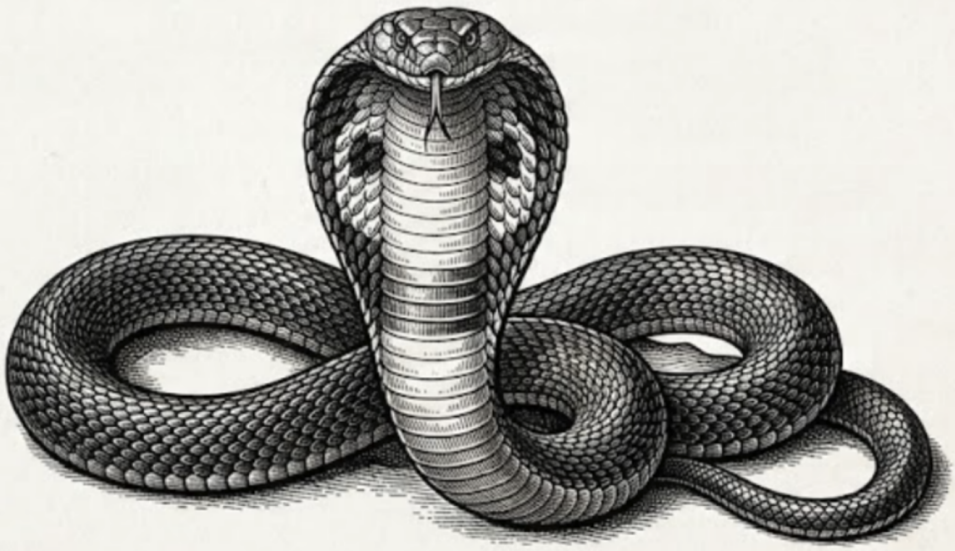
Pratik ve Hızlı Yazılım Geliştirme

Cobra

Programlama Dili

Basit sözdizimi, gerçek paralellik ve iki çalışma motoru



Baris AKIN © 2026

İçindekiler

Önsöz

Programlama dilleri çoğu zaman iki uçtan birine savrulur: ya yazması kolaydır ama yavaştır, ya da hızlıdır ama söz dizimi insanın gözünü yorar. **Cobra** bu ikilemi reddetmek için tasarlandı. Anlamlı girintiye (significant indentation) zorlamayan, blokları `end` ile kapatan sade bir söz dizimi sunar; buna karşılık arka planda iki ayrı çalışma motoruyla — bir ağaç-yürüten yorumlayıcı ve bir bayt-kodu sanal makinesi — ciddi bir performans sağlar. Üstelik pek çok dilin aksine bir **GIL** (Global Interpreter Lock) taşımaz: görevler birden çok CPU çekirdeğinde gerçekten paralel koşar.

Bu kitap, Cobra'yı sıfırdan, üretim düzeyinde öğretmek için yazıldı.

0.1 Bu kitap kimin için?

Bu kitabın okuru, en az bir programlama dilini temel düzeyde bilen bir geliştiricidir. Değişken, döngü ve fonksiyon gibi kavramların ne olduğunu bildiğinizi varsayıyoruz; ama Cobra'ya özgü hiçbir şeyi bildiğinizi varsaymıyoruz. Dili tanımıyor olmanız sorun değil — her şeyi baştan kuruyoruz.

0.2 Bu kitap nasıl düzenlendi?

Kitap iki yarımdan oluşur. İlk on bir bölüm dili **kullanmayı** öğretir: değişkenlerden ve tiplerden başlayıp koleksiyonlar, fonksiyonlar, nesne yönelimli programlama, hata yönetimi, modüller, eşzamanlılık ve standart kütüphaneye kadar ilerler. Son bölüm ise perdeyi aralayıp dili **kuranların** dünyasına girer: kaynağı tokenlara çeviren sözlük çözümleyiciyi (lexer), tokenları bir sözdizim ağacına dönüştüren ayrıştırıcıyı (parser) ve bu ağacı çalıştıran iki motoru anlatır.

Bölümler birbirinin üstüne kurulur; baştan sona okumanız önerilir. Yine de belirli bir konuya hızlıca bakmak isterseniz her bölüm kendi başına da anlaşılır.

0.3 Bu kitapta kullanılan kurallar

Metin boyunca O'Reilly geleneğindeki kenar notlarını kullanıyoruz:

> **Not** Konuyu tamamlayan, akılda tutulması yararlı bir ayrıntı.
>
> **İpucu** İşinizi kolaylaştıracak pratik bir öneri.
>
> **Uyarı** Sizi bir tuzağa veya yaygın bir hataya karşı uyaran kritik bir uyarı.

Kod örnekleri tek aralıklı (monospace) bloklar içinde verilir. Komut satırı örneklerinde `$` istem (prompt) işaretidir ve onu yazmazsınız:

```
$ cobra merhaba.cobra
```

Cobra kaynak kodu ise dilin kendi söz dizimiyle gösterilir:

```
let mesaj = "Merhaba, Cobra"
print(mesaj)
```

0.4 Kodu çalıştırmak

Tüm örnekler iki biçimde çalıştırılabilir:

```
$ cobra dosya.cobra          # ağaç-yürüten yorumlayıcı (varsayılan)
$ cobra --vm dosya.cobra     # bayt-kodu sanal makinesi (daha hızlı)
$ cobra                      # etkileşimli REPL
```

Bu kitabın temel aldığı sürüm **Cobra 0.10.0**'dır. İki motor da her örnek için birebir aynı çıktıyı üretecek biçimde tasarlanmıştır; bir fark görürseniz bu bir hatadır.

0.5 Teşekkür

Cobra; sade bir dilin de güçlü olabileceğine inanan herkese adanmıştır. Şimdi ilk satırımızı yazalım.

— Baris AKIN

Bölüm 1

Cobra'ya Giriş

Her programlama dili bir bahse girer. Bazıları hızın, bazıları güvenliğin, bazıları da soyutlama gücünün her şeyden önce geldiğine inanır. Cobra'nın bahsi ise sadeliktir: bir betik dilinin kodu okumayı ve yazmayı kolaylaştıran tarafıyla, onu gerçek işlere koşabilecek kadar güçlü kılan tarafını aynı anda elde tutabileceği. Bu bölümde Cobra'nın ne olduğuna, hangi tasarım kararlarının onu diğer dillerden ayırdığına ve daha ilk satırınızı yazmadan önce bilmeniz gereken birkaç temel kavrama bakacağız. Sonunda kendi "Merhaba, Cobra" programınızı iki ayrı motor üzerinde çalıştırmış, hatta dilin iç dünyasına bir göz atmış olacaksınız.

Kitabın geri kalanı bu bölümde atılan temelin üzerine inşa edilir. Bu yüzden buradaki örnekleri sadece okumakla yetinmeyin; bir terminal açın ve hepsini kendiniz çalıştırın. Cobra'yı öğrenmenin en hızlı yolu, onunla konuşmaktır.

1.1 Cobra Nedir?

Cobra, sade ve okunması kolay söz dizimine sahip bir betik (scripting) dilidir. Söz dizimi tanıdıktır ama hiçbir dilin birebir kopyası değildir: blokları girintiyle değil, açıkça `end` anahtar kelimesiyle kapatır; mantıksal işlemleri sembollerle değil, `and`, `or`, `not` kelimeleriyle ifade eder; ve modern bir dilden beklediğiniz pek çok şeyi (kapanışlar, yapılar, hata yönetimi, eşzamanlılık, zengin bir standart kütüphane) kutudan çıktığı haliyle sunar.

Cobra'yı asıl ilginç kılan iki özelliği vardır. Birincisi, dilin **iki ayrı çalışma motoru** üzerinde koşmasıdır. İkincisi, eşzamanlılığın gerçek anlamda paralel olmasıdır; yani **GIL içermez**. Bu ikisine birazdan ayrıntılı bakacağız.

Önce tasarım felsefesini anlamak gerekir, çünkü Cobra ile çalışırken vereceğiniz her kararın arkasında bu felsefe yatar.

1.2 Tasarım Felsefesi

Cobra birkaç net ilke etrafında tasarlanmıştır. Bunları baştan kavramak, dilin neden bazı şeyleri yaptığını ve bazı şeyleri kasıtlı olarak yapmadığını anlamanızı sağlar.

1.2.1 Sade Söz Dizimi

Cobra'nın birincil amacı sade söz dizimidir. Dil, yeni operatörler ve sembollerle dolup taşmak yerine, az sayıda iyi seçilmiş yapıya dayanır. Mantıksal operatörlerin sembol değil kelime olması bunun küçük ama anlamlı bir örneğidir:

```
let x = 7
if x > 0 and x < 10
    print("tek haneli pozitif")   # tek haneli pozitif
end
```

Burada `&&` yerine `and`, `||` yerine `or`, `!` yerine `not` yazarsınız. Kod İngilizce bir cümle gibi okunur. Bu, sadeliğin Cobra'da yalnızca bir slogan değil, somut bir tasarım tercihi olduğunu gösterir.

> **Not:** Cobra'da `else if` yoktur; bunun yerine tek kelime olan `elif` kullanılır. Mantıksal değerler küçük harfle yazılır: `true` ve `false`. "Hiçbir şey" değeri ise `null`'dur — `None`, `nil` veya `undefined` değil.

1.2.2 Girinti Değil, `end` ile Kapanan Bloklar

Pek çok modern dil blokları ya süslü parantezlerle (`{ }`) ya da anlamlı girintiyle belirler. Cobra üçüncü bir yolu seçer: her blok, onu açan anahtar kelimeyle başlar (`if`, `while`, `for`, `def`, `struct`…) ve `end` ile kapanır.

```
def selamla(isim)
    if isim == ""
        print("Merhaba, yabancı")
    else
        print("Merhaba, " + isim)
    end
end

selamla("Ada")   # Merhaba, Ada
```

Bunun pratik bir sonucu vardır: **girinti tamamen kozmetiktir.** Cobra, kodunuzu kaç boşlukla girintilediğinize aldırmaz; bloğun nerede bittiğini boşluktan değil, `end` kelimesinden anlar. Yine de okunabilirlik için kodunuzu düzgün girintilemeniz beklenir; sadece bunu yapmak *zorunda* olmadığınızı bilin.

> **İpucu:** Girintinin anlam taşıdığı dillerden geliyorsanız, "yanlış girinti yüzünden kodum çalışmıyor" sorununu Cobra'da hiç yaşamazsınız. Blok sınırlarını gözünüzle aramak yerine `end` kelimesini takip edin.

1.2.3 Tek Anlambilim, İki Motor

İşte Cobra'nın en sıra dışı yanı. Aynı dil iki farklı şekilde çalıştırılabilir:

1. **Ağaç-yürüten yorumlayıcı** (tree-walking interpreter): Kaynak kodu bir soyut söz dizimi ağacına (AST) çevirir ve bu ağacın üzerinde gezinerek programı doğrudan çalıştırır. Bu, varsayılan motordur.
2. **Bayt-kodu sanal makinesi** (bytecode VM): Kaynak kodu önce derleyip bayt koduna çevirir, sonra bu bayt kodunu yığın tabanlı bir sanal makinede çalıştırır. Bu motor, ağaç-yürütücüye göre kabaca **4 kat daha hızlıdır**.

Bu iki motor aynı anlambilimi (semantics) paylaşır. Yani bir Cobra programı hangi motorda çalışırsa çalışsın **birebir aynı çıktıyı** üretir. Bu yalnızca bir vaat değil; dilin test takımı bu eşitliği sürekli doğrular. Pratikte bu, geliştirme sırasında esnek ağaç-yürütücüyü kullanabileceğiniz, sonra tek bir bayrak ekleyerek aynı programı hızlı VM üzerinde koşturabileceğiniz anlamına gelir — davranışın değişeceğinden endişe etmeden.

> **Not:** İki motor arasında tek bir gözle görülür fark vardır: VM, tanımsız değişkenleri *derleme anında* yakalar (program o satıra gelmeden); ağaç-yürütücü ise o satır çalıştığında yakalar. Çalışma anı hata mesajları, satır numaraları dahil, iki motorda da aynıdır.

1.2.4 GIL İçermeyen Gerçek Paralellik

Pek çok betik dili, eşzamanlılığı bir "Global Yorumlayıcı Kilidi" (Global Interpreter Lock, GIL) ardında sunar. GIL, aynı anda yalnızca tek bir kodun çalışmasına izin verdiği için CPU-yoğun işler birden çok çekirdekten gerçekten faydalanamaz.

Cobra'da **GIL yoktur.** Bir `async def` çağrısı anında bir görev (task) döndürür ve bu görevler CPU çekirdekleri üzerinde gerçek anlamda paralel koşar. Bu, CPU-yoğun işlerin gerçekten hızlandığı anlamına gelir.

```
import "time"

async def is_yap(n)
    time.sleep(0.05)
    return n * n
end

let sonuclar = await [is_yap(1), is_yap(2), is_yap(3)]
print(sonuclar)   # [1, 4, 9]
```

Bu güç bir sorumluluk getirir: görevler gerçekten paralel çalıştığı için, paylaşılan değişebilir durumu açıkça eşzamanlamanız gerekir (`Mutex`, `Channel` gibi araçlarla). Eşzamanlılık konusunu kitabın ilerleyen bölümlerinde derinlemesine ele alacağız; şimdilik bilmeniz gereken tek şey, Cobra'nın bu konuda taviz vermediğidir.

1.2.5 Az Ama Yeterli

Cobra, başka dillerde alışık olduğunuz pek çok özelliği kasıtlı olarak içermez. f-string'ler, liste kavramaları (comprehensions), demetler (tuples), çoklu atama, `lambda`, `match`/`switch`, kümeler (sets) ve operatör aşırı yüklemesi bunlardan birkaçıdır. Bu eksiklikler bir kusur değil, sadelik felsefesinin bir sonucudur: her özellik dile bir karmaşıklık maliyeti getirir, ve Cobra bu maliyeti ancak gerçekten gerekli olduğunda öder.

> **Uyarı:** Başka bir dilden gelen alışkanlıkla `a, b = 1, 2` (çoklu atama) ya da `"x={x}"` (f-string) yazarsanız hata alırsınız. Cobra'da değer biçimlendirme `"x=" + str(x)` şeklinde, `+` ve `str(...)` ile yapılır. Bu kısıtlamaların listesini kitap boyunca yeri geldikçe hatırlatacağız.

1.3 Kodu Çalıştırma Yolları

Cobra'yı çalıştırmanın birkaç yolu vardır. Bu kitapta, kaynaktan çalıştırmak için `cobra` kullanacağız (derlenmiş bir `cobra` ikili dosyanız varsa, `cobra` yerine doğrudan `cobra` yazabilirsiniz). Hadi her bir yolu görelim.

1.3.1 Ağaç-Yürütücü ile Çalıştırma

En basit yol, bir `.cobra` dosyasını doğrudan vermektir. Bu, varsayılan ağaç-yürüten motoru kullanır:

```
cobra dosya.cobra
```

1.3.2 Bayt-Kodu VM ile Çalıştırma

`--vm` bayrağını eklediğinizde aynı program bayt-kodu sanal makinesinde koşar. Çıktı aynı kalır, ama yürütme yaklaşık 4 kat hızlanır:

```
cobra --vm dosya.cobra
```

CPU-yoğun bir programınız varsa `--vm` neredeyse her zaman daha iyi bir seçimdir.

1.3.3 Derleyip Çalıştırma

Bir programı önceden derleyip bayt-kodu dosyasına çevirebilirsiniz. `build` komutu `dosya.cobra` dosyasından `dosya.cobrac` üretir:

```
cobra build dosya.cobra      # dosya.cobrac üretir
cobra dosya.cobrac           # derlenmiş dosyayı çalıştırır
```

Derlenmiş bir `.cobrac` dosyasını çalıştırmak, ayrıştırma ve derleme adımlarını atladığı için başlangıçta biraz daha hızlıdır. Programı dağıtmak istediğinizde de işe yarar.

1.3.4 Yerel Makine Koduna Derleme

Programı çalıştırmanın üçüncü bir yolu daha var ve bu yol yorumlamayı büsbütün geride bırakır. `--native` ile Cobra, programınızın tiplerini çıkarsar ve onu **makine koduna** derler; artık araya bir yorumlayıcı girmeden programınızı doğrudan işlemci yürütür:

```
cobra --native dosya.cobra          # makine koduna derle, sonra çalıştır
cobra build --native dosya.cobra    # bağımsız çalıştırılabilir dosya üret
```

Üretilen çalıştırılabilir dosyanın içinde ne VM ne de Cobra çalışma zamanı vardır; çalışmak için kurulu hiçbir şeye ihtiyaç duymaz. Sayısal işlerde her iki yorumlayıcıdan da çarpıcı biçimde hızlıdır — ve birebir aynı çıktıyı üretir; proje bunu bayt bayt doğrular.

Yerel derleme, dilin tip-kararlı çekirdeğini destekler (fonksiyonlar, struct'lar, listeler, sözlükler, metinler, döngüler, `math` modülü). Closure, `async`, `try/catch` veya kalıtım kullanan bir program *yerel olarak derlenemez* diye bildirilir ve VM'de çalışır. Bunun nasıl işlediğini — ve dinamik tiplerin işi neden zorlaştırdığını — Bölüm 12'de göreceğiz.

1.3.5 Etkileşimli REPL

Argümansız `cobra` çalıştırdığınızda etkileşimli bir REPL (Read-Eval-Print Loop) açılır. Burada her satırı yazıp anında sonucunu görebilirsiniz — dili keşfetmenin ve küçük şeyleri denemenin en hızlı yolu budur:

```
cobra          # ağaç-yürütücü REPL
cobra --vm     # aynı REPL, bayt-kodu VM üzerinde
```

REPL, satırlar arasında durumu korur; yani bir satırda tanımladığınız değişkeni bir sonraki satırda kullanabilirsiniz:

```
> let x = 10
> let y = 20
> print(x + y)
30
```

REPL aynı zamanda **çok satırlı girişi** de anlar. Henüz kapanmamış bir blok (`def`, `if`, `while`, `for`), açık bir parantez veya köşeli ayraç yazarsanız, REPL girişin tamamlanmadığını fark eder ve bir `..` devam istemiyle (continuation prompt) sizden geri kalanı bekler:

```
> def kare(n)
..     return n * n
.. end
> print(kare(9))
81
```

Blok `end` ile kapandığında REPL girişi bütün olarak değerlendirir. Bu, REPL içinde bile fonksiyonlar, döngüler ve yapılar yazabilmenizi sağlar.

> **İpucu:** Bir özelliğin nasıl davrandığından emin değilseniz, dosya açıp dökümantasyona dalmadan önce REPL'de hızlıca deneyin. Cobra'yı öğrenirken en iyi arkadaşınız bu olacak.

1.4 İlk Programınız: Merhaba, Cobra

Artık teoriyi bir kenara bırakıp kod yazmanın vakti geldi. Sevdiğiniz metin düzenleyiciyle `merhaba.cobra` adında bir dosya oluşturun ve içine şunu yazın:

```
print("Merhaba, Cobra")
```

Şimdi çalıştırın:

```
cobra merhaba.cobra
```

Terminalde şunu görmelisiniz:

```
Merhaba, Cobra
```

Tebrikler, ilk Cobra programınızı yazdınız. Aynı dosyayı bayt-kodu VM üzerinde de çalıştırabilirsiniz; çıktı tıpatıp aynı olacaktır:

```
cobra --vm merhaba.cobra
# Merhaba, Cobra
```

`print` Cobra'nın yerleşik (built-in) bir fonksiyonudur. Birden fazla argüman alabilir ve onları aralarına boşluk koyarak yazdırır:

```
print("toplam:", 2 + 3)   # toplam: 5
print("a", "b", "c")      # a b c
```

`print`, sayıları otomatik olarak metne çevirir. Ama bir sayıyı başka bir metinle `+` ile birleştirmek isterseniz, önce `str(...)` ile açıkça dönüştürmeniz gerekir (Cobra'da f-string olmadığını hatırlayın):

```
let yas = 36
print("yaş " + str(yas))   # yaş 36
```

1.5 Yorumlar

Cobra'da yorumlar # işaretiyle başlar ve satırın sonuna kadar sürer. Yorumlar yorumlayıcı tarafından tamamen yok sayılır; kodunuzu açıklamak için vardır:

```
# Bu bir tam satır yorumudur
let pi = 3.14159   # bu da satır sonu yorumudur

# Aşağıdaki satır çalışmaz çünkü yorum içinde:
# print("bu çalışmaz")
```

Cobra'da yalnızca bu tek satırlık yorum biçimi vardır; çok satırlı blok yorumu (`/* ... */` gibi) yoktur. Birden çok satırı açıklamak isterseniz her satırın başına # koyarsınız.

1.6 Deyimler, İfadeler ve Satır Sonları

Cobra kodu **deyimlerden** (statements) oluşur. Bir deyim, çalıştırılabilen bir komuttur: bir değişken tanımı, bir atama, bir fonksiyon çağrısı, bir döngü gibi. Bir **ifade** (expression) ise bir değere indirgenen bir şeydir: `2 + 3`, `kare(9)`, `x and y` hep ifadelerdir.

Aradaki ayrım önemlidir ama Cobra bu konuda esnektir; örneğin atama bile bir ifadedir (bir değere indirgenir). Şimdilik kavraman gereken asıl kural şu:

Deyimler satır sonunda biter. Noktalı virgül yoktur.

Çoğu C-benzeri dilde her deyimin sonuna `;` koyarsınız. Cobra'da buna gerek yoktur — bir satırın bitmesi, deyimin de bittiği anlamına gelir:

```
let a = 1
let b = 2
print(a + b)   # 3
```

Yukarıdaki üç satır, üç ayrı deyimdir. Hiçbirinin sonunda noktalı virgül yoktur ve olmamalıdır.

Bu kuralın tek istisnası, parantezlerin içinde kalan satırlardır. Bir liste, sözlük veya fonksiyon çağrısı `(`, `[` ya da `{` ile açıldıysa, kapanana kadar birden çok satıra yayılabilir:

```
let sayilar = [
    1, 2, 3,
    4, 5, 6
]
print(len(sayilar))   # 6
```

Burada liste açık parantezler içinde olduğu için satır sonları deyimi bitirmez; Cobra köşeli ayraç kapanana kadar bekler.

> **Not:** Bir satıra birden çok deyim sıkıştırmak için bir ayraca (örneğin `;`) ihtiyaç duymazsınız çünkü öyle bir şey yok. Cobra'da "bir satır, bir deyim" kuralı sade ve nettir.

1.7 Tanı Araçları: `--ast` ve `--tokens`

Cobra, kodunuzun perde arkasında nasıl yorumlandığını görmeniz için iki tanı (diagnostic) bayrağı sunar. Bunlar günlük programlamada kullanmayacağınız, ama dili öğrenirken ya da bir sorunu derinlemesine anlamaya çalışırken çok değerli olan araçlardır.

`--tokens` bayrağı, kaynağı çalıştırmak yerine **ham simge akışını** (token stream) yazdırır. Yani Cobra'nın metninizi hangi parçalara böldüğünü görürsünüz: anahtar kelimeler, sayılar, operatörler, tanımlayıcılar:

```
cobra --tokens merhaba.cobra
```

`--ast` bayrağı ise bir adım ileri gider ve **ayrıştırılmış soyut söz dizimi ağacını** (AST) yazdırır — yani simgelerin nasıl yapılandırılmış bir programa dönüştürüldüğünü:

```
cobra --ast merhaba.cobra
```

Bu iki araç, Cobra'nın işlem hattının ilk iki aşamasına (sözcüksel çözümleme ve ayrıştırma) bir pencere açar. Şimdilik bu kadarını bilmeniz yeterli; bu bayrakların çıktısını okumayı ve dilin iç mimarisini ayrıntılı olarak Bölüm 12'de ele alacağız.

> **İpucu:** Bir söz dizimi hatasını anlayamadığınızda `--tokens` çıktısına bakmak çoğu zaman aydınlatıcıdır; Cobra'nın kodunuzu sizin sandığınızdan farklı böldüğünü hemen fark edebilirsiniz.

1.8 Kitap Haritası

Bu bölümde temeli attık: Cobra'nın ne olduğunu, hangi felsefeyle tasarlandığını, kodu nasıl çalıştıracağınızı ve dilin en temel kavramlarını gördük. Buradan sonra her bölüm bir katman daha ekleyecek.

Sıradaki bölümlerde değerlere ve değişkenlere (`let`, `const`, sayılar, metinler, mantıksal değerler), kontrol akışına (`if`/`elif`/`else`, `while`, `for ... in`, `range`), koleksiyonlara (listeler ve sözlükler, dilimleme), fonksiyonlara ve kapanışlara (kapanışlar, dekoratörler), yapılara (`struct`, `record`, `contract`, kalıtım), hata yönetimine (`try`/`catch`/`finally`, `throw`), modüllere ve standart kütüphaneye (`import`, `fs`, `math`, `json`, `http` ve daha fazlası) ve Cobra'nın gurur duyduğu GIL'siz eşzamanlılığa (`async`/`await`, `Mutex`, `Channel`) gireceğiz. Kitabın sonuna doğru, bu bölümde değindiğimiz `--ast` ve `--tokens` araçlarının ardındaki mimariyi — sözcüksel çözümleyiciden bayt-kodu sanal makinesine kadar — Bölüm 12'de derinlemesine inceleyeceğiz.

Her yeni kavramı, önce küçük ve çalışır örneklerle tanıtacak, sonra gerçek programlarda nasıl bir araya geldiklerini göstereceğiz. Tıpkı bu bölümde olduğu gibi, öğrenmenin en iyi yolu kodu kendiniz çalıştırmaktır.

1.9 Özet

Bu bölümde Cobra'nın temellerini öğrendik:

- **Cobra**, sade söz dizimli bir betik dilidir. Blokları `end` ile kapatır, mantıksal operatörleri `and`/`or`/`not` kelimeleriyle yazar ve girintiyi yalnızca kozmetik kabul eder.
- Dilin **iki çalışma motoru** vardır: varsayılan **ağaç-yürüten yorumlayıcı** ve yaklaşık 4 kat hızlı **bayt-kodu sanal makinesi** (`--vm`). İkisi aynı anlambilimi paylaşır ve birebir aynı çıktıyı üretir. Bunların ötesinde `--native`, programı **makine koduna derler** ve yorumlamayı tümüyle geride bırakır.
- Cobra **GIL içermez**; `async def` ile başlatılan görevler CPU çekirdeklerinde gerçek anlamda paralel koşar.

- Kodu çalıştırmanın yolları: `cobra dosya.cobra` (ağaç-yürütücü), `cobra --vm dosya.cobra` (VM), `cobra --native dosya.cobra` (makine koduna derlenmiş), `cobra build dosya.cobra` + `cobra dosya.cobrac` (derle ve çalıştır), ve argümansız `cobra` ile çok satırlı girişi destekleyen etkileşimli **REPL**.
- İlk programımızı `print("Merhaba, Cobra")` ile yazdık ve onu hem ağaç-yürütücüde hem VM'de çalıştırdık.
- **Yorumlar** `#` ile başlar. **Deyimler satır sonunda biter; noktalı virgül yoktur.** Parantez içindeki satırlar bu kuralın dışındadır.
- İki **tanı aracı** vardır: `--tokens` ham simge akışını, `--ast` ayrıştırılmış söz dizimi ağacını yazdırır. Ayrıntıları Bölüm 12'de göreceğiz.

Artık bir Cobra programı yazıp çalıştırabilir, dili REPL üzerinden keşfedebilir ve kodun nasıl yorumlandığına dair temel bir sezgiye sahip olursunuz. Bir sonraki bölümde değerleri ve değişkenleri ele alarak gerçek programlar yazmaya başlayacağız.

Bölüm 2

Temeller — Değişkenler, Tipler ve Operatörler

Her programlama dili, üzerine inşa edileceği birkaç temel taşla başlar: değerleri nereye koyacağımız (değişkenler), hangi türden değerlerle çalışabileceğimiz (tipler) ve bu değerleri nasıl birleştirip dönüştüreceğimiz (operatörler). Bu bölümde Cobra'nın bu üç temel taşını baştan sona ele alacağız. Cobra bilinçli olarak küçük ve okunabilir bir dil; bu yüzden burada öğreneceğiniz kurallar azdır ama tutarlıdır. Bir kez öğrendikten sonra dilin geri kalanı bu kuralların üzerine sorunsuzca oturur.

Tüm örnekleri çalıştırmak için bir `.cobra` dosyasına yazıp `cobra dosya.cobra` komutunu kullanabilir ya da `cobra` komutunu argümansız çalıştırarak etkileşimli REPL'de tek tek deneyebilirsiniz. Cobra'nın iki motoru vardır — varsayılan ağaç-yürüten yorumlayıcı ve `--vm` ile çalışan bayt-kodu sanal makinesi — ama ikisi de aynı sonuçları üretir, dolayısıyla bu bölümdeki hiçbir şey hangi motoru seçtiğinize bağlı değildir.

2.1 Değişkenler: `let` ve `const`

Cobra'da bir değişken oluşturmanın iki yolu vardır. `let` **değiştirilebilir** (mutable) bir bağ tanımlar; `const` ise **değişmez** (immutable) bir bağ.

```
let sayac = 0
const PI = 3.14159

print(sayac)   # 0
print(PI)      # 3.14159
```

Var olan bir değişkene yeni bir değer atamak için `let` tekrar yazılmaz; sadece adı ve `=` kullanılır:

```
let sayac = 0
sayac = sayac + 1
print(sayac)   # 1
```

Önemli bir kural: **daha önce tanımlanmamış** bir değişkene atama yapmak hatadır. Cobra, yazım hatalarını yakalayabilmek için her atamanın var olan bir bağı hedeflemesini ister.

```
let toplam = 10
toplma = 20      # HATA: 'toplma' tanımlı değil (yazım hatası yakalanır)
```

Yeni bir değişken istiyorsanız `let` ile bildirin; var olanı güncelliyorsanız `let` olmadan atayın. Bu ayrım, kazara yeni bir değişken oluşturup eski değeri gölgede bırakmanızı engeller.

> **Not** Cobra'nın iki motoru tanımsız değişkeni farklı zamanlarda yakalar: sanal makine (`--vm`) bunu derleme anında, ağaç-yürüten yorumlayıcı ise satır çalışırken yakalar. Hata mesajı her iki durumda da aynıdır.

2.1.1 Atama bir ifadedir

Cobra'da atama bir **deyim** değil, bir **ifadedir**: bir değer üretir. Bu, bazı durumlarda işinize yarar, ama çoğunlukla sadece kavramsal olarak bilmeniz yeterli olan bir ayrıntıdır.

```
let x = 0
let y = (x = 5)   # x'e 5 atanır, ve bu atamanın değeri de 5'tir
print(x)          # 5
print(y)          # 5
```

2.1.2 `const` bağlamayı dondurur, değeri değil

`const` ile tanımlanmış bir adı yeniden atamak hatadır. Aynı kapsamda onu `let` veya `const` ile yeniden bildirmek de hatadır.

```
const MAKS = 100
MAKS = 200        # HATA: bir sabit yeniden atanamaz
const MAKS = 5    # HATA: aynı kapsamda yeniden bildirilemez
```

Buradaki en sık karıştırılan nokta şudur: `const` **bağı** (yani adın hangi değere işaret ettiğini) dondurur, ama o değerin **iç içeriğini** değil. Bir listeyi veya sözlüğü `const` ile bağlarsanız, o adı başka bir değere yönlendiremezsiniz; ama listenin/sözlüğün içeriğini değiştirebilirsiniz.

```
const xs = [1]
xs.push(2)        # SERBEST: değerin içeriği değişiyor, bağ değil
print(xs)         # [1, 2]

xs = [9, 9]       # HATA: bağı değiştiremezsiniz
```

> **İpucu** Gerçekten değişmez bir veri yapısı istiyorsanız, `record` kullanın (sonraki bölümlerde göreceğiz). `const` sadece adın sabit kalacağını garanti eder, içeriğin değil.

2.1.3 İç kapsamda gölgeleme (shadowing)

Dış kapsamdaki bir `const` (ya da `let`), iç bir kapsamda yeni bir `let` ile **gölgelenebilir**. Bu, dıştaki bağı değiştirmez; sadece iç kapsam boyunca aynı adın yeni bir değere işaret etmesini sağlar.

```
const ad = "dış"

def goster()
    let ad = "iç"     # dış 'ad'ı gölgeler — hata değil
    print(ad)         # iç
end

goster()
print(ad)             # dış (değişmedi)
```

2.2 Tipler

Cobra'nın çekirdek değer tipleri sadedir. Bu bölümde sayısal ve basit tiplere odaklanıyoruz; liste, sözlük, fonksiyon ve struct gibi bileşik tipler kendi bölümlerinde ele alınacak.

Tip	Örnek	Açıklama
tamsayı (int)	`42`, `-7`	64-bit işaretli tam sayı
kayan nokta (float)	`3.14`, `2.0`	çift duyarlıklı ondalık
ondalık (decimal)	`19.90m`	tam/kesin aritmetik (finans)
string	`"merhaba"`	değişmez metin (yalnız çift tırnak)
boolean	`true`, `false`	doğruluk değeri (küçük harf)
null	`null`	"değer yok"

```
let n = 42            # int
let f = 3.14          # float
let para = 19.90m     # decimal
let s = "merhaba"     # string
let b = true          # boolean
let yok = null

print(type(n))        # INTEGER
print(type(f))        # FLOAT
print(type(s))        # STRING
print(type(b))        # BOOLEAN
print(type(yok))      # NULL
```

> **Not** Cobra'da boş/yok değeri her zaman `null`'dur. Başka dillerdeki `None`, `nil` veya `undefined` Cobra'da yoktur. Aynı şekilde booleanlar her zaman küçük harfli `true`/`false`'tur — `True`/`False` çalışmaz.

2.2.1 Tamsayılar 64-bittir

Cobra'nın tamsayıları 64-bit işaretlidir; yani yaklaşık $\pm 9.2 \times 10^{18}$ aralığını kapsar. Bu sınırın ötesine geçen "büyük tamsayı" (bignum) desteği **yoktur**; taşma olduğunda değer sarmalanır (wrap-around); otomatik olarak daha geniş bir tamsayı türüne büyümez.

2.2.2 Ondalık (decimal) — finans için kesin aritmetik

Kayan nokta sayıları (float) ikilik tabanda saklandığı için `0.1 + 0.2` gibi hesaplarda küçük yuvarlama hataları üretir. Para gibi kesinliğin şart olduğu durumlar için Cobra `decimal` tipini sunar. Bir sayının sonuna `m` ekleyerek ondalık değişmez (literal) yazarsınız:

```
let fiyat = 19.90m
let adet = 3
print(fiyat * adet)   # 59.70  (kesin, yuvarlama hatası yok)

print(decimal("0.1") + decimal("0.2"))   # 0.3  (tam olarak)
```

> **İpucu** Para, vergi, faiz gibi hesaplarda `float` yerine `decimal` kullanın. `float` bilimsel/mühendislik hesapları için, `decimal` ise finans için tasarlanmıştır.

2.3 Sayılarla çalışmak

2.3.1 int ve float karışımı float'a yükselir

Bir int ile bir float'ı bir aritmetik işlemde birleştirdiğinizde sonuç float olur. Buna **tip yükseltme** (promotion) denir.

```
print(1 + 2.5)        # 3.5   (sonuç float)
print(10 * 2.0)       # 20.0
print(1 == 1.0)       # true  (değerce eşit kabul edilir)
```

2.3.2 İki int arasında / TAMSAYI bölmesidir

Bu, başka dillerden gelenler için en sık şaşırtan kuraldır. İki tamsayı arasındaki `/` operatörü **tamsayı bölmesi** yapar: sonucu aşağı, sıfıra doğru keser.

```
print(7 / 2)          # 3    (tamsayı bölmesi — 3.5 değil!)
print(10 / 3)         # 3
```

Ondalıklı sonuç istiyorsanız operandlardan en az birini float'a çevirin:

```
print(float(7) / 2)   # 3.5
print(7 / 2.0)        # 3.5
```

> **Uyarı** Cobra'da `//` (taban bölmesi) operatörü **yoktur**. İki int arasındaki `/` zaten taban bölmesi gibi davranır; `7 // 2` yazmak söz dizimi hatasıdır.

2.3.3 Tip dönüşümleri

Bir değeri başka bir tipe çevirmek için adı tip adı olan yerleşik fonksiyonları kullanırsınız:

```
print(int(3.9))       # 3      (sıfıra doğru kırpar — yuvarlamaz)
print(int(-3.9))      # -3     (sıfıra doğru — aşağı değil)
print(float(7))       # 7.0
print(str(42))        # "42"   (sayı -> metin)
print(decimal(10))    # 10
```

Bu fonksiyonlar string ayrıştırma (parsing) için de çalışır:

```
print(int("100") + 1)       # 101
print(float("3.14"))        # 3.14
print(decimal("19.90"))     # 19.90
```

> **Not** `int(x)` ondalıklı bir sayıyı her zaman **sıfıra doğru** kırpar, matematiksel olarak aşağı yuvarlamaz. Yani `int(-3.9)` sonucu `-4` değil `-3`'tür. Gerçek yuvarlama için `math.round`, `math.floor`, `math.ceil` fonksiyonlarına bakın.

2.4 Operatörler

2.4.1 Aritmetik

```
print(5 + 3)          # 8
print(5 - 3)          # 2
print(5 * 3)          # 15
print(7 / 2)          # 3    (tamsayı bölmesi)
print(7 % 3)          # 1    (kalan / modulo)
```

`+` operatörü sayılarda toplama yapar, ama stringlerde **birleştirme** (concatenation) yapar:

```
print("merhaba " + "dünya")     # merhaba dünya
print("yaş: " + str(36))        # yaş: 36
```

> **Uyarı** `+` ya iki sayı ya da iki string ister; bir string ile bir sayıyı doğrudan toplayamazsınız. `"yaş: " + 36` hata verir; sayıyı önce `str(36)` ile metne çevirin. Cobra'da f-string veya metin araya yerleştirme (interpolation) yoktur; metinleri `+` ve `str(...)` ile kurarsınız.

2.4.2 Karşılaştırma

Karşılaştırma operatörleri her zaman bir boolean döndürür:

```
print(3 == 3)         # true
print(3 != 4)         # true
print(3 < 4)          # true
print(3 > 4)          # false
print(3 <= 3)         # true
print(4 >= 5)         # false
```

2.4.3 Mantıksal operatörler: `and`, `or`, `not`

Cobra'da mantıksal operatörler **sözcüktür** — `&&`, `||`, `!` değil. Bu, kodun daha okunabilir olmasını sağlar.

```
print(true and false)   # false
print(true or false)    # true
print(not true)         # false
```

`and` ve `or` **kısa devre** (short-circuit) yapar: sonucu belirlemeye yeten ilk operandı değerlendirdikten sonra durur. Daha da ilginci, bunlar bir boolean değil, **kararı veren operandın kendisini** döndürür.

```
print(0 or "x")         # x       (0 yanlış, ikinciye geçilir)
print("a" or "b")       # a       (ilki doğru, hemen döner)
print(1 and 2)          # 2       (ilki doğru, ikinci belirleyici)
print(0 and 2)          # 0       (ilki yanlış, hemen döner)
```

Bu davranış, varsayılan değer atamak için çok kullanışlıdır:

```
let ad = girilen_ad or "isimsiz"   # boşsa "isimsiz" kullan
```

2.5 Doğruluk (truthiness)

`if`, `while`, `and`, `or` ve üçlü ifade gibi yerlerde değerler bir boolean gibi yorumlanır. Cobra'da şu altı değer **yanlış** (falsy) sayılır:

- `false`
- `null`
- `0` (sıfır — int veya float)
- `""` (boş string)
- `[]` (boş liste)
- `{}` (boş sözlük)

Bunların dışındaki her şey **doğru** (truthy) sayılır.

```
if []
    print("bu çalışmaz")
else
    print("boş liste yanlıştır")    # bu çalışır
end

let liste = [1, 2, 3]
if liste
    print("dolu liste doğrudur")    # bu çalışır
end
```

> **İpucu** Bir listenin veya sözlüğün boş olup olmadığını kontrol etmek için `len(x) == 0` yazmanıza gerek yok; doğrudan `if not x` yeterlidir.

2.6 Artırılmış atama (augmented assignment)

`+=` `-=` `*=` `/=` `%=` operatörleri, bir bağı yerinde günceller. `x += 1` ifadesi tam olarak `x = x + 1` anlamına gelir; dolayısıyla `+` operatörünün tüm kurallarına (string birleştirme, int→float yükseltme, int/int tamsayı bölmesi) uyar.

```
let n = 10
n += 5     # n = n + 5
print(n)   # 15
n -= 3     # 12
n *= 2     # 24
n /= 5     # 4   (tamsayı bölmesi: 24 / 5 = 4)
n %= 3     # 1
print(n)   # 1

let mesaj = "mer"
mesaj += "haba"          # string birleştirme
print(mesaj)             # merhaba
```

Artırılmış atama yalnız basit değişkenlerde değil, **indeks hedeflerinde** ve **struct alanlarında** da çalışır:

```
let xs = [10, 20, 30]
xs[1] += 5
print(xs)               # [10, 25, 30]

let sayim = {"a": 1}
sayim["a"] += 1
print(sayim["a"])       # 2

struct Sayac
    def init()
        self.n = 0
    end
    def artir()
        self.n += 1     # alan üzerinde artırılmış atama
    end
end

let c = Sayac()
c.artir()
c.artir()
print(c.n)              # 2
```

> **Uyarı** Artırılmış atama bir `const`'a uygulanamaz; bir bağı değiştirdiği için `const TOPLAM = 0` üzerinde `TOPLAM += 1` yazmak hatadır.

2.7 Üçlü (ternary) ifade

Cobra, C ailesinden gelen `koşul ? a : b` üçlü ifadesini destekler: koşul doğruysa `a`, değilse `b` değerine eşittir. **Yalnız seçilen dal çalışır**, bu yüzden çalışmayan dalın yan etkileri tetiklenmez.

```
let yas = 20
let durum = yas >= 18 ? "yetişkin" : "çocuk"
print(durum)            # yetişkin
```

Üçlü ifade **sağ-ilişkilidir** (right-associative), yani zincirlendiğinde sağdan gruplanır:

```
# a ? b : c ? d : e   şu anlama gelir:   a ? b : (c ? d : e)
let puan = 75
let harf = puan >= 90 ? "A" : puan >= 80 ? "B" : puan >= 70 ? "C" : "F"
print(harf)             # C
```

Önceliği açısından üçlü ifade, `or`/`and`'den **daha gevşek**, fakat atamadan (`=`) **daha sıkı** bağlar. Bu sayede seçilen değeri doğrudan bir değişkene atayabilirsiniz:

```
let x = 5
let isaret = x > 0 ? "pozitif" : "negatif"   # = en gevşek olduğu için sorunsuz
print(isaret)           # pozitif
```

> **Not** Cobra'da `a if c else b` biçimindeki bir üçlü ifade **yoktur**; tek üçlü ifade biçimi `c ? a : b`'dir.

