BIMU4032
DERLEYİCİ TASARIMI

Bu Haftaki Konu Başlıkları

- **Bir Yukarıdan Aşağıya Ayrıştırıcıyı Otomatik Oluşturma**
- **Aşağı-itme Otomatı**
- **PDA'nın Soyut Makine Modeli**
- **PDA'nın Tanıdığı Dil**
- **LL(k) Ayrıştırmada Kullanılacak PDA Modeli**
- **Deterministik Ayrıştırma**
- **LL(k) Grameri**
- **LL(k) Ayrıştırma**
- **LL(k) Ayrıştırma Çizelgesinin Oluşturulması**

Bir Yukarıdan Aşağıya Ayrıştırıcıyı Otomatik Oluşturma

Otomatik oluşturulan bir yukarıdan aşağıya ayrıştırıcının temel ilkeleri ön hesaplama yapılarak el ile yazılması işlemlerinden çıkartılabilir. Bu yöntemde izin verilen gramer LL(k) gramer olarak adlandırılır. LL(k) ayrıştırma mekanizması aşağı-itme otomatına (Push-Down Automata) dayanır.

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 : Durumlar kümesi

∑ : Giriş alfabesi

Γ : Yığın alfabesi

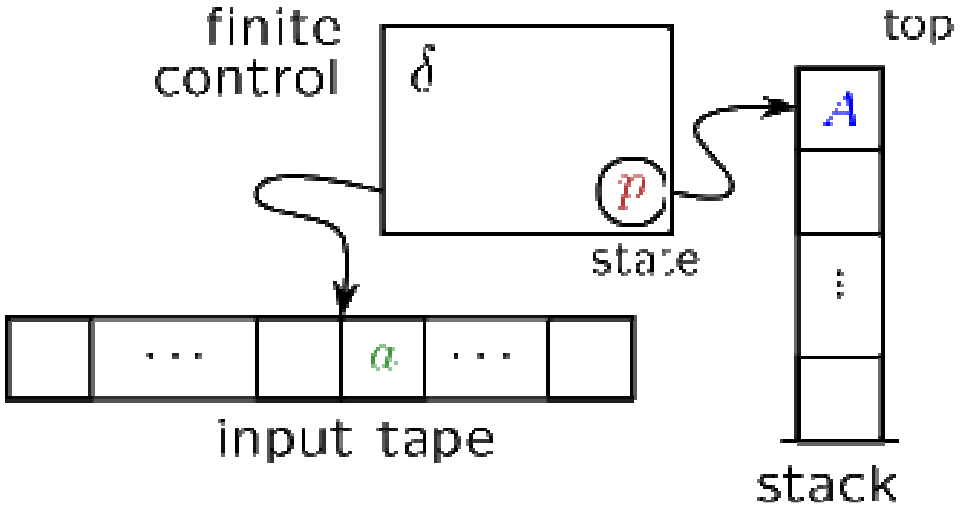
δ : Geçiş işlevi

q_0 : Başlangıç durumu ($q_0 \in Q$)

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

F : Uç durumlar kümesi

Bir PDA aşağıdaki gibi gösterilebilir.



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.

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

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

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

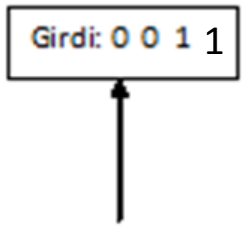
$F = \{q_1, q_4\}$, and

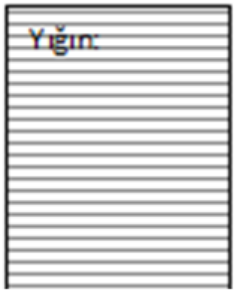
δ is given by the following table, wherein blank entries signify $\emptyset$.

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									

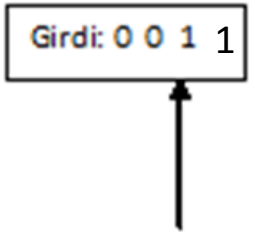
Aşağı-itme Otomatı

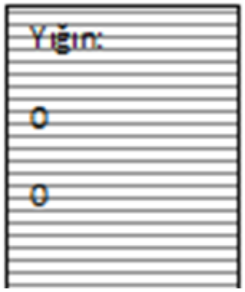
Bu tasarımı bir iki örnek ile anlamaya çalışalım. Örneğin girdimiz 0011 olsun.



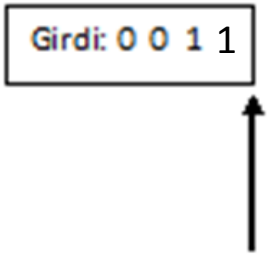


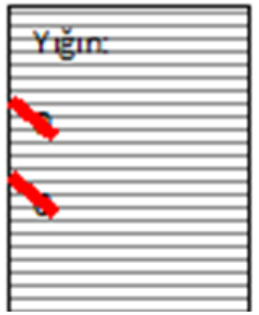
İlk durumda makinemiz girdinin ilk harfi olan 0′ı okuyacak ve 0 gördükçe yığına koyacak (push)





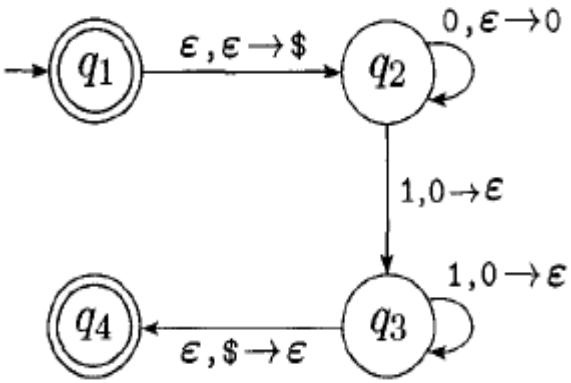
Yığına 0′ları koyduktan sonra her gördüğü 1 için yığından bir 0 çıkaracak (pop):





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.

