Giriş

```
puts 'Merhaba Dünya'
```

Programlama

- MİB
- Bellek
- Giriş/Çıkış

- Programı belleğe yükle (İşletim Sistemi)
- Denetimi programa ver (İşletim Sistemi)
- Bellekte sırayla çalışan buyruklar

- Sınırlı sayıda buyruklar $\rightarrow$ Buyruk kümesi (instruction set)
- Buyruğu veya işlem sonucunu tutan kayıt alanları $\rightarrow$ Kaydediciler (registers)
- Aritmetik ve Mantıksal işlemleri yerine getiren birim $\rightarrow$ ALB (ALU)

Oyuncak Makine

- Kaydediciler: sadece 1 tane $\rightarrow$ Akümülatör (Birikeç)
- Buyruk kümesi: 14 buyruk

İki sayıyı topla

```
start  load this
       add result
       store result
       load that
       add result
       store result
       load result
       print
       stop
this   3
that   5
result 0
```

Kaynak kod

Problemin çözümünü ilgili programlama dilinin sözcük ve kurallarıyla anlatan tarif

MİB'nin anladığı tek dil: makine dili

- Programın çalıştırılması: Kaynak kodla yapılan tarifin MİB'nin dilindeki buyruklara dönüştürülmesi

Tarifin hayata geçirilmesi ("programın çalıştırılması")

- Önce kaynak kodun tamamını makine diline çevir $\rightarrow$ Derleme (compile)
- Kaynak kodu (tarifi) bir programa girdi olarak vererek tarifteki her cümlenin gereğinin MİB'ne bu program tarafından yaptırılmasını sağla $\rightarrow$ Yorumlama (interprete)

- Kaynak kod bir tarifin hayata geçmesi için tek başına yeterli değil
- Sadece makine dilinde yazılan bir tarif doğrudan yeterli (ki onda bile bir tür işlemeye ihtiyaç var, bk. örnekte yapılan bellek ilklendirmeleri)
- Bir derleyiciye veya bir yorumlayıcıya ihtiyaç var

Derleme

(Aşırı basitleştirme içerir)

- Kaynak kodu hedef MİB'in buyruklarından oluşan makine diline çevir
- Bu işlem program çalıştırılmadan önce **bir seferliğine** yapılır
- Derlenmiş biçimdeki program çalıştırılır
- Bu modelde program işletim sistemi tarafından doğrudan yüklenerek çalıştırılıyor

```c
#include <stdio.h>

static int this   = 3;
static int that   = 5;
static int result = 0;

int main()
{
    result = this + that;

    printf("%d\n", result);
}
```

Nesne kodu

Object code

- Derleme sonucunda elde edilen imajı (ör. çalıştırılabilir kipte bir ikili program dosyası) anlatır
- Kaynak kodun devamında yer alan bir terim
- Terimde geçen nesneyi "Nesne Yönelimli"deki (Object Oriented) nesne ile karıştırmayın

Yorumlama

(Aşırı basitleştirme içerir; derlemeye göre daha da aşırı)

- "Yorumlayıcı" programı belleğe yükle
- Yorumlayıcı kaynak kodu okur; artık denetim yorumlayıcı programda
- Yorumlayıcı, kaynak koddaki anlamlı çalıştırma cümlelerini (ör. satırlar) sırayla yorumlar
- Yorumlama? Cümleyi anlamlandır ve gereğini MİB'ne (onun anladığı buyruklarla) yaptır
- Bu modelde program işletim sistemi tarafından yüklenenen bir yorumlayıcının aracılığıyla çalıştırılıyor

```
this = 3
that = 5

result = this + that

puts result
```

Çalışma zamanı

Önemli bir terim: "çalışma zamanı" $\rightarrow$ **runtime**

- Programın çalıştırılması süresince geçen zaman dilimini anlatıyor
- Derlenen programlarda, derlenmiş program imajının belleğe yüklenip MİB tarafından çalıştırılmaya başlandığı andan, sonlandığı ana kadar geçen süre
- Yorumlanan programlarda, kaynak kodun yorumlayıcı tarafından çalıştırılmaya başlandığı andan, sonlandığı ana kadar geçen süre

