Aşağı-itme Otomatı(Push-Down Automata)PDA

Sonlu otomatlar(FA) tüm context-free (bağlamdan bağımsız) dilleri tanıyamazlar çünkü sonlu hafızaları vardır. Fakat context-free dillerin tanınması için sınırsız miktarda bilgi depolanması gerekebilir. Örneğin $L = \{a^n b^n : n \geq 0\}$ dilindeki herhangi bir stringi incelediğimizde, sadece a'ların b'lerden önce gelip gelmediğini kontrol etmeyiz, aynı zamanda a'ları saymamız da gerekir. n sınırsız olduğu için, bu sayma işlemi sonlu otomatlarla (FA) yapılamaz. Sınır olmadan sayabilen bir otomata ihtiyacımız var.

Fakat başka örneklere bakacak olursak, mesela $\{ww^R\}$, sınırsız sayma becerisinden fazlasına ihtiyacımız var. Sembolleri depolayıp, sıralarını ters sırada karşılaştırabilen bir otomat gerekmektedir. Bunun için, depolama mekanizması sınırsız depolamaya izin veren bir yığın yapısı kullanabiliriz. Bunu da pushdown otomatlarla yapabiliriz.

- Düzgün dillerin hepsi ve düzgün olmayan dillerin bir kısmı bağlamdan bağımsız gramerler (CFG) kullanılarak oluşturulabilir.
- Düzgün diller düzgün ifadeler tarafından gösterilir ve DFA'lar tarafından kabul edilir.
- Context Free diller ise *context-free gramerler tarafından gösterilir.*

- Context Free diller *pushdown automata (yığın yapılı otomat)* tarafından kabul edilir. Yığın yapılı otomatlar yardımcı bellek olarak bir yığın kullanan gerekirci olmayan (non-deterministic) sonlu durumlu otomatlardır.
- Gerekirci (*deterministic*) pushdown automatlar context-free dillerin hepsini gösteremezler.

A **context-free grammar** (CFG) is a 4-tuple, G = (V, T, S, P), where V and T are disjoint sets, $S \in V$, and P is a finite set of rules of the form $A \rightarrow x$, where $A \in V$ and $x \in (V \cup T)^*$.

V = non-terminals or variables

T = terminals

S = Start symbol

P = Productions or grammar rules

Example

Let G be the CFG having productions:

$S \rightarrow aSa \mid bSb \mid c$

Then G will generate the language

$L = \{xcx^R \mid x \in \{a, b\}^*\}$

This is the language of odd palindromes - palindromes with a single isolated character in the middle.

Memory

- L’deki dizgileri tanıyabilecek ne tür bir belleğe ihtiyacımız vardır? Example: aaabbcbbaaa
- c’den önceki sembolleri hatırlayacak bir belleğe ihtiyaç duyarız.
- Dizginin c’ye kadar olan kısmını bir belleğe atarız (push) ve c ile karşılaştıktan sonra yığından sembolleri almaya başlarız (pop)ve bunları dizginin ortasından sonuna kadar olan herbir sembol ile karşılaştırırız.
- Eğer bunların hepsi eşleşiyorsa bu dizgi odd palindromdur.

Counting

- Example: $L = a^n c b^n$ aaaacbbbb
- Bu örnekte yine eşit sayıda a ve b olup olmadığını kontrol etmek için yığın kullanırız.
- c sembolü görülene dek yığına *a*'lar atılır ve daha sonra okunan her b sembolü için yığından bir a silinir.
- Eğer dizgi sonunda yığında hiç a kalmazsa dizgi L diline aittir.

Aşağı-itme Otomatı(Push-Down Automata)PDA

PDA içerik bağımsız gramerleri tanıyan makine modelidir. Biçimsel olarak PDA şu şekilde tanımlanır.

PDA = < Q, ∑, Γ, δ, q_0, Z_0, F>

Burada;

Q : Durumların sonlu kümesi

∑ : Giriş alfabesi

Γ : Yığın alfabesi (sonlu kümse)

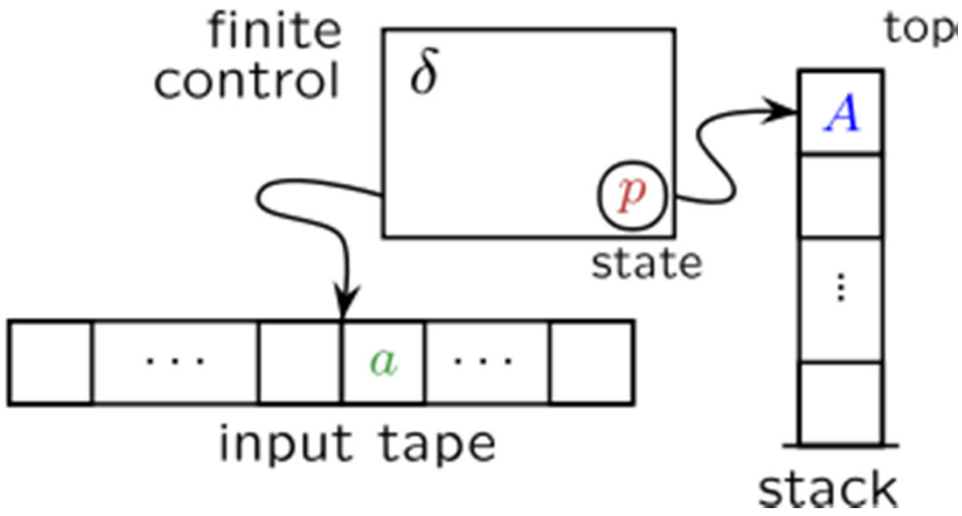
δ : Geçiş işlevi $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ (finite subsets of $Q \times \Gamma^*$)

q_0 : Başlangıç durumu ($q_0 \in Q$)

Z_0 : Yığın başlangıç simgesi ($Z_0 \in \Gamma$)

F : Kabul durumları kümesi

Bir PDA aşağıdaki gibi gösterilebilir.



Production rules

Örnek:

$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{a, b\}$

$\Gamma = \{0, 1\}$

q_0 başlangıç durumu

z = # (boş yığın gösterimi)

$F = \{q_3\}$

Production rules

δ geçiş fonksiyonu:

$\delta(q_0, a, \#) \rightarrow \{(q_1, 1\#), (q_3, \lambda)\}$

$\delta(q_0, \lambda, \#) \rightarrow \{(q_3, \lambda)\}$

$\delta(q_1, a, 1) \rightarrow \{(q_1, 11)\}$

$\delta(q_1, b, 1) \rightarrow \{(q_2, \lambda)\}$

Bu iki adım a'ların sayısını sayar ve b'lerin sayısı ile eşleştirir.

$\delta(q_2, b, 1) \rightarrow \{(q_2, \lambda)\}$

$\delta(q_2, \lambda, \#) \rightarrow \{(q_3, \lambda)\}$

This PDA is nondeterministic. Why?

Production rules

Bir sonlu durum otomatında (FSA) her bir kuraldan şu sonuç çıkarılır: Herhangi bir durumda iken bir sembol gördüğümüzde belirli bir duruma geçilir.

PDA'da ise hangi yeni duruma geçileceğine karar vermeden önce, yığındaki sembolü de bilmemiz gerekir.

Working with a stack:

- Yığının sadece en üstündeki elemana erişebiliriz.
- Yığının en üstündeki elemana erişebilmek için bu elemanı yığından çıkartırız. (POP)
- Input string'i her seferinde tek bir sembol olacak şekilde okunmalıdır.
- Otomatın o anki konfigürasyonları şunları içerir: o anki durum, girdi dizgisinde kalan karakterler ve yığının içeriği.

- Son örnekteki iki önemli geçiş:
- $\delta(q_1, a, 1) \rightarrow \{(q_1, 11)\}$, bir *a* okunduğunda yığına 1 ekler
- $\delta(q_1, b, 1) \rightarrow \{(q_2, \lambda)\}$, b okunduğunda, yığından 1 çıkarır.
- Aynı zamanda:
  $\delta(q_0, a, \#) \rightarrow \{(q_1, 1\#), (q_3, \lambda)\}$ kuralında, başlangıçta bir *a* gördüğünde q_3 kabul durumuna gidebilir.
  Tüm geçişler incelendikten sonra NPDA'nın, girilen dizgi $L=\{a^n b^n : n \geq 0\} \cup \{a\}$ diline ait olması durumunda q_3 kabul durumu ile sonlanacağı görülür.

Instantaneous description(Anlık tanım)

Given the transition function

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow (\text{finite subsets of } Q \times \Gamma^*)$$

a configuration, or *instantaneous description,* of M is a snapshot of the current status of the PDA. It consists of a triple:

(q, w, u)

where:

$q \in Q$ (Kontrol biriminin o anki durumu)

$w \in \Sigma^*$ (Girdi dizgisinin okunmayan kalan kısmı)

$u \in \Gamma^*$ (o anki yığın içeriği, en soldaki sembol yığının en üstündeki elemanı gösterir)

Instantaneous description

PDA'nın bir durumdan başka bir duruma geçmesi şu şekilde gösterilir:

$(q_1, aw, bx) \vdash (q_2, w, yx)$

Bir durumdan başka bir duruma birçok kural kullanarak geçmesi:

$(q_1, aw, bx) \vdash^* (q_2, w, yx)$

veya belirli bir PDA'ya ait geçişleri göstermek için:

$(q_1, aw, bx) \vdash_M^* (q_2, w, yx)$

Acceptance

If $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is a push-down automaton and $w \in \Sigma^*$, the string *w* is accepted by M if:

$$(q_0, w, \#) \vdash_M^* (q_f, \lambda, u)$$

for some $u \in \Gamma^*$ and some $q_f \in F$.

Bu şu anlama gelmektedir: başlangıç durumundan başlanır ve yığın boştur. *w* stringi işlendikten sonra bir kabul durumu ile sonlanır. Yığında kalan semboller önemli değildir. Buna kabul durumuyla tanıyan PDA (*acceptance by final state*) diyoruz.

Two types of acceptance

Bir de boş yığınla kabul eden (*acceptance by empty stack*) PDA var.

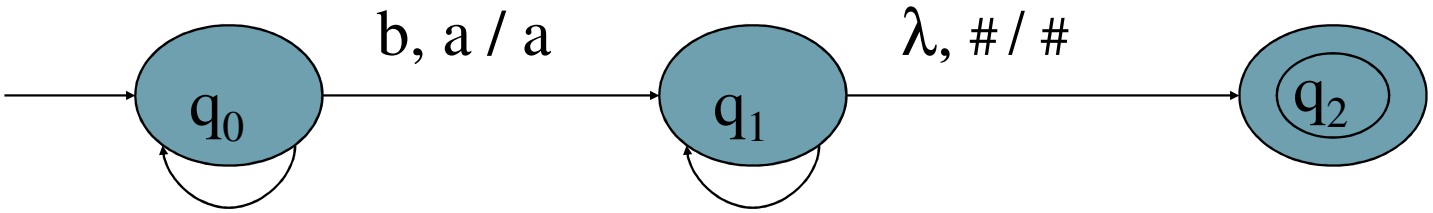
Başlangıç durumu ile başlanır, yığın boştur. *w* dizgisi işlendikten sonra yığında (# karakteri dışında) hiçbir sembol kalmaz.

Her iki tipteki kabul de birbirine eşdeğerdir. Kabul durumu ile L dilini kabul eden bir PDA oluşturabilirsek, boş yığın ile kabul eden bir PDA da bulabiliriz.

