## Konular

1. **Programlama dilleri kavramlarını neden öğrenmeliyiz?**
2. **Programlama Alanları**
3. **Dil Değerlendirme Ölçütleri**
4. **Dil Tasarımını Etkileyen Unsurlar**
5. **Dil Sınıfları**
6. **Dil Tasarımında Fayda-Zarar Dengesi**
7. **Gerçekleştirim Yöntemleri**

## Giriş

Programlama dilleri kavramları dersinin temel amacı, modern programlama dillerinin ana yapılarını tanıtmak ve derleyici tasarımına dair ön bilgiler vererek öğrenciyi hazırlamaktır.

## Programlama Dilleri Kavramlarını Neden Öğrenmeliyiz?

- *Fikirlerimizi daha iyi ifade edebilme*

Bir doğal dilin ne kadar güçlü olduğu insanların düşünme derinliğini etkilemektedir. İnsanlar nasıl ifade edeceklerini bilmedikleri yapıları zihinlerinde canlandırmazlar. Yazılım geliştiriciler de aynı kısıtlarla karşı karşıyadır. Kullandıkları programlama dili, kullanabilecekleri kontrol yapıları, veri yapıları ve soyutlamalar bakımından yazılımcıları kısıtlar. Programlama dillerindeki özelliklerin çeşitliliği bu tip kısıtları azaltmaktadır.
Bazı bilgisayar mühendisliği öğrencileri çalışma hayatında sadece 1-2 programlama dili kullanacaklarını hatta çalıştıkları yerde belirli dilleri kullanmaya zorlanacaklarını dolayısıyla programlama dilleri kavramları konusunda bilgi sahibi olamanın gerekli olmadığını düşünürler. Ancak programlama dilleri kavramlarına aşina olmak, bir dilin herhangi bir özelliğini, diğer bir dilde tasarlayabilme yeteneği kazandıracaktır.

- *Uygun programlama dilini seçebilmek için altyapımızı zenginleştirme*

Bir projede hangi dilin kullanılacağını seçerken en iy bildiğimiz dile yöneliriz ve o dilin gerçekten proje ihtiyaçlarını karşılamada yeterli olup olmadığına bakmayız. Eğer programlama dillerinde kullanılan yapılar konusunda daha geniş bilgimiz olursa daha iyi bir seçim yapabiliriz. Elbette bir dildeki yapının diğer bir dilde tasarlanabilme imkanı vardır ancak bu, hem maliyetli hem de daha az güvenli olması sebebiyle çok tercih edilen bir durum değildir. Onun yerine, istenen özelliği bünyesinde barındıran dili tercih etmek daha doğru bir yaklaşım olur.

- *Yeni bir dil öğrenmede kolaylık*

Bilgisayar programlama nispeten yeni bir alan ve tasarım yöntemleri, yazılım geliştirme araçları ve programlama dilleri halen sürekli bir gelişim içinde. Bilgisayar mühendisliği heyecan verici olduğu kadar sürekli öğrenmenin kaçınılmaz olduğu bir meslektir.

Öğrenilen programlama dilleri ve tasarım metodolojileri de bir zaman sonra popülerliğini yitirebilir. Bu durumda yeni bir dil öğrenmek gerekebilir. Yalnızca 1-2 dil bilen ve programlama dilleri kavramlarından haberdar olmayan bir kişi için zaten zor olan yeni bir dil öğrenme işi iyice zorlaşacaktır. Dillerin genel kavramlarını öğrenince, yeni öğrenilen dilde bu kavramların nasıl uygulandığına bakmak dil öğrenimini hızlandırıp kolaylaştırabilir. Örneğin nesneye dayalı programlama kavramını bilen bir kişinin Java dilini öğrenmesi bilmeyene kıyasla daha kolay olacaktır.

- *Bilinen bir dilin daha etkin kullanımı*

Çoğu modern programlama dili çok kapsamlı ve karmaşık yapıya sahiptir. Yazılım geliştiriciler genellikle bir dilin tüm özelliklerini bilmez ve kullanmaz. Programlama dilleri kavramlarını öğrenerek, bir dilin bilinmeyen ve kullanılmayan yönleri konusunda bilgi sahibi olup onları kullanmaya başlayabiliriz.

- *Gerçekleştirimin öneminin anlaşılması*

Gerçekleştirim konularının daha iyi anlaşılması, bir dilin neden o şekilde tasarlandığı konusunda bize fikir verir ve dili daha akıllıca kullanmaya başlarız. Kullandığımız veri yapılarını neden seçtiğimizi, bu seçim sonucunda ne olacağını bilirsek daha iyi yazılımcı olabiliriz. Ayrıca belirli tiplerdeki hataları ayıklamak kolaylaşır. Bunun yanısıra, bilgisayarın komutları ve yapıları nasıl işlediğini kafamızda canlandırabiliriz. Örneğin, belirli bir altyordamı çağırmanın karmaşıklığını bilmeyen bir kişi, bu altyordamı çok sık çağırdığında tasarımının etkinliği hakkında bir fikir yürütemez.

- *Genel programlama becerisinin artması*

Bazı dillerin seçilme ve yaygınlaşma sebebi en iyi, en güçlü dil oldukları için değildir. Bir problemin çözümü için kullanılacak dilin seçiminde söz sahibi olan kişilerin programlama dilleri kavramları konusunda bilgili olmaları gerekir. İlk ortaya çıktığında, daha zarif tasarımı, çok daha iyi kontrol yapıları olduğu için ALGOL60 dilinin Fortran'ın yerini alacağı düşünülüyormuş. Ancak yazılm geliştiriciler ve onların yöneticileri ALGOL60'ın kavramsal yapısını anlayamadıkları ve dili okunabilir ve anlaşılabilir bulmadıkları için beklenen şey gerçekleşmemiş. Dilin yararlı yönlerini farkedemeyecek insanların bu seçimlerde rol oynaması bu tip sonuçlar doğurabilir.

## Programlama Alanları

### 1. Bilimsel uygulamalar

1940'ların sonu ve 50'lerin başında ortaya çıkan ilk dijital bilgisayarlar bilimsel uygulamalar için icat edilmişlerdir. O zamanki bilimsel uygulamalarda basit veri yapıları ve çok fazla gerçel sayı işlemleri kullanılıyordu. En yaygın kullanılan yapılar diziler ve matrislerdi.

