

DERLEYİCİLER

İçindekiler

[SORU 1: Digraph ve Trigraphs](#)

[SORU 2: LR\(1\) Parçalama Algoritması](#)

[SORU 3: LR\(0\) parçalama algoritması](#)

[SORU 4: Earley Parçalama Algoritması \(Earley Parsing Algorithm\)](#)

[SORU 5: LL\(1\) Parçalama Algoritması](#)

[SORU 6: Maximal Munch \(Azami Lokma\) Yöntemi](#)

[SORU 7: Lisan-ı Kaime \(Dillerde Dik Aç, Orthogonal Languages\)](#)

[SORU 8: Kod Kelimesi](#)

[SORU 9: Mealy ve Moore Makineleri \(Mealy and Moore Machines\)](#)

[SORU 10: Preprocessor \(Ön işlem\)](#)

[SORU 11: Tek atama dili \(single assignment language\)](#)

[SORU 12: Hoare Mantığı \(Hoare Logic \)](#)

[SORU 13: Program doğruluğu \(Program correctness\)](#)

[SORU 14: Özyineli Geçiş Ağları \(Reursive Transition Networks\)](#)

[SORU 15: Turing Makinesi \(Turing Machine\)](#)

[SORU 16: Özyineli Sayılabilir Diller \(Recursively Enumerable Languages\)](#)

[SORU 17: Chomsky Hiyerarşisi \(Chomsky Hierarchy \)](#)

[SORU 18: Muntazam Diller \(Formal Languages\)](#)

[SORU 19: Değişken Tip Bağlama \(Dynamic Type Binding, Müteharrik Şekil Bağı\)](#)

[SORU 20: İşlem Önceliği \(Operator Precedence\)](#)

[SORU 21: Çok boyutlu diziler \(MultiDimensional Arrays\)](#)

[SORU 22: Sayma \(Enumeration, Tâdâd\)](#)

[SORU 23: Atomluluk \(Atomicity\)](#)

[SORU 24: Veri yapıları üzerinde fonksiyonlar](#)

[SORU 25: Filitreleme Tipi Fonksiyonlar \(Filter Type Functions\)](#)

[SORU 26: Biriktirme Tipi Fonksiyonlar \(Accumulator Type Functions\)](#)

[SORU 27: Bindirme Tipi Fonksiyonlar \(Mapping Style Functions\)](#)

[SORU 28: İçerik Bağımsız Gramerler için Pompalama Önsavı \(Pumping Lemma for Context Free Grammers\)](#)

[SORU 29: Düzenli İfadelerde Pompalama Önsavı \(Pumping Lemma for Regular Expressions\)](#)

[SORU 30: Pompalama Önsavı \(Pumping Lemma\)](#)

[SORU 31: İçerikten Bağımsız Gramer \(context free grammer, CFG\)](#)

[SORU 32: İçerikten bağımsız dil \(Context Free Language, CFL\)](#)

[SORU 33: EBNF \(Uzatılmış BNF, Extended Backus Normal Form\)](#)

[SORU 34: SableCC](#)

[SORU 35: Backus Normal Form \(BNF\)](#)

[SORU 36: Tek Geçişli Çevirici \(One Pass Assembler\)](#)

[SORU 37: Çevirici \(Assembler\)](#)

[SORU 38: NFA'den DFA'e çevirim \(Converting NFA to DFA\)](#)

[SORU 39: Belirsiz Sonlu Otomat \(Nondeterministic Finite Automat, NFA\)](#)

[SORU 40: Belirli Sonlu Otomat \(Deterministic Finite Automat, DFA\)](#)

[SORU 41: Dinamik Bağlantı Kütüphaneleri \(Dynamic Link Library \(.dll\)\)](#)

[SORU 42: XML \(extensible markup language , genişletilebilir işaretleme dili\)](#)

[SORU 43: Yorumlayıcı \(Interpreter\)](#)

[SORU 44: Bağlayıcı \(linker\)](#)

[SORU 45: Derleyici \(compiler\)](#)

[SORU 46: alt program \(subprogram, subroutine\)](#)

[SORU 47: fonksiyon göstericileri \(function pointer\)](#)

[SORU 48: otomat yönelimli programlama \(automata based programming\)](#)

[SORU 49: üst programlama yaklaşımı \(metaprogramming\)](#)

[SORU 50: fonksiyonel programlama \(functional programming\)](#)

[SORU 51: yapısal programlama \(structured programming\)](#)

[SORU 52: kapsülleme \(encapsulation\)](#)

[SORU 53: bit \(ikil\)](#)

[SORU 54: İşlem \(Process\)](#)

[SORU 55: Pointer \(Gösterici\) ve Diziler \(Arrays\)](#)

[SORU 56: Static Scoping \(Sabit Alanlı Değişkenler \)](#)

[SORU 57: Row Major Order \(Satır bazlı sıralama\)](#)

[SORU 58: Dyanmic Scoping \(dinamik alan değişkenleri\)](#)

[SORU 59: Coloumn Major Order \(Sütün bazlı sıralama\)](#)

SORU 1: Digraph ve Trigraphs

Basitçe bir programlama dilinde çeşitli sebeplerden dolayı bazı karakterlerin yazılması mümkün olmadığında digraph veya trigraph ismi verilen ve arka arkaya gelen 2 veya 3 karakterden oluşan ve aslında tek bir karakteri ifade etmek için kullanılan değerlere başvurulur.

Örneğin klavyede istenen karakterin bulunmaması veya bağlantıda istenen karakterin taşınmaması (remote shell veya telnet gibi ortamlarda kodlanması halinde) veya programlama ortamının istenen karakteri desteklememesi (mikro işlemci programlama durumları gibi) çeşitli hallerde bu özel karakter ikilileri veya üçlülerine başvurulur.

Aşağıda C dili için verilmiş bir tablo bulunmakadır ve bu tabloda bulunan karakterler C99 standartlarında belirlenmiştir ve bu standartları sağlayan bütün derleyicilerin (compiler) bu ikilileri ve üçlülerini desteklemesi beklenir:

Digraph Anlamı

<:	[
:>]
<%	{
%>	}
%:	#

Örneğin klasik olarak yazılan merhaba dünya mesajını ele alalım:

[View Code C](#)

```
1
2 #include <stdio.h>
3 int main(){
4     printf("merhaba dünya");
5 }
```

Yukarıdaki bu kodu aşağıdaki şekilde de yazabiliriz:

[View Code C](#)

```
1
2 %:include <stdio.h>
3 int main(<%
4     printf("merhaba dünya");
5 %>
```

Yukarıdaki kodun derlenip çalıştırılmış hali aşağıdaki şekildedir:

```
ses3:Downloads sadievrenseker$ cat deneme.c
%:include <stdio.h>
int main(<%
    printf("merhaba dünya");
%>
ses3:Downloads sadievrenseker$ gcc deneme.c
ses3:Downloads sadievrenseker$ ./a.out
merhaba dünyases3:Downloads sadievrenseker$
```

Görüldüğü üzere ilk yazılan merhaba dünya örneğinden hiç farkı yoktur ve aynı şekilde çalıştırılır.

Ayrıca trigraph listesi aşağıdaki şekilde verilebilir:

Trigraph Anlamı

??=	#
??/	
??'	^
??([
??)]
??!	
??<	{
??>	}
??-	~

Yukarıdaki haline göre kodumuzu yeniden yazacak olursak :

[View Code C](#)

```
1 ??=include <stdio.h>
2 int main()??<
3     printf("merhaba dünya");
4 ??>
```

```
ses3:Downloads sadievrenseker$ cat deneme.c
??=include <stdio.h>
int main()??<
    printf("merhaba dünya");
??>
ses3:Downloads sadievrenseker$ gcc -trigraphs deneme.c
ses3:Downloads sadievrenseker$ ./a.out
merhaba dünyases3:Downloads sadievrenseker$
```

Burada dikkat edilmesi gereken konulardan birisi de gcc derleyicisine parametre olarak -trigraphs veriliyor olmasıdır. Bazı derleyiciler bunu doğrudan desteklerken bazı derleyicilere, [derleme sırasında \(compile time\)](#) bu parametrenin özel olarak verilmesi gerekmektedir.

SORU 2: LR(1) Parçalama Algoritması

Algoritma, özellikle derleyici tasarımı konusunda sık kullanılan parçalama algoritmalarından birisidir. Algoritmanın ismi, iki kelimenin kısaltmasından gelmektedir. Buna göre ilk L harfi, left to right parsing (soldan sağa doğru parçalama) ve ikinci R harfi ise rightmost derivation (sağdan eksiltmeli) anlamındadır. Parantez içerisinde yer alan 1 sayısı ise, 1 look ahead, yani 1 sembol ilerisine bak anlamındadır. Bu durumda algoritma LR(0) algoritmasının bir seviye gelişmişisi olarak düşünülebilir. LR(0) algoritması ile tek farkı, ileri bakış değerinin 1 olması ve bu sayede LR(0) tarafından parçalanamayan bazı gramerleri parçalayabilmesidir.

Bu yazı kapsamında, öncelikle örnek bir gramerin DFA çizimini göreceğiz. Algoritmada, bu DFA çiziminin ardından gelen herhangi bir girdiyi (input) DFA üzerinde dolaşarak parçalamak gelir. Bunun için de örnek bir girdi vererek nasıl çalıştığını göstereceğiz. Son olarak LR(1) algoritmasının kısıtlarını ve yapamayacaklarını inceleyeceğiz.

Örnek:

$S \rightarrow aAd$

$S \rightarrow bBd$

$S \rightarrow aBe$

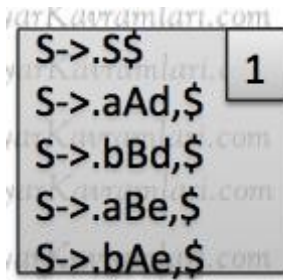
$S \rightarrow bAe$

$A \rightarrow c$

$B \rightarrow c$

Yukarıdaki şekilde verilen gramer (CFG şeklinde verilmiştir ve daha detaylı bilgi için CFG başlıklı yazı okunabilir) için DFA çizimi aşağıdaki şekilde yapılır.

Öncelikle başlangıç durumunu ele alıyoruz. Gramer tanımı itibariyle S, başlangıç ifade etmektedir. O halde aşağıdaki şekilde başlayabiliriz:

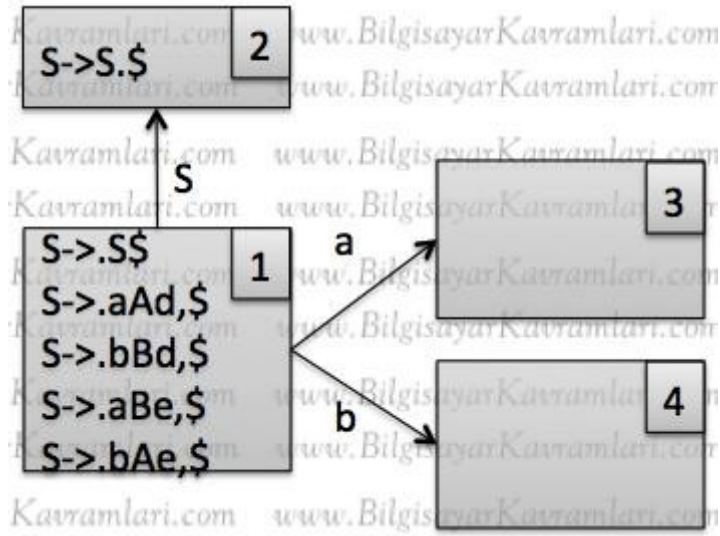


$S \rightarrow .S\$$	1
$S \rightarrow .aAd, \$$	
$S \rightarrow .bBd, \$$	
$S \rightarrow .aBe, \$$	
$S \rightarrow .bAe, \$$	

Yukarıda görüldüğü üzere, öncelikle S gelmesi ve sonrasında \$ sembolü olan girdinin bitmesi (end of input) halinde bu kabul anlamına gelmektedir. Dolayısıyla $S \rightarrow .S\$$ kuralı bizim bitişimizi ifade etmektedir. Burada . işareti, o ana kadar olan parçalama durumunu gösterir. Yani o ana kadar parçalanmış semboller ve o anda sıradaki sembolü göstermektedir.

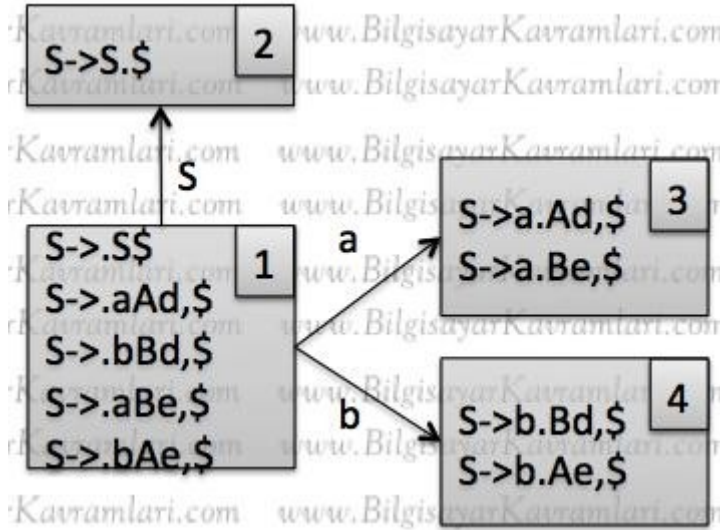
Yukarıdaki ilk kurala bakacak olursak, $S \rightarrow .S\$$ ifadesi, sırada S sembolü olduğunu göstermektedir. Bu durum aslında özel olarak ele alınması gereken bir durumdur. Buna göre sıradaki beklenen terim bir devamlı (non-terminal) olarak görülmektedir. Böyle özel bir halde, bu devamlının (non-terminal) tanımını içeren kurallar da DFA çizimindeki duruma eklenmelidir. Yukarıdaki durumda bulunan diğer 4 kuralın geliş sebebi işte budur.

Şimdi yukarıda çizdiğimiz 1 numaralı durumdan sonra devam için gelebilecek sembollerini inceleyelim. İlk kural itibariyle bir S, ikinci ve 4. kurallar itibariyle bir a ve üçüncü ve beşinci kurallar itibariyle de bir b sembolü beklenmektedir. Bütün bu semboller için çizdiğimiz durumdan devam yollarını çizmemiz gerekiyor:

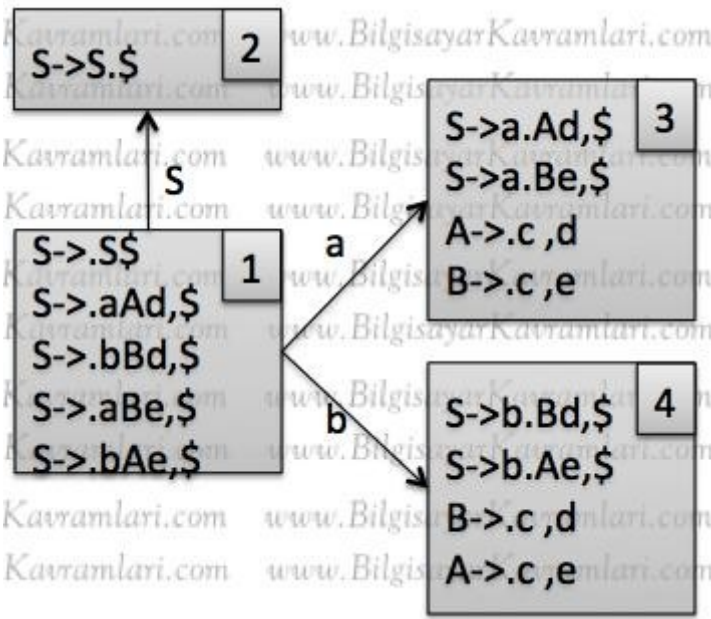


Yukarıdaki şekilde gösterildiği üzere, 1 numaralı durumdan çıkan 3 ayrı ok, 3 ayrı duruma geçişi temsil etmektedir. Bunlardan ilki olan 2 numaralı durumda, bir kaydırma (shift) işlemi gerçekleşmiş ve noktanın yeri bir kaydırılmıştır. $S \rightarrow S. \$$ gösterimi, S gelmesi halinde, 1 numaralı durumdan gidebileceğimiz ve gittiğimizde gramerimizde olacak değişikliği göstermektedir. Burada dikkat edilecek bir husus, sıradaki sembolün \$ olmasıdır. Yani bir şekilde girdinin (input) sonuna ulaşıldıysa, artık bu girdi kabul edilmiş demektir. Bu anlamda, 2 numaralı duruma bir kabul durumu denilebilir.

Şimdi, diğer durumları doldurmaya çalışalım:

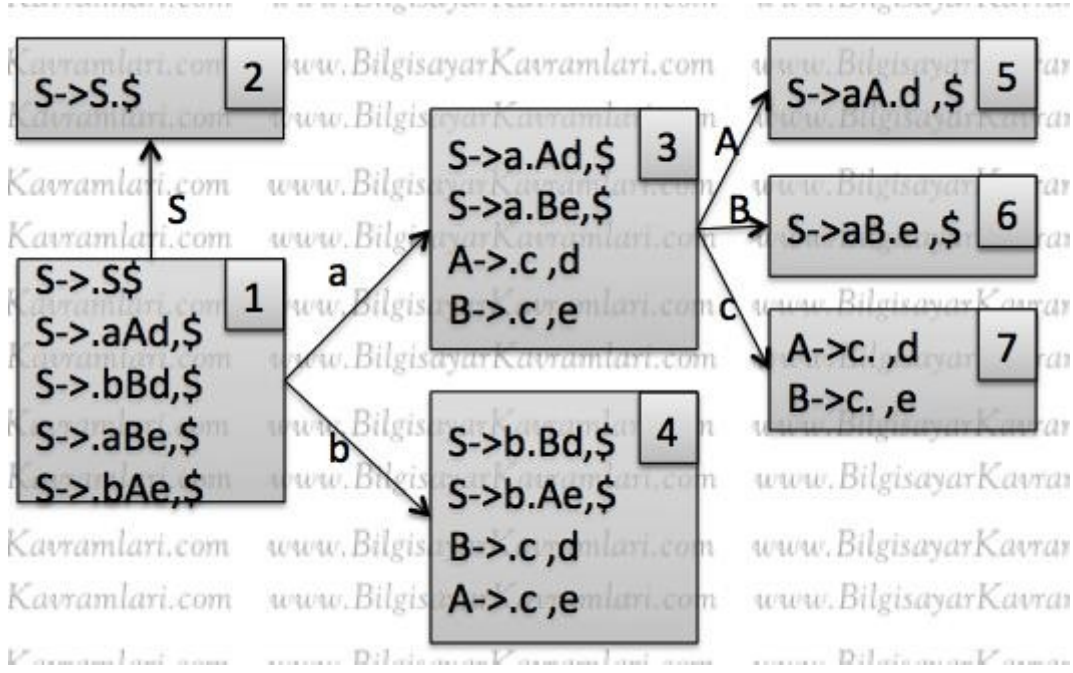


Yukarıda, her sembolün gelmesi durumunda yapılması gereken kaydırma işaretleri tutulmaktadır. Dikkat edilirse, yine sıradaki sembolün devamlı (non-terminal) olduğu koşullar ile karşı karşıyayız. Bu sembollerin açılımını yapıyoruz:

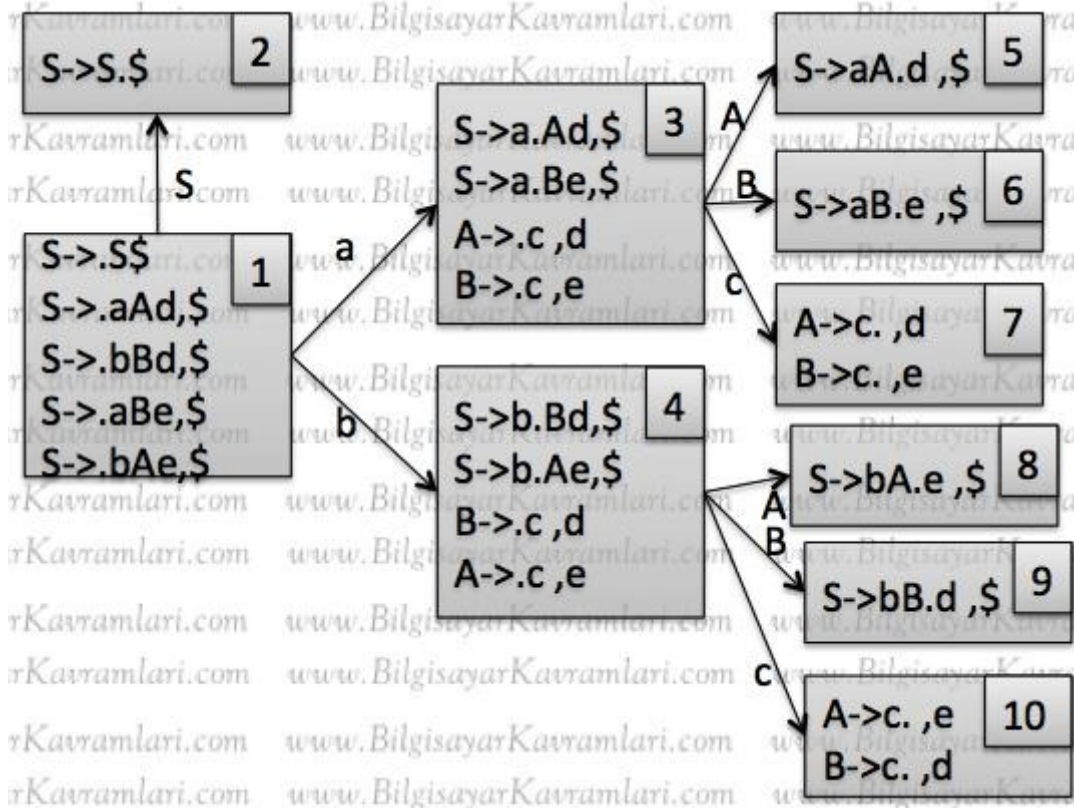


Gösterimimizde, 3. ve 4. durumlar için karşılaşılan devamlı (non-terminal) sembollerin tanımları yerleştirilmiştir. Örneğin 3. durumda, $S \rightarrow a \cdot A d, \$$ koşulu, bize a teriminin işlendiği sıradaki terimin A olduğu ve virgülden sonra \$ geldiği için look ahead (bakış) değerinin \$ olduğunu göstermektedir. Burada, sıradaki terimin A olmasından dolayı A teriminin açılımını yaparak $A \rightarrow c$ kuralını, duruma ekliyoruz. Bu eklemede dikkat edilecek iki husus bulunuyor. Birincisi eklenen yeni kuralda, henüz c sembolü işlenmediği için $A \rightarrow \cdot c$ şeklinde noktanın c sembolünden önce yerleştirilmesi, ikincisi ise virgülden sonraki terim olarak d sembolünün yerleştirilmesidir. Yani $S \rightarrow a \cdot A d, \$$ kuralında A gelmesi halinde sıradaki beklenen olan sembol d olacağı için $A \rightarrow \cdot c$ kuralında da bakış değerimiz (look ahead) d sembolü olacaktır.

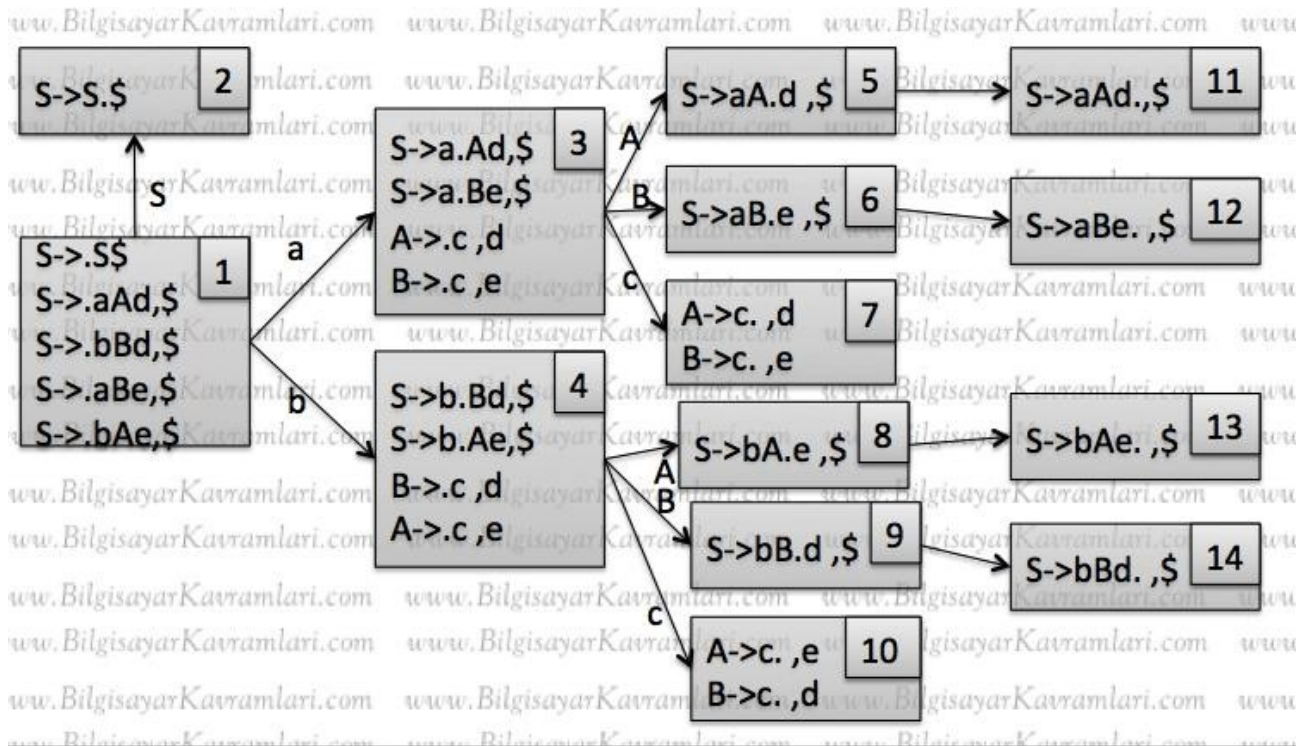
Burada LR(1) algoritmasının, LR(0) algoritmasından farkını görebiliriz, yani durumların içerisine yazılan kurallarda, bakış değerine göre aynı kural, farklı satırlar halinde yazılabilir. Örneğin $A \rightarrow \cdot c, d$ ile $A \rightarrow \cdot c, e$ aynı kural olan $A \rightarrow c$ kuralından gelmesine karşılık, devamında beklenen terimlerin farklı olmasından dolayı durumun içerisinde yer alır. Biz DFA çizmimize devam edelim, 3. durumda beklenen semboller ve bu sembollerin gelmesi durumunda gidilecek durumlar aşağıdaki şekilde çizilebilir:



Yukarıda, yeni eklenen 3 farklı durum bulunmaktadır. Burada özel olarak birşeyi belirtmek istiyorum, klasik birer kaydırma durumu olan 5. ve 6. durumların aksine 7. durum oldukça önemlidir ve iyi anlaşılması gerekir. 7. durumda iki adet ikame(reduce) işlemi bulunmaktadır. Yani bir c sembolü geldiğinde yerine A devamlısı veya B devamlısı (non-terminal) konulabilir. Aslında bu durum, LR(0) tam bir problemdir ve böyle bir durumu içeren LR(0) algoritması, hemen bu grameri parçalayamayacağını söyleyebilir. Ancak burada dikkat ederseniz c sembolünün A mı B mi olacağı devamında gelen d veya e sembolüne göre belirlenebilmektedir. O halde algoritmamız devamında gelecek olan sembole göre karar verebilir ve klasik ikame / ikame (reduce / reduce) problemi olan problemin üstesinden gelebilir. Bu durumu bir örnek girdiyi işlerken daha net göreceğiz. Sıradaki 4. durumu açalım:

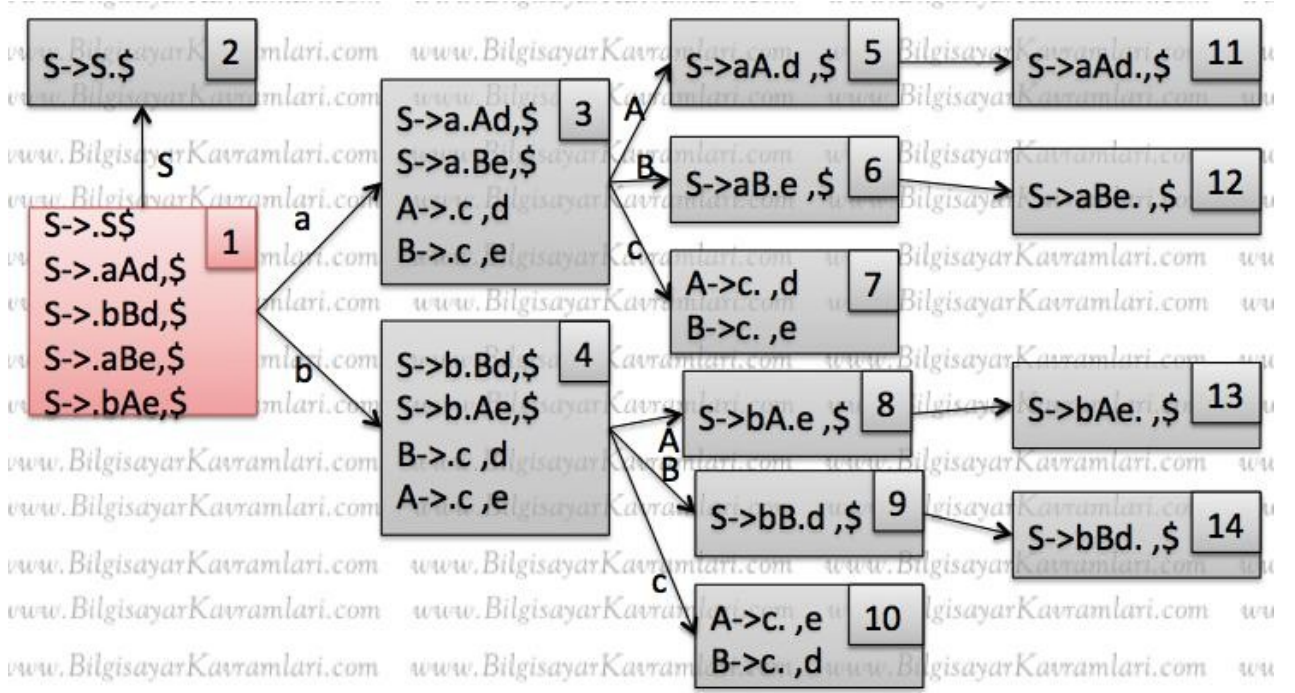


Görüldüğü üzere, bir önceki adıma benzer şekilde 3 durum çıktı ve bu durumlardan 10. durum, 7. duruma benzer şekilde bir ikame / ikame (reduce / reduce) durumudur. Buradan sonra hızlanarak bütün DFA'nın çizilmiş halini verelim:

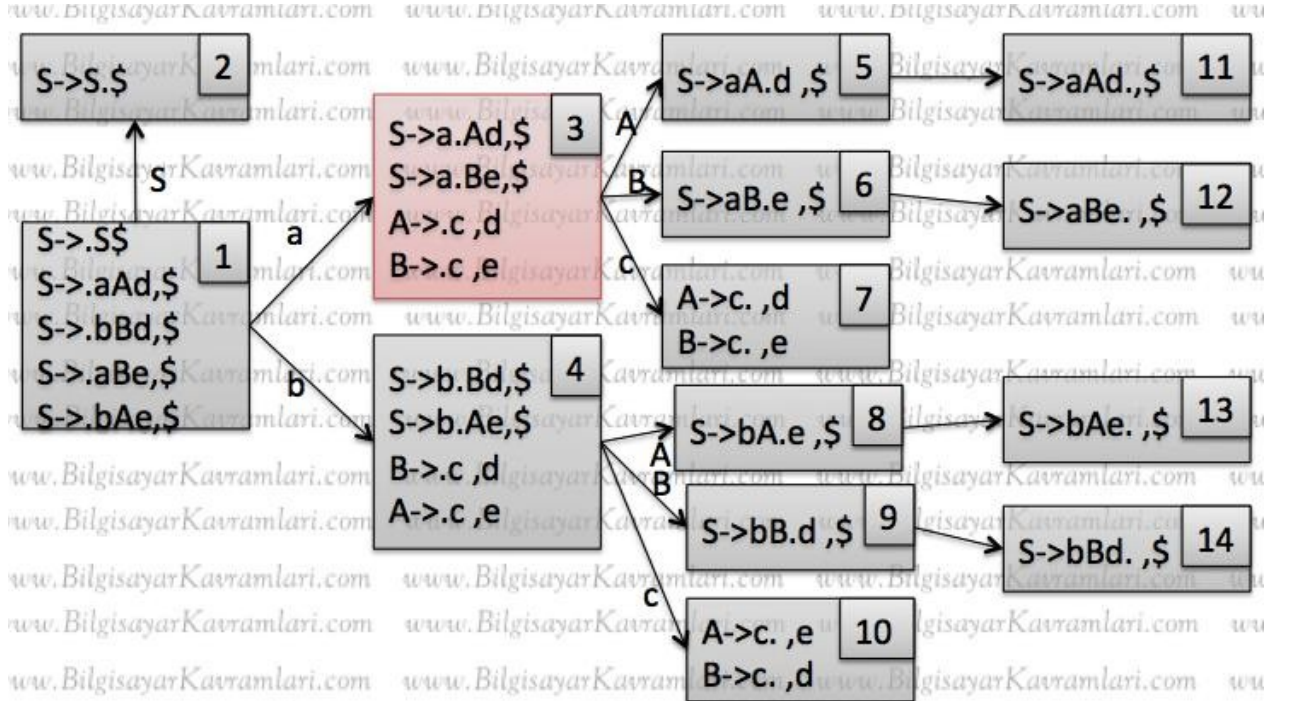


Son haliyle DFA çizimimizi tamamladık. Şimdi bu DFA'nın bir örnek girdi üzerinde nasıl çalıştığını görelim. Örnek girdimiz "acd" olsun ve bu girdiyi nasıl parçaladığımızı görelim.

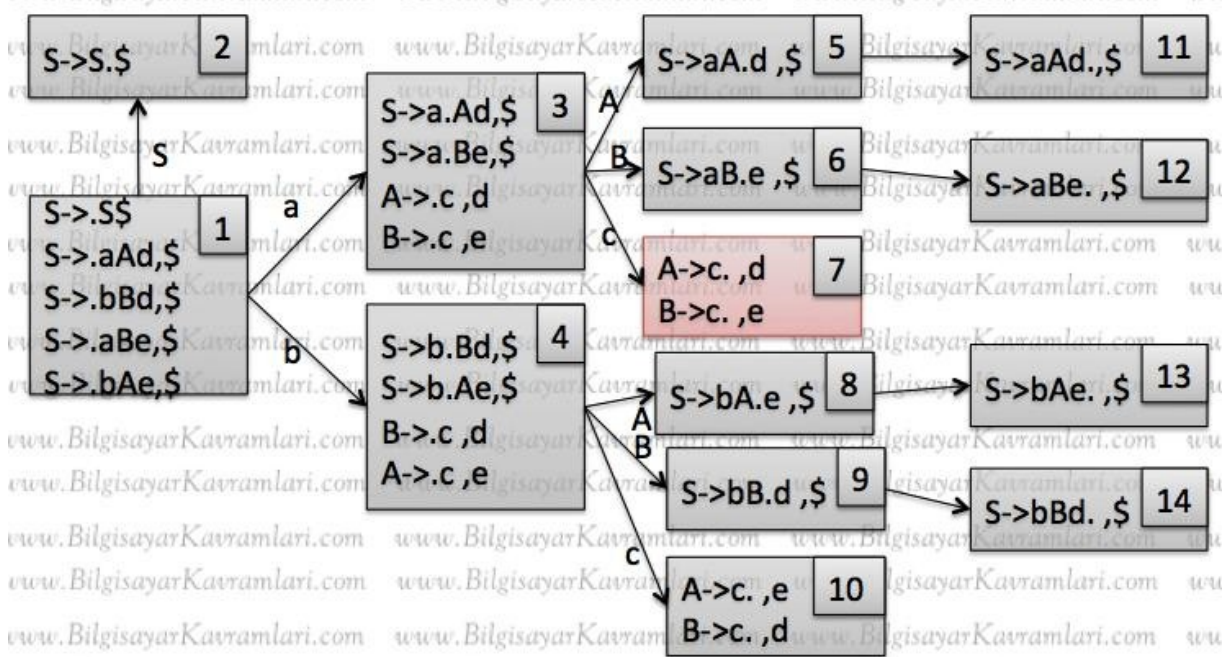
Öncelikle başlangıç durumundan başlıyoruz:



Bu durumdayken, ilk gelen girdi sembolümüz a olduğu için ilgili yolu izliyoruz:



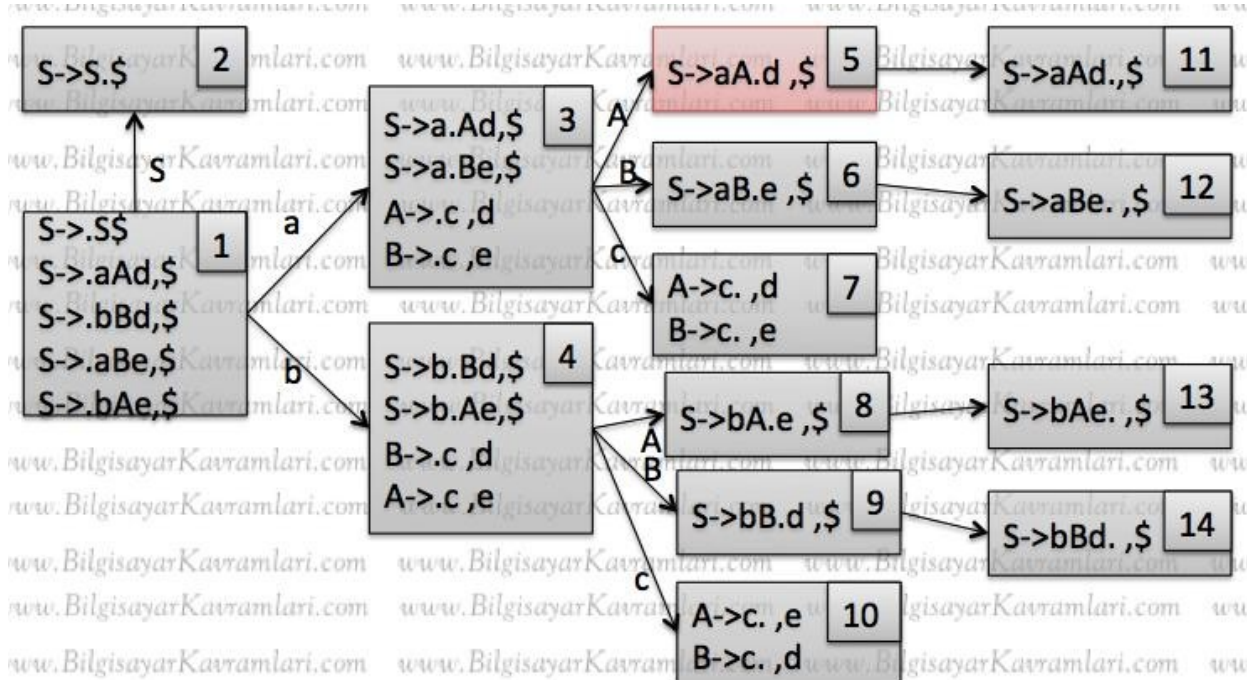
Buraya 1 numaralı düğümden geldik ve sıradaki sembol bir c dolayısıyla sıradaki yolu izleyerek 7 numaralı düğüme geçiyoruz:



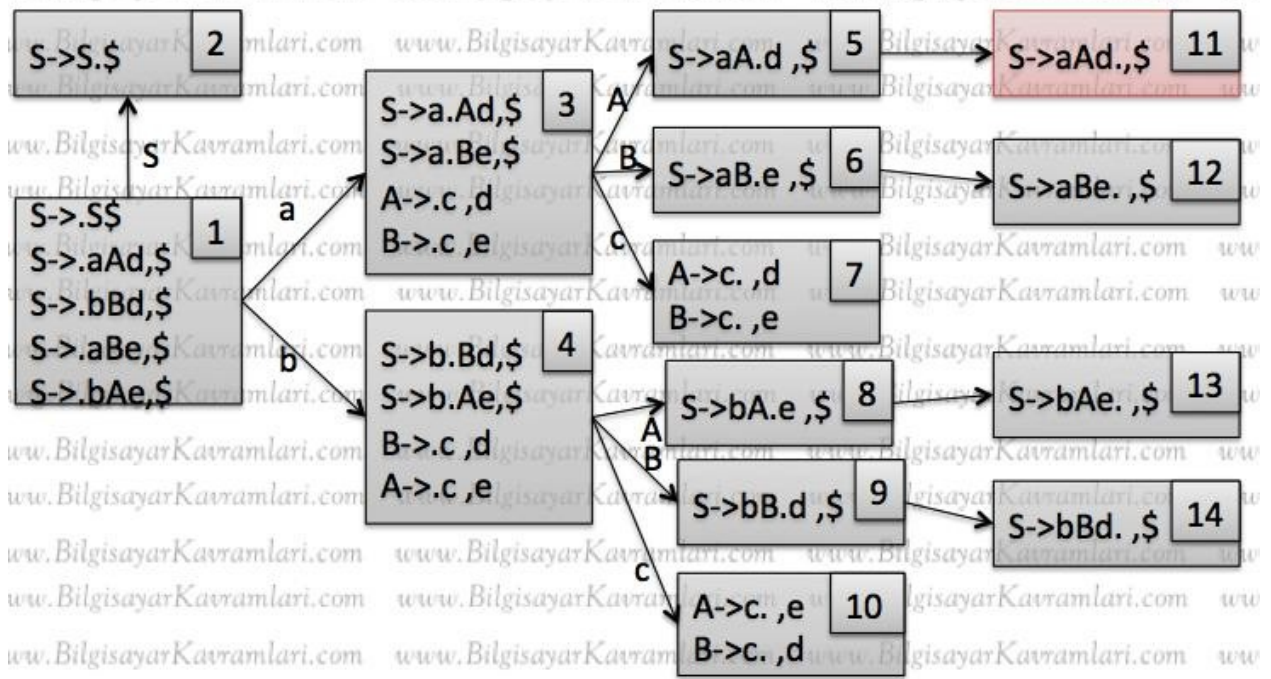
Şimdi artık bir ikame (reduce) durumu ile karşılaştığımız için, c yerine bir devamlı (non-terminal) koyabiliriz. Ancak sorun A veya B devamlılarından (non-terminal) hangisinin geleceğidir.

Girdimiz “acd” olduğuna göre ve biz c sembolünü işlediğimize göre sıradaki terime bakıp d olduğunu görebiliriz. O halde şu anda bulduğumuz c sembolü yerine sıradaki terim d olduğu için A konulması sonucuna varmak gayet kolaydır.

“13|.Ad” şeklinde girdimizi güncelliyoruz, yani 1. ve 3. durumları aştık ve sırada A terimini işlemek var.

