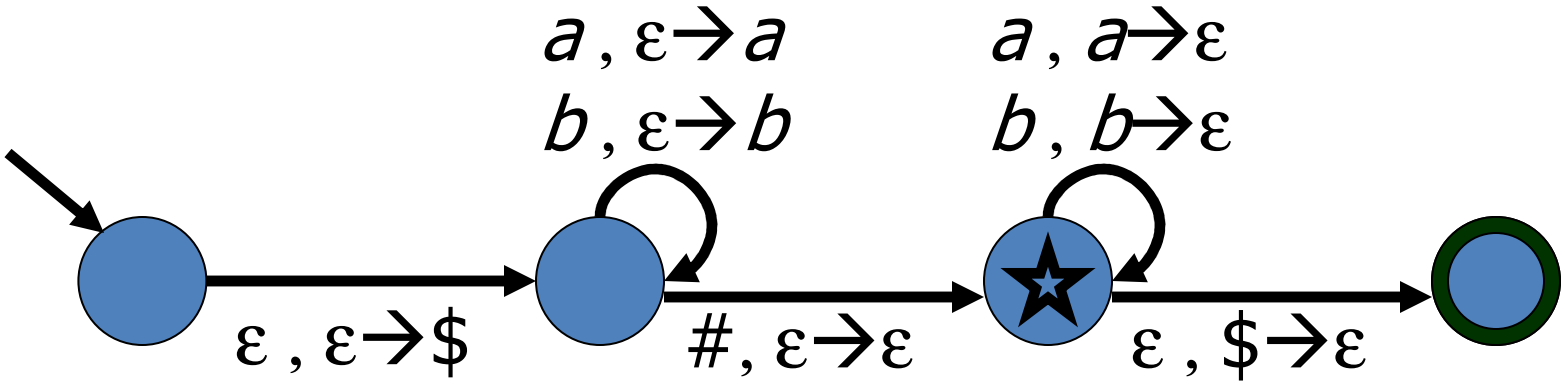


Input:

aaab# baaa

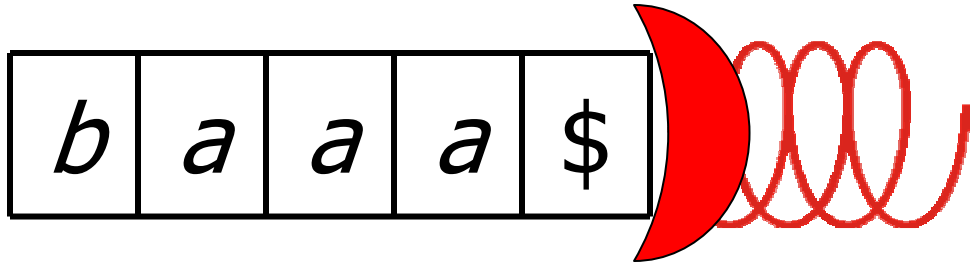


Versão Sipser's

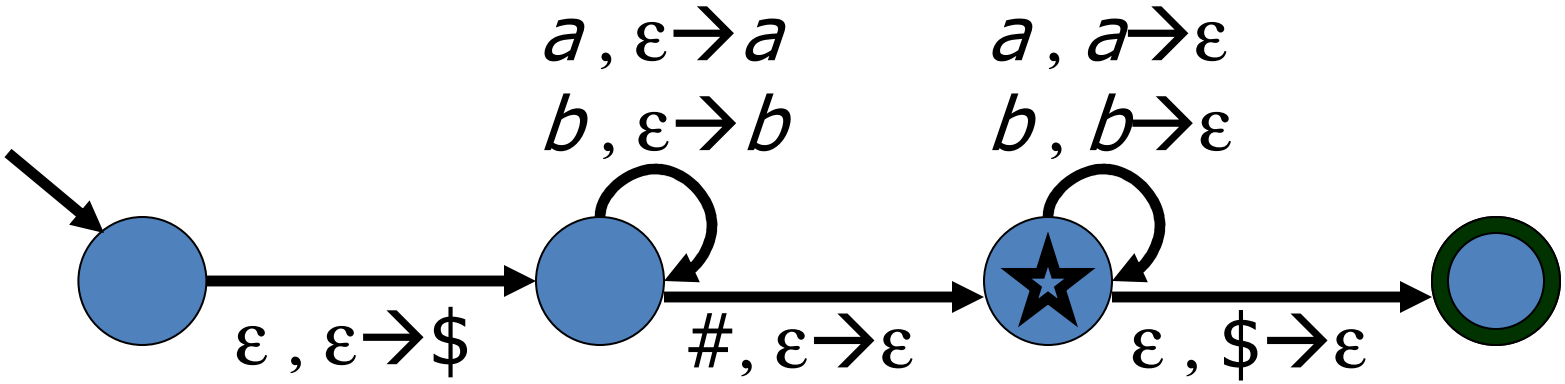


Input:

aaab# baaa

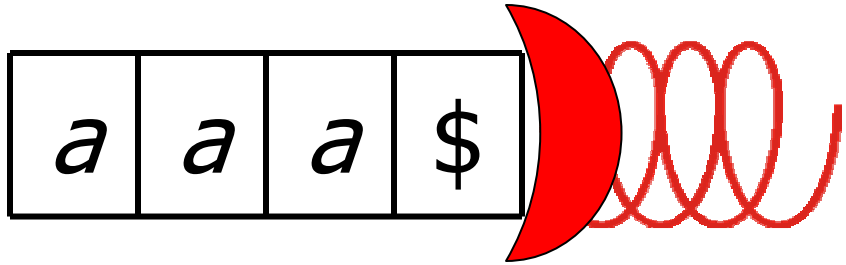


Versão Sipser's

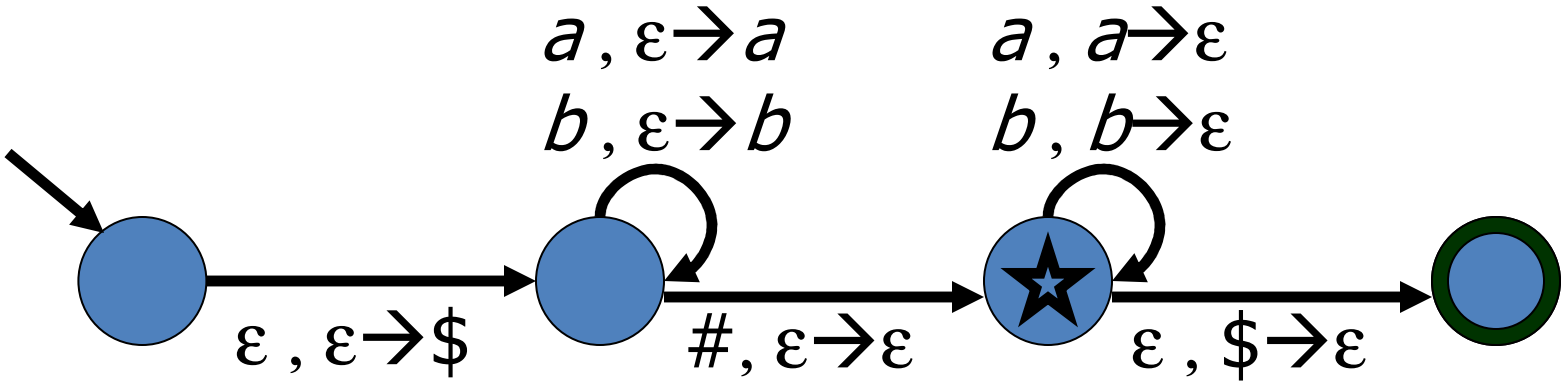


Input:

aaab# baaa

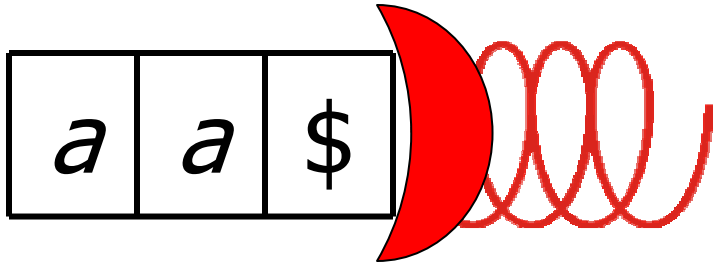


Versão Sipser's

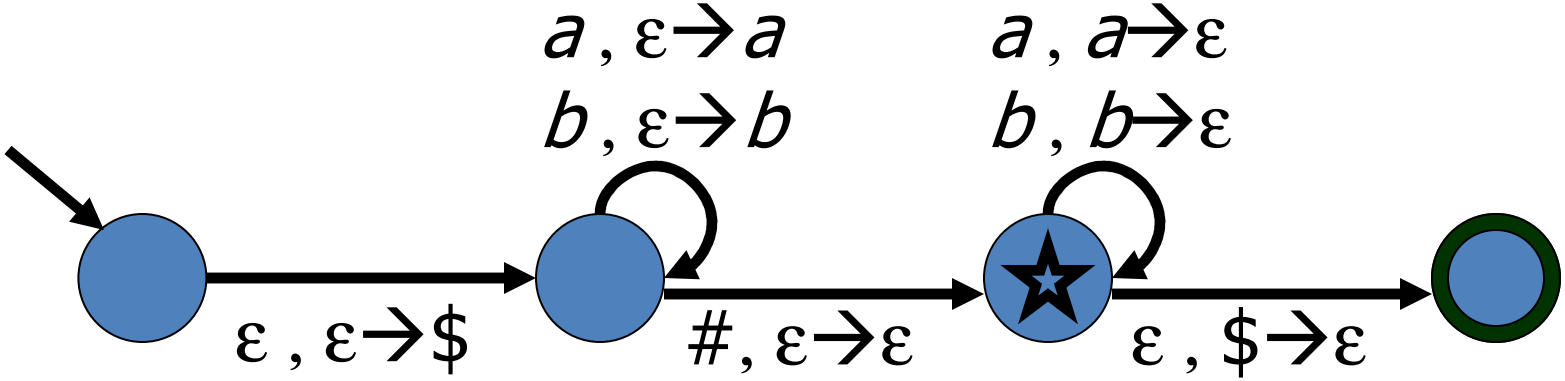


Input:

aaab# baaa

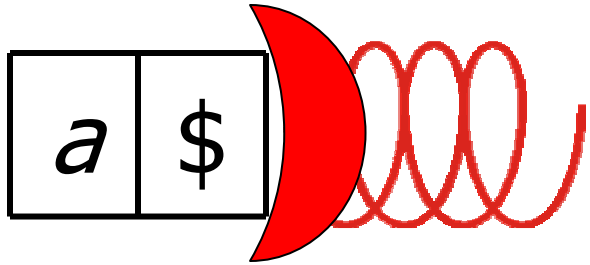


Versão Sipser's

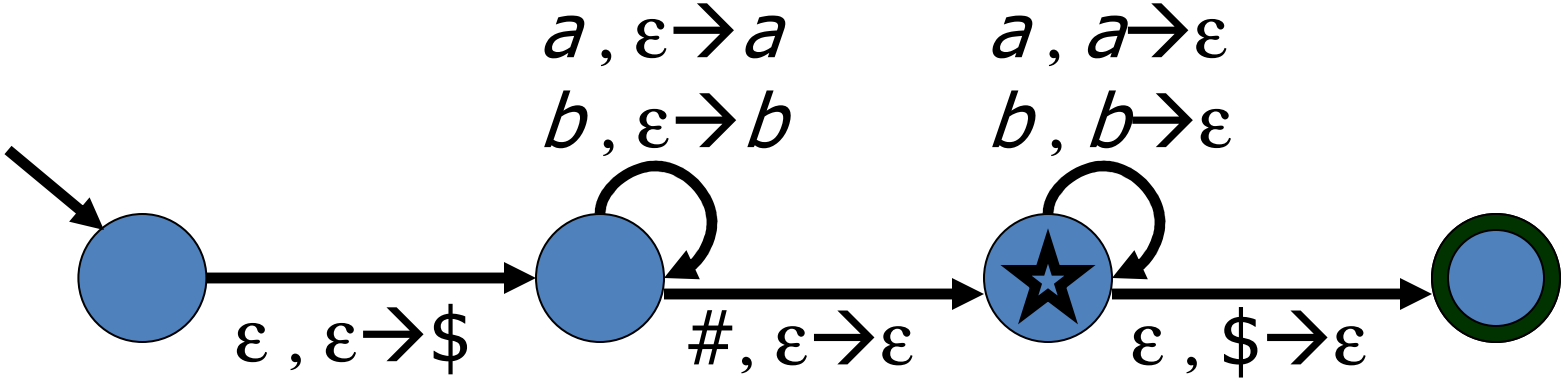


Input:

aaab# baaa

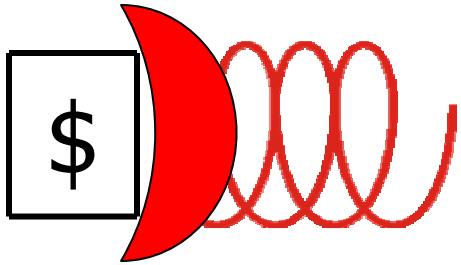


Versão Sipser's

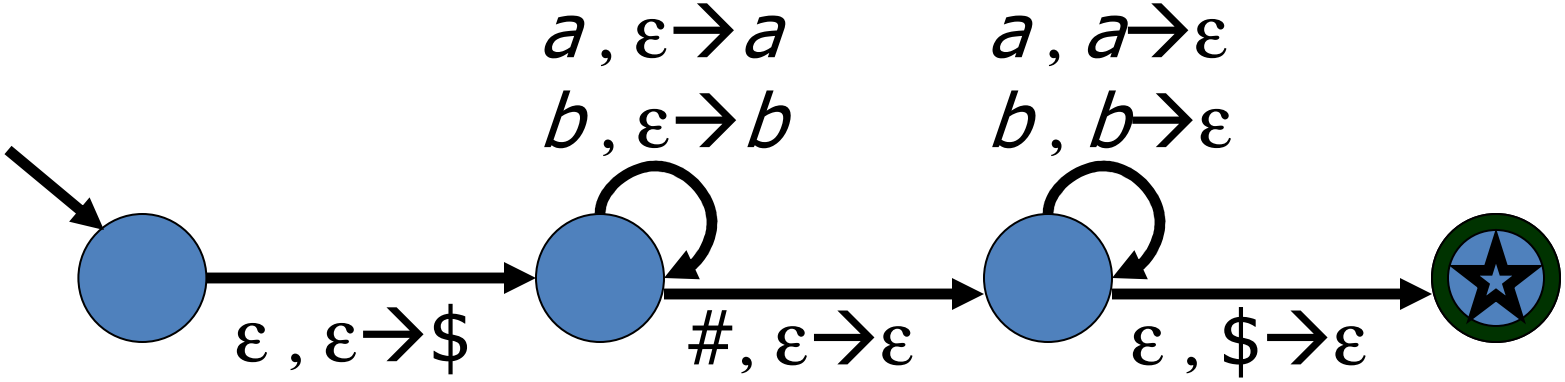


Input:

aaab# baaa



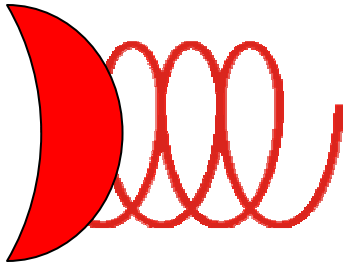
Versão Sipser's



Input:

aaab# baaa

ACCEPT!



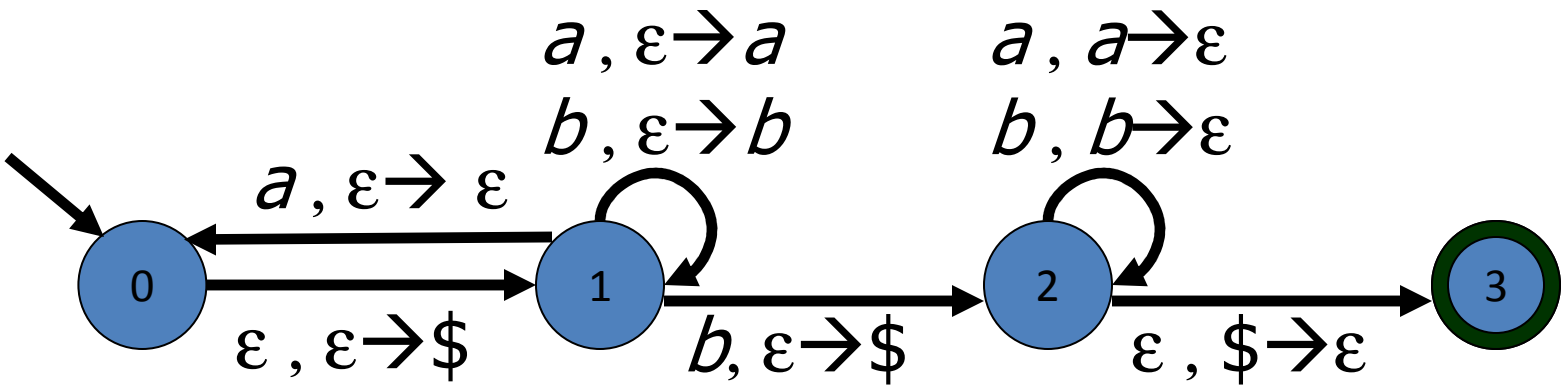
PDA
Definição Formal

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*.

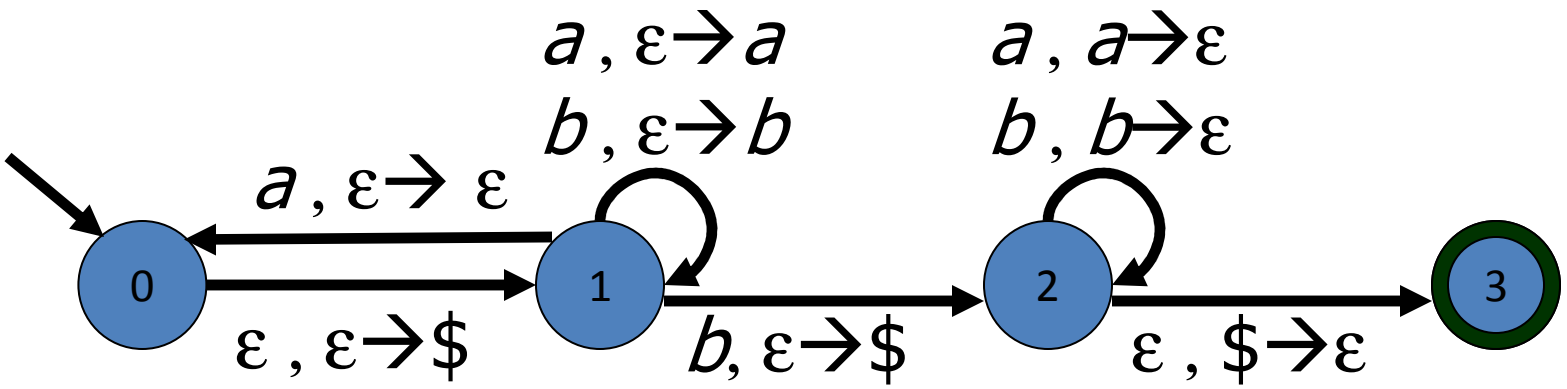
PDA
Definição Formal



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

(Sipser 2.6.a) Desenhe um PDA que aceite a linguagem

$L = \{ x \in \{a,b\}^* \mid n_a(x) = 2n_b(x) \}$

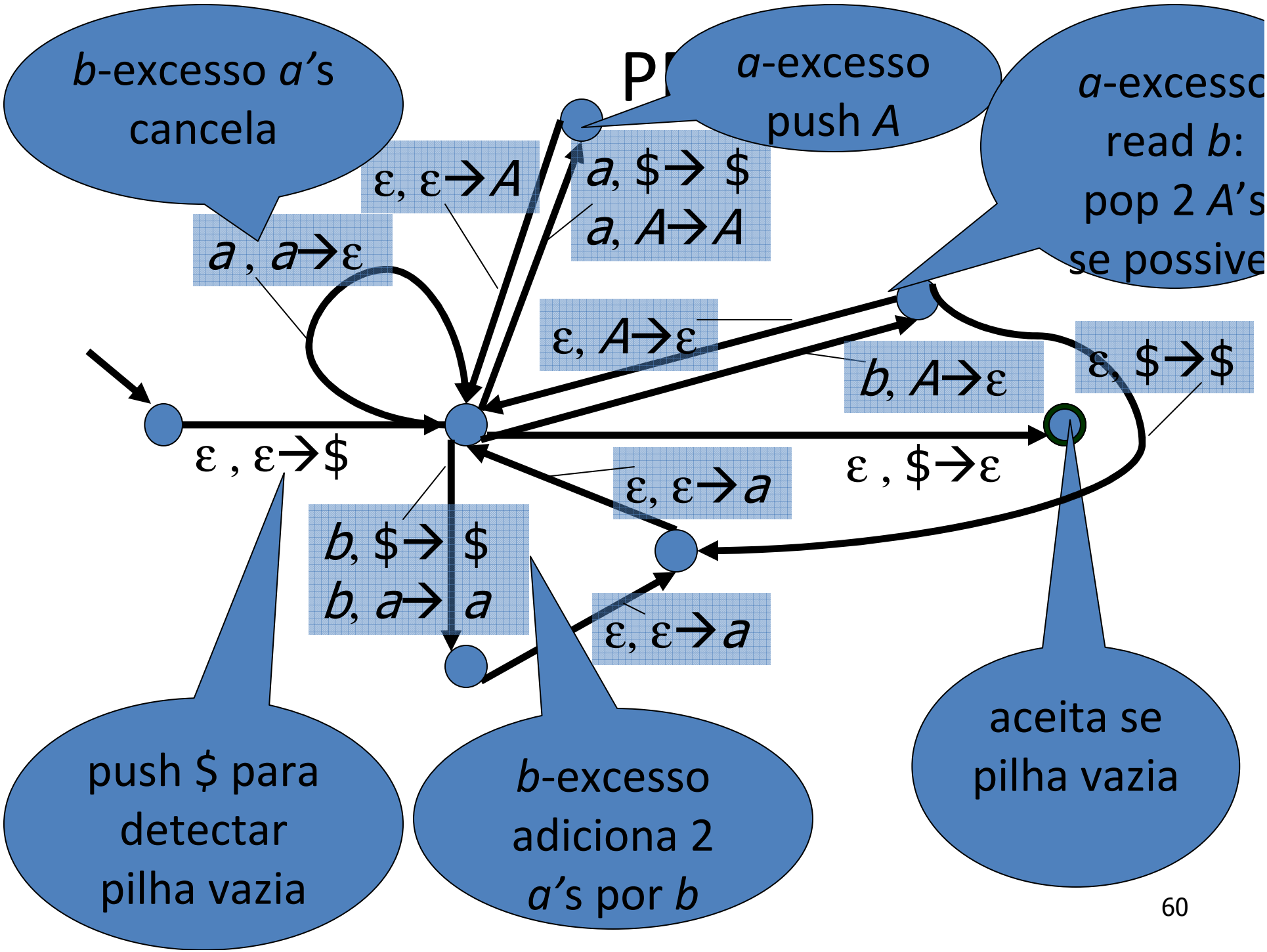
NOTA: O string vazio está em L.

