# BASH Programlama

Mehmet Tahir SANDIKKAYA

Ağustos 2000

## 2020 yılındaki düzenlemeye önsöz

Yirmi yıldan uzun zaman önce yazdığım bir yazı –bir nedenle– anımsayıverdiğim için tozlanmış manyetik izlerin arasından çıkarıldı. Yazının neye benzediğini de tamamen unutmuş olduğum için baştan sona bir kez okudum. İlginç biçimde, aklımda kalandan çok daha fazlasını anlatmış olduğumu, anlattıklarımın hâlâ geçerli olduğunu görüp şaşırdım. Yazının öğreticiliğini korumuş olduğunu gördüğüm için mutluyum.

Öte yandan, buna benzer bir kılavuzu bugün yazacak olsam pek çok anlatımı, pek çok sözcüğü, bazı konuların derinliğini, daha önemlisi yazının seslenişini oldukça değiştirirdim. Yazıda, bugün olsam belki deneyimle kazandığım için belki daha sonra karşılaşıp öğrendiğim için gördüğüm hatalar, hiç değilse kötü kullanımlar, kötü terminoloji var.

Buna karşın, özgün yazının içerdiği –okuyanın belki de hiç sezmeyeceği– kimi eksiklikler ya da gariplikler benim için kişisel belge sayılır. O nedenle, hatalı yazımların düzeltilmesi ve birkaç sözcükteki zorunlu saydığım değişiklikler dışında metin 2000 yılındaki vasfını korumaktadır. Bir farkla; bu düzenlemeyle yazıyı özgün HTML biçeminden daha okunaklı olacağını düşündüğüm LaTeX biçemine dönüştürmüş de oldum. Yazının kimi yerlerinde artık anlamlı olmasa da bunun izleri duruyor, şaşırmayın.

Özgün yazı, bir zamanlar hazırladığımız çevrimiçi bir dergi için yazılmıştı. Bu yazı daha sonra yine kendi hazırladığımız, pek ilgi görmeyen bir dağıtımın yardım kılavuzları içinde yer aldı. Türkiye'de hazırlanan erken dönem Linux dağıtımlarından *Gelecek Linux* ve *Pardus*'un ilk sürümlerinde de yardım kılavuzları içindeydi diye anımsıyorum.

Kasım 2020, İstanbul

# Önsöz (Konuyla ilgili değildir.)

Merhaba LinuxMag'cılar. Türkiye'nin hiçbir kuruma bağlı olmayan, tam anlamıyla özgür ve sansürsüz ilk ve tek[1] online Linux magazinine hoş geldiniz. Bu yazıda Linux işletim sisteminin gözde kabuklarından BASH'i[2] programlamak konusunda kısa ve yararlı bilgiler bulacaksınız. Fakat bu yazıyı okumanın bazı ön koşulları da var. Öncelikle bir Linux kullanıcısı olmalısınız! Bu da BASH kullanımı ve komutları hakkında yeterli bilgiye sahip olduğunuz manasına gelir. Unutmayın ki, bu yazının konusu BASH programlama. O yüzden yazı içinde komutlar hakkında sizce yeterli açıklamayı bulamazsanız darılmayın. Daha iyisi, siz bilmediğiniz komutların ne işe yaradığını öğrenmek için: `$ man <bilmediğiniz-komut>` yazın. Kılavuz (manual) sayfaları işinize yarayacaktır. Umarım...

# İki çift laf

**1. çift** Bu yazı ISO 8859-9 (Latin 5) yazı formatında, yani Türkçe olarak hazırlandı. Hatasız görünüm için tarayıcınızı Türkçe gösterecek şekilde ayarlayın.

**2. çift** HTML formatının azizliklerinden ötürü, aslında her bir komutun tek bir satır üzerinde kalması gerektiği durumlarda dahi ence daraltılmış sayfalarda tek satırda başlayıp bitmesi gereken bu satırlar alt satırlara kayabilir, ne yazık ki, HTML kullandığım sürece bu hatayı etkin biçimde gideremiyorum. Dileğim, tarayıcı pencerenizi ence olabildiğine geniş tutmanız ve/veya bu biçimde hatalı gösterilebilecek satırlar olduğunu akılda tutup hataları bertaraf etmeye çabalamanızdır.

[1] Aslına bakarsanız bildiğim kadarıyla Türkiye'de ne online ne offline, ne de kağıt üzerinde bir Linux dergisi yok. Daha neler, dergiler Linux hakkında ek bile vermiyor. Eylül ayında çıkacak bir "GNU Linux" dergisinden bahsediliyor. İnşallah.

[2] Bourne Again SHell: Stephen Bourne'un 1979'da geliştirdiği kabuğun yeni yetenekler eklenmişi.

