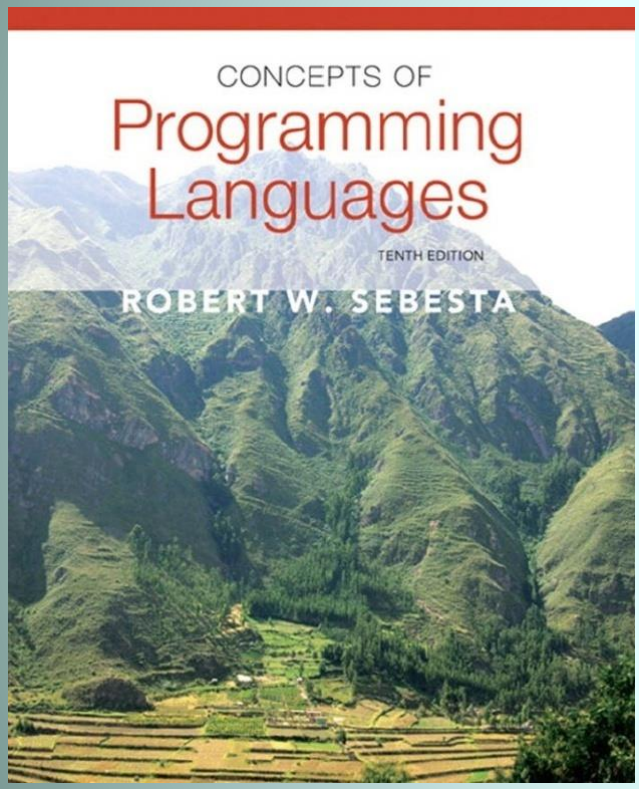
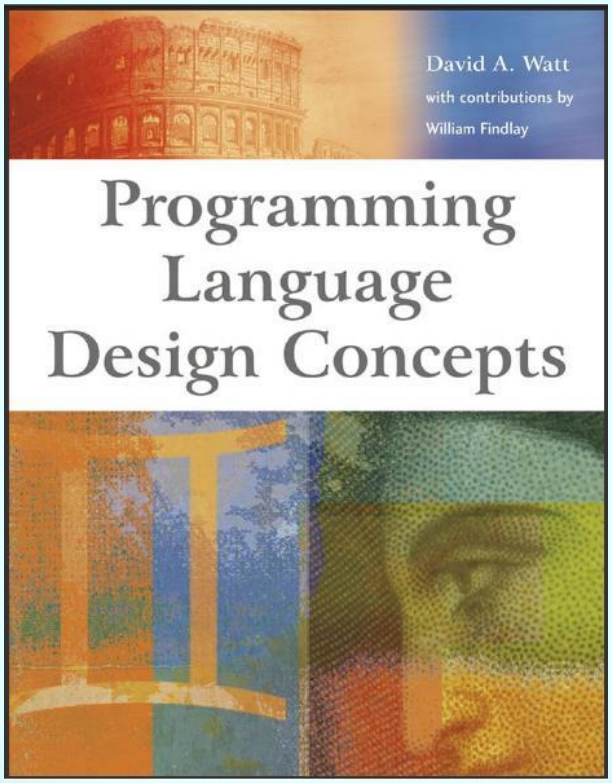
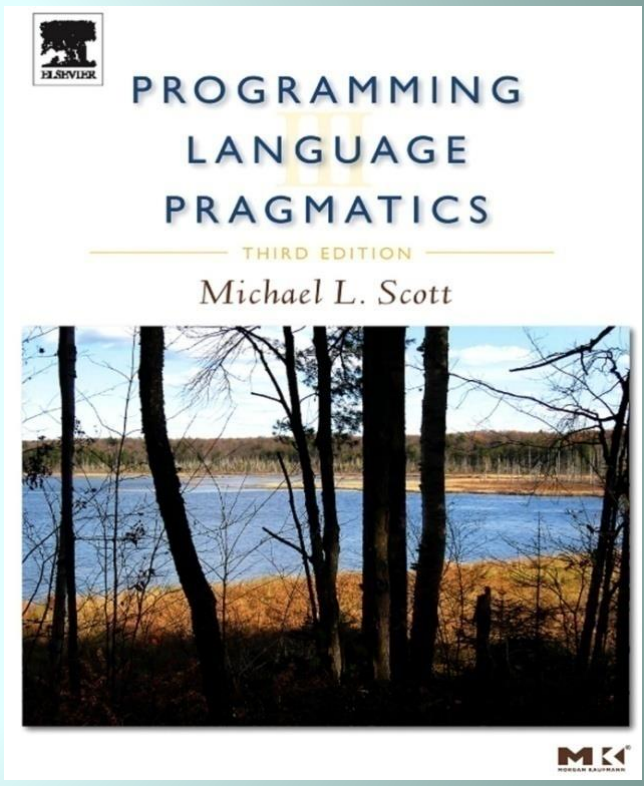
# Bölüm 1: Giriş



# Bölüm 1 Konular

1. Programlama Dilleri Kavramlarını Öğrenmenin Nedenleri
2. Programlama Alanları
3. Dil Değerlendirme Kriterleri
4. Dil Tasarımını Etkileyen Faktörler
5. Dil Kategorileri
6. Dil Tasarımında Verilen Ödünler (Trade-Offs)
7. İmplementasyon Metotları
8. Programlama Platformları (Environments)

# 1.1 Programlama Dilleri Kavramlarını Öğrenmenin Nedenleri

- Fikirleri ifade etme yeteneğinin arttırılması
- Uygun dilleri seçebilme bilincinin geliştirilmesi, altyapının sağlamlaştırılması
- Yeni diller öğrenebilme yeteneğinin geliştirilmesi
- İmplementasyonun (Gerçekleştirim) öneminin daha iyi anlaşılması ve kavranması
- Bilinen programlama dillerinin daha iyi kullanılması
- Bilgisayar biliminde kapsamlı ilerleme

# Fikirleri ifade etme yeteneğinin arttırılması

- Düşüncelerimizdeki derinliğin; düşüncelerimizi ilettiğimiz dilin ifade gücü tarafından etkilendiğine inanılmaktadır.

- Diğer bir deyişle insanların sözlü veya yazılı olarak tanımlayamadığı yapıları kavramlaştırmaları zordur.

# Fikirleri ifade etme yeteneğinin arttırılması

- Yazılım sürecindeki programcılar da benzer şekilde zorlanmaktadır. Programcıların yazılım geliştirdikleri programlama dilinin limitleri vardır. Örneğin o dilin kontrol yapıları, veri yapıları, soyutlamaları, vs. gibi
- Değişik yapıdaki bir çok programlama dili özelliğinin bilinmesi bu limitlerin azaltılmasını sağlamaktadır.
- Programcılar yeni dil yapıları ve kavramları öğrendikçe yazılım geliştirme ufukları artmaktadır

# Fikirleri ifade etme yeteneğinin arttırılması

- Herhangi bir programlama dilini kullanması için zorlanan bir programcının Programlama Dilleri kavramlarını ve yapılarını öğrenmesinin bir anlamı var mıdır?

# Fikirleri ifade etme yeteneğinin arttırılması

- Evet vardır. Bu yapılar ve kavramlar mecburi tutulan dilde simüle (benzetim veya taklit) edilebilir.
- Örneğin FORTRAN 90 dilinde substring arama fonksiyonu INDEX'tir. Pascal dilinde benzer bir fonksiyon yoktur fakat Pascal kullanması için zorlanan bir programcı bu fonksiyonun yaptığı işi altprogram yazarak çözebilir.
- Eğer dil seçiminde zorlama yoksa simüle edilen dildense gerçekten o özelliği barındıran dili kullanmak gerekir. Çünkü o özelliği simüle eden dil direk barındıran dile göre daha hantal ve daha karmaşıktır

# Uygun dilleri seçebilme bilincinin geliştirilmesi

- Bir programcının öğrendiği programlama dilinin ömrü çok uzun olmayabilir.
- Veya öğrendiği dillere eklenen yeni özellikleri bilmeyebilir.
- Bu programcıya yeni bir proje verildiğinde en uygun dili seçmek yerine, bildiği dili tercih etmesi beklenir. Bu programcı diğer programlama dillerinin özelliklerini ve kavramlarını bildiği takdirde daha verimli projeler gerçekleştirecektir.

# Yeni diller öğrenebilme yeteneğinin geliştirilmesi

- Bilgisayar programlama genç bir bilim dalı. Gerek programlama dillerinde gerek yazılım geliştirme araçlarında gerekse tasarım yöntemlerinde gelişim halen devam etmektedir. Bundan dolayı bilgisayar programcılığı ömür boyu öğrenme yeteneği isteyen bir bilim dalı olarak karşımıza çıkmakta.
- Bir veya iki programlama dili bilen ve programlama dilleri kavramlarını bilmeyen programcıların yeni programlama dilleri öğrenmeleri hem uzun zaman almaktadır hem de zor olmaktadır.

# Yeni diller öğrenebilme yeteneğinin geliştirilmesi

- Programlama dillerinin genel kavramlarını iyi bilen bir programcı, yeni öğreneceği dilde bu kavramların nasıl birleştirildiğini ve kullanıldığını daha rahat anlayacaktır. Programlama dilinin tasarımı hakkında bilgi sahibi olacaktır.
- Örneğin veri soyutlama (data abstraction) kavramını bilen bir programcı Java'da soyutlanmış data tiplerinin nasıl oluşturulduğunu daha iyi ve çabuk anlayacaktır. Veya nesne tabanlı programlama kavramını bilen bir kişi C++ veya Java'yı daha çabuk anlayacaktır.

# İmplementasyonun öneminin daha iyi anlaşılması

- Programlama dilleri kavramı öğrenilirken implementasyon sorunlarına da dikkat etmek gerekir. Çünkü  implementasyon sorunları programlama dilleri kavramlarını etkilemektedir. Bazen bir implementasyon sorununu bilmek, programlama dilinin nasıl tasarlandığı konusunu da anlamamıza yardımcı olmaktadır. Sonuç olarak implementasyon sorunları ve bunları aşmak için geliştirilmiş programlama dilleri tasarımları bilindiğinde, bildiğimiz bir programlama dili daha zekice kullanılmaktadır.

# İmplementasyonun öneminin daha iyi anlaşılması

- İmplementasyon detaylarını bilen bir programcı program bug (hatalarını) bulup düzeltebilir.
- İmplementasyonu iyi bilmenin bir avantajı da bilgisayarın değişik dil yapılarını nasıl çalıştırdığını kafamızda canlandırmamıza yardımcı olmasıdır.
- Bir avantajı da, aynı işi yapan farklı yapılar varsa ortaya çıkar. İmplementasyonu bilen programcı hangisinin daha verimli olduğunu anlayabilir.
- Bu derste implementasyona kısaca değinilecek. Daha detaylı bilgi, derleyici tasarımı dersinde verilecektir.

# Bilinen programlama dillerinin daha iyi kullanılması

- Programlama dilleri çok büyük ve karmaşıktır. Bu dilleri bilen bir programcının dilin tüm özelliklerini bilmesi pek rastlanan bir durum değildir.
- Programlama dilleri kavramını bilen bir programcı, daha önce bilmediği ve kullanmadığı özellikleri daha rahat öğrenip kullanabilir.

# Bilgisayar biliminde kapsamlı ilerleme

- Kendi dönemlerinde popüler olan programlama dillerinin en iyi programlama dili olmadığı zamanla görülmüştür.
- Örneğin ALGOL60'ın FORTRAN'ın yerine geçeceğine kesin gözüyle bakılmış, fakat 1960'lar da ALGOL'un okuma ve yazma karmaşıklığından dolayı insanlar bu düşüncelerinden vazgeçmeye başlamışlardır.
- ACABA JAVA SONSUZA KADAR KULLANILACAK MI?

# 1.2 Programlama Alanları

- **Bilimsel Uygulamalar**
  - Bilgisayarların ilk olarak kullanıldıkları alan sayısal uygulamaların ağırlıklı olduğu bilimsel çalışmalar olmuştur. Bu nedenle ilk geliştirilen programlama dilleri, sayısal programlama özelliklerini vurgulamışlardır.
  - Üniversitelerde ve bilimsel kuruluşlarda mühendislik veya matematik hesapları için kullanılırlar.
  - Genelde basit veri yapıları ve çok sayıda kayan nokta (floating point) hesaplamaları içerirler
  - En sık kullanılan veri yapıları diziler ve matrislerdir
  - Fortran
- **Ticari Uygulamalar**
  - Ticari uygulamalardaki veri işleme, sayısal hesaplamalardan sonra ilk olarak gelişen uygulama alanıdır.
  - Rapor üretmek, ondalık sayılar ve karakterler kullanmak
  - COBOL

# 1.2 Programlama Alanları

- **Yapay Zeka (Artificial Intelligence)**
  - Sayılar yerine semboller işlenir
  - Dizilerden ziyade genelde link listler kullanılır
  - Esnek dillerdir. Bazı AI uygulamalarında kod segmentleri program çalışırken meydana gelmektedir
  - LISP, PROLOG
- **Sistem Programlama**
  - İşletim sistemleri ve bilgisayar sisteminin programlama destek araçları sistem yazılımı olarak bilinir.
  - Devamlı kullanım nedeniyle verimlilik (etkinlik) gereklidir. Bu yüzden birleştirici diller (Assembly) yaygın olarak kullanılmaktadır.
  - Harici aygıtlara ulaşabilmesi için düşük seviyeli dillerin özelliklerini de içermesi gerekmektedir.
  - C (Örneğin UNIX işletim sistemi C ile yazılmıştır)
- **Web Yazılımı**
  - Dillerin seçmeci/derlemeci (eclectic) koleksiyonu: işaretleme (etiketleme) (markup) (örn. XHTML), betik (scripting) (örn. PHP), genel-amaçlı (örn. Java)

# 1.3 Dil Değerlendirme Kriterleri

- **Okunabilirlik (Readability)**: Programların okunabilme ve anlaşılabilme kolaylığı
- **Yazılabilirlik (Writability)**: Bir dilin program yazmada kullanılabilme kolaylığı
- **Güvenilirlik (Reliability)**: Şartlara (specifications) uygunluk (yani, şartları sağlaması)
- **Maliyet (Cost)**: En son toplam maliyet

# Dil Değerlendirme Kriterleri

| Karakteristik | Okunabilirlik | Yazılabilirlik | Güvenilirlik |
|---|---|---|---|
| Sadelik(Simplicity) | ✓ | ✓ | ✓ |
| Kontrol yapısı | ✓ | ✓ | ✓ |
| Veri tip ve yapısı | ✓ | ✓ | ✓ |
| Syntax tasarım | ✓ | ✓ | ✓ |
| Soyutlama desteği | | ✓ | ✓ |
| İfade gücü | | ✓ | ✓ |
| Tip kontrolü | | | ✓ |
| İstisna yönetimi | | | ✓ |
| Restricted aliasing | | | ✓ |

# Değerlendirme Kriterleri : Okunabilirlik

- Bir programlama dilinin değerlendirilmesinde en önemli kriterlerden birisi, programların okunabilme ve anlaşılabilme kolaylığıdır.
- Programlama dillerinin okunabilir olmaları, programlarda hata olasılığını azaltır ve programların bakımını kolaylaştırır.
- Bir programlama dilinde yer alan kavramlar, yapılar ve dilin sözdizimi, dilin okunabilirliğini doğrudan etkiler.

# Değerlendirme Kriterleri : Okunabilirlik

- Genel Kolaylık
- Ortogonallik
- Kontrol İfadeleri
- Veri Tipleri ve Veri Yapıları
- Syntax Tasarımı

# Genel Kolaylık

- Çok sayıda basit yapıdan oluşan bir dilin okunabilirliği az sayıda olandan daha azdır.
- Yönetimi kolay kullanışlı nitelikler (features) ve yapılar (constructs) kümesi bulunması **genel kolaylığı arttırır.**
- Özellik çokluğu (feature multiplicity) (aynı işi yapmanın birkaç tane yolu olması) istenmez

Örneğin JAVA'da bir değişkenin değerini 1(bir) artırmak istersek:

```
Count=Count+1
Count+=1
Count++
++Count
```

Yukarıdaki ifadeler birebir aynı değildir. Fakat yalnız kullanıldıklarında aynı anlama gelirler.