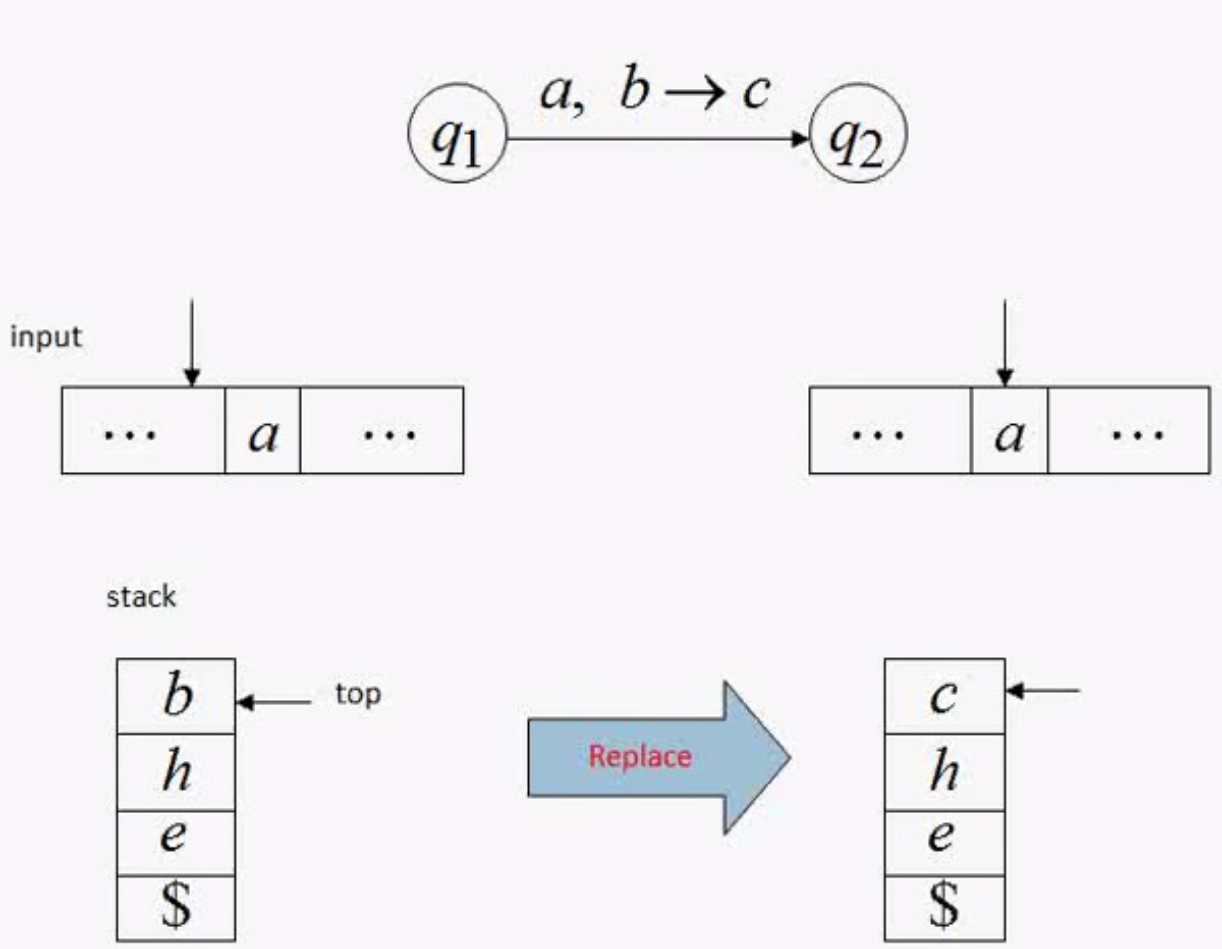


State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

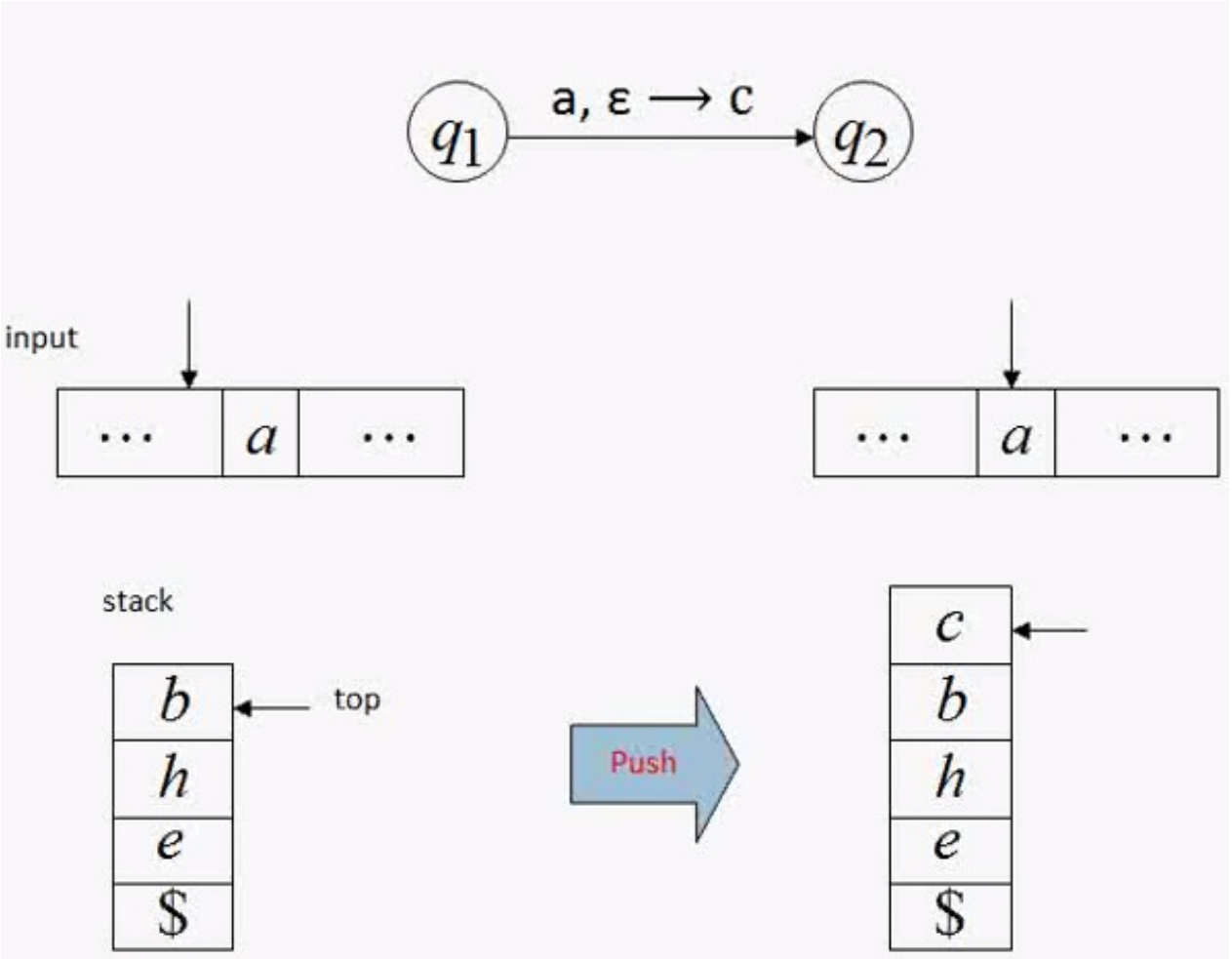
Yandaki makine tasarımımızı kısaca gözden geçirecek olursak. Kabul edilen durumlar q_1 ve q_4 durumlarıdır. Yani e (boş kelime) durumunu kabul için q_1, diğer kabul edilir durumlar için de q_4 durumu (state) tasarlanmıştır. Tasarımda bulunan q_2 durumu geçiş durumudur. Yani q_2 durumunda bulunduğu sürece 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.