Dinamik programlama dilleri

- Kaynak kod üzerinde çalışma zamanı dışında yapılan başka işlemler de var
- Bu süreçler de farklı şekilde adlandırılabiliyor, ör. derleme zamanı (compile time)
- Yorumlanan bir program dilinde kararlar çalışma zamanında dinamik olarak alındığından bu dillere "dinamik program dilleri" de deniliyor
- "Dinamik" teriminin karşı tarafındaki terim: "Statik"
- Bu nedenle kaynak kod üzerinde çalışma zamanı dışında gerçekleşen süreçler genel olarak "statik" terimiyle vasıflandırılıyor
- Örnek: Statik kod çözümlemesi

Yüksek/alçak seviye diller

Bilinmesinde yarar olan bir terim çifti

- Bir programlama dilinde sunulan soyutlamalarla ifade kabiliyeti ne kadar yüksek ise dil de o kadar "yüksek seviye" (high-level) bir dil oluyor
- Karşısındaki terim "alçak seviye" (low-level); soyutlamalar daha az, donanıma daha yakın (ve bir o kadar da denetim olanağı)
- Yüksek/alçak diyerek dilin kalitesine ilişkin bir sıfat oluşturmuyoruz, bu teknik bir tartışma
- Bunlar göreceli terimler, mutlak anlamda kullanmayın
- Örnek: Go, Ruby'ye göre alçak-seviye bir dildir, ama C'ye göre yüksek-seviyelidir
- Yorumlanan (dinamik) diller derlenen dillere göre hemen hemen daima yüksek-seviyeli

Derleme/Yorumlama

- "Hesaplama" (computing) süreçlerini anlamak için yararlı
- Günümüzde artık çok anlamlı terimler değil (bk. JIT, bytecode, garbage collector)
- Pek çok gerçeklemede "yorumlama" sürecinde bir tür derleme yapılıyor (çalışma zamanında)
- Derleme bazen doğrudan MİB'i hedeflemiyor, sanal bir MİB hedefleniyor (ör. Java sanal makinesi)

- Bu terimler programlama dilinin gerçeklemesiyle ilişkili; programlama diline ilişkin mutlak bir özelliği anlatmıyor
- Bir programlama dili, en azından kuramsal olarak, hem derlenen hem yorumlanan biçimde gerçeklenebilir
- Fakat dil (ortaya çıkışında belirlenmiş) doğası itibarıyla bir tür gerçeklemeyi daha etkin kılar veya bir tür gerçeklemeyi teknik olarak çok zorlaştırır
- "Derlenen/yorumlanan dil" yerine "Kaynak kodun derlenerek/yorumlanarak çalıştırılması öngörülen dil"
- Ör. Ruby, Python, Javascript yorumlanarak çalıştırılması öngörülen diller
- Ör. C, Go, Rust derlenerek çalıştırılması öngörülen diller

Derlenen dil

Avantajlar

- Çalışma zamanında yorumlama olmadığından (veya minimize edildiğinden) çok daha hızlı
- Bellek kullanımı daha az
- Sorunlar program çalışmadan önce (derleme aşamasında) yakalanabilir
- Lojistiği daha kolay; hedef platform için derlenmiş programın kurulumu yeterli, ayrıca bir yorumlayıcı kurmanıza gerek yok

Dezavantajlar

- Yazılması daha maliyetli (derleyiciyi mutlu etmek zorundasınız, tip bildirimleri gibi daha ayrıntılı tarifler gerekiyor)
- Çalışma zamanı üzerinde denetiminiz olmadığından "dinamik" işler çeviremezsiniz
- (C gibi en azından bir kısım dilde) Çalışma zamanında güvenlik açıkları

Yorumlanan dil

Avantajlar

- Geliştirme süresi daha kısa (arada zeki bir yorumlayıcı var, daha kısa lafla çok iş)
- Çalışma zamanı denetlenebildiğinden "dinamik" işler çevrilebilir
- Çalışma zamanı denetlenebildiğinden basit güvenlik açıkları yaşanmaz
- Daha "taşınabilir" (portable); yazdığınız kodun ilgili platformda çalışması için yorumlayıcının o platformda kurulu olması yeterli (fakat bk. lojistik)