- Aynı operasyonlar için farklı farklı özelliklerin/yapıların bulunmaması **genel kolaylığı arttırır.**

# Genel Kolaylık

- Minimum operatör aşırı yükleme (operator overloading)

Örneğin + operatörü birden fazla işlem için atanmış olabilir. Örneğin tamsayılarda toplama, reel sayılarda toplama, matrislerde toplama gibi. Bu özellik, operatör sayısını azalttığı için avantajlı bir durum olarak görülse de bazen programın okunabilirliğini azaltmaktadır. Bir dizideki tüm elemanların toplamı için + operatörünün programcı tarafından atandığını varsayalım. Bu programı okuyan bir kişi + operatörünün ne amaçla kullanıldığı hakkında şüpheye düşecektir. Bu da okunabilirliği azaltır. Operatör yüklemelerinin en az seviyede olması **genel kolaylığı arttırır.**

- SONUÇ OLARAK: BASİTLİK, OKUNABİLİRLİĞİ ARTIRIR AMA ÇOK FAZLA BASİTLİK TE TERSİ ETKİ YAPABİLİR. ÖRNEĞİN ASSEMBLER DİLİ EN BASİT DİL OLMASINA RAĞMEN OKUNABİLİRLİĞİ EN UYGUN DİL DEĞİLDİR.

# Genel Kolaylık (Özet)

- Yönetimi kolay özellikler/yapılar bulunması
- Aynı operasyonlar için farklı farklı özelliklerin/yapıların bulunmaması
  - Özellik çeşitliliğinin (feature multiplicity) (Aynı işi yapmanın birkaç tane yolu olması) minimum seviyede tutulması
- Operatör yüklemelerinin (Aşırı işlem yüklenmesinin ) en az seviyede olması

# Ortogonallik (Orthogonality)

- Ortogonallik (Orthogonality)
  - Bir programlama dilinin kontrol ve veri yapıları; çok küçük kümeye sahip ilkel yapılar ve bu yapıların çok az yolla birleştirilerek oluşmasına ortogonallik denir.
    - Dillerde bulunan özelliklerin birbirine ortogonal olması, yani birbirinden bağımsız olması kast edilir. Şayet bir dilde bulunan özellikler arasında ilişki bulunmuyorsa, dilin özelliklerinin genişlemiş olduğu düşünülebilir.
  - Her mümkün kombinasyon kurallara uygundur
  - Örneğin 4 tip ilkel yapı (integer, float, char ve double) ve 2 operatör (array ve pointer) olduğunu varsayalım. Bu 2 operatör kendi kendilerine ve ilkel veri yapılarına uygulanabilirse, çok sayıda veri yapısı elde edilebilir.
    - Bu durumda bir pointer, integer veya floatı gösterebilmektedir. Benzer şekilde arrayin elemanları pointer olabilmekte ve pointer da arraye işaret edebilmektedir.
    - Şayet dilde, ortogonallik bulunmasaydı, örneğin array işlemi, sadece float üzerinde çalışıyor olsaydı, bu durumda dildeki esneklik büyük ölçüde kaybedilmiş olacak ve örneğin integer arrayi bulunmayacağı gibi, integer arrayine işaret eden bir pointer veya her elemanı pointer olan bir integer arrayi de kullanılamayacaktı.

# Örnek: Hangisi ortogonaldir? Neden?

- IBM ve VAX32 için hazırlanmış 2 farklı assembler dilini bir örnek üzerinde inceleyip ortogonalliklerine bakarsak;

  Örnek: biri hafızada, diğeri registerde olan 2 tane 32 bitlik sayıyı toplayıp sonucu bu hafızalardan birine yazdırmak istersek;

- IBM:

```
A Reg1, memory_cell
AR Reg1, Reg2
```

*Semantiğine bakılırsa:*

*Reg1 ← contents (Reg1) + contents (memory_cell)*

*Reg1 ← contents (Reg1) + contents (Reg2)*

- Vax32:

```
ADDL operand_1,operand_2
```

*Semantiğine bakılırsa:*

*Reg1 ← contents (operand_1) + contents (operand_2)*

**Yukarıdaki örnekten görüldüğü gibi Vax32 tasarımının ortogonal olduğu görülmektedir. Tek bir komutla hem registerler hem de hafıza hücrelerinde işlem yapılabilmektedir.**

- C dilinde toplama işlemi (+) bulunmakta ve ortogonallik gereği mümkün olduğunca değişken yapısından bağımsız tutulmaktadır.
- Bir dilde ortogonallik özelliğinin fazla olması, dildeki kodlamayı kolaylaştırır ve dilin öğrenilmesini basit hale getirir. Şayet ortogonallik özelliği az olsaydı, bu durumda dili öğrenen veya kodlayan kişilerin istisna kurallarını da öğrenmesi gerekecekti. Örneğin İngilizce 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 ortogonallik özelliğinin çok fazla olması da sakıncalıdır. Aşırı derecede ortogonallik içerilmesi halinde, dili çalıştıracak bir derleyicinin geliştirilmesi imkansızlaşacağı gibi kodu kullanan kişilerin de hata yapma ihtimali artacaktır.

# Ortogonallik (Orthogonality)

- Fonksiyonel diller (örneğin Lisp) basit ve ortogonal dillerdir. Bu dillerde fonksiyonlar doğrudan parametrelere uygulanır. C, C++ ve Java'da olduğu gibi değişkenlerle ve atamalarla uğraşmaz. Fonksiyonel diller, basitlik ve ortogonallik özelliklerinden dolayı halen tercih edilmektedir.

# Ortogonallik (Orthogonality)

- Az sayıda ilkel/basit yapıların yine az sayıda yöntem kullanılarak birleştirilmesi
- Her kombinasyonun doğru (legal) olması

# Kontrol İfadeleri

- İyi organize edilmiş belli başlı kontrol deyimlerinin ve yapılarının bulunması (örn., while deyimi)

# Veri Tipleri ve Veri yapıları

- Bir programlama dilinde veri tip ve veri yapıları tanımlamadaki esneklik te okunabilirliğe katkı sağlamaktadır.
- Örnek: Bir dilde

```
Timeout=1
```

yazmak yerine

```
Timeout=true
```

yazmak okunabilirliği artırmaktadır.

# Veri Tipleri ve Veri yapıları

- Uygun yeri yapılarının iyi tanımlanmış olması

# Syntax Tasarımı

- Bir programın syntax tasarımı okunabilirlik için en önemli etkenlerden biridir
- Örneğin değişken isimlerini kısa tutmak okunabilirliği azaltır. Fortran 77'de değişken isimleri 6 karakterle sınırlandırılmıştır.
- Örnek: ogrenci_no biçiminde okunabilirliği artıracak bir değişken ismi verilemez.

# Syntax Tasarımı

- Özel kelimeler de (for, end, do, vb.) okunabilirliği etkiler.
- Birleşik durumların gösterim biçimi de okunabilirliği etkiler. Örneğin bir döngü ifadesi verilebilir.
- Özel kelimeler değişken ismi olarak tanımlanırsa da okunabilirlik etkilenir.
- Örneğin Fortran 95'te Do ve End özel kelimelerinin değişken ismi olarak ta tanımlanmasında bir sınırlama yoktur. Bu durum okunabilirliği azaltmaktadır.

# Syntax Tasarımı

- Bir diğer problem de anlam ve biçim.

Örneğin C programlama dilinde static kelimesi kullanılış biçimine göre farklı anlamlara gelmektedir.

Eğer bir fonksiyonun içinde değişken tanımlamak için kullanılıyorsa değişkenin derleme zamanında (compile time) oluşturulacağı anlamına gelmektedir.

Eğer tüm fonksiyonların dışında değişken tanımlamak için kullanılıyorsa, bu değişken bu dosyadan alınamaz ve ancak ve ancak tanımlandığı dosyada görülebilir.

# Syntax Tasarımı

- ÖRNEK: UNIX' in shell komutları çoğu zaman alakasız bir fonksiyonu göstermektedir. Bu da okunabilirliği azaltmaktadır.

grep komutu size herhangi bir şey anımsatıyor mu?

**g** harfi global komut olduğunu gösteriyor.

**re** regular expression

**p** regular ifadeyi içeren substring'lerin yazılması komutudur

Global Regular Expression Printer

# Syntax Tasarımı

- Tanıtıcı/kimlik belirtici (Identifier) biçimleri: Esnek kompozisyon
- Karmaşık ve birleşik ifadelerin oluşturulması için belirli özel deyimlerin (kelime ve metot) kullanılması
- Anlamlı anahtar sözcükler ve anlaşılabilir yapılar kullanılması

# Dil Değerlendirme Kriterleri

| Karakteristik | Okunabilirlik | Yazılabilirlik | Güvenilirlik |
|---|---|---|---|
| Sadelik(Simplicity) | ✓ | ✓ | ✓ |
| Kontrol yapısı | ✓ | ✓ | ✓ |
| Veri tip ve yapısı | ✓ | ✓ | ✓ |
| Syntax tasarım | ✓ | ✓ | ✓ |
| Soyutlama desteği | | ✓ | ✓ |
| İfade gücü | | ✓ | ✓ |
| Tip kontrolü | | | ✓ |
| İstisna yönetimi | | | ✓ |
| Restricted aliasing | | | ✓ |

# Yazılabilirlik

- Bir program yazılırken programlama dilinin ne kadar kolay olduğunun ölçülmesine yazılabilirlik denir. Okunabilirliği etkileyen faktörler (basitlik, ortogonallik, veri tipleri ve veri yapıları, syntax tasarımı ) yazılabilirliği de etkilemektedir.

# Yazılabilirlik

- Programlama dillerinin yazılabilirliği uygulamaya göre de değişebilir. Grafiksel arayüzü olan bir program tasarlanacağı zaman Visual Basic'in yazılabilirliği, C dilininkine göre daha iyidir.
- Bir işletim sistemi yazılacağı zaman ise, C dilinin yazılabilirliği, Visual Basic dilininkine göre daha iyidir.

# Yazılabilirlik

- Basitlik ve Ortogonallik
- Soyutlama (Abstraction) Desteği
- Anlamlılık (Açıklayıcılık) (Expressivity)

# Basitlik ve Ortogonallik

- Eğer bir programlama dilinde çok fazla sayıda farklı yapılar olursa programcının tüm bu yapıları öğrenmesi zordur. Programcı bildiği yapıları kullanmak isteyecek ve belki aynı işi yapan daha verimli bir yapının farkında bile olmayacaktır.
- Bu yüzden az sayıda ilkel yapı ve bunları birleştirmek için oluşturulan legal kombinasyonları (ortogonallik) kullanmak yazılabilirliği artırmaktadır.
- Çok fazla ortogonallik; okunabilirliği azalttığı gibi yazılabilirliği de azaltabilmektedir. Fazla ortogonallik programdaki hataları bulmayı zorlaşmaktadır.

# Soyutlama (Abstraction) desteği

- Karmaşık yapıları veya işlemleri, ayrıntıları göz ardı ederek tanımlayabilme ve kullanabilme yeteneğine soyutlama denir.
- Günümüzde tercih edilen programlama dillerinin tasarımındaki en önemli kavramlardan biridir.
- Programlama dilinde kullanılan soyutlamanın derecesi ve doğal ifade edilebilme özellikleri yazılabilirliği etkilemektedir.

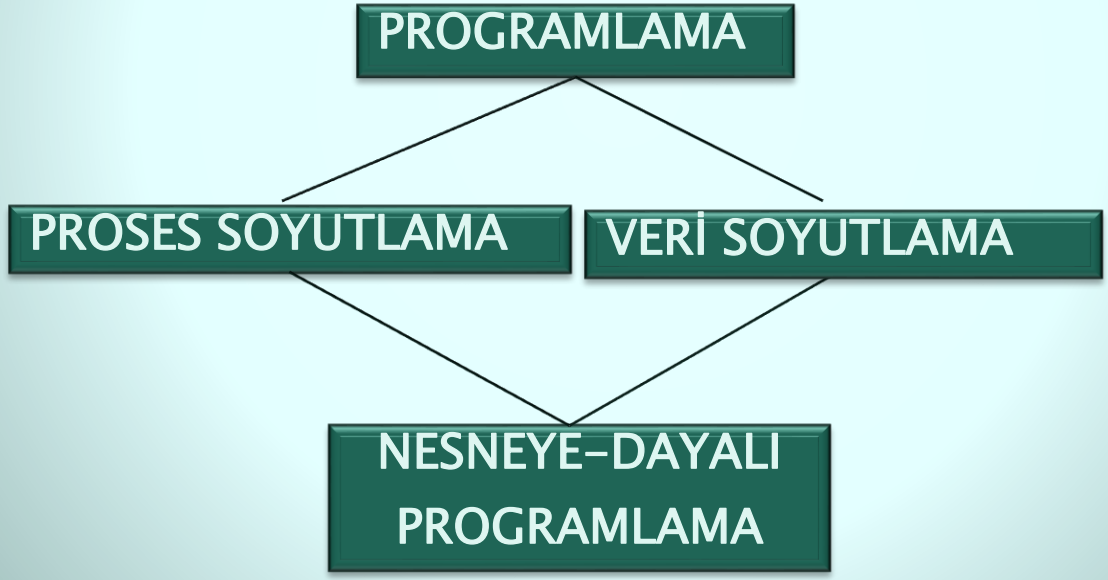
# Soyutlama (Abstraction) desteği

Programlama dillerinde 2 çeşit soyutlama sınıfı vardır. Proses (işlem) soyutlama ve veri soyutlama.

- Proses soyutlamaya en iyi örnek altprogramlardır. Örneğin sıralama yapan bir alt program kod yazarken tekrarları azaltır.
- Veri soyutlamaya bir örnek Fortran77 programlama dilinde binary bir ağaç tanımlanmasıdır. Fortran77 dili pointer ve heap ile dinamik hafıza yönetimini desteklememektedir. Fortran 77 kullanarak binary tree oluşturulabilir. Fakat oluşturmak zordur ve yazımı anlaşılabilir değildir. Fakat soyutlama özelliği olan Java gibi bir dil kullanıldığında binary tree'yi oluşturmak (2 pointer ve 1 integer ile) ve kullanmak çok daha kolaydır. Yazılabilirliği de Fortran'a göre çok iyidir.

# Soyutlama (Abstraction) desteği

İki soyutlama süreci, sınıf yapısında birleşmiştir.



# Anlamlılık (Açıklayıcılık)(Expressivity)

- İşlemleri belirtmenin nispeten elverişli yollarını kullanmak yazılabilirliği artırır.
- Örnek: Birçok modern dilin **`for`** ifadesini içermesi
- C programlama dilinde **`count++`** ifadesi **`count=count+1`** ifadesinden daha açıklayıcıdır. Yazılabilirliği artırır.

# Değerlendirme Kriterleri : Güvenilirlik

- Bir program tüm şartlar altında istenilen görevi doğru olarak yapıyorsa güvenilirdir denir. Bir programlama dilinin güvenilirliği, o dil kullanılarak geliştirilen programların güvenilir olmasıdır.
- Programlama dillerinde güvenilirlik, çeşitli faktörler tarafından belirlenir. Bunlara örnek olarak, dilde bulunan tip denetimi ve istisnai durum işleme verilebilir.
- Ayrıca programın doğruluğunun sağlanması için programlama ortamında sunulan araçlar da, güvenilir programlar geliştirilmesini etkiler.

# Değerlendirme Kriterleri : Güvenilirlik

- Tip Kontrolü (Type checking)
  - Tip hatalarının (errors) test edilmesi
- İstisna İşleme (Exception handling)
  - Yürütme zamanı (run-time) hatalarını yakalama ve düzeltme önlemlerinin alınması
- Farklı Adlandırma (Aliasing)
  - Aynı bellek konumu için iki veya daha fazla farklı referans metodunun (referencing method) varlığı iyi değildir
- Okunabilirlik ve yazılabilirlik
  - Bir algoritmayı ifade etmenin “doğal” yollarını desteklemeyen bir dil, ister istemez “doğal olmayan”, bu yüzden güvenilirliği azalmış yaklaşımları kullanacaktır

# Dil Değerlendirme Kriterleri

| Karakteristik | Okunabilirlik | Yazılabilirlik | Güvenilirlik |
|---|---|---|---|
| Sadelik(Simplicity) | ✓ | ✓ | ✓ |
| Kontrol yapısı | ✓ | ✓ | ✓ |
| Veri tip ve yapısı | ✓ | ✓ | ✓ |
| Syntax tasarım | ✓ | ✓ | ✓ |
| Soyutlama desteği | | ✓ | ✓ |
| İfade gücü | | ✓ | ✓ |
| Tip kontrolü | | | ✓ |
| İstisna yönetimi | | | ✓ |
| Restricted aliasing | | | ✓ |

# Tip Kontrolü (Type Checking)

- Programın çalışması veya derlenmesi süresince verilen programın tip hatalarının test edilmesine tip kontrolü (type checking) denir. Tip kontrol özelliği dilde güvenilirlik için önemlidir. Derleme zamanında (compile time) tip kontrolü yapmak, az maliyetli olduğundan tercih edilmektedir. Çalışma zamanında (run time) tip kontrolü maliyeti fazladır ve tercih edilmez. Programdaki tip hatalarının tespiti erken yapıldığından, hatalar erken fark edilmekte ve düzeltilmeleri sağlanmaktadır.