2.8 Yardımcı yerleşikler: `type` ve `hash`

`type(x)` bir değerin tip adını string olarak döndürür. Bu, hata ayıklamada ve çalışma anında tip kontrolünde işe yarar:

```
print(type(42))          # INTEGER
print(type(3.14))        # FLOAT
print(type("x"))         # STRING
print(type([1, 2]))      # LIST
print(type({"a": 1}))    # DICT
print(type(null))        # NULL
```

`hash(s)` bir değer için tamsayı bir özet (hash) üretir; aynı değer her zaman aynı özeti verir:

```
print(hash("merhaba") == hash("merhaba"))   # true
```

2.9 Operatör önceliği

Birden çok operatör bir araya geldiğinde hangisinin önce uygulanacağını **öncelik** (precedence) belirler. Aşağıdaki tabloda yukarıdan aşağı doğru öncelik artar (en altta en sıkı bağlayan):

Öncelik	Operatörler	Açıklama
en gevşek	`=`, `+=`, `−=`, …	atama (sağ-ilişkili)
	`? :`	üçlü ifade (sağ-ilişkili)
	`or`	mantıksal VEYA (kısa devre)
	`and`	mantıksal VE (kısa devre)
	`not`	mantıksal DEĞİL (tekli)
	`== != < > <= >=`	karşılaştırma
	`+ −`	toplama / çıkarma
	`* / %`	çarpma / bölme / modulo
	tekli `−`	negatif işaret
en sıkı	`f(...)`, `x[...]`, `x.y`	çağrı, indeks, alan erişimi

Bu tablodan çıkan bazı pratik sonuçlar:

```
print(2 + 3 * 4)         # 14   (* önce: 2 + 12)
print(not 3 == 3)        # false  (not (3 == 3) demektir)
print(1 + 2 == 3)        # true  (+ karşılaştırmadan sıkı: (1+2) == 3)
print(true or false and false)   # true  (and önce: true or (false and false))
```

Önceliği ister gruplama parantezleriyle (`(...)`) açıkça belirtin, ister okunabilirlik için tereddütte kaldığınızda parantez kullanın — Cobra'da gruplama parantezleri her zaman serbesttir.

2.10 Özet

Bu bölümde Cobra'nın temel yapı taşlarını öğrendik:

- **Değişkenler:** `let` değiştirilebilir bir bağ, `const` değişmez bir bağ tanımlar. Tanımsız bir değişkene atama hatadır. Atama bir ifadedir ve bir değer üretir. `const` yalnız **bağı** dondurur, değerin içeriğini değil (`const xs = [1]` → `xs.push(2)` serbest, `xs = ...` yasak). İç kapsamlar dış bir bağı `let` ile gölgeleyebilir.
- **Tipler:** int (64-bit), float, decimal (`19.90m`, finans için kesin), string, boolean (`true`/`false`) ve `null`. Boş/yok değeri her zaman `null`'dur.
- **Sayılar:** int/float karışımı float'a yükselir (`1 + 2.5` → `3.5`, `1 == 1.0` doğru). İki int arasında `/` tamsayı bölmesidir (`7 / 2` → `3`); ondalıklı sonuç için `float(7) / 2`. `//` operatörü yoktur.
- **Dönüşümler:** `int(x)` (sıfıra doğru kırpar), `float(x)`, `str(x)`, `decimal(x)` — hepsi string ayrıştırma da yapar.
- **Operatörler:** aritmetik `+ - * / %`; `+` stringlerde birleştirme yapar; karşılaştırma `== != < > <= >=`; mantıksal `and`/`or`/`not` (sözcük olarak). `and`/`or` kısa devre yapar ve **kararı veren operandı** döndürür (`0 or "x"` → `"x"`).
- **Doğruluk:** `false`, `null`, `0`, `""`, `[]`, `{}` yanlış sayılır; gerisi doğrudur.
- **Artırılmış atama:** `+= -= *= /= %=`; `x += 1` tam olarak `x = x + 1`'dir, değişken, indeks (`xs[i] += 1`) ve alan (`self.n += 1`) üzerinde çalışır, `const`'a uygulanamaz.
- **Üçlü ifade:** `koşul ? a : b`, sağ-ilişkili, yalnız seçilen dal çalışır; `=`'den sıkı, `or`/`and`'den gevşek bağlar.
- **Yardımcılar:** `type(x)` tip adını, `hash(s)` bir özeti döndürür.

Bu temellerle artık değerleri saklayabilir, dönüştürebilir ve birleştirebiliriz. Bir sonraki bölümde bu değerleri koşullar ve döngülerle bir araya getirerek programlarımıza akış kazandıracağız.

Bölüm 3

Kontrol Akışı

Bir programın değeri, yalnızca veriyi saklamasından değil, o veriye bakıp *karar verebilmesinden* gelir. Bir sayı sıfırdan büyük mü? Bir listede daha işlenecek eleman kaldı mı? Bir koşul sağlanana kadar bir işi tekrar tekrar yapmamız gerekiyor mu? İşte bu soruların yanıtını programa anlatma biçimine *kontrol akışı* (control flow) denir.

Bölüm 2'de değerleri, değişkenleri ve operatörleri gördük. Bu bölümde ise bunları bir araya getirip programın hangi satırların çalışacağına ve bir işin kaç kez tekrarlanacağına karar vermesini sağlayacağız. Cobra'nın kontrol akışı araçları bilinçli olarak azdır ve birbirine benzer; hepsi `end` anahtar kelimesiyle kapanan bloklar kullanır. Bu bölümü bitirdiğinizde koşullar (`if`/`elif`/`else`), iki tür döngü (`while` ve `for ... in`), tembel `range` dizileri ve döngü akışını yönlendiren `break`/`continue` ifadelerini rahatça kullanabiliyor olacaksınız.

Cobra'nın bütün blokları aynı kuralı izler: blok bir anahtar kelimeyle açılır (örneğin `if`, `while`, `for`), bir veya daha fazla satır gövdeden oluşur ve `end` ile kapanır. Girinti (indentation) tamamen kozmetiktir; okunabilirlik için kullanırız ama Cobra için anlamı yoktur. Blokları bağlayan tek şey `end`'tir.

3.1 Koşullar: `if`, `elif`, `else`

En temel karar yapısı `if` ifadesidir. Bir koşulu değerlendirir; koşul *doğruysa* (truthy) ilgili gövdeyi çalıştırır, değilse atlar.

```
let sicaklik = 32

if sicaklik > 30
    print("Bugün hava sıcak")
end
# Çıktı:
# Bugün hava sıcak
```

Birden fazla durumu sırayla denemek isterseniz `elif` (tek kelime — `else if` diye bir şey yoktur) ve `else` eklersiniz. Cobra koşulları yukarıdan aşağıya dener; *ilk doğru olan* dalın gövdesini çalıştırıp gerisini atlar. Hiçbiri doğru değilse `else` dalı çalışır (varsa).

```
let puan = 72

if puan >= 90
    print("AA")
```

```
elif puan >= 80
    print("BA")
elif puan >= 70
    print("BB")
else
    print("Geçti ya da kaldı, başka tabloya bak")
end
# Çıktı:
# BB
```

Buradaki üç dalın da koşulu teknik olarak doğru olabilir (72 hem `>= 70` hem de `>= 60`'tır), ama Cobra ilk eşleşen dalda durur. Bu yüzden koşulları en dar aralıktan en geniş aralığa doğru sıralamak doğru sonuç verir.

> **Not:** Cobra'da `elif` ve `else` zorunlu değildir. Tek başına bir `if` bloğu da tamamen geçerlidir. Ayrıca `else if` diye iki kelimelik bir biçim *yoktur*; doğru yazım tek kelime olan `elif`'tir.

3.1.1 Doğruluk (truthiness) kurallarını hatırlayalım

`if` ve `while`'ın koşulu illa `true`/`false` olmak zorunda değildir; Cobra herhangi bir değeri "doğru mu?" diye yorumlar. Bölüm 2'de gördüğümüz gibi şu değerler *yanlış* (falsy) sayılır:

- `false`
- `null`
- `0` (sayısal sıfır)
- `""` (boş string)
- `[]` (boş liste)
- `{}` (boş sözlük)

Bunların dışındaki her şey *doğru* (truthy) kabul edilir. Bu, koşulları çok okunaklı hâle getirir:

```
let isim = ""
if isim
    print("Merhaba " + isim)
else
    print("İsim boş")
end
# Çıktı:
# İsim boş

let liste = [1, 2, 3]
if liste
    print("Listede " + str(len(liste)) + " eleman var")
end
# Çıktı:
# Listede 3 eleman var
```

Mantıksal operatörlerin de kelime olduğunu hatırlayın: `and`, `or`, `not` (asla `&&`, `||`, `!` değil). `and` ve `or` kısa devre yapar ve kararı veren *operandı* döndürür:

```
let kullanici = null
let ad = kullanici or "misafir"   # kullanici falsy olduğu için "misafir"
print(ad)
# Çıktı:
# misafir
```

> **İpucu:** Sadece iki seçenek arasında *değer* seçmek istiyorsanız, koca bir `if/else` bloğu yazmak yerine üçlü (ternary) operatörü kullanabilirsiniz: `let durum = puan >= 50 ? "geçti" : "kaldı"`. Bu, `cond ? a : b` biçimindedir ve yalnızca seçilen dal çalışır.

3.2 `while` döngüsü

`while`, bir koşul doğru olduğu sürece gövdesini tekrar tekrar çalıştırır. Her turun başında koşula yeniden bakar; koşul yanlış olunca döngü biter ve program `end`'den sonraki satırdan devam eder.

```
let sayac = 1
while sayac <= 5
    print(sayac)
    sayac = sayac + 1
end
# Çıktı:
# 1
# 2
# 3
# 4
# 5
```

Dikkat edin: gövdenin içinde `sayac`'ı artırmazsak koşul hiçbir zaman yanlış olmaz ve döngü sonsuza kadar döner. Bir sayaç ya da birikim (accumulator) değişkeni tutmak `while` döngülerinde çok yaygındır. Klasik örnek, bir aralığın toplamını hesaplamaktır:

```
let n = 1
let toplam = 0
while n <= 100
    toplam += n        # toplam = toplam + n ile aynı
    n += 1
end
print(toplam)
# Çıktı:
# 5050
```

Burada iki değişken var: bir *sayaç* (`n`, nerede olduğumuzu izler) ve bir *birikim* (`toplam`, sonucu biriktirir). Bu ikili, döngülerin temel kalıbıdır.

> **Uyarı:** Bir `while` döngüsü yazarken kendinize "bu koşul *nasıl* yanlış olacak?" diye sorun. Eğer cevabınız yoksa sonsuz döngü yazmışsınız demektir. Genellikle gövdede sayacı ilerleten ya da koşulu değiştiren bir satır bulunmalıdır.

3.3 `for ... in`: bir şeyin üzerinde gezinmek

Çoğu zaman bir sayacı elle yönetmek istemeyiz; sadece bir koleksiyonun her elemanını sırayla işlemek isteriz. İşte `for ... in` tam da bunun içindir. Söz dizimi şöyledir:

```
for eleman in gezinilebilir
    # eleman ile bir şeyler yap
end
```

Cobra dört tür şey üzerinde gezinmenize izin verir: listeler, stringler, sözlük anahtarları ve `range(...)` dizileri. Hepsine tek tek bakalım.

3.3.1 Listeler üzerinde gezinmek

En sık karşılaşacağınız durum budur. `for`, listenin elemanlarını baştan sona, sırayla verir:

```
let meyveler = ["elma", "armut", "kiraz"]
for meyve in meyveler
    print(meyve.upper())
end
# Çıktı:
# ELMA
# ARMUT
# KIRAZ
```

3.3.2 Stringler üzerinde gezinmek (karakter karakter)

Bir string üzerinde gezindiğinizde Cobra size onu *karakter karakter* verir. Her tur, tek karakterlik bir string'tir:

```
for harf in "abc"
    print(harf)
end
# Çıktı:
# a
# b
# c
```

> **Not:** Stringler üzerinde gezinmek karakter karakter (rune) çalışır, ama bir string'i `s[i]` ile elle *indekslemek* bayt tabanlıdır ve çok baytlı UTF-8 karakterleri (örneğin Türkçe "ç", "ğ", "ş") bozabilir. Bir metni parçalara ayırırken `for` döngüsü ya da `re` modülü daha güvenli bir seçimdir.

3.3.3 Sözlük anahtarları üzerinde gezinmek (ekleme sırasıyla)

Bir sözlük (dict) üzerinde gezindiğinizde Cobra size *anahtarları* verir — hem de ekledikleri sırayla, çünkü Cobra'da sözlükler ekleme sırasını korur.

```
let yaslar = {"ali": 30, "veli": 25, "ayse": 35}
for isim in yaslar
    print(isim + ": " + str(yaslar[isim]))
end
# Çıktı:
# ali: 30
# veli: 25
# ayse: 35
```

Hem anahtar hem değeri bir arada gezinmek isterseniz `items()` kullanabilirsiniz; ama Cobra'da çoklu atama / açma (unpacking) olmadığını unutmayın. `items()` size `[anahtar, değer]` biçiminde iki elemanlı listeler verir ve siz bunları indeksleyerek okursunuz:

```
let yaslar = {"ali": 30, "veli": 25}
for ikili in yaslar.items()
    print(ikili[0] + " -> " + str(ikili[1]))
end
# Çıktı:
# ali -> 30
# veli -> 25
```

> **Uyarı:** `for anahtar, deger in d.items()` gibi bir yazım Cobra'da *çalışmaz* — dilde tuple ve unpacking yoktur. İki elemanlı listeyi `ikili[0]` ve `ikili[1]` ile okumanız gerekir.

3.4 `range`: sayı dizileri üretmek

Çoğu zaman bir şeyi belirli sayıda tekrarlamak ya da sayılar üzerinde gezinmek isteriz. `range`, bunun için arımetik (eşit aralıklı) bir sayı dizisi üretir. Üç biçimi vardır:

```
print(range(5))           # 0'dan 5'e kadar (5 dahil değil)
# Çıktı:
# [0, 1, 2, 3, 4]

print(range(2, 6))        # 2'den 6'ya kadar (6 dahil değil)
# Çıktı:
# [2, 3, 4, 5]

print(range(0, 10, 2))    # 0'dan 10'a, 2'şer adımla
# Çıktı:
# [0, 2, 4, 6, 8]
```

Yani:

- `range(stop)` → `0, 1, ..., stop-1`
- `range(start, stop)` → `start, start+1, ..., stop-1`
- `range(start, stop, step)` → `start`'tan başlayıp her seferinde `step` ekler, `stop`'a ulaşmadan durur.

Tipik kullanım, bir `for` döngüsünün başında:

```
for i in range(3)
    print("tur " + str(i))
end
# Çıktı:
# tur 0
# tur 1
# tur 2
```

3.4.1 `range` TEMBELdir (lazy) — gerçek bir liste değildir

`range` ekranda bir liste gibi yazdırılsa da aslında *tembel* (lazy) bir nesnedir. Tüm sayıları bellekte üretmez; gerektikçe hesaplar. `type()` ona sorduğunuzda `"RANGE"` döndürür — `"LIST"` değil:

```
let r = range(1, 10, 2)
print(type(r))    # "LIST" değil!
# Çıktı:
# RANGE

print(r[0])       # indekslenebilir
# Çıktı:
# 1

print(r)          # liste gibi yazdırılır
# Çıktı:
# [1, 3, 5, 7, 9]
```

Bir RANGE'i indeksleyebilir, üzerinde gezinebilir ve dilimleyebilirsiniz (slice); ama o gerçek bir liste *değildir*. Listelere özgü değiştirici metotları (örneğin `.push`) yoktur:

```
let r = range(3)
r.push(4)
# runtime error: line 2: RANGE has no method 'push'
```

> **İpucu:** Bir RANGE'i gerçek, değiştirilebilir bir listeye çevirmek isterseniz onun üzerinde gezinip yeni bir listeye eleman ekleyebilir veya `map`/`filter` gibi yüksek mertebeli fonksiyonlara verebilirsiniz; bunlar size yeni bir liste döndürür. `range`'in tembelliği, milyonlarca elemanlı bir aralıkta dönmenize hiçbir bellek maliyeti olmadan izin verdiği için bir avantajdır.

`range`'in büyük faydası budur: `for i in range(1000000)` yazmak bir milyon elemanlı liste oluşturmaz; her sayıyı tam ihtiyaç anında üretir.

3.5 `break` ve `continue`: döngü akışını yönlendirmek

Döngünün doğal akışına iki şekilde müdahale edebiliriz. `break` döngüyü *tamamen* sonlandırır; `continue` ise sadece *o anki turu* atlayıp bir sonrakine geçer. İkisi de hem `while` hem `for` döngülerinde çalışır.

`break` ile bir aramayı, hedefi bulunca durdurabiliriz:

```
let sayilar = [4, 8, 15, 16, 23, 42]
for s in sayilar
    if s > 20
        print("20'den büyük ilk sayı: " + str(s))
        break
    end
end
# Çıktı:
# 20'den büyük ilk sayı: 23
```

`continue` ile bazı elemanları atlayabiliriz. Aşağıda yalnızca tek sayıları topluyoruz:

```
let toplam = 0
for n in range(1, 6)
    if n % 2 == 0
        continue          # çift sayıları atla
```

```
    end
    toplam += n
end
print(toplam)             # 1 + 3 + 5
# Çıktı:
# 9
```

3.5.1 Döngü dışında `break`/`continue` bir AYRIŞTIRMA hatasıdır

`break` ve `continue` yalnızca bir döngünün içinde anlamlıdır. Onları döngü *dışında* kullanmaya çalışırsanız, program daha çalışmadan, ayrıştırma (parse) aşamasında hata verir:

```
break
# parse error: line 1: 'break' outside loop
```

Önemli bir incelik: bir döngünün *içine* yazılmış olsa bile, eğer `break`/ `continue` bir `def` (fonksiyon tanımı) gövdesinin içindeyse yine geçersizdir. Çünkü fonksiyon kendi içinde döngüden bağımsız bir bloktur; içerideki `break`'in dışarıdaki döngüye atlaması mümkün değildir:

```
for i in range(3)
    def f()
        break        # döngünün içine yazılmış olsa da fonksiyonun içinde
    end
end
# parse error: line 3: 'break' outside loop
```

> **Uyarı:** Bu hata *ayrıştırma* zamanında yakalanır, yani program hiç çalışmaya başlamadan. Bu iyi bir şeydir: yanlış yerleştirilmiş bir `break`/`continue`'yu çalışma zamanına bırakmaz, anında size bildirir.

3.6 Döngü değişkeni döngüden sonra yaşar

Bazı dillerde döngü değişkeninin ömrü döngü gövdesiyle sınırlıdır. Cobra'da *öyle değildir*: bir `for` döngüsünün değişkeni döngü bittikten sonra da erişilebilir kalır ve son aldığı değeri tutar.

```
for n in [1, 2, 3]
    # ...
end
print(n)
# Çıktı:
# 3
```

Bu davranış, döngüden sonra "en son nerede kaldık?" bilgisine ihtiyaç duyduğunuzda işe yarayabilir. Ama bir uyarı da getirir: aynı isimli bir değişkeni döngüden sonra başka bir amaçla kullanacaksanız, döngü değişkeninin o ismi hâlâ tuttuğunu unutmayın.

> **Not:** Aynı şey `while` için de geçerlidir — `while` içinde `let` ile tanımladığınız ya da güncellediğiniz değişkenler döngüden sonra da yaşar. Cobra'da bloklar (döngüler, `if`'ler) yeni bir kapsam (scope) açmaz; yalnızca `def` gövdeleri açar.

3.7 İç içe döngüler

Döngüleri iç içe yerleştirebilirsiniz. Bu, örneğin bir tablo (satır × sütun) üretmek için doğaldır. İçteki döngü, dıştaki döngünün her turu için baştan sona çalışır.

```
for i in range(1, 4)
    for j in range(1, 4)
        print(str(i) + " x " + str(j) + " = " + str(i * j))
    end
end
# Çıktı:
# 1 x 1 = 1
# 1 x 2 = 2
# 1 x 3 = 3
# 2 x 1 = 2
# 2 x 2 = 4
# 2 x 3 = 6
# 3 x 1 = 3
# 3 x 2 = 6
# 3 x 3 = 9
```

İç içe döngülerde `break` davranışına dikkat edin: `break` *yalnızca onu çevreleyen en içteki döngüyü* sonlandırır. Dıştaki döngü kaldığı yerden devam eder.

```
for i in range(2)
    for j in range(3)
        if j == 1
            break          # sadece içteki (j) döngüsünü kırar
        end
        print(str(i) + "," + str(j))
    end
end
# Çıktı:
# 0,0
# 1,0
```

Görüldüğü gibi içteki döngü `j == 1` olunca kırılır, ama dıştaki döngü `i = 0` ve `i = 1` için tekrar çalışır.

> **İpucu:** Bazı diller "etiketli `break`" (labeled break) ile en dıştaki döngüden tek hamlede çıkmaya izin verir; Cobra'da bu yoktur. İç içe bir döngüden tamamen çıkmanız gerekiyorsa, bir bayrak (flag) değişkeni kullanabilir ya da ilgili kodu bir `def` içine taşıyıp `return` ile çıkabilirsiniz.

3.8 Cobra'da OLMAYANLAR: `for(;;)`, `switch`, `match`

Başka dillerden gelen okuyucular için birkaç önemli farkı netleştirelim. Bu yapıların hiçbiri Cobra'da yoktur; yazmaya çalışırsanız hata alırsınız.

- **C tarzı `for(;;)` döngüsü yoktur.** Cobra'da `for` yalnızca `for x in gezinilebilir` biçiminde çalışır. Sayaç tabanlı bir döngü istiyorsanız `range` kullanın (`for i in range(n)`) ya da `while` ile elle bir sayaç yönetin.

- **`switch` / `case` yoktur.** Birden çok dalı karşılaştırmak için `if/elif/else` zincirini kullanın.
- **`match` (örüntü eşleme / pattern matching) yoktur.** Yine `if/elif/else` ile, ya da bir değeri davranışlara eşlemek için bir sözlük kullanabilirsiniz.

Bir `switch` özlemini `if/elif` ile şöyle karşılarsınız:

```
let komut = "kaydet"

if komut == "ac"
    print("Dosya açılıyor")
elif komut == "kaydet"
    print("Dosya kaydediliyor")
elif komut == "kapat"
    print("Dosya kapatılıyor")
else
    print("Bilinmeyen komut")
end
# Çıktı:
# Dosya kaydediliyor
```

Çok sayıda durum varsa, bir sözlüğü "atlama tablosu" (dispatch table) gibi kullanmak daha temiz olabilir — özellikle değerleri fonksiyonlarla eşlediğinizde:

```
def topla(a, b)
    return a + b
end

def cikar(a, b)
    return a - b
end

let islemler = {"+": topla, "-": cikar}
let f = islemler["+"]
print(f(10, 4))
# Çıktı:
# 14
```

> **Not:** Bir fonksiyon gövdesinin Cobra'da kendi satır(lar)ında olması *zorunludur*; `def topla(a, b) return a + b end` gibi tek satıra sıkıştırılmış bir yazım ayrıştırma hatası verir. Sözlüğün değerleri olarak fonksiyonları saklayabilmemiz, Cobra'da fonksiyonların birinci sınıf değer olmasından kaynaklanır (Bölüm 4'te bunu derinlemesine göreceğiz).

3.9 Özet

Bu bölümde Cobra'nın programlarına karar verdirme ve işleri tekrarlatma araçlarını gördük. Hepsi `end` ile kapanan, aynı sade kalıbı izleyen bloklardı:

- **Koşullar** `if cond ... elif cond ... else ... end` biçimindedir. Cobra dalları yukarıdan aşağıya dener ve ilk doğru olanı çalıştırır. Koşullar herhangi bir değeri *doğruluk* kuralına göre yorumlar: `false`, `null`, `0`, `""`, `[]` ve `{}` yanlış, diğer her şey doğrudur (Bölüm 2).

- **`while cond ... end`**, koşul doğru olduğu sürece döner. Tipik kalıbı bir *sayaç* ile bir *birikim* değişkenidir; koşulun bir gün yanlış olacağından emin olun, yoksa sonsuz döngü yazarsınız.
- **`for x in gezinilebilir ... end`** listeler, stringler (karakter karakter), sözlük anahtarları (ekleme sırasıyla) ve `range(...)` üzerinde gezinir.
- **`range`** üç biçimde gelir — `range(stop)`, `range(start, stop)`, `range(start, stop, step)` — ve *tembel* bir RANGE üretir: indekslenebilir, gezilebilir ve dilimlenebilir, ama gerçek bir liste değildir (`.push` yok) ve `type()` onun için `"RANGE"` döndürür.
- **`break`** döngüyü tamamen bitirir; **`continue`** o turu atlar. İkisi de `while` ve `for`'da çalışır. Bir döngü dışında — ya da döngü içindeki bir `def` gövdesinde — kullanmak bir *ayrıştırma* hatasıdır.
- Bir **döngü değişkeni döngüden sonra da yaşar** ve son değerini tutar.
- **İç içe döngülerde** `break` yalnızca onu çevreleyen en içteki döngüden çıkar; etiketli `break` yoktur.
- Cobra'da **C tarzı `for(;;)`, `switch` ve `match` yoktur**; bunların yerine `range`/`while`, `if`/`elif`/`else` ve gerektiğinde bir sözlükle atlama tablosu kullanırsınız.

Artık veriyi tutmayı (Bölüm 2) ve programın akışını yönlendirmeyi biliyoruz. Bir sonraki bölümde bu parçaları yeniden kullanılabilir birimler hâline getireceğiz: fonksiyonlar.

Bölüm 4

Koleksiyonlar — Listeler, Sözlükler ve Dilimleme

Bir programın değeri çoğu zaman tek başına duran değerlerde değil, o değerleri bir arada tutma biçiminde saklıdır. Bir alışveriş sepeti, bir kullanıcı kaydı, bir günün sıcaklık ölçümleri — bunların hepsi *koleksiyon* ister. Cobra bu iş için iki temel yapı sunar: sıralı bir diziyi tutan **liste** ve anahtarlardan değerlere eşleme yapan **sözlük** (dict). Bu iki yapı, yazacağınız Cobra programlarının omurgasını oluşturur.

Bu bölümde listeleri ve sözlükleri en baştan ele alacağız: nasıl oluşturulur, nasıl okunur, nasıl değiştirilir; hangi metotlar yerinde değişiklik yapar, hangileri yeni bir değer döndürür. Cobra'nın en güçlü ve en çok şaşırtan özelliklerinden biri olan **dilimleme** (slicing) için ayrı bir kısım ayırdık. Bir de dikkatli olunması gereken bir konu var: listeler ve sözlükler *referans tiptir*. `let b = a` yazdığınızda bir kopya değil, aynı nesneye ikinci bir isim elde edersiniz. Bunun nedenini ve sonuçlarını da göreceğiz.

Başlamadan önce küçük bir hatırlatma: Cobra'da bloklar `end` ile kapanır, satır sonları ifadeleri bitirir, yorumlar `#` ile başlar. Çıktıları kod örneklerinin içinde `#` yorumuyla göstereceğiz.

4.1 Liste oluşturmak

Bir liste, köşeli parantez içinde virgülle ayrılmış değerlerden oluşur. Cobra listeleri *heterojendir*: aynı listede farklı türde değerler bulunabilir.

```
let xs = [1, 2, 3]
let karisik = [1, 2, "x"]
let bos = []

print(xs)        # [1, 2, 3]
print(karisik)   # [1, 2, x]
print(len(bos))  # 0
```

`len(...)` yerleşik fonksiyonu listenin eleman sayısını verir. Boş liste `[]` ile yazılır ve uzunluğu sıfırdır.

> **Not** Köşeli parantezlerin içinde liste birden çok satıra yayılabilir. Cobra, `(`, `[` ve `{` içindeyken satır sonunu ifadenin sonu saymaz, bu yüzden uzun listeleri okunaklı biçimde dizebilirsiniz:
>
> ```
> let renkler = [
>     "kirmizi",
> ```

```
        "yesil",
        "mavi",
    ]
```

4.2 İndeksleme ve negatif indeks

Listenin elemanlarına sıfırdan başlayan bir indeksle ulaşırsınız.

```
let l = ["a", "b", "c", "d"]
print(l[0])   # a
print(l[2])   # c
print(len(l)) # 4
```

Cobra'nın hoş bir kolaylığı vardır: *negatif indeks* sondan saymaya yarar. `l[-1]` her zaman son elemandır, `l[-2]` sondan ikincidir.

```
print(l[-1])  # d
print(l[-2])  # c
```

Bu sayede "son eleman"a ulaşmak için `l[len(l) - 1]` yazmaya gerek kalmaz.

> **Uyarı** Liste sınırlarının dışına çıkan bir indeks (örneğin dört elemanlı bir listede `l[4]` ya da `l[-5]`) bir çalışma zamanı hatası üretir. İndekslemede sınır dışı değerler *kırpılmaz*. Bu, birazdan göreceğimiz dilimlemeden önemli bir farktır — dilimlemede sınır dışı değerler sessizce kırpılır.

4.3 İndeks ataması

Bir listenin belirli bir konumundaki değeri, o konuma atama yaparak değiştirebilirsiniz.

```
let l = [10, 20, 30]
l[0] = 99
l[-1] = 0
print(l)   # [99, 20, 0]
```

Augmented (artırımlı) atama da indeks hedeflerinde çalışır; `l[i] += 1` tam olarak `l[i] = l[i] + 1` anlamına gelir:

```
let sayac = [0, 0, 0]
sayac[1] += 5
print(sayac)   # [0, 5, 0]
```

4.4 Birleştirme ve eşitlik

`+` operatörü iki listeyi birleştirip *yeni* bir liste üretir. Kaynak listeler değişmez.

```
let a = [1, 2]
let b = [3, 4]
let c = a + b
print(c)   # [1, 2, 3, 4]
print(a)   # [1, 2]  (a değişmedi)
```

Listeler == ile *derinlemesine* karşılaştırılır: aynı uzunlukta ve karşılıklı elemanları eşitse iki liste eşittir. Bu karşılaştırma iç içe listeler için de geçerlidir.

```
print([1, 2, 3] == [1, 2, 3])     # true
print([1, [2, 3]] == [1, [2, 3]]) # true  (derin eşitlik)
print([1, 2] == [1, 2, 3])        # false
```

> **Not** Sayısal türlerin karışımında değer eşitliği geçerlidir: `1 == 1.0` doğrudur, dolayısıyla `[1, 2] == [1.0, 2.0]` da `true` döner.

4.5 Listeler referans tiptir

Burası dikkat isteyen kısım. Cobra'da bir listeyi başka bir değişkene atadığınızda *kopya oluşmaz*; iki değişken de aynı listeyi gösterir. Buna *takma ad* (alias) denir.

```
let a = [1, 2, 3]
let b = a        # b, a'nın kopyası DEĞİL — aynı listeye takma ad
b.push(4)
print(a)         # [1, 2, 3, 4]  (a da değişti!)
print(b)         # [1, 2, 3, 4]
print(a == b)    # true (zaten aynı liste)
```

`b` üzerinde yaptığınız değişiklik `a` üzerinden de görünür, çünkü ortada tek bir liste vardır. Aynı durum bir listeyi fonksiyona geçirdiğinizde de geçerlidir: fonksiyon içinde listeyi değiştirirseniz, çağıran taraf bu değişikliği görür.

```
def ekle_bir(liste)
    liste.push(99)
end

let veri = [1, 2]
ekle_bir(veri)
print(veri)   # [1, 2, 99]
```

Eğer gerçekten bağımsız bir kopya istiyorsanız, dilimlemeyi kullanarak sığ bir kopya çıkarabilirsiniz. `a[:]` ifadesi tüm elemanları içeren *yeni* bir liste döndürür (dilimlemeyi birazdan ayrıntılı göreceğiz):

```
let a = [1, 2, 3]
let kopya = a[:]   # yeni bir liste
kopya.push(4)
print(a)           # [1, 2, 3]  (etkilenmedi)
print(kopya)       # [1, 2, 3, 4]
```

> **İpucu** Birden çok satırda paylaşılan bir listeyi yanlışlıkla değiştirip beklenmedik sonuçlar alıyorsanız, ilk şüpheliniz takma ad olsun. "Neden bu liste de değişti?" sorusunun cevabı genellikle "çünkü ikisi aslında aynı liste"dir. Bağımsız bir kopya istediğinizde `liste[:]` kullanın.

4.6 Liste metotları

Listelerin üzerinde nokta ile çağrılan bir dizi metot vardır. Bunların bir kısmı listeyi *yerinde* değiştirir (mutator), bir kısmı yeni bir değer döndürür. İkisini ayırt etmek hata ayıklamayı çok kolaylaştırır.

Metot	Ne yapar	Yerinde mi?
`push(v)`	Sona eleman ekler	Evet (yerinde)
`pop()`	Son elemanı çıkarır ve döndürür	Evet (yerinde)
`reverse()`	Listeyi ters çevirir	Evet (yerinde)
`sort()`	Listeyi sıralar	Evet (yerinde)
`contains(v)`	Eleman var mı? `true`/`false`	Hayır (okur)
`find(v)`	İlk geçtiği indeks, yoksa `-1`	Hayır (okur)
`count(v)`	Kaç kez geçtiği	Hayır (okur)

Önce yerinde değişiklik yapanlar:

```
let xs = [3, 1, 2]
xs.push(4)
print(xs)          # [3, 1, 2, 4]

let son = xs.pop()
print(son)         # 4
print(xs)          # [3, 1, 2]

xs.sort()
print(xs)          # [1, 2, 3]

xs.reverse()
print(xs)          # [3, 2, 1]
```

`sort` ve `reverse` listeyi *yerinde* değiştirir ve listeyi yeniden döndürmezler; "sıralanmış kopya" almak isterseniz önce kopya çıkarın (`let sirali = xs[:]` sonra `sirali.sort()`).

Şimdi listeyi okuyan, değiştirmeyen metotlar:

```
let nesneler = ["elma", "armut", "elma", "kiraz"]
print(nesneler.contains("armut")) # true
print(nesneler.find("elma"))      # 0   (ilk geçtiği indeks)
print(nesneler.find("muz"))       # -1  (bulunamadı)
print(nesneler.count("elma"))     # 2
```

`find` bulunamayan eleman için `-1` döndürür; bu yüzden "var mı?" sorusu için `contains` daha okunaklıdır, "nerede?" sorusu için `find` uygundur.

4.6.1 Yerleşik fonksiyon biçimleri

`push` ve `pop` aynı zamanda yerleşik (builtin) fonksiyon olarak da çağrılabilir. `xs.push(v)` ile `push(xs, v)` aynı işi yapar; `xs.pop()` ile `pop(xs)` da öyle.

```
let xs = [1, 2]
push(xs, 3)       # xs.push(3) ile aynı
print(xs)         # [1, 2, 3]

let son = pop(xs) # xs.pop() ile aynı
print(son)        # 3
print(xs)         # [1, 2]
```

Hangisini tercih edeceğiniz tamamen üslup meselesidir. Metot biçimi (`xs.push(3)`) çoğu durumda daha akıcı okunur.

4.7 Dilimleme (slicing)

Dilimleme, bir dizinin bir *alt parçasını* almanın kısa ve güçlü yoludur. Genel söz dizimi `seq[low:high:step]` biçimindedir:

- `low` — başlangıç indeksi (dahil)
- `high` — bitiş indeksi (hariç)
- `step` — adım (varsayılan 1)

En güzel yanı, herhangi bir parçanın *atlanabilmesidir*. Atlanan parça mantıklı bir varsayılanla doldurulur.

```
let xs = [0, 1, 2, 3, 4, 5]

print(xs[1:4])   # [1, 2, 3]   (1'den 4'e kadar, 4 hariç)
print(xs[:2])    # [0, 1]      (baştan 2'ye kadar)
print(xs[3:])    # [3, 4, 5]   (3'ten sona kadar)
print(xs[:])     # [0, 1, 2, 3, 4, 5]  (tam kopya)
```

Son satıra dikkat: `xs[:]` listenin baştan sona *yeni bir kopyasını* döndürür. Az önce gördüğümüz "bağımsız kopya" hilesi tam olarak budur.

4.7.1 Negatif indeksler dilimde de çalışır

Tıpkı tekil indekslemede olduğu gibi, dilimleme sınırlarında da negatif sayılar sondan sayar.

```
let xs = [0, 1, 2, 3, 4, 5]
print(xs[-2:])   # [4, 5]      (son iki eleman)
print(xs[:-2])   # [0, 1, 2, 3] (sondan iki hariç hepsi)
print(xs[-3:-1]) # [3, 4]
```

4.7.2 Adım ve ters çevirme

Üçüncü parça, `step`, kaçar kaçar ilerleneceğini söyler. İki atlayarak gitmek için `2`, geriye gitmek için negatif bir adım verilir.

```
let xs = [0, 1, 2, 3, 4, 5]
print(xs[::2])   # [0, 2, 4]   (ikişer atlayarak)
print(xs[1::2])  # [1, 3, 5]
print(xs[::-1])  # [5, 4, 3, 2, 1, 0]  (tersine çevrilmiş kopya)
```

`xs[::-1]` ifadesi listenin ters çevrilmiş bir *kopyasını* döndürür — `reverse()` ile karıştırmayın: `reverse()` listeyi yerinde değiştirir ve hiçbir şey döndürmez, `xs[::-1]` ise orijinali bırakır, yeni bir liste verir.

4.7.3 Sınır dışı değerler kırpılır

Dilimlemenin tekil indekslemeden en önemli farkı budur: dilim sınırları listenin dışına taşarsa hata oluşmaz, değerler sessizce *kırpılır*.

```
let xs = [0, 1, 2]
print(xs[0:100]) # [0, 1, 2]   (100 yerine sona kırpıldı)
print(xs[-100:]) # [0, 1, 2]   (çok geriye gitti, başa kırpıldı)
print(xs[5:9])   # []          (tamamen dışarıda — boş liste)
```

Bu davranış, hesapladığınız indeksin sınırı aşıp aşmadığını tek tek kontrol etmekten kurtarır. `xs[i:i+3]` yazabilir, listenin sonuna yakın olsanız bile hata almayacağınızı bilirsiniz.

> **Not** Dilimleme listelerde yeni bir liste döndürür. Aynı söz dizimi *stringlerde* de çalışır, ama orada rune (UTF-8) tabanlıdır — yani baytlara değil karakterlere göre keser; bunu Bölüm 5'te ele alacağız. `range(...)` üzerinde dilimleme yaptığınızda ise sonuç tembel bir RANGE değeridir (bütün elemanları belleğe açmaz).

> **Uyarı** Cobra'da dilim *ataması* yoktur. `xs[1:3] = [9, 9]` gibi bir ifade geçerli değildir. Bir aralığı değiştirmek istiyorsanız, ya tek tek indeks atamasıyla (`xs[1] = 9`) ya da yeni bir liste kurup `+` ile birleştirerek (`xs[:1] + [9, 9] + xs[3:]`) ilerleyin.

4.8 Sözlükler (dict)

Sözlük, *anahtar*lardan *değer*lere eşleme yapan bir koleksiyondur. Süslü parantez içinde `anahtar: değer` çiftleriyle yazılır.

```
let kullanici = {"ad": "cobra", "yas": 3}
print(kullanici["ad"])   # cobra
print(kullanici["yas"])  # 3
```

Anahtarlar yalnızca string olmak zorunda değildir. Cobra'da anahtarlar **string, integer, boolean veya null** olabilir.

```
let d = {"ad": "cobra", 1: "one", true: "evet", null: "yok"}
print(d[1])      # one
print(d[true])   # evet
print(d[null])   # yok
```

Sözlükler **ekleme sırasını korur**: çiftleri yazdığınız ya da eklediğiniz sıra, üzerinde gezindiğinizde de aynı kalır.

4.8.1 Okuma, ekleme, güncelleme

`d[k]` ifadesi bir anahtarın değerini okur. `d[k] = v` ise anahtarı ekler (yoksa) ya da değerini günceller (varsa).

```
let yaslar = {"ali": 30, "veli": 25}
yaslar["ayse"] = 35        # yeni anahtar ekler
yaslar["ali"] = 31         # mevcut anahtarı günceller
print(yaslar)              # {ali: 31, veli: 25, ayse: 35}
```

Augmented atama sözlük değerlerinde de çalışır:

```
let puan = {"x": 10}
puan["x"] += 5
print(puan["x"])   # 15
```

> **Uyarı** Var olmayan bir anahtarı `d[k]` ile okumaya çalışmak bir *hatadır*, `null` döndürmez. Anahtarın varlığından emin değilseniz, önce `has` ile kontrol edin ya da güvenli okuma için `get` kullanın (her ikisini de aşağıda göreceğiz).

4.9 Sözlük metotları

Sözlüklerin de hem nokta ile çağrılan metot biçimi, hem de yerleşik fonksiyon biçimi vardır.

Metot	Yerleşik biçim	Ne yapar
`d.keys()`	`keys(d)`	Anahtarların listesi
`d.values()`	`values(d)`	Değerlerin listesi
`d.items()`	—	`[anahtar, değer]` çiftlerinin listesi
`d.has(k)`	`has(d, k)`	Anahtar var mı? `true`/`false`
`d.get(k, varsayilan?)`	—	Değer, yoksa varsayılan (ya da `null`)
`d.del(k)`	`del(d, k)`	Anahtarı siler

```
let yaslar = {"ali": 30, "veli": 25}

print(yaslar.keys())     # [ali, veli]
print(yaslar.values())   # [30, 25]
print(yaslar.has("ali")) # true
print(yaslar.has("xyz")) # false
```

`get` metodu, var olmayan bir anahtar için hata fırlatmak yerine bir varsayılan döndürmesiyle güvenli okumayı sağlar. Varsayılan vermezseniz, eksik anahtar için `null` döner.

```
let yaslar = {"ali": 30}
print(yaslar.get("ali", 0))   # 30
print(yaslar.get("zzz", 0))   # 0   (anahtar yok, varsayılan döndü)
print(yaslar.get("zzz"))      # null (varsayılan verilmedi)
```

`del` bir anahtarı kaldırır:

```
let yaslar = {"ali": 30, "veli": 25}
yaslar.del("veli")
print(yaslar)   # {ali: 30}
```

`keys`, `values` ve `has` için yerleşik fonksiyon biçimleri de aynı işi görür:

```
let d = {"a": 1, "b": 2}
print(keys(d))      # [a, b]
print(values(d))    # [1, 2]
print(has(d, "a"))  # true
del(d, "a")
print(d)            # {b: 2}
```

> **Not** Sözlükler de listeler gibi referans tiptir. `let b = a` aynı sözlüğe takma ad yapar; biri üzerinden eklenen anahtar diğerinden de görünür. Bağımsız bir kopya gerekiyorsa, anahtarları gezerek yeni bir sözlük kurmanız gerekir (sözlüklerde liste dilimlemesinin karşılığı yoktur).

4.10 Sözlükte gezinmek

Bir sözlüğün üzerinde `for ... in` ile doğrudan gezindiğinizde, döngü değişkeni *anahtarları* (ekleme sırasıyla) alır:

```
let yaslar = {"ali": 30, "veli": 25}
for ad in yaslar
    print(ad + " -> " + str(yaslar[ad]))
end
# ali -> 30
# veli -> 25
```

Hem anahtara hem değere aynı anda erişmek istediğinizde `items()` kullanırsınız. `items()` size `[anahtar, değer]` biçiminde iki elemanlı listelerden oluşan bir liste verir.

```
for cift in yaslar.items()
    print(cift[0], cift[1])
end
# ali 30
# veli 25
```

> **Uyarı** Cobra'da *unpacking* (çoklu çözme) yoktur. Yani `for anahtar, deger in d.items()` gibi bir yazım geçerli değildir. Bunun yerine çifti tek bir değişkene alır ve `cift[0]` (anahtar) ile `cift[1]` (değer) olarak indekslersiniz. Aynı kısıt `zip` ve `enumerate` çıktıları için de geçerlidir.