Determinism vs. non-determinism

- Gerekirci (deterministic) bir PDA'da herhangi bir girdi sembolü/yığın sembolü çifti için sadece tek bir geçiş olmalıdır.
- Non-deterministic bir PDA (NPDA)'da bir girdi sembolü/yığın sembolü çifti için geçiş tanımlanmayabilir ya da birçok geçiş tanımlanmış olabilir.
- NPDA'da, verilen bir girdi dizgisini işlemek için birçok yola olabilir. Yollardan bazıları verilen dizgiyi kabul edebilir. Diğer yollar ise kabul durumu olmayan bir durum ile sonlanabilir.
- Bir NPDA herhangi bir dizginin kabul edilmesi için otomatın takip etmesi gereken yolu tahmin edebilir.

Example: $a^n bc^n$



a, # / a#

a, a / aa

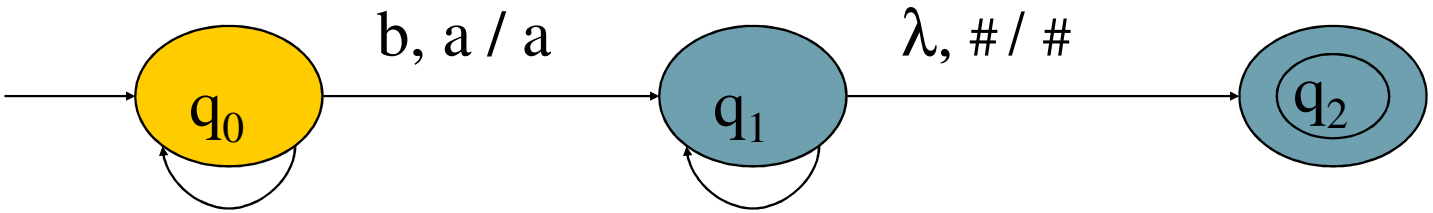
c, a / λ

$L = \{a^n bc^n \mid n>0\}$

Production rules for $a^n bc^n$

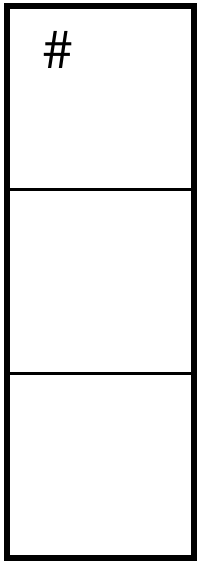
Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	$(q_0, a\#)$
2	q_0	a	a	(q_0, aa)
3	q_0	b	a	(q_1, a)
4	q_1	c	a	(q_1, λ)
5	q_1	λ	#	$(q_2, \#)$

Example: aabcc



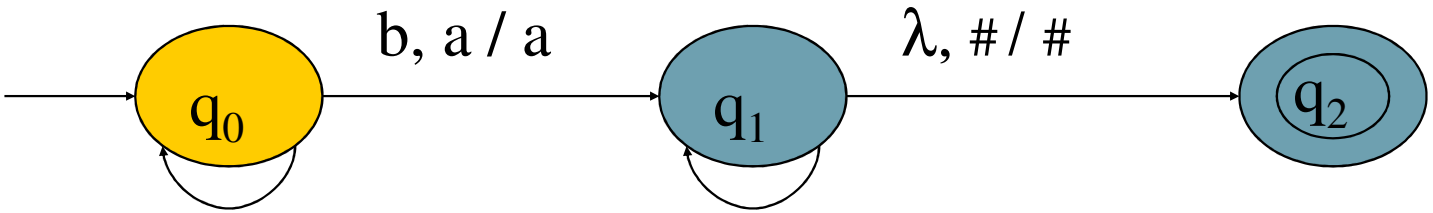
a, # / a#
a, a / aa

c, a / λ



a a b c c

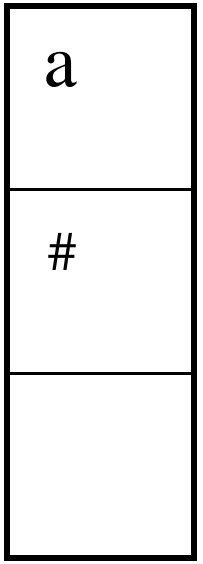
Example: aabcc



a, #/ a#

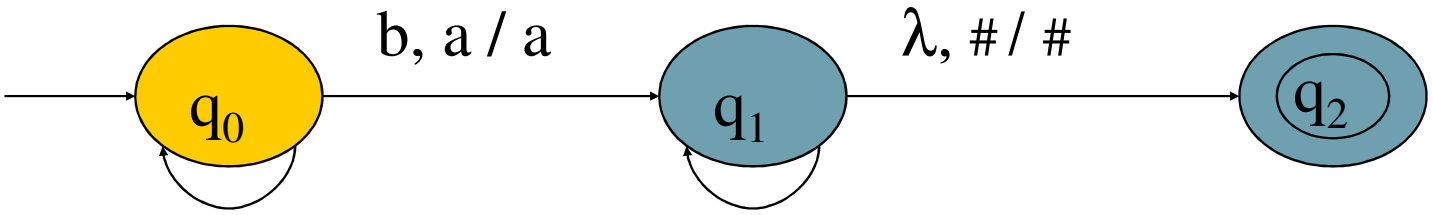
a, a / aa

c, a / λ



a b c c

Example: aabcc



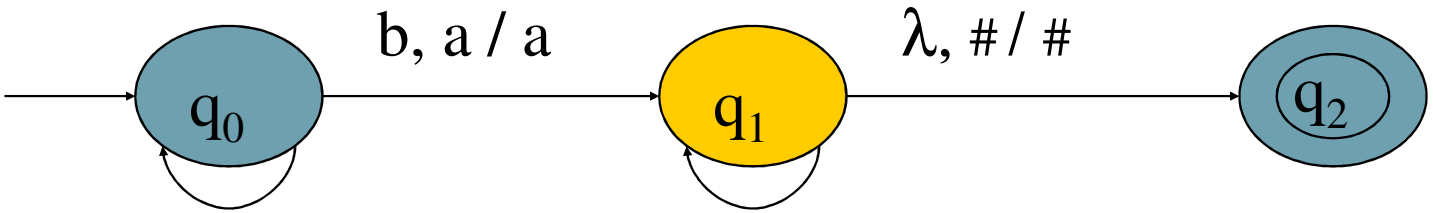
a, # / a#
a, a / aa

c, a / λ

b c c

a
a
#

Example: aabcc



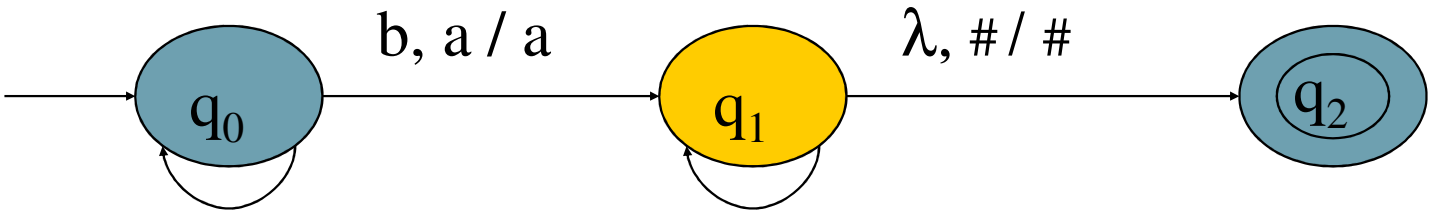
a, # / a#
a, a / aa

c, a / λ

a
a
#

c c

Example: aabcc



a, # / a#

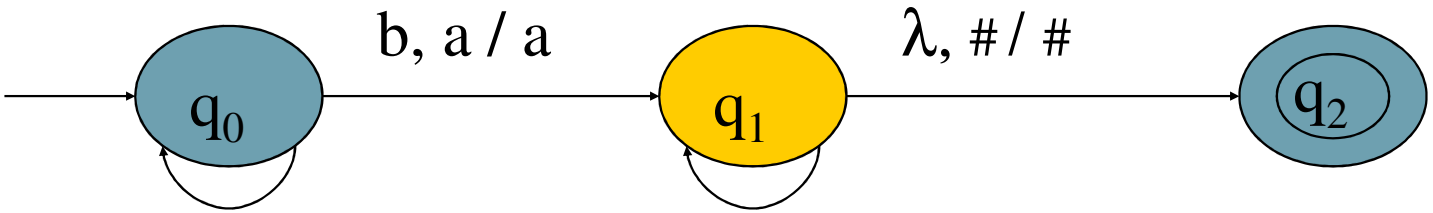
a, a / aa

c, a / λ

c

a
#

Example: aabcc

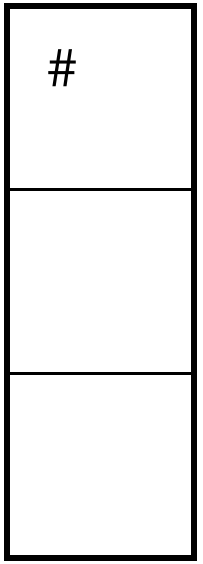


a, # / a#

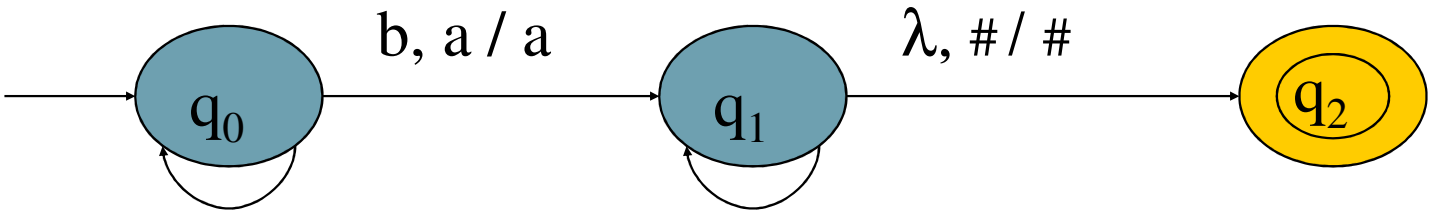
a, a / aa

c, a / λ

λ



Example: aabcc

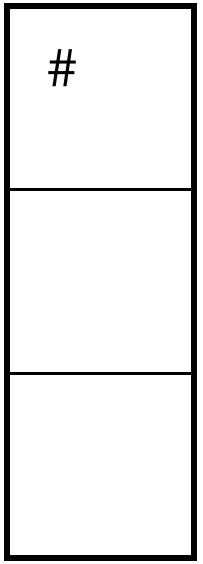


a, # / a#

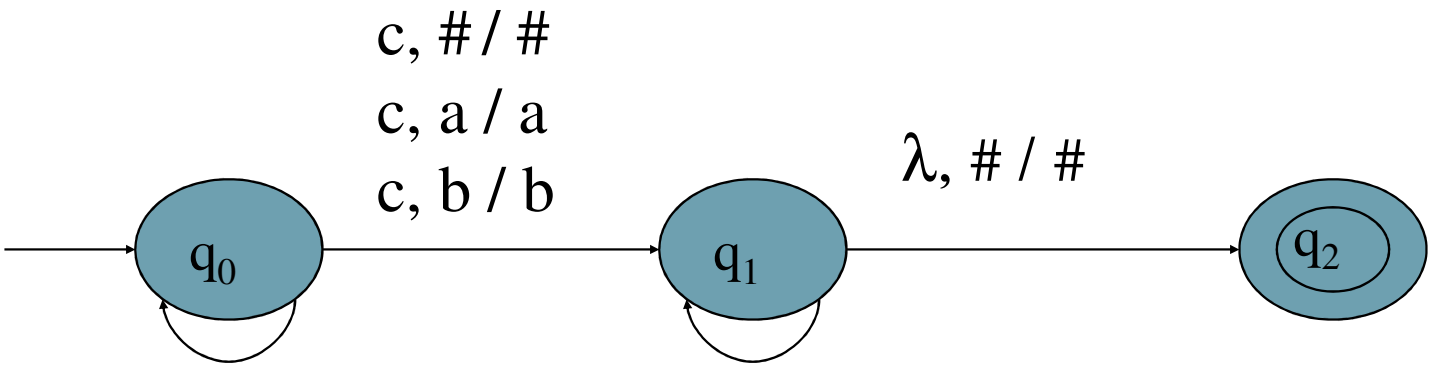
a, a / aa

c, a / λ

λ



Example: Odd palindrome



$L = \{xcx^R \mid x \in \{a, b\}^*\}$

Production rules for Odd palindromes

Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	(q_0, a#)
2	q_0	b	#	(q_0, b#)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_0, ba)
5	q_0	a	b	(q_0, ab)
6	q_0	b	b	(q_0, bb)
7	q_0	c	#	(q_1, #)
8	q_0	c	a	(q_1, a)
9	q_0	c	b	(q_1, b)
10	q_1	a	a	(q_1, λ)
11	q_1	b	b	(q_1, λ)
12	q_1	λ	#	(q_2, #)