# 1 Giriş

Linux'taki BASH[3]; C, Pascal, Fortran, Basic gibi sık kullanılan buyuru dillerinin birçok özelliğini taşır. BASH'te de onlardaki gibi buyruklardan, döngülerden, mantıksal ve –sınırlı da olsa– aritmetik işlemlerden söz edilebilir. Bu yönleriyle BASH kullanması ve programlaması kolay bir kabuktur.

Önce programlamanın ne olduğunu bir gözden geçirelim. BASH programlama ile anlattığımız bilgisayarın özel durumlarda çalıştıracağı bir veya daha fazla komuttan oluşan –genellikle– karışık işleri daha kısa yoldan (bizim belirleyeceğimiz yeni bir komutla) bitirmektir. Her gün kullandığımız `ls`, `mkdir`, `cp` gibi komutlar aslında BASH yazılırken tanımlanmış programlardır –diyebiliriz. Biz de BASH ile birlikte tanımlanmış ve sistemimizde halihazırda bulunan bu komutları kullanarak daha karışık programlar (dolayısıyla yeni komutlar) oluşturabiliriz.

## 1.1 En basit yoldan, ilk BASH programı

```
$ alias ll="ls -la --color | more"
```

Evet, yukarıdaki satırı yazıp "return" (enter) tuşuna bastıysanız ilk BASH programınızı yazdınız demektir. Bunu kontrol etmek için ekrana `l` yazıp iki kere "TAB" tuşuna basmayı deneyin. Karşınıza çıkacak listede `ll` ögesini görürseniz başardınız. Yukarıdaki satır benim en sık kullandığım komutlardan biri. Bu satır sayesinde kendimize `ll` şeklinde yeni bir komut yarattık ve bu komutun sistemde `ls -la --color | more` komutunu yerine getirmesini sağladık. `alias` komutu BASH altında bir komut veya komut takımı için yaftalar yaratır. Burada da siz bilgisayarınızı kapatana kadar `=` operatörünün sağ tarafındaki komut takımının yaftası işaretin sol tarafında girdiğiniz metin olacaktır. Eğer siz de benim gibi bunu bilgisayara her girişinizde tekrar yapmak istemezseniz bu komutu ev dizininizdeki `.bashrc` dosyasının en sonuna ekleyin. Bu dosya sizin bu tür yaftalarınızı ekleyebileceğiniz ve sistemin her açılışta okuyup içindeki yaftaları komutlar arasına eklemek için kullandığı bir dosyadır. Bu sayede, çok hızlıca, hem içinde bulunduğunuz dizinin içeriğini, hem gizli dosyaları, hem dosya ayrıntılarını renkli olarak görebilecek, hem de dosya sayısının fazla olduğu durumlarda listedekileri gözden kaçırmayacaksınız.

## 1.2 Klasik yoldan aynı BASH programı

Şimdi `ll` adlı bir dosya oluşturun.

```
$ touch ll
```

Şimdi de bu dosyanın içeriğini oluşturalım.

```
$ echo "ls -la --color | more" >> ll
```

Çok güzel. Gerçekte, işin zor kısmını bu şekilde bitirmiş olmamıza rağmen

[3]Benim bu yazı boyunca "BASH" yazdığım yerlere "BASH Kabuğu" yazanlar da oluyor. Bu kafanızı karıştırmasın. Aslında aynı şeyden bahsediliyor. Bu fark BASH'in açılımında gizli. BASH'in açılımının sonunda "shell" yani kabuk kelimesi zaten var. Bu yüzden "BASH" yerine "BASH Kabuğu" denmesi, "GAP Projesi" veya "HIV Virüsü" tarzında bir hatadır.

bu programın çalışması için biraz düzenlemeye ihtiyaç var. Şimdi bu yoldan yapılan programın üstteki örnekten ayrılan yönlerini belirleyelim:

1. `ll` dosyasının içerdiği komutu (komutları) ancak `ll` dosyasının bulunduğu dizine gidip `./ll` yazarak çalıştırabiliriz.
2. `ll` dosyasının içerdiği komutu (komutları) ancak BASH altında çalıştırabiliriz.
3. Yazdığımız programı bir dosya içinde (bu örnekte `ll`) sakladığımız için sistem kapandığında bu programı kaybetmeyiz.

Bu farklılıkları lehimize kullanmak için ne yapacağız? İlk işimiz şu sıkıntı veren `./ll` yazımından kurtulmak olsun. Bundan kurtulmak için sistemde önceden tanımlı olan bir değişkenin değerini, `PATH` çevresel değişkeninin değerini değiştirmeliyiz. Bu değişken sistemdeki çalıştırılabilir dosyaların yerlerini (kök dizininden itibaren hiyerarşik yapıdaki yerlerini) gösterir. Bu değişkenin değerini değiştirmek için `/~.bash_profile` dosyanızda değişiklikler yapmalısınız. `PATH` ile başlayan bir satır varsa bu satırın (ya da alt satıra geçmişse altındaki satırın) sonuna, çift tırnak işaretinin hemen soluna `:<kypkid>`[4] yazın. Eğer böyle bir satır yoksa, `PATH="<kypkid>"` yazın. Bu dosyada yapacağınız değişiklik sistemi bir sonraki başlatışınızda ve sonrakilerde etkin olacaktır. Sistemi başlatmadan hemen bunu kullanmak isterseniz; `export PATH=$PATH:<kypkid>` yazın.