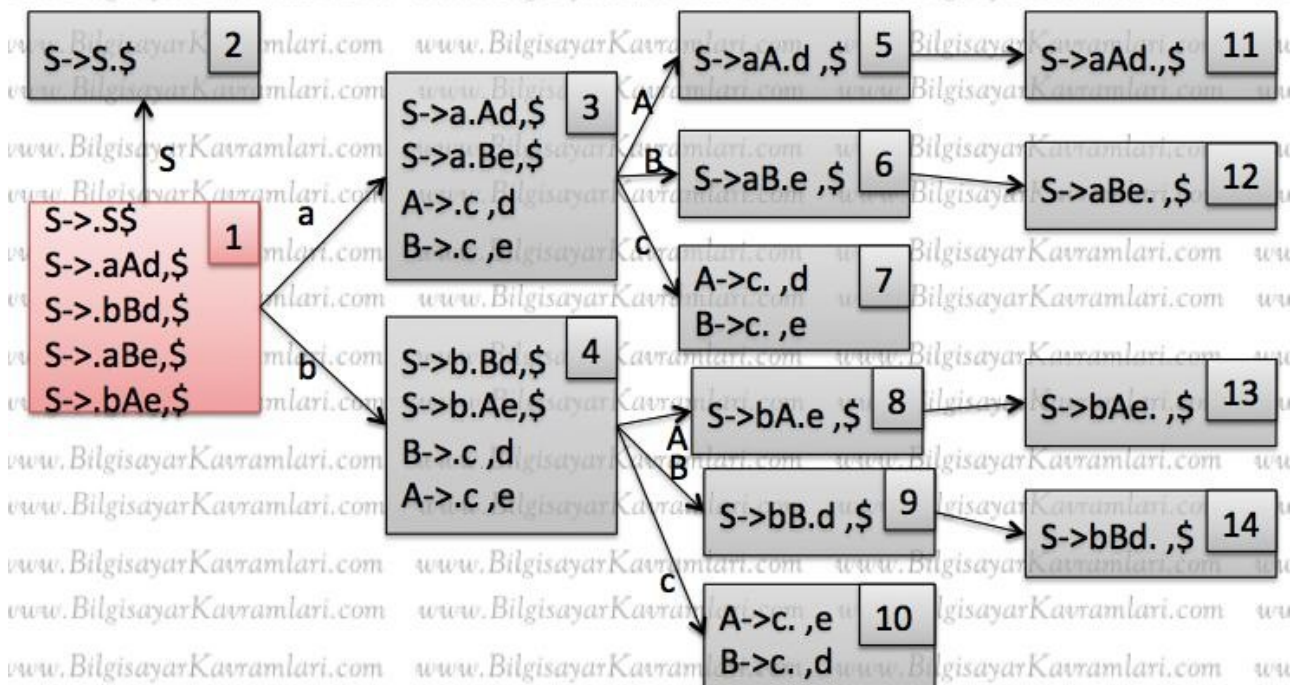


Bu bizi, 5. duruma götürür. Girdimiz, “135|.Ad” şeklini almıştır ve sıradaki girdimiz de d sembolüdür.



Şimdi yeni bir ikame (reduce) durumu olan 11. duruma ulaştık. Yapmamız gereken işlem aAd yerine S koymaktır. Girdimiz buraya geldikten sonra “ 1,3,5,11 | d. “ şeklini aldı. O halde sıradaki işlem 3,5,11 terimlerinin yerine (yani a,A,d terimleri olduğu görülebilir) S koymaktır:

“1.S” durumuna dönmüş oluruz ki bu da bizim 1. durumdan başlayarak S işletmemiz gerektiğini ifade eder:

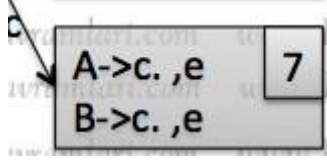


Şeklindeki başlangıç durumundan S sembolü işletilirse, kabul durumu olan 2. duruma ulaşılır ve bu girdinin kabul edildiği ve hatasız olduğu anlamına gelir.

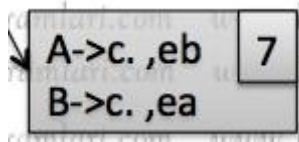
LR(1) algoritması başarıyla çalışmıştır ve LR(0) için problem olan bir reduce/reduce durumunu başarıyla çözmüştür.

Peki LR(1) algoritmasının limitleri var mıdır? Bütün gramerler LR(1) ile işletilip girdiler başarıyla çözülebilir mi?

Bu sorunun cevabı ne yazık ki hayırdır. Yani LR(1) de bazı limitlere sahiptir. Örneğin aşağıdaki şekilde bir reduce / reduce durumu LR(1) için de problem oluşturur.



Görüldüğü üzere burdaki ikame / ikame (reduce /reduce) problemi ne yazık ki takip eden karaktere bakılarak çözülememektedir. Yani tek bir sembole bakmak yeterli olmaz. Belki ikinci bir sembol, yani sıradaki sembolden sonra gelen sembol de tutularak çözülebilir:



Şayet gramer bize ikinci sembolde bir farklılık yakalama imkanı tanıyorsa o zaman çözüm bulunabilir denilebilir. Zaten bu şekilde 2 sonraki terime kadar bakan algoritmaya LR(2) ve daha fazla terime bakan algoritmalar da LR(n) şeklinde isim verilir (buradaki n herhangi bir sayı olabilir , örneğin LR(3) veya LR(15) gibi). Ancak dikkat edilecek bir husus, buradaki sayı arttıkça DFA çiziminin karmaşıklaştığı ve algoritmanın işletilmesinin zorlaştığıdır.

SORU 3: LR(0) parçalama algoritması

Algoritmanın ismi, iki harften L ve R harflerinden gelmektedir. İlk L harfi, Left to Right parsing (soldan sağa parçalamalı) anlamında, ikinci R harfi ise Rightmost derivation (sağdan azaltmalı) anlamındadır. Algoritmanın isminde bulunan 0 sayısı ise look ahead (ileri bakma sayısının) 0 olduğunu gösterir. Yani parçalama algoritmamız, sadece o andaki terim (harf) ile ilgilenmekte ve daha sonra gelecek olan terimlere bakmamaktadır.

Algoritma, basit bir DFA çizimi ile başlar ve dilin tanımında (genellikle CFG şeklinde verilir) bulunan durumlar bu sonlu otomat (finite automaton) üzerinde gösterilir. Ardından alınan bir girdi (input) için bu otomat üzerinde işlem yapılarak girdi parçalanmaya çalışılır.

Bu yazının içeriğinde algoritma incelenirken yukarıda bahsedildiği üzere iki adımda incelenecektir. İlk adımda DFA çizimini ve aşamalarını göreceğiz, ikinci adımda ise örnek bir girdi için algoritmanın DFA üzerinde nasıl çalıştığını göreceğiz.

Yazının sonunda ise LR(0) algoritmasının sınırlarını anlatacak ve parçalayamayacağı bazı özel dilleri ve neden parçalayamadığını açıklayacağız.

Örnek1 :

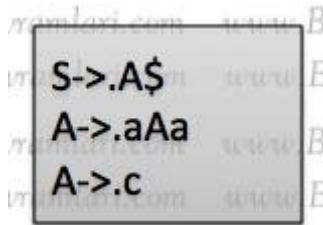
Örnek olarak aşağıdaki grameri ele alalım.

$S \rightarrow A\$$

$A \rightarrow aAa$

$A \rightarrow c$

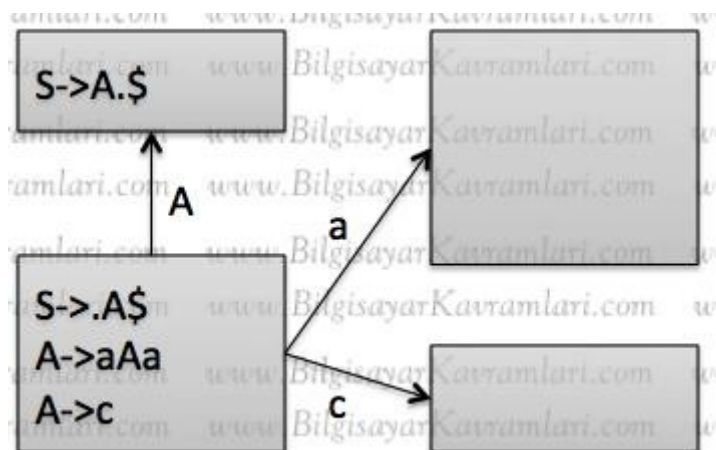
Bu gramer için öncelikle bir DFA oluşturarak işe başlayalım. Başlangıç durumunda (starting state) yukarıdaki dilin S durumu ele alınmalıdır:



Yukarıdaki şekilde, ilk önce $S \rightarrow A\$$ ifadesi yerleştiriliyor (şeklin sol alt köşesinde). Bu ifade yerleştirildiğinde bir nokta (.) sembolü, o ana kadar parçalanmış (işlenmiş, parse) kısmı tutmak üzere yerleştiriliyor. Buna göre $S \rightarrow .A\$$ ifadesi, henüz hiçbir terimin işlenmediğini ve sıradaki işlemenin A terimi olduğunu ifade ediyor.

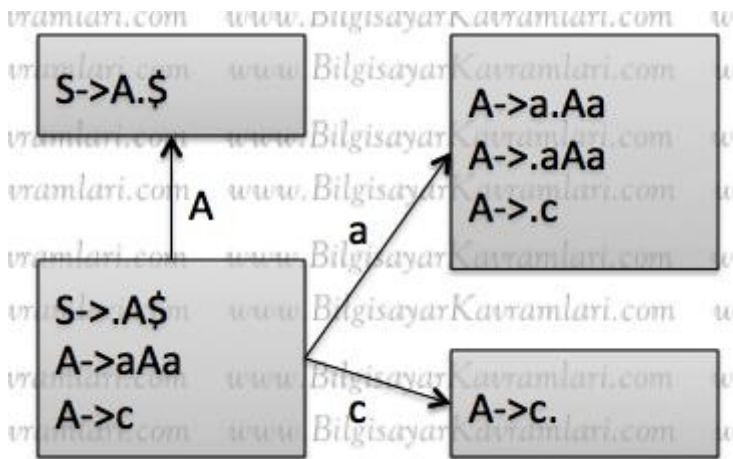
Bu durum aslında özel bir durumdur, yani sıradaki terim bir devamlı (non-terminal) ifadedir ve bu ifadeler büyük harfle gösterilir ve girdide (input) yer almaz ve gramerde daha karmaşık bir ifadeye dönüşüm için kullanılır (bkz. CFG konusu). Bu durumda, yani bir devamlının (non-terminal) sırada olduğu durumlarda bu devamlı da, gramerdeki tanımı itibarıyla açılır ve o andaki duruma eklenir. İçinde bulunduğumuz duruma, A devamlısının (non-terminal) açılımı olan kuralları da bu yüzden ekliyoruz.

Sırada gelebilecek ifadelerin işlenmesi var. Şu anda beklediğimiz ve gelebilecek semboller $\{A, a, c\}$ kümesindeki sembollerdir. Her sembol için farklı bir yol çiziyor ve yeni birer durum işaretliyoruz:



Yukarıdaki yeni haliyle, DFA çizimimiz 3 farklı yola gidebilmektedir. Bunlardan A gelmesi durumu basit bir durumdur ve bu durum shift (kaydırma) olarak değerlendirilir. Dikkat edilirse burada noktanın konumu ilerletilmiştir ve sıradaki terim \$ olarak bulunduğu için (\$

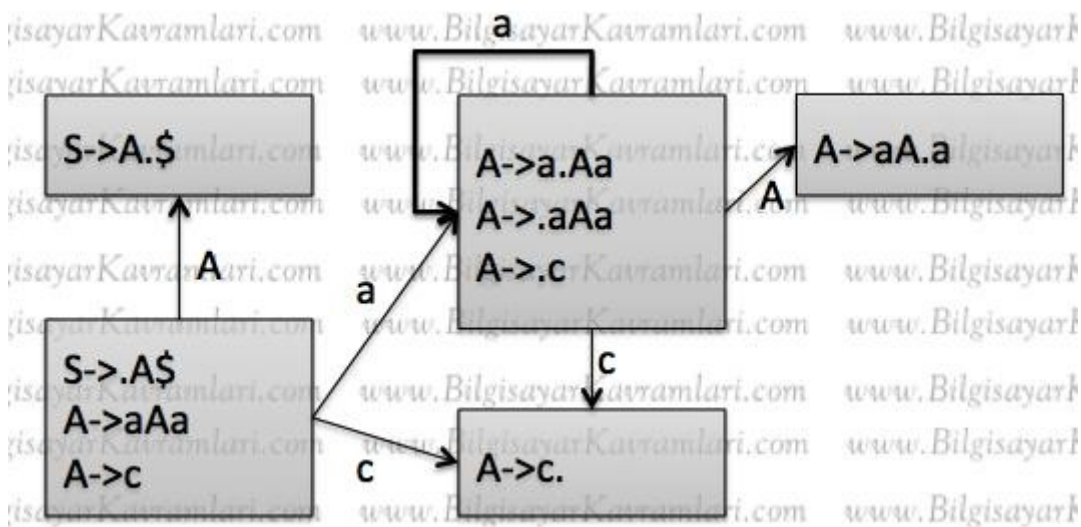
girdinin sonu, end of input anlamındadır) makine kabul durumu olarak biter. Diğer durumlar ise henüz doldurmadığımız durumlardır ve bu durumları sırasıyla inceleyelim.



Öncelikle “a” gelmesi durumuna bakalım. Bu durumda gramer taranımımızda bulunan ve a terimini bekleyen tek kural olan $A \rightarrow \cdot aAa$ kuralındaki nokta ilerletiliyor ve $A \rightarrow a \cdot Aa$ kuralı olarak yazılıyor. Ancak yine bir devamlı (non-terminal) olan A terimi ile karşılaşılıyor. Bu durumda, daha önce yaptığımız gibi bu devanlının (non-terminal) açılımını da durumun içerisine yazıyoruz.

Diğer bir ihtimal ise $A \rightarrow \cdot c$ beklentisini tatmin ederek $A \rightarrow c \cdot$ şeklindeki kaydırma (shift) işlemidir. Bu durum da yukarıdaki şekilde gösterilmiştir (sağ alt köşedeki durum). Bu gösterilen durum aslında özel bir durumdur ve buna ikame (reduce) durumu denir. Yani $A \rightarrow c$ şeklindeki açılım aslında c olan yere A yazılabileceği anlamını taşır ki bunu örnek bir girdinin parçalanması sırasında daha net anlayacağız.

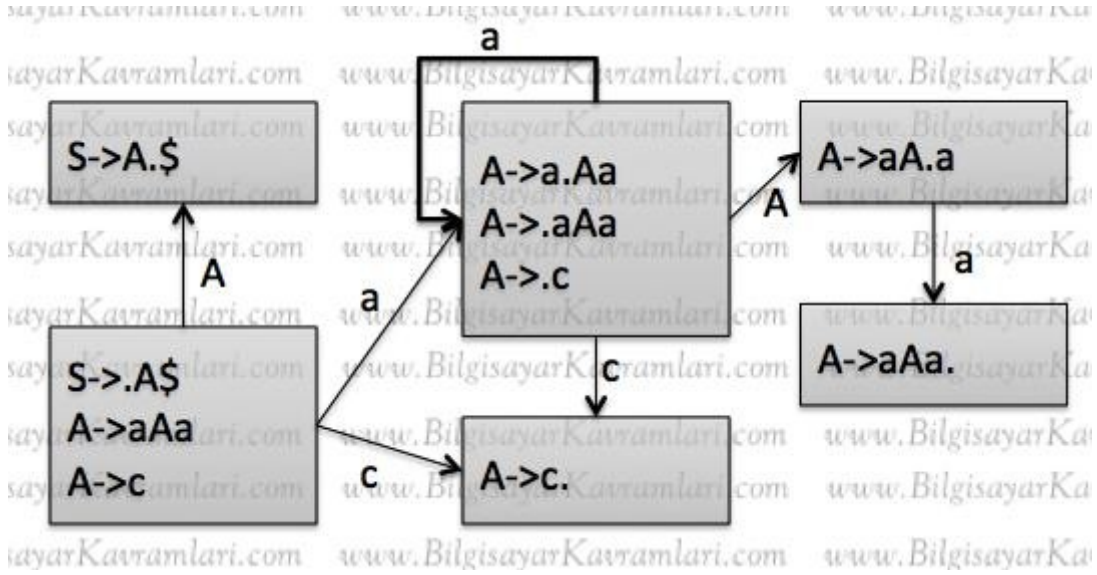
Sırada diğer beklentiler var. Sağ üst köşede duran durumdaki beklentilerimiz $\{A, a, c\}$ kümesindeki ifadelerdir ve bunların açılımlarını aşağıdaki şekilde gösterebiliriz:



Yukarıdaki yeni haliyle DFA çizimimizde gelebilecek 3 durum açılmış ve sadece $A \rightarrow aA.a$ ile gösterilen ve A gelmesi durumunu ifade eden ihtimal için yeni bir durum eklenmiştir. Diğer ihtimaller için mevcut durumlar kullanılmıştır. Burada a gelme ihtimali dikkate alınmalıdır.

Buna göre a geldiğinde yine aynı durumda olacağız ancak bu durumun altında yeni bir durum açılacaktır. Bunu parçalama anında göstereceğiz.

Gelelim son açılmamış durum olan ve bir önceki adımda yeni eklenen sağ üst köşedeki duruma:

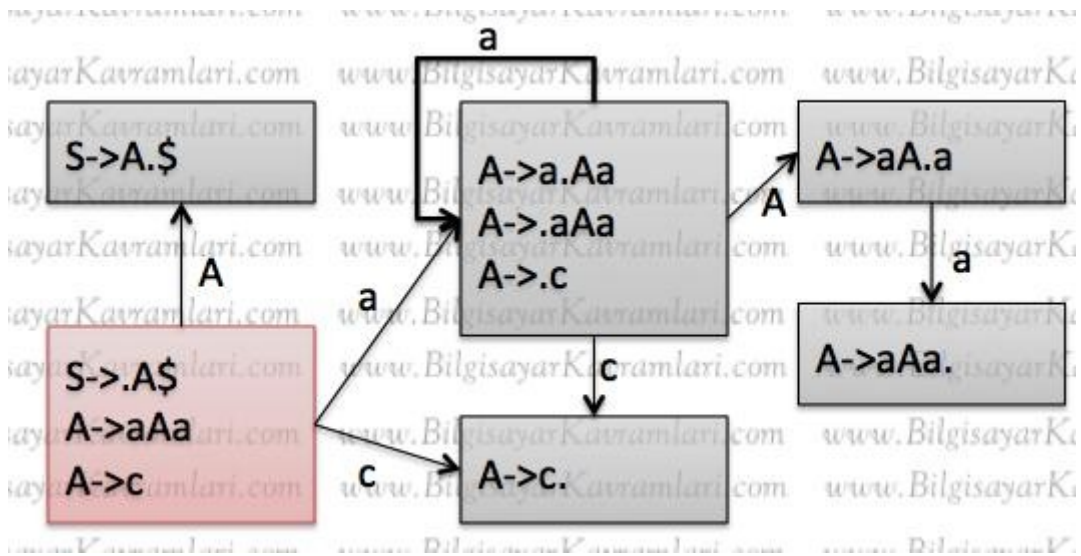


Yeni haliyle DFA çizimimizde, son ihtimalin beklentisi olan a terimi geldiğinde yine bir ikame(reduce) durumu eklenmiştir.

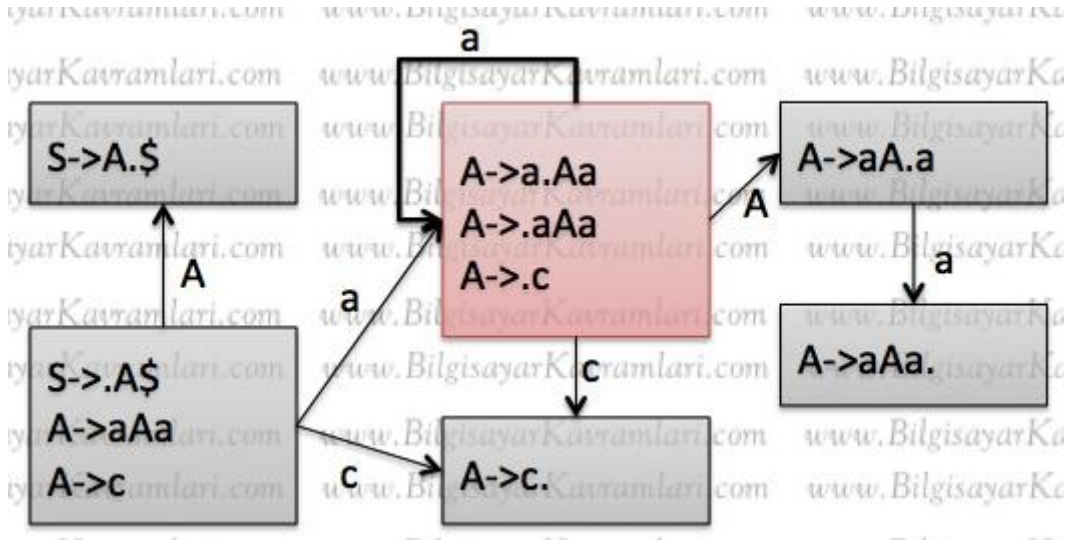
Artık DFA çizimimizi bitirdik. Bir sonraki aşama olan bir girdinin parçalanmasını adım adım inceleyebiliriz. Örnek olarak “aaca” girdisini parçalamak istiyor olalım.

Öncelikle bu girdinin parçalanması sırasında yaşanacak bir problemi göstermek istiyorum. Ardından bu problemin çözümü için DFA üzerinde ufak bir hile yapacağız.

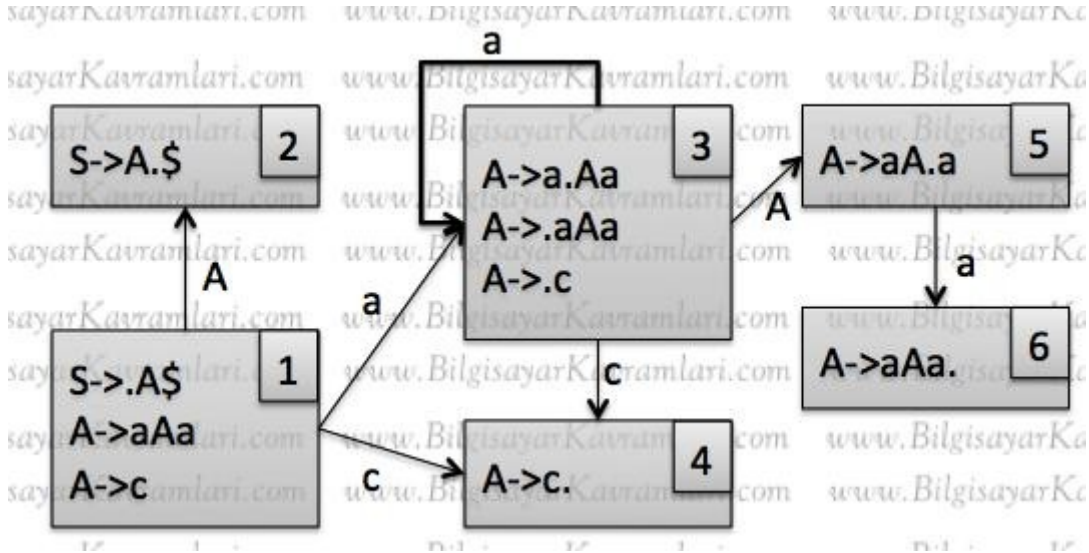
Girdimiz aaca olduğuna göre başlangıç durumunda parçalama işlemine başlıyoruz.



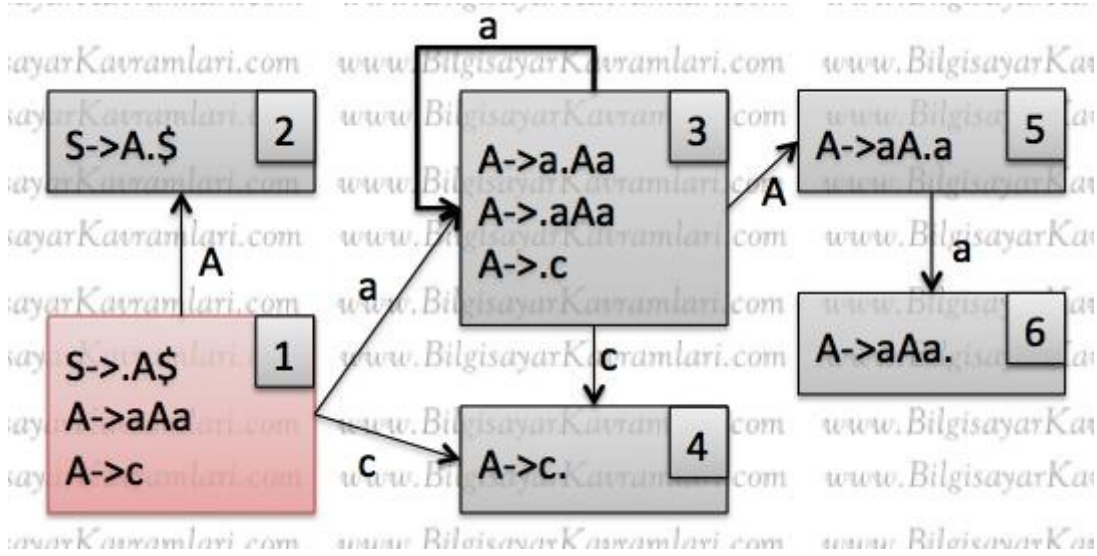
Başlangıç durumundayken, gelen ilk terim olan “a” için gideceğimiz durum DFA üzerinde açıkça tanımlanmış. Bu yola devam ediyoruz ve girdimizden “a” terimini işlediğimiz için siliyoruz. Yeni durumda girdimiz “abaa”t şeklinde düşünülebilir:



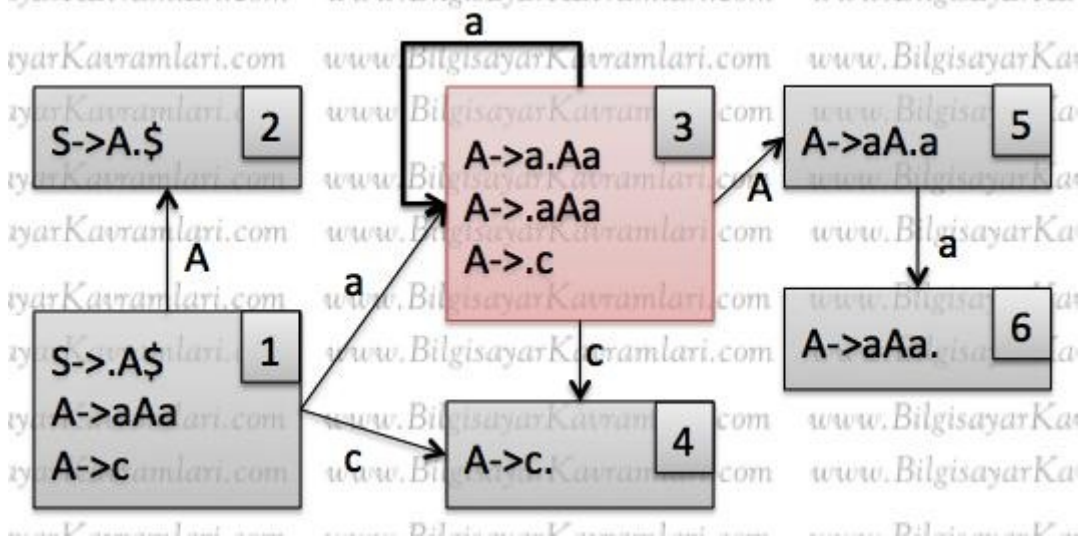
Sıradaki terim yine bir “a” sembolü. Yapacağımız işlem aynı durumda kalmak. Ve girdiyi “baa” şekline indirgemek. Ancak gel gelelim bu durumda kaç kere durduğumuzu kaybediyoruz. Yani “aba” girdisi, “aabaa” girdisi veya “aaabaaa” girdilerinin tamamı için bu durumda kalacağız ve bu gerçek bizim parçalama işlemimizi zora sokuyor. Bunun çözümü için durumlara birer sayı ataması yapmak akıllıca bir yöntemdir.



Yukarıdaki şekilde, her durumda (state) bir numara bulunmaktadır. Artık takip işlemi yapılabilir. Nasıl yaptığımızı baştan itibaren görelim. Yine “aaca” girdisi için başlangıç durumundan başlıyoruz:

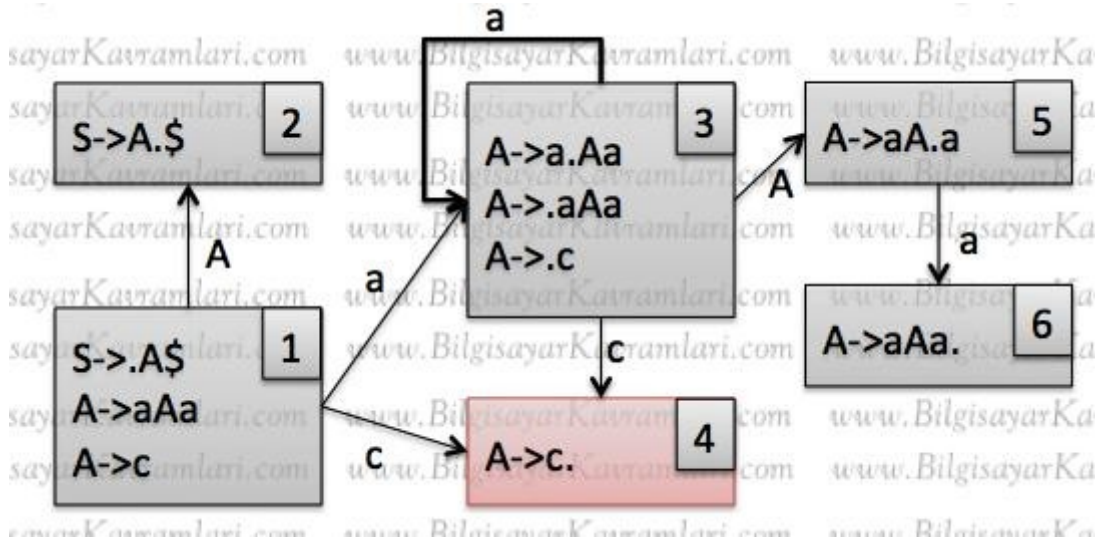


ilk terim için devam yolu olan “a” yolunu seçip 3. duruma geçiyoruz ve şimdiye kadar geçtiğimiz düğümleri bir listede tutuyoruz:

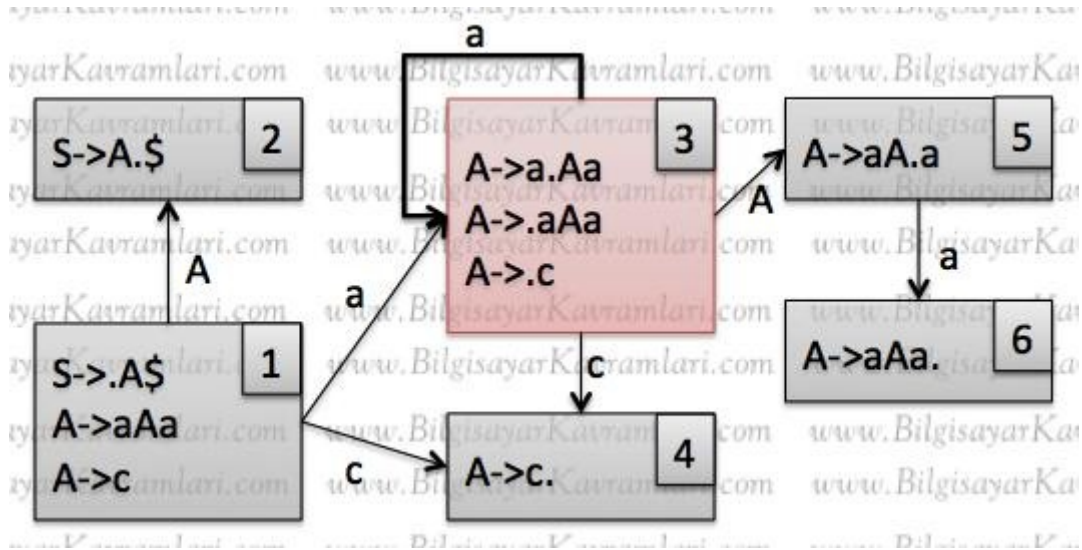


Yeni halinde girdimiz “13.aaa” şeklini almıştır. Devam edelim, sıradaki girdimiz bir “a” terimi. O halde, yine aynı durumda kalacağız ancak girdimiz “133.caa” şeklini alacak. Dikkat ederseniz artık kaç kere 3. durumda kaldığımızı sayabiliyoruz.

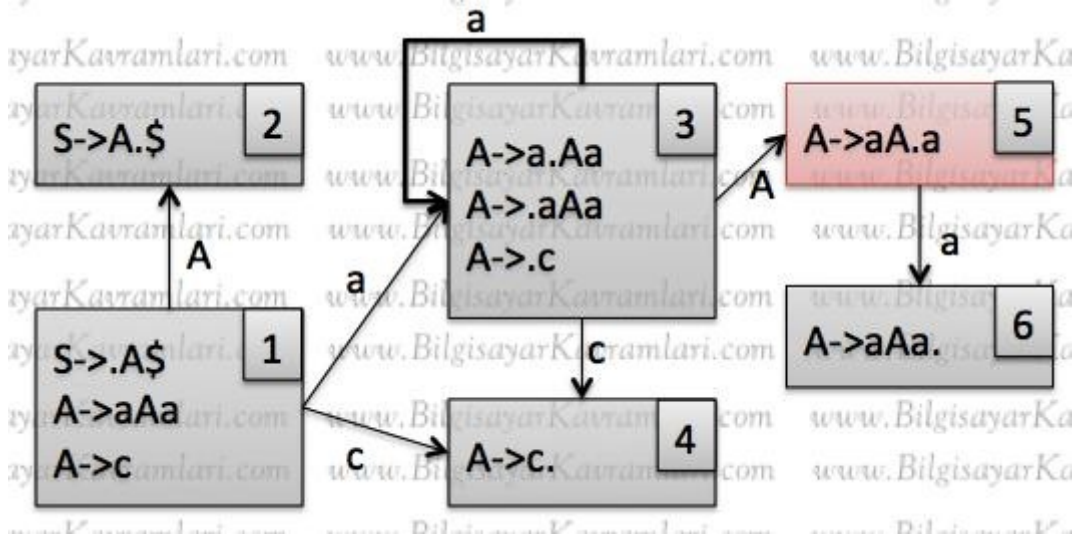
Devam edelim ve sıradaki terim olan “c” için DFA’i işletelim:



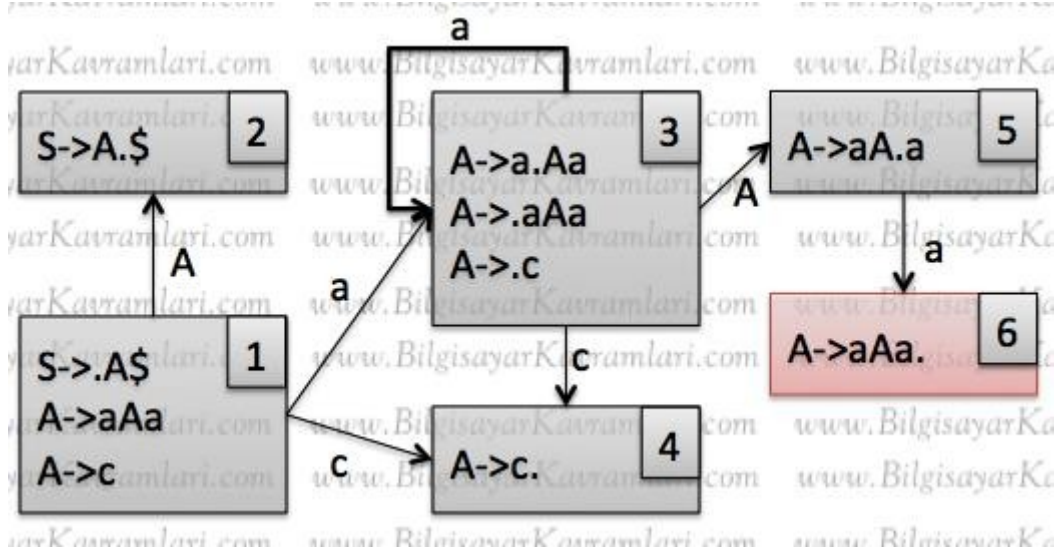
Son haliyle girdimizi “1334.aa” şekline getirebiliriz. Ancak burada dikkat edilecek bir husus, bir ikame (reduce) durumuna erişmiş olmamız. Bunun anlamı, girdimizde bulunan “c” terimi yerine A terimini yazabilecek olmamız. O halde aslında “133.Aaa” şeklinde girdimizi düşünebiliriz. Bu ikame (reduce) bizi bir önceki durum olan 3. duruma geri döndürdü:



Devam edelim ve son kaldığımız 3. durumdan A girdisini işleyelim:

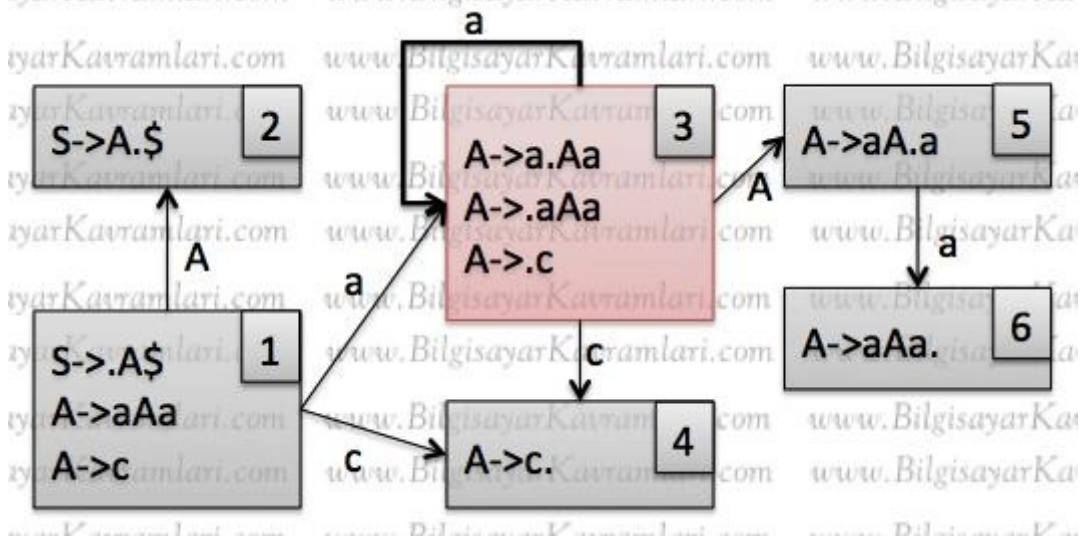


Girdimizin yeni hali, “1335.aa” şeklini almıştır. Devam edip sıradaki a harfini işleyelim:

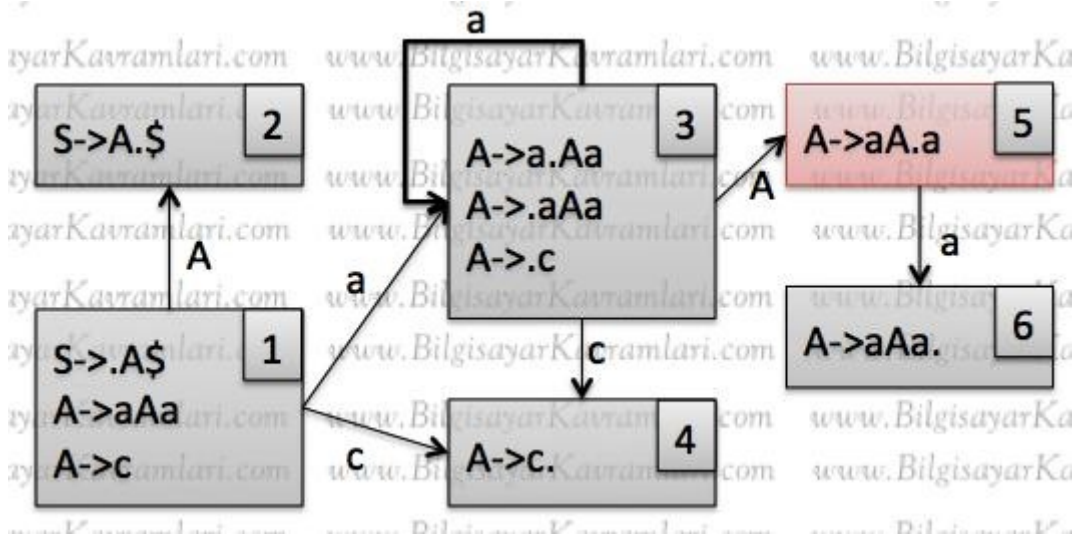


Son haliyle, 6. duruma ulaştık ve dikkat edilirse bu durumda bir ikame (reduce) işlemidir ve bizi A->aAa tanımının yerine A yazmaya zorlamaktadır. Bunun anlamı, “1335.aa” olan girdimizin önce “13346.a” olması ardından da 346 kısmı yerine A yazacağımızdır (3 sayısının a için 5 sayısının A için ve 6 sayısının a için geldiğini düşünebilirsiniz)

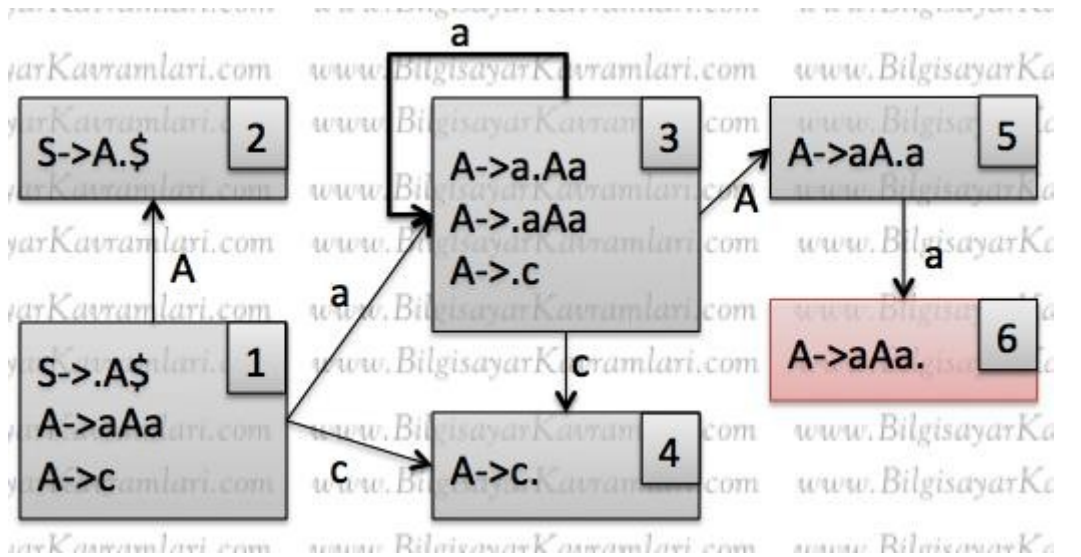
O halde “13.Aa” girdisine dönüşüyoruz ve sıradaki terim için yine 3. durumdan devam ediyoruz.



Benzer şekilde 5. duruma geçiş yapıyoruz:

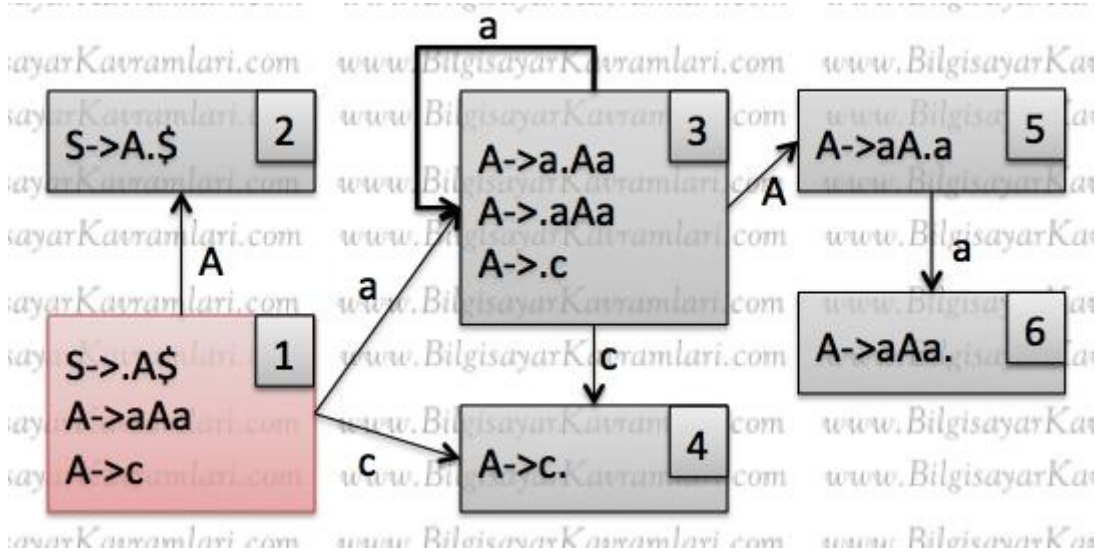


Ve yine bir önceki seferde olduğu gibi 6. duruma devam ediyoruz:



Neticede girdimiz yine aynı “1346” şekline geliyor. Burada yine bir önceki seferde olduğu gibi 346 yerine A koyabiliriz. Ve girdimiz “1A” şekline dönüşür.