Processing *abcba*

Rule #	Resulting state	Unread input	Stack
(initially)	q_0	abcba	#
1	q_0	bcba	a#
4	q_0	cba	ba#
9	q_1	ba	ba#
11	q_1	a	a#
10	q_1	-	#
12	q_2	-	#
	accept		

Processing *ab*

Rule #	Resulting state	Unread input	Stack
(initially)	q_0	ab	#
1	q_0	b	a#
4	q_0	-	ba#
	crash		

Processing *acaa*

Rule #	Resulting state	Unread input	Stack
(initially)	q_0	acaa	#
1	q_0	caa	a#
8	q_1	aa	a#
10	q_1	a	#
12	q_2	a	#
	crash		

Crashing

- What is happening in the last example?
- Bu örnekte *acaa* dizgisin ilk 3 karakteri işlendikten sonra q_1 durumundayız ve input string'inin hala işlenmemiş bir sembolü (*a*) var.
- Yığının en üstünde boş-yığın işareti var.
- Kural 12'ye göre, q_1 durumunda iken yığında # varsa, λ-geçişi ile q_2 durumuna geçebiliriz.

Crashing

- Bu nedenle otomat *aca* dizgisini kabul eder.
- Ancak q_2 durumundayız ve hala işlenmesi gereken son bir *a* vardır.
- Fakat böyle bir kural tanımlı değildir.
- Bir sonraki harekette *a* sembolünü işlemeye çalıştığımızda hata olur ve *acaa* dizgisi reddedilir.

Example: Even palindromes

Consider the following context-free language:

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

This is the language of all even-length palindromes over {a, b}.

$L = \{ww^R \mid w \in \{a, b\}^*\}$

- Semboller yığına girdiklerinin ters sırasında yığından çıkarılırlar. Bu özelliği kullanarak dizginin ilk kısmını okuduğumuzda sembolleri sırasıyla yığına atarız ve ikinci kısım için de o anki girdi sembolünü yığının en üstündeki sembol ile karşılaştırırız. Buradaki zorluk, dizginin ortasını bilmememiz. Yani w nerede sonlanıyor, w^R nerede başlıyor. Ancak otomatın gerekirci olmayan yapısı bize bu konuda yardım eder. NPDA dizginin ortasını doğru bir şekilde tahmin eder.

Production rules for Even palindromes

Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	(q_0, a#)
2	q_0	b	#	(q_0, b#)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_0, ba)
5	q_0	a	b	(q_0, ab)
6	q_0	b	b	(q_0, bb)
7	q_0	λ	#	(q_1, #)
8	q_0	λ	a	(q_1, a)
9	q_0	λ	b	(q_1, b)
10	q_1	a	a	(q_1, λ)
11	q_1	b	b	(q_1, λ)
12	q_1	λ	#	(q_2, #)

Example: Even palindromes

This PDA is non-deterministic.

Note moves 7, 8, and 9. Here the PDA is "guessing" where the middle of the string occurs. If it guesses correctly (and if the PDA doesn't accept any strings that aren't actually in the language), this is OK.

Example: Even palindromes

baab dizgisinin işlenmesi:

$(q_0, baab, \#) \vdash (q_0, aab, b\#)$
$\vdash (q_0, ab, ab\#)$
$\vdash (q_1, ab, ab\#)$
$\vdash (q_1, b, b\#)$
$\vdash (q_1, \lambda, \#)$
$\vdash (q_2, \lambda, \#)$ (accept)

Example: All palindromes

Consider the following context-free language:

$$L = pal = \{x \in \{a, b\}^* \mid x = x^R\}$$

This is the language of all palindromes, both odd and even, over {a, b}.

Production rules for All palindromes

Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	$(q_0, a\#), (q_1, \#)$
2	q_0	b	#	$(q_0, b\#), (q_1, \#)$
3	q_0	a	a	$(q_0, aa), (q_1, a)$
4	q_0	b	a	$(q_0, ba), (q_1, a)$
5	q_0	a	b	$(q_0, ab), (q_1, b)$
6	q_0	b	b	$(q_0, bb), (q_1, b)$
7	q_0	λ	#	$(q_1, \#)$
8	q_0	λ	a	(q_1, a)
9	q_0	λ	b	(q_1, b)
10	q_1	a	a	(q_1, λ)
11	q_1	b	b	(q_1, λ)
12	q_1	λ	#	$(q_2, \#)$

Production rules for All palindromes

Dizginin ikinci yarısını işlemeye başlamadan önceki her noktada 3 olasılık vardır:

1. Bir sonraki girdi karakteri hala dizginin ilk yarısındadır ve yığına atılması gerekir.

2. Bir sonraki girdi karakteri tek uzunluktaki dizginin ortadaki sembolüdür ve okunup, atılmalıdır. (yığına kaydetmeye gerek yok çünkü herhangi bir şey ile karşılaştırmayacağız)

3. Bir sonraki karakter çift uzunluklu dizginin ikinci yarısının ilk karakteridir.

Production rules for All palindromes

Why is this PDA non-deterministic?

Bu NPDA’nın ilk 6 kuralında seçilebilecek iki hareket vardır. Bu nedenle gerekirci değildir.

Production rules for All palindromes

- Bir PDA'nın her hareketinde üç ö koşul vardır: o anki durum, girdi dizgisinden işlenecek bir sonraki karakter ve yığının en üstündeki karakter.
- Kural 1'de, mevcut durum q_0, girdi dizgisinde işlenecek bir sonraki karakter *a* ve yığının en üstündeki karakter de boş-yığın göstergesidir. Bu kural için iki olası hareket vardır:

1) q_0 durumunda kalmak ve yığına *a#* atmak

ya da

2) q_1 durumuna gitmek ve yığına # atmak

Verilen önkoşullar kümesinden birden fazla olası hareket varsa otomat gerekirci değildir (nondeterminism)

Eğer M deterministic ise tek bir girdi/yığın konfigürasyonu için birden fazla harekete izin verilmez. Yani:

- yığın = Y ve input = X için, aynı durumdan aynı yığın değeri ve aynı girdi için başka bir hareket olamaz.
- λ-türetimler olabilir, FAKAT input = λ yığın = X için, aynı durumdan yığın = X iken başka bir hareket alamaz.

Non-determinism

- Başlangıçta non-deterministic olarak tanımlanan bazı PDA'lar aynı zamanda deterministic PDA olarak da tanımlanabilir.
- Fakat bazı CFL'ler doğası gereği non-deterministic'tir, örneğin:

Given L = pal = $\{x \in \{a, b\}^* \mid x = x^R\}$,

then L cannot be accepted by any DPDA.

Example:

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

This is the set of all strings over the alphabet $\{a, b\}$ in which the number of *a*'s is greater than the number of *b*'s. This can be represented by either an NPDA or a DPDA.

Example (NPDA):

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

Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	$(q_0, a\#)$
2	q_0	b	#	$(q_0, b\#)$
3	q_0	a	a	(q_0, aa)
4	q_0	b	b	(q_0, bb)
5	q_0	a	b	(q_0, λ)
6	q_0	b	a	(q_0, λ)
7	q_0	λ	a	(q_1, a)

Example (NPDA):

What is happening in this PDA?

We start, as usual, in state q_0. If the stack is empty, we read the first character and push it onto the stack. Thereafter, if the stack character matches the input character, we push both characters onto the stack. If the input character differs from the stack character, we throw both away. When we run out of characters in the input string, then if the stack still has an *a* on top, we make a free move to q_1 and halt; q_1 is the accepting state.

Example (NPDA):

Why is it non-deterministic?

Rules 6 and 7 both have preconditions of: the starting state is q_0 and the stack character is *a*. But we have two possible moves from here, one of them if the input is a *b*, and one of them any time we want (a λ-move), including if the input is a *b*. So we have two different moves allowed under the same preconditions, which means this PDA is non-deterministic.

Example (DPDA):

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

Rule #	State	Input	Top of Stack	Move(s)
1	q_0	a	#	$(q_1, \#)$
2	q_0	b	#	$(q_0, b\#)$
3	q_0	a	b	(q_0, λ)
4	q_0	b	b	(q_0, bb)
5	q_1	a	#	$(q_1, a\#)$
6	q_1	b	#	$(q_0, \#)$
7	q_1	a	a	(q_1, aa)
8	q_1	b	a	(q_1, λ)

Example (DPDA):

What is happening in this PDA?

Here being in state q_1 means we have seen more *a*'s than *b*'s. Being in state q_0 means we have **not** seen more *a*'s than *b*'s. We start in state q_0.

If we are in state q_0 and read a *b*, we push it onto the stack. If we are in state q_1 and read an *a*, we push it onto the stack. Otherwise we don't push *a*'s or *b*'s onto the stack. Any time we read an *a* from the input string and pop a *b* from the stack, or vice versa, we throw the pair away and stay in the same state.

When we run out of characters in the input string, then we halt; q_1 is the accepting state.

Aşağı-itme Otomatı

Örnek olarak çok klasik bir makine olan 0^n1^n probleminin çözüm makinesini PDA olarak göstermek isteyelim. Diğer bir deyişle makine bir giriş kelimesini alacak ve bu kelimedeki 0′ların sayısı 1′lerin sayısına eşitse ve 0'lar 1′lerden önce geliyorsa bu kelimeyi kabul edecek, eğer 0′ların sayısı ve 1′lerin sayısı eşit değil veya sıralamada bir hata varsa bu girdiyi kabul etmeyecektir. Örneğin;

ε : kabul, n= 0 için doğru (burada **ε** sembolü ile boş girdi kastedilmiştir)
01 : kabul , n = 1 için doğru
10 : ret , sıralama hatası, 0′lar 1′lerin önünde olmalı
0011: kabul n = 2 için doğru
0101 : ret, sıralama hatası , bütün 0′lar, 1′lerin önünde olmalı
00011: ret, 0′ların sayısı ile 1′lerin sayısı tutmuyor

Aşağı-itme Otomatı

Bu makinenin tasarımı için bir yığın kullanılacaktır. Eğer makineye gelen 0′ları sırasıyla koyarsak (push) ve 0′lar bittikten sonra gelen her 1 için yığından bir eleman alırsak (pop) bu durumda makine girdi kelimesi bittiğinde yığını boş bulursa (yığındaki harfler ile girdideki harfler aynı anda biterse) kelimeyi kabul edecek aksi halde reddedecektir.