Diğer meselenin anlaşılması için biraz da ön bilgi vereceğim. Herhangi bir GNU/Linux işletim sistemi iki parçadan oluşur: çekirdek (kernel) ve kabuk (shell). Çekirdek bilgisayarın donanımıyla bağlantı kuran ve bilgisayarı yöneten yazılımdır. "Linux" denilen aslında bu çekirdeğin adıdır. Kabuk ise kullanıcının kullandığı, çekirdeğe buyruk verilebilen arayüzün adıdır. Bu arayüz çoğunlukla "GNU" vakfının geliştirdiği programlarla etkin biçimde kullanılır. Kabuk, kullanıcının verdiği komutları çekirdeğin anlayacağı dile, yani sistem çağrılarına çevirir. Böylece kullanıcının bilgisayara hükmetmesi sağlanır. Linux sistemlerinde kullanılan çok sayıda kabuk vardır. Bunlar arasından tanımlı olan ve en sık kullanılan kabuk BASH'tir. Fakat, BASH kullanmayan biri sizin yazdığınız programları kullanmak isterse ne olacak? Programlarınızın kullanılabilmesi için, çekirdeğe o programın BASH için yazıldığını bildirmemiz gerekir. Bunun için de programımızın komutlarını yazdığımız dosyanın başına çekirdeğin hangi kabuk vasıtasıyla programı çalıştıracağını bildiren bir satır ekleriz. Bu, çekirdeğe gönderilen ve her kabuk için aynı biçimde kullanılan bir ifadedir.

Şimdi yukarıdaki iki eksiği telafi edilmiş olarak programı bir kez daha en baştan başlayarak yazalım.

```
$ mkdir ~/kypkid
$ export PATH=$PATH:~/kypkid
$ cd ~/kypkid
$ touch ll && echo "#!/bin/bash" > ll
$ echo "ls -la --color | more" >> ll
$ chmod 755 ll
```

[4] kendi yazacağınız programları koymak istediğiniz dizin

Evet, artık komut satırında `ll` yazıp klasik (ve en çok kullanacağınız) yoldan ilk BASH programınızı yaptınız. Yukarıda yaptıklarımı anlatayım.

- İlk satırdaki komut ile, ev dizinimde[5] kendi yazdığım programlarımı düzenli olarak yerleştireceğim bir dizin yarattım. Sisteminizin düzeni açısından bunu size de tavsiye ederim.

- İkinci satırda ise sistemin doğrudan çalıştırabileceği programların hangi dizinlerde bulunduğunu gösteren `PATH` çevresel değişkenine yarattığım dizini de ekledim. (Ayrıntılı bilgiyi ön tanımlı değişkenler başlığı altında bulabilirsiniz.)

- Üçüncü satırda yarattığım dizine geçtim.

- Dördüncü satırda `touch` komutu ile `ll` isminde bir dosya oluşturdum ve `echo` komutunu kullanarak tırnak içindeki yazıyı dosyaya aktardım. Aynı satırda çok sayıda komutu, sırasıyla ve birbiriyle ilintili çalıştırmak için komutlar arasına `&&` konulacağını hatırlatmalıyım. Gerçekte, `&&` operatörü ifade yazmakta kullanılır ve iki yanındaki ifade ya da buyrukların doğru olup olmadığını denetler. Buradaki kullanımıyla BASH önce solda kalan komutu çalıştırıp sonucunda hata olup olmadığına bakar. Hata varsa, zaten sağdaki komutu çalıştırmaya gerek kalmadan satırda yazılı ifadenin sonucu "yanlış" olarak belirlenmiş olur ve yapılacak işten tasarruf etmek için sağdaki komut hiç çalıştırılmaz. Soldaki komut hatalı çalışmamışsa, yani "doğru"ysa BASH ifadenin değerini öğrenmek için sağdaki komutun sonucunu da komutu çalıştırarak öğrenmek zorundadır. Böylece, bu yazımla, soldaki komut doğru çalıştıysa sağdakini çalıştır demiş oluruz.