Gelelim 1. adıma ve A gelmesi durumuna:

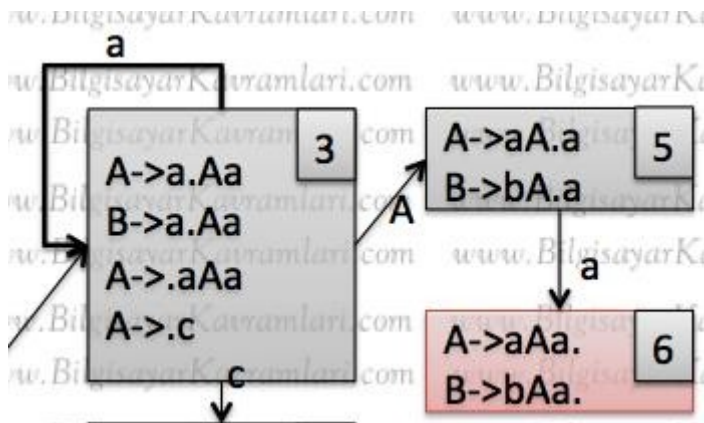


Bu şart, bizi 2. duruma yönlendiriyor ve 2. durumda da girdiyi kabul ediyoruz. Yani bu gramer “aaca” girdisini kabul etmektedir diyebiliriz.

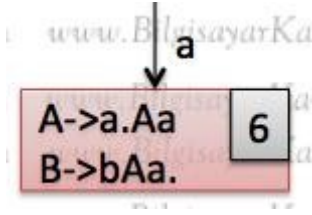
Farklı bir örnek olarak kabul edilmeyen bir girdiyi inceleyelim:

“aaca” olarak bir girdi verilmiş olsaydı, yukarıdaki gibi girdimiz “13.A” olana kadar aynı adımlarla gidecektik. Ancak 3. durumdan sonra A girdisi için devam ettiğimiz 5. durum ve ardından girdide başka bir terim kalmamasından dolayı 5. durumda çakılmamız ve 5. durumun bir kabul durumu olmayışı (tek kabul durumunun 2. durum olduğunu hatırlayın) bizim bu girdiyi kabul etmememiz demek olacaktı.

Gelelim LR(0) algoritmasının yeteneklerinin kısıtlarına. Ne yazık ki LR(0) algoritması, reduce/reduce (ikame/ikame) ve shift /reduce (kaydırma /ikame) çelişkilerine karşı dayanıksızdır. Yani aynı durumda birden fazla ihtimale ikame (reduce) bulunuyorsa veya aynı durumda hem ikame hem kaydırma durumları bulunuyorsa, bu durumda LR(0) algoritmasını kullanamayız.



Örneğin, yukarıdaki kısmi DFA çizimini ele alalım. Burada 6. durumda, iki farklı devamlıdan (non-terminal) birisine geçiş yapılacaktır. Bu devamlı (non-terminal) A da olabilir B de. İşte LR(0) algoritması bu ayrımı yapamadığı için çalışmasında problem olur. Farklı bir durumda shift/reduce (kaydırma/ikame) problemidir. Bunu da aşağıdaki şekilde gösterelim:



Örneğin yukarıdaki haliyle, gelen bir “a” terimi, bir shift (kaydırma) anlamına gelebileceği gibi (yukarıdaki 6. durumun ilk kuralı) bir ikame (reduce) anlamına da gelebilir (yukarıdaki şekildeki 6. durumun ikinci kuralı). Bu problem de bir önceki gibi LR(0) için çözümlenemeyecek bir durumdur ve bu şartları içeren DFA çizilmesi halinde girdiye bakılmaksızın o gramerin LR(0) için uygun olmadığı söylenebilir.

SORU 4: Earley Parçalama Algoritması (Earley Parsing Algorithm)

Bu algoritmanın amacı, verilen bir gramer ile verilen bir girdiyi parçalayarak varsa bu girdideki hatayı bulmak, veya anlamsal olarak bir gösterim elde etmektir.

Algoritmanın çalışması için bir girdi ve bir de dil tanımı yapılması gerekir. Bu yazı kapsamında konunun anlaşılması için örnek üzerinden açıklama yapılacaktır. Konuyu anlamaya aşağıdaki [dil tanımını \(CFG\)](#) ele alarak başlayalım:

$$S \rightarrow E$$

$$E \rightarrow bEb \mid aEa \mid \epsilon$$

Amacımız bu dil için baab şeklinde verilen bir girdiyi parçalamak olsun.

Earley algoritması, öncelikle parçalanacak olan girdinin her elemanının öncesinde ve sonrasında bierer durum analizi yapacak şekilde çalışmaya başlar.

Örneğimizde geçen baab girdisi için aşağıda gösterildiği gibi bir durum tablosu oluşturuyoruz:

b		a	a		b	
@1	@2	@3		@4	@5	

Yukarıdaki tabloda görüldüğü üzere her girdi elemanından önce ve sonrasına gelecek şekilde 5 farklı durumu işaretledik.

Algoritmanın amacı bu işaretlenen adımları doğru bir şekilde doldurabilmek. Başlangıç durumunda, dilimizde bulunan ve başlangıçta (S başlangıç kabul edilirse) erişilebilen bütün kural tanımlarını ilk duruma yazarak algoritmamıza başlıyoruz.

b	a	a	b
@1 S → .E @ 1 E → .bEb @ 1 E → .aEa @ 1 E → . @ 1 S → E. @ 1	@2	@3	@4 @5

Yukarıdaki gösterimden anlaşılacağı üzere kurallarımız arasında yer alan bütün durumlar ulaşılabilir kabul edilmiş ve bu durumların tamamı listelenmiştir. Ayrıca bu durumların tamamı 1. adımda gelen kurallar olduğu için her kuralın altına bu durumu belirten @1 sembolü eklenmiştir.

Algoritmamızın devamında ikinci adıma geçmek için gelen girdi sembolünü işletiyoruz.

Bu aşamada sorumuz, acaba girdimiz olan b sembolü geldiğinde bu kurallardan hangileri ilerler?

Noktaların konumuna dikkat edilirse, sırada b sembolü bekleyen tek kural, $E \rightarrow .bEb$ şeklinde ifade edilen kuraldır.

Bir sonraki adıma bu kuralda bulunan b sembolünü ilerleterek devam ediyoruz:

b	a	a	b
@1 S → . E @ 1 E → .bEb @ 1 E → .aEa @	@2 E → b.Eb @ 1	@3	@4 @5

1		
$E \rightarrow .$		
@ 1		
$S \rightarrow E.$		
@ 1		

Yeni halinde, noktayı bir sembol ilerlettik. Bu durumda noktadan sonra beklenen sembol E devamlısı (non-terminal) olmuş oluyor. O halde dilimizde bulunna E kuralının tanımını bu adıma ekliyoruz.

b	a	a	b
@1	@2	@3	@4
S → . E	E → b.Eb		
@ 1	@ 1		
E → .bEb	E → .bEb		
@ 1	@ 2		
E → .aEa	E → .aEa @		
@ 1	2		
E → .	E → .		
@ 1	@ 2		
S → E.			
@ 1			

Görüldüğü üzere dil tanımımızda bulunan bütün E kuralları ikinci adımda eklenmiştir. Bu kuralların eklendiği adımı ifade etmek için kuralların altına @2 işareti yerleştirilmiştir.

Tam bu noktada dikkat edilecek önemli bir husus, E teriminin boşluk olabileceğidir. O halde E terimini bekleyen bütün sembollerin atlatılması da söz konusudur. Yani hiçbir girdi gelmese bile E terimini ilerleterek bir sonraki terime devam edebiliriz. Bu yüzden listemizin başında bulunan kuralın E harfi ilerletilmiş halini de listenin sonuna ekliyoruz.

b	a	a	b
@1	@2	@3	@4
			@5

S → . E	E → b.Eb		
@ 1	@ 1		
E → .bEb	E → .bEb		
@ 1	@ 2		
E → .aEa	@ E → .aEa	@	
1	2		
E → .	E → .		
@ 1	@ 2		
S → E.	E → bE.b		
@ 1	@ 1		

Bir sonraki adımda gelen a girdisini bekleyen kurallar işletilecektir.

Şu anda a girdisini bekleyen tek kural E->.aEa kuralıdır. Bu kuralı işleterek sonraki adıma devam ediyoruz:

b	a	a	b
@1	@2	@3	@4
S → . E	E → b.Eb	E → a.Ea	@5
@ 1	@ 1	@ 2	
E → .bEb	E → .bEb		
@ 1	@ 2		
E → .aEa	@ E → .aEa	@	
1	2		
E → .	E → .		
@ 1	@ 2		
S → E.	E → bE.b		
@ 1	@ 1		

Yeni durumda, görüldüğü üzere nokta yine bir [devamlı \(non-terminal\)](#) sembolden önceyi göstermektedir. Bu durumda yine bu devamlı sembolün dil tanımında bulunan kuralları eklenmelidir:

b	a	a	b
@1	@2	@3	@4
S → . E	E → b.Eb	E → a.Ea	
@ 1	@ 1	@ 2	
E → . bEb	E → . bEb	E → . bEb	
@ 1	@ 2	@ 3	
E → .aEa	@ E → .aEa	@ E → .aEa	
1	2		
E → .	E → .	@ 3	
@ 1	@ 2	E → .	
S → E.	E → bE.b	@ 3	
@ 1	@ 1		

Eklenen yeni kurallar, eklendikleri adımı göstermek için bu defa @3 sembolü ile işaretlenmiştir.

Daha önce de belirtildiği üzere kurallarımız arasında E'nin boş olabilme ihtimali olduğu için listemize E'nin işletilmiş halini de ekliyoruz:

b	a	a	b
@1	@2	@3	@4
S → . E	E → b.Eb	E → a.Ea	
@ 1	@ 1	@ 2	
E → . bEb	E → . bEb	E → . bEb	
@ 1	@ 2	@ 3	
E → .aEa	@ E → .aEa	@ E → .aEa	
1	2		
E → .	E → .	@ 3	
@ 1	@ 2	E → .	

$S \rightarrow E.$	$E \rightarrow bE.b$	@ 3	
@ 1	@ 1	$E \rightarrow aE.a$	
		@ 2	

Algoritmamızı çalıştırmaya devam ediyoruz ve sıradaki girdi sembolü bir a harfi olduğu için, 3. adımda a harfi bekleyen kuralları ilerletiyoruz. Bu kural sadece $E \rightarrow aE.a$ kuralıdır ve bir a harfi kadar ilerler:

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	
@ 1	@ 1	@ 2	@ 3	
E → .bEb	E → .bEb	E → .bEb	E → .bEb	
@ 1	@ 2	@ 3	@ 4	
E → .aEa	@ E → .aEa	@ E → .aEa	E → .aEa	
1	2	@ 3	@ 4	
E → .	E → .	E → .	E → .	
@ 1	@ 2	@ 3	@ 4	
S → E.	E → bE.b	E → aE.a	E → aEa.	
@ 1	@ 1	@ 2	@ 2	

Yeni durumda, yine beklenen bir devamlı (non-terminal) olduğu için, bu adıma kadar alıştığımız üzere yine terimin açılımını listeye ekliyoruz.

İlave olarak E teriminin boş olabilme ihtimalini de listemizin sonuna ekliyoruz.

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	
@ 1	@ 1	@ 2	@ 3	
E → .bEb	E → .bEb	E → .bEb	E → .bEb	

@ 1	@ 2	@ 3	@ 4
$E \rightarrow .aEa$ 1	@ $E \rightarrow .aEa$ 2	@ $E \rightarrow .aEa$	$E \rightarrow .aEa$
$E \rightarrow .$	$E \rightarrow .$	@ 3 $E \rightarrow .$	@ 4 $E \rightarrow .$
@ 1 $S \rightarrow E.$	@ 2 $E \rightarrow bE.b$	@ 3 $E \rightarrow aE.a$	@ 4 $E \rightarrow aEa.$
@ 1	@ 1	@ 2	@ 2 $E \rightarrow aE.a$
			@ 3

Yukarıda dikkat edilecek bir durum da $E \rightarrow aEa.$ şeklinde sonunda nokta olan bir kurala ulaşmış olmamızdır. Bu tip noktanın sonda olduğu kurallara ikame (reduce) kuralları ismi verilmektedir. Bu kuralın gereği olarak aEa değerinin yerine E değeri konulabilir. Yukarıdaki tabloda görüldüğü üzere ikame edilecek değer (reduce) 2. adımdan gelmektedir. Dolayısıyla 2. adıma geri dönerek buradaki E değerini ilerletiyor ve buraya taşıyoruz:

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	
@ 1	@ 1	@ 2	@ 3	
E → .bEb	E → .bEb	E → .bEb	E → .bEb	
@ 1	@ 2	@ 3	@ 4	
E → .aEa	E → .aEa	E → .aEa	E → .aEa	
@ 1	@ 2	@ 3	@ 4	
E → .	E → .	E → .	E → .	
@ 1	@ 2	@ 3	@ 4	
S → E.	E → bE.b	E → aE.a	E → aEa.	
@ 1	@ 1	@ 2	@ 2	

$E \rightarrow aE.a$

@ 3

$E \rightarrow bE.b$

@ 1

Yukarıdaki 4. adıma eklediğimiz yeni kurala göre 1. adımdan gelen ve bu aşamada tamamlanan devamlı (non-terminal) değerini ilerletmiş olduk (shift).

Sonraki adıma geçebiliriz. Bunun için girdiden bize ulaşan sıradaki terimi yani b harfini okuyarak b harfi bekleyen bütün kurallarımızı ilerletiyoruz (shift).

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	E → b.Eb
@ 1	@ 1	@ 2	@ 3	@ 4
E → .bEb	E → .bEb	E → .bEb	E → .bEb	E → bEb.
@ 1	@ 2	@ 3	@ 4	@ 1
E → .aEa 1	@ E → .aEa 2	@ E → .aEa	E → .aEa	
E → .	E → .	@ 3	@ 4	
@ 1	@ 2	E → .	E → .	
S → E.	E → bE.b	@ 3	@ 4	
@ 1	@ 1	E → aE.a	E → aEa.	
		@ 2	@ 2	
			E → aE.a	
			@ 3	
			E → bE.b	
			@ 1	

Yukarıdaki şekilde 4. adımda b harfi gelmesini bekleyen bütün kurallar ilerletildi (shift) ve oluşan yeni kurallardan ilkinde yine bir devamlı (non-terminal) terim ile karşılaştık. Burada bu terimi açarak E devamlısının kurallarını listemize ekleyebiliriz.

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	E → b.Eb
@ 1	@ 1	@ 2	@ 3	@ 4
E → .bEb	E → .bEb	E → .bEb	E → .bEb	E → bEb.
@ 1	@ 2	@ 3	@ 4	@ 1
E → .aEa 1	@ E → .aEa 2	@ E → .aEa	E → .aEa	E → bEb
E → .	E → .	@ 3	@ 4	@ 5
@ 1	@ 2	E → .	E → .	E → aEa
S → E.	E → bE.b	@ 3	@ 4	@ 5
@ 1	@ 1	E → aE.a	E → aEa.	E → .
		@ 2	@ 2	@ 5
			E → aE.a	
			@ 3	
			E → bE.b	
			@ 1	

Daha önceki adımlara benzer şekilde, E tanımında boşluk olduğu için E terimini atlamış (shift) halini de kural listemize ekliyoruz:

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E→b.Eb	E → a.Ea	E → a.Ea	E→b.Eb
@ 1	@ 1	@ 2	@ 3	@ 4
E→.bEb	E→.bEb	E → .bEb	E → .bEb	E→bEb.
@ 1	@ 2	@ 3	@ 4	@ 1
E→.aEa	@ E→.aEa	@ E → .aEa	E → .aEa	E→.bEb
1	2	@ 3	@ 4	@ 5

$E \rightarrow .$	$E \rightarrow .$	$E \rightarrow .$	$E \rightarrow .$	$E \rightarrow .aEa$
@ 1	@ 2	@ 3	@ 4	@ 5
$S \rightarrow E.$	$E \rightarrow bE.b$	$E \rightarrow aE.a$	$E \rightarrow aEa.$	$E \rightarrow .$
@ 1	@ 1	@ 2	@ 2	@ 5
			$E \rightarrow aE.a$	$E \rightarrow bE.b$
			@ 3	@ 4
			$E \rightarrow bE.b$	
			@ 1	

Şimdi, 4. adımdan taşıdığımız ikinci kurala geçebiliriz. Bu kurala göre bir ikame (reduce) durumu ile karşılaştık çünkü nokta, kuralın sonunda bulunuyor:

$E \rightarrow bEb.$

Kuralın gereği olarak, bEb teriminin yerine E terimi koyarak kuralın geldiği @1. adımda ilerletiyoruz. Birinci adımda E terimini bekleyen tek kuralımız $S \rightarrow .E$ kuralıdır

b	a	a	b	
@1	@2	@3	@4	@5
S → . E	E → b.Eb	E → a.Ea	E → a.Ea	E → b.Eb
@ 1	@ 1	@ 2	@ 3	@ 4
E → .bEb	E → .bEb	E → .bEb	E → .bEb	E → bEb.
@ 1	@ 2	@ 3	@ 4	@ 1
E → .aEa	@ E → .aEa	@ E → .aEa	E → .aEa	E → .bEb
1	2	@ 3	@ 4	@ 5
E → .	E → .	E → .	E → .	E → .aEa
@ 1	@ 2	@ 3	@ 4	@ 5
S → E.	E → bE.b	E → aE.a	E → aEa.	E → .
@ 1	@ 1	@ 2	@ 2	@ 5



$E \rightarrow aE.a$

$E \rightarrow bE.b$

@ 3

@ 4

$E \rightarrow bE.b$

$S \rightarrow E.$

@ 1

@ 1

Görüldüğü üzere 5. adımda ulaştığımız durumlardan birisi E yerine S ikamesi durumudur. O halde aslında bir S [koşulunu](#) tamamladığımızı ve S'nin başlangıç durumu olduğunu hatırlarsak, aslında başlangıç şartını tamamladığımızı ve dolayısıyla da kabul durumuna ulaştığımızı söyleyebiliriz.

Bu duruma kadar olan geçişleri tekrar edecek olursak:

$S \rightarrow E$ @1

$E \rightarrow .bEb$ @2

$E \rightarrow .aEa$ @3

$E \rightarrow .$ @3

$E \rightarrow aE.a$ @3

$E \rightarrow aEa.$ @4

$E \rightarrow bE.b$ @4

$E \rightarrow bEb.$ @5

$S \rightarrow E$ @1

şeklinde adımları tamamlayarak başlangıç durumunu sağlamış oluyoruz. Kısacası bu verilen girdi değeri, bu verilen dil kuralları tarafından tanınmış ve kabul edilebilir.

SORU 5: LL(1) Parçalama Algoritması

LL(1) kelimesi, 3 konunun baş harflerinden oluşur.

- İlk L harfi, Left-to-Right scan, yani soldan sağa doğru tarama yaklaşımından,
- ikinci L harfi, Leftmost Derivation, soldan eksiltme yaklaşımından,
- 1 sayısı ise sadece 1 değer ileri bakılmasından (look ahead token)

Gelmektedir. Bu kavramların aynı anda kullanıldığı parçalama algoritması aşağıda adım adım anlatılacaktır.

Konuyu bir örnek üzerinden anlatmaya çalışalım. Öncelikle ileri bakma tablosunun (look ahead table) nasıl oluşturulduğunu anlatalım.

ÖRNEK 1:

Örneğin parçalama işlemini yapmak istediğimiz gramer aşağıdaki şekilde [CFG](#) yapısında verilmiş olsun:

$$S \rightarrow E\$$$
$$E \rightarrow \text{int}$$
$$E \rightarrow (E \text{ Op } E)$$
$$\text{Op} \rightarrow +$$
$$\text{Op} \rightarrow *$$

LL(1) algoritmasının ilk aşamasında, parçalama tablomuzu (parse table) oluşturuyoruz. Yukarıdaki dilbilgisi (grammar) için oluşturulan tablo aşağıda verilmiştir:

Tabloda, dikkat edileceği üzere dilimizdeki [nihayi elemanlar \(terminal, sonlular\)](#) tablonun kolon başlıklarında ve [devamlılar \(non-terminal\)](#) elemanlar ise satır başlarında yazılmıştır. Bu tabloda, her [devamlı \(non-terminal\)](#) için kabul edilen sonlular gösterilmiş ve hangi kural ile devam edileceği belirlenmiştir.

Bu tip tabloların çizimi sırasında öncelikle basitten başlamak kolaylık sağlar. Dil bilgisi tanımımızda en basit öge, Op olarak gösterilen operatördür ve + veya * sembollerinden birisini alabilir. Bu durumda, tabloda, Op ile + ve *'nın çakıştığı yerlere ilgili kuralları yazmak yeterlidir. Ardından E [devamlısına \(non-terminal\)](#) bakacak olursak, bu devamlı için de iki farklı kural tanımlanmıştır. Ayrıca bu iki kural da birer [sonlu \(terminal\)](#) ile başlamaktadır. Demek ki bizim E devamlısında bulunduğumuz durumdayken bir sonlu gelirse hangi kurala devam edeceğimiz bellidir. Nitekim tabloda da “int” gelmesi durumundaki ve “(“ sembolü gelmesi durumunda tanımlı işlemler yerleştirilmiştir.

Gelelim S ile tanımlı olan kurala. Bu kural $S \rightarrow E$ dönüşümünden dolayı, doğrudan E için tanımlı olan bütün durumları kendisine kopyalamıştır. Çünkü S durumundayken “int” veya “(“ sonluları gelirse kabul edilir ve hangi kuralın uygulanacağı bellidir.

Yukarıdaki tabloda, bazı hücrelerde bilgi varken bazılarında ise hiçbirşey yazmamaktadır. Bunun anlamı, parçalama işlemi sırasında yukarıda geçen durumlardayken, karşılığı boş olan kolondan bir sonlu (terminal) gelmesi halinde hata olduğudur.

Örneğin S devamlısı için sadece “int” ve “(“ sonluları tanımlıdır. Demek ki * veya + ile başlayan bir girdiyi (input) kabul etmemiz hatadır. Hiç sorgulamadan S durumundayken * gelmesi halinde hata verebiliriz.

Benzer şekilde kabul edeceğimiz bir girdi (input) geldiğinde de her adımda, sonraki adımı kesin (deterministic) bir şekilde nereye gideceğimizi biliyoruz.

Yukarıdaki tablonun, aşağıdaki şekilde de yazılması mümkündür:

Yeni tabloda, sadece dilbilgimizdeki her kural için bir sayı kullanılmıştır. Bu tablonun kesin (deterministic) olarak bir girdiyi nasıl işlediğini şimdi anlatmaya çalışalım:

Örneğin girdi olarak “(int + (int * int)) “ dizgisini alalım. Bu dizginin işlenmesi aşağıda verilmiştir:

Şekilde görüldüğü üzere, S durumundan başlayarak bütün adımlar teker teker sonunca kadar tablodan çıkarılarak işlenmiştir. İlk satırda, girdinin ilk sembolü olan (işareti için E geçişi kullanılmış, buradan E durumundayken bir parantez gelmesinde yapılacak tek ihtimal olan 3. geçiş kullanılmıştır. Ardından iki taraftaki (işaretleri birbirini yok etmiş ve E durumundayken gelen int sonlusu için işleme başlanmıştır. Yukarıdaki adımlar bu şekilde devam eder. Yani solda bulunan ve S ile başlayan parçalama işlemi, sağda bulunan girdiyi parçalamak için kullanılmaktadır. Bu sırada soldaki terimlerin en solunda bulunan sonluya karşılık sağdaki girdide aynı sonlu bulunuyorsa bunlar birbirini yok etmektedir. Şayet soldaki listenin en solunda bir [devamlı \(non-terminal\)](#) varsa bu durumda, sağdaki listenin en solunda bulunan sembole göre ilgili açma işlemi tablodan yapılır.

Yukarıdaki tabloda bulunan değerler aslında LL(1) algoritmasının ilk kümesini (First set) de içermektedir. Örneğin S kümesi için kabul edilen ilk değerler

$İlk(S) = \{ '(', 'int' \}$

olarak yukarıdaki tabloda gösterilmektedir. Bunun anlamı S durumundayken sadece bu kümedeki elemanların kabul edildiği ve bu elemanlar dışında bir eleman gelmesi durumunda hata verileceğidir.

ÖRNEK 2

Yukarıdaki örneğe benzer şekilde daha gerçekçi bir örneğin tablosunu oluşturmaya çalışalım.

Yukarıdaki örnekte verilen dilbilgisinin tablosunu oluşturalım. Tablonun satır başlıklarına, dilbilgisinde bulunan [devamlıları \(non-terminal\)](#) ve kolon başlıklarına da dilbilgisinde bulunan [sonluları \(terminal\)](#) yerleştirdik ve yukarıdaki boş tablonun nasıl doldurulduğunu adım adım işleyelim.

Şimdiki adımda tablodaki her hücreye gelecek değeri bulmamız gerekiyor. Basitten başlayarak karmaşığa doğru tabloyu dolduracağız. En basit tanım, TERM için yapılmış ve iki adet terminal ile tanımlanmıştır. Bu durumda tablonun ilgili hücreleri aşağıdaki şekilde

doldurulur:

Ardından ikinci karışıklık derecesine sahip EXPR teriminin tanımını tabloda işaretleyelim. EXPR teriminin tanımına bakıldığında 5 farklı ihtimal bulunmaktadır. Bunlardan 4 tanesi zaten sonlu (terminal) ile tanımlıdır. Bunları doğrudan işaretleyebiliriz:

Ayrıca ilave bir satırda EXPR devamılsının tanımının TERM devamlısı ile yapıldığı belirtilmiş. Bu tanıma göre EXPR durumundayken, TERM durumundan bir terim ile devam etmemiz söz konusu olabilir. Bu durumda TERM için geçerli olan bilgiler, EXPR için de kopyalanmalıdır. Yani TERM satırını odluğu gibi EXPR satırına kopyalıyoruz:

Yeni halinde tablomuza bakıldığında EXPR durumundayken gelebilecek sonlu terimler aşağıdaki küme ile ifade edilebilir:

İLK (EXPR) : { zero?, not, ++, — , id, const}

Son olarak STMT devamlısını tablomuzda işaretleyelim. STMT tanımının iki satırında sonlu ile başlamaktadır. Bu durumları aşağıdaki şekilde tabloya işaretleyelim.

STMT tanımının bir satırında ise EXPR tanımı bulunmaktadır. Bu durumda EXPR satırının aynısının STMT için de kopyalanması gerekir:

Yukarıdaki tablonun son halinde, örneğin bir STMT tanımının id veya const kabul edeceği görülmektedir. Ancak “;” veya “do” sonlularının STMT tanımında bulunması bir hata olarak kabul edilebilir.

Tanımda epsilon bulunması durumu

[CFG](#) tanımında bulunan özel bir durum da, tanımın herhangi birisinde epsilon sembolünün (ϵ , bazı kaynaklarda lambda (λ) olarak da geçer) bulunması durumudur. Bu durum özel olarak ele alınmalıdır çünkü bahsi geçen terimin yokluğu söz konusudur. Bu durumda uygulanabilecek en kolay yaklaşım epsilon terimi yokmuş gibi davranmak ve herşeyi işledikten sonra epsilon durumunu özel olarak ele almaktır.

ÖRNEK 3:

Epsilon durumunu içeren aşağıdaki örneği ele alalım:

Yukarıdaki bu durum için tablomuzu oluşturuyoruz:

	0-9	+	-
Number			
Sign			
Digits			
Digit			

Yukarıdaki tabloda dikkat edilirse epsilon ihtimali gösterilmemektedir. Zaten böyle bir ihtimal gösterilemez çünkü epsilon durumu yokluğu temsil eder. Tablomuzu sanki epsilon hiç yokmuş gibi dolduruyoruz:

	0-9	+	-
Number			
Sign		Sign-> +	Sign -> -

Digits			
Digit	Digit -> 0 1... 9		

Yukarıdaki hali ile tabloda [sonlu durumlar \(terminal\)](#) işaretlenmiştir. Sıra [devamlı \(non-terminal\)](#) durumları işaretlemeye geldi:

	0-9	+	-
Number		Number->Sign Digits	Number -> Sign Digits
Sign		Sign-> +	Sign -> -
Digits	Digits -> Digit Digits -> Digit Digits		
Digit	Digit -> 0 1... 9		

Yukarıdaki haliyle bütün durumlar işaretlendi. Şimdi epsilon'nu çalıştırabiliriz. Buna göre bir Sign devamlısının (non-terminal) epsilon yani boşluk olabilme ihtimali bulunmaktadır. Bu durumda Sign ile başlayan bir tanımda Sign terimi yokmuş gibi davranılabilir.

Number->Sign Digits

Satırını, normalde Sign ile başlıyor diye kabul etmiş ve Sign ile başlayan bütün terimleri buraya yerleştirmiştik. Epsilon durumu Sign terimi olmaması halini de düşünmemizi gerektirir çünkü Sign terimi boş olup Number devamlısı, Digits devamlısı ile başlayabilir. Bu durumda Digits satırı aynen Number satırına kopyalanmalıdır.

	0-9	+	-
Number	Digits -> Digit Digits -> Digit Digits	Number->Sign Digits	Number -> Sign Digits
Sign		Sign-> +	Sign -> -
Digits	Digits -> Digit Digits -> Digit Digits		
Digit	Digit -> 0 1... 9		

Yukarıdaki tabloda, son durum görülmektedir. Yukarıdaki tabloya göre Number terimi herhangi bir sonlu ile başlayabilir.

Yukarıdaki tabloda özel bir durum, verilen [CFG'nin](#) LL(1) olarak yazılmaya uygun olmamasıdır. Bunun sebebi bir hücrede birden fazla tanımın bulunmasıdır.

Takip Kümeleri (Follow Sets)

LL(1) algoritmasının önemli adımlarından birisi de takip kümelerini oluşturmaktır. Bu yazıda, buraya kadar İlk kümelerinin (First set) nasıl oluşturulduğunu ve nasıl kullanıldığını anlattık.

Yani bir durumdayken gelen ilk girdi (input) bilgisine göre hangi adıma geçeceğimiz veya hata verip vermeyeceğimizi nasıl anlayacağımızı gördük. Bu noktadan sonra Takip Kümelerini (Follow Set) göreceğiz ve dildeki herhangi bir durumdan sonra hangi sonluların (terminal) gelebileceğini takip edeceğiz.

Takip kümeleri, basitçe dildeki herhangi bir bilgiden sonra hangi bilginin gelebileceğidir. Örneğin daha önce açıklanan örnek 2’de while kelimesinden sonra do kelimesi gelebilir mi gelemmez mi? Sorusuna ancak takip kümesi ile cevap verilebilir.

Takip kümeleri nasıl oluşturulur sorusunu yine bir örnek ile açıklayalım:

Örnek 4:

$A \rightarrow Ax \mid yB$

$B \rightarrow zA \mid Cz$

$C \rightarrow y \mid z$

Yukarıda verilen dil tanımı için İlk (First) ve Takip (follow) kümelerini oluşturmaya çalışalım. Öncelikle ilk kümeleri aşağıdaki şekilde oluşturulabilir:

Basitten başlarsak, $C \rightarrow y$ ve $C \rightarrow z$ zaten verilmiş:

	x	y	z
A			
B			
C		$C \rightarrow y$	$C \rightarrow z$

İkinci adımda B için tablonun ilgili satırını dolduralım:

	x	y	z
A			
B			$B \rightarrow z$
C		$C \rightarrow y$	$C \rightarrow z$

B için tanımlı olan kurallardan birisi tabloya yazılmıştır. Ancak ikinci tanımda $B \rightarrow Cz$ şeklinde ilk eleman bir devamlı (non-terminal) olduğu için C satırı aynen kopyalanır:

	x	y	z
A			
B		$B \rightarrow Cz$	$B \rightarrow zB \rightarrow Cz$

C		C->y	C->z
---	--	------	------

Ardından A ile ilgili satıra geçelim:

	x	y	z
A		A->yB	
B		B->Cz	B->zB->Cz
C		C->y	C->z

Sonlu (terminal) ile başlayan kuralı tablomuza yerleştirdikten sonra [devamlı \(non-terminal\)](#) ile başlayan durumu inceleyelim:

A->Ax olduğu için burada hem sol döngü (left recursion) problemi bulunmakta hem de A ile başlayan bütün durumları ilk (first) kümesine almamız gerekmektedir. A zaten kendi tanımı olduğu için yeni bir eleman alınması söz konusu değildir. Yani tablomuz y elemanına iki durumdan erişebilir ve ikisi de aşağıdaki şekilde gösterilmiştir:

	x	y	z
A		A->yBA->Ax	
B		B->Cz	B->zB->Cz
C		C->y	C->z

Tabloya hızlıca bakınca, x ile başlama durumu hiçbir devamlı (non-terminal) için söz konusu değildir.

Ayrıca tablodaki bir hücrede birden fazla kural tanımı bulunduğu için (iki ayrı hücrede) bu dilbilgisinin (grammer) LL(1) algoritması ile parçalanması mümkün değildir.

Gelelim takip (follow) kümelerinin oluşturulmasına. Bu kümelerin oluşturulması sırasında izlenecek en kolay yol iki aşamada dili incelemektir. Birinci adımda, dil tanımındaki eşitliklerin sağ tarafına bakılır. Buna göre herhangi bir terimden sonra gelen terimler, takip (follow) kümesine alınır.

1. Adım

Dilbilgisi tanımının sağ tarafını alıntılacak olursak:

Ax | yB

zA | Cz

y | z

Yukarıdaki tanımlardan yola çıkarsak aşağıdaki kümeleri oluşturmamız mümkündür:

Takip (x) : { \$ }

Bunun sebebi dilin herhangi bir yerinde x'ten sonra bir sembol gelmemesidir.

Takip (y) : { \$, B }

takip kümelerinde, devamlı (non-terminal) eleman bulunmaması gerekir. Bu durumda yukarıdaki kümede bulunan B terimini açmamız gerekiyor. Herhangi bir şekilde devamlı eleman kümeye eklenmişse aşağıdaki dönüşümü yapabiliriz:

Takip (y) : { \$, İlk(B) }

Yani bir takip kümesinde bir devamlı (non-terminal) bulunması durumunda bu elemanın ilk kümesini yerine yazmamız mümkündür. İlk (B) ise tablodan çıkarılabileceği üzere {y,z} olarak yazılabilir:

Takip (y) : { \$, y, z } olarak yazılması mümkündür.

Takip (z) : { \$, A } yine devamlı (non-terminal) olduğu için bu elemanı da açıyoruz

Takip (z) : { \$, y } , A devamlısının ilk kümesi yerine yazılmıştır.

Takip (A) : { x, \$ }

Takip (B) : { \$ }

Takip (C) : { z }

2. Adım

Bir önceki adımda atlanan bir nokta, herhangi bir terimin tanımında bulunan elemanın son eleman olması durumunda, tanımı yapılan elemanın sonrasına gelen terimlerin aslında bu son elemanın da sonrasına gelmesi durumudur.

Örneğin dil tanımımızı hatırlayalım:

$A \rightarrow Ax \mid yB$

$B \rightarrow zA \mid Cz$

$C \rightarrow y \mid z$

Buna göre $C \rightarrow z$ durumu için, C'nin tanımının son elemanı olan z teriminden sonra gelen elemanlar aslında C'den sonra gelen elemanları da içermelidir. Çünkü C'den sonra gelen herhangi bir eleman z'nin sonrasına eklenebilir. Benzer durum $B \rightarrow Cz$ tanımı için de geçerlidir. Burada da z sondaki eleman olduğu için B'nin sonrasına gelen her eleman aslında z'nin de sonrasına gelebilir.

İşte ikinci adımda bu durumaları kümelere ekliyoruz.

Yine en alttan başlayarak yukarı doğru çıkalım:

Takip (x) : { \$ } şeklinde bulmuştuk. Bu elemanın (yani x'in) sonda olduğu bir tanım var mı?

Evet var $A \rightarrow Ax$ tanımı bu şartı sağlıyor. Öyleyse kümemizi aşağıdaki şekilde güncelliyoruz:

Takip(x) : { \$, Takip(A) } , bunun sebebi x teriminin A'nın tanımının sonunda bulunması ve A'yı takip eden bir elemanın x'i takip edebilecek olmasıdır.

Takip (y) : { \$, y, z } şeklinde bulmuştuk. Bu elemanın (yani y'nin) sonda olduğu bir tanım var mı?

Evet var $C \rightarrow y$ bu şartı sağlıyor. O halde kümemizi aşağıdaki şekilde güncelliyoruz:

Takip (y) : { \$, y, z, Takip(C) }

Takip (z) : { \$, y } şeklinde bulmuştuk. Bu elemanın (yani y'nin) sonda olduğu bir tanım var mı?

Evet var $C \rightarrow z$ bu şartı sağlıyor. O halde kümemizi aşağıdaki şekilde güncelliyoruz:

Takip (z) : { \$, y, Takip(C) }

Takip (C) : { z } şeklinde bulmuştuk. Bu elemanın (yani C'nin) sonda olduğu bir tanım var mı?

Hayır yok. Bu durumda Takip (C) yukarıdaki şekilde bulunmuştur. Bu takip kümesinin geçtiği önceki örnekleri son haline erdirtirebiliriz.

Takip (y) : { \$, y, z, z } ancak bir kümede bir eleman bir kere geçebileceği için

Takip (y) : { \$, y, z } olarak bulunur

Takip (z) : { \$, y, z } olarak bulunur.

Takip (B) : { \$ } şeklinde bulmuştuk. Bu elemanın (yani B'nin) sonda olduğu bir tanım var mı?

Evet var, $A \rightarrow yB$ bu şartı sağlayan bir kural. Öyleyse takip kümemiz aşağıdaki şekilde olur:

Takip (B) : { \$, Takip(A) }

Takip (A) : { x, \$ } şeklinde bulmuştuk. Bu elemanın (yani A'nın) sonda olduğu bir tanım var mı?

Evet var, $B \rightarrow zA$ bu şartı sağlayan bir kural. Öyleyse takip kümemiz aşağıdaki şekilde olur:

Takip (A) : { x, \$, Takip(B) }

Şimdiye kadar bulduğumuz kümeleri toplu olarak yazarsak:

Takip(x) : { \$, Takip(A) }

Takip (y) : { \$, y, z }

Takip (z) : { \$, y, z }

Takip (C) : { z }

Takip (B) : { \$, Takip(A) }

Takip (A) : { x, \$, Takip(B) }

şeklinde bir listeye erişiriz. Burada dikkat edilirse, Takip(A) ile Takip(B) arasında birbirini içeren (recursive) bir tanım bulunmaktadır. Diğer bir deyişle, A'dan sonra gelen bütün elemanlar B'den sonra ve B'den sonra gelen bütün elemanlar da A'dan sonra gelebilmektedir. Bu durumda bu iki küme birleştirilir.

Takip (B) : { \$, x }

Takip (A) : { x, \$ }

şeklinde iki küme de nihayete erdirilmiş olur.

Takip(x) : { \$, Takip(A) } ise aşağıdaki şekilde güncellenir:

Takip(x) : { \$, x }

Netice olarak özellikle seçilmiş bu zor örnekteki bütün takip kümeleri bulunmuştur.

Son bir örnek üzerinden dilimizde epsilon bulunması durumunda nasıl işleyeceğimizi anlatmaya çalışalım.

Örnek 5:

Aşağıdaki şekilde verilmiş grameri ele alalım:

$$A \rightarrow BxC \mid Cy$$

$$B \rightarrow zC$$

$$C \rightarrow xA \mid \epsilon$$

Buna göre tablomuzu ve takip kümelerini (follow set) çıkarmaya ve dilin LL(1) olup olmadığına karar vermeye çalışalım. Bu dilin özelliği, epsilon elemanını içeriyor olmasıdır bu yüzden dilimizde sanki epsilon yokmuş gibi çalışıp ardından epsilon durumunu özel olarak ele alacağız.

Öncelikle sonlulara olan geçişleri işaretleyelim:

	x	y	z
A			
B			$B \rightarrow zC$
C	$C \rightarrow xA$		

Şimdi devamlılara (non-terminal) olan geçişleri işaretleyelim:

	x	y	z
A	$A \rightarrow Cy$		$A \rightarrow BxC$
B			$B \rightarrow zC$
C	$C \rightarrow xA$		

Epsilonu dilimizden çıkarmamız halinde, parçalama tablomuz yukarıdaki halini alır. Epsilonu tablomuz eklersek (daha önce nasıl olduğunu Örnek 3'te gördük) aşağıdaki tabloyu elde ederiz:

	x	y	z
A	$A \rightarrow Cy$	$A \rightarrow Cy$	$A \rightarrow BxC$
B			$B \rightarrow zC$
C	$C \rightarrow xA$		

Bu aşamadan sonra geriye takip kümelerini oluşturmak kalıyor (follow set) ve yine sistemde epsilon yokmuş gibi işlem yapıyoruz (adımlar örnek 4'te detaylı olarak anlatılmıştır).

Takip(x) : { ilk(A) }

Takip (x) : {x,y,z}

Takip (y) : { \$, Takip (A) }

Takip (z) : { ilk(C) }

Takip (z) : { x }

Takip (A) : { \$, Takip (C) }

Takip (B) : { x }

Takip (C) : {y , \$, Takip (B) , Takip (A) }

Şayet epsilon durumunu değerlendirirsek, C değeri yok olabilir. Bu durum, aşağıdaki kurallar için sorun oluşturur:

$A \rightarrow BxC$

$B \rightarrow zC$

bunun sebebi C teriminin sonda bulunması ve aslında son terimin kaldırılması halinde

$A \rightarrow Bx$

$B \rightarrow z$

şeklinde kuralların çalışabileceğidir. Bu durumda Takip(x) kümesine Takip(A) ve Takip(B) kümesine Takip(z) eklenmelidir. Son durum aşağıdaki şekildedir:

Takip (x) : {x,y,z, **Takip(A)**}

Takip (y) : { \$, Takip (A) }

Takip (z) : { x }

Takip (A) : { \$, Takip (C) }

Takip (B) : { x , **Takip(z)** }

Takip (C) : {y , \$, Takip (B) , Takip (A) }

Şimdi takip kümelerini açarak yerine yazarsak:

Takip (x) : {x ,y ,z , \$ }

Takip (y) : { \$, x, y }

Takip (z) : { x }

Takip (A) : { \$, x, y }

Takip (B) : { x }

Takip (C) : {y , \$, x }

şeklinde son haliyle takip kümelerini (follow set) elde etmiş oluruz.

Son söz olarak, LL(1) parçalama algoritmasının, verilen bir girdi için, hangi terimden sonra hangi terimlerin gelmesi gerektiğini takip etmede hız sağladığı kesindir. Yazı zaten çok uzadığı için (şu anda 10. sayfadayım) LL(1) kullanılmaması durumunda aynı parçalama işleminin ne kadar uzun sürdüğünü ve hatta hata yaptığını göstermiyorum. Ancak okuyucu bunu kendisi de deneyebilir. Örneğin [geri izleme algoritması \(back tracking algorithm\)](#) kullanarak verilen bir [CFG \(içerikten bağımsız dil\)](#) ile bir dizgiyi parçalamayı deneyebilir.

SORU 6: Maximal Munch (Azami Lokma) Yöntemi

Algoritma, bilgisayar programlama veya bilgisayar bilimlerinde genelde derleyici tasarımı (compiler design) gibi dizgi (string) işleminin yoğun olduğu alanlarda büyük lokma (maximal munch) veya en uzun eşleşme (longest match) olarak geçmektedir. Buradaki amaç, anlık olarak en büyük lokmayı oluşturmak ve yutmak veya aranan koşulları sağlayan en uzun yapıyı koparmak olarak düşünülebilir. Tam olarak aynı olmasa da [aç gözlü yaklaşımına \(greedy approach\)](#) yakın bir yapıdır.