PDA - Exercício

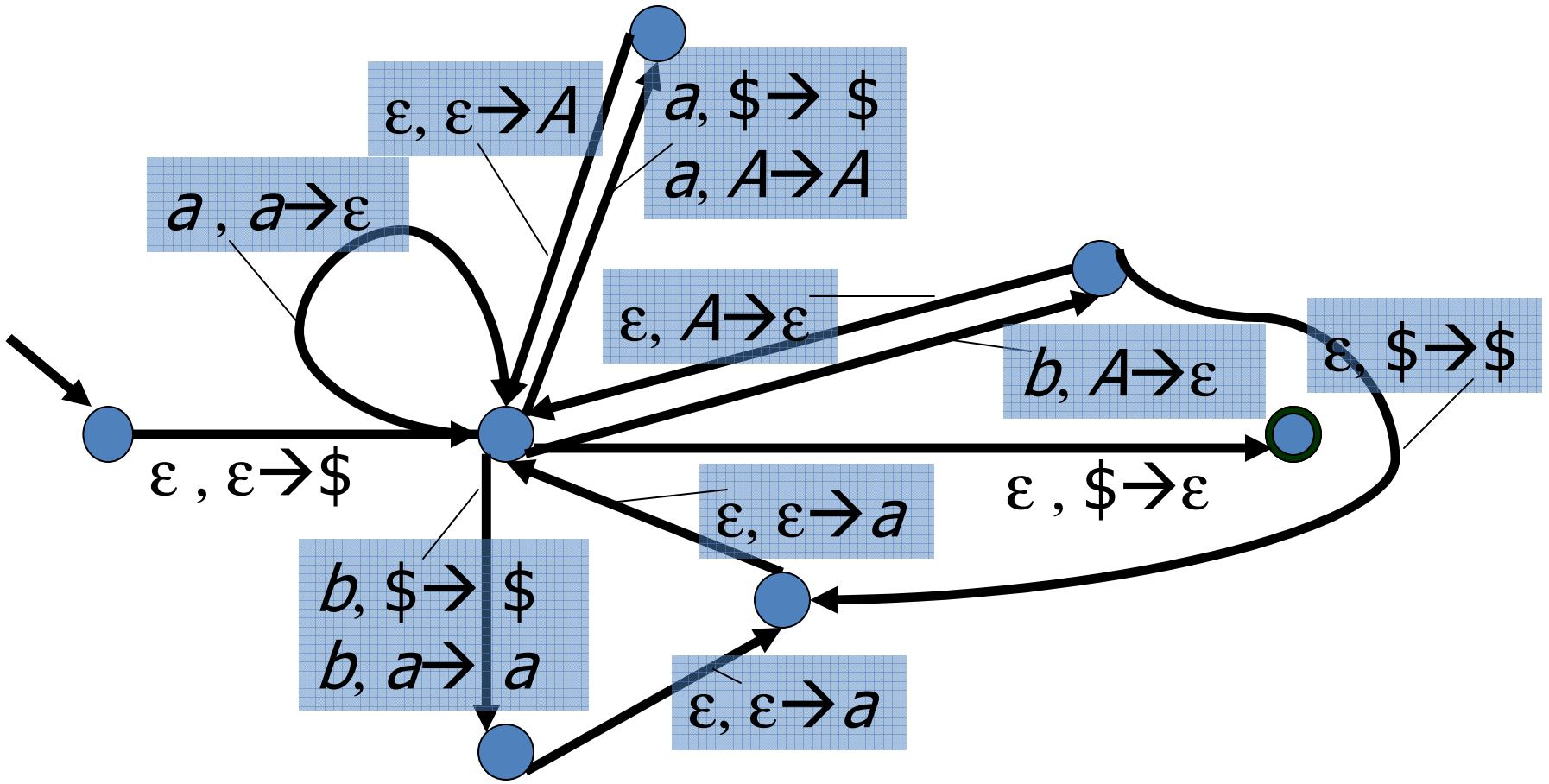
- Forneça PDA´s para reconhecer as seguintes linguagens:
  - $\{a^n b^n \mid n \in \aleph\}$
  - $\{a^n b^{2n} \mid n \in \aleph\}$
  - $\{a^{2n} b^n \mid n \in \aleph\}$
  - $\{a^i b^j c^k \mid i = j \text{ ou } i=k\}$
  - $\{w\, w^R \mid w \in \{0,1\}^*\}$
  - $\{w \in \{0,1\}^* \mid$ o número de 0´s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

PDA Exercício

A idéia é usar a pilha para "contar'' o número de *a'*s e/ou *b'*s necessários para obter um string válido. Se tiverem sido lidos *b*'s em excesso, a pilha deverá ter um número correspondente de *a'*s (dois para cada *b*). Por outro lado, para cada *a* lido em excesso não podemos ½ *b*… Ao invés disso, para cada *a* em excesso, empilhamos 2 "a-negativo", sendo um a-negativo representado pelo símbolo *A.* Vejamos como isso funciona:


b-excesso a's cancela
a-excesso push A
a-excesso read b: pop 2 A's se possive
ε, ε→A
a, $→ $
a, A→A
a , a→ε
ε, A→ε
b, A→ε
ε, $→$
ε , ε→$
ε , $→ε
ε, ε→a
b, $→ $
b, a→ a
ε, ε→a
push $ para detectar pilha vazia
b-excesso adiciona 2 a's por b
aceita se pilha vazia

PDA Exemplo.
Sem bolhas



Equivalência entre PDA e CFG

CFG → PDA

Gramáticas Lineares à direita podem ser convertidas para NFA's. Em geral, CFG's podem ser convertidas para PDA's.

Em "NFA → REX" foi útil considerar GNFA's como estágio intermediário. De modo similar, será útil considerar aqui PDA's Generalizados.

PDA's Generalizados

Um ***PDA Generalizado*** (GPDA) é como um PDA, exceto que permite substituir o topo da pilha por um string, e não apenas um caractere ou o string vazio. É fácil converter um GPDA para um PDA: basta substituir cada push de um string por uma sequência de push's simples.

CFG → PDA
Exemplo

Converta a gramática

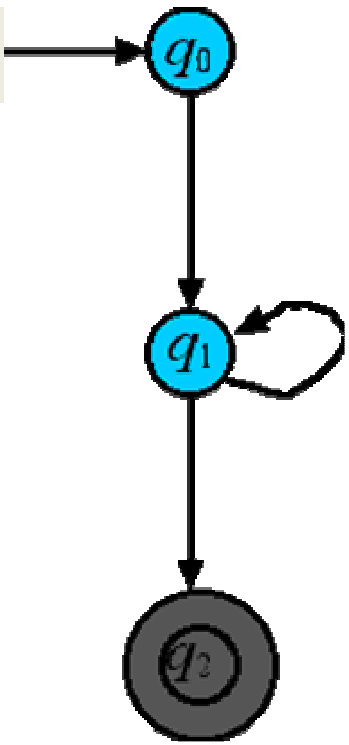
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

em um PDA. A idéia é simular as derivações gramaticais com o PDA.

CFG → PDA
Exemplo

Comece com três estados para o GPDA:

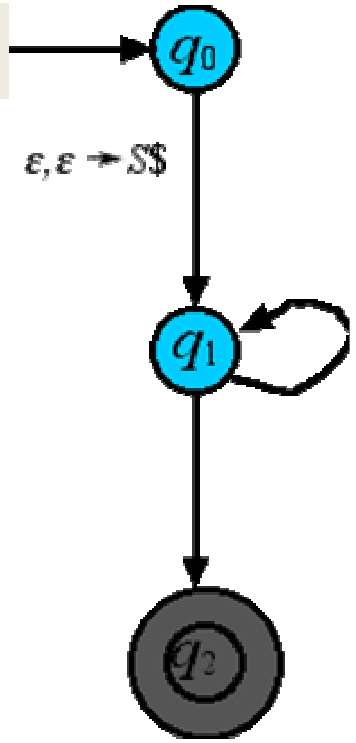
S → ε |*a* | *b* | *aSa* | *bSb*



CFG → PDA
Exemplo

A primeira transição empilha *S*$ de modo que se possa testar pilha vazia ($), e também iniciar a simulação (*S*).

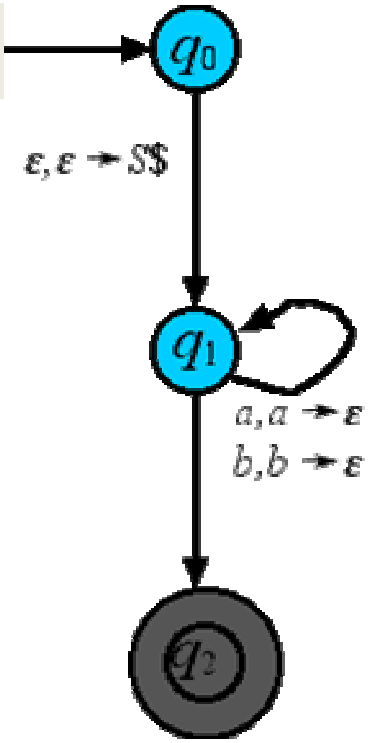
S → ε |*a* | *b* | *aSa* | *bSb*



CFG → PDA
Exemplo

Possibilite reading/popping de terminais de modo que se possa ler qualquer string de terminais gerado.

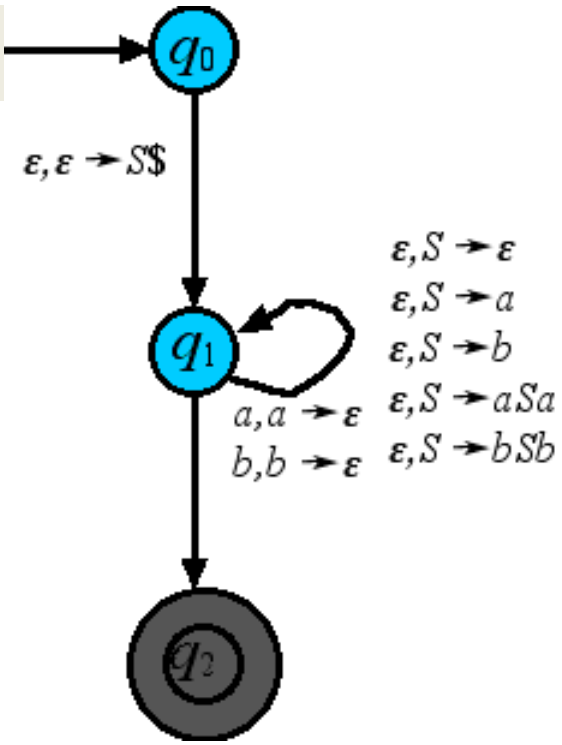
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



CFG → PDA
Exemplo

Simule todas as productions adicionando transições sem leitura.

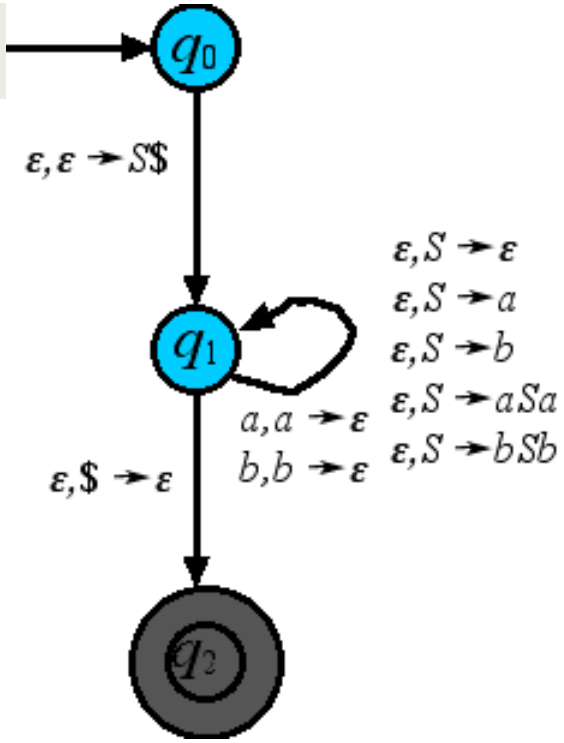
S → ε |*a* | *b* | *aSa* | *bSb*



CFG → PDA
Exemplo

Pop $ para aceitar quando a pilha está vazia (não deve haver mais nenhuma variável e todos os terminais devem ser lidos).

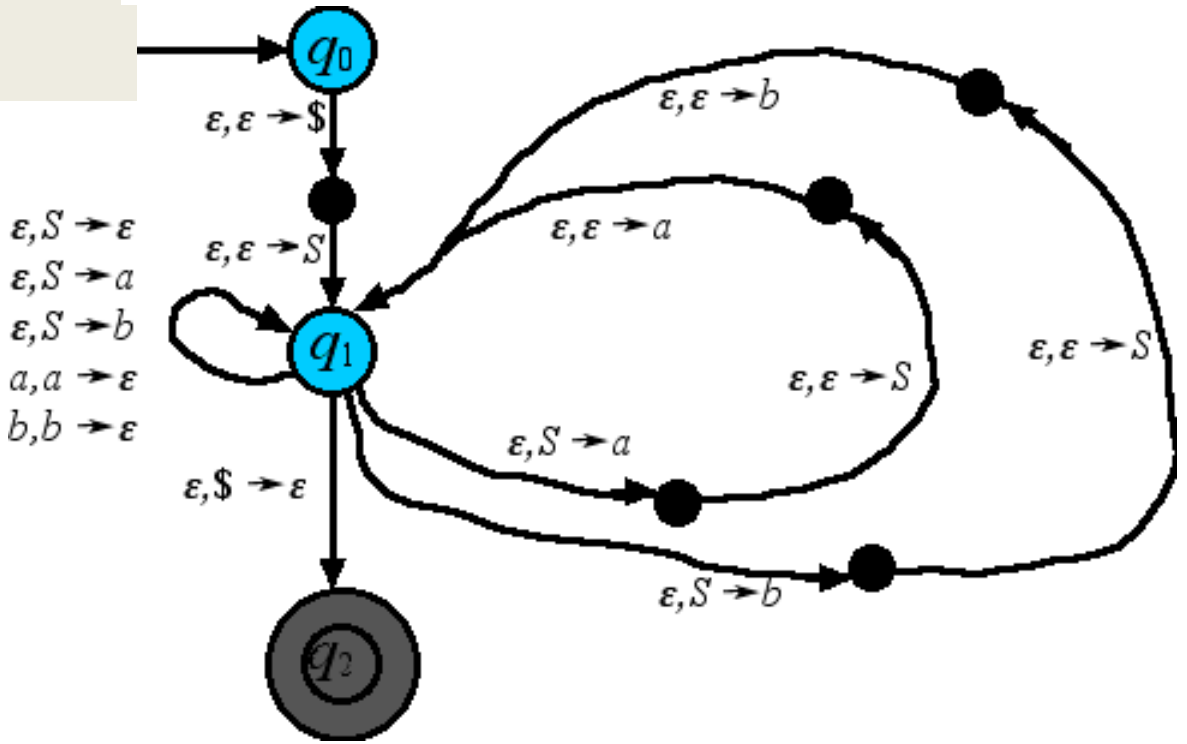
S → ε |*a* | *b* | *aSa* | *bSb*



CFG → PDA
Exemplo

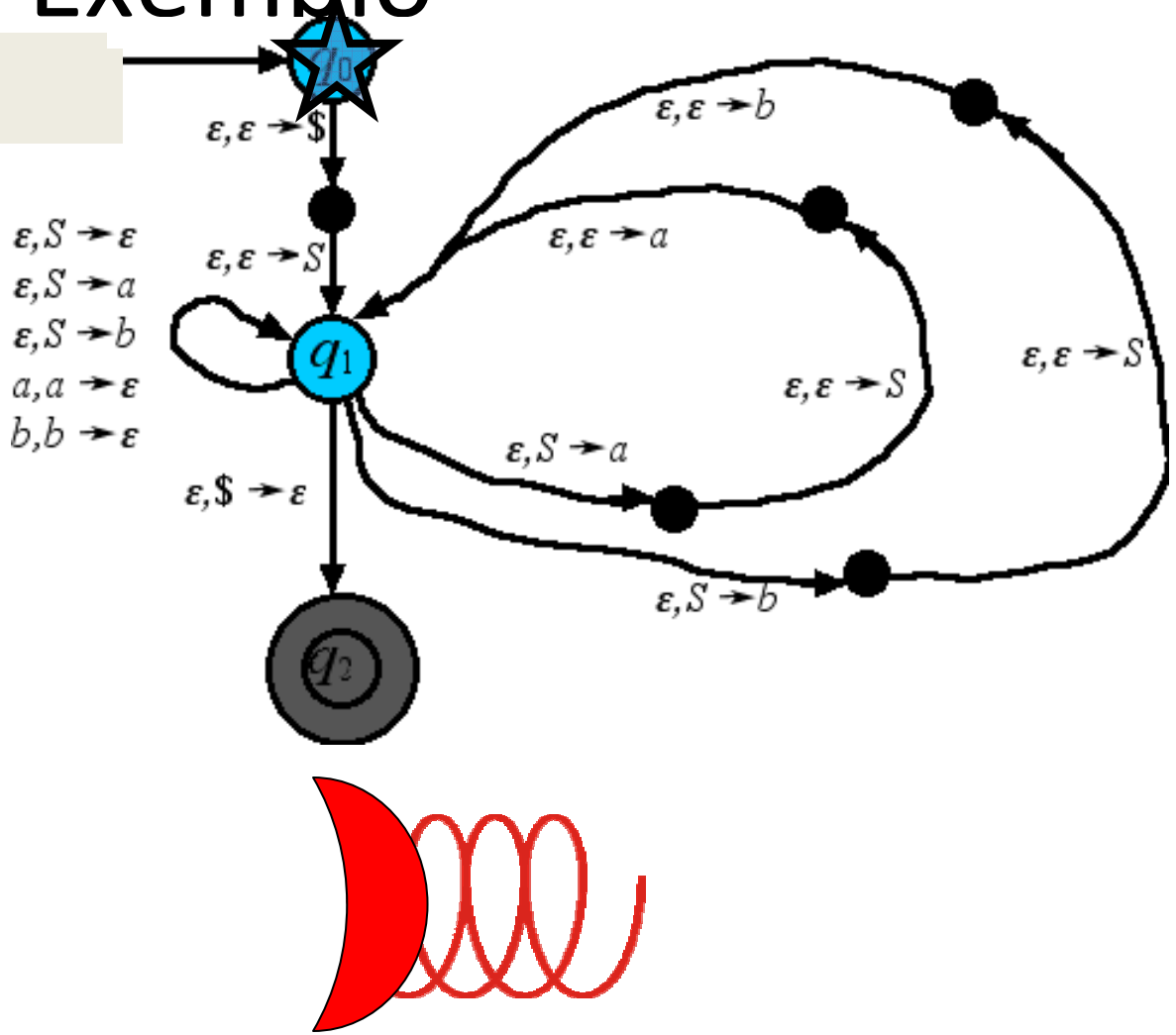
Converta o GPDA em um PDA usual, dividindo push's compostos.

