

Bölüm 7: Yapısal Programlama ve Akış Denetimi

CONCEPTS OF Programming Languages

TENTH EDITION

ROBERT W. SEBESTA

Programming Language Design Concepts

David A. Watt
with contributions by
William Findlay

PROGRAMMING LANGUAGE PRAGMATICS

THIRD EDITION

Michael L. Scott

BÖLÜM 7- Konular

- Yapısal Programlama
- Program Akış Denetim Deyimleri

7.1. YAPISAL PROGRAMLAMA



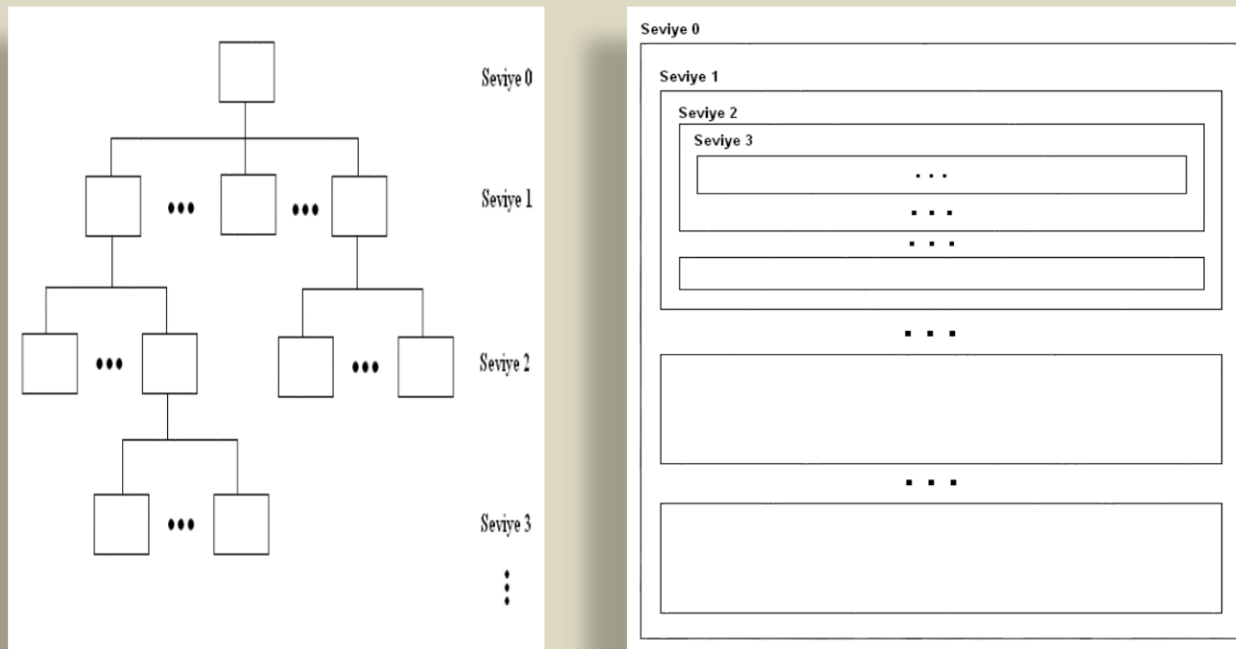
- Birinci bölümde incelenen yazılım yaşam döngüsünün en uzun süren ve en çok kaynak gerektiren aşamasının bakım aşaması olduğu, günümüzde yazılım mühendisliği çevrelerinin üzerinde anlaştığı bir noktadır.
- Bu nedenle, yazılımların bakım aşamasının kolaylaştırılması için anlaşılması kolay programlama dili yapılarının tasarımının önemi, geçmişe oranla daha da artmıştır.
- Yapısal programlama, program tasarımı ve yazılmasını kurallara bağlayan ve disiplin altına alan bir yaklaşımdır

7.1. YAPISAL PROGRAMLAMA

- Yapısal programlamanın amacı, program metninin incelenmesi ile programın işlevinin anlaşılmasını sağlayarak bakım aşamasının en temel işlemi olan program anlaşılabilirliğine katkıda bulunmaktadır.
- Yapısal programlamada problem çözümü daha kolay alt problemlere (modül) bölünür. Her bir alt problem (modül) daha düşük seviyedeki alt seviyelere bölünür.

7.1. YAPISAL PROGRAMLAMA

- Bu işlem, aşağıdaki şekilde de görülebileceği gibi her bir modülün kolaylıkla çözülebileceği seviyeye kadar devam eder.



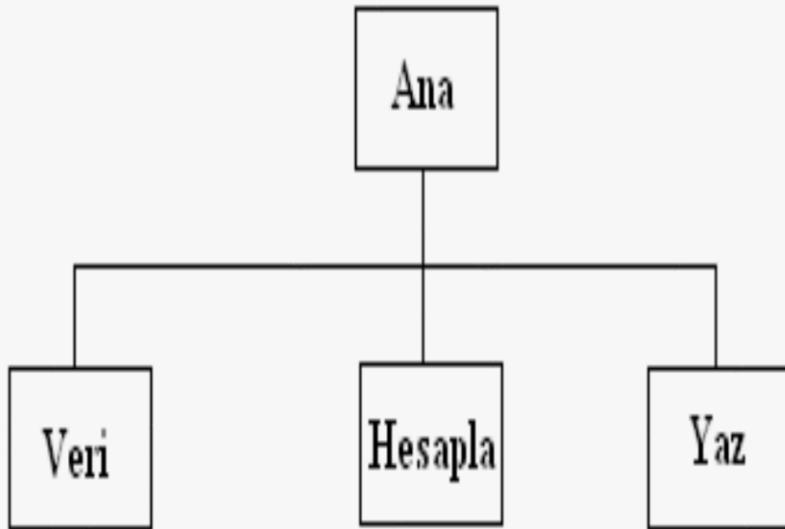
- En üst seviyede çözümün ana mantığının sergilendiği ana modül yer alır. Alt seviyelere indikçe izlenecek adımlar ile ilgili ayrıntılar artar.

7.1. YAPISAL PROGRAMLAMA

- Modüler program tasarımında her modül diğerlerinden bağımsız olmalıdır. Kontrol her modüle bir üst seviyedeki modülden geçmeli ve modül işlendikten sonra tekrar aynı modüle iletilmelidir.
- Modüllerin tanımlanmasında, (algoritma parçası olduğu için) sözde kod (pseudo-code) veya akış diyagramı kullanılır.
- Bir modülün çözümünde kullanılacak algoritma, sözde kod ile veya kullanılacak programlama dilinin yapısına uygun bir şekilde akış diyagramı ile ifade edilirse programa geçiş büyük ölçüde kolaylaşır.

7.1. YAPISAL PROGRAMLAMA

- Örnek
- *Problem:* 1'den n' ye kadar olan tam sayıların toplamını bulmak.
- Problem çok kolay olmasına rağmen modüler programlamaya bir örnek olması açısından aşağıdaki şekilde bir tasarım düşünelim;



Ana	
Veri	<input type="text"/>
Hesapla	<input type="text"/>
Yaz	<input type="text"/>

7.1. YAPISAL PROGRAMLAMA

- İlk yüksek seviyeli programlama dillerinden FORTRAN, programların okunabilirliğini artırdığı için makina ve birleştirici dillerine göre önemli bir gelişme sağlamıştır.
- Ancak FORTRAN, ağırlıklı olarak **go to** deyimlerine dayandığı için programlarda akış denetimini yönetmek güçtür.
- ALGOL 60 ve izleyen diller, FORTRAN'a göre daha iyi yapılar tanıtmış olmakla birlikte, programlama dillerinde yapısal programlama kavramı ağırlıklı olarak 1970'lerde yaygınlaşmıştır.

7.1. YAPISAL PROGRAMLAMA

- Yapısal programlama, programlardaki akış denetiminin, üç temel yapı olan Sıralı, Seçimli ve Yinelemeli yapılarının birleştirilmesi ile oluşturulmasını gerektirir:

Sıralı

Programlamadaki en temel yapıdır. Bir programda yer alan iki veya daha fazla program deyiminin görüldükleri sırada çalıştırılmasını sağlar.

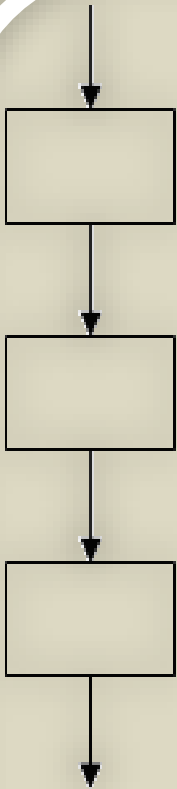
Seçimli

Bir programda yer alan iki veya daha fazla olası yol arasında bir seçim yapılmasını sağlar.

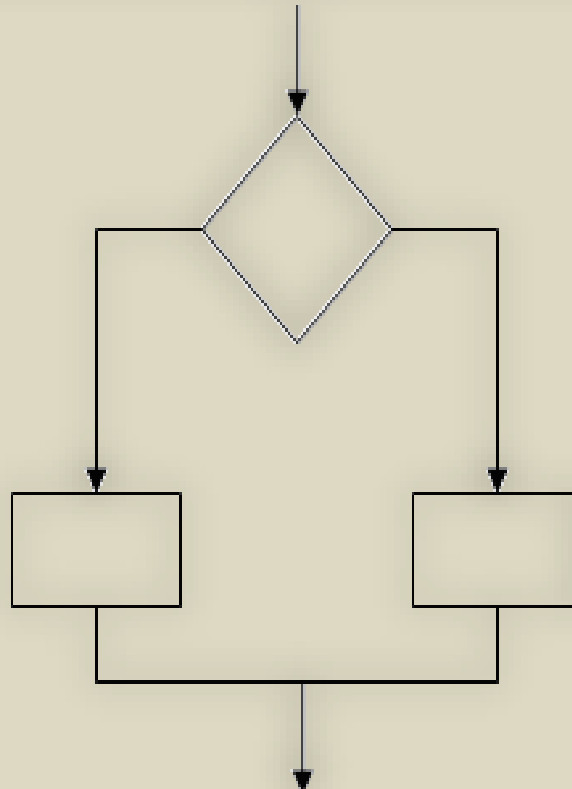
Yinelemeli

Bir programda yer alan bazı deyimlerin birden çok kez çalıştırılmasını sağlar.