# Tip Kontrolü (Type Checking)

- Örnek: Bir fonksiyonda kullanılan bir parametrenin tip kontrolünün programlama dili tarafından yapılamadığını varsayalım.
- Int olarak tanımlanması gereken 23 sayısı yerine float olarak aynı sayı girildiğinde ne olur?

# İstisna İşleme (Exception Handling)

- Yürütme zamanı (run-time) hatalarını yakalama ve düzeltme önlemleri alır.
- Ada, C++ ve Java  bu özelliği içermektedir, ama C ve Fortran gibi dillerde bu özellik yoktur. Daha sonraki konularda detaylı tartışılacaktır.

# Farklı Adlandırma (Aliasing)

- Aynı hafıza hücresine erişim için 2 veya daha fazla farklı ismin tanımlanmasıdır.
- Programlama dilleri için tehlikeli bir özellik olduğu kabul edilmektedir. Fakat bazı programlama dilleri bazı tip aliasing'lere izin vermektedir. Örneğin iki pointer aynı değişkeni göstermek için kullanılabilir.
- Programcının dikkatli olmasını gerektirmektedir. NEDEN?

# Okunabilirlik ve Yazılabilirlik

Bir algoritmayı ifade etmenin “doğal” yollarını desteklemeyen bir dil, ister istemez “doğal olmayan”, bu yüzden güvenilirliği azalmış yaklaşımları kullanacaktır.

NOT: Okunabilirlik, güvenilirliği 2 farklı yönden etkilemektedir.

- Okunabilirlik, yazılabilirliği etkiler. Yazılabilirlik te güvenilirliği etkilediğinden, doğal olarak okunabilirlik güvenilirliği etkiler.
- Okunabilirlik programın bakım ve onarım kısmını etkilediğinden, güvenilirliği de etkilemektedir. Okunabilirliği az olan programı anlayıp bakımını yapmak ta zor olacaktır.

# Değerlendirme Kriterleri : Maliyet

- Dili kullanacak programcıların eğitilmesi: Bu özellik basitlik ve ortogonallik ile alakalıdır. Ayrıca programcının tecrübesi de önemlidir.
- Programları yazma (özel uygulamalara yakınlık): Dilin yazılabilirliği ile alakalıdır.
- Programları derleme (compiling) maliyeti: Ada programlama dilinin ilk versiyonlarında derleme maliyetleri çok fazlaydı. Daha sonraki versiyonlarda maliyet azaltıldı.

# Değerlendirme Kriterleri : Maliyet

- Programları yürütme (executing) maliyeti

  Örneğin tip kontrolüne çok fazla run time ayıran bir programlama dili bu maliyeti artırır.
- SORU: Derleme maliyeti ile yürütme maliyeti arasında optimizasyon nasıl yapılabilir?
- Dil implementasyon sistemi:

  Ücretsiz derleyicilerin (compilers) kullanılabilirliği. Örneğin Java'nın derleyicisinin ücretsiz olması maliyeti azaltmaktadır. Veya daha pahalı bir donanımda çalışması gereken programlama dilleri maliyeti artırır.

# Değerlendirme Kriterleri : Maliyet

- Güvenilirlik (Reliability):

  Zayıf güvenilirlik yüksek maliyetlere yol açar. Örneğin bir uzay çalışmasında kullanılan programlama dili çökerse maliyet çok artar. Veya kriptoloji uygulamasında kullanılan bir programlama dilindeki güvenlik açığının maliyeti zarar verir.

- Programların bakım (maintaining) maliyeti

  Tüm maliyet içinde en yüksek oranda olan maliyettir. Bir yazılım projesinde eskiden maliyeti etkileyen en önemli parametre kodlama kısmı iken, şimdi bakım maliyeti en önemli parametre haline gelmiştir.

# Değerlendirme Kriterleri: Diğerleri

- Taşınabilirlik (Portability)
  - Programların bir implementasyondan diğerine taşınabilme kolaylığıdır. Dilin standartlaşma derecesi ile alakalıdır. BASIC'in standartlaşma derecesi kötüdür. Bu dilde yazılan bir programı başka bir implementasyona taşımak zordur.
  - Standartlaşma çok vakit alan maliyetli bir işlemdir. 1989 yılında C++'yı standartlaşmak için kurulan bir komite 1998 yılında çalışmasını tamamlamıştır.

# Değerlendirme Kriterleri: Diğerleri

- Genellik (Generality)
  - Çok çeşitli uygulamalarda uygulanabilirliği olması
- İyi-tanımlanmış olması (Well-definedness)
  - Dilin resmi tanımının kesinliği (doğruluğu) (precision) ve tamamlanmış olması (completeness)

# 1.4 Dil Tasarımını Etkileyen Faktörler

- Bilgisayar Mimarisi (Computer Architecture)
  - Diller, *von Neumann* mimarisi olarak bilinen yaygın bir bilgisayar mimarisi etrafında geliştirilir
- Programlama Tasarım Yöntemleri
  - Yeni yazılım geliştirme metodolojileri yeni programlama örneklerine bunların sonucunda da, yeni programlama dillerine yol açtı.
  - (örn., nesneye-dayalı (object-oriented) yazılım geliştirme)

# Bilgisayar Mimarisi Etkisi

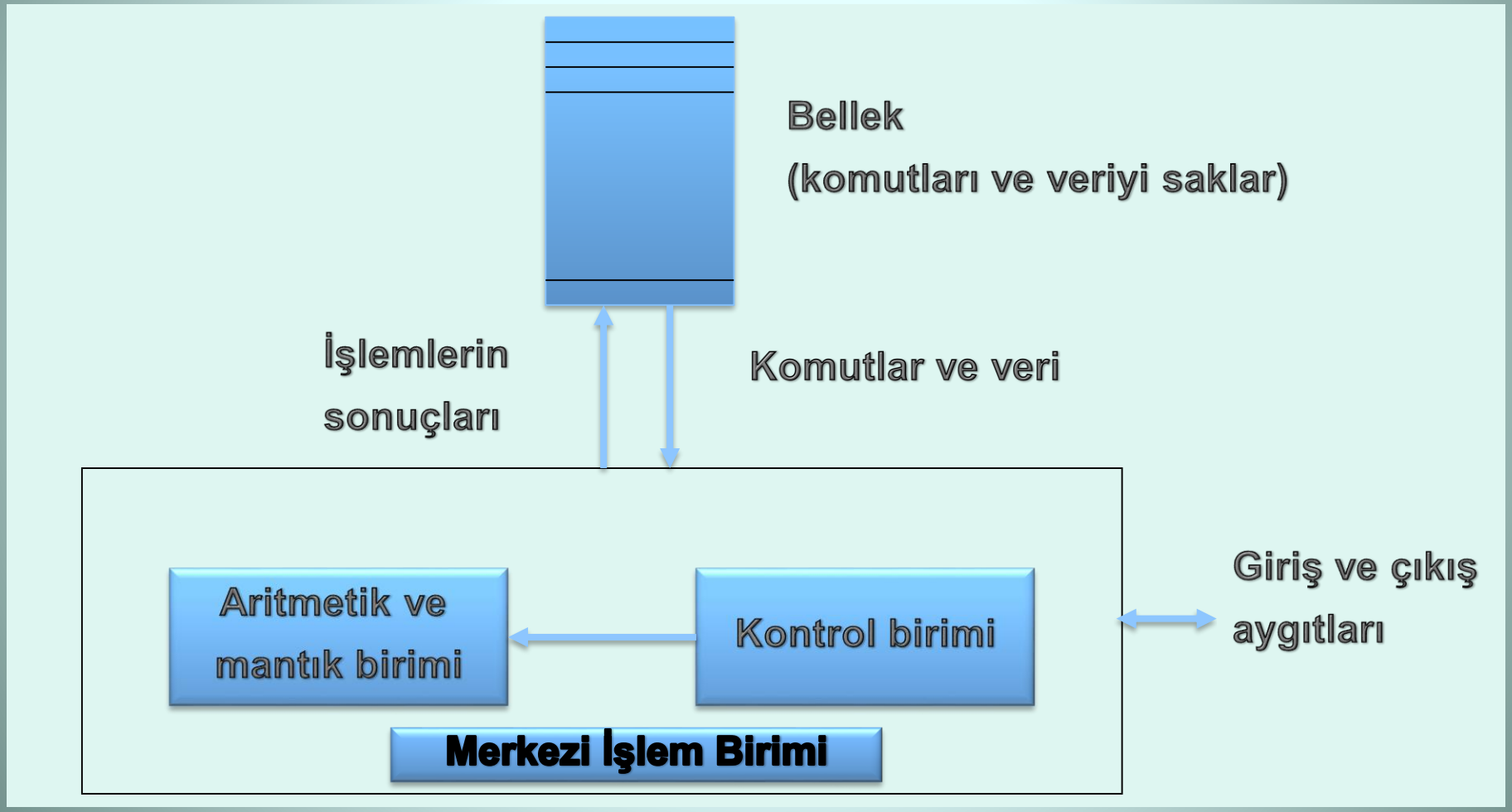
- Bilgisayarların temel mimarileri programlama dili tasarımında çok etkilidir.
- Son 50 yılda popüler olan programlama dillerinin çoğu ünlü bir bilgisayar mimarisi olan Von Neumann mimarisi dikkate alınarak tasarlanmıştır.
- Bu diller imperative diller olarak bilinirler.
- Von Neumann bilgisayarları nedeniyle imperatif diller (imperative languages) hakimdir

# Bilgisayar Mimarisi Etkisi

Von Neumann mimarisinde:

- Veri ve programlar bellekte saklanır
- Bellek, CPU'dan ayrıdır
- Komutlar ve veri bellekten CPU'ya iletilir (pipe)
- CPU'da ki işlemin sonucu hafızaya geri gönderilir.
- İmperatif dilleri esas alır.
- 1940'tan sonra üretilen bilgisayarların çoğunda Von Neumann (Princeton) mimarisi hakimdir.

# Von Neumann Mimarisi



- Von Neumann'dan dolayı İmperatif dillerin temelleri;
  - Değişkenler (Variables) bellek hücrelerini modeller
  - Atama ifadeleri (Assignment statements) iletim (piping) işlemi temeline dayanır
  - İterasyon (Iteration) bu mimaride tekrarı gerçekleştirmenin en verimli yoludur.

- İfadelerdeki operatörler CPU tarafından hafızadan alınırlar, ifade işlemden geçtikten sonra hafızaya geri gönderilir. Bu işlem atamanın sol tarafı (left side of assignment) olarak bilinir.

  Örnek:

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

- Von Neumann bilgisayarlarda iterasyon hızlıdır, çünkü komutlar hafızada birbirlerine yakın hücrelerde bulunmaktadır. Bir iterasyonda aynı komut bir çok kere çağrılacağından komutlardaki yakınlık hızı da beraberinde getirmektedir.
- Bazı uygulamalar için rekürsiflik daha doğal olmasına rağmen Von Neumann mimarisinin bu özelliğinden dolayı iterasyonlar, rekürsif ifadelere göre daha verimlidir.

- Makina kodunun bir Von Neumann mimarisinde işlenmesi işlemine fetch-execute işlemi denir. Çünkü komutlar hafızada bulunmaktadır ama CPU'da işlenmesi gerekmektedir. İşlenecek tüm komutların hafızadan CPU'ya getirilmesi gerekmektedir. Bir sonra işlenecek olan komutun adresi program counter diye bilinen registerlerla sağlanır.

- von Neumann mimarisi bilgisayarda Fetch-Execute basitçe aşağıdaki algoritmayla tanımlanabilir.

```
Program sayıcısını başlangıç durumuna getir
repeat
   program counter da adresi verilen komutu hafızadan getir (fetch)
   bir sonraki komutun adresini göstermesi için program counter'i bir artır
   komutu çöz (decode)
   komutu çalıştır (execute)
end repeat
```

- Fonksiyonel dillerde imperative dillerdeki değişkenler, atama ifadeleri ve iterasyonlar yoktur. Bu dillerde hesaplamalar yapılırken fonksiyonlar doğrudan değişkenlere uygulanır
- “Fonksiyonel dillerin bir çok yararlı özelliğine rağmen, bu dillerin imperative dillerin yerine geçme şansı günümüzde yoktur. Ancak günün birinde fonksiyonel dilleri verimli bir şekilde işleyecek bir Von Neumann mimarisi tasarlanırsa bu durum değişebilir” John Backus 1978.

- Başka bir grup bilim adamı ise imperative dillerin daha doğal olduğunu, fonksiyonel dilleri verimli gerçekleştiren bilgisayar mimari yapıları tasarlansa bile, halen imperative dillerin tercih edileceğini iddia etmektedir.

# Programlama Tasarım Yöntemlerinin Etkisi

- 1950'ler ve 1960'ların başlangıcı: Basit uygulamalar; makine verimliliğiyle ilgileniyordu
- 1970'lerde programların maliyeti donanımdan yazılıma doğru kaymaya başladı. Donanımların fiyatı düştü, yazılımcıların maliyeti ise arttı.
- 1960'ların sonları: İnsan verimliliği önem kazandı; okunabilirlik, daha iyi kontrol yapıları (control structures)
  - Yapısal programlama (structured programming)
  - Yukarıdan aşağıya (top-down) tasarım ve adım adım şekillendirme/rötüş (step-wise refinement)
- 1970'lerin sonları: İşleme-dayalı (Process-oriented) dan veriye-dayalıya (data-oriented) geçiş
  - Veri soyutlama (data abstraction)

  Veriye dayalı programlama çok tercih edilmesine rağmen işleme dayalı programlama da hiçbir zaman terk edilmedi. Özellikle concurrent dillerde halen tercih edilmekte. Örneğin Ada, C#, Java

- 1980'lerin ortaları: Nesneye-dayalı (Object-oriented) programlama
  - Veri soyutlama + miras + polimorfizm

    (Data abstraction + inheritance + polymorphism)
- Fonksiyonel ve mantıksal dillere de nesne tabanlı özellikler eklendi.
- CLOS (nesne tabanlı fonksiyonel dil)
- Prolog++ (nesne tabanlı mantıksal dil)

# 1.5 Dil Kategorileri

Programlama Paradigmaları

# Programlama Paradigmaları


- Bir **paradigma**, bir grubun ortak kararlarını temsil eden ve grubun konuya yeni bir şekilde bakışını sağlayan bir kavramsal şemadır.
- Programlama paradigmaları, bir programcının, problemlere çözüm üretmesini önemli derecede etkilerler.
- Farklı paradigmalar, farklı programlama stilleri getirir ve programcıların algoritmalara bakış şeklini değiştirirler.

# Programlama Paradigmaları

**Paradigma-Yönelik Diller:**

- Belirli bir paradigmayı destekleyen dillere **paradigma-yönelik** diller denir.
- Örneğin, Smalltalk ve Eiffel gibi diller nesneye yönelik programlama paradigmasını desteklerler. Benzer şekilde, FORTRAN ve Pascal, imperative paradigmayı desteklerler.
- Öte yandan, programlama dilleri ile paradigmalar arasında bire bir ilişki olması gerekli değildir. Bu bağlamda, bazı diller paradigma bağımsız olup, birden çok paradigmayı destekleyebilirler. Örneğin, C++ hem imperative hem de nesneye yönelik programların geliştirilmesini destekler.

# Programlama Paradigmaları

- Bir yazılım sisteminin geliştirilmesinde kullanılan tasarım yöntemi ve dil paradigması aynı ise, tasarım ve programlama eşlemeleri sağlanarak, yazılım geliştirme kolaylaştırılır.
- Paylaşılan kavramlar nedeniyle, programlama paradigmalarına göre gruplanan diller arasında yoğun benzerlik bulunmaktadır. Bu nedenle bir paradigma grubunun bir üyesini öğrenmek, aynı gruptaki bir diğer dili öğrenmeyi kolaylaştırır.
- *imperative*, *fonksiyonel*, *mantıksal ve nesneye yönelik* olmak üzere dört temel programlama paradigması vardır.

# Programlama Paradigmaları

- **Imperative (Zorunlu, komut merkezli, emirsel) Paradigmayı Destekleyen Diller**
  - *Imperative paradigma*'daki programlama dilleri işlem tabanlı olup bir program, bir dizi işlem olarak görülür. Komutlar ve prosedürler kullanarak değişkenleri güncellemesi ile karakterize edilir.
  - Programlardaki deyimler, birbirleri ile değişkenler aracılığı ile iletişim kurar. Bu dillerde bir program, bir dizi deyimden oluşur ve her deyimin çalışması, bellekteki bir veya daha fazla yerleşim değerinin değiştirilmesine neden olur.
  - Örneğin, iki değişkenin toplanması ve üçüncü bir değişkenin elde edilmesi, bu değerlerin birleştirilerek sonucun yeni bir yerleşimde saklanması olarak gösterilebilir.