4.11 Olmayanlar: kavrama ve küme

Diğer dillerden gelenlerin sıkça aradığı iki özelliği Cobra *bilinçli olarak* bulundurmaz:

- **Liste/sözlük kavraması (comprehension)** yoktur. Yani `[x * x for x in xs]` gibi bir söz dizimi geçerli değildir.
- **Küme (set)** veri tipi yoktur.

Bu, eksiklik değil, sadeliği koruma tercihidir. Kavrama yerine `map` ve `filter` yüksek-mertebe fonksiyonlarını ya da düz bir `for` döngüsünü kullanırsınız:

```
def kare(n)
    return n * n
end

let xs = [1, 2, 3, 4]
print(map(kare, xs))   # [1, 4, 9, 16]

# ya da döngüyle:
let kareler = []
for x in xs
    kareler.push(x * x)
end
print(kareler)         # [1, 4, 9, 16]
```

Küme davranışına (tekrarsız üyelik) ihtiyaç duyduğunuzda, sözlüğü bir küme gibi kullanmak yaygın bir kalıptır: anahtarları üye olarak tutar, değer olarak `true` koyarsınız. Üyelik kontrolü `has` ile yapılır. `map`, `filter`, `reduce` ve dostlarını Bölüm 6'da ayrıntısıyla ele alacağız.

4.12 Bir araya getirelim: küçük bir kelime sayacı

Öğrendiklerimizi tek bir örnekte toplayalım. Bir metindeki kelimeleri sayan, listeleri ve sözlükleri birlikte kullanan küçük bir program:

```
def kelime_say(metin)
    let sayim = {}
    for kelime in metin.split(" ")
        let k = kelime.strip().lower()
        if k == ""
            continue
        end
        if sayim.has(k)
            sayim[k] += 1
        else
            sayim[k] = 1
        end
    end
    return sayim
end

let sonuc = kelime_say("kedi kopek kedi kus kedi")
for cift in sonuc.items()
    print(cift[0] + ": " + str(cift[1]))
end
# kedi: 3
# kopek: 1
# kus: 1
```

Bu programda her aracı görüyoruz: bir sözlüğü baştan boş kuruyoruz, `has` ile anahtarın varlığını yokluyoruz, `+=` ile sayacı artırıyoruz, `items()` ile gezinip `cift[0]` ve `cift[1]` ile anahtar ve değere ulaşıyoruz. `del` kul-

lanmadan önce mevcut bir anahtarı `get` ile güvenle de okuyabilirdik; örneğin `sayim[k] = sayim.get(k, 0) + 1` tek satırda aynı işi yapar.

4.13 Özet

- **Listeler** köşeli parantezle yazılır (`[1, 2, "x"]`), heterojendir ve sıfırdan indekslenir. Negatif indeks sondan sayar (`l[-1]` son elemandır). `l[0] = v` ile yerinde atama, `+` ile birleştirme yapılır; `==` derinlemesine karşılaştırır.
- **Listeler referans tiptir.** `let b = a` bir kopya değil, takma ad oluşturur — biri üzerinden yapılan değişiklik diğerinden de görünür. Bağımsız bir sığ kopya için `a[:]` kullanın.
- **Liste metotları:** `push`, `pop`, `reverse`, `sort` yerinde değişiklik yapar; `contains`, `find` (yoksa `-1`), `count` listeyi okur. `push` ve `pop` için `push(xs, v)` / `pop(xs)` yerleşik biçimleri de vardır.
- **Dilimleme** `seq[low:high:step]` biçimindedir; her parça atlanabilir (`xs[:2]`, `xs[1:]`, `xs[:]`), negatif indeksler sondan sayar, negatif adım ters çevirir (`xs[::-1]`), sınır dışı değerler hata vermeden *kırpılır*. Listelerde yeni liste, stringlerde rune tabanlı sonuç (Bölüm 5), range'lerde tembel RANGE döner. **Dilim ataması yoktur.**
- **Sözlükler** süslü parantezle yazılır (`{"ad": "cobra", 1: "one"}`); anahtarlar string, int, boolean veya null olabilir; ekleme sırası korunur. `d[k]` okur (eksik anahtar *hatadır* — `has` ile kontrol edin veya `get` kullanın), `d[k] = v` ekler/günceller.
- **Sözlük metotları:** `keys`, `values`, `items`, `has`, `get(k, varsayilan?)`, `del`; çoğunun yerleşik biçimleri de var (`keys(d)`, `has(d, k)`, `del(d, k)`).
- **Gezinme:** `for ... in d` anahtarları verir; anahtar+değer için `for cift in d.items()` kullanın. Cobra'da **unpacking yoktur**, bu yüzden `cift[0]` (anahtar) ve `cift[1]` (değer) olarak indeksleyin.
- Cobra'da **kavrama (comprehension) ve küme (set) yoktur.** Bunların yerine `map`/`filter` (Bölüm 6) ya da düz bir `for` döngüsü kullanın; küme davranışı için değerleri `true` olan bir sözlük yaygın bir kalıptır.

Bölüm 5

Metinlerle Çalışmak

Bir programın dünyayla konuştuğu yer çoğu zaman metindir. Kullanıcıdan gelen bir satır, bir dosyanın içeriği, bir HTTP yanıtının gövdesi, ekrana bastığınız her şey... Bunların hepsi *string* (dizgi) olarak gelir ve string olarak gider. Bu bölümde Cobra'da metinleri nasıl kurduğunuzu, parçaladığınızı, aradığınızı ve dönüştürdüğünüzü göreceğiz.

Cobra'nın string modeli küçük ama keskin birkaç kuraldan oluşur. En önemlisini baştan söyleyelim: **stringler değişmezdir (immutable)**. Bir stringi yerinde değiştiremezsiniz; her "değişiklik" aslında *yeni* bir string üretir. Bu, ilk bakışta kısıtlayıcı görünse de hata ayıklamayı basitleştirir: elinizdeki bir stringin sizin sırtınızdan başkası tarafından değiştirilmeyeceğinden emin olabilirsiniz.

İkinci olarak Cobra'da **f-string veya string interpolasyonu yoktur**. Bazı dillerdeki `f"x={x}"` ya da `` `x=${x}` `` gibi gömülü ifade biçimleri burada çalışmaz. Metinleri `+` operatörü ve `str(...)` dönüştürücüsüyle elle kurarsınız. Bu bölüm boyunca bu deseni defalarca göreceksiniz.

5.1 Stringleri kurmak: `+` ve `str(...)`

En temel metin kurma aracı `+` operatörüdür. İki stringi yan yana birleştirir:

```
let ad = "Ada"
let soyad = "Lovelace"
print(ad + " " + soyad)
# Ada Lovelace
```

Ama `+` yalnızca string ile string'i birleştirir. Bir sayıyı doğrudan ekleyemezsiniz; önce onu `str(...)` ile stringe çevirmeniz gerekir:

```
let yas = 36
print("Yas: " + str(yas))
# Yas: 36
```

`str(...)` her değeri okunabilir bir metne dönüştürür: tam sayıları, ondalıkları, mantıksal değerleri, hatta listeleri ve sözlükleri.

```
print(str(3) + str(14))     # 314  (metin birlestirme, toplama DEGIL!)
print(str(3.14))            # 3.14
print(str(true))            # true
print(str([1, 2, 3]))       # [1, 2, 3]
```

> **Uyarı:** `str(3) + str(14)` sonucu `"314"`'tür, `17` değil. `+` iki string üzerinde *birleştirme* yapar, aritmetik toplama değil. Sayısal toplama istiyorsanız `str(...)` çağrılarını kaldırın: `3 + 14`.

Interpolasyonun yokluğu, birden çok değeri bir araya getirirken zincir zincir `+` ve `str(...)` yazmanız gerektiği anlamına gelir:

```
let urun = "kalem"
let adet = 12
let fiyat = 5
print("Siparis: " + str(adet) + " adet " + urun + ", toplam " + str(adet * fiyat) + " TL")
# Siparis: 12 adet kalem, toplam 60 TL
```

> **İpucu:** Çok parçalı mesajlarda `print` virgülle ayrılmış birden çok argüman da alır ve aralarına boşluk koyar: `print("Toplam", adet * fiyat, "TL")`. Bu, tek bir string *üretmez* ama ekrana basmak için pratiktir.

5.2 Kaçış dizileri

String literallerinin içine doğrudan yazamayacağınız karakterleri *kaçış dizileriyle* (escape sequence) eklersiniz. Cobra'da stringler yalnızca çift tırnakla yazılır, dolayısıyla en sık ihtiyaç duyacağınız kaçış, çift tırnağın kendisidir.

Dizi	Anlamı
`\n`	satır sonu (yeni satır)
`\t`	sekme (tab)
`\"`	çift tırnak karakteri
`\\`	ters bölü (backslash) karakteri

```
print("satir1\nsatir2")
# satir1
# satir2

print("Ad:\tAda")
# Ad:   Ada

print("o dedi \"selam\"")
# o dedi "selam"

print("yol: C:\\Users")
# yol: C:\Users
```

> **Not:** `\n` *tek* bir karakterdir (satır sonu), iki karakter değil. Bir stringin içinde gerçek bir ters bölü görmek istiyorsanız onu `\\` diye yazmalısınız; aksi halde Cobra bir sonraki karakterle birlikte kaçış dizisi oluşturmaya çalışır.

5.3 String metotları

Cobra'nın stringleri zengin bir metot kümesiyle gelir. Hepsi nokta çağrısıyla kullanılır (`s.upper()`) ve hiçbiri orijinal stringi değiştirmez — değişmezlik kuralı gereği hepsi *yeni* bir değer döndürür.

5.3.1 Büyük/küçük harf: `upper`, `lower`

```
print("merhaba".upper())   # MERHABA
print("MERHABA".lower())   # merhaba
```

5.3.2 Boşluk temizleme: `strip`, `lstrip`, `rstrip`

`strip` baştaki ve sondaki boşlukları kırpar; `lstrip` yalnızca soldakini, `rstrip` yalnızca sağdakini.

```
print("  bosluk  ".strip() + "|")    # bosluk|
print("   sol".lstrip() + "|")       # sol|
print("sag   ".rstrip() + "|")       # sag|
```

> **Uyarı:** Bu üç metot argüman almaz; her zaman *boşluk* karakterlerini (boşluk, sekme, satır sonu) temizler. `"xxabc".lstrip("x")` gibi bir çağrı hata verir. Belirli bir karakteri sökmek istiyorsanız `replace` kullanın.

5.3.3 Bölme ve birleştirme: `split`, `join`

`split` bir stringi parçalara ayırıp liste döndürür. Bir ayırıcı verirseniz ona göre böler:

```
print("a,b,c".split(","))      # ["a", "b", "c"]
print("a,b,,c".split(","))     # ["a", "b", "", "c"]
```

Argümansız çağırırsanız `split`, boşluklara göre böler ve aradaki çoklu boşlukları akıllıca yutar — metni "kelimelere" ayırmanın en pratik yolu budur:

```
print("Ali   Veli  Deli".split())   # ["Ali", "Veli", "Deli"]
print(len("a b   c".split()))       # 3
```

`join` ise bunun tersidir: bir listeyi araya ayırıcı koyarak tek bir stringe çeker. Ayırıcı, metodu üzerinde çağırdığınız stringtir:

```
print(", ".join(["x", "y", "z"]))   # x, y, z
print("-".join(["2026", "06", "27"]))   # 2026-06-27
```

5.3.4 Değiştirme: `replace`

Bir alt dizginin *tüm* geçtiği yerleri başka bir metinle değiştirir:

```
print("aaa".replace("a", "b"))           # bbb
print("yol/ile/yol".replace("/", "."))   # yol.ile.yol
```

5.3.5 Arama: `contains`, `startswith`, `endswith`, `find`, `count`

```
print("merhaba".contains("rha"))     # true
print("merhaba".startswith("mer"))   # true
print("merhaba".endswith("aba"))     # true
print("merhaba".find("h"))           # 3   (ilk gecisin indeksi)
print("merhaba".find("z"))           # -1  (bulunamadi)
print("banana".count("a"))           # 3   (kac kez geciyor)
```

> **Not:** `find` bulamadığında bir hata fırlatmaz, `-1` döndürür. Bu yüzden `if s.find(x) != -1` deyimi, "x bu stringin içinde var mı?" sorusunun klasik yazımıdır — ama çoğu zaman doğrudan `s.contains(x)` daha okunaklı olur.

5.3.6 Metotları zincirlemek

Her metot yeni bir string (ya da liste) döndürdüğü için çağrıları arka arkaya dizebilirsiniz. Soldan sağa okunur:

```
print("  a,b  ".strip().split(","))
# ["a", "b"]

print("  Ali, Veli , Deli  ".strip().split(","))
# ["Ali", " Veli ", " Deli"]
```

> **İpucu:** İkinci örnekte virgülden sonra gelen boşlukların hâlâ parçaların içinde durduğuna dikkat edin. Bunları da temizlemek için her parçaya tek tek `strip` uygulamanız gerekir — örneğin bir `for` döngüsünde ya da `map` ile.

5.4 İndeksleme ve dilimleme

Bir stringin tek bir konumuna köşeli parantezle erişebilirsiniz:

```
print("merhaba"[0])   # m
print("merhaba"[3])   # h
```

ASCII metinlerde bu kusursuz çalışır. Ama Cobra'da stringler altta UTF-8 bayt dizileri olarak saklanır ve **tek konum indekslemesi rune değil, BAYT tabanlıdır.** Türkçe karakterler (ş, ı, ğ, ö, ü, ç, İ…) UTF-8'de birden çok bayt kapladığı için, çok baytlı bir karakterin üzerine `[i]` ile düşmek bozuk bir sonuç verir:

```
let s = "şehir"
print(s[0])   # Å   (ş'nin ILK baytı — bozuk!)
```

İşte bu noktada **dilimleme (slicing)** devreye girer. Dilimleme rune tabanlıdır, yani UTF-8 güvenlidir ve Türkçe karakterlerde doğru sonuç verir:

```
let s = "şehir"
print(s[0:1])   # ş   (dogru — rune tabanli dilim)
print(s[1:3])   # eh
```

Aynı UTF-8 güvenliği ters çevirme ve sondan dilimlemede de geçerlidir:

```
print("şığöüç"[::-1])   # çüöğış   (tersine cevrilmis, dogru)
print("Türkçe"[::-1])   # eçkrüT
print("Türkçe"[-3:])    # kçe      (son uc rune)
```

Dilimleme söz dizimi `s[bas:son:adim]` biçimindedir; her parça atlanabilir, negatif indeksler sondan sayar, negatif adım tersine çevirir, sınır dışı değerler hata vermeden kırpılır — tıpkı listelerdeki gibi (bkz. Bölüm 4).

> **Uyarı:** `len` bir stringin **bayt** sayısını verir, rune (görünen karakter) sayısını değil. Çok baytlı karakterlerde ikisi farklıdır:
>
> ```
> print(len("abc"))   # 3   (3 ASCII karakter = 3 bayt)
> print(len("şğ"))    # 4   (2 karakter ama 4 bayt!)
> ```
>
> Görünen karakter sayısını saymanız gerekiyorsa metni rune'lara ayırmanın güvenli yolu, `re` modülüyle ya da dilimlemeyle çalışmaktır.

> **İpucu:** Pratik kural: **Türkçe (ya da herhangi bir Unicode) metinde tek karakter çekmeniz gerektiğinde `s[i]` yerine `s[i:i+1]` dilimini kullanın.** Karmaşık konumlandırma gerekiyorsa `re` modülüne geçin. Ham bayt indekslemesini yalnızca metnin tamamen ASCII olduğundan eminseniz kullanın.

5.5 Kod noktaları: `chr` ve `ord`

Bazen bir karakterin Unicode kod noktasını (numarasını) ya da tam tersini, bir numaradan karakteri elde etmeniz gerekir. Bunun için iki yerleşik fonksiyon vardır:

- `chr(n)` — `n` kod noktasına karşılık gelen tek karakterlik stringi verir.
- `ord(s)` — `s` stringinin ilk karakterinin kod noktasını verir.

```
print(chr(65))    # A
print(chr(351))   # ş
print(ord("A"))   # 65
print(ord("ş"))   # 351
```

Bunlar birbirinin tersidir: `chr(ord("ş"))` yine `"ş"` verir. Karakter aritmetiği yapmak (örneğin bir harfi alfabede kaydırmak) ya da kontrol karakterleri üretmek için kullanışlıdırlar:

```
# 'a'dan baslayarak ilk bes harf
let harfler = ""
for i in range(5)
    harfler = harfler + chr(ord("a") + i)
end
print(harfler)   # abcde
```

> **Not:** `chr`/`ord` kod *noktalarıyla* (rune) çalışır, baytlarla değil. Bu yüzden çok baytlı Türkçe karakterlerde de doğru numarayı verirler — ham bayt indekslemesinin aksine.

5.6 Düzenli ifadelere köprü: `re` modülü

Sabit metinlerle aramak için string metotları yeterlidir; ama *kalıplarla* (rakam dizisi, e-posta biçimi, birden çok ayırıcı…) çalışmanız gerektiğinde `re` modülü devreye girer. Burada yalnızca tadına bakacağız; ayrıntılar **Bölüm 11**'de.

```
import "re"

print(re.matches("[0-9]+", "abc123"))           # true   (kalip eslesiyor mu?)
print(re.find("[0-9]+", "abc123def456"))         # 123    (ilk eslesme)
print(re.find_all("[0-9]+", "abc123def456"))     # ["123", "456"]
print(re.replace("[aeiou]", "merhaba", "*"))     # m*rh*b*
print(re.split(",\\s*", "a, b,c,  d"))           # ["a", "b", "c", "d"]
```

Son örnekteki `re.split`, "virgül ve ardından sıfır veya daha fazla boşluk" kalıbına göre bölerek `split(",")`'nin beceremediği, değişken boşlukları da temizler.

> **Uyarı:** Cobra'nın `re` modülü RE2 söz dizimini kullanır. RE2 her zaman doğrusal zamanda çalışır (katastrofik geri izleme yoktur), ama bunun karşılığında **lookaround (ileri/geri bakış) ve backreference (geri başvuru) desteklemez.** Başka dillerden alıştığınız `(?=...)` ya da `\1` gibi kalıplar burada çalışmaz.

> **İpucu:** Kalıbınızdaki `\` karakterlerini Cobra string literalinde `\\` diye yazmanız gerektiğini unutmayın. Yukarıdaki `,\\s*` aslında düzenli ifade motoruna `,\s*` olarak ulaşır.

5.7 Özet

- Stringler **değişmezdir**: her metot orijinali bozmadan yeni bir değer döndürür.
- **F-string / interpolasyon yoktur**; metinleri `+` ve `str(...)` ile elle kurarsınız (`"deger=" + str(x)`).
- `+` stringlerde *birleştirmedir*; sayıları eklemeden önce `str(...)` ile çevirin.
- Kaçış dizileri: `\n`, `\t`, `\"`, `\\`.
- String metotları: `upper`, `lower`, `strip`, `lstrip`/`rstrip` (argümansız, yalnız boşluk), `split` (argümansızken boşluğa göre böler), `join`, `replace`, `contains`, `startswith`, `endswith`, `find` (indeks veya `-1`), `count`. Hepsi zincirlenebilir: `"  a,b  ".strip().split(",")`.
- Tek konum indekslemesi (`s[i]`) **bayt tabanlıdır** ve çok baytlı Türkçe karakterleri bozar; **dilimleme (`s[1:3]`, `s[::-1]`) rune tabanlı ve UTF-8 güvenlidir.** Unicode metinde tek karakter için `s[i:i+1]` kullanın.
- `len` bayt sayar; görünen karakter sayısı için dilimleme/`re` kullanın.
- `chr(n)` ↔ `ord(s)` kod noktası ile karakter arasında çevirir (rune tabanlı, UTF-8 doğru).
- Kalıplarla arama/değiştirme/bölme için `re` modülü (RE2; lookaround ve backreference yok). Ayrıntılar Bölüm 11'de.

Bölüm 6

Fonksiyonlar

Fonksiyonlar, programlarınızı tekrar kullanılabilir parçalara böldüğünüz yapı taşlarıdır. Cobra'da fonksiyonlar yalnızca "kod gruplama" aracı değildir; onlar tam anlamıyla **değerlerdir**: bir değişkene atayabilir, başka bir fonksiyona argüman olarak geçebilir, bir fonksiyondan geri döndürebilir ve çevrelerini "kapatarak" (closure) yaşatabilirsiniz.

Bu bölümde fonksiyon tanımının kurallarını, değer döndürmeyi, birinci sınıf değer olmanın ne anlama geldiğini, kapanışları (closure), Cobra'nın bilinçli olarak *eksik bıraktığı* özellikleri (varsayılan değer yok, `lambda` yok), yerleşik yüksek-mertebeli fonksiyonları ve son olarak dekoratörleri inceleyeceğiz. Cobra'nın felsefesi sade sözdizimidir; bu yüzden fonksiyonlar da gücünü çok sayıda özellikten değil, az sayıda özelliğin tutarlı davranışından alır.

6.1 Fonksiyon Tanımlamak: `def ... end`

Bir fonksiyon `def` anahtar kelimesiyle başlar, adı ve parantez içindeki parametreleriyle devam eder ve `end` ile kapanır. Cobra blokları girinti veya süslü parantezle değil, **`end`** ile kapatır.

```
def add(a, b)
    return a + b
end

print(add(2, 3))   # 5
```

Burada kritik bir kural vardır: **fonksiyon gövdesi `def` satırından SONRAKİ satırlarda olmalıdır.** Cobra'da satır içi (inline) gövde yoktur. Yani aşağıdaki gibi her şeyi tek satıra sıkıştıramazsınız:

```
# YANLIŞ — gövde def satırında olamaz
def add(a, b) return a + b end
```

Bu kural, dilin "her ifade bir satırda biter, bloklar `end` ile kapanır" ilkesinin doğal sonucudur. Gövde her zaman kendi satır(lar)ında yer alır.

> **Not:** `func` anahtar kelimesi `def` ile tam olarak eş anlamlıdır. İsterseniz `func add(a, b)` yazabilirsiniz; davranış birebir aynıdır. Bu kitapta tutarlılık için `def` kullanacağız.

Fonksiyonu çağırmak için adının ardından parantez içinde argümanları yazarsınız. Parantez, argüman olmasa bile zorunludur:

```
def greet()
    print("merhaba")
end

greet()   # merhaba
```

6.2 `return` ile Değer Döndürmek

Bir fonksiyon `return` ifadesiyle çağrıldığı yere bir değer geri verir. `return` çalıştığı anda fonksiyon biter; ondan sonraki satırlar çalışmaz.

```
def absolute(n)
    if n < 0
        return -n
    end
    return n
end

print(absolute(-7))   # 7
print(absolute(3))    # 3
```

6.2.1 `return` Olmadan: `null`

Cobra'da `return` olmayan bir fonksiyon otomatik olarak `null` döndürür. Bu, "hiçbir şey döndürmeyen" fonksiyonların aslında `null` döndürdüğü anlamına gelir.

```
def log(msg)
    print("[log] " + msg)
end

let sonuc = log("başladı")   # [log] başladı
print(sonuc)                 # null
```

6.2.2 Çıplak `return`

Bir değer belirtmeden sadece `return` yazabilirsiniz. Bu, fonksiyondan erken çıkmanın yoludur ve yine `null` döndürür:

```
def process(items)
    if len(items) == 0
        return            # erken çıkış; null döner
    end
    print("işleniyor: " + str(len(items)) + " öğe")
end

process([])          # (hiçbir şey yazmaz)
process([1, 2, 3])   # işleniyor: 3 öğe
```

> **İpucu:** Çıplak `return`'ü "koruma cümlesi" (guard clause) olarak kullanmak kodu okunaklı kılar: geçersiz durumları en başta eleyip ana mantığı girinti derinliğinden kurtarırsınız.

6.3 Fonksiyonlar Birinci Sınıf Değerlerdir

Cobra'da bir fonksiyon, tıpkı bir sayı veya bir metin gibi, bir değerdir. Onu bir değişkene atayabilirsiniz:

```
def add(a, b)
    return a + b
end

let f = add        # fonksiyon bir değerdir — parantez YOK
print(f(2, 3))     # 5
print(type(add))   # FUNCTION
```

Burada `add` (parantezsiz) fonksiyonun kendisini ifade eder; `add(2, 3)` ise onu *çağırır*. Bu ayrım, fonksiyonları başka fonksiyonlara geçirmenin temelidir. Fonksiyonları listelerde veya sözlüklerde de saklayabilirsiniz:

```
def topla(a, b) return a + b end
def cikar(a, b) return a - b end

let islemler = {"+": topla, "-": cikar}
print(islemler["+"](10, 4))   # 14
print(islemler["-"](10, 4))   # 6
```

> **Not:** Yukarıdaki `def topla(a, b) return a + b end` gibi yazımlar bu kitapta sadece *sözlük kuralını* göstermek için kısaltma gibi görünebilir; ancak unutmayın: gerçek kod yazarken gövde kendi satırında olmalıdır. Burada tek satırda görünmesi belge biçimlendirmesidir — kendi programlarınızda gövdeyi alt satıra alın.

6.4 Kapanışlar (Closures)

Cobra fonksiyonları, tanımlandıkları kapsamı (scope) "kapatır" — yani iç içe tanımlanmış bir fonksiyon, kendisini çevreleyen fonksiyonun değişkenlerine erişmeye devam eder, o fonksiyon çoktan dönmüş olsa bile. Buna **kapanış** (closure) denir.

```
def adder(n)
    def add(x)
        return x + n     # dıştaki n'yi yakalar
    end
    return add
end

let add5 = adder(5)
let add10 = adder(10)

print(add5(10))    # 15
print(add10(10))   # 20
print(add5(1))     # 6
```

`adder(5)` çağrısı geri döndükten sonra bile, döndürdüğü `add` fonksiyonu `n = 5` değerini hatırlar. Her `adder` çağrısı kendine ait bağımsız bir `n` yakalar; bu yüzden `add5` ve `add10` birbirini etkilemez. Kapanışlar, durum (state) taşıyan fonksiyonlar üretmenin güçlü bir yoludur:

```
def counter()
    let n = 0
    def next()
        n = n + 1
        return n
    end
    return next
end

let say = counter()
print(say())   # 1
print(say())   # 2
print(say())   # 3
```

Burada `next`, dıştaki `n`'yi sadece okumakla kalmaz, onu *değiştirir* de. Yakalanan değişken paylaşılan ve yaşayan bir bağlamadır.

6.5 Argümanlar: Tam Olarak Bildirildiği Kadar

Cobra fonksiyonları, bildirildikleri sayıda **konumsal** (positional) argüman alır — ne bir eksik, ne bir fazla. Bu konuda dil bilinçli olarak katıdır ve şunları **sunmaz**:

- **Varsayılan (default) değer yok.** `def f(a, b = 10)` diye bir şey yoktur.
- **Anahtar-kelime (keyword) argüman yok.** `f(b = 3)` diye çağıramazsınız.
- **Değişken sayıda argüman yok.** Kullanıcı fonksiyonlarında `*args` veya `**kwargs` bulunmaz.

```
def greet(name, greeting)
    return greeting + ", " + name + "!"
end

print(greet("Ada", "Merhaba"))   # Merhaba, Ada!

# greet("Ada")  -> HATA: fonksiyon iki argüman bekler
```

> **Uyarı:** Bir fonksiyonu yanlış sayıda argümanla çağırmak bir hatadır. "İsteğe bağlı parametre" davranışı istiyorsanız, bunu açıkça yönetmeniz gerekir: ya farklı isimli iki fonksiyon yazın ya da tek bir parametreyi dict olarak alıp `get(anahtar, varsayilan)` ile eksik değerleri doldurun.

Varsayılan değer benzeri bir davranış için yaygın bir desen, bir yapılandırma sözlüğü geçirmektir:

```
def connect(options)
    let host = options.get("host", "localhost")
    let port = options.get("port", 8080)
    return host + ":" + str(port)
end

print(connect({}))                       # localhost:8080
print(connect({"port": 9090}))           # localhost:9090
print(connect({"host": "db", "port": 5432}))   # db:5432
```

6.6 `lambda` Yok: İç İçe `def` Kullanın

Birçok dilde küçük, isimsiz fonksiyonlar için bir `lambda` sözdizimi vardır. Cobra'da **`lambda` yoktur.** Bunun yerine, "satır içi" bir fonksiyona ihtiyacınız olduğunda, iç içe bir `def` yazıp onu döndürürsünüz. Az önce gördüğümüz `adder` deseni tam olarak budur:

```
def make_multiplier(factor)
    def multiply(x)
        return x * factor
    end
    return multiply
end

let triple = make_multiplier(3)
print(triple(7))   # 21
```

Bu yaklaşım biraz daha uzun görünebilir, ama fonksiyona her zaman bir isim verdiği için yığın izlerinde (stack trace) ve hata mesajlarında okunabilirlik kazandırır. Sadelik adına ödenen küçük bir bedeldir.

6.7 Yüksek-Mertebeli Yerleşik Fonksiyonlar

Fonksiyonlar değer olduğu için, onları argüman olarak alan fonksiyonlar yazabilirsiniz. Bunlara **yüksek-mertebeli (higher-order)** fonksiyon denir. Cobra, en sık ihtiyaç duyulanları yerleşik olarak sunar. Hepsinin ortak bir özelliği vardır: **hepsi EAGER'dır**, yani sonucu tembel bir iterator değil, hemen oluşturulmuş bir **liste** olarak döndürürler.

6.7.1 `map(fn, list)`

Listenin her öğesine `fn`'i uygular, sonuçlardan yeni bir liste üretir:

```
def square(n)
    return n * n
end

print(map(square, [1, 2, 3, 4]))   # [1, 4, 9, 16]
```

6.7.2 `filter(fn, list)`

`fn`'in doğru (truthy) döndürdüğü öğeleri tutar:

```
def is_even(n)
    return n % 2 == 0
end

print(filter(is_even, [1, 2, 3, 4, 5, 6]))   # [2, 4, 6]
```

6.7.3 `reduce(fn, list, init?)`

Listeyi **soldan** katlayarak tek bir değere indirger. `fn` her adımda `fn(acc, elem)` biçiminde çağrılır: `acc` o ana kadar birikmiş değer, `elem` sıradaki öğedir. İsteğe bağlı `init` başlangıç değerini verir; verilmezse ilk öğe başlangıç olarak kullanılır.

```
def add(acc, x)
    return acc + x
end

print(reduce(add, [1, 2, 3, 4]))        # 10   (başlangıç = ilk öğe)
print(reduce(add, [1, 2, 3], 100))      # 106  (başlangıç = 100)
```

6.7.4 `zip(list, ...)`

Birden çok listeyi öğe öğe eşleştirir; her biri en kısa listeye kadar kesilir. Sonuç, alt listelerden oluşan bir listedir:

```
print(zip([1, 2, 3], ["a", "b", "c"]))   # [[1, "a"], [2, "b"], [3, "c"]]
print(zip([1, 2], ["a", "b", "c"]))      # [[1, "a"], [2, "b"]]  (kısaya göre)
```

6.7.5 `enumerate(list, start?)`

Her öğeyi indeksiyle birlikte `[indeks, değer]` çiftleri olarak verir. İsteğe bağlı `start` indeksin nereden başlayacağını belirler:

```
for pair in enumerate(["x", "y", "z"])
    print(str(pair[0]) + " -> " + pair[1])
end
# 0 -> x
# 1 -> y
# 2 -> z

print(enumerate(["a", "b"], 1))   # [[1, "a"], [2, "b"]]
```

> **Not:** Cobra'da çoklu atama ve açma (unpacking) yoktur; bu yüzden `for i, v in enumerate(...)` yazamazsınız. Çiftleri `pair[0]` ve `pair[1]` ile indeksleyin.

6.7.6 `any(list)` ve `all(list)`

`any`, listede en az bir doğru (truthy) değer varsa `true` döner; `all`, tüm değerler doğruysa `true` döner:

```
print(any([0, 0, 1]))   # true
print(any([0, 0, 0]))   # false
print(all([1, 2, 3]))   # true
print(all([1, 0, 3]))   # false
```

> **İpucu:** Liste kavraması (list comprehension) Cobra'da yoktur. `[x * 2 for x in xs]` yerine `map` kullanın; `[x for x in xs if cond]` yerine `filter` kullanın. İkisini birleştirmeniz gerekirse `filter`'ın sonucunu `map`'e besleyin: `map(f, filter(g, xs))`.

6.7.7 Paralel Sürümler: `pmap` ve `pfilter`

Cobra'da **GIL yoktur**, yani gerçek çok çekirdekli paralellik mümkündür. `map` ve `filter`'ın paralel kardeşleri olan `pmap` ve `pfilter`, işi CPU çekirdeklerine dağıtır. Çağrı biçimi `map`/`filter` ile aynıdır; sonuçlar giriş sırasını korur ve ilk hata aynen yayılır:

```
def heavy(n)
    return n * n
end

print(pmap(heavy, [1, 2, 3, 4]))   # [1, 4, 9, 16]  (paralel, sıra korunur)
```

CPU-yoğun işlerde `pmap`, fazladan hiçbir sözdizimi olmadan hız kazandırır. Paralelliğin ayrıntılarını — `async def`, `await`, `Mutex`, `Channel` — Bölüm 10'da derinlemesine ele alacağız.

6.8 Dekoratörler

Dekoratör, bir fonksiyonu dönüştüren veya kaydeden bir sarmalayıcıdır. Bir `def`'in hemen üstüne bir veya daha çok `@expr` satırı yazarak uygulanır.

En basit hali, `@d`, şu kuralı uygular:

```
@d
def ad(...) ... end
```

Bu, tam olarak şuna eşdeğerdir:

```
ad = d(ad)
```

Yani Cobra fonksiyonu tanımlar, ardından onu `d`'ye geçirir ve `d`'nin döndürdüğü değeri aynı isme yeniden bağlar. Bir örnekle bir fonksiyonu zamanlayan basit bir dekoratör yazalım:

```
import "time"

def timed(fn)
    def wrapper(x)
        let start = time.clock()
        let result = fn(x)
        let elapsed = time.clock() - start
        print("süre: " + str(elapsed) + "s")
        return result
    end
    return wrapper
end

@timed
def slow_square(n)
    return n * n
end

print(slow_square(9))
# süre: 0.000...s
# 81
```

Burada `slow_square`, tanımlandıktan sonra `timed(slow_square)`'ın döndürdüğü `wrapper` ile değiştirilmiştir. Artık `slow_square(9)` çağrısı aslında `wrapper(9)`'u çalıştırır.

6.8.1 Argümanlı Dekoratörler: `@d(args)`

Bir dekoratöre yapılandırma vermek isterseniz `@d(args)` biçimini kullanırsınız. Bu durumda Cobra önce `d(args)`'ı çağırır; sonuç (gerçek dekoratör olan bir fonksiyon) `def`'e uygulanır. Yani `@d(args)` şuna eşdeğerdir: `ad = d(args)(ad)`.

```
def repeat(times)
    def decorator(fn)
        def wrapper(x)
            let result = null
            let i = 0
            while i < times
                result = fn(x)
                i = i + 1
            end
            return result
        end
        return wrapper
    end
    return decorator
end

@repeat(3)
def shout(msg)
    print(msg + "!")
    return msg
end

shout("merhaba")
# merhaba!
# merhaba!
# merhaba!
```

`repeat(3)`, `decorator`'ı döndürür; `decorator` da `shout`'u sarmalar.

6.8.2 İstifleme (Stacking)

Bir `def`'in üstüne birden çok dekoratör koyabilirsiniz. Bunlar **istiflenir** ve uygulama sırası şu kuralı izler: **`def`'e en yakın olan önce uygulanır.** Yani:

```
@a
@b
def f(...) ... end
```

şuna eşdeğerdir: `f = a(b(f))` — önce `b`, sonra `a`. En içteki sarmalama `def`'e en yakın dekoratördür.

```
def upper(fn)
    def wrapper(s)
```

```
        return fn(s).upper()
    end
    return wrapper
end

def exclaim(fn)
    def wrapper(s)
        return fn(s) + "!"
    end
    return wrapper
end

@upper
@exclaim
def echo(s)
    return s
end

print(echo("merhaba"))   # MERHABA!
```

Burada önce `exclaim` (en yakın) uygulanır, `"merhaba!"` üretir; sonra `upper` onu sarmalayıp `"MERHABA!"` yapar.

6.8.3 Dekoratörler `async def` ile de Çalışır

Dekoratörler eşzamansız fonksiyonlara da uygulanabilir. Bir `async def`'in üstüne dekoratör koyduğunuzda, sarmalama kuralları aynen geçerlidir (eşzamanlılığın ayrıntıları Bölüm 10'da).

6.8.4 Kayıt (Registry) Deseni

Dekoratörlerin en sık karşılaşılan kullanımlarından biri, fonksiyonları bir kayıt tablosuna **kaydetmektir** — sarmalamadan, fonksiyonu olduğu gibi döndürerek. Web yönlendirmesi (routing) bunun klasik örneğidir:

```
let routes = {}

def route(path)
    def register(fn)
        routes[path] = fn
        return fn          # fonksiyonu değiştirmeden döndür
    end
    return register
end

@route("/home")
def home(req)
    return "hoş geldiniz"
end

@route("/about")
def about(req)
    return "hakkımızda"
end
```

```
# Şimdi routes, yol -> fonksiyon eşlemesini içerir:
print(routes["/home"]("istek"))    # hoş geldiniz
print(routes["/about"]("istek"))   # hakkımızda
```

Burada `route("/home")`, `register`'ı döndürür; `register` ise `home`'u `routes` sözlüğüne ekler ve onu **değiştirmeden** geri verir. Sonuçta `home` hâlâ normal bir fonksiyondur, ama aynı zamanda `routes` tablosuna da girmiştir. Bu desen, bir HTTP sunucusunun istek yollarını ilgili işleyicilere bağlamanın deyimsel (idiomatic) yoludur.

> **İpucu:** Bir dekoratörün fonksiyonu *sarmalaması* zorunlu değildir. Kayıt deseninde olduğu gibi, dekoratör fonksiyonu sadece bir yerde kaydedip aynen döndürebilir. Önemli olan tek kural: dekoratör, `def`'in yeni değeri olarak bağlanacak bir değer döndürmelidir.

6.9 Özet

Bu bölümde Cobra fonksiyonlarının tüm yüzlerini gördük:

- Fonksiyonlar `def ad(parametreler)` ile başlar, `end` ile kapanır ve **gövde her zaman `def` satırından sonraki satırlarda** yer alır — satır içi gövde yoktur. `func`, `def` ile eş anlamlıdır.
- `return` bir değer döndürür ve fonksiyonu sonlandırır. `return` olmayan bir fonksiyon `null` döndürür; çıplak `return` da erken çıkış yapıp `null` döndürür.
- Fonksiyonlar **birinci sınıf değerlerdir**: değişkenlere atanabilir, listelerde/sözlüklerde saklanabilir, geçirilebilir ve döndürülebilirler.
- Fonksiyonlar tanımlandıkları kapsamı **kapatır** (closure); iç içe tanımlanan bir fonksiyon, dıştaki değişkenleri yaşatır ve değiştirebilir.
- Fonksiyonlar **tam olarak bildirilen sayıda konumsal argüman** alır. Varsayılan değer, anahtar-kelime argüman ve `*args`/`**kwargs` **yoktur**.
- **`lambda` yoktur**; satır içi fonksiyon için iç içe bir `def` yazıp döndürün.
- Yüksek-mertebeli yerleşikler — `map`, `filter`, `reduce`, `zip`, `enumerate`, `any`, `all` — **hepsi EAGER'dır ve liste döndürür**. `reduce` soldan katlar (`fn(acc, elem)`). Paralel sürümler `pmap` ve `pfilter` GIL'siz gerçek paralellik sunar.
- **Dekoratörler** bir `def`'in üstündeki `@expr` satırlarıdır: `@d` → `ad = d(ad)`; `@d(args)` önce `d(args)`'ı çağırır. İstiflenirler (en yakın olan önce uygulanır) ve `async def` ile de çalışırlar. Kayıt deseni (`@route("/yol")`), fonksiyonları sarmalamadan kaydetmenin deyimsel yoludur.

Bir sonraki bölümde, durum ve davranışı bir arada paketlemenin yolu olan **struct**'lara (basit sınıflara) geçeceğiz — fonksiyonların metot olarak nasıl yaşadığını orada göreceksiniz.

Bölüm 7

Nesne Yönelimli Programlama

Önceki bölümde fonksiyonların durumu kapanışlar (closure) aracılığıyla nasıl taşıdığını gördük. Ancak çoğu programda durum ve davranış birlikte yaşar: bir banka hesabının bir bakiyesi *ve* o bakiyeyi değiştiren `deposit`/`withdraw` işlemleri vardır; bir geometrik şeklin boyutları *ve* bir `area` hesabı vardır. Bu ikisini tek bir paket altında toplamanın yolu **nesne yönelimli programlamadır**.

Cobra bu işi sade bir yapı taşıyla çözer: **`struct`**. Cobra'nın felsefesi az sayıda kavramın tutarlı davranmasıdır; bu yüzden burada onlarca anahtar kelime yerine tek bir temel fikir bulacaksınız — durumu (alanlar) ve davranışı (metotlar) bir arada tutan bir tip. Bu bölümde kurucuyu (`init`) ve örtük `self`'i, alan erişimini, örnek metotlarını, özellikleri (`get`/`set`), sınıf sabitlerini (`const`), statik metotları (`static def`), tek kalıtımı ve `super`'i, değişmez değer nesneleri için `record`'u ve arayüzlerin yerine geçen çalışma-zamanı sözleşmelerini (`contract` + `implements`) inceleyeceğiz. Cobra'nın bilinçli olarak *eksik bıraktığı* özellikleri de — operatör aşırı yükleme yok, çoklu kalıtım yok, `isinstance` yok — açıkça ele alacağız.

7.1 `struct`, `class`, `record`: Üç Anahtar Kelime

Cobra'da bir tip tanımlamanın üç anahtar kelimesi vardır ve ilk ikisi tam olarak **eş anlamlıdır**:

- **`struct`** — bir basit sınıf tanımlar. Bu kitabın ana anahtar kelimesi budur.
- **`class`** — `struct` ile birebir aynıdır. Başka dillerden gelenler için bir kolaylıktır; davranışta hiçbir fark yoktur.
- **`record`** — **değişmez** (immutable) örnekler üreten ve **değere göre** eşitlik kullanan özel bir tiptir. Bunu bölümün sonunda ayrıca ele alacağız.

```
class Counter
    def init()
        self.value = 0
    end
    def increment()
        self.value += 1
        return self.value
    end
end

let c = Counter()
print(c.increment())   # 1
print(c.increment())   # 2
print(type(c))         # Counter
```

> **Not:** `class` ve `struct` arasında seçim tamamen tarz meselesidir. Bu kitap tutarlılık için `struct` kullanır; ekibiniz `class`'ı tercih ediyorsa hiçbir şey değişmez.