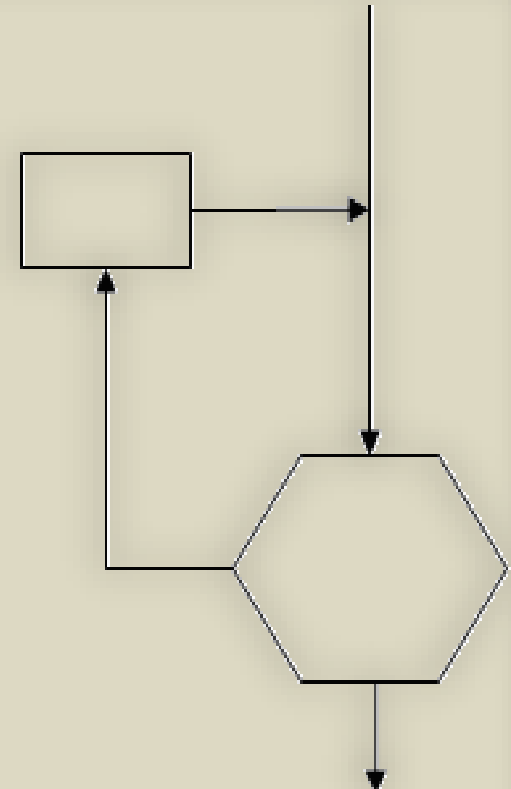
7.1. YAPISAL PROGRAMLAMA



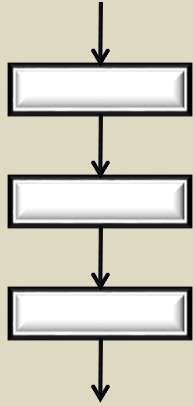
Sıralı



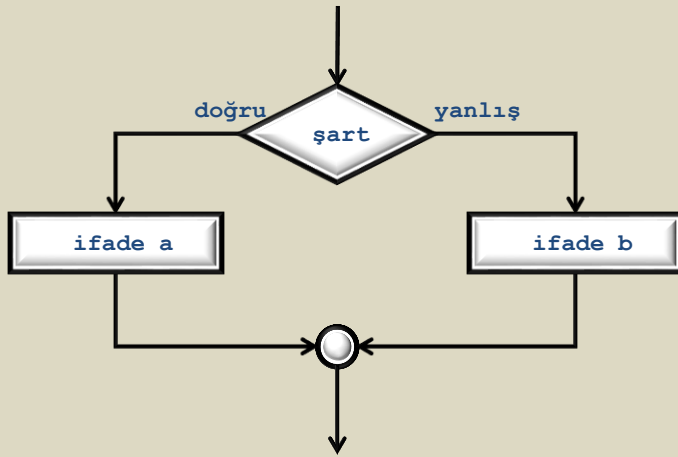
Seçimli



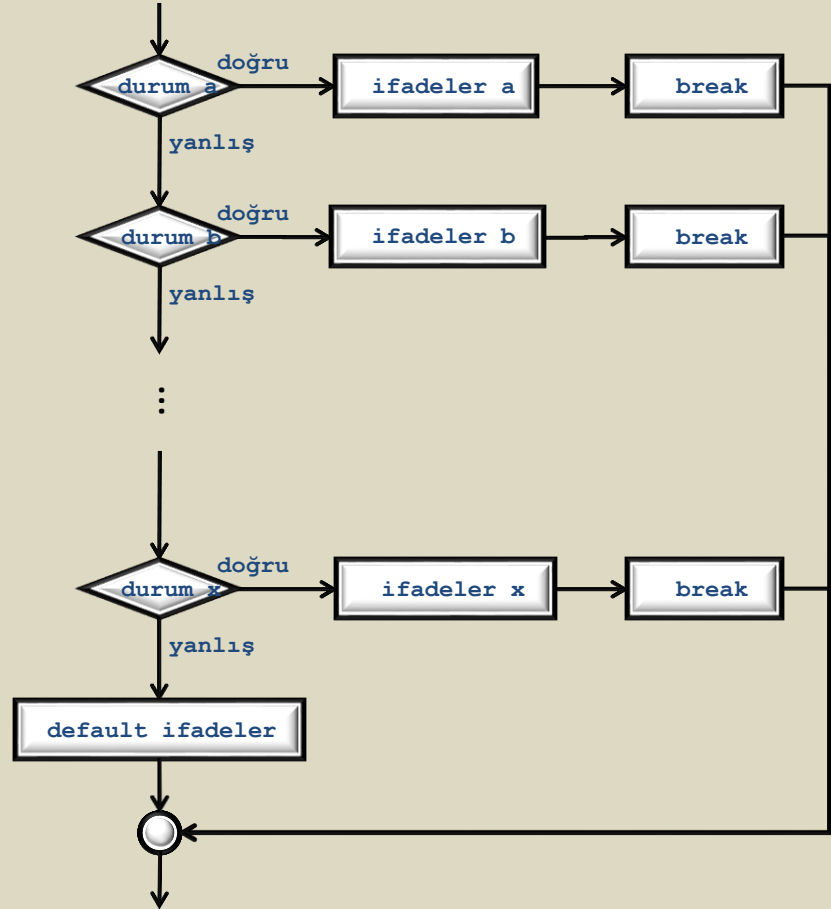
Yinelemeli



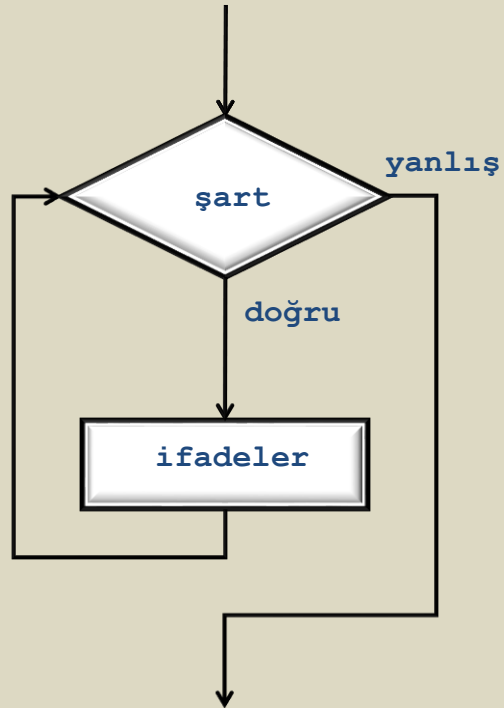
Sıra



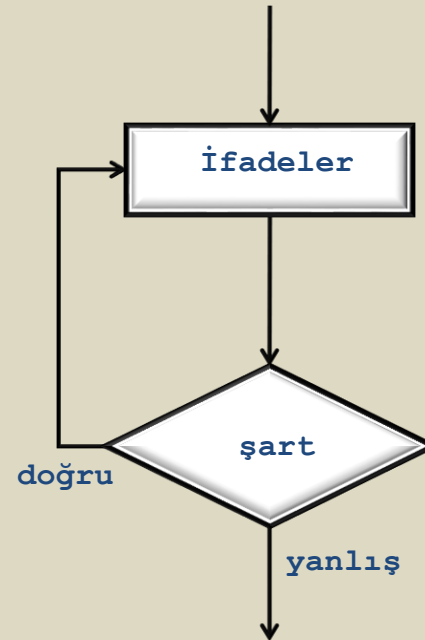
Seçim (if/else)



Seçim (çok-yollu)



Test öncesi yineleme
(**while** <test> **do** <stuff>)

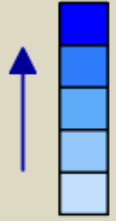


Test sonrası yineleme
(**do** <stuff> **while** <test>)

7.1. YAPISAL PROGRAMLAMA

- Bu açıklamadan anlaşıldığı gibi, yapısal programlama, programlarda koşulsuz olarak akışı değiştiren *goto* gibi deyimlere yer vermemektedir.
- 1970'li yıllarda yaygınlaşmaya başlayan yapısal programlama, günümüzde yararlarını kanıtlamış bir programlama tekniğidir.
- Yapısal programlamanın yararları aşağıda belirtilmiştir:
 - 1. Programların anlaşılabilirliği artar.
 - 2. Bir programın, hatalarının ayıklanması, sınanması ve düzeltilme zamanı kısalır.
 - 3. Bir programın niteliği, güvenilirliği ve etkinliği artar.

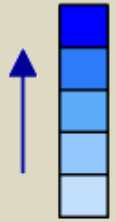
7.2. PROGRAM AKIŞ DENETİM DEYİMLERİ



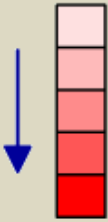
Akış Denetim
Deyimleri



Yazılabilirlik



Akış Denetim
Deyimleri



Okunabilirlik

- Yapısal programlamayı oluşturan, üç deyim yapısı, yani sıralı, seçimli ve yinelemeli yapılar, *imperative* programlamanın deyimlerini oluştururlar.
- Programlarda, belirli deyim ya da deyimlerin yinelemeli olarak çalıştırılması veya program akış yolları arasında seçim yapılması, denetim yapıları aracılığı ile sağlanır.
- Bir dilde çok sayıda akış denetim deyiminin bulunmasının dilin yazıla bilirliğini artırması nedeniyle, tüm programlama dilleri çok sayıda akış denetim deyimini içerirler.
- Öte yandan, bir dildeki akış denetim deyimlerinin sayısı arttıkça dilin okunabilirliği azalmaktadır.

7.2.1. Atama Deyimi

- En temel sıralı işlem deyimi, **atama deyimi** dir.
- *Imperative* programlama dillerinde sadece atama deyimi ile bir işlem yani değişkenlerin değerlerinin değiştirilmesi gerçekleştirilebilir. Seçimli ve yinelemeli deyimler atama deyimlerinin çalışma sırasını belirlemek için kullanılır.
- Atama deyiminin sözdizimi genel olarak aşağıdaki yapıdadır:
- **<hedef_değişken> <atama_işlemcisi> <ifade>**
- Bu söz dizimde *ifade*, 3 veya $a + 5$ bir değeri, *hedef değişken* ise bellekteki bir adresi göstermektedir. Atama deyiminin anlamı, *atama işlemcisinin* sağ tarafındaki değerin, atama işlemcisinin sol tarafındaki adrese kopyalanmasıdır.

7.2.1. Atama Deyimi

- **Atama İşleminde Uyumluluk:**
- Sağ taraftaki değerin, sol taraftaki değişkene kopyalanması işleminin gerçekleşmesi için sağ ve sol tarafların atama uyumlu olması gereklidir.
- Atama uyumluluk tip kavramı ile bağlantılıdır. Sağ ve sol tarafın farklı tiplerde olması durumunda bazı programlama dillerinde zorunlu tip dönüşümü (*coercion*) gerçekleşebilir veya tip uyumsuzluğu hatası oluşabilir.

7.2.1.1. Atama İşlemcisi

- FORTRAN, BASIC, PL/I, C, C++ ve Java atama işlemcisi olarak "=" işaretini kullanırlar. Eğer programlama dilinde eşit işareti aynı zamanda ilişkisel işlemci olarak da kullanılıyorsa (örneğin; PL/I ve BASIC), eşit işaretinin iki görevi karışıklığa neden olabilir.
- Örneğin PL/I'da; ***say1=say2=say3***
- deyimi, *say2=say3* mantıksal ifadesinin sonucunu *say1*'e atamayı gösterir. Ancak bu deyim, *say3*'ün değerinin *say2* ve *say1*'e atanması şeklinde algılanabilir.
- Bu ve benzeri karışıklıkları önlemek için ALGOL 60 ve onu izleyen diller, atama işlemcisi olarak ':=' kullanmışlardır.
- C'de ise "=" işareti atama sembolünü, "==" işareti ise mantıksal eşitlik işlemcisini göstermektedir. Bu nedenle C'de atama işlemcisi ve mantıksal eşitlik işlemcisi arasındaki benzerlik nedeniyle oluşabilecek hataların programcılar tarafından fark edilmesi güçtür.

	Atama İşlemcisi	Mantıksal Eşitlik İşlemcisi
ALGOL60	:=	=
C	=	==

7.2.1.2. Atama Deyimlerinin Tasarımı

- Atama işlemcisinin yanı sıra, atama deyimlerinin tasarımı programlama dillerinde değişiklik göstermiştir.
- FORTRAN, Pascal ve Ada'da atama deyimi, tek bir deyim olmalı ve atamanın hedefi tek bir değişken olmalı iken, çeşitli programlama dillerinde farklı tasarımlar uygulanmıştır.
- PL/I'da, atama işlemcisinin solunda birden çok değişken bulunabilir.
- C'de ise aynı amaçla farklı bir söz dizim kullanılmıştır. Aşağıdaki örnekte PL/I ve C' de iki atama deyimi örneği gösterilmektedir.

PL/I: toplam, sayac = 0; → Yandaki atama deyimi ile hem *toplam* hem de *sayac* değişkenine *sıfır* değeri verilmiştir.

C: toplam = sayac = 65; → Öncelikle *sayac* değişkenine 65 değeri verilir, daha sonra *sayac* değişkeninin değeri *toplam* değişkenine atanır.

7.2.1.2. Atama Deyimlerinin Tasarımı

- **C ve C++'da Atama Deyimleri:**
- Bir C programı aynı amaçla diğer bir dilde yazılmış bir program ile karşılaştırılırsa, C programının daha kısa olduğu görülür. Bunun bir nedeni sık kullanılan program deyimleri için kısa bir gösterim sağlayan bazı işlemcilerin C'de tanımlı olmasıdır.
- Diğer programlama dillerinden farklı olarak C ve C++'da, atama deyimleri için tanımlanan tasarımlar ***Koşullu Hedefler, Birleşik Atama, Artırma - Azaltma - Atama*** olmak üzere üç tanedir.

7.2.1.2.1. Koşullu Hedefler

Örnek

```
okunan ? say1 : say2 = 60;
```

- Bir atama deyiminin hedefi, bir koşula bağlı olarak belirlenebilir.
- Yandaki deyimin anlamı, *okunan* doğru ise, *say1*'e *60* değerinin atanması, doğru değilse *say2*'ye *60* değerinin atanmasıdır.