Dezavantajlar

- Daha yavaş
- Daha fazla bellek tüketimi
- Çalışma zamanında yaşanan sürpriz hatalar (derlenebilseydi çalıştırmadan önce yakalanabilirdi)
- Artan lojistik yük (yorumlayıcı kurulumu gerekiyor)

Yorumlanması öngörülen bir dilde programın çalışma süresi ve bellek tüketimini artırmak pahasına, programın geliştirme süresini azaltıyoruz

- Birim zamanda daha fazla iş
- Daha çabuk hayata geçen fikirler

(Ama ile başlayacak eleştirilere açık bir yargı)

Dinamik (yorumlanan) bir dilde geliştirici konforu hedeflenir

- Sistem kaynaklarını (MİB, bellek vs) daha konforsuz bir durumda tutmak pahasına

Günümüz trendleri

- Ayrım yine korunmakla birlikte her iki türün en iyi özellikleri dillere eklenebiliyor
- Yorumlanan dillerde tip bildirimleri
- Derlenen dillerde çalışma zamanını denetleyen eklemeler (ör. çöp toplayıcı)
- Teknik olarak geçerli, fakat pratikte hatalı kod parçalarını geliştirme aşamasında yakalayan zengin statik çözümlemeler ("lint"leme)

Ruby

Değişken

İsimlendirilmiş bellek hücresi

- Bellek hücresinde (bir tür) veri var
- Veriye anlamlı bir isimle erişiyoruz

```
kur = 8.96
dolar = 100.0

tl = kur * dolar
```

```
oran = 18.0 / 100
fiyat = 100.0

kdv = fiyat * oran
```

İsimlendirme

Söz dizimi (sentaks) kuralları

- İlk karakter İngilizce alfabedeki küçük/büyük harflerden biri veya alt tire (_) olmalı
- Varsa devam eden karakterlerde ilkine ilave olarak rakamlar kullanılabilir (ama ilk karakter rakam olamaz)

- Sadece değişkenler değil, metot adları, sabitler, sınıf/modül adları da (Ruby'de bunlar da birer sabit isim) isimlendirmenin kapsamında
- Bu isimlere genel olarak "tanımlayıcı" (identifier) deniliyor
- İsimlendirme söz dizimi kuralları $\rightarrow$ Tanımlayıcı söz dizimi kuralları

Uygun isimlendirme kod okunurluğunu çok artırır

- Her program bir öykü veya (uzunluğuna göre) bir roman
- İsimler bu öykünün kahramanları
- Anlamlı isimler öykünün okunmasını kolaylaştırıyor

Türkçe karakterler?

ç	ğ	ı	ö	ş	ü
Ç	Ğ	İ	Ö	Ş	Ü

- `ı` ve `İ`'ye dikkat! (`i` ve `I` Türkçe'ye özgü değil)
- Değişken adlarında Türkçe karakter çoğu durumda kullanabiliriz, ama kullanmamalıyız

- Programlama evrensel bir etkinlik
- Programlama dillerinin anahtar kelimeleri de İngilizce
- İsimlendirmeleri enternasyonal yapmakta yarar var; özellikle her dilden geliştiricinin katkı sunabileceği açık kaynak projelerde

```
exchange_rate = 8.96
usd = 100.0

tl = exchange_rate * usd

tax_rate = 18.0 / 100
price = 100.0

tax = price * tax_rate
```

İfadeler

- Değerlendirmeye (evaluation) konu ögeler
- Değerlendirme? Hesaplama, değer verme
- Örnek: `exchange_rate * usd`
- Bu bir aritmetik ifade
- Örnek: `usd = exchange_rate * usd`
- Bu (aritmetik ifade içeren) bir "atama" (assignment) ifadesi

Her ifade bir değer döner (değerlendirme sonrası)