7.2 İlk Yapımız: `struct` ve `init`

Bir `struct`, `struct` anahtar kelimesi, bir tip adı ve `end` ile kapanan bir gövdeden oluşur. Gövde içinde, kurucusu olan özel bir metot bulunur: **`init`**. `init`, tipi bir fonksiyon gibi çağırdığınızda (`BankAccount("Ada", 100)`) otomatik olarak çalışır ve yeni örneği başlatır.

Cobra'da tüm metotların gövdesinde, üzerinde çalışılan örneği temsil eden **örtük bir `self`** vardır. Onu parametre olarak yazmazsınız; her zaman hazırdır. Alanlara `self.alan` ile erişir ve `self.alan = değer` ile yazarsınız.

```
struct BankAccount
    def init(owner, balance)
        self.owner = owner          # alanlar self üzerinde tanımlanır
        self.balance = balance
    end

    def deposit(amount)
        self.balance = self.balance + amount
        return self.balance
    end

    def withdraw(amount)
        if amount > self.balance
            throw "yetersiz bakiye"
        end
        self.balance = self.balance - amount
        return self.balance
    end
end

let acc = BankAccount("Ada", 100)
print(acc.deposit(50))    # 150
print(acc.withdraw(30))   # 120
print(acc.owner)          # Ada
```

`BankAccount("Ada", 100)` çağrısı, `init`'i `owner = "Ada"` ve `balance = 100` ile çalıştırır; geriye yeni bir **örnek** (instance) döner. `deposit` ve `withdraw` metotları `self.balance`'ı okuyup günceller — durum, örneğin içinde yaşar.

> **İpucu:** `init`'in kendisi `return` yapmaz; Cobra otomatik olarak başlatılan örneği döndürür. Görevi yalnızca `self`'in alanlarını kurmaktır.

7.3 Alanlar: Okuma, Yazma ve Yeni Alan Ekleme

Bir örneğin alanlarına nokta ile erişirsiniz: `acc.balance` okur, `acc.balance = 0` yazar. Cobra'da alanlar önceden bir yerde "bildirilmek" zorunda değildir — bir örneğe **dışarıdan, dilediğiniz an yeni bir alan ekleyebilirsiniz**:

```
let acc = BankAccount("Ada", 100)

acc.note = "VIP musteri"     # init'te olmayan yeni bir alan
print(acc.note)              # VIP musteri
```

Bir örneği yazdırdığınızda Cobra onu `TipAdı{alan: değer, ...}` biçiminde gösterir; metin değerler tırnak içinde görünür. `type(örnek)` ise tipin adını bir metin olarak verir:

```
print(acc)         # BankAccount{owner: "Ada", balance: 100, note: "VIP musteri"}
print(type(acc))   # BankAccount
```

> **Uyarı:** Örnekler **kimliğe göre** karşılaştırılır (referans eşitliği): aynı alanlara sahip iki ayrı `BankAccount`, `==` ile eşit değildir. Değere göre eşitlik istiyorsanız `record` kullanın (bölümün sonunda).

7.4 Örnek Metotları

`init` dışındaki her `def`, tipin bir **örnek metodu** olur. Onları bir örnek üzerinden, parantezle çağırırsınız: `acc.deposit(50)`. Metot gövdesi her zaman `self` aracılığıyla örneğin alanlarına ve diğer metotlarına erişebilir. Augmented atama (`+=`) alanlarda da çalışır, bu yüzden `self.balance += amount` yazmak `self.balance = self.balance + amount`'un kısa halidir:

```
struct BankAccount
    def init(owner, balance)
        self.owner = owner
        self.balance = balance
    end

    def deposit(amount)
        self.balance += amount       # self.balance = self.balance + amount
        return self.balance
    end

    def transfer_to(other, amount)
        self.withdraw(amount)        # kendi metodunu self üzerinden çağır
        other.deposit(amount)        # başka bir örneğin metodunu çağır
    end

    def withdraw(amount)
        if amount > self.balance
            throw "yetersiz bakiye"
        end
        self.balance -= amount
        return self.balance
    end
end
```

> **Not:** Bir metodu çağırmak için parantez **zorunludur**: `acc.deposit(50)`. Parantezsiz `acc.deposit` yazarsanız metodu *çağırmaz*, metot değerinin kendisini elde edersiniz. (Az sonra göreceğimiz `get` özellikleri bunun kuralın tek istisnasıdır.)

7.5 Özellikler: `get` ile Hesaplanan Okumalar

Bazen bir değer, depolanmış bir alan değil, başka alanlardan **türetilmiş** bir sonuçtur. Bunu her seferinde bir metot çağrısıyla (`item.subtotal()`) almak yerine, sanki normal bir alanmış gibi (`item.subtotal`) okumak istersiniz. İşte `get` bunu sağlar: gövdesini `self` ile çalıştıran, ama **parantezsiz** okunan bir özellik tanımlar.

```
struct LineItem
    def init(name, unit_price, quantity)
        self.name = name
        self.unit_price = unit_price
        self.quantity = quantity
    end

    get subtotal()                  # özellik: item.subtotal (parantez YOK)
        return self.unit_price * self.quantity
    end
end

let item = LineItem("Klavye", 350, 3)
print(item.subtotal)    # 1050   (parantezsiz okunur)
item.quantity = 5
print(item.subtotal)    # 1750   (her okumada yeniden hesaplanır)
```

`subtotal` her okunduğunda gövde yeniden çalışır; bu yüzden `quantity` değişince sonuç da güncellenir. Dışarıdan bakıldığında sıradan bir alan gibi görünür, ama arkada bir hesaplama yatar.

7.6 Ayarlayıcılar: `set` ve Çift Yönlü Hesaplanan Alanlar

`get`'in eşi `set`'tir: `örnek.ad = değer` yazıldığında çalışan bir **ayarlayıcı**. Atanan değer, parametreye bağlanır. Bir `get`/`set` çiftini birlikte tanımlayarak, arka planda *başka* bir alanla desteklenen, çift yönlü hesaplanan bir alan yaratabilirsiniz. Klasik örnek bir sıcaklık dönüştürücüdür: veriyi tek bir alanda (Celsius) tutar, ama hem Celsius hem Fahrenheit olarak okunup yazılabilir.

```
struct Thermostat
    def init(celsius)
        self._celsius = celsius          # gerçek veri: geri-alan (_ ile)
    end

    get celsius()
        return self._celsius
    end
    set celsius(value)
        self._celsius = value
    end

    get fahrenheit()
        return self._celsius * 9 / 5 + 32
    end
    set fahrenheit(value)
        self._celsius = (value - 32) * 5 / 9
    end
```

```
end

let t = Thermostat(20)
print(t.celsius)        # 20
print(t.fahrenheit)     # 68

t.fahrenheit = 212      # setter çalışır: _celsius = 100
print(t.celsius)        # 100
print(t.fahrenheit)     # 212

t.celsius = 0
print(t.fahrenheit)     # 32
```

Burada dikkat edin: `fahrenheit` adında *depolanan bir alan yoktur*. Yalnızca `_celsius` saklanır; `fahrenheit` her okunup yazıldığında ona çevrilir. Öndeki alt çizgi (`_celsius`) Cobra'da özel bir anlam taşımaz — yalnızca "bu, içsel geri-alandır, doğrudan dokunmayın" diyen yaygın bir adlandırma kuralıdır.

> **Uyarı:** Bir ayarlayıcı, **kendi adıyla aynı alana yazmamalıdır**, yoksa sonsuz özyinelemeye girer. Şu kod `set celsius` içinde yine `self.celsius`'a yazdığı için ayarlayıcıyı tekrar tetikler ve yığını taşırır (stack overflow):
>
> ```
> set celsius(value)
>     self.celsius = value     # YANLIŞ: setter'ı yeniden çağırır -> sonsuz döngü
> end
> ```
>
> Doğrusu, `set celsius`'un `self._celsius` gibi *farklı* bir geri-alana yazmasıdır. Kural basittir: bir özellik ile onu destekleyen alanın adı farklı olmalıdır.

Getter ve setter'lar kalıtımla alt tiplere de geçer (kalıtımı birazdan göreceğiz).

7.7 Sınıf Sabitleri: `const`

Bir `struct` gövdesi içinde `const`, o tipe ait bir **sınıf sabiti** tanımlar. Belirli bir örneğe değil, tipin kendisine aittir ve `Tip.AD` ile okunur (bir örnek üzerinden, `örnek.AD`, de okunabilir). Tipe bağlı sabit değerler — bir fizik sabiti, varsayılan bir eşik, bir dönüşüm katsayısı — için idealdir.

```
struct Temperature
    const ABSOLUTE_ZERO_C = -273
    const FREEZING_C = 0

    def init(celsius)
        self.celsius = celsius
    end
end

print(Temperature.ABSOLUTE_ZERO_C)   # -273   (tip üzerinden)

let ice = Temperature(0)
print(ice.ABSOLUTE_ZERO_C)           # -273   (örnek üzerinden de okunur)
```

> **Not:** Bir `struct` *dışında* `const`, sıradan bir değişmez değişken bağlaması yapar (Bölüm 2). Bir `struct` *içinde* ise anlamı değişir ve bir sınıf sabiti olur. Bağlam belirleyicidir.

7.8 Statik Metotlar: `static def`

`static def`, bir örneğe değil **tipin kendisine** ait bir metot tanımlar. `self`'i yoktur ve `Tip.metot(...)` biçiminde çağrılır. En yaygın kullanımı, nesneyi farklı girdilerden üreten **fabrika metotlarıdır** (alternatif kurucular). `init`'i bir Celsius değeri beklerken, bir Fahrenheit değerinden örnek üreten bir yardımcı yazalım:

```
struct Temperature
    const FREEZING_C = 0

    def init(celsius)
        self.celsius = celsius
    end

    static def from_fahrenheit(f)        # fabrika: self yok
        return Temperature((f - 32) * 5 / 9)
    end

    static def freezing()
        return Temperature(Temperature.FREEZING_C)
    end
end

let boiling = Temperature.from_fahrenheit(212)
print(boiling.celsius)   # 100

let ice = Temperature.freezing()
print(ice.celsius)       # 0
```

Statik metotlar `self` taşımadığı için yalnızca argümanlarıyla ve sınıf sabitleriyle çalışır. Bir örnek üzerinde değil, doğrudan tip üzerinde düşünmenin doğal yoludur.

7.9 Kalıtım: `struct Dog < Animal`

Cobra **tek kalıtımı** destekler: bir tip, `<` ile başka bir tipi genişletebilir. Alt tip, üst tipin tüm metotlarını, özelliklerini ve sınıf sabitlerini devralır; istediğini **geçersiz kılabilir** (override) ve üst sürüme `super` ile erişebilir:

- `super.init(...)` — üst tipin kurucusunu çağırır (genellikle alt tipin `init`'inin ilk satırı).
- `super.metot(...)` — geçersiz kıldığınız bir metodun üst sürümünü çağırır.

Bir geometri örneğiyle görelim: ortak bir `Shape` tabanı ve onu genişleten `Circle` ile `Rectangle`. Her ikisi de `area`'yı kendine göre tanımlar.

```
struct Shape
    def init(name)
        self.name = name
    end

    def area()
        return 0                 # taban: alt tipler geçersiz kılar
    end
```

```
    def describe()
        return self.name + " alani: " + str(self.area())
    end
end

struct Circle < Shape
    const PI = 3.14159

    def init(radius)
        super.init("Daire")      # üst kurucuyu çağır
        self.radius = radius
    end

    def area()                   # geçersiz kılma
        return Circle.PI * self.radius * self.radius
    end
end

struct Rectangle < Shape
    def init(width, height)
        super.init("Dikdortgen")
        self.width = width
        self.height = height
    end

    def area()
        return self.width * self.height
    end

    def describe()               # geçersiz kıl ve üst sürümü genişlet
        return super.describe() + " (" + str(self.width) + "x" + str(self.height) + ")"
    end
end

let shapes = [Circle(2), Rectangle(3, 4)]
for s in shapes
    print(s.describe())
end
# Daire alani: 12.56636
# Dikdortgen alani: 12 (3x4)
```

Burada birkaç önemli nokta var. `Shape.describe`, kendi içinde `self.area()` çağırır; ama `self` çalışma zamanında gerçek örnektir (bir `Circle` ya da bir `Rectangle`), bu yüzden o örneğin geçersiz kıldığı `area` çalışır. Buna **polimorfizm** denir: tek bir `describe` kodu, alt tipe göre doğru `area`'yı seçer. `Rectangle.describe` ise `super.describe()` ile taban davranışı çağırıp sonuna kendi ek bilgisini iliştirir.

> **İpucu:** Bir alt tipin `init`'i, üst tipin alanlarına ihtiyaç duyuyorsa `super.init(...)`'i genellikle ilk satırda çağırın. Böylece taban alanlar (`self.name`) kurulduktan sonra alt tipe özel alanları eklersiniz.

7.10 `record`: Değişmez Değer Nesneleri

Bazı tipler bir **kimliği** değil, bir **değeri** temsil eder: bir para tutarı, bir 2B nokta, bir tarih. Bu tür "değer nesneleri" için iki şey isteriz: **değişmezlik** (bir kez kurulduktan sonra alanları değişmesin) ve **değere göre eşitlik** (aynı içerikteki iki tutar eşit sayılsın). `record` tam olarak bunu verir.

Bir `record`, `struct` gibi yazılır ama üç farkla:

1. Örnekleri **dondurulmuştur**: kurulduktan sonra bir alana yazmak hatadır.
2. ==, alan alan **değer** karşılaştırması yapar (kimlik değil).
3. Her örneğin, bazı alanları değiştirilmiş bir **kopyasını** üreten bir `.with(dict)` metodu vardır.

```
record Money
    def init(amount, currency)
        self.amount = amount
        self.currency = currency
    end

    def plus(other)
        if other.currency != self.currency
            throw "para birimleri uyusmuyor"
        end
        return Money(self.amount + other.amount, self.currency)
    end
end

let price = Money(100, "TRY")
let same  = Money(100, "TRY")
let other = Money(250, "TRY")

print(price == same)    # true    (değere göre eşitlik)
print(price == other)   # false

# .with -> bazı alanları değişmiş bir KOPYA (özgün nesne değişmez)
let raised = price.with({"amount": 150})
print(raised.amount)    # 150
print(raised.currency)  # TRY
print(price.amount)     # 100     (özgün değişmedi)

let total = price.plus(other)
print(total.amount)     # 350
print(total)            # Money{amount: 350, currency: "TRY"}
print(type(price))      # Money
```

Bir `record` değişmez olduğu için alanlarını "güncellemenin" tek yolu, yeni bir kopya üretmektir. `.with(dict)`, sözlükteki adlandırılmış alanları değiştirir, gerisini olduğu gibi kopyalar ve sonucu yine dondurur. Bir alanı doğrudan değiştirmeye çalışmak hata verir:

```
record Point
    def init(x, y)
        self.x = x
        self.y = y
    end
```

```
end

let p = Point(1, 2)
try
    p.x = 99
catch err
    print("hata:", err)    # hata: line 9: cannot modify immutable record Point
end
print(p.x)                 # 1   (değişmedi)
```

> **İpucu:** Bir tipin yalnızca verisini taşıdığını, hesaplama ya da değişen durum içermediğini düşünüyorsanız `record` seçin. Değere göre eşitlik ve değişmezlik, bu nesneleri sözlük anahtarı gibi düşünmeyi ve eşzamanlı kod içinde güvenle paylaşmayı kolaylaştırır.

7.11 `contract` ve `implements`: Çalışma-Zamanı Sözleşmeleri

Çoğu dilde, "bu tip şu metotlara sahip olmalı" güvencesini **arayüzler** (interface) verir. Cobra'da arayüz yoktur; yerine, ördek-tiplemeli (duck-typed) bir çalışma-zamanı kontrolü olan **`contract`** vardır. Bir sözleşme, gerekli metot adlarını her satıra bir tane yazarak listeler:

```
contract Serializable
    to_dict
    from_dict
end
```

`implements(değer, Sözleşme)`, bir değerin sözleşmedeki tüm metotlara sahip olup olmadığını çalışma zamanında kontrol eder ve `true`/`false` döndürür. Bu, "şu tipten mi?" diye sormak yerine "şunu yapabiliyor mu?" diye sormaktır — ki nesne yönelimli tasarımda genellikle daha sağlamı budur.

```
struct User
    def init(id, name)
        self.id = id
        self.name = name
    end

    def to_dict()
        return {"id": self.id, "name": self.name}
    end

    def from_dict(data)
        return User(data["id"], data["name"])
    end
end

struct Tag                         # to_dict var, ama from_dict YOK
    def init(label)
        self.label = label
    end
    def to_dict()
        return {"label": self.label}
    end
```

```
end

let u = User(1, "Ada")
print(implements(u, Serializable))            # true
print(implements(Tag("urgent"), Serializable)) # false   (from_dict eksik)

def save(value)
    if not implements(value, Serializable)
        throw "kaydedilemez: Serializable degil"
    end
    return value.to_dict()
end

print(save(u))    # {"id": 1, "name": "Ada"}
```

Burada `save`, kendisine geçilen değerin somut tipini umursamaz; yalnızca `Serializable` sözleşmesini karşılamasını ister. `User` ve gelecekte yazacağınız herhangi bir tip — `to_dict` ve `from_dict` sunduğu sürece — sorunsuz çalışır. `Tag` ise `from_dict`'i olmadığı için sözleşmeyi karşılamaz ve kontrol bunu yakalar. Aynı desen bir `Repository` sözleşmesi (`save`, `find`, `delete`) için de aynen geçerlidir: depolama katmanının somut sınıfını bağlamadan, yalnızca yetkinliğine güvenirsiniz.

> **Not:** `implements`, Cobra'nın `isinstance`/tip-kontrolü eksikliğine verdiği deyimsel yanıttır. "Bu nesne falanca tipten mi?" sorusunu sormak yerine "gereken metotlara sahip mi?" sorusunu sorarsınız — duck typing'in ruhuna uygun, gevşek bağlı bir tasarım.

7.12 Cobra'da Olmayanlar

Cobra'nın nesne modeli bilinçli olarak küçüktür. Başka dillerden tanıdığınız ve Cobra'da **bulunmayan** bazı özellikleri bilmek, geçersiz kod yazmaktan korur:

- **Operatör aşırı yükleme / dunder metotları yoktur.** `__eq__`, `__str__`, `__add__` gibi özel metotlar tanımlayamazsınız; `+` veya `==`'in kendi tipiniz için davranışını değiştiremezsiniz. (Bir istisna: `record`, `==` için değere göre eşitliği *yerleşik olarak* sağlar.) İki nesneyi toplamak için `a.plus(b)` gibi açık bir metot yazın.
- **Çoklu kalıtım yoktur.** Bir tip yalnızca tek bir üst tipi genişletebilir (`struct Dog < Animal`). Birden çok tabandan davranış paylaşmak isterseniz kompozisyon kullanın (bir alanda başka bir nesne tutun) veya ortak yeteneği bir `contract` ile ifade edin.
- **`isinstance` benzeri bir tip kontrolü yoktur.** "Bu nesne şu tipten mi?" diye sormak yerine `implements(değer, Sözleşme)` ile "şu metotlara sahip mi?" diye sorun. Hızlı bir tip adı karşılaştırması gerekiyorsa `type(x)` bir metin döndürür (`type(u) == "User"`), ama esnek tasarım için `contract` yeğlenir.

> **İpucu:** Bu eksiklikler bir kısıtlama gibi görünse de, çoğu zaman daha açık kod üretirler: gizli operatör davranışları yerine adlandırılmış metotlar, karmaşık kalıtım hiyerarşileri yerine sade kompozisyon ve sözleşmeler.

7.13 Özet

Bu bölümde Cobra'nın nesne yönelimli yüzünü baştan sona gördük:

- **`struct`** (ve eş anlamlısı **`class`**) durumu ve davranışı bir araya getiren basit bir sınıf tanımlar. **`init`** kurucudur ve her metotta **örtük `self`** vardır.
- Alanlara `p.x` ile erişir, `p.x = v` ile yazarsınız; örneklere **dilediğiniz an yeni alan** ekleyebilirsiniz. Örnekler `Tip{alan: değer}` biçiminde yazılır, **kimliğe göre** karşılaştırılır ve `type(örnek)` tip adını verir.
- Örnek metotları parantezle çağrılır (`acc.deposit(50)`). **`get ad()`** bir özelliği parantezsiz okunur kılar; **`set ad(v)`** atamada çalışır. Bir `get`/`set` çifti, içsel bir geri-alanla (`self._x`) desteklenen hesaplanan bir alan üretir — ayarlayıcı **farklı** bir alana yazmalıdır, yoksa sonsuz döngüye girer.
- **`const AD`**, bir `struct` içinde `Tip.AD` ile okunan bir sınıf sabiti tanımlar. **`static def`** bir örneğe değil tipe ait, `self`'siz bir metot tanımlar (`Tip.metot()`) — fabrika metotları için idealdir.
- **Tek kalıtım** `struct Alt < Ust` ile yapılır; **`super.init(...)`** ve **`super.metot(...)`** üst sürümlere erişir. Metotları geçersiz kılmak polimorfizmi mümkün kılar.
- **`record`** değişmez, **değere göre eşit** örnekler üretir; **`.with(dict)`** bazı alanları değişmiş bir kopya döndürür. Değer nesneleri için doğru araçtır.
- **`contract Ad ... end`** gerekli metot adlarını listeler; **`implements(değer, Sözleşme)`** bunları çalışma zamanında kontrol eder — arayüzlerin yerine geçen, duck-typed bir çözüm.
- Cobra'da **operatör aşırı yükleme/dunder metotları, çoklu kalıtım ve `isinstance` yoktur**; bunların yerine açık metotlar, kompozisyon ve sözleşmeler kullanılır.

Bir sonraki bölümde, programlar büyüdükçe kaçınılmaz olan **hataları** ele almaya — `try`/`catch`/`finally` ve `throw` ile sağlam, kurtarılabilir kod yazmaya — geçeceğiz.

Bölüm 8

Hata Yönetimi

Gerçek programlar nadiren ideal koşullarda çalışır. Kullanıcı boş bir form gönderir, bir dosya yarıda kalmış olur, bir HTTP isteği zaman aşımına uğrar, gelen JSON bozuktur. İyi yazılmış bir program bu durumlarda çökmek yerine onları *beklenen* birer olay gibi karşılar: hatayı yakalar, açık bir mesaj üretir, açtığı kaynakları kapatır ve mümkünse yeniden dener.

Cobra'nın hata yönetimi bilinçli olarak küçüktür. Tek bir blok yapısı (`try ... catch ... finally ... end`) ve tek bir komut (`throw`) vardır. Bazı dillerdeki gibi onlarca *exception* sınıfı ya da türe göre ayrı `catch` blokları yoktur. Bunun yerine: her değer atılabilir, `catch` her şeyi yakalar ve hatayı ayırt etmeyi (eğer gerekiyorsa) siz yaparsınız. Bu bölümde önce temel yapıyı, sonra hata değerlerinin nasıl tasarlanacağını, `finally`'nin garantisini ve hataların çağrı yığınında nasıl yayıldığını mühendislik senaryoları üzerinden göreceğiz.

8.1 `try / catch / finally`: Temel Yapı

Bir `try` bloğu, "başarısız olabilecek" kodu sarmalar. Hata oluşursa Cobra normal akışı durdurur ve kontrolü `catch` bloğuna geçirir. `finally` bloğu ise —ister hata olsun ister olmasın— **her zaman** çalışır.

```
try
    let parts = "a,b,c".split(",")
    print(parts[5])        # aralık dışı -> çalışma-zamanı hatası
catch err
    print("yakalandı:", err)
finally
    print("bitti")
end
```

Çıktı:

```
yakalandı: line 4: list index out of range: 5
bitti
```

Burada `parts[5]` geçersiz bir indeks olduğu için Cobra bir çalışma-zamanı hatası üretir; `print(parts[5])` hiç çalışmaz, akış doğrudan `catch err` bloğuna atlar. `err`, yakalanan hata değeridir (birazdan bunun ne tür bir değer olduğunu göreceğiz). En sonda `finally` bloğu çalışır.

Yapının söz dizimi şöyledir:

```
try
    ...                 # korunan kod
catch err               # 'err' opsiyoneldir
    ...                 # hata varsa çalışır
finally
    ...                 # her durumda çalışır
end
```

İki kuralı şimdiden aklınızda tutun:

- **En az `catch` ya da `finally` bulunmalıdır.** İçi boş bir `try ... end` geçersizdir.
- `catch` sonrası **değişken adı opsiyoneldir.** Hatanın değerini kullanmayacaksanız yalnızca `catch` yazabilirsiniz.

İçi boş bir `try` yazmaya çalışırsanız hata ayrıştırma (parse) aşamasında, yani program daha çalışmadan yakalanır:

```
try
    print("hi")
end
```

```
parse error: line 2: try requires catch or finally
```

`catch`'i ad bağlamadan da kullanabilirsiniz; hatanın ayrıntısı umurunuzda değilse bu yeterlidir:

```
try
    throw "ignored detail"
catch
    print("something failed")
end
```

```
something failed
```

> **Not:** `catch` bloğu, hangi hata oluşursa oluşsun çalışır. Cobra'da "şu tür hatayı yakala, bu tür hatayı yakalama" ayrımı yoktur. Hatayı ayırt etmek isterseniz bunu `catch` bloğunun *içinde* `type(err)` veya bir alan kontrolüyle kendiniz yaparsınız.

8.2 `throw`: Her Değer Atılabilir

Kendi hatalarınızı `throw` ile fırlatırsınız. Cobra'nın diğer dillerden ayrıldığı en önemli nokta burasıdır: **atılan değer herhangi bir tür olabilir** — string, integer, boolean, dict, list… Atılan değer ne ise, `catch` onu aynen yakalar.

```
# her değer atılabilir: integer, string, dict...
for v in [42, "metin", true]
    try
        throw v
    catch err
        print(type(err), "->", err)
    end
end
```

```
INTEGER -> 42
STRING -> metin
BOOLEAN -> true
```

En basit durumda bir string atmak yeterlidir. Ama gerçek bir sistemde hatanın yalnızca *mesajı* değil, **makine tarafından işlenebilir** ek bilgileri de olur: bir hata kodu, hangi alanın hatalı olduğu, belki HTTP durum kodu. Bu yüzden profesyonel kodda hatalar genellikle **dict** olarak atılır:

```
throw {"code": 404, "message": "kayıt bulunamadı"}
```

Bunu yakalayan taraf hem kullanıcıya mesaj gösterebilir hem de koda göre karar verebilir:

```
try
    throw {"code": 404, "message": "not found"}
catch err
    print(type(err))       # DICT
    print(err["code"])     # 404
    print(err["message"])  # not found
end
```

```
DICT
404
not found
```

8.2.1 Mühendislik Örneği: Girdi Doğrulama

Yapısal hata değerlerinin gerçek faydası, bir doğrulama (validation) katmanında ortaya çıkar. Aşağıdaki `validate_user`, kullanıcıdan gelen bir formu denetler ve ilk kuralı çiğneyen alanda `{"code", "field", "message"}` biçiminde bir hata atar. Hata değerini yapısal tutmak, çağıran tarafa "hangi alan, neden" bilgisini verir:

```
# --- Kullanıcı girdisi doğrulama, yapısal hata değerleriyle ---

# Doğrulama başarısızsa {"code", "field", "message"} biçiminde bir hata atar.
def validate_user(form)
    if not form.has("name") or form["name"].strip() == ""
        throw {"code": "EMPTY", "field": "name", "message": "isim zorunludur"}
    end
    if not form.has("age")
        throw {"code": "MISSING", "field": "age", "message": "yaş zorunludur"}
    end
    let age = int(form["age"])     # sayı değilse çalışma-zamanı hatası atar
    if age < 0 or age > 130
        throw {"code": "RANGE", "field": "age", "message": "yaş 0-130 aralığında olmalı"}
    end
    return {"name": form["name"].strip(), "age": age}
end

let forms = [
    {"name": "Ada", "age": "36"},
    {"name": "  ", "age": "20"},
    {"name": "Bob"},
```

```
    {"name": "Eve", "age": "200"},
    {"name": "Cem", "age": "otuz"}
]

for form in forms
    try
        let user = validate_user(form)
        print("OK   ->", user["name"] + ", " + str(user["age"]))
    catch err
        if type(err) == "DICT"
            print("HATA -> alan=" + err["field"] + " kod=" + err["code"] + ": " + err["message"])
        else
            # int() gibi çalışma-zamanı hataları düz string olarak gelir
            print("HATA -> " + err)
        end
    end
end
```

```
OK   -> Ada, 36
HATA -> alan=name kod=EMPTY: isim zorunludur
HATA -> alan=age kod=MISSING: yaş zorunludur
HATA -> alan=age kod=RANGE: yaş 0–130 aralığında olmalı
HATA -> line 11: cannot convert "otuz" to int
```

Son satıra dikkat edin: `int("otuz")` bizim attığımız bir hata değil, **Cobra'nın ürettiği bir çalışma-zamanı hatasıdır** ve bir dict değil, düz bir string olarak gelir. `catch` bloğunda `type(err) == "DICT"` kontrolü tam da bu iki durumu birbirinden ayırmamızı sağlar. Bir sonraki bölüm bu ayrımı netleştiriyor.

8.3 Çalışma-Zamanı Hataları Mesaj String'i Olarak Yakalanır

Sıfıra bölme, aralık dışı indeks, eksik dict anahtarı, tür dönüşüm hatası… Bunlar sizin `throw` ile attığınız değerler değildir; bunları **Cobra'nın çalışma zamanı** üretir. `catch` ile yakalandıklarında, değerleri o hatanın **mesaj string'idir**:

```
try
    let x = 10 / 0
catch err
    print(err)         # line 2: division by zero
    print(type(err))   # STRING
end
```

```
line 2: division by zero
STRING
```

Yani `err` burada bir nesne ya da özel bir "Exception" türü değil, sadece `"line 2: division by zero"` string'idir. Bu, Cobra'nın "her şey sade bir değerdir" felsefesinin doğrudan sonucudur ve pratikte çok kullanışlıdır: hatayı loglamak, kullanıcıya göstermek ya da içinde bir alt dize aramak için ekstra bir API öğrenmenize gerek kalmaz; metin metindir.

Bu davranış, dış dünyadan veri okuyan modüllerle birleştiğinde tam gücünü gösterir. Örneğin `json.parse` bozuk bir girdiyle karşılaşırsa fırlattığı hata da yine bir string'dir:

```
import "json"

let bad = "{ \"name\": \"ada\", }"
try
    let data = json.parse(bad)
    print(data)
catch err
    print("parse failed:", err)
end
```

```
parse failed: line 5: json.parse: invalid character '}' looking for beginning of object key string
```

> **İpucu:** Çalışma-zamanı hataları string, sizin yapısal hatalarınız ise genellikle dict olduğundan, bir `catch` bloğunda `type(err)` ile bu ikisini ayırt etmek yaygın ve sağlam bir kalıptır. Yukarıdaki doğrulama örneğinde tam olarak bunu yaptık.

8.4 Exception Türleri Yoktur

Birçok dilde hatalar bir sınıf hiyerarşisi oluşturur ve `catch (IOException e)` gibi yalnızca belirli bir türü yakalarsınız. **Cobra'da böyle bir şey yoktur.** Tek bir `catch` vardır ve o, ne atılmışsa onu yakalar. Avantajı sadeliktir: öğrenilecek tür hiyerarşisi, doğru `except` sırasını bulma derdi yoktur. Bedeli ise, "yalnızca şu hatayı yakala, ötekini yukarı bırak" davranışını kendinizin yazması gerektiğidir.

Bu davranışı `catch` içinde bir koşul + yeniden atma (`throw`) ile elde edersiniz:

```
def parse_port(s)
    try
        return int(s)
    catch err
        # Sadece dönüşüm hatasını ele alıyoruz; başka her şeyi yukarı bırak.
        throw {"code": "BAD_PORT", "message": "geçersiz port: " + s}
    end
end

try
    print(parse_port("8080"))   # 8080
    print(parse_port("xx"))     # buradan dict hata atılır
catch err
    print(err["code"] + ": " + err["message"])
end
```

```
8080
BAD_PORT: geçersiz port: xx
```

> **Uyarı:** `catch` her şeyi yakaladığı için, gerçekten ele alabildiğiniz hataları yakalamaya özen gösterin. Geniş bir `catch` içinde sadece `print` yapıp hatayı yutmak, programdaki gerçek bir kusuru gizleyebilir. Ele alamadığınız bir hatayı `throw err` ile bir üst katmana bırakmak çoğu zaman doğru olandır.

8.5 `finally` Her Zaman Çalışır

`finally` bloğunun tek bir görevi vardır ve onu kusursuz yapar: `try`'dan çıkış nasıl olursa olsun çalışmak. Normal bitişte, `catch` ile yakalanan bir hatada, hatta `return`/`break`/`continue` ile erken çıkışta bile `finally` çalışır. Bu yüzden `finally`, **kaynak temizliği** için biçilmiş kaftandır: açtığınız dosyayı, kilidi, bağlantıyı buraya kapatırsınız.

8.5.1 Mühendislik Örneği: Kaynak Temizliği

Aşağıdaki fonksiyon geçici bir dosyaya yazar, içeriğini işler ve `finally` içinde dosyayı **mutlaka** siler — işlem başarılı olsa da, ortada bir hata oluşsa da:

```
import "fs"

# --- Dosya/kaynak temizliği: finally her durumda çalışır ---

# Geçici bir dosyaya yazar, işler, sonra finally içinde MUTLAKA siler.
def process_report(path, lines)
    fs.write_lines(path, lines)
    try
        let total = 0
        for line in fs.read_lines(path)
            total += int(line)     # bozuk satırda çalışma-zamanı hatası atar
        end
        return total
    finally
        # Hata olsa da olmasa da geçici dosya temizlenir.
        fs.remove(path)
        print("temizlendi:", path, "var mı?", fs.exists(path))
    end
end

let tmp = "/private/tmp/cobra_rapor.txt"

# 1) Başarılı durum
print("toplam:", process_report(tmp, ["10", "20", "12"]))

# 2) Hatalı durum: dosya yine de silinmeli, hata yukarı yayılmalı
try
    process_report(tmp, ["5", "x", "9"])
catch err
    print("işlem başarısız:", err)
end
```

```
temizlendi: /private/tmp/cobra_rapor.txt var mı? false
toplam: 42
temizlendi: /private/tmp/cobra_rapor.txt var mı? false
işlem başarısız: line 11: cannot convert "x" to int
```

Çıktının sırasına dikkat edin. İlk çağrıda işlem başarılı olur; yine de "toplam: 42" yazdırılmadan **önce** `finally` çalışıp dosyayı siler. İkinci çağrıda `int("x")` hata atar; `return` hiç çalışmaz, ama `finally` yine de devreye girip dosyayı temizler — *ardından* hata, `process_report`'un dışındaki `try`'a yayılarak yakalanır. Yani `finally`, hatayı yutmaz; sadece akış o noktadan ayrılmadan önce işini görür.

8.5.2 `finally` İçinde Atılan Hata Kazanır

Peki ya hem `try` (ya da `catch`) bir hata atmışken, hem de `finally` *kendi* hatasını atarsa? Cobra'da kural nettir: **`finally` içindeki hata kazanır** ve ilk hatanın yerini alır.

```
try
    try
        throw "original"
    finally
        throw "from finally"
    end
catch err
    print(err)
end
```

```
from finally
```

`"original"` hatası `finally` içindeki `throw` tarafından gölgelenir ve dış `catch`'e ulaşan değer `"from finally"` olur.

> **Uyarı:** Bu, `finally` içinde hata atmaktan kaçınmak için iyi bir nedendir. Bir temizlik adımı başarısız olabiliyorsa (örneğin uzaktaki bir bağlantıyı kapatmak), onu kendi `try`'ı içine alın ki asıl hatayı gizlemesin.

8.6 `return`, `break` ve `continue` `try` ve `finally`'den Geçer

`try`/`finally`, akış kontrol komutlarınızın önüne *geçmez*. Bir `return`, `break` veya `continue`, `try` bloğunun içinden tetiklendiğinde önce `finally` çalışır, sonra komut etkisini gösterir. Bir fonksiyondan `try` içinde `return` ederken bile `finally` çalışır:

```
def f()
    try
        print("try")
        return "from try"
    finally
        print("finally")
    end
end
print(f())
```

```
try
finally
from try
```

`return "from try"` çalışmaya başlar; ama fonksiyon değeri döndürmeden önce `finally` bloğu çalışır ("finally" yazdırılır), *ardından* `"from try"` değeri geri verilir.

Aynı şey döngüler içindeki `break` ve `continue` için de geçerlidir:

```
for i in range(4)
    try
        if i % 2 == 0
            continue
        end
        print("tek:", i)
    finally
        print("  finally", i)
    end
end
```

```
  finally 0
tek: 1
  finally 1
  finally 2
tek: 3
  finally 3
```

`i` çift olduğunda `continue` döngünün sonraki adımına atlar — ama atlamadan önce o adımın `finally`'si çalışır. Yani kaynak temizliği, döngü erken kesilse bile güvende kalır.

8.6.1 `finally` İçindeki Yasaklar

`finally` bloğu temizlik içindir; akışı *yönlendiren* veya yeni isimler *tanıtan* ifadeler orada yasaktır. Somut olarak, `finally` içinde `return` ile bildirimler (`let`, `const`, `def`, `struct`…) **ayrıştırma (parse) hatasıdır** — program daha çalışmadan reddedilir.

```
def f()
    try
        return 1
    finally
        return 2
    end
end
print(f())
```

```
parse error: line 6: 'return' not allowed inside finally
```

Bir bildirim de aynı şekilde reddedilir:

```
try
    print("ok")
finally
    let x = 5
    print(x)
end
```

```
parse error: line 5: 'let' not allowed inside finally
```

> **Not:** Bu kısıtlama bilinçlidir. `finally` içinde bir `return` olsaydı, `try`'daki hatayı sessizce yutar ve akışı belirsizleştirirdi. `finally`'yi yalnızca "ne olursa olsun yapılacak temizlik" için kullanın; mantığı `try` ya da `catch` içinde tutun.

8.7 Hataların Çağrı Yığınında Yayılması

Bir hata oluştuğunda Cobra, onu yakalayacak **en içteki (en yakın) `try`**'ı arar. Hata, oluştuğu fonksiyondan çağıranına, oradan onun çağıranına… diye yukarı doğru "kabarcıklanır" (propagation). Yol üzerindeki ilk `try` onu yakalar. Hiçbir `try` yoksa hata programı durdurur.

```
def parse_age(s)
    let n = int(s)        # throws if s isn't a number
    return n
end

try
    print(parse_age("oops"))
catch err
    print("caught:", err)
end
```

```
caught: line 2: cannot convert "oops" to int
```

Hata `parse_age` içinde, 2. satırda doğdu; ama orada `try` yok. Bu yüzden hata `parse_age`'in çağrıldığı yere yayıldı ve oradaki `catch` onu yakaladı. Mesajın hâlâ `line 2` taşıdığına dikkat edin — hata, hangi katmanda yakalanırsa yakalansın, **doğduğu satırın** numarasını taşır.

Eğer hiçbir `try` araya girmezse, hata en üst seviyeye kadar çıkar ve programı sonlandırır:

```
def g()
    throw "boom from g"
end
def h()
    g()
end
print("before")
h()
print("after")
```

```
before
runtime error: line 2: uncaught throw: "boom from g"
```

`"before"` yazdırıldı, ama `h()` içindeki `g()` bir hata attı ve onu yakalayan kimse olmadığı için program durdu — `"after"` hiç çalışmadı. Yakalanmamış bir dict de aynı şekilde, içeriği gösterilerek raporlanır:

```
runtime error: line 2: uncaught throw: {"code": 500, "message": "kaboom"}
```

8.7.1 Mühendislik Örneği: Katmanlar Arası Hata Yayılımı

Gerçek uygulamalar katmanlıdır: en altta ham veriye dokunan bir **depo**, onun üstünde iş kurallarını uygulayan bir **servis**, en üstte de kullanıcıyla konuşan bir **uygulama sınırı**. İyi bir tasarımda alt katman ham hatayı atar; orta katman onu yakalayıp **anlamlı, yapısal bir hataya sarmalar** (orijinalini `cause` olarak saklayarak); en üst katman ise tüm hataları tek bir yerde, kullanıcı diline çevirerek karşılar.

```
import "json"

# --- Katmanlar arası hata yayılımı ---
# Katman 1 (depo): ham veriyi ayrıştırır.
# Katman 2 (servis): iş kurallarını uygular, alt katman hatasını sarmalar.
# Katman 3 (uygulama): kullanıcıya gösterilecek mesaja çevirir.

# Katman 1 — depo
def load_config(raw)
    return json.parse(raw)      # bozuk JSON -> çalışma-zamanı hatası (string)
end

# Katman 2 — servis: hatayı yakalayıp anlamlı, yapısal bir hataya sarmalar
def read_timeout(raw)
    let cfg = null
    try
        cfg = load_config(raw)
    catch err
        throw {"code": "CONFIG_INVALID", "message": "yapılandırma okunamadı", "cause": err}
    end
    if not cfg.has("timeout")
        throw {"code": "CONFIG_MISSING", "message": "timeout alanı yok"}
    end
    return cfg["timeout"]
end

# Katman 3 — uygulama sınırı: tek bir yerde tüm hataları ele alır
def show(raw)
    try
        print("timeout =", read_timeout(raw))
    catch err
        if type(err) == "DICT"
            print("[" + err["code"] + "] " + err["message"])
            if err.has("cause")
                print("    neden:", err["cause"])
            end
        else
            print("beklenmeyen hata:", err)
        end
    end
end

show("{\"timeout\": 30}")          # geçerli
show("{\"timeout\": 30")            # bozuk JSON -> sarmalanır
show("{\"retries\": 3}")           # alan eksik
```

```
timeout = 30
[CONFIG_INVALID] yapılandırma okunamadı
    neden: line 10: json.parse: EOF
[CONFIG_MISSING] timeout alanı yok
```

Bu kalıp, her katmanın "kendi diliyle" hata üretmesini sağlar. Uygulama sınırı, `json.parse`'ın ham mesajıyla uğraşmak zorunda kalmaz; `CONFIG_INVALID` kodunu görür, isterse `cause` alanından alt seviyedeki gerçek nedeni de loglar.

8.8 Hata Mesajları Satır Numarası Taşır

Önceki örneklerde defalarca gördüğümüz gibi, Cobra'nın ürettiği her çalışma-zamanı hatası `line N:` öneki taşır ve bu numara, hatanın **doğduğu** satırdır. Bu, hata ister doğduğu yerde yakalansın, ister birkaç fonksiyon yukarıda — hatta hiç yakalanmadan programı durdursun — değişmez. Yakalanmamış hatalar `runtime error:` önekiyle, yakalananlar ise yalnızca `line N: ...` string'i olarak gelir.

Bu tutarlılık, iki çalıştırma motoru için de geçerlidir: ister varsayılan ağaç-yürüten yorumlayıcı, ister `--vm` bytecode makinesi olsun, çalışma-zamanı hata mesajları satır numaralarıyla birlikte birebir aynıdır.