- Koşullu Hedefler (Perl)

`($flag ? $total : $subtotal) = 0`

eşittir

```
if ($flag) {  
    $total = 0  
} else {  
    $subtotal = 0  
}
```

7.2.1.2.2. Birleşik Atama

- Bir atama işlemcisi ile bir ikili işlemci birleştirilerek birleşik atama işlemcilerini oluştururlar. C ve C++'da, çeşitli ikili işlemciler için birleşik atama işlemcileri vardır.
- C'de; "+=", "-=", "*=", "/=", "%=" birleşik işlemcileri tanımlıdır.
- Aşağıdaki örneklerde verilen deyimlerin işlevleri aynıdır:

+

toplam += say1; \longleftrightarrow toplam = toplam + say1;

*

a *= 2; \longleftrightarrow a = a * 2;

7.2.1.2.3. Artırma - Azaltma - Atama

- C ve C++, artırma ve azaltma işlemlerini atama deyimi ile birleştiren ve iki sayısal işlemci içermektedir. Hem işlenenlerden önce yazılan bir işlemci olarak, hem de işlenenlerden sonra yazılan bir işlemci olarak tanımlanan artırma ve azaltma işlemcileri "++" ve "--", atama deyimlerinde veya ifadelerde bulunabilir.
- Aşağıdaki örneklerde sol tarafta tek deyimle verilen işlev, sağ tarafta verilen iki deyim ile gerçekleştirilebilir.

Örnek

sayac1 değişkeninin değeri 1 artırıldıktan sonra, *toplam* değişkenine atanmıştır.

`toplam = ++sayac1;` \longleftrightarrow `sayac = sayac+1;
toplam = sayac;`

say1 değişkeninin değeri *toplam* değişkenine atandıktan sonra, 1 artırılır.

`toplam = say1++;` \longleftrightarrow `toplam = say1;
say1 = say1+1;`

Başka örnek: `-toplam++` (*toplam* arttırılır, ondan sonra negatifi alınır)

7.2.1.2.3. Artırma - Azaltma - Atama

- Bu işlemciler, tam bir atama deyimi de oluşturabilirler.

Örnek

say1 değişkeninin değeri 1 artırılır.

`say1++;` \longleftrightarrow `say1 = say1+1;`

puan değişkeninin değeri 1 azaltılır.

`puan--;` \longleftrightarrow `puan = puan -1;`

7.2.2. Bir İfade Olarak Atama

- C-tabanlı diller, Perl, ve JavaScript'te atama durumu bir sonuç üretir ve bir operant olarak kullanılabilir,

```
while ((ch = getchar()) != EOF) {...}
```

ch = getchar() **başarılı; sonuç**(ch a aktar) **while** döngüsü için şartsal bir değer olarak kullanılır

- Dezavantaj: Başka tip yan etkiler.

7.2.3. Çoklu Atamalar

- Perl, Ruby, ve Lua çok hedefli ve çok kaynaklı atamalara izin verir

```
($first, $second, $third) = (20, 30, 40);
```

Hatta, aşağıdaki geçerli ve bir yer değiştirme uygulanır:

```
($first, $second) = ($second, $first);
```

7.2.4. Fonksiyonel Dillerde Atama

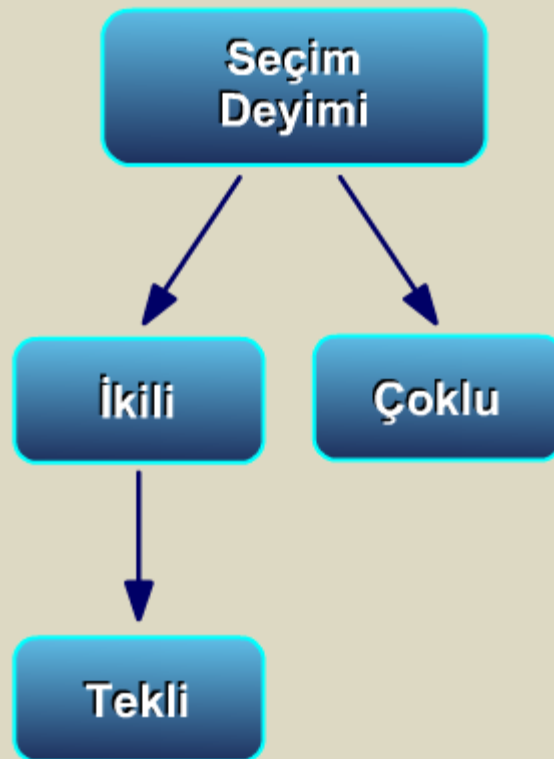
- Fonksiyonel dillerde tanıttıcılar (identifier) sadece değer adlarıdır.
- ML
 - İsimler `val` ve değer ile sınırlıdır İsimler
 - `val fruit = apples + oranges;`
 - Eğer `fruit` için başka bir `val` izlenecekse, o yeni ve farklı bir isimde olmalıdır.
- F#
 - F#'s yeni bir kapsam (scope) yaratmanın dışında ML 'in `val` ile aynıdır .

7.2.5. Karışık Biçim Ataması

- Atama ifadeleri karışık biçimde olabilir
- Fortran, C, Perl, ve C++'ta ,her tip sayısal değer her tip sayısal değişkene atanabilir
- Java ve C#'ta, sadece genişletici atama zorlaması yapılır
- Ada'da, atama zorlaması yoktur.

7.2.2. Seçim Yapıları

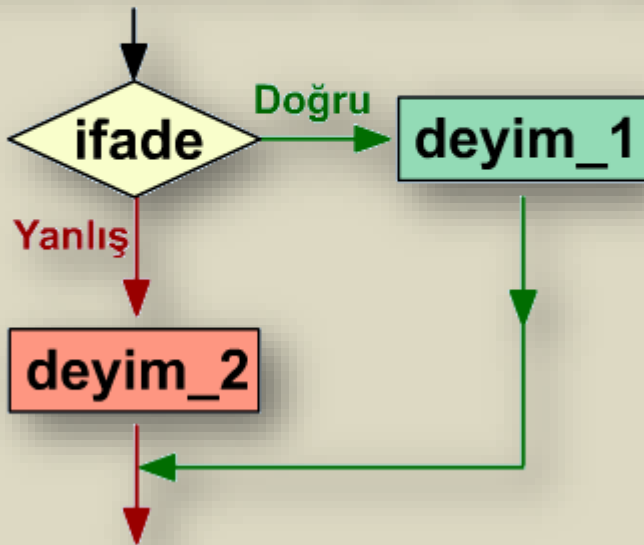
- Bir seçim deyimi, aşağıdaki şekilde gösterildiği gibi tekli, ikili veya çoklu akış yolları arasında seçim yapılmasını sağlar.



7.2.2.1. İkili ve Tekli Seçim Deyimleri

İkili Seçim Deyimi

if (kosul ifadesi) then deyim_1 else deyim_2

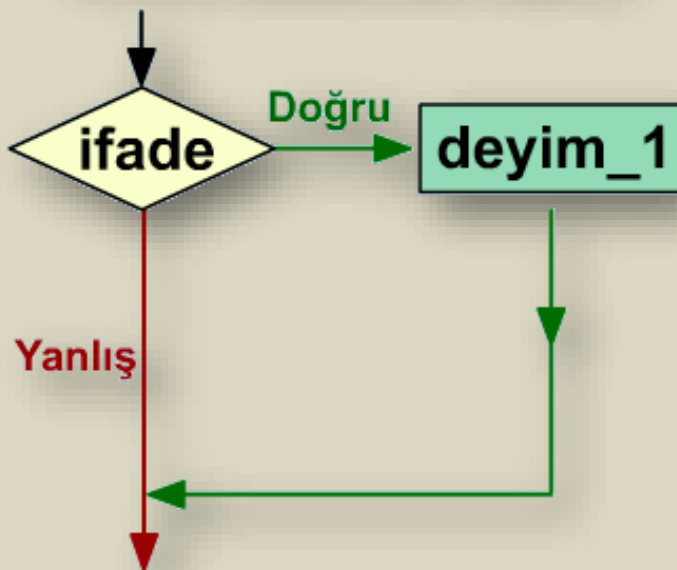


- İkili Seçim Deyimi:
- İki-yollu seçim deyiminin sözdizimi çeşitli programlama dillerinde değişmekle birlikte, genel yapısı yandaki şekilde görülmektedir.
- Bu yapının anlamı, koşul ifadesinin **doğru** sonucunu alması durumunda **deyim_1**'in, koşul ifadesinin **yanlış** sonucunu alması durumunda ise **deyim_2**'nin çalıştırılmasıdır.
- Tasarımla ilgili hususlar:
 1. Kontrol ifadesinin şekli ve tipi ne olacak?
 2. “**then**” ve “**else**” terimleri nasıl belirlenecek?
 3. İç içe geçmiş seçicilerin anlamları nasıl belirlenecek?

7.2.2.1. İkili ve Tekli Seçim Deyimleri

Tekli Seçim Deyimi

if (kosul ifadesi) then deyim_1



- **Tekli Seçim Deyimi:**
- Tekli seçim deyiminde sadece bir *if* deyimi ve koşul doğru olduğunda işlenecek deyim ya da deyimler yer alır.

7.2.2.1.1. Yuvalanmış *if* Deyimi

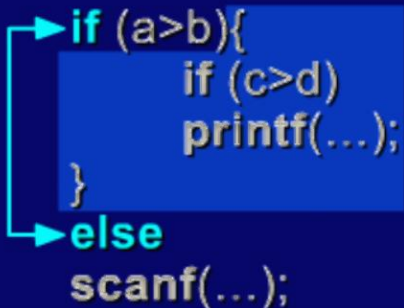
Örnek

```
if (a>b) then  
    if ( b>c)  
        then sonuc := 0  
    else sonuc:=1
```

- İki-yollu seçim deyimleri iç içe yuvalandığında, *else* deyiminin hangi *if* deyimine ilişkin olduğunun belirlenmesi güçleşir ve *sallanan-else (dangling else)* problemi oluşabilir.
- Örneğin yandaki şekilde görülen yapıda, *else* deyiminin hangi *if* deyimine ilişkin olduğu belirli değildir. Bunu belirlemek için dile ilişkin anlam kuralları tanımlanmalıdır.

7.2.2.1.1. Yuvalanmış *if* Deyimi

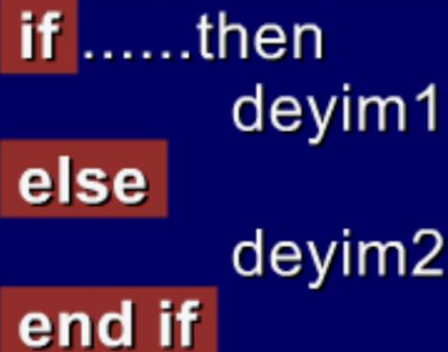
- **Birleşik Deyimler:**
- Programlama dillerinin (JAVA) durağan anlam kuralları, her ***else*** deyiminin kendisine en yakın eşleşmemiş ***then*** deyimi ile eşleştirilmesini gerektirir.
- Ancak yuvalanmış ***if*** deyimleriyle ilgili sorunun çözümü için ikinci ***if-then*** yapısı birleşik deyim yapılmalıdır.
- Birleşik deyimler, bir dizi deyimin tek bir deyime soyutlanmasını sağlarlar. Bu amaçla Pascal'da ***begin ... end*** yapısı, C, C++, C'dâ ise ***{ ... }*** yapısı kullanılmaktadır.



```
if (a>b){  
    if (c>d)  
        printf(...);  
}  
else  
    scanf(...);
```

7.2.2.1.2. *If* Deyimlerinin Sonunun Belirtilmesi

- C ve Pascal'da *if* deyimi sözdizimi, *then* veya *else*'den sonra tek bir deyim yer almasını, daha çok deyim bulunması durumunda ise birleşik deyim oluşturulmasını gerektirir.
- Bu sözdizimdeki eksiklik, örneğin C'de birleşik deyimlerin kapanışı için kullanılan "*}*" unutulsa bile, derleme sırasında sadece eksik "*}*" uyarısı verilmesidir.
- Bu sorunun çözümü için *if- then- else* yapısının sonunu belirten özel bir kelime kullanılmalıdır. Yandaki şekilde, çeşitli programlama dillerinde sağlanan yapı görülmektedir.
- QuickBASIC'te *then* veya *else*'den sonra bulunacak deyim, tek bir deyim ise *end if* kullanılmasına gerek yoktur.



The diagram shows a blue rectangular box with a white border. Inside, the text is arranged in a structured, indented manner. The word 'if' is in a red box, followed by '.....then'. Below this, 'deyim1' is indented. Then 'else' is in a red box, followed by 'deyim2' indented. Finally, 'end if' is in a red box at the bottom left, aligned with the 'if'.

```
if .....then
    deyim1
else
    deyim2
end if
```

- İç içe seçiciler

- Örneğin Java:

```
if (toplam==0)
    if (sayi==0) ...
    ...
else ...
```

- **else** hangi **if**'e bağlanır?