# Programlama Paradigmaları

- **Imperative (Zorunlu) Paradigmayı Destekleyen Diller**
  - Söz dizimi:
    - *Deyim-1;*
    - *Deyim 2;*
    - *……*
    - *……*
    - *Deyim-n;*


  - Imperative diller, yaygın olarak kullanılan ilk dil grubudur. Imperative programlama paradigması, C, FORTRAN, PL/I, Pascal, COBOL, Ada gibi birçok dil tarafından desteklenmektedir.

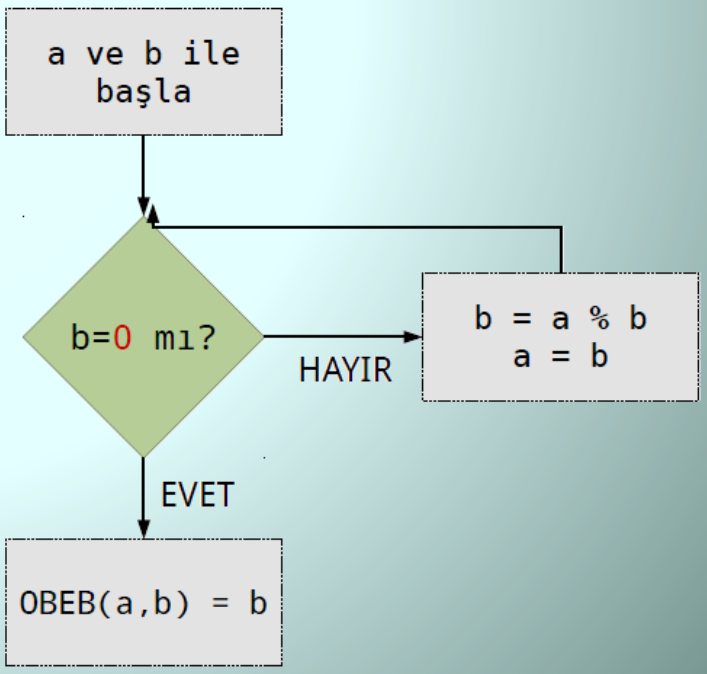
# Programlama Paradigmaları

## C'de OBEB (Öklit) Algoritması

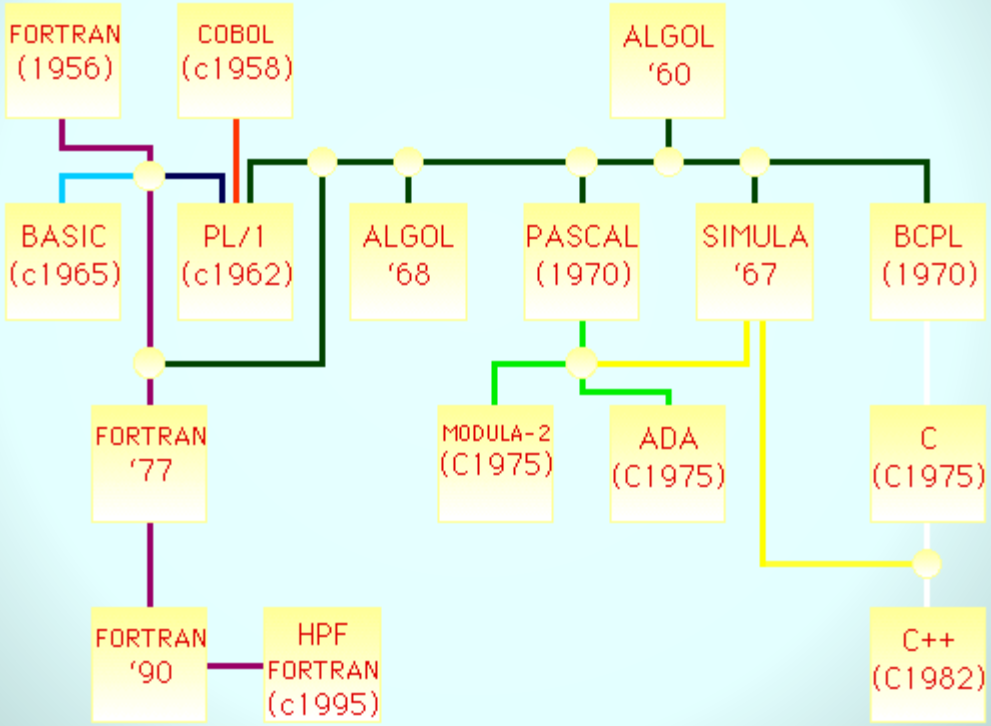
```
#include <stdio.h>
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

Algoritma, 2000 yıl önce Euclid tarafından keşfedilmiştir

*a ve b'nin OBEB'i, b ve a'nın b'ye bölümünden kalan sayının OBEB'iyle aynıdır*



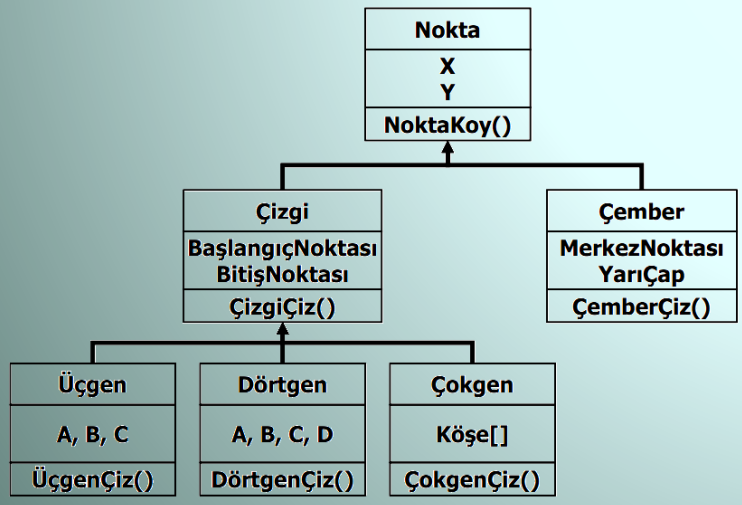
# Programlama Paradigmaları

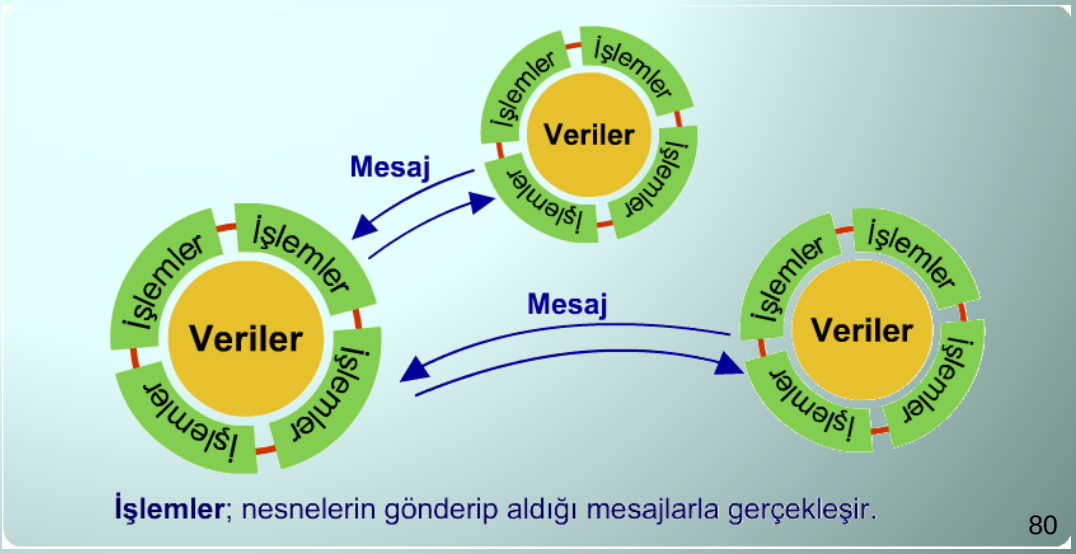



Imperative (Zorunlu, Komut merkezli) Diller

# Programlama Paradigmaları

- **Nesneye Yönelik Paradigmayı Destekleyen Diller**
  - Bu diller nesnelerin sınıflandırılması esasına dayanır.
  - Birbirini etkileyen nesnelerden oluşmuş diller olarak tanımlanabilir. Nesne, belli bir durumda paylaşılan işlemlerin bir topluluğudur. Veriler, fonksiyon ve prosedürler tek bir nesnede tutulabilir. Nesne temelli yaklaşımı kullanan dillere örnek olarak Simula, Smalltalk, C++, Java ve C# gibi diller verilebilir.



# Programlama Paradigmaları

- **Fonksiyonel Paradigmayı Destekleyen Diller**
  - Burada bilgisayara bir işlemi nasıl yapacağı bildirilir. Veriler ve sonucu elde etmek için *veriye uygulanacak fonksiyonel dönüşümler*, paradigmanın temelini oluşturur.
  - Sadece fonksiyonlar üzerine kurulmuş bir modeldir. Fonksiyonlar bir çok değer alır ve geriye sadece bir değer döndürürler. Fonksiyonlar başka fonksiyonları çağırır ya da başka fonksiyonun parametresi olur.

# Programlama Paradigmaları

- **Fonksiyonel Paradigmayı Destekleyen Diller**
  - Bu dillerde, alt yordamlar, fonksiyonlar (prosedürler) kullanılarak program daha alt parçalara bölünür.
  - İlk fonksiyonel dil Lisp'dir.
  - ML ve HASKELL modern fonksiyonel dillerdir. Bu diller fonksiyonlara rasgele değerler gibi davranmaktadır
  - ML ile Lisp karışık programlama dilleridir. Değişken ve atama işlemlerini desteklemektedir.
  - Haskell ise tamamen saf fonksiyonel bir dildir. Değişken ve atama işlemlerini desteklememektedir.

**Örnek**

(CAR (CAR '((A B) BC))) deyiminin liste ve fonksiyon bölümleri şunlardır:

| | |
|---|---|
| Liste | (A B) ve ((A B) B C) |
| Fonksiyon | CAR |

# Programlama Paradigmaları

## Scheme'de OBEB (Öklit) Algoritması

```
(define gcd2              ; 'gcd' is built-in to R5RS
  (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd2 (- a b) b))
          (else (gcd2 (- b a) a)))))
```

# Programlama Paradigmaları

- **Mantık Paradigmayı Destekleyen Diller**
  - Mantık programlama paradigmasında programlama, bir işin nasıl yapılacağının belirtilmesi yerine, ne yapılması istendiğinin belirtilmesi olarak görülür.
  - Bu diller, belirli bir koşulun varlığını kontrol ederek ve koşul sağlanıyorsa, uygun bir işlem gerçekleştirerek çalışırlar.
  - Bu modeldeki dillere en tanınmış örnek, Prolog programlama dilidir.

- Mantıksal diller kural tabanlı dillerdir. İmperative dillerde çözülecek algoritmanın tanımı ayrıntılı yapılır, komutların ve işlemlerin sırası bellidir.
- Mantıksal programlamada ise ayrıntı da yoktur. Sıralama da belli değildir. Programlama dili, sonucu elde etmek için hangi kuralı hangi sırayla kullanacağını belirlemelidir.
- Bu yaklaşım diğer dil kategorilerine göre çok radikaldir. Hiçbir kategoriye girmediği ve kendi başına ayrı bir kategori olduğu açıktır. En yaygın mantıksal dil Prolog'dur

# Programlama Paradigmaları

- **Mantık Paradigmayı Destekleyen Diller**
  - Mantık tabanlı bir dilin çalışması imperative bir dilin çalışmasına benzemekle birlikte, deyimler sıralı olarak işlenmez.
  - Bu dillerin sözdizimi genel olarak şu şekildedir:

```
koşul_1 -> hareket_1
koşul_2 -> hareket_2
..... ..... .....
koşul_n -> hareket_n
```

Örnek:

| | |
|---|---|
| Öğrenciler dersleri izler. | Murat bir insan mıdır? |
| Murat bir öğrencidir. | Murat ders izler mi? |
| Her öğrenci insandır. | |

| |
|---|
| Evet, Murat bir insandır. |
| Evet, Murat ders izler. |

# Programlama Paradigmaları

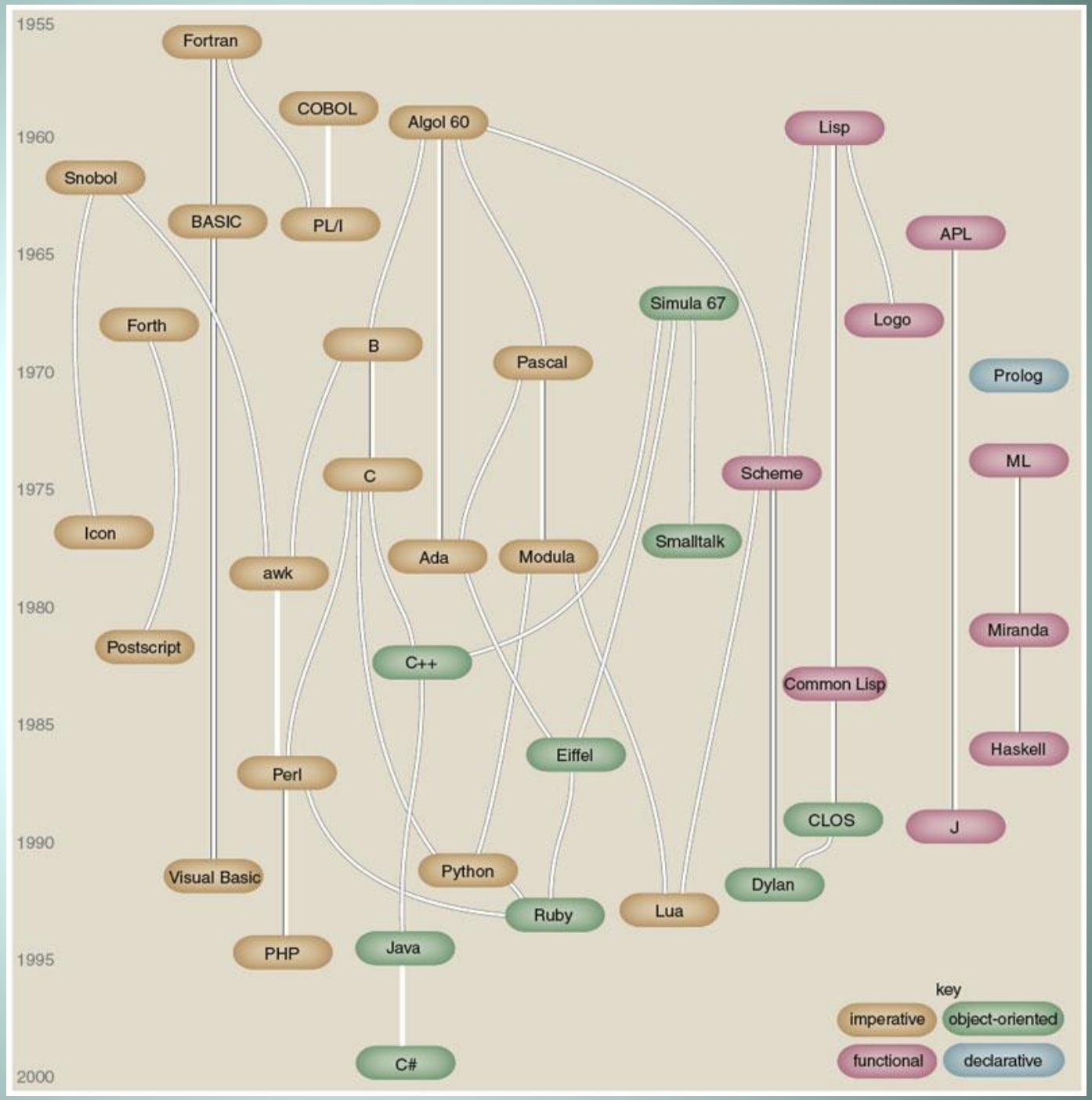
- **Mantık Paradigmayı Destekleyen Diller**
  - Doğal dil işleme, mantık programlamanın yaygın olarak kullanıldığı bir alandır.
  - Prolog dili, aşağıdaki örnekte olduğu gibi insan-makine etkileşimlerine uygundur. Prolog, özel bir mantık yürütme şeklini kullanarak kendisine yöneltilen sorgulara yanıt verir.
  - Gerçekler-Kurallar
    - 1-Ali insandır. 2-İnsanlar ölümlüdür. (Bilgisayar öğrenir)
  - Hedef
    - Ali ölümlüdür? (Bilgisayar gerçeklere bakarak cevap bulur
  - Sonuc:
    - Ali ölümlüdür.

# Programlama Paradigmaları

## Prolog'da OBEB (Öklit) Algoritması