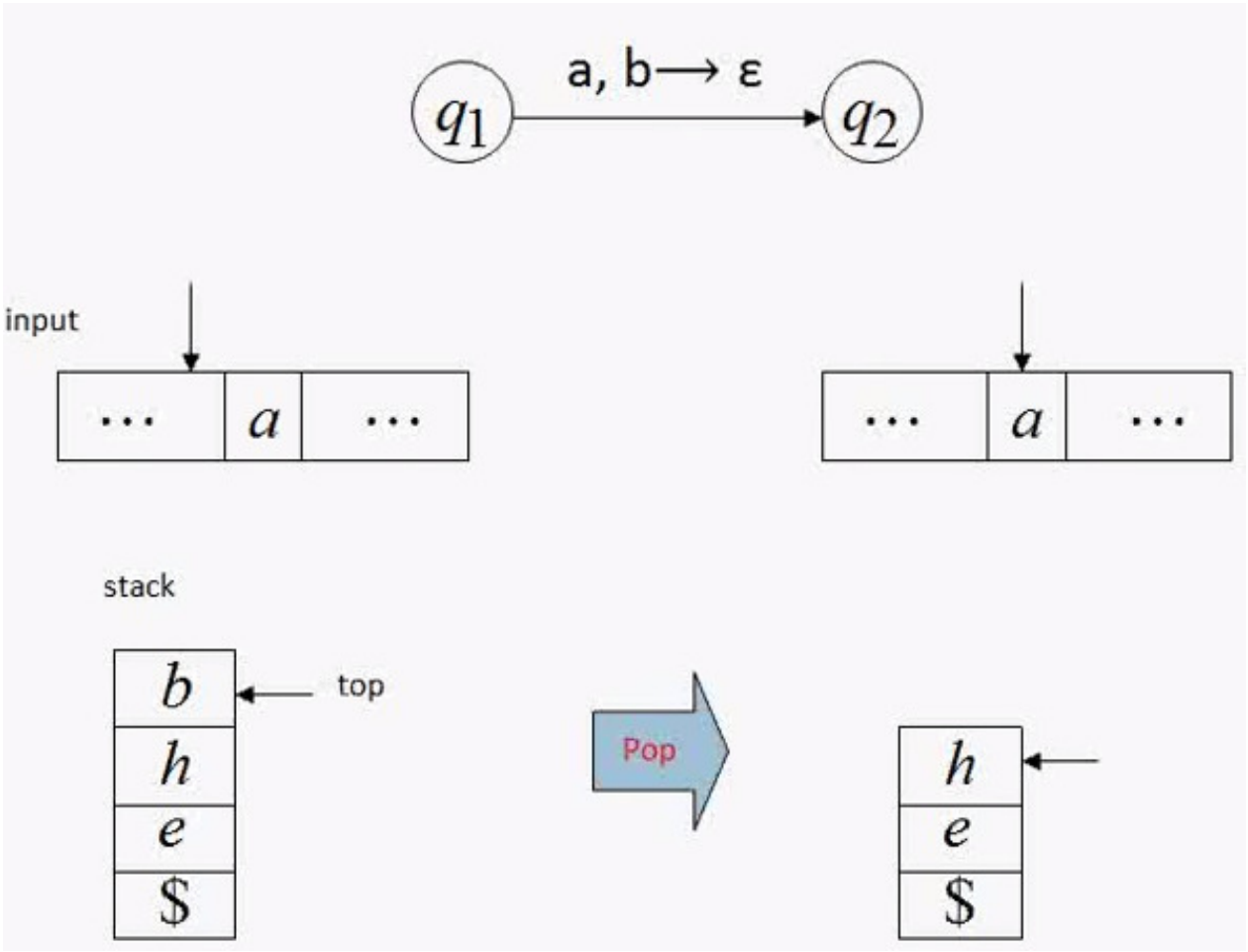
Aşağı-itme Otomatı



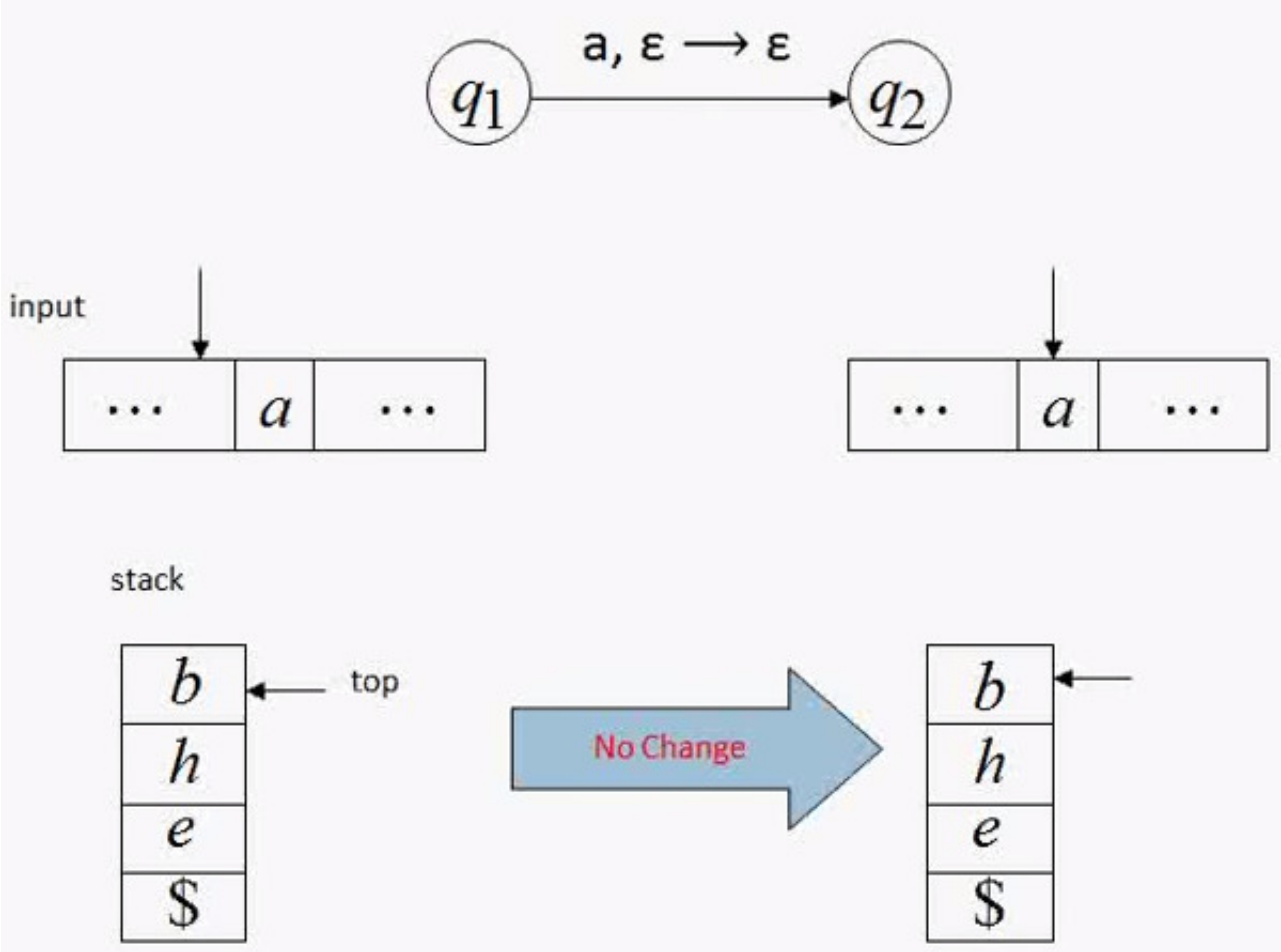
Aşağı-itme Otomatı



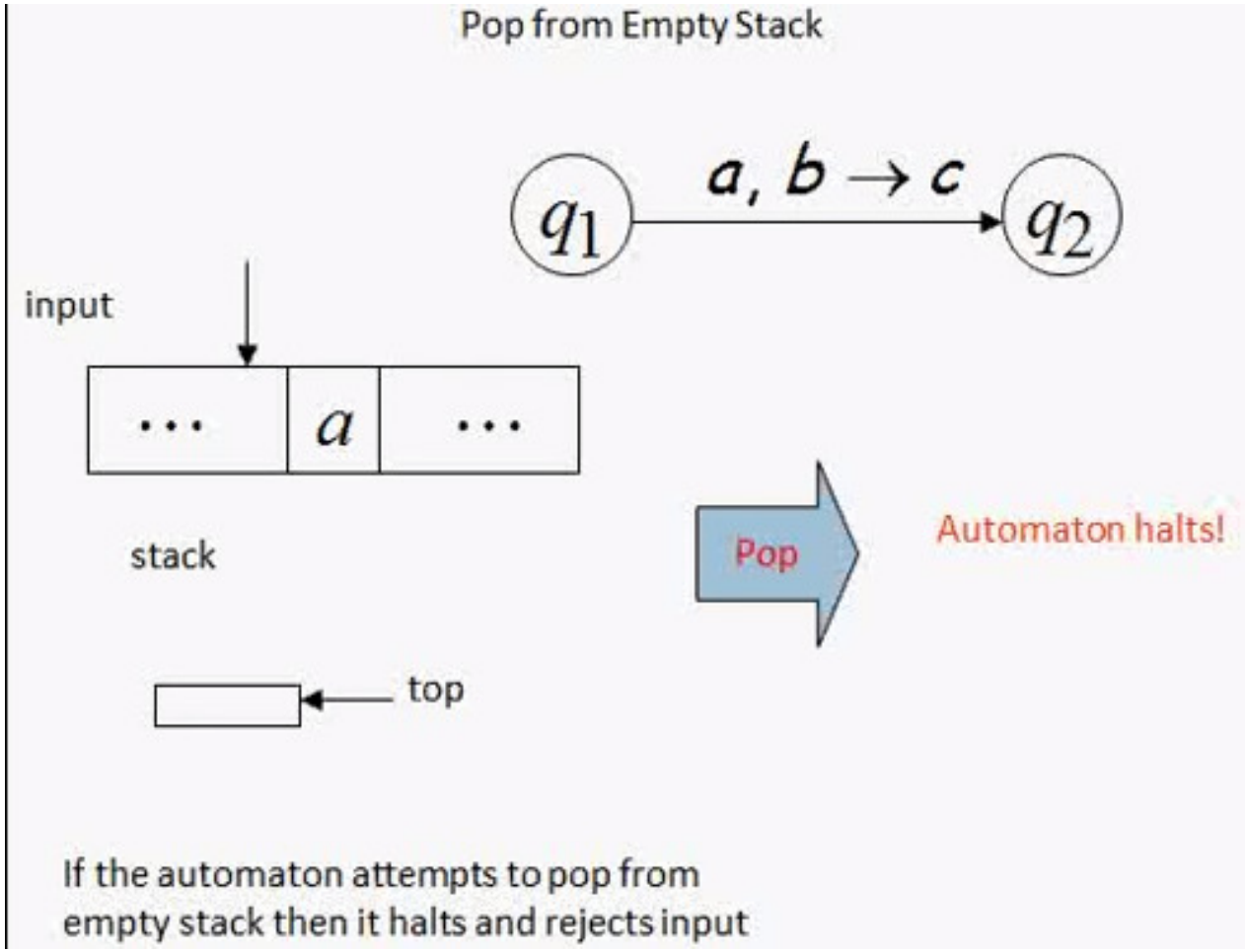
Aşağı-itme Otomatı



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)

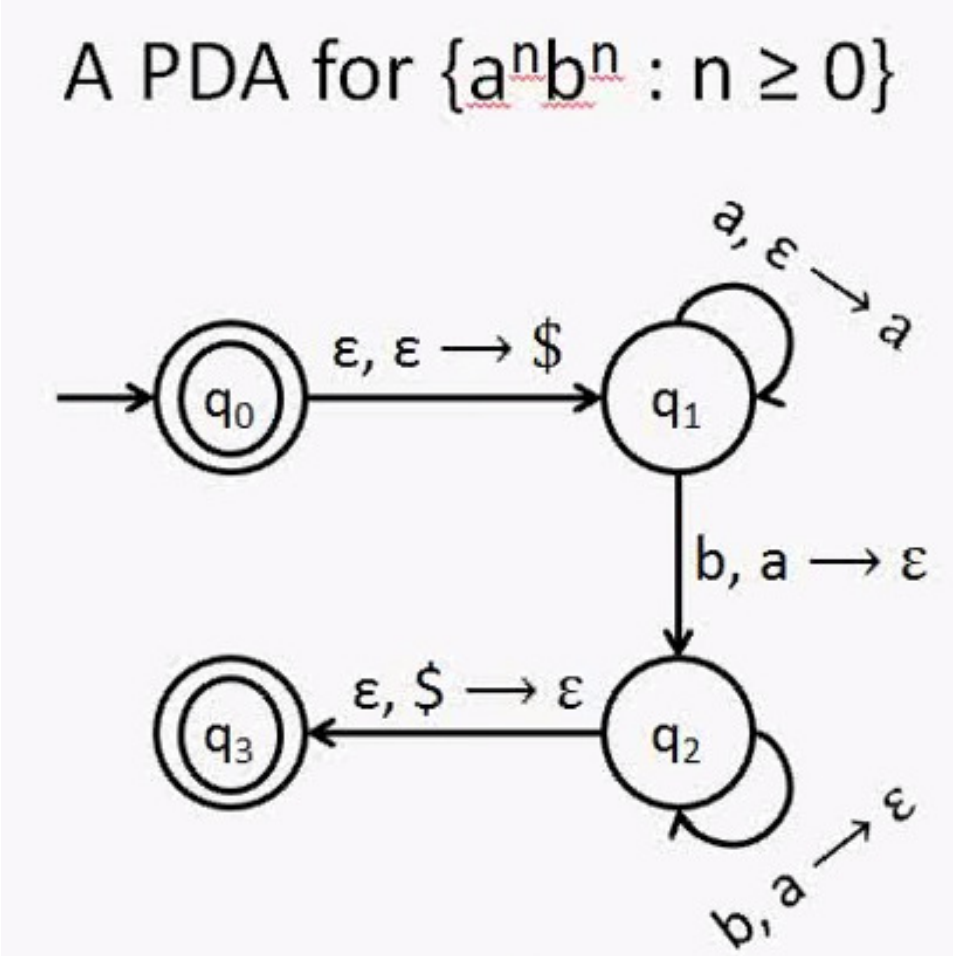
Aşağı-itme Otomatı

Is the stack empty?

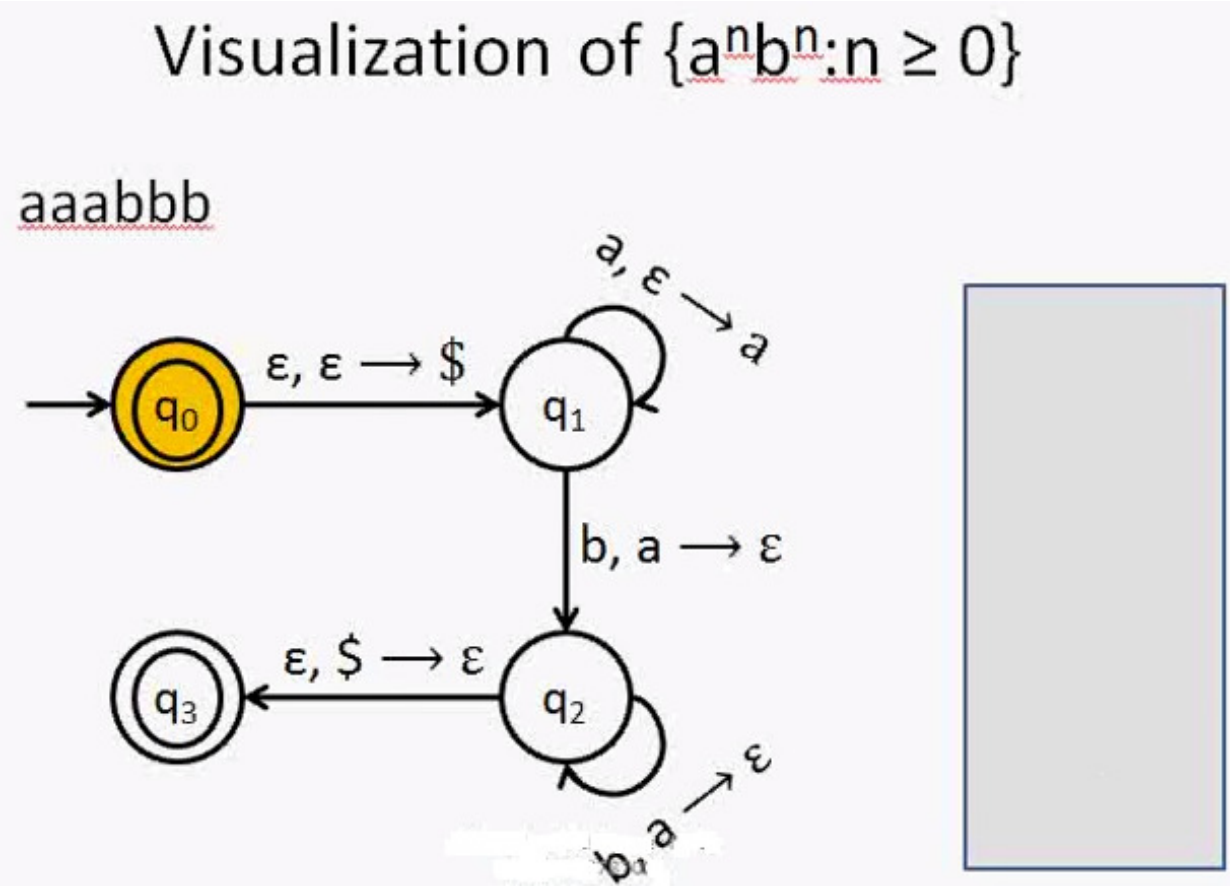
How can you check if the stack is empty?