- İlk **if** ile aynı hizada, ona mı bağlı?

- Java'nın statik anlam kuralı: **else** en yakın **if**'e gider.

- Perl'de bu problem yoktur çünkü komut kısımları mutlaka kıvrık parantezler içine konur.

- Else'in bağlı olduğu if kesinleştirilmek isteniyorsa:

```
if (toplam==0) {
    if (sayi==0) ...
}
else ...
```

■ FORTRAN 90 ve Ada çözümü – özel kelime ile bitirmek

□Örneğin Ada:

```
if ... then
    if ... then
        ...
    else
        ...
    end if
else
    ...
end if
```

```
if ... then
    if ... then
        ...
    end if
else
    ...
end if
```

□Avantaj: Okunabilirlik

7.2.2.2. Kısa Devre Değerlendirme

- Bir *if* deyiminde bulunan koşul ifadesi, basit bir ilişkisel ifade yerine birleşik bir ifade ise ***tam değerlendirme*** ve ***kısa devre değerlendirme*** olmak üzere iki tür değerlendirme olasılığı vardır.
- Tam değerlendirmede, ifadedeki her bileşen ayrı ayrı değerlendirilirken, kısa devre değerlendirmede, ifadenin sonucunun elde edilmesi için tek bir bileşenin sonucu yeterli ise, diğer ifadeler değerlendirilmez.
- Bir ifadede operant/operatörlerin tüm hesaplamalarını yapmaksızın sonucun bulunmasıdır.

- Örnek: $(13 * a) * (b / 13 - 1)$

Eğer $a == 0$ ise , diğer kısmı hesaplamaya gerek yok $(b / 13 - 1)$

- Kısa Devre Olmayan Problem

```
index = 0;
```

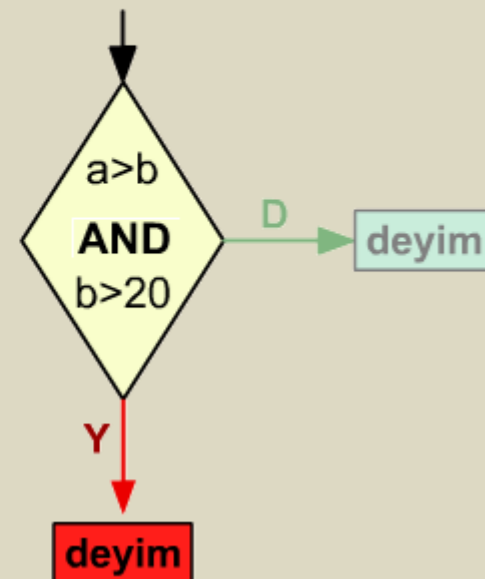
```
while (index <= length) && (LIST[index] != value)  
    index++;
```

$index=length$ **olduğunda**, $LIST[index]$ **indeksleme problemi ortaya çıkaracak** ($LIST$ dizisi $length - 1$ uzunluğunda varsayılmış)

7.2.2.2. Kısa Devre Değerlendirme

If (a > 10) **AND** (b > 20) then

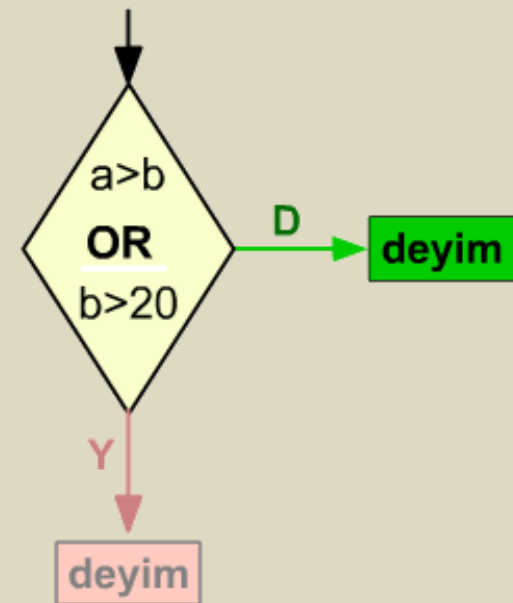
- $a > 10$ yanlış ise sonuç YANLIŞ
- if deyiminde, AND mantıksal işlemcisi kullanıldığı için, birinci ifadenin yanlış olması, sonucun yanlış olacağını belirlemektedir. Bu nedenle $a > 10$ ifadesinin yanlış sonucu vermesi durumunda, $b > 20$ ifadesinin değerlendirilmesine gerek kalmamaktadır.



7.2.2.2. Kısa Devre Değerlendirme

If (a > 10) **AND** (b > 20) then
 OR

- $a > 10$ doğru ise sonuç DOĞRU
- Birleşik ifadede OR işlemcisi kullanılırsa, birinci ifadenin *doğru* sonucu vermesi, ifadenin sonucunu belirlemek için yeterli olacaktır.



7.2.2.2. Kısa Devre Değerlendirme

- Bazı birleşik ifadelerin değerlendirilmesi, kısa devre değerlendirmeyi gerektirebilir.
- Örneğin, birleşik bir ifadedeki ikinci ilişkisel ifade için, birinci ifadenin sonucu önemli olabilir. Aşağıdaki *if* deyimi bu durumu örneklemektedir:

if (a != 0) **and** (36 / a > 2)



DOĞRU

ise ikinci ifadedeki bölme gerçekleşir. (36/a>2)

YANLIŞ

ise ikinci ifade değerlendirilmez.

7.2.2.2. Kısa Devre Değerlendirme

- Görüldüğü gibi *if* deyiminin koşulunun ikinci bölümündeki bölme işleminin yapılabilmesi için, koşulun birinci bölümündeki ilişkisel ifadesinin doğru olması yani *a* değişkeninin değerinin sıfırdan farklı olması gereklidir.
- Kısa devre değerlendirmenin uygulandığı durumlarda, birinci ifadeden sonraki ifadelerin değerlendirilmemesinin bütün yapı üzerinde etkisi olup olmayacağı önemlidir.

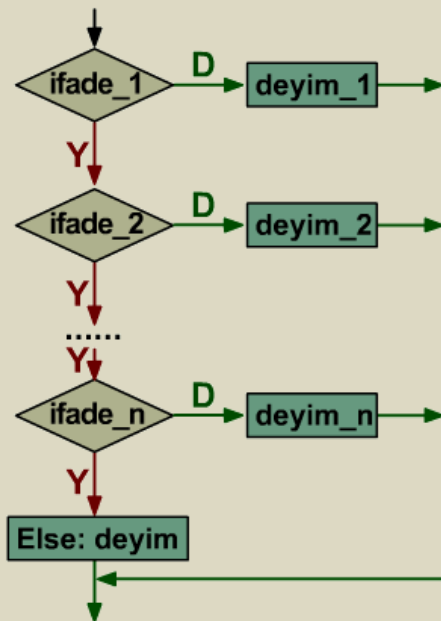
7.2.2.2. Kısa Devre Değerlendirme

- Örneğin C'de; ***if*** ($a \neq 0$) ***&&*** ($28 / a++ > 2$)... ***&&***-> *kısa devre- and* gibi bir deyimde, kısa devre değerlendirmenin, ***a*** değişkeni üzerindeki etkisi göz önüne alınmalıdır. Bu durumda, ***if*** deyiminin her işleyişinde, koşulun ikinci bölümü değerlendirilmeye bilineceği için, ***a*** değişkeninin değeri, ***if*** deyiminin her işleyişinde artırılmayacaktır.
- Pascal ve C'nin çoğu gerçekleştirimi, kısa devre değerlendirmeyi uygulamaktadır.
- Ada'da ise ***and then*** ve ***or else*** olmak üzere kısa devre değerlendirme için ayrı işlemciler tanımlıdır.

- C, C++, ve Java: kısa devre tespiti için bütün mantıksal operatörler(&& ve | |) için yapar, ama bit düzeyinde mantıksal operatörler(& and |) için yapmaz.
- Ruby, Perl, ML, F#, ve Python'da tüm mantık operatörleri için kısa devre tespiti yapılır.
- Ada: Programcının isteğine bağlıdır(kısa devre '**and then**' ve '**or else**' ile belirtilir)
- Kısa devre tespiti ifadelerdeki potansiyel yan etki problemini ortaya çıkarabilir
örnek. $(a > b) \quad || \quad (b++ \quad / \quad 3)$
- $a > b$ olduğu sürece b artmayacak

7.2.2.3. Çoklu Seçim Deyimi

- Bir programdaki akışı belirlemek için ikiden fazla yol olduğu zaman **çoklu seçim deyimi** kullanılır.
- Pascal' da Çoklu Seçim Yapısı:** Pascal'da çoklu seçim, *case* deyimleri ile ifade edilir. Çoklu seçim yapısının anlamı, *ifade* nin eşleştiği ilk *sabit_ifade* bulunduktan sonra, o *sabit_ifadesine* ilişkin deyim ya da deyimlerin çalıştırılması ve daha sonra akışın *case* yapısının dışına çıkmasıdır.

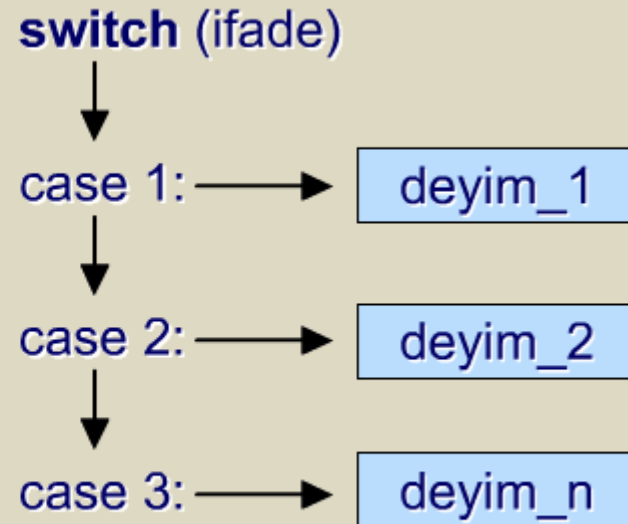


```
case ifade of
  sabit_ifade_1: deyim-1;
  sabit_ifade_2: deyim-2;
  .....
  sabit_ifade_n: deyim-n;
  else .....
end
```

7.2.2.3. Çoklu Seçim Deyimi

- **C' de Çoklu Seçim Yapısı:**
- Çoklu seçim yapısı için farklı bir tasarım, C'de yer almaktadır. C'deki tasarımda akış denetimi, bir *case* sabiti ile eşleme olsa da olmasa da, bir sonraki *case sabiti'* nin sınanması ile devam eder. Eğer, bir eşleme gerçekleştikten sonra diğerlerinin sınanması istenmiyor ve akışın çoklu seçim yapısının dışına çıkması isteniyorsa eşleme grubunun son deyimi olarak *break* deyimi kullanılmalıdır.

```
switch (ifade){  
    case sabit_ifade_1: deyim_1;  
    break;  
    case sabit_ifade_2: deyim_2;  
    break;  
    case sabit_ifade_n: deyim_n;  
    [default: deyim_n+1]  
}
```



7.2.2.3. Çoklu Seçim Deyimi

- Tasarımla ilgili hususlar:
 1. Kontrol ifadesinin tipi ve şekli nasıl olacak?
 2. Seçilebilir bölümler nasıl belirlenecek?
 3. Programın çoklu yapıdaki akışı sadece bir bölge ile mi sınırlı olacak?
 4. Seçimde temsil edilmeyen ifadelerle ilgili ne yapılacaktır?
- Tasarım yaparken C'nin switch kodu seçilirse
 1. Kontrol ifadeleri yalnızca tamsayı olabilir.
 2. Seçilebilir segmentler komut dizileri, bloklar veya bileşik komutlar olabilir.
 3. Herhangi bir segment numarası çalıştırılabilir bir yapı olabilir.
 4. `Default` cümlesi tanımlanmayan değerler için kullanılır.

❑ Erken çoklu seçme komutları:

1. FORTRAN arithmetic IF(üçlü seçme komutu)

IF(<aritmetik ifade>) N1, N2, N3

❑ Kötü özellikleri:

– Kılıflanmamış (not encapsulated); seçilebilir bölgeler programın içinde her yerde olabilir.

– GOTO gerekir.

2. FORTRAN hesaplanmış GOTO ve atanmış GOTO

- Modern çoklu seçme komutları

1. Pascal durumu (Hoare'in ALGOL W'ya katkısı)

case<ifade>of

SabitListe_1 : Komut_1;

...