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

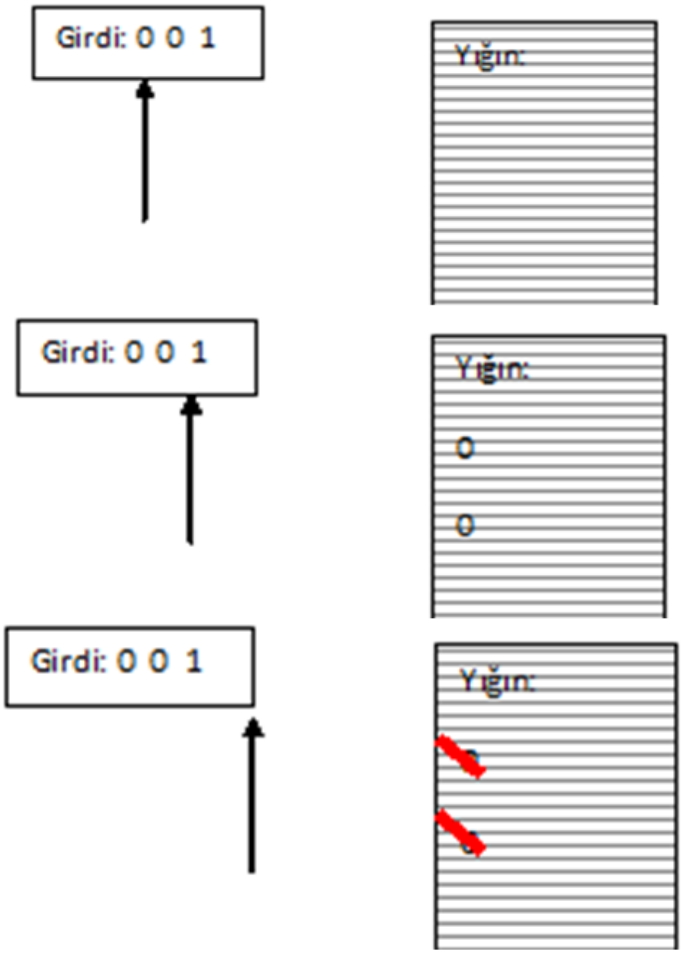
Input:	0			1			ϵ		
Stack:	0	$	ϵ	0	$	ϵ	0	$	ϵ
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \epsilon)\}$					
q_3				$\{(q_3, \epsilon)\}$				$\{(q_4, \epsilon)\}$	
q_4									

Aşağı-itme Otomatı

Bu tasarımı bir iki örnek ile anlamaya çalışalım. Örneğin girdimiz 0011 olsun.

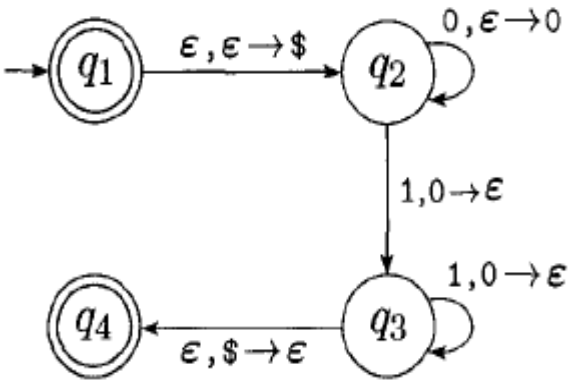
İlk durumda makinemiz girdinin ilk harfi olan 0′ı okuyacak ve 0 gördükçe yığına koyacak (push)

Yığına 0′ları koyduktan sonra her gördüğü 1 için yığından bir 0 çıkaracak (pop):



Aşağı-itme Otomatı

Görüldüğü üzere 2 adet 1 harfi için 2 adet 0 harfi yığından çıkarılmıştır (pop) ve sonuçta girdi kelimenin sonuna ulaşılmış ve yığında aynı anda boşalmıştır. Dolayısıyla 0011 kelimesini kabul ederiz. Eğer bu makine bir sonlu durum makinesi olarak çizilirse aşağıdaki şekil elde edilir.

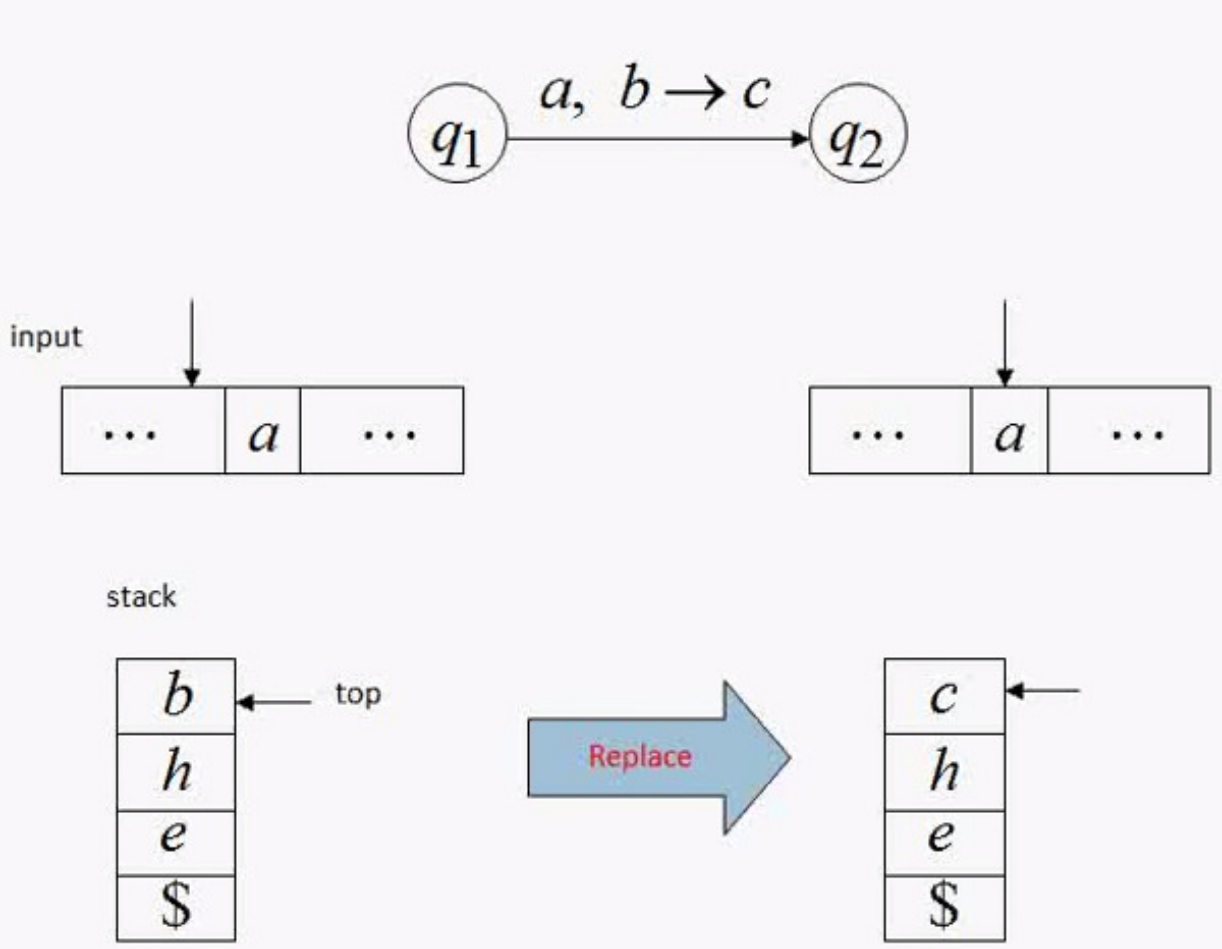


State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

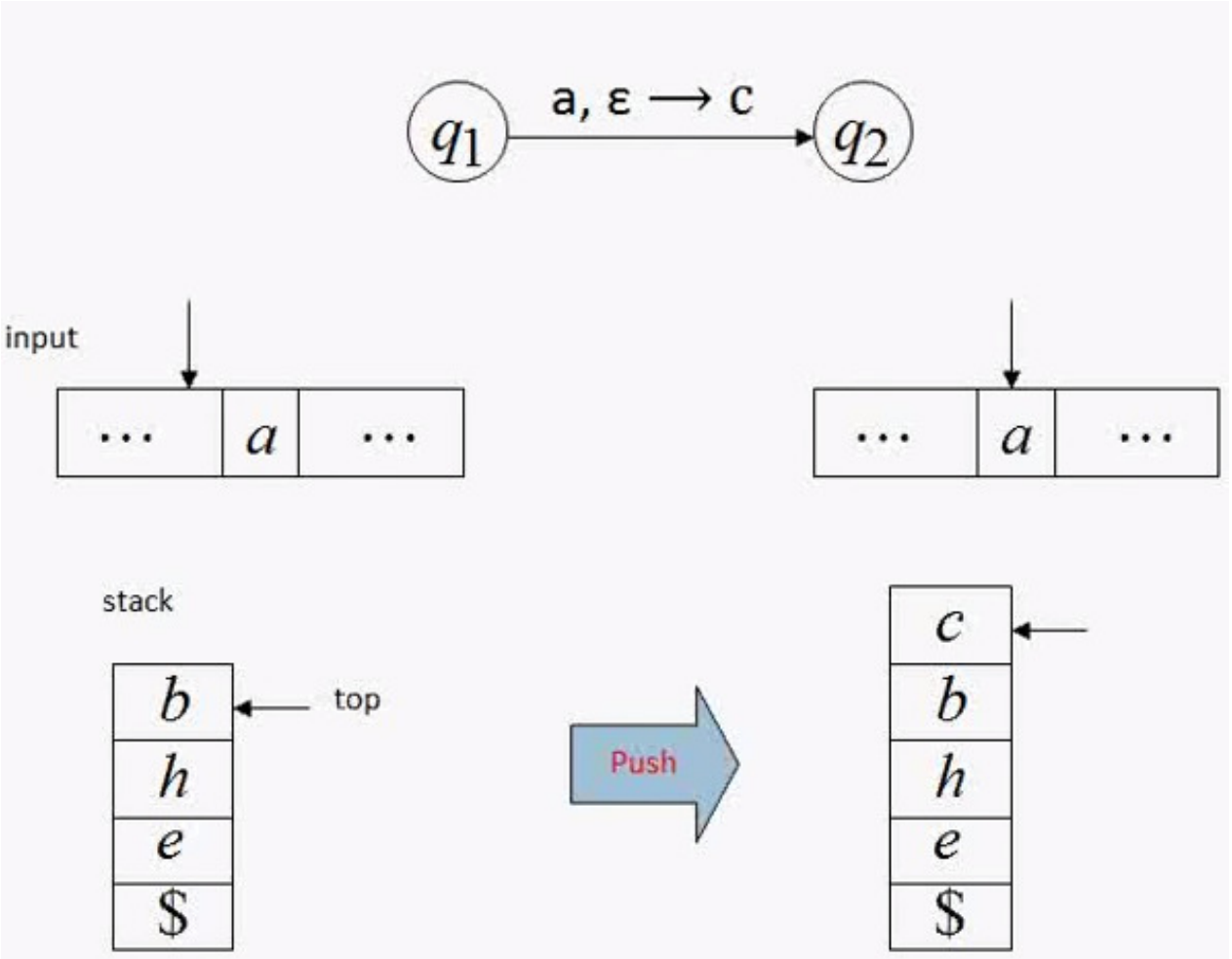
Yandaki makine tasarımımızı kısaca gözden geçirecek olursak. Kabul edilen durumlar q_1 ve q_4 durumlarıdır. Yani e (boş kelime) durumunu kabul için q_1, diğer kabul edilir durumlar için de q_4 durumu (state) tasarlanmıştır. Tasarımda bulunan q_2 durumu geçiş durumudur. Yani q_2 durumunda bulunduğu sürece makine girdiden bir harf okumakta ve bu harf 0 olmaktadır. Okunan harf 0 olduğu sürece de bu değer yığına konulmaktadır (push). Bu durum (yani q_2 durumu) girdiden bir harf olarak 1 geldiğinde bozulur. Şayet girdiden 1 harfi okunursa bu defa durum değiştirilerek q_3 durumuna geçilir ve bu q_3 durumunda da girdiden 1 harfi okundukça yığından 0 harfi çıkarılır (pop).