Örneğin [sözcük analizi sırasında \(lexical analysis\)](#), programlama dilinde gelen yazıların anlamlı kelimelere bölünmesi gerekmektedir. Diyelim ki aşağıdaki şekilde verilmiş bir satır olsun:

```
int a=b+c;
```

Bu satırı insan olarak biz okuduğumuz zaman, bu satırın içeriğinin a isimli bir değişkene b ve c değişkenlerinin toplamını koymak amacıyla olduğunu anlıyoruz. Ancak bilgisayar açısından henüz işlenmemiş olan bu [dizgi \(string\)](#) aslında bir boşlukla ayrılmış ve birisi 3 diğeri 6 karakter uzunluğunda iki kelimeden oluşan bir satırdır. Buradaki sembollerin taşıyabileceği çok sayıda anlam bulunduğundan [derleyici \(compiler\)](#) bu kelimeleri sözcük analiziden geçirmek zorundadır.

İlk harfi ele alalım ve soldan sağa işlemeye çalışalım. İlk kelimenin ilk harfi i harfi ve bu harf C dilinde farklı anlamdaki komutların başlangıcına işaret ediyor olabilir. Örneğin if, int gibi komutlar olabileceği gibi bir değişken veya fonksiyon ismi de olabilir. Demek ki ilk harften

sonra bir anlam çıkarmaya çalışıyoruz ancak ihtimalleri azaltarak sonuna kadar gitmek zorundayız. Ardından gelen n harfi ihtimalleri azaltıyor nihayetinde gelen t harfi ile kelime tamam oluyor. Böylelikle bir kelimeyi analiz etmiş oluyoruz. Benzer durum ikinci kelime için de geçerli ancak ikinci kelime 3 değişken ve 3 sembolden oluşmakta ve kelime bunlara teker teker bölünecek.

Algoritmanın çalışmasını daha iyi anlayabilmek için, sözcük analizinde (lexical analysis) algoritmanın nasıl kullanıldığını anlatalım.

Örneğin bize sözcük analizinde kullanılmak üzere düzenli ifadeler(regular expressions) verilmiş olsun. Bunları kullanarak sözcükleri analiz etmemiz istiyor ki, LEX (flex) gibi diller için de durum tam olarak bundan ibarettir.

Yaklaşımımız öncelikle [düzenli ifadeleri \(regular expressions\)](#), birer [sonlu durum otomatu \(finite state automaton\)](#) şekline çevirmek ve ardından gelen girdi cümlesini bu otomatlarda en fazla yolu gidecek şekilde parçalamak olacaktır.

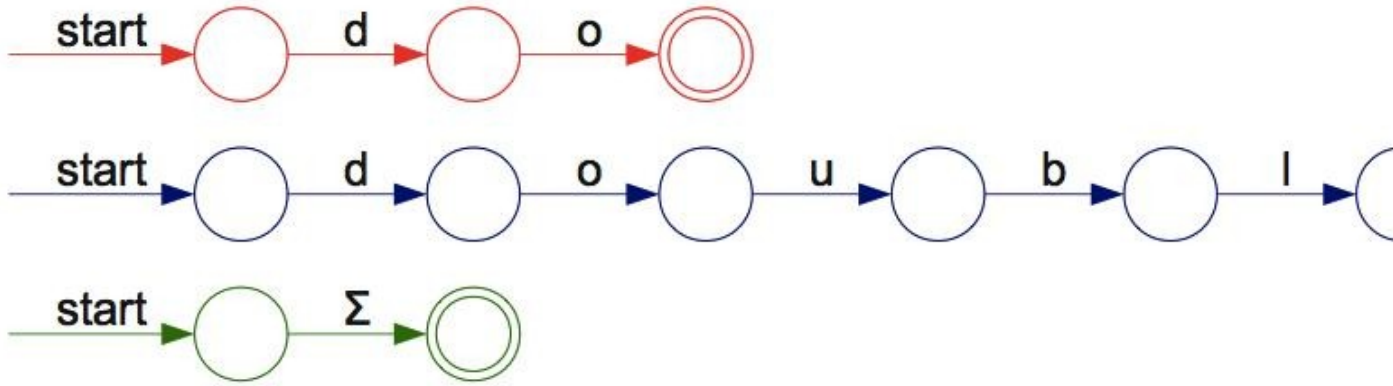
Örneğin kullandığımız 3 adet düzenli ifade aşağıdaki şekilde verilsin:

T_Do do

T_Double double

T_Mystery [A-Za-z]

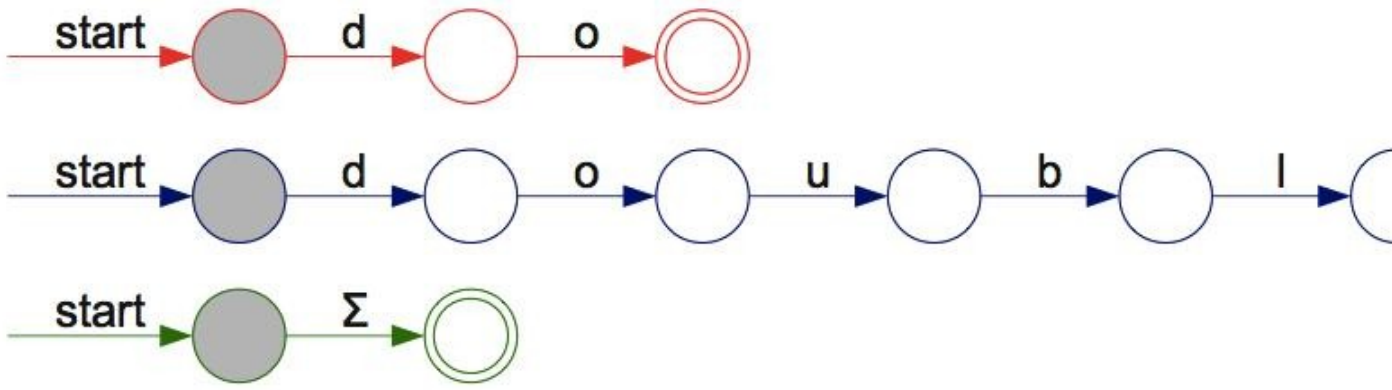
Buna göre bir kelime, do, double veya tek harfli herhangi başka bir kelime olma ihtimaline sahiptir.



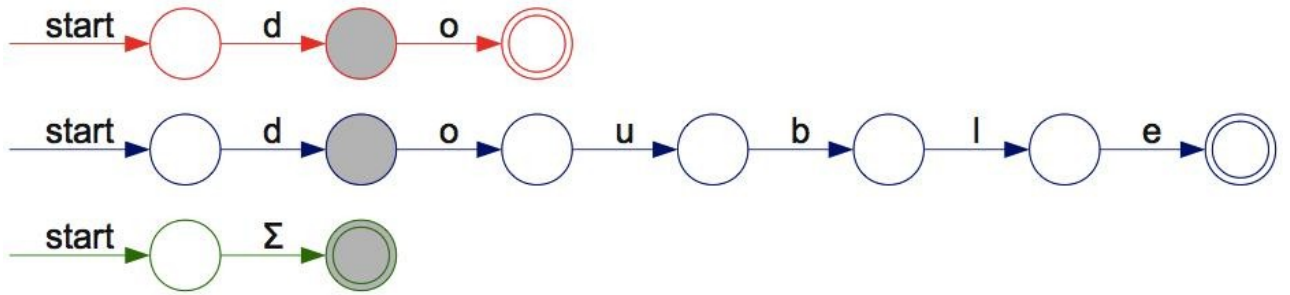
Yukarıdaki şekilde, 3 farklı ihtimali de içeren ayrı birer sonlu durum makinesi çizilmiştir. Bu makineleri kullanarak aşağıdaki örnek cümleyi analiz edelim:

“DOUBDOUBLE”

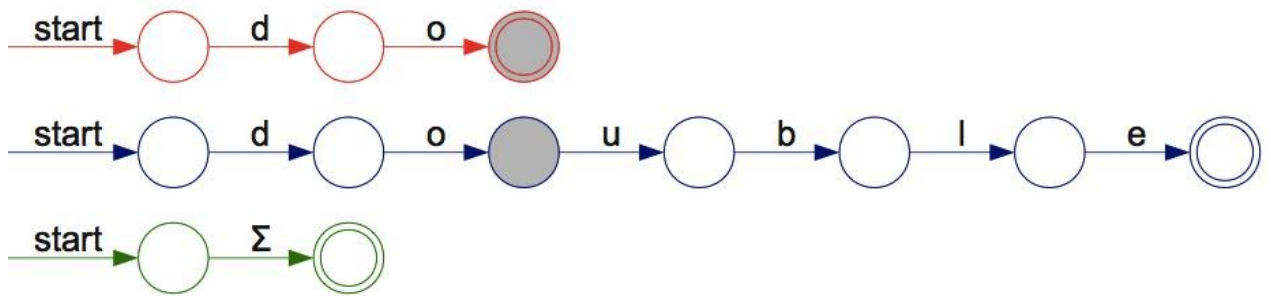
Bu kelimeyi yukarıdaki kurallara göre parçalamak istiyoruz:



Öncelikle, bütün makinelerimizde başlangıç durumuna gidiyoruz ve ardından gelen ilk harfe göre makinelerimiz çalışmaya başlıyor:

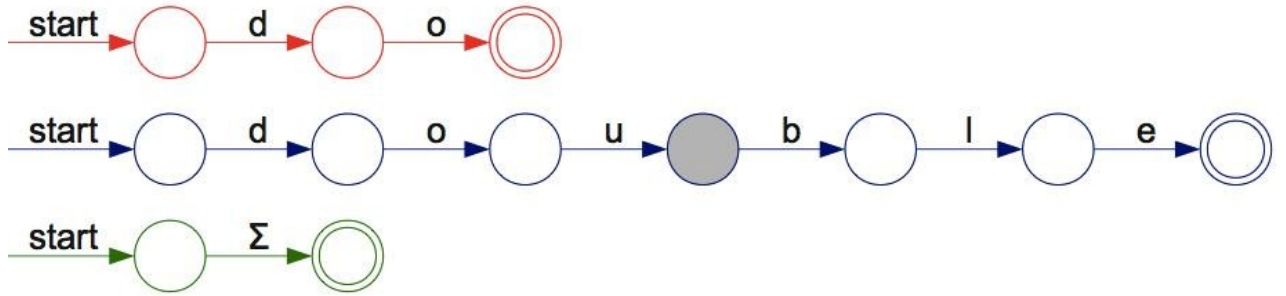


Yukarıda görüldüğü üzere, ilk harf olan d harfi bütün makineler için kabul edilir bir harftir. Bu durumda bütün makinelerdeki ilk geçiş, geçilmiş ve başlangıç durumundan, bir durum ileri gidilmiştir. Burada dikkat edilecek diğer bir nokta, makinelerimizden birisinin kabul durumuna (final state) gelmiş olmasıdır. Bu durumda son makinemiz, bize bu harfi yani “d” harfini kabul ettiğini söylüyor. Öyleyse biz d harfine bir gösterici koyarak bu harfi kabul ettiğimizi söyleyebiliriz ve bu gösterici bir sonraki güncellemeye kadar bekliyor. Şimdilik lokmamızda sadece “d” harfi var.

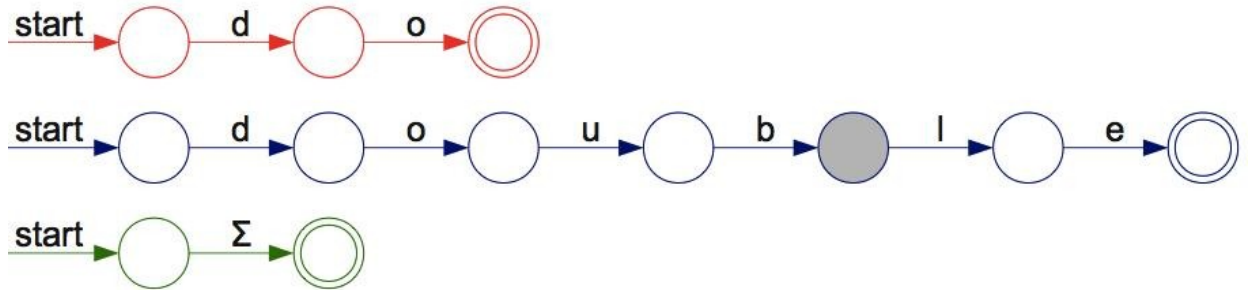


İkinci harf olan “o” harfi işlendiğinde ise sadece ilk iki makinede ilerleme sağlanmıştır. Son sırada yer alan ve 1 harf uzunluğundaki kelimeleri parçalamaya yarayan makine, kelime uzunluğu 1’den fazla olduğu için ilerleyemeden kalır. Tam bu noktada büyük lokma (maximal munch) algoritmasının anlamı ortaya çıkar. Buna göre artık son makineden bir sonuç çıkma ihtimali yoktur çünkü ilk iki makine her durumda, son makineye göre daha büyük bir lokma işleyecektir.

Ayrıca ilk makinemiz son durumuna (final state) ulaştığı için bir önceki adımda, lokmamızda olan “d” harfini güncelliyoruz, çünkü iki harfli daha uzun bir lokmayı kabul eden bir durum ile karşılaştık ve yeni lokmamız “do” oluyor.



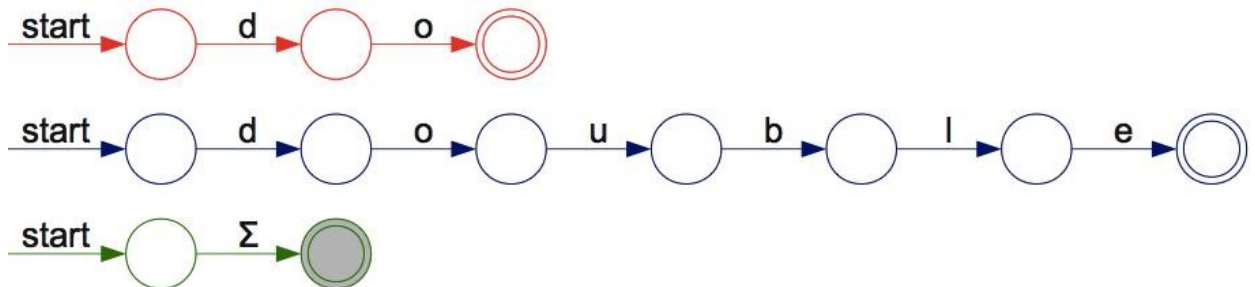
Ardından gelen u harfi için işleme devam etmektedir. Tek alternatif olan ikinci makinemizi çalıştırıyoruz. Şu ana kadar bitiş durumuna eriştiğimiz en uzun lokmamız “do” kelimesi.



Girdi olarak aldığımız kelimeden bir harf daha ilerleyerek b harfine kadar işlemiş oluyoruz. Girdi kelimemizin “DOUBDOUBLE” olduğunu hatırlarsak, şu anda 4. harfe kadar gelmişiz demektir. Ayrıca en büyük lokmamız henüz “do” kelimesi.

Bu aşamadan sonra ne yazık ki hiçbir makinemiz ilerleyemeyecek. Bunun sebebi şu ana kadar işlediğimiz kelimenin “DOUBD” olması ve bu kelimeyi kabul eden makinemizin bulunmaması.

Elimizde şimdiye kadar işlediğimiz en büyük lokma olan “do” kelimesini, ilk çıktı olarak basıyoruz. Ardından bu kelimenin devamı olan girdimizdeki 3. harfe yani u harfine geri dönerek işlemeye devam ediyoruz.

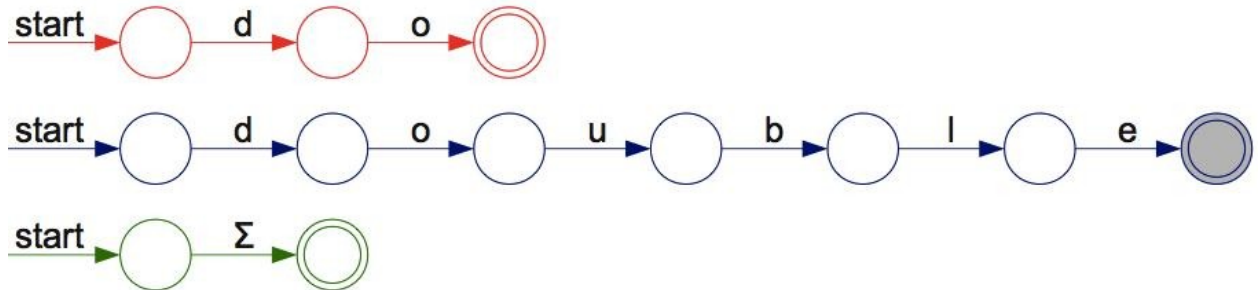


Kelimenin yeni başlangıç değeri için U kabul edilecek ve bu harf sadece 3. makineye uygun bulunacak. Bu durumda ikinci kelimemiz U olmuş oluyor. Benzer durum B için de geçerli ve üçüncü kelime olarak B harfini çıkarıyoruz:

DO+U+B

şeklinde şimdiye kadar parçalama işlemini tamamladık. Girdimiz “DOUBDOUBLE” kelimesiydi, bu kelimenin ilk 4 harfi böylece işlenmiş oldu gelelim diğer harflere.

Diğer harfler yeniden makineye girecek ve en büyük lokma oluşana kadar işlenecek.



Kısacası, geriye kalan harflerin 3 makine için işlenmesi sonucunda ikinci makine tarafından kabul edilmesi söz konusudur. Bu durumda parçalama işlemi aşağıdaki şekilde sonuçlanacaktır:

DO+U+B+DOUBLE

görüldüğü üzere en büyük lokma (maximal munch) algoritmasını basit bir girdi ve 3 makine üzerinde çalıştırdık ve bize, bu makineler tarafından üretilebilecek en uzun alternatifleri döndürdü.

SORU 7: Lisan-ı Kaime (Dillerde Dik Açılı, Orthogonal Languages)

Özellikle bilgisayar programlama dillerinde kullanılan bir terimdir. Buna göre dillerde bulunan özelliklerin birbirine dik olması, yani birbirinden bağımsız olması kast edilir. Kelime olarak, yöneylerin (vectors) dikliğinden esinlenilmiştir. Vektörlerin birbirine dik olması, aralarında ilişki bulunmaması (birbirine iz düşümünün sıfır olması) anlamına gelmektedir. Şayet bir dilde bulunan özellikler arasında da ilişki bulunmuyorsa, dilin özelliklerinin genişlemiş olduğu düşünülebilir.

Örneğin değişken tipleri olarak tamsayı (integer) ve küsurlu sayının (float) bulunduğu bir dil ve işlem(operator) olarak dizi (array) ve işaretçi (pointer) bulunduğu bir dili ele alalım. Bu dildeki 2 değişken tipi ve iki işlem şekli, şayet kaime arz ediyorsa, bu durumda bir işaretçi, tamsayı veya küsurlu sayıyı gösterebilmektedir. Benzer şekilde dizinin elemanları işaretçi olabilmekte ve işaretçi de diziye işaret edebilmektedir.

Şayet dilde, kaime bulunmasaydı, örneğin dizi işlemi, sadece küsurlu sayılar üzerinde çalışıyor olsaydı, bu durumda dildeki esneklik büyük ölçüde kaybedilmiş olacak ve örneğin tam sayı dizisi bulunmayacağı gibi, tam sayı dizisine işaret eden bir işaretçi veya her elemanı işaretçi olan bir tamsayı dizisi de kullanılamayacaktı.

C dilinde toplama işlemi (+) bulunmakta ve lisan-ı kaime gereği mümkün olduğunca değişken yapısından bağımsız tutulmaktadır.

Örneğin,

a+b şeklindeki bir işlemde, C dili, a ve b değişken değerlerini alıp [hafızaya yükledikten](#) sonra toplama işlemini yine hafızada tutmaktadır.

Yukarıdaki bu örnekte, a ve b değerleri, herhangi bir değişken tipinde olabilir. Örneğin tam sayı (int) veya karakter (char) yapısında olmasının bir mahsuru yoktur.

Ayrıca yukarıdaki a ve b değerleri birer [işaretçi \(pointer\)](#) da olabilir.

Örneğin;

```
int *a ;
int x[10];
a = x;
int b = 4;
a = a + b;
```

şeklindeki kodun anlamı, a işaretçisinin gösterdiği mevcut yerin 4 sonraki adresini göstermesidir. Buna göre, x dizisinin 4. elemanına işaret etmektedir. Yukarıdaki bu özellik, C dilinde bulunan ve işaretçi ile dizi ve değişken tipleri arasında bulunan diklik özelliğine istinat edilebilir.

Bir dilde diklik özelliğinin fazla olması, dildeki kodlamayı kolaylaştırır ve dilin öğrenilmesini basit hale getirir. Örneğin işaretçi kullanmayı öğrenen birisi yukarıdaki bu işlemleri ilave bir kural öğrenmeksizin kodlayabilir. Şayet diklik özelliği az olsaydı, bu durumda dili öğrenen veya kodlayan kişilerin istisna kurallarını da öğrenmesi gerekecekti. Örneğin ingilizce öğrenen birisi, fiillerin sonuna –ed eki getirerek geçmiş zaman yapılacağını öğrenir, ancak bunun istisnası olan kelimeleri de öğrenmek zorundadır. Şayet istisnası hiç olmasaydı, dili öğrenmek ve kullanmak çok daha kolay olacaktı.

Bir dildeki diklik özelliğinin çok fazla olması da sakıncalıdır. Aşırı derecede diklik içerilmesi halinde, dili çalıştıracak bir [derleyicinin \(compiler\)](#) geliştirilmesi imkansızlaştığı gibi kodu kullanan kişilerinde hata yapma ihtimali artar.

Örneğin C dilinde iki tip veri birliktelik ilişkisi bulunmaktadır (data associativity). Bunlar [diziler \(array\)](#) ve kayıtlardır (records, C dilinde özel olarak [yapı \(struct\)](#) ismi ile geçer). C dilinde bulunan fonksiyonlar, iki tipte de parametre alırken, sadece kayıt tipinde veri döndürebilmektedir. Dizi tipinde döndürmesi mümkün değildir. Bu durum dilin diklik özelliğine uymadığı gibi, dilde bulunan bu özelliklere de diklik eklenmesi halinde, hafıza yönetimi (Dizi boyutunun kestirilmesi) ve hafızanın ihali gibi pek çok problemi, kodu yazan kişinin kontrol etmesi gerekecekti. Çözüm olarak, C dilinin tasarımı sırasında, bu özelliklere diklik getirilmemiştir.

SORU 8: Kod Kelimesi

Haberleşmede kullanılan bir terimdir. Bir kod kelimesi (code word), belirli bir [teşrifatin \(protocol, protokol\)](#) anlamlı en küçük parçasıdır. Her kod kendi başına tek bir anlam ifade eder ve bu anlam yeganedir (unique).

Aynı yaklaşım programlama dilleri için de geçerlidir. Her programlama dilinde bulunan her kelime tek bir anlam ifade eder.

Örneğin bir programlama dilindeki “if” kelimesi, bu dildeki [kaynak kodda bulunan \(source code\)](#) kod kelimesidir (code word).

Benzer durum herhangi bir haberleşme teşrifatında da olabilir.

Kod kelimeleri kullanıldıkları yere göre kanal kelimeleri (channel code words) veya kaynak kelimeleri (source code words) olarak isimlendirilebilir. İlki haberleşme ikincisi ise programlama tabiridir.

Ancak kavramsal olarak kaynak kelimelerinin veri sıkıştırma (data compression) veya veri güvenliği (cryptography) gibi alanlarda kullanılması da mümkündür. Örneğin uzun bir kelimeyi, daha kısa bir kelime ile ifade etmenin anlamı, bu kelimenin yerine geçen bir kaynak kod kullanılmasıdır.

Benzer şekilde kanal kodları, gürültülü ortamlarda veri iletişimini güvenli hale getirmek için gereksiz ilave bilgiler içerebilir. Yani kod kelimeleri, yerine kullanıldıkları anlamdan uzun veya kısa olabilmektedirler.

Veri güvenliği açısından da bir kod kelimesi, yerine kullanıldığı kelimeye dönüşü sadece belirli kişiler tarafından yapılabilen şifreli metindir.

SORU 9: Mealy ve Moore Makineleri (Mealy and Moore Machines)

Bilgisayar bilimlerinde sıkça kullanılan sonlu durum makinelerinin ([finite state machine, FSM](#) veya [Finite State Automaton , FSA](#)) gösteriminde kullanılan iki farklı yöntemdir. Genelde literatürde bir FSM’in gösteriminde en çok moore makinesi kullanılır. Bu iki yöntem (mealy ve moore makinaları) sonuçta bir gösterim farkı olduğu için bütün mealy gösterimlerinin moore ve bütün moore gösterimlerinin mealy gösterimine çevrilmesi mümkündür.

Klasik bir FSM’de bir giriş bir de çıkış bulunur (input / output). Bu değerlerin nereye yazılacağı aslında iki makine arasındaki farkı belirler.

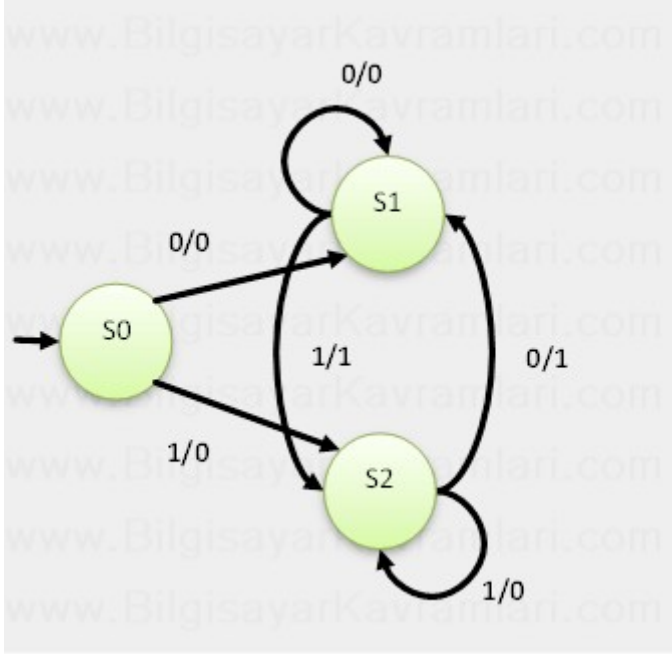
Moore makinelerinde çıkış değerleri [düğümlere \(node\)](#) yazılırken, giriş değerleri kenarlar (edges) üzerinde gösterilir.

Mealy makinelerinde ise giriş ve çıkış değerleri kenarlar (edges) üzerinde aralarına bir taksim işareti (slash) konularak gösterilir. Örneğin 1/0 gösterimi, girişin 1 ve çıktının 0 olduğunu ifade eder.

Basit bir örnek olarak [özel veya \(exclusive or, \$\oplus\$ \)](#) işlemini ele alalım ve her iki makine gösterimi ile de çizmeye çalışalım.

Girdi 1	Girdi 2	Çıktı
0	0	0
0	1	1
1	0	1
1	1	0

Klasik bir XOR kapısı, yukarıdaki [doğruluk çizelgesinde \(truth table\)](#) gösterildiği üzere iki giriş ve bir çıkıştan oluşur. Bizim makinemiz de ilk girdiden sonra ikinci girdiyi aldığı anda beklenen çıktıyı vermeli. Ayrıca makine sürekli olarak çalışmaya devam edecek. Örneğin 101011 şeklinde bir veri akışı sağlanması durumunda, sonuç olarak 1001001001 değerini hesaplamasını isteriz.



Yukarıdaki gösterim bir mealy makinesidir. Makinede görüldüğü üzere 3 farklı durum arasındaki geçişler üzerine iki adet değer yazılmıştır. Bu değerlerden ilki giriş ikincisi ise çıkış değeridir.

Makineyi beraber okumaya çalışalım.

Makinenin boşluktan bir ok ile başlayan durumu, yani örneğimizdeki S0 durumu, başlangıç durumudur. Bu durumdan başlanarak gelen değerlere göre ilgili duruma geçilir.

Örneğimizi hatırlayalım. Giriş olarak 101011 [dizgisini \(string\)](#) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Burada geçiş sırasında kullanılan kenarın üzerindeki değeri okuyalım: 1/0 bunun anlamı 1 geldiğinde geçilecek kenar olması ve çıktının 0 olmasıdır. Yani şu anda çıktımız 0

Ardından gelen değer 0 (yani şimdiye kadar 01 değerleri geldi). En son makinemizdeki durum, S2 durumuydu, şimdi yeni gelen değeri bu durumdaki kollardan takip ediyor ve S1'e giden 0/1 kenarını izliyoruz. Bu kolu izleme sebebimiz, S2 durumundan gidilen tek 0 girdisi kolu olmasıdır. Bu kol üzerindeki ikinci değer olan 1 ise, çıktının 1 olduğudur. Yani buraya kadar olan girdiyi alacak olsaydık 01 için 1 çıktısı alacaktık.

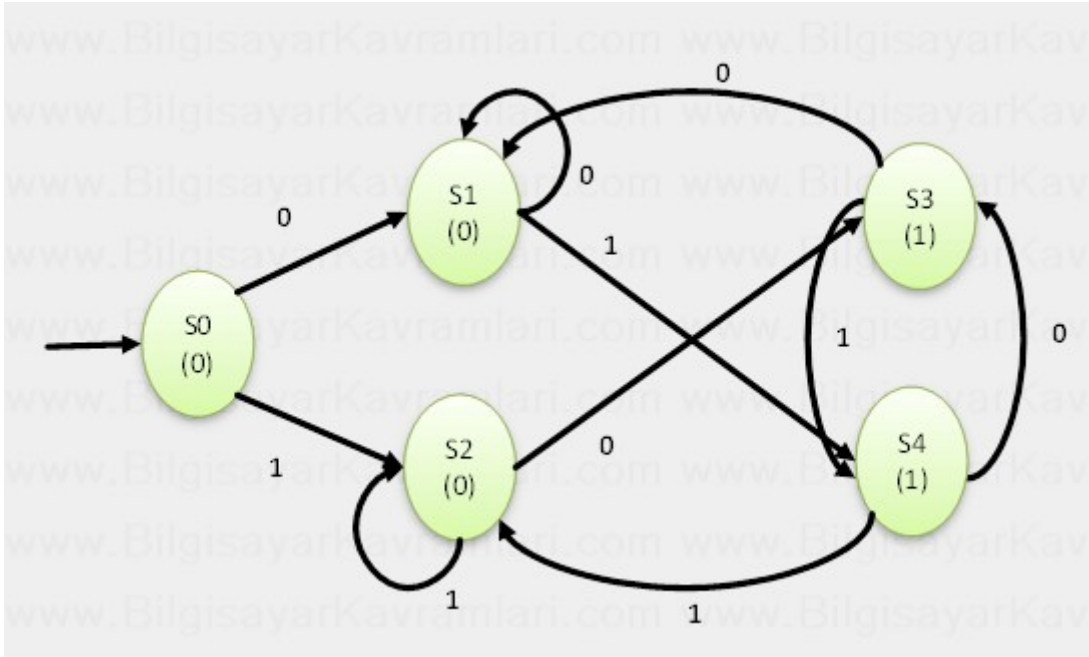
İşlemlere devam edelim ve durumları yazmaya çalışalım:

Gelen Değer	Durum	Çıkış	Yeni Durum
1	S0	0	S2

0	S2	1	S1
1	S1	1	S2
0	S2	1	S1
1	S1	1	S2
1	S2	0	S2

Yukarıdaki tablomuzun son halinde, çıkış değeri olarak 0 okunmuştur. Yani örneğimizin neticesi 0 olacaktır.

Aynı örneği moore makinesi olarak tasarlayacak olsaydık:



Moore makinesinde, mealy makinesine benzer şekilde boşluktan gelen bir ok, başlangıç durumunu belirtir.

Makinenin, durumlarında, mealy makinesinde olmayan değerler eklenmiştir. Bu değerler, ilgili durumdaki çıktıyı gösterir. Örneğin makinemiz S1 durumundayken çıktı 0 olarak okunabilir.

Şimdi moore makinesinde, aynı örneği çalıştırıp sonucu karşılaştıralım.

Giriş olarak 101011 [dizgisini \(string\)](#) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Bu geçiş sonucunda geldiğimiz S2 durumunda okunan çıktı değeri 0 yani sonuç şimdilik 0.

Ardından gelen 0 değeri ile S3 durumuna geçiyoruz ve çıktımız 1 oluyor. Çünkü S3 durumu 1 çıktısı veren durumdur. Bu şekilde durumları ve durumlar arasındaki geçişleri izlersek, aşağıdaki tabloyu çıkarabiliriz:

Gelen Değer	Durum	Çıkış	Yeni Durum
-------------	-------	-------	------------

1	S0	0	S2
0	S2	1	S3
1	S3	1	S4
0	S4	1	S3
1	S3	1	S4
1	S4	0	S2

Görüldüğü üzere, makinemiz, örnek girdi için S2 durumunda sonlanıyor ve bu durumda çıktımız 0 olarak okunuyor.

SORU 10: Preprocessor (Ön işlem)

Bilgisayar bilimlerinde, işlemciye çalışmadan önce yapılacak işleri ifade etmek için kullanılan bir terimdir. Çoğu programlama dili açısından, programın çalışması aşamasına geçilmeden önce (run-time) yapılacak işleri belirtir.

Genellikle bir betik (macro) şeklinde kayıtlı olarak dosyada duran ve programın [derlenmesi aşamasından \(compile time\)](#) önce veya sonra, ama çalıştırılmadan önce yapılan işleri ifade eder.

En çok kullanıldığı yerlerden birisi C dilidir. C dilindeki # işaretiyle başlayan satırların tamamı ön işlem komutlarıdır.

Örneğin “#include <stdio.h>” satırı, kodun derlenmesi aşamasından önce, kodun satırları işlenmeden önce, stdio kütüphanesini (standart input output) sisteme dahil eder.

Bu sayede stdio kütüphanesinde bulunan örneğin printf gibi fonksiyonları kullanabiliriz.

C dilindeki diğer bir ön işlem ise #define satırıdır. Örneğin

```
#define PI 3
```

Şeklindeki bir satırdan sonra, kodumuzdaki bütün PI değişkeni geçen değerler 3 olarak kabul edilir. Aslında buradaki işlem, derleme aşamasına geçilmeden önce, koddaki bütün PI ifadelerini 3 değeri ile değiştirmek anlamındadır.

Diğer meşhur ön işlem satırlarından birisi de #ifndef ve #endif ikilisidir. Özellikle bir başlık dosyasının (header file) kodlamasında kullanılan bu satırlar, şayet daha öncede tanımlı bir değer bulunmuyorsa ifndef satırı altına geçilmesini sağlar. Özellikle birden fazla yerden aynı anda #include edilen kütüphanelerde redefinition (tekrar tanımlama) hatasına düşmemek için kullanılır.

Örneğin bir header (.h) dosyası yazmak istiyoruz ve ismi de diyelim ki BilgisayarKavramları.h olsun. Bu dosyanın içerisinde tek satır bulunsun ve :

```
int a = 10;
```

yazıyor olsun. BilgisayarKavramları.h dosyasını her include eden dosyada a değişkeni otomatik olarak tanımlı olacaktır. Bu durumda örneğin X.c dosyamızdan include edilen bir

BilgisayarKavramları.h bulunsun bir de Y.h dosyasından benzer şekilde include edilme söz konusu olsun.

Şayet X.c dosyası ilave olarak Y.h dosyasını da include ederse, bu durumda X.c dosyası, Y.h içerisinde bulunan BilgisayarKavramları.h dosyasını da include etmiş olacak. Bu problemin çözümü için BilgisayarKavramları.h dosyasının içeriğini tek seferde çalışacak şekilde ayarlamamız gerekiyor. Bunu da derleyicinin, dosya içeriğine sadece bir kere girmesini sağlayan ifndef satırı ile yapıyoruz. Bu durumda dosyamızın içeriği aşağıdaki şekilde olabilir:

```
#ifndef BilgisayarKavramları
```

```
#define BilgisayarKavramları
```

```
int a = 10;
```

```
#endif
```

Yukarıdaki yeni dosyada, ilk include işleminden sonra BilgisayarKavramları değeri artık tanımlı olacak ve diğer include edilme durumlarında bir daha ifndef satırı geçilmeyerek a değişkenin tek sefer tanımlanması sağlanacaktır.

SORU 11: Tek atama dili (single assignment language)

Bilgisayar bilimlerinde kullanılan bir programlama dili tipidir. Bu dilde herhangi bir değişkene yalnızca bir kere atama yapılabilir. Literatürde sasl (single assignment language, tek atamalı diller) olarak geçen dil bu tipe bir örnektir.

Ayrıca C dilinden esinlenerek hazırlanmış ve C yazım kuralları ile uyumlu sacl (single assignment C language) dili de bir örnek olabilir.

Dillerin bu şekilde tasarlanmasındaki amaç, dilin derlenmesi sırasında belirli bir performans artışı sağlama kaygısıdır. Ayrıca paralel (parallel computing) ve çok işlemcili (multi processor) ortamlarda da hız artışı hedeflenir.

Bu konuda performans kaygısı ile geliştirilen farklı dil grupları da bulunur. Örneğin sıfır atamalı diller (zero assignment languages), içerisinde hiçbir değişkene atama operatörü bulunmayan dillerdir. Bu tip dillerde fonksiyonlar üzerinden atama yapılır. Yani dilde değişken yoktur, sadece fonksiyon çağrılarla değişken benzeri işlemler yapılır.

SORU 12: Hoare Mantığı (Hoare Logic)

Bilgisayar bilimlerinde, [program doğrulama \(program corectness\)](#) için kullanılan mantığın ismidir. Basitçe bir dizi matematik kuralları ile bir programı modellemeye ve programın doğruluğunu ispatlamaya (veya yanlışlığını göstermeye) yarayan mantıktır.

Bu mantığın bilgisayar dünyasında bir dil olarak modellenmesi sonucunda yine bir [muntazamn dil \(formal language\)](#) ortaya çıkar. Yani aslında bir [programın doğruluğu \(correctness\)](#) başka bir program tarafından denetlenebilir.

İşte tam bu noktada Hoare mantığı devreye girer ve bir programın nasıl denetleneceğine cevap arar. Bu mantığa göre hoare üçlüsü ismi verilen bir yapıdan faydalanılır. Buna göre her komut satırı veya program satırı ayrı bir yapıdır ve aşağıdaki şekilde gösterilebilir:

$$\{P\} C \{Q\}$$

Bu gösterimde $\{\}$ sembolleri arasında bulunan P ve Q programın çalışmasını doğrudan etkilemeyen ön ve son koşullardır (precondition postcondition). C ise ingilizcedeki command yani komut kelimesinin baş harfidir ve programlama dillerinde bulunan komutları ifade eder.

Yukarıdaki gösterimden anlaşılacağı üzere, bir programın doğruluğunu kontrol etmek için, programda bulunan her satırın ön koşulunu ve son koşulunu kontrol etme yaklaşımına hoare mantığı ismi verilir ve bu yaklaşımda şayet her komuttan önce doğru kontroller yapılır ve her satırdan sonra tam olarak beklenen durumlar belirlenirse programın doğruluğunun ispatlanabileceği iddia edilmektedir.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Hoare mantığının kendisine has bir gösterim şekli vardır. Buna göre programdaki komutlar ve bu komutlar arasındaki bağlantılar iki satır olarak gösterilir. Satırlardan üstteki, program komutlarını gösterirken alttaki de bu komutların hoare mantığındaki karşılığını gösterir.

Örnek hoare mantığı gösterimleri

Çok klasik olarak ihtiyaç duyulan bazı gösterimler aşağıda verilmiştir:

Boş komut

$$\{P\} \text{skip} \{P\}$$

Yukarıdaki gösterimde çizginin üzerinde herhangi bir komut bulunmamaktadır. Yani boş komut durumudur. Bu durumda komuttan önceki ve sonraki koşullar yani $\{P\}$ aynıdır.

Ardışık iki komut

Yukarıdaki iki ayrı satırın (S ve T satırları) [birleşimi \(conjunction\)](#) gösterilmiştir. Bu işlemde S satırının son koşulu olan ile T satırının ön koşulu olan Q aynı şarttır ve dolayısıyla iki satırın birleşmiş halinde ara koşul olacaktır. P ve R koşulları ise birleşmiş halin ön ve son koşulları olacaktır.

Koşullu komutlar

Yukarıdaki gösterim [veya operatörünün \(disjunction\)](#) ifadesidir. Buna göre bir B koşuluna bağlı olarak S veya T satırlarından birisinin çalışması yukarıdaki gösterimle mümkündür. Buradaki gösterimde P ön koşulu ile B koşulu aynı anda olduğunda S satırı, P ön koşulu olup B koşulu olmadığında da T satırı çalışmıştır. Bunun anlamı P ön koşulu her zaman olmak kaydıyla şayet B doğruysa S yanlışa T satırının çalışmasıdır.

SORU 13: Program doğruluğu (Program correctness)

Bilgisayar bilimlerinde bir programın istenen özellikleri yerine getirip getirememesine verilen isimdir. Buna göre şayet bir program, beklenen özellikleri tam ve eksiksiz yerine getiriyor, istenmeyen sonuçlar ortaya çıkmıyor ve program başladıktan sonra her durumda başarılı bir şekilde bitiyorsa bu programa tam doğru (total correctness) ismi verilir.

[Durma probeleminden \(halting problem\)](#) bilindiği üzere bir programın bitip bitmemesinin test edilmesi ayrı bir muammadır. Dolayısıyla tam doğru bir programın elde edilmesi veya ispatlanması farklı güçlükleri beraberinde getirir. Şayet bir program, kendinden beklenen özellikleri başarılı bir şekilde yerine getiriyor ancak bitip bitmemeyi garanti edemiyorsa (bir programın biteceği ispatlanamıyorsa) bu durumda programa kısmi doğru (partial correctness) ismi verilir.

Aslında programların doğruluğu (correctness), [karar problemlerinin \(decision problems\)](#) çözülmesidir.

Tam bu noktada bilgisayar bilimleri konusunda veciz sözleri ile konuya aydınlık getiren Dijkstranın bir sözünü hatırlamakta yarar var. “Testler hataların varlığını gösterebilir ama yokluğunu gösteremez.”

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Örneğin doğal sayılar kümesi üzerinde sayıların asal olup olmadığını test eden bir program yazdığımızı düşünelim. Bu program basitçe bir sayıyı alacak ve sayının çarpanlarını bulacaktır. Şayet sayının kendisi ve 1 dışında başka çarpanı yoksa bu sayıyı asal sayı ilan edecektir. Bu şekilde bir program yazmamızda herhangi bir sorun yoktur ancak programın biteceğini garanti edemeyiz. Elbette buradaki problem doğal sayıların bir sonu olup olmadığıdır. Yani program bütün asal doğal sayıları bulmaya çalışırsa sonsuz sayıda işlem yapması gerekir (ya da sonsuz bilinen bir değerse sonlu sayıda). Bu noktada bilgisayarın hafıza gibi kısıtları devreye girer. Yani sonsuz sayı işleyecek bir hafıza kapasitesi henüz bilgisayarlarda bulunmamaktadır.

Bu ispat sırasında programın bir lambda cebirine (lambda calculus) çevrilmesi ve bu cebirde doğruluğunun ispatlanması mümkündür. Bu tip ispat yöntemlerine program çıkarımı (program extraction) ismi verilir. Örneğin Curry Howard karşılığı (curry howard correspondence) yaklaşımı buna bir örnek olarak gösterilebilir.

Bir programın doğruluğunun tetkikine, program tetkiki (program verification) ismi verilir.

Program doğrulamanın kullanıldığı yerler

Program doğrulama, bir programın geliştirilme sürecinde, son adım olan test aşamasına gelinmeden hataların bulunması ve programın kodlanma hatta tasarım aşamasında hatalarının yakalanmasını sağladığı için oldukça önemlidir. Örneğin bitmiş bir programdaki bir hatanın test ile bulunması bütün tasarım ve kodlama adımlarına geri dönülmesini ve dolayısıyla hatanın telafisi için harcanan maliyetin yüksek olmasını sağlarken, daha henüz kodlama aşamсында hatanın tespiti maliyeti düşürmektedir.