```
gcd(A,B,G) :- A = B, G = A.
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

Şekil:
Brian Hayes, "The Semicolon Wars." American Scientist, July–August 2006, pp.299–303

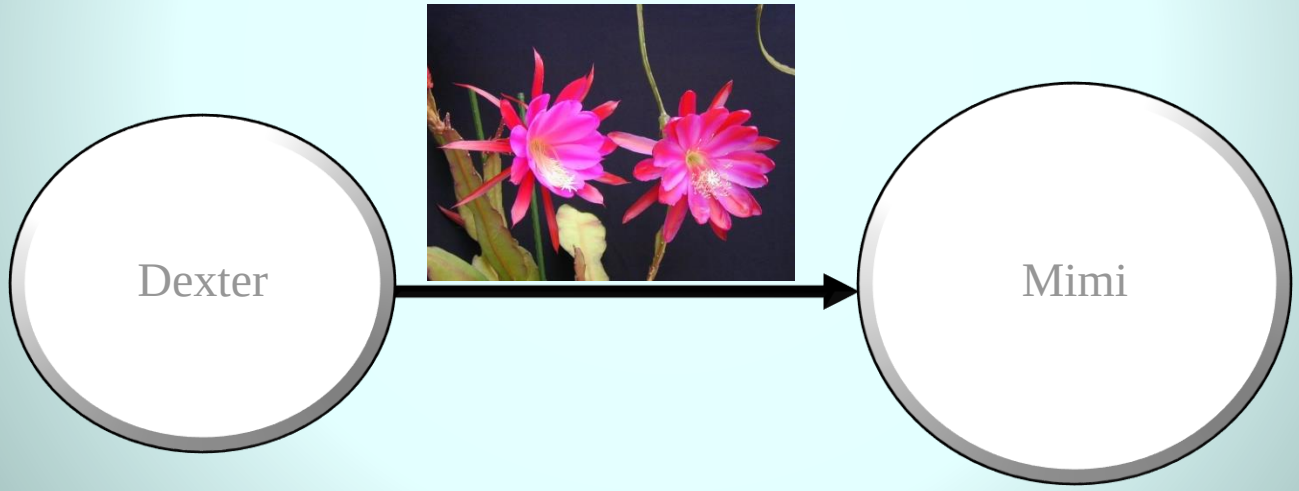


# Programlama Paradigmaları

- **Zorunlu (Imperative)**
  - Temel özellikleri değişkenler (variables), atama ifadeleri (assignment statements) ve iterasyondur (iteration)
  - Örnekler: C, FORTRAN, PL/I, Pascal, COBOL, Ada
- **Fonksiyonel (Functional)**
  - Hesaplamaları yapmanın temel yolu, fonksiyonları verilen parametrelere uygulamaktır
  - Örnekler: LISP, Scheme,ML
- **Mantıksal (Logic)**
  - Kural tabanlı (Rule-based) (kuralların sırası belirli değildir)
  - Örnek: Prolog
- **Nesneye-dayalı (Object-oriented)**
  - Veri soyutlama(Data abstraction), miras (inheritance), geç bağlama (late binding)
  - Örnekler: SIMULA 67, Smalltalk , C++ ve Java
- **İşaretleme (etiketleme) (Markup)**
  - Yeni; kendi başına programlama değildir, fakat Web dokümanlarında bilgiyi düzenlemek için kullanılır
  - Örnekler: XHTML, XML
- **Diğerleri**
  - Visual (görsel) diller
  - Scripting diller
  - Hibrid diller (markup dil/programlama dili)
  - Özel amaçlı diller
  - .....

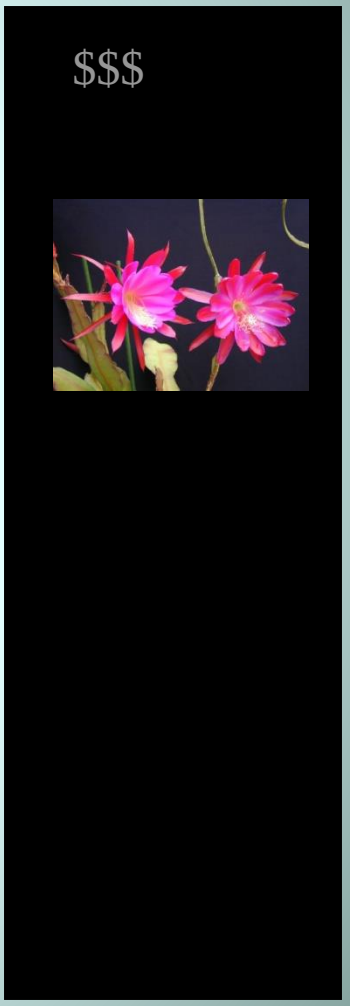
# Program Örneği

Dexter, Mimi'ye çiçek yollamak istemektedir

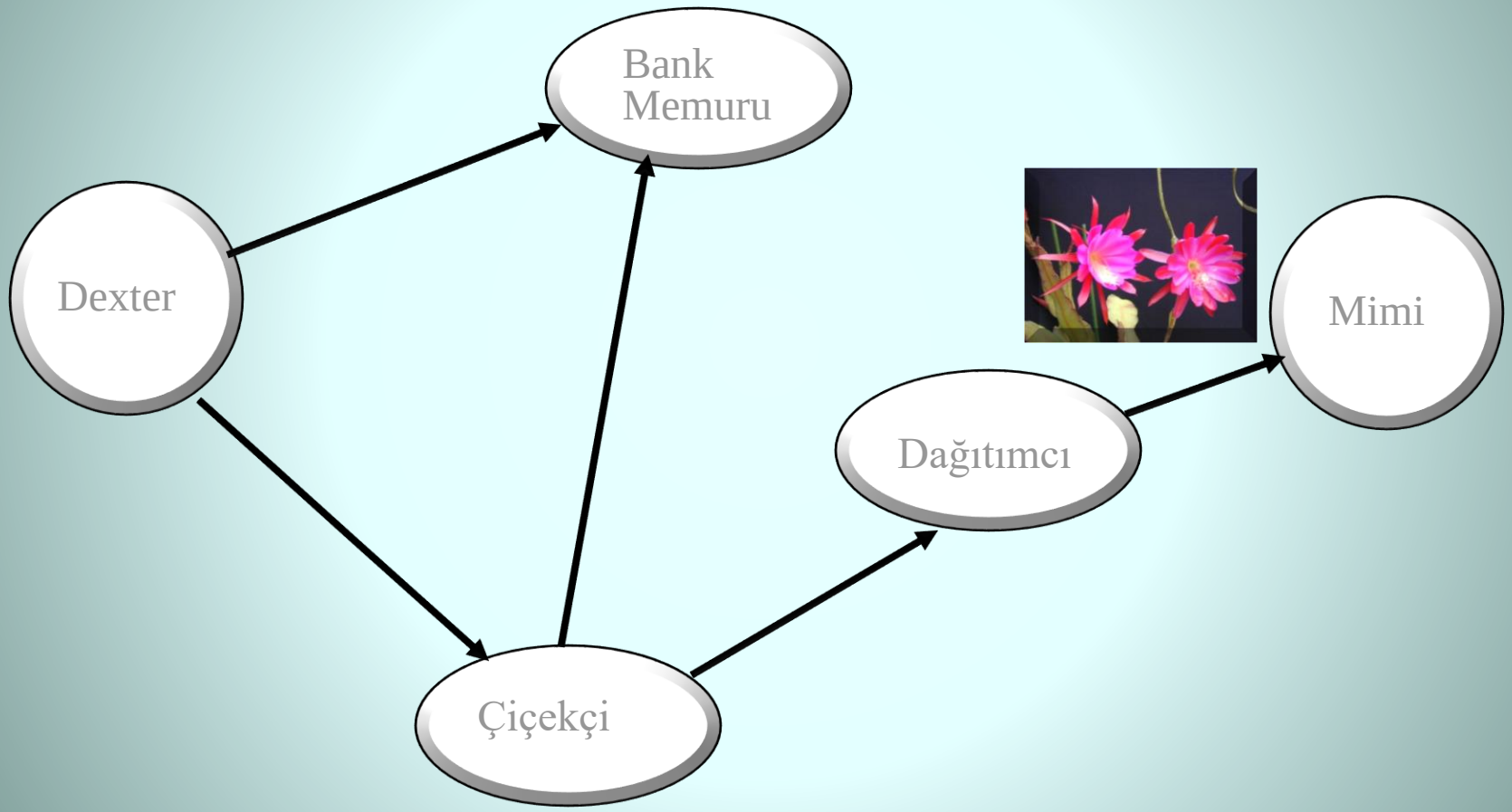


# Imperative Çözüm

- $$$ & çiçek için yer ayır
- Bankadan para al
- Parayı sakla
- Parayı al
- Çiçekçiye para ver
- Çiçekçiden çiçek al
- Çiçekleri sakla
- Gönderilen yere var
- Çiçekler al
- Çiçekleri yolla



# Nesne Tabanlı Çözüm



# Fonksiyonel Çözüm

- Neyin yapılmasına ihtiyaç var?
  - Nakit çekme
  - Çiçek satın alma
  - Çiçek dağıtma
- Hangi sırada olmalı?
  - Nakit -> Çiçek -> Dağıtım
- Fonksiyonların parametreleri ve dönüş değerleri vardır

```
Dagit( Satinal (Cek()))
```

# Mantıksal Çözüm

- Soruyu sor:

  *CicekAlir(Mimi)?*

- Kuralları ve gerçekleri görelim.

# Kurallar

CicekAlir(x): CicekDagitir(y, x)

CicekDagitir(y,x):

Cicekci(y) & CicekSiparisi(z, y) & Kisi(x)

CİcekSiparisi(x,y): Cicekci(y) & Kisi(x) &
ParasiVar(x) & ParaVerir(x, y)

# Gerçekler

ParasiVar(Dexter)

Cicekci(Faruk)

ParaVerir(Dexter, Faruk)

Şimdi soruyu sor:  CicekAlir(Mimi)?

???

Ne eksik?

# Gerçekler

Kopek(Dexter)

Kopek(Mimi)

CicekAlir(Mimi)?

HAYIR

# Problem Örneği

Dexter, Mimi'ye çiçek yollamak istiyor

# Programlama Paradigmaları

- Nesne tabanlı dil kavramının, imperative dillerde kullanılan prosedür tabanlı dil kavramından çok farklı olmasına rağmen imperative bir dilin nesne tabanlı programlamayı desteklemesini sağlamak çok zor değildir.
- Örneğin atama ve kontol ifadeleri C ve Java'da hemen hemen aynıdır (diğer taraftan diziler, altprogramlar ve Java'nın semantiği; C'den oldukça farklıdır).
- Benzer ifadeler nesne tabanlı programlamayı destekleyen fonksiyonel diller için de geçerlidir.

# Programlama Paradigmaları

## Visual (görsel) diller

- Visual diller ayrı bir kategori olarak görülse de, imperative dillerin alt kategorisindedir.
- En popüler olanı Visual BASIC.NET (VB.NET)
- Bu diller sürükle bırak (drag and drop) mantığıyla çalışırlar.
- Visual diller kullanıcıya rahat grafiksel ara yüzler tasarlamak için kullanılırlar. Örneğin bir düğme veya bir text kutusu gibi bir kontrol yapısı kodlamak için VB.NET'de tek bir tuşa basmak yeterlidir. Bu özellikler günümüzde tüm .NET dillerinde vardır.

# Programlama Paradigmaları

## Scripting diller

- Bazı bilim adamları bu dilleri ayrı bir kategori olarak görürler. Ama bu diller de imperative dillerdir. Bu dillere örnek verilirse; Perl, JavaScript, Ruby

# Programlama Paradigmaları

## Hibrid (melez) diller

- Son yıllarda hibrid dil (Markup/programlama dili) kategorisi denen bir kavram ortaya çıkmaktadır.
- Markup dilleri programlama dili değildir. Örneğin XHTML
- XHTML veya XML gibi markup dillerin bazı versiyonlarının içine bazı programlama dili kabiliyetleri serpiştirilmiştir.
- Örnek: JSTL (Java Server Pages Standart Tag Library)
- XSLT (eXtensible Stylesheet Language Transformations)

# Programlama Paradigmaları

## Özel amaç için tasarlanmış diller

- Son 50 yılda özel uygulamalar için bir çok programlama dili tasarlanmıştır.
- Örnek:

RPG (Report Program Generator): Ticari raporlar üretmek için tasarlanmıştır.

APT (Automatically Programmed Tools): Programlanabilir makinalar için tasarlanmıştır.

GPSS (General Purpose Simulation System): Sistem simulasyonu için tasarlanmıştır.

# 1.6 Dil Tasarımında Verilen Ödünler

- Dil gelişimi ile ilgili kriterler önceki bölümlerde anlatıldı. Bu kriterler bir programlama dili tasarlanırken gerekli çatı hakkında temel ipuçlarını verdi. Fakat bu çatıda birbiri ile çelişen bir çok durum var.
- “Programlama dili tasarlanırken çok önemli bir çok kriter vardır, ama bu kriterler birbirleri ile çelişmektedir. Bu çelişkili kriterlerle iyi bir dili tasarlamak, çok zor bir mühendislik görevidir.” Hoare (1973)

# 1.6 Dil Tasarımında Verilen Ödünler

- Güvenilirlik ve yürütme (execution) maliyeti
  - Çelişen kriterlerdir
  - Örnek: Java ‘da tüm dizilerin indislerinin doğru aralıkta olup olmadığının kontrolü yapılır. Bu işlem Java’da execution maliyetini çok fazla artırır.
  - C’de ise dizilerin indislerinin aralık kontrolü yapılmaz . Dolayısıyla C’de yazılan programın semantik olarak aynı olanı Java’da yazılırsa C’de yazılan program daha hızlı execution yapar.
  - Java programlar ise daha güvenilirdir. Java tasarımcıları güvenlik için execution verimliliğinden ödün vermişlerdir.

# 1.6 Dil Tasarımında Verilen Ödünler

- Okunabilirlik ve yazılabilirlik
  - Diğer çelişen kriterler
  - Örnek : APL pek çok güçlü operatör (ve çok sayıda yeni sembol) sağlayarak karmaşık hesaplamaların kolayca yapılabilmesine imkan sağlarken okunabilirliği azaltmaktadır.
  - «4 satırdan oluşan APL programın anlamak için 4 saat uğraştım» Daniel McCracken
  - APL tasarımcıları yazılabilirlik için okunabilirlikten ödün vermişlerdir.
- Yazılabilirlik ve güvenilirlik
  - Diğer çelişen kriterler
  - Örnek : C++ pointerları çok güçlü ve esnek olmasına rağmen güvenilirliği azdır. Bu yüzden Java'da pointer kavramı yoktur.