> **İpucu:** Bir hatayı loglarken `err`'i olduğu gibi yazdırmanız çoğu zaman yeterlidir; satır numarası zaten içindedir. Kendi yapısal hatalarınıza da bir bağlam (hangi işlem, hangi kayıt) eklerseniz, hata ayıklama çok daha hızlı olur.

8.9 Mühendislik Örneği: Yeniden Deneme (Retry) Döngüsü

Hata yönetiminin sık karşılaşılan bir kullanımı, *geçici* hatalarda işlemi **yeniden denemektir**. Ağ istekleri, kilitli kaynaklar veya kararsız servisler ikinci ya da üçüncü denemede başarılı olabilir. Aşağıdaki `with_retry`, bir fonksiyonu en çok `max_tries` kez dener; her başarısızlıkta tekrar dener, son deneme de başarısız olursa hatayı yukarı yayar. Burada `try`'ın bir döngü içinde nasıl yeniden çalıştırılabildiğini ve hatanın `throw err` ile nasıl yeniden atıldığını görüyoruz:

```
# --- Yeniden deneme (retry) döngüsü ---

# İlk iki çağrıda başarısız olan, üçüncüde başaran "kararsız" bir kaynak.
let attempts = 0
def fetch_user(id)
    attempts += 1
    if attempts < 3
        throw {"code": "UNAVAILABLE", "message": "geçici olarak erişilemiyor"}
    end
    return {"id": id, "name": "kullanıcı-" + str(id)}
end

# fn'i en çok max_tries kez dener; her başarısızlıkta yeniden dener,
# son deneme de başarısız olursa hatayı yukarı yayar.
def with_retry(fn, max_tries)
    let tries = 0
    while true
        tries += 1
        try
            return fn()
        catch err
            if tries >= max_tries
                print("son deneme de başarısız, hata yayılıyor")
                throw err
            end
            print("deneme " + str(tries) + " başarısız, yeniden denenecek")
        end
    end
end
```

```
def load()
    return fetch_user(7)
end

let user = with_retry(load, 5)
print("başarılı:", user["name"])
```

```
deneme 1 başarısız, yeniden denenecek
deneme 2 başarısız, yeniden denenecek
başarılı: kullanıcı-7
```

İlk iki deneme `UNAVAILABLE` hatası alır ve döngü tekrar döner; üçüncü deneme başarılı olunca `return fn()` çalışır ve `with_retry`'dan çıkılır. Eğer beş denemenin hepsi başarısız olsaydı, son `catch` `throw err` ile hatayı çağırana yayar ve onu en üstte yakalamak (ya da programın durmasına izin vermek) bizim sorumluluğumuzda olurdu.

8.10 Özet

Cobra'nın hata yönetimi, dilin geri kalanı gibi, az sayıda parçanın tutarlı davranışına dayanır:

- **Tek bir blok yapısı vardır:** `try ... [catch [err]] ... [finally] ... end`. En az `catch` ya da `finally` bulunmak zorundadır; aksi halde ayrıştırma hatası alırsınız.
- **`throw <değer>` her değeri atabilir** — string, integer, dict, list… `catch` onu aynen yakalar. Yapısal hatalar için `throw {"code": ..., "message": ...}` kalıbı, çağırana hem mesaj hem de makine-okur bilgi verir.
- **Çalışma-zamanı hataları mesaj string'i olarak yakalanır** (örn. `"line 3: division by zero"`). `type(err)` ile bunları kendi dict hatalarınızdan ayırt edebilirsiniz.
- **Exception türleri yoktur:** `catch` her şeyi yakalar. Seçici davranmak isterseniz `catch` içinde koşul kontrolü yapıp ele almadığınız hatayı `throw err` ile yeniden atarsınız.
- **`finally` her zaman çalışır** — normal bitişte, hatada ve `return`/`break`/`continue` ile erken çıkışta bile. Kaynak temizliğinin doğru yeri burasıdır. `finally` içinde atılan bir hata, asıl hatanın yerini alır (kazanır).
- **`return`, `break` ve `continue`, `try` ve `finally`'den geçer:** akış komutu etkisini göstermeden önce `finally` çalışır.
- **`finally` içinde `return` ve bildirimler (`let`, `const`, `def`, …) ayrıştırma hatasıdır.** `finally`'yi yalnızca temizlik için kullanın.
- **Hatalar çağrı yığınında en içteki `try`'a kadar yayılır;** yakalanmazsa programı `runtime error: ...` ile durdurur.
- **Hata mesajları, doğdukları satırın numarasını taşır** ve bu, iki çalıştırma motorunda (ağaç-yürüten ve `--vm`) birebir aynıdır.

Bu basit kurallarla; girdi doğrulama, bozuk veriyi yakalama, garantili kaynak temizliği, yeniden deneme ve katmanlar arası anlamlı hata yayılımı gibi gerçek mühendislik ihtiyaçlarının tamamını fazladan sözdizimi öğrenmeden karşılayabilirsiniz.

Bölüm 9

Modüller ve Paket Sistemi

Bir program büyüdükçe tek bir dosya hızla bir yük hâline gelir. Geometri hesaplarınız, yapılandırma sabitleriniz ve ana akışınız aynı dosyada iç içe geçtiğinde, neyin nereye ait olduğunu anlamak zorlaşır. Çözüm, kodu anlamlı parçalara — **modüllere** — bölmektir. Cobra'da her `.cobra` dosyası bir modüldür; bir modül diğerinin sunduklarını `import` ile içeri alır.

Cobra'nın modül sistemi iki sade ilke üzerine kuruludur. Birincisi: bir modül, **yalnızca açıkça `export` ile işaretlediğini** dışarıya açar — varsayılan gizliliktir, tıpkı ES modülleri veya Rust'ın `pub` anahtar kelimesi gibi. İkincisi: bir dosya **bir kez** çalıştırılır, sonucu önbelleğe alınır, ve yollar her zaman **içe aktaran dosyanın klasörüne göre** çözülür. Bu bölümde bu iki ilkeyi tüm ayrıntılarıyla göreceğiz: `import` ve `from ... import` biçimleri, gizlilik, "barrel" (çatı) modülleri, yerleşik modüllerin önceliği ve son olarak bir uygulamayı tek dosyaya derleyip paketleme.

9.1 Bir Modülü İçe Aktarmak: `import`

En temel biçim, bir dosyayı yoluyla içe aktarmaktır. `import`, dosyayı bir kez çalıştırır ve dosya adından türeyen bir **modül nesnesi** bağlar. Üyelere nokta (`.`) ile erişirsiniz:

```
import "lib/geometry.cobra"

print(geometry.PI)              # 3.14159
print(geometry.area_circle(2))  # 12.57
```

Modül nesnesinin adı, dosya adının `.cobra` uzantısı atılmış hâlidir: `lib/geometry.cobra` → `geometry`. Klasör yolu ada dahil değildir; yalnızca dosya adı sayılır.

> **Not:** Modül yolları **her zaman içe-aktaran dosyanın klasörüne görelidir**, programı nereden çalıştırdığınıza değil. Kök klasördeki `main.cobra`, `lib/` altındaki bir modülü `"lib/geometry.cobra"` ile gösterir; ama `lib/geometry.cobra`'nın kendisi, yanındaki `config.cobra`'yı sadece `"config.cobra"` ile gösterir — kendi klasörüne göre. (REPL'de "içe-aktaran dosya" olmadığı için yollar geçerli çalışma dizinine, yani cwd'ye göre çözülür.)

9.1.1 Takma Ad: `import ... as`

Bir modüle farklı bir ad vermek isterseniz `as` kullanırsınız. Bu, uzun dosya adlarını kısaltmak veya ad çakışmalarını önlemek için kullanışlıdır:

```
import "lib/geometry.cobra" as geo

print(geo.PI)            # 3.14159
print(geo.area_rect(8, 3))  # 24.0
```

9.1.2 Bir Kez Çalışır, Önbelleğe Alınır

Bir modül ne kadar çok kez içe aktarılırsa aktarılsın, dosya **yalnızca bir kez** çalıştırılır; sonraki `import`'lar önbellekteki aynı modülü verir. Bunu modülün en üst düzeyindeki bir yan etkiyle (örneğin bir `print`) gözlemleyebilirsiniz:

```
# loud.cobra üst düzeyde print içerir:
import "lib/loud.cobra"
import "lib/loud.cobra" as l2     # aynı modül; tekrar ÇALIŞMAZ
print(loud.X + l2.X)

loud.cobra yüklendi
2
```

"loud.cobra yüklendi" çıktısı iki `import`'a rağmen yalnızca bir kez görünür. Bu önbellekleme, hem performans hem de doğruluk için önemlidir: paylaşılan bir modülün durumu her yerde aynıdır.

> **Uyarı:** Modüller **döngü** oluşturamaz. `a.cobra`, `b.cobra`'yı içe aktarırken `b.cobra` da `a.cobra`'yı içe aktarırsa, Cobra bunu bir hata olarak bildirir:
>
> ```
> runtime error: ... circular import: "a.cobra"
> ```
>
> Döngüsel bağımlılık genellikle bir tasarım kokusudur; ortak parçayı üçüncü bir modüle çıkararak çözün.

9.2 Varsayılan Gizlilik: `export`

Cobra'nın modül sisteminin kalbinde tek bir kural vardır: **bir modül, yalnızca `export` ile işaretlediğini dışarıya açar.** İşaretlenmeyen her şey — değişkenler, fonksiyonlar, struct'lar — modüle özeldir ve dışarıdan görünmez. Bu, "varsayılan gizli" yaklaşımıdır; bir şeyi public yapmak bilinçli bir karar gerektirir.

```
# geometry.cobra
export const PI = 3.14159

export def area_circle(r)      # public
    return _round(PI * r * r)
end

def _round(x)                  # PRIVATE — `export` yok
    ...
end
```

Burada `area_circle` dışa açıktır, ama `_round` değildir. İçe aktaran taraf `geometry.area_circle(...)` çağırabilir, ancak `geometry._round(...)` çağırırsa hata alır — çünkü `_round` modülün public yüzeyinde yer almaz.

> **İpucu:** _ ön eki (örneğin `_round`, `_helper`) yalnızca bir **gelenektir**: "bu özeldir" mesajını okuyucuya verir. Asıl gizliliği sağlayan, _ değil, `export`'un yokluğudur. İsterseniz private bir yardımcıya _ koymadan da yazabilirsiniz; `export` olmadığı sürece gizli kalır.

`export` şu bildirimlerin hepsiyle çalışır: `let`, `const`, `def`, `struct` ve `record`:

```
export let DEFAULT_UNIT = "cm"
export const MAX = 100
export def area(r) ... end
export struct Circle ... end
export record Point ... end
```

9.3 Seçili İçe Aktarma: `from ... import`

`import`, modülün tamamını tek bir nesne olarak bağlar ve siz üyelere `modul.uye` ile erişirsiniz. Bazen yalnızca birkaç ada ihtiyacınız vardır ve her seferinde modül adını yazmak istemezsiniz. `from ... import` tam da bunu yapar: seçili public üyeleri **doğrudan** geçerli kapsama bağlar — prefix yok:

```
from "lib/geometry.cobra" import area_circle, PI

print(area_circle(2))   # 12.57   — "geometry." gerekmez
print(PI)               # 3.14159
```

İçe aktarılan bir adı yeniden adlandırmak için `as` kullanırsınız:

```
from "lib/geometry.cobra" import PI as PI_SABIT, area_rect as alan

print(alan(8, 3))       # 24.0
print(PI_SABIT)         # 3.14159
```

9.3.1 Hatalar İÇE-AKTARMA ANINDA Yakalanır

`from ... import`'ın en değerli özelliği, bir yazım hatasını veya dışa açılmamış bir adı **içe aktarma anında** — sonradan `null`'a erişen gizemli bir hata olarak değil — hemen bildirmesidir:

```
from "lib/geometry.cobra" import area_circel    # yazım hatası!
```

```
runtime error: ... module "lib/geometry.cobra" does not export "area_circel"
```

Aynı şey, var olan ama **dışa açılmamış** (private) bir adı istemeye çalıştığınızda da olur:

```
from "lib/geometry.cobra" import pow10          # var, ama export değil
```

```
runtime error: ... module "lib/geometry.cobra" does not export "pow10"
```

Her iki durumda da hata, programın geri kalanı çalışmadan önce ortaya çıkar. Bu, modüller arası sözleşmeleri (contract) erken doğrulamanın güçlü bir yoludur.

9.3.2 Hepsini Bağlamak: `from ... import *`

`from "yol" import *`, modülün **tüm** dışa açılan adlarını tek seferde geçerli kapsama bağlar. Bir modülün public API'sini prefix'siz olarak baştan sona kullanmak istediğinizde elverişlidir:

```
from "lib/geometry.cobra" import *

print(area_circle(2))     # 12.57
print(Circle(3).area())   # 28.27
```

> **Not:** `from ... import *` adları yalnızca **geçerli kapsama** bağlar; onları otomatik olarak *yeniden dışa açmaz*. Yani bir modülün içinde `import *` yapmanız, o adların sizin modülünüzün public yüzeyine eklenmesini sağlamaz. Bir adı yeniden yayınlamak için onu açıkça `export` etmeniz gerekir — bir sonraki bölümdeki "barrel" deseni tam olarak bunu gösterir.

9.4 Mühendislik Örneği: Çok Dosyalı Bir Uygulama

Şimdiye dek gördüğümüz parçaları gerçek bir uygulamada bir araya getirelim. Küçük bir "Geometri Atölyesi" kuracağız ve onu üç sorumluluğa böleceğiz: yapılandırma, geometri çekirdeği ve bunları tek çatı altında toplayan bir "barrel" modülü. Klasör yapısı şöyle olacak:

```
proje/
├── main.cobra            # giriş noktası
└── lib/
    ├── config.cobra      # uygulama sabitleri
    ├── geometry.cobra    # geometri çekirdeği (public API)
    └── shapes.cobra      # barrel: alt modülleri yeniden yayınlar
```

9.4.1 `lib/config.cobra` — Yapılandırma

En küçük modülle başlayalım: uygulama genelindeki sabitler. Hepsi `export` edilmiştir, çünkü amaçları zaten dışarıdan okunmaktır.

```
# config.cobra — uygulama genelindeki ayarlar.
# Yalnızca `export` ile işaretlenenler dışarıya açıktır.

export const APP_NAME = "Geometri Atölyesi"
export const VERSION  = "1.0.0"

# Hesaplamalarda kaç ondalık basamağa yuvarlanacağı.
export const PRECISION = 2
```

9.4.2 `lib/geometry.cobra` — Geometri Çekirdeği

Asıl iş burada. Bu modül, kardeş modülü `config.cobra`'yı içe aktarır (kendi klasörüne göre, sadece `"config.cobra"`), public alan fonksiyonları ve struct'lar sunar, ama yuvarlama yardımcılarını **özel** tutar:

```
# geometry.cobra — düzlem geometrisi temel işlemleri.
# Public API: PI, area_circle, area_rect, Circle, Rect.
# `_` ile başlayan adlar bir gelenektir; ASIL gizlilik `export`in
# yokluğundan gelir — işaretlenmeyen her şey modüle özeldir.

import "config.cobra"        # aynı klasördeki kardeş modül

export const PI = 3.14159

# Bir sayıyı config.PRECISION basamağa yuvarlayan özel yardımcı.
# `export` YOK → dışarıdan görünmez.
def _round(x)
    let factor = pow10(config.PRECISION)
    return float(int(x * factor + 0.5)) / factor
end

# Yalnızca bu modülün kullandığı bir başka özel yardımcı.
def pow10(n)
    let result = 1
    let i = 0
    while i < n
        result = result * 10
        i = i + 1
    end
    return result
end

export def area_circle(r)
    return _round(PI * r * r)
end

export def area_rect(w, h)
    return _round(w * h)
end

export struct Circle
    def init(r)
        self.r = r
    end
    def area()
        return area_circle(self.r)
    end
end

export struct Rect
    def init(w, h)
        self.w = w
        self.h = h
    end
    def area()
        return area_rect(self.w, self.h)
    end
end
```

Dikkat edin: `area_circle`, modül içinden private `_round`'u çağırabilir — bir modülün **kendi içinde** her şey görünürdür. Gizlilik yalnızca *dışarıya* karşıdır. Aynı şekilde `Circle.area()` metodu, üst düzeydeki `area_circle` fonksiyonunu doğrudan (prefix'siz) kullanabilir.

9.4.3 `lib/shapes.cobra` — "Barrel" Modülü

Bir **barrel** (çatı) modülü, birkaç alt modülün public adlarını tek bir noktada toplar; böylece kullanıcılar dört ayrı yerden değil, tek bir modülden içe aktarır. Bir adı yeniden yayınlamak için onu içeri alıp (`import ... as`) sonra açıkça `export` ederiz:

```
# shapes.cobra — "barrel" modülü.
# Alt modüllerin public adlarını tek bir çatı altında toplar; böylece
# kullanıcılar tek bir yerden içe aktarır. Yeniden-dışa-aktarmak için adı
# içeri alır (`import ... as`), sonra `export` ile yeniden yayınlarız.
from "geometry.cobra" import area_circle as _area_circle
from "geometry.cobra" import area_rect   as _area_rect
from "geometry.cobra" import Circle       as _Circle
from "geometry.cobra" import Rect         as _Rect

export let area_circle = _area_circle
export let area_rect   = _area_rect
export let Circle      = _Circle
export let Rect        = _Rect
```

> **İpucu:** Burada içeri aldığımız adlara _ ön eki koyup (`_area_circle`), sonra **temiz** adı `export` ettik. Bu küçük dans, gizli yerel ad ile public yeniden-dışa-açılan adı ayırır. Proje büyüdükçe, kullanıcılar tek bir `from "lib/shapes.cobra" import *` ile tüm geometri API'sine erişir — alt modüllerin nasıl bölündüğünü bilmelerine gerek kalmadan.

9.4.4 `main.cobra` — Giriş Noktası

Son olarak, her şeyi bir araya getiren giriş noktası. `config`'i modül nesnesi olarak (prefix'li) kullanırken, barrel'ı `import *` ile prefix'siz kullanıyoruz:

```
# main.cobra — uygulama giriş noktası.
# Modül yolları HER ZAMAN içe-aktaran dosyanın klasörüne görelidir;
# bu dosya kök klasörde olduğu için alt klasördeki modülleri "lib/..."
# ile gösteririz.

import "lib/config.cobra"                # modül nesnesi: config.APP_NAME
from "lib/shapes.cobra" import *         # barrel: tüm public adlar prefix'siz

print(config.APP_NAME + " v" + config.VERSION)

let bahce = Circle(5)
let havuz = Rect(8, 3)

print("Daire alanı: " + str(bahce.area()))
print("Dikdörtgen alanı: " + str(havuz.area()))
print("Serbest fonksiyon: " + str(area_circle(1)))
```

Programı kök klasörden çalıştırdığımızda:

```
cobra main.cobra

Geometri Atölyesi v1.0.0
Daire alanı: 78.54
Dikdörtgen alanı: 24.0
Serbest fonksiyon: 3.14
```

Aynı çıktıyı `cobra --vm main.cobra` ile de alırsınız — iki motor birebir aynı davranır. Bu küçük örnek, sağlıklı bir modül tasarımının tüm imzalarını taşır: her dosyanın tek bir sorumluluğu var, public API `export` ile bilinçle seçilmiş, yardımcılar gizli, ve bağımlılıklar tek yönlü akıyor (`main` → `shapes` → `geometry` → `config`).

9.5 Yerleşik Modüllerin Önceliği

Cobra, `math`, `fs`, `json`, `http` gibi bir dizi **yerleşik (native) modül** sunar. Bunlar `.cobra` uzantısı olmadan, sadece adlarıyla içe aktarılır:

```
import "math"
print(math.sqrt(16))    # 4.0
```

Önemli bir kural: **yerleşik modüller, aynı adlı dosyalara göre önceliklidir.** Klasörünüzde `math.cobra` adında bir dosya olsa bile, `import "math"` her zaman yerleşik `math` modülünü yükler — sizin dosyanızı değil:

```
# Klasörde math.cobra olsa bile:
import "math"
print(math.pi)          # 3.141592653589793  (yerleşik)
print(math.sqrt(16))    # 4.0
```

Kendi dosyanızı kastediyorsanız, ona uzantısını ekleyerek (`import "math.cobra"`) açıkça dosya yolu vermeniz gerekir; uzantısız ad her zaman yerleşiği seçer.

> **Uyarı:** Bu öncelik nedeniyle, kendi modüllerinize yerleşiklerle çakışan adlar (`math`, `os`, `time`, `json`, `re`, `http`, `fs`, `net`, `proc`, `test`, `runtime`) vermekten kaçının. Aksi halde uzantısız bir `import` her zaman yerleşiğe gider ve dosyanız hiç yüklenmez.

9.6 Derleme ve Paketleme: `cobra build` ve `--bundle`

Bir programı her çalıştırışınızda Cobra onu ayrıştırır ve derler. Bu adımı bir kez yapıp sonucu saklamak için `cobra build` kullanırsınız; bu, kaynak dosyanın yanına bir `.cobrac` **bytecode** dosyası yazar:

```
cobra build main.cobra
# wrote main.cobrac
```

Üretilen `.cobrac` doğrudan çalıştırılabilir ve ayrıştırma/derleme adımlarını atladığı için daha hızlı başlar:

```
cobra main.cobrac
# Geometri Atölyesi v1.0.0
# Daire alanı: 78.54
# ...
```

> **Not:** Sıradan bir `cobra build`, yalnızca ana dosyayı derler; içe aktarılan `.cobra` modülleri **çalışma anında diskten** yüklenmeye devam eder. Yani `.cobrac` dosyasını dağıtacaksanız, `lib/` klasörünü de yanında taşımanız gerekir — yoksa modüller bulunamaz.

9.6.1 Tek Dosyalık Paket: `--bundle`

Tüm uygulamayı tek, kendine yeten bir dosyaya gömmek için `--bundle` bayrağını eklersiniz. Bu, içe aktarılan **tüm `.cobra` modüllerini** `.cobrac` dosyasının içine gömer:

```
cobra build --bundle main.cobra
# wrote main.cobrac (3 modules bundled)
```

"3 modules bundled" — `config.cobra`, `geometry.cobra` ve `shapes.cobra`'nın üçü de gömüldü. Artık bu tek dosya, `lib/` klasörü silinmiş olsa bile çalışır:

```
rm -rf lib/          # modül dosyalarını kaldır
cobra main.cobrac    # yine de çalışır — modüller gömülü
# Geometri Atölyesi v1.0.0
# Daire alanı: 78.54
# Dikdörtgen alanı: 24.0
# Serbest fonksiyon: 3.14
```

Bu, bir Cobra uygulamasını dağıtmanın en temiz yoludur: tek bir `.cobrac` dosyası, harici kaynak bağımlılığı olmadan.

> **Uyarı:** `--bundle` yalnızca `.cobra` modüllerini gömer. Yerel (native) eklentiler (`.so` dosyaları) makine koduna derlendikleri için gömülmez; bunlar yine çalışma anında diskten yüklenir. Bir `.so` eklentisine bağımlı bir uygulamayı dağıtırken, paketin yanında eklentiyi de taşımalısınız.

9.7 Özet

Bu bölümde Cobra'nın modül sistemini baştan sona gördük:

- Her `.cobra` dosyası bir modüldür. `import "lib/x.cobra"`, dosyayı **bir kez** çalıştırır (sonucu önbelleğe alır) ve `x` adlı bir **modül nesnesi** bağlar; üyelere `x.uye` ile erişirsiniz. `import "x.cobra" as y` takma ad verir.
- Modül yolları **her zaman içe-aktaran dosyanın klasörüne görelidir** (REPL'de geçerli çalışma dizinine, cwd'ye). **Döngüsel** içe aktarmalar bir hatadır.
- **Varsayılan gizlilik**: bir modül yalnızca `export` ile işaretlediğini dışarıya açar. `export`; `let`, `const`, `def`, `struct` ve `record` üzerinde çalışır. İşaretsiz adlar (gelenekle `_helper` gibi) gizli kalır; gerçek gizliliği _ değil, `export`un yokluğu sağlar.
- `from "yol" import a, b as c`, seçili public üyeleri **doğrudan** kapsama bağlar (prefix'siz). Bir yazım hatası veya dışa açılmamış bir ad **içe-aktarma anında** hata verir — geç ve gizemli değil, erken ve net.
- `from "x" import *`, tüm public adları bağlar. Yeniden-dışa-açma için onları ayrıca `export` etmeniz gerekir; "barrel" modülleri tam olarak bunu yapar (`import ... as` ile alıp `export let` ile yeniden yayınlar).
- **Yerleşik (native) modüller**, aynı adlı dosyalara göre önceliklidir; uzantısız `import "math"` her zaman yerleşiği seçer.
- `cobra build dosya.cobra`, hızlı başlayan bir `.cobrac` bytecode üretir (modüller yine diskten yüklenir). `cobra build --bundle app.cobra`, içe aktarılan tüm `.cobra` modüllerini tek dosyaya gömerek kendine yeten bir paket oluşturur (`.so` eklentileri gömülmez).

Modüller, küçük programları gerçek uygulamalara dönüştüren yapıştırıcıdır: sorumlulukları ayırır, public API'leri bilinçle tasarlatır ve kodu yeniden kullanılabilir kılar. Bir sonraki bölümde, bu modüllerin sunduğu işlevleri **eşzamanlı** olarak — Cobra'nın GIL'siz gerçek paralelliğiyle — nasıl çalıştıracağımıza bakacağız.

Bölüm 10

Eşzamanlılık ve Gerçek Paralellik

Çoğu betik dili size eşzamanlılığın *yanılsamasını* verir. Tek bir yorumlayıcı kilidinin (GIL, Global Interpreter Lock) ardına gizlenirler: birden çok "iş parçacığı" yazabilirsiniz, ama herhangi bir anda yalnızca biri kod çalıştırır. Bu, ağ ve disk gibi *bekleme ağırlıklı* (I/O-bound) işlerde işe yarar, ancak hesaplama ağırlıklı (CPU-bound) bir işi dört çekirdeğe bölerek dört kat hızlandırmak istediğinizde duvara toslarsınız: çekirdekler vardır ama dil onları kullanamaz.

Cobra farklıdır. **Cobra'da GIL yoktur.** Bir görevi (`task`) başlattığınızda o görev kendi işletim sistemi iş parçacığı üzerinde, diğer görevlerle birlikte **gerçekten aynı anda**, birden çok CPU çekirdeğinde koşar. Bu, hesaplama ağırlıklı işin gerçekten paralelleştiği anlamına gelir; ekstra bir sözdizimi ya da numara gerekmez. Bunun bir bedeli vardır: paylaşılan değişebilir durumu artık siz açıkça korumak zorundasınız. Cobra bu korumayı sağlamanız için size sade ama eksiksiz bir araç takımı verir: görevler, kanallar, kilitler ve paralel yerleşikler.

Bu bölümde şunları göreceğiz: `async def` ile görev başlatmak ve `await` ile sonuçlarını toplamak; hataların `await` noktasında nasıl yeniden fırlatıldığı; paylaşılan durumu `Mutex` ve `RWMutex` ile korumak; `Channel` ile görevler arasında veri akıtmak ve işçi havuzu kurmak; `select` ile birden çok kanal üzerinde beklemek (zaman aşımı ve fan-in dahil); ve `pmap` / `pfilter` / `pforeach` / `preduce` ile tek satırda veri paralelliği. Tüm örnekler çalıştırılıp doğrulanmıştır.

10.1 GIL Yok: Görevler Gerçekten Paralel Koşar

Bir fonksiyonu `async def` ile tanımladığınızda, onu *çağırmak* gövdeyi hemen çalıştırmaz; bunun yerine **anında bir görev (task) döndürür** ve gövde arka planda, kendi iş parçacığında koşmaya başlar. Sonucu istediğinizde `await` kullanırsınız; `await`, görev bitene kadar bekler ve dönüş değerini verir.

```
import "time"

async def work(n)
    time.sleep(0.05)
    return n * n
end

# Üç görevi aynı anda başlat, sonra hepsini sırayla topla.
let results = await [work(1), work(2), work(3)]
print(results)            # [1, 4, 9]

# Tek bir görevi beklemek de aynı şekilde çalışır.
```

```
let single = await work(4)
print(single)             # 16
```

Burada üç `work(...)` çağrısı neredeyse aynı anda başlar. Her biri 50 ms uyusa da, paralel koştukları için tamamı yaklaşık 50 ms'de biter — toplam 150 ms değil. `await [t1, t2, t3]` bir **görev listesi** alır ve **sonuçları giriş sırasıyla** döndürür: hangisinin önce bittiği önemli değildir, çıktı her zaman `[1, 4, 9]` olur.

Bunun gerçekten paralel olduğunu hesaplama ağırlıklı bir işle kanıtlayabiliriz. Aşağıda aynı ağır hesabı önce seri, sonra paralel yapıyoruz:

```
import "time"
import "runtime"

print("cekirdek sayisi:", runtime.num_cpu())

# Saf CPU işi: hiçbir uyku yok, sadece hesap.
def heavy(n)
    let total = 0
    for i in range(3000000)
        total = total + (i % 7)
    end
    return total
end

async def aheavy(n)
    return heavy(n)
end

# Seri: dört işi peş peşe yap.
let t0 = time.clock()
for n in range(4)
    heavy(n)
end
let seri = time.clock() - t0

# Paralel: dört görevi aynı anda başlat ve topla.
let t1 = time.clock()
await [aheavy(0), aheavy(1), aheavy(2), aheavy(3)]
let paralel = time.clock() - t1

print("paralel seriden hizli mi:", paralel < seri)
```

Birden çok çekirdeği olan bir makinede çıktı şudur:

```
cekirdek sayisi: 10
paralel seriden hizli mi: true
```

Paralel sürüm seri sürümden ölçülebilir biçimde hızlıdır; çünkü dört `heavy` çağrısı dört ayrı çekirdekte gerçekten aynı anda koşar. Tek çekirdekli (veya GIL'li) bir dilde bu mümkün olmazdı.

> **Not:** Bir görevi başlatmak için onu `await` ile beklemek zorunda değilsiniz. `work(1)` çağrısı görevi hemen başlatır; dönen task değerini bir değişkende tutup işinizi yapar, sonucu daha sonra `await` ile alabilirsiniz. "Başlat ve unut" tarzı görevler için dönüşü hiç beklemeyebilirsiniz de — ama program biterken hâlâ koşan görevler düşürülür (sonuçları kaybolur).

> **İpucu:** Çok sayıda görevi bir döngüde başlatıp listede biriktirebilir, sonra tek bir `await` ile hepsini toplayabilirsiniz: `let tasks = []` … `tasks.push(work(i))` … `let all = await tasks`.

10.2 Görevlerdeki Hatalar: `await` Noktasında Yeniden Fırlar

Bir görevin gövdesinde bir hata oluşursa (bir `throw` ya da çalışma-zamanı hatası), bu hata görevi başlattığınız yerde *değil*, onu **`await` ettiğiniz noktada** yeniden fırlatılır. Böylece hatayı her zamanki `try/catch` ile yakalarsınız:

```
async def boom()
    throw "is patladi"
end

try
    await boom()
catch err
    print("yakalandi:", err)   # yakalandi: is patladi
end
```

Bu davranış önemlidir: görev arka planda sessizce patlayıp kaybolmaz. Hata, siz sonucu talep edene (`await`) kadar görevde "saklı" durur, sonra size geri fırlatılır. Bir görev listesini `await [...]` ile beklerken **ilk hata** (giriş sırasına göre) yayılır.

> **Uyarı:** Bir görevi hiç `await` etmezseniz, gövdesindeki bir hatayı hiçbir zaman görmezsiniz — hata o görevde saklı kalır ve program biterken görevle birlikte düşer. Bir görevin başarısını önemsiyorsanız onu mutlaka `await` edin.

10.3 Paylaşılan Durum Neden Tehlikeli?

Görevler gerçekten paralel koştuğundan, ikisi *aynı* değişebilir değere aynı anda yazarsa **yarış koşulu** (race condition) oluşur. Klasik örnek paylaşılan bir sayaçtır. `total = total + 1` tek bir adım gibi görünür ama aslında üç adımdır: oku, bir ekle, geri yaz. İki görev araya girerse güncellemelerden biri kaybolur.

```
# KİLİTSİZ — bilerek hatalı: kayıp güncelleme gösterimi
let total = 0
def heavy()
    let s = 0
    for i in range(20000)
        s = s + 1
    end
    return s
end
async def bump()
    for i in range(1000)
        heavy()            # araya girme şansını artırmak için gecikme
        total = total + 1
    end
end
await [bump(), bump(), bump(), bump()]
print("kilitsiz beklenen 4000, gercek:", total)
```

Bu programı birkaç kez çalıştırdığınızda çıktı değişir:

```
kilitsiz beklenen 4000, gercek: 4000
kilitsiz beklenen 4000, gercek: 3997
kilitsiz beklenen 4000, gercek: 3999
```

Bazen doğru (4000), çoğu zaman eksik. Bu, eşzamanlılığın en sinsi hatasıdır: arada bir doğru sonuç verir, bu yüzden testlerde gözden kaçar. Çözüm, paylaşılan durumu **açıkça senkronize etmektir**.

> **Uyarı:** Tasarım gereği, ayrı global değişkenlere dokunan görevler her zaman güvenlidir (global kapsam içeride senkronize edilir). Tehlike, *aynı* değeri paylaşıp eşzamanlı değiştirdiğinizde ortaya çıkar. Paylaşılan bir sayıyı, listeyi, dict'i veya struct alanını birden çok görev değiştirecekse, onu bir kilitle korumalısınız.

10.4 `Mutex` ile Karşılıklı Dışlama

`Mutex` (mutual exclusion — karşılıklı dışlama), aynı anda yalnızca bir görevin koruduğu bölgeye girmesine izin verir. `m.lock()` kilidi alır (başka biri tutuyorsa bekler), `m.unlock()` bırakır. İkisi arasındaki kod **kritik bölgedir**: aynı anda yalnızca bir görev oradadır.

Önceki yarış koşulunu bir `Mutex` ile düzeltelim:

```
let m = Mutex()
let total = 0

async def bump()
    m.lock()
    total = total + 1     # kritik bölge: aynı anda tek görev
    m.unlock()
end

let tasks = []
for i in range(100)
    tasks.push(bump())
end
await tasks
print(total)              # 100 — her zaman, istisnasız
```

Artık `oku-ekle-yaz` üçlüsü bölünmez (atomik) hale gelir; çıktı her çalıştırmada kesinlikle `100`'dür.

`Mutex`'in üçüncü bir yöntemi vardır: `try_lock()`. Kilidi *engellemeden* almaya çalışır; alabilirse `true`, kilit zaten tutuluyorsa beklemeden `false` döner.

```
let m = Mutex()
print(m.try_lock())     # true  — kilit boştu, alındı
print(m.try_lock())     # false — zaten kilitli, beklemeden döndü
m.unlock()
```

> **İpucu:** `lock` ile `unlock`'u her zaman eşleştirin. Kritik bölgede hata atabilecek kod varsa, kilidin bırakıldığından emin olmak için `try/finally` kullanın: `m.lock()` … `try` … işlem … `finally` … `m.unlock()` … `end`. Böylece bir hata bile kilidi sonsuza dek tutmaz.

10.4.1 Paylaşılan dict'i korumak

Bir `dict`'i (veya struct alanını) birden çok görevin eşzamanlı değiştirmesi tehlikelidir; Cobra'nın güvenlik ağı bunu yakalayıp `"data race in task"` biçiminde *yakalanabilir* bir hataya çevirir, ama bu yalnızca bir ağdır — doğru yöntem dict'i bir `Mutex` ile korumaktır:

```
let cache = {}
let m = Mutex()

async def store(k, v)
    m.lock()
    cache[k] = v          # paylaşılan dict yazımı: kilit altında
    m.unlock()
end

let tasks = []
for i in range(50)
    tasks.push(store("k" + str(i), i))
end
await tasks
print(len(cache))         # 50
print(cache["k49"])       # 49
```

> **Uyarı:** Paylaşılan bir *dict*'in ya da örnek alanının kilitsiz eşzamanlı değiştirilmesi, paylaşılan bir sayının kilitsiz değiştirilmesinden daha kötüdür: sessizce yanlış sonuç vermek yerine ölümcül bir veri yarışına yol açabilir. Birden çok görevin yazdığı her dict'i mutlaka kilitleyin.

10.5 `RWMutex`: Çok Okuyucu, Tek Yazıcı

Bazı paylaşılan durumlar *okuma ağırlıklıdır*: çok sayıda görev sürekli okur, arada bir biri yazar. Düz bir `Mutex` burada israftır, çünkü okuyucuların birbirini engellemesi için bir neden yoktur — yalnızca yazıcı herkesi dışlamalıdır. `RWMutex` (readers-writer lock) tam da bunu sağlar:

- `rlock()` / `runlock()` — **okuma kilidi**: aynı anda birçok okuyucu tutabilir.
- `lock()` / `unlock()` — **yazma kilidi**: tek, dışlayıcı; tutulduğunda hiçbir okuyucu giremez.
- `try_lock()` — yazma kilidini engellemeden dener.

```
let rw = RWMutex()
let config = {"limit": 10}

async def reader(id)
    rw.rlock()
    let v = config["limit"]    # birçok okuyucu aynı anda burada olabilir
    rw.runlock()
    return v
end

async def writer(v)
    rw.lock()
    config["limit"] = v        # yazarken kimse okuyamaz
    rw.unlock()
```

```
end

await [reader(1), reader(2), writer(20), reader(3)]
print(config["limit"])         # 20
```

> **İpucu:** Kuralı basit tutun: bir değeri yalnızca *okuyorsanız* `rlock`/`runlock`, *değiştiriyorsanız* `lock`/`unlock` kullanın. Yanlışlıkla yazma kilidi yerine okuma kilidi altında yazmak yarış koşulu yaratır.

10.6 `Channel`: Görevler Arasında Veri Akıtmak

Kilitler paylaşılan durumu *korur*; kanallar ise veriyi bir görevden diğerine *taşır*. Bir `Channel`, görevler arası tip-güvenli bir borudur: bir uçtan `send` ile değer koyarsınız, diğer uçtan `recv` ile alırsınız.

İki tür kanal vardır:

- `Channel()` — **tamponsuz**: `send`, bir `recv` onu alana kadar bekler (ve tam tersi). Bu, iki görev arasında senkron bir el sıkışmadır.
- `Channel(n)` — **tamponlu**: kanal `n` değere kadar tutabilir; tampon dolana kadar `send` beklemez.

```
# Tamponsuz kanal: bir görev üretir, ana akış tüketir.
let ch = Channel()
async def produce()
    ch.send(42)
end
produce()
print(ch.recv())          # 42

# Tamponlu kanal: kapatınca önce tampon boşalır, sonra null gelir.
let buf = Channel(3)
buf.send(1)
buf.send(2)
print(len(buf))           # 2  — bekleyen tamponlanmış değer sayısı
buf.close()
print(buf.recv())         # 1
print(buf.recv())         # 2
print(buf.recv())         # null — tampon boş + kanal kapalı
```

Dikkat edilecek iki nokta: `len(ch)` kanaldaki bekleyen değer sayısını verir; ve `close()` çağrıldıktan sonra `recv` önce tamponda kalan değerleri boşaltır, **sonra `null` döndürmeye başlar**. Bu `null`, "kanal bitti" sinyali olarak döngülerde kullanmak için idealdir.

`send`/`recv`'in engellemesiz kardeşleri de vardır: `try_send(v)` ve `try_recv()`.

```
let ch = Channel(1)
print(ch.try_send(1))   # true  — yer vardı, gönderildi
print(ch.try_send(2))   # false — tampon dolu, beklemeden döndü
print(ch.try_recv())    # 1
print(ch.try_recv())    # null  — kanal boş, beklemeden döndü
```

> **Not:** Kapalı bir kanala `send` yapmak hatadır. Genel kural: kanalı **gönderen taraf kapatır** (artık veri gelmeyeceğini yalnızca o bilir), alan taraf değil.

10.6.1 İşçi havuzu (worker pool)

Kanalların en güçlü kullanımı **işçi havuzudur**: sabit sayıda işçi görev, bir "işler" kanalından iş çeker, sonucu bir "sonuçlar" kanalına yazar. Bu desen, çok sayıda bağımsız işi sınırlı sayıda paralel işçiye dağıtmanın standart yoludur.

```
# 3 işçi, 8 işi paralel olarak öğütür.
let jobs = Channel(10)
let results = Channel(10)

async def worker()
    while true
        let job = jobs.recv()
        if job == null         # kanal kapandı: bu işçi işini bitirdi
            return null
        end
        results.send(job * job)
    end
end

# İşçi havuzunu başlat.
let pool = []
for i in range(3)
    pool.push(worker())
end

# İşleri besle, sonra kanalı kapat (işçilere "bitti" sinyali).
for n in range(1, 9)
    jobs.send(n)
end
jobs.close()

# Sonuçları topla.
let collected = []
for i in range(1, 9)
    collected.push(results.recv())
end
await pool                     # tüm işçilerin temiz bittiğinden emin ol

collected.sort()               # işçiler paralel, sıra değişebilir; sıralayalım
print(collected)               # [1, 4, 9, 16, 25, 36, 49, 64]
```

Üç işçi sekiz işi aralarında paylaşır. `jobs.close()`, her işçinin `recv` çağrısının sonunda `null` görmesini sağlar; böylece işçiler döngülerinden temizce çıkar. Sonuçların geliş sırası işçilerin hızına bağlı olduğundan deterministik değildir; bu yüzden karşılaştırmadan önce listeyi `sort()`'ladık.

> **İpucu:** İşçi sayısını işin türüne göre seçin. Hesaplama ağırlıklı işlerde `runtime.num_cpu()` kadar işçi tipik olarak en iyisidir; bekleme ağırlıklı (ağ/disk) işlerde çekirdek sayısından çok daha fazla işçi kullanabilirsiniz.

10.7 `select`: Birden Çok Kanal Üzerinde Beklemek

Bazen tek bir kanal değil, *birkaç* kanaldan hangisi önce hazır olursa onunla ilgilenmek istersiniz. `select(ops)` tam da bunu yapar: bir kanal listesi alır ve ilk hazır olan üzerinde işlem yaparak hangisinin seçildiğini anlatan bir dict döndürür: `{index, value, ok, send}`.

`ops` listesindeki her giriş ya bir kanaldır (alma/`recv` işlemi) ya da `[kanal, deger]` biçiminde iki elemanlı bir listedir (gönderme/`send` işlemi).

```
import "time"
let a = Channel()
let b = Channel()

async def fill(ch, v, d)
    time.sleep(d)
    ch.send(v)
end
fill(a, "a-sonuc", 0.02)   # a daha erken hazır olur
fill(b, "b-sonuc", 0.10)

let r = select([a, b])
print(r["index"], r["value"], r["ok"], r["send"])
# 0 a-sonuc true false   — index 0 (a) seçildi, bir recv'di (send=false)
```

`a` daha erken (`0.02 s`) hazır olduğu için `select` onu seçer: `index` 0, `value` alınan değer, `ok` kanalın açık olduğunu, `send` ise bunun bir gönderme değil alma işlemi olduğunu (`false`) söyler.