S → ε |*a* | *b* | *aSa* | *bSb*



CFG → PDA
Exemplo

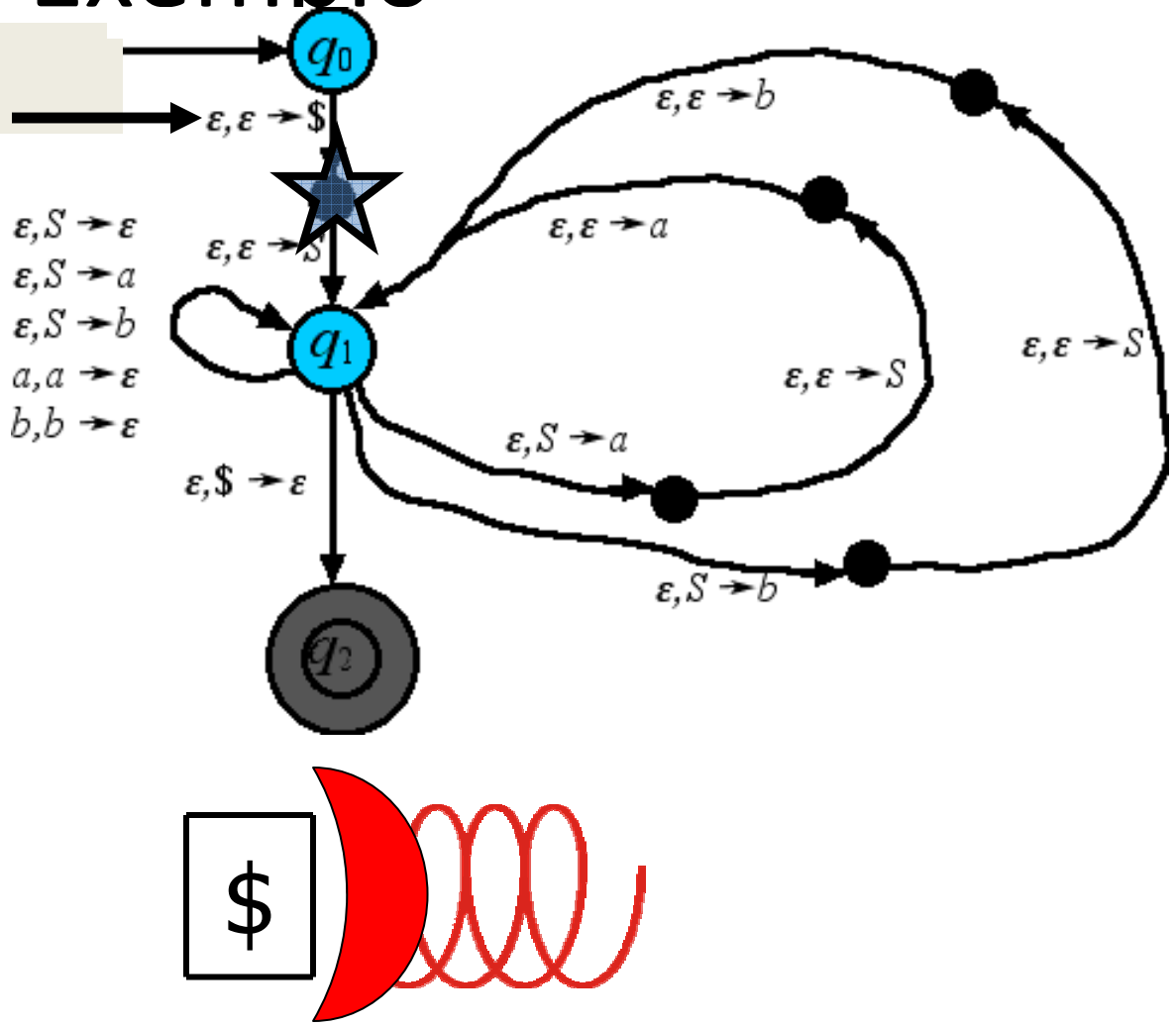
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



bbaabb

CFG → PDA
Exemplo

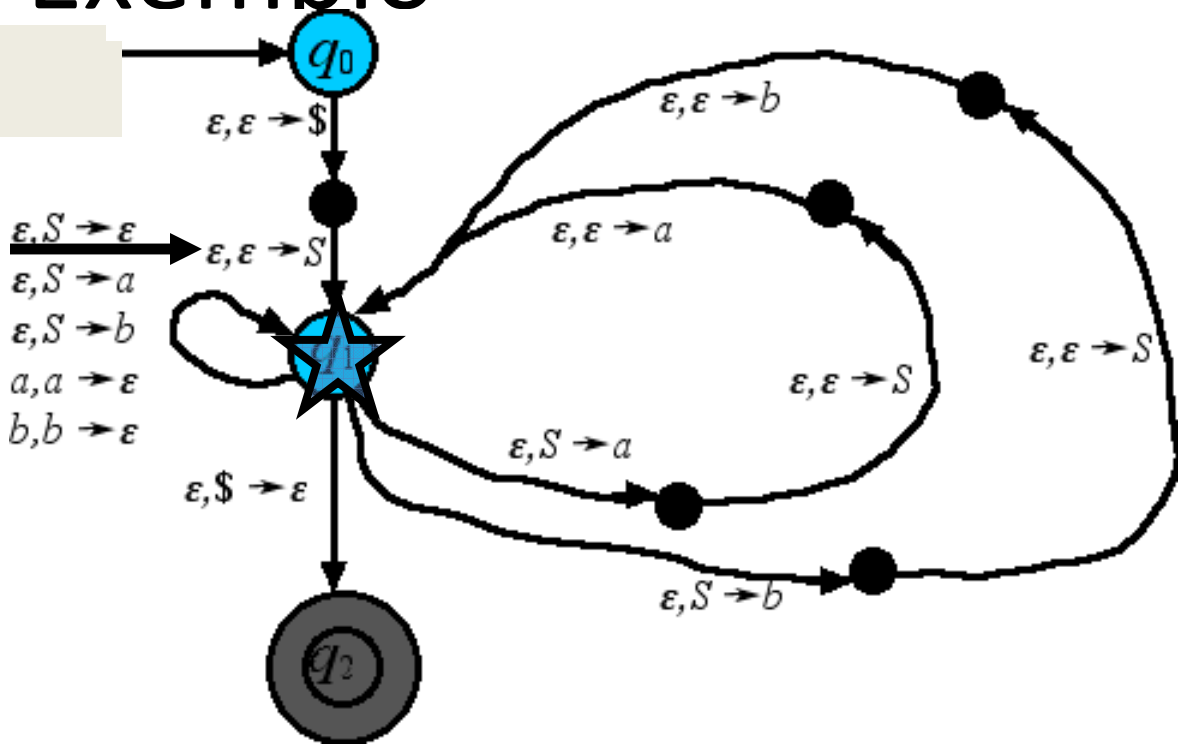
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



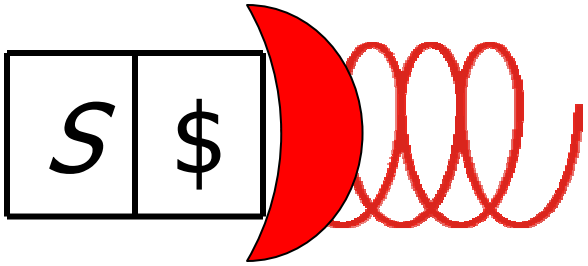
bbaabb

CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



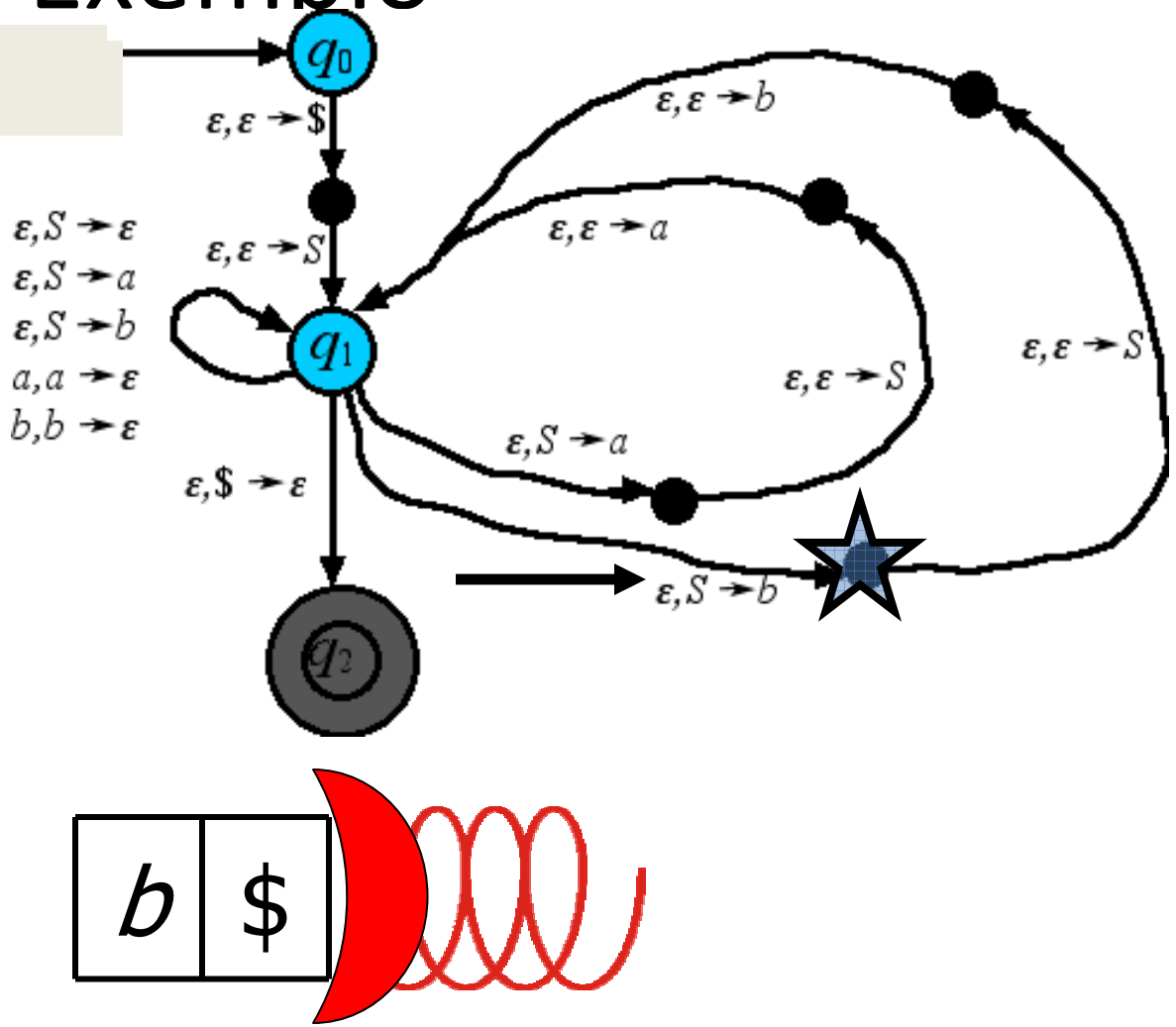
bbaabb



CFG → PDA
Exemplo

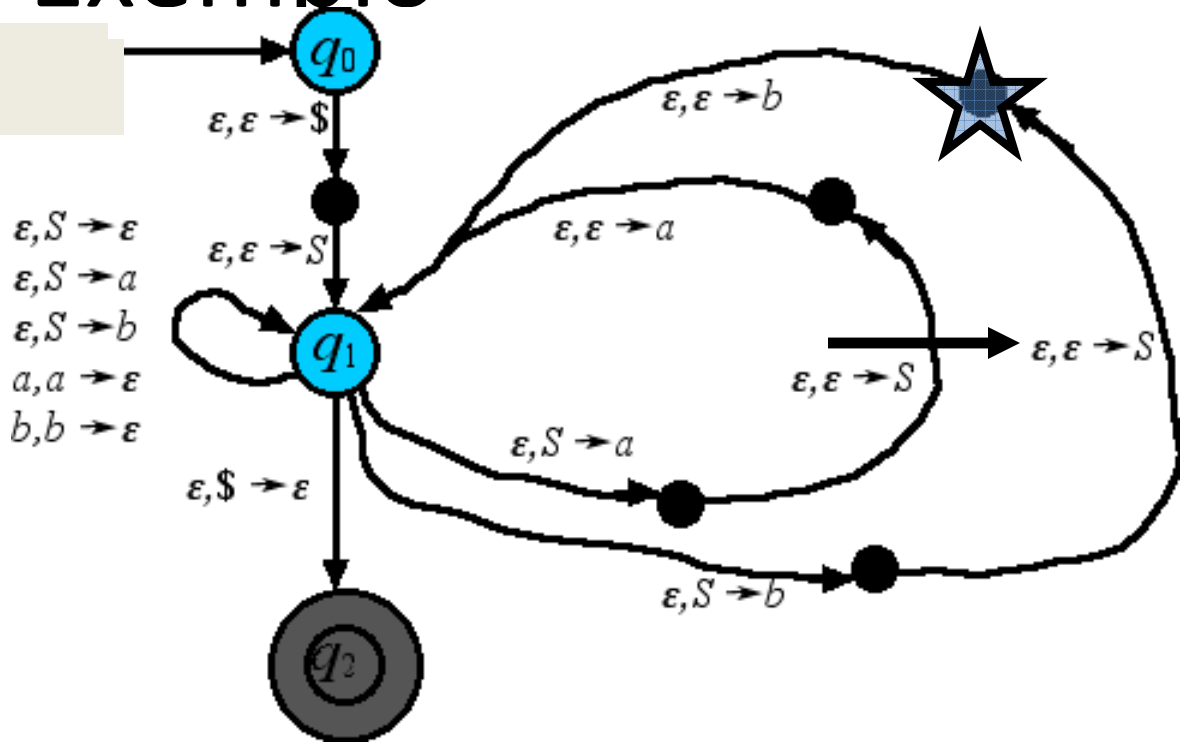
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