- Beşinci satırda yine `echo` komutu yardımıyla tırnak içindeki yazıyı bu kez dosyaya yazmak yerine dosyanın sonuna ekledim. İkisi arasındaki fark `>` ile `>>` kullanımından kaynaklanır. `>` kullandığınız zaman tırnak içindeki yazı dosyayı oluşturur. Dosyada daha önceden kaydedilmiş şeyler varsa bunlar silinip üzerine yazılacaktır. `>>` kullandığınız zaman tırnak içindeki yazı dosyanın sonuna eklenir ve daha önceden dosyada kayıtlı olan şeyler silinmez. Bilenler soracaklardır neden dosyayı bu yolla oluşturduğumu. Biraz çeşit olması zarar vermez. Ve elbette ki dilerseniz `pico`, `joe`, `ed` vb. metin editörlerini kullanabilirsiniz.

- Son satırda izinlerini değiştirmek yoluyla yazılan programı çalıştırılabilir hale getirdim.

Ve son olarak, az önce sözü geçen, kullanılan kabuğu belirtmek için kullandığım ifade dördüncü satırdaki tırnak içindeki ifadedir. Bu ifade genel olarak şöyle yazılabilir: (Bir dosya içine yazdığımızı varsayarak `echo` komutuyla birlikte veriyorum.) `$ echo "#!<programınızı_yorumlayacak_olan_kabuk>"`

[5] BASH'te `~` karakteri kullanıcının ev dizinine işaret eder. Mesela `ls ~` komutu ev dizininizin içeriğini görüntüler. Ya da `cd ~` komutu ev dizininize gitmenizi sağlar. `~/dosya.txt` ise ev dizininizdeki `dosya.txt` dosyasıdır.

Bütün işlemleri komut satırından gösteriyor olmam biraz kafa karıştırıcı olabilir bunun için bir de dosyanın içeriğinde ne olması gerektiğini ayrıca yazalım. Böylece bir metin editöründe görmeniz gereken hakkında daha fazla bilgi sahibi olursunuz. İşte metin editöründe görmeniz gereken iki satırlık programımız:

```
#!/bin/bash
ls -la --color | more
```

Yazdığımız programları nasıl çalıştıracağımızı anladığımıza göre işin zor kısmını bitirdik. Şimdi bu yolla yazacağımız programlarımızda kullanmak üzere buyurgan programlama dillerinde bulunan yapıların BASH altındaki görünüşlerine bakalım.

# 2 Değişkenler

BASH altında bir değişken tanımlamak için kullanılan operatör = operatörüdür. Bir değişken değeri ise değişken isminin önüne $ simgesi getirilerek okunur.

```
$ kedi=sarman
$ echo $kedi
sarman
$
```

Değişkenler oluşturulurken değişkene verilen ad = işaretinin soluna, değeri ise işaretin sağına yazılır. Yani, *sol-değer* değişken adı, *sağ-değer* değişkenin değerini belirleyen ifadedir. Yukarıdaki örnekte, birinci satırda, `kedi` değişkenine `sarman` değerini atadık. İkinci satırda `echo` komutu yardımıyla `kedi` değişkeninin değerinin ekrana basılmasını istedik. Değişkenin önündeki `$` simgesine dikkat edin. Üçüncü satır bilgisayarın bize cavabı. Dördüncü satırda ise bilgisayar bizden yeni komutlar bekliyor.

Bir değişkenin değerini değiştirmek için yeni değeri atamanız yeterli. Değişkenin değerinin boş olması için ise değişkene hiçbir şey atamanız yeter. Yani, sağ-değer atamadan = operatörü kullanılır. Değişkeni tanımsız yapmak, yani tamamıyla ortadan kaldırmak için ise `unset` komutu kullanılabilir.

```
$ kedi=tekir
$ echo $kedi
tekir
$ kedi=
$ echo $kedi

$
```

## 2.1 Değişken türleri

BASH kabuğu altında iki tür değişken vardır. Çevresel ve yerel değişkenler. Bu iki farklı değişkenin farkını anlatmak için çekirdek-kabuk yapısıyla ilgili

yeni şeylerden bahsetmem gerekiyor. Linux sistemlerde çalışan kabuk sayısı birden fazla olabilir. Daha doğrusu çok kullanıcılı bir sistemde her kullanıcı için diğerlerininkinden farklı bir kabuğun çalışması zorunluluktur. Böylece, her kullanıcının kabuğu diğerlerininkinden farklı olduğu için, her kullanıcının kabuğu bağımsız olarak çalışacak ve kabuk yardımıyla bilgisayara gönderilen komutlar birbiriyle karışmayacak veya birbirlerinin çalışmasını engellemeyecektir. Benzer şekilde sizin çalıştırdığınız bir BASH programının da sizin çalışmanızı engellememesi için program sizin kabuğunuza bağlı bir alt kabuk içinde çalıştırılacaktır. Değişkenlerin iki çeşit oluşu işte bu kabuk-alt kabuk ilişkisinden kaynaklanır. Yerel değişkenler sadece içinde çalıştıkları kabuk içinde tanımlıyken çevresel değişkenler bütün alt kabuklarda tanımlıdır. Bizim yukarıda yarattığımız değişkenler yerel değişkenlerdir. Bir değişkenin çevresel olduğunu bildirmek için değişken atamasının önüne `export` getirilir.

```
$ export kedi=tombik
```

Bu değişken artık bütün alt kabuklar tarafından tanınacaktır.