Son durumda şayet girdi boşsa ve yığın da boşsa q_4 durumuna yani kabul durumuna geçilir. Bunun dışındaki ihtimallerde q_2 ya da q_3 gibi kabul edilmeyen bir duruma takılır ve makine girdiyi reddeder.

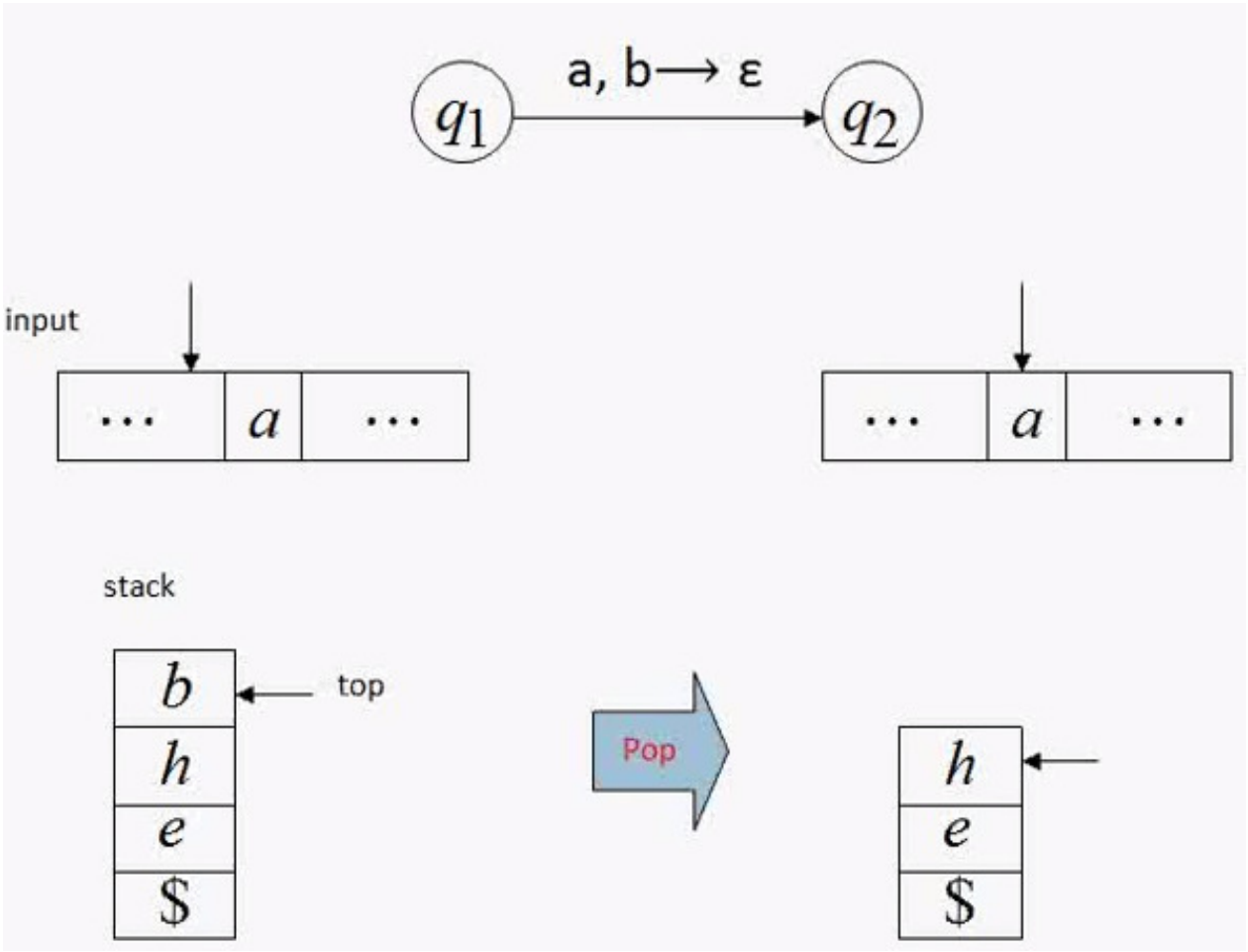
Aşağı-itme Otomatı



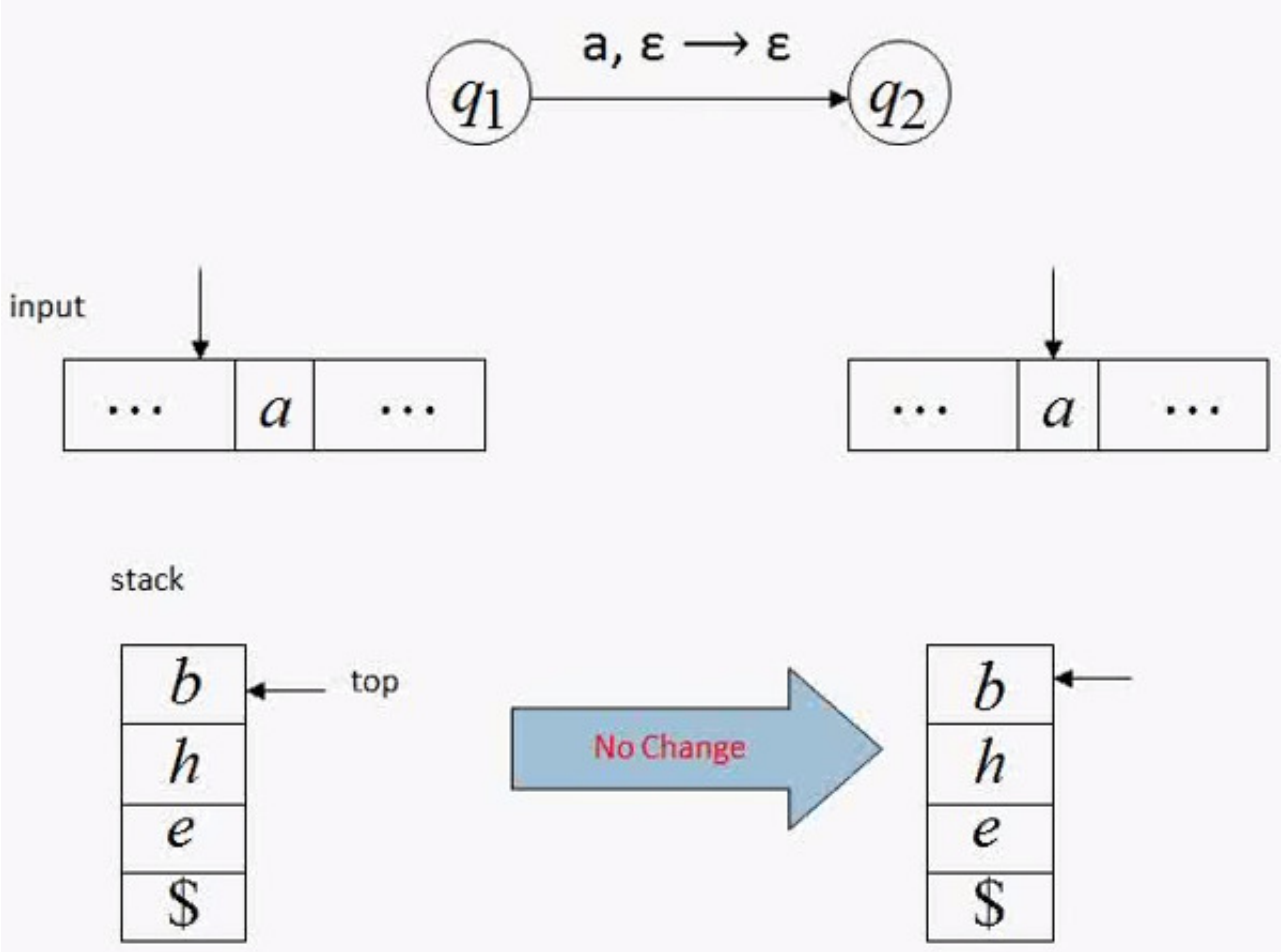
Aşağı-itme Otomatı



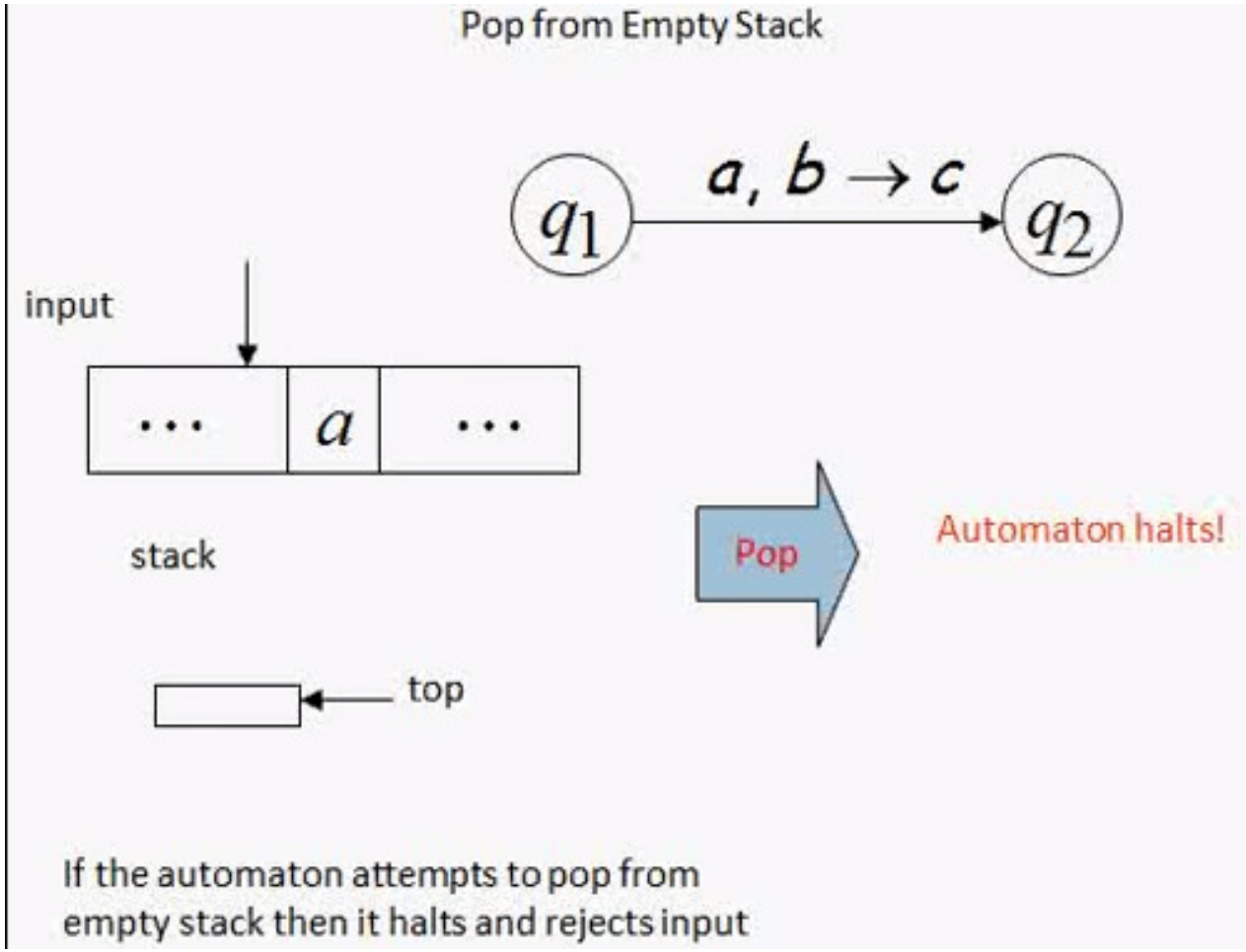
Aşağı-itme Otomatı



Aşağı-itme Otomatı



Aşağı-itme Otomatı



Aşağı-itme Otomatı

PDA Accept – Reject Status

- The PDA accepts when there exists a computation path such that:
  - The computation path ends in an accept state
  - All the input is consumed