bbaabb

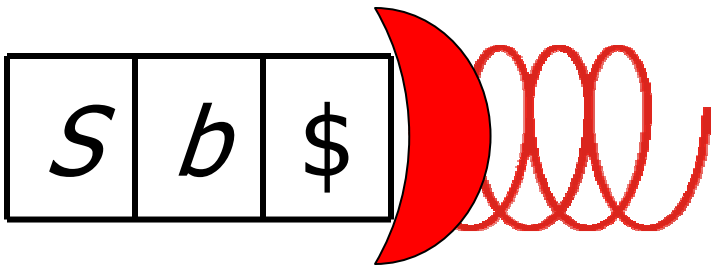


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

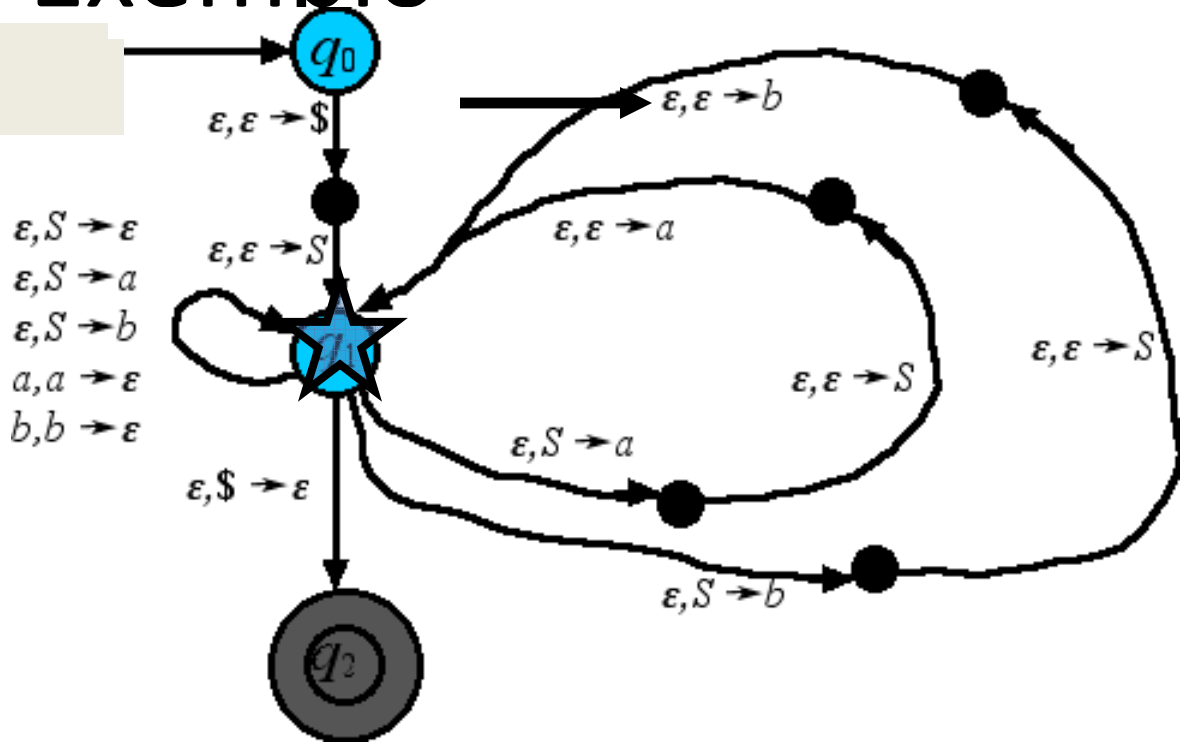


bbaabb

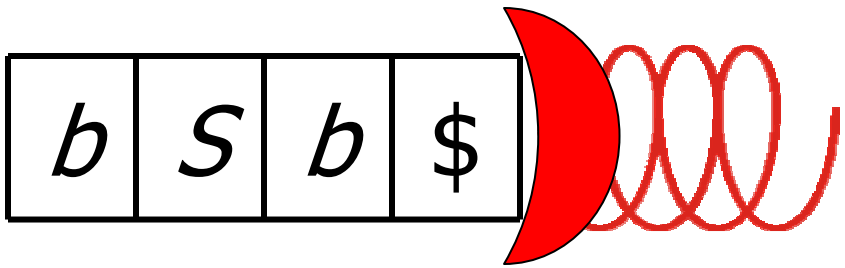


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

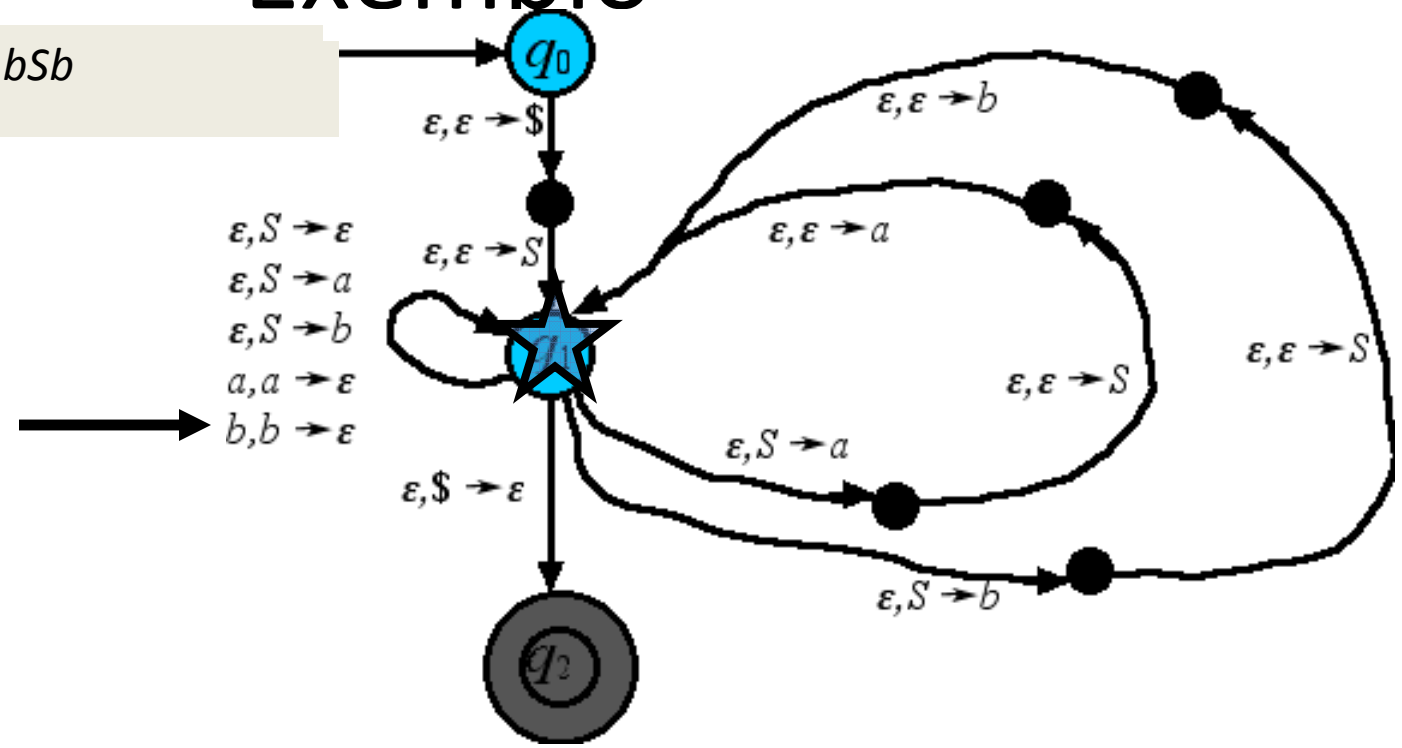


bbaabb

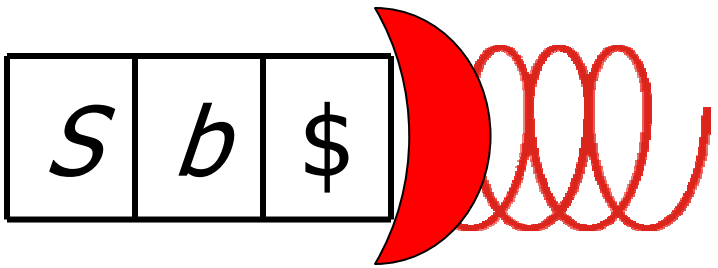


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



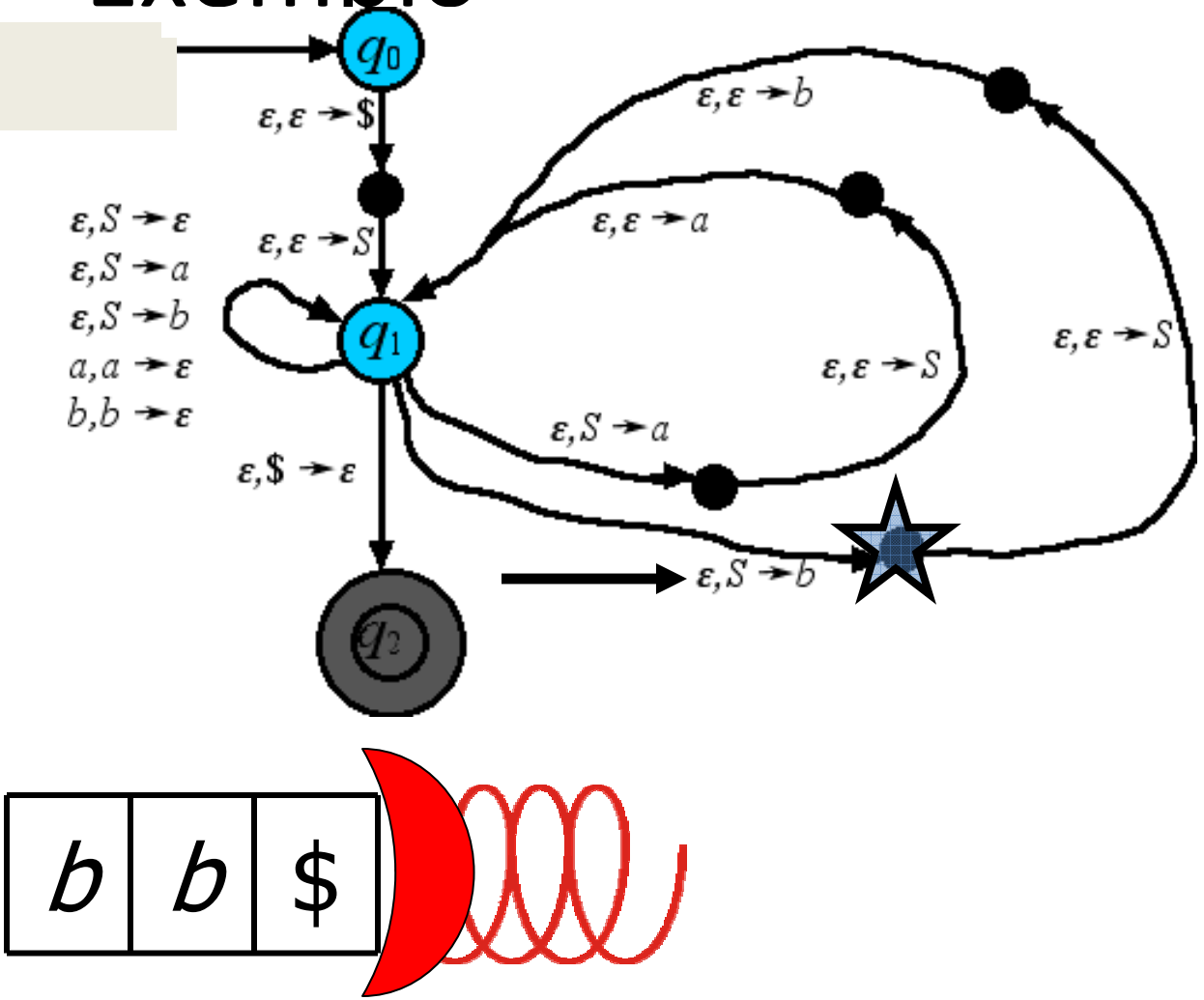
bbaabb



CFG → PDA
Exemplo

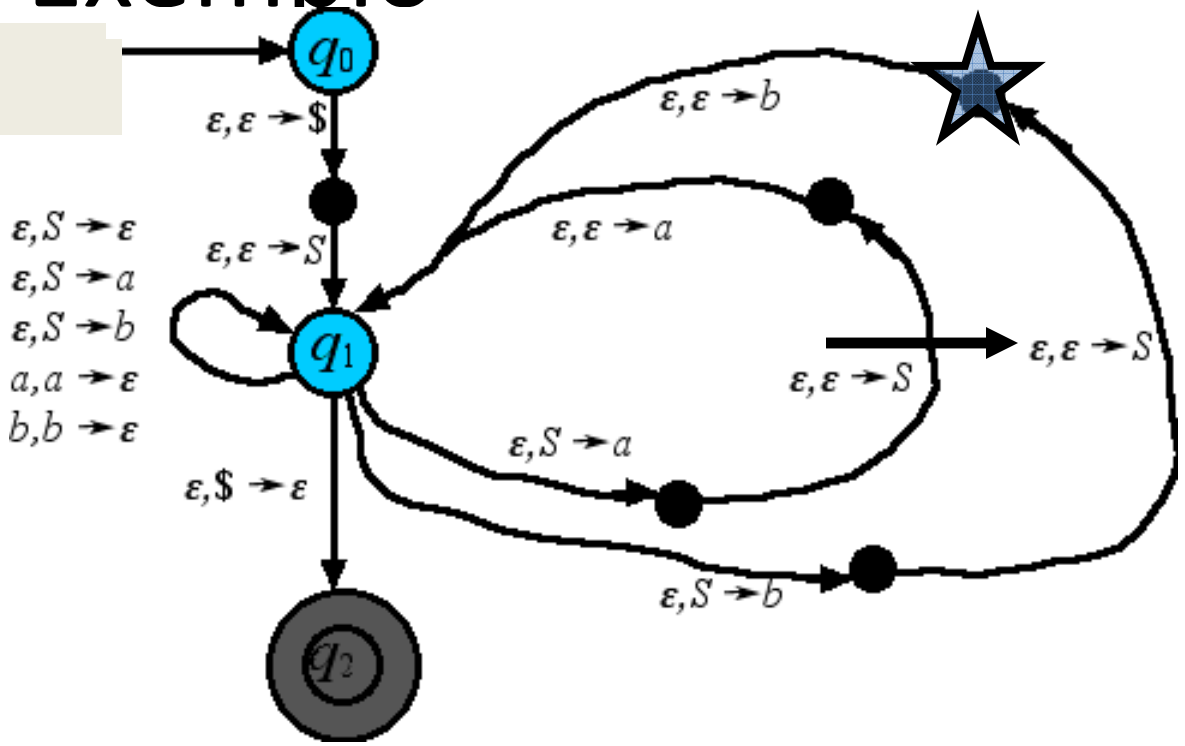
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