## 2.2 Öntanımlı değişkenler

BASH kabuğu çalışmaya başladığında sizin ona daha önceden belirttiğiniz veya özelleştirdiğiniz yeteneklerini kullanabilmek için bir dizi öntanımlı değişkeni değişik dosyalardan (çoğunu `.bash_profile` dosyasından) okur. Bu öntanımlı değişkenlerin hangileri olduğunu ve değerlerini görmek için `printenv` komutu kullanılabilir. Bunu yaptıysanız yukarıda bahsi geçen `PATH`in de ön tanımlı bir değişken olduğunu fark etmişsinizdir. Öntanımlı değişkenler sizin yaratacağınız değişkenlerle karışmaması için, teamül gereği, tamamı büyük harflerle yazılmışlardır. Sık kullanılan öntanımlı değişkenlerden bahsedelim.

**EDITOR** Öntanımlı olarak kullanmak istediğiniz metin editörü varsa onu bu değişkene ekleyebilirsiniz.

```
$ export EDITOR=/usr/bin/pico
```

Ya da bu satırı (baştaki `$` simgesi olmadan) `~/.bash_profile` dosyasına ekleyerek sistemi her açışınızda aynı değerin etkin olmasını sağlayabilirsiniz.

```
$ echo "export EDITOR=/usr/bin/pico" >> ~/.bash_profile
```

**HISTFILE** Sistemdeyken yazdığınız komutların hepsi sistem tarafında işletilirken hiçbir değişikliğe uğratılmadan (hatalı da olsa) burada belirtilen dosyaya da yazılır. Genelde `~/.bash_history` dosyası tanımlıdır. Merak ederseniz `cat ~/.bash_history` komutuyla bu dosyanın içeriğini dolayısıyla daha önce yazdığınız komutların listesini görebilirsiniz.

**HISTSIZE** `HISTFILE` ile tanımlanan dosyanın saklayacağı satır sayısının üst sınırı bu değişkenle tanımlıdır.

**HOME** Kullanıcının ev dizini bu değişkende tutulur.

**HOSTTYPE** Kullandığınız makinanın kullandığı işlemcinin (dolayısıyla sistemin) mimarisini verir.

**LOGNAME** Sistemdeki kullanıcı adınız.

**PATH** Yazılan komutların veya program isimlerinin hangi dizinlerde aranacağını belirten değişkendir. Kendi yazdığımız programların bulunduğu dizini de buraya ekleyerek bizim programlarımızın da komut satırından çalıştırılmasını sağlayabiliriz. (Nitekim yukarıda yaptık.) `PATH` değişkenine istediğiniz kadar dizin girebilirsiniz, dikkat edilmesi gereken her dizin arasına `:` koymak ve hiç boşluk bırakmamak.

**PS1** Sistemin sizden komut beklediğini belirtmek için satırın başına yazdığı karakterlerdir.

**PS2** Yazdığınız komut bir satırı geçerse, ikinci ve daha sonraki satırlarda satırın başını belirtmek için kullanılacak karakter dizisidir. `PS1` ve `PS2` değişkenlerinin değerleri belirlenirken kullanılan bazı özel karakter dizileri vardır.

| | |
|---|---|
| `\d` | Sistem tarihi |
| `\h` | Sistemin adı |
| `\s` | Kullanılan kabuğun adı |
| `\t` | Sistem saati |
| `\u` | Kullanıcı adı |
| `\w` | Kullanılan dizin |

**SHELL** Sistem açılışında kullanacağınız kabuk. Genelde `/bin/bash` olarak tanımlıdır.

**TERM** Kullanıcının çalıştığı terminal tipi. Örneğin, teletype `tty` ya da sanal terminal `xterm`, hatta körler için Braille konsol `brl`.

### 2.2.1 Kabuk Tanımları

Öntanımlı değişkenlerden söz etmişken BASH kabuğu altında bilmeniz gereken iki özel parametre tanımlı: `noclobber` ve `ignoreof`. Tanımların tamamı küçük harflerle yazılır, etkinleştirilmeleri için `set` komutu `-o`, kaldırılması için `+o` parametresiyle kullanılır. `noclobber` tanımı halihazırdaki bir dosyanın üzerine `>` ile yazılmasını engeller. `ignoreof\verb` ise "Ctrl" ve "D" tuşlarına aynı anda basılmasıyla sistemden çıkılması için etkinleştirilebilir.

```
$ set -o noclobber
$ cat dosya.1 > dosya.2
bash: dosya.2: Cannot clobber existing file
$ set +o noclobber
$ cat dosya.1 > dosya.2
$ set -o ignoreof
$
```

Sondan bir önceki komut yardımıyla `ignoreof` tanımı etkinleştirildikten sonra "Ctrl+D" tuşları terminalden çıkış komutu yerine geçecektir.