Ayrıca yazılan programın hatasının test aşamasında hiçbir zaman bulunamaması da söz konusu olabilir. Bu ihtimale karşı programın matematiksel olarak doğruluğunun ispatlanması çok önemlidir.

SORU 14: Özyineli Geçiş Ağları (Reursive Transition Networks)

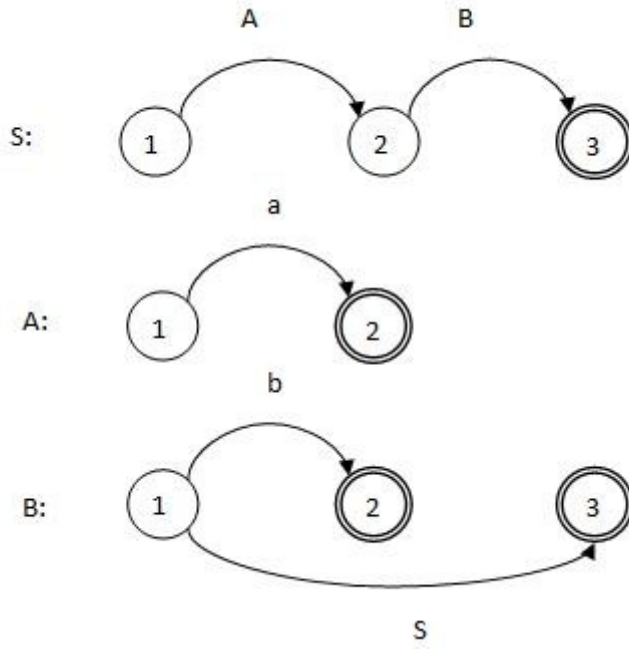
Veri modellemede kullanılan bir ağ şeklidir. Esas itibariyle [içerikten bağımsız dillerin \(context free grammars\)](#) görsel gösterimi için kullanılabilirler. Ağların yapısı [uzatılmış geçiş ağlarına \(augmented transition network\)](#) benzemekle birlikte en büyük farkı ve isminin özyineli olmasının da sebebi ağın kendini tekrarlama özelliğidir.

Daha basitçe bir [içerikten bağımsız dil \(CFG\)](#) S devamlısı (nonterminal) ile başlayan bir kurallar listesidir. Bu listedeki her kural bir devamlıdan (nonterminal) bir sonluya (terminal) doğru yapılan bir açıklamadır. Özyineli geçiş ağlarında bu açıklamada (yani [CFG](#) kurallarının sağ tarafında) da S devamlısı (nonterminal) bulunabilmektedir.

Örneğin aşağıdaki kuralları ele alalım:

$$S \rightarrow A B$$
$$A \rightarrow a$$
$$B \rightarrow b \mid S$$

Yukarıdaki son kuralda S bağlangıç devamlısına bir bağlantı kurulmuştur ve aşağıdaki şekilde görselleştirilebilir:



Yukarıdaki ağ yapısı verilen CFG örneğinin çizilmiş halidir. Buradaki çizimden de görüleceği üzere ağ yeniden başa döneme özelliğine sahiptir.

Doğal dil olarak yorumlanacak olursa bir cümlenin içinde alt cümlelerin bulunması bu ağ ile gösterilebilir. Örneğin aşağıdaki örnek cümleyi ele alalım:

Ali geldiğini söylemeyi unuttu.

Yukarıdaki örnekte 3 cümle iç içedir.

Ali gelmek

Ali söylemek

Ali unutmak

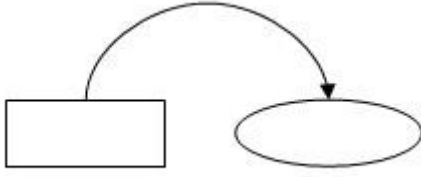
Basit bir yaklaşımla bir cümleyi özne – yüklem olarak tanımlayacak olursak yukarıdaki örnek cümlenin uyduğu yapı aşağıdaki şekilde olabilir:

C: Ö Y

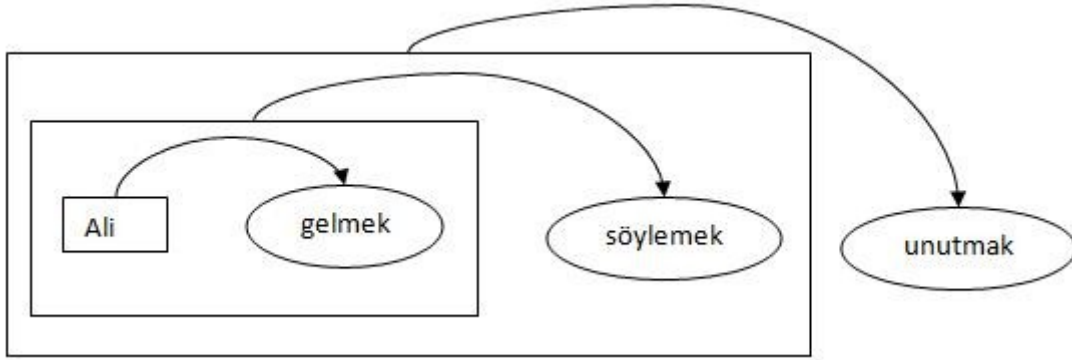
Ö : “Ali”

Y: gelmek | söylemek | unutmak

Yukarıdaki CFG yapısını şayet özyineli geçiş ağıımız (recursive transition network) ile gösterecek olursak:



Örneğin basit bir Özne-Yüklem ilişkisini yukarıdaki şekilde gösterelim. Yani özneleri dikdörtgen ve yüklemeleri yuvarlak ile gösterelim. Bu durumda örneğimiz:



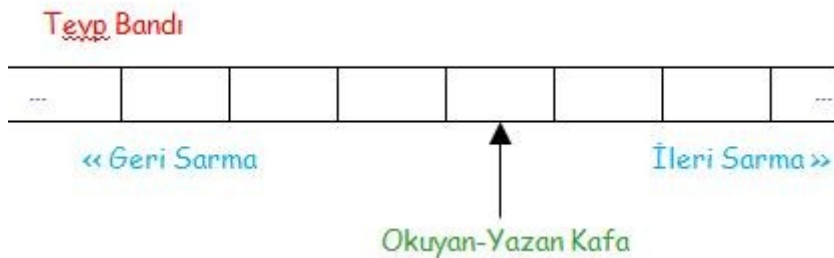
Görüldüğü üzere özyineli ağımızla gösterilen örnekte unutmak fiilinin başındaki isim kelime grubu “ali geldiğini söylemeyi” yine bir cümledir ve yapısı itibariyle bir önceki şekilde gösterilen şablonu içermektedir. Benzer şekilde söylemek fiilinin başındaki isim grubu da bir geçiş ağı (transition network) özelliği taşımaktadır

SORU 15: Turing Makinesi (Turing Machine)

Bilgisayar bilimlerinin önemli bir kısmını oluşturan [otomatlar \(Automata\)](#) ve [Algoritma Analizi \(Algorithm analysis\)](#) çalışmalarının altındaki dil bilimin en temel taşlarından birisidir. 1936 yılında Alan Turing tarafından ortaya atılan makine tasarımı günümüzde pek çok teori ve standardın belirlenmesinde önemli rol oynar.

Turing Makinesinin Tanımı

Basitçe bir kafadan (head) ve bir de teyp bandından (tape) oluşan bir makinedir.



Makinede yapılabilecek işlemler

- Yazmak

- Okumak
- Bandı ileri sarmak
- Bandı geri sarmak

şeklinde sıralanabilir.

Chomsky hiyerarşisi ve Turing Makinesi

Bütün teori bu basit dört işlem üzerine kurulmuştur ve sadece yukarıdaki bu işlemleri kullanarak bir işin yapılıp yapılamayacağı veya bir dilin bu basit 4 işleme indirgenip indirgenemeyeceğine göre diller ve işlemler tasnif edilmiştir.



Bu sınıflandırma yukarıdaki venn şeması ile gösterilmiştir. Aynı zamanda [chomsky hiyerarşisi \(chomsky hierarchy\)](#) için 1. seviye (type-1) olan ve Turing makinesi ile kabul edilebilen diller bütün tip-2 ve tip-3 dilleri yani içerik bağımsız dilleri ve düzenli dilleri kapsamaktadır. Ayrıca ilave olarak içerik bağımsız dillerin işleyemediği (üretmediği veya parçalayamadığı (parse)) $a^n b^n c^n$ şeklindeki kelimeleri de işleyebilmektedir. Düzenli ifadelerin işleyememesi konusunda bilgi için [düzenli ifadelerde pompalama savı \(pumping lemma in regular expressions\)](#) ve [içerik bağımsız dillerin işlemeyemesi için de içerik bağımsız dillerde pompalama savı \(pumping lemma for CFG\)](#) başlıklı yazıları okuyabilirsiniz.

Turing Makinesinin Akademik Tanımı

Turing makineleri literatürde akademik olarak aşağıdaki şekilde tanımlanır:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

Burada M ile gösterilen makinenin parçaları aşağıda listelenmiştir:

Q sembolü sonlu sayıdaki durumların kümesidir. Yani makinenin işleme sırasında aldığı durumardır.

Γ sembolü dilde bulunan bütün harfleri içeren alfabeyi gösterir. Örneğin ikilik tabandaki sayılar ile işlem yapılyorsa $\{0,1\}$ şeklinde kabul edilir.

Σ sembolü ile makineye verilecek girdiler (input) kümesi gösterilir. Girdi kümesi dildeki harfler dışında bir sembol taşıyamayacağı için $\Sigma \subseteq \Gamma$ demek doğru olur.

δ sembolü dilde bulunan ve makinenin çalışması sırasında kullanacağı geçişleri (transitions) tutmaktadır.

\diamond sembolü teyp bandı üzerindeki boşlukları ifade etmektedir. Yani teyp üzerinde hiçbir bilgi yokken bu sembol okunur.

q_0 sembolü makinenin başlangıç durumunu (state) tutmaktadır ve dolayısıyla $q_0 \subseteq Q$ olmak zorundadır.

F sembolü makinenin bitiş durumunu (state) tutmaktadır ve yine $F \subseteq Q$ olmak zorundadır.

Örnek Turing Makinesi

Yukarıdaki sembolleri kullanarak örnek bir Turing makinesini aşağıdaki şekilde inşa edebiliriz.

Örneğin basit bir kelime olan a^* düzenli ifadesini (regular expression) Turing makinesi ile gösterelim ve bize verilen aaa şeklindeki 3 a yı makinemizin kabul edip etmediğine bakalım.

Tanım itibariyle makinemizi aşağıdaki şekilde tanımlayalım:

$$M = \{ \{q_0, q_1\}, \{a\}, \{a, x\}, \{q_0 a \rightarrow a R q_0, q_0 x \rightarrow x L q_1\}, q_0, x, q_1 \}$$

Yukarıdaki bu makineyi yorumlayacak olursak:

Q değeri olarak $\{q_0, q_1\}$ verilmiştir. Yani makinemizin ik idurumu olacaktır.

Γ değeri olarak $\{a, x\}$ verilmiştir. Yani makinemizdeki kullanılan semboller a ve x'ten ibarettir.

Σ değeri olara $\{a\}$ verilmiştir. Yani makinemize sadece a girdisi kabul edilmektedir.

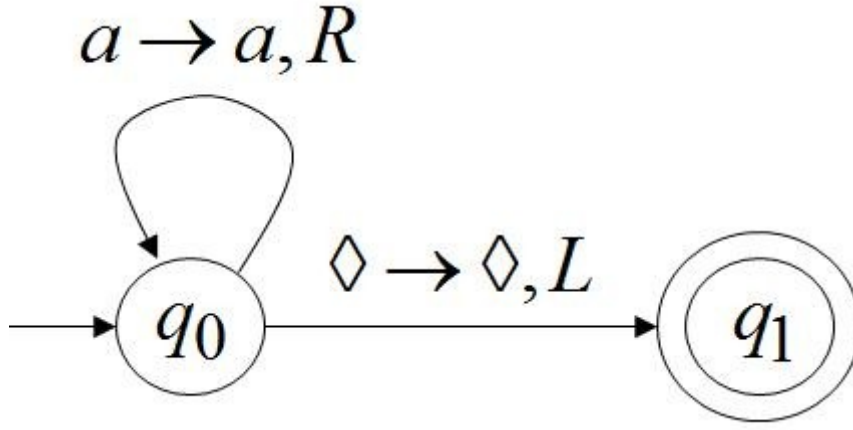
δ değeri olarak iki geçiş verilmiştir $\{q_0 a \rightarrow a R q_0, q_0 x \rightarrow x L q_1\}$ buraadki R sağa sarma L ise sola sarmadır ve görüleceği üzere Q değerindeki durumlar arasındaki geçişleri tutmaktadır.

\diamond değeri olarak x sembolü verilmiştir. Buradan x sembolünün aslında boş sembolü olduğu ve bantta hiçbir değer yokken okunan değer olduğu anlaşılmaktadır.

q_0 ile makinenin başlangıç durumundaki hali belirtilmiştir.

F değeri olarak q_1 değeri verilmiştir. Demek ki makinemiz q_1 durumuna geldiğinde bitmektedir (halt) ve bu duruma gelmesi halinde bu duruma kadar olan girdileri kabul etmiş olur.

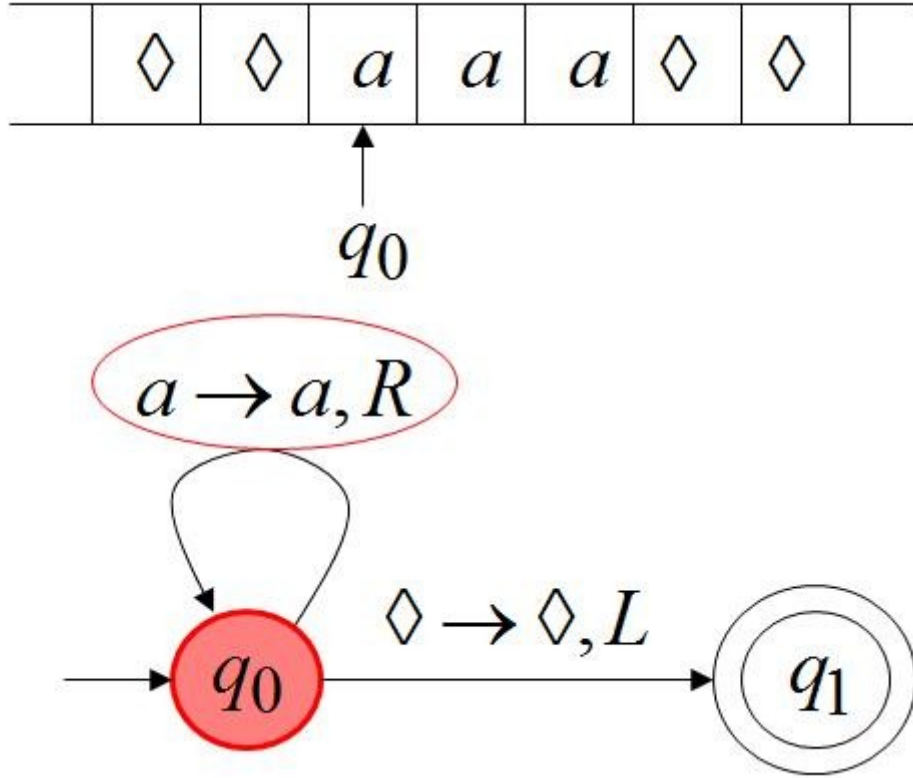
Yukarıdaki bu tanımlı görsel olarak göstermek de mümkündür:



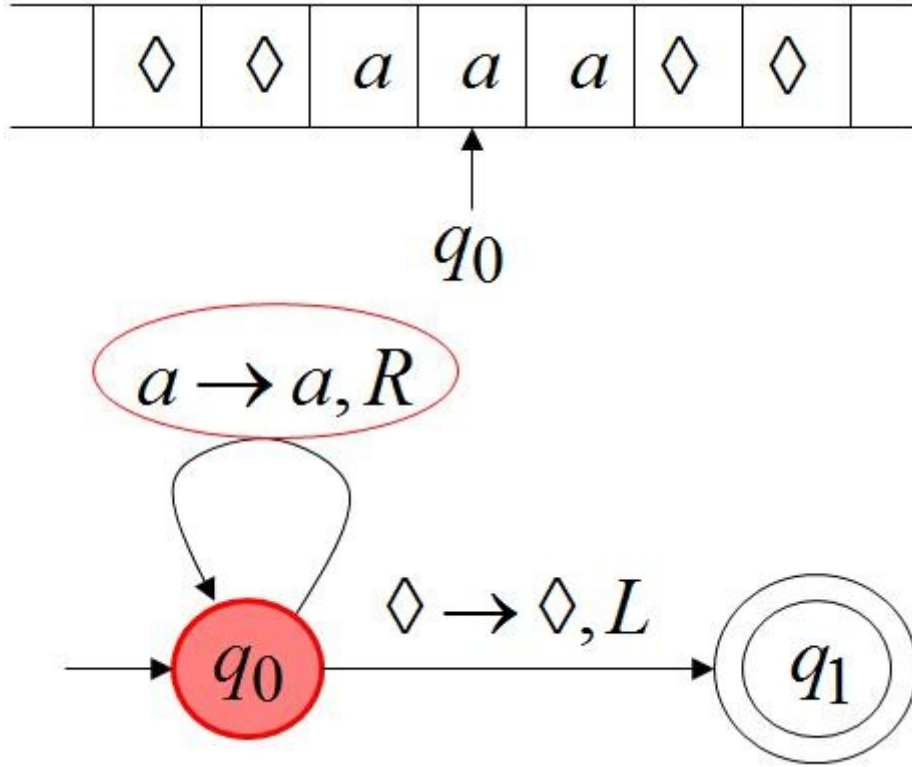
Yukarıdaki bu temsili resimde verilen turing makinesi çizilmiştir.

Makinemizin örnek çalışmasını ve bant durumunu adım adım inceleyelim.

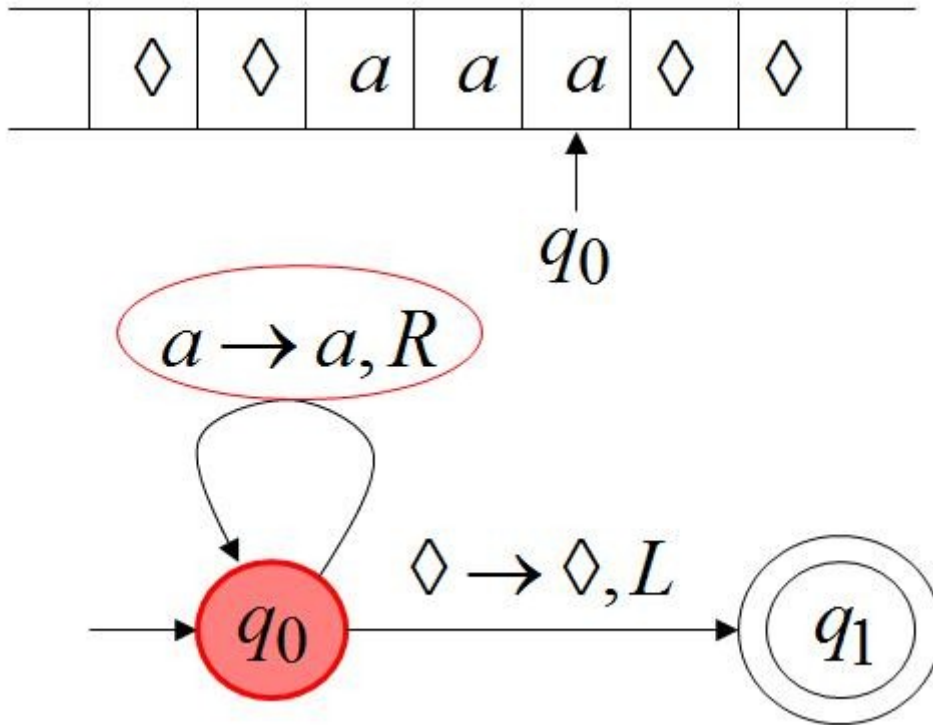
Birinci adımda bandımızda aaa (3 adet a) yazılı olduğunu kabul edelim ve makinemizin bu aaa değerini kabul edip etmeyeceğini adım adım görelim. Zaten istediğimiz de aaa değerini kabul eden bir makine yapabilmektir.



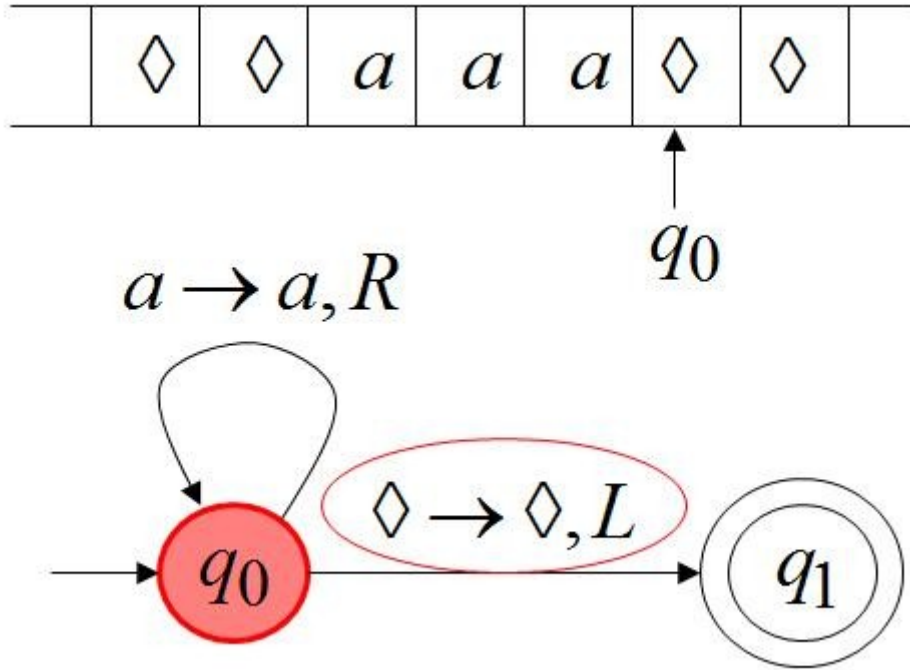
Yukarıdaki ilk durumda bant üzerinde beklenen ve kabul edilip edilmeyeceği merak edilen değerimiz bulunuyor. Makinemizin kafasının okuduğu değer a sembolü. Makinemizin geçiş tasarımına göre q_0 halinde başlıyoruz ve a geldiğinde teybi sağa sarıp yine q_0 durumunda kalmamız gerekiyor.



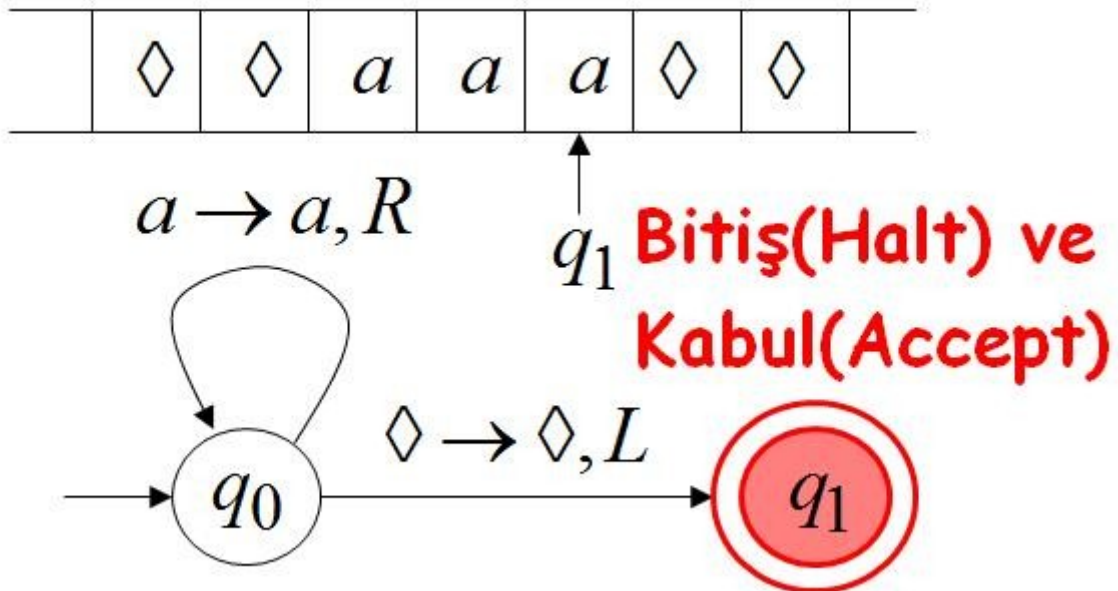
Yeni durumda kafamızın okuduğu değer banttaki 2. a harfı ve bu durumda yine q_0 durumundayken teybi sağa sarıp yine q_0 durumunda kalmamız tasarlanmıştır



3. durumda kafamızın okuduğu değer yine a sembolü olmakta ve daha önceki 2 duruma benzer şekilde q_0 durumundayken a sembolü okumanın sonucu olarak teybi sağa sarıp q_0 durumunda sabit kalıyoruz.



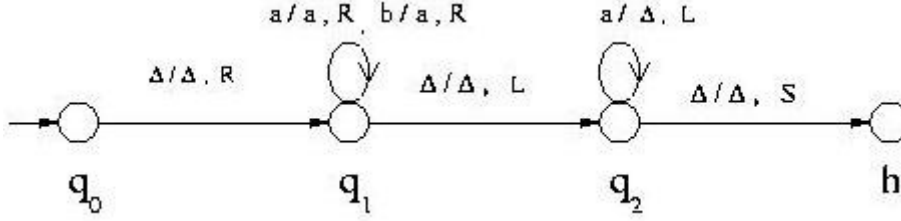
4. adımda teypten okuduğumuz değer boşluk sembolü x oluyor. Bu değer makinemizin tasarımında q_1 durumuna gitmemiz olarak tasarlanmış ve teybe sola sarma emri veriyoruz.



Makinenin son durumunda q_1 durumu makinenin kabul ve bitiş durumu olarak tasarlanmıştı (makinenin tasarımındaki F kümesi) dolayısıyla çalışmamız burada sonlanmış ve giriş olarak aaa girdisini kabul etmiş oluyoruz.

2. Örnek

Hasan Bey'in sorusu üzerine bir örnek makine daha ekleme ihtiyacı zuhur etti. Makinemiz $\{a,b\}$ sembolleri için çalışsın ve ilk durum olarak bandın en solunda başlayarak bandta bulunan sembolleri silmek için tasarlansın. Bu tasarımı aşağıdaki temsili resimde görülen otomat ile yapabiliriz:



Görüldüğü üzere makinemizde 4 durum bulunuyor, bunlardan en sağda olan h durumu bitişi (halt) temsil ediyor. Şimdi bu makinenin bir misal olarak “aabb” yazılı bir bandta silme işlemini nasıl yaptığını adım adım izah etmeye çalışalım.

Aşağıda, makinenin her adımda nasıl davranacağı bant üzerinde gösterilmiş ve altında açıklanmıştır. Sarı renge boyalı olan kutular, kafanın o anda üzerinde durduğu bant konumunu temsil etmektedir.

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

q0 durumunda başlıyoruz. Ve boşluk ile bandı sağa sarıyoruz:

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

a veya b değeri okundukça bant sağa sarılmaya devam ediyor ve q1 durumunda kalıyoruz.

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

Okunan değer b ise banda geri a değeri yazılıyor

◇	◇	a	a	a	b	◇
---	---	---	---	---	---	---

www.bilgisayarkavramlari.com

◇	◇	a	a	a	a	◇
---	---	---	---	---	---	---

En sağda boşluk değerini okuyunca (◇) Sağa sarma işlemi bitiyor ve geri dönüyoruz

◇	◇	a	a	a	a	◇
---	---	---	---	---	---	---

◇	◇	a	a	a	◇	◇
---	---	---	---	---	---	---

◇	◇	a	a	◇	◇	◇
---	---	---	---	---	---	---

◇	◇	a	◇	◇	◇	◇
---	---	---	---	---	---	---

◇	◇	◇	◇	◇	◇	◇
---	---	---	---	---	---	---

Tekrar boşluk (◇) görülünce makine bitiyor.

Geri sarma işlemi sırasında a değerleri silinmiş oluyor

Netice olarak Hasan Bey'in sorusuna temel teşkil eden ve örneğin q1 üzerindeki döngülerden birisi olan b/a,R geçişi, banttan b okunduğunda banta a değerini yaz manasındadır.

SORU 16: Özyineli Sayılabilir Diller (Recursively Enumerable Languages)

Muntazam dillerden (formal languages) birisi olan ve bu özelliği ile Mantık, Matematik ve Bilgisayar bilimlerinin çalışma alanına giren bir dil çeşididir. Sınıflandırma olarak Chomsky Hiyerarşisinde (Chomsky Hierarchy) 0. seviye olan (Type 0) bu dile uygun bütün diller birer düzenli ifade (regular expression) ile gösterilebilir.

Muntazam dil (formal language) olması dolayısıyla dilde üretilen her özyineli sayılabilir kelime kümesi (recursively enumerable set) bütün kelimelerden çıkarılabilecek güç kümesinin (power set) bir alt kümesi olmak zorundadır. Bu anlamda bir özyineli sayılabilir dili aşağıdaki üç farklı tanımla tanımlamak mümkündür:

Bir dilde bulunan alfabeden üretilebilen bütün kelimeleri kabul eden dil yani Σ sembolü ile dilimizdeki alfabe yi yani harfler kümesini gösterecek olursak L ile gösterilen dilimiz Σ^* ile üretilebilen kelimeler kümesidir.

[Turing makinesi \(Turing machine\)](#) ile işlenebilen veya [hesaplanabilir bir fonksiyon \(computable function\)](#) bulunabilen dildir. Burada dikkat edilecek nokta dilin sonsuz olabileceğidir. Yani dilimizde sonsuz sayıda tekrar bulunabilir. [Turing makinesi](#) ve [hesaplanabilirlik teorisi \(computability theory\)](#) dikkatle okunacak olursa dilin sonsuz olmasının bir sakıncası yoktur. Teorik olarak sonlu zamanda bitiyor olması yeterlidir ve dil sonsuz bile olsa sonlu zamanda işleyen bir makine veya fonksiyon bulunabilir.

Şayet bir dilde üretilen bir [dizgi \(string\)](#) için bir [Turing makinesi \(turing machine\)](#) veya [hesaplanabilir bir fonksiyon \(computable function\)](#) bulunuyorsa, diğer bir ifadeyle ürettiğimiz [turing makinesi](#) şu üç durumdan birisini yapıyorsa:

- [bitmek \(halt\)](#)
- [döngü \(loop\)](#)
- red etmek (reject)

bu durumda bu dil özyineli sayılabilir dildir denilebilir.

Özyineli sayılabilir diller temel olarak [chomsky hiyerarşisinde \(chomsky hierarchy\)](#) bulunan bütün diğer dilleri kapsar. Bu anlamda hiyerarşinin en alt seviye dilidir ve diğer dillere göre çok daha esnektir.

SORU 17: Chomsky Hiyerarşisi (Chomsky Hierarchy)

Bilgisayar bilimlerinin özellikle dil alanında yapılan çalışmalarında [muntazam dilleri \(formal languages\)](#) tasnif etmek için kullanılan bir yapıdır. Literatürde Chomsky–Schützenberger hiyerarşisi olarak da geçmektedir.

Bilindiği üzere ([muntazam diller \(formal languages\)](#) veya [CFG](#) yazısından da okunabileceği üzere) [muntazam dillerin](#) dört özelliği bulunur. Bunlar özellikle [içerikten bağımsız dillerin \(context free languages\)](#) da temelini oluşturmuştur.

- sonlular (terminals)
- devamlılar (nonterminals)
- üretim kuralları (devamlıların değerlerini belirleyen geçiş kuralları)
- Başlangıç devamlısı

Örneğin aşağıdaki [içerikten bağımsız dilbilgisini \(context free grammar\)](#) ele alalım

$$S \rightarrow AB$$

$$A \rightarrow Aa \mid \varepsilon$$

$$B \rightarrow b$$

Yukarıdaki bu [CFG](#) tanımındaki sonlular (terminals) $\{a,b,\varepsilon\}$, devamlılar (nonterminals) $\{S,A,B\}$ olarak tanımlanır. üretim kuralı olarak (production rules) S,A,B'nin açılımlarını

gösteren ve \rightarrow sembolü ile belirtilen satırlar bulunmaktadır. Son olarak başlangıç devamlısı (nonterminal) değeri olarak S kabul edilmiştir. Başlangıç devamlısı böyle bir kural bulunmamasına karşılık genelde S harfi (start kelimesinden gelmektedir) ile gösterilmekte ve ilk satırda bulunmaktadır.

Yukarıdaki bu CFG şayet [düzenli ifadeye \(regular expression\)](#) çevrilirse a^*b şeklinde yazılabilir. Aslında burada anlatılan değer $a^n b$ şeklinde de gösterilebilen istenildiği kadar a değeri alan (hiç almaya da bilir) ve sonra mutlaka b ile biten dildir.

Chomsky yukarıdaki şekilde tanımlanan [muntazam diller \(formal languages\)](#) için bir sınıflandırmaya gitmiş ve 4 seviye belirlemiştir.

- Type-0 (tip 0) Sınırlanmamış diller (unrestricted grammars)
- Type-1 (tip 1) İçerik duyarlı diller (context-sensitive grammars)
- Type-2 (tip 2) [İçerikten bağımsız diller \(context free languages\)](#)
- Type-3 (tip 3) [Düzenli diller \(regular grammars\)](#)

şeklinde 4 seviye isimlendirilmiştir. Bu seviyelerde iler gidildikçe (seviye arttıkça) dili bağlayan kurallarda sıkılaşmakta ve dil daha kolay işlenebilir ve daha belirgin (deterministic) bir hal almaktadır.

Tip-0 dilleri basitçe [turing makinesi \(turing machine\)](#) tarafından çalıştırılabilen ve bir şekilde bitecek olan dillerdir ([hesaplanabilir diller \(computable languages\)](#)). Örneğin [özyineli sayılabilir diller \(recursive enumerable languages\)](#) bu seviyeden kabul edilir.

Tip-1 dilleri için içerik duyarlı diller örnek gösterilebilir. Basitçe $\alpha A \beta \rightarrow \alpha \gamma \beta$ şeklindeki gösterime sahip bir dildir. Buradaki A devamlı (nonterminal) ve α, γ, β birer sonlu (terminal) terimdir. Bu sonlulardan α ve β boş harf (yani ϵ veya bazı kaynaklarda λ) olabilir ancak γ mutlak bir değere sahip olmalıdır (yani boş olamaz) Ayrıca bu tipte

$$S \rightarrow \epsilon$$

şeklinde bir kurala da izin verilmektedir. Burada S devamlısının sağ tarafta olmaması gerekmektedir. Bu diller doğrusal bağlı otomatlar (linear bounded automaton) ile gösterilebilir. Doğrusal bağlı otomatlar, [turing makinelerinin](#) özel bir halidir ve [Turing makinesinde](#) bulunan bantın sabit uzunlukta olduğu (çalışmanın sabit zaman sonra sona ereceği) kabul edilir.

Tip-2 diller ise [CFG](#) ile ifade edilebilen [içerikten bağımsız dillerdir \(context free languages\)](#). Bu dillerin özelliği $A \rightarrow \gamma$ şeklinde gösterilmeleridir. Buradaki γ değeri sonlular (terminals) ve devamlılar (nonterminals) olabilmektedir. Bu diller [aşağı sürüklemeli otomatlar \(push down automata PDA\)](#) tarafından kabul edilen dillerdir ve hemen hemen bütün programlama dillerinin temelini oluşturmaktadırlar. (programlama dillerinin neredeyse tamamı bu seviye kurallarına uymaktadırlar)

Tip-3 diller ise [düzeltilmiş ifadeler \(regular expressions\)](#) ile ifade edilebilen (veya üretilebilen veya parçalanabilen (parse)) dillerdir. Bu dillerin farkı

$$A \rightarrow \alpha \text{ ve}$$

$$A \rightarrow \alpha B$$

şeklindeki kurallar ile gösterilebilmeleridir.

Yukarıdaki bu tanımları bir tabloda toplamak gerekirse:

Seviye	Dil Örneği	Otomat Uygulaması	Kuralları
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow \alpha$ ve $A \rightarrow \alpha B$

Yukarıdaki seviyeler bütün dilleri kapsamak için yeterli değildir ayrıca yukarıda gösterilen seviyelere giren yukarıdaki tablo dışında diller de bulunmaktadır.

SORU 18: Muntazam Diller (Formal Languages)

Kısaca istisnası olmayan dillere muntazam dil diyebiliriz.

Muntazam diller bilgisayar bilimlerinde, mantıkta ve dilbilim (linguistic) çalışmalarında kullanılan bir dil ailesidir. Dilde bulunan bütün öğeler ve dilin ulaşabileceği sınırlar belirli kurallar dahilinde tanımlanabiliyorsa bu dillere muntazam dil ismi verilir.

Bu anlamda bilgisayar bilimlerinde bulunan bütün programlama dillerini bu ailede düşünmek mümkündür.

Temel olarak bir dildeki harfleri bu harflerden oluşabilecek [kelimelerin oluşma kurallarını \(morphology\)](#) ve bu [kelimelerin dizilimini \(syntax\)](#) muntazam bir şekilde belirleyebilen kurallarımız varsa. Ve bu kuralların istisnası yoksa bu dilin bir muntazam dil olduğu söylenebilir.

Muntazam dilimizi tanımlamak için öncelikle dilin en ufak ögesi olan harflerden başlamak gerekir.

Alfabe ve Harf tanımı

Bir dilde kullanılan sembollerin her birisine harf denilir. Bu harflerin listesine alfabe (bazı kaynaklarda abece) denilir.

Örneğin [haber mantığını \(predicate logic\)](#) ele alacak olursak bu mantığın ifade ettiği dil için $\wedge, \neg, \forall, ,)$, (sembolleri kullanılır. Ayrıca mantığımızdaki değişkenleri ve önermeleri gösteren x_0, x_1, x_2, \dots şeklinde sonsuz harf bulunmaktadır.

Kelime (word) tanımı

Bir alfabedeki sembollerin (harflerin) o dildeki kelimebilim(morphology) kurallarına uygun olarak oluşturduğu dizilime kelime ismi verilir. Kelimeler dildeki anlamlı en küçük

birimlerdir ve tek başlarına sadece bir birimlik bilgi ifade ederler. Daha karmaşık bilgi ifadeleri için cümlelere ihtiyaç duyulur. Ne yazık ki dilbilimde bir kelimenin anlattığı bilgi ile bir cümlemin anlattığı bilgi arasında kesin bir çizgi bulunmamaktadır. Örneğin bazı doğal dillerde tek kelime ile anlatılan bir duygu veya olay, başka dillere bir kelime grubu hatta bazan bir cümle ile çevrilebilmektedir.

Muntazam diller için de aynı durum söz konusudur. Dilin tasarımında hangi bilginin yeterli varlık oluşturacağı ve ne kadar detaya inileceği kararlaştırılır.

Hatta bir dilin var olması için sözdizim (syntax) seviyesinin bulunması gerekmez. Yani sadece kelimeler olan ve kelimelerin dizilmesi gerekmeyen diller de bulunabilir.

Bir dilde üretilebilecek kelimelerin belirli kurallarla tanımlanması ve bu kuralların dışına çıkan istisna bulunmaması durumunda bu dile muntazam dil denilebilir.

Örneğin [düzenli ifadeleri \(regular expressions\)](#) ele alırsak, bu ifadelerin kullanılması ile dilde oluşabilecek sonsuz sayıdaki kelime belirlenebilir.

$(a + b)^*$ şeklindeki bir düzenli ifadeden a ve b harfleri kullanılarak (istenilen sıra ve istenilen sayıda) kleene yıldızı sayesinde boş kümeyi de kapsayan bütün kelimeler üretilebilir.

Muntazam Dil tanımı

Yukarıdaki dilin öğelerini tanımladıktan sonra dili tanımlamak mümkündür.

Σ sembolü ile dilimizdeki alfabeyi yani harfler kümesini gösterecek olursak L ile gösterilen dilimiz Σ^* ile üretilebilen kelimeler kümesidir.

Bu tanıma ilave olarak muntazam dil kavramının matematik ve bilgisayar bilimlerinde geçen pek çok “dil” kelimesini karşıladığını söyleyebiliriz. Yani aslında matematik ve bilgisayar bilimlerinde geçen “dil” kelimesi aslında tam olarak “muntazam dile” karşılık gelmektedir.

Örneğin [içerikten bağımsız dil \(context free language\)](#) terimi aslında içerikten bağımsız muntazam dil anlamında kullanılmaktadır. Benzer şekilde düzenli diller (regular languages) terimi aslında düzenli muntazam dil (regular formal languages) anlamında kullanılmaktadır. Dolayısıyla matematik ve bilgisayar bilimlerinin uğraştığı bütün dilleri ve kullandıkları bütün dilleri muntazam dil (formal language) olarak tanımlamak mümkündür. Bunun tek istisnası bilgisayar bilimlerindeki yapay zeka konusunun altında geçen doğal dil işleme konusudur.

Muntazam dil çalışmaları

Muntazam diller üzerinde yapılan çalışmalardan birincisi bir muntazam dilin nasıl tanımlanacağıdır. Bu alanda yapılan bazı çalışmalar sonucunda elde edilen yaklaşımları aşağıdaki şekilde sayabiliriz:

- Muntazam bir dilbilgisi (formal grammer) tarafından üretilen dillere Muntazam dil (formal language) ismi verilir. Bu tanım [Chomsky Hiyerarşisi](#) tarafından sınıflandırılan dilbilgileri ile gösterilebilir.
- [Düzenli ifadelerin \(Regular expressions\)](#) bir örneği ile üretilebilen dillere muntazam dil ismi verilir.

- Bir otomat tarafından üretilen ([sonlu durum otomatu \(finite state automat\)](#) veya [Turing Makinesi \(Turing Machine\)](#) gibi) dillere muntazam dil ismi verilir.

Yukarıdaki bu tanımlama çalışmalarının yanında muntazam diller ile ilgili aşağıdaki sorulara da cevap aranmaktadır.

- Tanımlama yeteneği. Bir dilin tanımlayabilme gücü ve tanımlayabildiği dilin büyüklüğü nedir? Aynı güce ve yeteneğe sahip ikinci bir dil farklı bir şekilde olabilir mi? Bütün dilleri gösteren bir dil yapılabilir mi? şeklindeki soruları bu grup altında toplayabiliriz.
- Algılama yeteneği. Verilen bir kelimenin verilen bir dile ait olup olmadığının algılanması yeteneğidir. Elimizde muntazam bir dilbilgisi (grammar) olduğunu kabul edelim. Bu dilden olan veya olmayan bir kelime verildiğinde bu kelimenin analiz edilmesi ve dile ait olup olmadığının algılanması en verimli nasıl yapılabilir? sorusuna cevap arayan çalışmalardır.
- Karşılaştırma yeteneği. Elimizde iki dil bulunsun. Bu iki dilin birbirine göre farklarının algılanılması, bir dilde olup diğer dilde olmayan veya iki dilde ortak olan kelime ve cümlelerin tespiti gibi konularda çalışılan çalışma grubudur.

Yukarıdaki bu problemler genel olarak algoritma analizi (analysis of algorithms) altındaki karmaşıklık teorisi (complexity theory) alanın çalışma konularına girmektedir.

Diller üzerinde tanımlı işlemler

Diller üzerinde de farklı işlemlerin yapılması mümkündür. Aşağıda bu işlemler ve tanımları verilmiştir:

Dillerin kesişimi (intersection): Temel olarak bir dili o dildeki kurallar ve alfabe marifetiyle üretilebilecek kelimeler kümesi olarak tanımlarsak, iki dilin kesişimi aslında iki dilde üretilebilecek kelimelerin kesişimi olmuş olur. $L_1 \cap L_2$ şeklinde gösterilir.

Üleştirme işlemi (concatenation): İki dilden üleştirme yapmak için birinci dilden bir kelime ile ikinci dilden bir kelimenin arka arkaya eklenmesi ile yeni bir kelime oluşturulması kastedilir. Yani L_1 dilinden k kelimesi ile L_2 dilinden l kelimesi alınıp kl veya lk şeklinde kelime üretme işlemidir. L_1L_2 şeklinde gösterilir.