bbaabb

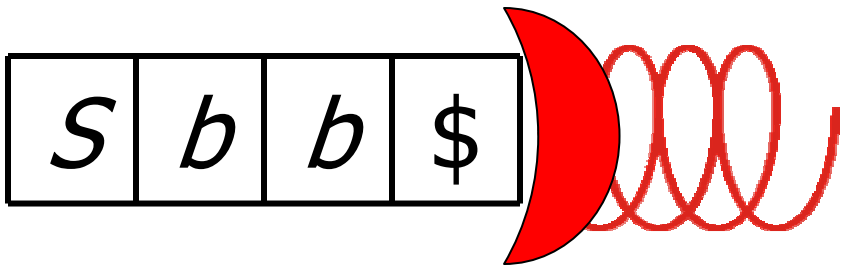


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

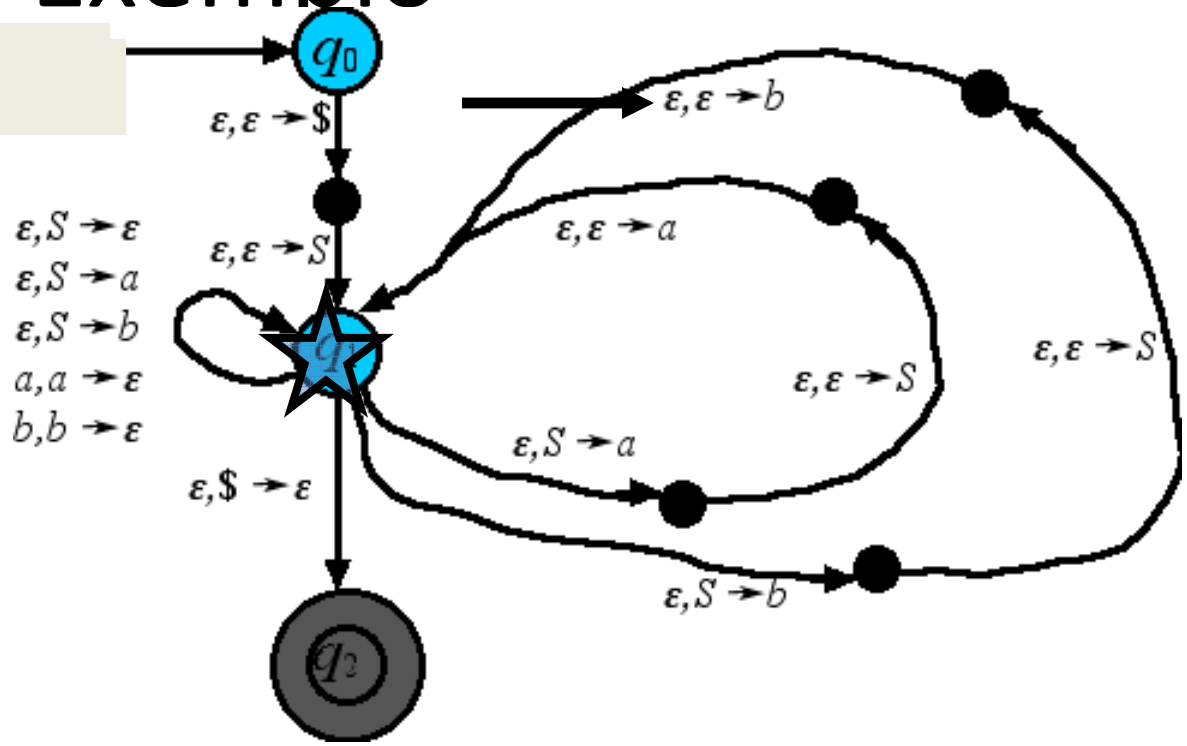


bbaabb

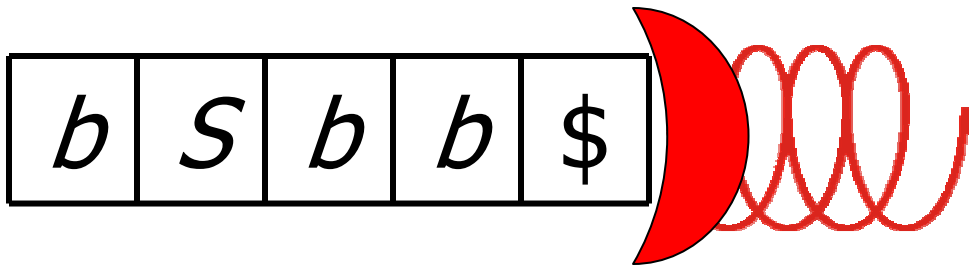


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

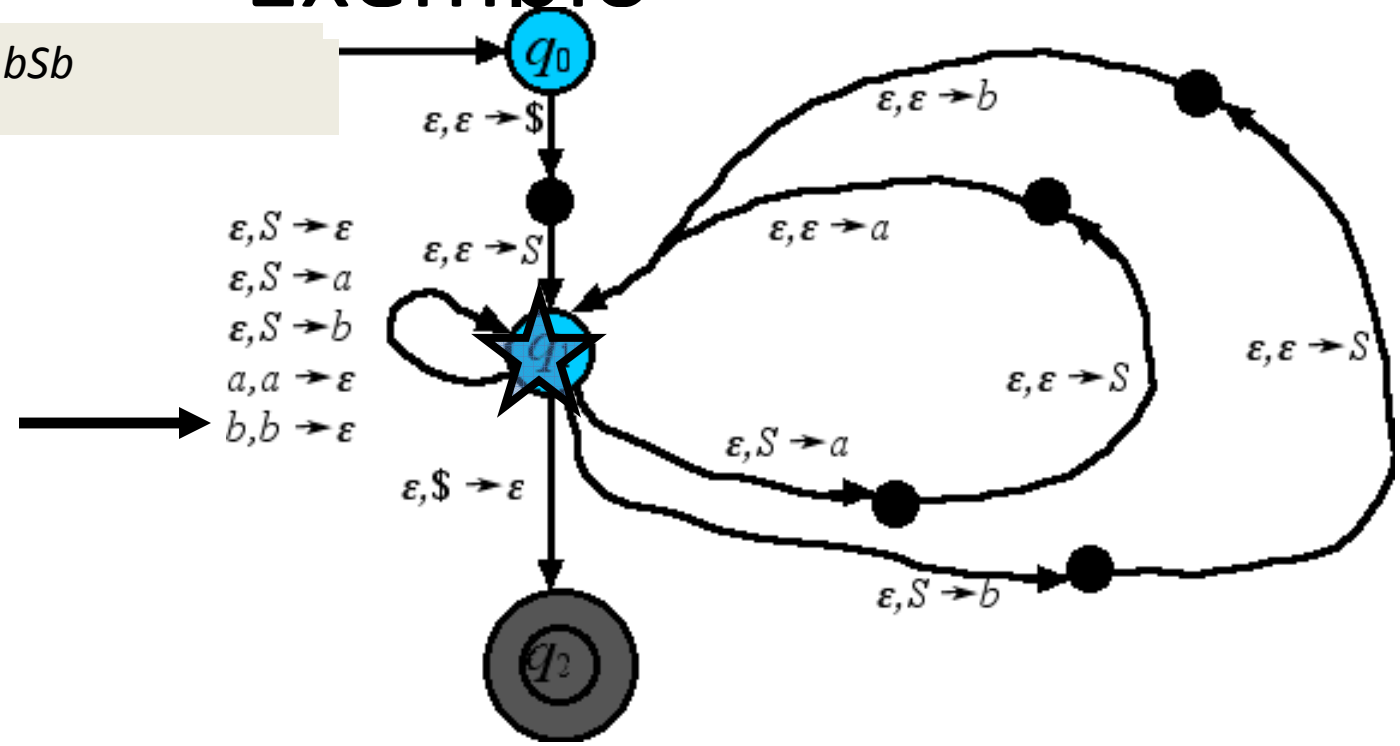


bbaabb

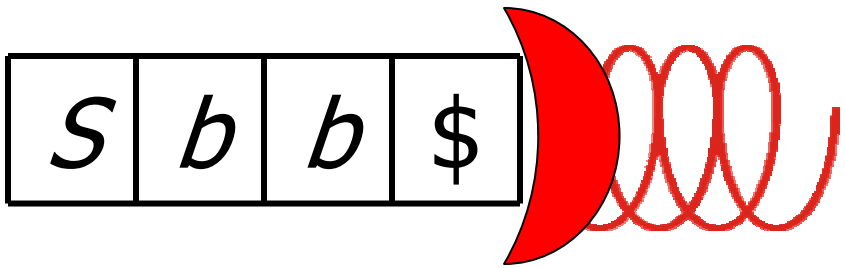


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

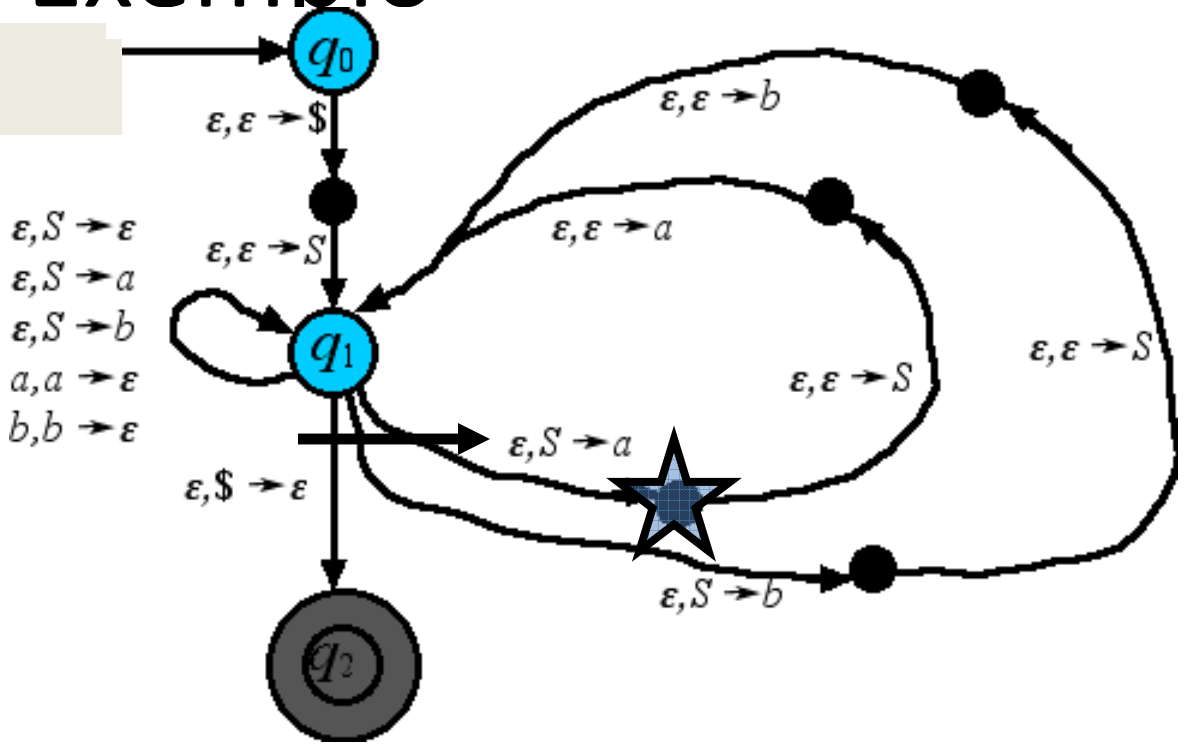


bbaabb

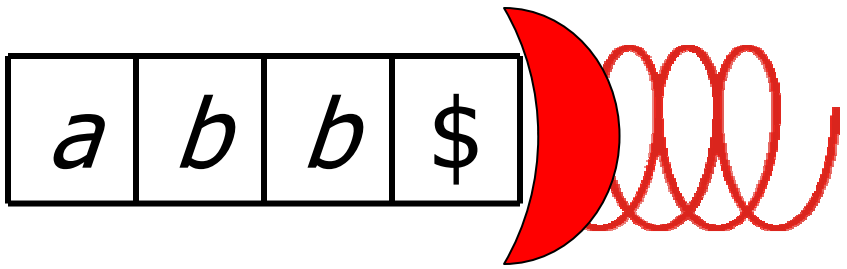


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

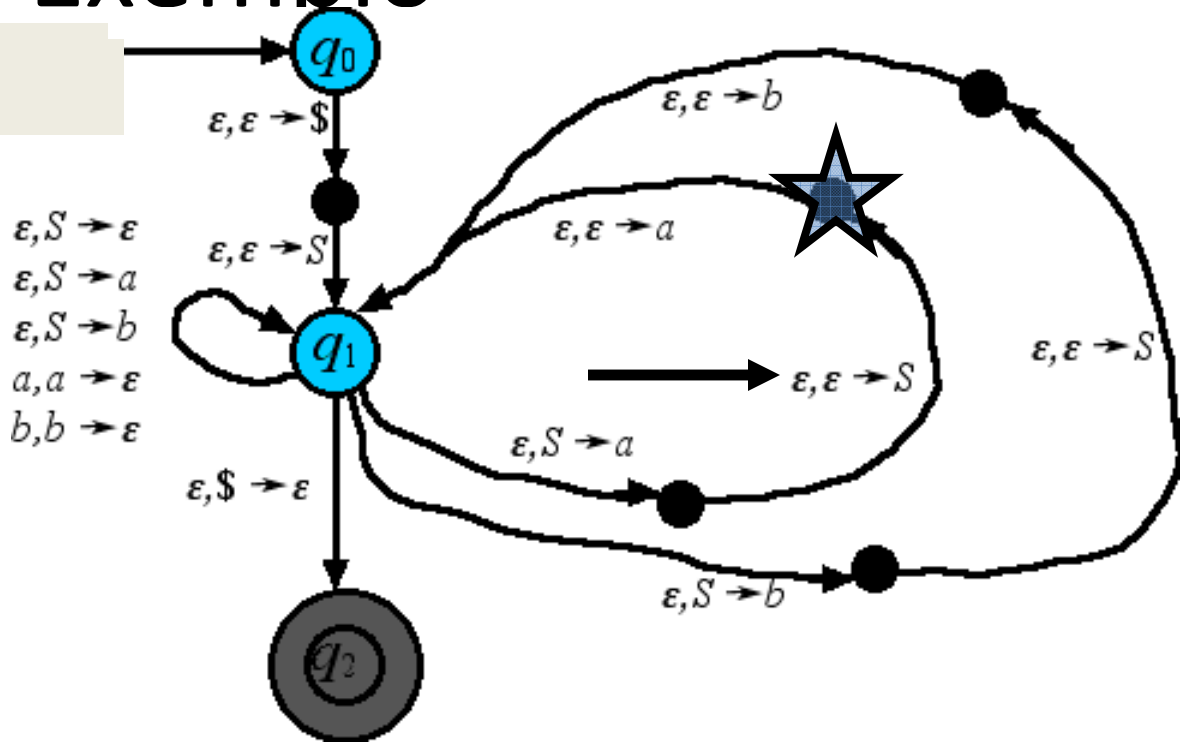


bbaabb

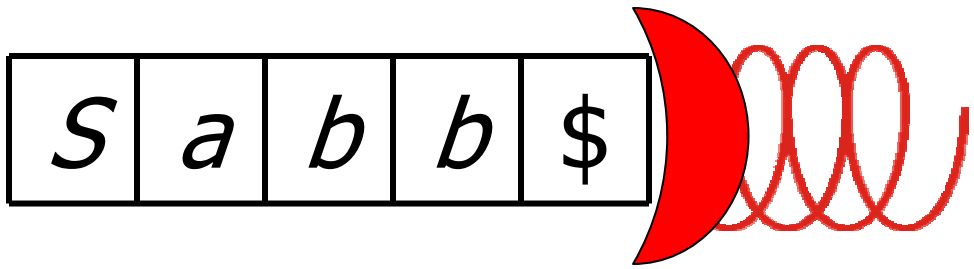


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

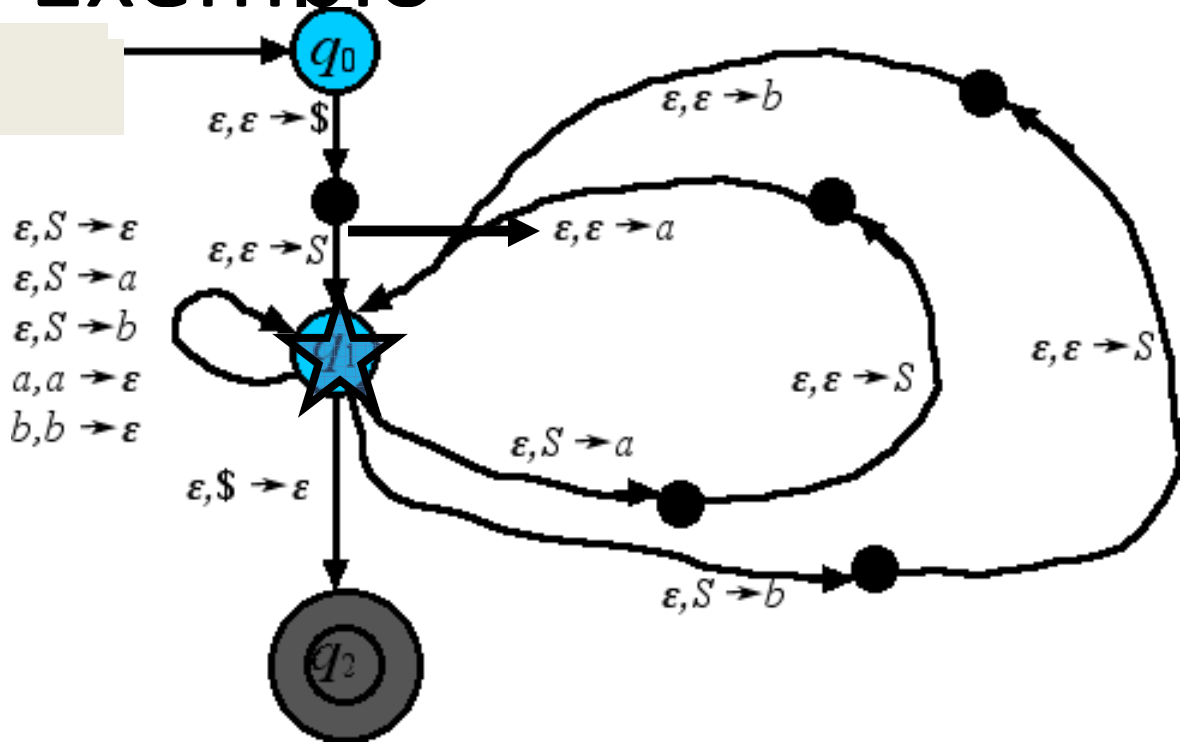


bbaabb

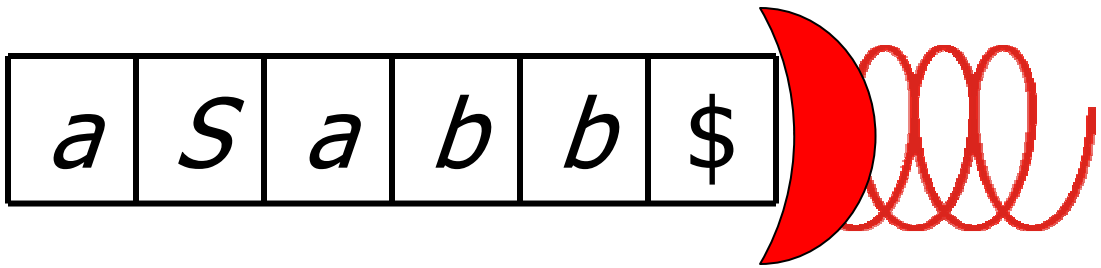


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

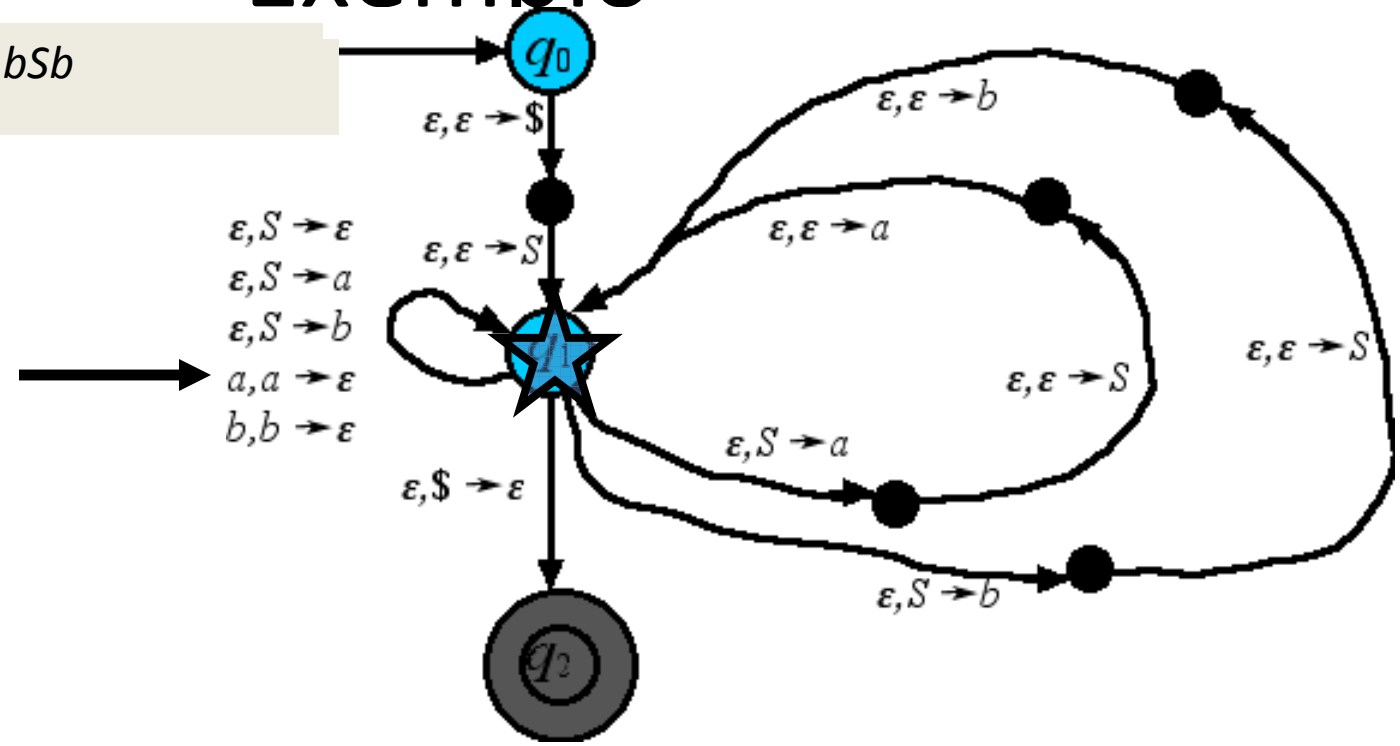


bbaabb

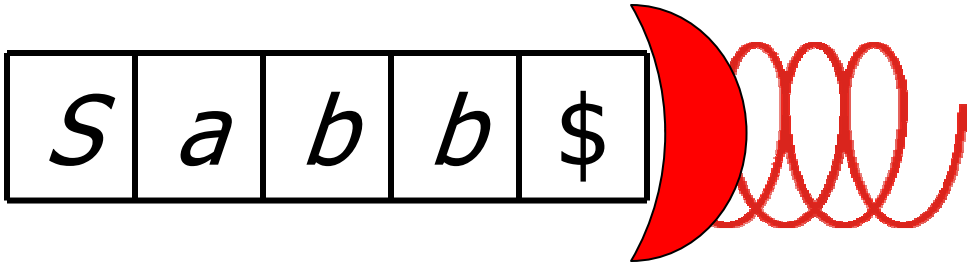


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

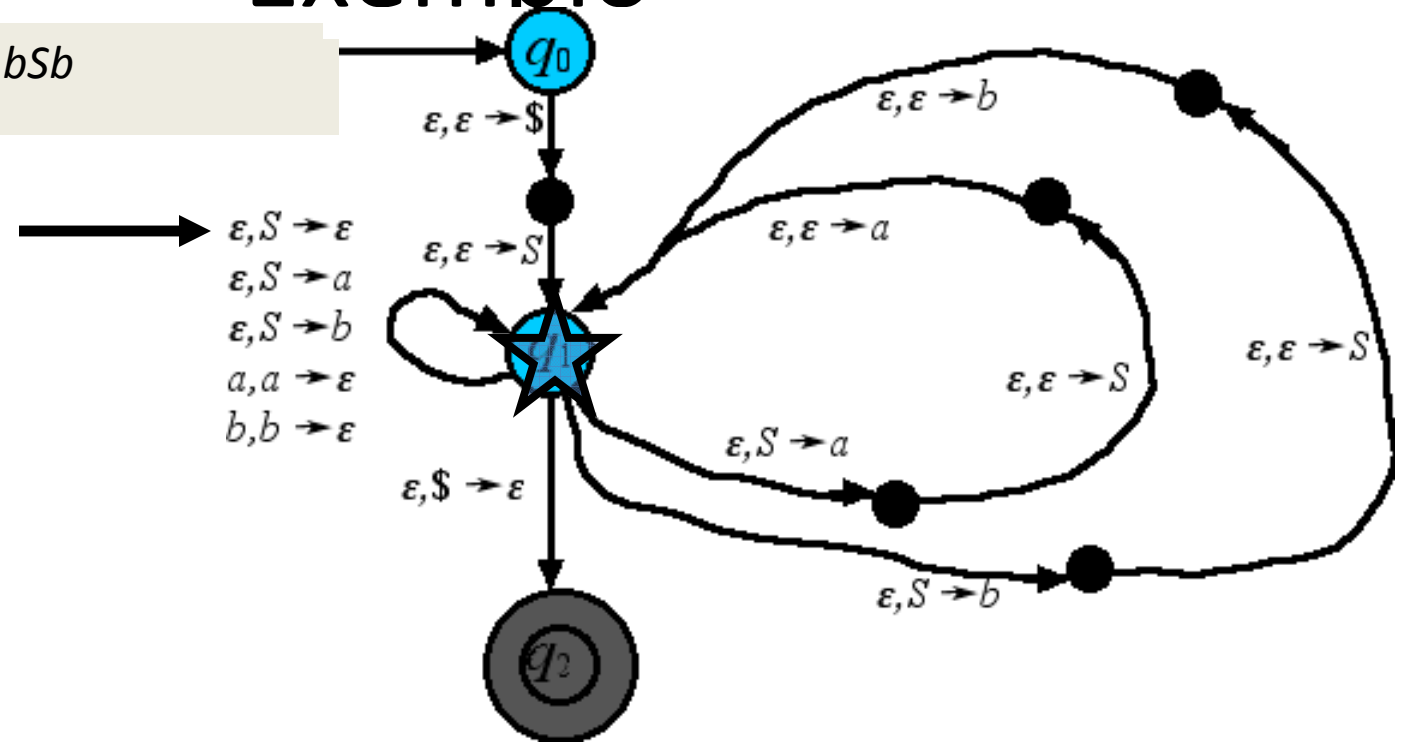


bbaabb

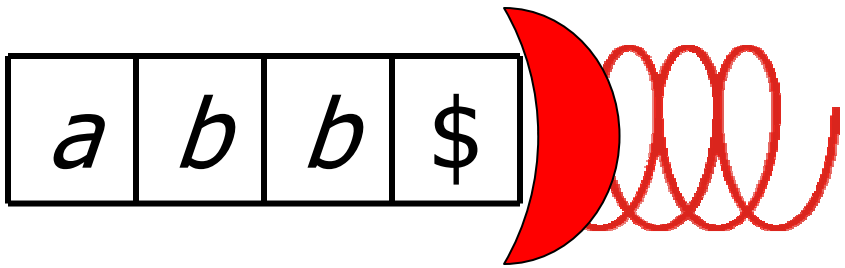


CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

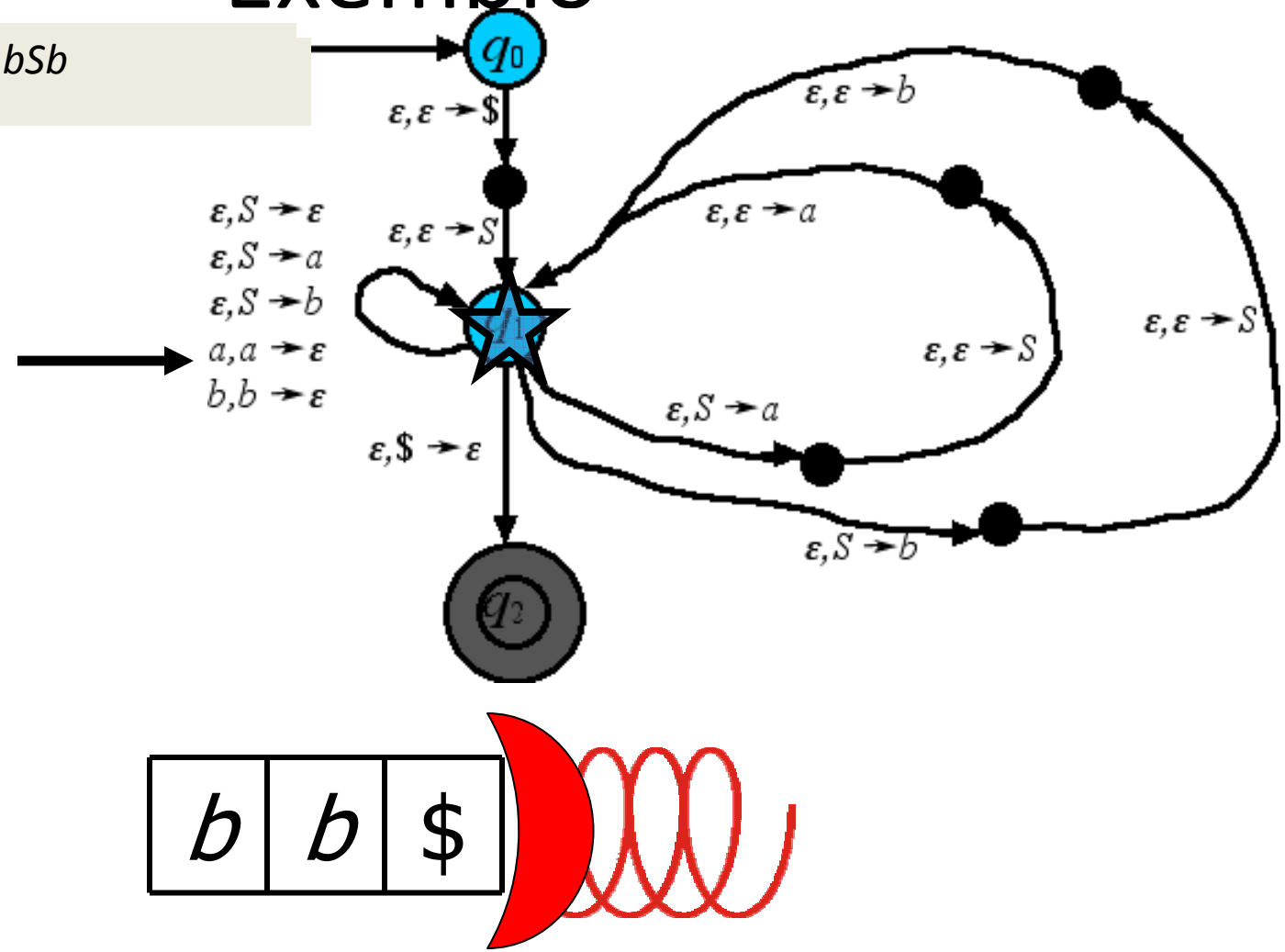


bbaabb



CFG → PDA
Exemplo

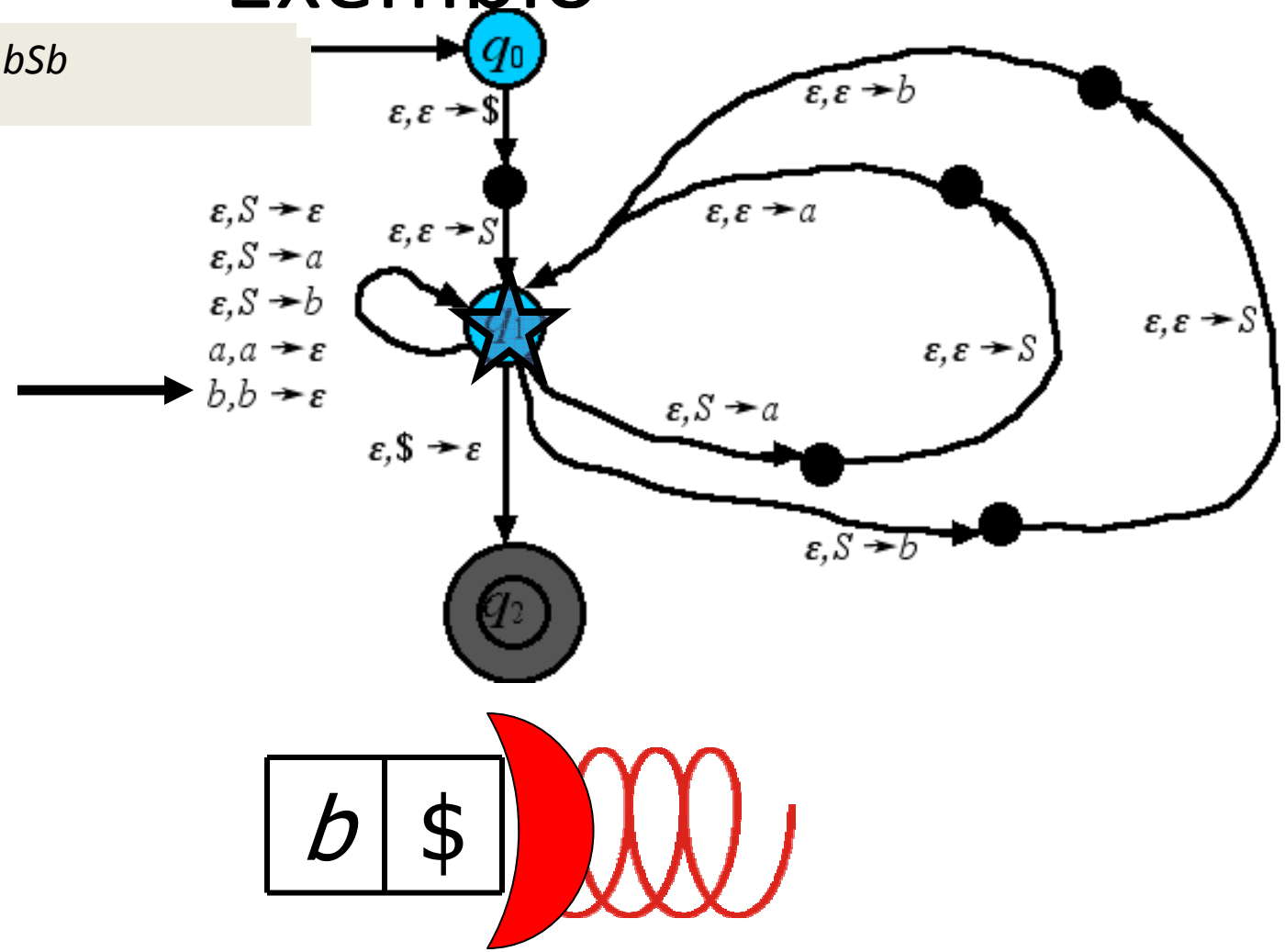
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



bbaabb

CFG → PDA
Exemplo

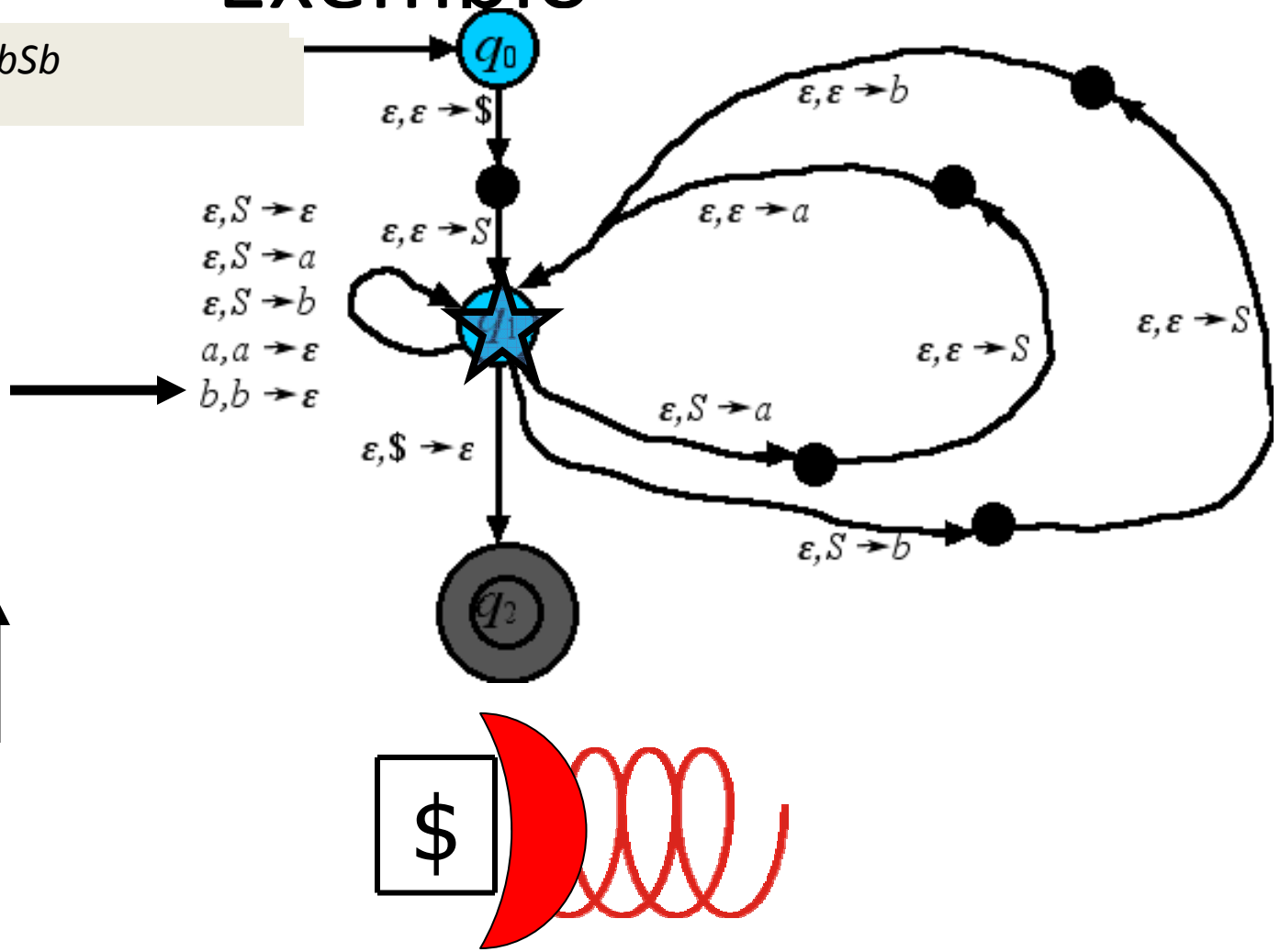
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



bbaabb

CFG → PDA
Exemplo

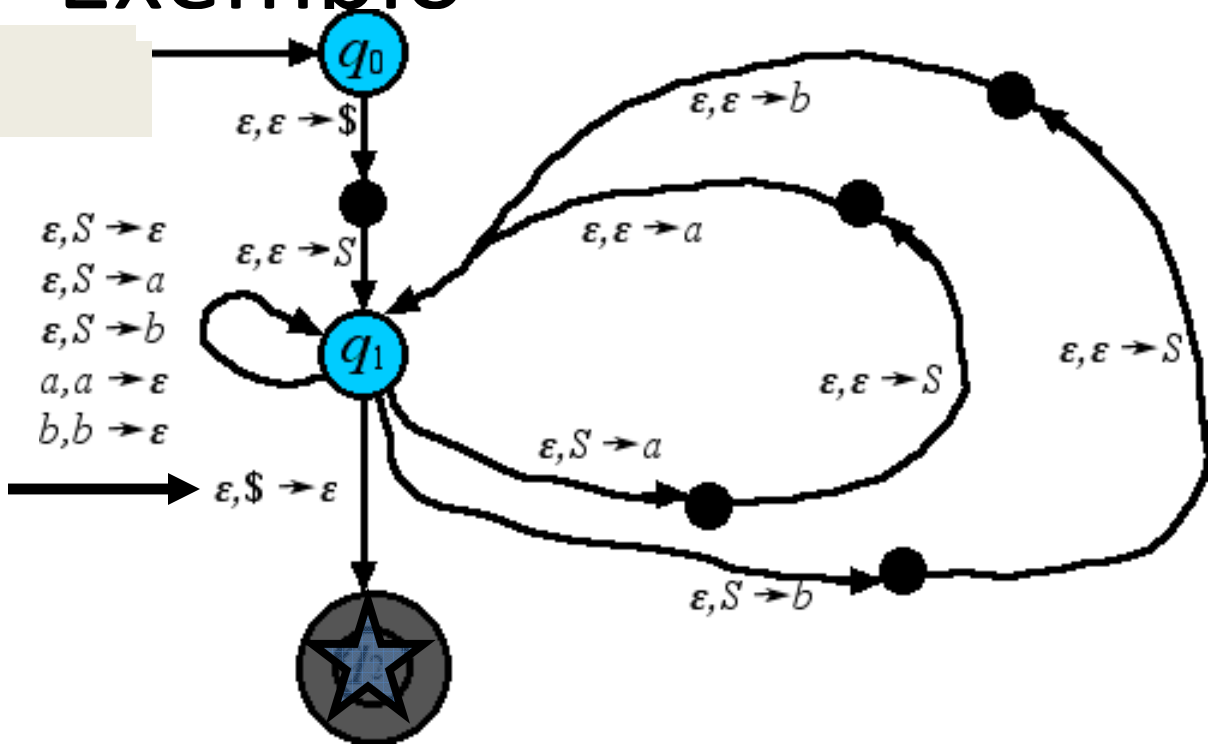
$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



bbaabb

CFG → PDA
Exemplo

$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



bbaabb

aceita!

CFG → PDA

Intuitivamente, toda derivação mais à esquerda pode ser simulada em um PDA como a seguir:

1. Empilhe *S*
2. Mude a variável no topo da pilha de acordo com a próxima regra a ser usada
3. Leia a entrada para obter o terminal que estiver no topo da pilha
4. Se a pilha estiver vazia ao ler toda a entrada, aceite. Senão, volte ao no. 2

Por outro lado, toda computação c/ aceitação necessariamente passa pelos passos acima e, portanto simula derivação mais à esq. em *G*.

Isso mostra que o PDA construído aceita a linguagem gerada pela gramática original.

Exercício

- Forneça PDA's para as CFG's das linguagens a seguir:
  - $\{a^i b^j c^k \mid i=j \text{ ou } j=k\}$
- Forneça PDA's para as CFG's:

a) $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$

b) $R \rightarrow X R X \mid S$

$S \rightarrow a T b \mid b T a$

$T \rightarrow X T X \mid X \mid \varepsilon$

$X \rightarrow a \mid b$

Gramática Sensível ao Contexto

Existe uma forma ainda mais geral de gramática. Em uma gramática **não** livre de contexto, em geral todo substring de variáveis/terminais pode ser substituído de uma vez. Por exemplo, sendo $\Sigma = \{a,b,c\}$ considere:

$S \rightarrow ASBC$

$A \rightarrow a$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Por razões técnicas, se toda regra é tal que o tamanho do lado esquerdo da regra é menor ou igual ao do lado direito, a gramática é chamada ***sensível ao contexto***.

Exercício

Qual é a linguagem gerada por:

$S \rightarrow ASBC$

$A \rightarrow a$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Exercício

Resposta: $\{a^n b^n c^n \mid n \geq 1\}$. Na próxima aula veremos que essa linguagem não é livre de contexto. Portanto, perturbar a propriedade de independência de contexto possibilita expandir a classe de linguagens.

PDA → CFG

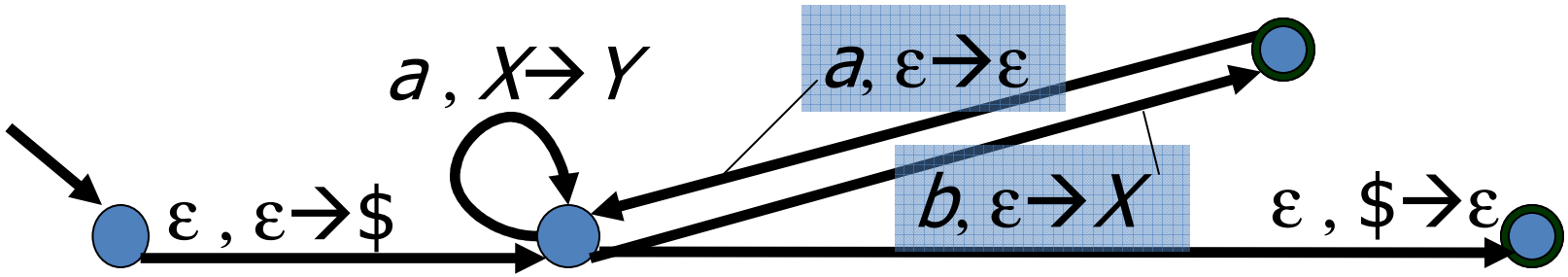
Para convertes PDA's para CFG's vamos precisar simular a pilha dentro de regras. Portanto, quanto mais simples as operações de pilha, mais fácil deverá ser isso. Além disso, outras restrições podem ser úteis. Portanto, vamos primeiro converter nosso PDA em um PDA tão simples quanto possível:

PPP → CFG
Hipótese Simplificadora

1. Hipótese PPP : A pilha apenas permite Pushs e Pops simples.
2. Um único estado de aceitação.
3. Pilha Vazia: Um string é aceito sse sua computação termina no estado de aceitação com a pilha vazia.

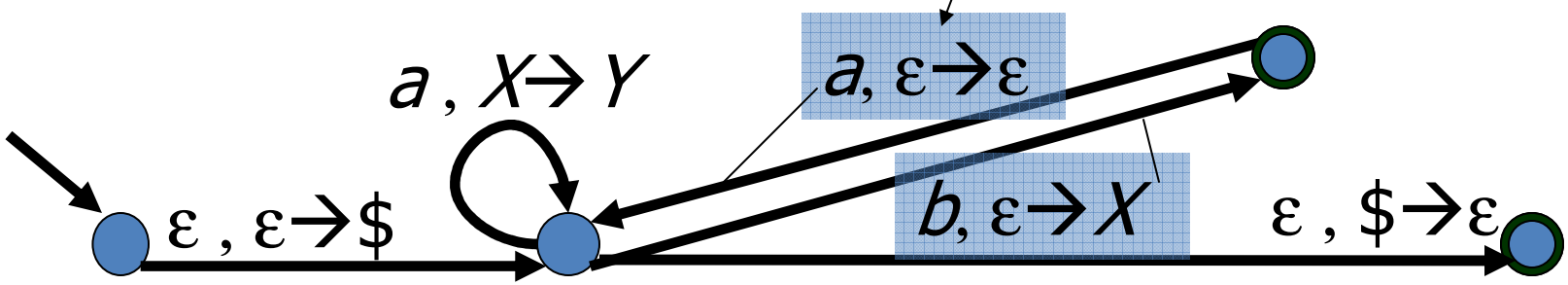
Vamos converter um exemplo típico nessa forma:

Simplificando o PDA
Exemplo Original



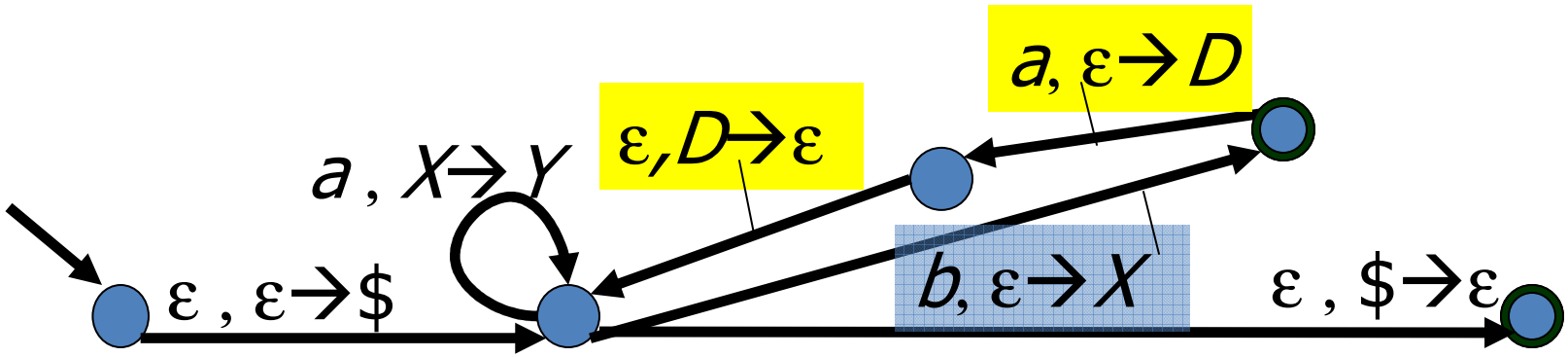
Simplificando o PDA
1. Push Pop Puro

1A) Garanta que a pilha está sempre ativa substituindo transição com pilha inativa por um push seguido imediatamente de um pop de um símbolo dummy.



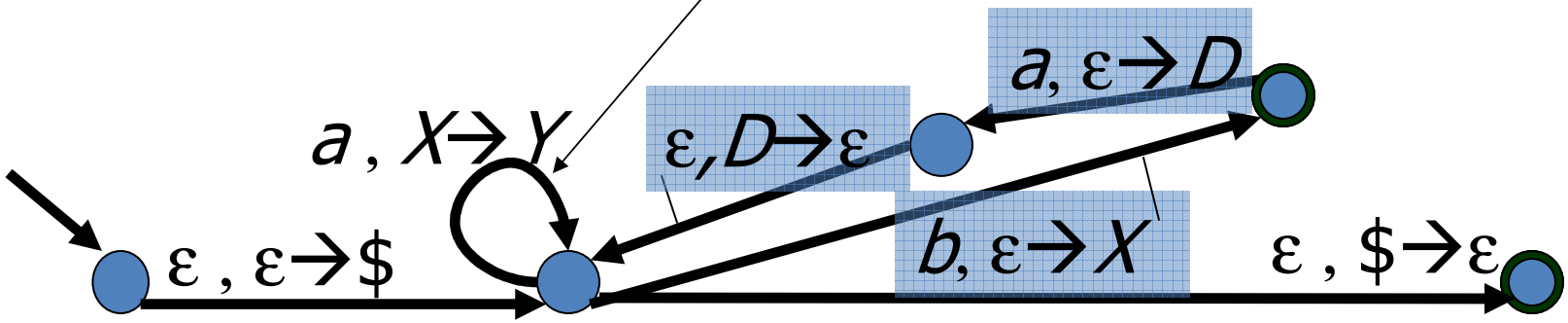
Simplificando o PDA
1. Push Pop Puro

1A) Garanta que a pilha está sempre ativa substituindo transição com pilha inactive por um push seguido imediatamente de um pop de um *novo* símbolo dummy.