# 3 Program dosyalarının içinde açıklama kullanmak

Yazdığımız programların kafa karıştırıcı bölümleri olabilir. Bu tür bölümleri daha sonra programı okurken hatırlamak veya programı değiştirmek isteyen bir başkasının anlamasını kolaylaştırmak için programın içine bazı açıklamalar yazmak mümkündür. Unutmayın, bir program bir kez yazılır, binlerce kez okunur. BASH programlarında # simgesi ile satır sonu arasındaki yazılar komut olarak işletilmez. # ile `#!` arasında fark vardır, bunları karıştırmamaya dikkat edin.

```
$ cat my_prog
#!/bin/bash
# Bu program LinuxMag'ı destekleyenler içindir.
echo "LinuxMag'ı destekliyorum!" # Ekrana istediğimizi yazdık.
$ my_prog
LinuxMag'ı destekliyorum!
$
```

# 4 Etkileşimli programlar

Etkileşimin temeli programın çalıştığı herhangi bir anda kullanıcının belirteceği bilgiler sayesinde programın yönetilmesidir. Bunun için `read` komutu kullanılabilir.

```
$ cat bilgi
#!/bin/bash
# Bu program kullanıcının adını sorar ve bilgi verir.
echo "Adiniz nedir?"
read -r isim
echo "Kullanıcı ismi "
echo "$LOGNAME"
echo " olan "
echo "$isim"
printf "\n"
echo "$EDITOR"
echo " programını tercih etmektedir." # Program bitti.
$ bilgi
Adiniz nedir?
Mehmet Tahir SANDIKKAYA
Kullanıcı ismi mts olan Mehmet Tahir SANDIKKAYA
/usr/bin/pico programını tercih etmektedir.
$
```

# 5 Aritmetik işlemler

BASH'in uzmanlığı insanla ilişki kurmaktır, aritmetik işlemler konusunda çok başarılı sayılmaz. (Bir anlamda sözelci bir kabuktur.) Ancak, nasıl yapılacağı bilinirse çoğu programlama dilindekinden hassas sonuçlar almak da mümkündür. BASH altında bir değişkenin değerinin aritmetik işlemlerde kullanılabilecek şekilde bir sayı (karakter olarak değil, sayı olarak kullanılacak bir değer) olduğunu BASH kabuğuna özel yöntemlerle anlatmak gerekir. Bunun için genelde `typeset` komutu `-i` parametresiyle birlikte kullanılır.

```
$ typeset -i sonuc
$ sonuc=4*7
$ echo $sonuc
28
$
```

Bir değişkene karakter olarak değil de sayı olarak *dört kere yedi* değerini atamak için `let` komutu da kullanılabilir.

```
$ let "sonuc=4*7"
$ echo $sonuc
28
$
```

BASH noktalı sayılarla işlem yapma yeteneğine sahip değildir. Fakat bu eksiği `bc` komutu yardımıyla kapatılabilir. Yapılan aritmetik işlemin çıktısı bir boru (pipe) yardımıyla `bc` programına aktarılır.

```
$ x=10
$ y=3
$ echo "$x/$y" | bc
3.3333333
```

# 6 Karşılaştırma yapmak

BASH kabuğunda karşılaştırmalar `test` komutu yardımıyla yapılır. `test` komutu karakter dizilerini ve sayıları kıyaslayabilir. Değişik kıyaslamalar için değişik parametreler kullanılır. `test` komutu ile yapılan kıyaslama doğru ise `?` değişkeninin değeri `0` olur. Kıyaslama yanlış ise `?` değişkeninin değeri `1` olur.

```
$ test 4 -eq 4
$ echo $?
0
$ test 3 -gt 7
$ echo $?
1
$
```

`test` komutundan sonra gelen kısım `[` ile `]` karakterleri arasına yazılabilir. Bu da `test` komutuyla aynı işi yapar.

```
$ [ 4 -eq 4 ]
$ echo$?
0
$ [ 5 -lt 2 ]
$ echo $?
1
$
```

`test` komutuyla kullanılabilen, dolayısıyla köşeli ayraç içinde de kullanılabilen sık kullanılan parametrelerin listesini aşağıda veriyorum.