Tümleyen işlemi (Complement): Bir dildeki [güç kümesinden \(power set\)](#) o dildeki dilbilgisine (grammar) göre çıkarılabilen kelimelerin farkıdır. Yani alfabede olan harflerle yazılabilecek bütün kelimelerin, dilde izin verilen kelimelerle [fark kümesidir](#). $\neg L$ şeklinde veya \overline{L} şeklinde gösterilir.

Kleene Yıldızı (Kleene Star): Bir dildeki alfabe üzerinde boş küme dahil olmak üzere üretilebilecek bütün kelimeler kümesidir. Bir anlamda güç kümesi (power set) ismi de verilebilir. Kullanımı aslında [düzenli ifadelerden \(regular expressions\)](#) alınmıştır.

Ters işlemi (reverse): Bir dildeki herhangi bir kelimenin yazılışının terse çevrilmiş halidir. Örneğin “abc”nin tersi “cba” şeklindedir. Daha akademik olarak iki kuralla tanımlanabilir.

- Şayet dilimizdeki boş kelimeyi ϵ sembolü ile gösterirsek $\epsilon^R = \epsilon$ olur yani boş kelimenin tersi yine boş kelimedir.

- Boş kelime dışındaki herhangi bir kelime için $w = x_1 \dots x_n$ şeklinde n adet harf ile yazıldığını düşünelim, bu durumda tersi $w^R = x_n \dots x_1$, olur

Bu durumda muntazam bir dil L için, $L^R = \{w^R \mid w \in L\}$ olarak tanım yapılabilir.

SORU 19: Değişken Tip Bağlama (Dynamic Type Binding, Müteharrik Şekil Bağı)

Programlama dillerinde bir değişkenin tipinin belirlenmesi iki türlü olabilir. Birinci tip tanımlamada değişkenin tipi sabit olarak atanır ve bir kere belirlendikten sonra değişmez (static binding). İkinci tip tanımlamada ise değişkenin tipi değişebilir (dynamic binding).

Bu tanımlama tiplerine geçmeden önce açıktan ve gizli bağlama şekillerine bakalım.

Sabit tip bağlamaları (Static Type Binding)

Şayet değişkenin (variable) tipi açıkça tanımlanıyor ve programcı tarafından belirleniyorsa bu tip tanımlamalara açıktan tanımlama (explicit declaration) şayet açıkça belirtilmiyor ancak içerisine konulan verinin tipine göre belirleniyorsa bu tip tanımlamalara da gizli bağlama ile tanımlama (implicit declaration) ismi verilir.

Örneğin

`int x; // C,C++,JAVA veya C# gibi dillerde`

tanımlamasında x değişkeninin tipi açıkça belirtilmiş ve tam sayıları (integer) alabileceği söylenmiştir. Buna mukabil:

`var x; // javascript`

`dim x; 'visual basic`

tanımlamasında x isminde bir değişken tanımlanmış ancak tipi belirsiz bırakılmıştır.

Örneğin FORTRAN dilinde I, J, K, L, M veya N harfleriyle başlayan değişkenler tam sayı (integer) ve diğer bütün tanımlamalar ise reel sayı olarak belirlenmiştir ve içsel olarak bu tanımlanma kendiliğinden yapılmış programcının bir tanımlama yapmasına gerek bırakılmamıştır.

Benzer şekilde PERL dilinde bazı özel semboller ile değişken tipleri belirlenir. Örneğin \$ sembolü ile başlayan bir değişken sabit bir sayı tutabilir (scalar) buna karşılık @ sembolü ile başlayan değişkenler [dizilerdir \(arrays\)](#) yine benzer şekilde % işareti ile başlayan değişkenler ise özet değerleri (hashing) tutmaktadır.

Hareketli Tip Bağlamaları (dynamic type binding)

Yukarıda açıklanan sabit bağlamalara (static binding) karşılık değişken bağlamalarda (dynamic binding) değişkenin (variable) tipi atandıktan sonra değişebilir.

Yani yukarıda açıktan (explicit) veya kapalı (implicit) olarak tip belirlendikten sonra değişmemektedir. Örneğin


```
int x;
```

tanımından sonra x değişkeninin değeri tamsayı olmaktadır.Veya

```
var x;
```

```
x=3;
```

satırlarından sonra x değişkeninin içerisine konulan değer tamsayı olduğu için değişken değeri bu şekilde kalmaktadır.

Buna karşılık hareketli bağlamalarda (dynamic binding) tip bir kere atandıktan sonra değişebilir. Örneğin:

```
bilgi = { 2 , 3 , 4 };
```

şeklindeki bir tanımla bilgi ismindeki değişkene bir dizi konulmuştur. Bu durumda bilgi değişkeninin bir dizi olduğu sonucuna varılır ve tipi bu şekilde atanır. Ancak yukarıdaki satırdan sonra aşağıdaki şekilde bir satır gelirse:

```
bilgi = "ali";
```

bu durumda değişkenin tipi [dizgi \(string\)](#) olarak yeniden atanmış olur ve bu satırdan sonra bu değişken üzerinde yapılan işlemler dizgi (string) işlemleri olarak kabul edilir.

Tip çıkarımı (Type inference, Şekil istidlali)

Miranda, Haskell ve ML gibi programlama dillerinde fonksiyonların tip çıkarımı yapması durumudur.

Örneğin ML dilinde aşağıdaki örneği ele alalım:

```
function alan (r) : 3.14 * r * r;
```

Yukarıda r yarıçapında bir dairenin alanını hesaplayan fonksiyon verilmiştir. Bu fonksiyonda dönen değerin tipi reel sayı olacaktır çünkü fonksiyon içerisinde 3.14 gibi reel bir sayı ile çarpım yapılmıştır.

İşte bu noktada programlama dili, fonksiyonun içeriğinden bir çıkarım yapmaktadır.

ML programlama dilinde çıkarım yapılamayan durumlarda programcının bir tipi elle belirtmesi istenir. Örneğin:

```
function carp(x): x * x;
```

yukarıdaki fonksiyonda x değerinin tipi bilinmediği için ve fonksiyonun dönüş tipi tahmin edilemeyeceği için programcının fonksiyonu aşağıdaki şekilde yazması gerekir:

```
function kare(x): int = x * x;
```


SORU 20: İşlem Önceliği (Operator Precedence)

Bilgisayar bilimlerinde önemli konulardan birisi olan programlama dillerinin işlemleri yapma sırasını belirler. Bir programlama dilinde işlem önceliği bir iki farklı unsura göre belirlenir. Öncelikle sonucu etkileyen ve işlemin matematiksel önceliğine göre bir tercih yapılır. Örneğin çoğu C benzeri dillerde [C dilindeki işlem öncelikleri](#) kullanılır.

İçerik

1. Soldan sağa öncelik (left to right precedence)
2. Sağdan sola öncelik (right to left precedence)
3. İşlemlerde kısa devre (Short Circuit)
4. İşlemlerde kısa devreden doğan Yan Etki (Side Effect)
5. İşlemlerde çalışma önceliğinden doğan Yan Etki (Side effect)

Bilindiği üzere programlama dilleri [kesin \(Deterministic\)](#) dillerdir ve muğlak ifadeler (ambiguity) izin verilmez. Dolayısıyla bu işlem önceliğinin yanında aynı öncelikli işlemlerde belirli bir sıraya göre yapılması gerekir. Bu aşamada soldan sağa öncelik (left to right precedence) ve sağdan sola öncelik (right to left precedence) şeklinde iki seçim yapılabilir.

Soldan sağa öncelik (left to right precedence)

Aynı önceliğe sahip işlemlerin soldan sağa doğru çalıştırılmasıdır. Aşağıdaki örneği ele alalım:

$$3 + 5 + 7 + 9$$

Programlama dili tasarlanırken şayet soldan öncelik verilirse yukarıdaki işlemin yapılma sırası aşağıdaki şekilde olur.

$$(((3+5) + 7) + 9)$$

Yukarıda görüldüğü üzere önce soldaki işlem sonra bu işlemin sonucu üzerinden sırayla soldan sağa diğer işlemler yapılır.

Sağdan sola öncelik (right to left precedence)

Aynı önceliğe sahip işlemlerin sağdan sola doğru çalıştırılmasıdır. Aşağıdaki örneği ele alalım:

$$3 + 5 + 7 + 9$$

Programlama dili tasarlanırken şayet sağdan öncelik verilirse yukarıdaki işlemin yapılma sırası aşağıdaki şekilde olur.

$$(3 + (5 + (7 + 9)))$$

Yukarıda görüldüğü üzere önce sağdaki işlem sonra bu işlemin sonucu üzerinden sırayla soldan sola doğru diğer işlemler yapılır.

İşlemlerde kısa devre (Short Circuit)

Mantıksal işlemlerde, [derleyici \(compiler\)](#) bazı işlemleri kısa devre yapabilir. Örneğin aşağıdaki en basit mantıksal işlemler olan [ve \(and\)](#) ve [veya \(or\)](#) işlemlerini ele alalım.

a	b	VEYA (Or)
0	0	0
0	1	1
1	0	1
1	1	1

a	b	VE (And)
0	0	0
0	1	0
1	0	0
1	1	1

Yukarıdaki tabloda görüldüğü üzere Ve işlemi sırasında a veya b taraflarından bir tanesinin 0 olması sonucun doğrudan 0 olmasına ve benzer şekilde Veya işlemi sırasında da tarafların bir tanesinin 1 olması sonucun 1 olmasına sebep olmaktadır.

Programlama dilleri tasarlanırken bu durumdan faydalanarak hız artışı sağlanır. Örneğin aşağıdaki [eğer satırını \(if statement\)](#) ele alalım:

```
if( a < 5 || b < 10)
```

Yukarıdaki bu kontrolde a'nın değerinin 5'ten küçük ve b'nin değerinin 10'dan büyük olması kontrol edilmiştir. Şayet a'nın değeri 5'ten küçükse bunun anlamı yukarıdaki Veya işleminin sonucunun doğru (true) çıkacağıdır ve bu durumda işlemin geri kalanı olan b<10 kontrolü yapılmaz.

Benzer şekilde aşağıdaki kontrolde de Ve işlemi kısa devre yapılabilir:

```
if( a < 5 && b < 10)
```

Yukarıda Ve kontrolü sırasında şayet a değişkeninin değeri 5'e büyük veya eşitse kontrolün bu ilk kısmı yanlış (false) olarak dönecektir. Bu durumda kontrolün ikinci kısmına bakılmasına gerek kalmayacak ve sonuç yanlış (false) olarak değerlendirilerek eğer satırına (if statement) girilmeyecektir.

İşlemlerde kısa devreden doğan Yan Etki (Side Effect)

İşlemlerin çalıştırılması sırasında beklenmeyen durumların oluşmasına yan etki ismi verilir. Örneğin aşağıdaki kodu inceleyelim:

```
if ( a < 5 || ++ b > 4)
```

Yukarıdaki satırda programı yazan programcı büyük ihtimalle a'nın 5'ten küçük olup olmadığını kontrol etmiş ve ayrıca b'nin değeri bir arttırıldığında 4'ten büyük olup olmadığını kontrol etmiştir.

Ancak yukarıdaki bu niyet gerçekleşmeyebilir. Bir önceki konuda gördüğümüz kısa devre (shor circuit) işlemi gerçekleşirse ve a'nın değeri 5'ten küçükse programlama dili kontrolün ilk kısmı doğru olduğu ve kontrol bir Veya işlemi olduğu için ikinci kısma bakmaksızın sonucu doğru ilan edecektir. Ne yazık ki programcının b'nin değerini 1 artırma beklentisi de gerçekleşmeyecektir.

Örnek bir çalışma için :

a=3 , b=7

ile yukarıdaki kontrol yapılırsa çalışma sonucunda b= 7 olarak kalır. Ancak

a=7 , b=7

ile yukarıdaki kontrol yapılırsa çalışma sonucunda b = 8 olarak değişir.

İşlemlerde çalışma önceliğinden doğan Yan Etki (Side effect)

Programlama dillerinin işlemleri soldan sağa veya sağdan sola doğru çalıştığını görmüştük. Bu durum bir takım beklenmedik yan etkilere sebep olabilir. Örneğin aşağıdaki kodu ele alalım:

```
int i = 1;
f() {
    i=i+2;
}
g() {
    i=i*2;
}
main() {
    int j = f() + g() + i++;
    print(j);
}
```

Yukarıdaki kodu incelediğimizde i isminde bir global değişken tanımlanmıştır. Bu değişken f fonksiyonunda 2 arttırılmış, g fonksiyonunda ise 2 ile çarpılmıştır.

Buradaki soru main fonksiyonunda j değişkeninin değerinin ne olacağıdır.

Şayet soldan sağa öncelikli çalıştırma yapılıyorsa bu durumda önce f sonra g fonksiyonları çalıştırılarak toplanacak ardından da i'nin değeri 1 arttırılarak sonuca eklenecektir. Yani sırasıyla:

f() -> i = 3 olacak

g() -> i = 6 olacak

j değeri için $3 + 6 + 7 = 16$ bulunacaktır.

Ancak çalışma sırası sağdan sola öncelikli olsaydı:

i'nin değeri 1 arttırılarak 2 olacaktı

g() -> i= 4 olacak

f() -> i = 6 olacak

sonuçta j için $6 + 4 + 2 = 12$ bulunacaktır.

Yukarıda da görüldüğü üzere programlama dilinin işlemlerin önceliği aynı olduğunda bile hangi sırayı kullandığı çok önemlidir. Yani toplama işleminin yer değiştirme özelliği bulunur ama bu özellik yukarıdaki şekilde yazılan kodlarda problem olabilir.

SORU 21: Çok boyutlu diziler (MultiDimensional Arrays)

Bilgisayar bilimlerindeki pek çok programlama dilinde birden fazla boyuttan oluşan dizilerin kullanılması mümkündür.

Örneğin bir ders çizelgesini, haftalık yemek listesini yada kişilerin aylık satışlarından oluşan bir tabloyu ele alalım. Günlük hayatta pek çok yerde tablolar kullanılmaktadır. Aynı zamanda matrisler (masfuf, matrix) matematikte küçümsenmeyecek bir öneme sahiptir.

İçerik

1.	Çok boyutlu dizilerin tanımlanması
2.	Çok boyutlu dizilerin kullanılması
3.	İkiden çok boyutlu diziler

Tutulan bilgi her ne olursa olsun şayet tablo gibi iki boyutlu bir bilgiyse ya da daha fazla boyutu varsa bu bilginin tek boyutlu dizilerle modellenmesi ve işlenmesi güçtür.

Bunun yerine programlama dillerinde birden fazla boyuttan oluşan [diziler \(array\)](#) kullanılabilir. Bu dizilere çok boyutlu dizi (multi dimensional array) ismi verilir.

Çok boyutlu dizilerin tanımlanması

Günümüzde oldukça yaygın olan C yazım kurallarına göre (C-syntax) bir matris aşağıdaki şekilde tanımlanabilir:

```
int a[3][3];
```

Yukarıdaki bu satırı, C/C++ programlama dilinde yazarsak hafızada bizim için 3×3 boyutlarında (3'e 3'lük) bir matris için yer açar (yani toplamda 9 hücreli ve 9 farklı sayı tutmaya yarayan bir yer) ve bu matrisin ismini a olarak tanımlar. Buradaki a matrisin (yani iki boyutlu dizinin (array)) ismidir ve herhangi bir değişken ismi verilebilir. Örnek olarak a ismi kullanılmıştır.

Yukarıdaki tanım satırını [JAVA](#) veya C# dillerinde aşağıdaki şekilde yapabiliriz:

```
int [][] a = new int[3][3];
```

Yukarıdaki bu yazımda farklı olan sadece yazılış şeklidir aslında yapılan iş aynıdır.

Yukarıdaki şekilde dizi tanımlandıktan sonra bu dizinin istenilen satır ve sütun değerlerine erişilebilir. Bu durumu aşağıdaki temsili resimden görebiliriz:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

Yukarıdaki resimde ilgili satır ve sütünlara erişmek için yazılması gereken indis değerleri verilmiştir. Yani örneğin aşağıdaki şekilde bir kod yazarsak:

```
a[1][2]=5;
```

Bu kod satırı ile a dizisinin 1. satırının 2. sütünuna 5 değeri konulmuş olur:

0	0	0
0	0	5
0	0	0

Burada önemli bir uyarı yapmak gerekir. Yukarıdaki satır ve sütun bilgileri tamamen bir kabule dayalıdır. Yani matrisin ilk indisi satır ikinci indisi sütun olarak gösterilmiş ve kullanılmıştır. Oysaki bilgisayarın hafızasında (RAM, Memory) tek boyutlu bir yapı vardır ve bizim iki boyutlu yapılarımız tek boyuta indirilerek tutulur. Dolayısıyla bizim satır ve sütun tutan indislerimiz bir kabule dayanır. Yani aslında aşağıdaki şekilde bir gösterim de pek âlâ doğrudur:

```
a[0][0]  a[1][0]  a[2][0]
a[0][1]  a[1][1]  a[2][1]
a[0][2]  a[1][2]  a[2][2]
```

Yukarıdaki her iki gösterimde bir kabule dayanır (aslında hafızada tek boyutlu olan bilginin iki boyutlu gösterilmesi bir kabuldür) ve programcı hangi kabulü isterse yapabilir ancak birisini kabul ederek bütün programcılık hayatı boyunca bu kabul üzerine devam edebilir.

Ancak programlama dili yazan arkadaşların (derleyiciler teorisi (compiler theory) gibi konular ile ilgilenen kişilerin) bu detayı bilmesinde yarar olabilir. Bu yüzden dilerlerse [satır bazlı sıra \(row major order\)](#) ve [sütun bazlı sıra \(column major order\)](#) başlıklı yazıları okuyabilirler.

Çok boyutlu dizilerin kullanılması

Çok boyutlu diziler genelde [döngüler \(loop\)](#) ile birlikte kullanılırlar. Bilindiği üzere aslında dizi (array) kavramının varlık sebebi birden fazla değişkeni hafızada bir arada tutmak ve kolayca ulaşmaktır. Dolayısıyla birden fazla değişkene erişirken indis numaralarından (satır ve sütun numaralarından) erişmek çoğu zaman avantajlı bir durumdur. İşte bu satır ve sütun numaraları üzerinde çalışan döngüler de çoğu zaman vaz geçilmez erişim araçlarıdır.

Örneğin aşağıdaki kodu ele alalım:

```
int a[5][5]; //diziyi tanımladık
//içine değer atıyoruz
for(int i = 0;i<5;i++){
```

```

        for(int i = 0;i<5;i++){
            a[i][j]=i+j;
        }
    }
    //değerleri bastırıyoruz
    for(int i = 0;i<5;i++){
        for(int j = 0;j<5;j++){
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
}

```

Yukarıdaki kodun çıktısı aşağıdaki şekildedir:

```

0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

```

Görüldüğü üzere yukarıdaki kod ile 5×5 boyutlarında bir matris tanımlanmış ve bu matrisin içerisine 01234 değerlerinden oluşan satır her seferinde 1 arttırılarak tekrarlanmıştır.

Yukarıda bu işlemi yapmaya yarayan kod değerlerin atandığı ve bastırıldığı iki bölüm olarak düşünülebilir ve her iki bölümde de dizinin elemanlarına döngüler marifeti ile erişilmiştir.

İkiden çok boyutlu diziler

Şimdiye kadar anlatılan diziler iki boyutluydu. Programlama dillerinde ikiden yüksek sayıdaki boyutlarda dizi tanımlamak da mümkündür. Temel olarak programlama dillerinin çoğunda matrisin boyutunun bir limiti yoktur yani 3 boyutlu 4 boyutlu yada 100 boyutlu diziler tanımlanabilir.

Aslında dizi tanımı sırasında tek limit bilgisayarın donanım ve işletim sistemi kaynaklarından doğar. Yani örneğin bir tam sayı değerinin (int) 2 bayt (byte) kapladığı bir işletim sisteminde 10000x10000x10000 boyutlarına sahip 3 boyutlu bir dizi tanımlanırsa ($10^4 \cdot 10^4 \cdot 10^4 = 10^{12} \times 2$ byte = 2×10^{12} byte \sim 1 terabyte) yaklaşık olarak 1TB yapar ki bu günümüz bilgisayarları için oldukça yüksek bir RAM miktarıdır.

Programlama dillerinde çok boyutlu diziler iki boyutlu dizilerde olduğu gibi boyut miktarını belirten ilave sayılar tanımlayarak gösterilir. Örneğin:

```
int a[5][5][5];
```

şeklindeki bir tanım C/C++ dillerinde veya

```
int [][][] a = new int[5][5][5];
```

şeklindeki bir tanım JAVA vey C# dillerinde 5x5x5 boyutlarında 3 boyutlu bir dizi tanımlamak için kullanılabilir.

Burdaki boyut sayısında bir limit bulunmamaktadır. Örneğin

```
int a[5][5][5][5][5];
```

şeklindeki bir satır ile 5 boyutlu (ve her boyutu 5 olan) bir dizi tanımlanabilir.

SORU 22: Sayma (Enumeration, Tâdâd)

Bilgisayar bilimlerinde alınabilecek alternatiflerin sayılması ve bu sayılan ihtimaller dışındaki ihtimallerin kabul edilmemesi durumudur (ihtimallerin tâdât edilmesi)

Örneğin programlama dillerinde bir değişkenin alabileceği değerleri tanımlayarak bu değişkene sadece bu değerlerden birisinin konulması sağlanabilir.

Örneğin C dilinde yazılmış aşağıdaki kodu ele alalım:

```
enum                                                    gunler{
pts,sal,car,per,cum,cts,paz
}gun;

main(){
gun=sal;
printf(“n%d”,gun);           //2.           gün           yani           1
gun=car;           //           sonraki           gun           yani           car           yani           2
printf(“n%d”,gun);
getch();
}
```

Yukarıdaki kod öncelikle gunler isminde bir enum tipi tanımlamakta ve bu tip marifetiyle gun ismindeki değişkenin alabileceği değerleri “pts,sal,car,per,cum,cts,paz” ihtimalleri ile sınırlamaktadır.

Kodun main fonksiyonunda ise bu değişken kullanılmış ve sırasıyla içerisine sal ve car değerleri verilerek bu değerlerin sayısal karşılıkları ekrana basılmıştır.

Yukarıdaki örnekten ve açıklamadan da anlaşılacağı üzere bir değişkenin ya da bir varlığın alabileceği değerler [kümesinin](#) tanımlanması işlemine tadat veya enumeration ismi verilmektedir.

SORU 23: Atomluluk (Atomicity)

Latince bölünemez anlamına gelen atom kökünden üretilen bu kelime, bilgisayar bilimlerinde çeşitli alanlarda bir bilginin veya bir varlığın bölünemediğini ifade eder.

Örneğin programlama dillerinde bir dilin atomic (bölünemez) en küçük üyesi bu anlama gelmektedir. Mesela C dilinde her satır (statement) atomic (bölünemez) bir varlıktır.

Benzer şekilde bir verinin bölünemezliğini ifade etmek için de veri tabanı, veri güvenliği veya veri iletimi konularında kullanılabilir.

Örneğin veri tabanında bir işlemin (transaction) tamamlanmasının bölünemez olması gerekir. Yani basit bir örnekle bir para transferi bir hesabın değerinin artması ve diğer hesabın

değerinin azalmasıdır (havale yapılan kaynak hesaptan havale yapılan hedef hesaba doğru paranın yer değiştirmesi) bu sıradaki işlemlerin bölünmeden tamamlanması (atomic olması) gerekir ve bir hesaptan para eksildikten sonra, diğer hesaba para eklenmeden araya başka işlem giremez.

Benzer şekilde işletim sistemi tasarımı, paralel programlama gibi konularda da bir işlemin atomic olması araya başka işlemlerin girmemesi anlamına gelir.

Örneğin sistem tasarımında kullanılan check and set fonksiyonu önce bir değişkeni kontrol edip sonra değerini değiştirmektedir. Bir değişkenin değeri kontrol edildikten sonra içerisine değer atanmadan farklı işlemler araya girerse bu sırada problem yaşanması mümkündür. Pekçok işlemci tasarımında buna benzer fonksiyonlar sunulmaktadır.

Genel olarak bölünmezlik (atomicity) geliştirilen ortamda daha düşük seviyeli kontroller ile sağlanır. Örneğin işletim sistemlerinde kullanılan [semafor'lar \(semaphores\)](#), kilitler (locks), koşullu değişkenler (conditional variables) ve monitörler (monitors) bunlar örnektir ve işletim sisteminde bir işlemin yapılması öncesinde bölünmezlik sağlayabilirler.

Kullanılan ortama göre farklı yöntemlerle benzer bölünmezlikler geliştirilebilir. Örneğin veritabanı programlama sırasında koşul (condition) veya kilit (lock) kullanımı bölünmezliği sağlayabilir.

SORU 24: Veri yapıları üzerinde fonksiyonlar

Ardışık veri yapıları (consecutive data structures) üzerinde çalışan fonksiyonlar temel olarak 3 grupta toplanabilir. Bu gruplarda amaç yazılabilecek fonksiyon tiplerini üç çatı altında toplamak ve bir [abstraction \(soyutlama\)](#) yaparak programcıya sadece istediği fonksiyonu parametre verebileceği bir ortam hazırlamaktır.

Bu işlem sırasında [fonksiyon göstericileri \(Function pointers\)](#) kullanılarak bir fonksiyon diğer fonksiyonlara parametre verilebilir.

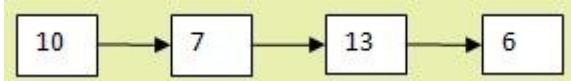
[Bindirme fonksiyonları \(mapping style functions\)](#) : Bu fonksiyonlarda amaç verilen veri yapısının her elemanını verilen fonksiyona tabi tutmaktır. (örneğin bir listedeki her elemanın bir artırılması)

[Biriktirme fonksiyonları \(accumulator style functions\)](#) : Bu fonksiyonlarda amaç verilen veri yapısı üzerine verilen biriktirici fonksiyonu uygulayarak tek bir sonuç elde etmektir (örneğin bir listenin tamamının toplamının bulunması)

[Filtreleme fonksiyonları \(Filtering style functions\)](#): Bu fonksiyonlarda amaç verilen bir veri yapısı üzerinde verilen bir filtreleme fonksiyonuna uymayan elemanların elenmesidir.

SORU 25: Filtreleme Tipi Fonksiyonlar (Filter Type Functions)

Bir veri yapısı üzerinde çalışan ve veri yapısında bulunan verileri, verilen bir fonksiyonu kullanarak eleyen (filtreleyen) fonksiyon tipidir. Bu durum aşağıdaki [bağlı liste \(linked list\)](#) örneğinden anlaşılabilir:



Örneğin yukarıdaki bağlı liste üzerinden tek sayıları eleyen bir fonksiyon yazmak istersek yapmamız gereken, tek sayıları veren aşağıdaki fonksiyonu bu bağlı liste üzerine filtre tipinde uygulamaktır.

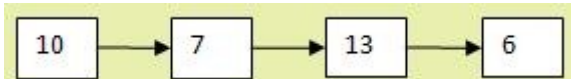
```
int tek(int a){  
  
return a%2==1;  
  
}
```

Yukarıdaki fonksiyon tek sayılarda 1 ve çift sayılarda 0 döndürmektedir (C dilinde 1'in olumlu (doğru, true) ve 0'ın olumsuz (yanlış, false) olduğunu hatırlayınız). Yukarıdaki bu fonksiyonu, [fonksiyon göstericisi \(function pointer\)](#) kullanarak aşağıdaki filter fonksiyonuna parametre verebiliriz:

```
node *filter ( node *root, int (*pt2Func)(int)){  
    node* iter = root;  
    while(iter->next != root){  
        if(pt2Func(iter->next->data)){  
            node * temp=iter->next;  
            iter->next = iter->next->next;  
            free(temp);  
        }  
        iter = iter->next;  
    }  
    if(pt2Func(root->data)){  
        node*temp=root;  
        root=root->next;  
        free(temp);  
    }  
    return root;  
}
```

SORU 26: Biriktirme Tipi Fonksiyonlar (Accumulator Type Functions)

Bir veri yapısı üzerinde çalışan ve veri yapısında bulunan verileri, verilen bir fonksiyonu kullanarak bir değişkende biriktiren fonksiyon tipidir. Bu durum aşağıdaki bağlı liste (linked list) örneğinden anlaşılabilir:



Örneğin yukarıdaki listede toplama fonksiyonu olarak aşağıdaki fonksiyonu uygulayacak olursak:

```
int topla(int a, int b){  
  
return a + b;  
  
}
```

sırasıyla yapılması gereken işlemler aşağıdaki şekildedir:

$$10 + 7 = 17$$

$$17 + 13 = 30$$

$$30 + 6 = 36$$

ve biriktirme tipi olan toplama fonksiyonundan dönen değer olarak sonuçta 36 çıkması beklenir.

Çoğu uygulamada biriktirme fonksiyonunun etkisiz elemanının, parametre olarak alınması da söz konusudur. Bu durumda listedeki işleme bu değer ile başlanabilir. C dilinde bu işi yapan genel bir fonksiyon yazmak mümkündür ve bu iş için fonksiyon göstericilerinden (function pointers) faydalanılmalıdır:

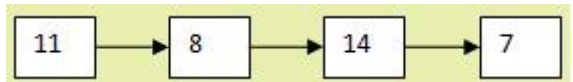
```
int accumulate (node *root, int nulvalue, int (*pt2Func) (int, int))
{
    int acc = (*pt2Func) (nulvalue, root->value);
    node * iter = root;
    while (iter != NULL) {
        iter = iter->next;
        acc = (*pt2Func) (acc, iter->value);
    }
    return acc;
}
```

SORU 27: Bindirme Tipi Fonksiyonlar (Mapping Style Functions)

Veri yapıları (data structures) üzerinde uygulanan [döngü \(loop\)](#) tiplerinden birisidir, literatürde haritalama tipi olarak da geçmektedir. Temel olarak bir veri tipi üzerindeki değişimi uygulamak için kullanılırlar. Örneğin aşağıdaki [bağlı listeyi \(linked list\)](#) ele alalım:



Yukarıdaki bu bağlı listedeki elemanların üzerine, sayı değerini 1 arttıran bir fonksiyon bindirildiğinde (map) aşağıdaki sonuç elde edilir:



Yani her eleman teker teker bu fonksiyona tabi tutulur ve sonuçta elde edilen listede yapısal bir değişim olmaz, sadece sayı değerleri değişir.

C dilinde bir map fonksiyonunun yazılması için [fonksiyon göstericilerine \(function pointers\)](#) ihtiyaç duyulur.

```
void map( node *root, int (*pt2Func) (int))
{
    node * iter = root;
    while (iter != NULL) {
```

```

printf("value:%d",iter->value);
iter->value=(*pt2Func)(iter->value);
iter=iter->next;
}
}

```

SORU 28: İçerik Bağımsız Gramerler için Pompalama Önsavı (Pumping Lemma for Context Free Grammers)

Bilgisayar bilimlerinde bir dilin, [içerik bağımsız gramer \(context free grammar, CFG\)](#) ile gösterilemeyeceğini ispatlamaya yarar. Yani pompalama ön savı sayesinde bir dilin CFG olmadığı ispatlanabilir ancak olduğu ispatlanamaz. Şayet pompalama önsavını geçemiyorsa CFG değildir denilebilir ancak geçmesi olmasını gerektirmez.

[Pomplama önsavı \(pumping lemma\)](#) kısaca bir dili aşağıdaki gramere uydurmaya çalışır:

$$s = uvxyz$$

Elimizde ispatı ile uğraştığımız L dili olsun ve yukarıdaki s kelimesini bu dilden bir kelime olarak üretelim. Ve bu kelime $|vxy| \leq p$, $|vy| \geq 1$ (p burada pompalama boyutudur) şartlarını sağlasın. Şimdi bu kelimeyi aşağıdaki şekilde pompayalım:

$$s = uv^i xy^i z$$

Şayet bu pompalama sırasında üretilen $i \geq 0$ için kelimelerde L dilindense o halde bu dil içerik bağımsız gramer CFG ile ifade edilebilir, şayet oluşan kelimeler L dili tarafından kapsanmıyorsa bu durumda da L dili, CFG olarak ifade edilemez sonucuna varılabilir.

Örnek:

Pompalama önsavını (pumping lemma) kullanarak aşağıdaki dilin CFG olmadığını ispatlayalım:

$$L = \{a^i b^i c^i \mid i > 0\}$$

Yukarıdaki dili, tanımımızdaki

$$s = uvxyz$$

şekline getirmeye çalışalım ve bu sırada $|vxy| \leq p$, $|vy| \geq 1$ şartlarını sağlamaya çalışalım. Bu durumda ilk kelimemiz:

$$s = a^l b^l c^l$$

olarak yazılacaktır. Bu yazımdan u ve z değerlerinin boş olabileceğini düşünersek

$$v = a$$

$$x = b$$

$$y = c$$

olarak yazılabilir. Bunun dışındaki şartların hiç birisi $s = uvxyz$ şartını ve $s = a^i b^i c^i \mid i > 0$ şartını aynı anda sağlamaz.

Şimdi ilk kelimemizi yazdığımıza göre, kelimemizi pompalayarak çıkan sonuçların yine bu dilde olup olmadığını kontrol edelim:

Yukarıdaki denemede $i=1$ için yazmıştık, şimdi $i=2$ için $uvxyz$ değerlerini bulmaya çalışalım.

Böyle bir değer bulunamaz çünkü

$$s = a^2 b^2 c^2$$

değerini ancak

$$s = uv^i xy^i z$$

pompalamasında $i=2$ yazarak sağlamaya çalışabiliriz ve bu durumda da x değeri olan b değeri 1 tane kalacağı için problem olacaktır. Daha basit bir ifadeyle CFG gösteriminde 3 değer birden aynı sayıda olması garanti edilemez, ancak iki değer aynı sayıda olabilir. Çünkü yukarıdaki v ve y değerleri pompalandığında artarken x değeri tek değer olarak kalacaktır.

Benzer şekilde

$$u = a$$

$$v = b$$

$$y = c$$

olması durumunda u ,

$$v = a$$

$$y = b$$

$$z = c$$

olması durumunda da z değerleri tek kalacak ve diğer harfler pompalanırken dengeyi bozacaktır.

SORU 29: Düzenli İfadelerde Pompalama Önsavı (Pumping Lemma for Regular Expressions)

Bir dilin [Düzenli ifade \(Regular expression\)](#) olup olmadığının belirlenmesi için kullanılan [pomplama önsavı \(pumping lemma\)](#). Basitçe düzenli ifadede olup olmadığı sınanacak bir w dili için (yani $L = w$ için)

$$w = xyz$$

şeklinde bir açılım sınanır. Buradaki sınama sırasında aşağıdaki koşulların sağlanması beklenir:

1. $|y| \geq 1$
2. $|xy| \leq p$
3. bütün $i \geq 0$ için, $xy^iz \in L$

Yukarıdaki üç şartı da sağlayan bir dil için düzenli ifadedir denilebilir. Ayrıca bu dilin düzenli ifade olabilmesi için yukarıdaki y teriminin üstü olan i değeri istenildiği kadar arttırılabilmeli ve elde edilen sonuç yine düzenli ifade (Regular expression) olmalıdır.

Daha basit bir ifadeyle, bir dilden üretilen bir terim üç parçaya bölündüğünde ortasındaki parça boş olmamalı, istenildiği kadar tekrarlanabilmeli ve çıkan bütün sonuçlar yine aynı dilden olmalıdır.

Örnek:

Pompalama önsavı konusunda en çok verilen örneklerden birisi $L = \{a^n b^n : n \geq 0\}$ dilinin bir düzenli ifade olmadığına ispatıdır. Bu örneği beraberce ispatlayalım:

Dilimizi pompalama önsavındaki kalıp olan $w = xyz$ şekline benzetmeye çalışalım. Bu durumda dilimizdeki üretilen en kısa terim için $w = a^p b^p$ olduğu söylenebilir ve bu terimi kalıba benzetirken xy için a ve z için b olduğunu söyleyebiliriz. Çünkü y 'nin sıfır boyutunda olamayacağını biliyoruz, bu durumda y değeri ya a ya da b olmalıdır. İki durumda birbirinin aynısıdır. Bu durumda

$$y=a$$

$$z=b$$

olarak kabul edelim.

Şimdi sonucumuzu pompalayalım ve bir sonraki beklentimiz olan xy^2z terimini üretelim. Bu durumda çıktımız $w = a^2b$ olacaktır. Dikkat edilirse bu yeni terim, dil tanımımız olan $L = \{a^n b^n : n \geq 0\}$ tanımı ile çelişmektedir ve bu dilin bir üyesi değildir.

Dolayısıyla $L = \{a^n b^n : n \geq 0\}$ dilinin bir düzenli ifade olmadığı veya düzenli ifade olarak yazılamayacağı ispatlanmış olur.

Örnek Çözüm :

Mesut Aydın bey efendinin sorusuna istinaden aşağıdaki çözümü yazıyorum:

Sorunun tam metnini kendileri yorum olarak bu sayfada yazmışlar ancak bir kerede buraya alıntılıyorum:

L dili için ($x \in (0,1)^*$ ve $W=XX^R$) Pumping lemma ile regürlüğünü araştırın. Örneğin : $x = 011$ olsun $x^R = 110$ olur ve $w=011110$ olur.

Zannediyorum x^R ile ifade edilmek istenen, x teriminin tersi oluyor.

Şimdi bakın böyle bir soruda doğrudan cevap olarak RE (Regular Expression) olamayacağını söyleyebilirim (biraz tecrübe ile biraz da olaya şu şekilde bakmanız gerekiyor) Bunun sebebi, basitçe herhangi bir RE için sadece bir pompalama noktasına izin verilmesidir. Diyelim ki x için değişim belirttiniz, bu durumda x^R 'nin buna bağlı değişmesi soruda bir şart olarak tanımlanmış, oysaki biz sadece tek noktada pompalama yapabiliyoruz ve dolayısıyla x değişirken x^R şeklinde verilen ikinci kısma müdahale edemiyoruz.

Daha akademik olarak durumu ifade edecek olursak:
Yukarıda verilen W non-regular (düzenli olmayan bir ifadedir).

İspatı : L dilini (L tanımıyla üretilebilecek kelimeler kümesini) öncelikle düzenli ifade kabul edelim (regular expression)

Bu kabul doğruysa, $n > 1$ uzunluğunda bir pompalama uzunluğu bulunmalıdır ki $W \in L$ olsun ve her $|w| \geq n$ için $x = klm$ yazılabilsin. (Burada klm üç harfini kullandım bunun sebebi aslında yukarıda anlatılan ve yazıda geçen xyz kullanırsak x harfinin sorudaki x ile karışması endişesidir. Soruda x harfi kullanıldığı için (L 'den üretilebilen herhangi bir kelime olarak) bu harfi tekrar kullanmadım). Ayrıca bu klm üç harfi için aşağıdaki şartlar da sağlanmalıdır:

- $|kl| \leq n$
- $|l| > 0$
- son olarak $kl^i m \in L$ bütün $i \geq 0$ değerleri için sağlanmalıdır.

Şimdi yukarıda yazdığımız ve pompalama önsavının tanımından gelen durumları çürüten bir örnek bulursak, bu L dilinin düzenli ifade olmadığını, [olmayana ergi \(burhan-ı mütanakis, proof by contradiction\)](#) yöntemi ile ispat etmiş olacağız:

Örneğin $X = 0^n$ olarak verilmiş olsun. Bu durumda $W = 0^n 0^n$ olacaktır çünkü 0^n teriminin tersi yine kendisidir.

Yukarıdaki örnek W için $W \in L$ rahatlıkla denilebilir. Çünkü soruda verilen tanım sağlamaktadır. Ayrıca dikkat ediniz W terimi X ve X^R terimlerinden mürekkep olduğu için bu terimin boyu her zaman çifttir yani $|W|$ her zaman çift bir sayıdır diyebiliriz. Çünkü X ile X^R kelimelerinin boyu eşittir. Yani bir kelimenin boyu ile tersinin boyu eşit olduğu için bu iki kelimeden oluşan W 'nin boyutu tabii olarak çift sayı olacaktır.

Pompalama önsavından öğrendiğimiz üzere, W terimi n adet 0 ile başladığına göre, herhangi bir j sayısı, $0 \leq j \leq n$ şartını sağlamak üzere, pompalama önsavından gelen tanım itibariyle $l = 0^j$ olur denilebilir. Yani diğer bir deyişle, klm dizilimindeki l teriminin pompalanacağını biliyoruz ve bu terim soruda verilen W ifadesini gösterebilmek için 0^j olarak seçilmeli ki pompalama neticesinde 0^n terimine ulaşılsın (veya 0^n terimine ulaşılan kadar pompalansın).

Örneğin $kl^i m$ ifadesi için $i = 0$ kabul edilirse $kl^0 m = km = 0^{n-j} 0^n$

olacaktır. Ayrıca bu terimin de bir W terimi olduğunu ve W 'nin tanımında bulunan şartları sağladığını söyleyebilmeliyiz.

Ancak, $n > n-j$ ($j > 0$ olduğu için) olduğuna göre ve ayrıca j teriminin tek veya çift olması ile ilgili herhangi bir bilgi olmadığına göre, j teriminin tek olması halinde km teriminin eleman sayısının tek sayı olduğu görülür. Bu durumda tanım itibariyle çift sayıda eleman içermesi gereken W teriminin tek sayıda terim içerebileceği de görülmüş olur. (Bu durumda bir çelişki

oluşuyor ve ispatını üzerine kurduğumuz üzere L dilinin RE olması kabulü çökmüş oluyor (contradiction)

Demek ki soruda verilen W terimi pompalama önsavına göre tek sayıda terim içerebilmektedir. Oysaki tanım itibariyle W bu şekilde bir terim asla olamaz.

Soruya bu şekilde cevap verdikten sonra, daha iyi anlaşılması açısından durumu şu şekilde izah edeyim:

Düzenli ifadelerde kullanabileceğimiz ve tekrar ifade eden tek işlemimiz (operator) ne yazık ki [kleene yıldızı \(kleene star\)](#) ismi verilen ve * sembolü ile gösterilen semboldür. Bu sembolün kaç kere tekrar edeceğini bilemeyiz.

Yukarıdaki soruda X için * kullanılması halinde X^R için yıldız kullanılamaz. İkisi için de kullanılırsa farklı miktarlarda tekrarlar olabilir. Bunun için iki veya daha fazla sayıdaki tekrar eden terimin birbirine eşit olması istenen durumlarda (ki bu örnek ve daha önce yazıda verdiğim örnekte de aynı durum söz konusu) RE olarak yazılamayacağını söyleyebiliriz çünkü kleen star kontrolsüz bir operatördür.

SORU 30: Pompalama Önsavı (Pumping Lemma)

Bilgisayar bilimlerinde dil tasarımı (language design, compiler design) konusunda önemli araçlardan birisidir. Bu önsava (lemma) göre şayet bir dil, bir herhangi bir gruba ([içerik bağımsız dil \(context free language\)](#) veya [düzenli ifadeler \(Regular expression\)](#) yada farklı bir dil grubu) dahil olarak kabul ediliyorsa, bu dil ne kadar pompalanırsa pompalansın yine aynı dil grubuna dahil olmalıdır.

Daha açık bir ifadeyle, bir dil sonsuz kümeyi kapsıyorsa (o dil kurallarıyla üretililecek sonuçların bir sınırı yoksa) bu dili ne kadar pompalarsak pompalayalım sonuçta yine aynı dil grubundan olmalıdır. Ve ancak bu sayede bu dilden üretililebilen bütün sonuçların aynı dil grubundan olduğu iddia edilebilir.

Yukarıdaki tanımlarda geçen pompalamak işlemi bir dil gramerinin bir kısmının şişirilmesi, arttırılması veya çoğaltılması olarak algılanabilir.

Pompalama önsavı genelde bir dilin iddia edildiği dil grubunda olmadığını ispatlamak için kullanılan bir çelişki ile ispat (proof by contradiction) yöntemidir.

SORU 31: İçerikten Bağımsız Gramer (context free grammar, CFG)

Bilgisayar bilimlerinde, dil tasarımı sırasında kullanılan bir gramer tipidir. Basitçe bir dilin kurallarını (dilbilgisini, grammar) tanımlamak için kullanılır.