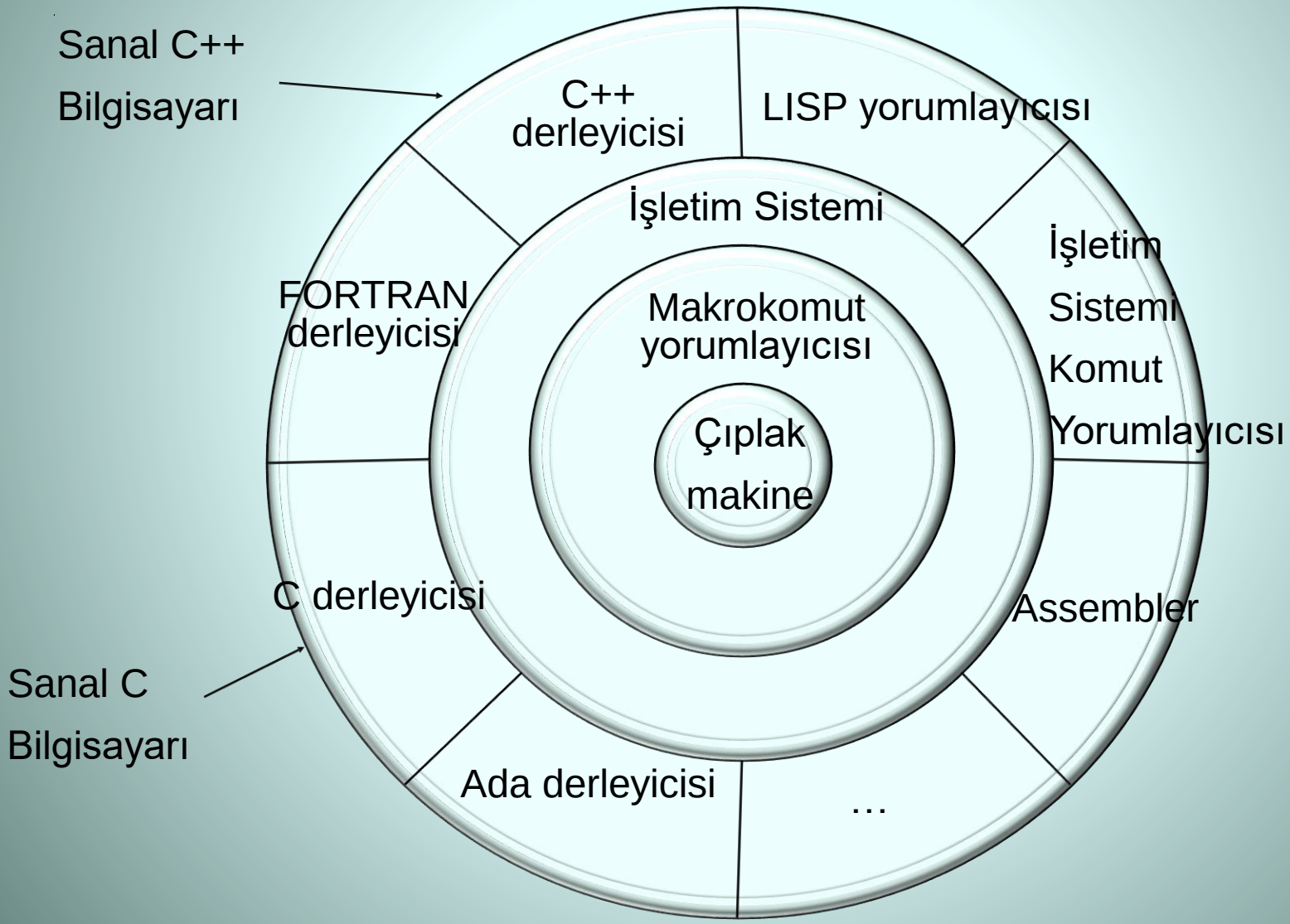
# 1.7 İmplementasyon Metotları

- Bilgisayarın iki temel bileşeni vardır. Dahili hafıza ve işlemci.
- Dahili hafıza, programları ve verileri depolamak için kullanılır.
- İşlemci ise mantıksal ve aritmetiksel işlemleri yapmak için tasarlanmış elektronik devrelerdir. İşlemci komutlara göre işlemleri yapmaktadır. Bu komutlara makrokomutlar da denir. Makro komutlar mikrokomut kümelerinden oluşur.

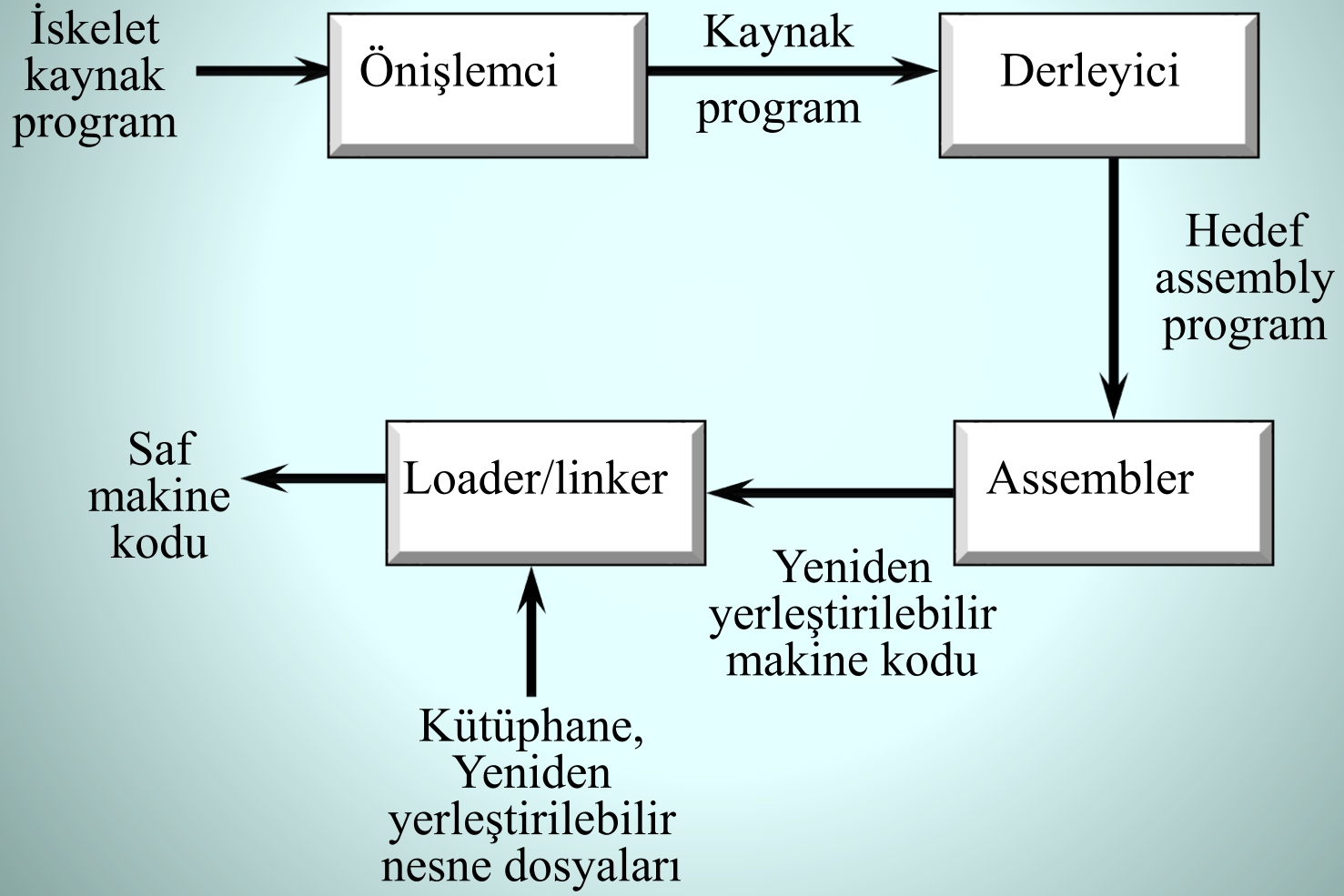
# 1.7 İmplementasyon Metotları

- Derleme (Compilation)
  - Programlar makine diline çevirilir
- Saf Yorumlama (Pure Interpretation)
  - Programlar yorumlayıcı (interpreter) olarak bilinen başka bir program tarafından yorumlanır
- Hibrit (melez--Hybrid) İmplementasyon Sistemleri
  - Derleyiciler (compilers) ve saf yorumcular (pure interpreters) arasındaki uzlaşmadır

# Bilgisayarın katmanlı (layered) görünümü

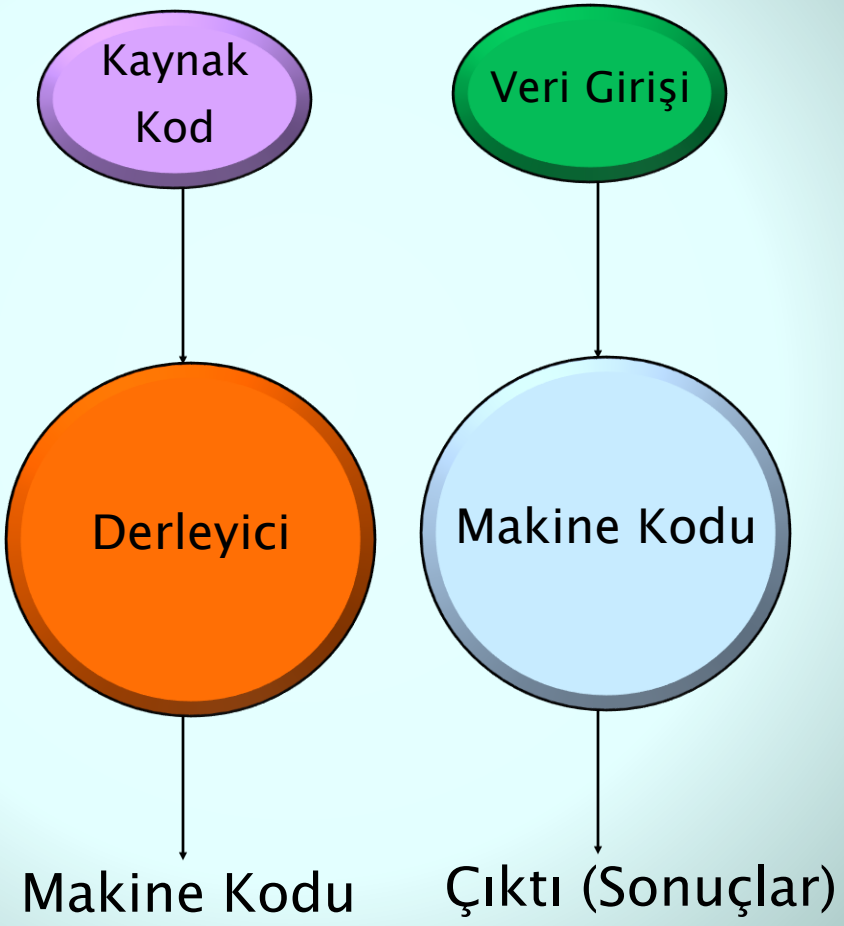


# Geleneksel Çevirici

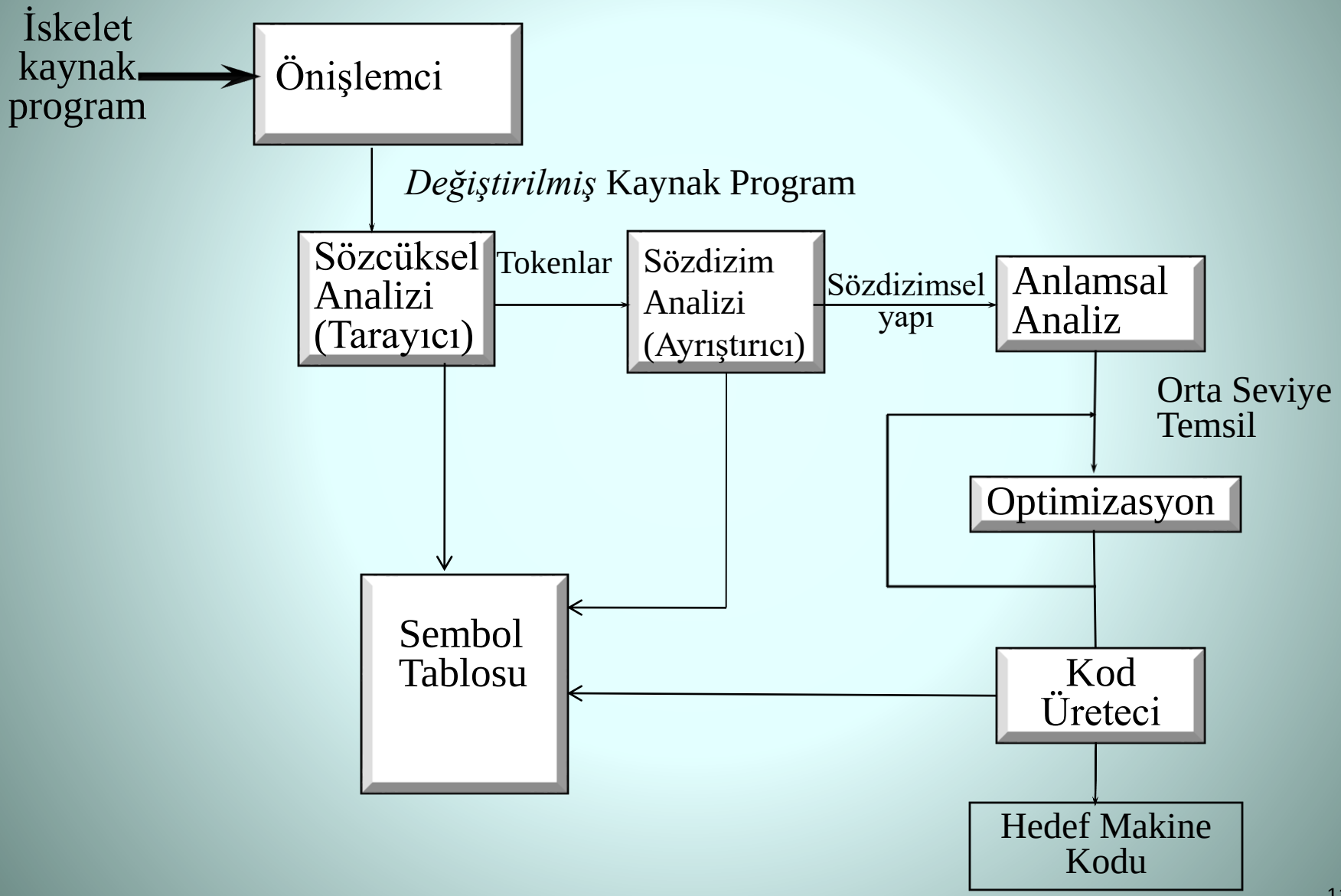


# Derleme (Compilation)

- Yüksek-düzeyli (high-level) programı (kaynak dil--source language), makine koduna (machine code--machine language) çevirir
- Yavaş çeviri, hızlı yürütme (execution)
- Derleme işlemi birkaç evreden oluşur:
  - Sözcüksel analiz (lexical analysis): kaynak programdaki karakterleri sözcüksel(lexical) birimlere çevirir
  - Sentaks Analiz (sözdizim-syntax analysis): sözcüksel birimleri *gramer/sözdizim ağaçlarına* (*parse trees)* dönüştürür, bu ağaçlar programın sözdizimsel (syntactic) yapısını gösterir
  - Semantik Analiz (anlamsal-Semantics analysis): ara (intermediate) kod üretilir
  - Kod üretimi: makine kodu üretilir

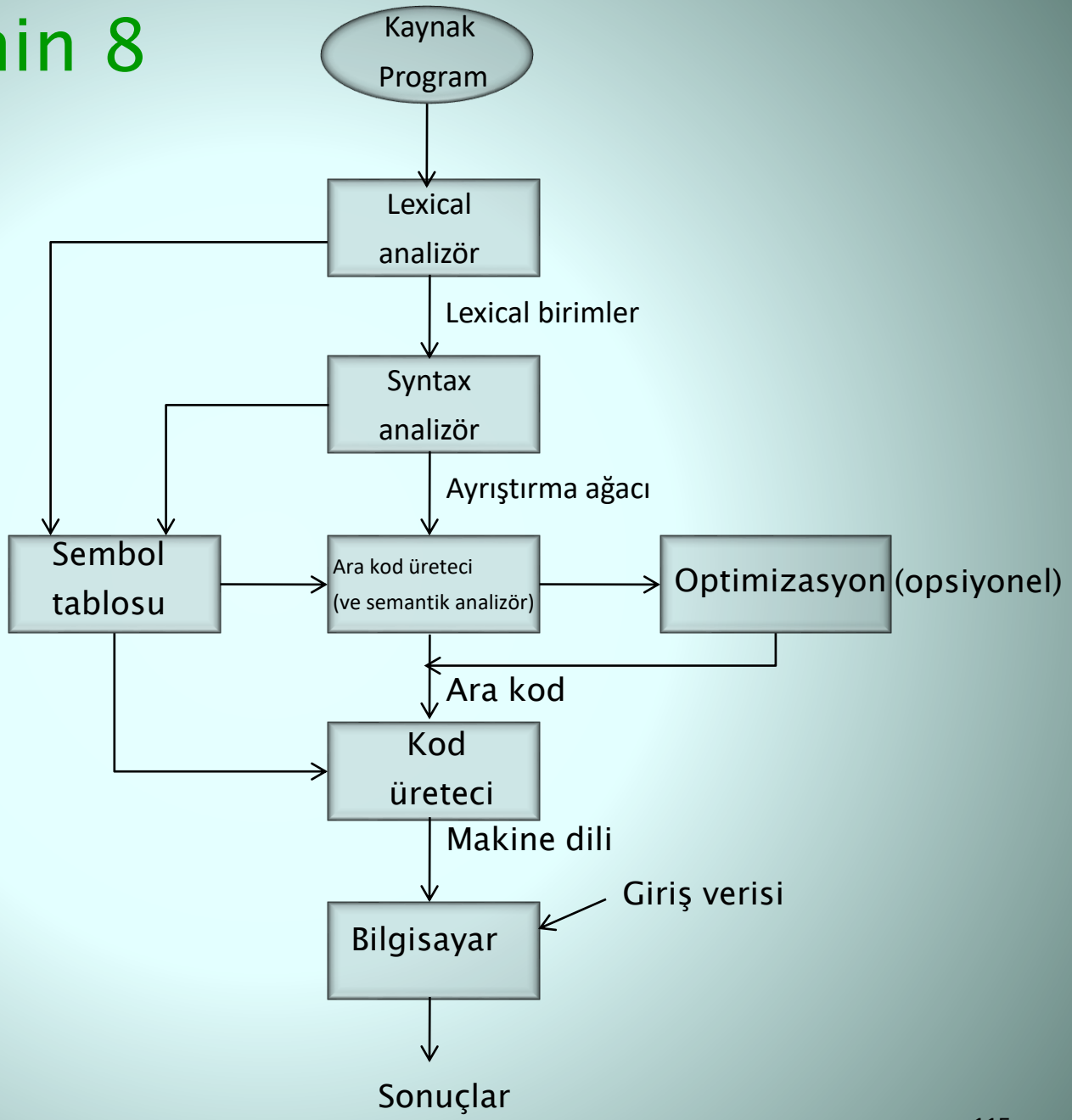

Kaynak Kod
Veri Girişi
Derleyici
Makine Kodu
Makine Kodu
Çıktı (Sonuçlar)