| Dosyaların karşılaştırılması | |
|---|---|
| `-b` | İlgili dosya bir blok aygıtı |
| `-c` | İlgili dosya bir karakter aygıtı |
| `-f` | İlgili dosya olağan bir dosya |
| `-h` | İlgili dosya bir başka dosyaya sembolik bağlı |
| `-r` | İlgili dosya okunabilir |
| `-s` | İlgili dosya var ve boş değil |
| `-w` | İlgili dosya değiştirilebilir |
| `-x` | İlgili dosya çalıştırılabilir |
| **Karakter dizilerinin karşılaştırılması** | |
| `!=` | Aynı değil |
| `=` | Aynı |
| `-n` | Boş dizi değil |
| `-z` | Boş dizi |
| **Mantıksal karşılaştırmalar** | |
| `!` | Değil |
| `-a` | VE |
| `-o` | VEYA |
| **Sayıların karşılaştırılması** | |
| `-eq` | Eşit |
| `-ge` | Büyük-eşit |
| `-gt` | Büyük |
| `-le` | Küçük-eşit |
| `-lt` | Küçük |

# 7 Kontrolü ele almanın vakti geldi

Bir BASH programının bizim girdiğimiz veya kendi edindiği bir veriye bakarak, ayrıca farklı iki veya daha çok veriyi kıyaslayarak farklı durumlarda farklı komut öbeklerini çalıştırması mümkündür. `if-else` kalıbı bu konuda en büyük

yardımcımız olacak. `if` komutundan sonra gelen karşılaştırma sonucu doğru olursa `then` komutundan sonraki komut öbeği işletilir. Karşılaştırmanın sonucu yanlış olursa (varsa) `else` komutundan sonraki komut öbeği işletilir. `if` komutu `fi` komutuyla son bulur.

```
$ cat kare_kup
#!/bin/bash
# Bu program 10'dan küçük sayıların küpünü,
# 10'dan büyük sayıların karesini alır.
echo "bir sayi giriniz."
read -r sayi
if [ "$sayi" -le 10 ] # if komutu başladı.
then # sayi 10'dan küçükse bu kısım çalıştırılacak.
  sayi=$((sayi*sayi*sayi))
# BASH'te çift parantezler arasında işlem yapılabilir.
  echo $sayi
else # sayi 10'dan büyükse bu kısım çalıştırılacak.
  sayi=$((sayi*sayi))
  echo $sayi
fi # if komutu bitti.
$ kare_kup
bir sayi giriniz.
5
125
$ kare_kup
bir sayi giriniz.
12
144
$
```

Bir başka kontrol yöntemi de `case` komutunu kullanmaktır. Bu komut yardımıyla elimizdeki değişkenin her bir değişik durumu için değişik komut öbekleri işletilebilir. `case` komutu `in` komutuyla birlikte kullanılır. Her `case` komutu `esac` komutuyla son bulur.

```
$ cat asal
#!/bin/bash
# Bu program 1'den 5'e kadar sayılar hakkında bilgi verir.
echo "bir sayi girin."
read -r sayi
case $sayi in
# case ve in komutları bir arada. değişken ikisinin arasında.
1) # genel kullanım "birinci_secenek)" şeklindedir.
echo "1 asal değildir."
# birinci seçeneğin oluşması durumunda çalıştırılacak
# komut öbeği buraya gelir.
;;
```

```
# seçenekler arasına çift noktalı virgül konur.
2)
echo "2 asaldır."
;;
3)
echo "3 asaldır."
;;
4)
echo "4 asal değildir."
;;
5)
echo "5 asaldır."
;;
*)
# yukarıdakiler dışında herhangi bir seçeneği belirtir.
# Seçenekler yazılırken ?,*,[,] gibi joker karakterlere
# izin vardır. Mesela; a*b seçeneği a ile başlayıp b
# ile biten bütün seçenekleri kapsar.
# Kullanıcı birle beş arası değer girmemişse
# kullanıcıyı uyarmak için bunu kullanalım:
echo "Bu program 1'le 5 arasındaki değerler için çalışır."
esac # case yapısının sonu
$ asal
bir sayi girin.
1
1 asal değildir.
$ asal
bir sayi girin.
5
5 asaldır.
```

# 8 Monoton programlar

BASH altında bir komutu veya komut grubunu birkaç kez çalıştırmamız gerekirse döngüleri kullanırız. İki çeşit döngü yapısı vardır: `while-do` ve `for-do`. `while-do` yapısında bir kıyaslama yapılır ve kıyaslama doğru olduğu sürece işleme devam edilir, `for-do` yapısında ise işlem, verilen bir sayı kere tekrarlanır. Her iki yöntemi de kullanarak 1'den 10'a kadar sayıları sırayla ekrana yazan programlar oluşturalım.

```
$ cat sayim1
#!/bin/bash
# bu program while-do yapısını kullanarak 1'den 10'a kadar
# sayıları ekrana yazacak.
sayi=0
```

```
while [ $sayi -le 10 ]
  # while komutundan hemen sonra kıyaslama getirilir.
do
  # çalıştırılacak olan komut öbeği do ile done arasına yazılır.
  sayi=$((sayi+1)) # sayi değişkeninin değeri 1 artırılıyor.
  echo $sayi # sayi değişkeninin değeri her seferinde yazılıyor.
  # sayi değişkeninin değeri 10'dan küçükse program komut
  # öbeğinin başına döner.
done
# while-do yapısı bitti.
$ cat sayim2
#!/bin/bash
# bu program for-do yapısını kullanarak 1'den 10'a kadar
# sayıları ekrana yazacak.
for sayi in 1 2 3 4 5 6 7 8 9 10
  # for komutu in komutuyla birlikte kullanılır.
  # in komutundan sonra değişkenin alacağı değerler
  # sırayla yazılır. Değişkenin alacağı değerler
  # yazılırken *,? gibi joker karakterler de kullanılabilir.
  # En alışıldık yollardan biri aritmetik hesap çalıştırarak
  # bir aralık oluşturmaktır:
  # for ((i=1;i<=5;i++)); do
  # Burada 1'den 5'e kadar aralık tanımlanıp kullanılmıştır.
  # do, noktalı virgülle ayrılmış ikinci komut olarak aynı satıra
  # alınabilir.
do # komut öbeği başlıyor.
  echo $sayi # sayi değişkeninin değeri akrana yazılıyor.
  # sırada değişkenin alacağı bir değer kaldıysa komut öbeğinin
  # başına dönülür.
done
# komut öbeği bitti.
```

# 9 Parametreler kolaylık sağlar

Komut satırından yazdığınız bir BASH programına parametreler gönderebilirsiniz. Bu parametreler sayesinde tek bir satırda program içinde kullanacağınız bütün değişkenlerin değerini programınıza bildirebilirsiniz. Böylece yazdığınız programların kullanıcılardan girdi bekleme zorunluluğu ortadan kalkar. Komut satırında, programınızın isminin ardından yazacağınız bütün parametreler çevresel değişkenlerde saklanır. `$#` değişkeninde komut satırından yazdığınız parametre sayısı tutulur. `$0` değişkeni ise yazdığınız komutun kendisini içerir. `$1` değişkeni komut satırından girilen ilk parametreyi, `$2` ise ikincisini gösterir. Parametre sayısı pratikte sınırsızca artırılabilir.

```
$ cat param
```

```
#!/bin/bash
echo "Yazdığınız komut "
echo "$0"
echo " ve "
echo "$#"
echo " parametreye sahip."
printf "\n"
echo "İlk iki parametre sırayla : "
echo "$1"
echo ", "
echo "$2"
$ param --parbir ++parti parüç
Yazdığınız komut param ve 3 parametreye sahip.
İlk iki parametre sırayla : --parbir, ++parti
$
```

# 10 Fonksiyon tanımlama

BASH programı yazılırken bazı komut öbekleri bir araya getirilerek istendiği zaman çalıştırılmaları sağlanabilir. Bir fonksiyonu tanımlamak için isminin önüne `function` sonuna `()` getirilir. Fonksiyonun kapsadığı komut öbeği `{` ile `}` karakterleri arasına alınır. Fonksiyon içindeki son komut noktalı virgülle bitmek zorundadır. Burada ayrıntılarını göstermesek de ilk fonksiyon parametreyle kullanılmış ve sonuç alınarak bir başka fonksiyona parametre olarak aktarılmıştır. İkinci fonksiyonun çıktısı da yakalanıp ekranda gösterilir. BASH fonksiyonları kullanılırken fonksiyonun ürettiği değeri almanın en uygun yolu fonksiyon biterken üretilen değeri `echo` ile ekrana yazdırmak; fonksiyonu çağırırken bu değeri `$()` içinde kullanarak ekrana yazdırılacak değeri yakalamaktır.

```
$ cat yari_kare
#!/bin/bash
# Bu program fonksiyon kullanarak girilen parametrenin karesini
# alır. Daha sonra bir başka fonksiyonla ikiye böler.
function kare_alma_fonksiyonu()
{
  ((sonuc=$1*$1))
  echo "$sonuc";
}
function yarilama_fonksiyonu()
{
  ((sonuc=$1/2));
  echo "$sonuc";
}
kare=$(kare_alma_fonksiyonu $1)
yari=$(yarilama_fonksiyonu "$kare")
```

```
echo "$yari"
$ yari_kare 4
8
$
```

## Sonsöz (Konuyla ilgili değildir.)

Bu yazıda elimden geldiği kadarıyla BASH programlamayı anlatmaya çalıştım. Tahmin ediyorum ki birçok eksik ve/veya hata bulunabilir. Fakat, eksik noktalar bıraktıysam da konu özelinde eksiklik olmaması için özel bir gayret gösterdiğimi de belirtmek isterim. Bu yazıda düzeltilecek/yazıya eklenecek bir şeyler olduğunu düşünüyorsanız lütfen bana haber verin.