Örneğin:

$S \rightarrow a$

Yukarıdaki dil tanımında bir büyük harfle gösterilen (S) bir de küçük harfle gösterilen (a) sembolleri bulunmaktadır. Bu satır, S devamlısının(nonterminal) a sonuncusuna(terminal) dönüştüğünü göstermektedir. Kısaca dildeki kuralları ifade etmek için büyük harfli semboller

devamlıları (nonterminal) ve küçük harfli semboller sonucuları (terminal) ifade etmekte, \rightarrow ok işareti ise, işaretin solundaki devamlının (nonterminal), sağındaki sembolle gösterilebileceğini ifade etmektedir.

Bir [dili içerikten bağımsız \(context-free\)](#) yapan, o dilin bir [belirsiz aşağı sürüklemeli otomat \(non deterministic pushdown automata\)](#) tarafından üretilebilir olmasıdır.

İçerikten bağımsız diller, programlama dilleri olarak sıklıkla kullanılmaktadır, bilinen çoğu programlama dili aslında birer içerikten bağımsız dil özelliğindedir ve bu dillerin kurallarının tanımlandığı gramerlerde içerikten bağımsız gramerlerdir (context free grammer).

Bu anlamda, [YACC](#) gibi programlama ortamlarında, bir dil tasarlamak ve içerikten bağımsız kurallar yazarak dili tanımlamak mümkündür.

CFG gösterimi ne yazık ki doğal diller (natural languages) için kullanılamaz. Ya da kullanılsa bile bir doğal dilin tamamını kapsayacak bir CFG gösterimi çıkarılamaz. Örneğin doğal dillerde [kelimebiliminin \(lexicology\)](#) bir parçası olarak sıkça kullanılan [uyum \(agreement\)](#) veya atıf (reference) kullanımları CFG ile gösterilemeyen özelliklerdir. Yani daha net bir ifadeyle CFG gösterimi için dildeki anlamların belirli olması gerekir. Çeşitli durumlarda belirsizlik içeren doğal diller için ise bu durum imkansızdır.

CFG tanımı

Temel olarak bir içerik bağımsız gramer dört özellik içermelidir. Bunlar sonlular (terminals), devamlılar (nonterminals), bağlantılar (relation) ve başlangıç sembolü (starting symbol) olarak sıralanabilir. Bir gramerin tanımı sırasında kullanılan bu kümeler aşağıdaki şekilde yazılabilir:

$$G = (V , \Sigma , R , S)$$

Bu gösterimdeki gramer (G) , V devamlıları, Σ sonluları, R bağlantıları, S ise başlangıç sembolünü göstermektedir.

Örneğin

$$S \rightarrow aSb \mid ab$$

şeklinde tanımlanan bir dilde:

$$G = (\{S\} , \{a,b\} , \{S \rightarrow aSb \mid ab\} , S)$$

gösterimi kullanılabilir.

SORU 32: İçerikten bağımsız dil (Context Free Language, CFL)

Bilgisayar bilimlerinde bir dilin tasarımı sırasında, içerik bağımsız bir gramer ile oluşturulması durumudur. Basitçe bir [aşağı sürüklemeli otomat \(push down automata\)](#) tarafından kabul edilen dil çeşididir. Bazı kaynaklarda bağlamdan bağımsız dil olarak da geçmektedir.

Örneğin çok meşhur $L = \{a^n b^n, n > 0\}$ dilini ele alalım. Bu dil örneğinin bu kadar meşhur olmasının ve önemli olmasının sebebi bir düzenli ifade (regular expression) ile yazılmasının imkansız oluşu ancak içerikten bağımsız dil ile yazılmasının mümkün olmasındandır.

Şimdi bu dilin aşağı sürüklemeli otomatını aşağıdaki şekilde çıkarabiliyoruz:

$$\begin{array}{lll} \delta(q_0, a, z) & = & (q_0, a) \\ \delta(q_0, a, a) & = & (q_0, a) \\ \delta(q_0, b, a) & = & (q_1, x) \\ \delta(q_1, b, a) & = & (q_1, x) \\ \delta(q_1, b, z) & = & (q_f, z) \end{array}$$

$$\delta(state_1, read, pop) = (state_2, push)$$

Yukarıdaki [PDA \(push down automaton\)](#) tasarımında dikkat edilirse iki durum (state) arasındaki geçişler ile yukarıdaki dili tasarlamak mümkündür. Bu sayede bu dilin bir içerik bağımsız dil olduğu söylenebilir.

Ayrıca yukarıdaki tanımda kullandığımız ” içerik bağımsız bir grammer ile oluşturulması durumu” ifadesini de açıklayarak buna da bir örnek verelim ve dilimizin ($L = \{a^n b^n, n > 0\}$) CFG (context free grammer, içerik bağımsız gramer) karşılığını aşağıda yazalım:

$$S \rightarrow aSb \mid ab$$

Yukarıdaki yazılışta, dilin sonucu ab veya aSb olarak çıkacaktır ancak S devamlısı (nonterminal) bitmek için bir sonuncuya (terminal) ihtiyaç duyacaktır bu değer de yine ab olacaktır.

Sonuçta yukarıdaki gramer ile istenilen uzunlukta sırasıyla a ve b lerden oluşsan dil tasarlanabilir ve üretilen bütün dillerde a ’nın sayısı ile b ’nin sayısı eşittir.

SORU 33: EBNF (Uzatılmış BNF, Extended Backus Normal Form)

Bilgisayar bilimlerinde dil tasarımı konusunda kullanılan [backus normal şeklinin \(backus normal form\)](#) özel bir halidir. Basitçe standart BNF’te yazılan kuralların birleştirilerek daha sade yazılmasını hedefler.

Bu durumu aşağıdaki örnek üzerinden görebiliriz:

Örneğin BNF olarak yazılan dilimize göre:

$$\langle EGER \rangle ::= \text{if}(\langle KOSUL \rangle) \mid \text{if}(\langle KOSUL \rangle) \text{ else}$$

şeklinde bir satırı bulunsun. Bu satırın anlamı dilimizde bir EGER döz dizilimi (syntax), if komutu ve parantez içinde bir koşuldan oluşabilir veya bu if ve parantez içerisindeki koşulu bir else komutu izleyebilir.

Yukarıdaki bu BNF yazılımını EBNF olarak aşağıdaki şekilde yazabiliriz:

$$\langle EGER \rangle ::= \text{if}(\langle KOSUL \rangle) [\text{else}]$$

Yukarıdaki bu yeni satırda dikkat edileceği üzere köşeli parantezler arasında bir else komutu bulunmaktadır. Bunun anlamı, EGER komutu “if(KOSUL)” olarak tanımlanır ve şayet istenirse bu komuta ilave olarak else komutu eklenebilir. Yani köşeli parantez içerisindeki komut isteğe bağlıdır.

Yukarıdaki bu yeni yazılım aslında sadece gösterimde bir farklılık oluşturmaktadır. Bunun dışında, EBNF’in kullanım alanı ve işlevi BNF ile aynıdır.

EBNF’in BNF’ten farklı olarak getirdiği ifade şekilleri aşağıda listelenmiştir:

İfade		Kullanımı
Tanımlama	definition	=
Üleştirme	concatenation	,
Bitirme	termination	;
Seçim (Veya)	separation	
Çift Tırnak	double quotation marks	” ... “
Tek Tırnak	single quotation marks	‘ ... ‘
İsteğe bağlı	option	[...]
Tekrarlı	repetition	{ ... }
Gruplama	grouping	(...)
Yorum	comment	(* ... *)
Özel dizilim	special sequence	? ... ?
Hariç	exception	-

Yukarıdaki tabloda ilk 6 ifade standart BNF gösteriminde de kullanılan ifadelerdir. Ancak son 6, koyu renkle yazılmış ifade EBNF için gelen yeni eklentilerdir.

Bu kullanımlardan isteğe bağlı (option) olma durumunu gördük. Şimdi diğer durumları inceleyelim:

Tekrarlı ifade ({ } işaretleri arasındaki ifadeler), [düzenli ifadelerde \(regular expression\)](#) kullanılan * işlemine benzetilebilir. Bu işlem basitçe bir bilginin istenildiği kadar tekrar edilmesi anlamına gelir.

Örneğin programlama dillerinin çoğunda kullanılan C tipi yorum’u düşünelim (comment). Bu yorumlarda istenilen kelimeler yazılabilir. Bu durumda yorum satırının tanımı aşağıdaki şekilde olabilir

<YORUM> ::= “/*” , { <harf> } , “*/”

<harf> ::= a | b | ... | z

Yukarıdaki EBNF tanımında a’dan z’ye kadar olan harfler, <harf> olarak tanımlanmış, ardından bu tanım <YORUM> içerisinde istenildiği kadar tekrarlanabilir anlamında { } işaretleri arasına yerleştirilmiştir.

EBNF’de ilave olarak dil tasarımcısının istediği yere kendi yorumlarını eklemesi de mümkündür. Buna göre tasarımcı (* *) işaretleri arasına istediği bilgiyi yazabilmektedir. Bu bilgi BNF işlemine tabi tutulmamaktadır.

EBNF'in belki de BNF'e göre en büyük eklentisi, hariç (fark) işlemidir. Yani bir bilgi grubundan başka bir bilgi grubunun çıkarılması durumudur.

Mesela bir [dizgi \(String\)](#) tanımı sırasında çift tırnaklar arasında herhangi bir yazı yazılabilir. Ancak bu yazının içerisinde çift tırnak bulunamaz çünkü bu durumda dizginin bittiğini belirten çift tırnak ile karışıklık oluşur. Bunu ifade için aşağıdaki EBNF gösterimini inceleyelim:

$$\langle \text{DIZGI} \rangle ::= \text{‘ ’ ‘}, \{ \text{?bütün karakterler?} - \text{‘ ’ ‘} \}, \text{‘ ’ ‘}$$

Yukarıdaki yeni kuralda, bütün karakterlerden, çift tırnak karakteri ayrı tutulmuştur. Yine yukarıdaki gösterimde ?bütün karakterler? tanımı, özel bir dizilim görüntüsüdür.

SORU 34: SableCC

SableCC 1998 yılında Étienne Gagnon tarafından bir yüksek lisans tezi olarak hazırlanmış ve dil geliştirmekte kullanılan, JAVA üzerinde çalışan, nesne yönelimli bir geliştirme ortamıdır.

Temel olarak SableCC üzerinde bir dil geliştirmek için aşağıdaki adımların takip edilmesi gerekir:

1. Dilde bulunacak olan kelimeler (lexicons) için bir kelime tanımı (lexical definition) yapılmalıdır.
2. Tanım dosyası hazırlandıktan sonra SableCC bu dosya ile birlikte çalıştırılır.
3. SableCC 2. adımda JAVA dilinde kod üretmiş olur. Bu aşamadan sonra anlambilimsel (semantic) analiz kısmı yazılabilir. JAVA dilinde anlambilimsel analiz, kod üretici kod ve kod iyileştirmesi (code optimizer) konularından bir veya bir kaçını yazdıktan sonra dil için gerekli son adıma geçilir.
4. Bu adımda 3. adımdaki kodumuz ile 2. adımdaki lexer ve parser (parçalayıcı) kodlar birleştirilir.
5. son olarak JAVA derleyicisi ile kod oluşturulup yeni dilimizde yazılmış bir kod için çalıştırılır.

SORU 35: Backus Normal Form (BNF)

Bilgisayar bilimlerinde genellikle bir dil tanımlamada ve bu dilin gramerini (Dil bilgisini) belirlemekte kullanılan gösterim biçimidir.

Basitçe dil bir dil tanımında başlayarak Terminal (sonuncu) ve Non-Terminal (Devamlı) terimler kullanarak tanımlanmaktadır.

Örneğin aşağıda basit bir örneği verilmiştir:

$$\langle \text{dil} \rangle ::= \langle \text{harf} \rangle | \langle \text{imla} \rangle$$
$$\langle \text{harf} \rangle ::= a|b|\dots|z$$
$$\langle \text{imla} \rangle ::= . \mid |, | ?$$

Yukarıda bir dil tanımı yapılmış ve bu dilde harf veya imlalar bulunduğu söylenmiştir. Buna göre dilimizdeki harfler a'dan z'ye kadar olup imla işaretleri de “.”, “?” işaretleridir. Bir ifadenin (nonterminal veya terminal) alabileceği alternatifler | işareti ile ayrılır. Yani harf ya “a”, ya “b” ya... şeklinde sayılan alternatiflerden birisi olabilir demektir. (aralarında [veya \(or\)](#) bağlantısı varmış gibi düşünülebilir)

Yukarıda da görüldüğü üzere BNF gösteriminde nonterminal'ler (devamlılar) $\langle \rangle$ işaretleri arasında belirtilmektedir.

Yukarıdaki dilimize aşağıdakine benzer bir ek yapılırsa:

$$\langle \text{dil} \rangle ::= \langle \text{harf} \rangle \langle \text{dil} \rangle \mid \langle \text{harf} \rangle \mid \langle \text{imla} \rangle$$
$$\langle \text{harf} \rangle ::= a|b|\dots|z$$
$$\langle \text{imla} \rangle ::= .|.|,|?$$

Yukarıda fark edilsin diye koyu gösterilen ek marifetiyle dilimizde bir döğü elde edilmiş ve bu sayede dilimizdeki harfler istenildiği kadar tekrar edilmiştir. Yani ilk örnekte dilimizde sadece tek bir harf veya tek bir imla işareti bulunabilirken yeni eklentimiz ile dilimizde istediğimiz kadar harf bulunabiliriz. Elbette bu durumun bir harf veya imla ile bitmesi gerekmektedir.

SORU 36: Tek Geçişli Çevirici (One Pass Assembler)

Tek geçişli bir [çeviricinin \(assembler\)](#) karşılaştığı en büyük problem çeviricinin kaynak koddaki (Assembly dilindeki koddaki) değişken ve etiketlerin kodun ilerleyen kısımlarında tanımlanma ihtimalidir. Bu durumda kodun geri dönerek daha sonradan tanımlanan bilgilerin önceki adreslere yazılması mümkün olmaz.

Tek geçişli çeviricilerde bu problemi çözmek için iki farklı yöntem kullanılabilir:

1. İleride kullanılacak olan etiketlerin (labels) önceden tanımlanmasıdır. Yani kodda sonradan tanımlanan bir etiket bırakılmaması durumudur.

2. Makine dilindeki kodun [hafızada \(RAM\)](#) üretilmesi yöntemi. Bu yöntemde göre [yükleyicinin \(loader\)](#) görevi de atlanarak kod doğrudan hafızada üretilmekte ve çevirici (assembler) daha sonradan değerlerini bulduğu etiketleri hafızadaki ilgili adreslere yazmaktadır.

SORU 37: Çevirici (Assembler)

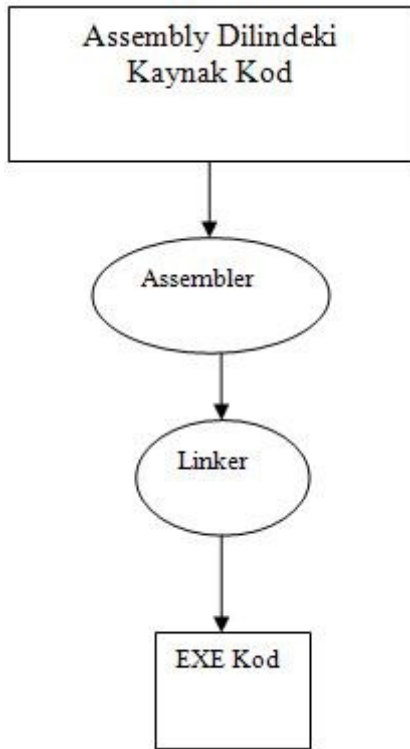
Bilgisayar bilimlerinde iki farklı kavram için assembler kelimesi kullanılmaktadır. Birincisi Assembly dili adı verilen ve makine diline (machine language) çok yakın düşük seviyeli (low level language) için kullanılan ve nesne kodunu (object code) makine koduna (machine code) çeviren dildir. İkincisi ise birleştirmek, monte etmek anlamında örneğin nesne yönelimli dillerde nesnelerin birleştirilmesi monte edilip büyük parçaların çıkarılması anlamında kullanılmaktadır.

Assembly dili, CPU üzerinde programcıya tanınmış olan yönergeleri (instructions) olarak bunları işlemcinin doğrudan çalıştırdığı makine kodları haline getirir. CPU üzerindeki bu yönergeler genelde üretici tarafından belirlenir ve tamamen teknoloji bağımlıdır. Diğer bir deyişle aynı kod farklı yapıdaki işlemciler üzerinde çalışmaz. Hatta çoğu zaman aynı mimarinin farklı nesilleri arasında bile sorun yaşanır.

Assembly dili macro destekleyen ve desteklemeyen olarak ikiye ayrılır. Tarihi gelişim sürecinde ihtiyaç üzerine dilde macro yazmak ve yazılan bu macroları tekrar tekrar kullanmak imkanı doğmuştur. Bu tip assembly dilini makine diline çeviren çeviricilere de macroassembler adı verilir.

Assembly dilleri temel olarak fonksiyon(function) veya prosedür (procedure) desteği barındırmazlar. Hatta döngü ve koşul operatörleri de yok denilebilir. Bunun yerine programın içerisinde istenen bir satıra (adrese) gidilmesini sağlayan GOTO komutu kullanılır.

Yapısal programlama (structured programming) dokusuna ters olan bu yaklaşım dilin performans kaybı ve düşük seviye olmasından kaynaklanmaktadır.



Yukarıdaki şekilde assembly kaynak kodunda verilen bir girişin assembler'dan geçerek [bağlayıcı \(linker\)](#) marifetiyle çalışabilir (Exe code) haline gelişini tasvir edilmiştir.

SORU 38: NFA'den DFA'e çevirim (Converting NFA to DFA)

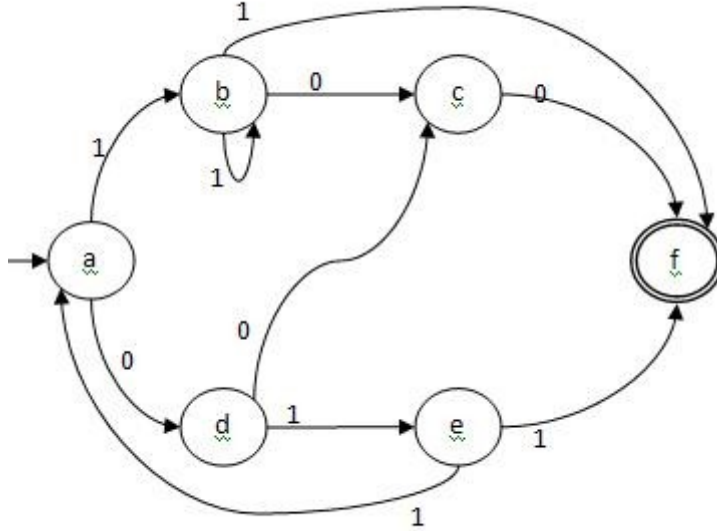
Bu yazıda belirsiz sonlu otomattan(NFA) Belirli sonlu otomata (gerekirici sonlu otomat, nedensel sonlu otomat, deterministic finite automata) dönüştürmenin nasıl yapıldığı anlatılmaktadır.

Basitçe bir iki adımlık işlemler izlenerek bu dönüşüm gerçekleştirilebilir:

1. Öncelikle gerekircilik (determinism) açısından birbiri ile özdeş olan kümler çıkarılmalıdır (subset construction)
2. Bu kümelerin diğer kümeler ile olan ilişkisi çıkarılmalıdır.

Sonuçta elde eden kümelerin birer durum göstermesi ve bu durumlar arası bağlantıların yukarıdaki ikinci adımda çıkarılmış ilişkisi belirli bir sonlu otomat (DFA) üretmiş olur.

Bu işlemi aşağıdaki örnekte inceleyelim:



Yukarıda belirsiz bir sonlu otomaton gösterilmiştir. Bu yapıyı belirli sonlu otomata adım adım çevirelim.

Öncelikle lambda (ϵ, λ) ile ulaşılabilen durumları sıralayalım :

$\{a\}$: A , tek elemanlı bu küme başlangıç elemanını içerir, bunun dışında yukarıdaki otomatonda herhangi bir elemana lambda ile ulaşılması söz konusu değildir.

Şimdi bir önceki adımda ürettiğimi A kümesinden gidilebilecek kelime ihtimallerini yazalım

A1 (A kümesinden 1 değeri ile ulaşılabilir olan durumlar) : $\{b\}$: B (bu kümeye B ismini verelim)

A0 (A kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{d\}$: C (bu kümeye C ismini verelim)

B1 (B kümesinden 1 değeri ile ulaşılabilir olan durumlar): $\{b,f\}$: D (bu kümeye D ismini verelim)

B0 (B kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{c\}$: E (bu kümeye E ismini verelim)

C1 (C kümesinden 1 değeri ile ulaşılabilir olan durumlar): $\{e\}$: F (bu kümeye F ismini verelim)

C0 (C kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{c\}$: E (bu kümeye zaten E ismini vermiştik)

D0 (D kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{c\}$: E (bu kümeye zaten E ismini vermiştik)

D1 (D kümesinden 1 değeri ile ulaşılabilir olan durumlar): $\{e\}$: F (bu kümeye zaten F ismini vermiştik)

E0 (E kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{f\}$: G (bu kümeye G ismini verelim)

E1 bu ihtimal boş kümedir dolayısıyla işlem yapılmıyor

F0 bu ihtimal boş kümedir dolayısıyla işlem yapılmıyor

F1 (F kümesinden 1 değeri ile ulaşılabilir olan durumlar): $\{a,f\}$: H (bu kümeye H ismini verelim)

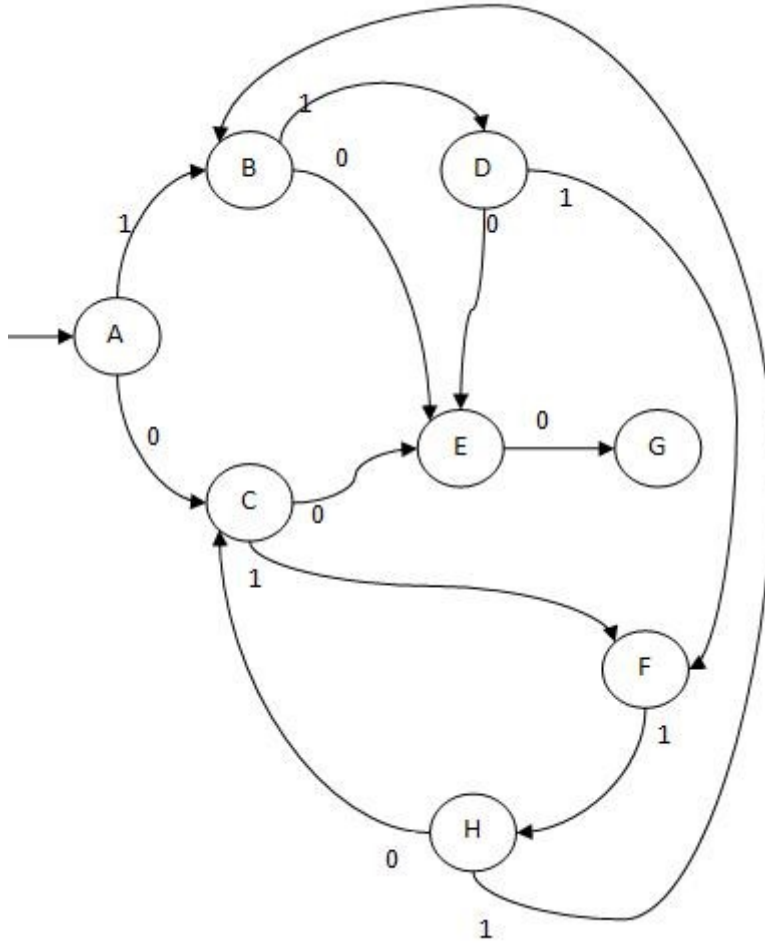
G0 bu ihtimal boş kümedir dolayısıyla işlem yapılmıyor

G1 bu ihtimal boş kümedir dolayısıyla işlem yapılmıyor

H0 (H kümesinden 0 değeri ile ulaşılabilir olan durumlar): $\{d\}$: C (bu kümeye zaten C ismini vermiştik)

H1 (H kümesinden 1 değeri ile ulaşılabilir olan durumlar): $\{b\}$: B (bu kümeye zaten B ismini vermiştik)

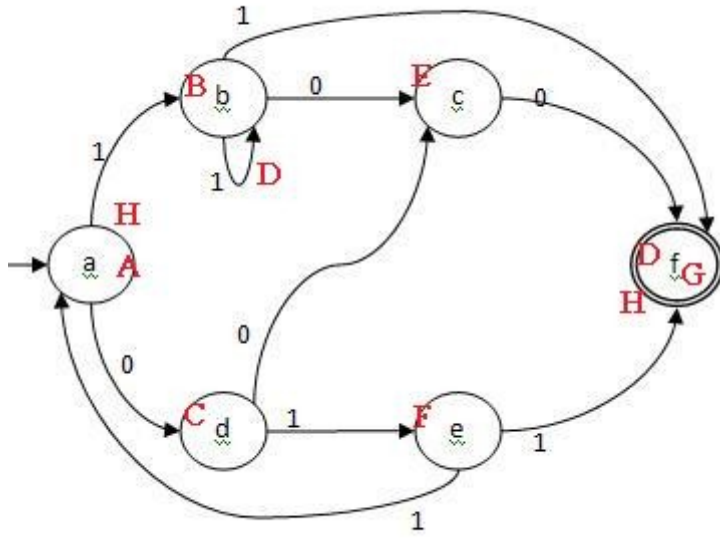
Yukarıda bulunan bütün alt kümelerin inşası tamamlandı ve bu kümelerin diğer kümelere gidiş koşulları belirlendi şimdi bu kümeler arasında yukarıda da listelenmiş olan geçişleri çizebiliriz:



Yukarıdaki yeni otomatımızın belirli (nedensel, gerekirci) olduğunu söyleyebiliriz çünkü her durumda gelen kelimeye göre gidilebilecek tek ihtimal bulunmaktadır.

Kümelerin NFA üzerinde gösterimi

Bu kısmı Tarık Bey'in sorusu üzerine ekliyorum. Yukarıdaki kümeleri graf üzerinde kırmızı büyük harf ile gösterdim. Bu sayede kümelerin takibi sanırım daha kolay olacaktır.



Yukarıdaki şekilde şaşılması gereken bir husus, aynı düğüme birden fazla küme ismi geliyor olmasıdır. Bu normaldir ve doğrudur ancak kümelerin birden fazla düğüm içerebildiğine dikkat etmek gerekir. Örneğin f düğümü, D,H ve G kümelerinin üyesidir. Ancak tek başına sadece G kümesine üyedir. H kümesine a düğümü ile ve D kümesine b düğümü ile üyedir.

Örnek 1

Bir düzenli ifade (regular expression) olarak aşağıdaki örnek verilmiş olsun. Bu örneği sonlu otomat (finite state automata) olarak göstermeye çalışalım:

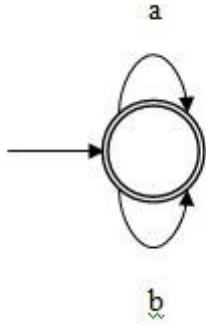
a^*b^*

Yukarıdaki düzenli ifadeyi tanımak için öncelikle bu ifadeden üretilebilecek örnekleri görelim:

Üretilebilecek en kısa dizgi (String): ϵ (yani boş küme olacaktır, bu bazı kitaplarda λ sembolü ile de gösterilir). Çünkü kleene yıldızının üreteceği sonuçlar arasında boş küme de bulunmaktadır.

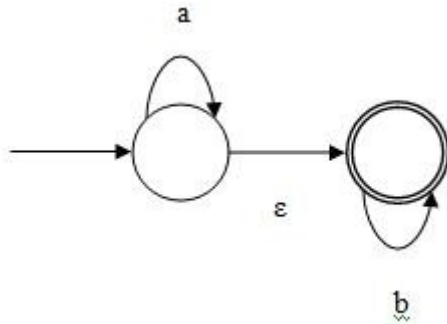
İkinci dizgimiz (String) : ab olacaktır ve bu üretme işlemi aab , abb, aabb, aaab, abbb şeklinde devam edecektir.

Yukarıdaki bu düzenli ifadeyi göstermek için aşağıdaki sonlu otomatı çizebiliriz (genelde sık yapılan bir hata olduğu için aşağıdaki **hatalı örneği** çizmek istiyorum):



Yukarıdaki şekilde anlatılmak istenen tek bir durum (State) oluşudur ve bu durum hem başlangıç (okla belirtilmiştir) hem de bitiş durumu (çift çizgi ile belirtilmiştir) olduğudur. Bu durum üzerinde hem a hem de b harfleri ile istenildiği kadar dönülebilir.

Yukarıdaki hata, yukarıdaki FSM'in örneğin ba gibi bir kelimeyi de kabul etmesidir. Oysaki düzenli ifademiz buna izin vermemektedir. **Doğrusu** aşağıdaki şekilde olmalıdır.



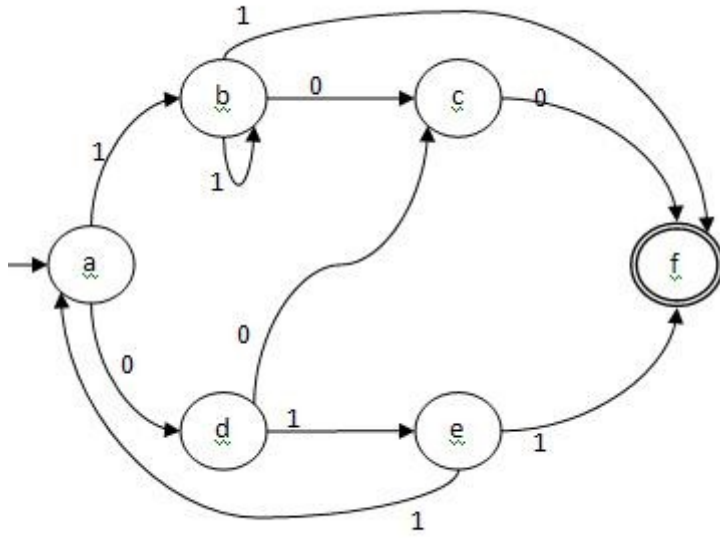
Yukarıdaki çizim doğru çizimdir. Görüldüğü üzere otomatımız hem boş kelimeyi hem de a ve b ile üretilebilecek (Sırası değişmeden) bütün kelimeleri desteklemektedir.

SORU 39: Belirsiz Sonlu Otomat (Nondeterministic Finite Automat, NFA)

DFA (deterministic finite automat) belirli sonlu otomatların (özdevinirlerin) tersine her durumdan gidişin karışık olduğu ve her durum için bir sonraki kelimedeye nereye gidileceğinin belirli olmadığı otomatlardır.

Basitçe DFA kurallarına uymayan bütün otomatlar NFA olarak adlandırılabilir.

Aşağıda bir örnek üzerinde durumu inceleyelim:



yukarıdaki örnekte belirsiz durumlar bulunmaktadır. Örneğin b durumunda iken 1 kelimesi ile hem f durumuna hem de yine b durumuna gitmek mümkündür. Veya e durumundaki ike 1 kelimesi ile f veya a durumlarına geçmek de mümkündür. Bu belirsizlik yüzünden NFA'nın bilgisayarlar tarafından algılanması ve kullanılması zor olmaktadır. Ancak insanlar tarafından NFA daha kolay anlaşılır ve kullanılır yapılardır.

Örnek 1 (Vildan hn.'ın isteği üzerine bu iki örneği ekliyorum umarım yardımcı olur)

Bir düzenli ifade (regular expression) olarak aşağıdaki örnek verilmiş olsun. Bu örneği sonlu otomat (finite state automata, Sonlu Özdevinirler) olarak göstermeye çalışalım:

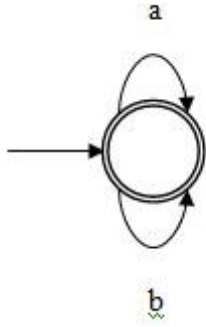
a^*b^*

Yukarıdaki düzenli ifadeyi tanımak için öncelikle bu ifadeden üretilebilecek örnekleri görelim:

Üretilebilecek en kısa dizgi (String): ϵ (yani boş küme olacaktır, bu bazı kitaplarda λ sembolü ile de gösterilir). Çünkü kleene yıldızının üreteceği sonuçlar arasında boş küme de bulunmaktadır.

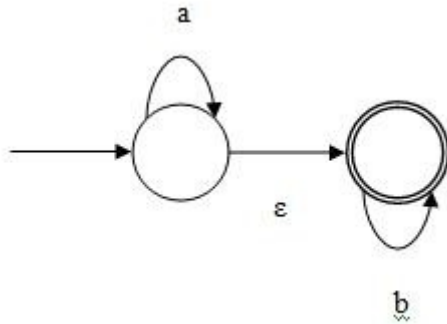
İkinci dizgimiz (String) : ab olacaktır ve bu üretme işlemi aab , abb, aabb, aaab, abbb şeklinde devam edecektir.

Yukarıdaki bu düzenli ifadeyi göstermek için aşağıdaki sonlu otomatı çizebiliriz (genelde sık yapılan bir hata olduğu için aşağıdaki **hatalı örneği** çizmek istiyorum):



Yukarıdaki şekilde anlatılmak istenen tek bir durum (State) oluşudur ve bu durum hem başlangıç (okla belirtilmiştir) hem de bitiş durumu (çift çizgi ile belirtilmiştir) olduğudur. Bu durum üzerinde hem a hem de b harfleri ile istenildiği kadar dönülebilir.

Yukarıdaki hata, yukarıdaki FSM'in örneğin ba gibi bir kelimeyi de kabul etmesidir. Oysaki düzenli ifademiz buna izin vermemektedir. **Doğrusu** aşağıdaki şekilde olmalıdır.



Yukarıdaki çizim doğru çizimdir. Görüldüğü üzere otomatımız hem boş kelimeyi hem de a ve b ile üretilebilecek (Sırası değişmeden) bütün kelimeleri desteklemektedir.

Örnek 2:

Biraz daha karmaşık kabul edebileceğimiz bir örneği çözmeye çalışalım:

$$(a+b)^*ab+c$$

Yukarıdaki ifadeyi analiz edip anlamaya çalışalım. Dilin üreteceği en küçük dizgiden (String) başlayalım ve uzatarak ihtimalleri deneyelim:

c

ab

aab

abab

aaab

bbab

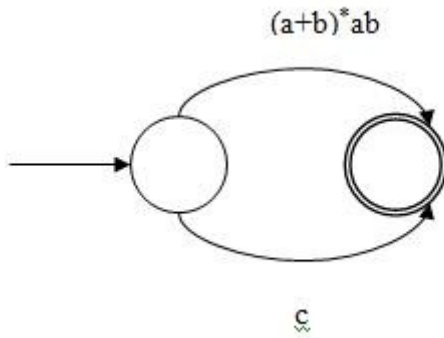
şeklinde giden üretim listemiz bulunuyor. Yukarıdaki düzenli ifadeye bakıldığında görüleceği üzere + (ikinci + sembolü) ifadeyi ikiye bölmüştür. Yani ifademizi:

$(a+b)^*ab$

veya

c

ifadelerinden birisini üretecek gibi düşünebiliriz. Bu durumu FSM ile gösterecek olursak:



şeklinde düşünülebilir. Elbette yukarıdaki çizim tam bir FSM değildir. Yani kollardan üstteki kolu açarak çizmemiz gerekir ancak fikir vermesi açısından + sembolleri (veya anlamındaki semboller) iki kol olarak düşünülebilir.

Yukarıdaki bu ifadeyi daha açık halde yazacak olursak ve üst kolu analiz edersek

$(a+b)^*ab$

ifadesini de ikiye bölmek mümkündür:

$(a+b)^*$

ve

ab

olarak bölünebilir.

Yukarıda bu üç ifade üleştirilmiştir (concatenate) yani

ABC

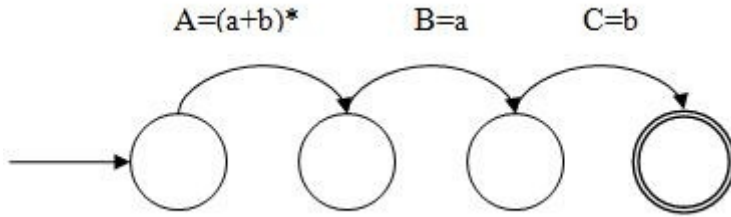
üleştirilmesi olarak düşünülürse

$$A = (a+b)^*$$

$$B = a$$

$$C = b$$

olarak düşünülebilir. Bu durumdaki üleştirme işlemleri aşağıdaki şekilde çizilebilir:

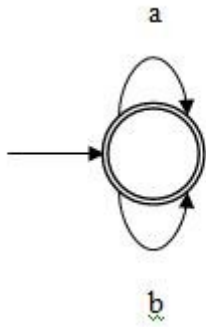


Yukarıdaki bu yeni çizimde üleştirme işlemi görülmüştür.

Yukarıdaki üleştirme işleminde de $(a+b)^*$ ifadesi kapalı olarak bırakılmıştır. Son olarak bu ifadeyi nasıl göstereceğimizi inceleyelim:

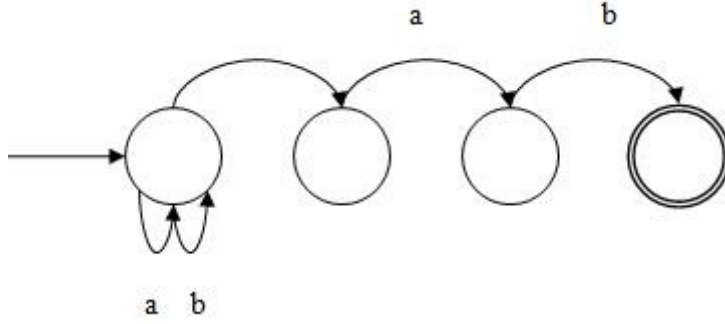
$$(a+b)^*$$

ifadesi de görüldüğü üzere aslında $a+b$ ifadesinin kleene yıldızının (kleene star) uygulanmış halidir. Dolayısıyla aşağıdaki şekilde gösterilebilir:

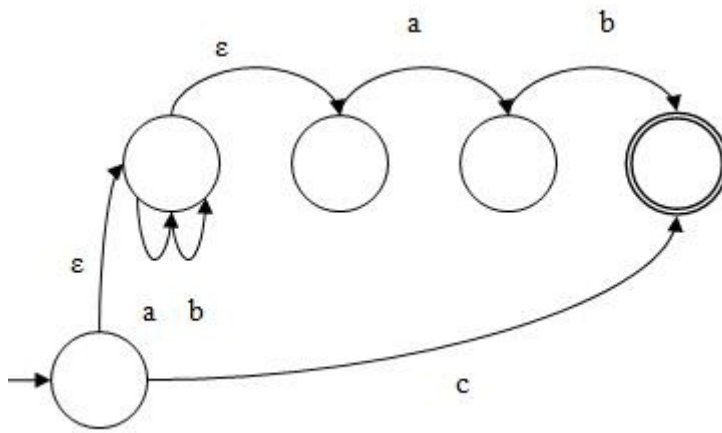


yukarıdaki şekilde görüleceği üzere bütün a ve b ile üretilebilecek (ve boş küme dahil) ihtimaller kapsamaktadır.

Son aşamada bütün bu alt FSM çizimlerimizi birleştiriyoruz. Önce sondan başalayarak son iki çizimi birleştirelim:



Yukarıdaki ifade $(a+b)^*ab$ düzenli ifadesinin sonlu otomatı olmaktadır. Buna ilk otomatımızı da eklersek sonucu buluruz:



Yukarıda FSM'ini bulmak istediğimiz düzenli ifade olan $(a+b)^*ab+c$ ifadesinin son hali görülmektedir.

Yukarıdaki soru çözümü sırasında izlenen yöntem parçala fethet yöntemidir. Problem cevabını bildiğimiz basit parçalara bölünmüş ve sonra birleştirilmiştir. Problemde özel olarak karşılaşılabilecek en temel 3 durum olan veya (+), üleştirme (concatenation) ve kleene yıldızı durumlarını içeren bir örnek seçtim. Düzenli ifadeleri çevirirken istisnalar hariç bu üç duruma indirgeyerek çizim yapabilirsiniz.

SORU 40: Belirli Sonlu Otomat (Deterministic Finite Automat, DFA)

Sonlu otomatların özel bir halidir. Bu özel hal aşağıdaki 3 durumu içermelidir:

1. Her durumdan (State) gidilecek koşulun tek bir durum göstermesi. Yani bir durumda başka duruma geçerken bir kelime ile sadece bir duruma gidilebilmesi
2. Tek bitiş durumunun bulunması (final state)
3. Lambda (veya epsilon) kelimesinin durumlar arası geçişte yer almaması

Örneğin aşağıdaki sonlu otomatı (aynı zamanda sonlu durum makinesi (finite state machine) de denilmektedir) inceleyelim:

Yukardaki otomatta, başlangıç soldan gelen ok ile gösterilmiş q1 ve bitiş çift halka içerisine alınmış durum (düğüm, node, state) ile gösterilmiş olan q2'dir.

Yukarıdaki otomatın özellikleri kontrol edildiğinde belirli sonlu otomat (deterministic finite automaton) olmasını gerektiren koşullar şöyledir:

1. adımdaki şartı sağlar çünkü herhangi bir durumda diğerine giderken belirsizlik söz konusu değildir. Örneğin q1 durumundayken 0 gelince tek bir yere (yine q1) ve 1 gelince tek bir yere (q2) gidilmektedir. Şayet herhangi birisi için birden fazla alternatif bulunsaydı bu durumda [belirsiz sonlu otomat \(nondeterministic finite automat, nfa\)](#) denilebilirdi.

2. adımdaki şartı da sağlamaktadır çünkü otomatta hiç lambda geçişi (boş geçiş) yoktur. bütün geçişler bir değerle (kelime) yapılmaktadır.

3. adımı da sağlamaktadır çünkü tek bitiş durumu söz konusudur (bu bitiş de q2 durumudur)

dolayısıyla yukarıdaki otomatın belirli (Deterministic) olduğu söylenebilir

Sonlu Durum Makinelerinin Gösterimi

Bazı kaynaklarda, sonlu durum makinelerinin ifadesi için, yukarıdaki yazıda yer alan görsel gösterimin yerine küme gösterimi veya tablo gösterimi de kullanılmaktadır.

Buna göre durum makinesinin geçişlerini ve durumlarını gösteren bir şekil tablo hazırlanabilir. Yukarıdaki makine için aşağıdaki tablo aynı makineyi tutmaktadır:

	q1	q2	q3
q1	0	1	
q2	1	0	
q3	1,0		

Yukarıdaki gösterime göre geçişlerin tutulduğu bir matris oluşmuştur. Bu matris, yönlü bir şekil (graph) için kullanılmasından mütevellit asimetrik yapıdadır.

Aynı sonlu durum makinesi aşağıdaki tablo ile de tutulabilir:

	0	1
q1	q1	q2

q2 q3 q2
q3 q2 q2

Yukarıdaki yeni tabloda, kolon başlıkları gelen parametreyi (geçiş değerini) ve tablo içeriği de geçilecek olan durumu(state) tutmaktadır.

Bu tabloda da bir hücrede birden fazla durum(state) bulunması halinde NFA, her hücrede tek durum (state) bulunması halinde DFA yorumu yapılabilir.

Yukarıdaki tablo gösterimlerine ilave olarak küme halinde tutulabilir. En çok kullanılan yazım aşağıdaki şekildedir:

$(\Sigma, S, S_0, \delta, F)$

Buna göre Σ sembolü bir [alfabeyi \(alphabet\)](#) ifade etmektedir. Yani dilimizde kabul edilen girdilerin her birisini gösteren küme. Örneğin yazı boyunca kullandığımız DFA için $\{0,1\}$ kümesidir denilebilir.

S kümesi, otomattaki durumların kümesidir. Yani örnek otomatımızda $\{q_1, q_2, q_3\}$ durumları bulunmaktadır

S_0 ise başlangıç durumudur (state). Örnek otomatımızda q_1 başlangıç durumudur.