- Ruby'de ilkel değerlerin bizzat kendisi de ifade
- Örnek: `100.0`
- `usd = 100.0` atama ifadesinde önce sağ taraf değerlendirilir, dönen değer (`100.0`) sol taraftaki değişkene atanır

Ruby'de her şey bir ifadedir

- IRB'de girilen bir satır `enter` tuşu ile yorumlayıcıya gönderilir
- IRB, satırı bir bütün halde ifade olarak yorumlar
- İfadenin döndüğü değer `#=>` ile belirtilir

Aritmetik operatörler

- Operatör $\rightarrow$ İşleç
- Sayısal türde değerleri operatörlerle düzenleyerek dönüş değeri yine sayısal türde olan aritmetik ifadeler kurabiliyoruz
- Sayısal tür? Tam sayı, Gerçel sayı, Rasyonel sayı
- Aritmetik operatörler beklediğiniz gibi: +, −, ∗, /
- Ayrıca iki operatör: % modülüs ve ∗∗ üs alma operatörleri

Sayısal tür

- Tam sayı ve gerçel sayılar
- Gerçel sayılarla kurulan ifadelere dikkat! `18.0 / 100` yerine `18 / 100` yazılırsa?

Ruby'de Rasyonel sayıların gösterimi için özel bir söz dizimi kullanılıyor

```
tax_rate = 18/100r
```

- Daha okunur
- Bunu nasıl kullanacağız? Göründüğü gibi, ör. `18/100r * 100.0`

Tür dönüşümleri yapılabilir

- Değer nesneleri üzerinde çalıştırılacak iki metot: `to_i` ve `to_f`
- Bir değeri tamsayıya çevirmek için `to_i`, ör. `18.9.to_i #=> 18`
- Bir değeri gerçel sayıya çevirmek için `to_f`, ör. `18/100r.to_f #=> 0.18`
- Değer bu dönüşümü desteklemeli

Fonksiyonlar

```
value = Math.sin(0.5236) # 0.5236 ~ Pi/6 ~ 30 derece
```

- Matematikte aşina olduğumuz bir trigonometrik fonksiyon: `sin`
- `<fonksiyon>(girdi listesi) → çıktı`
- Fonksiyonlara giriş değerlerini argümanlar yoluyla iletiyoruz, örnekte 30 derece `sin` fonksiyonuna iletiliyor
- Fonksiyon (isminin yansıttığı) hesaplamayı yapıp bir değer dönüyor, örnekte `0.5`
- Ruby'de bir fonksiyona geçirilen argümanlar etrafında parantez kullanmanız her zaman gerekmiyor

Matematiksel fonksiyonlardan bir parça farklı olarak programlamada yazacağınız fonksiyonlar:

- Hiç argüman istemeyebilir
- Birden fazla argüman isteyebilir
- Bir değer dönmeyebilir
- Dönecekse sadece tek bir değer döner (bazı dillerde, ör. Go, birden fazla değer dönülebilir)

Fonksiyon veya Metot

Ruby gibi Nesne Yönelimli dillerde fonksiyon yerine metot adlandırması tercih ediliyor

- Bu bir isimlendirme inceliği (bazı nedenleri var, gelecekte daha ayrıntılı değineceğiz)
- Bundan sonra fonksiyon değil metot diyeceğiz

Metotlarla ilk karşılaşmamız:

```
puts 'Merhaba Dünya'
```

- `puts` bir metot (yani fonksiyon)
- Metotlara genel olarak bir nesne üzerinde `.` operatörüyle erişiyoruz
- Fakat bu örnekte metot bir nesne üzerinden değil doğrudan çağrılıyor
- Bu konuya gelecekte değineceğiz

Nesne Nokta Metot notasyonu

Notasyona dikkat edin! `18.to_f #=> 18.0`

- Noktanın solunda bir değer: `18`, sağında ise bir metot: `to_f` bulunuyor
- Noktanın solundaki "değer" aslında bir "nesne" (object)
- Nesnelere `.` operatörü yoluyla bir mesaj iletiyoruz
- Mesaj $\rightarrow$ Metot
- Nesne mesajın gereğini yerine getiriyor (ilgili metot çağrılıyor)