- 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.**

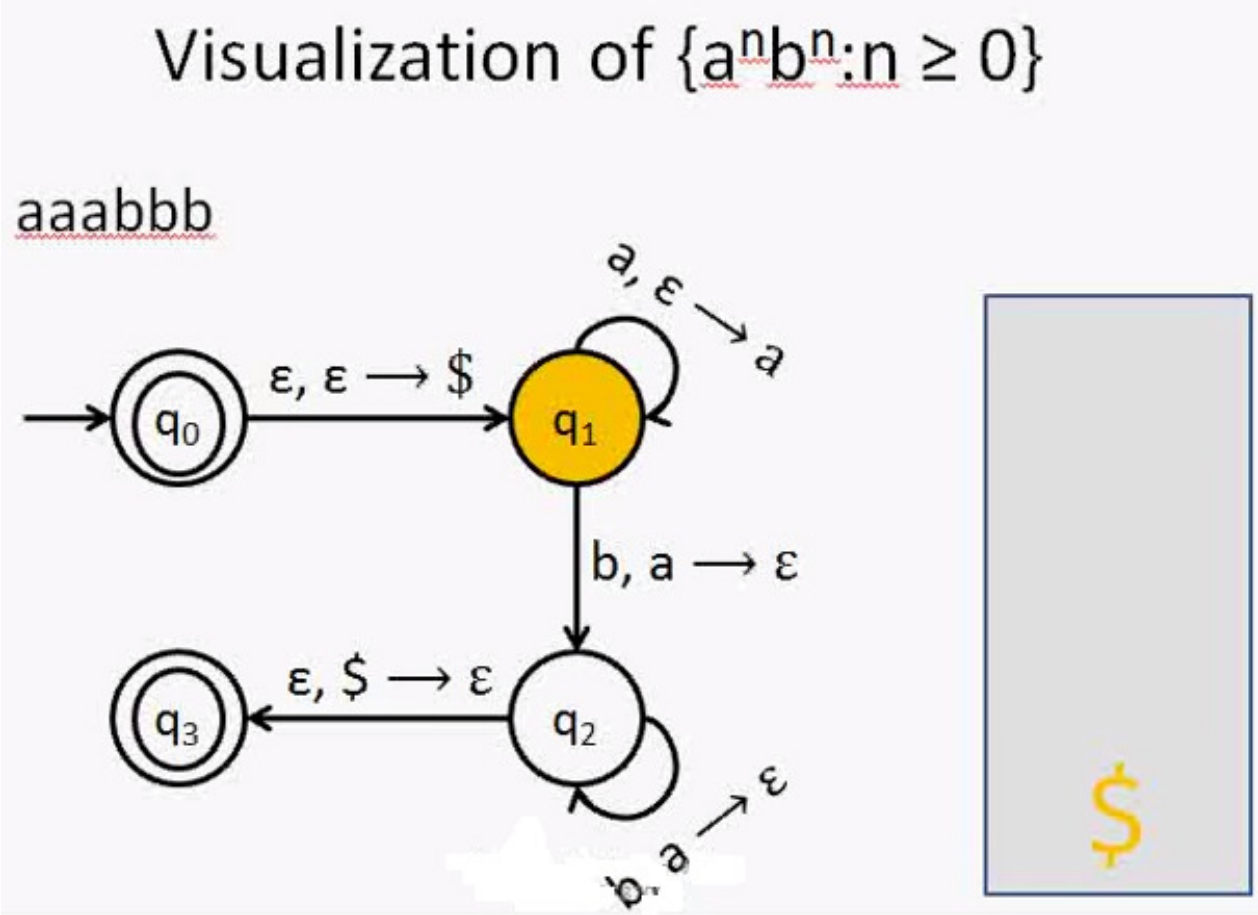
Aşağı-itme Otomatı



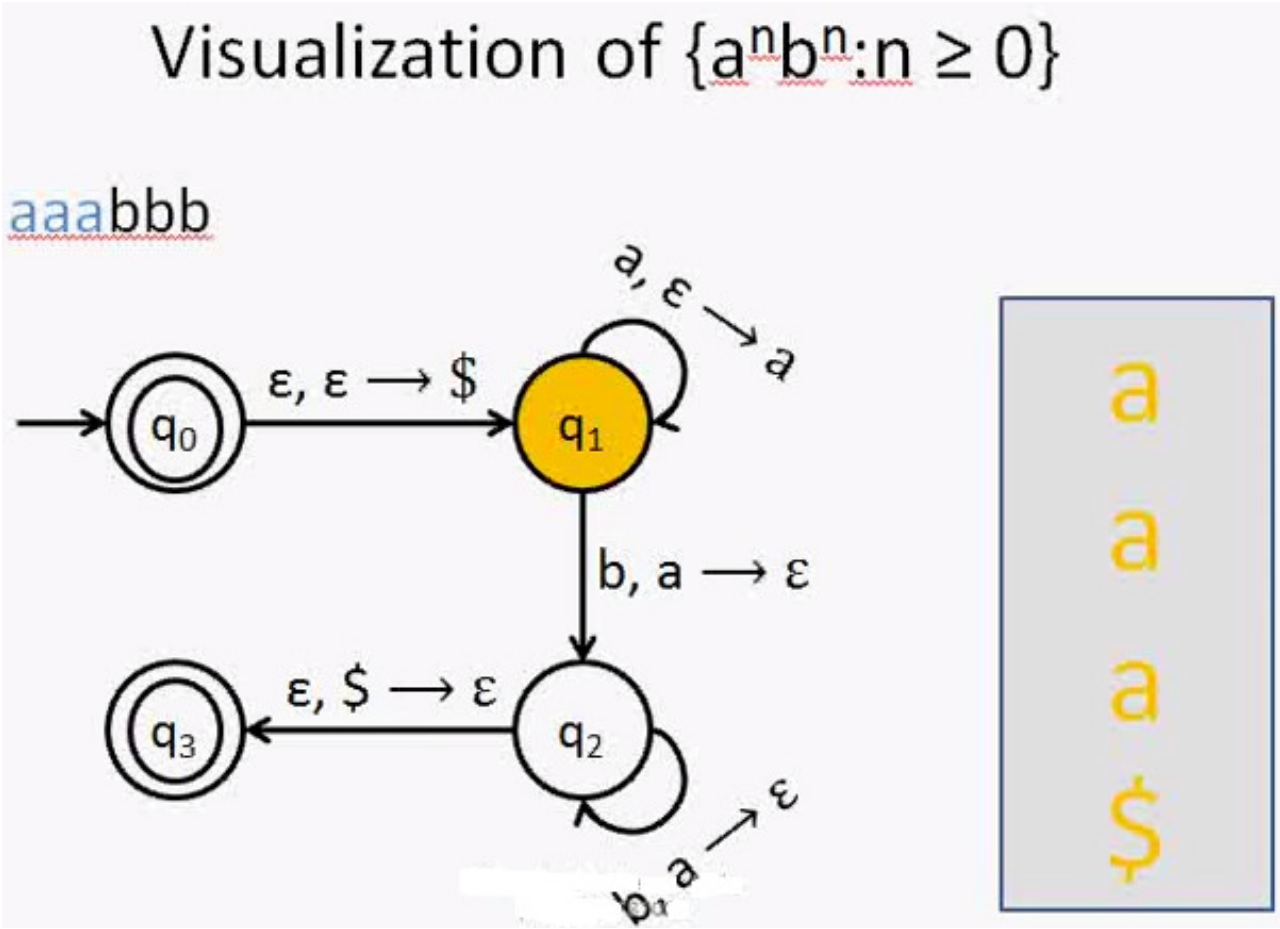
Aşağı-itme Otomatı



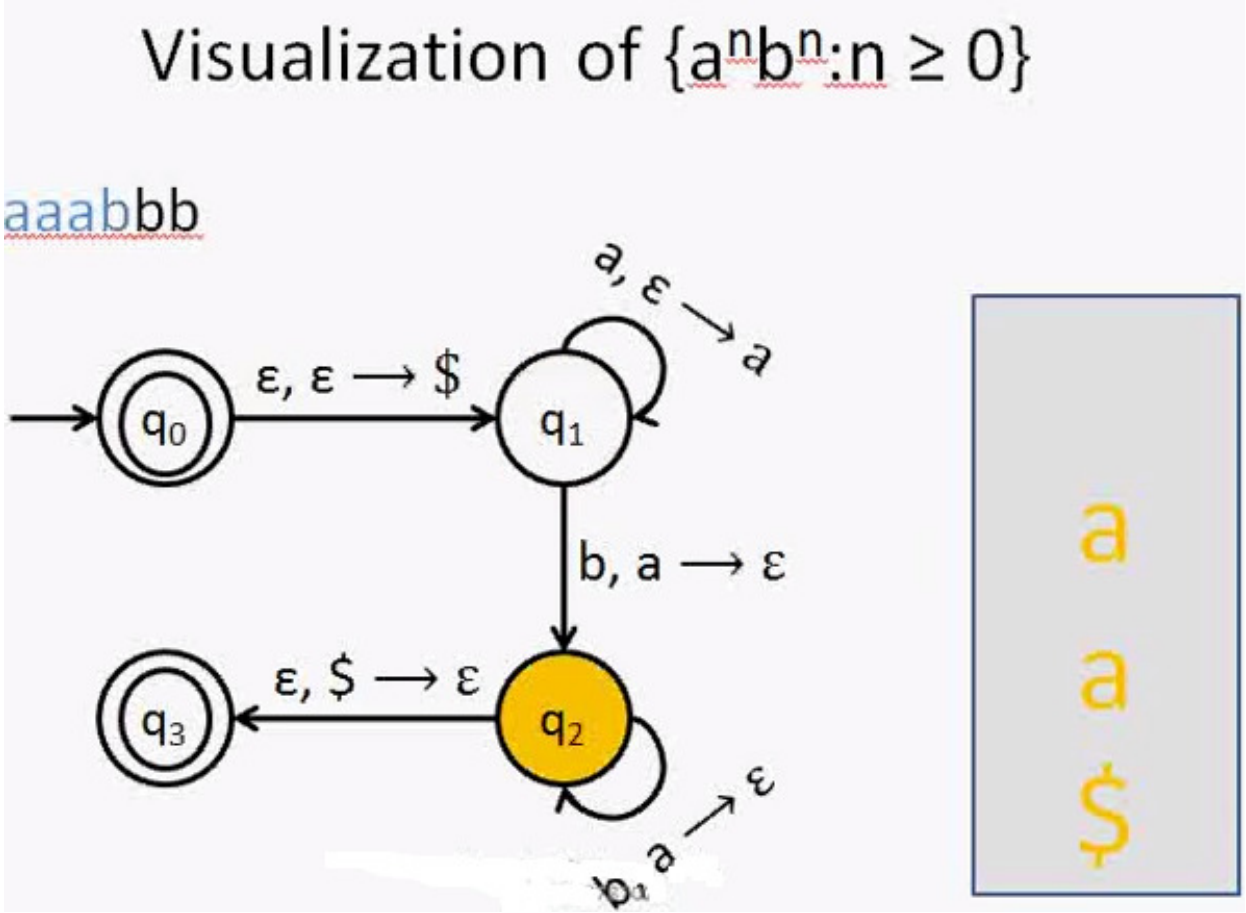
Aşağı-itme Otomatı



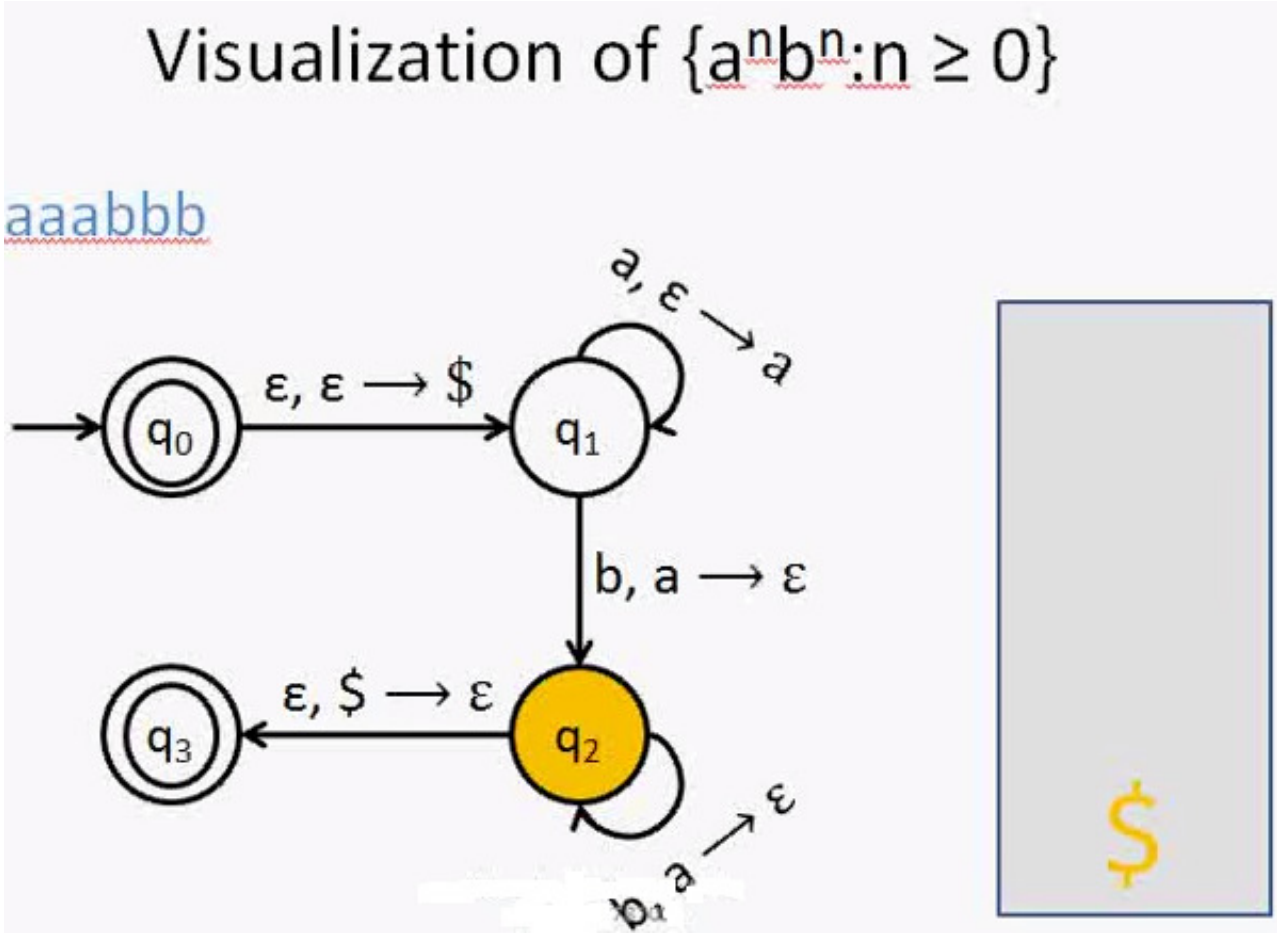
Aşağı-itme Otomatı



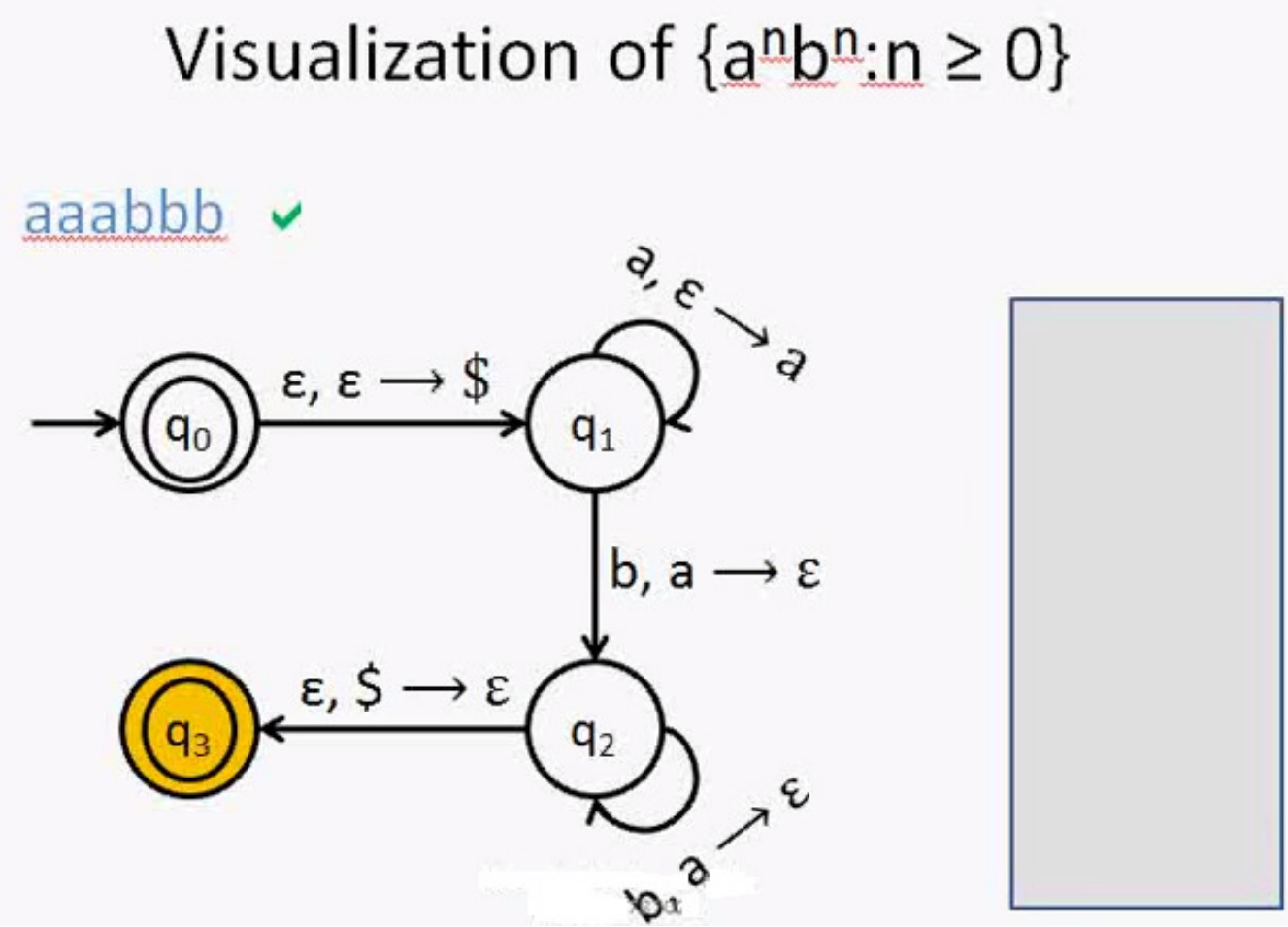
Aşağı-itme Otomatı



Aşağı-itme Otomatı



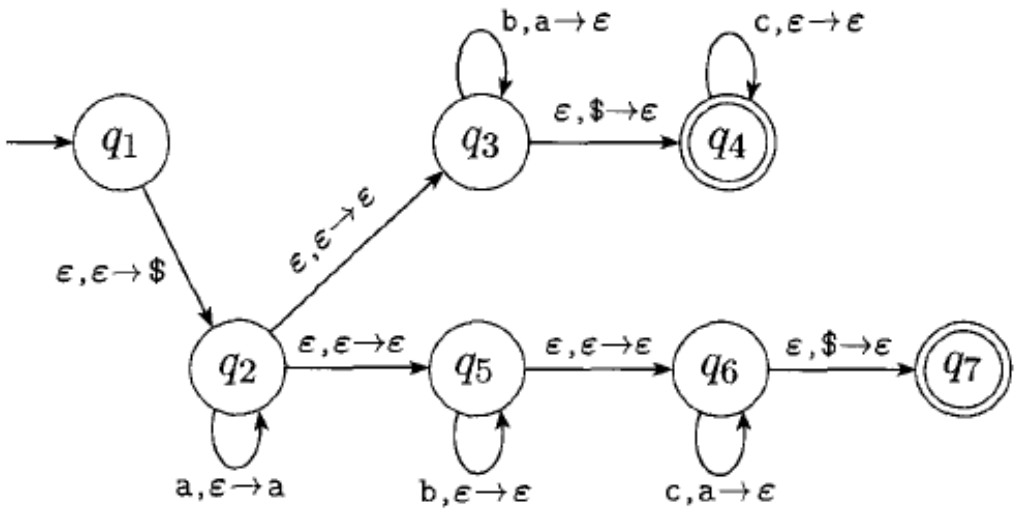
Aşağı-itme Otomatı



Aşağı-itme Otomatı

This example illustrates a pushdown automaton that recognizes the language

$$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$$



State diagram for PDA M_2 that recognizes
$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

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