  - (no requirement for the stack)
- The PDA rejects when all the paths:
  - Either end in a non-accepting state
  - Or are incomplete (meaning that at some point there is no possible transition under the current input and stack symbols)

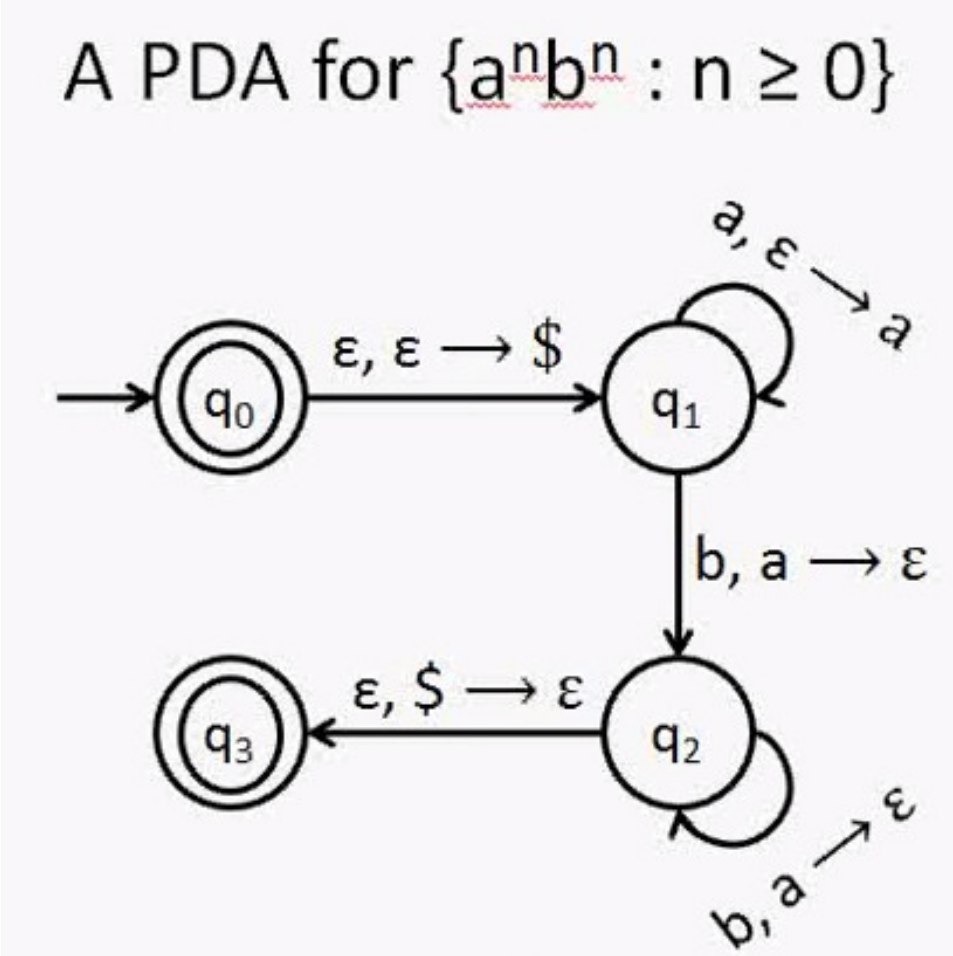
Aşağı-itme Otomatı

Is the stack empty?

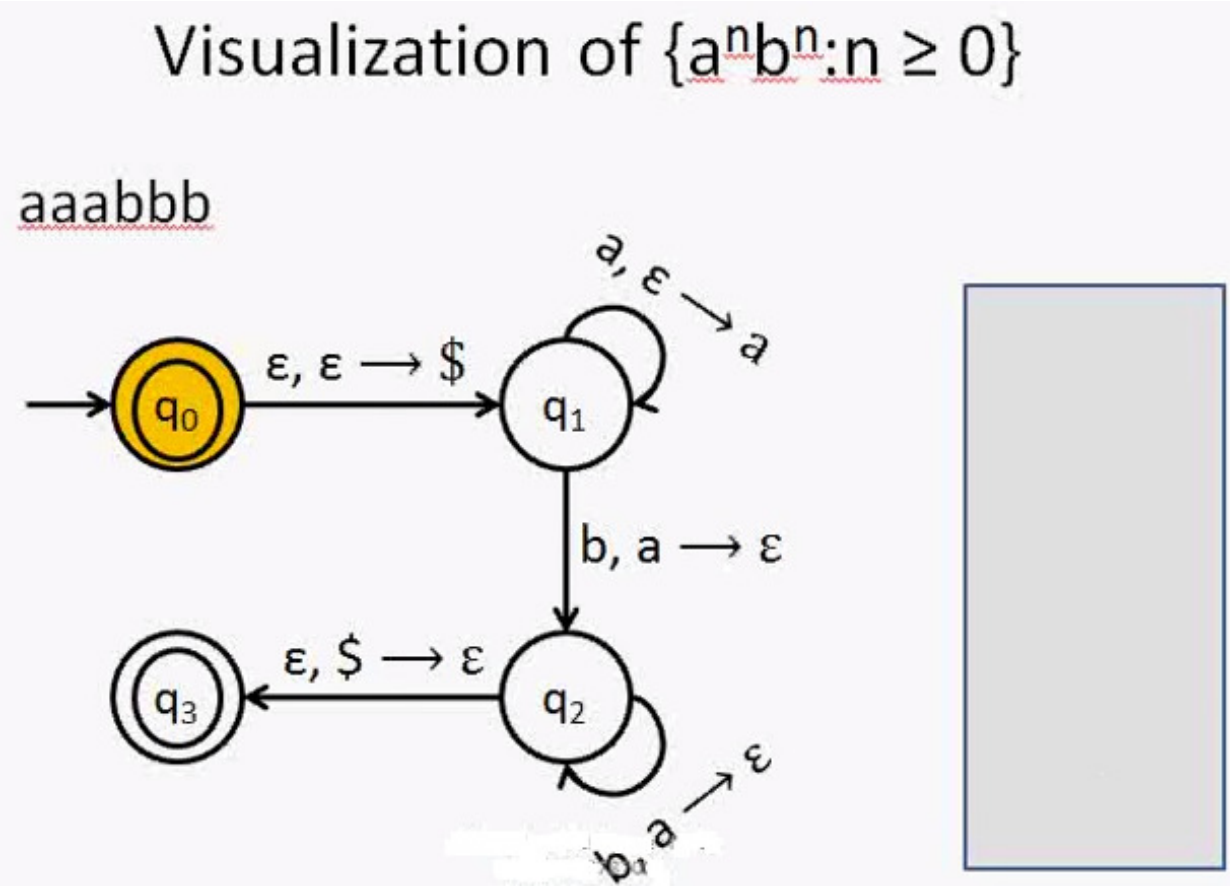
How can you check if the stack is empty?

- What we usually do is to place a special symbol (for example a $) at the bottom of the stack.
- Whenever we find the $ again we know that we reached the end of the stack.
- **In order to accept a string there is no need for the stack to be empty.**