`select`'in ikinci bir biçimi engellemesizdir: `select(ops, default)`. Hiçbir işlem hazır değilse beklemez, doğrudan `default`'u döndürür.

```
let empty = Channel()
let r2 = select([empty], "bos")
print(r2)                  # bos — hiçbir kanal hazır değildi
```

10.7.1 `select` ile zaman aşımı

`select`'in en yaygın kullanımı **zaman aşımıdır**: bir iş kanalı ile bir zamanlayıcı kanalını aynı anda bekleyip, hangisi önce gelirse ona göre davranırsınız. İş zamanlayıcıdan önce gelirse sonucu kullanırsınız; zamanlayıcı önce gelirse "zaman aşımı" dersiniz.

```
import "time"
let work = Channel()
let timeout = Channel()

# Yavaş iş: 200 ms sonra sonucu gönderir.
async def slow()
    time.sleep(0.20)
    work.send("bitti")
end
slow()

# Zamanlayıcı: 50 ms sonra ateşler.
```

```
async def fire()
    time.sleep(0.05)
    timeout.send(true)
end
fire()

let r = select([work, timeout])
if r["index"] == 0
    print("sonuc:", r["value"])
else
    print("zaman asimina ugradi")    # zamanlayıcı önce geldi
end
```

İş 200 ms sürdüğü, zamanlayıcı ise 50 ms'de ateşlediği için çıktı şudur:

```
zaman asimina ugradi
```

10.7.2 Fan-in: çok üreticiyi tek kanalda toplamak

`select`'in kardeş deseni **fan-in**'dir: birden çok üretici görevin çıktısını tek bir kanalda toplamak. Burada `select`'e gerek bile yoktur — tüm üreticiler aynı kanala yazar, ayrı bir görev tüm üreticileri bekleyip kanalı kapatır, tüketici de kanal `null` dönene kadar okur:

```
let out = Channel(10)

async def producer(name, count)
    for i in range(count)
        out.send(name + "-" + str(i))
    end
end

let producers = [producer("A", 3), producer("B", 3)]

# Tüm üreticiler bitince kanalı kapatan ayrı bir görev.
async def closer()
    await producers
    out.close()
end
closer()

# Kanal kapanıp boşalana (null) kadar oku.
let received = []
while true
    let v = out.recv()
    if v == null
        break
    end
    received.push(v)
end
received.sort()            # iki üretici paralel; sırayı sabitle
print(received)            # ["A-0", "A-1", "A-2", "B-0", "B-1", "B-2"]
```

`closer` görevi, kanalı ne çok erken (üreticiler bitmeden) ne de hiç kapatmamak gibi iki tuzaktan kaçınmanın temiz yoludur: `await producers` tüm üreticilerin bittiğinden emin olur, ardından `out.close()` tüketiciye "akış bitti" der.

10.8 Veri Paralelliği: `pmap`, `pfilter`, `pforeach`, `preduce`

Çoğu paralellik ihtiyacı aslında tek bir kalıba iner: "şu listenin her elemanına şu işlemi uygula." Bunun için elle görev kurmanıza bile gerek yok; Cobra'nın **paralel yerleşikleri** işi tüm çekirdeklere otomatik dağıtır. Bunlar sıradan `map`/`filter`/`foreach`/`reduce` ile aynı çağrı biçimine sahiptir; tek farkları paralel koşmalarıdır.

```
def sq(n)
    return n * n
end
def is_even(n)
    return n % 2 == 0
end
def add(a, b)
    return a + b
end

print(pmap(sq, [1, 2, 3, 4]))           # [1, 4, 9, 16]  — sıra korunur
print(pfilter(is_even, [1, 2, 3, 4, 5, 6]))  # [2, 4, 6]
print(preduce(add, [1, 2, 3, 4, 5]))    # 15

# pforeach yan etki için: paylaşılan listeyi kilitle.
let out = []
let m = Mutex()
def collect(n)
    m.lock()
    out.push(n)
    m.unlock()
end
pforeach(collect, [10, 20, 30])
out.sort()
print(out)                              # [10, 20, 30]
```

Önemli iki garanti vardır:

1. **Sonuç sırası korunur.** `pmap` ve `pfilter`, elemanları paralel işlese de çıktıyı giriş sırasına göre düzenler. Yukarıda `pmap(sq, [1,2,3,4])` her zaman `[1, 4, 9, 16]` verir.
2. **İlk hata yayılır.** Bir eleman üzerindeki işlem hata atarsa, `map`'te olduğu gibi giriş sırasına göre **ilk** hata `pmap`'ten dışarı fırlatılır.

`pforeach` ise sonuç döndürmez; sadece her eleman için fonksiyonu çağırır (yan etki için). Yukarıdaki örnekte `collect` paylaşılan `out` listesine yazdığı için onu bir `Mutex` ile koruduk; aksi halde paralel `push`'lar yarışırdı. İşçiler paralel çalıştığından `out`'a ekleme sırası değişebilir, bu yüzden karşılaştırmadan önce `sort()`'ladık.

> **İpucu:** Pür (yan etkisiz) bir dönüşümünüz varsa `pmap` çoğu zaman elle görev kurmaktan hem daha kısa hem daha güvenlidir: kilit yok, kanal yok, sıra zaten garanti. Yan etki gerektiğinde (`pforeach`) paylaşılan durumu kilitlemeyi unutmayın.

Not: Paralelliğin bir başlangıç maliyeti vardır (iş parçacığı kurulumu). Çok küçük listelerde veya eleman başına işin çok hafif olduğu durumlarda düz `map` daha hızlı olabilir. `pmap`'i, eleman başına iş gerçekten ağır olduğunda tercih edin.

10.9 HTTP Sunucusu: İşleyiciler Paralel Koşar

Eşzamanlılık yalnızca sizin `async def`'lerinizle ortaya çıkmaz; bazı yerleşik modüller işleyicilerinizi *sizin adınıza* paralel çağırır. En önemlisi `http.serve`'dür: her gelen istek için işleyici fonksiyonunuzu ayrı bir görevde çağırır. Yani iki istek aynı anda gelirse, işleyiciniz **aynı anda iki kez** koşabilir.

Bunun pratik sonucu şudur: bir web sunucusunda paylaşılan durumu (istek sayacı, oturum tablosu, önbellek) tıpkı kendi görevlerinizde olduğu gibi bir kilitle korumalısınız.

```
import "http"

let m = Mutex()
let hits = 0

def handler(req)
    m.lock()
    hits = hits + 1            # paylaşılan sayaç: işleyiciler paralel koşar
    let n = hits
    m.unlock()
    return "istek #" + str(n)
end

# http.serve(":8080", handler)   # her istek kendi görevinde çalışır
```

Uyarı: `http.serve` işleyicilerinde paylaşılan bir dict'i veya sayacı kilitsiz değiştirmek, tam da yük altında (çok sayıda eşzamanlı istek) patlayan ve yerel testte gözden kaçan türden bir hatadır. Sunucu kodunda paylaşılan her değişebilir durumu kilitleyin.

10.10 Özet

Cobra'da eşzamanlılık bir yanılsama değil, gerçektir: **GIL yoktur** ve görevler birden çok CPU çekirdeğinde gerçekten paralel koşar. Bu güç, paylaşılan durumu açıkça senkronize etme sorumluluğuyla birlikte gelir. Bu bölümde gördüklerimiz:

- **Görevler ve `await`.** `async def` ile tanımlanan bir fonksiyonu çağırmak hemen bir görev döndürür ve gövde arka planda paralel koşmaya başlar. `await task` sonucu, `await [t1, t2, ...]` ise birden çok görevin sonuçlarını **giriş sırasıyla** toplar. Bir görevdeki hata, görevi başlattığınız yerde değil, **`await` noktasında** yeniden fırlatılır; `await [...]`'te ilk hata yayılır.
- **Yarış koşulları.** İki görev aynı değeri kilitsiz değiştirirse güncellemeler kaybolur (sayılar) ya da ölümcül hata oluşur (dict/struct alanları). Ayrı globaller güvenli, *paylaşılan* değerler değildir.
- **`Mutex`.** `lock` / `unlock` ile kritik bölgeyi tek görevle sınırlayın; `try_lock` engellemeden dener. Paylaşılan sayaç, liste ve dict'i bununla koruyun.
- **`RWMutex`.** Okuma ağırlıklı durum için: birçok `rlock`/`runlock` okuyucu aynı anda, ama tek `lock`/`unlock` yazıcı dışlayıcı.
- **`Channel`.** `Channel()` (tamponsuz, senkron el sıkışma) ve `Channel(n)` (tamponlu). `send` / `recv` / `close` / `try_send` / `try_recv` / `len(ch)`. Kapanıştan sonra `recv` önce tamponu boşaltır, sonra `null` döndürür — bu "akış bitti" sinyalidir. Kanalı **gönderen taraf** kapatır. İşçi havuzu deseninin temelidir.

- **`select`.** Birden çok kanal üzerinde bekleyip ilk hazır olanı seçer; sonuç `{index, value, ok, send}`. Zaman aşımı ve fan-in desenlerinin anahtarıdır. `select(ops, default)` engellemesizdir.
- **Paralel yerleşikler.** `pmap` / `pfilter` / `pforeach` / `preduce` veri paralelliğini tek satıra indirir: **sonuç sırası korunur**, **ilk hata yayılır**, ekstra sözdizimi gerekmez.
- **`http.serve`** işleyicileri paralel koşar — paylaşılan durumu kendi görevlerinizdeki gibi kilitleyin.

Bir sonraki adım: bu ilkelleri gerçek bir programda birleştirmek. Hesaplama ağırlıklı bir işiniz varsa önce `pmap`'i deneyin; üretici-tüketici akışınız varsa `Channel` ile bir işçi havuzu kurun; paylaşılan bir kaynağı birden çok görev değiştiriyorsa onu bir `Mutex` (ya da okuma ağırlıklıysa `RWMutex`) ardına alın. GIL'in olmaması, bu desenlerin gerçek hız kazancı vermesini sağlar — yeter ki paylaşılan durumu disipline koruyun.

Bölüm 11

Standart Kütüphane

Bir dili "kullanılabilir" yapan şey çoğu zaman çekirdek söz diziminden çok, yanında gelen pillerdir: dosya okumak, bir API yanıtını ayrıştırmak, süre ölçmek, metin doğrulamak, bir komut çalıştırmak. Cobra bu işlerin tamamını küçük ama özenle tasarlanmış bir dizi **yerleşik modülle** karşılar. Bu modüller diğer dillerdeki gibi paket yöneticisinden indirilmez; yorumlayıcının içinde hazır gelir ve adlarıyla içe aktarılır:

```
import "math"
print(math.sqrt(16))   # 4.0
```

Bir modülün üyelerine her zaman nokta ile erişilir (`math.sqrt`, `fs.read`, `json.parse`). Yerleşik modüller, aynı adı taşıyan yerel `.cobra` dosyalarına göre önceliklidir; yani `import "math"` her zaman gerçek matematik modülünü getirir. Bir modülün başarısızlıkları (var olmayan dosya, geçersiz JSON, ağ hatası) **yakalanabilir** çalışma zamanı hatalarıdır: `try/catch` ile ele alabilirsiniz.

Bu bölümde standart kütüphanenin tamamını tek tek geziyoruz. Her modülü oyuncak bir örnekle değil, karşınıza gerçekten çıkacak küçük bir mühendislik göreviyle tanıtacağız: bir log dosyasını işlemek, bir API yanıtını çözmek, bir e-posta adresini doğrulamak, bir alt süreç çalıştırmak. Tüm çalıştırılabilir örnekler gerçekten koşturulup çıktıları doğrulanmıştır; kod bloklarındaki `#` ile başlayan çıktı yorumları, programın gerçek çıktısıdır.

> **Not:** Cobra'da string biçimleme (f-string, interpolasyon) yoktur. Çıktıları `print` birden çok argümanla ("a:", x) veya `+` ile `str(...)` birleştirerek üretiriz. Bu bölümdeki örneklerde her iki üslubu da göreceksiniz.

11.1 fs — Dosya Sistemi

`fs` modülü dosya girişi/çıkışını sağlar. Tüm yollar çalışma dizinine (cwd) görelidir. Üyeler: `read`, `write`, `append`, `read_lines`, `write_lines`, `exists`, `size`, `remove`, `rename`, `mkdir`, `list_dir`, `is_dir`, `cwd`.

Tipik bir görev: bir uygulamanın ürettiği log dosyasını satır satır okuyup seviyelere göre özetlemek. Aşağıdaki program önce örnek bir log yazar, sonra onu okuyup `INFO`/`WARN`/`ERROR` sayımlarını çıkarır ve yalnızca hata satırlarını ayrı bir dosyaya ayıklar.

```
# Bir uygulama log dosyasini olusturup satir satir isleyelim.
import "fs"
```

```
let log_path = "/private/tmp/cobrabook/app.log"

# Once ornek bir log yazalim (gercek hayatta bunu uygulama uretir).
fs.write_lines(log_path, [
    "2026-06-27 09:01:12 INFO  baslatildi",
    "2026-06-27 09:01:13 WARN  yavas yanit",
    "2026-06-27 09:02:01 ERROR baglanti koptu",
    "2026-06-27 09:02:05 INFO  yeniden denendi",
    "2026-06-27 09:02:06 ERROR zaman asimi"
])

# Dosya gercekten olustu mu?
print("var mi:", fs.exists(log_path))      # var mi: true
print("boyut:", fs.size(log_path) > 0)     # boyut: true

# Seviyelere gore say.
let counts = {}
for line in fs.read_lines(log_path)
    let fields = line.split(" ")
    let level = ""
    for f in fields
        if f == "INFO" or f == "WARN" or f == "ERROR"
            level = f
        end
    end
    if level != ""
        counts[level] = counts.get(level, 0) + 1
    end
end

for pair in counts.items()
    print(pair[0] + ": " + str(pair[1]))
end
# INFO: 2
# WARN: 1
# ERROR: 2

# Sadece ERROR satirlarini ayri bir dosyaya ozetleyelim.
def has_error(l)
    return l.contains("ERROR")
end
let errors = filter(has_error, fs.read_lines(log_path))
fs.write(log_path + ".errors", "\n".join(errors))
print("hata satiri:", len(fs.read_lines(log_path + ".errors")))  # hata satiri: 2

# Temizlik.
fs.remove(log_path)
fs.remove(log_path + ".errors")
print("temizlendi:", not fs.exists(log_path))   # temizlendi: true
```

Burada `read_lines`/`write_lines` çiftinin dosyayı satır listesi olarak ele alması işimizi kolaylaştırdı; tüm içeriği tek string olarak isteseydik `read` ve `write` kullanırdık. `append` ise var olan bir dosyanın sonuna ekler (log biriktirmek için idealdir). Dizinlerle çalışmak için `mkdir`, `list_dir`, `is_dir` ve geçerli dizini veren `cwd`

vardır.

> **İpucu:** `filter` gibi yüksek mertebeli işlevlere fonksiyon adı geçirmek gerekir; Cobra'da `lambda` yoktur. Bu yüzden `has_error`'ı önce `def` ile tanımlayıp adını `filter`'a verdik.

> **Uyarı:** `fs.read` var olmayan bir dosyada hata fırlatır. Önce `fs.exists(path)` ile kontrol edin ya da çağrıyı `try/catch` içine alın.

11.2 math — Matematik

`math` sayısal hesaplamalar için temel sabitleri ve işlevleri sağlar: `pi`, `e`, `sqrt`, `abs`, `floor`, `ceil`, `round`, `pow`, `sin`, `cos`, `tan`, `log`, `exp`, `min`, `max`, `random`. Küçük bir ölçüm dizisinin ortalamasını ve standart sapmasını hesaplayalım.

```
# Bir olcum dizisinin temel istatistiklerini math ile hesaplayalim.
import "math"

let samples = [12.5, 9.0, 15.5, 11.0, 13.0]

# Ortalama
let total = 0.0
for x in samples
    total = total + x
end
let mean = total / float(len(samples))

# Standart sapma (nufus)
let var_sum = 0.0
for x in samples
    var_sum = var_sum + math.pow(x - mean, 2)
end
let stddev = math.sqrt(var_sum / float(len(samples)))

print("ortalama:", mean)                            # ortalama: 12.2
# round() tam sayi dondurur; ondalik basamak icin float'a bolmek gerekir.
print("std sapma:", math.round(stddev * 100) / 100.0) # std sapma: 2.16
print("en kucuk:", math.min(samples))               # en kucuk: 9.0
print("en buyuk:", math.max(samples))               # en buyuk: 15.5

# Banker yuvarlamasi: tam ortadaki 0.5 en yakin CIFT sayiya gider.
print("round(2.5):", math.round(2.5))               # round(2.5): 2
print("round(3.5):", math.round(3.5))               # round(3.5): 4
print("pi:", math.round(math.pi * 1000) / 1000.0)   # pi: 3.142
```

İki noktaya dikkat edin. Birincisi, ortalamayı hesaplarken `float(len(...))` kullandık: Cobra'da iki tam sayının bölümü **tam sayı bölmesidir** (`5 / 2`, 2'dir), bu yüzden ondalık sonuç istiyorsak en az bir tarafı `float` yapmalıyız. İkincisi, `math.round` **banker yuvarlaması** kullanır: tam ortadaki bir değer (`2.5`) en yakın *çift* sayıya yuvarlanır, yani `2.5` → `2` ama `3.5` → `4`. Bu, büyük veri kümelerinde yuvarlama sapmasını azaltan standart davranıştır.

`math.min` ve `math.max` hem ayrı argümanlar (`math.max(3, 7, 1)`) hem de tek bir liste (`math.max(samples)`) ile çağrılabilir. Rastgele sayı için `math.random()` `[0, 1)` aralığında bir float verir.

> **Not:** `math.round` bir **tam sayı** döndürür. Belirli sayıda ondalık basamağa yuvarlamak istediğinizde örnekteki gibi önce ölçekleyip (`* 100`), yuvarlayıp, ardından bir `float`'a bölün (`/ 100.0`).

11.3 os — İşletim Sistemi

`os` modülü süreç ortamını açar: `args` (betik yolu + argümanlar), `env`, `setenv`, `platform`, `time`. Bir komut satırı aracının yapacağı gibi argümanları ve ortam değişkenlerini okuyalım.

```
# os ile calisma ortamini ve komut satiri argumanlarini okuyalim.
import "os"

# os.args() bir liste dondurur; [0] calistirilan betigin yolu, sonrasi
# kullanici argumanlaridir.
let args = os.args()
print("betik:", args[0].endswith(".cobra"))   # betik: true
print("platform:", os.platform() == "darwin" or os.platform() == "linux" or os.platform() == "windows")  #

# Ortam degiskeni: ikinci arguman varsayilan deger.
let mode = os.env("APP_MODE", "development")
print("mod:", mode)                            # mod: development

# Bir degisken atayip geri okuyalim.
os.setenv("APP_MODE", "production")
print("yeni mod:", os.env("APP_MODE", "development"))  # yeni mod: production

# os.time() Unix epoch saniyesini ondalik (float) olarak dondurur.
print("zaman makul mu:", os.time() > 1700000000.0)     # zaman makul mu: true
```

`os.args` ve `os.platform` birer **işlevdir**; parantezle çağrılmaları gerekir (`os.args()`, `os.platform()`). `os.env(name, default?)` ikinci argüman olarak bir varsayılan alır; değişken tanımlı değilse onu döndürür, varsayılan da verilmemişse `null` verir. Bu, yapılandırmayı ortamdan okuyan programlar için ergonomiktir: `os.env("PORT", "8080")`.

> **İpucu:** `os.time()` Unix epoch'tan bu yana geçen saniyeyi *float* olarak verir; takvim biçimlemesi için onu `time.format`'a geçirebilirsiniz (bkz. bir sonraki kesitten önceki `time` modülü).

11.4 json — JSON

`json` modülünün iki işlevi vardır: `parse` (metin → değer) ve `stringify(value, indent?)` (değer → metin). Çözümlemede dict **anahtar sırası korunur** ve tam sayılarla ondalıklar ayırt edilir. Bir HTTP istemcisinden gelmiş gibi bir API yanıtını ayrıştıralım.

```
# json ile bir API yanitini ayristirip yeniden seri hale getirelim.
import "json"

# Bir HTTP istemcisinden gelmis gibi bir yanit metni.
let response = "{\"user\": {\"id\": 42, \"name\": \"Ada\", \"active\": true}, \"roles\": [\"admin\", \"edi

let data = json.parse(response)
print("ad:", data["user"]["name"])       # ad: Ada
print("rol sayisi:", len(data["roles"])) # rol sayisi: 2
print("ilk rol:", data["roles"][0])      # ilk rol: admin
print("aktif mi:", data["user"]["active"])  # aktif mi: true

# Bir alan ekleyip geri stringe cevirelim. Anahtar sirasi korunur.
data["user"]["verified"] = true
let out = json.stringify(data["user"])
print(out)   # {"id":42,"name":"Ada","active":true,"verified":true}

# Girintili (pretty) cikti.
print(json.stringify(["a", "b"], 2))
# [
#   "a",
#   "b"
# ]

# String olmayan anahtarlar stringe doner.
print(json.stringify({1: "bir", 2: "iki"}))  # {"1":"bir","2":"iki"}
```

Ayrıştırılan JSON nesneleri Cobra'nın olağan dict ve listelerine dönüşür; iç içe yapılara `data["user"]["name"]` gibi zincirlenerek erişirsiniz. `stringify`'a ikinci argüman olarak bir girinti sayısı verirseniz okunabilir (pretty) çıktı alırsınız; vermezseniz boşluksuz kompakt biçim üretilir. Cobra dict anahtarları tam sayı veya boolean da olabildiğinden, bunlar JSON'a çevrilirken stringe dönüştürülür (`{1: ...}` → `{"1": ...}`), çünkü JSON yalnızca string anahtar tanır.

> **Uyarı:** `json.parse` geçersiz JSON'da hata fırlatır. Dışarıdan gelen veriyi her zaman `try/catch` ile sarın:
>
> ```
> try
>     let data = json.parse(text)
> catch err
>     print("gecersiz JSON:", err)
> end
> ```

11.5 time — Zaman

`time` hem süre ölçümü hem de takvim işlemleri sağlar: `now`, `clock`, `sleep`, `format`, `parse`, `parts`. Süre ölçmek için **tek artan (monotonik)** `clock`'u kullanın; takvim işlevleri (`format`/`parse`/`parts`) UTC'de çalışır ve strftime yönergelerini destekler (`%Y %m %d %H %M %S %a %A %b %B %p %I %j %y %%`).

```
# time ile sure olcelim ve zaman damgalarini bicimlendirelim.
```

```
import "time"

# clock() tek artan (monotonik) bir saattir; sure olcumu icin idealdir.
let start = time.clock()
let total = 0
for i in range(1000000)
    total = total + i
end
let elapsed = time.clock() - start
print("toplam dogru mu:", total == 499999500000)  # toplam dogru mu: true
print("sure olculdu mu:", elapsed >= 0.0)          # sure olculdu mu: true

# Sabit bir epoch zaman damgasini bicimlendirelim (UTC).
# 1700000000 = 2023-11-14 22:13:20 UTC
let ts = 1700000000
print(time.format(ts, "%Y-%m-%d %H:%M:%S"))   # 2023-11-14 22:13:20

# Bir tarihi ayristirip parcalarina ayiralim.
let parsed = time.parse("2026-06-27", "%Y-%m-%d")
let p = time.parts(parsed)
print("yil:", p["year"], "ay:", p["month"], "gun:", p["day"])  # yil: 2026 ay: 6 gun: 27
print("haftanin gunu (Pzt=0):", p["weekday"])  # haftanin gunu (Pzt=0): 5
```

`clock` mutlak takvim zamanı değil, yalnızca *geçen süreyi* ölçmek içindir: ölçümün başında ve sonunda çağırıp farkı alırsınız. Mutlak zaman damgası için `now()` vardır. `format(ts, fmt)` bir epoch zaman damgasını okunabilir metne çevirir; `parse(s, fmt)` bunun tersini yapar ve `parts(ts)` zaman damgasını year/month/day/hour/minute/second/weekday/yearday alanlarına ayıran bir dict döndürür. Haftanın günü Pazartesi = 0 konvansiyonuyla verilir, yani 27 Haziran 2026 bir Cumartesi'dir (5).

> **İpucu:** Bir işlemi belirli bir süre duraklatmak için `time.sleep(saniye)` kullanın; argüman bir float'tır, yani `time.sleep(0.05)` 50 milisaniye bekler. Bunu özellikle eşzamanlı örneklerde bir sunucunun ayağa kalkmasını beklemek için kullanacağız.

11.6 re — Düzenli İfadeler

`re` modülü metin eşleştirme ve çıkarma sağlar: `matches`, `find`, `find_all`, `groups`, `replace`, `split`. Altında RE2 motoru yatar; bu, doğrusal zaman garantisi verir ama **geri başvuru (backreference) ve lookaround desteklemez**. Klasik bir doğrulama görevi olan e-posta kontrolüyle başlayalım.

```
# re ile e-posta dogrulama ve alan cikarma yapalim.
import "re"

let email_pat = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"

let addresses = ["ada@example.com", "gecersiz@", "bob.smith@mail.co.uk"]
for addr in addresses
    print(addr + " -> " + str(re.matches(email_pat, addr)))
end
# ada@example.com -> true
```

```
# gecersiz@ -> false
# bob.smith@mail.co.uk -> true

# Bir log satirindan tarih ve seviye alanlarini gruplarla cikaralim.
let line = "2026-06-27 ERROR baglanti koptu"
let g = re.groups("(\\d{4}-\\d{2}-\\d{2})\\s+(\\w+)", line)
print("tarih:", g[1], "seviye:", g[2])  # tarih: 2026-06-27 seviye: ERROR

# Metindeki tum sayilari bulalim.
print(re.find_all("\\d+", "3 elma, 12 armut, 7 erik"))  # ["3", "12", "7"]

# Birden cok boslugu tek bosluga indirgeyelim.
print(re.replace("\\s+", "a   b     c", " "))  # a b c

# CSV benzeri bir satiri virgul VEYA noktali virgulle bolelim.
print(re.split("[,;]", "ad;soyad,yas"))  # ["ad", "soyad", "yas"]
```

İşlevlerin imzalarına dikkat edin: `matches`, `find`, `find_all`, `groups` ve `split` desen ile metni bu sırayla alır (`re.matches(desen, metin)`); `replace` ise üç argüman alır: `re.replace(desen, metin, yeni)`. `groups` ilk eşleşmeyi `[tüm_eşleşme, grup1, grup2, ...]` listesi olarak döndürür; bu yüzden ilk yakalama grubuna `g[1]` ile eriştik. Bir gruba hiç katılım olmazsa o konumda `null` bulunur. `replace` içinde `$1`, `$2` ile yakalama gruplarına başvurabilirsiniz.

> **Uyarı:** Cobra string'lerinde `\d`, `\s`, `\w` gibi yönergeleri yazarken ters eğik çizgiyi **kaçırmanız** gerekir: `"\\d+"`. Aksi halde string ham `\d` içermez. Ayrıca RE2 olduğu için `(?=...)` (lookahead) veya `\1` (backreference) gibi desenler **çalışmaz** — bunlara ihtiyaç duyarsanız mantığı kod tarafında ele almalısınız.

> **Not:** Cobra string'leri bayt-indekslidir; bir alt diziyi `s[i]` döngüsüyle elle kesmek UTF-8 metni bozabilir. Metin ayıklama için döngü yerine `re` işlevlerini (veya rune-güvenli dilim `s[a:b]`'yi) tercih edin.

11.7 http — HTTP İstemci ve Sunucu

`http` hem bir istemci hem de bir sunucu içerir.

İstemci tarafı: `get(url, headers?)`, `post(url, body, headers?)` ve genel `request(method, url, body?, headers?)`. Hepsi `{status, body, headers}` biçiminde bir dict döndürür. Ağ hatası yakalanabilir bir hatadır; 2xx olmayan durum kodları ise hata sayılmaz, sonucun bir parçasıdır.

```
import "http"
import "json"

# Bir JSON API'sini cagiralim (cikti ortama/aga baglidir).
let r = http.get("https://api.example.com/users/42")
if r["status"] == 200
    let user = json.parse(r["body"])
    print("ad:", user["name"])
else
    print("hata kodu:", r["status"])
end
```

> **Not:** Yukarıdaki `http.get` örneği gerçek bir ağ çağrısı yaptığından çıktısı çalıştırıldığı ortama bağlıdır; bu yüzden burada sabit bir `#` çıktı göstermiyoruz. Aşağıdaki sunucu örneği ise tamamen yerel olduğundan gerçekten çalıştırılıp doğrulanmıştır.

Sunucu tarafı: `serve(addr, handler, limit?)` belirtilen adreste bloklar ve her istek için Cobra işleyicinizi `{method, path, query, body, headers}` dict'i ile çağırır. İşleyici ya bir string (200 olur) ya da bir `{status?, body?, headers?}` dict'i döndürür. **İşleyiciler paraleldir** (istek başına ayrı bir görev), bu yüzden paylaşılan durumu açıkça kilitlemeniz gerekir. İsteğe bağlı `limit` verilirse, sunucu o kadar istek karşıladıktan sonra geri döner — bu, sonlu örnekleri test etmek için pratiktir.

Aşağıdaki program tek bir betikte hem küçük bir JSON API sunucusu başlatır hem de ona istemciyle istek atar. Paylaşılan `hits` sayacını bir `Mutex` ile koruduğumuza dikkat edin.

```
# http ile kucuk bir JSON API sunucusu kuralim ve istemciyle cagiralim.
import "http"
import "json"
import "time"

# Paylasilan sayac; isleyiciler PARALEL calistigi icin Mutex ile koruyoruz.
let hits = 0
let m = Mutex()

def handler(req)
    m.lock()
    hits = hits + 1
    let n = hits
    m.unlock()

    if req["path"] == "/ping"
        return "pong"
    end
    if req["path"] == "/stats"
        # Dict dondurursek JSON govdesi olarak elle seri hale getiririz.
        return {"status": 200,
                "body": json.stringify({"hits": n}),
                "headers": {"Content-Type": "application/json"}}
    end
    return {"status": 404, "body": "bulunamadi"}
end

# limit=3: 3 istek karsilandiktan sonra serve() geri doner (test icin pratik).
async def run_server()
    http.serve("127.0.0.1:8088", handler, 3)
end

let srv = run_server()
time.sleep(0.2)   # sunucunun ayaga kalkmasini bekle

# Istemci tarafi.
let base = "http://127.0.0.1:8088"
let r1 = http.get(base + "/ping")
print("ping:", r1["status"], r1["body"])   # ping: 200 pong

let r2 = http.get(base + "/stats")
```

```
let stats = json.parse(r2["body"])
print("stats hits:", stats["hits"])         # stats hits: 2

let r3 = http.get(base + "/yok")
print("yok:", r3["status"], r3["body"])     # yok: 404 bulunamadi

await srv
```

Sunucuyu bir `async def` içinde başlatıp `time.sleep` ile kısa bir süre bekledik; böylece istemci ilk isteği atmadan önce sunucu dinlemeye başladı. İşleyicideki `hits` sayacı her istekte arttığından `/stats` çağrıldığında değeri 2'dir (önce `/ping`, sonra `/stats`). Bir işleyici hata fırlatırsa Cobra bunu 500'e çevirir ve sunucu çalışmaya devam eder.

> **Uyarı:** İşleyiciler paralel koştuğu için paylaşılan herhangi bir değişkeni (sayaç, önbellek, dict) bir `Mutex` ile koruyun. Bir dict'i kilitsiz eşzamanlı değiştirmek ölümcül bir hatadır.

11.8 proc — Süreçler

`proc` harici komutlar çalıştırır. İki işlevi vardır: `run(cmd, args?, options?)` komutu doğrudan (kabuk yorumlaması olmadan) çalıştırır; `shell(cmdline, options?)` ise bir kabuk satırını yorumlar (boru/pipe, değişken, joker karakter çalışır). İkisi de `{code, stdout, stderr}` döndürür. `options` olarak `input` (standart girdi), `dir` (çalışma dizini) ve `env` verilebilir.

```
# proc ile harici komutlar calistiralim.
import "proc"

# run(cmd, args) komutu kabuk yorumlamasi olmadan dogrudan calistirir.
let r = proc.run("echo", ["merhaba", "dunya"])
print("cikis kodu:", r["code"])          # cikis kodu: 0
print("stdout:", r["stdout"].strip())    # stdout: merhaba dunya

# shell(...) bir kabuk satirini yorumlar; boru (pipe) ve degiskenler calisir.
let s = proc.shell("printf 'a\\nb\\nc\\n' | wc -l")
print("satir sayisi:", s["stdout"].strip())  # satir sayisi: 3

# Standart girdiyi options uzerinden besleyelim.
let up = proc.run("tr", ["a-z", "A-Z"], {"input": "cobra"})
print("buyuk harf:", up["stdout"].strip())   # buyuk harf: COBRA

# Basarisiz bir komutun cikis kodu sonuctur, hata firlatmaz.
let bad = proc.shell("exit 3")
print("hatali kod:", bad["code"])        # hatali kod: 3
```

`run` ile `shell` arasındaki ayrım güvenlik açısından önemlidir: dışarıdan gelen girdiyi bir komuta argüman olarak geçirecekseniz, kabuk enjeksiyonundan kaçınmak için `run`'ı argüman listesiyle tercih edin (`proc.run("grep", [desen, dosya])`). Bir komutun başlatılamaması (örneğin var olmayan program) yakalanabilir bir hatadır; ama sıfırdan farklı bir çıkış kodu hata değil, normal bir sonuçtur — yukarıdaki `exit 3` örneğinde olduğu gibi `code` alanından okunur.

İpucu: Bir komutun başarısını her zaman `code` alanından kontrol edin; bazı araçlar tanı mesajlarını `stderr`'e yazar ama yine de 0 ile çıkar, bazılarıysa tersini yapar.

11.9 net — TCP Soketleri

`net` ham TCP soketleri sağlar (opak tanıtıcılar üzerinden): `connect(addr)`, `listen(addr)`, `accept(listener)`, `send(conn, data)`, `recv(conn, max)` (EOF'ta `null`), `recv_line(conn)`, `close(handle)`. Bu, HTTP'nin altındaki katman gerektiğinde veya kendi protokolünüzü yazarken işe yarar. Minik bir yankı (echo) sunucusuyla istemcisini tek betikte kuralım.

```
# net ile minik bir TCP yanki (echo) sunucusu ve istemcisi.
import "net"
import "time"

let addr = "127.0.0.1:8099"

# Sunucu: bir baglanti kabul eder, bir satir okur, geri yollar, kapatir.
async def server()
    let listener = net.listen(addr)
    let conn = net.accept(listener)
    let line = net.recv_line(conn)
    net.send(conn, "yanki: " + line)
    net.close(conn)
    net.close(listener)
end

let srv = server()
time.sleep(0.1)   # sunucunun dinlemeye baslamasini bekle

# Istemci: baglanir, bir satir yollar, yaniti okur.
let c = net.connect(addr)
net.send(c, "merhaba\n")
let reply = net.recv(c, 1024)
print(reply.strip())   # yanki: merhaba
net.close(c)

await srv
```

Sunucuyu eşzamanlı bir görev (`async def`) olarak başlattık; `accept` bir bağlantı gelene kadar bloklar, bu yüzden onu ayrı bir görevde çalıştırıp ana akışta istemciyi koşturuyoruz. `recv(conn, max)` en fazla `max` bayt okur ve akış bittiğinde `null` döndürür; satır temelli protokoller için `recv_line` daha pratiktir. Tanıtıcılar (bağlantılar ve dinleyiciler) iş bitince `close` ile kapatılmalıdır.

Not: `net` katmanı baytlar ve string'lerle çalışır; uygulama düzeyinde bir protokol (çerçeveleme, uzunluk önekleri vb.) sizin sorumluluğunuzdadır. Çoğu durumda doğrudan `http` modülünü kullanmak daha kolaydır; `net`'i yalnızca özel bir protokole ihtiyaç duyduğunuzda tercih edin.

11.10 runtime — Çalışma Zamanı ve Bellek

Cobra belleği elle serbest bırakmaz; bir çöp toplayıcı (GC) ölü değerleri otomatik geri kazanır. `runtime` modülü bu mekanizmaya profil çıkarma ve kıyaslama amacıyla pencere açar: `gc()` (zorla toplama), `free_os_memory()` (topla, sonra belleği işletim sistemine geri ver), `mem()` (bir istatistik dict'i), `set_gc_percent(pct)` (çöp toplama yüzdesini ayarlar, önceki değeri döndürür), `num_goroutines()` (çalışan görev sayısı), `num_cpu()`.

```
# runtime ile bellek kullanimini olcelim.
import "runtime"

print("cpu cekirdegi:", runtime.num_cpu() >= 1)   # cpu cekirdegi: true

# Olcumden once temiz bir taban cizgisi alalim.
runtime.gc()
let before = runtime.mem()["heap_alloc"]

# Bol bol bellek tuketen bir is yapalim: 100 bin elemanli liste.
let big = []
for i in range(100000)
    big.push(i * i)
end

let after = runtime.mem()["heap_alloc"]
print("liste olustu:", len(big) == 100000)  # liste olustu: true
print("bellek artti mi:", after > before)    # bellek artti mi: true

# GC istatistiklerine bakalim.
let m = runtime.mem()
print("gc sayisi makul:", m["num_gc"] >= 0)  # gc sayisi makul: true

# Cop toplama yuzdesini ayarla: onceki degeri dondurur.
let prev = runtime.set_gc_percent(200)
print("onceki yuzde bir tam sayi:", type(prev) == "INTEGER")  # onceki yuzde bir tam sayi: true
runtime.set_gc_percent(prev)
```

`mem()` zengin bir dict döndürür: `alloc`, `total_alloc`, `sys`, `heap_alloc`, `heap_sys`, `heap_objects`, `num_gc`, `pause_total_ns`, `gc_cpu_fraction`. Bir ölçüm yaparken tipik akış, ölçümün başında `gc()` çağırıp temiz bir taban çizgisi almak, işi yapmak ve `heap_alloc` farkına bakmaktır. `num_goroutines()` çalışan görev/işleyici sayısını verir (Cobra'da GIL olmadığı için her görev kendi iş parçacığında gerçekten paralel koşar); `num_cpu()` çekirdek sayısını verir ve paralel işleri boyutlandırmak için kullanışlıdır.

> **İpucu:** Bellek ölçümleri doğası gereği gürültülüdür; GC arka planda istediği zaman çalışabilir. Karşılaştırma yaparken tek bir okumaya değil, eğilime bakın ve ölçümden hemen önce `runtime.gc()` ile durumu sabitleyin.

11.11 test — Birim Test Çerçevesi

Cobra'nın içinde küçük bir xUnit tarzı test çerçevesi gelir. Bir işlevi test olarak işaretlemek için `@test.it` dekoratörünü kullanırsınız (ya da özel adlı bir test için `case(name, fn)` ile kaydedersiniz). İddialar başarısızlıkta hata fırlatır; `run()` hepsini çalıştırır, bir rapor basar ve `{passed, failed, total, ok}` dict'i

döndürür. İddialar: `assert(cond, msg?)`, `assert_eq(actual, expected, msg?)` (== gibi değer eşitliği), `assert_neq(a, b, msg?)` ve `assert_raises(fn, msg?)` (`fn` hata fırlatırsa geçer).

İki küçük işlevi (`slugify` ve `divide`) test eden gerçek bir test paketi yazalım.

```
# test modulu ile kucuk bir test paketi yazalim.
import "test"

# Test edilecek kod (gercekte ayri bir modulden import edilirdi).
def slugify(s)
    return s.strip().lower().replace(" ", "-")
end

def divide(a, b)
    if b == 0
        throw "sifira bolme"
    end
    return a / b
end

# @test.it bir fonksiyonu test olarak isaretler.
@test.it
def slug_bosluk_tireye_doner()
    test.assert_eq(slugify("  Merhaba Dunya "), "merhaba-dunya")
end

@test.it
def slug_zaten_temiz()
    test.assert_eq(slugify("kod"), "kod")
    test.assert_neq(slugify("A B"), "A B")
end

@test.it
def bolme_dogru()
    test.assert_eq(divide(10, 2), 5)
    test.assert(divide(9, 3) == 3, "9/3 == 3 olmali")
end

@test.it
def sifira_bolme_firlatir()
    # assert_raises, fonksiyon cagrildiginda hata firlatirsa gecer.
    def call_zero()
        return divide(1, 0)
    end
    test.assert_raises(call_zero)
end

# case() ile ozel adli bir test de kaydedebiliriz.
def negatif_kontrol()
    test.assert(divide(-6, 2) == -3)
end
test.case("negatif bolme", negatif_kontrol)

let summary = test.run()
print("ozet:", summary["passed"], "/", summary["total"], "ok:", summary["ok"])
```

Programın çıktısı şöyledir:

```
ok    slug_bosluk_tireye_doner
ok    slug_zaten_temiz
ok    bolme_dogru
ok    sifira_bolme_firlatir
ok    negatif bolme

5 passed, 0 failed, 5 total
ozet: 5 / 5 ok: true
```

`assert_eq` **değer eşitliği** kullanır (tıpkı == gibi), bu yüzden iç içe liste ve dict'leri derinlemesine karşılaştırır. `assert_raises`'e bir **işlev** geçeriz; çerçeve onu sizin yerinize çağırır ve bir hata fırlatmasını bekler — bu yüzden `divide(1, 0)`'ı doğrudan değil, `call_zero` adlı küçük bir sarmalayıcı işlevle veriyoruz (Cobra'da `lambda` olmadığını hatırlayın). Bir iddia başarısız olursa çerçeve onu yakalar, ilgili testi başarısız işaretler ama diğer testleri çalıştırmaya devam eder; sonunda dönen dict'teki `ok` alanı tüm paketin geçip geçmediğini özetler.

> **İpucu:** `run()`'ın döndürdüğü `ok` alanını CI/sürekli entegrasyon betiklerinde kullanın: `false` ise çıkış kodunu sıfırdan farklı yaparak yapıyı başarısız sayabilirsiniz.

11.12 Özet

Bu bölümde Cobra'nın standart kütüphanesinin tamamını gerçek görevler üzerinden gezdik:

- **fs** — dosya ve dizin işlemleri; bir log dosyasını satır satır işledik.
- **math** — sabitler ve sayısal işlevler; banker yuvarlamasına ve tam sayı bölmesi tuzağına dikkat çektik.
- **os** — argümanlar, ortam değişkenleri, platform ve zaman; `args`/`platform` birer işlevdir.
- **json** — anahtar sırasını koruyarak ayrıştırma ve seri hale getirme; bir API yanıtını çözdük.
- **time** — süre ölçümü için monotonik `clock` ve UTC takvim işlevleri (strftime yönergeleri).
- **re** — RE2 tabanlı eşleştirme; e-posta doğrulama ve alan çıkarma yaptık (geri başvuru/lookaround yok).
- **http** — `{status, body, headers}` döndüren istemci ve paralel işleyicili bir sunucu; küçük bir JSON API kurduk.
- **proc** — `run`/`shell` ile alt süreçler; `{code, stdout, stderr}`.
- **net** — ham TCP; minik bir yankı sunucusu yazdık.
- **runtime** — GC ve bellek muhasebesi; bir liste oluştururken bellek artışını ölçtük.
- **test** — yerleşik birim test çerçevesi; `@test.it`, `case` ve iddialarla gerçek bir test paketi yazdık.