δ kümesi ise geçişlerin tutulduğu kümedir. Örnek otomatımızda geçişler için şöyle bir küme yazabiliriz $\delta = \{ q_1 \rightarrow q_1, 0 ; q_1 \rightarrow q_2, 1 ; q_2 \rightarrow q_3, 0 ; q_2 \rightarrow q_2, 1 ; q_3 \rightarrow q_2, 0 ; q_3 \rightarrow q_2, 1 \}$ olarak yazılabilir.

F ise bitiş durumunu gösterir. Yani bir gidi (input) bu durumda bitiriyorsa kabul edilir (accept). Diğer durumlarda bitmesi veya gidilemeyen bir devam olması durumunda ise red edilir (reject). Örnek otomatımızda q_2 bitiş durumudur.

Kısaca yukarıdaki özellikler tek bir kümeler kümesinde toplanırsa, aşağıdaki gibi bir gösterim ortaya çıkar:

$\{ \{0,1\} , \{q_1, q_2, q_3\} , q_1 , \{ q_1 \rightarrow q_1, 0 ; q_1 \rightarrow q_2, 1 ; q_2 \rightarrow q_3, 0 ; q_2 \rightarrow q_2, 1 ; q_3 \rightarrow q_2, 0 ; q_3 \rightarrow q_2, 1 \} , q_2 \}$

Yukarıdaki bu küme en başta çizilen durum makinesini göstermek için yeterlidir.

SORU 41: Dinamik Bağlantı Kütüphaneleri (Dynamic Link Library (.dll))

Microsoft tarafından windows işletim sistemi üzerinde kullanıma açılan ve çalışma sırasında bağlanmaya izin verilen kütüphane yaklaşımıdır. Basitçe Linux ortamlarındaki .o (object file (nesne dosyası)) benzetilebilir. Bu dosyaların amacı birden fazla program tarafından kullanılan kütüphaneleri içermeleri ve her programın gerekli oldukça ilgili kütüphaneden dosyayı okumasıdır.

Windows öncesi microsoft işletim sisteminde (windows 3.x ve DOS gibi) kullanılan yaklaşımda program tek bir dosyadan oluşmaktaydı. Dolayısıyla programın kullandığı her

kütüphane o programa özgü olarak bulunduruluyordu. İşletim sisteminin programın çalıştırılması sırasında herhangi bir bağlama yapması söz konusu değildi.

Windows 95 ile birlikte 32bit sisteme geçen windows, Unix ve dolayısıyla Linux dünyasında da rahatlıkla kullanılabilen object code kavramı windows dünyasına kazandırılmış oldu. Buradaki amaç ortak kullanılan fonksiyonları bir dosyada bulundurmak ve her programın çalıştırılması sırasında önce [bağlayıcı \(linker\)](#) tarafından bu dosyaların bağlanması ve ardından programın [yükleyici \(loader\)](#) tarafından hafızaya yüklenerek çalıştırılmasıydı.

Bu sayede aynı ortak fonksiyonu kullanan programların herbirisi için kod tekrarı olmayacağı gibi güncellemeler de tek elden takip edilebilecekti. Programcılığın modüler yaklaşımının bir ürünü olan .dll dosyaları günümüzde windows işletim sisteminde yaygın olarak kullanılmaktadır. Bu dosyalar derlenmiş (compiled) kod olduğu için gerekli görülmesi durumunda şifrelenebilmekte ve orjinal kodu koruma altında tutabilmektedir.

Gelişen web teknolojileri ile birlikte .dll dosyalarını windowsun web sunucusu (web server) olan IIS (internet information server) ile de uyumlu hale getiren microsoft, şu anda geliştirilen web projelerini .dll olarak sunucularda barındırıp internet kullanıcılarının erişimine açmaktadır.

.dll dosyalarının bir diğer avantajı da [hafızaya](#) bir kere yüklenen dosyaların birden fazla program tarafından paylaşılması bu sayede hafızanın verimli kullanılmasıdır.

SORU 42: XML (extensible markup language , genişletilebilir işaretleme dili)

XML dilinin çıkış amacı makinelerin birbiri ile konuşurken kategorize olmuş bir dil (veya protokol) üzerinden konuşmalarını sağlamaktır. Aslında XML'in çıktığı yıllara bakıldığında verilerin çok çeşitli şekillerde bir standarda uymaksızın saklandığını ve işlendiğini görmek mümkündür. Gelişen İnternet teknolojisinin de etkisiyle birden fazla bilgisayarın (veya programın) birbiri ile bir standart üzerinde anlaşarak konuşmaları ihtiyacı doğmuştur. İşte tam da bu noktada W3C tarafından ortaya atılan, verileri programcını istediği gibi kategorize ederek tutabileceği bir standart olan XML doğmuş ve hızla kullanımı artmıştır.

XML verileri tasnif etmek ve bu tasnif edilmiş verileri arzu edildiği gibi okuyabilmenin dışında ilave özellikleri ile veri standartları üzerinde denetimler de yapabilmektedir. Örneğin istenilen standartlara uymayan bir XML dökümanını [DTD](#), XSD gibi ilave yapılar ile tespit etmek mümkündür.

XML platform bağımsız bir standart olarak çıkmıştır ve amacı bütün platformları ortak bir yapı altında toplayarak iletişim güçlüğüne asgari düzeye indirmektir. Bu durum doğal olarak her platformda XML için bir ilave yazılım ihtiyacı doğurmuştur. Bu konuda en önemli kütüphane standartları DOM ve SAX kütüphaneleridir. Bu kütüphaneler hemen bütün modern diller tarafından dile eklenmiş ve programcının kullanımına sunulmuştur.

SORU 43: Yorumlayıcı (Interpreter)

Bilgisayar dünyasında yorumlayıcı terimi bir programı veya bir komutu çalıştırmaya yarayan programlar için kullanılır. Genellikle derleyiciler ile karışan ve çoğu zaman aynı görevi icra eden yorumlayıcılar da derleyiciler gibi kodu bir dilden başka bir dile çevirme işlemini yerine getirirler. Basitçe görevleri ve tipleri aşağıdaki şekilde listelenebilir:

1. Kaynak kodu çalıştırmak
2. Bir kaynak kodu çalıştırılabilir farklı bir kod haline tercüme etmek
3. Daha önceden hazırlanmış olan (çoğu zaman önceden derlenmiş bir koddur) kodları yeri geldiğinde çalıştırmak

Perl, Python diller 2. tip dillere örnekken [JAVA](#) sanal makinesi ve Pascal dilleri 3. seviye dillere örnek olarak gösterilebilir. Çünkü java ve pascal'da daha önceden hazırlanmış olan program parçaları çalışma zamanında (execution time) [bağlanmaktadır](#).

Derleyiciler (compiler) ile Yorumlayıcılar (Interpreter) arasındaki farklar:
Basitçe, bir kaynak kodu hedef koda çevirdikten sonra çalıştıran ve dolayısıyla koddaki hataları yakalama işlemini ve kodun iyileştirilmesini daha kod çalıştırmadan yapan çeviricilere derleyici, kodu satır satır veya bloklar halinde çalıştırıp sırası gelmeyen satırları hiç çalıştırmayan bu satırlardaki hataları hiçbir zaman göremeyen ve kodun bütününe ait iyileştirmeleri yapamayan çeviricilere de yorumlayıcı (interpreter) adı verilmektedir.

Genel kanının tersine bir dilin derleyici veya yorumlayıcı özelliği yoktur. Yani C dili için sadece derleyicisi bulunan bir dildir demek yanlış olur. Bu durum bütün diller için geçerlidir. Her dil için bir derleyici veya yorumlayıcı tasarlanabilir. Ama daha genel bir bakışla, her dilin aslında yorumlayıcı (interpreter) yapısında bir çalışması olduğunu söylemek yanlış olmaz. Sonuçta bilgisayarın işlemcisinde anlık olarak tek bir işlem yapılabilmektedir ve çalışması istenen kod, işlemciye sırayla verilecek ve satır satır çalıştırılacaktır.

Genelde bir ortam yazılan dilin çalıştırılmasına kadar geçen sürede ya bir derleyici yada bir yorumlayıcı kullanılmaktadır. Gelişmekte olan teknolojiyle iki programı birden kullanan diller de türemiştir. Örneğin [JAVA](#) dilinde kod önce derlenerek byte code adı verilen ve sadece java sanal makinelerinde (java virtual machine) çalıştırılabilen bir kod üretilmektedir. Bu üretilen ara kod daha sonra java sanal makinasında bir yorumlayıcı yapısına uygun olarak çalıştırılmaktadır.

SORU 44: Bağlayıcı (linker)

Bir [derleyici](#) tarafından üretilmiş olan kodları bağlayarak işletim sisteminin çalıştırabileceği tek bir kod üreten programdır.

Günümüzde hızla gelişen programlama ihtiyaçları sonucunda programlamada modüler yaklaşıma geçilmiştir. Buna göre büyük bir yazılım küçük alt parçalara bölünmekte ve her parça ayrı ayrı işlenerek büyük program elde edilmektedir. Yapısal programlamanın da çıkış sebeplerinden birisi olan bu yaklaşıma göre dillerde fonksiyon desteği gelmiş ve değişik parametrelere göre aynı kodun farklı sonuçlar üretmesi sağlanmıştır. Daha sonradan gelişen nesne yönelimli programlama bu konuda bir sonraki nesil olarak kabul edilebilir. Nesne yönelimli programlamada, programlar nesnelere bölünerek farklı bir yaklaşım izlenmiştir.

Bu yaklaşımların [derleyici](#)lere yansması da uzun sürmemiş, daha yapısal programlamanın ilk geliştiği günlerde [derleyici](#)ler de farklı kütüphaneler ve bu kütüphaneleri birleştirmeye yarayan harici programlar kullanmaya başlamışlardır.

Kodun birden fazla parçaya bölünmesi ve her parçanın ayrı ayrı üretilmesi durumunda bu parçaların birleştirilmesi ve tek bir program halinde üretilmesinden sorumlu olan programlara bağlayıcı (linker) adı verilmektedir.

SORU 45: Derleyici (compiler)

Basitçe bir dilde yazılmış olan kodu (kaynak kodu yada source code) istenilen başka bir kod haline dönüştüren programdır. Genelde üretilen bu kod ortama göre çalıştırılabilir kod (executable code) olarak üretilmektedir. Ancak bir derleyicinin daha doğru tanımı bir dildeki kodu başka dile çeviren program olarak yapılabilir. Örneğin C dilinde yazılan bir programı PASCAL diline çeviren programlara derleyici adı verilebilir. Derleyicinin diğer bir tanımı ise daha üst seviye bir dilden daha alt seviyeli bir dile tercüme olarak kabul edilebilir. Buna göre örneğin C dilinden Assembly veya makine dili gibi daha alt dillere tercüme ile derleyici kavramı daha da sınırlandırılmış olarak kabul edilebilir. Derleyiciler günümüzde daha çok bir dilde yazılmış koddan, işletim sistemi ve donanım bağımlı kodların üretilmesinde kullanılmaktadırlar. Bu üretim sırasında ya doğrudan işletim sisteminin anlayacağı ve çalıştıracağı kodları üretirler ya da işletim sisteminde bulunan veya yine dil bağımlı olarak çalışan [bağlayıcı \(linker\)](#) programların anlayacağı ara kodları üretirler.

Derleyiciler bu kod üretmesi sırasında, üretilen kodun en verimli şekilde üretilmesi için kod iyileştirmesi (optimisation) da yapmaktadırlar. Yani hedef dildeki çalışma süresi ve [hafıza](#) ihtiyacı en az olan kodu üretmek bir derleyicinin daha başarılı olma kriterlerinden birisidir.

Aynı zamanda kaynak kodda (source code) bulunan hataların yakalanması bu hataların programcıya bildirilmesi de derleyicilerin diğer görevlerinden birisidir.

Derleyiciler (compiler) ile Yorumlayıcılar (Interpreter) arasındaki farklar:

Basitçe, bir kaynak kodu hedef koda çevirdikten sonra çalıştıran ve dolayısıyla koddaki hataları yakalama işlemini ve kodun iyileştirilmesini daha kod çalıştırmadan yapan çeviricilere derleyici, kodu satır satır veya bloklar halinde çalıştırıp sırası gelmeyen satırları hiç çalıştırmayan bu satırlardaki hataları hiçbir zaman göremeyen ve kodun bütününe ait iyileştirmeleri yapamayan çeviricilere de yorumlayıcı (interpreter) adı verilmektedir.

Genel kanının tersine bir dilin derleyici veya yorumlayıcı özelliği yoktur. Yani C dili için sadece derleyicisi bulunan bir dildir demek yanlış olur. Bu durum bütün diller için geçerlidir. Her dil için bir derleyici veya yorumlayıcı tasarlanabilir. Ama daha genel bir bakışla, her dilin aslında yorumlayıcı (interpreter) yapısında bir çalışması olduğunu söylemek yanlış olmaz. Sonuçta bilgisayarın işlemcisinde anlık olarak tek bir işlem yapılabilir ve çalışması istenen kod, işlemciye sırayla verilecek ve satır satır çalıştırılacaktır.

Genelde bir ortam yazılan dilin çalıştırılmasına kadar geçen sürede ya bir derleyici yada bir yorumlayıcı kullanılmaktadır. Gelişmekte olan teknolojiyle iki programı birden kullanan diller de türemiştir. Örneğin [JAVA dilinde](#) kod önce derlenerek byte code adı verilen ve sadece java sanal makinelerinde (java virtual machine) çalıştırılabilen bir kod üretilmektedir. Bu üretilen ara kod daha sonra java sanal makinasında bir yorumlayıcı yapısına uygun olarak çalıştırılmaktadır.

Tek geçişli (one pass) ve çok geçişli (multi pass) derleyiciler:

Bu başlıkta geçiş ile kastedilen kavram, bir derleyicinin kaynak kodu baştan sona kadar okumasıdır. Yani tek geçişli derleyicilerde kaynak kod baştan başlanıp sona kadar bir kere okunmakta buna mukabil çok geçişli derleyicilerde (örneğin iki geçişli bir derleyicide) birden çok kereler (örneğin iki kere) kaynak kod baştan sona kadar taranmaktadır. Tek geçişli derleyiciler tahmin edileceği üzere çok geçişlilere göre çok daha hızlı

çalışmaktadırlar. Ancak bazı durumlarda dilin tasarımı tek geçişli derleyicilere izin vermemektedir. Örneğin kodun sonlarına doğru kodun başında yapılan bir tanımlı etkileyecek bir işlem yapıldığını düşünün bu durumda tek geçişle bu olayın algılanması ve kodun doğru şekilde derlenmesinin yapılması mümkün olamaz. Tek geçişli derleyicilerin diğery bir eksik yanı ise kod iyileştirmesi sırasında kodun üzerinden sadece bir kere geçtiğı için kodun önceki satırlarında bulunan ve daha sonradan anlaşılan iyileştirmelerin yapılamamasıdır.

Sık	kullanılan	bazı	terimler:
kaynaktan kaynağı	derleyici (source to source compiler):	Bir dilden başka bir dile kod çeviren	
derleyicilerdir.	Örneğin C++ dilinden	JAVA diline çevirmek	gibi.
çapraz derleyici (crosscompiler):	Çalıştığı ortam dışında farklı bir ortam için kod üreten		
derleyicidir.	Örneğin Linux işletim sisteminde,	Windows işletim sistemi için kod üretmek	
			gibi.
Tam zamanında derleyici (just in time compiler):	Genelde ortam bağımsız ve ara seviye		
	kodların kullanıldığı JAVA veya .NET gibi dil aileleri için kodun son çalıştırıldığı ortamda		
	kodun iyileştirilmesini hedefleyen derleyici grubudur. Yani örneğin java kodu byte koda		
	çevrildikten sonra sanal java makinesi bu kodu zamanında derleyici ile çalıştırarak		
	zamanlamada iyileştirmeyi hedeflemektedir.		

SORU 46: alt program (subprogram, subroutine)

bir programın herhangi bir alt parçasına verilen isimdir. Daha resmî tanımı için ilave olarak bu alt parçanın belirli bir amaca yönelik olması gerektiğı söylenebilir. Yani programın herhangi bir alt parçası olmasının yanında bir amaç için bölünmüş parça'ya alt program diyebiliriz. Basitçe dilde bulunan fonksiyon (function), prosedür (procedure) , metod(method) veya herhangi bir blok için (if, while, for bloğı gibi) alt program tâbiri kullanılabilir.

Alt programlarda değişken kontrolü dilin özelliğine göre değişmektedir. (bkz. [sabit alanlı değişkenler](#) ve [dinamik alan değişkenleri](#))

Aşağıda örnek bir kod ve her kod parçası için bir alt program bölümü verilmiştir:

Yukarıda örnek bir 3 boyutlu savaş oyununun java kodunun bir kısmı bulunmaktadır. Bu koddaki alt programlar farklı renkler ile işaretlenmiştir. Buna göre alt program kavramı, resimde de görüleceği üzere her if, else, fonksiyon veya döngü parçasıdır. Daha fazla bilgi için [yapısal programlama](#) başlığına bakabilirsiniz.

SORU 47: fonksiyon göstericileri (function pointer)

fonksiyon göstericilerinin amacı, programlama dilinde bulunan fonksiyonları gösteren birer referans bilgisi tutmaktır. Bu sayede gösterilmekte olan fonksiyon için [hafızada](#) ayrılmış olan yere erişmek ve dolayısıyla örneğin fonksiyonun yerel değişkenlerine ulaşmak mümkündür. Aşağıda C dilinde yazılmış bir fonksiyon göstericisi kullanan kod örneği verilmiştir:

```
#include
#include

void func(int);

main(){
    void (*fp)(int);

    fp = func;

    (*fp)(1);
    fp(2);
}

void
func(int arg){
    printf("%dn", arg);
}
```

Yukarıdaki kodda “func” isminde bir fonksiyon tanımlanmıştır. Ayrıca void tipinde dönüş değerine sahip ve int tipinde parametre alan bir fonksiyon göstericisi “fp” tanımlanmıştır. Dikkat edilirse “func” fonksiyonunun ve “fp” göstericisinin hem parametreleri hem de dönüş değerleri aynıdır. Bu durum bir göstericinin fonksiyonu göstermesi (refer etmesi) için gereklidir. Bu gösterme işlemi atama satırı olan `fp=func;`

satırı ile yapılmaktadır. artık bu satırdan sonra “fp” göstericisine verilen her değer “func” fonksiyonuna verilmiş gibi icra edilir. Yani yukarıdaki kod çalıştırıldığında ekranda önce 1 sonra 2 görülmektedir.

Karıştırılmaması gereken bir nokta:
`void (*fp)(int);`
`void *fp(int);`
Yukarıdaki iki satır birbirinden farklıdır. İlk satır bir fonksiyon göstericisini, ikinci satır ise gösterici döndüren bir fonksiyonu tanımlarken kullanılmalıdır. Yani yukarıdaki iki satır aynı değildir.

Fonksiyon göstericilerinin fonksiyonlara parametre olarak verilmesi.

bir fonksiyon parametre olarak bir fonksiyon göstericisini alabilir. Aşağıda bunu yapan temsili kod verilmiştir:

```
#include
#include
int func(int);
void PassPtr(int (*pt2Func)(int))
{
    int sonuc = (*pt2Func)(12);
    printf("%d", sonuc);
}

main() {
    int (*fp)(int);
    fp = func;
    PassPtr(fp);
}

int
func(int arg){
    return ++arg;
}
```

Yukarıdaki örnek kodda, func ismindeki fonksiyonu gösteren fp isminde bir fonksiyon göstericisi tanımlanmıştır. Bu gösterici PassPtr fonksiyonuna parametre olarak verilmiştir. Bu kod çalıştırıldığında ekranda 13 sayısı görülür çünkü, func fonksiyonunu gösteren fp göstericisi PassPtr fonksiyonunun içinden çağırılmış ve değer olarak 12 parametresi atanmıştır. Fonksiyon incelenirse parametre olarak aldığı sayıyı bir arttırdığı görülür. Bu durumda ekrandaki değer 13 olmaktadır. Dikkat edilirse yukarıdaki kod ile, bir fonksiyona başka bir fonksiyon verilebilmektedir. Bu sayede genel amaçlı fonksiyonlar yazılarak ve bu fonksiyonlar parametre olarak geçirilerek programlamada avantaj elde edilebilmektedir.

Fonksiyon göstericilerinden dizi oluşturmak:

Fonksiyon göstericileri de birer gösterici olduğu için normal bir göstericiye yapılan her şey bu

göstericilere de yapılabilir. Bunlardan birisi de bir dizi tanımlamaktır. Aşağıdaki örnek kodu inceleyelim:

```
#include
#include
typedef int (*pt2Function) (int);
int
func(int arg){
    return ++arg;
}
int main(){
    pt2Function funcArr1[10] = {NULL};
    int (*funcArr2[10])(int) = {NULL};
    funcArr1[0] = funcArr2[1] = func;
    printf("%dn", funcArr1[1](12));
}
```

Yukarıdaki örnek kodda, iki adet fonksiyon gösterici dizisi tanımlanmıştır. İlk dizi olan funcArr1 dizisi, typedef marifeti ile tanımlanmıştır ve bu tip tanımı daha önceden yapılmıştır. İkinci dizi olan funcArr2 dizisi ise daha önceden tanımlanmış herhangi bir tip kullanmaksızın tanımlanmıştır. Sonuçta ekranda 13 sayısını gördüğümüz bu yukarıdaki kodda fonksiyon göstericisi, bir dizinin elemanı olarak tutulmakta ve çağrılmaktadır.

SORU 48: otomat yönelimli programlama (automata based programming)

otomat yönelimli programlama yaklaşımı, kaynağını otomatlar (automata)'dan alır ve sonlu durum makinaları (finite state machine, FSM) ile tasarlanan bir makinanın kodlanmasını hedefler.

Basitçe C dilindeki switch komutlarının dallanmasına benzer bir şekilde her durumdan bir sonraki duruma geçiş yapan bu programlama yaklaşımında amaç durumlar arası geçişin tasarıma uygun olarak kolay bir şekilde gerçekleşmesidir. Bunun için çeşitli dil çevirici araçlar olduğu gibi günümüz dillerinin pek çoğunda kullanılan eylem bazlı programlama (event based programming) aslında bir otomat yönelimli programlama tipidir.

Bu eylem bazlı programlama yaklaşımında yapılan, her eylem için bir alt program tanımlayarak, gerçekleşen olaylar sonucunda bu alt programlara yönlendirme yapılmasıdır. Örneğin visual basic, C++ veya JAVA gibi dillerde ekrandaki bir düğmeye (button) tıklanması durumunda bir fonksiyonun çağırılması veya javascript için onclick event, (tıklama eylemi) bu yaklaşıma birer örnektir.

SORU 49: üst programlama yaklaşımı (metaprogramming)

Üst programlama, mevcut programlama yaklaşımlarının üzerinde yeni bir yaklaşım geliştirerek programlama yapan programlama yapma anlamına gelir. Yani üst programlama ile bir program geliştirilirken, alt programlama yöntemleri harmanlanır ve kod üretilir.

Bu yaklaşımın ilginç kullanımlarından birisi de kendi kendini programlayan programların üretilmesidir. Yani üretilen kod, başka bir programa ait olmayıp bizzat üreticinin kendi parçası olmaktadır. Her iki ihtimalde de, üst programlama ile kast edilen, programın bir program üretmesidir.

Üst programın yazıldığı dile üst dil, alt programların her birisine de nesne dil veya nesne program adı verilmektedir. üst dil ile nesne dilin aynı olması durumunda, yani kendi kendini programlama durumunda buna da yansıma (relection) adı verilir.

Günümüzde en çok kullanılan örneklerine sunucu tarafı betiklerde (server side scripting) rastlanmaktadır. Örneğin PHP ile yazılan aşağıdaki örnek kodda 10 kere ekrana “bilgisayar.kavramlari.com” basan kodu üreten örnek kod verilmiştir.

```
echo " ";
for ($i=0;$i<10;$i++){
echo "

bilgisayarkavramlari.com

";
}
echo " ";
?>
```

yukarıdaki kodda [HTML kodunu](#) da kapsayan satırlar üretilmiş ve PHP dili ile HTML dilinde kod üretilmiş olmuştur.

Diğer sık kullanım alanlarından birisi ise derleyici teorisinde (compiler theory) sık kullanılan araçlar olan [lex](#) ve [yacc](#) programlarıdır. Bu programlar aracılığı ile bir programlama dili üretmek oldukça basittir ve bu programları kullanacak olan kişi örneğin yacc için basitçe tanımı yapılmış bir [parçalama ağacını \(parse tree\)](#) girdi olarak vermekte bu ağaçtan ise bir C kodu üretilmektedir.

SORU 50: fonksiyonel programlama (functional programming)

Programlama yaklaşımlarından birisi olan fonksiyonel programlama günümüz dillerinin neredeyse tamamında kullanılmaktadır. Bu yaklaşımda matematik fonksiyonlarında olduğuna benzer bir şekilde alt programlar tanımlanmakta ve bu alt programların değişik argümanlar ile çalışması sağlanmaktadır. Bu yaklaşım basitçe:

- Kod tekrarını engellemekte ve aynı kodun farklı şartlar için tekrar tekrar çalışmasını sağlamaktadır
- Kodun okunabilirliğini arttırmakta ve kod analizini daha kolay hale getirmektedir.
- Programın tasarlanması aşamasında tasarımcıya modüler yaklaşım yapmasını sağlamaktadır.

Bir dilin fonksiyonel olması dilde fonksiyon veya prosedüre benzeri özellikler bulundurması ile sağlanır. Bu özellikleri tanımlamak gerekirse:

- Sıfır veya daha fazla argüman ile giriş yapılabilen
- Sıfır veya daha fazla argüman ile çıkış yapabilen
- İç yapısında dilin izin verdiği alt programları barındıran yapıdır.

Yukarıdaki bu maddeleri bir [kara kutu \(blackbox\)](#) yaklaşımı olarak da düşünebiliriz. Yani alt programların, dış dünya ile (programın geri kalanı ile) olan tek bağlantıları almış oldukları ve geri döndürmüş oldukları argümanlardır (parametrelerdir).

Aşağıda bir fonksiyon örneği verilmiştir: ([JAVA](#), [C](#), [C++](#), [C#](#) dillerinde kabul edilir koddur):

```
int      toplama      (      int      a,      int      b) {  
return      a      +      b;  
}
```

Yukarıdaki kod incelendiğinde, a ve b, fonksiyona verilen argümanlardır (parametrelerdir). Dönüş değeri olarak tam sayı (integer) tipi kullanılmış ve bu durum return komutu ile belirtilmiştir. a+b eylemi ise bir alt programdır. Yani bu fonksiyon çağrıldığında icra edilen program parçasıdır.

Fonksiyonel programlama, [yapısal programlamanın](#) gerektirdiği bir yaklaşımdır. Buna göre fonksiyonel programlama kullanılan bütün diller [yapısal programlama](#) yaklaşımına uygundur denilebilir. Ancak tersi doğru değildir. Her ne kadar [yapısal programlamanın](#) tanımında bir alt programın varlığı zarurî olsa da bu alt program basit bir if bloğu olarak da düşünülebilir. Yani okuyucu kod blokları ile fonksiyonları karıştırmamalıdır.

[Nesne yönelimli programlama](#) yaklaşımınlarında fonksiyonel programlama kullanılmaktadır. Dolayısıyla her [nesne yönelimli programlama](#) yaklaşımı, fonksiyonel programlamayı barındırmaktadır. [Nesne yönelimli programlama](#) terminolojisinde, fonksiyonlara metod ismi verilmektedir.

SORU 51: yapısal programlama (structured programming)

yapısal programlama 1900lü yılların ortalarında programlama taleplerinin artması ile gelişen bir programlama felsefesidir. Buna göre programların analizi, tasarımları, kodlaması ve testleri arasındaki mantık uyumunu sağlamak amacıyla bir standarda gidilmiş ve aşağıdaki yapı çıkmıştır. Yapısal programlama amaç problemi alt parçalara bölerek bu parçaların çözümlerinin birleştirilmesidir. Bu yönüyle [parçala fethet \(Divide and conquer\)](#) yaklaşımı olarak kabul edilebilir.

yapısal programlamanın ortaya atılmasındaki sebepler:

1. goto komudunun karmaşıklığı: goto (atla, git veya jump) satırı bir kodun analizi, okunabilirliği ve testlerini neredeyse imkansız hale sokabilecek kadar karışmasını sağlayabilir. Bunun en basit sebebi akışların kontrol edilemez halde olmasıdır. Her ne kadar tersi iddialar da bulunsun güncel dillerin pek çoğu goto komutlarını sakıncalı bulmuş olsalar gerek bu komdu dilin doğal bir özelliği değil ama ek bir özelliği gibi barındırmaktadırlar.

2. tasarımda kullanılan yöntemlerin uyarlanma zorluğu: yapısal programlama öncesinde tam olarak ortaya atılmış ve genel kullanıma sahip, formal bir analiz ve tasarım sistemi bulunmuyordu. Yapısal programlama düşünme mantığında bir yenilik getirmesi hasebiyle tasarım ve analiz aşamalarında da güncel akış çizelgeleri (flow chart) öncülük etmiştir.

Yapısal programlama sahip bir dilde kontrol işlemleri (şartlar) aşağıdaki şekilde üçe ayrılırlar:

1. Akış (sequence) bir alt programdan diğerine geçiş işlemi. (fonksiyon veya prosedür, bkz. prosedürel programlama)
2. iki alt programdan birisini bir [bool mantık işlemine](#) göre çalıştırmak. (if , eğer)
3. bir şart sağlanana kadar bir alt programın çalıştırılması (döngüler, loop , iteration, for, while)

yukarıdaki şartları destekleyen bir dil yapısal programlama mantığına sahiptir denilebilir. Yukarıdakilerden birisinin eksik olması yapısal programlamaya sahip olmaması için yeterlidir. Günümüzde gelişen ihtiyaçlar ile artık tek bir yaklaşıma sahip diller yoktur. Bunun yerine pek çok farklı yaklaşımlara sahip diller bulunmaktadır. Örneğin C, C++, JAVA gibi diller yapısal programlayı yukarıdaki şartları destekledikleri için barındırmaktadırlar. Ancak bu diller farklı yaklaşımları da bünyelerinde barındırmaktadır.

Aşağıda yapısal programlama yaklaşımına göre tasarım yapmayı ve bir [akış çizelgesi \(flow chart\)](#) çizmeyi adımlara bölmüş yaklaşım verilmiştir:

1. Yapılması istenen adımları basit bloklara böl
2. Her bloğu tek bir çıkışı olacak şekilde yeniden tasarla veya böl
3. Bu çıkış noktalarını kullanarak blokları birbirine bağla
4. Tekrarlı bloklar için döngüleri ve döngülerin koşullarını tanımla
5. Çatallanma (dallanma, fork) için şartları tanımla (if)
6. Bağlantı eksikleri bulunan blokların son bağlantılarını tamamla

Yukarıdaki adımlar tanımlandıktan sonra basit yorumlar yapılabilir. Örneğin bir değişken tanımlayarak ilk değer olarak 0 atarsak, her bloğa girişinde bir arttırsak ve her bloktan çıkışında bir azaltsak program sonunda değişkenin değerinin 0 olması beklenir.

Yapısal programlama, bir programlama yaklaşımı olup, güncel gelişmelerle birlikte kullanılmaya devam etmektedir. Örneğin nesne yönelimli programlama yaklaşımlarını kullanan dillerin neredeyse tamamı yapısal programlamayı da bünyelerinde barındırmaktadır.

SORU 52: kapsülleme (encapsulation)

genel olarak bir bilginin soyut bir yapı içerisine konulmasına verilen isimdir. En çok ağ teknolojilerinde ve nesne yönelimli programlama dünyasında kullanılır.

Nesne Yönelimli Programlama için anlamı bir sınıfın (class) bilgilerinin dışarıya kapalı olması ve bu sınıfın her türlü veri iletişiminin kontrol altındaki metodlar ile yapılmasıdır.

Ağ teknolojileri için anlamı, katmanlı mimaride (OSI katmanı veya Internet katmanları gibi), her katman arasında verinin bir kapsüle konularak diğer (alt veya üst) katmana geçirilmesidir. Buna göre gönderilmek istenen veri her alt katmana indikçe, indiği katmandaki başlık bilgileri eklenerek yeni bir kutuya konulmuş gibi paketlenir. Ulaştığı yerde ise her katman kendisi ile ilgili kutuyu açarak görevini icra eder.

SORU 53: bit (ikil)

Bilgisayar dünyasında ikili tabandaki (binary) tek haneli bir sayıyı ifade eder. Yani bir bit değeri 1 veya 0 olabilir. Bu aslında elektronik sinyali olarak yüksek (1) veya düşük (0) gerilimde akım demektir.

bir bit, 1 veya 0 değeri alabildiğine göre her bit değerinin 2 farklı değer alması mümkündür. Bu durumda örneğin iki sistemde yazılmış 8 haneli bir sayı (8 bitlik bir sayı) 2 üzeri 8 farklı değer = 256 farklı değer alabilir. Örneğin 11010010 sayısı, 8 bitlik bir sayıdır.

8 bitlik sayılara bilgisayar dünyasında kısaca byte adı da verilmektedir.

Bit değerleri için kullanılan birimler aşağıda listelenmiştir:

<u>Metrik</u> Gösterim		<u>İkili</u> (binary) Gösterim	
İsim	Metrik Değeri	İsim	Değeri
<u>kilobit</u> (kbit)	10^3	<u>kibibit</u> (Kibit)	2^{10}
<u>megabit</u> (Mbit)	10^6	<u>mebibit</u> (Mibit)	2^{20}
<u>gigabit</u> (Gbit)	10^9	<u>gibibit</u> (Gibit)	2^{30}
<u>terabit</u> (Tbit)	10^{12}	<u>tebibit</u> (Tibit)	2^{40}
<u>petabit</u> (Pbit)	10^{15}	<u>pebibit</u> (Pibit)	2^{50}
<u>exabit</u> (Ebit)	10^{18}	<u>exbibit</u> (Eibit)	2^{60}
<u>zettabit</u> (Zbit)	10^{21}	<u>zebibit</u> (Zibit)	2^{70}
<u>yottabit</u> (Ybit)	10^{24}	<u>yobibit</u> (Yibit)	2^{80}

SORU 54: İşlem (Process)

Bir işletim sistemi üzerinde herhangi bir dil ile kodlanmış ve bir compiler (derleyici) ile derlenmiş ve daha sonra [hafızaya](#) yüklenerek işlemcide çalıştırılan programlara verilen isimdir.

Genel anlamda her program bir process olarak düşünülebilir, ancak bir programın birden fazla processi olabileceği gibi her process, yeni başka processlerde üretebilir (fork) . İşletim sisteminin tasarımına göre değişmekle birlikte [işlemler \(process\)](#) kendi adres alanında (own adress space) çalışırlar ve hafıza koruması (memory protection) uygulanır. Bu sayede bir işlemin, başka işlemlerin bilgisine erişmesi engellenmiştir.

İşlemler arası iletişim (Inter process communication (IPC)), aynı bilgisayarda çalışan farklı programların haberleşmesini hedef alır, bu durum ağ üzerinde birbiriyle haberleşen bilgisayarlara benzetilebilir.

SORU 55: Pointer (Gösterici) ve Diziler (Arrays)

Pointer (gösterici) basitçe bir değişkenin bir [hafıza](#) alanını göstermesi demektir. C dilinde pointerlar:

Veritipi *pointeradı;

Şeklinde tanımlanır. Burada veritipi int, char, float gibi değişken tipleridir. Pointer adı ise bir değişken adının taşınması gereken özellikleri taşıyan ve tanımlandığı scope (geçerlilik alanında) yaşayacak olan değişken adıdır.

Bir değişkenin başına & işareti geldiğinde ise bu değişkenin adresini temsil eder. Yani değişken hafızada nereye konulduysa bu konulan yerin adresine bu işaret ile ulaşılabilir.

Örneğin,

```
int *p;
```

```
int a=10;
```

```
p=&a;
```

satırından sonra hafızada integer değer gösteren p isminde bir pointer tanımlanmış olur. Bunu hafızda temsili olarak gösterecek olursak:

Adres değeri	içeriği	matıksal ismi
A101		
A102		
A103		
A104		
A105		
A106		
A107	A116	p
A108		
A109		
A110		
A111		
A112		
A113		
A114		
A115		
A116	10	a
A117		
A118		
A119		
A120		

Yukarıdaki temsili resimde, ilk sütun adres değerlerini temsil etmektedir, buna göre hafızanın a101 ile a120 numaralı adresleri arasındaki bilgiler gösterilmiştir (bu değerler hexadecimal olup temsili olarak yazılmıştır)

Kodda bulunan p ve a değerlerinin hafızda hangi bölüme atanacağı çalıştırma sırasında (execution time) belirlenir dolayısıyla tam olarak nerede bulunacağı bilinemez ancak p tanımlandıktan sonra a107 adresine ve a tanımlandıktan sonra a116 adresinin ayrıldığını kabul edelim. Bu durumda p'nin değeri a107 numaralı adreste olacaktır.

Kodda a değerine 10 konulmuştur bu durumda a'nın değerinin durduğu a116 numaralı adreste sayısal olarak 10 yazacaktır.

Kodda a'nın taşıdığı adres değeri, p'nin içine atılmıştır. Bu durumda p'nin değeri a'nın adresi olacaktır.

Aşağıdaki örnek kodu çalıştırınız ve yorumlayınız:

```

#include

int main(){

int a=10;

int *p;

p=&a;

printf(“%dn”,*p); // p’nin gösterdiği yeri basar

printf(“%dn”,p); // p’nin değerini yani, p’nin gösterdiği yerin adresini basar

printf(“%dn”,a); // a’nın değerini basar

printf(“%dn",&a); //a’nın adresini basar

printf(“%dn",&p); //p’nin adresini basar

}

```

C dilinde her dizi bir pointer her pointer da doğal bir dizidir.

```

char                                str[80],                                *p1;
p1 = str;

```

Burada p1, str dizisinin – stringinin – ilk elamanının adresinin degerini alır. Yani string adi, aslında o stringin hafızadaki baslangıç adresini = stringin ilk karakterinin adresini tutmaktadır. str dizisinin 5. elemanına erismek için ise;

```
str[4] veya *(p1+4)
```

ifadelerini kullanırız. Her ikisinin de anlami aynidir.

C’de dizi elemanlarına 2 sekilde ulasilir: pointer kullanimi ile ve indis kullanimi ile. Indis kullanimi gelistirme ve anlama bakimindan bir kolaylik saglasa da, hiz önemli bir konu oldugundan C programcileri genelde dizi erisimini pointer kullanarak yaparlar. Asagida ayni isi yapan iki fonksiyon yeralmaktadır:

```

void                                putstr(char                                *s)
{
int                                t;
for                                (t=0;                                s[t];                                ++t)                                putchar(s[t]);
}
void                                putstr(char                                *s)
{
while                                (*s)                                putchar(*s++);
}

```

C dilinde bir pointer'ı array gibi kullanmak için:

```
int *p = (int *) malloc ( sizeof(int)*10);
```

şeklinde bir satır yazmak yeterlidir. Burada klasik dizi tanımından farksız bir işlem yapılmış, hafızada 10'luk bir alan açılmış ve tipi integer olarak tanımlanmıştır.

Pointer'ın farkı bu alana erişim şeklindedir. Yani dizi tanımlarında olduğu gibi köşeli parantezle (a[3] gibi) erişmek yerine pointer üzerinde işlem yapmak gereki ancak yöntem bir önceki örnekte olduğu gibi

```
printf("%dn",*(p+4)) şeklinde işleyebilmektedir.
```

SORU 56: Static Scoping (Sabit Alanlı Değişkenler)

Static scoping ile en yakın değişken bindingin değeri atanır. Basit bir şekilde programın metni okunup bu işlem yapılabilir. Programın çalışırken (runtime) oluşturduğu stack içeriğine bakılmasına gerek yoktur. Sadece metine bakması yeterli olduğu için bu tarz scopinglere lexical scoping adı da verilir. Static scope, kodun anlaşılmasını daha kolay hale getirdiği için daha modüler kodlar yazılmasını sağlar. Ters olarak dynamic scoping ise programcının bütün olası stack değerlerini ve karşılaşılabileceği olasılıkları hesaplamasını gerektirdiği için itici olabilir.

Örneğin aşağıdaki kod hem static hem de dinamik (dynamic) scoping ile çalıştırılabilir:

```
int x = 0;
```

```
int f () { return x; }
```

```
int g () { int x = 1; return f(); }
```

Şayet static scoping kullanılırsa g fonksiyonunun döndüreceği değer 0 olur çünkü, static scopingin o sırada fonksiyon stackinde ne olduğu ile ilgisi yoktur ve x değerinin son hali olan 0'ı alır.

tersi olarak dynamic scoping kullanarak bu kod çalıştırılacak olsaydı g fonksiyonunun döndüreceği değer 1 olacaktır. Çünkü g fonksiyonu terk edilmeden önce x in değeri 1 di ve bu bilgi stackten alınır.

SORU 57: Row Major Order (Satır bazlı sıralama)

Bilgisayar bilimlerinde, rowmajor order veya column major order ile, çok boyutlu (multidimensional) dizilerin (array) dogrusal hafızada (linear memory) tutulma şekli kastedilmektedir. Arraylerin hafızada tutulma biçimleri özellikle diller arasında veri geçişi yaparken kritik rol oynar. Ayrıca diziye erişim şekli doğru biliniyorsa ve bu bilgi doğru kullanılıyorsa performansı oldukça etkileyen bir özelliktir.

Row-major order

Satırların arka arkaya saklandığı linear memory modelinin ismidir. C dilinin kullandığı model de row-major modeldir.

Örneğin:

1	2	3
4	5	6

Masfuf'unun (matrixinin) C dilinde tanımı:

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

şeklinde yapılır ve bu dizinin hafızadaki tutlma biçimi aslında:

1 2 3 4 5 6

Şeklindedir. Yani A[satır][sütün] şeklinde verilen bir dizini için

offset = satır*sütünsayısı + sütün

Şeklinde					verilebilir.
Kolon	bazlı	sıralamada	ise	aynı	matrix:
1		2			3
4	5	6			

aşağıdaki şekilde hafızada tutulacaktır.

1 4 2 5 3 6

SORU 58: Dyanmic Scoping (dinamik alan değışkenleri)

Dynamic scoping, fonksiyon değışkenlerinin fonksiyonlar ile birlikte stackte tutulmasını hedefler. Buna göre fonksiyonların içinde tanımlanmış olan değışkenler, o fonksiyon çalışırken geçerli olur ve o fonksiyonun içinde atanmış olan değeri korur. Bir fonksiyondan başka bir fonksiyon çağrılmış olsa ve çağrılan fonksiyonda değışkenin değeri değışmiş olsa da çağırnan fonksiyona geri döndüğünde fonksiyonun değışken değeri geri yüklenir.

Örneğin aşağıdaki kod hem static hem de dynamic scoping ile çalışabilir:

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

Şayet static scoping kullanılırsa g fonksiyonunun döndüreceği değeri 0

olur çünkü, static scopingin o sırada fonksiyon stackinde ne olduğu ile

ilgisi yoktur ve x değeri son hali olan 0ı alır.

tersi olarak dynamic scoping kullanarak bu kod çalıştırılacak olsaydı g fonksiyonunun döndüreceği değer 1 olacaktır. Çünkü g fonksiyonu terk edilmeden önce x in değeri 1 di ve bu bilgi stackten alınır.

SORU 59: Coloumn Major Order (Sütün bazlı sıralama)

Bilgisayar bilimlerinde, rowmajor order veya column major order ile, çok boyutlu (multidimensional) [dizilerin \(array\)](#) dogrusal hafızada (linear memory) tutulma şekli kastedilmektedir.

Arraylerin hafızada tutulma biçimleri özellikle diller arasında veri geçişi yaparken kritik rol oynar. Ayrıca diziye erişim şekli doğru biliniyorsa ve bu bilgi doğru kullanılıyorsa performansı oldukça etkileyen bir özelliktir.

Column major order

Sütünların arka arkaya saklandığı linear memory modelinin ismidir. FORTRAN dilinin kullandığı model de column-major modeldir.

Örneğin:

1	2	3
4	5	6

aşağıdaki şekilde hafızada tutulacaktır.

1 4 2 5 3 6