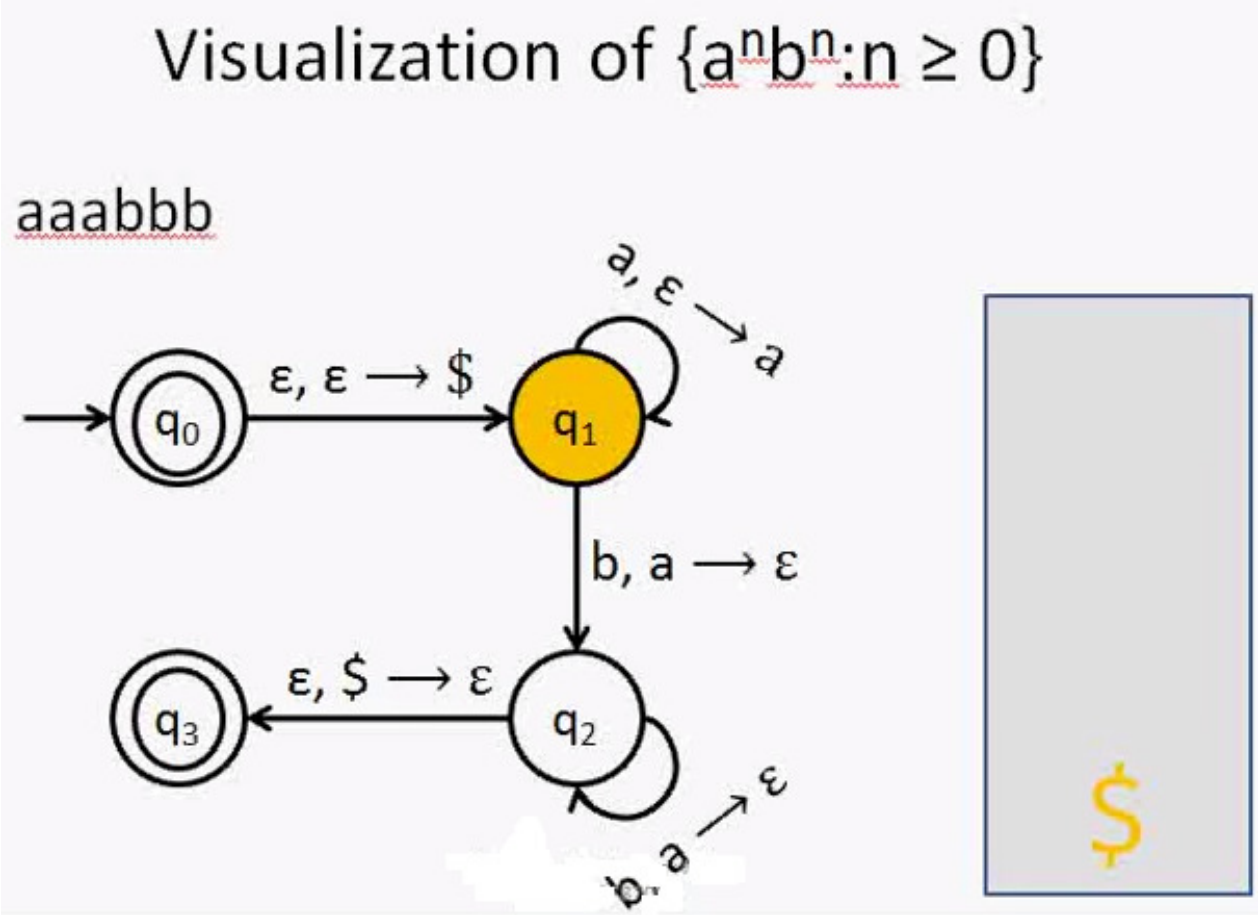
Aşağı-itme Otomatı



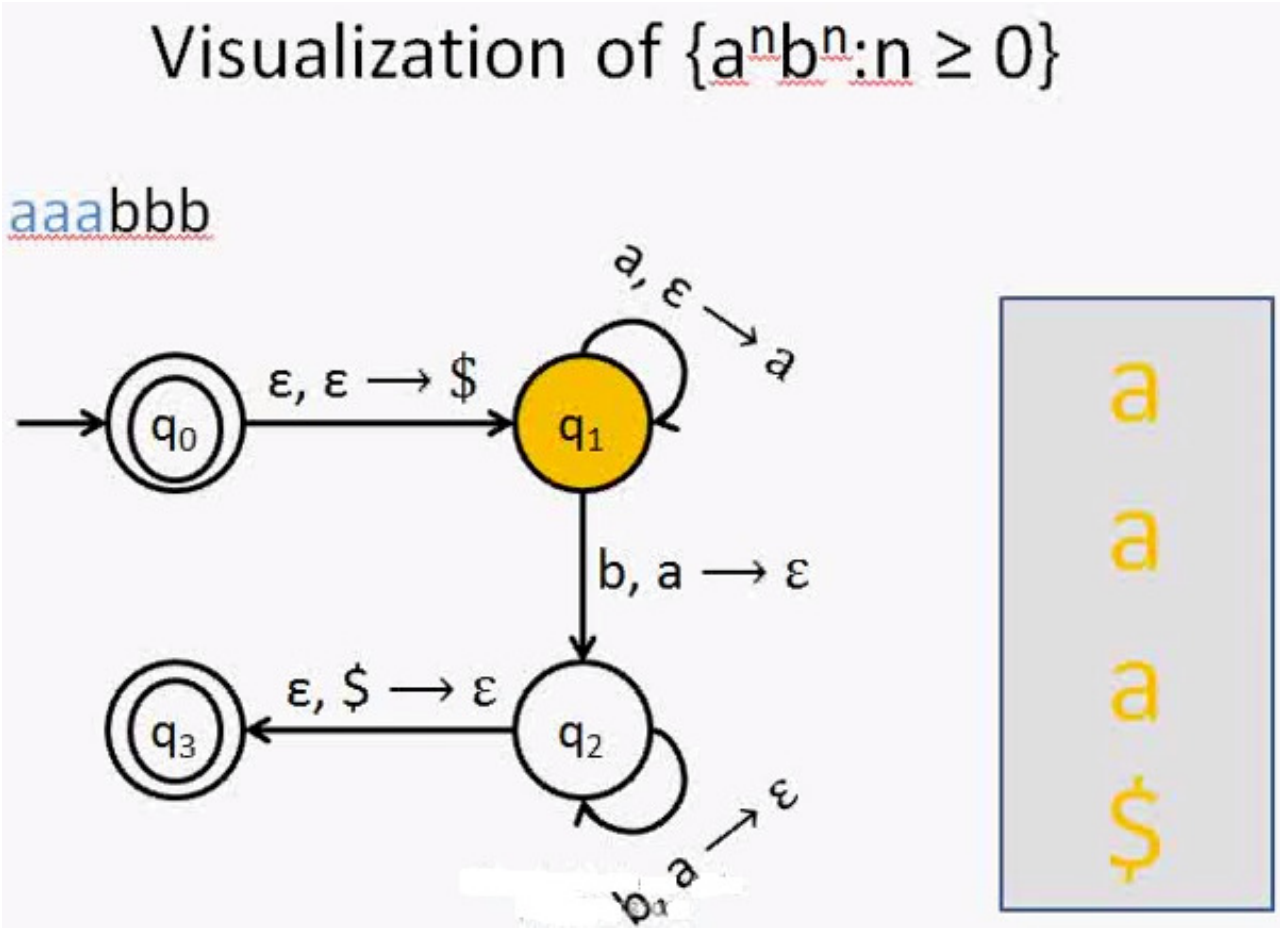
Aşağı-itme Otomatı



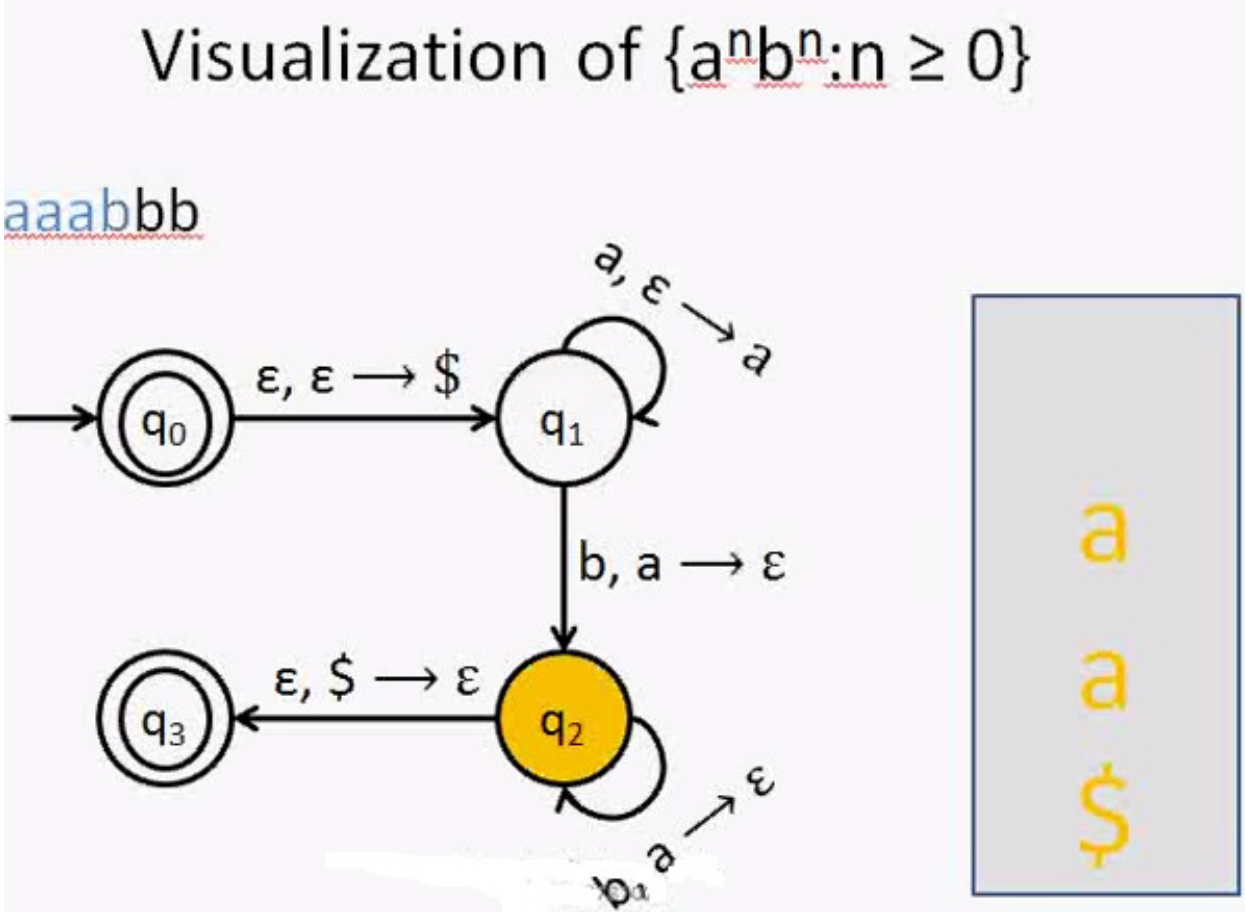
Aşağı-itme Otomatı



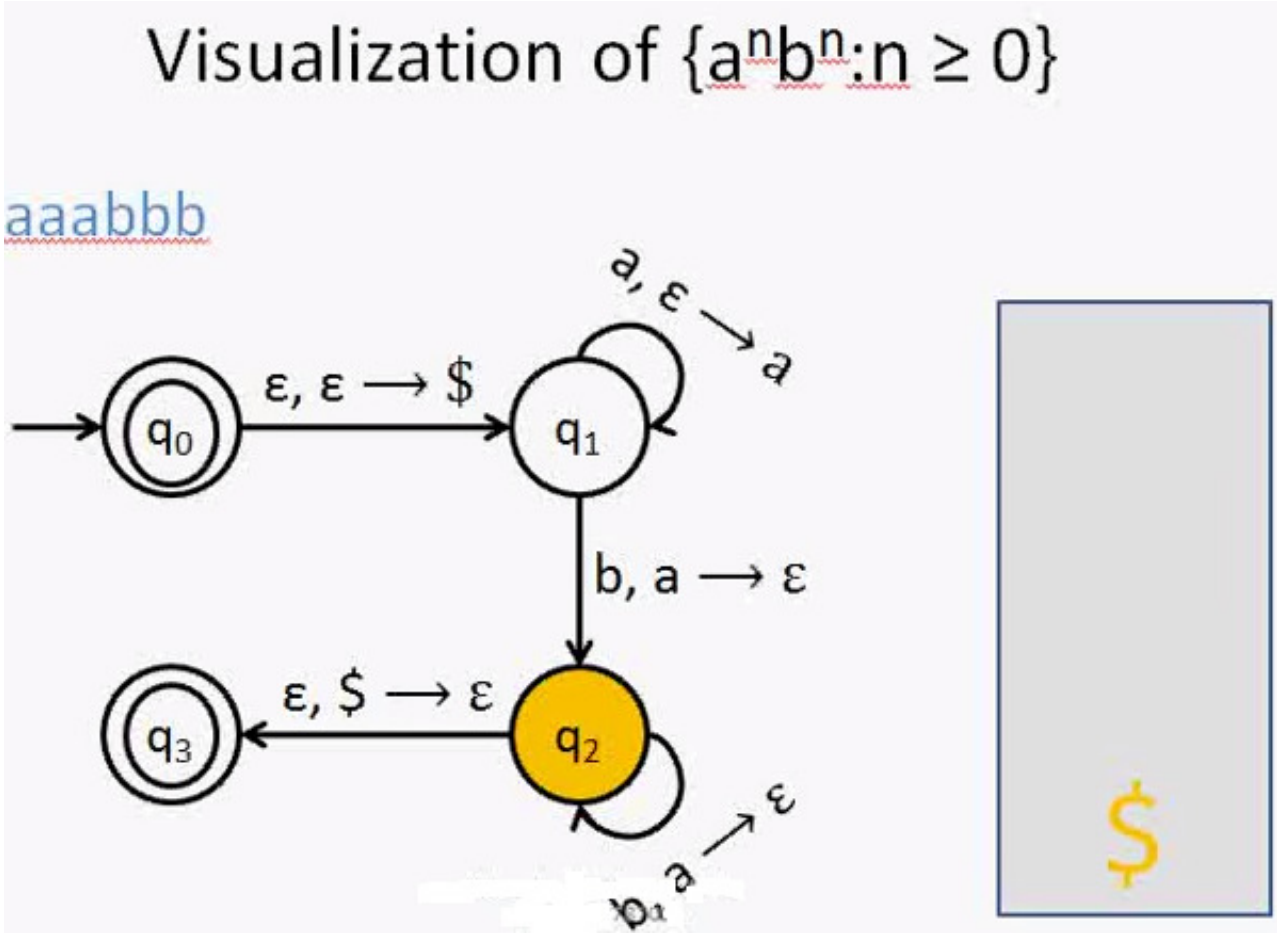
Aşağı-itme Otomatı



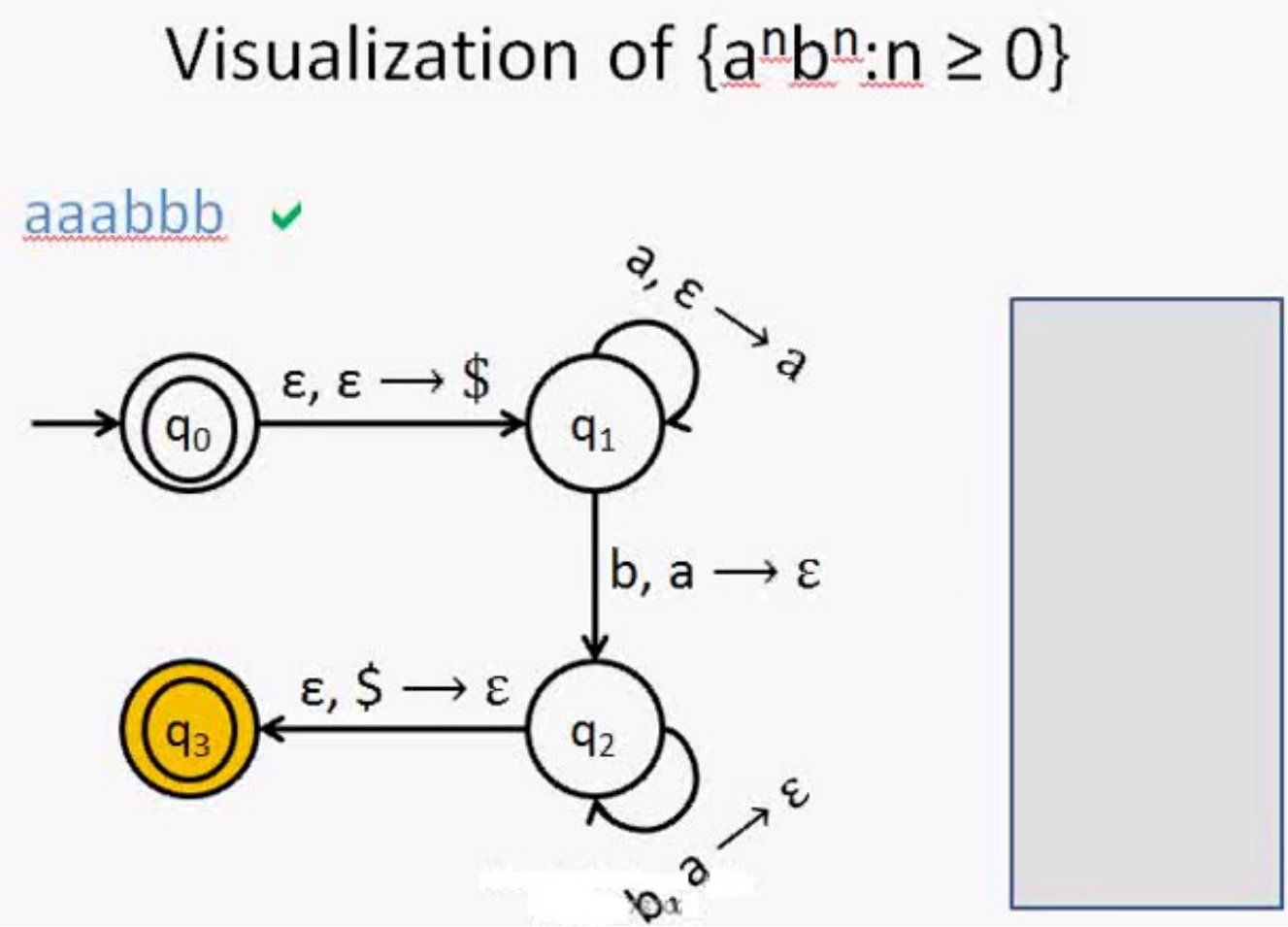
Aşağı-itme Otomatı



Aşağı-itme Otomatı



Aşağı-itme Otomatı



PDA'nın Soyut Makine Modeli

PDA makinesinin iki türlü hareketi vardır;

Normal Hareket:

- Şeritten bir simge okunur
- Okuma kafası bir sağ hücreye geçer
- Yığının en üstündeki simge silinir (pop).
- Yığının üstüne, yığın simgelerinden oluşan bir dizi eklenir (push). Yığına eklenen dizi boş (ε) da olabilir.

ε Hareketi:

- Şeritten simge okunmaz ve okuma kafası yer değiştirmez. Dolayısıyla ε hareketi şeritteki simgeden bağımsız bir harekettir.
- Yığının en üstündeki simge silinir (pop)
- Yığının üstüne yığın simgelerinden oluşan bir dizi eklenir (push)

PDA'nın Tanıdığı Dil

Her PDA giriş alfabesindeki simgelerden oluşan dizilerin bir kısmını tanır bir kısmını tanımaz. Dizileri tanıma açısından PDA'nın iki alt modeli vardır.

- *Kabul Durumuyla Tanıyan PDA:* Eğer bir dizinin tüm simgeleri okunduktan sonra PDA bir kabul durumda bulunursa bu dizi PDA tarafından tanınır. Dizinin tüm simgeleri okunduktan sonra yığın içeriğine bakılmaz.

- *Boş Yığınla Tanıyan PDA:* Dizinin tüm simgeleri okunduktan sonra eğer yığın boşalmışsa dizi PDA tarafından tanınır.

PDA'nın Tanıdığı Dil

Örnek: $L=\{ww^R \mid w \in (0 \mid 1)^*\}$ tanımlanmıştır.
Bu dili türeten gramer;