# Derleme


İskelet kaynak program
Önişlemci
Değiştirilmiş Kaynak Program
Sözcüksel Analizi (Tarayıcı)
Tokenlar
Sözdizim Analizi (Ayrıştırıcı)
Sözdizimsel yapı
Anlamsal Analiz
Orta Seviye Temsil
Optimizasyon
Sembol Tablosu
Kod Üreteci
Hedef Makine Kodu

# Bir Derleyicinin 8 Fazı

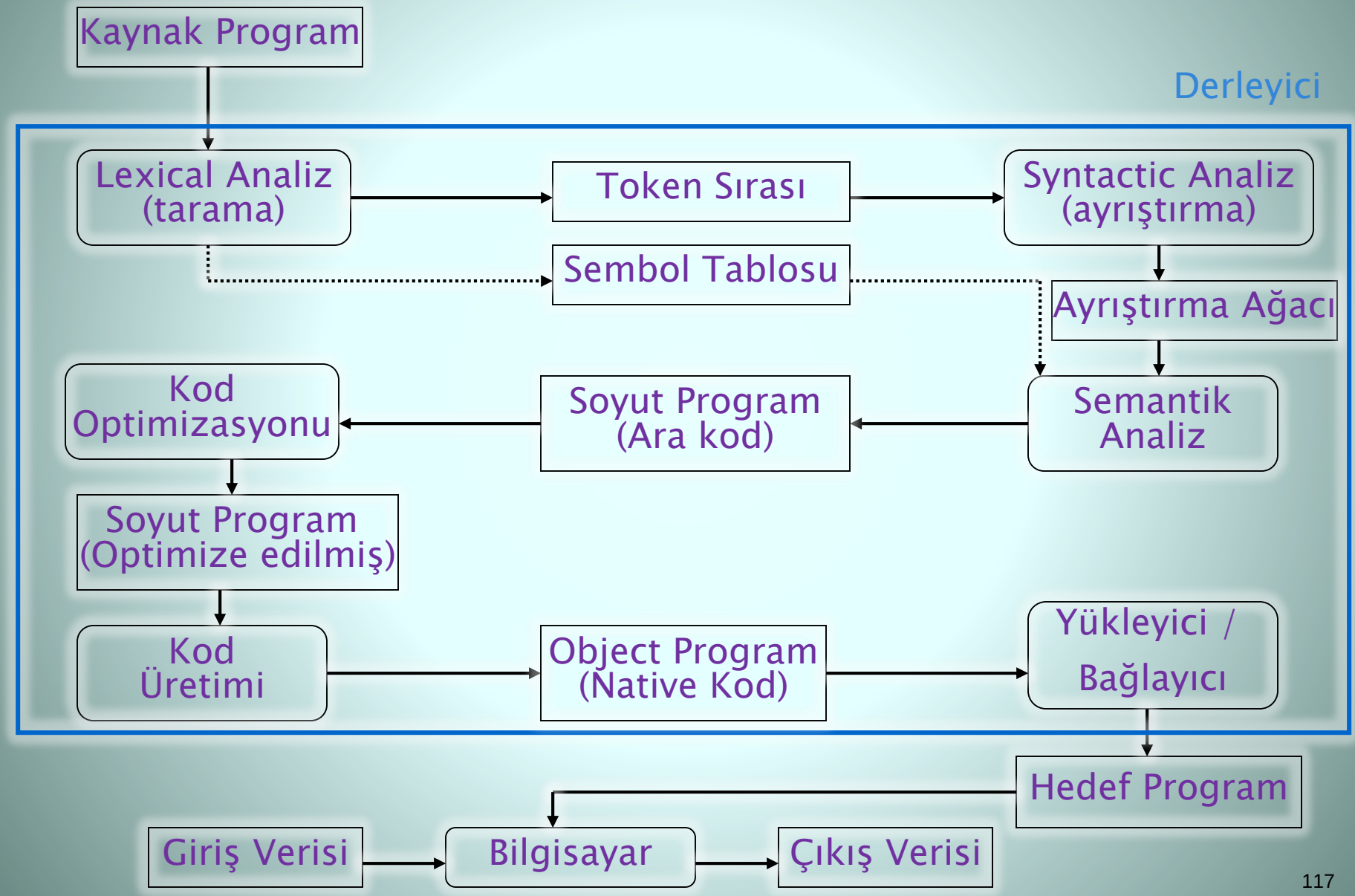
- 6 ana faz
  1. Lexical analizör
  2. Syntax analizör
  3. Semantic analizör
  4. Ara kod üreteci
  5. Kod optimizsyonu
  6. Kod üreteci
- İki aktivite daha (yukarıdaki 6 fazla paralel)
  1. Sembol tablosu yöneticisi
  2. Hata yöneticisi



# Biraz daha derleme terminolojisi

- **Yükleme Modülü (Load module)** (yürütülebilir görüntü–executable image): kullanıcı ve sistem kodu birlikte
- **Bağlama ve Yükleme (Linking and loading)**: sistem programını toparlama ve kullanıcı programa bağlama

# Derleme ve Yürütme



# Makine kodunun yürütülmesi

- (Bulma-yürütme-çevrimi) Fetch-execute-cycle (bir von Neumann mimarisi üzerinde)

```
initialize the program counter (program sayacını başlat)
repeat forever
    fetch the instruction pointed by the counter
                                (sayacın gösterdiği komutu bul)
    increment the counter   (sayacı arttır)
    decode the instruction  (komutu çöz)
    execute the instruction (komutu yürüt)
end repeat
```

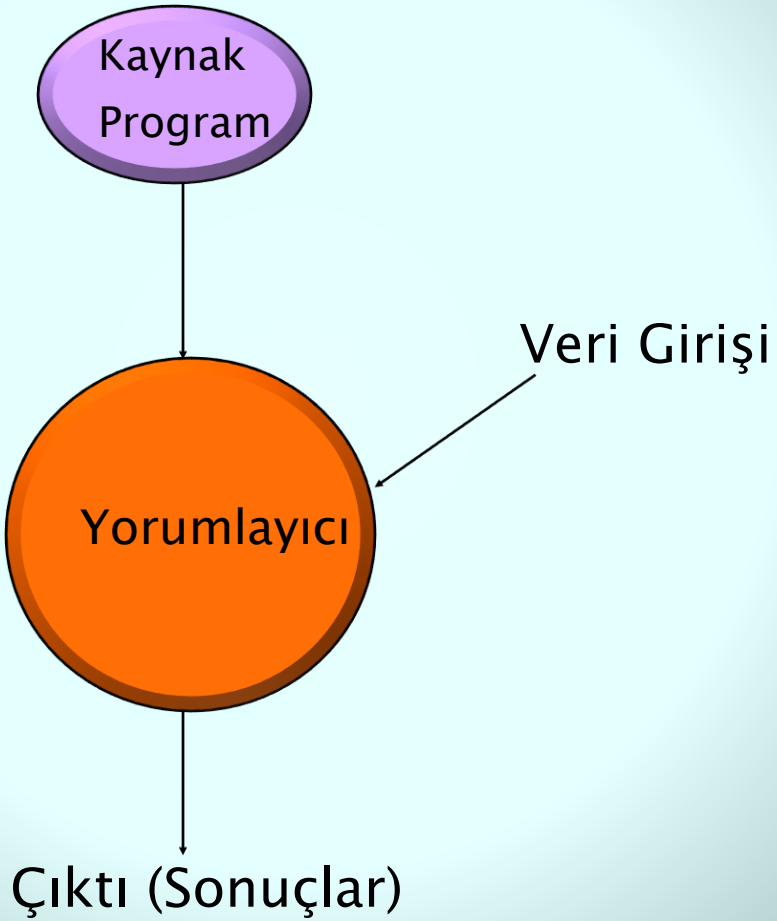
# Von Neumann Darboğazı (Bottleneck)

- Bir bilgisayarın belleği ve işlemcisi arasındaki bağlantı hızı o bilgisayarın hızını belirler
- Program komutları (instructions) çoğu kez yukarıda bahsedilen bağlantı hızından çok daha hızlı yürütülebilir; bağlantı hızı bu yüzden bir *darboğaza* (*bottleneck)* sebep olur
- Von Neumann darboğazı (bottleneck) olarak bilinir; bilgisayarların hızını sınırlayan birinci faktördür

# Saf Yorumlama (Pure Interpretation)

- Çeviri (translation) yoktur
- Programların implementasyonu daha kolaydır (çalışma-zamanı hataları (run-time errors) hemen ve kolayca gösterilir)
- Daha yavaş yürütme (Derlenmiş programlardan 10 ile 100 kat daha yavaştır)
- Çoğu kez daha fazla boş alana ihtiyaç duyar
- Yüksek-düzeyli (high-level) dillerde gittikçe daha nadir kullanılmaktadır
- Bazı Web yazı dillerde (Web scripting languages) kullanımı önem kazanır (örn., JavaScript)

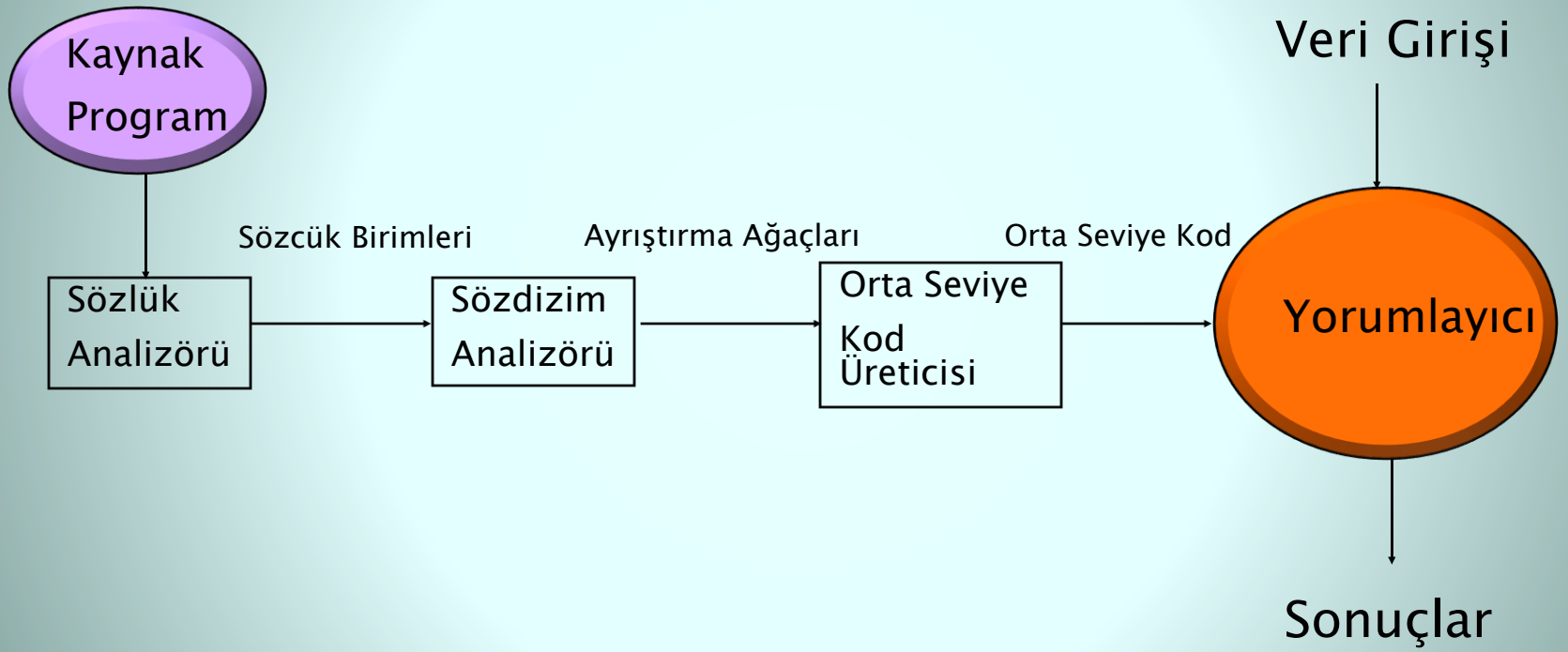
# Saf Yorumlama (Pure Interpretation) İşlemi



# Hibrit İmplementasyon Sistemleri

- Derleyiciler (compilers) ve saf yorumlayıcılar (pure interpreters) arasında bir uzlaşmadır
- Bir yüksek-düzeyli dil programı kolayca yorumlanabilecek bir ara (intermediate) dile çevrilir
- Saf yorumlamadan daha hızlıdır
- Örnekler
  - Perl programları, yorumlamadan önce hataları tespit etmek için kısmen derlenir
  - Java'nın ilk implementasyonları hibritti; ara form, yani ***bayt kodu*** (*byte code)*, bir bayt kodu yorumlayıcısı (byte code interpreter) ve bir çalışma-zamanı(run-time) sistemi olan herhangi bir makineye (*Java Sanal Makinesi - Java Virtual Machine*) taşınabilirlik sağlar (bu makinede çalışır)

# Hibrit İmplementasyon İşlemi



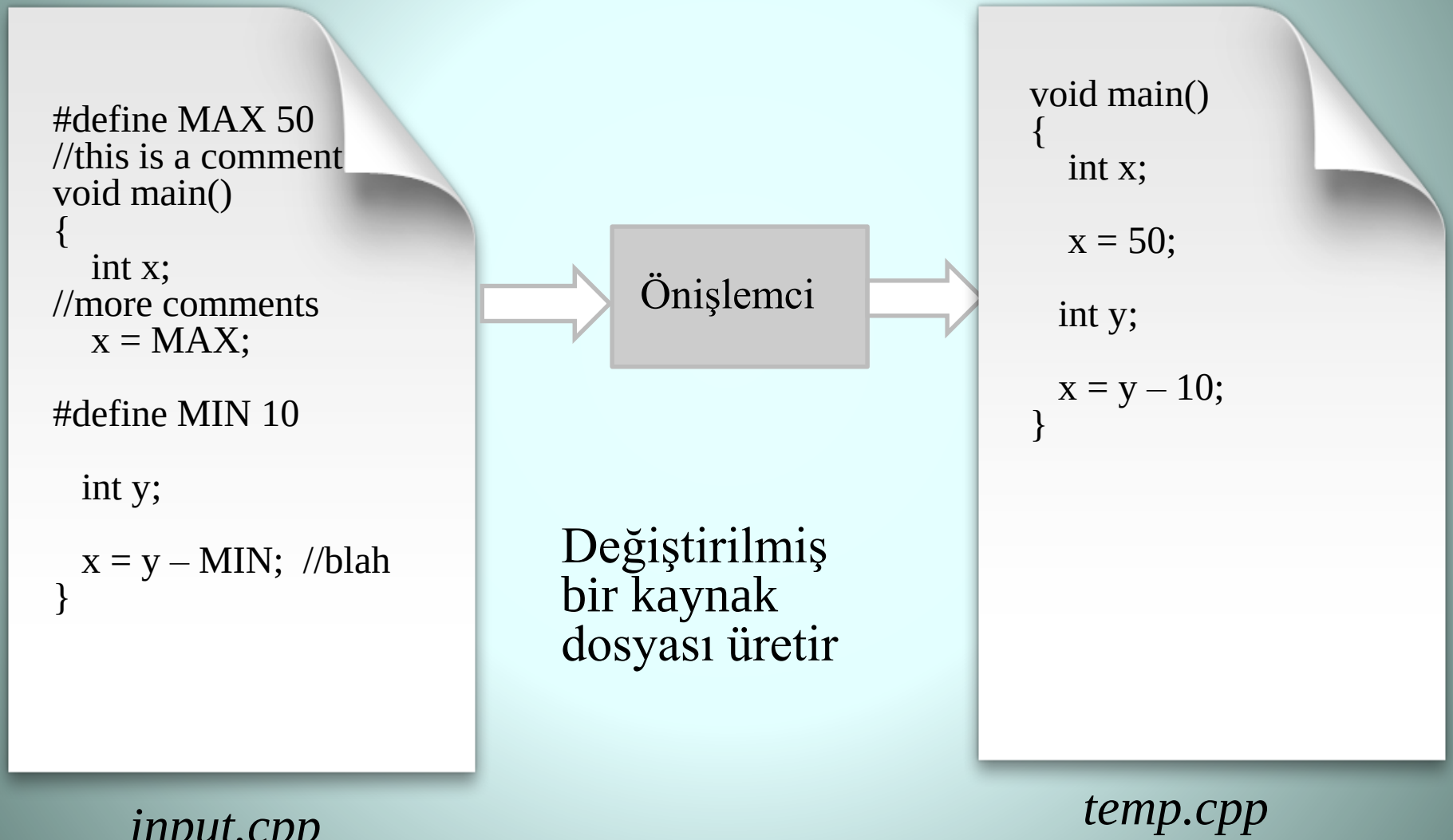
# Tam-zamanında (Just-in-Time) Implementasyon Sistemleri