$V_N = \{S\}$

$V_T = \{0, 1, c\}$

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

Bu dili tanıyan PDA:

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

Q : $\{q_0, q_1\}$

$\sum$: $\{0, 1, c\}$

Γ : $\{0, 1, Z_0\}$

PDA'nın Tanıdığı Dil

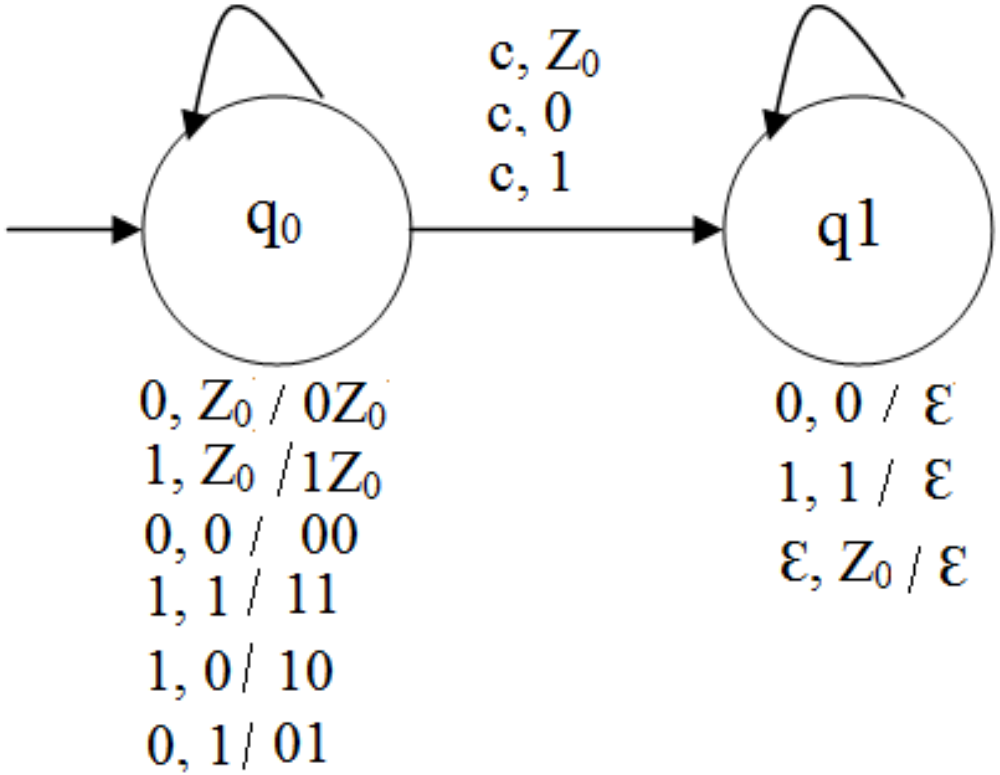
PDA'nın çalışma ilkesi:

- Önce w'nun simgelerini okur ve okuduğu her simgeyi yığına ekler. Ekleme sırasında PDA q_0 durumundadır.
- c'yi okuduğunda w'nun bittiğini anlar, yığında bir değişiklik yapmaz ve durum değiştirerek q_1 durumuna geçer.
- Sonra w^R nin simgelerini okur ve okuduğu her simge için eğer yığının tepesinde aynı simge varsa bunu yığından çıkarır.
- Giriş dizisinin tüm simgeleri okunduktan sonra eğer yığının tepesinde Z_0 varsa bunu da siler ve yığını boşaltır.