$V_N = \{S\}$

$V_T = \{0, 1\}$

P: $S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$

Bu dili tanıyan PDA:

$M = < Q, \sum, \Gamma , \delta , q_0, Z_0, F >$

$Q = \{q_0, q_1\}$

$\sum = \{0, 1,\}$

$\Gamma = \{A, B, Z_0\}$

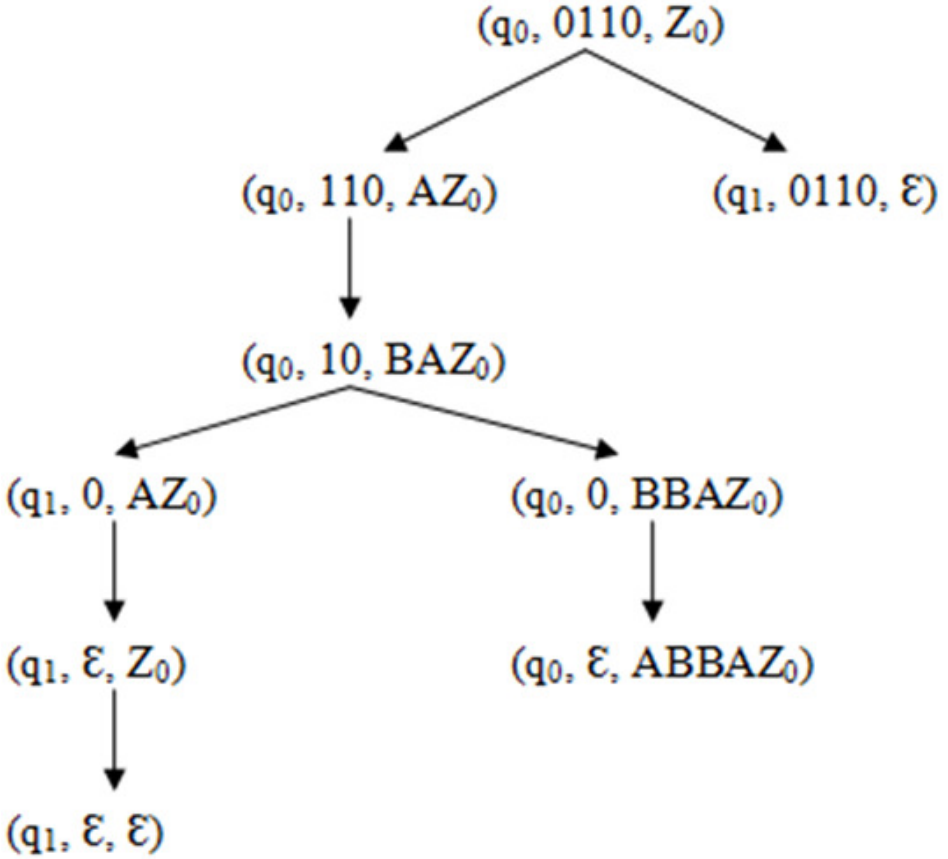
PDA'nın çalışma ilkesi:

- w'nun simgelerini okur ve yığına okuduğu her 0 için A, 1 için B ekler. PDA'nın yığın alfabesi giriş alfabesindeki simgeleri içerebileceği gibi bu simgeleri içermeyebilir de.
- w'nun bitip w^R'nin başladığını belirten özel bir simge yoktur. Okunan simge eğer bir önceki simgenin aynısı ise bu simge w'nun bir sonraki simgesi olabileceği gibi w^R'nin ilk simgesi de olabilir. Bu koşullarda iki hareket tanımlamak gerekir. Birinci harekette PDA ekleme durumunda kalır (q_0) ve yığına ekleme yapar. İkinci harekette PDA simge durumuna (q_1) geçer ve yığının tepesindeki simgeyi siler.

PDA'nın Tanıdığı Dil

δ :

$\delta(q_0, 0, Z_0) = (q_0, AZ_0)$
$\delta(q_0, 1, Z_0) = (q_0, BZ_0)$
$\delta(q_0, \varepsilon, Z_0) = (q_1, \varepsilon)$
$\delta(q_0, 0, A) = \{(q_0, AA), (q_1, \varepsilon)\}$
$\delta(q_0, 1, A) = (q_0, BA)$
$\delta(q_0, 0, B) = (q_0, AB)$
$\delta(q_0, 1, B) = \{(q_0, BB), (q_1, \varepsilon)\}$
$\delta(q_1, 0, A) = (q_1, \varepsilon)$
$\delta(q_1, 1, B) = (q_1, \varepsilon)$
$\delta(q_1, \varepsilon, Z_0) = (q_1, \varepsilon)$



W=0110 dizisinin bu PDA tarafından tanınması $(q_0, 0110, Z_0)$ başlangıç ifadesinden $(q_1, \varepsilon, \varepsilon)$ ifadesine ulaşılabildiği için makine 0110 dizisini tanır.