Hatırlanacak ortak kalıplar: yerleşik modüller adlarıyla içe aktarılır ve üyelerine nokta ile erişilir; modül hataları yakalanabilir, bu yüzden dış dünyayla (dosya, ağ, JSON) konuşan kodu `try/catch` ile sarın; ve paralellik söz konusu olduğunda (HTTP işleyicileri, görevler) paylaşılan durumu bir `Mutex` ile koruyun. Bu modüller, Cobra'nın sade söz dizimiyle birleştiğinde, kısa ve okunaklı ama gerçek işler yapan programlar yazmanıza yeter.

Bölüm 12

Dilin İç Yapısı — Lexer'dan Sanal Makineye

Önceki bölümlerde Cobra'yı *kullanan* tarafta durduk: değişkenler, fonksiyonlar, yapılar, modüller. Bu son bölümde masanın diğer tarafına geçiyoruz. Cobra kaynak kodunu çalıştırılabilir davranışa çeviren makineyi söküp inceleyeceğiz: kaynak metin nasıl önce kelimelere, sonra bir ağaca, en sonunda da çalışan bir programa dönüşüyor?

Cobra'nın ilginç bir mimari kararı var: tek değil, **iki** yürütme motoru var ve ikisi de aynı dili çalıştırıyor:

- **Ağaç-yürüten motor** — varsayılan. Soyut sözdizim ağacını (AST) doğrudan gezerek değerlendirir. Yazması ve okuması en kolay olan motor budur.
- **Derleyici + sanal makine (VM)** — `--vm` bayrağıyla devreye girer. Önce kaynağı kompakt bir **bayt-koduna** derler, sonra bunu bir yığın (stack) tabanlı sanal makinede çalıştırır. Kabaca birkaç kat daha hızlıdır.

Bu bölüm boyunca küçük bir Cobra programını alıp her iki motorun boru hattında adım adım izleyeceğiz. Anlatılan her şey gerçek araç çıktısından gelir; örnekleri kendiniz de üretebilirsiniz.

İzleyeceğimiz program iki satır:

```
let n = 2
print(n + 1)
```

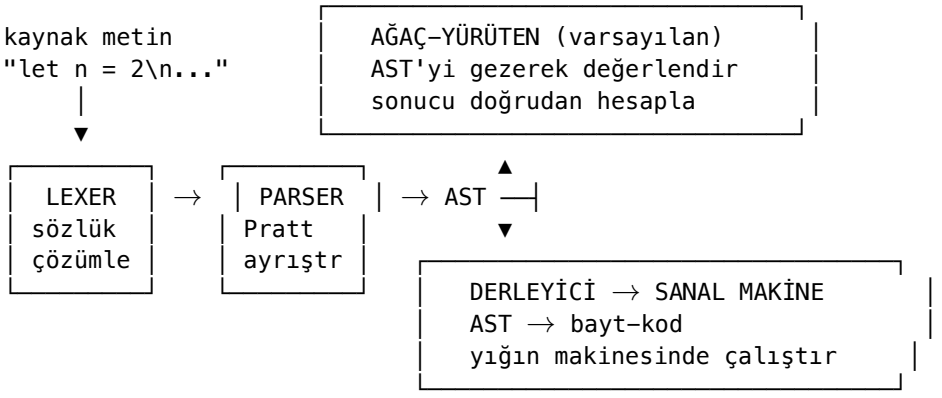
İki motoru da çalıştırın; aynı sonucu verirler:

```
$ cobra ornek.cobra          # ağaç-yürüten
3
$ cobra --vm ornek.cobra     # bayt-kod VM
3
```

> **Not:** "Aynı davranış" bir tesadüf değil, mimari bir zorunluluktur. Aritmetik, doğruluk (truthiness), indeksleme, dilimleme, metot çağrısı ve gömülü fonksiyonlar gibi *anlambilim* tek bir yerde tanımlıdır. İki motor da bu tek tanıma başvurur. Böylece doğal olarak hizalı kalırlar — birinin verdiği yanıtı diğeri de verir.

12.1 Genel Mimari — Kaynaktan Sonuca

Her iki motor da aynı ilk iki aşamayı paylaşır. Kaynak metin önce bir **token** akışına, sonra bir **soyut sözdizim ağacına (AST)** çevrilir. Yollar ancak bundan sonra ayrılır:



Hangi yolun seçileceğine komut satırı bayrağı karar verir. Aynı kaynak, seçiminize göre farklı boruya yollanır:

- `cobra dosya.cobra` — lexer + parser + **ağaç-yürüten**.
- `cobra --vm dosya.cobra` — lexer + parser + **derleyici + VM**.
- `cobra --tokens dosya.cobra` — sadece **lexer**'ın çıktısını basar.
- `cobra --ast dosya.cobra` — lexer + parser'ın ürettiği **AST**'yi basar.

Son iki bayrak bize bu bölümde kullanacağımız iki teşhis aracını verir: `--tokens` lexer'ın gördüğünü, `--ast` ise parser'ın kurduğu ağacı gösterir. İkisini de boyunca kullanacağız.

12.2 Lexer — Metni Token'lara Bölmek

Lexer (sözlük çözümleyici) işini tek bir cümleyle özetleyebiliriz: ham karakter akışını anlamlı parçalara — **token**'lara — ayırmak. Bir token üç bilgi taşır:

- **Tür:** token'ın ne olduğu — `IDENT` (tanımlayıcı), `INT` (tamsayı), `PLUS` (artı), `NEWLINE` (satır sonu) gibi.
- **Metin:** token'ın kaynaktaki gerçek karakterleri — `"n"`, `"2"`, `"+"`.
- **Satır:** hata mesajlarında kullanılmak üzere token'ın geçtiği kaynak satırı.

Lexer, girdi metninde tek bir okuma kafası gibi soldan sağa ilerler. O anki karaktere bakar, gerekirse bir-iki karakter ileriye göz atar, ve bir token üretip kafasını ilerletir. Bu işlemi metin bitene kadar tekrarlar. Ürettiği her token bir sonraki aşamaya — parser'a — akar.

Lexer'ın iki ilginç iç durumu vardır ve Cobra'nın noktalı virgülsüz, girinti-dostu sözdiziminin sırrı bunlarda saklıdır: **henüz hiç gerçek token üretildi mi?** ve **şu an kaç açık parantezin içindeyiz?** İkisine de birazdan döneceğiz.

12.2.1 Boşluk ve yorumların atlanması

Lexer bir token üretmeden önce anlamsız karakterleri sessizce yutar: boşluklar, sekmeler ve **yorumlar**. Cobra'da yorum `#` ile başlar ve satır sonuna kadar sürer; lexer `#` görür görmez satırın geri kalanını atlar. Yorumlar hiçbir zaman token'a dönüşmez — parser onların varlığını bile bilmez.

12.2.2 İki karakterli operatörler: ileriye göz atmak

`=` ile `==`, ya da `+` ile `+=` aynı karakterle başlar. Lexer hangisinin kastedildiğini **bir karakter ileriye göz atarak** anlar. Önce o anki karakteri görür (`+`), sonra imleci ilerletmeden bir sonrakine bakar: eğer `=` ise bu bir `+=` token'ıdır ve iki karakter birden tüketilir; değilse sade bir `+`'tır.

Aynı önden-bakış deseni `==`, `!=`, `<=`, `>=`, `-=`, `*=`, `/=`, `%=` için de çalışır. Yanlış tahminin maliyeti yoktur, çünkü ileriye göz atmak hiçbir şeyi tüketmez — sadece sıradaki karaktere bakar.

12.2.3 NEWLINE: noktalı virgül yerine satır sonu

Cobra'da deyimler noktalı virgülle değil, **satır sonuyla** biter. Lexer bunu özel bir `NEWLINE` token'ı üreterek halleder. Ama iki ince ayar vardır.

Birincisi — parantez içindeki satır sonları önemsizdir. Lexer kaç açık parantezin (`(`, `[`, `{`) içinde olduğunu bir sayaçla takip eder. Her açılışta sayaç artar, her kapanışta azalır. Sayaç sıfırdan büyükken bir satır sonu görülürse, bu yok sayılır:

```
açılış  (  [  {   →  sayaç++   →  içerideki '\n' yok sayılır (boşluk gibi)
kapanış )  ]  }   →  sayaç--
sayaç == 0 iken   →  '\n' bir NEWLINE token'ı üretir (deyim sınırı)
```

Bu sayede liste ve sözlük literalleri ile fonksiyon çağrıları birden çok satıra yayılabilir; parser yine tek bir mantıksal deyim görür:

```
let renkler = [
    "kirmizi",
    "yesil",
    "mavi"
]
```

İkincisi — baştaki satır sonları yutulur. İlk gerçek token'dan *önce* gelen boş satırların parser için bir anlamı yoktur (henüz sonlandıracak bir deyim yoktur). Lexer "ilk gerçek token üretildi mi?" durumunu tutarak bu baştaki satır sonlarını sessizce atar. Ayrıca arka arkaya gelen boş satırlar tek bir `NEWLINE`'a indirgenir.

12.2.4 Sayılar ve `m` ondalık eki

Lexer bir rakam görünce ardışık rakamları okuyup bir sayı token'ı kurar. *Her iki yanında da rakam olan* bir nokta varsa bu sayı bir `FLOAT` olur: `1.5` evet, ama `1.` veya `.5` hayır. Sayının sonunda bir `m` harfi varsa ve onu bir harf ya da rakam izlemiyorsa, bu bir **tam ondalık** (DECIMAL) literalidir — para hesapları için kullanılan `19.90m` gibi.

> **İpucu:** `m` ekinin koşulu önemlidir: `m`'den sonra harf veya rakam gelmemelidir. `42m` bir DECIMAL'dir, ama `42memo` öyle değil — orada `42` sayı, `memo` ise ayrı bir tanımlayıcı olarak okunur. Bu, sayıdan hemen sonra gelen değişken adlarının yanlışlıkla yutulmasını önler.

12.2.5 Anahtar kelime mi, tanımlayıcı mı?

Lexer bir harf dizisi okuduğunda — örneğin `let` ya da `print` — bunun bir **anahtar kelime** mi yoksa sıradan bir **tanımlayıcı** mı olduğuna karar vermelidir. Çözüm basittir: okunan metin önce bilinen anahtar kelimeler tablosunda aranır. Eşleşirse o anahtar kelimenin türü (`LET`, `IF`, `DEF`…), eşleşmezse `IDENT` üretilir. Yani `let` bir `LET`, `print` ise bir `IDENT`'tir.

12.2.6 Örnek programın token'ları

Şimdi gerçek çıktıyı görelim. `--tokens` bayrağı lexer'ın ürettiği akışı aynen basar:

```
$ cobra --tokens ornek.cobra
line 1   LET      "let"
line 1   IDENT    "n"
line 1   =        "="
line 1   INT      "2"
line 1   NEWLINE  "\\n"
line 2   IDENT    "print"
line 2   (        "("
line 2   IDENT    "n"
line 2   +        "+"
line 2   INT      "1"
line 2   )        ")"
line 2   NEWLINE  "\\n"
```

Dikkat edin: `let` bir anahtar kelimeye (`LET`) çözüldü, ama `n` ve `print` birer `IDENT` olarak kaldı. İki satır arasındaki `NEWLINE` deyim sınırını işaretliyor. `print(...)` içindeki parantez sayacı bir artırıp azalttığı için, orada olası bir satır sonu yok sayılırdı.

12.3 Parser — Token'lardan Ağaca

Parser (ayrıştırıcı) token akışını bir **soyut sözdizim ağacına** (AST) çevirir. AST, programın yapısını ağaç biçiminde temsil eder: hangi ifade hangi ifadenin içinde, hangi operatör hangi işlenenlere uygulanıyor.

Cobra, **Pratt ayrıştırması** kullanır — operatör önceliğine duyarlı, yukarıdan-aşağı çalışan bir tekniktir. Ünlü olmasının nedeni, ifade önceliğini (`*`, `+`'dan önce gelir) çok zarif biçimde ele almasıdır.

12.3.1 Öncelik merdiveni

Her operatörün bir **öncelik düzeyi** vardır. En düşükten en yükseğe doğru sıralandıklarında ortaya şu merdiven çıkar:

```
en düşük ┌─ atama          x = ...
         │  üçlü (ternary)  kosul ? a : b
         │  mantıksal or    a or b
         │  mantıksal and   a and b
         │  mantıksal not   not a
         │  eşitlik         ==  !=
```

```
          │  karşılaştırma   <  >  <=  >=
          │  toplama         +  -
          │  çarpma          *  /  %
          │  tekli (prefix)  -x  not x
          │  çağrı           f(...)
en yüksek └─ indeksleme      liste[i]
```

Bu sıralama sayesinde `2 + 3 * 4` doğru biçimde `2 + (3 * 4)` olarak ayrışır: çarpmanın önceliği toplamadan yüksek olduğu için, parser `3 * 4`'ü tek bir alt ağaç olarak toplar, sonra onu toplamanın sağ tarafı yapar.

12.3.2 Prefix ve infix kuralları

Pratt ayrıştırmasının özü iki kural kümesidir. Her token türü için parser şunu bilir:

- **prefix kuralı:** token bir ifadenin *başında* geldiğinde ne yapılacağı — `-x` içindeki `-`, ya da tek başına duran bir tamsayı literali, ya da bir açılış parantezi (gruplama).
- **infix kuralı:** token iki ifadenin *arasında* geldiğinde ne yapılacağı — `a + b` içindeki `+`, ya da `a == b` içindeki `==`.

İlginç bir nokta: `(` ve `[` de birer infix kurala sahiptir. `f(x)` ifadesinde `(`, soldaki `f` ifadesinden *sonra* gelir; yani bir infix operatör gibi davranır ve çok yüksek öncelik (çağrı/indeksleme) taşır. Benzer şekilde `xs[i]` içindeki `[` soldaki `xs`'i "yutar". Böylece `f(x)` bir **çağrı düğümü**, `xs[i]` bir **indeks düğümü** olur.

12.3.3 Pratt çekirdeği

Tüm sihir tek bir döngüde toplanır. Bir ifade ayrıştırılırken parser önce o anki token'ın prefix kuralını çalıştırıp bir "sol taraf" elde eder. Sonra sağdaki token *yeterince yüksek öncelikliyken* infix kuralları sol tarafı tekrar tekrar "yutmaya" devam eder:

```
ifade_ayrıştır(öncelik):
    sol = curToken.prefix_kuralı()         # önce baştaki ifade
    döngü: peekToken'ın önceliği > öncelik iken
        sol = peekToken.infix_kuralı(sol)  # sol taraf büyüyor
    sol'u döndür
```

`n + 1` için: önce `n` (bir tanımlayıcı) prefix kuralıyla alınır. Sonra sıradaki token `+` olduğundan ve önceliği yeterli geldiğinden, `+`'ın infix kuralı çalışır; o da sağ tarafı (`1`) ayrıştırıp bir **infix düğümü** kurar — sol işleneni `n`, operatörü `+`, sağ işleneni `1` olan bir düğüm.

12.3.4 AST düğümleri

AST'deki her düğüm bir dil yapısını temsil eder. Örnek programımız şu düğümleri üretir:

- Bir **let deyimi** düğümü: adı `n`, değeri bir tamsayı düğümü.
- O **tamsayı düğümü**: değeri `2`.
- Bir **ifade deyimi** düğümü: içinde bir çağrı.
- O **çağrı düğümü**: çağrılan `print`, argümanı bir infix düğümü.
- O **infix düğümü**: sol `n`, operatör `+`, sağ `1` (bir tamsayı düğümü).

Her düğüm türü yalnızca kendi anlamı için gereken bilgiyi tutar: bir tamsayı düğümü sadece sayısal değerini, bir infix düğümü ise sol-operatör-sağ üçlüsünü tutar. Bu düğümler bir araya gelip programın tam yapısını oluşturur.

12.3.5 Örnek programın AST'si

`--ast` bayrağı AST'yi, her düğümün yeniden-metinleştirilmiş biçiminde gösterir:

```
$ cobra --ast ornek.cobra
let n = 2
print((n + 1))
```

`n + 1`'in `(n + 1)` olarak parantezli basılmasına dikkat edin. Bu rastlantı değil: her infix düğümü kendini açıkça parantezleyerek yeniden metinleştirir, böylece ayrıştırma yapısı gözle görünür hale gelir. Yani parser gerçekten de `print` çağrısının argümanı olarak bir toplama düğümü kurmuş.

> **Not:** Parser, sözdizimsel kuralları çalışma zamanından *önce* dayatır. Örneğin döngü dışındaki bir `break`, `finally` bloğu içindeki bir `return`, ya da `not x == 3` gibi öncelik incelikleri (parser onu `not (x == 3)` yapar; çünkü `not` karşılaştırmalardan daha gevşek bağlanır) hep bu aşamada çözülür.

12.4 Ağaç-Yürüten Motor — AST'yi Doğrudan Çalıştırmak

En basit motor budur ve Cobra'nın varsayılanıdır. Ağaç-yürütücü, AST'yi kökten başlayarak gezer ve her düğüm için "bu ne anlama geliyor?" sorusunu *yerinde* yanıtlar. Bir tamsayı düğümü görürse bir tamsayı değeri üretir; bir infix düğümü görürse önce iki tarafını değerlendirir, sonra operatörü uygular.

Hesaplanan değerler birer **Cobra değeridir**: tamsayı, float, string, liste, sözlük, fonksiyon, yapı örneği ve benzeri. Değişken bağları (binding) bir **ortam** (environment) içinde tutulur. İç içe kapsamlar bir zincir oluşturur: bir değişken aranırken önce en içteki ortam, bulunamazsa onu çevreleyen ortam yoklanır — ta ki en dış (global) ortama kadar.

Örnek programımız bu motorda şöyle akar:

1. Program düğümü, deyimleri sırayla gezer.
2. `let n = 2`: önce sağ taraf (`2` tamsayı düğümü) değerlendirilir ve bir tamsayı değeri üretir; bu değer ortamda `n` adına bağlanır.
3. `print(n + 1)`: çağrı düğümü değerlendirilir. Önce `n + 1` (bir infix düğümü) hesaplanır — `n` ortamdan okunur (`2`), `+` uygulanır, sonuç `3` olur. Sonra `print` gömülü fonksiyonu `3` argümanıyla çağrılır ve `3` ekrana basılır.

12.4.1 Kontrol akışının yukarı taşınması

Bir blok değerlendirilirken motor, deyimleri sırayla işler — ama `return`, `break`, `continue` ya da bir hata ortaya çıkarsa bloğu hemen terk etmesi gerekir. Bunu, bu sinyalleri özel değerlere sararak yapar. Bir blok her deyimden sonra "bu bir kontrol akışı sinyali mi?" diye bakar; öyleyse kalan deyimleri atlayıp sinyali bir üst seviyeye geri verir:

```
blok_değerlendir(blok, ortam):
    sonuç = null
    her deyim için blok içinde:
        sonuç = değerlendir(deyim, ortam)
        eğer sonuç bir kontrol-akışı sinyaliyse (return/break/continue/hata):
            sonuç'u hemen yukarı döndür   # kalan deyimleri atla
    sonuç'u döndür
```

Bir `return` değeri sarılı bir "dönüş değeri" olarak yukarı taşınır ve fonksiyon sınırında açılarak gerçek sonuca dönüşür. `break` ve `continue` benzer biçimde en yakın döngüye kadar yükselip orada tüketilir. Bu, özyinelemeli bir yürütücü yazmanın klasik ve okunması en kolay yoludur — ama her düğümde tür ayrımı yapmak ve her değer için yeni nesne üretmek nedeniyle en hızlısı değildir. İşte VM'in var olma sebebi tam olarak budur.

12.5 Derleyici + Sanal Makine — Bayt-Koduna İnmek

İkinci motor, AST'yi doğrudan yürütmek yerine önce **bayt-koduna** (bytecode) derler: yığın tabanlı bir sanal makine için kompakt bir komut akışı. Sonra bu bayt-kodu sıkı, kısa bir döngüde çalıştırır. Ağacı tekrar tekrar gezmek yerine, önceden hazırlanmış düz bir komut listesini hızla işler.

12.5.1 Opcode'lar

Bayt-kodun yapı taşı **opcode**'dur (işlem kodu): sanal makinenin anlayabileceği tek bir ilkel komut. Her opcode bir bayttır ve isteğe bağlı **operand**'lar alabilir. Cobra'nın kullandığı opcode'lardan bazıları:

- `OpConstant <i>` — sabit havuzundaki `i` numaralı değeri yığına iter.
- `OpSetGlobal <slot>` / `OpGetGlobal <slot>` — global bir değişkeni yazar/okur.
- `OpSetLocal <slot>` / `OpGetLocal <slot>` — yerel bir değişkeni yazar/okur.
- `OpGetBuiltin <i>` — `print`, `len` gibi gömülü bir fonksiyonu yığına iter.
- `OpBinary <op>` — yığından iki değer çeker, bir ikili operatör uygular, sonucu geri iter. Operand hangi operatör olduğunu söyler (+, −, ∗, …).
- `OpCall <n>` — `n` argümanla bir çağrı yapar.
- `OpPop` — yığının tepesindeki değeri atar (deyim sonucu kullanılmıyorsa).
- `OpHalt` — ana programın sonu.

Her opcode'un operand genişlikleri sabittir. Örneğin `OpConstant` 2 baytlık bir operand (sabit havuzu indeksi) alır, `OpGetBuiltin` 1 baytlık, `OpPop` ise hiç operand almaz. VM, komut işaretçisini her komuttan sonra tam bu genişlik kadar ilerletir.

İkili operatörlerin hepsi için ayrı ayrı opcode tanımlamak yerine tek bir `OpBinary` kullanılır; operandı hangi operatörün uygulanacağını belirtir. Aynı sıralama (`+ - * / % == != < > <= >=`) hem derleyici hem VM tarafında paylaşılır, böylece operatör numaraları her iki tarafta da aynı anlama gelir.

12.5.2 Derleyici ne yapar?

Derleyici de AST'yi gezer — tıpkı ağaç-yürütücü gibi — ama değer *hesaplamak* yerine opcode *üretir*. Bir tamsayı düğümü görünce onu sabit havuzuna ekler ve bir `OpConstant` yayar. Bir infix düğümü görünce önce sol tarafı, sonra sağ tarafı derler (ikisi de yığına değer bırakacaktır), en sonunda bir `OpBinary` ekler.

Önemli bir ayrıntı: değişkenler isimle değil, **slot numarasıyla** ele alınır. Derleme sırasında bir **sembol tablosu** her isme bir tamsayı indeks atar (global, yerel, serbest ya da gömülü kapsamlara göre). Çalışma anında VM artık isim aramaz, doğrudan bir diziyi indeksler — bu büyük bir hız kazancıdır.

12.5.3 Örnek programın bayt-kodu

İşte örneğimizin derlenmiş hali. Aşağıdaki, bayt-kodun insan-okunur bir **dökümüdür** (sökümü); soldaki sayılar her komutun bayt adresidir:

```
=== KOMUTLAR ===
0000 OpConstant 0        # sabit[0] = 2'yi yığına it
0003 OpSetGlobal 0       # global slot 0'a (n) ata
0006 OpGetBuiltin 0      # gömülü 'print'i it
0008 OpGetGlobal 0       # n'yi it
0011 OpConstant 1        # sabit[1] = 1'i it
0014 OpBinary 0          # 0 = "+" uygula  →  n + 1
0016 OpCall 1            # 1 argümanla çağır → print(3)
0018 OpPop               # sonucu yığından at (deyim)
0019 OpHalt              # program sonu

=== SABİTLER ===
0: INTEGER 2
1: INTEGER 1
```

Token akışını, AST'yi ve bu bayt-kodu yan yana koyduğunuzda boru hattının tamamını görürsünüz: `2` ve `1` literalleri **sabit havuzuna** (constant pool) taşındı, `n` bir **global slota** indirgendi, `+` bir `OpBinary 0`'a, `print` çağrısı ise `OpGetBuiltin 0` + `OpCall 1` ikilisine dönüştü.

Adres sütununa dikkat: `0000`, `0003`, `0006`… Aradaki boşluklar operandların kapladığı baytlardır. `OpConstant` 1 (opcode) + 2 (operand) = 3 bayt yer kaplar, bu yüzden ardından gelen komut `0003`'tedir; `OpGetBuiltin` 1 + 1 = 2 bayttır, bu yüzden ondan sonraki komut `0008`'dedir. VM, komut işaretçisini tam bu kadar ilerletir.

12.5.4 OpHalt ve sınır-kontrolsüz döngü

Derleyici, ürettiği komut akışının sonuna her zaman bir `OpHalt` ekler. Bunun amacı performanstır. VM'in iç döngüsü her komuttan sonra "akışın sonuna ulaştık mı?" diye kontrol etmek zorunda kalsaydı, bu kontrol her adımda boşa zaman harcardı. Bunun yerine, akışın sonunda bir `OpHalt`'a *garanti* çarpacağını bildiği için döngü hiçbir sınır kontrolü yapmaz; `OpHalt`'a geldiğinde sadece durur ve geri döner. Sıcak döngüden tek bir karşılaştırmayı bile çıkarmak, milyonlarca komut işleyen bir VM'de hissedilir bir kazanç sağlar.

12.5.5 Sanal makine nasıl çalışır?

VM bir **yığın makinesidir**. Hesaplamalarını bir değer yığını üzerinde yapar: değerleri iter (push), çeker (pop), ve sonuçları geri iter. Önceden ayrılmış sabit boyutlu birkaç dizi tutar: değer **yığını**, **global** değişkenler dizisi ve çağrı **çerçeveleri** (frames). Bir de **yığın işaretçisi** vardır; yığının tepesinin nerede olduğunu söyler.

Çekirdek, opcode'lar üzerinde dönen tek bir döngüdür: sıradaki opcode'u oku, ne yapması gerekiyorsa yap, komut işaretçisini ilerlet, tekrarla. Örnek programımızın yürütülmesini adım adım izleyelim — sağdaki sütun her komuttan *sonraki* yığını gösteriyor:

adres	komut	yığın (sonra)
0000	OpConstant 0	`[2]`
0003	OpSetGlobal 0	`[]` (n = 2)
0006	OpGetBuiltin 0	`[print]`
0008	OpGetGlobal 0	`[print, 2]`
0011	OpConstant 1	`[print, 2, 1]`
0014	OpBinary 0	`[print, 3]`
0016	OpCall 1	`[null]` (print 3 bastı)
0018	OpPop	`[]`
0019	OpHalt	—

`OpBinary 0`'ın iki değeri (`2` ve `1`) çekip yerine tek bir sonucu (`3`) koymasına dikkat edin — bir yığın makinesinin tipik hareketi. `OpCall 1` ise `print`'i ve argümanını yığından alıp çağrıyı yürütür; geriye `print`'in dönüş değeri (`null`) kalır, onu da `OpPop` atar.

12.5.6 Çerçeveler (frames) ve kapanışlar (closures)

Her fonksiyon çağrısı bir **çerçeve** (frame) açar. Bir çerçeve şunları tutar: o an çalışan fonksiyon, o fonksiyona özel komut işaretçisi, ve yerel değişkenlerin yığında nereden başladığı (taban işaretçisi). Bir fonksiyon çağrıldığında yeni bir çerçeve "itilir"; fonksiyon döndüğünde çerçeve "çekilir", yığın çağrıdan önceki haline geri sarılır ve dönüş değeri bırakılır. Çerçeveler de sabit boyutlu bir dizide tutulur; bu dizinin derinliği özyineleme sınırını belirler.

Kapanışlar (closures), çevreleyen kapsamdaki değişkenlere başvuran fonksiyonlardır. Bir fonksiyon, kendi dışındaki bir değişkene ("serbest değişken") erişiyorsa, derleyici bunu fark eder ve o değişkeni fonksiyonun kapanışına bağlar. Çalışma anında ayrı opcode'lar bu serbest değişkenlere erişir. Sonuç olarak, bir fonksiyon onu üreten bağlam çoktan kapanmış olsa bile değişkenini hatırlamaya devam eder. Klasik örnek bir sayaç üretici:

```
def sayac_yap()
    let n = 0
    def artir()
        n = n + 1     # 'n' serbest değişken — sayac_yap'tan kapanışa taşınır
        return n
    end
    return artir
end

let c = sayac_yap()
print(c())
print(c())
print(c())
```

Her iki motor da aynı sonucu verir:

```
$ cobra closure.cobra        # ağaç-yürüten
1
2
3
$ cobra --vm closure.cobra   # VM
1
2
3
```

12.5.7 Süperkomutlar: bir hız optimizasyonu örneği

VM'de gerçek bir mühendislik dokunuşu **süperkomut**lardır. Fonksiyon gövdelerinde en sık görülen desenlerden biri `yerel_değişken OP tamsayı_literali` biçimindedir — `n < 2`, `i + 1` gibi. Normalde bu üçlü, üç ayrı komut gerektirir: yerel değişkeni it, sabiti it, ikili operatörü uygula. Derleyici bu sık deseni fark edip **tek bir kaynaşık komuta** indirir. Üç komutluk dağıtım yerine tek komut çalışır; üstelik bu komutun tamsayılar için özel bir hızlı yolu vardır ve yavaş, genel operatör mantığına hiç uğramadan sonucu hesaplar. Programcı hiçbir şey yapmaz — kazanç tamamen derleyici ile VM arasındaki anlaşmadan gelir.

12.6 Yerel Derleme — Yorumlayıcıyı Geride Bırakmak

Şimdiye kadar gördüğümüz iki motorun ortak bir özelliği var: çalışma anında *bir şey programı okur ve ne yapacağına karar verir*. Ağaç-yürüten AST düğümlerini okur, VM opcode'ları okur. Bu okuma — yani dağıtım (dispatch) — yorumlayıcı olmanın bedelidir ve hiçbir ince ayar onu tümüyle ortadan kaldıramaz.

Üçüncü bir yol var ve Cobra onu da açar: **programı makine koduna derleyip yorumlamayı büsbütün bırakmak.**

```
cobra build --native mandelbrot.cobra
```

Sonuç bağımsız bir çalıştırılabilir dosyadır. İçinde VM yok, bayt-kodu yok, Cobra çalışma zamanı yok — yalnızca sizin programınız, işlemcinin doğrudan yürüttüğü komutlar olarak.

12.6.1 Engel: dinamik tipler

Bir betik dilinin en başta neden yorumladığının cevabı basittir: tipleri önceden bilmez. VM `x + y` işlemini yürütürken çalışma anında sormak zorundadır: bunlar tam sayı mı? ondalık mı? metin mi? Bu yüzden her değer *kutulanmış* yaşar — tipini taşıyan bir yığın nesnesine işaretçi — ve her işlem hem kontrolün hem kutunun bedelini öder.

Makine kodu böyle çalışamaz. İşlemcideki `ADD` komutu iki tam sayıyı yazmaçlarda toplar; soru sormaz. Onu üretebilmek için derleyicinin, derleme anında, `x` ve `y`'nin tam sayı olduğunu *bilmesi* gerekir.

12.6.2 Çıkış yolu: tip çıkarımı

Cobra burada gerçek programlara dair bir gözlem yapar: dil dinamik tipli olsa da, **yazılan kod ezici çoğunlukla tip-kararlıdır**. Ondalık olarak başlayan bir değişken ondalık kalır; tam sayı döndüren bir fonksiyon tam sayı döndürmeye devam eder. Tip değiştirme özgürlüğü vardır, ama neredeyse hiç kullanılmaz.

Bu yüzden yerel derleyici tipleri talep etmek yerine *kanıtlar*. Her değişkene, parametreye, alana ve dönüş değerine boş başlayan bir tip "hücresi" verir; sonra tüm programı tekrar tekrar dolaşarak öğrendiği her şeyi bu hücrelere işler:

- boş bir hücreye tam sayı atanması onu tam sayı yapar;
- tam sayı tutan bir hücre bir ondalıkla karşılaşırsa ondalığa yükselir — dilin çalışma anında yaptığı terfinin aynısı (`1 + 2.5` sonucu `3.5`'tir);
- `null` tutan bir hücre bir struct ile karşılaşırsa o struct olur (ikili ağaçtaki `Node(null, null)` böyle doğru çıkarsanır).

Dolaşma, hiçbir şey değişmeyene kadar tekrarlanır — bir *sabit nokta* (fixpoint). Durulduğunda programdaki her adın tek bir somut makine tipi vardır ve derleyici gerçek komutlar üretebilir. Bir ad gerçekten sabitlenemiyorsa derleyici tahmin yürütmez: programı *yerel olarak derlenemez* diye bildirir ve hangi yapının buna engel olduğunu söyler.

12.6.3 Ortaya ne çıkar

Tipler bilindiğinde çeviri doğrudandır:

Cobra	yerel kod
`let i = 0`	yazmaçta 64-bit tam sayı
`let x = 1.5`	yazmaçta 64-bit ondalık — kutu yok, tahsis yok
7 alanlı `struct Body`	düz bir yapı: 7 ondalık, bitişik yerleşimli
`let xs = []`	yerel, büyüyebilen dizi
`p.x`	sabit bir bellek konumu — ad araması yok
`fib(n - 1)`	makine çağrı komutu

Farkın en çarpıcı göründüğü satır struct satırıdır. VM'de `b.vx * b.vx` şu demektir: alanı adıyla bul, bir ondalığı kutudan çıkar, bir tanesini daha çıkar, çarp, sonuç için yeni bir kutu tahsis et. Yerel kodda ise bu, iki yazmaç arasındaki tek bir çarpma komutudur.

12.6.4 Yakın değil, birebir aynı

Farklı cevap veren hızlı bir motorun değeri olmaz; bu yüzden iki yorumlayıcıyı yöneten kural bunu da yönetir: **çıktı bayt bayt aynı olmalıdır.** Bu sözü tutmak için iki incelik gerekti.

Ondalık *yazdırma* birebir uymalı — Cobra `3.0` değerini `3` değil `3.0` olarak yazar ve bilimsel gösterime tam olarak belirli eşiklerde geçer — bu yüzden yerel ikili dosya, yorumlayıcının kullandığı biçimlendirme mantığının aynısını taşır.

Ondalık *aritmetik* daha incedir. Modern işlemciler kaynaşık çarp-topla (FMA) komutu sunar: `a*b + c` işlemini iki yerine tek yuvarlamayla, tek adımda hesaplar. Daha hızlıdır ve tartışmasız daha doğrudur — ama iki kez yuvarlayan yorumlayıcıdan *son basamakta farklı bir sayı* üretir. Cobra bu yüzden her ondalık işlemden sonra açık bir yuvarlama bariyeri koyar ve kaynaşmayı yasaklar. Sonuçları VM'inkiyle bit bit eşit tutmak için bir miktar hızdan feragat eder — bir dil çalışma zamanı için doğru takas budur.

12.6.5 Bedeli ve kazancı

Yerel derleme dilin tip-kararlı çekirdeğini destekler: fonksiyonlar, `init` ve metotlarıyla struct'lar, listeler, sözlükler, metinler, `while` / `for-in` / `if` ve `math` modülü. Closure'lar, `async`, `try/catch` ve kalıtım derlenmez — her biri "her ada tek somut tip" modelini kendi yolundan kırar (örneğin bir `try` bloğu, programdaki *her* hata verebilecek işlemin kontrollü bir yol ve satır numarası taşımasını gerektirir; bu da tam olarak yerel kodun kaçınmak için var olduğu maliyettir). Bunları kullanan programlar VM'de, değişmeden ve cezalandırılmadan çalışır.

Kazancı ise bambaşka bir performans sınıfıdır. Klasik sayısal ölçütlerde — n-body, binary-trees, spectral-norm, mandelbrot — yerel ikili dosya takımı 0.28 saniyede bitirir. CPython 5.99 saniye sürer. Python'u çalışma anında makine koduna derleyen izleme-tabanlı bir JIT olan PyPy 0.44 saniye. Yerel ikili dosyanın ısınma süresi yoktur, çalışma zamanı yoktur ve onu çalıştıran makinede kurulu hiçbir şeye ihtiyaç duymaz.

Bu bölümden çıkarılacak asıl ders ölçüt tablosu değildir. Asıl ders şudur: kutular ve dağıtım döngüsü — yorumlayıcı bölümlerinin optimize etmek için onca emek harcadığı her şey — *dilin* özsel parçası hiçbir zaman olmadı. Onlar, tipleri bilmemenin bedeliydi. Tipleri öğrenin, bedel ortadan kalkar.

12.7 Paylaşılan Anlambilim — İki Motor, Tek Hakikat

Buraya kadar iki motor ayrı görünüyor: biri AST gezer, diğeri bayt-kod yürütür. Ama dilin *anlamı* — `1 + 2` neye eşittir, bir liste nasıl dilimlenir, bir alan ataması ne yapar — **tek bir yerde** tanımlıdır. Bu, Cobra'nın iki motoru hizalı tutmasının anahtarıdır.

Anlambilim, AST'den de bayt-koddan da bağımsız, saf "değer üstü işlemler" olarak yaşar. Her iki motor da bu tek tanıma başvurur. Birkaç örnek:

İkili operatörler. VM, tamsayı ve metin için satır içi hızlı yollara sahiptir; ama geri kalan tüm durumlar (float, decimal, liste + liste, eşitlik kuralları…) ağaç-yürütücünün de kullandığı tam aynı ortak operatör mantığına düşer. `1 + 2`'nin tamsayı, `1 + 2.5`'in float'a yükseltme, `"a" + "b"`'nin birleştirme, liste + liste'nin yeni bir liste üretmesi — hepsi tek yerde tanımlıdır.

Doğruluk (truthiness). `if`, `while`, `and`/`or` ve `?:` hangi değerin "doğru" sayıldığına tek bir ortak kuralla karar verir. VM'in koşullu atlama opcode'u da bu aynı kuralı kullanır; böylece boş bir liste her iki motorda da "yanlış" sayılır.

İndeksleme ve dilimleme. `xs[i]` ve `xs[düşük:yüksek:adım]` mantığı tek bir ortak işlemde yaşar. VM'in indeksleme ve dilimleme opcode'ları bu ortak işleme yönlendirir. Bu yüzden negatif adımlı ters çevirme (`xs[::-1]`) gibi ince davranışlar bile iki motorda **bit-bit** aynıdır:

```
let xs = [10, 20, 30, 40]
print(xs[1:3])
print(xs[::-1])
```

```
$ cobra slice.cobra          # ağaç-yürüten
[20, 30]
[40, 30, 20, 10]
$ cobra --vm slice.cobra     # VM
[20, 30]
[40, 30, 20, 10]
```

Alan erişimi ve atama. `p.x` (okuma) ve `p.x = v` (yazma) mantığı da paylaşılır. Buradaki incelik şudur: property *getter* ve *setter*'ları (parantezsiz erişimi yakalayan özel metotlar) her iki motorda da çalışmak zorundadır. Bir setter bulunduğunda, ortak alan-atama mantığı onu ister ağaç-yürütücünün fonksiyon biçiminde, ister derlenmiş kapanış biçiminde olsun çağırabilir. Aşağıdaki yapı, Celsius değerini içeride tutar ama Fahrenheit üzerinden okunup yazılabilir:

```
struct Sicaklik
    def init(c)
        self._c = c
    end
    get fahrenheit()
        return self._c * 9.0 / 5.0 + 32.0
    end
    set fahrenheit(f)              # t.fahrenheit = 212.0 bunu çağırır
        self._c = (f - 32.0) * 5.0 / 9.0
    end
end

let t = Sicaklik(0.0)
t.fahrenheit = 212.0
print(t._c)
print(t.fahrenheit)
```

Hem setter (`t.fahrenheit = 212.0`) hem getter (`t.fahrenheit`) iki motorda da aynı yoldan geçer:

```
$ cobra setter.cobra         # ağaç-yürüten
100.0
```

```
212.0
$ cobra --vm setter.cobra    # VM
100.0
212.0
```

İki motoru birbirine bağlayan zarif bir köprü vardır: VM başlatıldığında, derlenmiş bir kapanışı nasıl çalıştıracağını ortak anlambilime "öğretir". Böylece ortak mantık, kendisi VM'e bağımlı olmadan, gerektiğinde derlenmiş bir fonksiyonu çağırabilir. Bu, iki bileşen arasındaki döngüsel bağımlılığı kırarken anlambilimi tek noktada tutan akıllıca bir tasarımdır.

> **Uyarı:** İki motor arasında *bir* bilinen davranış farkı vardır: tanımsız bir değişkene başvurmak. Ağaç-yürütücü bunu çalışma anında yakalar; VM ise isimleri derleme zamanında slotlara çözdüğü için aynı hatayı *derleme sırasında* bildirir. Sonuç aynıdır (program reddedilir), ama hatanın yakalandığı an farklıdır.

Paylaşılan anlambilimin pratik faydası, projenin kıyaslama (benchmark) takımında somutlaşır: her iş yükü hem ağaç-yürütücüde hem VM'de çalıştırılır ve **ikisinin de birebir aynı çıktıyı üretmesi** şart koşulur. Bu çıktı eşitliği bir testtir; biri diğerinden saparsa derleme bozulur.

12.8 Özet

Bu bölümde Cobra'nın iç işleyişini baştan sona, gerçek araç çıktıları üzerinden izledik:

- **Mimari.** Kaynak metin önce ortak iki aşamadan geçer — **lexer** (token) ve **parser** (AST) — sonra iki motordan birine gider: doğrudan değerlendiren **ağaç-yürütücü** ya da derleyip **VM**'de çalıştıran yol. `--tokens`, `--ast`, `--vm` bayrakları bu boruları açar.
- **Lexer.** Karakterleri token'lara böler. `NEWLINE` deyim sonudur ama parantez içinde yok sayılır; `==`/`+=` gibi operatörler ileriye göz atılarak tanınır; sayılar ve `m` ekli ondalıklar okunur; `#` yorumları atlanır; anahtar kelimeler tanımlayıcılardan bir tablo aramasıyla ayrılır.
- **Parser.** Öncelik-duyarlı **Pratt ayrıştırması**: prefix/infix kuralları ve atamadan indekslemeye uzanan öncelik merdiveni birleşip doğru iç içe AST'yi kurar.
- **Ağaç-yürütücü.** AST'yi düğüm düğüm gezer; değerleri yerinde hesaplar, ortamda saklar, kontrol akışını sarmalayıcılarla yukarı taşır. Basit ve okunabilir.
- **Derleyici + VM.** Derleyici AST'yi **opcode**'lara ve bir **sabit havuzuna** çevirir; isimleri slotlara indirir. VM bunları sıkı bir dağıtım döngüsünde yürütür — `OpHalt` ile sınır-kontrolsüz, **çerçeve**'ler ve **kapanış**'larla. Süperkomutlar sıcak desenleri tek komuta kaynaştırır.
- **Paylaşılan anlambilim.** Dilin anlamı — ikili operatörler, doğruluk, indeksleme, dilimleme, alan erişimi — tek bir noktada yaşar. İki motor da bu ortak tanıma başvurur; bir köprü, derlenmiş kapanışları ortak mantığa bağlar. Sonuç, çıktı eşitliğiyle test edilen ve garanti edilen tek bir dildir — iki bedende.