PDA'nın Tanıdığı Dil

δ :

$\delta(q_0, 0, Z_0) = (q_0, 0Z_0)$
$\delta(q_0, 1, Z_0) = (q0, 1Z_0)$
$\delta(q_0, c, Z_0) = (q_1, Z_0)$
$\delta(q_0, 0, 0) = (q_0, 00)$
$\delta(q_0, 1, 0) = (q_0, 10)$
$\delta(q_0, 0, 1) = (q_0, 01)$
$\delta(q_0, 1, 1) = (q_0, 11)$
$\delta(q_0, c, 0) = (q_1, 0)$
$\delta(q_0, c, 1) = (q_1, 1)$
$\delta(q_1, 0, 0) = (q_1, \varepsilon)$
$\delta(q_1, 1, 1) = (q_1, \varepsilon)$
$\delta(q_1, \varepsilon, Z_0) = (q_1, \varepsilon)$



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:

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

$Q = \{q_0, q_1\}$

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

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

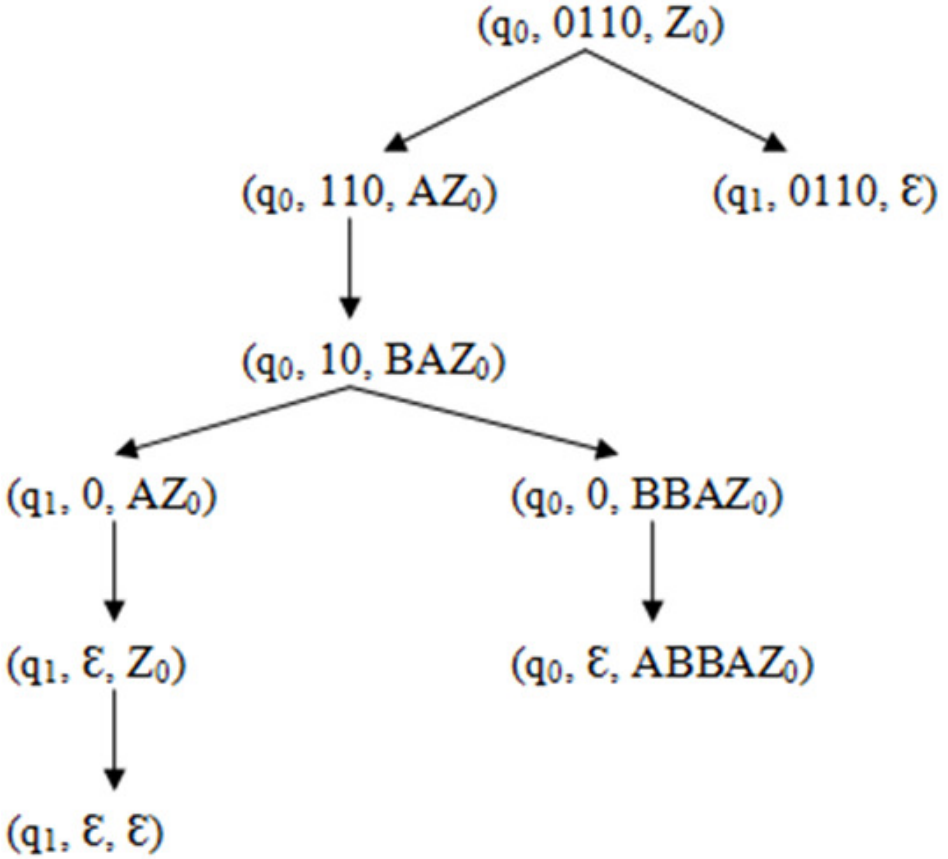
PDA'nın çalışma ilkesi:

- 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 içermeyebilirde.
- 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.

Top-down parsers *(LL(1), recursive descent)*

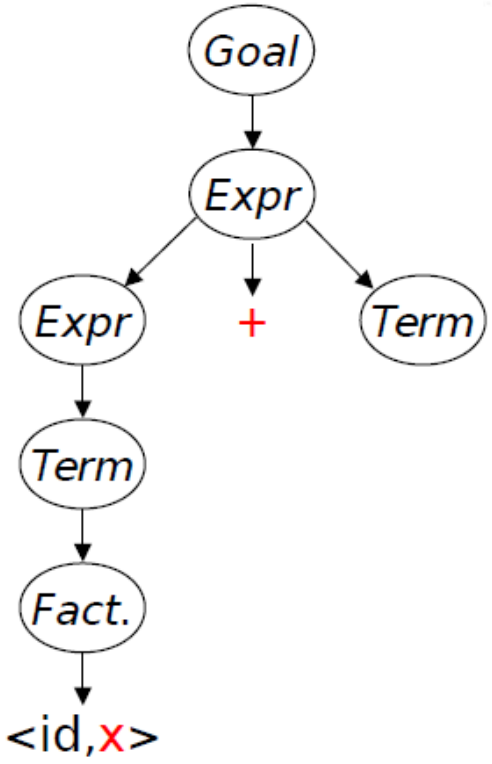
- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x - 2 * y

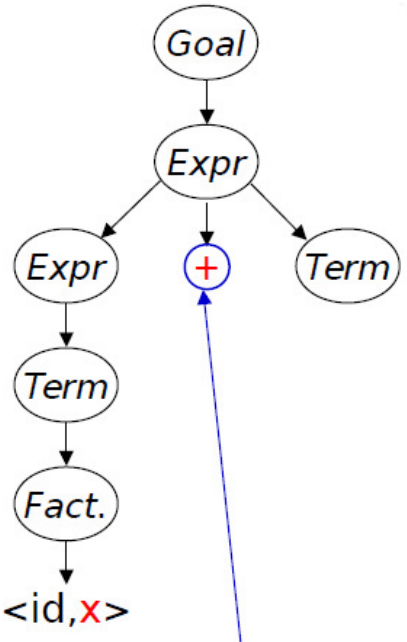
Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y



Let's try x - 2 * y :

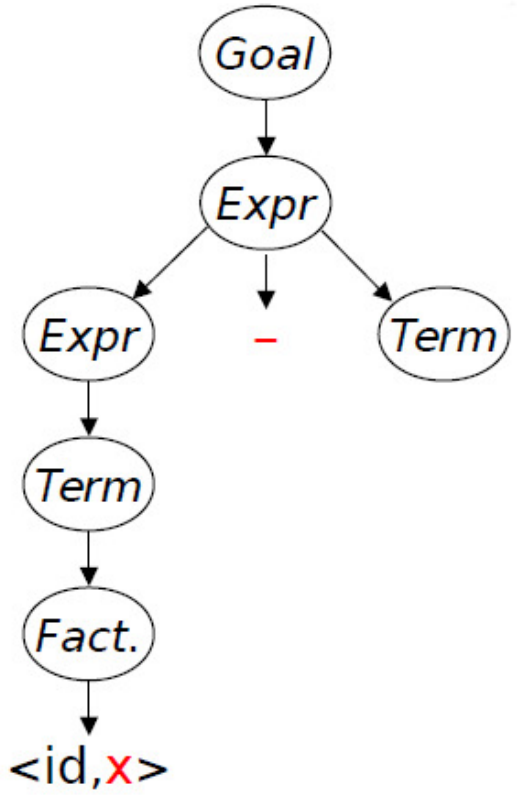
Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y



This worked well, except that "-" doesn't match "+"
The parser must backtrack to here

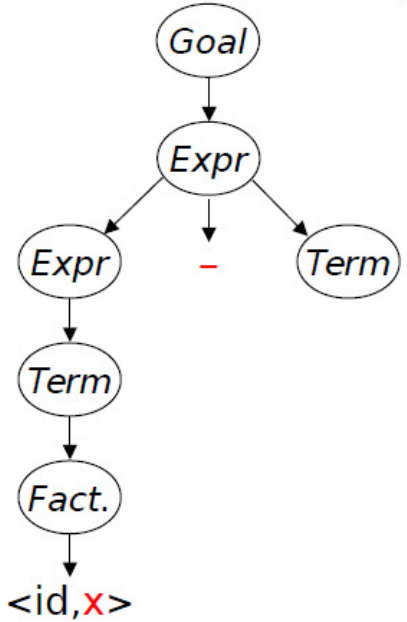
Continuing with x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
3	*Expr* - *Term*	↑x - 2 * y
4	*Term* - *Term*	↑x - 2 * y
7	*Factor* - *Term*	↑x - 2 * y
9	<id,x> - *Term*	↑x - 2 * y
9	<id,x> - *Term*	x ↑- 2 * y
—	<id,x> - *Term*	x - ↑2 * y



Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



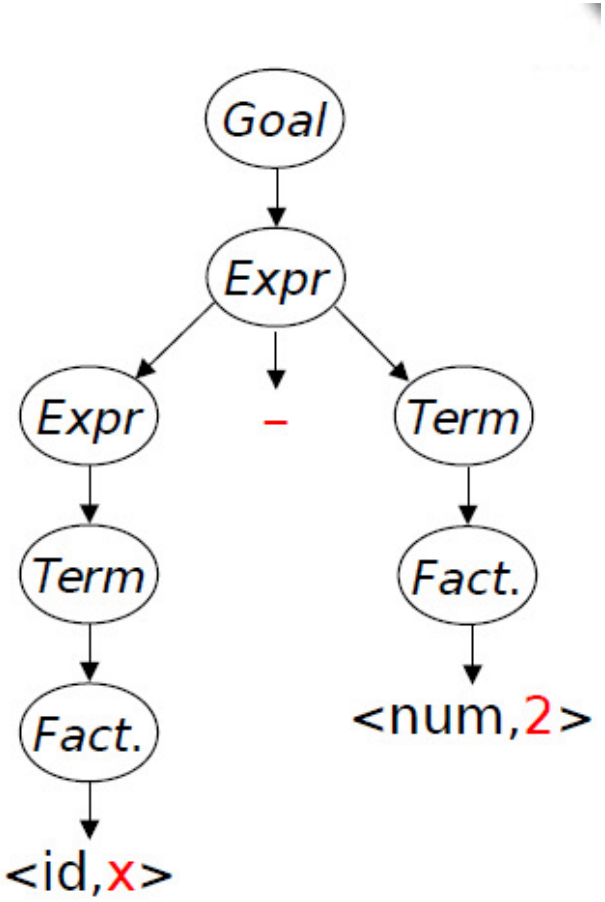
This time, "–" and "–" matched

We can advance past "–" to look at "2"

⇒ Now, we need to expand *Term* - the last *NT* on the fringe

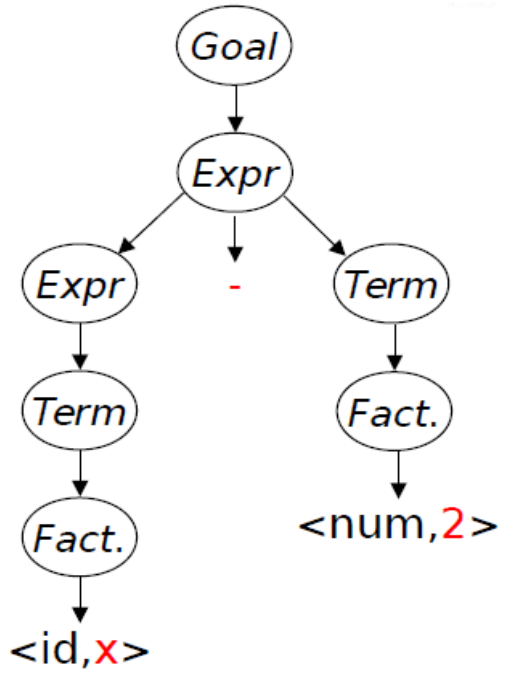
Trying to match the "2" in x - 2 * y :

Rule	*Sentential Form*	*Input*
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



Trying to match the "2" in x - 2 * y :

Rule	*Sentential Form*	*Input*
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



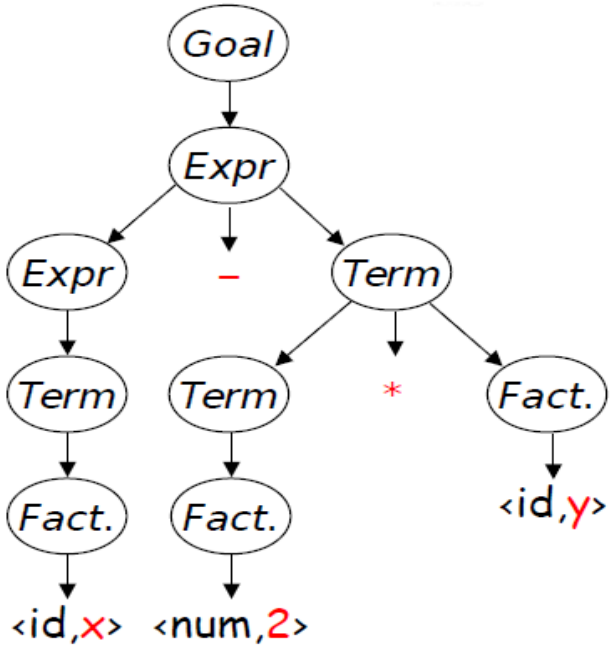
Where are we?

- "2" matches "2"
- We have more input, but no *NT*s left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Trying again with "2" in x - 2 * y :

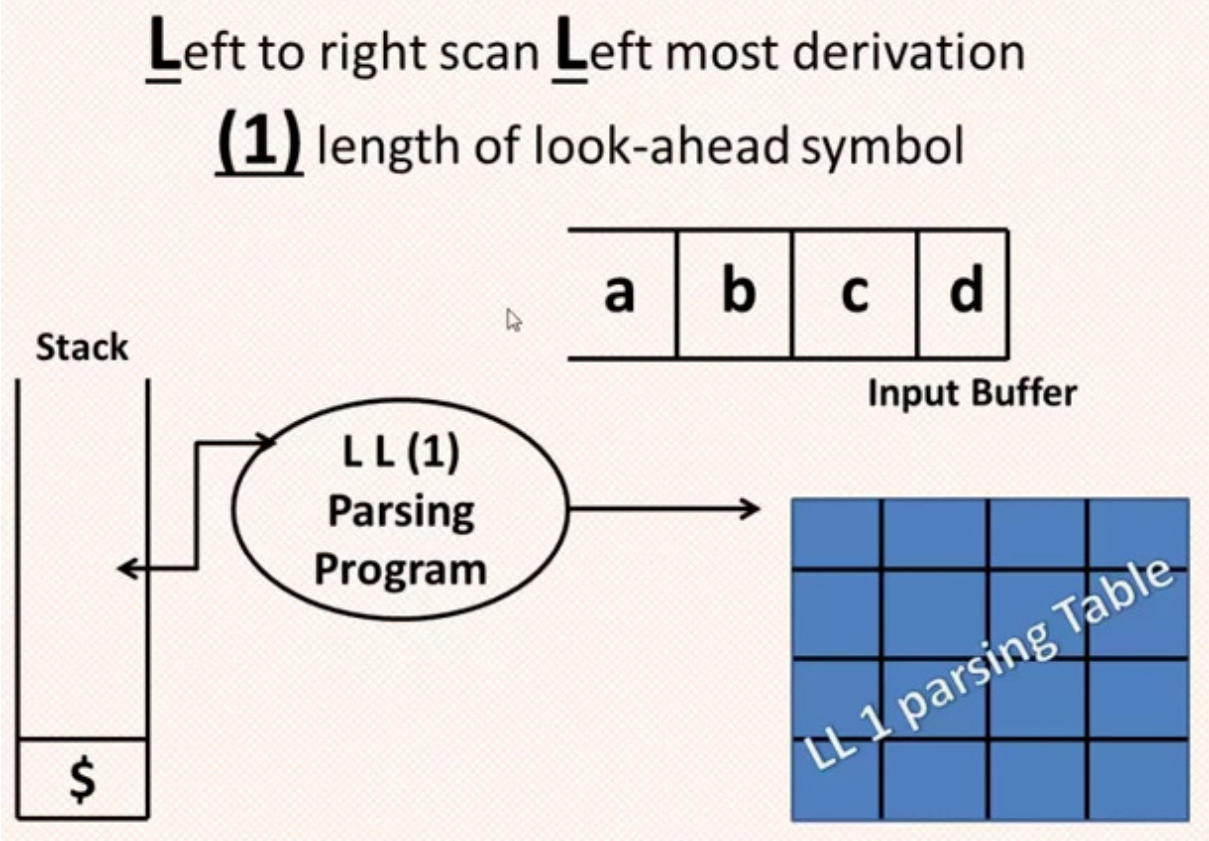
Rule	*Sentential Form*	*Input*
—	<id,x> - *Term*	x - ↑2 * y
5	<id,x> - *Term* * *Factor*	x - ↑2 * y
7	<id,x> - *Factor* * *Factor*	x - ↑2 * y
8	<id,x> - <num,2> * *Factor*	x - ↑2 * y
—	<id,x> - <num,2> * *Factor*	x - 2 ↑* y
—	<id,x> - <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> - <num,2> * <id,y>	x - 2 * y↑



This time, we matched & consumed all the input

⇒ Success!

LL(k) Ayrıştırmada



LL(k) Ayrıştırmada Kullanılacak PDA Modeli

LL(k) grameri G=<V_N, V_T, P, S> olsun

Bu gramer tarafından türetilen dilin ayrıştırılmasında 3 durumlu bir PDA kullanılır. Yukarıdan-aşağıya ayrıştırma algoritmasına uygun biçimde oluşturulan bu PDA kısaca **PDA(↓)** olarak gösterilir.

$M = <Q, \Sigma, \Gamma, \delta, q_0, Z_0, \Phi>$

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

$\Sigma = V_T$

$\Gamma = V_N \cup V_T \cup \{Z_0\}$

$F = \{q_2\}$

δ : 1) $\delta(q_0, \varepsilon, \varepsilon) = (q_1, S)$

2) $\surd a \in V_T$: $\delta(q_1, a, a) = (q_1, \varepsilon)$

$\surd (A \Rightarrow \beta) \in P$: $\delta(q_1, \varepsilon, A) = (q_1, \beta)$

3) $\delta(q_1, Z_0, \lambda) = (q_2, \varepsilon)$

LL(k) Ayrıştırmada Kullanılacak PDA Modeli

- Başlangıçta PDA q_0 durumundadır ve yığında Z_0 vardır.
- İlk olarak bir ε hareketi ile yığına S eklenir ve PDA q_1 durumuna geçer.
- q_1 durumundayken yığının üstünde bir terminal simge varsa ve giriş dizisinden okunan simge ile aynı ise yığından bu simge silinir ve giriş dizisinden bir sonraki simge okunur.
- q_1 durumundayken yığının üstünde bir nonterminal (A) varsa, ayrıştırma çizelgesi yardımıyla bu değişkene ilişkin kurallardan birisi seçilir ve bu kuralın sağ tarafı (β) ε hareketi ile yığına eklenir.
- q_1 durumundayken yığının üstünde Z_0 varsa ε hareketi ile Z_0 yığından silinir
- Sonuçta giriş dizisinin sonuna ulaşıldığında yığın boşalmışsa ve sonlu denetim birimi q_2 durumundaysa sözcük tanınmıştır.

LL(k) Ayrıştırmada Kullanılacak PDA Modeli

Tanımlanan PDA hem boş yığınla hem de uç durumla tanıyan bir PDA'dır. Doğru bir sözcük için hem yığın boşalmakta, hem de sonlu denetim birimi bir uç duruma **(q_2)** ulaşmaktadır.

S -> aS | cA

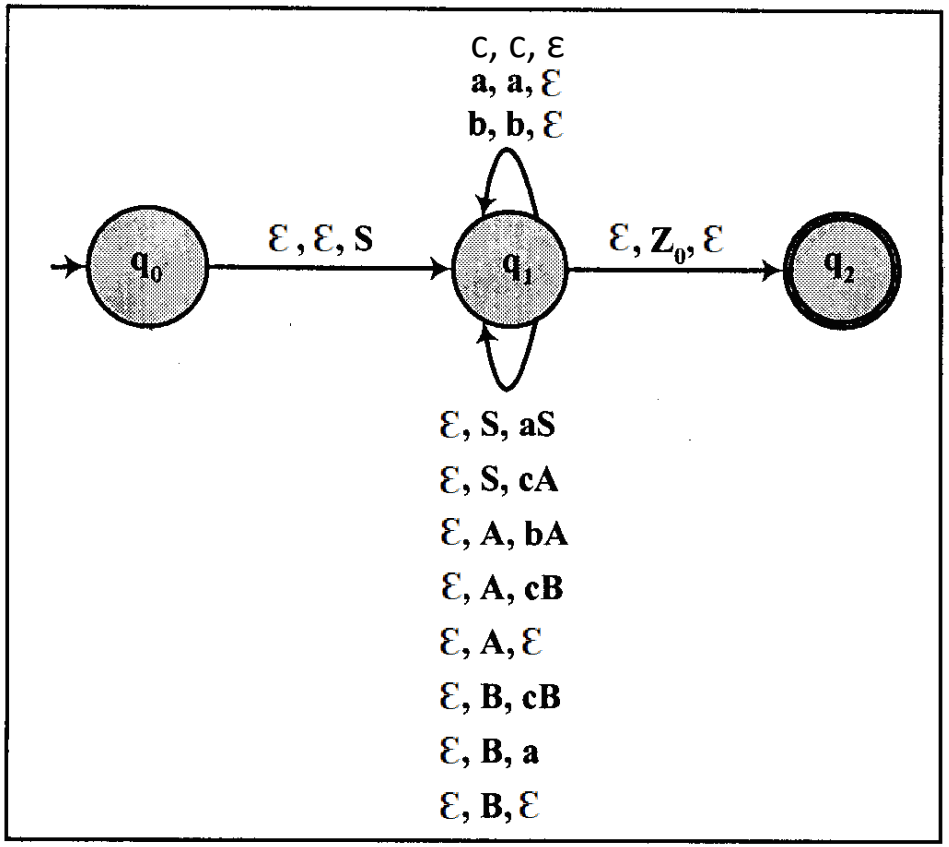
A -> bA | cB |ε

B -> cB | a | ε

Gramer1

LL(k) Ayrıştırmada Kullanılacak PDA Modeli

Bu örnek grameri için LL(1) ayrıştırmada kullanılacak PDA(↓) modeli şekilde görülmektedir.



Örnek:

$G = (V, \Sigma, R, S)$ şeklinde tanımlı bir CFG olsun ve $L = \{wcw^R : w \in \{a, b\}^*\}$ dilini oluşturursun.

$V = \{S, a, b, c\}$

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

$R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c)$

Bu dili tanıyan bir PDA olan $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$ olarak tanımlanabilir.