Ruby'de hemen her şey bir nesne

- Nesneleri `.<metot>` söz dizimiyle uyarıyoruz

İlkel veri türleri

- Sayısal türler ilkel (primitive) veri türlerinin en yaygın örneği
- Pek çok programlama dilinde bir diğer önemli veri türü: “dizgi” (string)

Dizgi

```
message = 'Merhaba Dünya'

puts message
```

- Örnekteki `'Merhaba Dünya'` değeri bir dizgi (string)
- Çift tırnak veya tek tırnak kullanabiliriz

```ruby
who = 'Dünya'
message = "Merhaba #{who}"

puts message
```

- Çift tırnakta Ruby dizgi değerini özel olarak yorumlar → "Dizgi Enterpolasyonu" (String Interpolation)
- #{} arasına istediğiniz karmaşıklıkta bir Ruby kodu yazabilirsiniz
- Yorumlayıcı #{} arasındaki kodu bir ifade olarak değerlendirir ve dönüş değerini yerine koyar
- Bu örnekte tek tırnak kullanılsaydı `message` dizgisi olduğu gibi (literal) yorumlanacaktı

Dizgiler programlama dillerinde çok temel bir veri türü

- Her bir tespih tanesi bir “karakter” (character, char) olan bir tespih gibi

Karakter

- Dizgilerin yapıtaşları; kabaca harfler, rakamlar ve noktalama işaretleri
- Bunlara ilave kontrol karakterleri var: boşluk, satır sonu, sekme gibi
- Karakterler belirli sayıda bitlik bir bilgiyle kodlanıyor
- En bilineni 7 bitlik ASCII: American Standard Code for Information Interchange
- Türkçe gibi dile özgü karakterler ASCII tabloda yok
- Bunun yerine günümüzde UTF-8 gibi daha evrensel kodlama standartları kullanılıyor
- Yine de ASCII tabloya hakim olmalısınız (örneğin UTF-8 ASCII'nin bir tür üst sürümü)

ASCII Tablo

Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@
1	SOH (start of heading)	33	!	65	A
2	STX (start of text)	34	"	66	B
3	ETX (end of text)	35	#	67	C
4	EOT (end of transmission)	36	$	68	D
5	ENQ (enquiry)	37	%	69	E
6	ACK (acknowledge)	38	&	70	F
7	BEL (bell)	39	'	71	G
8	BS (backspace)	40	(	72	H
9	TAB (horizontal tab)	41	)	73	I
10	LF (NL line feed, new line)	42	*	74	J
11	VT (vertical tab)	43	+	75	K
12	FF (NP form feed, new page)	44	,	76	L
13	CR (carriage return)	45	-	77	M
14	SO (shift out)	46	.	78	N
15	SI (shift in)	47	/	79	O
16	DLE (data link escape)	48	0	80	P

Ruby'de karakterler özel bir veri türü değildir

- Ama örneğin C gibi bazı programlama dillerinde çoğunlukla `char` adında özel bir veri türüdür (Ruby'den farklı olarak C programlama dilinde dizgi veri türü yoktur)

Ruby'de bir karakterin ASCII tablodaki onluk tabanda kodunu öğren: `.ord`

```
'a'.ord
' '.ord
"\n".ord
"\t".ord
```

Onlu tabanda verilen bir kodu karakteri içeren dizgiye çevir: `.chr`

```
97.chr
```

Özel karakterler

- `"\n"` $\rightarrow$ Satır sonu
- `"\t"` $\rightarrow$ Sekme
- Bunlar en yaygınları, bunların dışında ters bölü karakteriyle nitelendirilen başka kodlar da var

Beyaz boşluk (whitespace)

- Kabaca; boşluk, satır sonu ve sekme karakterlerine deniliyor (ama başkaları da var)
- Dizgi içinde kullanılmadığında, kaynak kod ayrıştırılırken bu karakterler göz ardı edilir veya kodun söz dizimsel olarak farklı parçalarını birbirinden ayırır