İlk yüksek-seviyeli diller bu ihtiyaçları karşılamak üzere ortaya çıktı. Rakipleri assembly dili olduğu için etkinlik en önemli ölçüt oldu. İlk önce Fortran sonra ALGOL60 ve benzerleri ortaya çıkmıştır. Fortran'dan daha yaygın kullanılan başka bir dil ortaya çıkmamıştır ve günümüzde de hala kullanıldığı görülmektedir.

### 2. İş uygulamaları

Bilgisayarlar 50'lerde iş uygulamaları için kullanılmaya başlandı, özel bilgisayarlar ve programlama dilleri geliştirildi. İş uygulamaları için geliştirilen ilk başarılı yüksek-seviyeli dil COBOL'dur. Günümüzde kullanılan sürümü 60'larda ortaya çıkmıştır. Ayrıntılı raporlama, yüksek kesinlikte ondalık sayıları ve karakter verisini tanımlama ve

depolama, ondalıklı aritmetik işlemler tanımlama gibi işeri desteklemektedir.

**3. Yapay zeka**

Yapay zeka uygulamaları, nümerikten ziyade sembolik işlemlerin kullanıldığı uygulamalardır. İsimlerden oluşan semboller sayıların yerine işlem görür. Diziler yerine genelde bağlı listeler kullanılarak sembolik işlemler gerçekleştirilir. Diğer programlama dillerinden daha fazla esnekliğe ihtiyaç duyarlar. Çrneğin bazı yapay zeka uygulamalarında çalışma zamanında kod eklenmesine izin verilebilmektedir.

İlk yapay zeka uygulamaları için geliştirilen dil LISP, 1959'da ortaya çıkmıştır. 1990 öncesi geliştirilen çoğu yapay zeka uygulaması LISP veya ona çok yakın dillerde geliştirilmiştir. 70'lerin başında Prolog ile mantıksal programlama kullanılarak yazılan yapay zeka uygulamaları bulunmaktadır. Son zamanlarda C ile yazılmış bazı uygulamalar da karşımıza çıkmaktadır. Yaygın olarak Prolog ve LISP'ın bir lehçesi olan Scheme kullanılmaktadır.

**4. Sistem programlama**

İşletim sistemleri ve donanım sürücüleri gibi bilgisayar sistem destek yazılımları hepbirlikte sistem yazılımı olarak adlandırılırlar. Bu yazılımlar, neredeyse sürekli çalıştıkları için etkinlik çok önemlidir; düşük-seviye işlemleri desteklemelidirler.

60'lar ve 70'lerde PL/I, BLISS ve Extended ALGOL gibi diller kullanılmış olsa da çoğu sistem yazılımı daha genel amaçlı diller olan C ve C++ ile yazılmıştır. UNIX işletim sisteminin neredeyse tamamı C ile yazılmıştır.

C dilinin bazı özellikleri onu sistem programlama için uygun kılmaktadır. Düşük seviyeli işlemleri destekleyen, etkin işletimi olan bir dildir ve kullanıcıyı çok fazla güvenlik kısıtıyla boğmaz. Sistem programcıları C dilinin tanıdığı özgürlüklerden oldukça tatmin olsalar da sistem programcısı olmayanlar C dilinin büyük sistemlerde kullanımını tehlikeli bulmaktadır.

**5. Web yazılımı**

Web yazılımları HTML gibi işaretleme dillerinden- ki bunlar programlama dili olarak sınıflandırılmazlar- Java gibi genel amaçlı dillere kadar geniş bir yelpazede dil kullanımından oluşur. PHP veya JavaScript gibi betik dilleri ile kullanılarak HTML'nin işlem yeteneği artırılabilir.

**Dil Değerlendirme Ölçütleri**

Programlama dillerini değerlendirirken ve birbirleriyle kıyaslarken kullanılan genel ölçütler şöyledir:

1- Okunabilirlik

- Genel sadelik
- Ortogonallik
- Veri tipleri ve kontrol ifadeleri
- Sözdizimi tasarımı

2- Yazılabilirlik
- Sadelik ve Ortogonallik
- Soyutlama desteği
- Anlatım

3- Güvenilirlik
- Tip kontrolü
- İstisna işleme
- Örtüşme
- Okunabilirlik ve yazılabilirilik

4- Maliyet
- Programlama dilinin öğretimi
- Yazım maliyeti
- Derleme maliyeti
- Çalışma maliyeti
- Gerçekleştirim
- Güvenilirlik
- Bakım maliyeti

**1. Okunabilirlik**

Bir dili değerlendirmede kullanılan en önemli ölçütlerden biri, o dilde yazılan programların okunabilir ve anlaşılabilir olma özelliğidir. 1970'lerden önce yazılım geliştirme denince akla ilk gelen kod yazmaktı. 70'lerde geliştirilen "yazılım yaşam döngüsü kavramı" ile kodlamanın rolü oldukça küçülmüş ve bu döngünün ana parçası olarak "yazılımın bakımı" (maintenance) kabul edilmiştir. Yazılımın bakımı, maliyeti etkilemektedir. Bakım kolaylığı programın okunabilirliği ile doğrudan ilişkilidir. Böylece okunabilir olma, programlama dillerinin gelişiminde bir dönüm noktası olmuştur. Daha önce makineye odaklanan anlayıştan, keskin bir şekilde insan odaklı yaklaşıma geçilmiştir.

- **Genel sadelik, basitlik**

Az sayıda basit yapıya sahip olan programlama dillerini öğrenmek çok daha kolaydır. Eğer o dilde aynı işi yapmanın birden fazla yolu varsa o programlama dili karmaşıklaşır. Örneğin Java'da bir değişkenin değerini 1 artırmak için dört farklı yol kullanılabilir:

| | |
|---|---|
| count = count + 1 | count++ |
| count += 1 | ++count |

Son iki ifade kullanım durumuna göre farklı anlamlar taşıyabilse de, tekil olarak aynı sonucu vermekte, aynı anlama gelmektedirler.