$\Delta =$ { ((p, e, e), (q, S)), (T1)

((q, e, S), (q, aSa)), (T2)

((q, e, S), (q, bSb)), (T3)

((q, e, S), (q, c)), (T4)

((q, a, a), (q, e)), (T5)

((q, b, b), (q, e)), (T6)

((q, c, c), (q, e))} (T7)

abbcbba için geçişler aşağıdaki tabloda verilmiştir.

State	Unread Input	Stack	Transition Used
p	$abbcbba$	e	-
q	$abbcbba$	S	T1
q	$abbcbba$	aSa	T2
q	$bbcbba$	Sa	T5
q	$bbcbba$	$bSba$	T3
q	$bcbba$	Sba	T6
q	$bcbba$	$bSbba$	T3
q	$cbba$	$Sbba$	T6
q	$cbba$	$cbba$	T4
q	bba	bba	T7
q	ba	ba	T6
q	a	a	T6
q	e	e	T5

Deterministik Ayrıştırma

- Ayrıştırma işlemi, ancak belirli özellikleri taşıyan gramerler için deterministik bir sürece dönüştürülebilir. Ayrıştırmanın deterministik bir sürece dönüştürülmesinde kullanılan ortak teknik ise **ileri-bakış (lookahead)** tekniğidir.
- Deterministik yukarıdan aşağı ayrıştırmayı ileri bakış tekniği kullanarak gerçekleştirmeyi sağlayan içerik bağımsız gramerler LL(k) gramerleri olarak adlandırılır.
- Buna karşılık deterministik aşağıdan-yukarı ayrıştırmayı ileri bakış tekniği kullanarak gerçekleştirmeyi olanak veren gramerler ise LR(k) gramerler olarak adlandırılır.

LL(k) Grameri

LL(k) adlandırmasında:

- İlk L ayrıştırılacak sözcüğün soldan sağa doğru tarandığını,
- İkinci L, ayrıştırıcının soldan (sol öncelikli) türetme gerçekleştirdiğini,
- k ise ileri-bakış simgesi sayısını gösterir.

Buna göre bir CFG'nin **LL(k)** grameri olması, bu gramer tarafından türetilen stringlerin soldan-sağa doğru taranarak ve k ileri-bakış simgesi kullanılarak, yukarıdan-aşağıya türetme ile deterministik olarak ayrıştırılabileceğini gösterir.

S ⟶ aS | AB | B

A ⟶ abA | ab

B ⟶ BB | ba

Gramer2

LL(k) Grameri

Yukarıdan-aşağıya ayrıştırma işlemlerinde ileri-bakış tekniğinin kullanılması şu şekilde gösterilebilir:

Örnek olarak *Gramer–2* ile türetilen **w = aababbbaba** sözcüğünün ayrıştırılmasına bakarsak; bu gramerin üç **S** kuralı vardır. Ayrıştırma işleminin deterministik olmasını sağlamak için, ayrıştırmanın her adımında tek bir yeniden yazma kuralı kullanılabilmelidir. **w**'nin ilk simgesi **a**'dır. **S** kurallarından ilk ikisi kullanılarak **a** ile başlayan ifadeler türetilebilir. Ancak (**S** ⟹ **B**) kuralı kullanılarak **a** ile başlayan ifadeler türetilemez. Böylece **w**'nin ilk simgesine baktığımızda kullanılabilecek kural sayısı üçten ikiye indirilebilir. Ancak bu deterministik ayrıştırma için yeterli değildir. Oysa **w**'nin ilk iki simgesine bakılırsa, **aa** ile başlayan ifadenin türetilebilmesi için ilk olarak (**S** ⟹ **aS**) kuralının kullanılması gerektiği görülür. **S** ⟹ **aS** kuralı ile **w**'nin ilk simgesi türetilmiş olur.

LL(k) Grameri

Bu aşamada sonraki iki simge **ab**'dir. **ab** simgeleri, **S** kuralarından (**S ⟹ aS**) ya da (**S⟹AB**)'nin kullanılabileceğini gösterir; ancak bu iki kuraldan hangisinin kullanılacağına karar vermek için yetersiz kalır. Bunun anlamı, Gramer–2 tarafından türetilen ifadeler yukarıdan-aşağıya ayrıştırılırken, ayrıştırmanın deterministik olması için iki ileri-bakış simgesi de yeterli değildir. Başka bir deyişle Gramer–2 bir **LL(2)** grameri değildir. Aslında bu gramer bir **LL(k)** grameri değildir. **w=abababababab**…. Örneğine bakarsak, bu sözcüğün türetilmesinde kullanılan ilk kural (**S ⟹ aS**) ya da (**S ⟹ AB**) olabilir. Bu iki kuraldan hangisinin kullanıldığını belirleyebilmek için ya sözcük sonunu ya da **bb** örüntüsünü görmek gerekir.

LL(k) Ayrıştırma

LL(k) gramerleri tarafından türetilen diller için kullanılabilen deterministik ayrıştırma LL(k) ayrıştırma olarak adlandırılır. LL(k) ayrıştırma işleminde PDA modeli ile birlikte ayrıştırma çizelgesi olarak adlandırılan bir çizelge kullanılır.

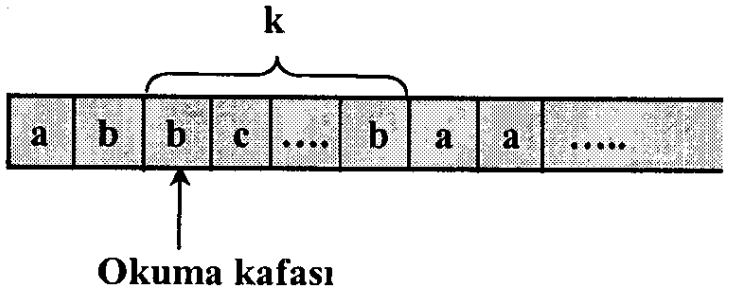
LL(k) Ayrıştırma Çizelgesinin Oluşturulması

Ayrıştırma çizelgesi, ayrıştırmadaki belirsizliği ortadan kaldırmak ve ayrıştırmanın deterministik bir biçimde sürdürülmesini sağlamak için kullanılan bir çizelgedir. PDA modeli incelendiğinde, yığının üstünde bir değişken bulunduğunda belirsizliğin ortaya çıktığı görülür. Ayrıştırma çizelgesi, ileri-bakış simgelerine bakılarak, ilgili değişkenle ilgili yeniden yazma kurallarından hangisinin kullanılacağını belirler ve böylece ayrıştırmadaki belirsizliği ortadan kaldırır. Ayrıştırma çizelgesinde:

- Gramerdeki her nonterminal simge için bir satır,
- İleri-bakış simgelerinin her birleşimi için de bir sütun (Okuma kafasının üzerinde bulunduğu simge, gramerin bir terminal olabileceği gibi, **dizi sonu (ds)** da olabilir) bulunur.

Çizelgenin her elemanında ise hangi yeniden yazma kuralının kullanılacağı, ya da sözcüğün yanlış olduğunun anlaşılacağı gösterilir.

LL(k) Ayrıştırma Çizelgesinin Oluşturulması



ileri-bakış simgeleri, okuma kafasının üzerinde bulunduğu simgeden başlayan k simgedir. O an okunan simge ileri-bakış simgelerine dâhildir ve k'nın değeri en az 1'dir.

- Terminal simgeleri a, b ve c olan bir LL(1) gramerinin ayrıştırma çizelgesinde başlıkları a, b, c ve ds olan 4 sütun bulunacaktır.
- Terminal simgeleri a, b ve c olan bir LL(2) gramerinin ayrıştırma çizelgesinde başlıkları aa, ab, ac, ba, bb, bc, ca, cb, cc, ads, bds, cds ve ds olan 13 sütun bulunacaktır.
- Terminal simgeleri a, b ve c olan bir LL(3) gramerinin ayrıştırma çizelgesinde 40 sütun bulunacaktır.

LL(k) Ayrıştırma Çizelgesinin Oluşturulması

Aşağıda *gramer-1* için oluşturulan ayrıştırma çizelgesi görülmektedir.

S -> aS | cA

A -> bA | cB |ε

B -> cB | a | ε

	a	b	c	ds
S	S ⇒ aS	yan	S ⇒ cA	yan
A	yan	A ⇒ ba	A ⇒ cB	A ⇒ λ
B	B ⇒ a	yan	B ⇒ cB	B ⇒ λ

“yan” ifadesi, bu koşullarda sözcüğün yanlış olduğunun anlaşılacağını göstermektedir

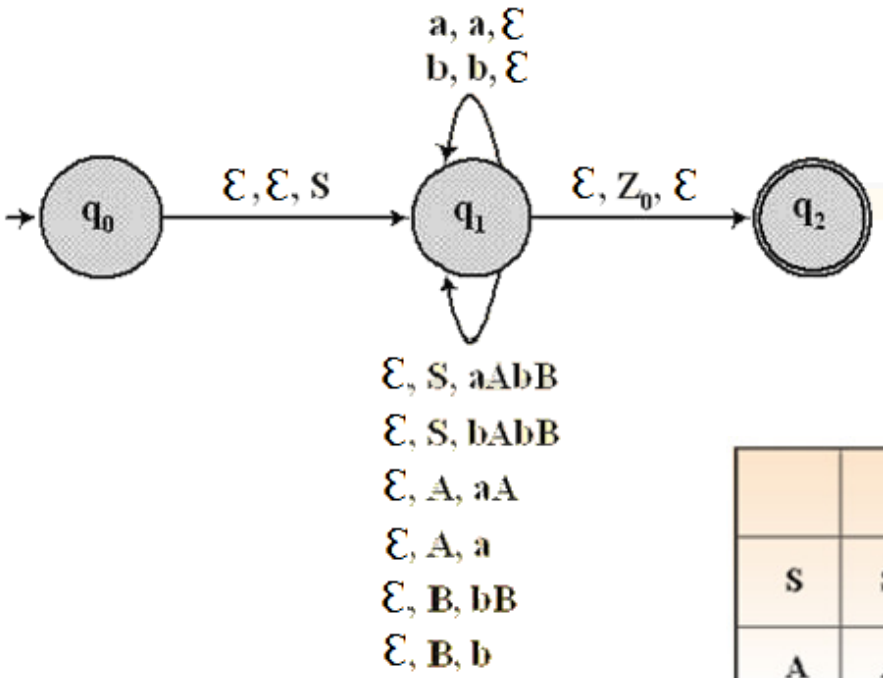
LL(k) Ayrıştırma Çizelgesinin Oluşturulması

Örnek

S -> aAbB | bAbB
A -> aA | a
B -> bB | b

Bu gramerin türettiği dilin ifadelerinin yukarıdan-aşağıya deterministik biçimde ayrıştırılabilmesi için bir ileri-bakış simgesi yeterli değildir. Ancak iki ileri-bakış simgesi kullanıldığında yukarıdan-aşağıya deterministik ayrıştırma yapılması mümkündür. Başka bir deyişle bu gramer bir LL(2) grameridir

LL(k) Ayrıştırma Çizelgesinin Oluşturulması



	aa	ab	ba	bb	ads	bds	ds
S	S⇒aAbB	yan	S⇒bAbB	yan	yan	yan	yan
A	A⇒aA	yan	yan	yan	A⇒a	yan	yan
B	yan	yan	yan	B⇒bB	yan	B⇒b	yan