SabitListe_n : Komut_n

end

- Tasarım seçimleri (Pascal):
 1. Seçme ifadesi herhangi bir sıra tipinde (int, boolean, char, enum) olabilir.
 2. Seçilen bölgede tek komut ta olabilir, bir grup komut ta.
 3. Yapının her yürütülmesinde bir bölgesi kullanılır.
 4. Pascal'da (Wirth), karşılığı olmayan seçme yapıldığında sonuç belirsizdir, 1984 ISO Standardına göre bu işlem hata mesajı verir.
 5. Birçok dilde bu durumlar için otherwise veya else terimleri konulmuştur.

2. C, C++ ve Java **switch**

switch (<ifade>)

```
{  
    Sabitİfade_1 : ifade_1;  
    ...  
    Sabitİfade_n : ifade_n;  
    [default:ifade_n+1]  
}
```

- Tasarım seçimleri: (**switch** için)
 1. Kontrol ifadesi sadece tam sayı tipi olabilir.
 2. Seçilebilir kısımda da birkaç komut peş peşe veya bir blok içinde olabilir.
 3. Birden çok kısım peş peşe çalıştırılabilir. Seçilebilir kısımların sonunda örtülü bir sapma yok. Açıkça "break" komutu ile "switch" sonuna gidilebilir
 4. default terimi tanımlanmamış kontrol ifadesi değerleri içindir; eğer default olmazsa uygun kontrol değeri çıkmaması durumunda switch bir şey yapmaz.

3. C# Tasarım seçimleri: (switch için)

1. C gibidir, sadece birden çok kısmın yürütülmesine izin vermez.
2. Her kısmın mutlaka "break" veya "goto" ile sonlandırılması gerekir.

```
switch (indeks) {  
    case1: gotocase3;  
    case3: tek +=1;  
           toplamtek+= indeks;  
           break;  
           case2: gotocase4;  
    case4: cift+= 1;  
           toplamcift+= indeks;  
           break;  
    default: Console.WriteLine("switchiçinde hata, indeks =  
           %d\n", indeks);  
}
```

4. Ada'nın **case**'i Pascal'ın **case** 'ine çok benzer, şu farkla ki:

1. Sayısal değişkenlere (Integer, Boolean, Char, Enum) ek olarak aşağıdaki tipleri de kabul eder:
 - Alt değer kümeleri, örneğin: 10..15.
 - Mantıksal OR işleçleri,
örneğin: 1..5 | 7 | 15..20.
2. Seçme ifadesi sabitlerinin tamamen kapsanması gerekir:
 - Genellikle **others** terimi ile sağlanır.
 - Bu komutu güvenilir yapar.
3. Seçilen komut grubu bitince case komutunu takip eden komut ile program devam eder. "break" veya "goto" kullanılmaz.

```
case <ifade> is
  when SabitListe_1 => Komut_1 grubu;
  ...
  when SabitListe_n => Komut_n grubu;
  [when others => Komut_diğer grubu;]
end case;
```

5. PHP Tasarım seçimleri: (**switch** için)

1. C gibidir, sadece indeks ifadesi "string" "integer" veya "double" olabilir..
2. Yürütülmesi sırasında indeks değerine tam uyan case sabiti aranır.
Hiçbiri uymazsa switch bir şey yapmadan bir sonraki komuta devreder..

```
switch (indeks) {  
    case 1: goto case 3;  
    case 3: tek +=1;  
            toplamtek += indeks;  
            break;  
    case 2: goto case 4;  
    case 4: cift += 1;  
            toplamcift += indeks;  
            break;  
    default: printf("switch içinde hata, indeks = %d\n", indeks);  
  
}
```

- Birçok durumda switch veya case komutları yetersizdir:
 - Örneğin basit bir sabit yerine mantıksal bir ifade gerektiğinde.
- Çok seçimler iki yollu seçme komutlarının genişletilmesi ile de elde edilebili (ALGOL 68, FORTRAN 90, Ada)

Ada

```
if ...  
    then ...;  
elsif ...  
    then ...;  
else ...;  
end if;
```

Ada

```
if ...  
    then ...;  
else  
    if ...  
        then ...;  
    else ...;  
    end if;  
end if;
```

C

```
if (...)  
    ...;  
else if (...)  
    ...;  
else ...;
```


- Ada'nın çoklu seçme komutu:
 - İç içe geçmiş if'lerden çok daha kolay okunabilir.
 - Her aşamada mantıksal değerlendirme yapılmasını sağlar.

7.2.2.3.1. Case Yapısının Ortak Noktaları

- Bir önceki sayfada görülen yapı, çeşitli programlama dilleri arasında farklılık göstermekle birlikte aşağıdaki ortak noktaları içerir:
 - ① İki-yollu bir seçim deyimi için, sadece iki olası değeri olan bir koşul ifadesi gerekiyken, çoklu seçim yapılarında akış yolu olasılıkları daha fazla sayıda olduğu için, seçimin dayandığı ifade farklı niteliktedir. *İfade*, sıralı bir tipte olmalıdır.
 - ② Her sabit bölümü, tek bir sabit veya bir sabit aralığı içerebilir. *0...9* veya *"a".. "z"* gibi aralıklar case sabiti olarak tanımlanabilir.
 - ③ Case yapısında yer alan sabitler ardışık olmak zorunda değildir. Yani, "A" için yazılan bir case deyiminden sonra "B" için bir case deyimi olmadan "C" için bir case deyimi bulunabilir.
 - ④ Case yapısındaki sabitler özgün olmalıdır. Eğer sabitler arasında çakışma olursa, bu hata derleme zamanında yakalanabilir.

7.2.3. Yineleme Yapıları

- Yineleme yapıları, bir programda yer alan tek bir deyimin veya bir dizi deyimin hiç çalıştırılmamasını, bir kez çalıştırılmasını veya daha çok kez çalıştırılmasını sağlayan **döngüler** (*loop*) oluştururlar.
- Yinelemenin, mantıksal bir ifadenin değeri ile denetlenmesi ile **mantıksal denetimli döngüler** oluşturulurken, yinelemenin, bir sayaç değerinin belirli bir değere ulaşması ile denetlendiği durumda, **sayaç denetimli döngüler** oluşur.

7.2.3. Yineleme Yapıları

- Tasarım Sorunları:
 1. Döngü değişkeninin tipi ve kapsamı nedir?
 2. Döngü değişkeninin döngü bittiğinde değeri nedir?
 3. Döngü değişkeninin değeri döngü içinde değiştirilebilmeli midir? Değiştirilebilirse bu döngü kontrolünü etkilemeli midir?
 4. Döngü parametreleri bir kez mi değerlendirilmelidir yoksa her döngüde mi?

7.2.3. Yineleme Yapıları

Önce Sınanan Döngü

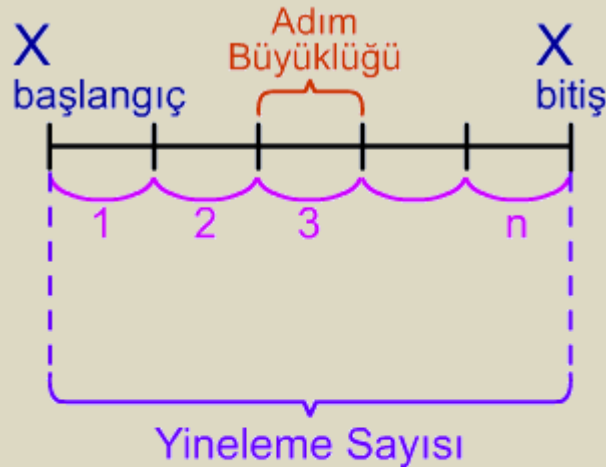


Sonra Sınanan Döngü



- Bir döngüdeki denetim mekanizması döngünün başında gerçekleştirilirse, **önce sınanan** (*pretest*) döngü, döngünün sonunda gerçekleştirilirse, **sonra sınanan** (*posttest*) döngü olarak nitelendirilir. Döngünün başı ve sonu arasındaki deyimler ise döngünün gövdesi olarak adlandırılır.
- Bir döngüdeki yinelemenin denetlenme türüne göre (*mantıksal, sayaç-denetimli veya birleşimi*) ve denetim mekanizmasının döngüdeki yerine (*başlangıçta veya sonda*) göre, programlama dillerinde çeşitli tasarımlar geliştirilmiştir.

7.2.3.1. Sayaç Denetimli Döngüler



X: Döngü Değişkeni

- Bir sayaç denetimli döngüde, **döngü değişkeni** adı verilen bir değişken sayaç değerini gösterir.
- Bu sayaç değerinin **başlangıç değeri**, **bitiş değeri** ve ardışık iki değeri arasındaki farkı gösteren **adım büyüklüğü**, bir sayaç denetimli döngü gövdesinin kaç kez yineleneneceğini belirler.
- Başlangıç değerinden başlayan döngü değişkeni, her yineleme için, adım büyüklüğü değerine göre artırılacak veya azaltılacaktır.

7.2.3.1.1. FORTRAN IV

- FORTRAN IV'da sayaç denetimli döngülerin genel şekli aşağıda belirtilmiştir. FORTRAN IV'da başlangıç, bitiş ve adım değerleri için sadece tamsayı sabitler ve değişkenler kullanılabilir.

DO etiket **değişken** = başlangıç, bitis [adım]

DO 300 **I = 1** , 20

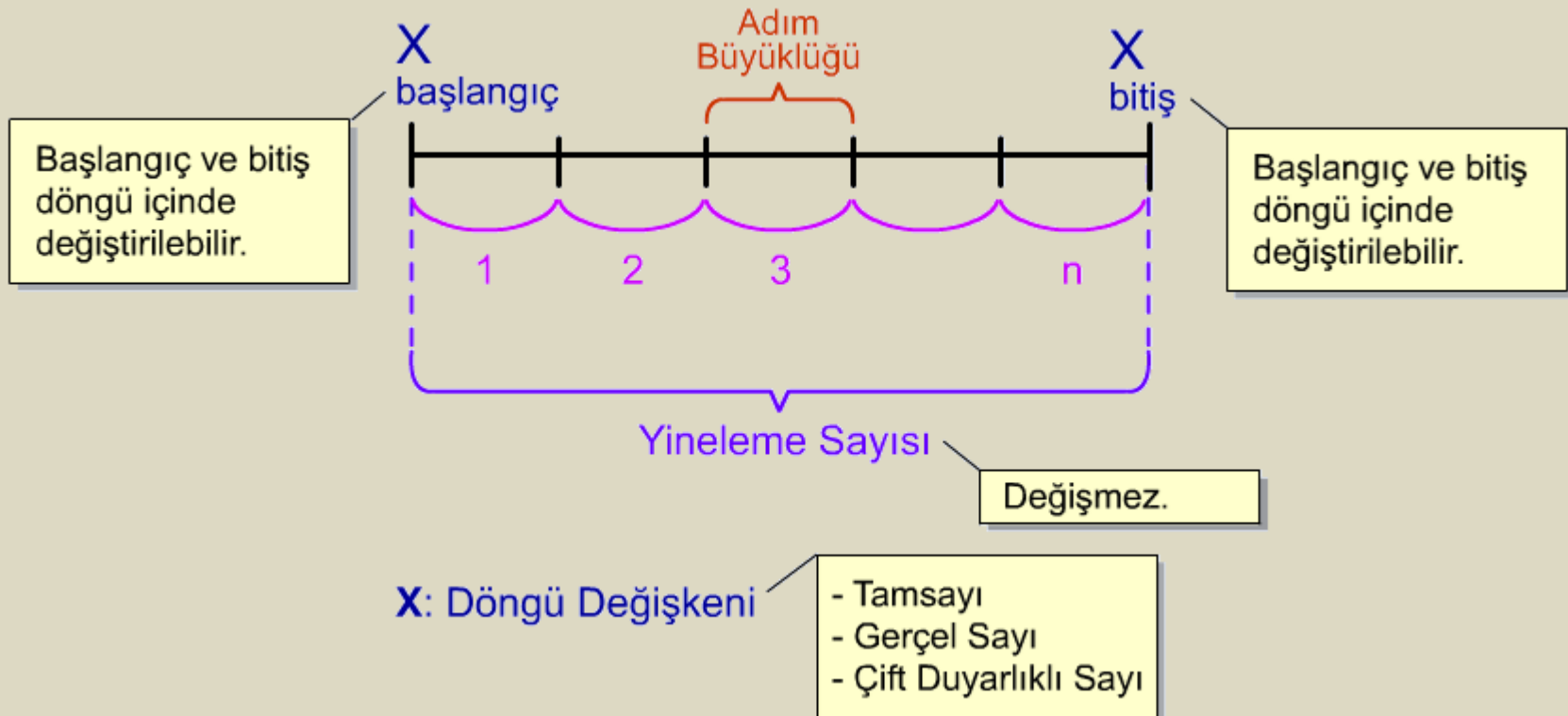
Döngü değişkeninin değeri her zaman, küçükten büyüğe doğru ilerlemelidir.

Döngü değişkeninin değeri, döngü gövdesinde değiştirilemez ve adım büyüklüğü varsayılan olarak 1 değerindedir.

7.2.3.1.2. FORTRAN77 ve FORTRAN90

- FORTRAN77 ve FORTRAN90'da sayaç denetimli döngü değişkeni, tamsayı, gerçel sayı veya çift duyarlıklı sayı olabilir. Başlangıç, bitiş ve adım değerleri, sabit ve değişkenlerin yanı sıra ifade olabilir ve bu değerler bir kez döngünün başında değerlendirildikten sonra döngünün kaç kez yineleneyeceği hesaplanır.
- Dolayısıyla, döngü içinde bu değerler değiştirilse bile, döngünün kaç kez yineleneyeceği belirlenmiş olduğu için, döngünün yinelenme sayısı etkilenmez.

7.2.3.1.2. FORTRAN77 ve FORTRAN90



7.2.3.1.3. ALGOL 60

- ALGOL 60'da sayaç denetimli döngü değişkeni, tamsayı veya gerçel sayı tipinde olabilir. ALGOL 60'daki döngü tasarımının önemli özelliklerinden birisi, başlangıç, bitiş ve adım değerlerinin döngü içinde değiştirilebilmesi ve döngü bitiş ve adım değerlerinin her yinelemede yeniden değerlendirilmesidir.

<for_deyimi>->for deg:=<eleman>{,<eleman>} do <deyim>

<eleman> -><ifade>

|<ifade> step <ifade> until <ifade>

|<ifade> while <mantıksal_ifade>

7.2.3.1.3. ALGOL 60

- Yukarıda ALGOL 60'daki sayaç denetimli döngü tasarımının BNF tanımı görülmektedir. Bu tasarımda, bir sayaç denetimi ve bir mantıksal ifade birleştirilebilmektedir. Çok işlevli bir döngü oluşturmayı amaçlayan bu tasarım, esneklik sağlamakla birlikte karmaşıklığa neden olmaktadır.

```
for sayac :=1,7,86,110 do  
    liste[sayac]:= 0
```

```
for sayac := 1 step 1 until 22 do  
    liste[sayac] :=0
```

```
for j=3, j+1 while (sayac<=36) do  
    liste[j]:=0
```

```
for k:=1,9,28 step 2 until 38, 4 * k while (k<280) do  
    toplam := toplam + k
```

7.2.3.1.4. ADA

```
Say : Float := 1.35;  
for Say in 1..10 loop  
  toplam=toplam+Say;  
end loop;
```

- Ada
for var **in** [**reverse**] `discrete_range` **loop**
...
end loop
- Tasarım Seçenekleri:
 1. Döngü değişkeni var'ın tipi `discrete_range`'in tipidir. Döngü içinde tanımlıdır (örtülü tanımlama).
 2. Döngü değişkeni döngü dışında tanımlı değil.
 3. Döngü değişkeni döngü içinde değiştirilemez fakat `discrete_range` değiştirilebilir; bu döngüyü etkilemez.
 4. `discrete_range` bir kez hesaplanır.
 5. Döngüye sadece başından girilebilir..

7.2.3.1.5. Python

- Python

for döngü değişkeni **in** nesne:

- döngü gövdesi

[**else**:

- else cümlesi]

- Nesne sıklıkla bir aralığı(**range**) temsil eder. Liste değerleri ise parantez içinde (`[2, 4, 6]`), yada **range** fonksiyonu kullanılarak ifade edilir (**range**(5), 0, 1, 2, 3, 4 değerlerin döndürür.
- The loop variable takes on the values specified in the given range, one for each iteration Döngü değişkeni aralıkta (range) gösterilen değerleri alırlar.
- Döngü normal bir şekilde sona ererse isteğe bağlı olarak else bloğu yürütülür.

7.2.3.1.5. F#

- F#

- Sayaçları değişkenler gerektiren ve fonksiyonel dillerde değişkenleri yok olduğundan, sayaç kontrollü döngü özyinelemeli fonksiyonlar ile simüle edilmelidir.

```
let rec forLoop loopBody reps =  
    if reps <= 0 then ()  
    else  
        loopBody()  
        forLoop loopBody, (reps - 1)
```

forloop adında loopbody parametrelerini kullanan recursive bir fonsiyon tanımlanır.

- () Hiçbir olay gerçekleşmiyor ve hiçbir değer dönmüyor anlamına gelmektedir.

7.2.3.1.6. Pascal

- Pascal'daki sayaç denetimli döngü tasarımı aşağıda görülmektedir:
- ***for** deg := ilk_değer (**to** / **down to**) bitiş_değeri **do** deyim*
- Pascal'da döngü değişkeninin sıralı bir tipte olması gerekir ve döngü değişkeninin değeri döngü içinde değiştirilemez. Döngü değişkeninin başlangıç ve bitiş değerleri ise döngü içinde değiştirilebilirler, ancak sadece döngünün başında bir kez değerlendirildikleri için döngünün çalışmasını etkilemezler.

7.2.3.1.6. Pascal

- Örneğin aşağıdaki şekilde görülen Pascal programında, döngü gövdesi içinde *ilk* ve *son*'a yapılan değişiklikler, döngünün başlangıçta belirlenen yineleme sayısını etkilemez ve döngü, ilk başta belirlendiği gibi 10 kez yinelenir.
- Aşağıda Pascal'da sayaç-denetimli döngü örneklenmektedir:

```
ilk:=1;  
son:=10;  
for k:=ilk to son do → Döngü 10 kez yinelenir.  
begin;  
  writeln(k, ilk, son);  
  ilk:=5  
  son:=30 } → İlk ve son'un değiştirilmesi yineleme sayısını  
end;
```


7.2.3.1.7. C

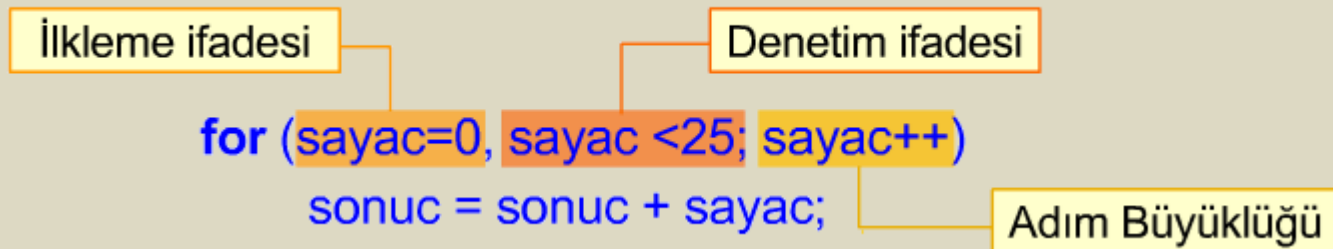
- C'deki sayaç denetimli döngü tasarımında belirli bir döngü değişkeni ya da başlangıç, bitiş ve adım değerleri yoktur. Döngünün genel şekli aşağıda görülmektedir:

for (ifade_1; ifade_2; ifade_3)

- *ifade_1* ilkleme ifadesi olup *for* deyiminde sadece bir kez değerlendirilir.
- *ifade_2*, döngü denetim ifadesi olup, döngü gövdesinin her çalıştırılışından önce değerlendirilir.
- *ifade_3* ise, adım büyüklüğünü ve döngü gövdesini oluşturmaktadır.

7.2.3.1.7. C

- Örneğin, aşağıdaki for deyiminde, *sayac=0* ifadesi, döngü için ilkleme ifadesi, *sayac<25* ifadesi, döngünün sona ermesi için gerekli denetim ifadesini ve *sayac++* ise adım büyüklüğünü göstermektedir.



- C'deki tasarımda döngü değişkenin ilklenme ifadesi, adım ifadesi ve bitiş ifadesi seçimliktir. Bu nedenle bu ifadeler, bir *for* deyiminde bulunmayabilir. Örneğin, *ifade_2* bir *for* deyiminde bulunmazsa, döngü sonu denetim yapılamayacağı için sonsuz döngü oluşur.

7.2.3.1.7. C

Örnek yazılım formatları:

- `for (k=1;k<50; k+=2)`
- `for (k=5;k<=n; k++)`
- `for (x=50;x>10;x--)`
- `for (;x<10;x++)` /* başlangıç değeri daha önce atanmış olmalı */
- `for (x=2;x<n;)` /* x döngü sayacı döngü içinde değiştirilmeli */

7.2.3.1.7. C

- C'deki tasarımda tanımlı bir döngü değişkeni olmadığı için tüm değişkenler, döngü içerisinde değiştirilebilirler.
- C'deki *for* döngü tasarımının bir diğer özelliği de, döngünün devamını belirleyen sına ifadesinin tek bir ifade olması gerekliken, diğer ifadeler için böyle bir sınırlama olmamasıdır..

```
for (sayac =0; toplam =0; sayac < 100; sayac++){  
    toplam = toplam +sayac;  
    printf ("sayac = %d, toplam =%d \n", sayac, toplam);  
}
```

- **C++** iki şekilde C ile farklılık gösterir :
 1. Kontrol ifadesi Boolean olabilir.
 2. Başlangıç ifadesi değişken tanımları içerebilir.
- **Java ve C#**
 - C++'tan farkı, kontrol ifadesinin Boolean (mantıksal) olma zorunluluğudur. Java döngüye ortadan girişe izin vermezken C# verir.

7.2.3.2. Mantıksal Denetimli Döngüler

- Bir mantıksal denetimli döngüde, döngü gövdesinin yinelenme kararı bir mantıksal ifadenin değeri ile belirlenir. Bu nedenle mantıksal denetimli döngüler, sayaç denetimli döngülerden daha basit ve daha genel amaçlı bir yapı sağlarlar.
- Mantıksal denetimli döngüler, hem önce sınanan hem de sonra sınanan özellikte olabilirler. Günümüzde popüler olan programlama dillerinin çoğu, hem **önce sınanan** hem de **sonra sınanan** özellikte mantıksal denetimli döngüler içermektedirler. Sonra sınanan döngülerde, mantıksal ifade sınanmadan önce döngü değişkeni bir kez çalıştırılacağı için, kullanımlarına dikkat edilmelidir.

7.2.3.2. Mantıksal Denetimli Döngüler

C

QuickBasic

Pascal

Ada

C'de mantıksal denetimli döngüler için kullanılan yapı aşağıdaki şekilde görülmektedir.

```
while (mantıksal_ifade) } önce sınanan  
deyim
```

```
do  
deyim  
while (mantıksal_ifade) } sonra sınanan
```

7.2.3.2. Mantıksal Denetimli Döngüler

C

QuickBasic

Pascal

Ada

QuickBASIC'te mantıksal denetimli döngüler için dört yapı sağlanmıştır. Böylece hem önce sınanan hem de sonra sınanan mantıksal döngüler için, koşul doğru olduğu sürece ve koşul doğru olana kadar çalıştırılacak yinelemeli yapılar sağlanmıştır.

	Koşul Doğru Olduğu Sürece	Koşul Doğru Olana Kadar
önce sınanan	Do While mantıksal_ifade Loop	Do Until mantıksal_ifade Loop
sonra sınanan	Do Loop While mantıksal_ifade	Do Loop Until mantıksal_ifade

7.2.3.2. Mantıksal Denetimli Döngüler

C

QuickBasic

Pascal

Ada

Pascal'da, koşul doğru olana kadar döngü gövdesinin yinelendiği **repeat....until** yapısı tanımlıdır.

Repeat

.....

Until mantıksal_ifade

7.2.3.2. Mantıksal Denetimli Döngüler

C

QuickBasic

Pascal

Ada

Ada'da, işletim sistemleri gibi özel yazılımların gerek duyabileceği sonsuz çalışma özelliğinde döngüler tanımlanması için bir yapı tanımlanmıştır.

```
loop
.....
end loop
```

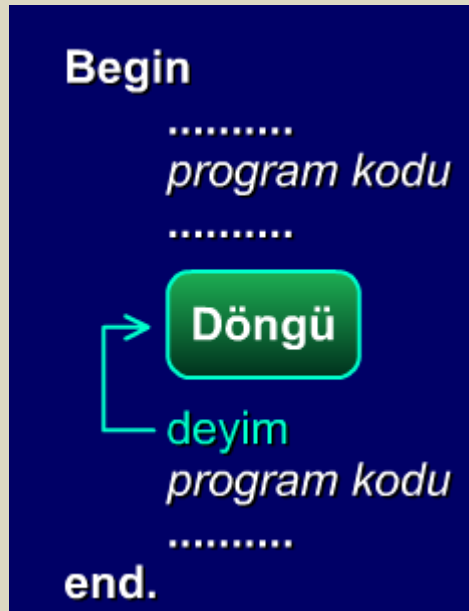
7.2.3.2. Mantıksal Denetimli Döngüler

- **Java** kontrol ifadesinin Boolean olma zorunluluğu dışında C ve C++'a benzer (Ve döngü yapısına sadece başlangıçta girilir. Java'da `goto` deyimi yoktur. C# ta ise döngü yapısına ortadan da girilebilir.)
- **F#**
 - Sayaç denetimli döngülerde olduğu gibi mantıksal denetimli döngülerde de recursive fonksiyonlar kullanılır.

```
let rec whileLoop test body =  
    if test() then  
        body()  
        whileLoop test body  
    else ()
```

- Whileloop özyinelemeli fonksiyonu test ve body parametreleriyle tanımlanır. Test parametresi mantıksal kontrolü yapar body ise kontrolün doğru olduğu durumdaki yapılacak işlemi temsil eder.

7.2.4. Akış Denetimini Koşulsuz Değiştirme



- Yapısal programlama için, bir program içinde deyimlerin akışı denetlenmelidir. Ancak akış denetiminin değiştirilmesini sağlayan deyimler, yapısal programlama ilkelerine uymayan yapılar içerebilirler.
- Programlama dillerinde yer alan *goto* deyimi ve döngülerden erken çıkış için veya döngünün bir geçişinin normalden önce tamamlanması için kullanılan deyimler, bu deyimlere örnek oluşturmaktadır.

7.2.4.1. *goto* Deyimi

- *goto* deyimi, bir programda akış denetimini koşulsuz olarak değiştirmeyi sağlayan deyimdir.
- Akışı yönetmek için güçlü bir deyim olmakla birlikte, akışı koşulsuz olarak değiştirme, bir programdaki deyimlerin sırasını rasgele olarak belirleyebildiği için, sorunlara yol açmaktadır.
- Programlama dilleri alanındaki araştırmalar, *goto* deyiminin programların okunabilirliğini ve güvenilirliğini azaltarak, bakım aşamasını ve programların etkin çalışmasını güçleştirdiğini göstermiştir.
- (Bunu ifade eden bir benzeştirmede *goto* deyiminin yer verildiği programlar, *sphagetti kodu* olarak nitelendirilmiştir)



7.2.4.1. *goto* Deyimi

Begin

.....
program kodu
.....

Döngü

GOTO

program kodu

end.

- Bugüne kadar geliştirilen genel amaçlı dillerin çoğunda *goto* deyimine yer verilmekle birlikte, *goto* deyiminin yol açtığı sorunlar nedeniyle, günümüzde popüler olan dillerin *goto* deyimine yaklaşımları, *goto* deyimine dilde yer vermek, ancak kullanımını kısıtlayan bir tasarım uygulamaktır.
- *goto* deyiminin kullanımını kısıtlayan programlama dillerine örnek olarak Pascal ve C verilebilir.
- Java'da yoktur.
- C#'ta *goto* komutu kullanılabilir (switch bloğuyla beraber).
- Döngü çıkış komutları (break, last) kamufle edilmiş *goto*'lardır. Ancak çıkış döngünün hemen sonuna olduğundan okunabilirliği ve güvenliği bozmazlar.

Etiket şekilleri:

- 1. İşaretsiz tam sayı sabitler: Pascal (üst üste nokta ile) FORTRAN (üst üste noktasız)
- 2. Üst üste nokta ile değişkenler: ALGOL 60, C
- 3. Değişkenler<< ... >>içinde: Ada
- 4. Değişkenler etiket: PL/I
 - Bu değişkenlere değer atanabilir ve parametre olarak alt programlara geçirilebilirler.
 - Çok esnek yapar ama aynı ölçüde de okunamaz ve gerçekleştirilemez yapar.

7.2.4.2. Döngüler için Aktarma Deyimleri

Exit, Break

Normalden önce çıkışı sağlar.

- Yapısal programlama kuralları, bir döngünün tek bir başlangıç ve tek bir çıkış noktası olmasını gerektirir. Ancak döngülerden normalden önce çıkış veya döngünün bir yinelemesinin bırakılarak, yeni bir yinelemeye başlanması, döngülerde gerek duyulan bir işlemdir.

Continue, Cycle

Döngü içinde bir bölümün atlanmasını sağlar.

- Goto* deyimlerinin yol açtığı sorunlar nedeniyle birçok programlama dilinde, bu gibi durumlarda aktarımı sağlamak için *goto* deyimleri yerine, mantıksal denetimli veya sayaç denetimli döngülerden normalden önce çıkışı sağlayan (*exit*, *break* gibi) veya döngü içinde bir bölümün atlanmasını sağlayan (*continue*, *cycle* gibi) deyimler tanımlanmıştır.

7.2.4.2.1. QuickBASIC ve Ada'da Döngüler için Aktarma Deyimleri

- QuickBASIC' te *exit* Deyimi:
- QuickBASIC'teki *exit* deyimi, sayaç denetimli veya mantıksal denetimli döngülerden normalden önce çıkış sağlar. Bu çıkış, çoğunlukla döngü içinde oluşan ve koşul ifadesi veya sayacı ile sınırlanmayan bir olay nedeni ile gerçekleşir.
- Aşağıda verilen program parçasında QuickBASIC'te *exit* deyimi ile döngüden erken çıkış örneklenmektedir.

```
for j=1 to 5
  input miktar
  if miktar < 0 then exit for
  toplam = toplam + miktar
end
```

miktar değişkeninin değeri sıfırdan küçük olduğu zaman, *exit* deyim ile *for* döngüsü sona erdirilmektedir.

7.2.4.2.1. QuickBASIC ve Ada'da Döngüler için Aktarma Deyimleri

- **Ada'da *exit* Deyimi:**
- Ada'da mantıksal denetimli döngüler için, sınamanın döngü başından veya döngü sonundan farklı bir yerde yapılmasını sağlayan bir tasarım vardır. Bu tasarımda sınama yeri, döngü gövdesi içinde programcı tarafından belirlenen herhangi bir deyim olabilir. Ada'daki bu yapı aşağıdaki şekilde görülmektedir.
- Ada'daki tasarımda, exit deyimi koşullu olarak veya koşulsuz olarak kullanılabilir.

```
loop
....
exit when koşul;
....
end loop;
```

exit deyiminde belirtilen koşul sağlanırsa (örneğin; `sonuc >= 1000` gibi) döngü sona erer. Aksi durumda devam eder.

7.2.4.2.2. C'de Aktarma Deyimleri

- C' de **break** Deyimi:
- C'de sayaç denetimli veya mantıksal denetimli döngülerden normalden önce çıkış, **break** deyimi kullanılarak sağlanabilir. **break** deyimi çalıştırılır çalıştırılmaz, döngü dışına çıkışı sağlar.
- Aşağıdaki şekilde, **break** deyiminin kullanımı örneklenmektedir. Bu örnekte, *okunan* değişkeninin 2'ye göre *mod* işleminin sonucu sıfırdan büyük olduğu sürece *while* döngüsü yineleneyecektir. Ancak istisnai bir durum, okunan değişkeninin 2'ye göre *mod* işleminin sonucunun sıfırdan büyük olması ve aynı zamanda 7'ye göre modunun sıfır olmasıdır. Bu durumda, **break** deyimi işlenecek ve döngü dışına çıkılacaktır.

```
scanf("%d", &okunan);  
while okunan mod 2 > 0 {  
    if okunan mod 7 = 0 break;  
    toplam = toplam +okunan;  
    scanf("%d", &okunan);  
}
```

özel durum: *okunan* değişkeni *mod* 7'ye göre sıfır olduğunda **break** deyimi ile döngü dışına çıkılır.

7.2.4.2.2. C'de Aktarma Deyimleri

- **C' de *continue* Deyimi:**
- C' de döngülerde akışı değiştirmek için *break* deyimine ek olarak denetimi en içteki döngünün sınaama deyimine aktaran ***continue*** deyimi tanımlıdır. ***continue*** deyimi, döngüyü sona erdirmeden döngü gövdesinde bulunulan noktadan döngü kapanış deyimine kadar olan deyimlerin atlanmasını ve döngünün bir sonraki yinelemeye devam etmesini sağlar. FORTAN90'da ise aynı işlev, *CYCLE* deyimi tarafından sağlanmaktadır.
- Aşağıda verilen C kodu, *continue* deyiminin kullanımını örneklemektedir. Bu örnekte, *sayi* değişkeni için okunan değer, 10'dan büyük ise okunan değer, *gozlemler* değişkenine eklenecektir. *sayi* için okunan değer, 10'dan küçük ise ekleme işlemi yapılmayacak ve döngü bir sonraki yinelemesine geçecektir. Döngü, *gozlemler* değişkeninin değeri 500'den küçük olduğu sürece devam edecektir.

```
while ( gozlemler <500)
{scanf( "%d",&sayi);
if (sayi <10) continue;
gozlemler += sayi;
}
```

özel durum: sayi değeri 10'dan küçük ise döngü bir sonraki yinelemesine geçer.

7.2.4.2.2. C'de Aktarma Deyimleri

- **Aktarma Deyimleri'nin Kullanımı:**
- Programlama dilinde, *break / continue* ya da benzeri işleve sahip deyimlerin yer alması, dilin yazılabilirliğine katkıda bulunur.
- Ancak bu deyimlerin kullanılmaları zorunlu değildir. Örneklerden de anlaşıldığı gibi, *break / continue* deyimlerinin işlevi, yuvalanmış seçimli deyimler kullanarak da sağlanabilir.
- Buna ek olarak *continue / cycle* deyimleri, goto deyimine benzer şekilde döngüdeki işlemlerin akışını değiştirerek programların izlenmelerini güçleştirdikleri için, kullanımları sınırlandırılmalıdır.

Aktarma Deyimleri

- C , C++, Java, Perl ve C#'ta koşulsuz, etiketsiz bir kademe çıkış **break**.
- Java, C#: bir öncekine ilaveten, koşulsuz etiketli birkaç kademeli çıkış **break**.
- Perl: koşulsuz etiketli birkaç kademeli çıkış **last**.
- Bütün bu dillerde ayrıca, döngüyü bitirmeyen, ancak kontrol kısmına gönderen, **break ile aynı özelliklerde continue**.
- Java ve Perl de **continue komutlarının etiketi de olabilir**.

7.2.4.3. Veri Yapılarına Dayalı Döngüler

- Kavram: bir veri yapısını (data structure) ve sırasını döngünün kontrolü için kullanmak.
- Kontrol mekanizması: varsa veri yapısının bir sonraki elemanını dönen bir fonksiyon, yoksa döngü biter.
- C'de **for** bu amaçla kullanılabilir, Örnek:
 - `for (p=hdr; p; p=sonraki(p)) {...}`
 - `for (p=root; p!=NULL; traverse(p)) {...}`

7.2.4.3. Veri Yapılarına Dayalı Döngüler

- PHP: reset, current, prev ve next fonksiyonları
 - current pointer'ın o andaki işlediği dizi elemanını temsil eder.
 - next current değerini bir sonraki elemana taşır.
 - reset current değerini dizinin ilk elemanına taşır.

```
reset $list;                // current göstericisini başa alır.  
print("ilk sayı: " + current($list) + "<br/>");  
while($current_value= next($list))  
    print("sonraki sayı: " + $current_value+ "<br/>");
```

- Java: Iterator metotları kullanılarak (next, hasNext ve remove) yapılabileceği gibi, "foreach" gibi davranan "for" kullanılarak veri yapısına dayalı döngü yapılabilir. Örneğin önceden tanımlanmış "myList" isimli bir Stringimiz varsa

```
for (String myElement : myList) { ... }
```


7.2.4.3. Veri Yapılarına Dayalı Döngüler

- C# ve F# (ve diğer .NET dilleri)'ta Java 5.0'a benzeyen kapsamlı kütüphane sınıfları vardır (diziler, listeler, yığınlar ve kuyruklar). Bu yapılarda bulunan elemanların tümünü `foreach` döngüsüyle gezebiliriz.

```
List<String> names = new List<String>();  
names.Add("Bob");  
names.Add("Carol");  
names.Add("Ted");  
foreach (Strings name in names)  
    Console.WriteLine ("Name: {0}", name);
```

{0} yazılan yere “`name`”in değeri yazılır.