Bir başka okunabilirlik sorunu da işleç aşırı yüklemesidir (operator overloading) ve bir sembolün birden fazla anlamının olması demektir. Aslında bu kullanışlı bir durum gibi gözükse de, kullanıcı işleçlere kendi aşırı yüklemesini yaptığında -ve eğer bunu çok mantıklı şekilde yapmadıysa- okunabilirlik azalmaktadır. Örneğin, + işaretinin aşırı yükleme ile iki dizinin tüm elemanlarının toplanması işini yaptığını düşünelim. Bu durum kodu okuyacak kişi

için oldukça kafa karıştırıcı olacaktır, çünkü iki dizinin toplama işlemine tabi tutulması çok farklı bir işlemdir.

- **Ortogonallik**

Programlama dillerinde ortogonallik, bir dilin kontrol ve veri yapılarını oluşturmak için, az sayıda ilkel yapının az sayıda yolla birleştirilmesidir. Bur birleşimler dilin kurallarına uygun ve anlamlıdır. Örnek olarak veri yapılarını düşünelim. Bir dilde 4 tane ilkel veri tipi olduğunu varsayalım (integer, float, double, char) ve işaretçi(pointer) ve dizi(array) işleçleri olsun. Bu iki işlecin kendilerine ve 4 tane ilkel veri tipine uygulanması ile çok sayıda veri yapısı tanımlanabilir.

Ortogonallikten yoksun bir dilde istisnai durumlar fazlalaşır. Örneğin işaretçi desteği olan bir dilde işaretçilerin tüm veri yapılarını işaret etmesi beklenir. Ancak işaretçiler diziyi işaret edemiyorsa çok sayıda veri yapısını kullanıcıların tanımlayabilmesi mümkün olmaz.

Ortogonalliğin bir tasarım kavramı olarak kullanımını, IBM mainframe ve VAX minicomputerlarda kullanılan assembly ifadelerini karşılaştırarak açıklayalım:

IBM'de

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

Burada Reg1 ve Reg2 iki tane registerı temsil etmektedir.

```
Reg1 <--- contents(Reg1) + contents(memory_cell)
Reg1 <--- contents(Reg1) + contents(Reg2)
```

VAX'da

```
ADDL   operand_1, operand_2
```

Burada operandların ne olduğu önemli değildir. Register veya bellek hücresi olabilirler.

```
operand_2 <--- contents(operand_1) + contents(operand_2)
```

VAX komut tasarımı ortogonaldir. Bir komutla hem register hem de bellek hücresi toplanabilir. IBM komut tasarımı ise ortogonal değildir. Toplama için A ve AR olmak üzere iki komut gerekmektedir ve sonuç bellek hücresine doğrudan atanamamaktadır.

Ortogonallik, basitlikle de doğrudan ilişkilidir. Ortogonallik arttıkça istisnalar azalmaktadır. Daha az istisna, dili öğrenmede kolaylık sağlar. (ingilizce, past tense, düzensiz fiiller)

Yüksek seviyeli dillerde ortogonallik eksikliğini C dilindeki bazı istisnaları ve kuralları hatırlayarak açıklayalım. C'deki iki veri yapısı olan array ve struct ın fonksiyonlardan döndürülmesi ortogonallik eksikliğine bir örnektir. Fonksiyonlar array  döndüremezler ancak struct döndürebilirler. Ayrıca, bir struct içindeki üyeler void ve aynı

tipten bir struct olamaz. Bir array, void ve fonksiyon haricindeki tüm tiplerdeki veriyi tutabilir. Diziler bir fonksiyona ancak referans olarak gönderilebilirler çünkü dizinin adı dizinin ilk elemanının adresini ifade eder.
İçerik bağımlılığı için bir örnek: a+b işleminde eğer a float tutan bir bellek bölgesini işaret ediyorsa, b değişkeni buna göre ölçeklendirilir. Burada a'nın tipi, b üzerinde nasıl bir işlem yapılacağını belirler.

Çok fazla ortogonallik de sorun oluşturabilir. Örneğin ALGOL68'de her yapının bir tipi vardır ve bu tipler konusunda kısıtlama yoktur. Bunun yanında bu yapıların çoğu bir değer de üretir. Bu durum gereksiz karmaşıklığa sebep olur. Dolayısıyla bir dilin basitliği, **az sayıda** ilkel yapının kombinasyonlarına ve ortogonalliğin belli bir sınırda kullanılmasına bağlıdır.

Fonksiyonel dillerin basitlik ve ortogonallik anlamında iyi bir denge sağladıklarına inanılmaktadır. LISP gibi bir fonksiyonel dilde temel olarak, verilen parametrelere fonksiyonların uygulanması ile hesaplamalar gerçekleştirilmektedir. Bunun tersi olarak, C, C++, Java gibi buyurgan (imperative) dillerde hesaplamalar, değişkenler ve atama ifadeleriyle gerçekleştirilir. Fonksiyonel diller genel basitlik konusunda en iyi olanlardır. Her işlemi tek yapıda (fonksiyon çağrısı) gerçekleştirirler. Etkinlik sorunu gibi sorunlardan ötürü daha geniş kullanım alanına ulaşamamışlardır.

- **Veri tipleri ve kontrol ifadeleri**

While ifadesinin olmadığı bir dilde goto kullanarak while aşağıdaki şekilde tanımlanabilir:

```
while (inc<20) {
    while (sum < 100){
        sum = sum + inc;
    }
    Inc++;
}
```

```
loop1:
    if (inc>=20) goto exit;
    loop2:
    if (sum>100) goto next;
    sum = sum + inc;
    goto loop2;
next:
    Inc++;
    Goto loop1;
exit:
```

Goto kullanılmaması için programlama dili yeterli sayıda kontrol ifadesi içermelidir. Bu, programın yukarıdan aşağı okunabilir olmasını sağlar.

Veri tipleri de yeterli ve açıklayıcı olmalıdır. Mantıksal ifadeleri belirtmek için kullanılan boolean veri tipinin olmadığı bir dilde, bir değişkene 1 ve 0 atanarak true ve false benzetimi yapılabilir. Ancak timeout=1 ifadesi açık bir ifade değildir, mantıksal bir değişken olduğu anlaşılmamaktadır.

- **Sözdizimi tasarımı**

*Özel kelimeler*

Bir dilin okunabilirliği bünyesinde kullanılan özel kelimelerle yakından ilgilidir (while, class, for,...). Birleşik ifadeleri gösteren kelimeler de önem taşımaktadır. Örneğin C'de bir grup ifade { } parantezleri arasına yazılır. Her tür döngü bloğu da aynı iki parantez kullanılarak belirtilir. Bu yüzden ayırt etmesi oldukça zordur. Fortran95 ve Ada dillerinde farklı kelimeler kullanılarak bu sorun ortadan kaldırılmıştır. If - endif, end loop

*Biçim ve anlam*

Özel kelimelerin anlamı program içinde her yerde aynı olmalı, duruma göre değişmemelidir. Örneğin C dilinde *static* kelimesi bir fonksiyon içinde kullanılırsa, o değişkenin derleme zamanında yaratılacağını gösterir. Tüm fonksiyonların dışında kullanıldığında ise o değişkenin sadece tanımlandığı dosyada kullanılabileceğini, başka dosyalar tarafından kullanılamayacağını gösterir.

**2. Yazılabilirlik**

Yazılabilirlik, bir programlama dilinin belirli bir alanda program yazmada ne kadar kolaylıkla kullanabileceğinin ölçüsüdür. Okunabilirliği etkileyen özelliklerin çoğu yazılabilirliği de etkiler. Programı yazan kişi, yazım sürecinde geri dönüp yazdıklarını tekrar tekrar okuyacağı için, program kolaylıkla okunabilmelidir. Bu noktada programlama dilinin hangi amaç için kullanıldığı da çok önemlidir. Görsel bir uygulama için C dilini seçmek yerine bu uygulamalarda yazılabilirliği daha farklı olan Visual Basic dilini kullanmak daha mantıklıdır. Aynı şekilde Sistem programlama için de C dilinin yazılabilirliği Visual Basic'ten çok farklıdır.

- **Sadelik ve Ortogonallik**

Eğer bir dil çok sayıda yapıdan oluşuyorsa, bazı özelliklerden bazı yazılımcıların haberdar olmaması doğaldır. Bu durum, dilin bazı özelliklerinin yanlış kullanılmasına, bazılarının da hiç kullanılmamasına sebep olabilir. Çok fazla sayıda temel yapının olması tercih edilen bir durum değildir.

Ortogonalliğin çok fazla olması da istenen bir özellik değildir ve yazılabilirliği olumsuz etkiler. İlkel yapıların her türlü ullanımına izn verilmesi hata ayıklamayı zorlaştırır. Derleyici tarafından algılanamayan kod yanlışlıklarına sebebiyet verebilir.

- **Soyutlama desteği**

Soyutlama, karmaşık yapıların veya işlemlerin birçok detayını gözardı edecek şekilde tanımlayabilme ve kullanabilme yeteneğidir. (karmaşıklıkların gizlenmesi)

Programlama dilleri iki tip soyutlamayı destekler: süreç ve veri soyutlama. Süreç tipi soyutlamaya en basit örnek bir sıralama algoritmasının bir altyordam olarak tanımlanmasıdır. Sıralama kullanılması gereken her yerde bu yordam çağrılarak defalarca aynı kodun yazılması önlenebilir.

Veri soyutlaması için örnek de, düğümlerinde int değerler tutan bir ikili ağaç olabilir. Böyle bir ikili ağacı gerçekleştirebilmek için Fortran77 gibi, işaretçileri ve dinamik bellek kullanımını desteklemeyen bir dilin kullanımındansa, düğümlerin 2 işaretçi ve bir int ile basit bir sınıf olarak tanımlanabildiği, C++ ve Java gibi dilleri kullanmak çok daha kolaydır.

- **Anlatım**

Bir dilin hesaplamaları tanımlamak için külfetli yollar yerine daha etkin yöntemleri desteklemesidir.

**Örn:** count = count + 1  yerine count++, while yerine for gibi.

**3. Güvenilirlik**

Eğer bir dil, tanımlamalarını her koşulda gerçekliyorsa, güvenilir olarak tabir edilir.

- **Tip kontrolü**

Derleme ya da çalışma zamanında tip hatalarını kontrol etme işlemine denir. Bu kontrolün derleme zamanında yapılması tercih edilen bir durumdur çünkü çalışma zamanında yapılması daha maliyetlidir. Programdaki hatalar ne kadar erken saptanırsa o kadar az maliyetle düzeltilebilirler. Java'da neredeyse tüm değişkenler ve ifadeler derleme zamanında kontrol edilirler.

- **İstisna işleme (Exception Handling)**

Çalışma zamanında oluşan hatalar karşısında araya girip düzeltme önlemleri alması ve çalışmaya devam etmesi bir dil için çok önemli bir özelliktir. Bu özellik, Ada, C++ ve Java'da bulunurken C ve Fortran'da bulunmamaktadır.

- **Örtüşme (Aliasing)**

Kabaca, aynı bellek hücresine erişim için iki veya daha fazla farklı ad kullanılmasıdır.Bir programlama dili için tehlikeli bir özellik olarak kabul edilir. Çoğu dilde kullanım örneklerine rastlayabiliriz. Örneğin aynı değişkeni işaret eden birden fazla işaretçi kullanımı durumunda yazılımcı değişken değer takibini iyi yapmalıdır.

- **Okunabilirlik ve Yazılabilirlik**

Gereken algoritmayı doğal yollarla ifade edemeyen bir dilde, doğal olmayan yollarla bazı yapılar simüle edilecektir. Doğal olmayan yaklaşımlar da güvenilirliği zedeler. Programı yazmak ne kadar kolaysa, yazılan kodun doğru olma olasılığı da o kadar yüksektir.

**4. Maliyet**

Bir programlama dilinin toplam maliyetini etkileyen birçok faktör vardır. Bunlardan ilki **programlama dilinin öğretimidir**, ve bu maliyetin büyüklüğü, dilin ortogonalliği ve kolaylığı ile yazılımcının

deneyimine bağlıdır. Bir dilin çok güçlü olması, o dili öğrenmenin zor olacağı anlamına gelmez ama genellikle de öyledir.

İkincisi **program yazma maliyetidir**. Dilin belirli bir uygulama için kullanılmaya ne kadar uygun olduğu ile de kısmi olarak ilgili olan dilin yazılabilirliğine bağlı bir maliyettir. Yüksek seviyeli dillerin ortaya çıkmasındaki temel düşünce bir yazılım üretmenin maliyetini düşürmektir. İyi bir yazılım geliştirme ortamında hem programcı eğitimi hem de program yazma maliyetleri büyük ölçüde azaltılabilir.

Üçüncü faktör **derleme maliyetidir**. Gelişmiş derleyici tasarımları ile bu maliyet azaltılabilir.

Bir programlama dilinin tasarımı, o dilde yazılmış programların **çalışma maliyeti** üzerinde büyük etkiye sahiptir. Çalışma zamanında birçok tip kontrolü gerektiren bir dil, derleyicinin kalitesinden bağımsız olarak, programın hızlı çalışmasına engel olacaktır. Her ne kadar çalışma etkinliğinin önemi ilk ortaya çıkan dillerde ilk sırada geliyor da olsa, günümüzde daha az önemli olarak düşünülmektedir.

Maliyeti etkileyen bir diğer faktör de dilin **gerçekleştirim** sistemidir. Java dilinin çok çabuk kabul görmesinin nedenlerinden biri özgür derleyici/yorumlayıcı sistemine sahip olmasıdır. Gerçekleştirimi maliyetli olan veya sadece pahalı sistemler üzerinde çalışabilen sistemler için tasarlanmış dillerin genel kabul görmesi düşük bir olasılığa sahiptir.

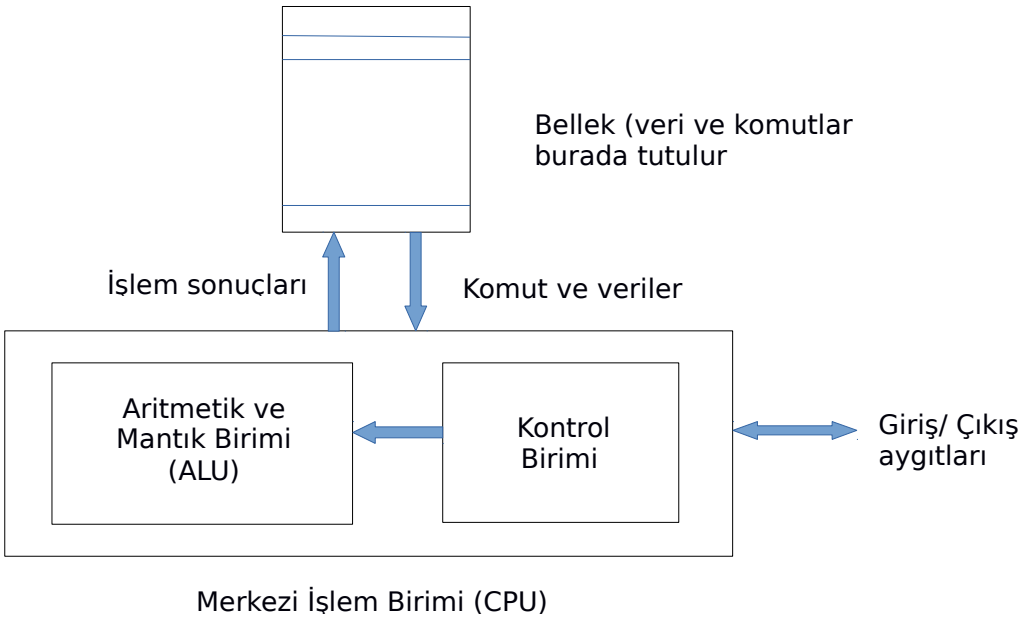
Düşük **güvenilirlik** de maliyeti etkileyen bir unsurdur. Hayati önem taşıyan bir sistemde yazılımın çökmesinin maliyeti oldukça yüksek olabilir. Hayati önem taşımayan sistemlerde de güvenilir olmayan yazılımlar, gelecek işlerin veya davaların kaybedilmesine sebep olabilir.

Son olarak, programların **bakım maliyetlerine** değinebiliriz. Bakım maliyeti hem düzeltmeleri hem de programa yapılacak yeni eklemeleri içermektedir. Dilin okunabilirliği bu maliyeti yüksek oranda etkiler. Yazılım bakımları genellikle programı yazan kişi tarafından değil de başka bir yazılımcı tarafından yapılmaktadır. Dolayısıyla kodun rahatlıkla okunup anlaşılabilmesi çok önemlidir. Uzun süre kullanılan geniş kapsamlı yazılımların bakım maliyetleri, geliştirim maliyetinin 2-4 katına çıkabilmektedir.

## Dil tasarımını etkileyen unsurlar

### 1. Bilgisayar Mimarisi

Son 50 yıldır, popüler programlama dillerinin çoğu **von Neumann** mimarisi adı verilen yaygın olarak kullanılan bilgisayar mimarisini temel alarak geliştirilmiştir. Bu diller buyurgan (imperative) diller olarak adlandırılırlar. Von Neumann mimarisinde veri ve komutlar aynı bellekte tutulur. Komutları işleyen merkezi işlem birimi (CPU), bellekten ayrıdır. Dolayısıyla önce veri ve komutlar işlem görmek için işlemciye aktarılmalıdır ve işlemlerin sonucu da işlemciden belleğe aktarılmalıdır (Şekil 1). 1940'lardan beri üretilen bilgisayarların tümü bu mimariyi kullanır.




**Şekil 1.** von Neumann mimarisi

Bu mimariden kaynaklı olarak, buyurgan dillerin temel unsuru değişkenlerdir. Bir atama işleminin sağ tarafı işlemcide gerçekleştirilir ve eşitliğin solundaki değer de belleğe gönderilir. Bu mimaride iterasyon hızlıdır. Çünkü komutlar bellekte ardışık hücrelerde tutulurlar ve bir grup kodun tekrarı için yalnızca bir dallanma komutu (branch instruction) yeterli olmaktadır.

Makine kodunun von Neumann mimarisinde çalışması getir-çalıştır döngüsü (fetch-execute cycle) olarak adlandırılır. Çalıştırılacak her komutun bellekten işlemciye getirilmesi gerekir. Bir sonra işlenecek komutun bellek adresi, program sayacı (program counter) adlı yazmaçta (register) tutulur.
Getir-çalıştır döngüsü şu algoritmayla basitçe anlatılabilir:

*Program sayacını ilkleme*
*Sonsuza kadar tekrarla*
    *Program sayacının gösterdiği adresteki komutu getir*
    *Program sayacını bir sonraki komutu işaret edecek şekilde artır*
    *Komutu çözümle*
    *Komutu çalıştır*
*tekrar sonu*

Fonksiyonel dillerin bu mimari yapısı ile yaygın kullanılması ve buyurgan dillerin yerini alması pek mümkün gözükmemektedir. Çünkü fonksiyonel dillerde buyurgan dillerdeki veri tipleri, atama işlemleri ve iterasyonlar olmadan programlama yapılabilmektedir. Bu mimari, buyurgan dillerin etkin çalışması için daha uygundur.

## 2. Programlama Metodolojileri

70’lerin sonunda, altyordama dayalı program tasarımından veriye dayalı tasarım metodolojisine geçiş başlamıştır. Veriye dayalı sistemler, en basit anlatımıyla, veri tasarımına vurgu yapan soyut veri tiplerinin kullanımına odaklanan sistemlerdir. Veri soyutlamanın etkin bir şekilde uygulanabilmesi soyutlamayı destekleyen dillerin kullanılması ile mümkündür. 1970’lerden sonra geliştirilen çoğu programlama dilinde soyutlama desteği bulunmaktadır.

Veri tabanlı yazılım geliştirmenin son adımı da 1980’lerde ortaya çıkan nesneye dayalı tasarım olmuştur. Nesneye dayalı metodoloji veri soyutlaması ile başlar, veriye erişimin metodlar aracılığıyla kontrollü olarak yapılması, kalıtım ve dinamik metod bağlamanın eklenmesi ile gelişir. Kodun tekrar kullanılmasını sağlandığından, kalıtım bu metodoloji tarafından benimsenmiştir. İlk nesneye dayalı programlama dili Smalltalk, yaygın olarak kullanılmamıştır.

### Dil Sınıfları

Programlama dillerini dört sınıfa ayırmak mümkündür: buyurgan, fonksiyonel, mantıksal ve nesneye dayalı diller.

Aslında nesneye dayalı programlama desteği olan dilleri ayrı bir sınıfa ayırmak gerekmeyebilir. En popüler nesneye dayalı diller buyurgan dillerden gelişmiştir. Nesneye dayalı programlama paradigması alt yordama dayalı programlama paradigmasından çok farklı olsa da , nesne desteği için buyurgan bir dile getirilen eklentiler çok da fazla değildir. Örneğin C ve Java’daki kontrol ve atama yapıları neredeyse aynıdır. Bu bakış açısıyla fonksiyonel dilleri de nesneye dayalı programlama destekleyen dillerin bir uzantısı olarak görebiliriz.

Görsel diller de buyurgan dillerin alt kategorisidir. En popülerleri .NET dilleridir. Scripting (betik) dillerini ayrı bir kategoride tanımlayanlar olsa da Perl, JavaScript ve Ruby gibi diller her anlamda buyurgandır. Mantıksal programlama dilleri kural tabanlı dillerdir. Buyurgan dillerde olduğu gibi komutların sıralı bir şekilde yazılması gerekmemektedir. Kuralların tanımlanma sırası yoktur. Örn: Prolog.

### Dil tasarımında fayda-zarar dengesi

Dil değerlendirme ölçütlerinde çatışan iki kriter güvenilirlik ve çalışma maliyetidir. Java’da tüm dizi indisleri doğru sınırlar içinde mi diye kontrol edilmektedir. Eğer diziler üzerinde çok işlem yapılan bir kodumuz varsa bu kontrol çok fazla çalışma maliyeti getirir. C’de dizi sınırları kontrol edilmez ve mantıksal olarak aynı olan Java kodundan daha hızlı çalışır ancak Java’nın güvenilirliği daha yüksektir.

Güvenilirlik ve yazılabilirlik arasındaki çatışma da dil tasarımında çok sık karşılaşılan bir durumdur. C++’ta bulunan ve veri adreslemede esneklik yaratan işaretçiler Java’da güvenlik sebebiyle kullanılmamıştır.

## Gerçekleştirim Yöntemleri

Makine dili bir komutlar bütünüdür. Herhangi başka bir destekleyici yazılımın olmadığı durumlarda donanımın anlayabileceği tek dildir. Makine dili olarak yüksek seviyeli bir dil kullanılsaydı bu çok yüksek maliyetli ve esnek olmayan bir çözüm olurdu. İşletim sistemi ve diğer yazılımlar, makine dili üzerinde ayrı bir katman gibi düşünülebilir.

Programlama dilleri 3 genel metottan herhangi biri ile gerçekleştirilebilir.

### 1. Derleme

Bu yöntemde yazılan programlar makine diline dönüştürülürler ve makinelerde doğrudan çalıştırılabilirler. Bu yöntemin adı ***derleyici gerçekleştirimi***dir. Bu yöntemin en büyük avantajı, programın bir defa makine diline dönüşümü yapıldıktan sonra çok hızlı çalışmasıdır. C, C++, COBOL ve Ada dilleri derleyici gerçekleştirimi kullanırlar. Derleyicinin dönüştürdüğü dile ***kaynak dil*** (source language) denir. Derleme ve çalıştırma süreçleri birçok adımda gerçekleştirilir ve adımlar Şekil2'de gösterilmiştir.

**Sözcüksel analizci**(lexical analyzer) kaynak programdaki karakterleri **sözcüksel birimlere**(lexical units) dönüştürür. Sözcüksel birim dediklerimiz, tanımlayıcılar, özel kelimeler, işleçler ve noktalama işaretleridir. Program içinde yazılan komut satırları sözcüksel analizci tarafından yok sayılır çünkü bu satırlar derleyici için bir anlam ifade etmez.

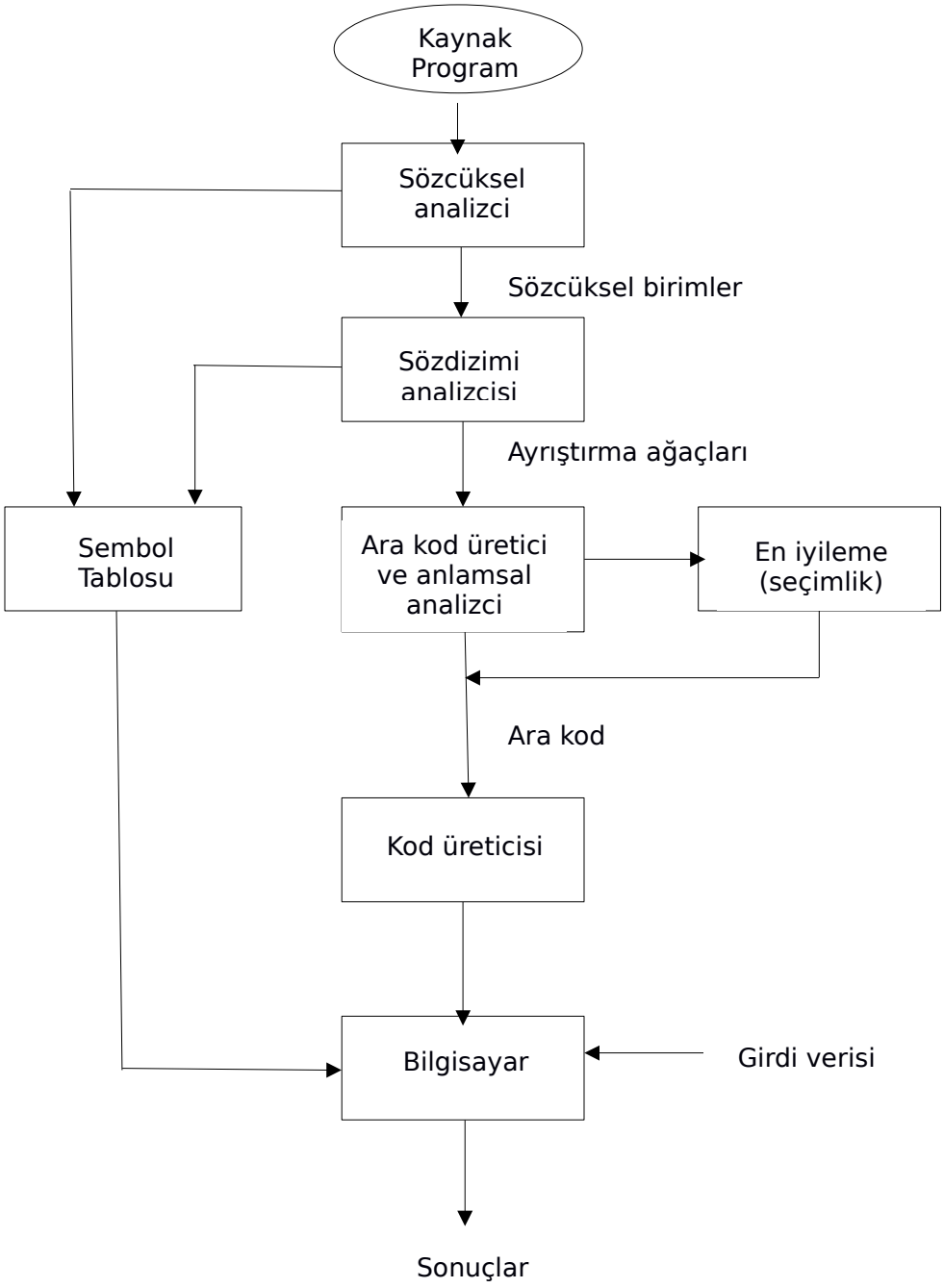
**Sözdizimi analizcisi** (syntax analyzer), sözcüksel analizciden aldığı sözcüksel birimleri ayrıştırma ağacı (parse tree) adı verilen hiyerarşik yapıları oluşturmak için kullanır. Ayrıştırma ağaçları programın sözdizimsel (syntactic) yapısını gösterir.

**Ara kod üreticisi** (intermediate code generator), kaynak dil ve makine dili arasında başka dilde bir program üretir. Ara kodlar bazen assembly diline çok benzerler, bazen de assembly kodudurlar. Diğer durumlarda da assembly kodundan bir seviye daha yukarıdadırlar. **Anlamsal analizci** (semantic analyzer), ara kod üreticisine eklenmiş bir yapıdır. Sözdizimi analizi esnasında tespiti mümkün olmayan tip hataları gibi hatalar anlamsal analizcide yakalanır.

**En iyileme**(optimization), derleme işleminin seçimlik kısmıdır. Programları, genellikle de ara kodu daha küçük ve/veya daha hızlı yaparak iyileştirir. Gerçekte bazı derleyiciler kayda değer bir iyileştirme yapamazlar. Bu tipş derleyiciler programın hızlı çalışmasının çok da önemli olmadığı durumlara uygundur. Makine dili üzerinde en iyileme yapmak zor olduğundan, çoğu en iyileme ara kod üzerinde yapılır.

**Kod üretici** (code generator), en iyilenmiş ara kodu makine koduna çevirir.

**Sembol tablosu** (symbol table), derleme işlemi için bir veritabanı işlevi görür. Sembol tablosunun ana içeriği kullanıcının tanımladığı her ismin tipi ve niteliğidir. Bu tablodaki bilgiler sözdizimi ve sözcüksel analizciler tarafından yerleştirilir ve anlamsal analizci ve kod üreticisi tarafından kullanılır.




**Şekil 2.** Derleme süreci

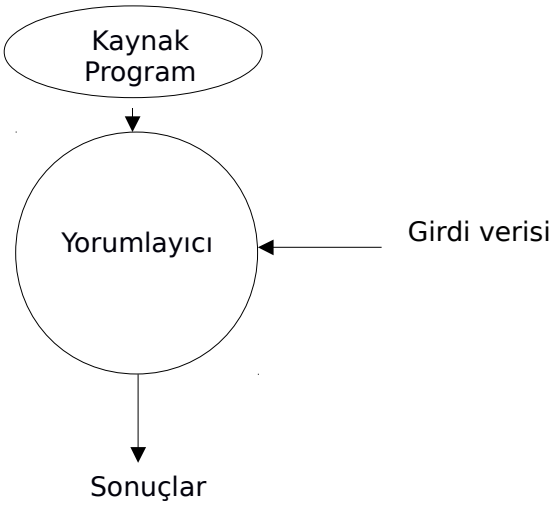
Üretilen makine kodu, donanım üzerinde doğrudan çalıştırılabilir olsa da yanında başka kodların çalıştırılmasına ihtiyaç duyar. Çoğu kullanıcı programı işletim sisteminden gelen bazı kodlara gereksinim duyar. Derleyici, gerektiğinde sistem programlarına çağrı yapar. Derleyici tarafından üretilen makine kodu çalıştırılmadan önce işletim sisteminin sağlayacağı programlar bulunmalı ve kullanıcı programına bağlanmalıdır. Kullanıcı programları ile sistem

programlarının bağlanması işine **linking** işlemi denir. Kullanıcı kodu ve sistem kodu birlikte **yükleme modülü** (load module) veya **çalıştırılabilir görüntü** (executable image) adını alır. Linking işlemi **linker** denilen bir sistem programı tarafından gerçekleştirilir. Linker kullanıcı programlarını sistem programları ile bağlamanın yanında, kütüphanelerdeki başka kullanıcı programları ile de bağlar.

Bilgisayarın hızını bellek ve işlemci araasındaki veri yolunun hızı belirler. Von Neumann mimarisindeki bilgisayarlar için hız limitinin ana sebebi von Neumann darboğazı olarak adlandırılır. Paralel bilgisayarların ortaya çıkışındaki ana motivasyon da von Neumann darboğazıdır.

## 2. Saf Yorumlama

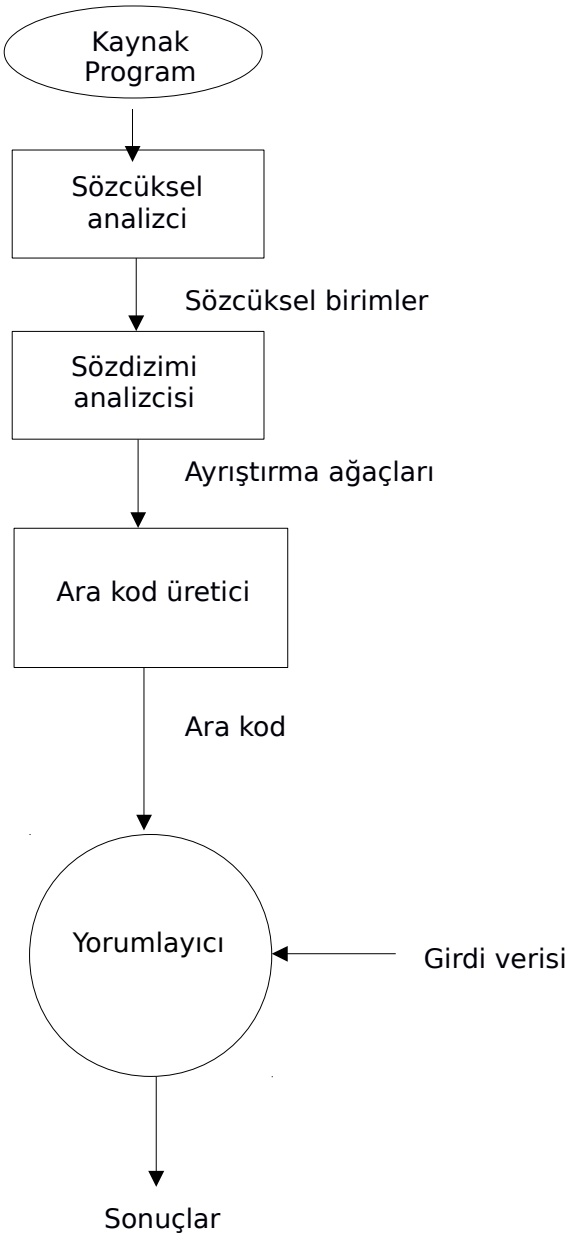
Bu yaklaşımda programlar yorumlayıcı (interpreter) adı verilen başka bir programda yorumlanırlar. Kod çevrimi gibi bir işlem olmaz. Yorumlayıcı programlar ile makine yazılımla simüle edilir, getir-çalıştır döngüsü makine komutlarını değil, yüksek seviyeli bir dilde yazılmış programın yapılarını kullanır. Dil için sanal bir makine yaratılmış olur. Saf yorumlayıcının avantajı hata ayıklamayı kod düzeyinde yapabilmesidir. Örneğin dizi indisi hatalarının hangi dizide, kodun hangi satırında yapıldığını yazılımcıya bildirir. Diğer taraftan derleyici sistemlere göre 10-100 kat yavaşlığı ile çok ciddi dezavantaj sahibidir. Diğer dezavantajı da daha fazla alana ihtiyaç duymasıdır. APL, SNOBOL, LISP, JavaScript ve PHP bu metodolojiyle geliştirilmiştir.




**Şekil 3.** Saf Yorumlama

## 3. Hibrid Gerçekleştirim Sistemleri

Kolay yorumlama yapabilmek için yüksek seviyeli dilin ara dile çevrilmesi prensibine dayanırlar. Bu yöntem saf yorumlamadan daha hızlıdır çünkü kaynak kod yapıları sadece bir kere çözümlenir. Ara kodun makine koduna çevrilmesi yerine ara kod yorumlanır (Şekil 4). Perl, Java (ara koduna byte code denir). Just in Time (JIT) sistemlerde ara kod üretilir, çalışma zamanında ara dilin metodları çağrıldıkça makine koduna çevrilir. Java ve .NET




**Şekil 4.** Hibrid Gerçekleştirim Sistemleri