Cobra'nın iki motorlu tasarımı, dil tasarımının temel bir gerçeğini güzel biçimde gösterir: **sözdizimi ve anlambilimi, yürütme stratejisinden ayırın.** Lexer ile parser bir kez yazılır ve paylaşılır; anlambilim bir kez yazılır ve paylaşılır; geriye kalan tek seçim, AST'yi nasıl çalıştıracağınızdır — basit ve açık mı (ağaç-yürütücü), yoksa hızlı mı (VM). Cobra ikisini birden sunar ve sizi seçim yapmaya zorlamadan, her ikisinin de aynı yanıtı vereceğine güvenmenizi sağlar.

Bölüm 13

cobranum — Sayısal Hesaplama

Cobra'nın `[1, 2, 3]` listesi genel amaçlıdır: içine her tür değeri koyar, büyütür, küçültürsünüz. Bu esnekliğin bedeli vardır. Listedeki her eleman ayrı bir **kutulu** nesnedir — bir tam sayı bile bellekte bir işaretçinin arkasında durur, dağınık adreslerde yaşar ve her erişimde bir yönlendirme gerektirir. Birkaç eleman için bu görünmezdir; ama 512×512'lik iki matrisi çarpmak, bir milyon ölçümün ortalamasını almak ya da on bilinmeyenli bir doğrusal sistemi çözmek istediğinizde kutulama hem belleği hem de işlemci önbelleğini boşa harcar.

cobranum bu boşluğu doldurur: Cobra'ya **yoğun, tipli, N-boyutlu bir dizi** (`ndarray`) ekler. Bir cobranum dizisi, arka planda **ardışık bir `float64` tamponudur** — kutu yok, işaretçi takibi yok, önbellek dostu. Üzerindeki işlemler vektörize ve çok çekirdeklidir; macOS'ta matris işlemleri Apple Accelerate'in BLAS/LAPACK rutinlerine, eleman bazlı işlemler ise elle yazılmış ARM NEON (SIMD) çekirdeklerine ve birden çok iş parçacığına gider. Sonuç, Cobra'yı bir betik dilinden çıkarıp doğrusal cebir, istatistik, optimizasyon, diferansiyel denklemler ve fizik simülasyonu yazabileceğiniz bir sayısal hesaplama aracına dönüştürür.

Bu bölüm bir "fonksiyon listesi" değildir. Önce dizinin nasıl çalıştığını — bellek modeli, yayılım kuralları, sayısal kararlılık — kuracağız; sonra kütüphaneyi gerçek bilimsel problemlerle çalıştıracağız: en küçük kareler, asal bileşen analizi, PageRank, diferansiyel denklem çözümü, ısı difüzyonu, Newton yöntemi, lojistik regresyon. Her kod bloğu cobranum'a karşı gerçekten koşturulmuştur; `#` ile başlayan çıktı yorumları programın gerçek çıktısıdır.

cobranum çekirdeğin parçası değil, bir **Go eklentisidir**. `cobranum.so`'yu içe aktararak kullanırsınız; bu yüzden onu yükleyebilen (eklenti uyumlu) bir cobra ikili dosyasına ihtiyaç vardır.

13.1 13.1 ndarray: bellek modeli

Bir cobranum dizisi iki şeyden oluşur: bir **biçim** (`shape`) — her boyutun uzunluğu — ve düz, ardışık bir `float64` tamponu. Veri **satır-öncelikli** (row-major) saklanır: 2 boyutlu bir `[r, c]` dizisinde `[i][j]` elemanı tamponda `i*c + j` konumundadır. Bu düzen, bir satırı baştan sona taramayı bir bellek sıçraması olmaktan çıkarır; modern işlemcilerin önbelleği tam da bu erişim desenini sever.

cobranum 1 ve 2 boyutlu dizileri destekler — vektörler ve matrisler, sayısal işlerin büyük çoğunluğu. Bir dizi **opak bir tutamaçtır**: doğrudan yazdırmak yerine içeriğini `to_list` (iç içe Cobra listesi), `str` (okunur özet) veya `shape` (boyut) ile görürsünüz.

```
import "cobranum.so"
```

```
let np = cobranum

let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.shape(M))      # [2, 3]
print(np.str(M))        # ndarray(shape=[2, 3], data=[[1, 2, 3], [4, 5, 6]])
```

13.2 13.2 Dizi oluşturma

Dizileri (iç içe) listelerden kurabilir ya da doğrudan üretebilirsiniz. Üreticiler büyük bir Cobra listesi oluşturmaz; veriyi doğrudan Go tarafında ayırırlar, bu yüzden milyonlarca eleman için bile ucuzdurlar.

```
np.array([1, 2, 3])             # 1B vektör
np.array([[1, 2], [3, 4]])      # 2B matris

np.zeros([2, 3])                # tümü 0
np.ones([4])                    # [1, 1, 1, 1]
np.full([2, 2], 7.0)            # sabitle doldur
np.eye(3)                       # birim matris
np.arange(0, 10, 2)             # [0, 2, 4, 6, 8] (son hariç)
np.linspace(0.0, 1.0, 5)        # [0, 0.25, 0.5, 0.75, 1] (uç dahil)
```

`arange` bir sayma dizisidir (başlangıç, bitiş hariç, adım); `linspace` ise iki uç arasını **tam olarak `count` noktaya** böler — sayısal integral ve grafik ızgaraları için doğru araç budur.

13.3 13.3 İndeksleme, inceleme, yeniden şekillendirme

```
let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.ndim(M), np.size(M))            # 2  6
print(np.get(M, 1, 2))                   # 6   ([1][2] elemanı)
np.set(M, 0, 0, 99)                      # yerinde tek eleman ataması
print(np.to_list(np.reshape(M, [3, 2]))) # [[99, 2], [3, 4], [5, 6]]
```

`get`/`set` tek elemanlara erişir (negatif indeks sondan sayar). `reshape` aynı sayıda elemanı yeni bir biçimde, yeni bir kopya olarak verir.

13.4 13.4 Eleman bazlı işlemler

Aritmetik, aynı biçimli iki dizide eleman eleman çalışır; ya da bir skalerle.

```
let a = np.array([1, 2, 3])
let b = np.array([10, 20, 30])

np.add(a, b)     # [11, 22, 33]
np.sub(b, a)     # [9, 18, 27]
np.mul(a, 2)     # skaler -> [2, 4, 6]
np.div(b, a)     # [10, 10, 10]
np.pow(a, 2)     # [1, 4, 9]
np.neg(a)        # [-1, -2, -3]
```

Tek-değişkenli matematik de eleman bazlıdır:

```
np.sqrt(np.array([1, 4, 9]))     # [1, 2, 3]
np.exp(np.array([0, 1]))         # [1, 2.718...]
np.log(np.array([1, 2.71828]))   # [0, 0.999...]
np.sin(a)   np.cos(a)   np.abs(np.array([-1, 2, -3]))
```

13.5 13.5 Yayılım (broadcasting)

Boyutları farklı ama uyumlu iki diziyi işleme sokmak için cobranum **yayılım** kullanır: küçük olan, büyük olana eşleşene kadar "gerilir". Kural basittir — bir boyut ya eşit olmalı ya da 1 olmalıdır (1 ise tekrarlanır). Bir uzunluk-`n` vektör, satır vektörü `(1, n)` gibi davranır.

Sol biçim	Sağ biçim	Sonuç	Anlam
`(r, c)`	`(c,)`	`(r, c)`	satır vektörü her satıra
`(r, c)`	`(r, 1)`	`(r, c)`	sütun vektörü her sütuna
`(r, c)`	skaler	`(r, c)`	sabit her elemana

```
let X = np.array([[1, 2, 3], [4, 5, 6]])
let row = np.array([10, 20, 30])          # (3,) -> satırlara yayılır
print(np.to_list(np.add(X, row)))         # [[11, 22, 33], [14, 25, 36]]

let col = np.array([[100], [200]])        # (2,1) -> sütunlara yayılır
print(np.to_list(np.add(X, col)))         # [[101, 102, 103], [204, 205, 206]]
```

Yayılım, "her özniteliğden ortalamasını çıkar" ya da "her satırı normuna böl" gibi işlemleri tek satıra indirir; bu bölümdeki istatistik ve makine öğrenmesi örneklerinde sürekli kullanacağız.

13.6 13.6 Füzyon ve yerinde işlemler

Bir hesap zincirini ayrı çağrılarla yazmak basittir ama her çağrı yeni bir dizi ayırır. `sqrt(a*a + b*b)` dört geçiş ve üç geçici dizi demektir. cobranum iki kaçış yolu sunar:

- **Füzyonlu** çekirdekler tüm hesabı tek geçişte yapar: `hypot(a, b)`, `fma(a, b, c)` (yani `a*b + c`), `axpy(α, x, y)` (yani `α*x + y`).

- **Yerinde** çeşitler (`*_into`) sonucu önceden ayrılmış bir tampona yazar, yani sıcak döngüler hiç bellek ayırmaz.

```
np.hypot(a, b)           # sqrt(a*a + b*b), tek füzyonlu geçiş

let dst = np.zeros([3])
np.mul_into(dst, a, a)    # dst = a*a
np.add_into(dst, dst, b)  # dst = dst + b
np.sqrt_into(dst, dst)    # dst = sqrt(dst)
```

Kural: bir hesap çekirdeği bir döngünün içindeyse, füzyonlu ya da yerinde biçimi tercih edin — ara dizileri ve bellek ayırmayı ortadan kaldırırlar.

13.7 13.7 İndirgemeler ve istatistik

İndirgeme, bir diziyi tek bir sayıya (ya da bir eksen boyunca bir vektöre) toplar.

```
let s = np.array([4, 8, 15, 16, 23, 42])
print(np.sum(s), np.mean(s))          # 108  18
print(np.min(s), np.max(s), np.prod(s))
print(np.var(s), np.std(s))           # 151.66...  12.31...
print(np.norm(s))                     # 53.42...  (karelerin toplamının kökü)
print(np.argmin(s), np.argmax(s))     # 0  5
print(np.to_list(np.cumsum(s)))       # [4, 12, 27, 43, 66, 108]
print(np.to_list(np.clip(s, 10, 20)))  # [10, 10, 15, 16, 20, 20]
```

2 boyutlu bir dizide **eksen** vererek satır ya da sütun indirgersiniz (eksen 0 sütunlar boyunca aşağı, eksen 1 satırlar boyunca):

```
let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.to_list(np.sum(M, 0)))   # [5, 7, 9]   (sütun toplamları)
print(np.to_list(np.sum(M, 1)))   # [6, 15]     (satır toplamları)
```

İstatistiğin daha ileri ölçütleri bu yapı taşlarından kurulur. Örneğin bir veri matrisinin **kovaryans matrisi**, ortalanmış verinin `CᵀC / n` çarpımıdır; 13.11.5'te bunu PCA için kullanacağız.

13.8 13.8 Lineer cebir

Matris çarpımı, devrik, ters, determinant ve doğrusal çözümler — sayısal hesaplamanın bel kemiği — macOS'ta Accelerate (BLAS/LAPACK) üzerinden yürür.

```
let A = np.array([[1, 2], [3, 4]])
let B = np.array([[5, 6], [7, 8]])
```

```
print(np.to_list(np.matmul(A, B)))   # [[19, 22], [43, 50]]
print(np.dot(np.array([1,2,3]), np.array([4,5,6])))  # 32  (1B nokta çarpım)
print(np.to_list(np.transpose(A)))   # [[1, 3], [2, 4]]
print(np.det(A))                     # -2.0
print(np.to_list(np.inv(A)))         # [[-2, 1], [1.5, -0.5]]
print(np.trace(A))                   # 5
print(np.to_list(np.diag(A)))        # [1, 4]
```

Önemli alışkanlık: `A x = b`'yi çözmek için tersi alıp çarpmak (`inv(A)·b`) yerine `solve(A, b)` kullanın. `solve`, LU çarpanlamasıyla çalışır; hem daha hızlı hem de sayısal olarak daha kararlıdır — özellikle kötü koşullu matrislerde fark belirgindir (bkz. 13.11.3).

```
print(np.to_list(np.solve(np.array([[3, 2], [1, 2]]), np.array([5, 5]))))  # [0, 2.5]
```

13.9 13.11 Bilimsel örnekler

Buradan sonrası tam, çalışan programlardır. Her biri bir sayısal yöntemi tanıtır, ardından cobranum'la birkaç satırda uygular. Çıktılar gerçektir.

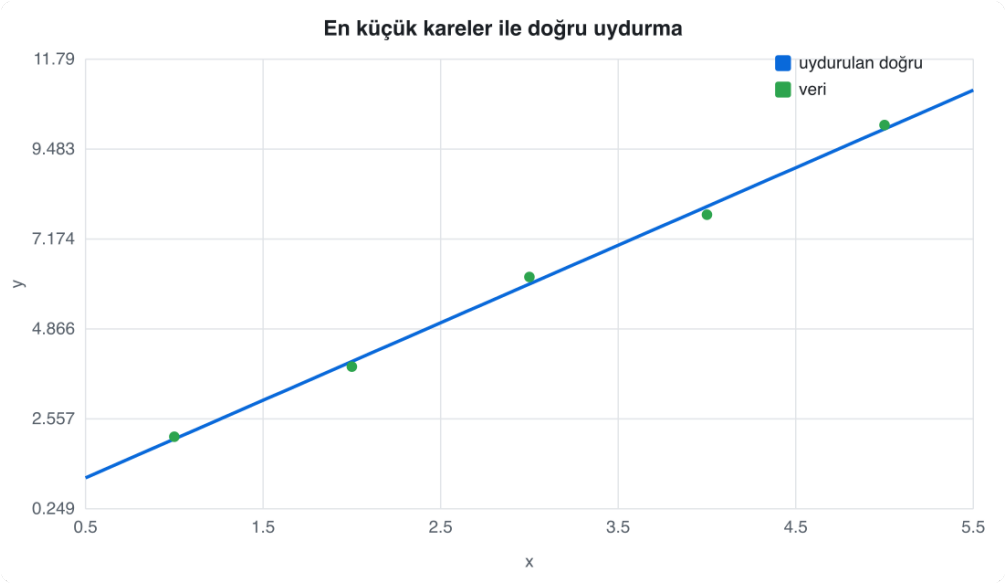
13.9.1 13.11.1 En küçük kareler ve uyum iyiliği (R^2)

Veri noktalarına bir doğru uydurmak istiyoruz: $y = w_0 + w_1 x$. En küçük kareler çözümü, hata karelerini en aza indiren ağırlıklardır ve kapalı biçimde $w = (X^T X)^{-1} X^T y$ ile verilir; burada X'in ilk sütunu sabit terim için 1'dir. Uyumun ne kadar iyi olduğunu **belirleme katsayısı** $R^2 = 1 - SS_{res}/SS_{tot}$ ölçer (1'e ne kadar yakınsa o kadar iyi).

```
let xs = [1.0, 2.0, 3.0, 4.0, 5.0]
let ys = [2.1, 3.9, 6.2, 7.8, 10.1]
let rows = []
for x in xs
  push(rows, [1.0, x])
end
let X = np.array(rows)
let y = np.reshape(np.array(ys), [5, 1])
let Xt = np.transpose(X)
let w = np.matmul(np.matmul(np.inv(np.matmul(Xt, X)), Xt), y)

let resid = np.sub(y, np.matmul(X, w))
let ss_res = np.sum(np.mul(resid, resid))
let dev = np.sub(y, np.mean(y))
let ss_tot = np.sum(np.mul(dev, dev))

print("w:", np.to_list(w))            # [[0.05], [1.99]]   (~ y = 2x)
print("R^2:", 1.0 - ss_res / ss_tot)  # 0.9973
```



Şekil 13.1: En küçük kareler ile bir doğru uydurma.

13.9.2 13.11.2 Ridge (Tikhonov) regülarizasyonu

Öznitelikler birbirine bağımlı olduğunda X^TX neredeyse tekil olur ve ağırlıklar patlar. **Ridge** çözümü köşegene küçük bir λ ekler: $w = (X^TX + \lambda I)^{-1} X^Ty$. Bu, ağırlıkları sıfıra doğru çeker (büzüştürür) ve çözümü kararlı kılar.

```
let lam = 1.0
let I = np.eye(2)
let wr = np.matmul(np.matmul(np.inv(np.add(np.matmul(Xt, X), np.mul(I, lam))), Xt), y)
print("ridge w:", np.to_list(wr))   # [[0.29], [1.89]]  (sıfıra doğru büzüşmüş)
```

13.9.3 13.11.3 Koşullandırma ve kararlılık

Hilbert matrisi `H[i][j] = 1/(i+j+1)` kötü koşulluluğun ders kitabı örneğidir: boyutu büyüdükçe çözüm girdideki minik hatalara aşırı duyarlı hale gelir. `x = [1,1,1,1]` için `b = Hx` üretip geri çözelim; sonucun tam 1'lerden ne kadar saptığı koşullanmanın etkisidir.

```
let Hrows = []
for i in range(4)
  let r = []
  for j in range(4)
    push(r, 1.0 / (i + j + 1))
  end
  push(Hrows, r)
end
let Hm = np.array(Hrows)
let b = np.matmul(Hm, np.reshape(np.array([1,1,1,1]), [4,1]))
print(np.to_list(np.solve(Hm, np.reshape(b, [4]))))
# [0.9999..., 1.0000..., 0.9999..., 1.0000...]  -- küçük sapmalar koşullanmadan
```

13.9.4 13.11.4 Özdeğerler — kuvvet yinelemesi ve deflasyon

Bir matrisle bir vektörü tekrar tekrar çarpıp her adımda normalleştirirsek, vektör **baskın özvektöre**, Rayleigh oranı $v^T A v$ ise onun **özdeğerine** yakınsar. İlk özdeğeri bulduktan sonra matristen $\lambda_1 v_1 v_1^T$ çıkararak (deflasyon) ikinciyi de elde ederiz.

```
def power_iter(A, iters)
  let n = np.shape(A)[0]
  let v = np.ones([n, 1])
  for i in range(iters)
    let Av = np.matmul(A, v)
    v = np.div(Av, np.norm(Av))
  end
  return [v, np.get(np.matmul(np.transpose(v), np.matmul(A, v)), 0, 0)]
end

let A = np.array([[4, 1, 0], [1, 3, 1], [0, 1, 2]])
let r1 = power_iter(A, 200)
let v1 = np.reshape(r1[0], [3])
let A2 = np.sub(A, np.mul(np.outer(v1, v1), r1[1]))   # deflasyon
let r2 = power_iter(A2, 200)
print("özdeğer 1:", r1[1], " özdeğer 2:", r2[1])      # 4.7320...  3.0
```

13.9.5 13.11.5 Asal bileşen analizi (PCA)

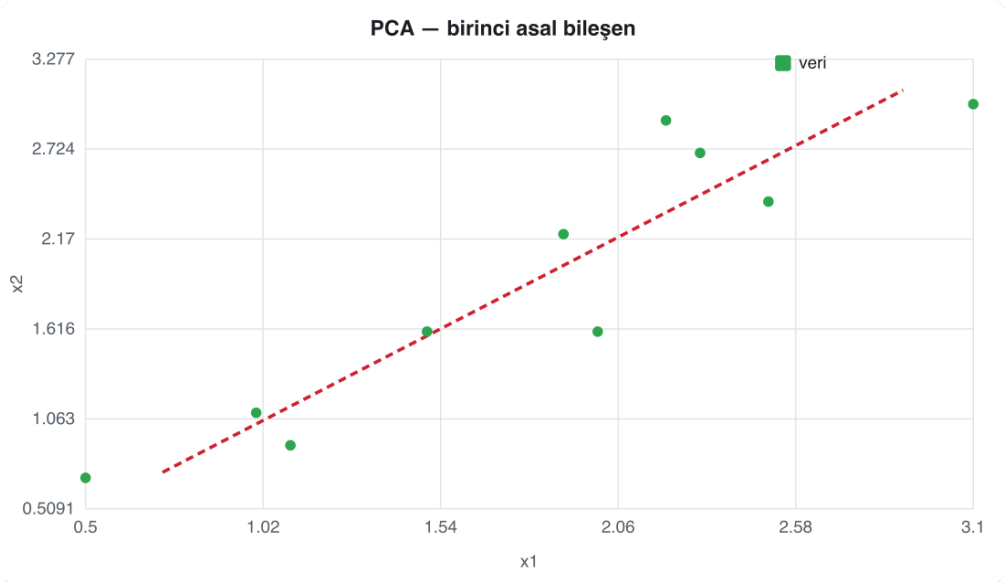
PCA, verideki en çok varyans taşıyan yönü bulur. Yöntem: veriyi ortala, kovaryans matrisini kur, onun baskın özvektörünü al — bu **birinci asal bileşendir**. Açıkladığı varyans oranı $\lambda_1 \,/\, \mathrm{iz}(C)$'dir.

```
let D = np.array([[2.5,2.4],[0.5,0.7],[2.2,2.9],[1.9,2.2],[3.1,3.0],
                  [2.3,2.7],[2.0,1.6],[1.0,1.1],[1.5,1.6],[1.1,0.9]])
let m = np.shape(D)[0]
let mu = np.mul(np.sum(D, 0), 1.0 / m)
let Dc = np.sub(D, mu)
let C = np.mul(np.matmul(np.transpose(Dc), Dc), 1.0 / m)
let pc = power_iter(C, 300)
print("PC1:", np.to_list(np.reshape(pc[0], [2])))     # [0.678, 0.735]
print("açıklanan varyans:", pc[1] / np.trace(C))      # 0.963  (%96)
```

13.9.6 13.11.6 PageRank

PageRank, bir geçiş matrisinin baskın özvektörüdür — yani kuvvet yinelemesinin bir başka uygulaması. Sütunları olasılık olan (sütun-stokastik) bir bağlantı matrisinde başlangıç dağılımını yineler, her adımda toplamına böleriz.

```
# bağlantılar: 0->1,2 ; 1->2 ; 2->0 ; 3->2
let M = np.array([[0, 0, 1, 0], [0.5, 0, 0, 0], [0.5, 1, 0, 1], [0, 0, 0, 0]])
let rank = np.full([4, 1], 0.25)
for i in range(100)
  rank = np.div(np.matmul(M, rank), np.sum(np.matmul(M, rank)))
end
print(np.to_list(np.reshape(rank, [4])))   # [0.4, 0.2, 0.4, 0]
```



Şekil 13.2: Veri bulutu ve birinci asal bileşen ekseni.

13.9.7 13.11.7 Sayısal integral — yamuk ve Simpson

Bir eğrinin altındaki alanı ızgara üzerinde tahmin ederiz. **Yamuk kuralı** uçları yarım ağırlıklar; **Simpson kuralı** `1,4,2,4,…,4,1` ağırlıklarıyla çok daha hızlı yakınsar. `∫₀^π sin = 2` ile sınayalım:

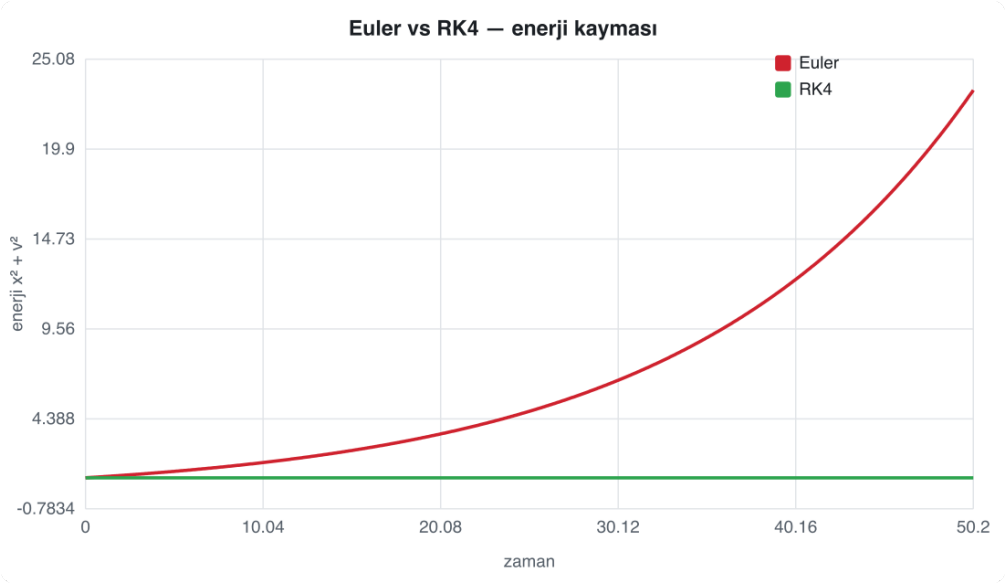
```
let n = 1001                          # tek -> çift sayıda aralık
let x = np.linspace(0.0, 3.141592653589793, n)
let y = np.sin(x)
let wlist = []
for i in range(n)
  if i == 0 or i == n - 1
    push(wlist, 1.0)
  elif i % 2 == 1
    push(wlist, 4.0)
  else
    push(wlist, 2.0)
  end
end
let h = 3.141592653589793 / (n - 1)
print((h / 3.0) * np.dot(np.array(wlist), y))   # 2.0000000000  (Simpson)
```

13.9.8 13.11.8 Diferansiyel denklemler — Euler'e karşı RK4

Bir başlangıç değer problemini `y' = f(y)` adım adım entegre ederiz. En basiti **Euler** (`y ← y + dt·f(y)`); ama hatası birikir. **Dördüncü mertebe Runge–Kutta (RK4)** dört eğim örneğini ortalar ve çok daha doğrudur. Harmonik osilatörde (`y' = Ky, K = [[0,1],[-1,0]]`) enerji `x²+v²` korunmalıdır; Euler enerjiyi şişirir, RK4 sabit tutar.

```
let A = np.array([[0, 1], [-1, 0]])
def rk4(s, dt)
  let k1 = np.matmul(A, s)
```

```
  let k2 = np.matmul(A, np.add(s, np.mul(k1, dt / 2.0)))
  let k3 = np.matmul(A, np.add(s, np.mul(k2, dt / 2.0)))
  let k4 = np.matmul(A, np.add(s, np.mul(k3, dt)))
  let toplam = np.add(np.add(k1, np.mul(k2, 2.0)), np.add(np.mul(k3, 2.0), k4))
  return np.add(s, np.mul(toplam, dt / 6.0))
end
let s = np.array([[1], [0]])
for i in range(1000)
  s = rk4(s, 0.0628318)            # ~10 periyot
end
print("durum:", np.to_list(s), "enerji:", np.get(s,0,0)*np.get(s,0,0) + np.get(s,1,0)*np.get(s,1,0))
# durum: [[~1.0], [~0.0]]   enerji: 0.99999...   (RK4 enerjiyi korur)
```



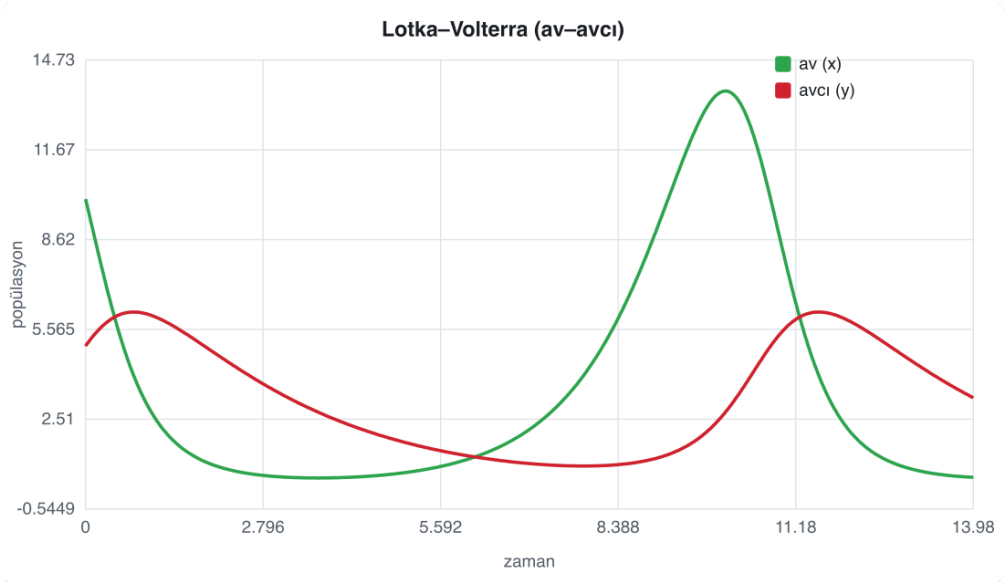
Şekil 13.3: Aynı adım boyutunda Euler enerjiyi şişirirken RK4 sabit tutar.

13.9.9 13.11.9 Lotka–Volterra (av–avcı)

Doğrusal olmayan bir sistemi de aynı RK4 ile çözebiliriz. Av `x` ve avcı `y` popülasyonları `x' = 1.1x − 0.4xy`, `y' = −0.4y + 0.1xy` ile birbirini kovalar. Durumu `2×1` bir dizi tutar; `get` ile bileşenleri okuyup türevi kurarız.

```
def lv(s)
  let x = np.get(s, 0, 0)
  let y = np.get(s, 1, 0)
  return np.array([[1.1*x - 0.4*x*y], [-0.4*y + 0.1*x*y]])
end
def rk4f(s, dt, f)
  let k1 = f(s)
  let k2 = f(np.add(s, np.mul(k1, dt/2.0)))
  let k3 = f(np.add(s, np.mul(k2, dt/2.0)))
  let k4 = f(np.add(s, np.mul(k3, dt)))
 return np.add(s, np.mul(np.add(np.add(k1, np.mul(k2,2.0)), np.add(np.mul(k3,2.0), k4)), dt/6.0))
end
let st = np.array([[10.0], [5.0]])
```

```
for i in range(50)
  st = rk4f(st, 0.1, lv)
end
print("t=5'te:", np.to_list(st))   # [[0.68...], [1.73...]]
```



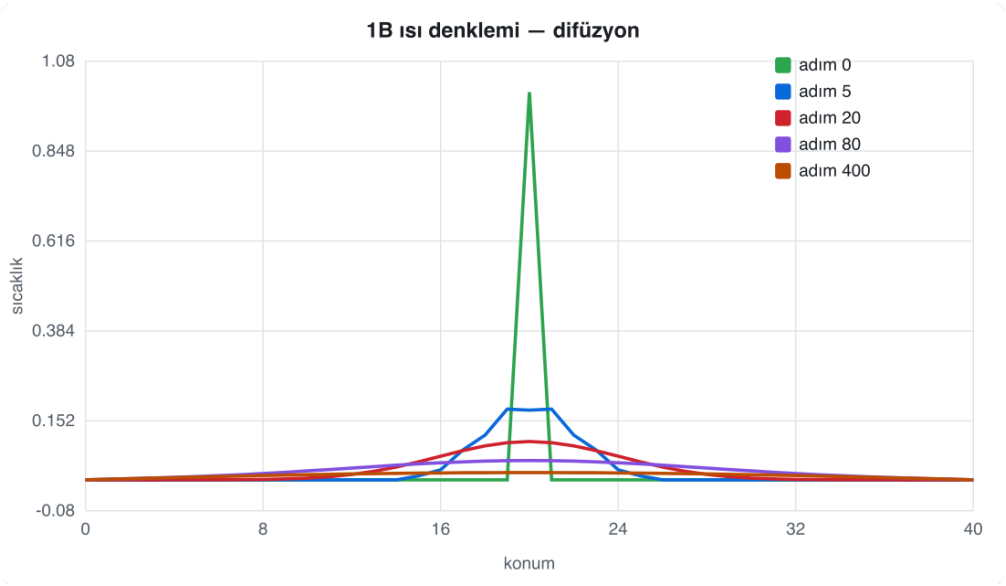
Şekil 13.4: Av ve avcı popülasyonlarının zamanla salınımı.

13.9.10 13.11.10 Isı denklemi — Laplacian matrisiyle

cobranum'da dizi dilimleme yoktur, ama komşu erişimi gerektiren bir kısmi diferansiyel denklemi **matris çarpımına** çevirebiliriz. 1B ısı denkleminin ayrık Laplacian'ı üç köşegenli `(1, −2, 1)` bir matristir; açık şema `u ← u + α·(L u)`'dur. Ortadaki bir sıcaklık çıkıntısı zamanla yayılır.

```
let m = 11
let Lrows = []
for i in range(m)
  let r = []
  for j in range(m)
    if i == j
      push(r, -2.0)
    elif i == j - 1 or i == j + 1
      push(r, 1.0)
    else
      push(r, 0.0)
    end
  end
  push(Lrows, r)
end
let Lap = np.array(Lrows)
let u = np.zeros([m, 1])
np.set(u, 5, 0, 1.0)              # ortaya çıkıntı
for t in range(200)
  u = np.add(u, np.mul(np.matmul(Lap, u), 0.4))
```

```
  np.set(u, 0, 0, 0.0)             # Dirichlet sınırları
  np.set(u, m - 1, 0, 0.0)
end
print(np.to_list(np.reshape(u, [m])))   # simetrik, pürüzsüz bir çan
```



Şekil 13.5: Başlangıçtaki sıcaklık çıkıntısının zamanla difüzyonu.

13.9.11 13.11.11 Newton yöntemi — doğrusal olmayan sistem

Newton yöntemi `F(x) = 0`'ı, her adımda Jacobian'ı `J` çözerek bulur: `x ← x − J⁻¹F(x)`. (Tersi almak yerine `solve(J, F)` kullanırız.) $x^2+y^2=4$ ile $xy=1$'i çözelim.

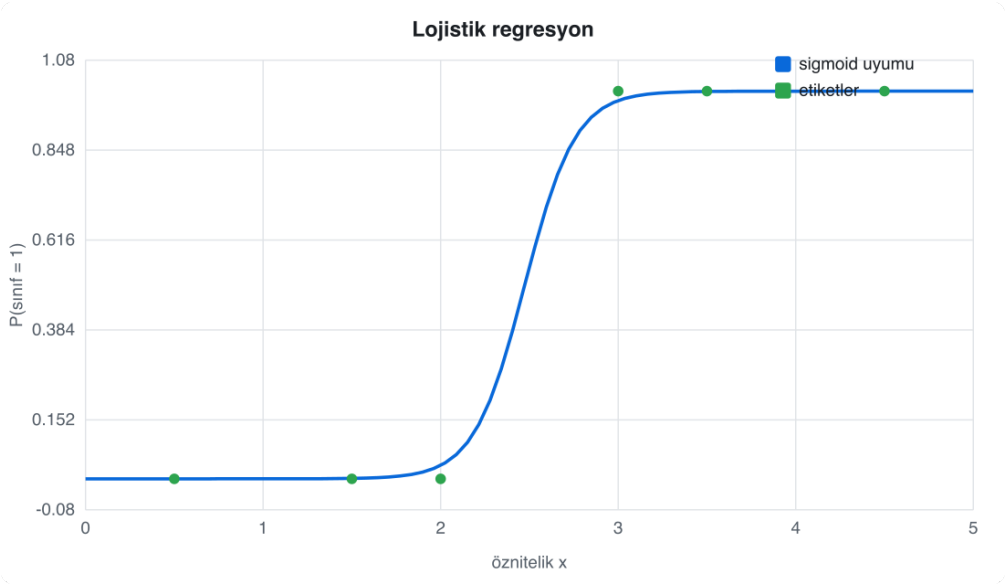
```
let s = np.array([[2.0], [0.5]])
for it in range(20)
  let x = np.get(s, 0, 0)
  let y = np.get(s, 1, 0)
  let F = np.array([[x*x + y*y - 4.0], [x*y - 1.0]])
  let J = np.array([[2.0*x, 2.0*y], [y, x]])
  s = np.sub(s, np.reshape(np.solve(J, np.reshape(F, [2])), [2, 1]))
end
print(np.to_list(s))   # [[1.9318...], [0.5176...]]
```

13.9.12 13.11.12 Lojistik regresyon

İkili sınıflandırmayı `σ(z) = 1/(1+e⁻ᶻ)` sigmoidi ve gradyan inişiyle eğitiriz. Gradyan `Xᵀ(p − y)/n`'dir; başarıyı **log-kayıp** ile ölçeriz.

```
let X = np.array([[1,0.5],[1,1.5],[1,2.0],[1,3.0],[1,3.5],[1,4.5]])
let y = np.reshape(np.array([0,0,0,1,1,1]), [6,1])
let Xt = np.transpose(X)
def sigmoid(z)
```

```
  return np.div(1.0, np.add(1.0, np.exp(np.neg(z))))
end
let w = np.zeros([2, 1])
for it in range(3000)
  let p = sigmoid(np.matmul(X, w))
  w = np.sub(w, np.mul(np.matmul(Xt, np.sub(p, y)), 0.5 / 6.0))
end
let p = sigmoid(np.matmul(X, w))
print("olasılıklar:", np.to_list(np.reshape(p, [6])))
# [0.000006, 0.0026, 0.053, 0.963, 0.998, 0.99999]  -- iki sınıf ayrışmış
```



Şekil 13.6: Eğitilmiş sigmoid sınıfları ayırıyor.

13.9.13 13.11.13 Fizik — eğik atış menzili

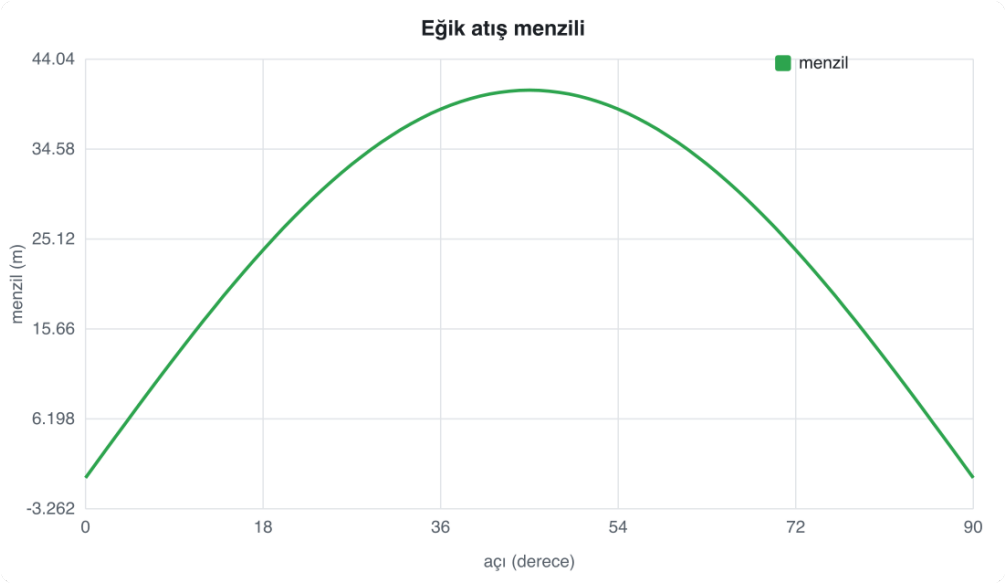
Vektörleştirmenin gücü: bir merminin menzilini `R = v²·sin(2θ)/g` ile onlarca açı için **aynı anda**, döngüsüz hesaplarız.

```
let aci = np.linspace(0.0, 90.0, 91)
let rad = np.mul(aci, 3.141592653589793 / 180.0)
let R = np.mul(np.sin(np.mul(rad, 2.0)), 20.0 * 20.0 / 9.81)
print("en uzak indeks:", np.argmax(R))   # 45  (45 derecede en uzak)
```

13.9.14 13.11.14 Sinyal — hareketli ortalama

Dilimleme olmadan bir sinyali, bantlı bir matrisle çarparak yumuşatırız: her satırı yalnızca komşularını `1/3` ağırlıkla toplar.

```
let sig = np.array([1, 2, 3, 4, 5, 6, 7, 8])
let k = 8
```



Şekil 13.7: Menzil, 45 derecede tepe yapar.

```
let Brows = []
for i in range(k)
  let r = []
  for j in range(k)
    if j == i - 1 or j == i or j == i + 1
      push(r, 1.0 / 3.0)
    else
      push(r, 0.0)
    end
  end
  push(Brows, r)
end
print(np.to_list(np.matmul(np.array(Brows), np.reshape(sig, [k, 1]))))
# [[1], [2], [3], [4], [5], [6], [7], [5]]
```

13.10 13.12 Sınırlar ve tasarım ipuçları

cobranum bilinçli olarak küçüktür. Şu an kapsamadıkları ve etrafından dolaşma yolları:

- **Karşılaştırma / maskeleme yok** (`a > b`, `where`, filtreleme). Bu yüzden "daireye düşen noktaları say" türü Monte Carlo'lar dizi-doğal değildir; bunun yerine integral tabanlı tahmin (13.11.7) ya da `random` modülüyle skaler döngü kullanın.
- **Dilimleme yok** (yalnızca `get`/`set`). Komşu erişimi gerektiren işleri (türev, evrişim, PDE) bir **operatör matrisiyle** ifade edin — ısı denklemi ve hareketli ortalama örneklerindeki gibi.
- **Yalnızca 1B ve 2B**; özdeğer/SVD doğrudan yok (kuvvet yinelemesi + deflasyon ile elde edilir).
- Bir hesabı sıcak bir döngüde tekrarlıyorsanız, ayrı çağrılar yerine **füzyon** (`hypot`, `fma`) ve **yerinde** (`*_into`) biçimleri kullanın.

13.11 13.13 Hızın kaynağı

cobranum üç katmandan hız alır. Birincisi **yoğun bellek**: bir `ndarray` kutulu nesneler değil ardışık bir tampondur, önbellek dostudur. İkincisi **çok çekirdek**: eleman bazlı işlemler ve indirgemeler, Cobra'nın GIL'siz eşzamanlılığı sayesinde çekirdeklere bölünür. Üçüncüsü **donanım çekirdekleri**: macOS'ta matris çarpımı, ters ve çözüm Apple Accelerate'in BLAS/LAPACK rutinlerine, eleman bazlı matematik ise elle yazılmış ARM NEON (SIMD) çekirdeklerine gider; bunlar AMX matris koprosesörünü ve geniş vektör yönergelerini kullanır.

13.12 13.14 API özeti

Grup	İşlevler
Oluşturma	`array`, `zeros`, `ones`, `full`, `eye`, `arange`, `linspace`
İnceleme	`shape`, `ndim`, `size`, `reshape`, `to_list`, `str`, `get`, `set`
Eleman bazlı	`add`, `sub`, `mul`, `div`, `pow`, `neg`
Tek-değişkenli	`sqrt`, `exp`, `log`, `sin`, `cos`, `abs`
Füzyonlu / yerinde	`hypot`, `fma`, `axpy`, `add_into`, `sub_into`, `mul_into`, `div_into`, `sqrt_into`, `exp_into`, `log_into`, `sin_into`, `cos_into`, `abs_into`, `neg_into`
İndirgemeler	`sum`, `mean`, `min`, `max`, `prod` (+ 2B'de eksen), `var`, `std`, `norm`, `argmin`, `argmax`, `cumsum`, `clip`
Lineer cebir	`matmul/dot`, `transpose`, `inv`, `det`, `solve`, `trace`, `diag`, `outer`, `concat`

cobranum, Cobra'nın betik rahatlığını sayısal hesaplamanın gücüyle birleştirir: birkaç satırda bir regresyon uydurur, bir sistem çözer, bir PDE'yi ilerletir ya da bir modeli eğitirsiniz — hepsi tek bir küçük eklentiyle.