7.2.4.3. Veri Yapılarına Dayalı Döngüler

- Ruby blokları kod dizileridir ve bloklar `do` `end` kodları arasında tanımlanmıştır
 - Bloklar döngü oluşturabilmek için metotlarla beraber kullanılabilirler.
 - Önceden tanımlanmış döngü metotları (`times`, `each`, `upto`):

```
3.times {puts "Hey!"}
list.each {|value| puts value}
```

(`list` bir dizi; `value` ise bir blok parametresi)

```
1.upto(5) {|x| print x, " "}
```
 - Ruby bir `for` komutuna sahiptir fakat `for` komutunu çalıştırabilmek için `upto` metoduna dönüştürmesi gerekmektedir.

7.2.4.3. Veri Yapılarına Dayalı Döngüler

- Ada
 - Ada dilinde döngü aralığı ile dizi indisi arasında ilişki kurulabilir.

```
subtype MyRange is Integer range 0..99;  
MyArray: array (MyRange) of Integer;  
for Index in MyRange loop  
    ...MyArray(Index) ...  
end loop;
```

7.2.5. Güvenlikli Komutlar (Guarded Commands)

- Dijkstra (1975) tarafından yeni ve farklı seçme ve döngü yapıları önerilmiştir.
- Amaç: yeni bir programlama metodolojisini desteklemek –program geliştirmesi sırasında doğrulama
- Paralel eşzamanlı programlama için önemlidir ve iki dilsel mekanizmayı temel alır (CSP ve Ada)

7.2.5. Güvenlikli Komutlar (Guarded Commands)

1. Seçme

- Form

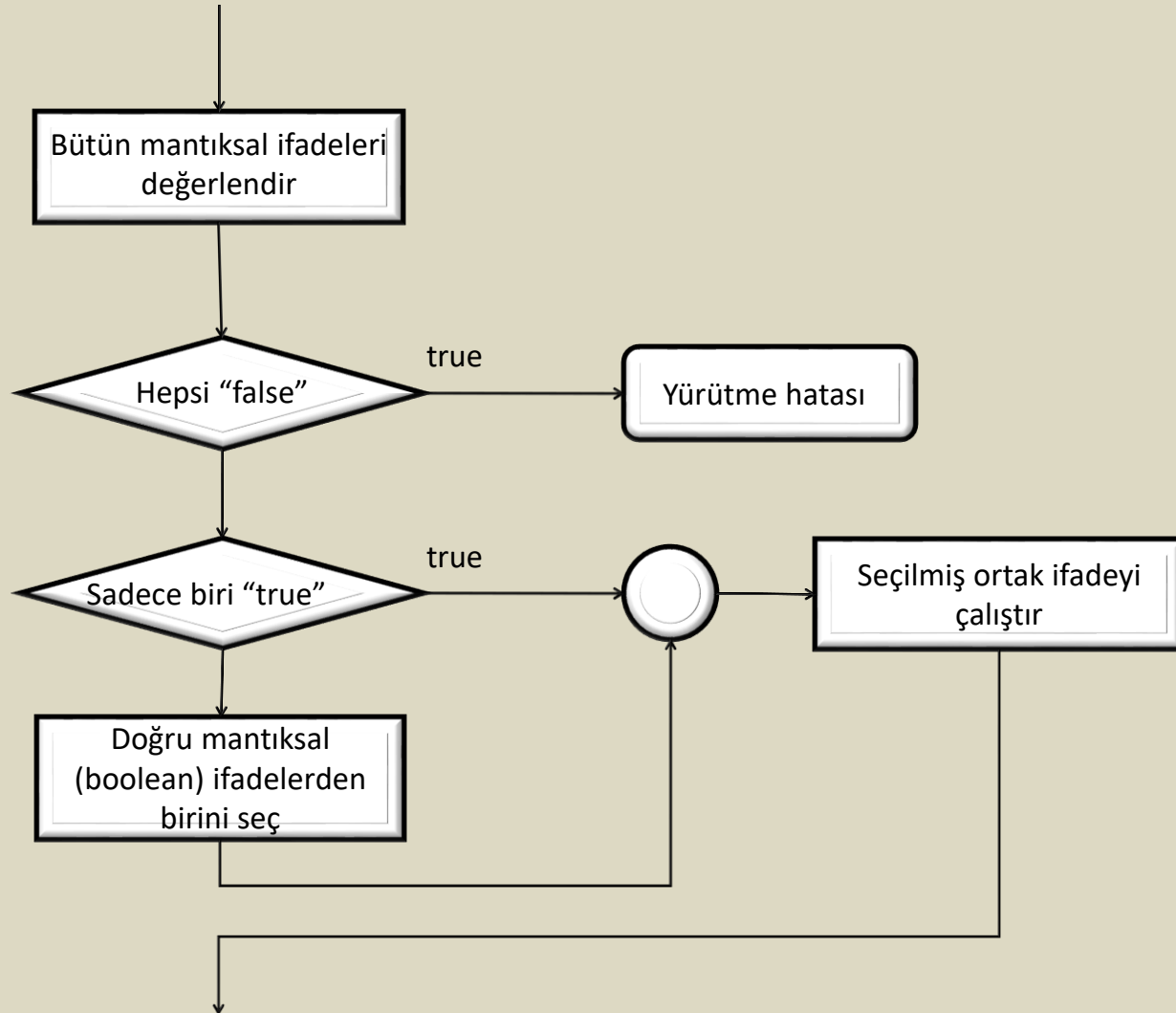
```
if <Boolean expr> -> <statement>  
[] <Boolean expr> -> <statement>  
...  
[] <Boolean expr> -> <statement>  
fi
```

- Anlamı: yapıya ulaşıldığında

- Tüm boolean ifadeleri değerlendir.
- Eğer birden fazla doğru ifade varsa non-deterministik olarak birini seç.
- Eğer doğru ifade yoksa çalışma zamanı hatası ver.

- Düşünce: Eğer değerlendirme sırası önemli değilse, program böyle bir seçim yapmamalıdır

Seçme Güvenlikli Komutlar



7.2.5. Güvenlikli Komutlar (Guarded Commands)

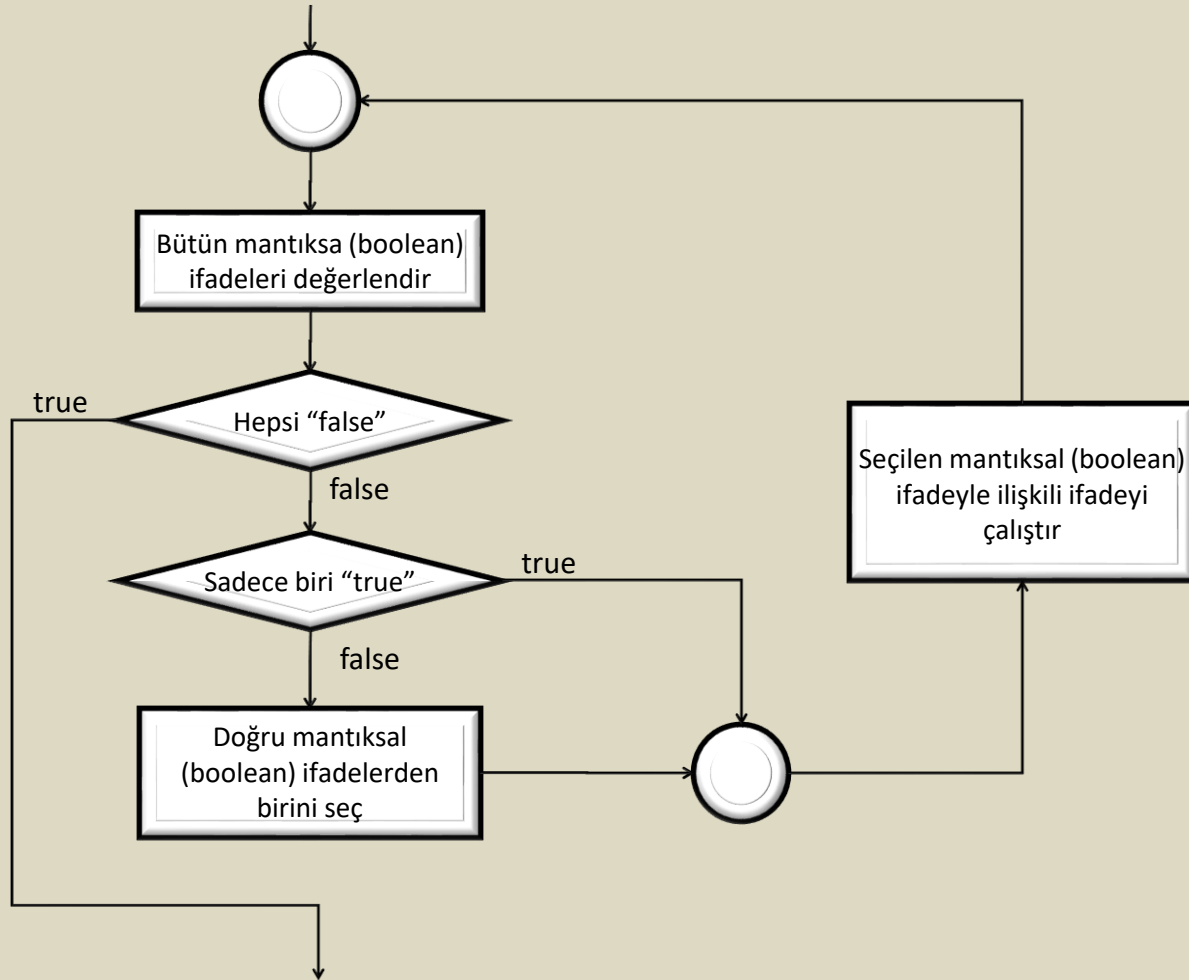
2. Döngü

- Form

```
do<boolean> -> <statement>  
[]<boolean> -> <statement>  
...  
[] <boolean> -> <statement>  
od
```

- Anlamı: Her döngüde,
 - Bütün boolean ifadeleri değerlendir;
 - Eğer birden fazlası doğruysa birini nondeterministic olarak seç;
 - Eğer hiçbirisi doğru değilse, döngüden çık.
- Düşünce: Eğer değerlendirme sırası önemli değilse, program böyle bir seçim yapmamalıdır.

Döngü Güvenlikli Komutlar



7.2.5. Güvenlikli Komutlar (Guarded Commands)

- Kontrol komutları ve program doğrulaması (verification) arasında güçlü bir etkileşme vardır.
- goto kullanılan bir programda doğrulama çok zorlaşır.
- Doğrulama seçmeler ve mantıksal döngüler kullanılırsa yapılabilir.
- Doğrulama "önlemleri komutlarla" görece olarak daha basittir

Özet

- Bu bölümde, modern programlama açısından büyük öneme sahip olan yapısal programlama kavramı ve programlarda akış denetimi sağlayan deyimler ve yapılar incelenmiştir.
- Bu kapsamda; Atama Deyimi, Seçim Yapıları, Yineleme Yapıları ve Akış Denetimini Koşulsuz Değiştirme Deyimleri ele alınmıştır.
- Bu kısımda bahsettiğimiz seçme ve ön kontrollü döngüler dışındaki diğer kontrol komutları dilin büyüklüğü ile kolay yazılabilirlik arasındaki tercih sorunudur.
- Fonksiyonel ve mantıksal dillerdeki kontrol yapıları bu kısımda bahsettiğimiz yapılardan farklıdır.

Kaynaklar

- Roberto Sebesta, Concepts Of Programming Languages, International 10th Edition 2013
- David Watt, Programming Language Design Concepts, 2004
- Michael Scott, Programming Languages Pragmatics, Third Edition, 2009
- Zeynep Orhan, Programlama Dilleri Ders Notları
- Mustafa Şahin, Programlama Dilleri Ders Notları
- Ahmet Yesevi Üniversitesi, Programlama Dilleri Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları
- David Evans, Programming Languages Lecture Notes
- Oscar Nierstrasz, Programming Languages Lecture Notes