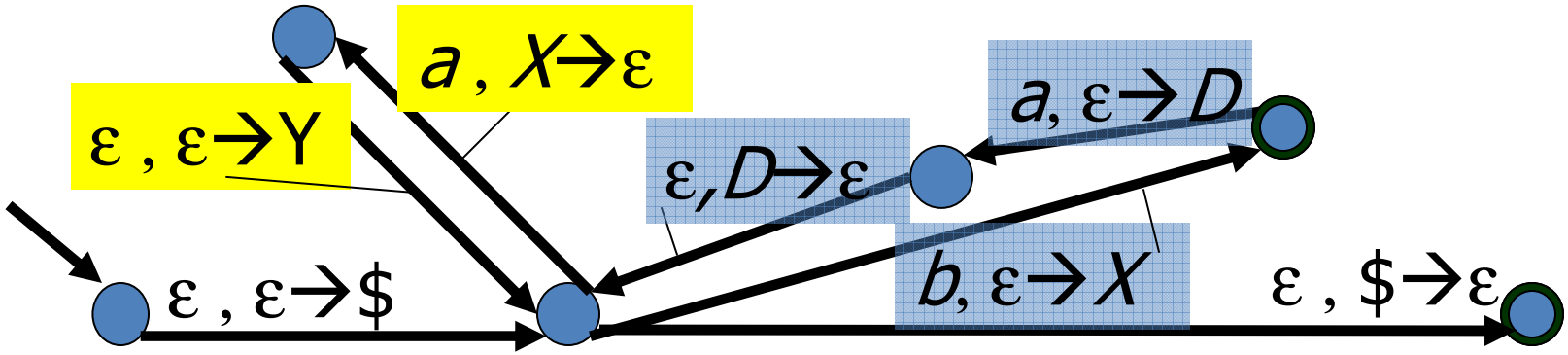
Simplificando o PDA
1. Push Pop Puro

1B) Toda transição que substitui o topo da pilha deve ser convertido em um pop seguido de push.



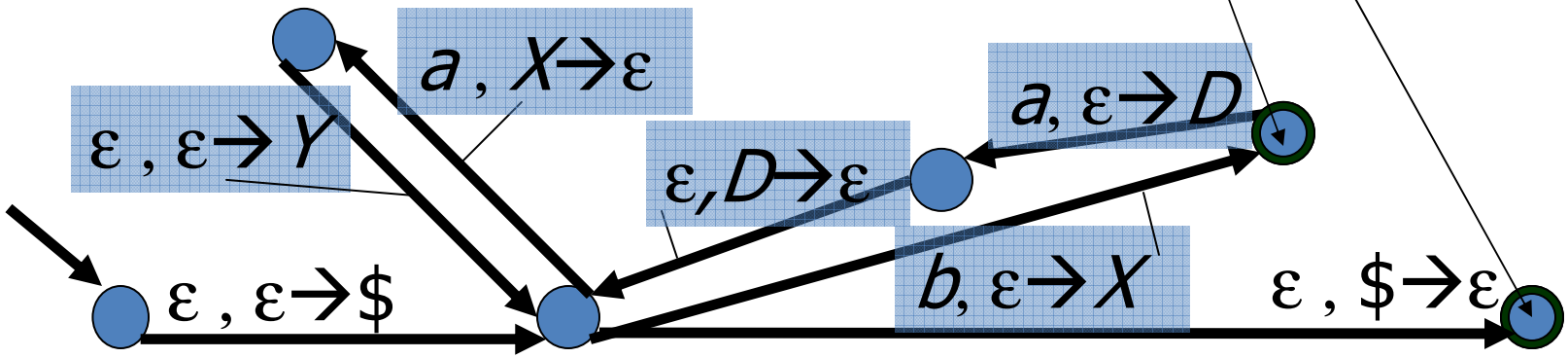
Simplificando o PDA 1. Pure Push Pop

1B) Toda transição que substitui o topo da pilha deve ser convertido em um pop seguido de push.



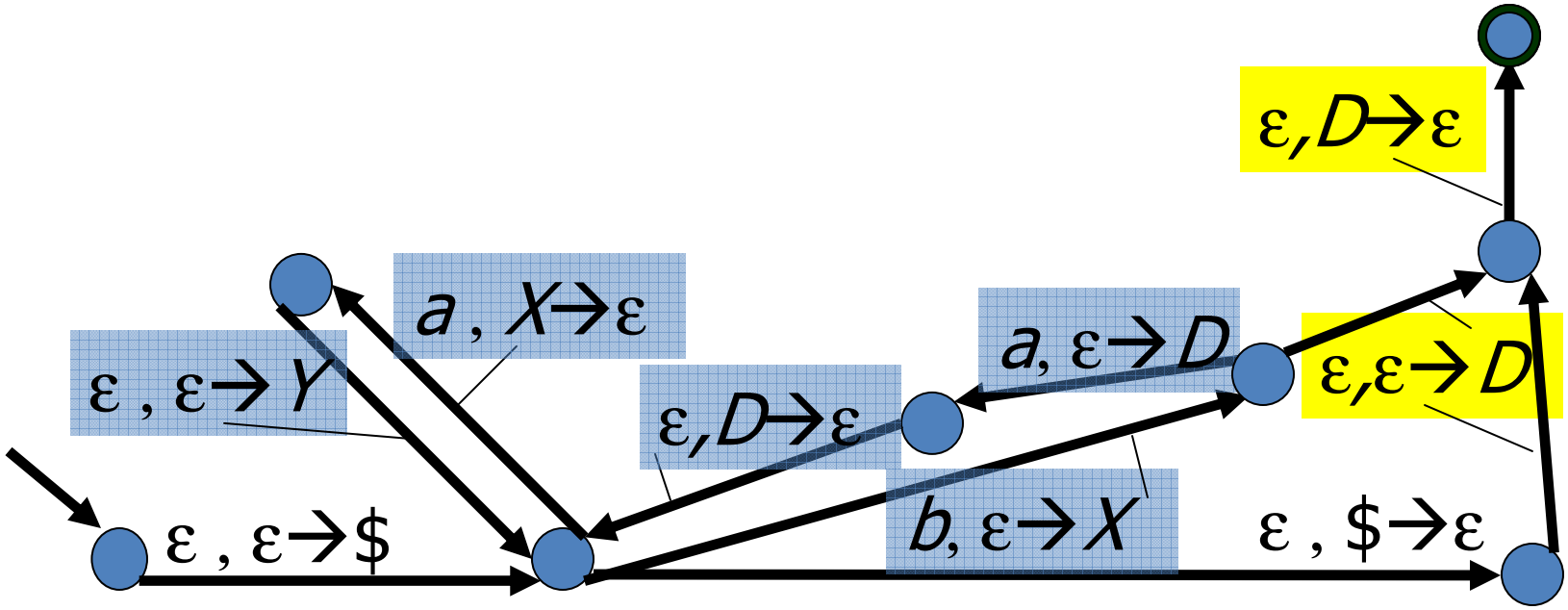
Simplificando o PDA
2. Estado de Aceitação Único

Desligue os estados de aceitação originais e conecte-os a um novo estado de aceitação (não esquecendo que a pilha não pode ser ignorada).



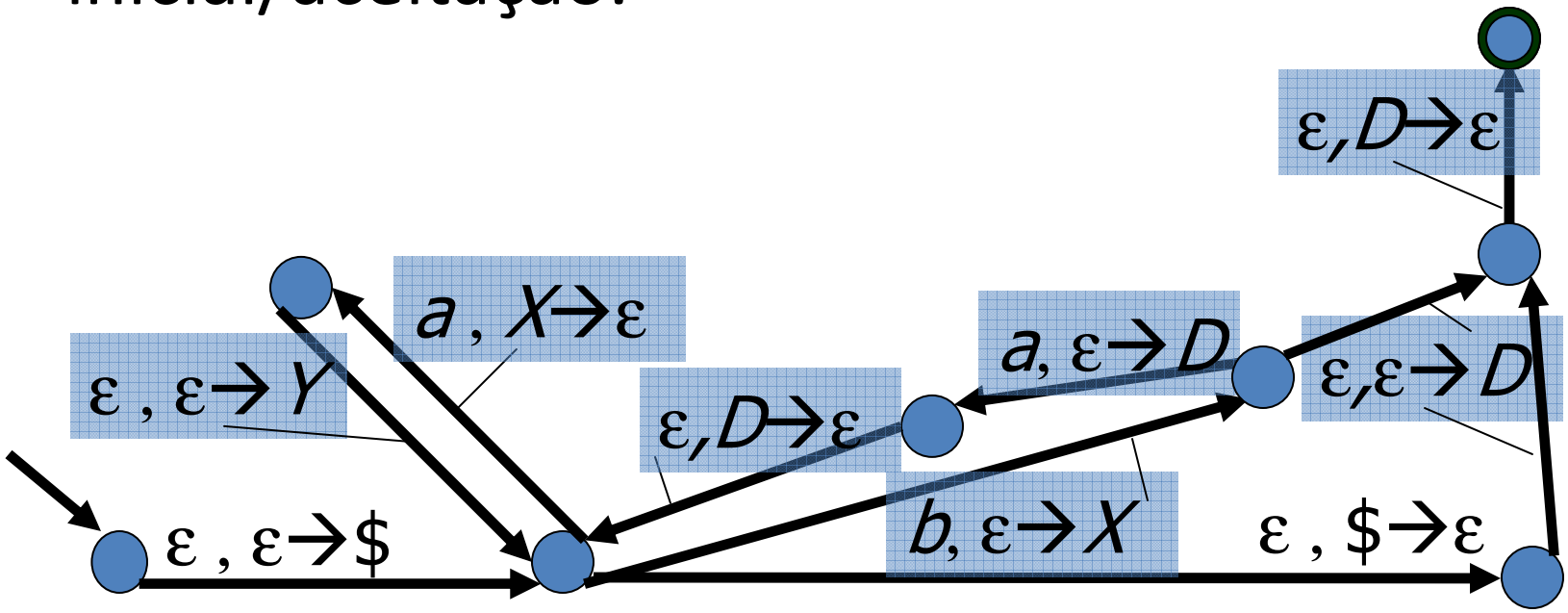
Simplificando o PDA
2. Estado de Aceitação Único

Desligue os estados de aceitação originais e conecte-os a um novo estado de aceitação (não esquecendo que a pilha não pode ser ignorada).



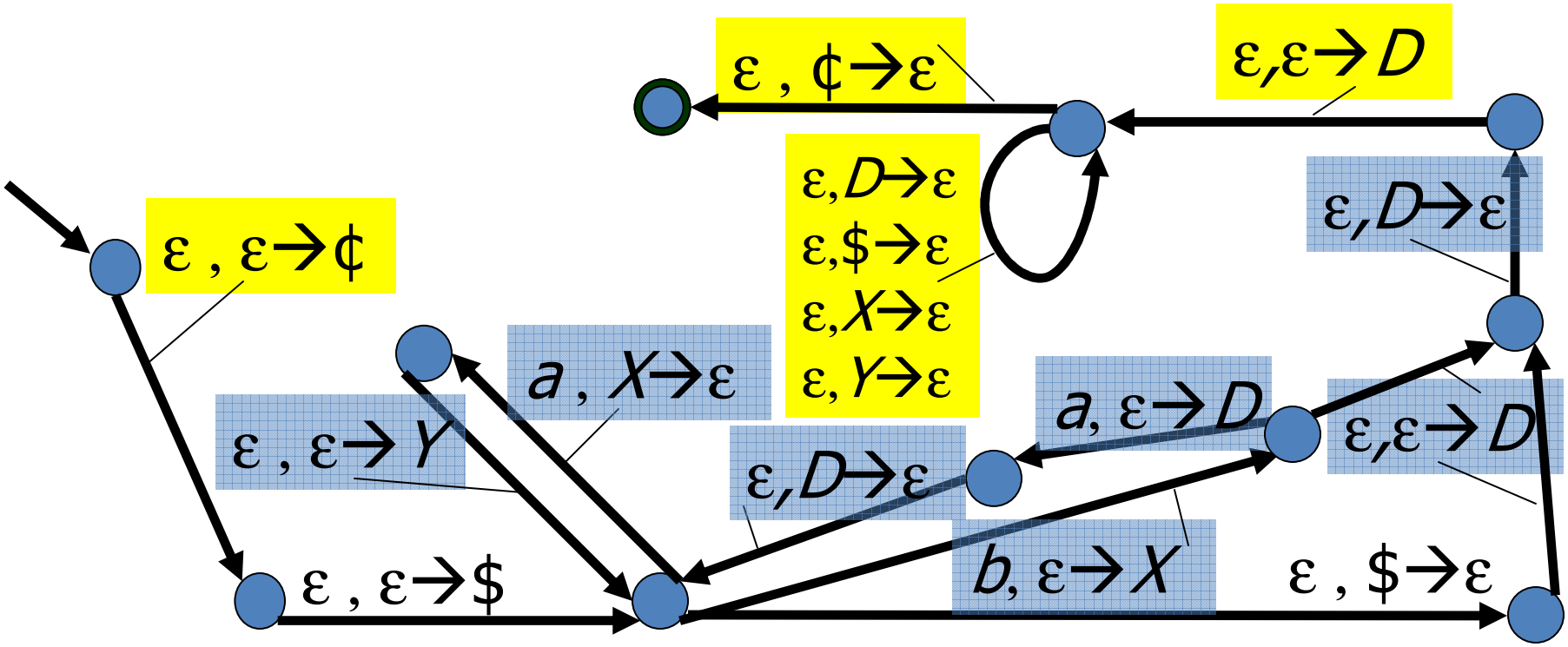
Simplificando o PDA
3. Pilha Vazia

Garanta que o conteúdo da pilha é esvaziado, adicionando um novo símbolo dummy marcador de pilha vazia e novos estados inicial/aceitação.

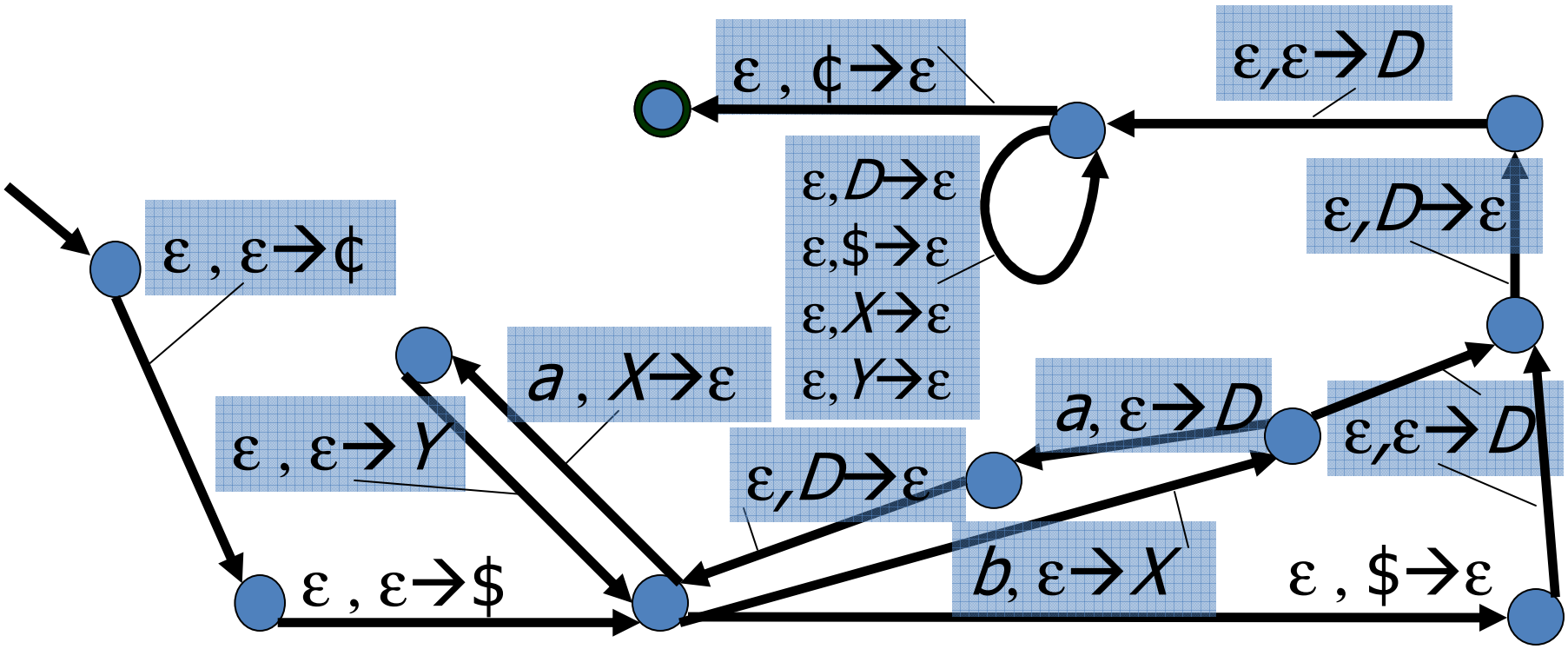


Simplificando o PDA
3. Pilha Vazia

Garanta que o conteúdo da pilha é esvaziado, adicionando um novo símbolo dummy marcador de pilha vazia e novos estados inicial/aceitação.



Simplificando o PDA
Resultado



PDA → CFG

Uma vez que o PDA seja convertido nessa forma mais restrita, podemos convertê-lo em uma CFG por meio de um procedimento padrão.

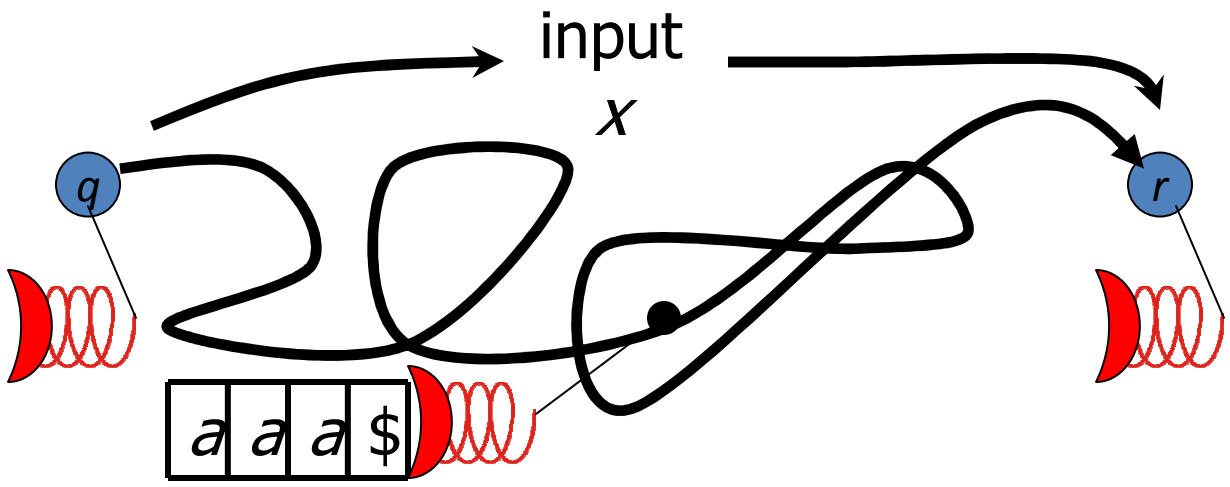
Agora que caminhos de aceitação começam e terminam com a pilha vazia, é possível considerar caminhos desse tipo entre *quaisquer dois estados* e recursivamente gerar todos esses caminhos. Essa relação recursiva entre caminhos dará origem à recursão que é característica de gramáticas livres de contexto.

PDA → CFG
Recursão sobre Caminhos

Notação: dados dois estados *q,r* do PDA, e um string *x* do dado alfabeto de entrada, a notação

$$q\text{-}x \rightarrow r$$

significa que é possível ir de *q* para *r* lendo a entrada *x*, começando e terminando com pilha vazia:



Q: Expresse aceitação em termos dessa notação.

PDA → CFG
Recursão sobre Caminhos

R: Para nosso PDA restrito, com estado de aceitação único q_F, um string x é aceito sse q_0-$x \rightarrow q_F$

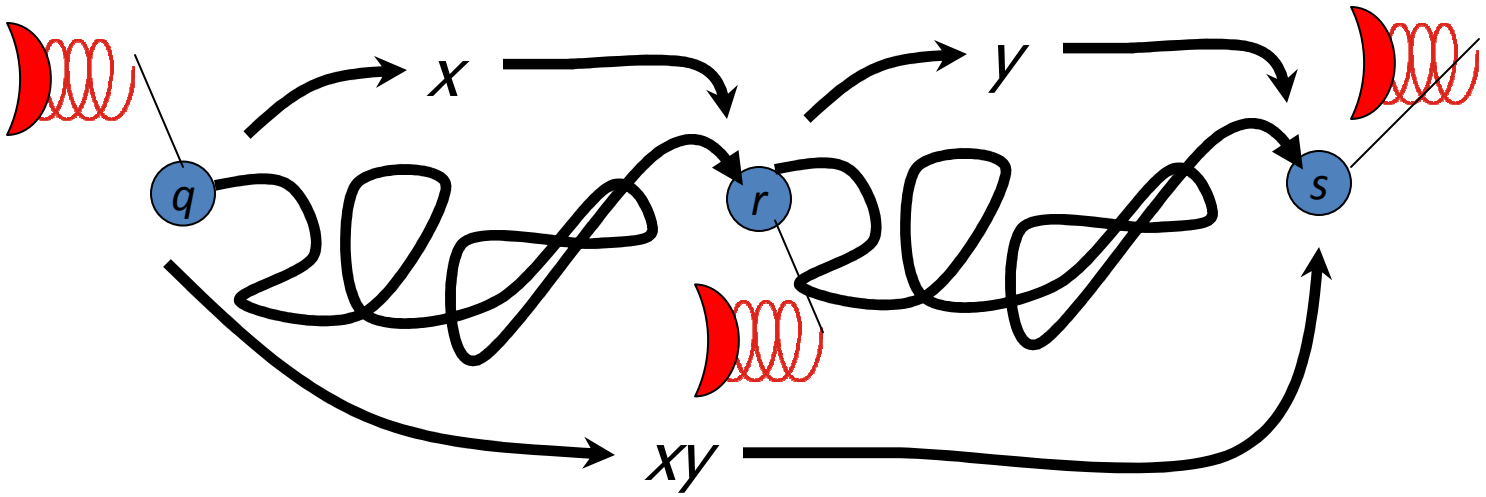
Portanto, todos os strings aceitos são gerados se podemos gerar todas as "triples" que satisfazem q-$x \rightarrow r$. Isso é feito de modo recursivo sobre o comprimento do caminho:

1. Regra-Base: Podemos sempre considerar que o string vazio leva de q para q sem modificar a pilha, já que nada é lido:

$$q\text{-}\varepsilon \rightarrow q$$

PDA → CFG
Recursão sobre Caminhos

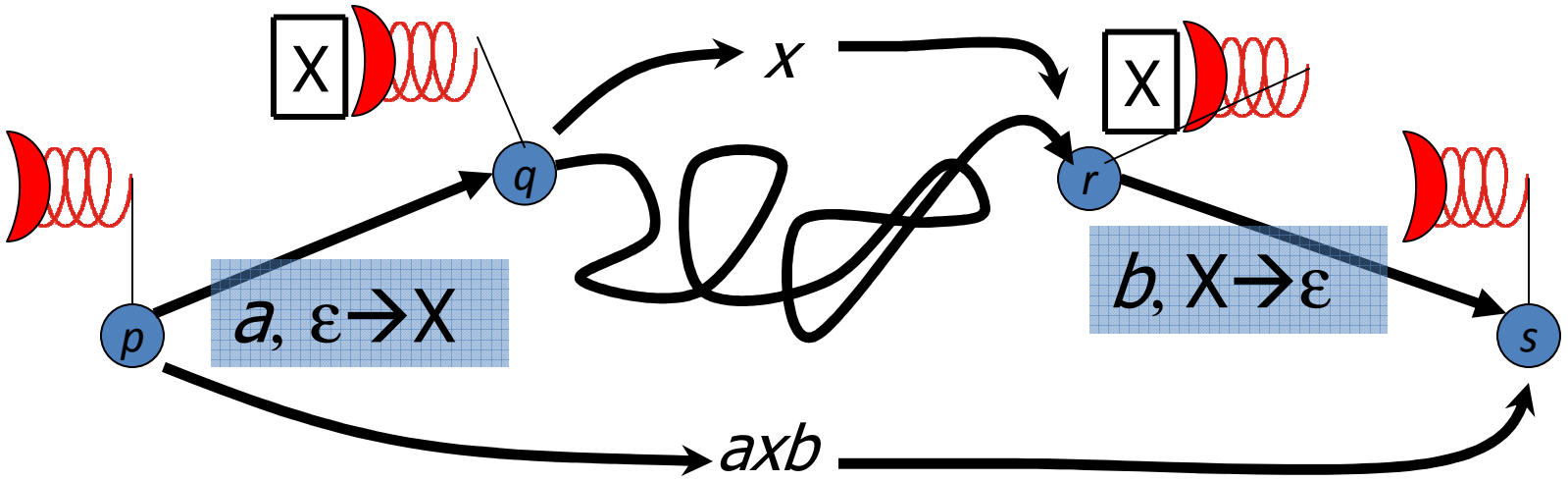
2. Regra de Recursão Transitive: Se podemos ir de *q* para *r* sem afetar a pilha, assim como de *r* para *s*, então podemos combinar os caminhos para obter um caminho de *q* para *r*. I.e: *q-x→r* e *r-y→s* implica *q-xy→s*



PDA → CFG
Recursão sobre Caminhos

3. Regra de Recursão Push-Pop: Se podemos ir de *q* para *r* sem afetar a pilha, e empilhamos um símbolo X indo de *p* para *q,* o qual é desempilhado ao ir de *r* para *s,* então vamos de *p* para *s* sobre pilha vazia:

q-*x*→*r* e (*q*,X)∈ δ(*p*, *a*, ε) e (*s*, ε)∈ δ(*r*,*b*, X) implica *p*-*axb*→*s*



PDA → CFG
Recursão sobre Caminhos

LEMA: Toda tripla $q\text{-}x \rightarrow r$ deve poder ser gerada indutivamente por uma das regras (1), (2) ou (3).

Prova. Por indução sobre o comprimento n do caminho

$$q\text{-}x \rightarrow r.$$

Caso Base ($n = 0$): x é o string vazio e tais caminhos são gerados pela regra (1).

Indução ($n > 0$): Siga o caminho, iniciando com pilha vazia. Existem 2 possíveis situações:

I. A pilha é esvaziada em algum ponto intermediário.

II. A pilha nunca fica vazia antes do final.

PDA → CFG
Recursão sobre Caminhos

Caso I. Em algum ponto intermediário, digamos estado *s*, a pilha é esvaziada. Então divida o caminho em 2 partes, cada qual com sua parte do string de entrada, e cada um começando e terminando com a pilha vazia. I.e. divida *x* como *x* = *uv* tal que *q*-*u* →*s* e *s*-*v* →*r*. Então aplique a regra (2).

PDA → CFG
Recursão sobre Caminhos

Caso II. A pilha não fica vazia em nenhum estado intermediário. Portanto, a primeira transição faz push (nenhum pop) de um símbolo X que apenas é desempilhado na última transição. Seja *s* o estado destino da primeira transição, e *t* o estado origem da última transição. Então podemos ir de *s* para *t* sob pilha vazia, lendo algum string *u*. Além disso, $(s,X) \in \delta(p,a,\varepsilon)$, $(r,\varepsilon) \in \delta(t,b,X)$ e $x = aub$. Essa é exatamente a situação à qual a regra (3) se aplica.

Isso completa a prova.

PDA → CFG
A Gramática

As três regras para gerar todos os caminhos nos dão uma gramática para gerar todos os rótulos desses caminhos. A gramática terá variáveis escritas como A_{qr} a qual irá gerar todos os strings x para os quais q-$x \rightarrow r$.

Q: Supondo isso, qual deve ser a variável inicial?

PDA → CFG
A Gramática –Símbolos

R: $S = A_{q^0 q^F}$ Isso porque são aceitos exatamente os strings x para os quais vale $q_0\text{-}x \rightarrow q_F$.

Além dessa variável inicial, as outras variáveis em V são todos os A_{qr} para os quais existe um caminho de q para r que começa e termina com pilha vazia.[1]

O conjunto de símbolos terminais Σ é o alfabeto de entrada do PDA.

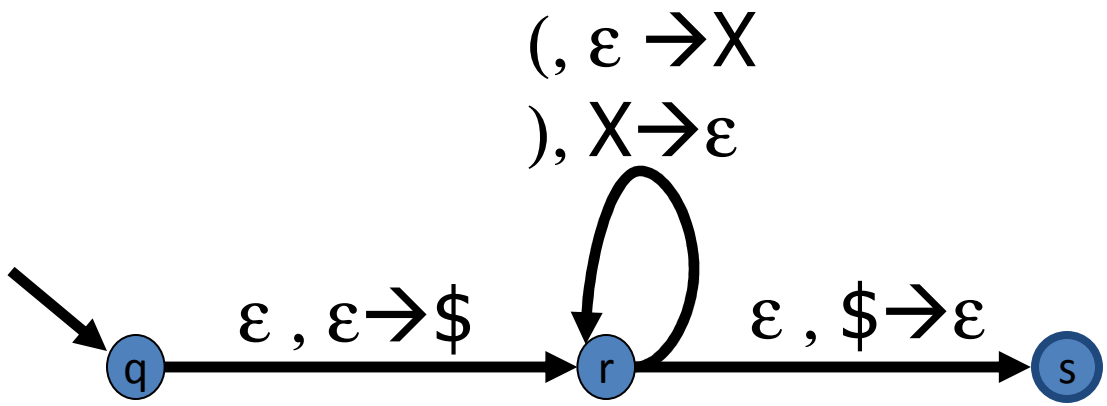
PDA → CFG
A Gramática –Regras

As regras são exatamente (1), (2) e (3):

1. Inclua uma regra $A_{qq} \rightarrow \varepsilon$ para cada estado q do PDA.
2. Inclua uma regra $A_{pr} \rightarrow A_{pq} A_{qr}$ para cada tripla p,q,r tais que A_{pr}, A_{pq} e A_{qr} estão em V.
3. Inclua uma regra $A_{ps} \rightarrow aA_{qr}b$ para cada p,s,q,r tais que A_{ps} e A_{qr} estão em V, e existem no PDA transições $(q,X) \in \delta(p,a,\varepsilon)$, e $(s,\varepsilon) \in \delta(r,b,X)$ para um mesmo símbolo de pilha X.

PDA → CFG
Exemplo

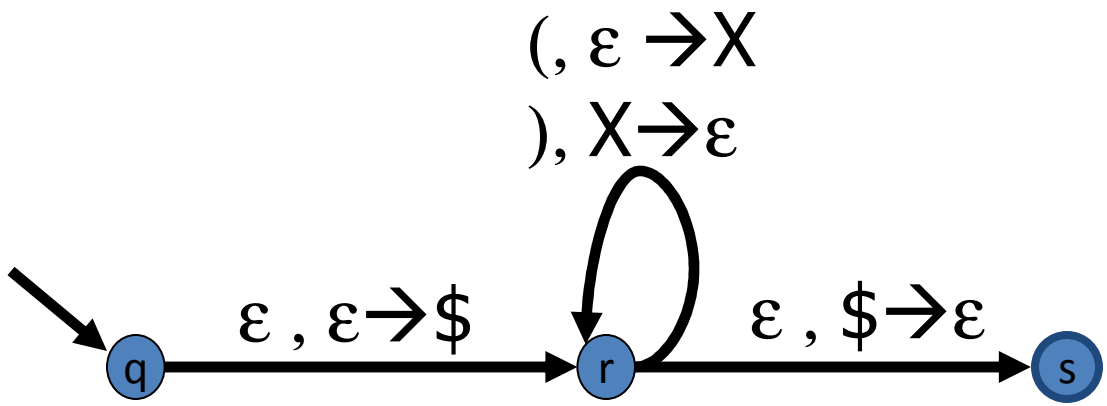
O PDA abaixo já está na forma correta (a forma restrita PPP) :



Q: Qual é a linguagem aceita?

PDA → CFG
Exemplo

R: "BP" = parenteses balanceados. O número de X's na pilha reflete o nível corrente de aninhamento de parenteses

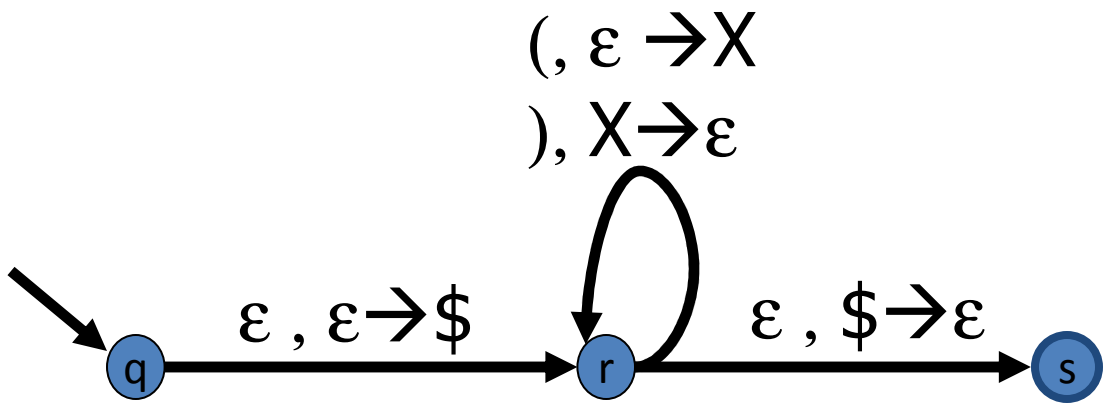


Q: Quais são as variáveis da gramática equivalente? Qual é a variável inicial?

PDA → CFG
Exemplo

R: $V = \{A_{qs}, A_{qq}, A_{rr}, A_{ss}\}$, $S = A_{qs}$

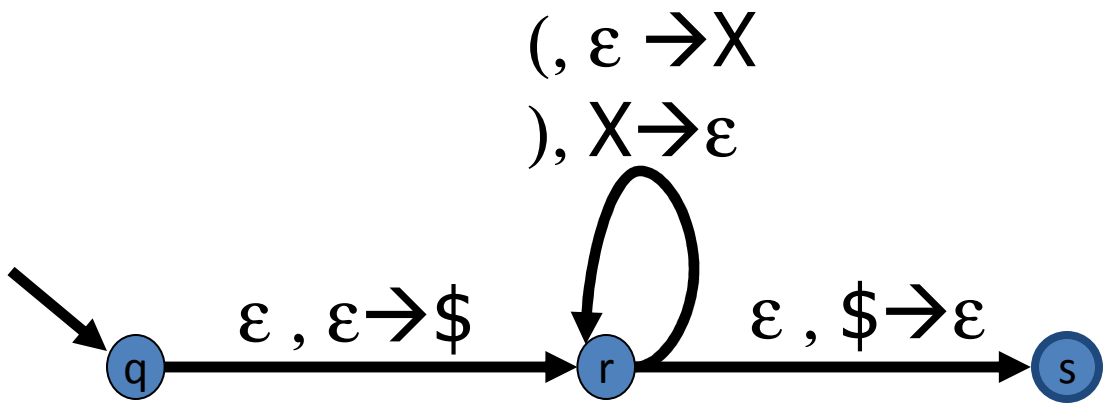
Não precisamos de A_{rq}, A_{sq}, A_{sr} : direção errada. Não precisamos de A_{qr} ou A_{rs} porque não se pode empilhar ou desempilhar $ estando em *r*.



Q: Que produções obtemos da regra (1)?

PDA → CFG
Exemplo

R: $A_{qq} \rightarrow \varepsilon$, $A_{rr} \rightarrow \varepsilon$, $A_{ss} \rightarrow \varepsilon$



Q: Que produções obtemos da regra (2)?

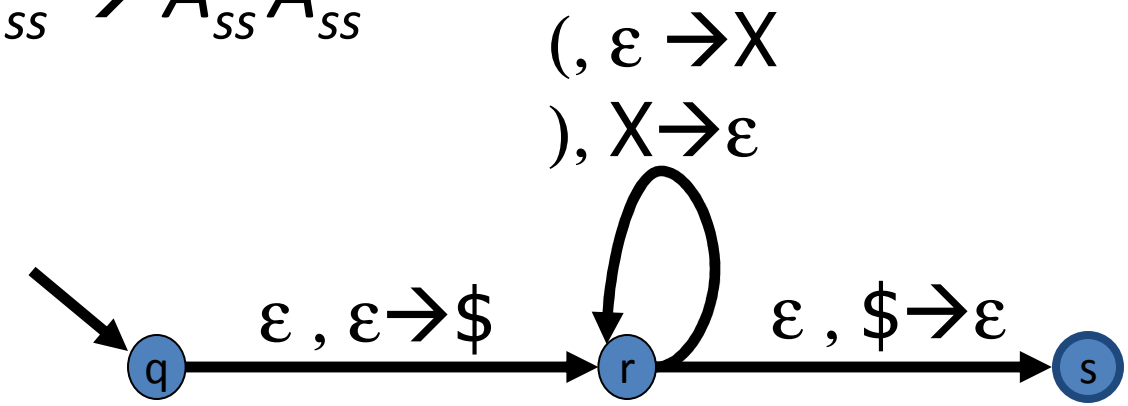
PDA → CFG
Exemplo

A: $A_{qs} \rightarrow A_{qq} A_{qs} \mid A_{qs} A_{ss}$

$A_{qq} \rightarrow A_{qq} A_{qq}$

$A_{rr} \rightarrow A_{rr} A_{rr}$

$A_{ss} \rightarrow A_{ss} A_{ss}$



Q: Que produções obtemos da regra (3)?

PDA → CFG
Exemplo

A: $A_{qs} \rightarrow A_{rr}$, $A_{rr} \rightarrow (A_{rr})$

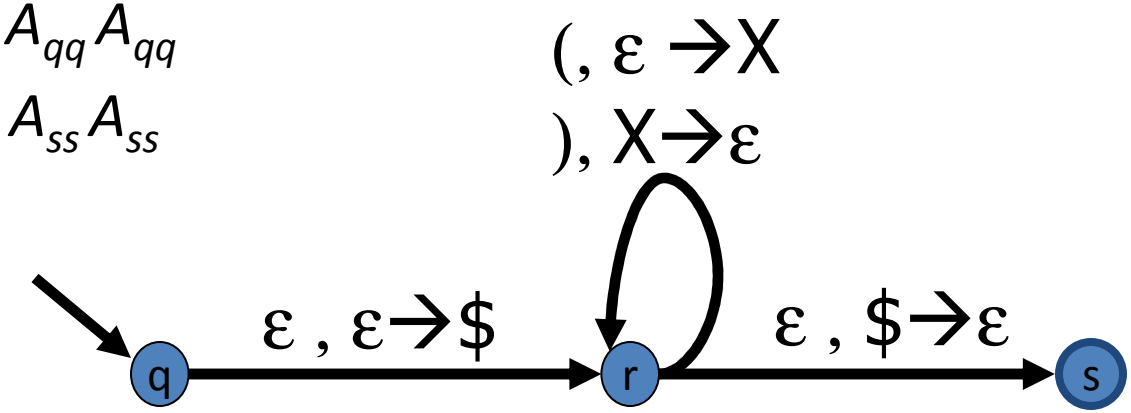
Portanto a gramática é:[1]

$A_{qs} \rightarrow A_{rr} \mid A_{qq} A_{qs} \mid A_{qs} A_{ss}$

$A_{rr} \rightarrow \varepsilon \mid A_{rr} A_{rr} \mid (A_{rr})$

$A_{qq} \rightarrow \varepsilon \mid A_{qq} A_{qq}$

$A_{ss} \rightarrow \varepsilon \mid A_{ss} A_{ss}$



Q: Alguma simplificação óbvia?

PDA → CFG
Exemplo

A: Aparentemente A_{qq} e A_{ss} são puramente auto-referenciaveis, portanto a única maneira de concluir uma derivação em que ocorram seria "apagá-las". Então podemos remover A_{qq}, A_{ss} desde que sejam substituídas por ε:

$A_{qs} \rightarrow A_{rr} \mid A_{qq} A_{qs} \mid A_{qs} A_{ss}$

$A_{rr} \rightarrow \varepsilon \mid A_{rr} A_{rr} \mid (A_{rr})$

$A_{qq} \rightarrow \varepsilon \mid A_{qq} A_{qq}$

$A_{ss} \rightarrow \varepsilon \mid A_{ss} A_{ss}$

Torna-se:

$A_{qs} \rightarrow A_{rr} \mid A_{qs}$

$A_{rr} \rightarrow \varepsilon \mid A_{rr} A_{rr} \mid (A_{rr})$

PDA → CFG
Exemplo

$A_{qs} \rightarrow A_{rr} \mid A_{qs}$

$A_{rr} \rightarrow \varepsilon \mid A_{rr} A_{rr} \mid (A_{rr})$

Renomeir as variáveis para obter:

$S \rightarrow T \mid S$

$T \rightarrow \varepsilon \mid TT \mid (T)$

Resposta final (*S* não é necessário, pois o seu papel é apenas gerar *T*):

$T \rightarrow \varepsilon \mid TT \mid (T)$

Exercício

- Converta para CFG's os PDA's das seguintes linguagens:
  - $\{wcw^R \mid w \in \{a, b\}^*\}$
  - $\{a^n b^n \mid n \in \mathbb{N}\}$
  - $\{a^n b^n c^m d^m \mid n,m \in \mathbb{N}\}$