- Önce programları bir ara (intermediate) dile çevirir
- Sonra ara dili makine koduna derler (machine code)
- Makine kodu sürümü (version) birbirini takip eden çağırmalar (subsequent calls) için tutulur
- JIT sistemleri Java programlarında çok kullanılır
- .NET dilleri bir JIT sistemiyle geliştirilmiştir

# Önişlemciler (Preprocessors)

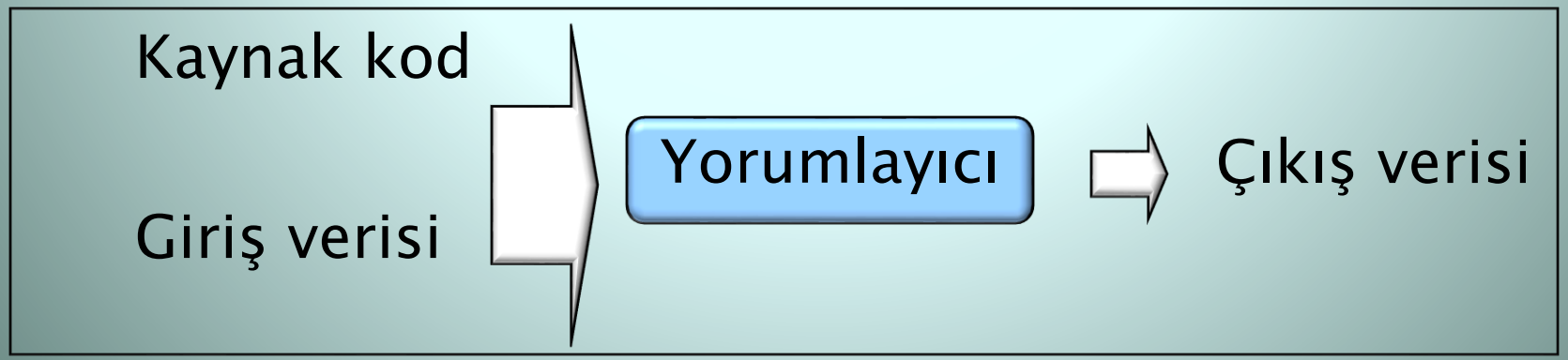
- Önişlemci makroları (macros (instructions-komutlar)) başka bir dosyadan dahil edilecek kodu belirtmek için yaygın olarak kullanılır
  - Tüm yorumları kaldırır
  - Dil büyük-küçük harf duyarlı değilse tüm program metnini büyük harf ya da küçük harfe değiştirebilir
- Bir önişlemci, gömülü (embedded) önişlemci makrolarını genişletmek için programı derlenmesinden hemen önce işler
- İyi bilinen bir örnek: C önişlemcisi
  - **`#include`**, **`#define`**, ve benzeri makroları genişletir

# Önişlemciler( Preprocessors)



# Derleyicilere Karşı Yorumlayıcılar

- Çeviri yürütmeden ayrı mı?
  - Evet – Derleyici (Compiler)
  - Hayır – Yorumlayıcı (Interpreter)
- Birleşik uygulama (örneğin Java)

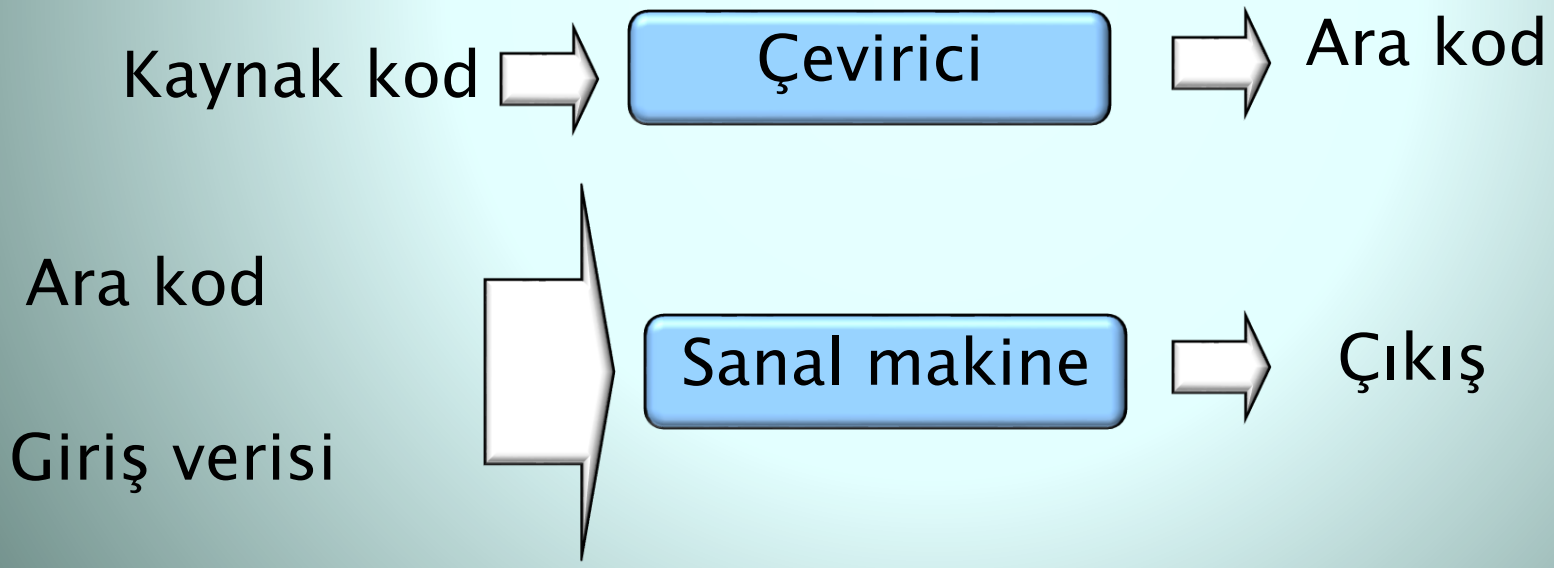



# Neden Derleyici

- Temel mühendislik prensipleri
  - doğruluk – erken statik hata kontrolü
  - maliyet – derleme program dağıtma maliyetini düşürür.
  - performans – hızlı çalışır
    - bir defa derle (maliyet), birçok defa yürüt (fayda)
- Yazılımın/fikirlerin korunması


# Neden Yorumlayıcı

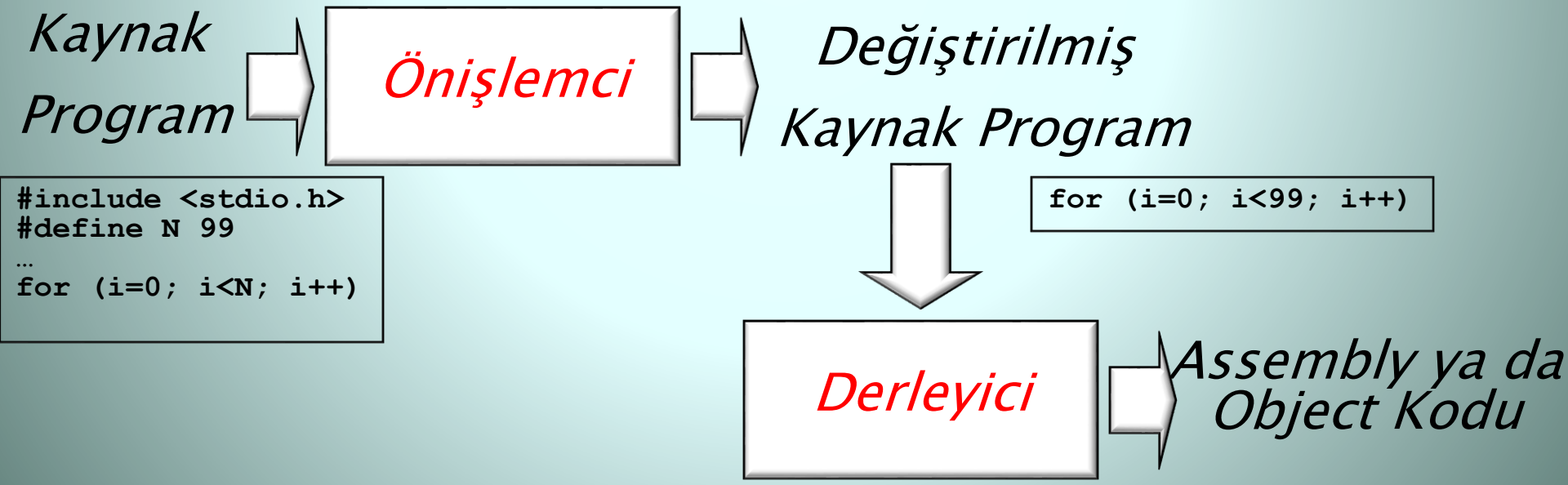
- Esneklik (yürütüm zamanı oluşan geç bağlantılar nedeniyle)
- Yürütüm zamanı durum desteği
  - komut dosyaları (Perl, Shells, Python,TCL)
  - dinamik ortamlar (Basic, APL, LISP)
  - sanal makineler (JVM, Emulators, CPUs).



# Çok Geçişli Derleyiciler

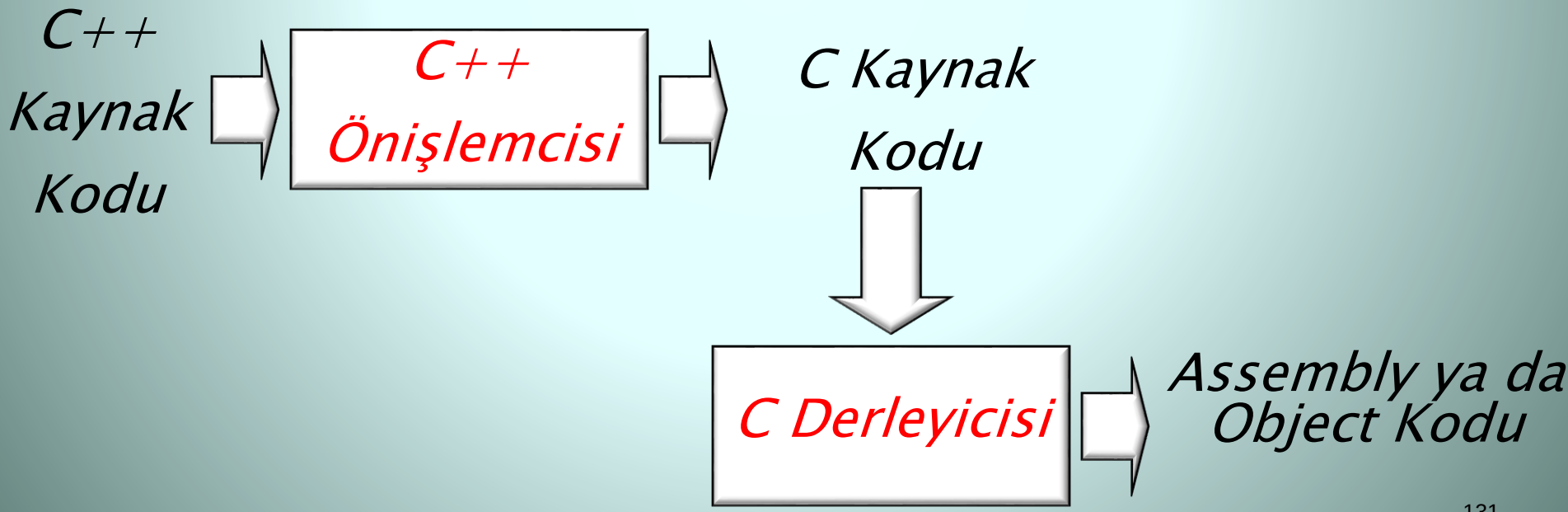
- Karmaşıklığı nasıl çözümleyelim?
  - Program kitaplığı ile (dili basit tutarak, örneğin Java).
  - Katmanlaştırarak (her seferinde bir problem üzerinde yoğunlaşarak)
    - Peş peşe fazlardan oluşan çözüm.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```



# Başka Dile Derleme

- Bazı derleyiciler çevirici diline derler (assembly code)
  - Yüksek oranda optimize olur
- Bazı diğer derleyiciler başka üst seviye dile derleyebilir.
  - Var olan dilin optimizasyonunu kullanılır.
  - Taşınabilirliği artırır, karmaşıklığı azaltır.



# 1.8 Programlama Platformları

- Yazılım geliştirmede kullanılan araçlar (tools) koleksiyonuna programlama platformu denir.
- Bu platform; bir dosya sistemi, text editor, linker, ve compiler içerebilir veya çok fazla birleşik tool'dan oluşan ve tool'lar arası geçişi grafiksel ara yüzle sağlanan bir platformda olabilir.
- Tüm bunlardan dolayı yazılım geliştirme sisteminin kabiliyetini ölçmek için tek yöntem programlama dilinin karakteristiği değildir.

# 1.8 Programlama Platformları

- UNIX
  - Eski bir işletim sistemi ve araç koleksiyonudur.
  - UNIX'in ilk versiyonlarının en büyük eksiği araçları arasında geçiş için grafiksel özelliğinin olmamasıydı. Bu eksiklik yüzünden öğrenilmesi ve kullanılması zor bir platformdu. Bu eksiklik günümüzde giderildi ve unix'e GUI özelliği eklendi. UNIX GUI'lere örnek: Solaris Common Desktop Environment (CDE), GNOME ve KDE. Bu GUI'ler sayesinde Unix'i Windows ve Macintosh sistemlerine benzer şekilde görünmesini sağlamıştır.

# 1.8 Programlama Platformları

- Borland JBuilder
  - Java için bir tümleşik geliştirme platformu (integrated development environment)
  - Compiler, editor, debugger ve dosya sistemini birleşik olarak kullanıma sunan bir platformdur. Bu dört özelliğe grafiksel ara yüzle erişilir. Jbuilder, Java yazılımı geliştirmek için güçlü ve kompleks bir sistemdir.
- Microsoft Visual Studio.NET
  - Büyük, karmaşık bir görsel platform (visual environment)
  - Pencere ara yüzlerle araçlarına erişilir. Çok fazla yazılım geliştirme aracına sahiptir.
  - Bu platform aşağıda isimleri verilen 5 farklı dilde program geliştirmek için kullanılır. C#, Visual BASIC.NET, Jscript (Javascript'in Microsoft versiyonu), J# (Java'nın Microsoft versiyonu), veya C++ ile programlama yapılır

- NetBeans, öncelikli kullanım amacı Java ile web uygulaması gerçekleştirmektir. Aynı zamanda Javascript, Ruby, ve PHP'yi de desteklemektedir.
- Hem NetBeans hem de Visual Studio programlama platformu oldukları gibi aynı zamanda framework özelliği de taşırlar.

# Özet

- Programlama dillerini öğrenmenin öneminin bazı nedenleri:
  - Farklı yapıları kullanabilme kapasitemizi arttırması
  - Dilleri daha akıllıca seçebilir hale gelmek
  - Yeni dilleri öğrenmeyi kolaylaştırması
- Dil değerlendirme kriterlerinin en önemlileri:
  - Okunabilirlik (Readability), yazılabilirlik (writability), güvenilirlik (reliability), maliyet (cost)
- Dil tasarımı etkileyen ana faktörler makine mimarisi (machine architecture) ve yazılım geliştirme metodolojileridir
- Programlama dillerinin implementasyonunun ana metotları: derleme (compilation), saf yorumlama (pure interpretation), ve hibrit implementasyon

# 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, Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları