GELİŞMİŞ DÜZEY BASH BETİK PROGRAMI

Yazar: Mendel Cooper©

Telif Hakları

KISIM 1 GİRİŞ

Kabuk bir komut yorumlayıcısıdır. Sadece işletim sistemi çekirdeğiyle kullanıcı arasındaki katmana yalıtım sağlamakla kalmaz, oldukça güçlü bir programlama dilidir. *Komut dosyası (script)* adlı bir kabuk programı, sistem çağrıları, araçları, yardımcı programları ve derlenmiş ikili programları bir arada "yapıştırarak" uygulama oluşturmaya yardımcı kolay kullanımlı bir araçtır. Hemen hemen tüm UNIX komutları, yardımcı programlar ve araçlar bir kabuk betiği tarafından çağrılabilir. Bu yeterli değilse, testler ve döngü yapıları gibi iç kabuk komutları, komut dosyalarına ek güç ve esneklik sağlar. Kabuk betikleri, yönetimsel sistem görevleri ve diğer rutin tekrarlayan işleri yerine getirirken tam gelişmiş bir yapısal programlama dilinin getirdiği gelişmelere gereksinim duymaz.

BÖLÜM 1 NEDEN KABUK PROGRAMLAMA?

Sistem yönetimi alanında uzman olmak için komut dosyası yazmak zorunda olunmasa da kabuk betikleri konusunda geçerli bilgi sahibi olunması şarttır. Bir Linux makine ilk açılırken sistem yapılandırmasını ve kurulum hizmetlerini geri yüklemek için `/etc/rc.d` dizini içindeki kabuk betiklerini çalıştırır. Bu başlatma komut dosyalarını ayrıntılı olarak anlamak, bir sistemin davranışlarını analiz etmek ve muhtemelen değiştirmek için esastır.

Kabuk betiklerini yazmayı öğrenmek zor değildir, çünkü kabuğa-özel operatörler ve seçeneklerin [1] sayısı azdır ve betikler küçük küçük parçacıklar halinde yazılabilir. Sözdizimi basit ve anlaşılırdır, komut satırında yardımcı programların yürütülmesine ve zincirlemesine benzemektedir ve öğrenilmesi gereken birkaç "kural" yeterlidir. En kısa kodlar genellikle ilk seferde doğru olarak çalışır ve daha uzun olanların hata ayıklanması basittir.

Bir kabuk betiği karmaşık bir uygulamanın prototipini çıkarmanın "hızlı ve kirli" bir yöntemidir. Bir kabuk betiğinin işlevi az ve yavaş bile olsa, proje geliştirme sürecine şöyle bir katkıda bulunabilir: Uygulama yapısı, ortaya çıkabilecek yapısal bozukluklara karşın C, C++, Java ve Perl gibi üst-düzey programlamaya gidilmeden test edilir ve bozukluklar için öngörülen prosedür (gerekli değişiklik veya düzeltmelerin yapılması) sağlanır.

Kabuk programlama sayesinde Unix geleneğinin bir parçası olan karmaşık projelerin daha basit alt görevlere bölünmesi mümkündür. Bileşen ve yardımcı programların zincirleme metoduyla daha üst düzeyde veri işlemesi ve programlar arası bilgi aktarımı sağlanabilir. Birçok insan en uygun bir problem çözme aracı olarak Perl gibi, yeni nesil ve güçlü özellikleri bünyesinde bulunduran programlama dillerini, daha iyi ve estetik bulmaktadırlar. Bu gibi diller her insan için her şeyi ancak düşünme süreçlerini değiştirmeye zorlama pahasına sunarlar.

Kabuk programlama hangi durumlarda kullanılmamalıdır?

- Yoğun miktarda kaynak isteyen görevler, hız faktörünün önemli olduğu sıralama, karma, vb.
- Özellikle ağır kayan nokta işlemleri ve matematiksel prosedürler.
- Görev kritik uygulamalar
- Güvenlik uygulamaları
- Birbirine bağımlı alt bileşenlerden oluşan programlar
- Kapsamlı dosya işlemlerinin gerekli olduğu durumlar (Bash seri dosya erişimi sağlar, ancak satırsal okuduğu için verimsiz kalır.)
- Çok boyutlu diziler
- Bağlantılı listeler veya ağaç gibi veri yapıları
- Grafik veya GUI oluşturmak veya işlemek için
- Sistem donanımına doğrudan erişim ihtiyacı
- Bağlantı noktası ya da soket Girdi/Çıktı gerekliyse
- Eski bir kodu kitaplığı veya arabirimi ile kullanmanız gerekliyse
- Özel, kapalı kaynak uygulamaları (kabuk betikleri mutlaka açık kaynak kodludur.)

Yukarıdakilerden herhangi biri sizin için geçerliyse, Perl, Tcl, Python, veya C, C++ veya Java gibi daha güçlü bir betik dilini tercih etmelisiniz. O zaman bile, bir kabuk gibi prototip uygulama hala yararlı bir gelişme adımı olabilir.

Bash "Bourne Again Shell" e karşılık gelen kısaltmanın adıdır. Bash, UNIX komut dosyası için "de facto standart" haline gelmiştir. Bash bazı özellikleri Korn kabuktan elde etmiştir [2].

Elinizdeki bu materyal, kabuk ve betik programlamayı size öğreticidir. Kabuk özelliklerini göstermek için örneklere dayanır. Mümkün olduğu kadar, örnek betikleri test edilmiştir ve bazıları gerçek yaşamda yararlı olabilir. Okuyucu (bir şey-veya-diğer.sh) kaynak arşivinden gerçek örnekler kullanmalıdır. [3] Sonra ne olacağını görmek için onları çalıştırın. Kaynak arşivin mevcut olmadığı durumlarda, HTML, PDF veya metin oluşturan sürümlerinden kes-ve-yapıştır yapın.

Unutmayınız ki, bazı betiklerin özellikleri açıklanmadan size tanıtılmıştır ve bu nedenle okuyucunun bazı bölümleri atlaması gerekiyor. Aksi belirtilmedikçe, kitabın yazarı bir çok örnek betikler yazmıştır.

Notlar:

[1] İç kabuk özellikleri taşıyan kabuk iç yerleşikleri (built-in) olarak adlandırılır.

[2] `ksh88` özelliklerinin çoğu, güncelleştirilmiş `ksh93` özelliklerinden birkaçı `Bash` adı altında birleştirilmiştir.

[3] `chmod u+rx` komutu adı belirtilen dosyaya okuma ve çalıştırma izni verir.

BÖLÜM 2 İLK SATIRI #! ile BAŞLAYAN KOMUT DOSYALARI

En basit haliyle, bir komut dosyası, bir dosyada depolanan sistem komutlarının listesinden başka bir şey değildir. Dosyaya komutları yazdığımız için en azından komutu oluşturan dizeleri her seferinde yeniden yazma zahmetinden kurtuluruz.

ÖRNEK 2.1 TEMİZLEME: **/var/log** GÜNLÜK DOSYALARI TEMİZLEMEK İÇİN BİR KOMUT DOSYASI

```
# cleanup
# Run as root, of course.

cd /var/log
cat /dev/null > messages
messagescat /dev/null > wtmp
echo "Logs cleaned up."
```

Burada olağandışı bir şey yok, sadece konsoldan (`xterm`) birer birer çağrılacak komut dizisi var. Bir komut dosyasının komutlarını bir betik dosyasının içine yerleştirsek, defalarca yeniden yazmak gereksizdir. Komut belirli bir uygulama için, değiştirilmiş özelleştirilmiş veya genelleştirilmiş olabilir.

ÖRNEK 2.2 TEMİZLEME: YUKARIDAKİ PROGRAM İÇİN GELİŞTİRİLMİŞ YAYGIN BİR SÜRÜM

```
#!/bin/bash# cleanup, version 2
# Run as root, of course.

LOG_DIR=/var/log
ROOT_UID=0          # Only users with $UID 0 have root privileges.
LINES=50            # Default number of lines saved.
E_XCD=66            # Can't change directory?
E_NOTROOT=67        # Non-root exit error.

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "Must be root to run this script."
    exit $E_NOTROOT
fi

if [ -n "$1" ]
# Test if command line argument present (non-empty).then
    lines=$1
else lines=$LINES   # Default, if not specified on command line.
fi

# Stephane Chazelas suggests the following,
```

```
#+ as a better way of checking command line arguments,
#+ but this is still a bit advanced for this stage of the tutorial.
#

#E_WRONGARGS=65 #Non-numerical argument (bad arg format)
#
#case "$1" in
#""      ) lines=50;;
#*[!0-9]*) echo "Usage: ' basename $0' file-to-cleanup"; exit
$E_WRONGARGS;;
#*       ) lines=$1;;
#esac
#
#* Skip ahead to "Loops" chapter to decipher all this.

cd $LOG_DIR

if [' pwd' != "$LOG_DIR"] # or if ["$PWD" != "$LOG_DIR"]
                          # NOT in /var/log?

then
    echo "Can't change to $LOG_DIR."
    exit $E_XCD
fi    # Double check if in right directory, before messing with log
file.

# far more efficient is
#
# cd/var/log || {
# echo "Cannot change to necessary directory." >#2
# exit $E_XCD
#}

tail -$lines messages > mesg.temp    # Saves last message of log file
mv mesg.tmp messages

# cat /dev/null > messages
#* No longer needed, as the above method is safer.

cat /dev/null > wtmp # ' : > wtmp' and ' > wtmp' have the same effect.
echo "Logs cleaned up."

exit 0

# A zero return value from the script upon exit
#+ indicates success to the shell.
```

Tüm sistem günlüğünü silip temizlemek istemeyebilirsiniz, bu ilk betiğin bir başka türü mesaj günlüğünün son bölümünde tutulur. Böylece, önceden yazılmış komut dosyalarını etkinlik artırımı amacıyla iyileştirebilirsiniz.

Komut dosyasının başındaki iki harflik (#!) karakter dizisi bu dosya ile belirtilen komutların belirtilen komut yorumlayıcısına verileceğini sisteme söyler. #! aslında bir-iki baytlık özel işaretleyici belirtir: bir dosya türü veya bu durumda bir yürütülebilir kabuk (bu konu hakkında daha fazlasını görmek için **man magic** yazınız.) Hemen ardından bir yol adı yazılır. Bu komut dosyası komutlarını yorumlayan programın bulunduğu yoldur.

Bu program bir kabuk, bir programlama dili, veya bir yardımcı program olabilir. Bu komut yorumlayıcısı, daha sonra, betiğin en üstteki (birinci) satırından itibaren açıklamaları atlayarak betik komutlarını çalıştırır [2].

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

İlk satır olarak, yukarıdaki başlık satırlarından her birini içeren komut dosyaları, farklı birer komut yorumlayıcısını çağırır. `/bin/sh` Linux sisteminin varsayılan kabuğu `bash`'i ve diğerleri de karşılık gelen diğer başkalarını çağırır [3]. UNIX işletim sisteminin diğer başka ticari sürümlerinde Bourne Shell'in varsayılan olarak `#!/Bin/sh` satırıyla birlikte kullanılması, birkaç Bash'e özgü özelliklerin (komut POSIX [4] `sh` standardına uygun olacaktır) feda edilmesine rağmen taşınılabilirliği artırır.

`#!` satırında verilen yolun doğru olmasına dikkat edilmelidir, aksi takdirde hata mesajıyla karşılaşılır. Komut dosyası, yalnızca genel sistem komutları veya hiçbir iç kabuk direktifini kullanan bir komut içermiyorsa #! ihmal edilebilir.

Yukarıda değişken atama satırı LINES=50 kabuğa-özgü bir yapı olduğu için Örnek 2'nin, ilk satırında #! yazılması gereklidir. Unutmayınız ki, `#!/bin/sh` herhangi bir Linux makinede `/bin/bash` için varsayılan kabuktur.

Bu dokümanda benimsenen öğretici yaklaşım "modüler" olarak nitelendirilmelidir, bir komut dosyası oluşturmayı size teşvik ediyoruz. Demirbaş türünden kod parçacıkları gelecekte yararlı olabilir. Sonunda oldukça geniş çok iyi bir rutin kütüphanesi oluşturabilirsiniz. Örnek olarak, aşağıdaki komut dosyası parametrelerin doğru sayıda çağrılıp çağrılmadığını test eder.

```
if [ $# -ne Number_of_expected_args ]
then
  echo "Usage: `basename $0` whatever"
  exit $WRONG_ARGS
fi
```

Notlar

[1] UNIX'in bazı versiyonları (4.2 BSD tabanlı olanlar) `!`'dan sonra boşluk gerektiren dört baytlık bir sihirli sayı alırlar, `#! /bin/sh`.

[2] Kabuk satır komut yorumlayıcısının (**sh** veya **bash**) gördüğü ilk şey #! satırı olacaktır. Bu satır # ile başladığı için, komut yorumlayıcısı nihayet komut dosyasını yürüttüğü zaman doğal olarak bir yorum olarak yorumlanacaktır. Bu satır, komut yorumlayıcısını arayarak zaten amacına hizmet etmiştir.

[3] Bu, bazı sevimli hilelere olanak tanır.

KISIM 2 TEMEL KONULAR

BÖLÜM 3 ÇIKIŞ VE ÇIKIŞ DEĞERİ

Çıkış komutu, bir C programını bir komutla sonlandırır. Ayrıca komutun ana gövdesi için kullanılabilir bir değer de döndürür. Her komut, istenirse bir çıkış durumunu bir dönüş durumu olarak getirebilir. Başarısız bir hata kodu sıfır olmayan bir değer ve başarılı bir komut komut sıfır döndürür. UNIX komutları çoğunlukla program ve yardımcı programları başarılı bir şekilde bitirirse sıfır çıkış kodunu getirir. Aynı şekilde, bir komut ve betik kendi içindeki fonksiyonların çıkış durumunu getirir. İşlev ya da komut dosyasında yürütülen son komut çıkış durumunu belirler. Komut dosyasından çıkış için gerekli nnn komutu kabuğun nnn çıkış durumunu sağlar. (nnn- 0 - 255 aralığında bir ondalık sayıdır).

Komut çıkış durumu komut dosyasının son komutunun çıkış değeridir. Bir komut dosyası hiçbir parametresi olmayan bir çıkış ile sona erdiğinde, komut çıkış durumu betikte son çalıştırılan komutun çıkış durumudur. Bir işlevin sonlandırılmasından önce, $? işlevi ile gerçekleştirilen son komut son yürütülen komutun çıkış değerini okur. Bash kabuğunda işlevlerin dönüş değerleri bu yolla verilir. Komut dosyası sona erdiğinde, $? komut satırından geleneksel olarak başarısızsa, 1-255 aralığında bir tamsayı veya 0, başarılıysa, döndürür.

ÖRNEK 3.1 ÇIKIŞ VE ÇIKIŞ DURUMU

```
#!/bin/bash
echo hello

echo $?     # Returned because command executed successfully.
lskdf       # Unrecognized command

echo $?     # Non-zero exit status returned because command failed to
execute

echo

exit 113    # Will return 113 to shell.
            # To verify this, type "echo $?" after script terminates.

# By convention, an ' exit 0' indicates success.
#+ while a non-zero exit value means an error or anomalous condition.


$? is especially useful for testing the result of a command in a script
```

(bkz.Örnek 12.27 ve Örnek 12.23)

Mantıksal "değil" nitelendiricisi bir test veya komut sonucunun ters değerini getirir ve bu onun çıkış durumunu etkiler.

ÖRNEK 3.2 ! KULLANARAK TERS KOŞUL İŞLEMİ

```
true                                          # the "true" built-in.
echo "exit status of \"true \" = $?"          #0

!true

echo "exit status of \"! true \" = $?"        #1


# Note that the "!" needs a space.
#    !true leads to a "command not found" error

# Thanks S.C.
```

Bazı çıkış değerleri farklı kodlarla ifade edilir, ve bunların betiğe özgü anlamları vardır, kullanıcı betik kodu değiştiremez.

BÖLÜM 4 ÖZEL KARAKTERLER

Betikleri açıklamada kullanılan en geçerli seçenek # özel karakterini kod içinde kullanmaktır.

#! dizisi özel olduğundan # ile başlayan satırlara açıklama yazılır.

```
# Bu bir açıklama satırıdır.
```

Açıklamalar komut sonunda da yer alabilir.

```
echo "A comment will follow."   # Comment here
```

Açıklama satırında boşluk karakterleri sorunsuz kullanılır.

```
# A tab preceeds this comment.
```

Açıklama yazmak zorunlu değildir, ancak programcılar tavsiye eder.

Bir satırda açıklama yaptıktan sonra kod yazmanıza izin verilmez. Sonraki komut için yeni bir satıra geçilmelidir.

Açıklama içinde # karakterini ancak esc ile kullanabilirsiniz. Bu şekliyle açıklamaya devam edebilirsiniz.

Aynı şekilde, # karakteri belirli parametre değiştirme yapıları ve sayısal sabit ifadeler için de kullanılabilir.

```
echo "The # character does not begin a comment here."
echo ' The # character does not begin a comment here'
echo The \# character does not begin a comment here.
echo The # here begins a comment here.

echo ${PATH#*:}           # Parameter substitution not a comment.
echo $((2#101011))        # Base conversion, not a comment.

# Thanks S.C.
```

karakterinden çıkış için '' ' \ özel karakterlerinden biri kullanılır.

;

Komut ayırıcısı [noktalı virgül] aynı satırda iki veya daha çok komut koymanıza izin verir.

```
echo hello ; echo there
```

Not: Noktalı virgülün ; bazen çıkış karakteriyle ;; gösterilmesi gerekir.

;; (Çift noktalı virgül)

case cümleleri için terminator özel karakteri.

```
case "variable" in

abc) echo "$variable=abc" ;;
xyz) echo "$variable=xyz" ;;

esac
```

. (Nokta)

Nokta komutu kaynak için eşdeğer bir bash yerleşiğidir (bkz. Örnek: 11-16).

Bir dosya adının parçası olarak da . (nokta) kullanılır. Dosya adları ile çalışırken, gizli dosya adlarının nokta ile başladığını bilmeniz gerekir. `ls` gizli dosyaları göstermez.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
  -rw-r--r-- 1  bozo      4034 Jul 18 22:04 data1.addressbook
  -rw-r--r--  1  bozo      4602 May 25 13:58 data1.addressbook.bak
  -rw-r--r--  1  bozo      877  Dec 17 2000  employment.addressbook

bash$ ls -al
total  14
  drwxrwxr-x 2     bozo bozo     1024 Aug 29 20:54 ./
  drwx------ 52    bozo bozo     3072 Aug 29 20:51 ../
  -rw-r--r--1  4034  bozo bozo     4034 Jul 18 22:04 data1.addressbook
  -rw-r--r--  1     bozo bozo     4602 May 25 13:58 data1.addressbook.bak
  -rw-r--r--  1     bozo bozo     877 Dec 17 2000   employment.addressbook
  -rw-rw-r-- 1     bozo bozo     0   Aug 29 20:54  .hidden-file
```

Dizin adlarında geçerli çalışma dizini tek bir noktayla temsil edilir; iki nokta üst dizini gösterir.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo
```

Nokta, genellikle bir dosyanın taşıma komut hedefi (dizin) olarak görünür.

```
bash$ cp /home/bozo/current_work/junk/* .
```

.

(Nokta) karakter eşleme.

Bir düzenli ifadenin parçası olarak karakter eşlerken, "nokta" uzunluğu sadece bir (bir adet) karakterle eşleşir.

“

Çift tırnak.

"STRING" tipi metin dizelerini yazmakta kullanılan özel karakter. (bkz. Bölüm 6)

‘

Tek tırnak.

Çift tırnağa göre daha güçlüdür. (bkz. Bölüm 6)

,

Virgül operatörü aritmetik işlemler dizisini bir araya bağlar. Tümü değerlendirilir, ancak sonuncusu döndürülür.

```
let  “t2 = ((a=9, 15/3))” # Set “a” and calculate “t2”.
```

\

Çıkış [ters bölü çizgisi]. \X karakterinin çıkış çizgisidir. X'i açıklama etkisi verir. Aynı etkiyi 'X' de verir. Çift tırnak ve tek tırnağı açıklamak için kullanılabilir. (Çıkış karakterlerin daha kapsamlı açıklaması için bkz. Bölüm-6.)

/ Dosya yol ayırıcısı. [eğik çizgi]

`/home/bozo/projects/Makefile` ifadesinde olduğu gibi dosya adının bileşenlerini ayırır.

Bu aynı zamanda bölme aritmetik işlemcisidir.

‘

komut değiştirme. [vazgeçmek]

'Komut' bir değişkene atama yapmak için komutun çıkışını yapar.

:

: boş komut. [iki nokta üstüste]

Bu NOP'un kabuk eşdeğeridir. (no-op, hiçbir şey yapma anlamına gelen no-operation ifadesinin benzetimidir.) `doğru` yerleşiğinin kabuğa karşılık gelen eşanlamlısı olarak düşününüz.

İki nokta üst üste bir bash yerleşiğidir ve çıkış kodu `doğru`'ya denk düşen 0'dır.

```
:
echo $?      # 0
```

Sonsuz döngü :

```
while :
do
   operation-1
   operation-2
   …
   operation-n
done

# same as :
# while true
# do
# …
# done
```

`if/then` test ifadesinin yer tutucusu :

```
if condition

then : # Do nothing and branch ahead

else
   take-some-action
fi
```

Nerede bir işlem bekleniyorsa, oraya bir tutucu sağlayınız (bkz. Örnek 8-2 ve varsayılan parametreler)

```
: ${username='whoami'}

# ${username='whoami'} without the leading :
#                      gives an error unless "username" is a command or builtin …
```

Komutun beklendiği yere yer tutucu sağlayın. Buraya bir doküman yerleştirin. (bkz. Örnek 17-9)

Değişken dizilerini değerlendirmek için parametre değiştirme uygun bir tekniktir. (Örnek 9-12)

```
: ${HOSTNME?} ${USER?} ${MAIL?}
#Prints error message if one or more of essential environmental variables not set.
```

Değişken açma/altdize yenileme :

Yeniden yönlendirme operatörü > ile birlikte kullanıldığında, değiştirme izni olmaksızın bir dosyanın uzunluğu sıfıra iner. Dosya önceden yok ise, oluşturur.

```
: > data.xxx    # File "data.xxx" now empty.
                # Same effect as cat /dev/null >data.xxx
                # However, this does not fork a new process, since ":" is a built-in.
```

(bkz. Örnek 12.11)

Yeniden yönlendirme operatörü >> dosya erişim/değişiklik zamanını günceller (: >> yeni_dosya). Dosya mevcut değilse, yenisini gösterir. `touch` komutu ile aynı etkiyi yaratır. Ancak bu, sadece kullanıcı dosyaları için geçerlidir, aktarımlı, sembolik ve belirli özel dosyalar için (genellikle işletim sistemi ile ilgili olan) geçerli değildir.

Bu operatör, tavsiye edilmediği halde, bir açıklama satırını da başlatabilir. # açıklama satırı her karakteri kabul eder, ancak, : ile başlayıp . ile bitenlerde durum farklıdır.

```
: (if [$ x-eq 3])  hata üreten bir yorumdur.
```

":" şifre ve kullanıcı adlarını getiren passwd dosyalarında ve $PATH sistem yol gösterici değişkeninde veri alan ayıracı olarak da kullanılır.

```
bash$ echo $PATH /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

!

Bir test veya çıkış durumunun yönünün zıttını verir (bkz. Örnek 3-2.) Fonksiyonlar için çıkış değerini tersine döndürmekte kullanılır. Yapısal cümlelerin anlamlarını da tersine çevirir. Örneğin, = işlecinin ters değerini veren yapı != olur. ! operatörü `Bash` anahtar sözcüklerinden biridir.

Programcılık açısından bakıldığında, ! değişken değerlerinin değerini değiştirir. Bu ifade indirekt referans kavramına neden olmuştur.

Diğer bir açıdan bakıldığında, komut satırından ! girildiğinde Bash, tarih mekanizmasını çağırır (bkz. Ek-F) Unutmayınız ki, bir betiğin içinden tarih mekanizması devreden çıkmıştır.

*

Asterisk joker karakteri olan * dosya adını geneller; kurallı ifadelerde uzunluğu 0 ve daha büyük olan karakterleri temsil eder.

*

Aritmetik operatör olan * çarpma işlemi için kullanılır.

**

Çift yıldız operatörü, üslü kuvvet alımında kullanılır.

?

Test operatörü olan ? belirli işlemler için test koşulu ve cümlelerini oluşturmada ve ayırmada kullanılır.

Çift parantez yapısı içinde ? C-stilindeki üçlü operatör olarak görev yapar. (bkz. Örnek 9-25).

Parametre değiştirme ifadesinde ? bir değişkenin değerinin atanıp atanmadığını test eder.

?

Soru işareti joker karakteri. Dosya adlarının uzunluğu, sadece ve sadece bir olan, karakter genellemelerini ifade eder. Bu genelleme kurallı ifadelerde de geçerlidir.

$

Değişken değeri.

```
var1=5
var2=23skidoo
echo $var1  # 5
echo $var2  # 23skidoo
```

Bir değişken adının önünde $ öneki varsa, değişkenin değeri anlaşılmalıdır.

$ satır sonu.

Bir kurallı ifadede, $ metin satırının sonunu ifade eder.

${}

Parametre değişimi.

$*, $@

Konumsal parametreler.

$? çıkış durumu değişkeni

Bir komut, bir fonksiyon veya betiğin çıkış değerini tutan çıkış durumu değişkeni $? ile gösterilir.

$$ işlem tanımlayıcısı değişkeni

Betiğe ait olan işlem tanımlayıcısı, işletim sisteminde $$ değişkeninde tutulur.

()

Komut grubu.

```
(a=hello; echo $a)
```

Parantez içindeki komut listesi altkabuğu başlatır. Altkabukta parantez içindeki değişkenler, betiğin geri kalanında görünür değildir. Betiği çalıştıran ana süreç, alt kabukta oluşturulan değişkenleri okuyamaz.

```
a=123 (a=321; )
echo "a=$a"       # a=123
# "a" within parentheses acts like a local variable.
```

Dizilimin ilk değerlerini başlatma

```
array=(element1 element2 element3)
```

```
{xxx,yyy,zzz,...}
```

Kıvrık parantezin açılımı

```
grep Linux file*.{txt,htm*}
# Finds all instances of the word "Linux"
# in the files "fileA.txt", "file2.txt", "fileR.html", "file-87.html", etc.
```

Bir komut dosyasında, kıvrık parantez içinde virgülle ayrılmış dosya belirtimleri yer alabilir [1]. Dosya adı açılımları parantez içindeki dosya belirtimleri için uygulanır. Parantez içine boşluk yazılmamalıdır, boşluk karakterleri tırnak içinde veya değiştirme özel karakteriyle yazılır.

```
echo {file1,file2}\ :{\A," B", '  C'}

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

{}

Kod bloğu. [Kıvrık parantez] Ayrıca bir "açık grup" olarak da anılan bu yapı anonim fonksiyon oluşturmada kullanılır. Ancak fonksiyonun tersine, kod bloğu içindeki değişkenler, tüm betik boyunca görünürdür.

```
bash$ { local a; a=123; }
bash: local: can only be used in a function
```

```
a=123
{ a=321; }
echo "a= $a"      # a = 321 (value inside code block)

# Thanks, S.C.
```

Kıvrık parantez içindeki kod bloğu G/Ç (Girdi/Çıktı) yönlendirmesi yapılabilir.

ÖRNEK 4.2 KOD BLOKLARI VE G/Ç YÖNLENDİRMESİ

```
#!/bin/bash
# Reading lines in /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "First line in $File is:"
echo "$line1"
echo "$Second line in $File is:"
echo "$line2"
exit 0
```

ÖRNEK 4.2 KOD BLOĞU SONUÇLARININ BİR DOSYAYA YAZILMASI

```
#!/bin/bash
# rpm-check.sh
# Queries an rpm file for description, listing, and whether it can be
installed.
# Saves output to a file.
#
# This script illustrates using a code block.

SUCCESS=0
E_NOARGS=65

if [-z "$1" ]
then
  echo "Usage: ' basename $0' rpm-file"
  exit $E_NOARGS

fi
{
  echo
  echo "Archive Description:"
  rpm –qpi $1                 # Query description.
  echo
  echo "Archive Listing:"
  rpm –qpl $1                 #Query listing.
  echo rpm –i -- test $1      # Query whether rpm file can be installed.
  if [ "$?" –eq $SUCCESS ] then
    echo "$1 can be installed."
  else
    echo "$1 cannot be installed."
  fi
  echo
} > "$1.test"            # Redirects output of everything in block to file.
```

```
echo  "Results of rpm test in file $1.test"

# See rpm man page for explanation of options.

exit 0
```

Yukarıdaki gibi (parantez) içindeki bir komut grubunun aksine, kıvrık parantez içindeki kod bloğu, normalde altkabuğu çalıştırmayacaktır.

{ } \;

Yol adı.

Çoğunlukla find yapıları içinde kullanılır. Kabuk yerleşiği değildir.

find komut dizisinin `-exec` seçeneği genellikle `;` (noktalı virgül) ile biter. Kabuk tarafından yorumlanmaması için değiştirme özel karakteriyle gösterilir.

[ ]

test.

[] içindeki test ifadesi. Unutmayınız ki, sol köşeli parantez [kabuğa yerleşik testin bir parçası (ve ona bir kısaltma) olup, `/usr/bin/test` dış komutuna bir bağlantı değildir.

[[ ]]

[[]] içindeki test ifadesi (kabuk anahtar sözcüğü).

[ ] dizilim elemanı.

Bir dizilim bağlamında, [] o dizilimin her elemanını numaralandırır.

```
array[1]=slot_1
echo ${array[1]}
```

[ ]

karakter aralığı.

Bir kurallı ifadenin parçası olarak, köşeli parantezler eşleşecek karakterlerin aralığını belirler.

(( ))

Tamsayı açılımı. (( )) arasında yazılan ifadeyi açar ve değerlendirir.

> &> >& >> <

Yönlendirme.

`betikadı > dosyadı` betikadı'nın çıktısını dosyadı'na yönlendirir. dosyadı varsa da, üzerine yazar.

`komut &> dosyadı` komutun hem `stdout` ve hem de `stderr`'ini dosyadı'na yönlendirir.

`komut > &2 stderr` komutun `stdout`'unu `stderr`'e yönlendirir.

`betikadı >> dosyadı` betikadı'nın çıktısını dosyadı'na ekler. `dosyadı` mevcut değilse, oluşturulur.

İşlem değiştirme

`(komut)>`

`<(komut)`

< ve > karakterleri dizgi karşılaştırma operatörü olarak görev görür.

Diğer farklı bağlamlarda < ve > karakterleri tamsayı karşılaştırma operatörü olarak da görev görür. Bkz. Örnek 12-6.

<<

Sonuna yönlendirme 17.Bölümde açıklanacaktır.

<, >

ASCII karşılaştırması.

```
veg1=carrots
veg2=tomatoes
if [[ "$veg1" < "$veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo "this implies nothing about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

\<, \>

Kurallı ifadede kelime sınırı.

```
bash$ grep '\<the\>' metindosyası
```

|

Oluk

Önceki komutun çıktısını bir sonrakinin girdisine veya kabuğa geçirir. Bu, komutları zincirlemek için bir yöntemdir.

```
echo ls -l | sh
# Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".cat *.

lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

Oluk, işlem süreçleri arasındaki iletişimin klasik bir modelidir, bir işlem sürecinin `stdout`'unu diğerinin `stdin`'ine yollar. Tipik olarak, `cat` veya `echo` gibi bir komut, veri akışını bir "filtre"ye oluklar. Filtre, girdisini işlenmek üzere dönüştüren bir komuttur.

```
cat $filename | grep $search_word
```

Komut ya da komutların çıktısı bir betiğe oluk yapılabilir.

```
#!/bin/bash
# uppercase.sh : Changes input to uppercase.

tr ' a-z' ' A-Z'
# Letter ranges must be quoted
#+ to prevent filename generation from single-letter filenames.
exit 0
```

Şimdi, `ls -l` komutunun çıktısını yukarıdaki betiğe oluklayalım.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--  1 BOZO BOZO        109 APR 7  19:49  1.TXT
 -RW-RW-R-- 1 BOZO BOZO        109 APR 14 16:48  2.TXT
 -RW-R—_R-- 1 BOZO BOZO        725 APR 20  20:56 DATA-FILE
```

Bir işlem, kendinden önce gelen işlemin `stdout`'unu `stdin` olarak okumak zorundadır. Eğer böyle olmazsa, veri akışı bloke edilir ve oluğun davranışı beklenen şekliyle gerçekleşmez.

```
cat file1 file2 | ls -l | sort
# The output from "cat file1 file2" disappears.
```

Oluk, bir alt işlem süreci olarak çalışır, ve bu nedenle betik değişkenlerinin değerini değiştiremez.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable=$variable"           # variable = initial_value
```

Olukta en az bir komutun çalışması başarısız sonlanırsa, oluğun çalışması zamanından önce kesintiye uğrar. Oluğun kesilmesiyle (kesik oluk), SIGPIPE sinyali yollanır ve bu durum bildirilir.

>|

Kuvvet yönlendirmesi (`noclobber` seçeneği ayarlanmış olsa bile). Bir dosyanın üzerine zorla yazar.

||

VEYA mantıksal operatörü.

Bir test yapısı olarak || operatörü, bağlantılı test koşullarından herhangi biri doğruysa 0 (başarılı) bir dönüş yapar.

&

Arka planda iş çalıştırır. Bir komutu izleyen & komutu arka planda çalıştıracaktır.

```
bash$ sleep 10 &
[1]  850
[1]+ Done sleep 10
```

Bir komut dosyası içinde, komutları ve hatta döngüleri arka planda çalıştırabilirsiniz.

Örnek 4-3. Bir döngüyü arka planda çalıştırmak

```
#!/bin/bash
background –loop.sh

for i in 1 2 3 4 5 6 7 8 9 10   # First loop.
do
  echo –n "$i "
done &                          # Run this loop in background.
                                # Will sometimes execute after second loop.
echo                            # This ' echo' sometimes will not display.

for i in 11 12 13 14 15 16 17 18 19 20  # Second loop.
do
  echo –n "$i"
done

#==========================================================================

# The expected output from the script:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Sometimes, though, you get:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (The second ' echo' doesn't execute. Why?)

# Occasionally also:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (The first ' echo' doesn't execute. Why?)

# Very rarely something like:
# 11 12 13 14 15 16 17 18 19 20
# The foreground loop preempts the background one.

exit 0
```

Arkaplanda çalışan betik içindeki bir komut bir tuş vuruşu için beklerken betiğin askıda kalmasına neden olabilir. Neyse ki, bunun çaresi vardır.

&&

VE mantıksal operatörü. Bir test yapısında, && operatörü yalnızca, eğer bağlı test koşullarından her ikisi de doğruysa 0 (başarılı) bir dönüş döndürür.

-

Seçeneği, ön-ek. Bir komut ya da filtreleme için seçim bayrağı. Operatör için ön-ek.

```
COMMAND -[Option1][Option2][…]

ls -al

sort -dfu $filename

set -- $variable
```

```
if [$file1 -ot $file2 ]
then
  echo "File $file1 is older than $file2."
fi

if [ "$a" -eq "$b" ]
then
  echo "$a is equal to $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47]
then
  echo "$c equals 24 and $d equals 47."
fi
```

-

Stdin veya stdout a/dan yönlendirme. [dash]

```
(cd /source/directory && tar cf - . )  | ( cd /dest/directory  &&  tar xpvf -)

# Move entire file tree from one directory to another
# [courtesy Alan Cox a.cox@swansea.ac.uk, with a minor change]

# 1) cd /source/directory        Source directory, where the files to be moved are.
# 2) &&                          "And-list": if the ' cd' operation successful, then
execute the next command.
```

```
# 3) tar cf - .                    The ‘ c’ option ‘ tar’ archiving command creates a
new archive,
#                                  the ‘ f’ (file) option, following ‘ -‘  designates
the target file as stdout,
#                                  and do it in current directory tree (‘ .’) .
$ 4) |                             Piped to …
# 5 ( ... )                        a subshell
# 6) cd /dest/directory            Change to the destination directory.
# 7) &&                            “And-list”, as above
# 8) tar xpvf -                    Unarchive (‘ x’), preserve ownership and file
permissions (‘ p’),
#                                  and send verbose messages to stdout (‘ v’),
#                                  reading data from stdin (‘ f’ followed by ‘ -‘ ).
#
#                                  Note that ‘ x’ is a command, and ‘ p’, ‘ v’, ‘ f’ are
options.
# Whew!
# More elegant than, but equivalent to:
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
#
# cp –a /source/directory/dest     also has same effect.


bunzip2 linux-2.4.3.tar.bz2 | tar xvf –
# -- uncompress tar file--  | --then pass it to “tar”–
# If “tar” has not been patched to handle “bunzip2”,
# this needs to be done in two discrete steps, using a pipe.
# The purpose of the exercise is to unarchive “bzipped” kernel source.


Bu bağlamda “-“ tekil halde Bash operatörü değildir, ama tar, cat gibi belirli UNIX
yardımcı programlarının stdout’a yazarken tanıdığı bir seçenektir.

bash$ echo “whatever” | cat –
whatever
```

Bir dosyanın beklendiği durumlarda, `-` çıkışı `stdout`'a yönlendirir. (bazen `tar cf` ile kullanılır.) veya bir dosyadan değil de, yerine `stdin`'den girdi bekler.

Bir oluk içindeki filtrenin dosya odaklı yardımcı program olması yöntemidir.

```
bash$ file
usage: file [bciknvzL] [-f namefile] [-m magicfiles] file…
```

Komut satırında, dosya bir hata iletisiyle kendisi tarafından başarısız olur.

Daha faydalı bir sonuç için ”–“ ekleyin. Kabuğun bir kullanıcı girişi bekliyor olmasına neden olur.

```
bash$ file –
abc
standard input:

bash$ file –
#!/bin/bash
standard input:
```

Şimdi komut `stdin`'den girdi kabul eder ve onu analiz eder.

-

`stdout`'un diğer komutlar için olukla işlem yapmasında kullanılır.

Bir dosyanın başına satır eklemek gibi beceri isteyen işlerde yardımcı olur.

Bir dosyanın içindeki verilerin başkasıyla karşılaştırılmasında `diff` kullanılır.

```
grep Linux file1 | diff file2  -
```

Gerçek dünyadan bir örnek olarak son gün değişen tüm dosyaların yedeğini, `tar` kullanarak almayı son olarak göstereceğiz.

```
#!/bin/bash
# Backs up all files in current directory modified within last 24 hours
#+ in a "tarball" (tarred and gzipped file).

NOARGS=0
E_BADARGS=65

if [ $#=$NOARGS]
then
  echo "Usage: ' basename $0' filename"
```

```
  exit $E_BADARGS
fi

tar cvf – ‘ find . –mtime -1 –type f –print’ > $1.tar
gzip $1.tar

# Stephane Chazelas points out that the above code will fail
#+ if there are too many files found
#+ or if any filenames contain blank characters.
# He suggests the following alternatives:
#-------------------------------------------------------------------------
#  find . –mtime -1 –type f –print0 | xargs -0
#     using the GNU version of “find”.
#  find . –mtime -1 –type f –exec tar rvf “$1.tar” ‘ {}’\;
#     portable to other UNIX flavors, but much slower.
#-------------------------------------------------------------------------
exit 0
```

`-` ile başlayan dosya adlarının yeniden yönlendirme ile birlikte kullanılması sorun olabilir. Betik tarafından bu kontrol yapılabilir ve dosya adının önüne uygun bir ön-ek eklenebilir, örneğin `./-FILENAME,$PWD/-FILENAME, veya $PATHNAME/-FILENAME.`

Bir değişkenin değeri – ile başlıyorsa, benzer şekilde sorun yaratabilir.

```
var=”n”
echo $var
# Has the effect of “echo –n”, and outputs nothing
```

-

Önceki çalışma dizini [dash]. `cd -` önceki çalışma dizinine değişiklik yapar. `$OLDPWD` çevre değişkenini kullanır.

Burada kullanılan "-" ile az önce değinilen "-" yönlendirme operatörünü karıştırmayınız. Bu ikisinin yorumlanması koşullara bağlıdır.

-

Eksi. Bir aritmetik işlemde eksi.

=

Eşittir. Aritmetik operatörü.

```
a=28
echo $a   # 28
```

Farklı bir bağlamda, "=" operatörü bir karakter dizisini karşılaştırmada kullanılır.

+

Artı. Toplama aritmetik operatörü.

Farklı bir bağlamda, + sıralı ifade operatörüdür.

+

Seçenek. Bir komut ya da filtreleme için Seçenek bayrağı.

Bazı komut ve yerleşikler + operatörünü bazı seçenekleri etkinleştirmek için ve – operatörünü de onları etkisiz kılmak için kullanırlar.

%

Modulus operatörü. Bir bölmenin kalanını modulus ile ifade edebiliriz. Farklı bir bağlamda, % operatörü karakter dizi sıra ve desenlerinin eşleştirilmesinde de kullanılır.

~

Ana dizin. [tilde] Bu `$HOME` değişkenine karşılık gelir. `~bozo` bozo'nun ana dizinidir ve `ls ~ bozo` bunun içeriğini listeler.

`~/` geçerli kullanıcının ev dizinidir, ve `ls ~/` bunun içeriğini listeler.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
~+
```

geçerli çalışma dizini. `$PWD` iç değişkenine karşılık gelir.

~-

önceki çalışma dizini. `$OLDPWD` iç değişkenine karşılık gelir.

^
satır-başlangıcı.

Bir sıralı ifadede, "^" bir metin satırının başını gösterir.

Denetim Karakterleri

Terminal davranışını değiştirir veya metin görüntüler. Bir kontrol karakteri CTRL+tuş kombinasyonudur.

- **`Ctl - C`**

  Bir ön plan işi sonlandırır.

- **`Ctl – D`**

  Bir kabuk oturumunu kapatır ve çıkar. (exit ' e benzer.)

  "EOF" (dosya sonu). Standart girdiden girişi de sonlandırır.

- **Ctl-G**

  BEL Bip

- **Ctl-H**

  Geri tuşu.

```
#!/bin/bash
# Embedding Ctl-H in a string.

a="^H^H"                # Two Ctl-H's (backspaces).
echo "abcdef"           # abcdef
echo -n "abcdef$a "     #abcd f
#   Space at end ^           ^ Backspaces twice
echo -n "abcdef$a"      # abcdef
#  No space at end             Doesn't backspace (why?).
                  # Results may not be quite as expected.
echo; echo
```

- **Ctl-J**

  Satır başı

- **Ctl-L**

  Sayfa ilerletme karakteri (terminal ekranını temizler). Bu, `clear` komutu ile aynı etkiye sahiptir.

- **Ctl-M**

  Yeni satır.

- **Ctl-U**

  Bir satır girişini siler.

- **Ctl-Z**

  Bir ön plan işi duraklatır.

Boşluk karakteri

Bir ayırıcı olarak görev yapar, komutları veya değişkenleri ayırır. Boşluk için boş karakter, tab, boş satırlar, veya bunların herhangi bir kombinasyonu kullanılabilir. Bazı bağlamlarda, değişken atama gibi, boşluk karakterine izin verilmez, ve bu sözdizimi hatasına neden olur.

Bir komut dosyasının eylemi üzerinde boş satırların hiçbir etkisi yoktur, ve bu nedenle görsel fonksiyonel bölümlere ayırmada kullanışlıdır.

$IFS, bazı komutlar için girdi alanlarını ayıran özel değişken, hiçbir değer atanmadığı sürece boşluk karakterine eşittir.

Notlar

[1] Kabuk parantez açılımı yapar. Komut açılımın sonucu üzerine davranır.

[2] Özel durum: Oluğun bir parçası olan parantez içindeki kod bloğu bir altkabuk halinde çalışabilir.

```
ls  |  { read firstline; read secondline; }
# Error. The code block in braces runs as a subshell,
# so the output of “ls” cannot be passed to variables within the block.
echo “First line is $firstline; second line is $secondline” # Will not
work.

#Thanks, S.C.
```

BÖLÜM 5 DEĞİŞKEN ve PARAMETRELERE GİRİŞ

Değişkenler programlama dillerinin ve komut dosyalarına ait betik dillerinin merkezinde yer alır. Aritmetik işlem ve miktar manipülasyonu yapılması, dize ayrıştırılması ve semboller (başka bir şeyi temsil eden belirteçler) ile soyut olarak çalışmakta gereklidir. Değişken dediğimizde anlaşılması gereken herhangi bir veri maddesi değildir, fakat bilgisayar belleğininin belirli bir konumunda veya konumlarında tutulan veri maddesidir.

5.1 DEĞİŞKEN DEĞİŞİMİ

Bir değişkenin adı, değeri için yer tutucu olarak görev yapar, en kısa deyişle veri tutar. Değişkenin değerine referans etmeye ise değişken değişimi denir.

$

Değişkenin adını ve değerini dikkatli bir şekilde ayırt etmeliyiz. Değişken1 bir değişken adı ise, $Değişken1 değeri, içerdiği veri öğesi için bir başvuru, referanstır. Bir değişkenin $ ön-ekinin bulunmaksızın, değerinin olmaması ancak henüz tanımlanmamış olması veya değer atanmamış olması, dışa aktarımı veya bir sinyali temsil etmesi nedeniyle mümkündür (bkz.Örnek 30.5) Atama = sembolü ile (var1=27), bir oku cümlesi ile ve bir döngünün tekrar edilen baş satırında (for var2 in 1 2 3) yapılabilir.

Çift tırnak içine bir değişken yazılması ile değişkene referans başvuru yapılması arasında doğrudan ilişki kurulması yanlıştır. Birincisi "zayıf referans" olarak bilinir. Tek tırnak kullanımı değişken adının sözcük olarak kullanımına neden olur, ve herhangi bir değişim olmaz. Bu ikincisi "tam referans" veya "güçlü referans" olarak bilinir. Detaylı açıklamalar için bkz.Bölüm 6.

Unutmayınız ki $değişken ${değişken} ifadesinin basitleştirilmiş bir diğer yazılışıdır. $değişken sözdizimi hataya sebep olarak düşünüldüyse, uzun formuyla da çalışabilir (bkz. Bölüm 9.3)

ÖRNEK 5.1 DEĞİŞKEN ATAMA ve DEĞİŞTİRME

```
#!/bin/bash

#Variables: assignment and substitution

a=375

hello=$a

# -------------------------------------------------------------------------

#No space permitted on either side of = sign when initalizing variables.

#    If "VARIABLE =value",

#+ script tries to run "VARIABLE" command with one argument, "=value".

#    If "VARIABLE= value",

#+ script tries to run "value" command with

#+ the environmental variable "VARIABLE" set to "".

#-------------------------------------------------------------------------

echo hello               # Not a variable reference, just the string
"hello".

echo $hello

echo ${hello}            # Identical to above.


echo"$hello"

echo "${hello}"


echo


hello="A B    C     D"

echo $hello              # A B C D

echo "$hello"            # A B   C      D

# As you see, echo $hello  and echo "$hello" give different results.

# Quoting a variable preserves whitespace.

echo


echo ' $hello'           # $hello
```

```
#   Variable referencing disabled by single quotes,
#+ which causes the "$" to be interpreted literally.


# Notice the effect of different types of quoting.


hello=                  # Setting it to a null value.
echo "\$hello (null value) = $hello"
#    Note that setting variable to a null value is not the same as
#+  unsetting it, although the end result is the same (see below).
# ------------------------------------------------------------------------
#    It is permissible to set multiple variables on the same line,
#+  if separated by white space.
#    Caution, this may reduce legitibility, and may not be portable.
var1=variable1    var2=variable2   var3=variable3
echo
echo "var1=$var1   var2=$var2   var3=$var3"
#   May cause problems with older versions of "sh".
# ------------------------------------------------------------------------
echo; echo


numbers="one two three"
other_numbers="1 2 3"
# If whitespace within a variable, then quotes necessary.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"     #  other_numbers = 1  2  3
echo
echo "uninitialiazed_variable = $uninitialiazed_variable"
#  Uninitialized variable has null value (no value at all).
uninitialized_variable=              #   Declaring, but not initializing it
#+ (same as setting it to a null value, as above).
```

```
echo "uninitialized_variable = $uninitialized_variable"
                                          #  It still has a null value.
uninitialized_variable=23                 #  Set it.
uninitialized_variable                    #  Unset it.
echo "uninitialized_variable  =  $uninitialized_variable"
                                          #  It still has a null value.
echo

exit 0
```

Henüz atanmamış bir değişkenin değeri -atanan değer hiçtir anlamında- yoktur (sıfır değildir!). Bir değişkenin değer atanmadan kullanılması ciddi bir hatadır.

5.2 DEĞİŞKEN ATAMA

=

Atama operatörü (öncesi ve sonrasında boşluk yazılmamalıdır.)

Bu operatörleri, atamadan çok test amaçlı kullanılan = ve -eq ile karıştırmayınız.

Bağlam yönelimli olarak düşünülebildiği için = bir atama veya test operatörü olabilir.

ÖRNEK 5.2 DEĞİŞKEN ATAMANIN EN BASİT HALLERİNE ÖRNEKLER

```
#!/bin/bash
echo
# When is a variable "naked", i.e., lacking the ' $'in front?
# When it is being assigned, rather than referenced.
# Assignment
a=879
echo "The value of \"a\" is $a"
```

```
# Assignment using ‘ let’
let a=16+5
echo “The value of \”a\” is now $a”
echo
# In a ‘ for’ loop (really, a type of distinguishedassignment)
echo –n “The values of \”a\” in the loop are “
for a in 7 8 9 11
do
  echo –n “$a “
done

echo
echo
# In a ‘ read’ statement (also a type of assignment)
echo –n “Enter \”a\” “
read a
echo “The value of \”a\” is now $a”
echo
exit 0
```

ÖRNEK 5.3 DEĞİŞKEN ATAMANIN BASİT ve KARMAŞIK HALLERİNE ÖRNEKLER

```
#!/bin/bash
a=23                    # Simple case
echo $a
b=$a
echo $b
# Now, getting a little bit fancier (command substitution).
```

```
a='echo Hello!'          # Assigns result of ' echo'
echo $a
#  Note that using an exclamation mark (!) in command substitution
#+ will not work from the command line,
#+ since this triggers the Bash "history mechanism."
#  Within a script, however, the history functions are disabled.
a='ls -l'                #  Assigns result of ' ls -l' command to ' a'
echo $a                  #  Unquoted, however, removes tabs and newlines.
echo
echo "$a"                #  The quoted variable preserves whitespace.
#  (bkz.Referans başlıklı Bölüm.)
exit 0
```

$ (...) mekanizması kullanılarak değişken atama, ters tırnak yönteminden daha yenidir.

```
# From /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

5.3 BASH DEĞİŞKENLERİNİN TÜRÜ TANIMLANMAZ

Bash değişkenleri, diğer birçok programlama dillerinde kullanılan değişkenlerden farklı olarak tür- tanımsızdır. Gerçekte, Bash değişkenleri karakter dizeleridir, ancak, bağlama göre değişecek şekilde Bash, tamsayı işlemlerine izin verir ve değişkenler üzerinde karşılaştırma yapılmasına izin verir. Belirleyici faktör, değişkenin değerinin sadece rakamlardan oluşup oluşmadığıdır.

ÖRNEK 5-4. TAMSAYI veya KARAKTER DİZESİ?

```
#!/bin/bash
#  int-or-string.sh
#  Integer or string?

a=2334
let  "a  += 1"
echo "a= $a "
echo

b=${a/23/BB}
echo "b = $b"
declare -i b
echo "b = $b"

let "b += 1"
echo "b = $b"
echo

c=BB34
echo "c = $c"
d=${c/BB/23}
echo "d = $d"
let "d += 1"
echo "d = $d"

#  Variables in Bash are essentially untyped.
exit 0
```

Türsüz değişkenler hem istenir hem istenmez özellikler taşır. Betikte esnekliğe izin verir ve daha fazla kod yazmak kolaydır. Ancak, özensiz programlama alışkanlıklarıyla birlikte hata yapılmasına sebep olabilir.

Betik değişkenlerinin türü ile ilgili sorumluluk programcıya ait olmalıdır. Bash bunu sizin için yapmaz.

5.4 ÖZEL DEĞİŞKEN TÜRLERİ

Yerel değişkenler

Sadece bir kod bloğu veya fonksiyon içindeki görünür değişkenlerdir. (bkz.Yerel değişkenler ve fonksiyonlar.)

5.5 ÇEVRESEL DEĞİŞKENLER

Kabuk ve kullanıcı arabiriminin davranışını etkileyen değişkenler

Daha genel bir bağlamda, her bir işlem sürecine ait bir çevre düşünülebilir. Diğer bir deyişle, bir grup değişken sürec içinde işlemin işlemesi için çevreye başvuru referansı yapar. Bu anlamda, kabuk diğer her süreç gibi davranır.

Bir kabuk her başladığında, kendi çevresel değişkenlerine karşılık gelen kabuk değişkenleri oluşturur. Yeni kabuk değişkenlerinin güncellenmesi veya eklenmesi, kabuğun çevresini güncellemesine neden olur, ve kabuğun tüm alt işlem süreçleri (çalıştırdığı komutlar) çevreyi kalıtsal olarak kullanır.

Çevreye ayrılmış olan alan sınırlıdır. Çok fazla sayıda veya veya aşırı alan kullanan ortam değişkenleri sorunlara neden olabilir.

```
bash$ eval "'seq 10000 | sed –e ' s/.*/export var&=ZZZZZZZZZZZ/''"


bash$ du

bash: /usr/bin/du: Argument list too long


(Thank you, S.C. for the clarification, and for providing the above
example.)
```

Çevresel değişkenlere betik tarafından değer atanırsa, bunlar "dışa aktarılmalıdır", bir başka deyişle, betiğe yerel olan çevreye rapor edilir. **`export`** komutunun işlevi de tam budur.

Bir komut dosyası, sadece kendi alt süreçlerine değişken aktarır, aktarılan komut veya süreçlere başlangıç değeri atanır. Komut satırından çağrılan bir betik komut satırı çevresine değişken aktaramaz. Alt süreçler kendilerini yaratan üst süreçlere değişken aktaramazlar.

Konum parametreleri

Komut satırından betiğe geçirilen argümanlar, - $0, $1, $2, $3 ... $0 betiğin kendi adıdır, $1 birinci, $2 ikinci, $3, üçüncü argümandır, vb. $9'dan sonra argümanlar parantez içine alınmalıdır , ${10}, ${11}, ${12}.

ÖRNEK 5.5 KONUMSAL PARAMETRELER

```
#!/bin/bash

#  Call this script with at least 10 parameters, for example

#  ./scriptname 1 2 3 4 5 6 7 8 9 10

echo

echo "The name of this script is \"$0\"."

#  Adds ./ for current directory

echo "The name of this script is \"'basename $0'\"."

#  Strips out path name info (see ' basename')

echo
```

```
if  [ -n  "$1" ]             #  Tested variable is quoted.
then
   echo "Parameter #1 is $1"         # Need quotes to escape #
fi

if  [ -n "$2" ]
then
   echo "Parameter #2 is $2"
fi
if [ -n "$3" ]
then
   echo "Parameter #3 is $3"
fi
# . . .
if [ -n "${10}" ]  # Parameters  >$9 must be enclosed in {brackets}.
then
   echo "Parameter #10 is ${10}"
fi
echo
exit 0
```

Bazı komut dosyaları çağrıldıkları ada bağlı olarak değişik işlemler yapar.

Çağrılan komutu çalıştırmak için, komutun ismini tutan $0 değeri kontrol edilir.

Betiğin tüm diğer adlarına sembolik linklerin mevcut olması gereklidir.

Bir komut dosyasının, komut satırından girilmesi gereken parametre beklentisi varsa, ve komut dosyası parametresiz çağrıldıysa bunun sonucu olarak boş değişken atanır; bu çoğu zaman istenmeyen bir sonuçtur. Bunu önlemenin bir yolu, konumsal parametre kullanarak atama deyiminin her iki yönü için fazladan bir karakter eklemektir.

```
variable1_=$1_

#  This will prevent an error, even if positional parameter is absent.

critical_argument01=$variable1_

#  The extra character can be stripped off later, if desired, like so.

variable1=${variable_/_/}     #  Side effects only if $variable1_ begins
with "_".

#  This uses one of the parameter substitution templates discussed in
Chapter 9.

#  Leaving out the replacement pattern results in a deletion.

#  A more straightforward way of dealing with this is

#+  to simply test whether expected positional parameters have been passed.

if [ -z  $1 ]

then

   exit $POS_PARAMS_MISSING

fi
```

Örnek 5-6. Wh, whois alan adı arama

```
#!/bin/bash

#  Does a ' whois domain-name' lookup on any of 3 alternate servers :

#                                    ripe.net, cw.net, radb.net

#  Place this script, named ' wh' in usr/local/bin


# Requires symbolic links:

# ln –s /usr/local/bin/wh /usr/local/bin/wh-ripe

# ln –s /usr/local/bin/wh /usr/local/bin/wh-cw

# ln –s /usr/local/bin/wh /usr/local/bin/wh-radb


if [ -z "$1" ]

then

   echo "Usage: ' basename $0' [domain-name]"

   exit 65

fi
```

```
ase ‘ basename $0’ in
#  Checks script name and calls proper server
       “wh”       )  whois $1@whois.ripe.net;;
       “wh-ripe” )  whois $1@whois.ripe.net;;
       “wh-radb” )  whois $1@whois.radb.net;;
       “wh-cw”    )  whois $1@whois.cw.net;;
       *          )  echo “Usage: ‘ basename $0’ [domain-name]
esac
exit 0
```

`shift` komutu konumsal parametreleri yeniden tanımlar, görevi parametreleri birer pozisyon sol tarafa kaydırmaktır.

```
#!/bin/bash
#  Using ‘ shift’ to step through all the positionalparameters.
#  Name this script something like shft,
#+ and invoke it with some parameters, for example
#                  ./shft a b c def 23 skidoo
until [ -z “$1” ]     # Until all parameters used up …
do
  echo –n “$1 “
  shift
done
echo                  #  Extra line
exit 0
```

`shift` komutu da bir fonksiyona geçirilen parametreler üzerinde çalışır.

Notlar

[1] Betiği çalıştıran işlem süreci $0 komutunu çalıştırır. Gelenek gereği, bu parametre komut dosyasına verilen isimden ibarettir. Yardım için bkz. `execv` sayfası.

BÖLÜM 6 REFERANS

Referans bir karakter dizesini tırnak içine almak demektir. Karakter dizesinde yer alan özel karakterlerin yeniden yorumlanması veya kabuk ya da kabuk betiği tarafından genişlemesi etkisi vardır. (Bir karakter sözcük anlamından farklı bir anlamı varsa özeldir örneğin * joker karakteri gibi.)

```
bash$ ls –l [Vv] *

-rw-rw-r--        1  bozo  bozo      324  Apr 2  15:05  VIEWDATA.BAT

-rw-rw-r--        1  bozo  bozo      507  May 4 14:25  vartrace.sh

-rw-rw-r--        1  bozo  bozo      539  Apr 14 17:11 viewdata.sh
```

```
bash$ ls –l ‘ [Vv]*’

ls: [Vv]*: No such file or directory
```

Bazı program ve yardımcı programlar referans karakter dizesinin içindeki özel karakterleri genişletip yeniden yorumlayabilirler. Bu, referans kullanımının önemini gösterir, şöyle ki çağıran program onu genişletmesine izin verdiği halde hala komut satırı kabuktan korunmaktadır, referansın önemi buradadır.

```
bash$ grep ‘ [Ff]irst’ *.txt

file1.txt:This is the first line of file1.txt.

 file2.txt:This is the First line of file2.txt.
```

Bilginiz için, `grep [Ff]irst *.txt` çalışmayacağı doğaldır.

Bir değişkene referans etmek için, çoğu zaman onu çift tırnak içine almak gerekir. Bu, değişken isminin içindeki özel karakterlerin hepsini korur, sadece $ dışında, ‘(ters tırnak), ve \ \(kaçış). $ özel karakteri (yukarıda bkz.Örnek 5.1) değişkene referans etmeye izin verir, değişkenin değerini göstermeye yarar.

Sözcük bölmeyi engellemek için çift tırnak işaretini kullanın. [1] Tırnak içine yazılmış bir bir argüman kendini boşluk karakterleri bile içerse, sadece bir bütün sözcükmüş gibi gösterir.

```
variable1="a variable containing five words"

COMMAND This is $variable1                    # Executes COMMAND with 7 arguments:

# "This" "is" "a" "variable" "containing" "five" "words"


COMMAND "This is $variable1"                  # Executes COMMAND with 1 argument:

# "This is a variable containing five words"


variable2=""                                  # Empty.


COMMAND $variable2 $variable2 $variable2      #  Executes COMMAND with

                                              #  no arguments.

COMMAND "$variable2" "$variable2" "$variable2" #  Executes COMMAND with

                                              #  3 empty arguments.

COMMAND "$variable2 $variable2 $variable2"    #  Executes COMMAND with

                                              #  1 argument (2 spaces).


# Thanks, S.C.
```

Argümanları tırnak işareti içine kapatarak echo cümlesi içine almak sadece kelimeleri bölmek (bir veya birden fazla parçaya ayırmak) gerektiğinde zorunludur.

ÖRNEK 6.1 İLGİNÇ DEĞİŞKENLERİN echo İLE GÖSTERİLMESİ

```
#!/bin/bash

# weirdvars.sh: Echoing weird variables.

var="'(]\\{}\$\""

echo $var           # ' (]\{}$"

echo "$var"         # ' (]\{}$"      Doesn't make a difference.
```

```
echo

IFS='\'

echo $var         # ' (] {}$"      \ converted to space.

echo "$var"       # ' (]\{}$"


# Examples above supplied by S.C.

exit 0
```

Tek tırnak (''), çift tırnağa benzer şekilde çalışır ancak, değişkenlere referans vermeye izin vermez. Nedeni, $ özel karakteri bu noktada işlevsel değildir.

Tek tırnak içindeki ' dışında kalan özel karakterlerin her biri kelime halinde ele alınır.

Tek tırnağı çift tırnağın daha özelleştirilmiş bir hali olarak düşünebilirsiniz.

Kaçış karakteri olan \ bile karakter olarak yorumlanır. Tek tırnağı tek tırnak içine yazamazsınız.

```
echo "Why can't I write ' s between single quotes"

echo

#  The roundabout method.

echo ' Why can'\''t I write ' "'"'s between single quotes'

echo

#    |-------|  |----------|    |------------------------|

#  Three single-quoted strings, with escaped and quoted single quotes
between.

#  This example courtesy of Stephane Chazelas.
```

Tek karakterleri \ ile yazmaya *kaçma* yöntemi denir. Bir karakterin hemen öncesindeki \ karakteri kabuğa o karakterin harf halinde ele alınmasını söyler.

`echo` ve `sed` gibi, belirli komutlar ve yardımcı programlar ile bir karakteri kaçma yöntemiyle yazamayabilirsiniz – çünkü bu, o karakterin özel anlamını değiştirir.

Belirli özel karakterlerin anlamları

`echo` ve `sed` ile kullanılırken

`\n`

Yeni bir satıra geçer.

`\r`

Satır sonundan bir sonrakinin başında kalır.

`\t`

Belirli sayıda boş karakter (yatay)

`\v`

Dikey `\t`

`\b`

Bir önceki karakteri siler.

`\a`

Bip veya uyarı.

`\0xx`

oktal `0xx` eşdeğerinin ASCII değeri.

ÖRNEK 6.2 KAÇMA YÖNTEMİYLE YAZILAN KARAKTERLER

```
#!/bin/bash
#  escaped.sh: escaped characters

echo;  echo

echo "\v\v\v\v"                      # Prints \v\v\v\v

#  Use the -e option with ' echo' to print escaped characters.

echo -e "\v\v\v\v"                   #  Prints 4 vertical tabs.
echo -e "\042"                       #  Prints " (quote, octal ASCII
character 42).

# The $'\X' construct makes the -e option unnecessary.

echo $'\n'                           #  Newline.
echo $'\a'                           #  Alert (beep).

#  Version 2 and later of Bash permits using the $'\xxx' construct.
echo $ ' \t \042 \t'                  # Quote (") framed by tabs.

#  Assigning ASCII characters to a variable.
# -----------------------------------------------------------------------
Quote=$'\042'                        # " assigned to a variable.
Triple_underline=$'\137\137\137'     #  137 is octal ASCII code for ' _'.
Echo "$triple_underline UNDERLINE $triple_underline"

ABC=$'\101\102\103\010'              # 101, 102, 103 are octal A, B, C.
echo $ABC

echo; echo

escape=$'\033'                       #  033 is octal for escape.
echo "\"escape\" echoes as $escape"  # no visible output.

echo;  echo

exit 0
```

Örnek 35.1 $ ' ' için bir başka örnektir. Burada, karakter dizesinin genişleme yapısı verilmiştir.

\"

Kaçma yöntemiyle harf olarak yazar.

```
echo "Hello"                                        # Hello

echo "\"Hello\", he said."                          # "Hello", he said.
```

\$

Kaçma yöntemiyle harf olarak yazar. (İzleyen değişken adına referans verilmeyecektir)

```
echo "\$variable01"               # results in $variable01
```

\\

Kaçma yöntemiyle harf olarak yazar.

```
echo "\\"                         # results in \
```

\ özel karakterinin davranışı kendisinin de \ ile yazılıp yazılmadığına göre değişir. Referans veya komut yer değişiminde bulunulup bulunmadığı ile ilgili de değişir.

```
# Simple escaping and quoting
echo \z                  #  z
echo \\z                 # \z
echo ' \z'               # \z
echo ' \\z'              # \z
#  Command substitution
echo ' echo \z'          #  z
echo ' echo \\z'         #  z
echo ' echo \\\z'        # \z
echo ' echo \\\\z'       # \z
echo ' echo \\\\\\z'     # \z
echo ' echo "\z"'        # \z
echo ' echo "\\z"'       # \z
# Here document
cat <<EOF
\z
EOF                      # \z

cat <<EOF
\z
```

```
EOF                     # \z
cat <<EOF
\\z
EOF                     # \z
# These examples supplied by Stephane Chazelas.
```

Bir karakter dizesine atanan ve bir karakter dizesini oluşturan elemanlar kaçma yöntemiyle yazılabilir, ancak kaçma karakteri tek başına bir değişkene atanamaz.

```
variable=\
echo "$variable"
# Will not work - gives an error message:
# test.sh : command not found
# A "naked" escape cannot safely be assigned to a variable.
#
# What actually happens here is that "\" escapes the newline and
#+ the effect is            variable=echo "$variable"
#+                          invalid variable assignment

variable=\
23skidoo
echo "$variable"            # 23skidoo
                            # This works, since the second line
                            #+ is a valid variable assignment.
variable=\
#        \^      escape followed by space
echo "$variable"            # space

variable=\\
echo "$variable"            # \
```

```
variable=\\\

echo "$variable"

# Will not work – gives an error message :

# test.sh: \: command not found

#

#  First escape escapes second one, but the third one is left "naked",

#+ with same result as first instance, above.


variable=\\\\

echo "$variable"             # \\

                             # Second and fourth escapes escaped.

                             # This is o.k.
```

Sözcük bölmeyi engellemek için boşluk karakterini kaçma yöntemiyle yazınız.

```
file_list="/bin/cat  /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"

# List of files as argument(s) to a command.

# Add two files to the list, and list all.

ls –l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "-------------------------------------------------"

#What happens if we escape a couple of spaces?

ls –l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list

#Error:the first three files concatenated into a single argument to ' ls –l'

#        because the two escaped spaces prevent argument (word)
```

Çok satırlı komutlar yazmak için de kaçma yöntemi kullanılır. Normalde, satırların her biri farklı bir satırdır. Ancak satır sonunda kaçış karakteri yazılırsa newline karakteri ile aynı sonucu doğurur; yani newline karakteri kaçma yöntemiyle yazılmış gibi düşünün. Komut sırası bir sonraki satıra geçer.

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Repeating Alan Cox's directory
# but split into two lines for increased legibility.
# As an alternative:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# See note below.
# (Thanks, Stephane Chazelas.)
```

Bir komut satırı oluk karakteri olan | ile biterse, kaçma yöntemini kullanmak muhakkak gerekli değildir. Buna rağmen, bir sonraki satıra geçen kodu kaçma yöntemiyle kullanmak iyi bir programlama alışkanlığıdır.

```
echo "foo
bar"
#foo
#bar

echo

echo ' foo
bar'            # No difference yet.
#foo
#bar

echo
echo foo\
bar             #  Newline escaped.
#foobar
```

```
echo


echo "foo\

bar"            # Same here, as \ still interpreted as escape within weak
quotes.

#foobar


echo


echo ' foo\

bar'            # Escape character \ taken literally because of strong
quoting.

#foo\

#bar

# Examples suggested by Stephane Chazelas.
```

Notlar

[1] Bu bağlamda "sözcük bölme", bir karakter dizesini birden fazla sayıya ve sonlu argümanlara bölme anlamındadır.

BÖLÜM 7 TESTLER

Eksiksiz her programlama dili, bir koşul için test yapabilir, daha sonra test sonucuna göre davranış gösterebilir. Bash'in bu amaçla **`test`** komutu, çeşitli parantez ve parantez operatörleri ve **`if/then`** yapısı vardır.

7.2 DOSYA TEST OPERATÖRLERİ

Doğru döndürür eğer …

-e

Dosya varsa

-f

Dosya normal bir dosya ise (dizin ya da aygıt dosyası değilse)

-s

Dosya boyutu 0 (sıfır) değilse

-d

Dosya bir dizin ise

-b

Dosya bir blok aygıtı ise (flopi, cdrom, vb.)

-c

Dosya bir karakter aygıtı ise (klavye, modem, ses kartı, vb.)

-P

Dosya bir oluk ise

-h

Dosya bir sembolik link ise

-L

Dosya bir sembolik link ise

-S

Dosya bir soket ise

-t

Dosya (tanımlayıcısı) bir terminal aygıtı ile ilişkili ise

Bu test, verilen bir betikteki `stdin ([ -t 0 ])` veya `stdout ([ -t 1 ])` 'un terminal olup olmadığını kontrol etmede kullanılır.

-r

Dosya (test'i çalıştıran kullanıcı için) okuma iznine sahipse

-w

Dosya (test'i çalıştıran kullanıcı için) yazma iznine sahipse

-x

Dosya (test'i çalıştıran kullanıcı için) çalıştırma iznine sahipse

-g

`set-group-id (sgid)` bayrağı dosya veya dizinde başlatıldıysa

Bir dizinde `sgid` bayrağı ayarlı ise, o dizin içinde oluşturulan bir dosya, dizinin sahibi olan gruba aittir, ille de dosyayı oluşturan kullanıcı grubuna ait olmak zorunda değildir. Bu bir çalışma grubu tarafından paylaşılan bir dizin için kullanışlıdır.

-u

Dosyanın `set-user-id (suid)` bayrağı ayarlı ise

`Root` tarafından sahiplenilmiş `set-user-id` bayrağına ayarlı, ikili moddaki bir dosya sıradan bir kullanıcı tarafından çağrılsa bile, kök ayrıcalıklarına sahiptir. [1] Bu sistem donanımına erişim için gerekli yürütülebilir dosyalar için (`pppd` ve `cdrecord` gibi) kullanışlıdır. SUID bayrağı eksik olan ikili dosyalar, kök-olmayan bir kullanıcı tarafından çağrılamaz.

```
-rwsr-xr-t       1 root        178236 Oct  2  2000 /usr/sbin/pppd
```

SUID bayrağı ayarlanmış bir dosyanın izinleri arasında *s* harfi gösterilir.

-k

`Yapışkan bit` ayarlı ise

Genellikle "yapışkan bit" olarak bilinen *kaydet-metin-modu* bayrağı, dosya izninin özel bir türüdür. Bir dosyada bu bayrak kümesi varsa, bu dosya daha hızlı erişim için önbellekte tutulacaktır. [2] Bir dizine ayarlarsanız, yazma hakkı iznini kısıtlar. Yapışkan bit ayarı, dosya veya dizin listeleme üzerindeki izinlere bir *t* harfi ekler.

```
drwxrwxrwt       7  root           1024  May 19 21:26 tmp/
```

Bir kullanıcı yapışkan biti olmayan bir dizin sahibi ise, ancak bu dizine yazma izni varsa, bu dizindeki dosyalardan sadece sahip olduğu dosyaları silebilir. Bu `/tmp` gibi herkese açık bir dizinde, kullanıcıların yanlışlıkla birbirlerinin dosyalarını silmelerini veya üzerlerine yazmalarını engeller.

-O

Dosyanın sahibi iseniz

-G

Dosyanın grup kimliği sizinki ile aynı ise

-N

Son okunduğundan bu yana dosya değiştirilmiş ise

`f1 –nt f2`

Dosya `f1 f2`'den daha yeni ise

`f1 -ot f2`

Dosya `f1` `f2`'den daha eski ise

`f1 –ef f2`

`f1` ve `f2` dosyaları aynı dosya için sabit bağlantılı ise

!

"değil" – yukarıdaki testlerin anlamını tersine çevirir (koşul mevcut değilse `doğru` döndürür).

Örnek 29.1 , Örnek 10.7, Örnek 10.3, Örnek 29.3, ve Örnek A.2 dosya test operatörlerin kullanımını gösterir.

Notlar

[1] Unutmayınız ki, *suid* ikilileri güvenlik deliği açabilir ve *suid* bayrağının kabuk betikleri üzerinde hiçbir etkisi yoktur.

[2] Modern UNIX sistemlerinde, yapışkan bit artık sadece dizinlerde kullanılmaktadır, dosyalar için kullanılmamaktadır.

7.3 KARŞILAŞTIRMA OPERATÖRLERİ (ikili)

tamsayı karşılaştırması

-eq

eşittir

```
if [ "$a" -eq "$b" ]
```

-ne

eşit değildir

```
if [ "$a" -ne "$b" ]
```

-gt

büyüktür

```
if [ "$a" -gt "$b" ]
```

-ge

büyük veya eşittir

```
if [ "$a" -ge "$b" ]
```

-lt

küçüktür

```
if [ "$a" -lt "$b" ]
```

-le

küçük veya eşittir

```
if [ "$a" -le "$b" ]
```

<

küçüktür (çift parantez içinde)

```
((“$a” < “$b”))
```

<=

küçük veya eşittir (çift parantez içinde)

```
((“$a” <= “$b”))
```

>

büyük veya eşittir (çift parantez içinde)

```
((“$a” >= “$b”))
```

karakter dizesi karşılaştırması

=

eşittir

```
if [ “$a” = “$b” ]
```

==

eşittir

```
if [ “$a” = “$b” ]
```

= için bir eşdeğerdir.

```
[[ $a == z* ]]    # true if $a starts with an “z” (pattern matching)
[[ $a == “z*” ]]  # true if $a is equal to z*
[  $a == z* ]     # file globbing and word splitting take place
[ “$a” == “z*” ]  # true if $a is equal to z*
# Thanks, S.C.
```

!=

eşit değildir

```
if [ “$a” != “$b” ]
```

Bu operatör `[[ ... ]]` içinde kalan eşleştirilen kalıp desenleri kullanır.

<

küçüktür, ASCII alfabe sırasını takiben

```
if [[ “$a” < “$b” ]]

if [ “$a” \< “$b” ]
```

Unutmayınız ki `[]` yapısı içinde `"<"` nin kaçma yöntemiyle yazılması gerekir.

>

büyüktür, ASCII alfabe sırasını takiben

```
if [[ “$a” > “$b” ]]

if [ “$a” \> “$b” ]
```

Unutmayınız ki `[]` yapısı içinde `">"` nin kaçma yöntemiyle yazılması gerekir.

Bu karşılaştırma operatörünün uygulanışı Örnek 26-4 'te gösterilmiştir.

-z

Karakter dizesi uzunluğu "boş", yani uzunluğu sıfırdır.

-n

Karakter dizesi uzunluğu "boş" değildir.

`-n` test seçeneği karakter dizesinin muhakkak parantez içine yazılmasını gerektirir. `! -z` seçenekleri tırnak içine yazılmamış bir karakter dizesi ile kullanıldığında, veya test parantezi içinde sadece referans edilmemiş bir karakter dizesi kullanıldığında (bkz.Örnek 7.5) komut çalışır, ancak bu alışkanlık tavsiye edilmez. Önerilen, her bir test dizesine alıntı yapmaktır.

ÖRNEK 7.4 ARİTMETİK ve DİZE KARŞILAŞTIRMALARI

```
#!/bin/bash

a=4

b=5

#  Here "a" and "b" can be treated either as integers or strings.

#  There is some blurring between the arithmetic and string comparisons,

#+ since Bash variables are not strongly typed.

#  Bash permits integer operations and comparisons on variables

#+ whose value consists of all-integer characters.

#  Caution advised.


if [ "$a" -ne "$b" ]

then

   echo "$a is not equal to $b"

   echo "(arithmetic comparison)"

fi


echo


if [ "$a" != "$b" ]

then

   echo "$a is not equal to $b."

   echo "(string comparison)"

fi


#  In this instance, both "-ne" and "!=" work.

echo

exit 0
```

ÖRNEK 7.5 BİR KARAKTER DİZESİNİN BOŞ OLUP OLMADIĞINI TEST ETME

```
#!/bin/bash
# str-test.sh: Testing null strings and unquoted strings,
# but not strings and sealing wax, not to mention cabbages and kings . . .

# Using if [ . . . ]

# If a string has not been initialized, it has no defined value.
# This state is called “null” (not the same as zero).

if [ -n $string1 ]      #  $string1 has not been declared or initialized.
then
   echo “String \”string\” is not null.”
else
   echo “String \”string1\” is not null.”
fi
# Wrong result.
# Shows $string1 as not null, although it was not initialized.

echo

#  Lets try it again.
if [ -n “$string1” ]      # This time, $string1 is quoted.
then
   echo “String \”string1\” is not null.”
else
   echo “String \”string1\” is not null.”
fi   # Quote strings within test brackets!
```

```
echo

if [ $string1 ]     # This time, $string1 stands naked.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
#  This works fine.
#  The [ ] test operator alone detects whether the string is null.
#  However it is good practice to quote it ("$string1").
#
#  As Stephane Chazelas points out,
#     if [ $string 1 ] has one argument, "]"
#     if [ "$string 1" ] has two arguments, the empty "$string1" and "]"

echo

string1=initialized
if [ $string1 ]          # Again, $string1 stands naked.
then
   echo "String \"string1\""
else
   echo "String \"string1\""
fi

# Again, gives correct result.
# Still, it is better to quote it ("$string1"),

string1="a = b"
```

```
if [ $string1 ]    # Again, $string1 stands naked.
then
   echo "String \"string1\" is not null."
else
   echo "String \"string1\" is null."
fi
# Not quoting "$string1" now gives wrong result!

exit 0
# Also, thank you, Florian Wisser, for the "heads-up".
```

ÖRNEK 7.6 zmost

```
#!/bin/bash
# View gzipped files with ' most'
NOARGS=65
NOTFOUND=66
NOTGZIP=67

if [ $# -eq 0 ]    # same effect as: if [ -z "$1" ]
# $1 can exist, but be empty: zmost "" arg2 arg3
then
   echo "Usage: ' basename $0' filename" >&2
   #  Error message to stderr.
   exit $NOARGS
   # Returns 65 as exit status of script (error code).
fi

filename=$1
```

```
if [ ! –f "$filename" ]    # Quoting $filename allows for possible spaces.
then
   echo "File $filename not found!" >&2
   #  Error message to stderr.
   Exit $NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
#  Using bracket in variable substitution.
then
   echo "File $1 is not a gzipped file!"
   exit $NOTGZIP
fi

zcat $1 | most
#  Uses the file viewer ' most' (similar to ' less')
#  Later versions of ' most' have file decompressioncapabilities.
#  May substitute ' more' or ' less', if desired.

exit $?     # Script returns exit status of pipe.
#  Actually "exit $?" unnecessary, as the script will, in any case,
#  return the exit status of the last command executed.
```

compound comparison

-a

mantıksal ve

ifade1 –a ifade2

hem ifade1 ve hem de ifade2 doğru olduğu halde doğru olur.

-o

mantıksal veya

ifade1 –o ifade2 ifade1 veya ifade2 her ikisi veya biri doğru olduğu halde doğru olur.

Bunlar && veya || benzeri Bash karşılaştırma operatörlerinin çift parantez içindeki hallerine benzer.

```
[[ condition1 && condition2 ]]
```

-o ve –a operatörleri test komutu ile birlikte çalışır veya tek bir test parantezi içinde yer alır.

```
if [ "$exp1" -a "$exp2" ]
```

Örnek 8.3 ve Örnek 26.8 bileşik karşılaştırma işleçlerinin çalışır hali için görülebilir.

Notlar

[1] S.C.'nin bir bileşik testte düzeltmesini yaptığı gibi, karakter dizesi değişkenine referans vermek belirli Bash sürümleri için yeterli olmayabilir. `[ -n "$string" -o "$a" = "$b" ]` Eğer `$string` boş ise hataya neden olabilir. Güvenli yol, boş değişkenlere ekstra birer karakter eklemektir, `[ "x$string" != x -o "x$a" = "x$b" ]` ("x'ler" birbirini iptal eder).

7.4 İÇ İÇE `if/then` KOŞULLU TESTLER

`if/then` koşullu test yapıları iç içe kullanılabilir. && bileşik operatörü kullanmak ile aynı sonucu verir.

```
if  [ condition1 ]
then
    if [ condition2 ]
    then
      do-something  # But only if both "condition1" and "condition2" valid.
    fi
fi
```

İç içe koşullu test örneği için bkz.Örnek 35.4.

7.5 TEST BİLGİNİZİ TEST EDİN

X sunucuyu kurmak için sistem kurallarının uyacağı `xinitrc` dosyası mevcuttur. Aşağıdaki alıntıda bu dosyanın içindeki çok sayıda `if/then` testi mevcuttur.

```
if [ -f $HOME/.Xclients ]; then
    exec $HOME/.Xclients
elif  [  -f /etc/X11/xinit/Xclients ]; then
   exec /etc/X11/xinit/Xclients
else
    # failsafe settings. Although we should never get here
    # (we provide fallbacks in Xclients as well) it can't hurt.
    xclock –geometry 100x100-5+5 &
    xterm –geometry 80x50-50+150 &
    if [ -f /usr/bin/netscape –a –f /usr/share/doc/HTML/index.html ]; then
            netscape /usr/share/doc/HTML/index.html &
    fi
fi
```

Yukarıdaki alıntıda hangi test yapıları vardır? `grep`, `sed`, ve sıralı ifadeleri önceden hatırlayarak önce tüm dosyayı, , ardından `/etc/X11/xinit/xinitrc` dosyasını incelemeniz ve oradaki `if/then` test yapılarını analiz etmeniz gerekebilir.

BÖLÜM 8 İŞLEMLER ve İLGİLİ KONULAR

8.1 OPERATÖRLER

atama

değişken atama

Başlatma ve bir değişkenin değerini değiştirme

=

Çok-amaçlı atama operatörü, aritmetik ve karakter dizesi atamaları için kullanılır.

```
var=27
category=minerals    # No spaces allowed after the "=".
```

= test operatörü ile = atama operatörü farklı anlam taşır.

```
#   = as a test operator

if [ "$string1" = "$string2" ]
# if [ "X$string1" = "X$string2" ] is safer,
# to prevent an error message should one of the variables be empty
# (The prepended "X" characters cancel out.)
then
   command
fi
```

aritmetik operatörler

+

artı

-

Eksi

*

Çarpma

/

Bölme

**

üs alma

```
#  Bash, version 2.02, introduced the "**" exponentiation operator.

let "z=5**3"

echo "z = $z"       # z = 125
```

%

`modulo`, veya `mod` (Bir tamsayı bölme işleminin kalanını döndürür)

```
bash$ echo ' expr 5 % 3'

2
```

Bu operatör diğer pek çok şeyi yaptığı gibi, belirli bir aralık içindeki sayıları üretmede de kullanım alanı bulur.

Örnek 9.21 ve Örnek 9.22 ve program çıktısının biçimlendirilmesine örnek olarak, bkz.Örnek 26.7 ve Örnek A.7.

Hatta, Örnek A.16, asal sayı üretmek için de kullanılabilir. Modulo çeşitli sayısal formüllerde de kendini gösterir.

ÖRNEK 8.1 EN BÜYÜK ORTAK BÖLEN

```
#!/bin/bash
# gcd.sh: greatest common divisor
          Uses Euclid's algorithm

#  The "greatest common divisor" (gcd) of two integers
```

```
#+ is the largest integer that will divide both, leaving no remainder.

#  Euclid's algorithm uses successive division.
#  In each pass,
#+ dividend <--- divisor
#+ divisor <--- remainder
#+ until remainder = 0.
#+ The gcd = dividend, on the final pass.
#
#  For an excellent discussion of Euclid's algorithm, see
#  Jim Loy's site, http://www.jimloy.com/number/euclids.htm

#  ------------------------------------------------------
#  Argument check
ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: ' basename $0' first-number second-number"
  exit $E_BADARGS
fi
# ------------------------------------------------------------

gcd()
{
  dividend=$1            #  It does not matter
  divisor=$2             #+ which of two is larger.
                         #  Why?
  remainder=1            #  If uninitialized variable used in loop,
                         #+ it results in an error message
                         #+ on first pass through loop.

  until [ "$remainder" -eq 0 ]
  do
     let "remainder = $dividend % $divisor"
     dividend=$divisor  # Now repeat with 2 smallest numbers.
     divisor=$remainder
  done                   # Euclid's algorithm
}                        # Last $dividend is the gcd.


echo; echo "GCD of $1 and $2 = $ dividend"; echo

# Exercise :
# --------
#  Check command-line arguments to make sure if they are integers,
#+ and exit the script with an appropriate error message if not.

exit 0
```

+=

"artı-eşit" (değişken değerini bir sabit kadar artırma)

`let "var += 5"` var değişken değerini 5 ile artırır.

`-=`

"eksi-eşit" (değişken değerini bir sabit kadar azaltır)

`*=`

"çarp-eşit" (değişken değerini bir sabit ile çarpar)

`let "var *= 4" var` değişken değerini 4 ile çarpar.

`/=`

"bölü-eşittir" (değişken değeri bir sabit ile bölünür)

`%=`

"mod-eşit" (değişken değerinin bir sabit ile bölünmesinden kalan)

Aritmetik operatörler çoğunlukla bir expr veya let ifadesinde kullanılır.

ÖRNEK 8.2 ARİTMETİK İŞLEMLERİ KULLANMA

```
#!/bin/bash
# Counting to 6 in 5 different ways.

n=1; echo –n "$n "

let "n = $n + 1"        #let n = n + 1" also works.
echo –n "$n "

: $((n = $n + 1))
#   ":" necessary because otherwise Bash attempts
#+ to interpret "$[ n = $n + 1 ]" as a command
#  Works even if "n" was initialized as a string.
echo -n "$n "

n=$[ $n + 1 ]
#  Works even if "n" was initialized as a string.
#* Avoid this type of construct, since it is obsolete and nonportable.
echo –n  "$n "; echo

# Thanks, Stephane Chazelas.

exit 0
```

Bash tamsayı değişkenleri aslında değer aralıkları -2147483648 ile 2147483647 aralığında değişen uzun (32-bit) tamsayılardır. Bu sınırların dışında bir değere sahip değişken alan işlemler hatalı sonuç verir.

```
a=2147483646
echo “a = $a”                     #  a = 2147483646
let “a+=1”                        #  Increment “a”.
echo “a = $a”                     #  a = 2147483647
let “a+=1”                        #  increment “a” again, past the limit.
echo “a = $a”                     #  a = -2147483648
                                  #         ERROR (out of range)
```

Bash kayan noktalı aritmetiği anlamaz. Ondalık nokta içeren sayıları karakter dizesi olarak değerlendirir.

```
a=1.5

let “b= $a + 1.3”                 # Error.
#  t2.sh: let: b = 1.5 + 1.3 :  syntax error in expression (error token is
“.5 + 1.3”)

echo “b = $b”                     # b=1
```

Kayan nokta hesaplamaları veya matematik kütüphanesi fonksiyonları gerektiren betiklerde bc kullanılır.

Bit-düzeyi operatörleri. Bit-düzeyinde işlem yapan operatörler kabuk betiklerinde pek görünmezler. Onların temel görevi yuvalardan veya soketlerden değer okuma ve değer işlemektir. “ Bit çevrimi “yüksek hızda çevrime izin veren C veya C++ gibi derlenmiş dillerle daha çok ilgilidir.

Bit-düzeyi operatörleri

<<

Bit-düzeyi sola kaydır (Kaydırılan her pozisyon için değer 2 ile çarpma işlemiyle çarpılmış gibi artar.)

<<=

“bit-düzeyi-sola-kaydır-ve-eşittir”

`let "var <<= 2"` var değişkeninin değerini 2 bit sola kaydırır (4 ile çarpılır)

>>

Bit düzeyi sağa kaydır (Kaydırılan her pozisyon için değer 2 ile bölme işlemiyle bölünmüş gibi azalır.)

>>=

“sağa-kaydır-ve-eşittir (<<= operatörünün tersi)

&

bit-düzeyinde-ve

&=

"bit-düzeyinde ve-eşittir"

|

bit-düzeyinde VEYA

|=

"bit-düzeyinde VEYA-eşittir"

~

"bit-düzeyinde olumsuzlanan"

!

bit-düzeyinde TERSİ

^

bit-düzeyinde XOR

^=

"bit-düzeyinde XOR-eşittir"

mantıksal operatörler

&&

ve (mantıksal)

```
if [ $condition1 ]   &&  [ $condition2 ]
     #  Same as:   if [ $condition1 –a $condition2 ]
     #  Returns true if both condition1 and condition2 hold true ...

     if [[ $condition1 && $condition2 ]]      # Also works.

     #  Note that && operator not permitted within [. . .] construct.
```

&&, farklı bağlamlarda, bir ve-listesinde komutları birleştirmek için de kullanılabilir.

||

veya (mantıksal)

```
if [ $condition1 ] || [ $condition2 ]
# Same as: if [ $condition1 -o $condition2 ]
# Returns true if either condition1 or condition2 holds true.

if [[ $condition1 || $condition2 ]]      # Also works.

# Note that || operator not permitted within [. . .] construct.
```

Bash bir mantıksal operatör bağlantılı her cümlenin çıkış durumunu kontrol eder.

ÖRNEK 8.3 && ve || KULLANAN BİLEŞİK KOŞULLU TESTLER

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
    echo "Test #1 succeeds."
else
    echo "Test #1 fails."
fi

# ERROR:       if [ "$a" -eq 24 && "$b" -eq 47]
#              attempts to execute '  [ "$a" -eq 24 '
#              and fails to finding matching ' ]'.
#
#    if [[ $a -eq 24 && $b -eq 24 ]]   Works
#    (The "&&" has a different meaning in line 17 than in line 6.)
#    Thanks, Stephane Chazelas.


if [ "$a" -eq 98 ] || [ "$b" -eq 47]
then
   echo "Test #2 succeeds."
else
   echo "Test #2 fails."
fi


#  The -a and -o options provide
#+ an alternative compound condition test.
#  Thanks to Patrick Callahan for pointing this out.

if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
   echo "Test #3 succeeds."
else
   echo "Test #3 fails."
fi
```

```
if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Test #4 succeeds."
else
  echo "Test #4 fails."
fi


a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
    echo "Test #5 succeeds."
else
    echo "Test #5 fails."
fi

exit 0
```

`&&` ve `||` operatörleri de bir aritmetik bağlamda kullanım alanı bulmaktadır.

```
bash$  echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 ||0))
1 0 1 0
```

çeşitli operatörler

,

Virgül operatörü

Virgül operatörü iki veya daha fazla aritmetik işlemi birbiri ardı sıra işleme koyar. Tüm işlemler (olası yan etkileri ile) değerlendirilir, ancak son işlem döndürülür.

```
let "t1 = ((5+3, 7 - 1, 15 - 4))"
echo "t1= $t1"                          # t1 = 11

let "t2 = ((a=9, 15 / 3))"              # Set "a" and calculate "t2".
echo "t2 = $t2   a= $a"                 # t2 = 5    a = 9
```

Virgül operatörü çoğunlukla döngüler içinde kullanılır.

8.2 NÜMERİK SABİTLER

Bir sayının özel bir öneği veya sembol sistemi yoksa, kabuk betiği o sayıyı ondalık tabanda (10'luk sistem) yorumlar. Örneğin, 0'dan sonra gelen bir sayı sekizli sayı sistemindedir (8 tabanı). 0x sonrasında gelen bir sayı onaltılı sayı sistemindedir (16 tabanı). # içine gömülmüş bir sayı TABAN#NUMARA olarak değerlendirilir (Bu seçeneğin kullanılırlığı, aralık kısıtlamaları nedeniyle sınırlıdır).

ÖRNEK 8.4 NÜMERİK SABİTLERİN TEMSİL EDİLMESİ

```
#!/bin/bash
#  numbers.sh: Representation of numbers.
#  Decimal
let "dec = 32"
echo "decimal number = $dec"
#  Nothing out of the ordinary here.

#  Octal: numbers preceded by ' 0'  (zero)
let "oct = 071"
echo "octal number = $oct"
#  Expresses result in decimal.

#  Hexadecimal: numbers preceeded by ' 0x' or ' 0X'
let "hex = 0x7a"
echo "hexadecimal number = $hex"
# Expresses result in decimal.
#  Other bases: BASE#NUMBER
#  BASE between 2 and 64.

let "bin = 2#111100111001101"
echo "binary number = $bin"
```

```
let “b32 = 32#77”

echo “base-32 number = $b32”


let “b64 = 64#@_”

echo “base-64 number = $b64”

#

#  This notation only works for a limited range (2 – 64)

#  10 digits + 26 lowercase characters + 26 uppercase characters + @ + _

echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))


#  Important note:

#   Using a digit out of range of the specified base notation

#+  will give an error message.


let “bad_oct = 081”

# numbers.sh: let: oct = 081: value too great for base (error token is
“081”)

#          Octal numbers use only digits in the range of 0 – 7.


exit 0

# Thanks, Rich Bartell and Stephane Chazelas, for clarification.
```

KISIM 3 ORTA DÜZEY KONULAR

BÖLÜM 9 DEĞİŞKENLER (yeniden)

9.1 İÇ DEĞİŞKENLER

Yerleşik değişkenler

Bash betik davranışını etkileyen değişkenlerdir.

`$BASH`

Bash ikilisinin bulunduğu yoldur.

`$BASH_ENV`

Bir Bash başlangıç dosyasına işaret eden ve bir betik çalıştırıldığında okunacak olan çevre değişkenidir.

`$BASH_VERSINFO[n]`

Sisteme yerleştirilmiş olan Bash sürümü hakkında sürüm bilgisi içeren 6 elemanlı bir dizidir. `$BASH_VERSION`'a benzer, ama biraz daha detaylısıdır.

```
#  Bash version info:
for n in 0 1 2 3 4 5
do
    echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done
```

```
# BASH_VERSINFO[0] = 2                      # Major version no.
# BASH_VERSINFO[1] = 05                     # Minor version no.
# BASH_VERSINFO[2] = 8                      # Patch level.
```

```
# BASH_VERSINFO[3] = 1                                  # Build version.
# BASH_VERSINFO[4] = release                            # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu              # (same as $MACHTYPE).
```

`$BASH_VERSION`

Sisteme yerleştirilmiş olan Bash sürümüdür.

```
bash$ echo $BASH_VERSION
2.04.12(1) –release

tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

`$BASH_VERSION` değişkeninin değerini kontrol ederek hangi kabuğun çalıştığını belirleyebilirsiniz.

`$DIRSTACK`

Dizin yığını içindeki en üst pozisyonda bulunan değerdir (`pushd` ve `popd` tarafından etkilenir).

Bu yerleşik değişken, `dirs` komutuna karşılık gelir, ancak `dirs` dizin yığınının tüm içeriğini gösterir.

`$EDITOR`

Betiğin çağırdığını varsaydığımız editor'dür, `vi` veya `emacs`.

`$EUID`

"etkin" kullanıcı kimliği numarasıdır.

`su` dahil, geçerli kullanıcının varsayılan kimlik numarasıdır.

Unutmayınız ki, `$EUID` muhakkak ki `$UID` ile aynı değildir.

`$FUNCNAME`

Geçerli fonksiyonun adıdır.

```
xyz23 ()
{
   echo "$FUNCNAME now executing."   # xyz23 now executing.
}
xyz23
echo "FUNCNAME = $FUNCNAME"
```

`$GLOBIGNORE`

Dosya eşleştirmesinin dışında kalan dosya-adı listesidir.

`$GROUPS`

Geçerli kullanıcının ait olduğu gruplardır.

Geçerli kullanıcı için `/etc/passwd` dosyasında kayıtlı, grup kimlik numaralarının listesidir (dizisi).

```
root# echo $GROUPS
0

root# echo ${GROUPS[1]}
1

root# echo ${GROUPS[5]}
6
```

`$HOME`

Genellikle, kullanıcının ana dizinidir, `/home/username` (Bkz.Örnek 9.12)

`$HOSTNAME`

`hostname` komutuyla, sistem açılışında çalışan bir `init` betiği içinde, sistem adı atanır.

`$HOSTTYPE`

Ana bilgisayar türüdür.

`$MACHTYPE` değişkeni gibi, sistem donanımı tanımlar.

```
bash$ echo $HOSTTYPE
i686
```

`$IFS`

Girdi (input) alanı ayıracıdır.

Bu varsayılan beyaz boşluk karakterlerine (boşluk, sekme, ve satır sonu) denk düşer, ancak örneğin virgülle ayrılmış bir veri dosyasını ayrıştırmak için de kullanılabilir. Unutmayınız ki, `$IFS` değişkeni ilk karakter olarak `$*` tutar. Bkz. Örnek 6.1.

```
bash$ echo $IFS | cat –vte
$

bash$ bash –c ' set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```

`$IFS` boşluk karakterini diğer karakterlerden farklı olarak ele alır.

ÖRNEK 9.1 $IFS ve BOŞLUK KARAKTERİ

```
#!/bin/bash

# $IFS treats whitespace differently than other characters.

output_args _one_per_line()

{

     for arg

     do echo “[$arg]”

     done

}

echo; echo “IFS=\” \””

echo “-------“


IFS=” “

var=”a  b c    “

output _args_one_per_line $var     #  output_args_one_per_line ‘ echo “ a  b
c   “’

#

# [a]

# [b]

# [c]


echo; echo “IFS=:”

echo “-----“


IFS=:

var=”:a::b:c:::”                   # Same as above, but substitute “:” for
“ “.

output_args_one_per_line $var

#

# []

# [a]
```

```
# []

# [b]

# [c]

# []

# []

# []

# The same thing happens with the "FS" field operator in awk.

# Thank you, Stephane Chazelas.


echo

exit 0

(Thanks, S.C. for clarification and examples.)
```

`$IGNOREEOF`

`EOF` karakterini yoksay: Oturum kapatılmadan önce, kabuk kaç adet dosya-sonunu (`control-D`) yok sayacak?

`$LC_COLLATE`

Değeri genellikle `.bashrc` veya `/etc/profile` dosyalarında atanan bu değişken, dosya adı açılımlarını ve joker karakter eşleşmelerini okur. Yanlış ya da bilmeyerek kullanılması beklenmeyen sonuçlara neden olur.

Bash sürümü 2.05 itibariyle, dosya adı açılımında, artık parantez içindeki karakter aralıklarında küçük-büyük harf ayırımı yapılmamaktadır. Örneğin, `ls [A-M]*` hem `Dosya1.txt` ve `dosya1.txt`'e karşılık gelir. Parantez eşleştirmesinde alışılmış davranışı elde etmek için `/etc/profile` ve `/` veya `~/.bashrc` içerisinde şu atamayı yapılmalıdır: `export LC_COLLATE=C`

`$LC_CTYPE`

Bu iç değişken, dosya adı açılımı ve joker karakter eşleştirmesi sırasında geçen karakterleri yorumlamada kullanılır.

`$LINENO`

Bu değişken, içinde bulunduğu kabuk betiğinin satır numarasını verir. Sadece içinde geçtiği komut dosyasında önemi vardır, ve esas olarak hata ayıklama amacıyla kullanılır.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_       # Save it.
echo "At line number $LINENO, variable \"v1\" = $v1"
echo "Last command argument processed = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

`$MACHTYPE`

Makine türüdür.

Sistem donanımını tanımlar.

```
bash$ echo $MACHTYPE
1686-debian-linux-gnu
```

`$OLDPWD`

Eski çalışma dizinidir ("ESKİ-baskı-çalışma-dizini", bulunduğunuz önceki dizindir).

`$OSTYPE`

İşletim sistemi türüdür.

```
bash$ echo $OSTYPE
linux-gnu
```

`$PATH`

İkililerin bulunduğu yolu verir, çoğunlukla `/usr/bin/, /usr/X11R6/bin/, /usr/local/bin` vb.

Bir komut verildiğinde, kabuk otomatik olarak, yürütülebilir için tanımlı bulunan yoldaki dizinler arasında bir karma (hash) tablo araması yapar. Yol `$PATH` adında bir çevre değişkeninde saklanır, bu değişken iki nokta üst üste ile ayrılmış ve dizin adlarının bulunduğu bir listedir. `$PATH` değişkeninin tanımı, normal olarak, sistemde `/etc/profile` ve `/ ` veya `~/.Bashrc`'de saklanır.

```
bash$ echo $PATH

/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

`PATH=${PATH}:/opt/bin` geçerli dizinin sonuna `/opt/bin` dizinini ekler. Bir komut dosyasında, geçici olarak bu şekilde bir dizini yola eklemek mümkündür. Komuttan çıkıldığında orijinal `$PATH` geri saklanır (bir alt süreç (betik) ana sürecin (kabuk gibi) çevresini değiştiremez).

Geçerli çalışma dizini ./ genellikle `$PATH` tarafından güvenlik önlemi olması açısıyla atlanır.

`$PIPESTATUS`

Son çalıştırılan borunun çıkış durumu. İlginçtir ki, bu değişken son çalıştırılan komutun çıkış durumunu vermez.

```
bash$ echo $PIPESTATUS

0


bash$ ls –al | bogus_command

bash: bogus_command: command not found

bash$ echo $PIPESTATUS

141


bash$ ls –al | bogus_command

bash: bogus_command: command not found

bash$ echo $pipestatus

141


bash$ ls –al | bogus_command

bash: bogus_command: command not found

bash$ echo $?

127
```

`$PPID`

Bir sürecin `$PPID`'si üst sürecin süreç kimliği `(pid)` 'dir. [1]

`pidof` komutu ile karşılaştırınız.

`$PS1`

Komut satırında görülen ana istemdir.

`$PS2`

Ek girdi beklendiği zaman görülen ikincil istemdir. `">"` ile gösterilir.

`$PS3`

`select` döngüsü içinde görüntülenen üçüncül istemdir (bkz.Örnek 10.28).

`$PS4`

`-X` seçeneği ile bir komut dosyası başlatılırken her çıktı satırının başında gösterilen dördüncül istemdir. `"+"` olarak gösterilir.

`$PWD`

Şu anda bulunduğunuz çalışma dizinidir.

Bu `pwd` yerleşik komutuna benzerdir.

```
#!/bin/bash
E_WRONG_DIRECTORY=73
clear     #  Clear screen.
TargetDirectory=/home/bozo/projects/GreatAmericanNovel
cd $TargetDirectory
echo "Deleting stale files in $TargetDirectory."
if [ "$PWD" != "$TargetDirectory" ]
then    # Keep from wiping out wrong directory by accident.
   echo "Wrong directory!"
```

```
    echo "In $PWD, rather than $TargetDirectory!"
    echo "Baling out!"
    exit $E_WRONG_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*           # Delete dotfiles.
# rm -f .[^.]* ..?* to remove filenames beginning with multiple dots.
# (shopt -s dotglob; rm -f *)   will also work.
# Thanks, S.C. for pointing this out.

# Filenames may contain all characters in the 0 - 255 range, except "/".
# Deleting files beginning with weird characters is left as an exercise.
# Various other operations here, as necessary.
echo
echo "Done."
echo "Old files deleted in $TargetDirectory."
echo

exit 0
```

`$REPLY`

Bir değişkenin `read` ile okunmayan ve varsayılan değerini tutar. Menüleri seçmek için de kullanılır, ancak seçilen değişkenin nesne numarasını tutar, değerini tutmaz.

```
#!/bin/bash
echo
echo -n "What is your favorite vegetable?"
read
```

```
echo “Your favorite vegetable is $REPLY.”
# REPLY holds the value of last “read” if and only if
# no variable supplied.

echo
echo
echo –n “What is your favorite fruit? “
read fruit

echo”Your favorite fruit is $fruit.”
echo “but . . .”
echo “Value of \$REPLY is stil $REPLY.”
#  $REPLY is still set to its previous value because
#  the variable $fruit absorbed the new “read” value.
echo

exit 0
```

`$SECONDS`

Komut dosyasının kaç saniye çalıştığını saklar.

```
#!/bin/bash
ENDLESS_LOOP=1
INTERVAL=1

echo
echo “Hit Control-C to exit this script.”
echo
```

```
while [ $ENDLESS_LOOP ]
do
    if [ "$SECONDS" -eq 1 ]
    then
        units=second
    else
        units=seconds
    fi
    echo "This script has been running $SECONDS $units."
    sleep $INTERVAL
done

exit 0
```

`$SHELLOPTS`

Etkin kabuk seçenekleri listesi, salt okunur bir değişkendir.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

`$SHLVL`

Bash hangi kabuk düzeyinde yuvalanmıştır? Komut satırında, `$SHLVL 1` ise, bir komut dosyası içinde bu değer 2'ye çıkacaktır.

`$TMOUT`

`$TMOUT` çevre değişkeni sıfır olmayan bir *zaman* değerine ayarlanmış ise, kabuk istemi *zaman* saniyesi geçtikten sonra zaman aşımına uğrayacaktır. Bu da oturumdan çıkışa (logout) neden olur.

Ne yazık ki bu, sadece kabuk komut istemi konsol veya bir `xterm`'den girdi (input) beklerken çalışır. Zamanlamalı giriş için bu iç değişkeni düşünmek yerinde olurdu, ama örneğin okuma `(read)` ile birlikte, `$TMOUT` bu bağlamda çalışmaz ve kabuk betikleri için neredeyse kullanışsızdır. (`ksh` sürümünde zamanlanmış okumanın çalıştığı bildirilmiştir.)

Zamanlanmış girdiyi bir komut dosyasında uygulamaya koymak kesinlikle mümkündür, ama karmaşık düzenlemeler gerektirir. Bunu sağlayacak yöntemlerden biri de, komut dosyası zaman aşımına uğradığı zaman sinyal veren bir zamanlama döngüsü ayarlamaktır. Bu yaklaşım, zamanlama döngüsü tarafından üretilen kesmeyi yakalayacak (Örnek: 30-5'e bakınız) bir sinyal işleme rutinini de gerektirir.

ÖRNEK 9.2 ZAMANLAMALI GİRDİ

```
#!/bin/bash

# timed-input.sh

# TMOUT=3            useless in a script

TIMELIMIT=3   # Three seconds in this instance, may be set to different
value.

PrintAnswer()

{

  if [ "$answer" = TIMEOUT ]

  then

   echo $answer

  else       # Don't want to mix up the two instances.

    echo "Your favorite veggie is $answer"

    kill $!  # Kills no longer needed TimerOn function running in
background.

             # $! is PID of last job running in background.

  fi

}
```

```
TimerOn()
{
  sleep $TIMELIMIT && kill –s 14 $$ &
  # Waits 3 seconds, then sends sigalarm to script.
}
Int14Vector()
{
   answer="TIMEOUT"
   PrintAnswer
   exit 14
}

trap Int14Vector 14   # Timer interrupt (14) subverted for our purposes.
echo "What is your favorite vegetable "
TimerOn
read answer
PrintAnswer
#  Admittedly, this is a kludgy implementation of timed input,
#+ however the "-t" option to "read" simplifies this task.
#  See "t-out.sh", below.
#  If uou need something elegant…
#+ consider writing the application in C or C++,
#+ using appropriate library functions, such as ' alarm' and ' setitimer'.
exit 0
```

Alternatif olarak `stty` kullanılır.

ÖRNEK 9.3 ZAMANLANMIŞ GİRDİ (yeniden)

```
#!/bin/bash
# timeout.sh

# Written by Stephane Chazelas,
# and modified by the document author.
INTERVAL=5                # timeout interval

timedout_read() {
  timeout=$1
  varname=$2
  old_tty_settings='stty -g'
  stty -icanon min 0 time ${timeout}0
  eval read $varname     # or just     read $varname
  stty "$old_tty_settings"
  # See man page for "stty"
}
echo; echo -n "What's your name? Quick! "
timedout_read $INTERVAL your_name

# This may not work on every terminal type.
# The maximum timeout depends on the terminal.
# (it is often 25.5 seconds).

echo
if [ ! -z "$your_name" ]   # If name input before timeout. . .
then
   echo "Your name is $your_name."
else
```

```
   echo "Timed out."
fi

echo
# The behavior of this script differs somewhat from "timed-input.sh".
# At each keystroke, the counter resets.

exit 0
```

Belki de en basit yöntem okumak (`read`) için `-t` seçeneğini kullanmaktır.

ÖRNEK 9.4 ZAMANLI OKUMA

```
#!/bin/bash
# t-out.sh (per a suggestion by "syngin seven)

TIMELIMIT=4         # 4 seconds
read –t $TIMELIMIT variable <&1

echo
if [ -z "$variable" ]
then
  echo "Timed out, variable stil unset."
else
  echo "variable = $variable"
fi
exit 0
```

`$UID`

Kullanıcı kimlik numarasıdır.

`/etc/passwd` dosyasında kayıtlı olan, geçerli kullanıcının kullanıcı kimlik numarasıdır.

Bu, `su` yoluyla geçici olarak başka bir kimlik üstlenmiş olsa bile, geçerli kullanıcı'nın gerçek kimliğidir. `$UID` komut satırından veya bir komut dosyası içinden değiştirilemeyen, salt okunur bir değişkendir, ve `id` yerleşik değişkeninin karşılığıdır.

ÖRNEK 9.5 BEN KÖK MÜYÜM?

```
#!/bin/bash
# am-i-root.sh    Am I root or not?

ROOT_UID=0   #  Root has $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]   # Will the real root please stand up?
then
  echo "You are root."
else
  echo "You are just an ordinary user (but mom loves you just the same)."
fi

exit 0

# ============================================================== #
# Code below will not execute, because the script already exited.
# An alternate method of getting to the root of matters:

ROOTUSER_NAME=root
username='id -nu'              # Or...  username='whoami'
```

```
if  [ "$username" = "$ROOTUSER_NAME" ]
then
  echo "Rooty, toot, toot. You are root."
else
  echo "You are just a regular fella."
fi
```

Bkz. Örnek 2.2.

`$ENV`, `$LOGNAME`, `$MAIL`, `$TERM`, `$USER`, ve `$USERNAME` değişkenleri birer Bash yerleşiği değildir. Bunlar, çoğu kez Bash başlangıç dosyalarından birinde çevre değişkeni olarak ayarlanır. `$SHELL`, kullanıcının oturum açma kabuğunun adıdır, `/etc/passwd`'den veya "`init`" komut dosyasından ayarlanabilir, ve bu değişken de Bash yerleşiği değildir.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt
bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Konum Parametreleri

`$0, $1, $2`, vb.

Konum parametreleridir. Komut satırından betiğe, bir fonksiyona geçirilir, veya bir değişkene ayarlanır. (bkz. Örnek 5-5 ve Örnek 11-11)

`$#`

Komut satırı argümanlarının sayısı [2] veya konum parametrelerini verir. (bkz. Örnek 34.2)

`$*`

Tek bir sözcük (word) olarak görülen konumsal parametrelerin tümünü verir.

`$@`

`$*` ile aynıdır, ancak her parametre yorumsuz ve değişikliğe uğramadan geçirilen birer karakter dizgesidir. Bunun anlamı şudur: Argüman listesi içindeki her parametre ayrı bir sözcük (word) olarak görülür.

ÖRNEK 9.6 arglist: ARGÜMANLARI $* ve $@ ile GÖRÜNTÜLEME

```
#!/bin/bash
# Invoke this script with several arguments, such as “one two three”.
E_BADARGS=65
if [ ! –n “$1” ]
then
  echo “Usage: ‘ basename $0’ argument1 argument2 etc”
  exit $E_BADARGS
fi
```

```
echo

index=1

echo "Listing args with \"\$*\":"

for arg in "$*"   # Doesn't work properly if "$*" isn't quoted.

do

  echo "Arg #$index  = $arg"

  let "index+=1"

done              # $* sees all arguments as single word.

echo "Entire arg list seen as single word."


echo

index=1

echo "Listing args with \"\$@\":"

for arg in "$@"   do

  echo "Arg #$index  = $arg"

  let "index+=1"

done             # $@ sees arguments as separate words.

echo "Arg list seen as separate words."


echo

exit 0
```

Kaydırma (`shift`) komutunun ardından `$1` kaybolacaktır, `$@` kalan komut satırının kalan parametrelerini verir.

```
#!/bin/bash

# Invoke with ./scriptname 1 2 3 4 5


echo "$@"      # 1 2 3 4 5

shift

echo "$@"      # 2 3 4 5
```

```
shift
echo "$@"      # 3 4 5

# Each "shift" loses parameter $1.
# "$@" then contains the remaining parameters.
```

`$@` özel parametresi kabuk betiklerine girdi filtrelemek için bir araç olarak kullanım bulur. `$@` yapısı, `stdin`'den veya komut için parametre olarak verilen dosyalardan betiğe girdi `(input)` kabul eder. (bkz. Örnek 12-17 ve 12-18)

`$IFS` ayarına bağlı olarak `$*` ve `$@` parametreleri bazen tutarsız ve şaşırtıcı davranışlar sergiler.

ÖRNEK 9.7 TUTARSIZ $* ve $@ DAVRANIŞI

```
#!/bin/bash

# Erratic behavior of the "$*" and "$@" internal Bash variables,
# depending on whether these are quoted or not.
# Word splitting and linefeeds handled inconsistently.
# This example script by Stephane Chazelas,
# and slightly modified by the document author.

Set – "First one* "second" *third:one" "" "Fifth: :one"
# Setting the script arguments, $1, $2, etc.

echo
echo ' IFS unchanged, using "$*"'
c=0
for i in "$*"               # quoted
do echo "$((c+=1)): [$i]"   # This line remains the same in every instance.
```

```
                                # Echo args.
done
echo ---

echo ‘ IFS unchanged, using $*’
c=0
for i in $*                     # unquoted
do echo “$((c+=1)): [$i]”
done
echo ---

echo ‘ IFS unchanged, using $@’
c=0
for i in $@
do echo “$((c+=1)): [$i]”
done
echo ---

IFS=:
echo ‘ IFS=”:”, using “$*”’
c=0
for i in “$*”
do echo “$((c+=1)): [$i]”
done
echo ---

echo ‘ IFS=”:”, using $*’
c=0
for i in $*
do echo “$((c+=1)): [$i]”
```

```
done
echo ---

var=$*
echo ' IFS=":", using "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo ' IFS=":", using $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo ' IFS=":", using $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo ' IFS=":", using $var (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
```

```
done
echo ---


echo ‘ IFS=":", using "$@"’
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo ‘ IFS=":", using $@’
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

echo ‘ IFS=":", using $var (var=$@)’
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo ‘ IFS=":", using $var (var="$@")’
c=0
for i in "$var"
```

```
do echo "$((c+=1)): [$i]"
done
echo ---

echo ' IFS=":", using $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Try this script with ksh or zsh -y.

exit 0
```

@$ ve $* parametreleri, yalnızca çift tırnak arasındayken farklılık gösterir.

ÖRNEK 9.8 $IFS DEĞERİ BOŞKEN $* ve $@

```
#!/bin/bash
# If $IFS set, but empty,
# then "$*" and "$@" do not echo positional params as expected.

mecho()         # Echo positional parameters.
{
   echo "$1,$2,$3";
}
```

```
IFS=""          # Set, but empty.
set a b c       # Positional parameters.

mecho "$*"      # abc,,
mecho $*        # a,b,c

mecho $@        # a,b,c
mecho "$@"      # a,b,c

# The behavior of $* and $@ when $IFS is empty depends
# on whatever Bash or sh version being run.
# It is therefore inadvisable to depend on this "feature" in a script.
# Thanks, S.C.

exit 0
```

Diğer Özel Parametreler

`$-`

Betiğe aktarılan bayraklardır.

Bu aslında Bash'e uyarlanan bir `ksh` yapısıdır ve ne yazık ki Bash betiklerinde güvenilir bir şekilde çalışmıyorlar. Olası bir kullanımı, bir betiğin etkileşimli olup olmadığını kendi kendine test etmesidir.

`$!`

Arka planda çalışan son işe ait `PID` (işlem kimliği) dir.

`$_`

Bir önceki çalıştırılan komutun son argümanını saklayan özel değişkendir.

ÖRNEK 9.9 ALT ÇİZGİ DEĞİŞKENİ

```
#!/bin/bash

echo $_                  # /bin/bash

                  # Just called      /bin/bash to run the script.

du >/dev/null     # So no output from command.

echo $_           # du

ls –al >/dev/null       # So no output from command.

echo $_           # -al   (last argument)

.

.

echo $_           # :
```

$?

Bir komut, fonksiyon veya komut dosyasının çıkış durumudur (bkz. Örnek 23.3).

$$

Betiğin kendi işlem kimliğidir, genellikle "tekil" dosya adları oluşturmak için kullanılır (bkz. Örnek A.13, Örnek 30.6, Örnek 12.23, ve Örnek 11.20).

Notlar

[1] Şu anda çalışan komut dosyasına ait PID, tabii ki $$ ile verilir.

[2] "Argüman" ve "parametre" kelimeleri sık sık birbirinin yerine kullanılmaktadır. Bu doküman bağlamında bir betiğe ya da fonksiyona geçirilen değişken anlamında ve tamamen eş anlamda kullanılmıştır.

9.2. Karakter Dizelerini Değiştirme

Bash pek çok dize işleme operasyonlarını destekler. Ne yazık ki, bu araçlar tek bir odakta birleşmemiştir. Bazıları parametre değiştirmenin bir alt kümesi olup, diğerleri UNIX `expr` komutunun işlevselliğini altında yer almaktadır. Bu zihin karışıklığını bir kenara bırakın, tutarsız komut sözdizimi ve işlevselliğin örtüşmesine neden olur.

Bash karakter dizesi için ayrılmış olan işlemler için şaşırtıcı derecede destek sağlar. Ne yazık ki, bu araçlar tek odakta birleşmekten şu anda yoksundur. Bazıları, parametrelerin birbirleri yerine konması, veya bunun alt-kümesidir. Ve diğerleri UNIX expr komutunun işlevselliği altında incelenir. Bunun sonucu olarak, tutarsız komut sözdizimi ve işlevselliğin örtüşmesi sayılabilir. Kafa karışıklığı da bir diğer sayılması gereken ve istenmeyen sonuçtur.

Karakter Dizesi Uzunluğu

```
${#string}


expr length $string


expr "$string" : '.*'
```

```
stringZ=abcABC123ABCabc


echo ${#stringZ}                 # 15
echo ‘ expr length $stringZ’       # 15
echo ‘ expr “$stringZ” : ‘ .*’’     # 15
```

Dizenin Başında Eşleşen Altdize Uzunluğu

```
expr match "$string" '$substring'
```

`$substring` bir sıralı ifadedir.

```
expr "$string" : '$substring'
```

`$substring` bir sıralı ifadedir.

```
stringZ=abcABC123ABCabc
#       |------|
echo ‘ expr match “$stringZ” ‘ abc[A-Z]*.2’’      # 8
echo ‘ expr “$stringZ” : ‘ abc[A-Z]*.2’’          # 8
```

İndis

```
expr index $dize $altdize
```

`$dize` ile eşleşen `$altdize`'nin ilk karakterinin sayısal konumudur.

```
stringZ=abcABC123ABCabc
echo ‘ expr index “$stringZ” C12’          # 6
# C position.
echo ‘ expr index “$stringZ” 1c’                # 3
# ‘ c’ (in #3 position) matches before ‘ 1’.
```

Bu `C`'deki `strchr()` fonksiyonuna yakın eşdeğerdir.

Karakter Dizesi İçinden Alıntı Almak

`${dize: konum}`

`$dize`'nin `$konum` pozisyonundan başlayarak sonrasındaki tüm karakterleri alır ve döndürür.

`$dize` parametresi "*" veya "@" ise, `$konum` pozisyonundan başlayan konumsal parametreleri [1] verir.

`${dize:konum:uzunluk}`

`$dize`'nin `$konum` pozisyonundan başlayarak sonrasındaki `uzunluk` sayısı kadar karakteri alır ve döndürür.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       0-based indexing.
echo ${stringZ:0}            # abcABC123ABCabc
echo ${stringZ:1}            # bcABC123ABCabc
echo ${stringZ:7}            # 23ABCabc

echo ${stringZ:7:3}          # 23A
# Three characters of substring
```

`$dize` parametresi `"*"` veya `"@"` ise, `$konum` pozisyonundan başlayarak sonrasındaki en fazla `$uzunluk` sayısı kadar olan konumsal parametreyi alır ve döndürür.

```
echo ${*:2}       # Echoes second and following positional parameters.
echo ${@:2}       # Same as above.
echo ${*:2:3}     # Echoes three positional parameters, starting at second.
```

```
expr substr $dize $konum $uzunluk
```

`$dize` içinde `$konum` pozisyonundan başlayarak sonrasındaki `$uzunluk` sayısı kadar karakteri sıralamak için kullanılır.

```
stringZ=abcABC123ABCabc
#       123456789......
#       1-based indexing.
echo ‘ expr substr $stringZ 1 2’
echo ‘ expr substr $stringZ 4 3’
```

```
expr match "$dize" '\($altdize\)'
```

`$dize` karakter dizesinin başından başlayarak `altdize` sıralı ifadesine uyan karakterleri sıralamak için kullanılır.

```
expr "$dize" : '\($altdize\)'
```

`$dize` karakter dizesinin başından başlayarak `altdize` sıralı ifadesine uyan karakterleri sıralamak için kullanılır.

```
stringZ=abcABC123ABCabc
#       ========
echo ‘ expr match “$stringZ” ‘ \(.[b-c]*[A-Z]..[0-9]\)’     # abcABC1
echo ‘ expr “$stringZ” : ‘ \(.[b-c]*[A-Z]..[0-9]\)’’        # abcABC1
echo ‘ expr “$stringZ” : ‘ \(…….\)’’                    # abcABC1
# All of the above forms give an identical result.
```

```
expr match "$dize" '.*\($altdize\)'
```

`$dize` karakter dizesinin sonundan başlayarak `$altdize` sıralı ifadesine uyan karakterleri sıralamak için kullanılır.

```
expr "$dize" : '.*\($altdize\)'
```

`$dize` karakter dizesinin sonundan başlayarak `$altdize` sıralı ifadesine uyan karakterleri sıralamak için kullanılır.

```
stringZ=abcABC123ABCabc
#               ======
echo ‘ expr match “$stringZ” ‘ .*\([A-C][A-C][A-C][a-c]*\)’’       # ABCabc
echo ‘ expr “$stringZ= : ‘ .*\(……\)’’
```

Karakter Alt Dizelerini Ortadan Kaldırma

```
${dize#altdize}
```

`$dize` dizesinin ilk karakterinden başlayarak `$altdize` karakter dizesi ile eşleşen en uzun dizeleri yok eder.

```
${dize##altdize}
```

`$dize` dizesinin ilk karakterinden başlayarak `$altdize` karakter dizesi ile eşleşen en kısa dizeleri yok eder.

```
stringZ=abcABC123ABCabc
#       |----|
#       |----------|
echo ${stringZ#a*C}                 # 123ABCabc
# Strip out shortest match between ‘ a’ and ‘ C’.
echo ${stringZ##a*C}          # abc
# Strip out longest match between ‘ a’ and ‘ C’.
```

`${dize%altdize}`

`$dize` dizesinin son karakterinden başlayarak `$altdize` karakter dizesi ile eşleşen en kısa kısımları yok eder.

`${dize%%altdize}`

`$dize` dizesinin son karakterinden başlayarak `$altdize` karakter dizesi ile eşleşen en kısa kısımları yok eder.

```
stringZ=abcABC123123ABCabc
#                       ||
#        |---------------|

echo ${stringZ%b*c}                 # abcABC123ABCa
# Strip out shortest match between ‘ b’ and ‘ c’, fromback of
$stringZ.

echo ${stringZ%%b*c}          # a
# Strip out longest match between ‘ b’ and ‘ c’, fromack of $stringZ.
```

ÖRNEK 9.10 DOSYA DEĞİŞİKLİĞİ İLE, GRAFİK DOSYA FORMATLARINI DÖNÜŞTÜRME

```
#!/bin/bash

#  cvt.sh:

#  Converts all the MacPaint image files in a directory to "pbm" format.

#  Uses the "macptopbm" binary from the "netpbm" package,

#+ which is maintained by Brian Henderson (bryanh@giraffe-data.com).

#  Netpbm is a standard part of most Linux distros.

OPERATION=macptopbm

SUFFIX=pbm               # New filename suffix.

if [ -n "$1" ]

then

      directory=$1       # If directory name given as a script argument...

else

      directory=$PWD     # Otherwise use current working directory.

fi


#  Assumes all the files in the target directory are MacPaint image files,

#+ with a ".mac" suffix.

for file in $directory/*      # Filename globbing.

do

      filename=$  {file%.*c}  #  Strip ".mac" suffix off filename

                              #+ (' .*c' matches everything

                              #+ between ' .' and ' c', inclusive).

      $OPERATION $file > $filename.$SUFFIX

                              #  Redirect conversion to new filename.

      rm -f $file             #  Delete original to new filename.

      echo "$filename.$SUFFIX" # Log what is happening to stdout.

done

exit 0
```

`${dize/altdize/yerdeğiştirme}`

`$dize` dizesi içinde kalan *`$altdize`* karakter dizesinin ilk eşleşmesini *`$yerdeğiştirme`* ile yerdeğiştirir.

`${dize//altdize/yerdeğiştirme}`

`$dize` dizesi içinde kalan *`$altdize`* karakter dizesinin her eşleşmesini *`$yerdeğiştirme`* ile yerdeğiştirir.

```
stringZ=abzABC123ABCabc
echo ${stringZ/abc/xyz} # xyzABC123ABCabc
                        # Replaces first match of ‘ abc’ with ‘ xyz’.
echo ${stringZ//abc/xyz}      # xyzABC123ABCxyz
                        # Replaces all matches of ‘ abc’ with # ‘ xyz’.
```

`${dize/#altdize/yerdeğiştirme}`

`$dize` dizesinin sol baş tarafından itibaren `$altdize` eşleşmesi varsa `$altdize` `$yerdeğiştirme` ile yerdeğiştirir.

`${dize/%altdize/yerdeğiştirme}`

`$dize` dizesinin sağ baş tarafından itibaren `$altdize` eşleşmesi varsa `$altdize` `$yerdeğiştirme` ile yerdeğiştirir.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ} # xyzABC123ABCabc

                         # Replace front-end match of ‘ match’ with ‘ xyz’.


echo ${stringZ/%abc/XYZ} # abcABC123ABCXYZ

                         # Replaces back-end match of ‘ abc’ with ‘ xyz’.
```

9.2.1 awk KULLANARAK KARAKTER DİZELERİNİ DEĞİŞTİRME

Bir Bash betiği yerleşik düzenine alternatif olarak awk karakter dizesi işlemlerini çağırarak karakter değiştirme yapabilir.

ÖRNEK 9.11 ALTDİZGELERİN SEÇİLİP ALINMASI İÇİN ALTERNATİF YOLLAR

```
#!/bin/bash
# substring-extraction.sh

String=23skidoo1
#      012345678  Bash
#     123456789   awk
# Note different string indexing system:
# Bash numbers first character of string as ‘ 0’.
# Awk numbers first character of string as ‘ 1’.

echo ${String:2:4}                  #  position 3 (0-1-2), 4 characters long
                             # skid


# The awk equivalent of ${string:pos:length} is substr(string,pos,length).
echo | awk ‘
{ print substr(“’”${String}”’”,3,4)      # skid
}

‘
#  Piping an empty “echo” to awk gives it dummy input,
#+ and thus makes it unnecessary to supply a filename.

exit 0
```

9.2.2 DAHA FAZLASI

Karakter dizesi değiştirme işleminin komut dosyası içinde yapılması ve expr komut listesinin ilgili bölümü için bkz. Bölüm 9.3.

Komut dosyası örnekleri için bkz.

Örnek 12.6
Örnek 9.13
Örnek 9.14
Örnek 9.15
Örnek 9.17

Notlar

[1] Komut satırı değişkenleri veya bir fonksiyona geçirilen parametreler için de geçerlidir.

9.3 PARAMETRELERİ YERİNE KOYMAK

Değişkenler ile oynamak ve/veya büyütmek

`${parametre}`

`$parametre` ile aynıdır, yani parametre değeri. Bazı durumlarda sadece *`${parametre}`* şekli daha belirgin olduğu için kullanılır.

Değişkenleri karakter dizeleri ile birleştirmek için de kullanılabilir.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin/      # Add /opt/bin to $PATH for duration of script.
echo "New \$PATH = $PATH"
```

`${parametre-varsayılan}`

Parametre değeri tespit edilmemişse, varsayılanı döndürür.

```
echo ${username-' whoami'}
# echoes the result of ' whoami', if variable $username is still unset.
```

Bu kullanım neredeyse *`${parametre:-varsayılan}`* kullanımına eşittir. Parametre bildirilmiş olup ve değeri de boş olduğu takdirde, fazladan yazılan iki nokta üst üste : farklı sonuç döndürür.

```
#!/bin/bash

username0=
#username0 has been declared, but is set to null.
echo "username0 = ${username0-' whoami'}"
# Will not echo.

echo "username1 = ${username1-' whoami'}"
# username1 has not been declared.
# Will echo.

username2=
# username2 has been declared, but is set to null.
echo "username2 = ${username2:-' whoami'}"
# Will echo because of :- rather than just - in condition test.

exit 0
```

${parametre=varsayılan}, ${parametre:=varsayılan}

Parametre değeri tespit edilmemişse, varsayılan değere atar.

Her iki kullanım da birbirine hemen hemen eşittir. *$parametre* bildirilmiş olup, değeri de boş olduğu takdirde [1], iki nokta üst üste : farklı sonuç döndürür.

```
echo ${username='whoami'}
# Variable "username" is now set to ' whoami'.
```

${parametre+alt_değer}, ${parametre:+alt_değer}

Parametre değeri tespit edildiyse, alt_değeri kullanır, aksi takdirde boş karakter dizesi yazdırır.

Her iki kullanım da birbirine hemen hemen eşittir. *$parametre* bildirilmiş olup ve değeri de boş olduğu takdirde iki nokta üst üste : farklı değer döndürür. Aşağıdaki örneğe bkz.

```
echo "###### \${parameter+alt_value} ########"
echo

a=${param1+xyz}
echo "a = $a"     # a =

param2=
a=${param2+xyz}
echo "a = $a"           # a = xyz

param3=123
a=${param3+xyz}
echo"a = $a"            # a = xyz

echo
echo "###### \${parametre:alternatif_değer} ########"
echo

a=${param4:+xyz}
echo "a = $a"           # a =

param5=
a=${param5:+xyz}
echo "a = $a"     # a =
# Different result from a=${param5+xyz}

param6=123
a=${param6+xyz}
echo "a= $a"            # a = xyz
```

${parametre?hata_msj},${parametre:?hata_msj}

Parametre değeri tespit edildiyse kullanır, aksi takdirde hata_mesajı yazdırır.

ÖRNEK 9.12 PARAMETRELERDE YERİNE GEÇMEYİ ve İKİ NOKTA ÜST ÜSTEYİ : KULLANMAK

```
#!/bin/bash

# Check some of the system's environmental variables.
#  If, for example, $USER, the name of the person at the console, is not
set,
#+ the machine will not recognize you.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Name of the machine is $HOSTNAME."
  echo "You are $USER."
  echo "Your home directory is $HOME."
  echo "Your mail INBOX is located in $MAIL."
  echo
  echo "If you are reading this message,"
  echo "critical environmental variables have been set."
  echo
  echo

# ------------------------------------------------------------------------
--------

#  The ${variablename?} construction can also check
#+ for variables set within the script.

ThisVariable=Value-of-ThisVariable
#  Note, by the way, that string variables may be set
#+ to characters disallowed in their names.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".

echo
echo

: ${ZZXy23AB?"ZZXy23AB has not been set."}
#  If ZZXy23AB has not been set,
#+ then the script terminates with an error message.

# You can specify the error message.
# : ${ZZXy23AB?"ZZXy23AB has not been set."}

# Same result with:           dummy_variable=${ZZXy23AB?}
```

```
#                       dummy_variable=${ZZXy23AB?"ZXy23AB has not been
set."}
#
#                       echo ${ZZXy23AB?} >/dev/null

echo "You will not see this message, because script terminated above."

HERE=0
exit $HERE   # Will *not* exit here.
```

Parametrelerin yer değiştirmesi ve/veya büyütülmesi. Aşağıdaki ifadeler **match** ***in*** **expr** dize işlemlerinin tamamlayıcısıdır (bkz. Örnek 12.6). Çoğunlukla bu belirli olanlar, dosya yollarının adlarını ayrıştırmak için kullanılır.

Değişken uzunluğu / Karakter alt dizesinin yok edilmesi

`${#d}`

`String length` (`$d` değişkenindeki karakter sayısı). Bir dizi için, **${# dizi}** dizideki ilk elemanın uzunluğu.

Aykırı Durumlar:

- ${#*} ve ${#@} konumsal parametrelerin sayısını verir.
- Bir dizi için, ${#dizi[*]} ve ${#dizi[@]}dizideki eleman sayısını verir.

ÖRNEK 9.13 BİR DEĞİŞKENİN UZUNLUĞU

```
#!/bin/bash

# length.sh

E_NO_ARGS=65

if [ $# -eq 0]    # Must have command-line args to demo script.

then

   echo "Invoke this script with one or more command-line arguments."

   exit $E_NO_ARGS

fi
```

```
var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Length of var01 = ${#var01}"

echo "Number of command-line arguments passed to script = ${#@}"
echo "Number of command-line arguments passed to script = ${#*}"

exit 0
```

${d#Desen}, ${d##Desen}

$d değişkeninden baştan başlayarak sona doğru, en kısa/en uzun $Desen eşleşmesini yok eder.

Örnek A.8 bir kullanım örneğini göstermektedir:

```
#  Function from "days-between.sh" example.
#  Strips leading zero(s) from argument passed.
strip_leading_zero ()   # Better to strip possible leading zero(s)
{                       # from day and/or month
   val=${1#0}           # since otherwise Bash will interpret them
   return $val          # as octal values (POSIX.2, sect 2.9.2.1).
}
```

Bir başka örnek kullanım:

```
echo ' basename $PWD'    # Basename of current working directory.
echo "${PWD##*/}"       # Basename of current working directory.
echo
echo ' basename $0'            # Name of script.
```

```
echo $0                  # Name of script.

echo "${0##*/}"          # Name of script.

echo

filename=test.data             # data

echo "${filename##*.}"     # Extension of filename.
```

${d%Desen}, ${d%%Desen}

$d değişkeninden sondan başlayarak başa doğru, en kısa/en uzun $Desen eşleşmesini yok eder.

Bash Sürüm 2 ek seçenekler ekler.

ÖRNEK 9.14 PARAMETRE YERİNE KOYARKEN DESEN EŞLEŞTİRMESİ

```
#!/bin/bash

# Pattern matching using the     # ## % %% parameter substitution operators.


var1=abcd12345abc6789

pattern1=a*c        # * (wild card) matches everything between a – c.


echo

echo "var1 = $var1"              # abcd12345abc6789

echo "var1 = ${var1}"            # abcd12345abc6789        (alternate form)

echo "Number of characters in ${var1} = ${#var1}"

echo "pattern1 = $pattern1"      # a*c (everything between ' a' and ' c')

echo


echo '${var1#$pattern1}  =' "${var1#$pattern1}"     #         d12345abc6789

# Shortest possible match, strips out first 12 characters  abcd12345abc6789

#                                      ^^^^^                |-|
```

Yazar Mendel Cooper'ın izni alınarak ticari olmayan amaçla Türkçe'ye çevrilmiştir.

```
echo ' ${var1##$pattern1} =' "${var1##$pattern1}"     #                    6789
# Longest possible match, strips out first 12   characters abcd12345abc6789
#                                     ^^^^^                 |-----------|
echo; echo

pattern2=b*9                # everything between ' b' and ' 9'
echo "var1 = $var1"        # Still  abcd12345abc6789
echo "pattern2 = $pattern2"
echo

echo ' ${var1%pattern2}   =' "${var1%$pattern2}"      #    abcd12345a
#  Shortest possible match, strips out last 6 characters abcd12345abc6789
#                                       ^^^^                          |----|
echo ' ${var1%%pattern2}  = =' "${var1%%$pattern2}"     # a

# Longest possible match, strips out last 12 characters   abc12345abc6789
#                                     ^^^^                   |------------|

# Remember, # and ## work from the left end of string,
#           % and %% work from the right end.

echo

exit 0
```

ÖRNEK 9.15 DOSYA ADLARINA YENİ AD VERİLMESİ

```
#!/bin/bash

#                   rfe
#                   ---

# Renaming file extensions.
#
#          rfe old_extension new_extension
#
# Example:
# To rename all *.gif files in working directory to *.jpg,
#          rfe gif jpg

ARGS=2
E_BADARGS=65

if [ $# -ne “$ARGS” ]
then
   echo ”Usage: ‘ basename $0’ old_file_suffix new_file_suffix”
   exit $E_BADARGS
fi

for filename in *.$1
# Traverse list of files ending with 1st argument.
do
  mv $filename ${filename%$1}$2
  #  Strip off part of filename matching 1st argument,
  #+ then append 2nd argument.
done

exit 0
```

Değişken büyütmesi / Karakter alt dizesinin yer değiştirmesi

Bu yapılar *ksh* tarafından benimsenmiştir.

`${d:poz}`

`d` değişkeninin, `poz` bağıl konumundan itibaren büyütülmesi.

`${d:poz:uz}`

`d` değişkeninin `poz` bağıl konumundan itibaren en fazla `uz` karakter uzunluğu kadar büyütülmesi. Bu operatörün kullanımına örnek için bkz. Örnek A-14.

`${d/Desen/Yedek}`

`d` değişkeni içinde bulunan `Desen`'in ilk eşleşmesi `Yedek` ile yerdeğiştirir.

`Yedek` yok sayıldığı takdirde `Desen` ile olan ilk eşleşme silinir.

`${d//Desen/Yedek}`

Global yerdeğiştirme. `d` değişkeni içinde bulunan `Desen`'in her eşleşmesi `Yedek` ile yerdeğiştirir.

Yukarıdaki gibi, `Yedek` yok sayılırsa, tüm `Desen` olageImeleri silinir.

ÖRNEK 9.16 KARAKTER DİZELERİNİ AYRIŞTIRMAK İÇİN DESEN EŞLEŞTİRMEYİ KULLANMAK

```
#!/bin/bash

var1=abcd-1234-defg

echo “var1 = $var1”

t=${var1#*-*}
```

```
echo "var1 (with everything, up to and including first – stripped out) =
$t"

#  t=${var1#*-} works just the same,

#+ since # matches the shortest string,

#+ and * matches everything preceding, including an empty string.

# (Thanks, S.C. for pointing this out.)

t=${var1##*-*}

echo "If var1 contains a \"-\", returns empty string...   var1 = $t"


t=${var1%*-*}

echo "var1 (with everything from the last – on stripped out) = $t"


echo

# ----------------------------------------

path_name=/home/bozo/ideas/thoughts.for.today

# ----------------------------------------

echo "path_name = $path_name"

t=${path_name##/*/}

echo "path_name, stripped of prefixes = $t"

# Same effect as t='basename $path_name' in this particular case.

#  t=${path_name%/}; t=${t##*/}    is a more general solution,

#+  but still fails sometimes.

#  If $path_name ends with a newline, then ' basenam $path_name' will not
work,

#+  but the above expression will.

# (Thanks, S.C.)


t=${path_name%/*.*}

# Same effect as   t='dirname $path_name'
```

```
echo “path_name, stripped of suffixes = $t”
# These will fail in some cases, such as “../”, “/foo////”, # “foo/”, “/”.
#  Removing suffixes, especially when the basename has no suffix,
#+ but the dirname does, also complicates matters.
# (Thanks, S.C.)

echo
t=${path_name:11}
echo “$path_name, with first 11 chars stripped off = $t”

t=${path_name:11:5}
echo “$path_name, with first 11 chars stripped off, length 5 = $t”
echo

t=${path_name/bozo/clown}
echo “$path_name with \”bozo\” replaced by \”clown\” = $t”

t=${path_name/today/}
echo “$path_name with \”today\” deleted= $t”

t=${path_name//o/O}
echo “$path_name with all o’s capitalized = $t”

t=${path_name//o/}
echo “$path_name with all o’s deleted = $t”

exit 0
```

${d/#Desen/Yedek}

d değişkeninin öneki, *Desen* ile eşleşiyorsa *Desen* yerine *Yedek* yazılır.

${değişken/%Desen/Yedek}

d değişkeninin soneki, *Desen* ile eşleşiyorsa *Desen* yerine *Yedek* yazılır.

ÖRNEK 9.17 KARAKTER DİZESİNİN ÖNEKİYLE veya SONEKİYLE EŞLEŞEN DESENLER

```
#!/bin/bash
# Pattern replacement at prefix / suffix of string.
v0=abc1234zip1234abc    # Original variable.
echo "v0 = $v0"         # abc1234zip1234abc
echo

# Match at prefix (beginning) of string.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDE1234zip1234abc
                        # |---|
# Match at suffix (end) of string.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #               |----|
echo
#  --------------------------------------------------------
#  Must match at beginning / end of string,
```

```
#+ otherwise no replacement results.

#  -------------------------------------------------------

v3=${v0/#123/000}       # Matches, but not at beginning.

echo "v3 = $v3"         # abc1234zip1234abc

                        # NO REPLACEMENT.

v4=${v0/%123/000}       # Matches, but not at end.

echo "v4 = $v4"         # abc1234zip1234abc

                        # NO REPLACEMENT.

exit 0
```

${!değişken_önek*}, ${!değişken_önek@}

Önceden bildirilen ve *değişken_önek* ile başlayan tüm değişkenler ile eşleşir.

```
xyz23=whatever
xyz24=

a=${!xyz*}              # Expands to names of declared variables beginning
with "xyz".
echo "a = $a"           # a = xyz23 xyz24

a=${!xyz@}              # Same as above.
echo "a = $a"           # a = xyz23 xyz24

# Bash, version 2.04, adds this feature.
```

Notlar

[1] Etkileşimsiz komut dosyalarında $parametre değeri boşsa, betik 127 hata koduyla sonlanır ("komut bulunamadı" hatası ile birlikte).

9.4 YAZIM DEĞİŞKENLERİ: declare veya typeset

Bildirim veya yazım kümesi yerleşikleri (bunlar tamamen eşanlamlıdır) değişkenlerin özelliklerini kısıtlamaya yarar. Bu bazı programlama dilleri için mevcut olan çok zayıf bir yazım şeklidir. **`declare`** komutu Bash sürüm 2 veya üstüne özgüdür. **`typeset`** komutu `ksh` komut dosyalarında da çalışır.

`declare / typeset seçenekleri`

`-r` *salt okunur*

```
declare –r değişken1
```

(**`declare -r değişken1`** ile **`readonly değişken1`** aynı şekilde çalışır)

Bu `C` **`const`** tip niteleyicisi ile hemen hemen eşdeğerdir. Salt okunur bir değişkenin değerini değiştirmek girişimi bir hata iletisiyle başarısız olur.

`-i` *tamsayı*

```
declare –i number
# The script will treat subsequent occurrences of “number” as an integer.

number=3
echo “number = $number”    # number = 3

number=three
echo “number = $number”    # number = 0
# Tries to evaluate “three” as an integer.
```

Unutmayınız ki, belirli aritmetik işlemler, `expr` ve `let` tanımlamalarına gerek kalmaksızın bildirilen tamsayı değişkenlerini kabul ederler.

`-a` *dizi*

```
declare –a endeksler
```

`endeksler` değişkeni bir dizi olarak ele alınacaktır.

`-f` *fonksiyonlar*

```
declare –f
```

Bir komut dosyasında argümansız çalıştırılan declare –f satırı o betikte daha önce tanımlanmış bütün fonksiyonların listelenmesine neden olur.

```
declare –f fonksiyon_adı
```

Bir betikte çalıştırılan `declare –f fonksiyon_adı` satırı sadece `fonksiyon_adı`'na karşılık gelen fonksiyonu listeler.

`-X` *`export`*

```
declare –x değişken3
```

Bu komut, betiğin kendi ortamı dışına taşımak için kullanılabilir bir değişkeni bildirir.
değişken=$değer

```
declare –x değişken3=373
```

`declare` komutu bir değişkenin özelliklerinin atandığı aynı cümle içinde değişkene değer atamaya yarar.

ÖRNEK 9.18. DEĞİŞKENLERİ YAZMAK İÇİN declare KULLANILMASI

```
#!/bin/bash

func1 ()
{
echo This is a function.
}

declare –f                    # Lists the function above.
echo

declare –i var1               # var1 is an integer.
var1=2367
echo "var1 declared as $var1"
var1=var1+1                   # Integer declaration eliminates the need for
' let'.
echo "var1 incremented by 1 is $var1."
# Attempt to change variable declared as integer
echo "Attempting to change var1 to floating value, 2367.1."
var1=2367.1                   # Results in error message, with no change to
variable.
echo "var1 is still $var1"

echo

declare –r var2=13.36         # ' declare' permits setting a variable
property
                              #+ and simultaneously assigning it a value.
echo "var2 declared as $var2"       # Attempt to change readonly variable.
Var2=13.37                    # Generates error message, and exit from
script.
```

```
echo "var2 is still $var2"          # This line will not execute.

exit 0                              # Script will not exit here.
```

9.5 DEĞİŞKENLERE YAPILAN DOLAYLI REFERANSLAR

Bir değişkenin değerinin ikinci bir değişkenin adı olduğunu varsayalım. İkinci değişkenin değerini bir şekilde birinciden almak mümkün müdür? Örneğin, `a = alfabe_harfi` ve `alfabe_harfi = z` ise, `a` değişkenine yapılan bir referans `z` değerini döndürür mü? Bu gerçekten de yapılabilir, ve *dolaylı referans* olarak adlandırılır. Seyrek olarak kullanılan `eval` gösterimi ile bu mümkündür, örneğin `var1=\$$var2.`

ÖRNEK 9.19 DOLAYLI REFERANSLAR

```
#!/bin/bash

# Indirect variable referencing.


a=letter_of_alfabet

letter_of_alfabet=z


echo

# Direct reference.

echo "a = $a"

# Indirect reference.

eval a=\$$a

echo "Now a = $a"

echo
```

```
# Now, let's try changing the second order reference.

t=table_cell_3

table_cell_3=24

echo "\"table_cell_3\" = $table_cell_3"

echo –n "dereferenced \"t\" = "; eval echo \$$t

# In this simple case,

#   eval t=\$$t; echo "\"t\" = $t"

# also works (why?).

echo

t=table_cell_3

NEW_VAL=387

table_cell_3=$NEW_VAL

echo "Changing value of \"table_cell_3\" to $NEW_VAL."

echo "\"table_cell_3\" now $table_cell_3"

echo –n "dereferenced \"t\" now "; eval echo \$$t

# "eval" takes the two arguments "echo" and "\$$t" (set equal to
$table_cell_3)

echo


# (Thanks, S.C., for clearing up the above behavior.)


# Another method is the ${!t} notation, discussed in "Bash, version 2"
section.

# See also example "ex78.sh"

exit 0
```

ÖRNEK 9.20 DOLAYLI BİR REFERANSI *awk*'a GEÇİRMEK

```
#!/bin/bash
# Another version of the "column totaler" script
# that adds up a specified column (of numbers) in the target file.
# This uses indirect references.

ARGS=2
E_WRONGARGS=65
if [ $# -ne "$ARGS" ] # Check for proper no. of command line args.
then
      echo "Usage: ' basename $0'filename column-number"
      exit $E_WRONGARGS
fi

filename=$1
column_number=$2
#===== Same as original script, up to this point =====#
# A multi-line awk script is invoked by   awk '  .... '

# Begin awk script.
# ------------------------------------------------
awk "
{ total += \$${column_number}  # indirect reference
}
END {
     print total
     }
     " "$filename"
# ------------------------------------------------
# End awk script.
```

```
# Indirect variable reference avoids the hassles

# of referencing a shell variable within the embedded awk script.

# Thanks, Stephane Chazelas.


exit 0
```

Dolaylı referansın bu yöntemi biraz zordur. İkinci derece değişken değerini değiştirirse, birinci dereceden değişkene (yukarıdaki örnekteki gibi) doğru referans verilmesi gerekir. Neyse ki, Bash sürüm 2 ile tanıtılan *${!değişken}* gösterimi, dolaylı referansı daha basit hale getirmiştir (bkz. Örnek 35.2).

9.6 $RANDOM: RASGELE TAMSAYI OLUŞTURMAK

$RANDOM 0-32767 aralığında bir *sözde-rasgele* tamsayı döndüren iç Bash fonksiyonudur (sabitle karıştırmayınız). $ RANDOM şifreleme anahtarı oluşturmak için *kullanılmamalıdır*.

ÖRNEK 9.21 RASGELE SAYILARI OLUŞTURMAK

```
#!/bin/bash


# $RANDOM returns a different random integer at each invocation.

# Nominal range: 0 – 32767 (signed 16-bit integer).


MAXCOUNT=10

count=1


echo

echo "$MAXCOUNT random numbers:"

echo "-----------------"

while [ "$count" -le $MAXCOUNT ]         # Generate 10 ($MAXCOUNT) random
integers.
```

```
do
  number=$RANDOM
  echo $number
  let "count += 1"     # Increment count.
done
echo "-------------"

# If you need a random int within a certain range, use the ' modulo'
operator.
# This returns the remainder of a division operation.
RANGE=500
echo
number=$RANDOM
let "number %= $RANGE"
echo "Random number less than  $RANGE --- $number"
echo
# If you need a random int greater than a lower bound,
# then set up a test to discard all numbers below that.
FLOOR=200
number=0      #initialize
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
done

echo "Random number greater than $FLOOR ---  $number"
echo

# May combine above two techniques to retrieve random number between
# two limits.
number=0   #initialize
```

```
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"   # Scales $number down within $RANGE.
done
echo "Random number between $FLOOR and $RANGE --- $number"
echo

# Generate binary choice, that is, "true" or "false" value.
BINARY=2
number=$RANDOM
T=1

let "number %= $BINARY"
# let "number >>= 14"   gives a better random distribution
# (right shifts out everything except last binary digit).
if [ "$number" -eq $T ]
then
  echo "TRUE"
else
  echo "FALSE"
fi
echo

# May generate toss of the dice.
SPOTS=7   # Modulo 7 gives range 0 - 6.
DICE=2
ZERO=0
die1=0
die2=0
```

```
# Tosses each die separately, and so gives correct odds.
  while [ "$die1" -eq $ZERO ]     # Can't have a zero come up.
  do
    let "die1 = $RANDOM % $SPOTS" # Roll first one.
  done

  while [ "$die2" -eq $ZERO ]
  do
    let "die2 = $RANDOM % $SPOTS" # Roll second one.
  done
let "throw = $die1 + $die2
echo "Throw of the dice = $throw"
echo
exit 0
```

RANDOM nasıl bir rasgeledir? Bunu test etmenin en iyi yolu RANDOM tarafından oluşturulan "rasgele" sayıların dağılımını izleyen bir komut dosyası yazmaktır. RANDOM zarını birkaç kez yuvarlayalım.

ÖRNEK 9.22 RANDOM ZARINI YUVARLAMAK

```
#!/bin/bash
# How random is RANDOM?
RANDOM=$$     # Reseed the random number generator using script process ID.

PIPS=6        # A die has 6 pips.
MAXTHROWS=600 # Increase this, if you have nothing beter to do with your
time.
throw=0       # Throw out.

zeroes=0      # Must initialize counts to zero.
```

```
ones=0        # since an uninitialized variable is null, not zero.
twos=0
threes=0
fours=0
fives=0
sixes=0


print_result ()
{
echo
echo “ones =   $ones”
echo “twos =   $twos”
echo “threes = $threes”
echo “fours =  $fours”
echo “fives =  $fives”
echo “sixes =  $sixes”
echo
}


update_count()
{
case “$1” in
  0) let “ones += 1”;;     # Since die has no “zero”, this corresponds to
     1.
  1) let “twos += 1”;;     # And this to 2, etc.
  2) let “threes += 1”;;
  3) let “fours += 1”;;
  4) let “fives += 1”;;
  5) let “sixes += 1”;;
esac
}
echo
```

```
while [ "$throw" -lt "$MAXTHROWS" ]

do

  let "die1 = RANDOM % $PIPS"

  update_count $die1

  let "throw += 1"

done


print_result

# The scores should distribute fairly evenly, assuming RANDOM is fairly
random.

# With $MAXTHROWS at 600, all should cluster around 100, plus-or-minus 20
or so.

#

# Keep in mind that RANDOM is a pseudorandom generator,

# and not a spectacularly good one at that.

# Exercise (easy):

# ---------------

# Rewrite this script to flip a coin 1000 times.

# Choices are "HEADS" or "TAILS".


exit 0
```

Son örnekte de gördüğümüz gibi, RANDOM üretecinin her çağrıldığında yeniden başlatılması en doğrusudur. Her seferinde RANDOM için aynı başlangıcı kullanmak aynı sayıların üretilmesine neden olacaktır. (C programlama dilindeki *`random()`* fonksiyonunun davranışı gibidir.)

ÖRNEK 9.23 RANDOM'u YENİDEN BAŞLATMAK

```
#!/bin/bash
# seeding-random.sh: Seeding the RANDOM variable.

MAXCOUNT=25       # How many numbers to generate.
random_numbers ()
{
count=0
while [ "$count" -lt "$MAXCOUNT" ]
do
  number=$RANDOM
  echo -n "$number "
  let "count += 1"
done
}

echo; echo
RANDOM=1          # Setting RANDOM seeds the random number generator.
random_numbers

echo; echo
RANDOM=1          # Same seed for RANDOM...
random_numbers    #...reproduces the exact same number series.
                  #
                  # When is it useful to duplicate a "random" number series?
echo; echo
RANDOM=2          # Trying again, but with a different seed...
random_numbers    # gives a different number series.
echo; echo
```

```
# RANDOM=$$  seeds RANDOM from process id of script.
# It is also possible to seed RANDOM from ‘ time’ or‘ date’ commands.

# Getting fancy...

SEED=$(head -1 /dev/urandom | od –N 1 | awk ‘ { print$2 }’)

#  Pseudo-random output fetched

#+ from /dev/urandom (system pseudo-random device-file),

#+ then converted to line of printable (octal) numbers by “od”,

#+ finally “awk” retrieves just one number for SEED.

RANDOM=$SEED

random_numbers

echo; echo
exit 0
```

`/dev/urandom` aygıt dosyası, `$RANDOM` değişkeninden çok daha fazla "rasgele" sözde-rasgele sayılar üretmeyi sağlar. **`dd if=/dev/urandom of=targetfile bs=1 count=XX`** iyi dağıtılmış sözde-rasgele numaralardan oluşan bir dosya yaratır. Ancak bu numaraları komut dosyası içinde bir değişkene atamak için (yukarıdaki örnekte olduğu gibi) `od` aracılığıyla filtre veya `dd` kullanmak (bkz. Örnek 12.41) gibi bir çözüm gerektirir.

Bir komut dosyası içinde sözde-rasgele sayı üretmek için başka diğer araçlar da vardır. `awk` bunu yapmanın kolay bir yoludur.

ÖRNEK 9.24 SÖZDE-RASGELE SAYILAR, awk KULLANILARAK

```
#!/bin/bash
# random2.sh: Returns a pseudorandom number in the range 0 – 1.

# Uses the awk rand() function.

AWKSCRIPT=' { srand(); print rand() } '

# Command(s) / parameters passed to awk

# Note that srand() reseeds awk's random number generator.


echo –n "Random number between 0 and 1 = "

echo | awk "$AWKSCRIPT"


exit 0


# Exercises:

# ---------

# 1) Using a loop construct, print out 10 different random numbers.

#      (Hint: you must reseed the "srand()" function with a different seed

#      in each pass through the loop. What happens if you fail to do this?)

# 2) Using an integer multiplier as a scaling factor, generate random

# numbers

#    in the range between 10 an 100.

# 3) Same as exercise #2, above, but generate random integers this time.
```

9.7 ÇİFT PARANTEZ YAPISI

`let` komutuna benzer olarak, ((…)) yapısı aritmetik açılım ve değerlendirmeye fırsat verir. En basit haliyle, **`a = $((5 + 3))`** "a" için "5 + 3" veya 8 ataması yapar. Ancak, bu çift parantez yapısı aynı zamanda Bash değişkenlerinin C tipi manipülasyonunu sağlayan bir mekanizmadır.

ÖRNEK 9.25 DEĞİŞKENLERİN C-TİPİ MANİPÜLASYONU

```
#!/bin/bash

# Manipulating a variable, C-style, using the ((...)) construct.


echo

(( a = 23 ))      # Setting a value, C-style, with spaces on both sides of
the "=".

echo "a (initial value) = $a"

(( a++ ))         # Post-increment ' a', C-style.

echo "a (after a++) = $a"

(( a-- ))         # Post-decrement ' a', C-style.

echo "a (after a--) = $a"

(( ++a ))         # Pre-increment ' a', C-style.

echo "a (after ++a) = $a"

(( --a ))         # Pre-decrement ' a', C-style.

echo "a (after --a) = $a"

echo

(( t = a<45?7:11 ))   # C-style trinary operator.

echo "If a < 45, then t = 7, else t = 11."

echo "t = $t "        # Yes!

echo


# -----------------
```

Yazar Mendel Cooper’ın izni alınarak ticari olmayan amaçla Türkçe’ye çevrilmiştir.

```
# Easter Egg alert!
# -----------------
#  Chet Ramey apparently snuck a  bunch of undocumented C-style constructs
#+ into Bash (actually adapted from ksh, pretty much).
#  In the Bash docs, Ramey calls ((...)) shell arithmetic,
#+ but it goes far beyond that.
#  Sorry, Chet, the secret is now out.
#  See also “for” and “while” loops using the ((...)) construct.
#  These work only with Bash, version 2.04 or later.

exit 0
```

BÖLÜM 10 DÖNGÜLER ve AYRIMLAR

Kod blokları ve işlemleri, gerekli yapılandırmaya sahip kabuk betikleri için sahip olunan bir anahtardır. Döngü ve ayrım yapıları bunu gerçekleştirmek için gerekli araçları sağlar.

10.1 DÖNGÜLER

Döngü kontrol koşulu doğru sağlandığı sürece, bir komut listesini tekrar eden kod bloğudur.

for döngüleri

for (in)

Bu temel döngü yapısıdır. C karşılığından önemli ölçüde farklıdır.

for *arg* in [*liste*]

do

command(s)...

done

Döngünün her tekrarında *arg* ile belirtilen değer *liste*de verilen değişkenleri teker teker alır.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# In pass 1 of the loop, $arg = $var1
# In pass 2 of the loop, $arg = $var2
# In pass 3 of the loop, $arg = $var3
# ...
# In pass N of the loop, $arg = $varN
# Arguments in [list] quoted to prevent possible word splitting.
```

Argüman `listesi` joker karakter içerebilir.

do ile for aynı satırda yazılacak ise listeden sonra noktalı virgül konur.

for *`arg`* in [*`list`*]; do

ÖRNEK 10.1 BASİT for DÖNGÜLERİ

```
#!/bin/bash
# List the planets.
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet
done
echo
# Entire ‘ list’ enclosed in quotes creates a single variable.
for planet in “Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto”
do
  echo $planet
done
exit 0
```

Her **`[liste]`** elemanı birden fazla parametre içerebilir. Parametreler gruplar halinde işlenirse bu yararlıdır. Bu gibi durumlarda, (bkz. Örnek 11-11) `[liste]` de bulunan her elemanın ayrıştırılmasını sağlayabilmek ve her elemanın hangi konumda bulunduğunu atayabilmek için `set` komutunu kullanınız.

ÖRNEK 10.2 LİSTEDE İKİ PARAMETRESİ BULUNAN for DÖNGÜSÜ

```
#!/bin/bash
# Planets revisited.
# Associate the name of each planet with its distance from the sun.
for planet in “Mercury 36” “Venus67” “Earth 93” “Mars 142” “Jupiter 483”
do
```

```
  set -- $planet  # Parses variable "planet" and sets positional parameters.
  # the "--" prevents nasty surprises if $planet is null or begins with a dash.

  # May need to save original positional parameters, since they get owerwritten.
  # One way of doing this is to use an array,
  #        origianl_parameters=("$@")
  echo "$1             $2,000,000 miles from the sun"
  #-------two  tabs---concatenate zeroes onto parameter $2
done
#  (Thanks, S.C., for additional clarification.)
exit 0
```

`for` döngünün `listesi` bir değişken halinde de verilebilir.

ÖRNEK 10.3 *Fileinfo*: BİR DEĞİŞKEN OLARAK VERİLEN DOSYA LİSTESİNDE İŞLEM YAPMAK

```
#!/bin/bash
# fileinfo.sh
FILES="/usr/sbin/privatepw"
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind         # List of files you are curious about.
                     # Threw in a dummy file, /usr/bin/fakefile.
echo
for file in $FILES
do
  if [ ! -e "$file" ]         # Check if file exists.
  then
    echo "$file does not exist.";  echo
```

```
    continue                    # On to next.
  fi
  ls –l $file | awk ‘ { print $9 “
  whatis ‘ basename $file’     # File info.
  echo
done

exit 0
```

`for` döngüsüne ait `liste` içinde dosya isimlerinde kullanılan joker karakter geçebilir.

ÖRNEK 10.4 BİR for DÖNGÜSÜ KULLANARAK DOSYALAR İLE ÇALIŞMA

```
#!/bin/bash
# list-glob.sh: Generating [list] in a for-loop using “globbing”.
echo
for file in *
do
  ls –l “$file”  # Lists all files in $PWD (current directory).
  # Recall that the wild card character “*” matches everything,
  # however, in “globbing”, it doesn’t match dot-files.
  # If the pattern matches no file, it is expanded to itself.
  # To prevent this, set the nullglob option
  # (shopt –s nullglob).
  # Thanks, S.C.
done
echo; echo

for file in [jx]*
do
```

```
  rm –f $file    # Removes only files beginning with “j” or “x” in $PWD.

  echo “Removed file \”$file\””.

done

echo

exit 0
```

Döngünün `in [liste]` bileşeni yazılmazsa, betiğe komut satırında verilen argüman listesi, yani `$@` ile for döngüsü işleme konur. (İyi bir örnek için bkz. Örnek A.16).

ÖRNEK 10.5 for DÖNGÜSÜNDE in [liste] YAZILMAZSA

```
#!/bin/bash
#  Invoke both with and without arguments
for a
do
  echo –n “$a ”
done

#  The ‘ in list’ missing, therefore the loop operates on ‘ $@’
#+  (command-line argument list, including whitespace).

echo

exit 0
```

`for` döngüsünde `[liste]` elemanları komutla belirlenebilir (bkz. Örnek 12.38, Örnek 10.10 and Örnek 12.33).

ÖRNEK 10.6 for DÖNGÜSÜNDEKİ [liste] BİR KOMUTLA YER DEĞİŞEBİLİR

```
#!/bin/bash

# A for-loop with [list] generated by command substitution.

NUMBERS=”9 7 3 8 37.53”

for number in ‘ echo $NUMBERS’   # for number in 9 73 8 37.53
do
  echo –n “$number “
done

echo

exit 0
```

Şimdi daha karmaşık bir başka örnekte `[liste]` yaratmak için komut ile yer değiştirelim.

ÖRNEK 10.7 İKİLİ DOSYALAR İÇİN grep YER DEĞİŞİMİ

```
#!/bin/bash
# bin-grep.sh: Locates matching strings in a binary file.

# A "grep" replacement for binary files.
# Similar effect to "grep -a"

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Usage: ' basename $0' string filename"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "File \"$2\" does not exist."
  exit $E_NOFILE
fi

for word in $( strings "$2" | grep "$1" )
# The "strings" command lists strings in binary files.
# Output then piped to "grep", which tests for desired string.
do
  echo $word
done

# As S.C. points out, the above for-loop could be replaced with the simpler
#    strings "$2" | grep "$1" | tr -s "$IFS" ' [\n*]'

# Try something like "./bin-grep.sh mem /bin/ls" to exercise this script.

exit 0
```

Buna benzer örnekler çoğaltılabilir.

ÖRNEK 10.8 SİSTEMDEKİ TÜM KULLANICILARI LİSTELEMEK

```
#!/bin/bash
# userlist.sh

PASSWORD_FILE=/etc/passwd
n=1           #  User number

for name in $(awk ' BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
# Field separator = :    ^^^^^^
```

```
# Print first field              ^^^^^^^^
# Get input from password file              ^^^^^^^^^^^^^^^^^
do
  echo "USER #$n = $name"
  let "n += 1"
done

# USER #1 = root
# USER #2 = bin
# USER #3 = daemon
# ...
# USER #30 = bozo

exit 0
```

Komut yer değiştirmesi sonucunda `[liste]` ile ilgili son örnek.

ÖRNEK 10.9 BİR DİZİN İÇİNDEKİ TÜM İKİLİ DOSYALARIN SAHİBİNİ BULMAK

```
#!/bin/bash
# findstring.sh:
# Find a particular string in binaries in a specified directory.

directory=/usr/bin
fstring="Free Software Foundation"   # See which files come from the FSF.

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
  #  In the "sed" expression,
  #+ it is necessary to substitute for the normal "/" delimiter
  #+ because "/" happens to be one of the characters filtered out.
  #  Failure to do so gives an error message (try it).
done

exit 0

#  Exercise (easy):
#  ----------------
#  Convert this script to taking command-line parameters
#+ for $directory and $fstring.
```

`for` döngüsünün çıktısı bir komuta veya komutlara oluk yapılabilir.

ÖRNEK 10.10 BİR DİZİNDEKİ SEMBOLİK BAĞLARI LİSTELEME

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.

ARGS=1                  # Expect one command-line argument.

if [ $# -ne "$ARGS" ]  # If not 1 arg...
then
  directory='pwd'       # current working directory
else
  directory=$1
fi

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type l )"   # -type l = symbolic links
do
  echo "$file"
done | sort

# As Dominik ' Aeneas' Schnitzer points out,
#+ failing to quote $( find $directory -type l )
#+ will choke on filenames with embedded whitespace.

exit 0
```

Önceki örneğe yapılan bu küçük değişikliğin gösterdiği gibi bir döngü işlevini yaparken `stdout` yönlendirmesi de kullanabilir.

ÖRNEK 10.11 BİR DİZİNDEKİ SEMBOLİK BAĞLARIN BİR DOSYAYA KAYDEDİLMESİ

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.

ARGS=1                  # Expect one command-line argument.
OUTFILE=symlinks.list   # save file

if [ $# -ne "$ARGS" ]  # If not 1 arg...
then
  directory='pwd'       # current working directory
else
  directory=$1
fi

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type 1 )"
do
  echo "$file"
```

```
done | sort > "$OUTFILE"
#           ^^^^^^^^^^^^
exit 0
```

Döngüye alternatif olan ve C programcıları için çok tanıdık bir sözdizimi vardır. Bu çift parantez gerektirir.

ÖRNEK 10.12 C-BENZERİ for DÖNGÜSÜ

```
#!/bin/bash
# Two ways to count up to 10.

echo

# Standard syntax.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

# +==========================================+
# Now, let's do the same, using C-like syntax.
Limit=10
for ((a=1; a <= LIMIT; a++)) # Double parentheses, and "LIMIT" with no "$".
do
  echo -n "$a "
done                          # A construct borrowed from ' ksh93'.

echo; echo
# +==========================================+
# Let's use the C "comma operator" to increment two variables
simultaneously.
for ((a=1, b=1; a <= LIMIT; a++, b++))    # The comma chains together
operations.
do
  echo -n "$a-$b "
done

echo; echo

exit 0
```

(bkz. Örnek 26.7, 26.8, ve A.7).

Şimdi, `for` döngüsünün gerçek bağlamda kullanılmasına bir örnek gösterim.

ÖRNEK 10.13 TOPLU MODDA efax KULLANMA

```
#!/bin/bash

EXPECTED_ARGS=2
E_BADARGS=65

if [ $# -ne $EXPECTED_ARGS ]
# Check for proper no. of command line args.
then
  echo "Usage: ' basename $0' phone# text-file"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "File $2 is not a text file"
  exit $E_BADARGS
fi

fax make $2              # Create fax formatted files from text files.

for file in $(ls $2.0*)  # Concatenate the converted files.
                         # Uses wild card in variable list.

do
  fil="$fil $file"
done

efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # Do the work.

# As S.C. points out, the for-loop can be eliminated with
#    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
# but it's not quite as instructive [grin].

exit 0
```

while

Bu yapı döngünün başındaki koşulu test eder ve sonuç doğru olduğu sürece (0 çıkış kodu) döngü işlemeye devam eder. `for` döngüsünün aksine, bir `while` döngüsü döngü tekrarlarının önceden bilinmediği durumlarda kullanım alanı bulur.

```
while [condition]
do
  command...
done
```

for/in döngülerinde olduğu gibi do; test ile aynı satıra yazılırsa, noktalı virgül koymanız gerekir.

```
while [condition]; do
```

Bazı özelleşmiş **while** döngüleri, `getopts` örneğinde olduğu gibi, burada verilmiş olan standart şablondan farklılık gösterir.

ÖRNEK 10.14 BASİT while DÖNGÜSÜ

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n suppresses newline.
  var0='expr $var0 + 1'   # var0=$(($var0+1)) also works.
done

echo

exit 0
```

ÖRNEK 10.15 BİR BAŞKA while DÖNGÜSÜ

```
#!/bin/bash

echo

while [ "$var1" != "end" ]          # while test "$var1" != "end"
do                                  # also works.
  echo "Input variable #1 (end to exit) "
  read var1                         # Not ' read $var1'(why?).
  echo "variable #1 = $var1"        # Need quotes because of ' #'.
  # If input is ' end', echoes it here.
  # Does not test for termination condition until top of loop.
  echo
done

exit 0
```

Bir **while** döngüsünün birden fazla koşulu olabilir. Son koşul da değerlendirildikten sonra, döngünün ne zaman sonlanacağına karar verilir. Ancak bu, biraz daha farklı bir döngü söz dizimi gerektirmektedir.

ÖRNEK 10.16 BİRDEN ÇOK KOŞULLU while DÖNGÜSÜ

```
#!/bin/bash

var1=unset
previous=$var1

while echo "previous -variable = $previous"
      echo
      previous=$var1
      [ "$var1" != end ]       # Keeps track of what $var1 was previously.
      # Four conditions on "while", but only last one controls loop.
      # The *last* exit status is the one that counts.
do
echo "Input variable #1 (end to exit) "
  read var1
  echo "variable #1 = $var1"
done

# Try to figure out how this all Works.
# It's a wee bit tricky.

exit 0
```

`for` döngüsünde olduğu gibi `while` döngüsü de çift parantez yapısını kullanarak C-benzeri söz dizimini kabul eder (bkz. Örnek 9-25).

ÖRNEK 10.17 C-BENZERİ SÖZ DİZİMİNİN while DÖNGÜSÜNDE KULLANIMI

```
#!/bin/bash
# wh-loopc.sh: Count to 10 in a "while" loop.

LIMIT=10
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done            # No surprises, so far.

echo; echo

# +=========================================================================+

# Now, repeat with C-like syntax.

((a=1))         # a=1
# Double parentheses permit space when setting a variable, as in C.
```

```
while (( a <= LIMIT ))    # Double parentheses, and no "$" preceeding
variables.
do
  echo -n "$a "
  ((a += 1))    # let "a+=1"
  # Yes, indeed.
  # Double parentheses permit incrementing a variable with C-like syntax.
done

# Now, C programmers can feel right at home in Bash.

exit 0
```

while döngüsünün sonuna yazılan bir < ile `stdin` bir dosyaya yönlendirilebilir.

until

Döngünün başındaki koşul test edilir, yanlış olduğu sürece döngü tekrar eder (while döngüsünün aksine).

```
until [condition-is-true]
do
  command...
done
```

Unutmayınız ki, bazı programlama dillerinde sonlanma koşulunun döngünün sonunda yer aldığı benzer yapılar bulunmaktadır.

`for/in` döngülerinde olduğu gibi, do ile test koşulu aynı satıra yazılacaksa sonuna noktalı virgül konur.

```
until [condition-is-true] ; do
```

ÖRNEK 10.18 until DÖNGÜSÜ

```
#!/bin/bash
until [ "$var1" = end ] # Tests condition here, at top of loop.
do
  echo "Input variable #1 "
  echo "(end to exit)"
  read var1
  echo "variable #1 = $var1"
done

exit 0
```

10.2 İÇ İÇE DÖNGÜLER

Bir iç içe döngü bir döngü içinde bir döngüdür, yani bir dış döngünün içindeki iç döngüdür. Dıştaki döngünün ilk geçişi tamamlaması içteki döngünün çalışmasını tetikler, içteki döngü çalışmasını sonuna kadar sürdürür. Sonra dıştaki döngünün ikinci geçişi tekrar iç döngüyü tetikler. Dıştaki döngü bitinceye kadar bu tekrarlanır. Tabii ki, iç veya dış döngü içindeki bir kesinti bu süreci kesintiye uğratacaktır.

ÖRNEK 10.19 İÇ İÇE DÖNGÜ

```
#!/bin/bash
# Nested “for” loops.
outer=1             # Set outer loop counter.
# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo “Pass $outer in outer loop.”
  echo “----------------------“
  inner=1           # Reset inner loop counter.
  # Beginning of inner loop.
  for b in 1 2 3 4 5
  do
    echo “Pass $inner in inner loop.
    let “inner+=1”  # Increment inner loop counter.
  done
  # End of inner loop.
  let “outer+=1”    # Increment outer loop counter.
  echo              # Space between output in pass of outer loop.
done
# End of outer loop.
exit 0
```

İç içe "while" döngülerine bir örnek için bkz.Örnek 26.4, ve bir "while" döngüsü "until" döngüsü içinde ne yapar örneği için bkz.Örnek 26.5.

10.3 DÖNGÜ KONTROLÜ

Döngü Davranışını Etkileyen Komutlar

break, **continue**

Break ve **continue** döngü kontrol komutları [1] diğer programlama dillerindeki benzerleri ile birebir örtüşür. Break komutunun çalışmasıyla, döngü içindeki tüm komutlar sonraki yinelemeye başlanmak üzere ileriye sarılır.

ÖRNEK 10.20 BİR DÖNGÜ İÇİNDE break ve continue ETKİLERİ

```
#!/bin/bash

LIMIT=19  # Upper limit

echo

echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=20

while [ $a -le "$LIMIT" ]

do

  a=$(($a+1))

  if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]   # Excludes 3 and 11

  then

    continue   # Skip rest of this particular loop iteration.

  fi

  echo -n "$a "

done
```

```
# Exercise:
# Why does loop print up to 20?
echo; echo
echo Printing Numbers 1 through 20, but something happens after 2.
##############################################################
# Same loop, but substituting ‘ break’ for ‘ continúe
a=0
while [ “$a” –le “$LIMIT” ]
do
  a=$(($a+1))
  if [ “$a” –gt 2 ]
  then
    break   # Skip entire rest of loop.
  fi
  echo –n “$a “
done
echo; echo; echo
exit 0
```

break komutu isteğe bağlı bir parametre alabilir, parametresiz olanı gömüle olan en içteki döngüyü sonlandırır.

ÖRNEK 10.21 BİRDEN FAZLA DÖNGÜ SEVİYESİNDEN break KOMUTUYLA ÇIKMAK

```
#!/bin/bash
# break-levels.sh: Breaking out of loops.
# “break N” breaks out of N level loops.
for outerloop in 1 2 3 4 5
do
  echo –n “Group $outerloop:   “
```

```
  for innerloop in 1 2 3 4 5
  do
    echo –n "$innerloop "
    if [ "innerloop" –eq 3 ]
    then
      break  # Try   break 2   to see what happens.
             # ("Breaks" out of both inner and outer loops.)
    fi
  done
  echo
done

echo
exit 0
```

continue komutu **break** komutuna benzer şekilde, isteğe bağlı bir parametre alır. Parametresiz olanı kendi döngüsü içindeki halihazırdaki döngüyü kırarak sonraki yinelemeyi çalıştırır. **continue N** kendi döngü düzeyinde diğer tüm yinelemeleri sonlandırır ve döngünün N. seviyesinden yinelemeleri çalıştırmaya devam eder.

ÖRNEK 10.22 YÜKSEK BİR DÖNGÜ DÜZEYİNDEN DEVAM ETMEK

```
#!/bin/bash
# The "continue N" command, continuing at the Nth level loop.
do outer in I II III IV V            # outer loop
do
  echo; echo –n "Group $outer: "
  for inner in 1 2 3 4 5 6 7 8 9 10   # inner loop
  do
    if [ "$inner" –eq 7 ]
    then
```

```
      continue 2  # Continue at loop on 2nd level, that is "outer loop".
                  # Replace above line with a simple "continue"
                  # to see normal loop behavior.
    fi
    echo -n "$inner " # 8 9 10 will never echo.
  done
done

echo; echo
# Exercise:
# Come up with a meaningful use for "continue N" in a script.
exit 0
```

continue N yapısını anlamak ve anlamlı bir bağlamda kullanmak zordur. Belki de bundan kaçınmak en doğrusudur.

Notlar

[1] `while` ve `case` gibi diğer döngü komutları anahtar sözcük oldukları halde bunlar kabuk yerleşiğidir.

10.4 TEST ve AYRIMLAR

case ve **select** yapıları teknik olarak döngü değildir, çünkü bir kod bloğunun çalışmasını yinelemezler. Döngüler ile ortak yönü bloğun altındaki ve üstündeki koşullara göre program akışının yönetilmesidir.

Bir kod bloğu içinde program akışının kontrol edilmesi

case (in) /esac

case yapısı kabuktaki C/C++ **switch** eşdeğeridir. Test koşullarına bağlı olarak kod bloklarından sadece birinin ayrımına ait olan komut akışına izin verir. Bunu birden fazla if/then/else cümlesi yerine kullanılabilen ve menü tercihlerine ait olan işlemler için hizmet eden bir yardımcı olarak düşünebilirsiniz.

```
case "$değişken" in
"$koşul1")
komut...
;;
"$koşul2")
komut...
;;
esac
```

- Değişkenlere referans vermek zorunlu değildir çünkü, sözcüğü bütünlüğünü korumuştur.
- Her test koşulunun sonuna bir sağ parantez konur.
- Koşula it olan komut bloğunun sonuna çift noktalı virgül konur ;; .
- **case** bloğunun sonuna bitiş cümlesi olarak **esac** konur.

ÖRNEK 10.23 case KULLANIMI

```
#!/bin/bash
echo; echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "Lowercase letter";;
  [A-Z]   ) echo "Uppercase letter";;
  [0-9]    ) echo "Digit";;
  *        ) echo "Punctuation, whitespace, or other";;
esac  # Allows ranges of characters in [square brackets].

# Exercise:
# --------
# As the script stands, # it accepts a single keystroke,
# Change the script so it accepts continuous input,
# reports on each keystroke, and terminates only when "X" is hit.
# Hint: enclose everything in a "while" loop.

exit 0
```

ÖRNEK 10.24 case KULLANARAK MENÜ OLUŞTURMAK

```
#!/bin/bash
# Crude address database
clear # Clear the screen.
echo "          Contact List"
echo "Choose one of the following persons:"
echo
```

```
echo “[E]vans, Roland”
echo “[J]ones, Milred”
echo “[S]mith, Julie”
echo “[Z]ane, Morris”
echo

read person

case “$person” in
# Note variable is quoted.
  “E” | “e” )
  # Accept upper or lowercase input.
  echo
  echo “Roland Evans”
  echo “4321 Floppy Dr.”
  echo “Hardscrabble, CO 80753”
  echo “(303) 734-9874”
  echo “(303) 734-9832 fax”
  echo “revans@zzy.net”
  echo “Business partner & old friend”
  ;;
# Note double semicolon to terminate
# each option.
  “J” | “j” )
  echo
  echo “Milred Jones”
  echo “249 E. 7th St., Apt. 19”
  echo “New York, NY 10009”
  echo “(212) 533-2814”
  echo “(212) 533-9972 fax”
```

```
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb 11"
  ;;
  Add info for Smith & Zane later.
          * )
   # Default option.
   # Empty input (hitting RETURN) fits here, too.
   echo
   echo "Not yet in database."
   ;;
esac


echo


#  Exercise:
#  --------
#  Change the script so it accepts continuous input,
#+ instead of terminating after terminating after displaying just one
address.
exit 0
```

Son derece akıllı bir **case** kullanımı komut satırı parametrelerini test etmeyi içerir.

```
#!/bin/bash
case "$1" in
"") echo "Usage: ${0##*/} <filename>"; exit 65;; # No command-line parameters,
                                                 # or first parameter empty.
# Note that ${0##*/} is ${var##pattern} param substitution. Net result is $0.
-*) FILENAME=./$1;;   # If filename passed as argument ($1) starts with a dash,
                      # replace it with ./$1
                      # so further commands don't interpret it as an option.

*)  FILENAME=$1;;     # Otherwise, $1.
esac
```

ÖRNEK 10.25 case DEĞİŞKENİNİ ÜRETMEK İÇİN KOMUT YER DEĞİŞİMİNİ KULLANMAK

```
#!/bin/bash
# Using command substitution to generate a "case" variable.

case $( arch ) in    # "arch" returns machine architecture.
i386 ) echo "80386-based machine";;
i486 ) echo "80486-based machine";;
i586 ) echo "Pentium-based machine";;
i-686 ) echo "Pentium2+-based machine";;
*      ) echo "Other type of machine";;
esac

exit 0
```

case yapısı joker karakterlerle yazılan desenleri ve karakter dizelerini filtreleyebilir.

ÖRNEK 10.26 KARAKTER DİZESİNİN BASİT EŞLEŞMESİ

```
#!/bin/bash
# match-string.sh: simple string matching
match_string ()
{
  MATCH=0
  NOMATCH=90
  PARAMS=2     # Function requires 2 arguments.
  BAD__PARAMS=91
  [ $# -eq $PARAMS ] || return $BAD_PARAMS
  case "$1" in
  "$2") return $MATCH;;
  *   ) return $NOMATCH;;
  esac
}

a=one
b=two
c=three
d=two

match_string $a      # wrong number of parameters
echo $?              # 91
match_string $a $b   # no match
echo $?              # 90

match_string $b $d   # match
echo $?              # 0

exit 0
```

ÖRNEK 10.27 ALFABETİK GİRDİNİN DENETİMİ

```
#!/bin/bash
# Using "case" structure to filter a string.
SUCCESS=0
FAILURE=-1
isalpha () # Tests whether *first character* of input string is alphabetic.
{
İf [ -z "$1" ]
then
  return $FAILURE
fi

case "$1" in
[a-zA-Z]*) return $SUCCESS;;
         ) return $FAILURE;;
esac
}             # Compare this with "isalpha ()" function in C.

isalpha2 ()   # Tests whether *entire string* is alphabetic.
{
  [ $# -eq 1 ] || return $FAILURE
  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
                *) return $SUCCESS;;
  esac
}
```

```
check_var ()   # Front-end to isalpha().
{
if isalpha "$@"
then
  echo "$* = alpha"
else
  echo "$* = non-alpha"   # Also "non-alpha" if no argument passed.
fi
}

a=23skidoo
b=H3llo
c=-What?
d='echo $b'    # Command substitution.

check_var $a
check_var $b
check_var $c
check_var $d
check_var      # No argument passed, so what happens?

# Script improved by S.C.
exit 0
```

select

Korn Kabuğu tarafından benimsenen **select** yapısı, menü oluşturmada kullanılabilecek başka bir araçtır.

select *değişken* [in *liste*]

do

command...

break

done

Bu kod, değişken listesi seçeneklerinden birini girmesi için kullanıcıya sorar. Unutmayınız ki, değişebilse de PS3 istemini (yani varsayılan #?) kullanır.

ÖRNEK 10.28 select KULLANARAK MENÜ OLUŞTURMAK

```
#!/bin/bash

PS3='Choose your favorite vegatable: '   # Sets theprompt string.

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"

do

  echo

  echo "Your favorite veggie is $vegetable."

  echo "Yuck!"

  echo

  break   # if no ' break' here, keeps looping forever.

done

exit 0
```

in *liste* yazılmadığı takdirde, **select** yapısının içine gömüldüğü fonksiyona veya betiğe geçirilen komut satırı argümanları arasından seçim yapılır.

Bunu, aşağıdaki kod yapısında **in** *liste* yazılmadığı takdirde olması gereken davranış ile karşılaştırınız.

for *değişken* [**in** *liste*]

ÖRNEK 10.29 MENÜ OLUŞTURMAK İÇİN BİR FONKSİYONDA select KULLANIMI

```
#!/bin/bash

PS3='Choose your favorite vegetable: '

echo

choice_of()

{

select vegetable

# [in list] omitted, so ' select' uses arguments passed to function.

do

  echo

  echo "Your favorite veggie is $vegetable."

  echo "Yuck!"

  echo

  break

done

}

Choice_of beans rice carrots radishes tomatoes spinach

#         $1    $2   $3      $4        $5        $6

#         passed to choice_of() function

exit 0
```

bkz.Örnek 35.3.

Bölüm 12 İÇ KOMUTLAR ve YERLEŞİKLER

İç yerleşikler Bash araç seti içinde yer alır. Bunun nedeni birinci olarak, yerleşikler dış komutlardan daha hızlı çalışır, böylece performansları daha büyüktür. İkincisi, dış komutlar kendilerinden farklı olarak başka bir süreci başlatmakta ve bununla birlikte kabuğun içsel komutlarına direkt erişim ihtiyacı duymaktadırlar.

Bir komut veya kabuğun kendisi bir görevi yürütmek için yeni bir alt işlem başlatabilir. Bu süreç çatallama olarak bilinir. Bu yeni sürece "çocuk" ve onu üreten yani çatallayan sürece "ana" denir. Çocuk süreç çalışmasını sürdürdüğü zaman boyunca, ana süreç de varlığını sürdürüyor olmalıdır.

Genellikle bir Bash yerleşiği, betik içinde çalışırken bir alt süreci başlatmaz. Betik içinde bir alt sürecin çatallanmasına neden olarak çoğunlukla dış sistem komutları verilir.

Sistem komutu bir yerleşikle eş anlamlı, yani aynı adı taşıyabilir, ancak Bash onu içsel olarak yeniden gerçekleştirecektir. Örneğin Bash **echo** komutu, aynı davranışı üretmesine rağmen `/bin/echo` ile aynı kategoride değildir.

```
#!/bin/bash

echo “This line uses the \”echo\” builtin.”

/bin/echo “This line uses the /bin/echo system command.”
```

Bir anahtar sözcük ya özel amaçlı sözcük, ya simge, veya operatördür. Anahtar sözcüklerin kabuk için her zaman özel bir anlamı vardır, ve gerçekten kabuğun söz dizimini yaratan temel yapı taşlarından biri oldukları bilinir. Örneğin, "for", "while", "do", ve "!" anahtar sözcüklerdir. *Yerleşikle* benzer olarak, anahtar sözcük Bash içine sabit olarak kodlanır, yerleşiklerden farklı olarak anahtar kendi başına bir komut değildir, ve bu nedenle daha kapsamlı bir komut yapısının parçasıdır. [1]

I/O

echo

Bir ifade veya değişkeni `stdout` aygıtına yazar (bkz.Örnek 5.1).

```
echo Hello

echo $a
```

echo komutu kaçış karakterlerini yazdırmak için `-e` seçeneğini gerektirir. bkz.Örnek 6.2.

Normalde her **echo** komutu, bir satır sonu karakteriyle birlikte terminale yazar, ancak `-n` seçeneği bu karakterin yazılmasına izin vermez.

echo komutu oluktan aşağı bazı komutları besleyebilme özelliğine sahiptir.

```
if echo "$VAR" | grep -q txt    # if [[ $VAR = *txt* ]]

then

  echo "$VAR contains the substring sequence \"txt\""

fi
```

Komut yer değişimi ile birlikte kullanılırsa, **echo** komutu bir değişkeni atama özelliğine sahiptir.

```
a='echo "HELLO" | tr A-Z a-z'
```

bkz.Örnek 12.15, Örnek 12.2, Örnek 12.32, ve Örnek 12.33.

Dikkat etmelisiniz ki, **echo 'komut'** komut tarafından çıktısı olan bütün linefeed karakterleri siler.

$IFS (iç alan ayıracı) değişkeni normal olarak boşluk karakterlerinin kümelerinden biri olarak \n (linefeed) karakterini içerir. Bu nedenle, *komut* çıktısı ne zaman linefeed karakterine rastlarsa, kesintiye uğratılır çünkü argüman olarak **echo** komutuna dönülmelidir. **echo** bu argümanları çıktı olarak boşluk karakteriyle yer değiştirilmiş halde yazar.

```
bash$ ls –l /usr/share/apps/kjezz/sounds

-rw-r--r--          1 root     root      1407  Nov 7 2000 reflect.au

-rw-r--r--           1 root     root      362   Nov 7 2000 seconds.au


bash$ echo ‘ ls –l /usr/share/apps/kjezz/sounds’

total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au –rw-r—r-- 1 root
root 362 Nov 7 2000 seconds.au
```

Kabuk yerleşiği olarak bu komut `/bin/echo` ile aynı değildir. Ancak gerçekleşim olarak Bash tarafından ele alınan /bin/echo komutu aynı çıktıyı üretir.

```
bash$ type –a echo

echo is a shell builtin

 echo is /bin/echo
```

printf

printf print komutunun biçimlendirilmiş halidir; **echo** komutunun gelişmiş halidir. C dili kütüphane fonksiyonunun sınırlı bir değişiğidir, ve söz dizimi bakımından farklı kullanım alanları vardır.

printf *karakter-dizesi-biçimi...parameter...*

Bu `/bin/printf` veya `/usr/bin/printf` komutlarının Bash yerleşik sürümüdür. Detaylı anlatım için bkz. **printf** yardım sayfası (sistem komutu) .

Bash eski sürümlerinin destekleyip desteklemediği bilinmemektedir.

ÖRNEK 11.1. printf KULLANIMI

```
#!/bin/bash

# printf demo


PI=3.14159265358979

DecimalConstant=31373

Message1="Greetings,"

Message2="Earthling."


echo


printf "Pi to 2 decimal places = %1.2f" $PI

echo

printf "Pi to 9 decimal places = %1.9f" $PI   # It even rounds off
correctly.

printf "\n"                                    # Prints a line feed,

                                               # equivalent to ' echo'.

printf "Constant = \t%d\n" $DecimalConstant   # Inserts tab (\t)

printf "%s %s \n" $Message1 $Message2

echo

# ============================================#

# Simulation of C function, ' sprintf'.

# Loading a variable with a formatted string.

echo

Pi12=$(printf "%1.12f" $PI)

echo "Pi to 12 decimal places = $Pi12"

Msg='printf "%s %s \n" $Message1 $Message2'

echo $Msg; echo $Msg

# As it happens, the ' sprintf' function can now beaccessed

# as a loadable module to Bash, but this is not portable.

exit 0
```

Hata mesajlarının biçimlendirilerek yazdırılması **printf** komutunun sıkça kullanılmasına bir örnektir.

```
E_BADDIR=65

var=nonexistent_directory

error()

{

  printf "$@" >&2

  # Formats positional params passed, and sents them to stderr.

  echo

  exit $E_BADDIR

}

cd $var || error $"Can't cd to %s." "$var"

# Thanks, S.C.
```

read

`stdin` aygıtından bir değişkenin değerini "okur", yani klavyeden gelen verinin okunmasına yarar. Bir dizi içindeki değerlerin okunması için `-a` seçeneği kullanılır (bkz.Örnek 26.2).

ÖRNEK 11.2 DEĞİŞKENE read KOMUTU İLE ATAMA YAPILMASI

```
#!/bin/bash

echo -n "Enter the value of variable ' var1': "

# The -n option to echo suppresses newline.

read var1

# Note no ' $' in front of var1, since it is being set.

echo "var1 = $var1"


echo

# A single ' read' statement can set multiple variables.
```

```
echo –n “Enter the values of variables ‘ var2’ and ‘var3’ (separated by a
space or tab): “

read var2 var3

echo “var2 = $var2       var3 = $var3”

# If you input only one value, the other variables(s) will remain unset

# (null).

exit 0
```

Komutla ilişkili olmayan bir değişken, **read** komutu ile birlikte kullanılırsa, girdi olarak alınan veri alınarak, adanmış bir değişken olan $REPLY'e atanabilir.

ÖRNEK 11.3 read KOMUTU DEĞİŞKENSİZ KULLANILIRSA NE OLUR?

```
#!/bin/bash

echo

# ---------------------------#

# First code block.

echo –n “Enter a value: “

read var

echo “\”var\” = “$var””

# Everything as expected here.

# ---------------------------#


echo

echo –n “Enter another value: “

read            #  No variable supplied for ‘ read’,therefore...

                #+ Input to ‘ read’ assigned to default variable, $REPLY.

var=”$REPLY”

echo “\”var\” = “$var””

# This is equivalent to the first code block.

echo

exit 0
```

Normal olarak \ karakterinin girilmesiyle, `read` komutu okuduğu `newline` karakterin yazılmasına izin vermez. `-r` seçeneği `\` karakterini olduğu gibi, yani harf olarak yazdırır.

ÖRNEK 11.4. read KOMUTUNUN ÇOKLU-SATIR GİRDİ OKUMASI

```
#!/bin/bash

echo

echo "Enter a string terminated by a \\, then press <ENTER>."

echo "Then, enter a second string, and again press <ENTER>."

read var1     # The "\" suppresses the newline, when reading "var1".

              #     first line \

              #     second line

echo "var1 = $var1"

#     var1 = first line second line

# For each line terminated by a "\",

# you get a prompt on the next line to continue feeding characters into

# var1.

echo; echo

echo "Enter another string terminated by a \\ , then press <ENTER>."

read -r var2  # The -r option causes the "\" to be read literally.

              #     first line \

echo "var2 = $var2"

#     var2 = first line \

# Data entry terminates with the first <ENTER>.

echo


exit 0
```

read komutunun ilginç bazı özellikleri vardır, aralarında istem satırı, ve hatta ENTER tuşuna basılmaksızın klavye hareketlerini okuyabilme sayılabilir.

```
# Read a keypress without hitting ENTER.

read –s –n1 –p “Hit a key “ keypress

echo; echo “Keypress was “\”$keypress\””.”

# -s option means do not echo input.

# -n N option means accept only N characters of input.

# -p option means echo the following prompt before reading input.

# Using these options is tricky, since they need to be in the correct

# order.
```

`-t` seçeneği zamanlamalı girdi okur (bkz.Örnek 9.4).

read komutunun bir başka özelliği, değişken değeri olarak `stdin` aygıtına yönlendirilmiş olan bir dosyadan veri kabul etmesidir. Dosya birden fazla satır içeriyorsa, yalnızca ilk satırda bulunan veri değişkene atanır. Birden fazla parametre okunacaksa, her değişkene boşluk karakteriyle ayrılan bir karakter dizesi değer olarak atanır. Dikkat şarttır!

ÖRNEK 11.5 read KOMUTUNUN DOSYA YÖNLENDİRME YAPARKEN KULLANIMI

```
#!/bin/bash

read var1 <data-file

echo “var1 = $var1”

# var1 set to the entire first line of the input file “data-file”


read var2 var3 <data-file

echo “var2 = $var2   var3 = $var3”

# Note non-intuitive behavior of “read” here.

# 1) Rewinds back to the beginning of input file.

# 2) Each variable is now set to a corresponding string,

#    separated by whitespace, rather than to an entire line of text.

# 3) The final variable gets the remainder of the line.

# 4) If there are more variables to be set than whitespace-terminated

#    strings on the first line of the file, then the excess variables

#    remain empty.
```

```
echo "------------------------------------------------"
# How to resolve the above problem with a loop:

while read line
do
   echo "$line"
done <data-file
# Thanks, Heiner Steven for pointing this out.
echo "------------------------------------------------"

# Use $IFS (Internal File Separator variable) to split a line of input to
# "read", if you do not want the default to be whitespace.

echo "List of all users:"
OIFS=$IFS; IFS=:       # /etc/passwd uses ":" for field separator.
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O redirection.
IFS=$OIFS              # Restore original $IFS.
# This code snippet also by Heiner Steven.
exit 0
```

read komutuna **oluk**tan çıktı iletmek, veya **echo** komutuyla değişkenlere atama yapmak mümkün değildir.

cat komutunun çıktısı oluğa verildiğinde, bu kod doğru çalışacaktır.

```
cat file1 file2 |
while read line
do
echo $line
done
```

Dosya sistemi

cd

Alışılagelen `cd` (dizin değiştir) komutu belirli bir dizinle çalışması gereken betikler tarafından kullanılır.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[Alan Cox tarafından daha önce sözü edilen örnekten alınmıştır.]

`cd` komutundaki `-P` (fiziksel) seçeneği sembolik bağların yok sayılmasına neden olur.

`cd -` komutu `$OLDPWD` ile belirtilen önceki çalışma dizinine geçer.

pwd

Çalışma dizinini yazdırır. Bu, kullanıcının (ya da betiğin) geçerli dizinini (bkz.Örnek 11.6) verir. Etkisi, yerleşik değişken olan `$PWD` değerinin okunması ile aynıdır.

pushd, popd, dirs

Bu komut seti çalışma dizinlerini imlemek için geliştirilmiş bir mekanizmadır, yani dizinler arasında düzenli bir şekilde ileri ve geri hareket etmeye yarayan bir araçtır. Son-giren-ilk çıkar yığıtı dizin isimlerini takip etmede kullanılır. Komut seçenekleri, dizin yığınlarının çeşitli yönlendirilmelerine izin verir.

`pushd dizin-adı` komutu, *`dizin-adı`* yolunu dizin yığıtına ekler ve aynı anda geçerli çalışma dizini *`dizin-adı`* olarak değişir.

popd dizin yığıtının en üstünde yer alan dizin yolu adını alarak siler ve aynı anda geçerli çalışma dizini yığıttan alınan o dizin olarak değişir.

dirs dizin yığıtının içeriğini listeler (`$DIRSTACK` değişkeni ile karşılaştırın). Başarılı bir şekilde çalışan **pushd** veya **popd** komutları otomatik olarak **dirs** komutunu çağıracaktır.

Geçerli çalışma dizini üzerindeki dizin adı değişikliklerini sabit kodlamaksızın çeşitli değişiklikler yapmayı gerektiren betikler, bu komutlardan hallice yararlanabilirler. Unutmayınız ki, bir komut dosyası içinden erişilebilir örtük `$DIRSTACK` değişkeni, dizin yığınının içeriğini tutmaktadır.

ÖRNEK 11.6 GEÇERLİ ÇALIŞMA DİZİNİNİN DEĞİŞTİRİLMESİ

```
#!/bin/bash

dir1=/usr/local

dir2=/var/spool

pushd $dir1

# Will do an automatic ‘ dirs’ (list directory stackto stdout).

echo “Now in directory ‘ pwd’.” # Uses back-quoted pwd’.

# Now, do some stuff in directory ‘ dir1’.

pushd $dir2

echo “Now in directory ‘ pwd’.”

# Now, do some stuff in directory ‘ dir2’.

echo “The top entry in the DIRSTACK array is $DIRSTACK.”

popd

echo “Now back in directory ‘ pwd’.”

# Now, do some more stuff in directory ‘ dir1’.

popd

echo “Now back in original working directory ‘ pwd’.”

exit 0
```

Değişkenler

let

let komutu değişkenler üzerinde aritmetik işlemler yapar. Birçok durumda, **expr** ifadesinin daha az karmaşık bir versiyonu gibi işlev görür.

ÖRNEK 11.7 let KOMUTUYLA ARİTMETİK İŞLEM YAPABİLME

```
#!/bin/bash

echo

let a=11          # Same as ‘ a=11’

let a=a+5         # Equivalent to let “a = a + 5”

                  # (double quotes and spaces make it more readable)

echo “11 + 5 = $a”

let “a <<= 3”     # Equivalent to let “a = a << 3”

echo “\”\$a\” (=16) left-shifted 3 places = $a”

echo “a /= 4”     # Equivalent to let “a = a / 4”

echo “128 / 4 = $a”

let “a -= 5”      # Equivalent to let “a = a – 5”

echo “32 – 5 = $a”

let “a = a * 10”  # Equivalent to let “a = a * 10”

echo “27 * 10 = $a”

let “a %= 8”      # Equivalent to let “a = a % 8”

echo “270 modulo 8 = $a (270 / 8 = 33, remainder $a)”

echo

exit 0
```

eval

`eval arg1 [arg2] ... [argN]`

Bir liste halindeki argümanları komutlara çevirmektedir (bir komut dosyası içinde kod oluşturma için kullanışlıdır).

ÖRNEK 11.8 eval KOMUTUNUN ETKİSİNİN GÖSTERİLMESİ

```
#!/bin/bash

y='eval ls -l' # Similar to y='ls -l'

echo $y      # but linefeeds removed because "echoed" variable is unquoted.

echo

echo "$y"      # Linefeeds preserved when variable is quoted.

echo; echo

y='eval df'    # Similar to y='df'

echo $y        # but linefeeds removed.

#  When LF's not preserved, it may make it easier to parse output,

#+ using utilities such as "awk".

exit 0
```

ÖRNEK 11.9 ÇIKIŞIN ZORLANMASI

```
#!/bin/bash

y='eval ps ax | sed -n ' /ppp/p' | awk ' { print $1 }'

# Finding the process number of ' ppp'.

kill -9 $y    # Killing it

# Above lines may be replaced by

#  kill -9 ' ps ax | awk ' /ppp/ { print $1 }'

chmod 666 /dev/ttyS3

# Doing a SIGKILL on ppp changes the permissions

# on the serial port. Restore them to previous state.

rm /var/lock/LCK..ttyS3   # Remove the serial port lock file.

exit 0
```

ÖRNEK 11.10 BİR "rot13" SÜRÜMÜ

```
#!/bin/bash
# A version of "rot13" using ' eval'.
# Compare to "rot13.sh" example.
setvar_rot_13()               # "rot13" scrambling
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}

setvar_rot_13 var "foobar"  # Run "foobar" through rot13.
echo $var                   # sbbone
echo $var | tr a-z n-za-m   # foobar
                            # Back to original variable.
# This example by Stephane Chazelas.
exit 0
```

eval komutu riskli olabilir, ve makul bir alternatifinin var olması durumunda kullanımdan kaçınılmalıdır. **`eval $KOMUTLAR`** komutundaki *$KOMUTLAR* içeriği **rm –rf *** gibi bazı tatsız sürprizler içerebilse de, olduğu gibi çalıştırılır. Bilinmeyen kişiler tarafından yazılmış bilinmeyen kod üzerinde **eval** komutunu çalıştırmak çok tehlikeli olabilir.

set

set komutu iç betik değişkenlerinin değerini değiştirir. Bir kullanım alanı, betik davranışını belirlemeye yardımcı seçenek bayrakları arasında geçiş yapmaktır. Başka bir uygulaması, **`set ' command'`** gibi bir komutun sonucunu gören bir betiğin konumsal parametrelerini ilk duruma getirmektir. Komut daha sonra komut çıktı alanlarını ayrıştırabilir.

ÖRNEK 11.11 KONUMSAL PARAMETRELER İLE set KOMUTUNU KULLANMAK

```
#!/bin/bash

# script "set-test"

# Invoke this script with three command line parameters,

# for example, "./set-test one two three".

echo

echo "Positional parameters before set \'uname –a\' :"

echo "Command-line argument #1 = $1"

echo "Command-line argument #2 = $2"

echo "Command-line argument #3 = $3"

echo

set ' uname –a'  # Sets the positional parameters tothe output

                 # of the command ' uname –a'

echo "Positional parameters after set \'uname –a\' :"

# $1, $2, $3, etc. reinitialized to result of ' uname –a'

echo "Field #1 of ' uname –a' = $1"

echo "Field #2 of ' uname –a' = $2"

echo "Field #3 of ' uname –a' = $3"

echo

exit 0
```

`set` komutunu seçeneksiz veya argümansız çağırmak, başlangıç durumuna getirilen tüm çevresel ve diğer değişkenleri listeler.

```
bash$ set

AUTHORCOPY=/home/bozo/posts

 BASH=/bin/bash

 BASH_VERSION=$'2.05.8(1)-release'

 ...

 XAUTHORITY=/home/bozo/.Xauthority

_=/etc/bashrc

variable22=abc

variable23=xzy
```

`set` komutunu `--` seçeneği ile kullanmak bir değişkenin içeriğini konumsal parametrelere açık bir şekilde atar. `--` seçeneğini izleyen hiçbir değişken yoksa, konumsal parametrelerin değeri belirsiz hale getirilir.

ÖRNEK 11.12 KONUMSAL PARAMETRELERİN YENİDEN ATANMASI

```
#!/bin/bash

variable="one two three four five"

set -- $variable

# Sets positional parameters to the contents of "$variable".

first_param=$1

second_param=$2

shift; shift         # Shift past first two positional params.

remaining_params="$*"

echo

echo "first parameter = $first_param"              # one

echo "second parameter = $second_param"            # two

echo "remaining parameters = $remaining_params"    # three four five

echo; echo
```

```
# Again.
set -- $variable
first_param=$1
second_param=$2
echo "first parameter = $first_param"             # one
echo "second parameter = $second_param"           # two
# ===================================================
set --
# Unsets positional parameters if no variable specified.
first_param=$1
second_param=$2
echo "first parameter = $first_param"             # (null value)
echo "second parameter = $second_param"           # (null value)

exit 0
```

bkz. Örnek 10.2 ve Örnek 12.39.

unset

unset komutu bir kabuk değişkeninin değerini null yaparak siler. Unutmayınız ki, bu komut konumsal parametreleri etkilemez.

```
bash$ unset PATH
bash$ echo $PATH
bash$
```

ÖRNEK 11.13 BİR DEĞİŞKENİN DEĞERİNİ SIFIRLAMAK

```
#!/bin/bash

# unset.sh: Unsetting a variable.

variable=hello                          # Initialized.

echo "variable = $variable"

unset variable                          # Unset.

                                        # Same effect as   variable=

echo "(unset variable = $variable"    # $variable is null.

exit 0
```

export

export komutu çalışan betik veya kabuğun tüm alt süreçleri tarafından kullanılabilir değişkenler yapar. Ne yazık ki, değişkenleri betik ya da kabuğu çağıran ana sürece geri vermek için hiçbir yol yoktur. **export** komutunun önemli kullanım alanlarından biri de başlangıç dosyalarıdır, çevre değişkenlerini başlangıç durumuna getirmek (sıfırlamak) ve izleyen kullanıcı süreçleri tarafından erişilebilir kılmak sayılabilir.

ÖRNEK 11.14 GÖMÜLÜ awk KOMUT DOSYASINA BİR DEĞİŞKEN GEÇİRMEK İÇİN export KULLANILMASI

```
#!/bin/bash

# Yet another version of the "column totaler" script (col-totaler.sh)

# that adds up a specified column (of numbers) in the target file.

# This uses the environment to pass a script variable to ' awk'.

ARGS=2

E_WRONGARGS=65

if [ $# -ne "$ARGS" ]  # Check for proper no. of command line args.

then
```

```
   echo "Usage: ' basename $0' filename column-number"
   exit $E_WRONGARGS
fi
filename=$1
column_number=$2
#===== Same as original script, up to this script =====#
export column_number
# Export column number to environment, so it's available for retrieval.

# Begin awk script.
# ------------------------------------------------
awk ' { total += $ENVIRON["column_number"]
}
END { print total }' $filename
# ------------------------------------------------
# End awk script.

# Thanks, Stephane Chazelas.
exit 0
```

`export var1=xxx` örneğinde olduğu gibi, tek işlemde değişkenleri sıfırlamak ve dış ortama taşımak mümkündür.

declare, typeset

declare ve **typeset** komutları değişkenlerin özelliklerini belirtmeye ve/veya kısıtlamaya yarar.

readonly

declare –r ile aynıdır, bir değişkeni salt-okunur veya sabit kılar. Değişkeni değiştirme girişimleri bir hata mesajı ile başarısızlıkla sonuçlanacaktır. Bu C dili const tür niteleyicisinin kabuk analoğudur.

getopts

Bu güçlü araç, komut dosyasına geçirilen komut satırı argümanlarını ayrıştırır. Bu getopt dış komutunun ve C programcılarına tanıdık olan **getopt** kütüphane fonksiyonunun Bash analoğudur. Birden fazla seçeneği [2] ve ilişkili argümanları birleştirerek bir komut dosyasına geçirmeye izin verir (örneğin **`scriptname -abc -e /usr/local`**).

getopts yapısı iki örtük değişken kullanır. `$OPTIND` argüman göstericidir *(OPTsiyön INDeksi)* ve `$OPTARG` *($OPTsiyön ARGümanı)* bir seçeneğe eklenmiş olan (opsiyonel) argümandır. Bildirimde seçenek adını takip eden iki nokta üst üste, bu seçenekle ilişkili bir argümanla seçeneği etiketler.

getopts yapısı genellikle bir while döngüsü içinde paketlenmiş olarak gelir. Bu döngü, seçenekleri ve argümanları birer birer işler, ardından sonraki adıma geçmek için örtük `$OPTIND` değişkeni bir azaltılır.

1. Komut satırından komut dosyasına (betiğe) geçirilen argümanların önüne birer eksi (-) veya artı (+) gelmelidir. – veya + öneki **getopts** komutunun komut satırı argümanlarını *seçenek* olarak tanımasını sağlar. Aslında, **getopts** – veya + öneki almayan argümanları işleme koymayacaktır ve öneki olmayan ilk argümandan itibaren seçenek işlemlerini sonlandıracaktır.
2. **getopts** şablonu standart **while** döngüsünden biraz farklıdır, çünkü koşul parantezinden yoksundur.
3. **getopts** yapısı eski ve daha az güçlü **getopt** dış komutunun yerini almıştır.

```
while getopts ":abcde:fg" Option
# Initial declaration.
# a, b, c, d, e, f, and g are the options (flags) expected.
# The : after option ' e' shows it will have an argument passed with it.
do
  case $Option in
    a ) # Do something with variable ' a'.
    b ) # Do something with variable ' b'.
    ...
    e ) # Do something with ' e', and also with $OPTARG,
        # which is the associated argument passed with option ' e'.
    ...
```

```
    g) # Do something with variable ' g'.
  esac
done
shift $(($OPTIND - 1))
# Move argument pointer to next.
# All this is not nearly as complicated as it looks <grin>.
```

ÖRNEK 11.15 BİR KOMUT DOSYASINA GEÇİRİLEN SEÇENEKLERİ / DEĞİŞKENLERİ OKUMANIZ İÇİN getopts KULLANIMI

```
#!/bin/bash
# ' getopts' processes command line arguments to script.
# The arguments are parsed as "options" (flags) and associated arguments.
# Try invoking this script with
# ' scriptname -mn'
# ' scriptname -oq qOption' (qOption can be some arbitrary string.)
# ' scriptname -qXXX -r'
#
# ' scriptname -qr'    - Unexpected result, takes "r" as the argument to
# option "q"
# ' scriptname -q -r'  - Unexpected result, same as above
#  If an option expects an argument ("flag:"), then it will grab
#  whatever is next on the command line.
NO_ARGS=0
E_OPTERROR=65
if [ $# -eq "$NO_ARGS" ] # Script invoked with no command-line args?
then
  echo "Usage: ' basename $0' options (-mnopqrs)"
  exit $E_OPTERROR       # Exit and explain usage, if no argument(s) given.
```

```
fi

# Usage: scriptname –options

# Note: dash (-) necessary


while getopts ":mnopq:rs" Option

do

  case $Option in

    m     ) echo "Scenario #1: option –m-";;

    n | o ) echo "Scenario #2: option –$Option-";;

    p     ) echo "Scenario #3: option –p-";;

    q     ) echo "Scenario #4: option –q-, with argument \"$OPTARG\"";;

    # Note that option ' q' must have an associated argument,

    # otherwise it falls through to the default.

    r | s ) echo "Scenario #5: option –$Option-"'';;

    *     ) echo "Unimplemented option chosen.";;

  esac

done

shift $(($OPTIND – 1))

# Decrements the argument pointer so it points to next argument.


exit 0
```

Betik Davranışı

kaynak, . (nokta komutu)

Bu komut, komut satırından çağrıldığında, bir komut dosyasını çalıştırır. Bir komut dosyası içinde, bir **`kaynak dosya-adı`**, `dosya-adı` dosyasını yükler. Bu bir C/C ++ **`#include`** direktifinin kabuk komut dosyası eşdeğerdir. Birden fazla komut dosyasının ortak bir veri dosyası veya fonksiyon kütüphanesi kullanması durumlarında yararlıdır.

ÖRNEK 11.16 BİR VERİ DOSYASININ "DAHİL EDİLMESİ"

```
#!/bin/bash
. data-file    # Load a data file.
# Same effect as "source data-file", but more portable.
#  The file "data-file" must be present in current working directory,
#+ since it is referred to by its ' basename'.
# Now, reference some data from that file.
echo "variable1 (from data-file) = $variable1"
echo "variable3 (from data-file) = $variable3"
let "sum = $variable2 + $variable4"
echo "Sum of variable2 + variable4 (from data-file) = $sum"
echo "message1 (from data-file) is \"$message1\""
# Note:                              escaped quotes
print_message This is the message-print function in the data-file.
exit 0
```

Yukarıdaki Örnek 11.16 için dosya `veri-dosyası`. Aynı dizinde bulunması gerekir.

```
# This is a data file loaded by a script.
# Files of this type may contain variables, functions, etc.
# It may be loaded with a ' source' or ' .' commandyba shell script.
# Let's initialize some variables.
variable1=22
variable2=474
variable3=5
variable4=97
message1="Hello, how are you?"
message2="Enough for now. Goodbye."
```

```
print_message ()
{
# Echoes any message passed to it.
  if [ -z "$1" ]
  then
    return 1
    # Error, if argument missing.
  fi
  echo
  until [ -z "$1" ]
  do
    # Step through arguments passed to function.
    echo –n "$1"
    # Echo args one at a time, suppressing line feeds.
    echo -n " "
    # Insert spaces between words.
    shift
    # Next one.
  done
  echo
  return 0
}
```

exit

Bir betiği kayıtsız ve şartsız olarak sona erdirir. **exit** komutu isteğe bağlı olarak bir tamsayı argüman alabilir, bu tamsayı betiğin çıkış durumu olarak kabuğa döndürülür. Tüm betikleri **`exit 0`** ile sonlandırmak iyi bir alışkanlıktır, bu kod başarılı çıkış kodu olarak bilinir ve betiğin başarılı çalıştığını gösterir.

Bir komut dosyası, argümansız olarak **exit** ile sonlandırılırsa, betiğin çıkış durumu, **exit** komutunu saymazsak betiğin çalıştırdığı son komutun çıkış durumu ile aynıdır.

exec

Bu kabuk yerleşiği mevcut süreci belirli bir komut ile yer değiştirir. Normal olarak, kabuk bir komut ile karşılaştığında, komutu gerçekten yürütmek için bir çocuk süreç yaratır. **exec** yerleşiği kullanıldığında, kabuk hiçbir şey yaratmaz ve **exec**'edilen komut kabuk ile yer değiştirir. Bir komut dosyası içinde kullanıldığında, bu nedenle, **exec**'edilen komut sonlandırıldığında, betikten çıkışa zorlanır. **exec** bir komut dosyası içinde göründüğünde, bu nedenle, muhtemelen son komut olacaktır.

ÖRNEK 11.17 exec ETKİLERİ

```
#!/bin/bash

exec echo “Exiting \”$0\”.”   # Exit from script here.

# ---------------------------------

# The following lines never execute.

echo “This echo will never echo.”

exit 99                       #  This script will not exit here.

                              #  Check exit value after script terminates

                              #+ with an ‘ echo $?’.

                              #  It will *not* be 99.
```

ÖRNEK 11.18 KENDİ KENDİNİ exec YAPAN BETİK

```
#!/bin/bash

# self-exec.sh


echo

echo “This line appears ONCE in the script, yet it keeps echoing.”

echo “The PID of this instance of the script is stil $$.”

#     Demonstrates that a subshell is not forked off.
```

```
echo  "====================== Hit Ctrl-C to exit ================"
sleep 1
exec $0   #  Spawns another instance of this same script
          #+ that replaces the previous one.
echo "This line will never echo!"  # Why not?
exit 0
```

Bir **exec**, dosya tanıtıcılarını yeniden atamak için de hizmet vermektedir. **exec <zzz-dosya,** `zzz-dosya` ile `stdin`'i yer değiştirir (bkz. Örnek 16.1).

find komutuyla birlikte `–exec` seçeneğini kullanmak, **exec** kabuk yerleşiği ile aynı *değildir*.

shopt

Bu komut, kabuk seçeneklerini anında değiştirmeye izin verir (bkz. Örnek 24-1 ve Örnek 24-2). Genellikle Bash başlangıç dosyalarında görünür, ama aynı zamanda komut dosyalarında da kullanımları vardır. Bash sürüm 2 veya daha ilerisine ihtiyacı vardır.

```
shopt –s cdspell
# Allows minor misspelling directory names with ' cd'
command.
```

Komutlar

true

Başarılı bir (sıfır) çıkış durumu döndüren bir komuttur, ama başka hiçbir şey yapmaz.

```
# Endless loop
while true   # alias for ":"
do
    operation-1
    operation-2
    ...
    operation-n
    # Need a way to break out of loop.
done
```

false

Başarısız bir çıkış durumu döndüren bir komuttur, ama başka hiçbir şey yapmaz.

```
# Null loop
while false
do
   # The following code will not execute.
   operation-1
   operation-2
   …
   operation-n
   # Nothing happens!
done
```

type [cmd]

which dış komutuna benzer şekilde, **type cmd** "cmd" için tam yol adını verir. **which** komutundan farklı olarak type bir Bash yerleşiğidir. type ile birlikte kullanılan –a seçeneği *anahtar kelimeleri* ve *yerleşikleri* belirler, ve aynı isimlerdeki sistem komutlarını da bulur.

```
bash$ type ' ['

[ is a shell builtin

bash$ type -a ' ['

[ is a shell builtin

 [ is /usr/bin/[
```

hash [cmds]

Belirtilen komutların yol adlarını (kabuk adres tablosunda) kaydeder, böylece kabuk veya komut dosyasının bu komutlara yapılan sonraki çağrılarında `$PATH` değişkenini araması gerekmez. Argümansız çağrıldığında **hash**, sadece adreslenen komutları listeler. `-r` seçeneği adres tablosunu sıfırlar.

help

help KOMUT kabuk yerleşik KOMUT'unun kısa bir kullanım özetini arar. Yerleşikler için bu whatis'in bir karşılığıdır.

```
bash$ help exit

exit: exit [n]

    Exit the shell with a status of N. If N is omitted, the exit status

    is that of the last command executed.
```

Notlar

[1] Burada kural dışı olan time komutudur, komut olduğu halde resmi Bash dokümantasyonu tarafından anahtar olarak tanınır.

[2] Seçenek, betik davranışlarını açıp kapayan bir bayrak olarak davranan bir argümandır. Belli bir seçenek ile ilişkili argüman, bayrağın açık veya kapalı olan davranışını belirtir.

11.1. İş Kontrol Komutları

Aşağıdaki belirli bazı iş kontrol komutları argüman olarak bir "iş kimliği" alır. Bölümün sonundaki tabloya bakınız.

jobs

İş numarası vererek, arka planda çalışan işleri listeler. **ps** kadar yararlı değildir.

İş ve işlem süreci kavramlarını birbirine karıştırmak çok kolaydır. **kill, disown** ve **wait** gibi bazı yerleşikler argüman olarak ya bir iş numarası veya bir işlem süreci numarası ister. **fg**, **bg** ve **jobs** komutları sadece bir iş numarası kabul eder.

```
bash$ sleep 100 &

[1] 1384

bash$ jobs

[1]+ Running                 sleep 100 &
```

"1" iş numarasıdır (işler mevcut kabuk tarafından korunur), ve "1384" işlem süreci numarasıdır (süreçler sistem tarafından korunur). Bu iş/süreci öldürmek için **kill %1** veya **kill 1384** komutlarından her ikisi de çalışır.

Teşekkürler, S.C.

disown

Kabukta bulunan etkin işler tablosundan iş(ler)i çıkarır, siler.

fg, bg

fg komutu arka planda çalışan bir işi ön plana geçirir. **bg** komutu askıda kalan bir işi yeniden başlatır ve arka planda çalıştırır. Hiçbir iş numarası belirtilmezse, **fg** veya **bg** komutu şu anda çalışan iş üzerinde hareket eder.

wait

Arka planda çalışan tüm işler sona erene kadar veya seçenek olarak belirtilen iş numarası veya işlem kimliği sonlandırılana kadar betiğin çalışmasını durdurur. Sonlandırılması beklenen komutun çıkış durumunu döndürür.

Bir arka plan işin yürütülmesi tamamlanmadan (bu korkunç bir yetim süreç yaratacaktır) bir komut dosyasının çıkmasını önlemek için **wait** komutunu kullanabilirsiniz.

ÖRNEK 11.19 DEVAM ETMEDEN ÖNCE BİR İŞLEM SÜRECİNİN BİTİRİLMESİNİ BEKLEMEK

```
#!/bin/bash

ROOT_UID=0    # Only users with $UID 0 have root privileges

E_NOTROOT=65

E_NOPARAMS=66


if [ “$UID” –ne “$ROOT_UID” ]

then

  echo “Must be root to run this script.”

  # “Run along kid, it’s past your bedtime.”

  exit $E_NOTROOT

fi


if [ -z “$1” ]

then

  echo “Usage: ‘ basename $0’ find-string”

  exit $E_NOPARAMS

fi


echo “Updating ‘ locate’ database...”

echo “This may take a while.”
```

```
Updatedb /usr &     # Must be run as root.
wait
# Don't run the rest of the script until ' updatedb'finished.
# You want the database updated before looking up the file name.
locate $1
# Without the wait command, in the worse case scenario,
# the script would exit while ' updatedb' was still unning,
# leaving it as an orphan process.
exit 0
```

İsteğe bağlı olarak, **wait** argüman olarak bir iş kimliği alabilir, örneğin, **wait %1** veya **wait $PPID**. İş kimliği tablosuna bakınız.

Bir komut dosyası içinde, bir komutu arka planda (&) ile çalıştırmak, **ENTER** tuşuna basılana kadar komut dosyasının askıda kalmasına neden olabilir. Bunun daha çok, `stdout`'a yazan komutlarda meydana geldiği görülmektedir. Bu önemli bir sıkıntı olabilir.

```
#!/bin/bash
# test.sh
ls –l &
echo "Done."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo            34 Oct 11 15:09 test.sh
```

background komutundan sonra bir **wait** yerleştirilmesi buna bir çare olur gibi görünüyor.

```
#!/bin/bash

# test.sh

ls –l &

echo “Done.”

wait
```

```
bash$ ./test.sh

Done.

 [bozo@localhost test-scripts]$ total 1

 -rwxr-xr-x    1 bozo     bozo            34 Oct 11 15:09 test.sh
```

Komutun çıktısını bir dosyaya ya da `/dev/null` için yönlendirmek de bu sorunu halleder.

suspend

Bu **Kontrol-Z**'ye benzer bir etkiye sahiptir, ancak kabuğu askıya alır (kabuğun ana sürecinin uygun bir zamanda devam etmesi gerekir).

logout

İsteğe bağlı olarak bir çıkış durumu belirterek, bir giriş kabuğundan çıkar.

times

Komutları çalıştırmak için kullanılan sistem saati ile ilgili, aşağıdaki şekildeki gibi, istatistikler verir:

```
0m0.020s 0m0.020s
```

Kabuk betiklerinin profili, verimliliği ve karşılaştırmalı değerlendirmesini yapmak çok yaygın olmadığı için, komutun bu yeteneği çok sınırlı bir değer taşımaktadır.

kill

Uygun bir *sonlandırma* sinyali göndererek, bir süreci zorla sonlandırır (bkz. Örnek 13.4).

ÖRNEK 11.20 KENDİSİNİ ÖLDÜREN BİR KOMUT DOSYASI

```
#!/bin/bash
# self-destruct.sh
kill $$  # Script kills its own process here.
         $ Recall that "$$" is the script's PID.
echo "This line will not echo."
# Instead, the shell sends a "Terminated" message to stdout.
exit 0
```

`kill -l` tüm sinyalleri listeler. **`kill -9`** "emin öldürmektir", sadece **kill** ile ölmeyi inatla reddeden bir işlem sürecini sonlandırır. Bazen, `kill -15` çalışır. Bir "zombi süreci", yani, ana süreci sonlandırılmış bir süreç öldürülemez (zaten ölü bir şey öldürülemez), fakat **init** genellikle er ya da geç onu temizleyecektir.

command

command KOMUT bildirimi "KOMUT" komutu için takma ad ve fonksiyonları devre dışı bırakır.

Bu betik komutlarının işlenmesini etkileyen üç kabuk bildiriminden biridir. Diğer ikisi, builtin ve enable direktifleridir.

builtin

builtin YERLEŞİK_KOMUT çağrılması "YERLEŞİK_KOMUT" komutunu aynı isme sahip fonksiyonları ve dış sistem komutlarını geçici olarak devre dışı bırakarak, bir kabuk yerleşiği olarak çalıştırır.

enable

Bir kabuk yerleşik komutunu ya etkinleştirir veya devre dışı bırakır. Örnek olarak, **enable –n kill** kabuk yerleşiği olan kill'i devre dışı bırakır, böylece daha sonra Bash ne zaman **kill** ile karşılaşırsa, o zaman `/bin/kill` çağrılacaktır.

enable komutuna verilen `-a` seçeneği tüm kabuk yerleşiklerini, etkin olup olmadıklarını göstererek listeler. enable komutu `-f dosyadı` seçeneği ile çalıştırılırsa yerleşik, bir paylaşılan kitaplık (DLL) modülü olarak düzgün derlenmiş bir hedef dosyasından yüklenir. [1].

autoload

Bu *ksh* otomatik yükleyici tarafından yüklenen Bash için bir kapıdır. Otomatik yükleme ile, "autoload" beyanı yapan bir işlev ilk çağrı sırasında dışardan bir dosyadan yükleyecektir. [2] Bu sistem kaynaklarından tasarruf sağlar.

Unutmayınız ki, **autoload** çekirdek Bash kurulumunun bir parçası değildir. **enable –f** (yukarıya bakınız) ile yüklenmesi gerekir.

Tablo 11-1. İş kimlikleri

Gösterim	Anlamı
%N	İş numarası [N]
%S	Karakter dizesi *S* ile başlayan işin (komut satırından) çağrılması
%?S	İçinde karakter dizesi *S* olan işin (komut satırından) çağrılması
%%	"geçerli" iş (ön planda durdurulan veya arka planda başlatılan son iş)
%+	"geçerli" iş (ön planda durdurulan veya arka planda başlatılan son iş)
%-	Son iş
$!	Son arka plan işlem süreci

Notlar

[1] Bir dizi yüklenebilir yerleşik için C kaynağı genellikle `/usr/share/doc/bash-?.??/functions` dizininde bulunur.

Unutmayınız ki, **enable** komutunun `-f` seçeneği tüm sistemlere taşınabilir değildir.

[2] **autoload** ile aynı etki typeset –fu ile elde edilebilir.

BÖLÜM 12 DIŞ FİLTRELER, PROGRAMLAR ve KOMUTLAR

Standart UNIX komutları sayesinde kabuk betikleri çok yönlülük açısından üst seviyelere gelmiştir. Betiklerin gücü, basit programlama yapıları ile sistem komutları ve kabuk bildirimleri arasında bağlantı kurabilmesi nedeniyledir.

12.1. Temel Komutlar

Başlangıç seviyesinde öğrenilmesi gereken temel komutlar

ls

Temel dosya "listesi" komutu. Kullanımı çok elverişli olan bu komutun gücü yadsınamaz. Örneğin, öztekrarlı seçeneği olan `-R` ile birlikte kullanıldığında ls komutu dizin yapısını ağaç-şeklinde listeler. Diğer ilginç seçenekler şöyle sıralanabilir, `-S`, dosyaların büyüklük sırasına göre listeler, `-t`, dosyaları değiştirilme zamanına göre sıralar, ve `-i`, dosyanın karakteristik özelliklerinin kayıtlı olduğu inode sistem tanımını gösterir (bkz. Örnek 12-3).

ÖRNEK 12.1 ls KULLANIMIYLA BİR CDR DİSKE YAZMAK İÇİN İÇİNDEKİLER TABLOSUNUN OLUŞTURULMASI

```
#!/bin/bash

SPEED=2                   # May use higher speed if your hardware supports

IMAGEFILE=cdimage.iso

CONTENTSFILE=contents

DEFAULTDIR=/opt   # Make sure this directory exists.

# Script to automate burning a CDR.

# Uses Joerg Schilling's "cdrecord" package.

# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)
```

```
#  If this script invoked as an ordinary user, need to suid cdrecord
#+ (chmod u+s /usr/bin/cdrecord, as root).
if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Default directory, if not specified on command line.
else
    IMAGE_DIRECTORY=$1
fi
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# The "l" option gives a "long" file listing.
# The "R" option makes the listing recursive.
# The "F" option marks the file types (directories get a trailing /).
echo "Creating table of contents."
mkisofs -r -o $IMAGFILE $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image ($IMAGEFILE)."
cdrecord -v -isosize speed=$SPEED dev=0,0 $IMAGEFILE
echo "Burning the disk."
echo "Please be patient, this will take a while."
exit 0
```

cat, tac

cat, *art arda bağla* anlamına gelen bir kısaltmadır. Komut olarak kullanıldığında, bir dosyanın içeriğini `stdout` çıktısı olarak gösterir. Yeniden yönlendirme (> ya da >>) ile birleştirildiğinde, dosyaları birleştirerek yazar.

```
cat filename cat file.1 file.2 file.3 > file.123
```

cat komutunun `-n` seçeneği hedef dosya(lar)ın tüm satırlarının başında ardışık sayılar gösterir. `-b` seçeneği sadece boş olmayan satırların başına sayı atar. `-v` seçeneği `^` notasyonunu kullanarak, basılması mümkün olmayan bazı özel karakterleri de yazar. `-s` seçeneği birden fazla boş satırı tek bir boş satır haline getirir.

bkz. Örnek 12.21 ve Örnek 12.27.

tac, *cat* komutunun listelediği dosya içeriğini tersine çevirir, dosya sondan başa doğru gösterilir.

rev

Bir dosyanın her bir satırını tersine çevirir, karakterler sondan başa doğru `stdout`'ta gösterilir. Etkisi **tac** gibi değildir; çünkü satır sıralarını korur.

```
bash$ cat file1.txt
This is line 1.
 This is line 2.

bash$ tac file1.txt
This is line2.
 This is line1.

bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cp

Bu dosya kopyalama komutudur. **`cp dosya1 dosya2`** `dosya2` yerine `dosya1` içeriğini kopyalar, `dosya2` halihazırda oluşturulmuş ise içeriği `dosya1` ile aynı olur. (bkz. Örnek 12.5)

En çok ve ayrıntılı olarak kullanılan bir seçenek de arşiv bayrağı `-a` (bir dizin ağacını kopyalamaya yarar.), ve `-r` ve `-R` öztekrar bayraklarıdır.

mv

Bu komut, bir dosyanın *yer değiştirmesi* (taşınması) ile ilgilidir. `cp` ve `rm` komutlarının ard arda kullanımıyla yaratılan etkiyle eşdeğerdir. Dosya bir başka yere kopyalanır, kopya öncesi ait olduğu yerden silinir. Bu, bir dizine birden çok dosya taşımak için, ya da bir dizinin isim değişikliği ile ilgili kullanılabilir. Bir betik içinde `mv` kullanımı için bkz. Örnek 9.15 ve Örnek A.3.

Etkileşimli olmayan bir komut dosyası içinde **mv** kullanıcı girişini atlamak için `-f` (*force*) seçeneği ile kullanılır. Bir dizin önceden oluşturulmuş bir dizine taşındığında, hedef dizininin bir alt dizini haline gelir.

```
bash$ mv source_directory target_directory

bash$ ls -lF target_directory

total 1

 drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 source_directory/
```

rm

Bir dosya veya dosyaları silmeye (yok etmeye) yarar. `-f` seçeneği ile kullanılırsa, salt-okunur dosyalar dahil edilmek üzere silinir, ve kullanıcı girişi olmayan bir betik yazılması için yararlıdır.

`-r` öztekrar bayrağı ile birlikte kullanıldığında, bu komut dizin ağacı altındaki tüm dosyaları de siler.

rmdir

Dizin silme amacıyla kullanılır. Bu komutun başarılı olması için dizinin tüm dosyalarının boş olması gereklidir; görünmez, yani "nokta ile başlayan tüm dosyalar" da komutla birlikte tek seferde silinir.

mkdir

Dizin oluşturur, yeni bir dizin oluşturur. **`mkdir -p proje/programlar/Aralık`** isimli bir dizin oluşturur. *-p* seçeneği gerekli üst dizinleri otomatik olarak oluşturur.

chmod

Dosya için yetkilendirilmiş bazı özelliklerin atanmasını sağlar (bkz Örnek 11.9).

```
chmod +x filename

# Makes "filename" executable for all users.

chmod u+s filename

# Sets "suid" bit on "filename" permissions.

# An ordinary user may execute "filename" with same privileges as the
file's owner.

# (This does not apply to shell scripts.)
```

```
chmod 644 filename

# Makes "filename" readable/writeable to owner, readable to others

# (octal mode).
```

```
chmod 1777 directory-name

# Gives everyone read, write, and execute permission in directory,

# however also sets the "sticky bit".

# This means that only the owner of the directory,

# owner of the file, and, of course, root

# can delete any particular file in that directory.
```

chattr

Dosya için yetkilendirilmiş bazı özelliklerin atanmasını sağlar. Bu komut, farklı haldeki söz dizimi nedeniyle, yukarıdaki **chmod** ile farklı etki yaratmak zorunda değildir, etkisi aynıdır, açıklaması ise sadece bir *ext2* dosya sistemi üzerinde çalıştığıdır.

ln

Önceden oluşturulan dosyalara bağlantılar yaratır. Çoğu zaman tercih edilen `-s` seçeneği, direkt bağlantı kurmadan "sembolik" bağlantı kurmaya yarar.

Bağlantı kurulan dosyaya birden fazla isim ile başvuru (referans) yapılabilir ve aliasing olarak anılan indirekt isimlendirme yönteminden daha etkilidir (Örnek 5.6).

`ln -s eskidosya yenidosya` önceden oluşturulan `eskidosya` ile yeni yaratılmış `yenidosya` arasında sembolik bağlantı yapar.

Notlar

[1] Dosya ismi . ile başlayan `~/.Xdefaults`. Böyle dosyalar normal `ls` listelenmesinde gösterilmezler ve yanlışlıkla (istemeyerek yazılan) `rm -rf *` tarafından silinemez. Nokta ile başlayan dosyalar genellikle kurulum aşamasında kullanılır ve bir kullanıcının ana dizinindeki yapılandırma dosyalarıdır.

12.2 GELİŞMİŞ KOMUTLAR

Daha gelişmiş kullanıcılar için komutlar.

find

-exec *KOMUT* \;

find komutunun bulduğu dosyalar üzerinde *KOMUT* komutunu yürütür. Sözdizimi açısından *KOMUT* `\;` ile sonlanır (`;` kaçış karakteri ile yazılmıştır, bunun nedeni harfi harfine ve tam olarak komutlar sırasıyla kabuk tarafından **find** komutuna geçirilmelidir). *KOMUT* `{}` içeriyorsa, **find** seçilen dosyanın tam yol adını yer değiştirir.

```
bash$ find ~/ -name ' *.txt'

/home/bozo/.kde/share/apps/karm/karmdata.txt

 /home/bozo/.kde/share/apps/karm/karmdata.txt

 /home/bozo/misc/irmeyc.txt

 /home/bozo/test-scripts/1.txt
```

```
find /home/bozo/projects –mtime 1

# Lists all files in /home/bozo/projects directory tree

# that were modified within the last day.
```

```
find /etc –exec grep ' [0-9] [0-9]*[.][0-9][0-9]*[.][0-9][0-0]*[.][0-9][0-
9]*' {} \;

# Finds all IP addresses (xxx.xxx.xxx.xxx) in /etc directory files.

# There a few extraneous hits – how can they be filtered out?

# Perhaps by:

find /etc –type f –exec cat ' {}' \; | tr –c ' .[:digit:]' \n' \ | grep
' ^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'

# [:digit:] is one of the character classes

# introduced with the POSIX 1003.2 standard.

# Thanks, S.C.
```

find komutunun –exec seçeneği ile kullanımını, exec kabuk yerleşiği ile karıştırmayınız.

ÖRNEK 12.2 Badname, GEÇERLİ DİZİNDE BULUNAN DOSYA ADLARINDAN KÖTÜ veya BOŞLUK KARAKTERİ İÇERENLERİ SİLER.

```
#!/bin/bash
# Delete filenames in current directory containing bad characters.
for filename in *
do
badname='echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p'
# Files containing those nasties:     + { ; " \ = ? ~ ( ) < > & * | $
rm $badname 2>/dev/null    # So error messages deep-sixed.
done
# Now, take care of file containing all manner of whitespace.
find . -name " * *" -exec rm -f {} \;
# The path name of the file that "find" finds replaces the "{}".
# The ' \' ensures that the ' ;' is interpreted literally, as end of command.
exit 0

# ------------------------------------------------------------------
# Commands below this line will not execute because of "exit" command.
# An alternative to the above script:
find . -name ' *[+{;"\\=?~()<>&*|$ ]*' -exec rm -f ' {}\;
exit 0
# (Thanks, S.C.)
```

ÖRNEK 12.3 inode DÜĞÜMÜNE GÖRE BİR DOSYANIN SİLİNMESİ

```
#!/bin/bash
# idelete.sh: Deleting a file by its inode number.
#  This is useful when a filename starts with an illegal character,
```

```
#+ such as ? or -.
ARGCOUNT=1                      # Filename arg must be passed to script.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72
if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: ' basename $0' filename"
  exit $E_WRONGARGS
fi
if [ ! -e "$1" ]
then
  echo "File \""$1"\" does not exist."
  exit $E_FILE_NOT_EXIST
fi

inum='ls -i | grep "$1" | awk ' {print $1}''
# inum = inode (index node) number of file
# Every file has an inode, a record that hold its physical address info.
echo; echo -n "Are you absolutely sure you want to delete \"$1\" (y/n)? "
read answer
case "$answer" in
[nN]  echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Deleting file \"$1\".";;
esac
find . -inum $inum -exec rm {} \;
echo "File "\"$1"\" deleted!'
exit 0
```

find kullanan betikler için bkz.Örnek 12.22, Örnek 4.4, ve Örnek 10.9. Bu karmaşık ve güçlü komut hakkında daha detaylı bilgi için yardım sayfasına (manpage) başvurunuz.

xargs

Argümanları bir komuta vermek için kullanılan bir filtredir, ve aynı zamanda komutların kendilerinin birleştirilmesi için de bir araçtır. Bir veri akışını filtre ve işlem yapacak komutlar için yeterince küçük parçalar halinde böler. `backquotes` ile yer değiştirebilir, ancak fazla argüman hatası yüzünden `backquotes` başarısız sonuçlansa da, **xargs** genellikle başarıyla çalışır. Normal olarak, **xargs** `stdin`'den ya da bir oluktan okur, ama bir dosyanın çıktısı da ona girdi olarak verilebilir.

xargs için varsayılan komut echo komutudur. Bunun anlamı şudur: **xargs** komutuna oluk yoluyla yapılan girdi, satır besleme (linefeed) ve diğer boşluk karakterlerin yok edilmiş hallerini içerebilir.

```
bash$ ls -l

total 0

 -rw-rw-r--      1  bozo   bozo

 -rw-rw-r--      1  bozo   bozo


bash$ ls –l | xargs

total 0 –rw-rw-r-- 1 bozo  bozo 0 Jan 29 23:58 file1 –rw-rw-r—1 bozo bozo 0
Jan 29 23:58 file2
```

`ls | xargs -p -l gzip` geçerli dizindeki her dosyayı, her seferde bir tane olmak üzere **gzip**'le sıkıştırır, başlamadan önce her biri için kullanıcıya onay sorar.

İlginç bir **xargs** seçeneği olan `-n` *NN*, geçirilen argüman sayısını en fazla *NN* ile sınırlar.

`ls | xargs -n 8 echo` mevcut dizindeki dosyaları 8 sütun halinde listeler.

Bir başka kullanışlı seçenek, **find -print0** veya **grep -lZ** ile birlikte kullanılan `-0`'dır. Bu boşluk veya tırnak işareti içeren argümanlarla da çalışmaya olanak verir.

```
find / -type f –print0 | xargs -0 grep –liwZ GUI | xargs -0 rm -f
```

```
grep –rliwZ GUI / | xargs -0 rm -f
```

Yukarıdaki örneklerden her ikisi de "GUI" içeren herhangi bir dosyayı silecektir.
(Teşekkürler, S.C.)

ÖRNEK 12.4 SİSTEM GÜNLÜĞÜNÜ İZLEMEK İÇİN GÜNLÜK DOSYASININ xargs KULLANMASI

```
#!/bin/bash

# Generates a log file in current directory

# from the tail end of /var/log/messages.

# Note: /var/log/messages must be world readable

# if this script invoked by an ordinary user.

#         #root chmod 644 /var/log/messages


LINES=5

(date; uname –a ) >>logfile

# Time and machine name

echo ------------------------------------------------------------------
>>logfile

tail -$LINES /var/log/messages | xargs | fmt –s >>logfile

echo >>logfile

echo >>logfile


exit 0
```

ÖRNEK 12.5 xargs KULLANARAK, GEÇERLİ DİZİNDEKİ DOSYALARIN BİR BAŞKA DİZİNE KOPYALANMASI: copydir

```
#!/bin/bash
# Copy (verbose) all files in current directory
# to directory specified on command line.
if [ -z "$1" ]   # Exit if no argument given.
then
  echo "Usage: ' basename $0' directory-to-copy-to"
  exit 65
fi
ls . | xargs -i -t cp ./{} $1
# This is the exact equivalent of
#    cp * $1
# unless any of the filenames has "whitespace" characters.
exit 0
```

expr

Çok amaçlı ifade değerlendirici : Argümanları verilen işleme göre birleştirir ve değerlendirir. (Argümanlar boşluklarla ayrılmış olmalıdır). İşlemler aritmetik, karşılaştırma, string, veya mantıksal olabilir.

```
expr 3 + 5
```

8 döndürür.

```
expr 5 % 3
```

2 döndürür.

```
expr 5 \* 3
```

15 döndürür.

Çarpma operatörü, bir aritmetik ifade olarak expr ile birlikte kullanıldığında hemen önüne kaçma karakteri yazılmalıdır.

```
y= ‘ expr $y + 1’
```

Bir değişkenin değerini artırır. **`let y=y+1`** ve **`y=$(($y+1))`** ile aynı etkiye sahiptir. Bu aritmetik açılıma bir örnektir.

```
z= ‘ expr altdize $dize $konum $uzunluk’
```

`$konum`'dan başlayarak `$uzunluk` kadar karakteri altdize olarak seçip çıkarır.

ÖRNEK 12.6 expr KULLANIMI

```
#!/bin/bash
# Demonstrating some of the uses of ‘ expr’
# ====================================
echo
# Arithmetic Operators
# ---------- ---------
echo “Aritmetic Operators”
echo

a=’expr 5 + 3’
echo “5 + 3 = $a”

a=’expr $a + 1’
```

```
echo

echo "a + 1 = $a"

echo "(incrementing a variable)"


a='expr 5 % 3'

# modulo

echo

echo "5 mod 3 = $a"

echo

echo


# Logical Operators

# ------- ---------

#  Returns 1 if true, 0 if false,

#+ opposite of normal Bash convention.

echo "Logical operators"

echo


x=24

y=25

b='expr $x = $y'          # Test equality.

echo "b = $b"             # 0  ( $x -ne $y )

echo


a=3

b='expr $a \> 10'

echo ' b='expr $a \> 10', therefore...'

echo "If a > 10, b = 0 (false)"

echo "b = $b"             # 0 ( 3 ! -gt 10 )

echo
```

```
b='expr $a \< 10'
echo "If a < 10, b = 1 (true)"
echo "b = $b"             # 1  ( 3 -lt 10 )
echo
# Note escaping of operators.
b='expr $a \<= 3'
echo "If a <= 3, b = 1 (true)"
echo "b = $b"
# There is also a "\>=" operator (greater than or equal to).


echo
echo
# Comparison Operators
# ---------- ---------
echo "Comparison Operators"
echo
a=zipper
echo "a is $a"
if [ ' expr $a = snap' ]
# Force re-evaluation of variable ' a'
then
   echo "a is not zipper"
fi
echo
echo


# String Operators
# ------ ---------
echo "String Operators"
echo
```

```
a=1234zipper43231

echo "The string being operated upon is \"$a\"."

# length: length of string

b='expr length $a'

echo "Length of \"$a\" is $b."

# index: position of first character in substring

#        that matches a character in string

b='expr index $a 23'

echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."


# substr: extract substring, starting position & length specified

b='expr substr $a 2 6'

echo "Substring of \"$a\", starting at position 2, \

and 6 chars long is \"$b\"."


#  The default behavior of the ' match' operations is to

#+ search for the specified match at the ***beginning*** of the string.

#

#        uses Regular Expressions

b='expr match "$a" ' [0-9]*''                #  Numerical count.

echo Number of digits at the beginning of \"$a\" is $b.

b='expr match "$a" ' \([0-9]*\)''            #  Note that escaped parentheses

#                    ==      ==              + trigger substring match.

echo "The digits at the beginning of \"$a\" are \"$b\"."

echo

exit 0
```

: operatörü **match** yerini alabilir. Örneğin, yukarıdaki kodda **`b='expr $a : [0-9]*'`** tam olarak **`b='expr match $a [0-9]*'`** ile eşdeğerdir.

```
#!/bin/bash

echo

echo "String operations using \"expr \$string : \" construct"

echo "=================================================="

echo

a=1234zipper5FLIPPER43231

echo "The string being operated upon is \"'expr "$a" : ' \(.*\)''\"."

#     Escaped parentheses grouping operator.

#      ***************************

#+          Escaped parentheses

#+           match a substring

#      ***************************


#  If no escaped parentheses...

#+ then ' expr' converts the string operand to an integer.

echo "Length of \"$a\" is ' expr "$a" : ' .*''."   #length of string

echo "Number of digits at the beginning of \"$a\" is ' expr "$a" : ' [0-
9]*''."

# ------------------------------------------------------------------------
#

echo

echo "The digits at the beginning of \"$a\" are ' expr "$a" : ' \([0-
9]*\)''."

echo "The first 7 characters of \"$a\" are ' expr "$a" : ' \(.......\)''."

# Again, escaped parentheses force a substring match.

echo "The last 7 characters of \"$a\" are ' expr "$a": ' .*\(.......\)''."

echo

exit 0
```

Yukarıdaki örnek, *kaçma karakterli parantezin* `--\( ... \)--` bir karakter alt dizesini eşleştirmek için düzenli ifade ayrıştırma ile birlikte gruplama operatörünün **expr** tarafından nasıl kullanıldığını göstermektedir .

Perl ve sed'in çok daha üstün dize ayrıştırma yetenekleri vardır. Bir komut dosyası içinde kısa bir **Perl** ve **sed** "alt yordamı" (bkz. Bölüm 34.2) **expr** kullanmaya alternatif olan cazip bir seçenektir.

Karakter dizesi işlemleri hakkında daha fazla bilgi için bkz. Bölüm 9.2.

12.3. SAAT / TARİH KOMUTLARI

Saat ve tarihin işlenmesi

date

Basitçe çağrıldığında, **date** `stdout`'a tarih ve saati yazdırır. Bu komutun ilginç olduğu nokta, biçimlendirme ve ayrıştırma seçenekleridir.

ÖRNEK 12.7 date KULLANIMI

```
#!/bin/bash

# Exercising the ‘ date’ command

echo “The number of days since the year’s beginning is ‘ date +%j’.”

# Needs a leading ‘ +’ to invoke formatting.

# %j gives day of year.


echo “The number of seconds elapsed since 01/01/1970 is ‘ date +%s’.”

#  %s yields number of seconds since “UNIX epoch” began,

#+ but how is this useful?


prefix=temp

suffix=’eval date +%s’  # The “+%s” option to ‘ date’ is GNU-specific.
```

```
filename=$prefix.$suffix


echo $filename
#  It's great for creating "unique" temp filenames,
#+ even better than using $$.
# Read the ' date' man page for more formatting options.


exit 0
```

-u seçeneği UTC (Eşgüdümlü Evrensel Zaman) verir.

```
bash$ date
Fri Mar 29 21:07:39 MST 2002


bash$ date –u
Sat Mar 30 04:07:42 UTC 2002
```

zdump

Belirli bir zaman diliminde zamanı yazdırır.

```
bash$ zdump EST
EST Tue Sep 18 22:09:22 2001 EST
```

time

Bir komut çalıştırmak için çok ayrıntılı zamanlama istatistiklerini verir.

time ls –l / şöyle bir şey verir:

```
0.00user 0.01system 0:00.05elapsed 16%CPU(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

Önceki bölümdeki çok benzer **times** komutuna ayrıca bakınız.

Bash 2.0 sürümü itibariyle, **time** bir oluk içindeki hafif değiştirilmiş davranışı ile kabuk için ayrılmış bir sözcük haline gelmiştir.

touch

Bir dosyanın erişim / değişiklik zamanlarını geçerli sistem saatine veya diğer belirtilen süreye güncellemek için yardımcıdır, ama aynı zamanda yeni bir dosya oluşturmak için de yararlıdır. `touch zzz` komutu `zzz` adlı dosyanın daha önce varolmadığını varsayarak, `zzz` adlı sıfır uzunlukta yeni bir dosya oluşturur. Boş dosyaların bu şekilde zaman-damgalanması, tarih bilgisini saklamak için yararlıdır, örneğin bir proje üzerinde yapılan değişikliklerin takip edilmesi gibi.

touch komutu `: >> yenidosya` veya `>> yenidosya` (sıradan dosyalar için) ile eşdeğerdir.

at

İş kontrol komutu, **at**, verilen bir komut grubunu belirli bir zamanda çalıştırır. Görünüşte, **crond** komutuna benzer, ancak **at** bir komut setinin bir kez yürütülmesi için özellikle yararlıdır.

`at 2pm Ocak 15` o zamanda çalıştırmak için bir dizi komut ister. Bu komutlar, kabuk-betiğine uyumlu olmalıdır, çünkü uygulanabilir bütün amaçlar için, kullanıcı, bir defada bir yürütülebilir kabuk betiğini bir satıra yazmaktadır. Girdi **Ctl-D** ile sona erer.

at, ya `-f` seçeneğini ya da girdi yeniden yönlendirmesini `(<)` kullanarak, bir dosyadan komut listesini okur . Bu dosya, tabii ki etkileşimli olmamasına rağmen, yürütülebilir bir kabuk betiğidir. Özellikle akıllı bir yöntem, bir dizi farklı komutu yürütmek için dosyaya **run-parts** komutunu dahil etmektir.

```
bash$ at 2:30 am Friday < at-jobs.list

job 2 at 2000-10-27 02:30
```

batch

batch iş kontrol komutu **at** komutuna benzer, ancak sistem yükü 0.8'in altına düştüğünde komut listesini çalıştırır. **at** gibi, `-f` seçeneği ile bir dosyadan komutları okuyabilir.

cal

Özenle biçimlendirilmiş bir aylık takvimi `stdout`'a yazdırır. İçinde bulunduğumuz yılı veya geniş bir geçmiş ve gelecek yıl aralığını yazdırabilir.

sleep

Bu, **wait** döngüsünün kabuk eşdeğeridir. Hiçbir şey yapmadan, belirtilen saniye kadar duraklar. Bu zamanlama için veya belirli bir olayı sık sık kontrol etmesi gereken arka planda çalışan süreçler için yararlıdır (bkz. Örnek 30.6).

```
sleep 3
# Pauses 3 seconds.
```

sleep komutu varsayılan olarak saniye alır, ama dakika, saat veya gün de belirtilebilir.

```
sleep 3 h
# Pauses 3 hours!
```

usleep

Mikrouyku ("u" Yunanca "mu" olarak, veya mikro öneki ile okunur). Bu yukarıdaki **sleep** ile aynıdır, ama mikrosaniye aralıklarla "uyur". Bu ince ayar gerektiren zamanlamalar için, ya da çok sık aralıklarla yoklanması gereken devam eden bir süreç için kullanılabilir.

```
usleep 30

# Pauses 30 microseconds.
```

usleep komutu en çok doğru zamanlamayı sağlayamaz, bu nedenle kritik zamanlama döngüleri için uygun değildir.

hwclock, clock

hwclock komutu makinenin donanım saatine erişir veya ayarlar. Bazı seçenekler root yetkileri gerektirir. `/etc/rc.d/rc.sysinit` başlangıç dosyası, açılışta otomatik olarak sistem zamanını donanım saatine ayarlamak için **hwclock** komutunu kullanır.

clock komutu **hwclock** ile eş anlamlıdır.

12.4 METİN İŞLEME KOMUTLARI

Metin ve metin dosyaları etkileyen komutlardır.

sort

Çoğu zaman, bir oluk içinde bir filtre olarak kullanılan dosya sıralayıcıdır. Bu komut bir metin akışını veya dosyayı ileri veya geri yönde, veya çeşitli tuşlar ya da karakter pozisyonlarına göre sıralar. `-m` seçeneğini kullanarak, önceden sıralanmış girdi dosyalarını birleştirir. Bilgi sayfası bu komutun pek çok yeteneklerini ve seçeneklerini listeler. bkz. Örnek 10.9, Örnek 10.10 ve Örnek A.9.

tsort

Topolojik sıralama, boşlukla ayrılmış metin dizelerini çifter çifter okur ve girdi kalıplarına göre sıralar.

uniq

Bu filtre sıralanmış bir dosyadaki yinelenen satırları kaldırır. Genellikle sort ile bağlı bir olukta görülür.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# Concatenates the list files,
# sorts them,
# removes duplicate lines,
# and finally writes the result to an output file.
```

Oldukça kullanışlı olan `-c` seçeneği, girdi dosyasının satır başlarına satırın oluş numarasını ekler.

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.

bash$ uniq -c testfile
      1 This line occurs only once.
       2 This line occurs twice.
       3 This line occurs three times.

bash$ sort testfile | uniq -c | sort -nr
      3 This line occurs three times.
       2 This line occurs twice.
       1 This line occurs only once.
```

`sort GİRDİDOSYASI | uniq -c | sort -nr` komutu `GİRDİDOSYASI` üzerindeki satırlara ait *oluş frekansı* listesi üretir**.** (`-nr` seçeneği sayıları tersten sıralar.) Bu şablon, günlük dosyaları ve sözlük listelerinin analizinde ve bir belgenin sözcük yapısının incelenmesi gereken her yerde kullanım alanı bulur.

ÖRNEK 12.8 SÖZCÜK FREKANSI ANALİZİ

```
#!/bin/bash
# wf.sh: Crude word frequency analysis on a text file.
# Check for input file on command line.
ARGS = 1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne "$ARGS" ]  # Correct number of arguments passed to script?
then
  echo "Usage: ' basename $0' filename"
  exit $E_BADARGS
fi
if [ ! -f "$1" ]       # Check if file exists.
then
  echo "File \"$1\" does not exists."
  exit $E_NOFILE
fi
###############################################################
# main()
sed -e ' s/\.//g' -e ' s/ /\
/g' "$1" | tr ' A-Z' ' a-z' | sort | uniq -c | sort -nr
#                            ========================
```

```
#                               Frequency of occurrence
#  Filter out periods and
#+ change space between words to linefeed,
#+ then shift characters to lowercase, and
#+ finally prefix occurrence count and sort numerically.
###########################################################
# Exercises:
# 1) Add ' sed' commands to filter out other punctuation, such as commas.
# 2) Modify to also filter out multiple spaces and other whitespace.
# 3) Add a secondary sort key, so that instances of equal occurrence
#+   are sorted alphabetically.
exit 0
```

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.
```

```
bash$ ./wf.sh testfile
      6 this
       6 occurs
       6 line
       3 times
       3 three
       2 twice
       1 only
       1 once
```

expand, unexpand

expand filtresi sekmeleri boşluk karakterine dönüştürür. Bu, genellikle bir oluk içinde kullanılır.

unexpand filtresi boşluk karakterini sekmelere dönüştürür. Bu, expand komutunun etkisini tersine çevirir.

cut

Dosyalardan veri alanlarını ayıklamak için bir araçtır. Bu, awk'ta ortaya çıkan **`print $N`** komutuna benzer, ancak daha kısıtlısıdır. Bir komut dosyasında **cut** kullanımı, **awk** kullanımından daha kolay olabilir. Özellikle önemli olan seçenekler `-d` (sınırlayıcı) ve `-f` (alan belirteci) seçenekleridir.

Bağlanan dosya sistemlerinin bir listesini elde etmek için **cut** kullanımı:

```
cat /etc/mtab | cut –d ‘  ‘  –f1,2
```

OS ve çekirdek sürümünü listelemek için **cut** kullanımı:

```
uname –a | cut –d” “ –f1,3,11,12
```

Bir e-posta klasöründen mesaj başlıklarını ayıklamak için **cut** kullanımı:

```
bash$ grep ‘ ^Subject:’ read-messages | cut –c10-80

Re: Linux suitable for mission-critical apps?

 MAKE MILLIONS WORKING AT HOME!!!

 Spam complaint

 Re: Spam complaint
```

Bir dosyayı ayrıştırmak için **cut** kullanımı:

```
# List all the users in /etc/passwd.
FILENAME=/etc/passwd
for user in $(cut –d: -f1 $FILENAME)
do
  echo $user
done
# Thanks, Oleg Philon for suggesting this.
```

cut –d ‘ ‘ –f2,3 dosyadı ile **awk –F'[ ]' ' { print $2, $3 }' dosyadı**

eşdeğerdir.

Ayrıca bkz. Örnek 12.33.

paste

Farklı dosyaları bir tek, çok-sütunlu dosya içine birleştirmek için bir araçtır. **cut** ile birlikte sistem günlük dosyaları oluşturmak için yararlıdır.

join

Bunu, **paste**'in özel amaçlı bir kuzeni olarak düşünün. Bu güçlü bir yardımcı, iki dosyanın anlamlı bir şekilde birleştirilmesini sağlar, aslında bir ilişkisel veritabanının basit bir sürümünü oluşturur.

join komutu, tam olarak iki dosya üzerinde çalışır, ancak sadece ortak etiketli alanları olan satırları birarada yapıştırır (genellikle sayısal bir etiket), ve sonucu `stdout`'a yazar. Karşılaştırmaların düzgün çalışması için birleştirilecek dosyalar etiketli alana göre sıralanmalıdır.

```
File: 1.data

100 Shoes

200 Laces

300 Socks


File: 2.data

100 $40.00

200 $1.00

300 $2.00
```

```
bash$ join 1.data 2.data

File: 1.data 2.data

 100 Shoes $40.00

 200 Laces $1.00

 300 Socks $2.00
```

Etiketli alan çıktıda sadece bir kez görünür.

head

Bir dosyanın başını `stdout`'a listeler (varsayılan `10` satırdır, ancak bu değiştirilebilir). İlginç olan bir dizi seçeneğe sahiptir.

ÖRNEK 12.9 HANGİ DOSYALAR KOMUT DOSYASIDIR?

```
#!/bin/bash

# script-detector.sh: Detects scripts within a directory.

TESTCHARS=2    # Test first 2 characters.

SHABANG='#!'   # Scripts begin with a "sha-bang."

for file in *  # Traverse all the files in current directory.
```

```
do
  if [[ ‘ head –c$TESTCHARS “$file”’ = “$SHABANG” ]]
  #  The ‘ -c’ option to “head” outputs a specified
  #+ number of characters, rather than lines (the default).
  then
    echo “File \”$file\” is a script.”
  else
    echo “File \”$file\” is *not* a script.”
  fi
done

exit 0
```

ÖRNEK 12.10 ON HANELİ RASGELE SAYILAR ÜRETİLMESİ

```
#!/bin/bash
# rnd.sh: Outputs a 10-digit random number
# Script by Stephane Chazelas.
head –c4 /dev/urandom | od –N4 –tu4 | sed –ne ‘ 1s/.* /ø’
# ============================================================== #
# Analysis
# head:
# -c4 option takes first 4 bytes.
# od:
# -N4 option limits output to 4 bytes.
# -tu4 option selects unsigned decimal format for output.
# sed:
# -n option, in combination with “p” flag to the “s” command,
# outputs only matched lines.
```

```
# The author of this script explains the action of 'sed', as follows.
# head –c4 /dev/urandom | od –N4 –tu4 | sed –ne ' 1s/.*//p'
# Assume output up to "sed"
# is 0000000 1198195154\n
# sed begins reading characters: 0000000 1198195154\n.
# Here it finds a newline character,
# so it is ready to process the first line (0000000 1198195154).
# It looks at its <range><action>s. The first and only one is
#   range     action
#   1         s/.* //p
# The line number is in the range, so it executes the action:
# tries to substitute the longest string ending with a space in the line
# ("0000000 ") with nothing (//), and if it succeeds, prints the result
# ("p" is flag to the "s" command here, this is from the "p" command).
# sed is now ready to continue reading its input. (Note that before
# continuing, if –n option had not been passed, sed would have printed
# the line once again).
# Now, sed reads the remainder of the characters, and finds the end of
file.
# It is now ready to process its 2nd line (which is also numbered ' $' as
# it's the last one).
# It sees it is not matched by any <range>, so its job is done.
# In few word this sed command means:
# "On the first line only, remove any character up to the right-most space,
# then print it."
# A beter way to do this would have been:
#          sed –e ' s/.* //;q'
# Here, two <range><action>s (could have been wrritten
#          sed –e ' s/.* //' –e q):
#   range                    action
```

```
#   nothing (matches line)   s/.* //

#   nothing (matches line)   q (quit)

# Here, sed only reads its first line of input.

# It performs both actions, and prints the line (substituted) before
quitting

# (because of the "q" action) since the "-n" option is not passed.

# ================================================================ #

# A simpler alternative to the above 1-line script would be:

#           head -c4 /dev/urandom | od -An -tu4

exit 0
```

Ayrıca bkz. Örnek 12.30.

tail

Bir dosyanın sonunu `stdout`'a listeler (varsayılan `10` satırdır). Yaygın olarak, `-f` seçeneğini kullanarak, sistem günlük dosyası değişikliklerini takip etmek için kullanılır, bu seçenek bu dosyaya eklenen satırları verir.

ÖRNEK 12.11 SİSTEM GÜNLÜĞÜNÜ İZLEMEK İÇİN tail KULLANILMASI

```
#!/bin/bash

filename=sys.log

cat /dev/null > $filename; echo "Creating / cleaning out file."

#  Creates file if it does not already exist.

#+ and truncates it to zero length if it does.

#  : > filename   and   > filename also work.

tail /var/log/messages > $ filename

# /var/log/messages have world read permission for this to work.

echo "$filename contains tail end of system log."

exit 0
```

Ayrıca bkz. Örnek 12-4, Örnek 12-30 ve Örnek 30-6.

grep

Düzenli ifadeler kullanan çok amaçlı bir dosya arama aracıdır. Bu, aslında saygıyla bilinen **ed** satır editöründe bir komut/filtresidir, açılımı **g/re/p** – *global – regular expression – print*.

grep *kalıp*[*dosya…*]

kalıp oluşları için hedef dosya(lar)ı arar, *kalıp* düz metin ya da düzenli ifade olabilir.

```
bash$ grep ' [rst]ystem.$' osinfo.txt

The GPL governs the distribution of the Linux operating system.
```

Hedef dosya (lar) belirtilmemişse, **grep**, bir olukta olduğu gibi `stdout` üzerinde bir filtre olarak çalışır.

```
bash$ ps ax | grep clock

765 tty1     S      0:00 xclock

 901 pts/1    S      0:00 grep clock
```

`-i` seçeneği harf duyarsız bir aramaya neden olur.

`-w` seçeneği yalnızca tam sözcükleri eşleşir.

`-l` seçeneği sadece eşleşmelerin bulunduğu dosyaları listeler, ancak eşleşen satırları listelemez.

`-r` (özyineli) seçeneği geçerli çalışma dizini ve altındaki tüm alt dizinlerdeki dosyaları arar.

`-n` seçeneği eşleşen satırları birlikte satır numaraları ile birlikte listeler.

```
bash$ grep –n Linux osinfo.txt

2:This is a file containing information about Linux.

 6:The GPL governs the distribution of the Linux operating system.
```

`-v` (ya da – `ters-eşleşme`) seçeneği eşleşmeleri *filtreden geçirir*.

```
grep pattern *.txt | grep –v pattern2
# Matches all lines in “*.txt” files containing “pattern1”,
# but ***not*** “pattern2”.
```

`-c (--count)` seçeneği eşleşmelerin sayısal karşılığı olan sayıyı verir, eşleşmeleri listelemez.

```
grep –c txt *.sgml    # (number of occurrences of “txt” in “*.sgml” files)

# grep –cz .
# means count (-c) zero-separated (-z) items matching “.”
# that is, non-empty ones (containing at least 1 character).
#
printf ‘ a b\nc d\n\n\n\n\n\000\n\000e\000\000\nf’ |grep –cz .     # 4
printf ‘ a b\nc d\n\n\n\n\n\000\n\000e\000\000\nf’ |grep –cz ‘ $’   # 5
printf ‘ a b\nc d\n\n\n\n\n\000\n\000e\000\000\nf’ |grep –cz ‘ ^’   # 5
#
printf ‘ a b\nc d\n\n\n\n\n\000\n\000e\000\000\nf’ |grep –cz ‘ $’   # 9
# By default, newline chars (\n) separate items to match.
# Note that the –z option is GNU “grep” specific.
# Thanks, S.C.
```

Birden fazla hedef dosya ile birlikte çağrıldığında, **grep** hangi dosyanın eşleşmeleri içerdiğini belirtir.

```
bash$ grep Linux osinfo.txt misc.txt

osinfo.txt:This is a file containing information about Linux.

 osinfo.txt:The GPL governs the distribution of the Linux operating system.

 misc.txt:The Linux operating system is steadily gaining in popularity.
```

Sadece bir tek hedef dosyayı ararken dosya adının gösterilmesini zorlamak için **grep** komutunun ikinci argümanı olarak `/dev/null` dosyasını vermeniz yeterlidir.

```
bash$ grep Linux osinfo.txt /dev/null

osinfo.txt:This is a file containing information about Linux.

 osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

Başarılı bir eşleşme varsa, grep `0` çıkış durumunu döner, bu bir komut dosyası içinde durum testi yapmaya olanak verir, özellikle çıktının gösterilmesini engellemeye yarayan `-q` seçeneği ile birleşik kullanıldığında yararlıdır.

```
SUCCESS=0                      # if grep lookup succeeds

word=Linux

filename=data.file

grep –q “$word” “$filename”    # The “-q” option causes nothing to echo to
stdout.

if [ $? –eq $SUCCESS ]

then

  echo “$word found in $filename”

else

  echo “$word not found in $filename”

fi
```

Örnek 30.6 bir sistem günlüğünde bir sözcük kalıbını aramak için **grep**'in nasıl kullanılacağını gösteriyor.

ÖRNEK 12.12 BİR KOMUT DOSYASINDA grep'in EMÜLASYONU

```
#!/bin/bash

# grp.sh: Very crude reimplementation of ‘ grep’

E_BADARGS=65
```

```
if [ -z "$1" ]    # Check for argument to script.
then
  echo "Usage: ' basename $0' pattern"
  exit $E_BADARGS
fi
echo

for file in *     # Traverse all files in $PWD.
do
  output=$(sed -n /"$1"/p $file)   # Command substitution.

  if [ ! -z "$output" ]            # What happens if "$output" is not
quoted?
  then
    echo -n "$file: "
    echo $output
  fi          #  sed -ne "/$1/s|^|${file}: |p" is equivalent to above.
  echo
done
echo

exit 0
# Exercises:
# 1) Add newlines to output, if more than one match in any given file.
# 2) Add features.
```

egrep ile **grep -E** aynıdır. Bu, düzenli ifadelerin biraz daha farklı, genişletilmiş kümesini kullanır, böylece biraz daha esnek yapabilirsiniz.

fgrep ile **grep -F** aynıdır. Harfi harfine bir arama yapar (düzenli ifadeli değil), dediklerine göre bu aramayı hızlandırır.

agrep yaklaşık eşleşmeler yaparak **grep**'in yeteneklerini genişletir. Arama dizesi ile ortaya çıkan eşleşme, belirtilen karakter sayısı kadar farklı olabilir. Bu yardımcı, çekirdek Linux dağıtımının bir parçası değildir.

Sıkıştırılmış dosyaları aramak için, **zgrep**, **zegrep** veya **zfgrep** kullanın. Bunlar, sıkıştırılmış olmayan dosyalar üzerinde de çalışır, **grep**, **egrep**, **fgrep**'ten daha yavaş olsa da. Özellikle, karışık bir dosya seti, bazıları sıkıştırılmış, bazıları sıkıştırılmış olmayan dosyalar üzerinden arama yapmak için kullanışlıdır.

bzip edilmiş dosyaları aramak için, **bzgrep** kullanın.

look

look komutu **grep** gibi çalışır, ama bir "sözlük", sıralanmış bir sözcük listesi üzerinden arama yapar. Varsayılan olarak, **look** `/usr/dict/words` ile bir eşleyişi arar, ancak farklı bir sözlük dosyası da belirtilebilir.

ÖRNEK 12.13 BİR LİSTEDEKİ SÖZCÜKLERİN GEÇERLİLİK İÇİN KONTROL EDİLMESİ

```
#!/bin/bash

# lookup: Does a dictionary lookup on each word in a data file.

file=words.data  # Data file from which to read words to test.

echo


while [ “$word” != end ]  # Last word in data file.

do

  read word      # From data file, because of redirection at end of loop.

  look $word > /dev/null  # Don’t want to display lines in directory file.

  lookup=$?      # Exit status of ‘ look’ command.


  if [ “$lookup” -eq 0 ]

  then
```

```
    echo "\"$word\" is valid."
  else
    echo "\"$word\" is invalid."
  fi
done < "$file"   # Redirects stdin to $file, so "reads" come from there.
echo
exit 0
# ----------------------------------------------------------------
# Code below line will not execute because of "exit" command above.
# Stephane Chazelas proposes the following, more concise alternative:
while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" is valid."
   else echo "\"$word\" is invalid."
   fi
done <"$file"
exit 0
```

sed, awk

Betik dilleri, özellikle metin dosyalarını ve komut çıkışını ayrıştırmak için uygundur. Tek başlarına veya oluk ve kabuk betikleri ile birleşik olarak gömülebilirler.

sed

Etkileşimli-olmayan "dizgi editörü"dür, toplu moda birçok **ex** komutunun kullanılmasına izin verir. Kabuk betiklerinde pek çok kullanım alanı bulmaktadır.

awk

Programlanabilir dosya çıkarıcı ve biçimlendiricidir, yapılandırılmış metin dosyaları içindeki alanları (sütunları) değiştirmek veya ayıklamak için iyidir. Sözdizimi C'ye benzer.

wc

wc bir dosyadaki veya Girdi/Çıktı akışındaki "kelime sayısı"nı verir:

```
bash $ wc /usr/doc/sed-3.02/README

20     127     838 /use/doc/sed-3.02/README

[20 lines  127 words  838 characters]
```

wc -w sadece kelime sayısı verir.

wc -l sadece satır sayısı verir.

wc -c sadece karakter sayısı verir.

wc -L sadece en uzun satırın uzunluğunu verir.

Geçerli çalışma dizininde kaç tane *.txt* dosyasının bulunduğunu saymak için **wc** kullanımı:

```
$ ls *.txt | wc -l

# Will work as long as none of the "*.txt" files have a linefeed in their
name.

# Alternative ways of doing this are:

#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .

#      (shopt -s nullglob; set -- *.txt; echo $#)

# Thanks, S.C.
```

İsimleri d-h aralığındaki harfler ile başlayan tüm dosyaların boyutlarını toplamak için **wc** kullanımı :

```
bash$ wc [d-h]* | grep total | awk ' {print $3}'

71832
```

Bu kitabın ana kaynak dosyasında kaç defa "Linux" sözcüğünün geçtiğini bulmak için **wc** kullanımı:

```
bash$ grep Linux abs-book.sgml | wc –l

50
```

Ayrıca bkz. Örnek 12.30 ve Örnek 16.7.

Belirli bazı komutlar **wc** işlevselliğinin bir kısmını seçenek olarak alır.

```
... | grep foo | wc –l

# This frequently used construct can be more concisely rendered.

... | grep –c foo

# Just use the "-c" (or "—count") option of grep.

# Thanks, S.C.
```

tr

Karakter çeviri filtresidir.

Uygun olacak şekilde tırnak ve/veya parantez kullanmanız gerekir. Tırnak, kabuğun **tr** komut dizisindeki özel karakterleri yeniden yorumlamasına engel olur. Parantezler, kabuk tarafından açılımının önlenmesi için tırnak içine alınması gerekir.

`tr "A-Z" "*" <dosyadı` ve **`tr A-Z \* <dosyadı`** komutlarının her ikisi de dosyadaki tüm büyük harfleri yıldız (asteriks) olarak değiştirir ve `stdout`'a yazar. Bazı sistemlerde bu çalışmayabilir, ama **`tr A-Z ' [**]'`** mutlaka çalışacaktır.

`-d` seçeneği bir karakter aralığını siler.

```
echo "abcdef"                  # abcdef

echo "abcdef" | tr -d b-d      # aef

tr -d 0-9 < filename

# Deletes all digits from the file "filename".
```

`--squeeze-repeats` (veya `-s`) seçeneği ardışık karakter dizelerinin tümünü, sadece ilk harfi koruyarak, siler. Bu seçenek, fazladan yazılmış olan boşluk karakterlerini yok etmek için yararlıdır.

```
bash$ echo "XXXXX" | tr --squeeze-repeats ' X'

X
```

`-c` "tümleyici" seçeneği eşleşen karakterleri *tersine* çevirir. Bu seçenekle, **tr** sadece belirtilen kümeye uymayan karakterler üzerinden hareket eder.

```
bash$ echo "acfdeb123" | tr -c b-d +

+c+d+b++++
```

Unutmayınız ki, **tr** POSIX karakter sınıflarını tanır. [1]

```
bash$ echo "abcd2ef1" | tr ' [:alpha:]' -

----2--1
```

ÖRNEK 12.14 toupper: DOSYANIN TÜMÜNÜ BÜYÜK HARFE DÖNÜŞTÜRÜR.

```
#!/bin/bash
# Changes a file to all uppercase.
E_BADARGS=65

if [ -z "$1" ]  # Standard check for command line arg.
then
  echo "Usage: ' basename $0' filename"
  exit $E_BADARGS
fi

tr a-z A-Z < "$1"
# Same effect as above, but using POSIX character set notation:
#        tr ' [:lower:]' ' [:upper:]' <"$1"
# Thanks, S.C.
exit 0
```

ÖRNEK 12.15 lowercase: ÇALIŞMA DİZİNİNDE BULUNAN TÜM DOSYA ADLARINI KÜÇÜK HARFE DEĞİŞTİRİR.

```
#!/bin/bash
#
# Changes every filename in working directory to all lowercase.
#
# Inspired by a script of John Dubois,
# which was translated into Bash by Chet Ramey,
# and considerably simplified by Mendel Cooper, author of this document.
```

```
for filename in *                   # Traverse all files in directory.
do
   fname='basename $filename'
   n='echo $fname | tr A-Z a-z'     # Change name to lowercase.

   if [ "$fname" != "$n" ]          # Rename only files not already
lowercase.
   then
     mv $fname $n
   fi
done
exit 0

# Code below this line will not execute because of "exit".
# --------------------------------------------------------#
# To run it, delete script above line.
# The above script will not work on filenames containing blanks or
newlines.
# Stephane Chazelas therefore suggests the following alternative:
for filename in *     # Not necessary to use basename,
                      # since "*" won't return any file containing "/".
do n='echo "$filename/" | tr ' [:upper:]' ' [:lower:]'
#                             POSIX char set notation.
#                     Slash added so that trailing newlines are not
#                     removed by command substitution.
  # Variable substitution:
  n=${n%/}            # Removes trailing slash, added above, from filename.
  [[ $filename == $n ]] || mv "$filename" "$n"
                      # Checks if filename already lowercase.
done
exit 0
```

ÖRNEK 12.16 du: DOS'tan UNIX'e METİN DOSYASI DÖNÜŞTÜRME.

```
#!/bin/bash

# du.sh: DOS to UNIX text file converter.

E_WRONGARGS=65

if [ -z "$1" ]

then

  echo "Usage: ' basename $0' filename-to-convert"

  exit $E_WRONGARGS

fi

NEWFILENAME=$1.unx

CR='\015'  # Carriage return.

# Lines in a DOS text file end in a CR-LF.

tr –d $CR < $1 > $NEWFILENAME

# Delete CR and write to new file.

echo "Original DOS text file is \"$1\"."

echo "Converted UNIX text file is \"$NEWFILENAME\"."

exit 0
```

ÖRNEK 12.17 rot13: ULTRA ZAYIF ŞİFRELEME.

```
#!/bin/bash

# rot13.sh: Classic rot13 algorithm, encryption that might fool a 3-year
old.

# Usage: ./rot13.sh filename

# or     ./rot13.sh <filename

# or     ./rot13.sh and supply keyboard input (stdin)


cat "$@" | tr ' a-zA-Z' ' n-za-mN-ZA-M'   # "a" goes to "n", "b" to "o", etc.

# The ' cat "$@"' construction

# permits getting input either from stdin or from files.

exit 0
```

ÖRNEK 12.18 "KRİPTO-ALINTI" BULMACALAR OLUŞTURMA

```
#!/bin/bash

# crypto-quote.sh: Encrypt quotes

# Will encrypt famous quotes in a simple monoalphabetic substitution.

#  The result is similar to the "Crypto Quote" puzzles

#+ seen in the Op Ed pages of the Sunday paper.


key=ETAOINSHRDLUBCFGJMQPVWZYXK

# The "key" is nothing more than a scrambled alphabet.

# Changing the "key" changes the encryption.

# The ' cat "$@"' construction gets input either from stdin or from files.

# If using stdin, terminate input with a Control-D.

# Otherwise, specify filename as command-line parameter.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"

# Will work on lowercase, uppercase, or mixed-case quotes.

# Passes non-alphabetic characters through unchanged.


# Try this script with something like

# "Nothing so seeds reforming as other people's habits."

# --Mark Twain

#

# Output is:

# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."

# --BEML PZERC

# To reverse the encryption:

# cat "$@" | tr "$key" "A-Z"


#  This simple-minded cipher can be broken by an average 12-year old

#+ using only pencil and paper.

exit 0
```

tr değişkeleri

tr programının iki tarihi çeşidi vardır. BSD sürümü parantez kullanmaz **(tr a-z A-Z)**, ama SysV sürümü kullanır **(tr ' [a-z]' ' [A-Z]'**). GNU sürümünde harf aralıklarının parantez içine alınması zorunludur.

fmt

Basitçe bir dosya biçimlendiricidir, uzun satırlı metin çıktılarını "kaydırmak" için bir oluğun içinde filtre olarak kullanılır.

ÖRNEK 12.19 BİÇİMLENDİRİLMİŞ DOSYA LİSTESİ.

```
#!/bin/bash

WIDTH=40                  # 40 columns wide.

b='ls /usr/local/bin'     # Get a file listing...

echo $b | fmt –w $WIDTH

# Could also have been done by

#  echo $b | fold - -s –w $WIDTH

exit 0
```

Ayrıca bkz. Örnek 12-4.

col

Bu filtre bir girdi akışından ters satır beslemelerini kaldırır. Ayrıca boşluk karakterlerin yerine eşdeğer sekmeleri yazmaya çalışır. **col** programının başlıca kullanımı **groff** ve **tbl** gibi bazı metin işleme araçlarının çıktısını filtrelemesidir.

column

Sütun biçimlendiricidir. Bu filtre uygun yerlerde sekmeler ekleyerek, liste-türünden metinleri iyi-formatlanmış tablolara dönüştürür.

ÖRNEK 12.20 BİR DİZİN LİSTESİNİ BİÇİMLENDİRMEK İÇİN column KULLANILMASI

```
#!/bin/bash
#This is a slight modification of the example file in the "column" manpage.
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t
#  The "sed ld" in the pipe deletes the first line of output,
#+ which would be "total        N",
#+ where "N" is the total number of files found by "ls -l".
# The -t option to "column" pretty-prints a table.
exit 0
```

colrm

Sütun kaldırma filtresidir. Bu bir dosyadan sütunları (karakterleri) kaldırır, ve belirtilen sütun aralığından arınmış halde, dosyayı `stdout`'a yazar. **`colrm 2 4 <dosyadı`** dosyadı adlı metin dosyasının her satırından ikinci ve dördüncü karakterler arasında kalanları kaldırır.

Dosya sekme veya yazdırılamayan karakterleri içeriyorsa, bu öngörülemeyen davranışlara neden olabilir. Bu gibi durumlarda, bir oluk yaratarak **colrm** öncesinde **expand** ve **unexpand** kullanmayı düşünün.

nl

Satır numaralandırması filtresidir. **`nl dosyadı`** dosyadı adlı dosyayı, boş olmayan her satırın başına ardışık birer satır numarası ekleyerek `stdout'a` yazdırır. Dosya adı belirtilmezse, `stdin` üzerinde çalışır.

nl çıktısı `cat -n` ile çok benzeşir, ancak, varsayılan olarak **nl** boş satırları listelemez.

ÖRNEK 12.21 nl: KENDİ KENDİNİ NUMARALANDIRAN KOMUT DOSYASI.

```
#!/bin/bash
# This script echoes itself twice to stdout with its lines numbered.

# ' nl' sees this as line 3 since it does not numberblank lines.
# ' cat -n' sees the above line as number 5.
nl ' basename $0'
echo; echo  # Now, let's try it with ' cat -n'

cat -n ' basename $0'
# The difference is that ' cat -n' numbers the blank lines.
# Note that ' nl -ba' will also do so.
exit 0
```

pr

Basım biçimlendirme filtresidir. Dosyaları (veya `stdout`) yazılı çıktı veya ekranda görüntülemek üzere uygun bölümler halinde sayfalandırır. Sahip olduğu çeşitli seçenekler, satır ve sütun manipülasyonuna izin verir, satırları birleştirir, kenar boşluklarını ayarlar, satırları numaralandırır, sayfa başlıkları ekler ve dosyaları birleştirir, ve diğer seçenekler sayılabilir. **pr** komutu **nl**, **paste**, **fold**, **column** ve **expand** işlevselliklerinin çoğunu bünyesinde barındırır.

`pr –o 5 –width=65 fileZZZ | more` fileZZZ dosyasının ekranda kenar boşlukları 5 ve 65 olan güzel bir listesini çıkarır.

Özellikle yararlı bir başka seçeneği de, çift satır aralıklı liste çıkarmaya zorlayan `-d` seçeneğidir. (**sed -G** ile aynı etkiye sahiptir).

gettext

Lokalizasyon ve programların metin çıktılarının yabancı dillere tercümesi için geliştirilmiş bir GNU programıdır. Öncelikle C programları için tasarlanmış olsa da, **gettext** kabuk komut dosyalarında da kullanım alanı bulmaktadır. bkz. *`info page`*

iconv

Dosya (lar)ı farklı bir kodlamaya (karakter kümesi) dönüştürmek için geliştirilmiş bir yardımcı programdır. Başlıca kullanım alanı lokalizasyon içindir.

recode

Bunu daha detaylı bir **iconv** sürümü olarak düşününüz. Bir dosyayı farklı bir kodlamaya dönüştürmeye yarayan bu çok yönlü program, standart Linux yüklemesinin bir parçası değildir.

TeX, gs

TeX ve **Postscript**, biçimlendirilmiş video görüntü kopyası veya baskısı hazırlamakta kullanılan metin biçimlendirme dilleridir.

TeX Donald Knuth'un ayrıntılı dizgi sistemidir. Genellikle yazılan bir kabuk betiğinin bu biçimlendirme dillerinden birine iletilen tüm seçenekleri ve argümanları içine alması uygundur.

Ghostscript (**gs**) bir GPL Postscript yorumlayıcısıdır.

groff, tbl, eqn

groff, bir diğer metin ve gösterim biçimlendirme dilidir. Bu, UNIX **roff / troff** gösterim ve dizgi paketinin gelişmiş GNU sürümüdür. *Man sayfaları* **groff** kullanır (bkz.Örnek A-1).

tbl tablo işleme programı **groff**'un parçası olarak kabul edilir, çünkü işlevi tablo biçimlendirmesini **groff** komutlarına dönüştürmektir.

eqn denklem işleme programı da, benzer şekilde **groff**'un bir parçasıdır, ve işlevi denklem biçimlendirmesini **groff** komutlarına dönüştürmektir.

lex, yacc

lex sözcüksel analizcisi, kalıp eşleştirmeler yapmak için programlar üretir. Linux sistemlerinde bu program **flex** ile yer değiştirmiştir.

yacc programı bir dizi belirtime dayalı bir ayrıştırıcı oluşturur. Linux sistemlerinde bu program **bison** ile yer değiştirmiştir.

Notlar

[1] Bu sadece **tr** GNU sürümü için doğrudur, genellikle ticari UNIX sistemlerinde mevcut olan genel sürümü için doğru değildir.

12.5 DOSYA ve ARŞİVLEME KOMUTLARI

Arşivleme

tar

Standart UNIX arşivleme programıdır. Orijinal olarak, *Tape ARchiving* sözcüklerinin kısaltması olan bu program, her türlü arşivlemeyi teyp sürücülerinden düzenli dosyalara hatta `stdout`'a kadar, her türlü hedef aygıt için işleyebilen genel amaçlı bir paket haline gelmiştir (bkz Örnek 4-4). GNU tar çeşitli sıkıştırma filtrelerini kabul edecek şekilde düzeltilmiştir, örneğin **tar czvf arşiv_adı.tar.gz** * bir dizin ağacındaki tüm dosyaları, geçerli çalışma dizininde (**$PWD**) bulunan adı nokta ile başlayan tüm dosyalar dışında, ardışık olarak arşivler ve gzip'ler. [1]

Bazı yararlı **tar** seçenekleri:

1. `-c` yeni arşiv oluşturur.
2. `-x` mevcut arşiv dosyaları ayıklar.
3. `--delete` mevcut arşiv dosyaları siler.

   Bu seçenek, manyetik teyp cihazlarında çalışmaz.

4. `-r` mevcut arşivin sonuna dosyaları ekler.
5. `-A` mevcut arşivin sonuna *tar* dosyalarını ekler.
6. `-t` mevcut arşivin içeriğini listeler.
7. `-u` arşivi güncelleştirir.
8. `-d` belirtilen dosya sistemi ile arşivi karşılaştırır.
9. `-z` arşivi gzip'ler.
   `-c` veya `-x` seçeneği ile birlikte kullanılmasına göre sıkıştırır veya açar.
10. `-j` arşivi bzip2'ler.

    gzip ile sıkıştırılmış bozuk bir tar arşivinin verilerini kurtarmak zor olabilir. Önemli dosyaları arşivlerken, birden çok yedekleme yapınız.

shar

Kabuk arşivleme programıdır. Kabuk arşiv dosyaları sıkıştırılma olmadan da birleştirilebilirler, ve ortaya çıkan arşiv aslında #/bin/sh başlığı ile tamamlanmış ve gerekli tüm arşiv açma komutlarını içeren bir kabuk betiğidir. shar arşivleri hala İnternet haber gruplarında gözükmektedir, ama **tar/gzip** hoş bir şekilde **shar**'ın yerini almıştır. **unshar** komutu **shar** arşivlerini paketten çıkarmada kullanılır.

ar

Arşivler için oluşturma ve işleme programı, özellikle ikili nesne dosyası kitaplıkları için kullanılır.

cpio

Bu özel arşivleme kopyalama komutu (kopyalama girdi ve çıktı) artık nadiren görülmektedir, **tar/gzip** bu komutun yerine geçmiştir.

ÖRNEK 12.22 BİR DİZİN AĞACINI TAŞIMAK İÇİN cpio KULLANIMI

```
#!/bin/bash
# Copying a directory tree using cpio.
ARGS=2
E_BADARGS=65
if [ $# -ne "$ARGS"]
then
  echo "Usage: ' basename $0' source destination"
  exit $E_BADARGS
fi
```

```
source=$1

destination=$2

find "$source" –depth | cpio –admvp "$destination"

# Read the man page to decipher these cpio options.

exit 0
```

ÖRNEK 12.23 BİR rpm ARŞİVİNİ AÇMA

```
#!/bin/bash

# de-rpm.sh: Unpack an ' rpm' archive

E_NO_ARGS=65

TEMPFILE=$$.cpio                              # Tempfile with "unique" name.

                                              # $$ is process ID of script.

if [ -z "$1" ]

then

  echo "Usage: ' basename $0' filename"

exit $E_NO_ARGS

fi


rpm2cpio < $1 > $TEMPFILE                     # Converts rpm archive into cpio
archive.

cpio –-make-directories –F $TEMPFILE          # Unpacks cpio archive.

rm –f $TEMPFILE                               # Deletes cpio archive.
```

Sıkıştırma

gzip

Standart GNU/UNIX sıkıştırma programıdır, **compress**'in yerini almıştır. Sıkıştırmayı açma komutu **gunzip**'tir ve bu da **gzip -d** ile eşdeğerdir.

zcat filtresi *gzip'li* dosyayı `stdout`'a açar, bu da muhtemelen bir oluğa ya da yönlendirmeye girdi olacaktır. Bu, aslında, sıkıştırılmış dosyalar üzerinde çalışan bir **cat** komutudur (eski **compress** programı ile işlenmiş dosyalar da dahil olmak üzere). **zcat** komutu **gzip -dc** ile eşdeğerdir.

Bazı ticari UNIX sistemlerinde, **zcat**, **uncompress -c** ile eş anlamlıdır, ve *gzip'lenmiş* dosyalar üzerinde çalışmaz.

Ayrıca bkz. Örnek 7.6.

bzip2

Özellikle büyük dosyalar üzerinde çalışan, **gzip**'ten genellikle daha verimli (ama daha yavaş) bir alternatif sıkıştırma programıdır. İlgili sıkıştırmayı açma programı **bunzip2**'dir.

uncompress, compress

Bu ticari UNIX dağıtımlarında bulunan eski ve özel bir sıkıştırma programıdır. Daha verimli olan **gzip** programı, büyük ölçüde yerine geçmiştir. **compress** ile sıkıştırılan dosyalar **gunzip** ile açılabildiği halde, Linux dağıtımları genellikle, **compress**'in uyumluluğu için aynı şekilde çalışan bir açma programı da dahil etmişlerdir.

znew komutu *compress* ile sıkıştırılmış dosyaları *gzip* dosyalarına dönüştürür.

sq

Yine başka bir sıkıştırma programıdır, sadece sıralı ASCII sözcük listeleri üzerinde çalışan bir filtredir. Bu, bir filtre için standart yürütme sözdizimini kullanır, **sq < input-file > output-file**. Hızlıdır, ama gzip kadar verimli değildir. İlgili sıkıştırmayı açma filtresi **unsq** programıdır ve **sq** gibi çağrılır.

sq çıktısı daha fazla sıkıştırma için olukla gzip'e girdi olarak verilebilir.

zip, unzip

Çapraz platform dosya arşivleme ve DOS *pkzip.exe* ile uyumlu bir sıkıştırma programıdır. Zip'lenmiş "arşiv"ler "tar"lara göre İnternet'te daha kabul edilebilir bir aktarım ortamı gibi görünmektedir.

unarc, unarj, unrar

Bu Linux programları, DOS *arc.exe, arj.exe* ve *rar.exe* programları ile sıkıştırılmış arşivlerin açılmasına izin verir.

Dosya Bilgileri

file

Dosya türlerini tanımlamak için bir yardımcıdır. **`file dosyadı`** komutu, `dosyadı` için dosya özelliklerini döndürecektir, bu `ASCII, metin` veya `veri` olabilir. Linux / UNIX dağıtımına bağlı olarak `/usr/share/magic, /etc/magic`, veya `/usr/lib/magic` dosyalarında bulunan sihirli sayılara referans verir.

`-f` seçeneği, **file** komutunun toplu modda çalışmasına neden olur, belirli bir dosyadan analiz etmek için dosya listesini okutur. `-z` seçeneği, sıkıştırılmış bir hedef dosya üzerinde kullanıldığı zaman, sıkıştırılmamış dosya türünü analiz etme girişimi için zorlar.

```
bash$ file test.tar.gz

test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16
13:34:51 2001, os: Unix




bash$ file –z test.tar.gz

test.tar.gz: GNU tar archive (gzip compressed data, deflated, last
modified: Sun Sep 16 13:34:51 2001, os: Unix)
```

ÖRNEK 12.24 C PROGRAMI DOSYALARINDAN YORUMLARI ORTADA KALDIRMA

```
#!/bin/bash

# strip-comment.sh: Strips out the comments (/* COMMENT */) in a C program.


E_NOARGS=65

E_ARGERROR=66

E_WRONG_FILE_TYPE=67

if [ $# -eq “$E_NOARGS” ]

then

  echo “Usage: ‘ basename $0’ C-program-file” >&2  # Error message to
stderr.

  exit $E_ARGERROR

fi


# Test for correct file type.

type=’eval file $1 | awk ‘ { print $2, $3, $4, $5 }’
```

```
# "file $1" echoes file type...
# then awk removes the first field of this, the filename...
# then the result is fed into the variable "type".
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "This script works on C program files only."
  echo
  exit $E_WRONG_FILE_TYPE
fi
# Rather cryptic sed script:
Sed '
/^\/\*/d
/.*\/\*/d
'  $1
# Easy to understand if you take several hours to learn sed fundementals.

#  Need to add one more line to the sed script to deal with
#+ case where line of code has a comment following it on same line.
#  This is left as a non-trivial exercise.

# Also, the above code deletes with a "*/" or "/*",
# not a desirable result.
exit 0

# Code below this line will not execute because of 'exit 0' above.
# Stephane Chazelas suggests the following alternative:
usage() {
```

```
  echo "Usage: ' basename $0' C-program-file" >&2
  exit 1
}
WEIRD='echo -n -e ' \377''    # or WEIRD=$'\377'
[[ $# -eq 1 ]] || usage
case ' file "$1"' in
  *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
      | tr ' \377\n' ' \n\377' \
      | sed -ne ' p;n' \
      | tr -d ' \n' | tr ' \377' ' \n';;
  *) usage;;
esac

# This is still fooled by things like:
# printf("/*");
# or
# /*   /* buggy embedded comment */
#
# To handle all special cases (comments in strings, comments in string
# where there is a \", \\" ...) the only way is to write a C parser
# (lex or yacc perhaps?).
exit 0
```

which

which komut-xxx "komut-xxx"in tam yolunu verir. Bu, belirli bir komut veya programın sistemde yüklü olup olmadığını bulmak için yararlıdır.

```
bash$ which rm
/usr/bin/rm
```

whereis

Yukarıda tanımlanan **which** komutuna benzer olarak, **whereis komut-xxx** sadece "komut-xxx"in değil, ama aynı zamanda *man sayfasının* da tam yolunu verir.

```
bash$ whereis rm

rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis dosyaxxx *whatis* veritabanında "dosyaxxx" arar. Bu, sistem komutlarının ve önemli yapılandırma dosyalarının belirlenmesi için yararlıdır. Bunu basitleştirilmiş bir **man** komutu olarak da düşünebilirsiniz.

```
bash$ whatis whatis

whatis                (1)   - search the whatis database for complete words
```

ÖRNEK 12.25 /usr/X11R6/bin DİZİNİNİ ARAŞTIRMAK

```
#!/bin/bash

# What are all those mysterious binaries in /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"

# Try also "/bin", "/usr/bin", "/usr/local/bin", etc.


for file in $DIRECTORY/*

do

  whatis ' basename $file'  # Echoes info about the binary.

done

exit 0
```

```
# You may wish to redirect output of this script, like so:
# ./what.sh >>whatis.db
# or view it a page at a time on stdout,
# ./what.sh | less
```

Ayrıca bkz. Örnek 10.3.

vdir

Ayrıntılı dizin listesini gösterir. Etkisi **ls -l** komutuna benzer.

Bu, GNU dosya araçlarından biridir.

```
bash$ vdir
total 10
 -rw-r--r--      1 bozo  bozo      4034 Jul 18 22:04  data1.xrolo
 -rw-r--r--      1 bozo bozo       4602 May 25 13:58  data1.xrolo.bak
 -rw-r--r--      1 bozo bozo        877 Dec 17 2000   employment.xrolo

bash$ ls –l
total 10
 -rw-r--r--      1 bozo  bozo      4034 Jul 18 22:04  data1.xrolo
 -rw-r--r--      1 bozo bozo       4602 May 25 13:58  data1.xrolo.bak
 -rw-r--r--      1 bozo bozo        877 Dec 17 2000   employment.xrolo
```

shred

Silmeden önce rasgele bit desenlerini birden çok kez üzerine yazarak güvenli şekilde bir dosya siler. Bu komut Örnek 12.41 ile aynı etkiye sahiptir, ama daha kapsamlı ve zarif bir şekilde yapar.

Bu, GNU dosya araçlarından biridir.

Bir dosya üzerinde **shred** kullanılması, gelişmiş teknolojileri kullanarak içeriğin bir kısmının veya tamamının geri kazanılmasını engellemez.

locate, slocate

locate komutu, sadece bu amaç için saklanan bir veritabanını kullanarak dosya arar. **slocate** komutu **locate** komutunun (**slocate**'in öteki adı olarak tanımlanabilir) güvenli olan sürümüdür.

```
bash$ locate hickson

/usr/lib/xephem/catalogs/hickson.edb
```

strings

Bir ikili veya veri dosyasında yazdırılabilir dizgileri bulmak için **strings** komutunu kullanınız. Bu, hedef dosyada bulunan yazdırılabilir karakter dizilerini listeler. Bu, bir çekirdek bellek dökümünün hızlıca incelenmesi için ya da bilinmeyen bir grafik görüntü dosyasına bakmak için kullanım alanı bulabilir. (`strings image-file | more` `JFIF` gibi bir şey gösterebilir, ki bu dosyayı bir *jpeg* grafik olarak tanımlamak olacaktır). Bir komut dosyasında, muhtemelen grep veya sed ile **strings** çıktısını ayrıştırmak isteyebilirsiniz. Bkz.Örnek 10.7 ve Örnek 10.9.

ÖRNEK 12.26 BİR "GELİŞMİŞ" *strings* KOMUTU

```
#!/bin/bash

# wstrings.sh: “word-strings” (enhanced “strings” command)

#

#  This script filters the output of “strings” by checking it

#+ against a standard word list file.
```

```
#  This effectively eliminates all the gibberish and noise,
#+ and outputs only recognized words.
# ============================================================
#                   Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=65
E_NOFILE=66
if [ $# -ne $ARGS ]
then
  echo “Usage: ‘ basename $0’ filename”
  exit $E_BADARGS
fi
if [ -f “$1” ]                          # Check if file exists.
then
    file_name=$1
else
    echo “File \”$1\” does not exist.”
    exit $E_NOFILE
fi
# ============================================================


MINSTRLEN=3                              #  Minimum string length.
WORDFILE=/usr/share/dict/linux.words     #  Dictionary file.
                                         #  May specify a different
                                         #+ word list file
                                         #+ of format 1 word per line.
wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
# Translate output of 'strings' command with multiple passes of 'tr'.
#  "tr A-Z a-z" converts to lowercase.
```

```
#  "tr '[:space:]'" converts whitespace characters to Z's.
#  "tr -cs '[:alpha:]' Z" converts non-alphabetic characters to Z's,
#+ and squeezes multiple consecutive Z's.
#  "tr -s '\173-\377' Z"  converts all characters past 'z' to Z's
#+ and squeezes multiple consecutive Z's,
#+ which gets rid of all the weird characters that the previous
#+ translation failed to deal with.
#  Finally, "tr Z ' '" converts all those Z's to whitespace,
#+ which will be seen as word separators in the loop below.

# Note the technique of feeding the output of 'tr' back to itself,
#+ but with different arguments and/or options on each pass.

for word in $wlist                                  # Important:
                                                    # $wlist must not be quoted
here.
                                                    # “$wlist” does not work.
                                                    # Why?
do
  strlen=${#word}                                   # String length.
  if [ "$strlen" -lt "$MINSTRLEN" ]                 # Skip over short strings.
  then
    continue
  fi
  grep -Fw $word "$WORDFILE"                        # Match whole words only.
done

exit 0
```

Karşılaştırmalar

diff, patch

diff: Esnek bir dosya karşılaştırma programıdır. Hedef dosyaları sırayla satır-satır karşılaştırır. Kelime sözlüklerini karşılaştırma gibi bazı uygulamalarda, dosyaların olukla **diff** programına verilmeden önce **sort** ve **uniq** yoluyla filtrelenmesi yararlı olabilir. `diff dosya-1 dosya-2` dosyalardaki farklı satırları çıktı olarak listeler, her bir satırın ait olduğu dosya, satır başında düzeltme işareti (`^`) ile gösterilir.

diff komutu ile birlikte kullanılan `--side-by-side` seçeneği, karşılaştırılan her dosyayı, satır satır, ayrı sütunlarda, eşleşmeyen satırlar işaretli halde gösterir.

diff komutunun farklı önyüzleri mevcuttur, örneğin **spiff**, **wdiff, xdiff** ve **mgdiff** komutları bunların arasındadır.

diff komutu, karşılaştırılan dosyalar aynı ise 0 çıkış durumu döner, farklı ise 1 çıkış durumu döner. Bu bir kabuk betiği içindeki bir test yapısı içinde kullanım alanı bulabilir .

diff için yaygın bir kullanım alanı da, **patch** ile kullanılmak üzere fark dosyaları oluşturmaktır.

`-e` seçeneği **ed** veya **ex** betikleri için uygun dosyaları çıktı olarak üretir.

patch: Esnek sürüm programıdır. **diff** tarafından oluşturulan bir fark dosyası verildiğinde, **patch**, paketi önceki sürümünden daha yeni bir sürüme yükseltebilir. Yeni gözden geçirilmiş paketin tümüne kıyasla, nispeten daha küçük bir "fark" dosyası dağıtmak uygun olacaktır. Çekirdek "yamaları" Linux çekirdeğinin dağıtımlarında oldukça sık tercih edilen yöntem haline gelmiştir.

```
patch -p1 <patch-file
# Takes all the changes listed in 'patch-file'
# and applies them to the files referenced therein.
# This upgrades to a newer version of the package.
```

Çekirdeğe yama yapmak:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Upgrading kernel source using 'patch'.
# From the Linux kernel docs "README",
# by anonymous author (Alan Cox?).
```

diff komutu özyineli olarak dizinleri de karşılaştırabilir (varolan dosya adları için).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
 Only in /home/bozo/notes1: file03
 Only in /home/bozo/notes2: file04
```

gzip'lenmiş dosyaları karşılaştırmak için **zdiff** kullanınız.

diff3

Bir seferde üç dosyayı karşılaştıran genişletilmiş **diff** versiyonudur. Bu komut, başarılı bir şekilde yürütülmesi halinde 0 çıkış değeri ile döner, ama ne yazık ki bu, karşılaştırma sonuçları hakkında bilgi vermez.

```
bash$ diff3 file-1 file-2 file-3

====

 1:1c

   This is line 1 of "file-1".

 2:1c

   This is line 1 of "file-2".

 3:1c

   This is line 1 of "file-3".
```

sdiff

Bir çıktı dosyasında birleştirilmek üzere iki dosyayı karşılaştırır ve/veya düzenler. Etkileşimli olan doğası yüzünden, bir komut dosyasında kullanımı çok azdır.

cmp

cmp komutu yukarıdaki **diff** programının daha basit bir versiyonudur. **diff** iki dosya arasındaki farkları rapor etse de, **cmp** onların sadece hangi noktalarda farklı olduğunu gösterir.

Karşılaştırılan dosyalar aynı ise **diff** gibi, **cmp** da 0 çıkış durumunu döner, ve dosyalar farklı ise 1 döner. Bu, bir kabuk betiği içindeki test yapısında kullanım alanı bulabilir.

ÖRNEK 12.27 BİR KOMUT DOSYASI İÇİNDE İKİ DOSYAYI KARŞILAŞTIRMAK İÇİN cmp KULLANIMI.

```
#!/bin/bash

ARGS=2 # Two args to script expected.

E_BADARGS=65

E_UNREADABLE=66
```

```
if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` file1 file2"
  exit $E_BADARGS
fi


if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Both files to be compared must exist and be readable."
  exit $E_UNREADABLE
fi


cmp $1 $2 &> /dev/null # /dev/null buries the output of the "cmp" command.
# Also works with 'diff', i.e., diff $1 $2 &> /dev/null
if [ $? -eq 0 ]        # Test exit status of "cmp" command.
then
  echo "File \"$1\" is identical to file \"$2\"."
else
  echo "File \"$1\" differs from file \"$2\"."
fi
exit 0
```

gzip'lenmiş dosyalarda **zcmp** kullanınız.

comm

Çok yönlü dosya karşılaştırma programıdır. Yararlı olması için dosyaların sıralanmış olması kriteri gerekir.

comm *–seçenekler dosya-1 dosya-2*

`comm dosya-1 dosya-2` çıktı olarak 3 sütun üretir.

- sütun 1 = sadece dosya-1' e özgü olan satırlar
- sütun 2 = sadece dosya-2' ye özgü olan satırlar
- sütun 3 = her ikisine de ortak olan satırlar

Seçenekler bir veya daha fazla sütunun çıktı olarak görüntülenmesini engeller.

- `-1` sütun 1'i engeller.
- `-2` sütun 2'yi engeller.
- `-3` sütun 3'i engeller.
- `-12` hem sütun 1'i hem sütun 2'yi engeller, vb.

Yardımcı Programlar

basename

Bir dosya adından yol bilgisini çıkarır, sadece dosya adı kalır. **`basename $0`** yapısı, komut dosyasına adını bildirir, yani kendisini çağıran komut dosyasının adı. Bu, eğer, örneğin bir komut dosyası eksik argümanla çağrıldığı takdirde, "kullanım" mesajları için kullanılabilir,

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Bir dosya adından **basename** bilgisini çıkarır, sadece yol bilgisi kalır.

basename ve **dirname** herhangi bir rasgele dizgi üzerinde çalışabilir. Argümanın, mevcut bir dosyaya ait olması gerekmez, hatta bir dosya adı olması bile gerekmez (bkz. Örnek A-8).

ÖRNEK 12.28 basename ve dirname

```
#!/bin/bash

a=/home/bozo/daily-journal.txt

echo "Basename of /home/bozo/daily-journal.txt = `basename $a`"

echo "Dirname of /home/bozo/daily-journal.txt = `dirname $a`"

echo

echo "My own home is `basename ~/`."         # Also works with just ~.

echo "The home of my home is `dirname ~/`."  # Also works with just ~.

exit 0
```

split

Bir dosyayı küçük parçalar halinde bölmek için yardımcı programdır. Genellikle disketler üzerinde yedekleme veya e-posta ya da yüklemeye hazırlık yapmak amacıyla büyük dosyaları bölmek için kullanılır.

sum, cksum, md5sum

Sağlama oluşturmak için araçtır. *Sağlama* matematiksel olarak bir dosyanın bütünlüğünü kontrol etmek amacıyla, dosyanın içeriğine göre hesaplanan bir sayıdır. Bir komut dosyası güvenlik amacıyla sağlama listesine başvurabilir, önemli sistem dosyalarının içeriğinin değiştirilmediği veya bozulmadığını sağlamak gibi. Güvenlik uygulamaları için, 128-bit **md5sum** (ileti özeti sağlama toplamı) komutunu kullanınız.

```
bash$ cksum /boot/vmlinuz

1670054224 804083 /boot/vmlinuz



bash$ md5sum /boot/vmlinuz

0f43eccea8f09e0a0b2b5cf1dcf333ba /boot/vmlinuz
```

Unutmayınız ki, **cksum** aynı zamanda hedef dosyanın boyutunu da, bayt cinsinden, gösterir.

ÖRNEK 12.29 DOSYA BÜTÜNLÜĞÜNÜN KONTROL EDİLMESİ

```
#!/bin/bash
# file-integrity.sh: Checking whether files in a given directory
#                    have been tampered with.
E_DIR_NOMATCH=70
E_BAD_DBFILE=71
dbfile = File_record.md5
# Filename for storing records.

set_up_database ()
{
  echo ""$directory"" > "$dbfile"
  # Write directory name to first line of file
  md5sum "$directory"/* >> "$dbfile"
  # Append md5 checksums and filenames.
}

check_database ()
{
  local n=0
  local filename
  local checksum
  # ------------------------------------------- #
  # This file check should be unnecessary,
  #+ but better safe than sorry.
```

```
  if [ ! -r "$dbfile" ]
  then
    echo "Unable to read checksum database file!"
    exit $E_BAD_DBFILE
  fi
  # ----------------------------------------- #
  while read record[n]
  do
    directory_checked="${record[0]}"
    if [ "$directory_checked" != "$directory" ]
    then
      echo "Directories do not match up!"
      # Tried to use file for a different directory.
      exit $E_DIR_NOMATCH
    fi

    if [ "$n" -gt 0 ]   # Not directory name.
    then
      filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
      # md5sum writes records backwards,
      #+ checksum first, then filename.
      checksum[n]=$( md5sum "${filename[n]}" )

      if [ "${record[n]}" = "${checksum[n]}" ]
      then
        echo "${filename[n]} unchanged."
      else
        echo "${filename[n]} : CHECKSUM ERROR!"
        # File has been changed since last checked.
      fi
```

```
    fi


    let "n+=1"
  done <"$dbfile" # Read from checksum database file.
}
# ================================================== #
# main ()
if [ -z "$1" ]
then
  directory="$PWD"      # If not specified,
else                    #+ use current working directory.
  directory="$1"
fi
clear                   # Clear screen.
# ------------------------------------------------------------------ #
  if [ ! -r "$dbfile" ] # Need to create database file?
  then
    echo "Setting up database file, \""$directory"/"$dbfile"\"."; echo
    set_up_database
  fi
# ------------------------------------------------------------------ #
check_database          # Do the actual work.
echo
#  You may wish to redirect the stdout of this script to a file,
#+ especially if the directory checked has many files in it.


#  For a much more thorough file integrity check,
#+ consider the "Tripwire" package,
#+ http://sourceforge.net/projects/tripwire/.
exit 0
```

Kodlama ve Şifreleme

uuencode

Bu yardımcı program ikili dosyaları ASCII karakterler halinde kodlar, böylece bir e-posta iletisinin gövdesinde ya da bir haber grubundaki göndermeye iletim için en uygun hale getirir.

uudecode

Bu kodlamayı tersine yapar, uuencode'lanmış dosyaların şifresini çözer ve orijinal ikililer haline getirir.

ÖRNEK 12.30 KODLANMIŞ DOSYALARI uudecode İLE ÇÖZMEK

```
#!/bin/bash
lines=35          # Allow 35 lines for the header (very generous).
for File in *     # Test all the files in the current working directory...
do
  search1=`head -$lines $File | grep begin | wc -w`
  search2=`tail -$lines $File | grep end | wc -w`
  #  Uuencoded files have a "begin" near the beginning,
  #+ and an "end" near the end.
  if [ "$search1" -gt 0 ]
  then
    if [ "$search2" -gt 0 ]
    then
      echo "uudecoding - $File -"
      uudecode $File
    fi
```

```
  fi

done

#  Note that running this script upon itself fools it
#+ into thinking it is a uuencoded file,
#+ because it contains both "begin" and "end".
# Exercise:
# Modify this script to check for a newsgroup header.
exit 0
```

fold -s komutu (muhtemelen bir oluk içinde) Usenet haber gruplarından indirilebilen şifresi uudecode ile çözülmüş uzun metin mesajlarını işlemek için yararlı olabilir.

mimencode, mmencode

mimencode ve **mmencode** komutları multimedya olarak kodlanmış e-posta eklerini işler. Posta kullanıcı arayüzleri (örneğin **pine** veya **kmail** gibi) normalde otomatik olarak ele almasına rağmen, bu dikkate değer yardımcı programlar e-posta eklerinin manuel olarak komut satırından ya da bir toplu iş olarak kabuk betiğinden değiştirilmesine izin verir.

crypt

Bir zamanlar, bu standart UNIX dosya şifreleme programıydı. [2] Şifreleme yazılımının dışarı aktarımını yasaklayan siyasi amaçlı hükümet düzenlemeleri UNIX dünyasında ortadan kaybolan **crypt** ile sonuçlandı, ve hala da Linux dağıtımlarının çoğundan eksiktir. Neyse ki, programcılar buna bir dizi iyi alternatif ile geldi, aralarında yazarın kendisine ait olan cruft da bulunmaktadır (bkz. Örnek A.5).

Çeşitli

make

İkili paketleri oluşturmak ve derlemek için yardımcı programdır. Bu aynı zamanda kaynak dosyaların artımlı değişiklikleri tarafından tetiklenen herhangi bir işlem grubu için de kullanılabilir.

Dosya bağımlılıklarının ve yapılacak işlemlerin bir listesi olan `Makefile`, **make** komutu tarafından kontrol edilir.

install

cp'ye benzer özel amaçlı dosya kopyalama komutudur, ama izinleri ve kopyalanan dosyaların özelliklerini ayarlama yeteneğine sahiptir. Bu komut, yazılım paketlerini yüklemek için biçilmiş kaftan gibi görünüyor, ve bu şekilde `Makefile`'larda da sık sık kendini gösterebilir (*`make install:`* bölümünde). Aynı şekilde yükleme betiklerinde de kullanım alanı bulur.

ptx

ptx [hedefdosya] komutu, hedef dosyanın devşirimli indeksini (çapraz-referans listesi) verir. Gerekli olursa, bu, ayrıca, filtre edilebilir veya bir oluk içerisinde biçimlendirilebilir.

more, less

Bir metin dosyasını veya akış dizgisini stdout'ta her seferinde bir sayfa dolusu görüntüler. Bu bir komut dosyasının çıktısını filtre etmek için de kullanılabilir.

Notlar

[1] **tar czvf arşiv_adı.tar.gz *** geçerli çalışma dizini *altındaki* dizinlerde bulunan adı nokta ile başlayan dosyaları da dahil eder. Bu belgelenmemiş bir GNU **tar** "özelliği" dir.

[2] Bu, bir simetrik blok şifredir, "açık anahtar"lı şifrelemenin aksine, tek bir sistem ya da yerel ağ üzerindeki dosyaları şifrelemek için kullanılır. Açık anahtarlı şifrelemeye iyi bilinen bir örnek **pgp** programıdır.

12.6 İLETİŞİM KOMUTLARI

Bilgi ve İstatistikler

host

DNS kullanarak, ad veya IP adresine göre bir İnternet ana bilgisayarı hakkında bilgi arar.

vrfy

Internet e-posta adresini doğrular.

nslookup

IP adresine göre bir ana bilgisayarda İnternet "ad sunucusu arama"sı yapar. Bu bir komut dosyası içinde, ya etkileşimli veya etkileşimli olmayacak şekilde çalıştırılabilir.

dig

nslookup komutuna benzer şekilde, bir ana bilgisayarda internet "ad sunucusu arama"sı yapar. Bir komut dosyası içinde, ya etkileşimli veya etkileşimli olmayacak şekilde çalıştırılabilir.

traceroute

Bir uzak ana bilgisayara gönderilen paketler tarafından alınan yolu izler. Bu komut bir LAN, WAN içinde veya İnternet üzerinden çalışır. Uzak ana bilgisayar bir IP adresi tarafından belirtilebilir. Bu komutun çıktısı **grep** veya **sed** tarafından bir oluk içinde filtre edilebilir.

ping

Bir yerel veya uzak ağ üzerindeki tüm diğer makinelere bir "ICMP ECHO_REQUEST" paketi yayımlar. Bu, ağ bağlantılarnıı test etmek için bir tanı aracıdır, ve dikkatli bir şekilde kullanılmalıdır.

Başarılı bir **ping**, 0 çıkış durumu döner. Bu bir komut dosyası içinde test edilebilir.

```
bash$ ping localhost

PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of
data.

 Warning: time of day goes back, taking countermeasures.

 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255
time=709 usec

 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255
time=286 usec

 --- localhost.localdomain ping statistics ---

 2 packets transmitted, 2 packets received, 0% packet loss

 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois

DNS (Alan Adı Sistemi) araması yapar. `-h` seçeneği sorgulanacak *whois* sunucusunu belirtmeye izin verir. Bkz. Örnek 5.6.

finger

Bir ağ üzerinde belirli bir kullanıcı hakkında bilgiye erişir. İsteğe bağlı olarak, bu komut kullanıcının `~/.plan, ~/.project,` ve `~/.forward` dosyalarını da, varsa, görüntüleyebilir.

```
bash$ finger bozo

Login: bozo                                Name: Bozo Bozeman

Directory: /home/bozo                       Shell: /bin/bash

On since Fri Aug 31 20:13 (MST) on tty1 1 hour 38 minutes idle

On since Fri Aug 31 20:13 (MST) on pts/0 12 seconds idle

On since Fri Aug 31 20:13 (MST) on pts/1

On since Fri Aug 31 20:31 (MST) on pts/2 1 hour 16 minutes idle

No mail.

No Plan.
```

Güvenlik düşünceleri nedeniyle, birçok ağ **finger** ve ilgili geri plan yordamını devre dışı bırakmıştır. [1]

Uzaktan Ana Makine Erişimi

sx, rx

sx ve **rx** komut seti *xmodem* protokolünü kullanarak, bir uzak ana bilgisayara veya bir uzak ana bilgisayardan dosya aktarmak için hizmet vermektedir. Bunlar genellikle **minicom** gibi bir iletişim paketinin parçasıdır.

sz, rz

sx ve **rx** komut seti *zmodem* protokolünü kullanarak, bir uzak ana bilgisayara veya bir uzak ana bilgisayardan dosya aktarmak için hizmet vermektedir. *zmodem*'in *xmodem* üzerinde bazı avantajları vardır, daha fazla iletim hızı ve kesintiye uğrayan dosya transferlerinin yeniden başlaması gibi. **sx** ve **rx** gibi, genellikle bir iletişim paketinin parçasıdır.

ftp

Uzak bir ana bilgisayardan dosya indirme/karşıya yükleme yapmak için yardımcı program ve protokoldür. Bir ftp oturumu komut dosyası içinde otomatik hale getirilebilir (Bkz. Örnek 17.7, Örnek A.5, ve Örnek A.13).

cu

Uzak bir sisteme bağlantı kurar (***C***all ***U***p) ve basit bir terminal olarak bağlanır. Bu telnet'in bir çeşit versiyonudur.

uucp

UNIX'ten *UNIX*'e *kopyalar*. Bu UNIX sunucuları arasında dosya aktarmak için bir iletişim paketidir. Bir kabuk betiği, **uucp** komut dizisini işlemek için etkili bir yoldur.

İnternet ve e-posta'nın ortaya çıkışından bu yana, **uucp** unutularak yok olma noktasına gelmiştir, ama İnternet bağlantısının uygun olmadığı durumlarda hala mükemmel olarak uygulanabilir kalabilmiştir.

telnet

Bir uzak ana bilgisayara bağlanmak için yardımcı program ve protokoldür. Telnet protokolü güvenlik delikleri içerir ve bu nedenle muhtemelen kaçınılmalıdır.

rlogin

`Uzaktan oturum açma`, uzak ana bilgisayarda bir oturum başlatır. Bu komutun güvenlik sorunları vardır, bu yüzden yerine **ssh** kullanınız.

rsh

`Uzak kabuk`, bir uzak ana bilgisayar üzerinde komut(lar) yürütür. Güvenlik sorunları vardır, bu yüzden yerine **ssh** kullanınız.

rcp

`Uzaktan kopyalama`, iki farklı ağ üzerindeki makineler arasında dosyaları kopyalar. **rcp** ve güvenlik etkileri olan benzeri araçları bir kabuk betiği içinde kullanmak tavsiye edilmez. Bunun yerine, **ssh** ya da **expect** betiğini kullanmayı düşününüz.

ssh

`Güvenli kabuk`, uzak bir ana bilgisayarda oturum açar ve orada komutları yürütür. **telnet**, **rlogin, rcp** ve **rsh** yerine kullanılabilen bu güvenli program, kimlik doğrulama ve şifreleme kullanır. Ayrıntılar için *man sayfasına* bakınız.

Yerel Ağ

write

Terminalden-terminale iletişim için bir araçtır. Başka bir kullanıcıya terminalinizin (konsol veya xterm) satırlarını göndermenizi sağlar. Bir terminale yazma erişimini ortadan kaldırmak için **mesg** komutu kullanılır.

write etkileşimli olduğundan, komut dosyası içinde normalde kullanım alanı bulmaz.

Posta

mail

Bir kullanıcıya e-posta iletisi gönderir.

Bu komut satırı posta istemcisi, bir komut dosyası içine gömülü komut olarak çalışıyor.

ÖRNEK 12.31 KENDİSİNİ POSTALAYAN BİR KOMUT DOSYASI

```
#!/bin/sh
# self-mailer.sh: Self-mailing script
ARGCOUNT=1              # Need name of addressee.
E_WRONGARGS=65
if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` addressee"
  exit $E_WRONGARGS
```

```
fi

# ============================================================================
cat $0 | mail -s "Script \"`basename $0`\" has mailed itself to you." "$1"
# ============================================================================
# ----------------------------------------
#  Greetings from the self-mailing script.
#  A mischievous person has run this script,
#+ which has caused it to mail itself to you.
#  Apparently, some people have nothing beter
#+ to do with their time.
# ----------------------------------------
exit 0
```

vacation

Bu yardımcı program otomatik olarak e-postalara cevap olarak, alıcının tatilde olduğunu ve geçici olarak müsait olmadığını yanıt olarak gönderir. Bu, **sendmail** ile bağlantılı olarak, bir ağ üzerinde çalışır ve çevirmeli POPmail hesapları için geçerli değildir.

Notlar

[1] Bir *geri plan yordamı* terminal oturumuna bağlı olmayan bir arka plan işlemidir. Geri plan yordamları belirtilen zamanlarda, belirli olaylar tarafından açıkça tetiklenen belirli hizmetleri yerine getirirler.

"Geri plan yordamı"nın Yunanca karşılığı cin, hızır gibi sözcüklerdir, ve UNIX geri plan yordamları kendilerine atanan görevleri yürütürken neredeyse doğaüstü ve gizemli bir yoldan sessizce perde arkasında dolaşırlar, anlamına gelir.

12.7 TERMİNAL KONTROL KOMUTLARI

Konsol veya terminali etkileyen komutlar

tput

Terminal başlatmak ve / veya bu konuda `terminfo` verisinden bilgi alıp getirir. Çeşitli seçenekleriyle belirli bazı terminal işlemlerine izin verir. **tput clear** aşağıda anlatılan **clear** ile eşdeğerdir. **tput reset** aşağıda anlatılan **reset** ile eşdeğerdir.

```
bash$ tput longname

xterm terminal emulator (XFree86 4.0 Window System)
```

Unutmayınız ki, **stty** bir terminal kontrolü için belirlenen daha güçlü bir komut setini programcıya sunmaktadır.

reset

Terminal parametrelerini sıfırlar ve metin ekranını temizler. **clear** komutunda olduğu gibi, imleç ve komut istemi, terminalin sol üst köşesinde yeniden belirir.

clear

clear komutu açıkça konsol veya xterm'deki metin ekranını temizler. İmleç ve komut istemi, ekranın veya xterm penceresinin sol üst köşesinde yeniden belirir. Bu komut, komut satırından çağrılabilir veya bir komut dosyasında kullanılabilir. Bkz. Örnek 10.24.

script

Bu yardımcı program bir konsol veya xterm penceresinde kullanıcının komut satırından gerçekleştirdiği tüm tuş vuruşlarının kaydını tutar (bir dosyaya kaydeder). Bu, aslında, bir oturumun kaydını oluşturmaktır.

12.8 MATEMATİK KOMUTLARI

"Sayıları gerçekleştirmek"

factor

Bir tamsayıyı asal faktörlerine ayrıştırır.

```
bash$ factor 27417

27417: 3 13 19 37
```

bc, dc

Bunlar esneklik ve isteğe bağlı duyarlılık gerektiren hesaplama araçlarıdır.

bc'nin belli belirsiz C'ye benzeyen sözdizimi vardır.

dc, yığıta (stack) dayalıdır ve RPN ("Reverse Polish Notation") kullanır.

İkisini karşılaştırdığımızda, komut dosyalarında kullanmak için **bc** daha faydalı görünüyor. Bu oldukça iyi bir UNIX yardımcı programdır, ve bu nedenle bir olukta kullanılabilir.

Bash kayan noktalı hesaplamaları işleyemez, ve bazı önemli matematiksel fonksiyonlar için gerekli olan operatörlerden yoksundur. Neyse ki, **bc** kurtarmaya ve yardıma gelir.

Bir komut dosyası değişkenini hesaplamak için **bc** kullanılan basit bir şablonu aşağıda görüyoruz. Burada komut yer değiştirmesi de kullanılmıştır.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

ÖRNEK 12.32 BİR İPOTEK ÜZERİNDE YAPILAN AYLIK ÖDEMELER

```
#!/bin/bash
# monthlypmt.sh: Calculates monthly payment on a mortgage.
# This is a modification of code in the "mcalc" (mortgage calculator)
#+ package,
# by Jeff Schmidt and Mendel Cooper (yours truly, the author of this
#+ document).
# http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz [15k]
echo
echo "Given the principal, interest rate, and term of a mortgage,"
echo "calculate the monthly payment."
bottom=1.0
echo
echo -n "Enter principal (no commas) "
read principal
echo -n "Enter interest rate (percent) " # If 12%, enter "12", not ".12".
read interest_r
echo -n "Enter term (months) "
read term
interest_r=$(echo "scale=9; $interest_r/100.0" | bc)  # Convert to decimal.
```

```
# "scale" determines how many decimal places.
interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)
top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)
echo; echo "Please be patient. This may take a while."
let "months = $term - 1"
# ============================================================================
for ((x=$months; x > 0; x--))
do
  bot=$(echo "scale=9; $interest_rate^$x" | bc)
  bottom=$(echo "scale=9; $bottom+$bot" | bc)
# bottom = $(($bottom + $bot"))
done
# ----------------------------------------------------------------------------
#  Rick Boivie pointed out a more efficient implementation
#+ of the above loop, which decreases computation time by 2/3.
# for ((x=1; x <= $months; x++))
# do
#   bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
# done

#  And then he came up with an even more efficient alternative,
#+ one that cuts down the run time by about 95%!
# bottom=`{
#     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
#     for ((x=1; x <= $months; x++))
#     do
#         echo 'bottom = bottom * interest_rate + 1'
#     done
#     echo 'bottom'
#     } | bc`      # Embeds a 'for loop' within command substitution.
```

```
# ============================================================================

# let "payment = $top/$bottom"

payment=$(echo "scale=2; $top/$bottom" | bc)

# Use two decimal places for dollars and cents.

echo

echo "monthly payment = \$$payment"# Echo a dollar sign in front of amount.

echo


exit 0

# Exercises:

# 1) Filter input to permit commas in principal amount.

# 2) Filter input to permit interest to be entered as percent or decimal.

# 3) If you are really ambitious,

#    expand this script to print complete amortization tables.
```

ÖRNEK 12.33 TABAN ÇEVRİMİ

```
:/bin/bash
################################################################################
# Shellscript : base.sh - print number to different bases (Bourne Shell)
# Author : Heiner Steven (heiner.steven@odn.de)
# Date : 07-03-95
# Category : Desktop
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
################################################################################
# Description
#
# Changes
# 21-03-95 stv fixed error occuring with 0xb as input (0.2)
################################################################################
# ==> Used in this document with the script author's permission.
# ==> Comments added by document author.
NOARGS=65
PN=`basename "$0"`                              # Program name
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`   # ==> VER=1.2
Usage () {
    echo "$PN - print number to different bases, $VER (stv '95)
usage: $PN [number ...]
If no number is given, the numbers are read from standard input.
```

```
A number may be
    binary (base 2) starting with 0b (i.e. 0b1100)
    octal (base 8) starting with 0 (i.e. 014)
    hexadecimal (base 16) starting with 0x (i.e. 0xc)
    decimal otherwise (i.e. 12)" >&2
    exit $NOARGS
} # ==> Function to print usage message.

Msg () {
    for i # ==> in [list] missing.
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # Determine base of the number
    for i      # ==> in [list] missing...
    do         # ==> so operates on command line arg(s).
        case "$i" in
            0b*)                ibase=2;;     # binary
            0x*|[a-f]*|[A-F]*) ibase=16;;    # hexadecimal
            0*)                 ibase=8;;     # octal
            [1-9]*)             ibase=10;;    # decimal
            *)
                Msg "illegal number $i - ignored"
                continue;;
        esac


        # Remove prefix, convert hex digits to uppercase (bc needs this)

        number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`

        # ==> Uses ":" as sed separator, rather than "/".

        # Convert number to decimal

        dec=`echo "ibase=$ibase; $number" | bc`   # ==> 'bc' is calculator
utility.

        case "$dec" in

            [0-9]*)     ;;                          # number ok

            *)          continue;;                  # error: ignore

        esac

        # Print all conversions in one line.

        # ==> 'here document' feeds command list to 'bc'.

        echo `bc <<!

            obase=16; "hex="; $dec

            obase=10; "dec="; $dec

            obase=8; "oct="; $dec
```

```
            obase=2; "bin="; $dec
!
    ` | sed -e 's: : :g'
    Done
}

while [ $# -gt 0 ]
do
    case "$1" in
        --) shift; break;;
        -h) Usage;;   # ==> Help message.
        -*) Usage;;
        *) break;;    # first number
    esac              # ==> More error checking for illegal input would be
useful.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else                  # read from stdin
    while read line
    do
        PrintBases $line
    done
fi
```

bc çağırmanın alternatif bir yöntemi, komut yer değiştirmesi bloğunun içine gömülü olarak here belgesinin kullanılmasını içermektedir. Bu özellikle bir komut dosyasının, **bc** komutuna bir dizi seçenek ve komut geçirmesi gerektiği zamanlarda uygun olur.

```
variable=`bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
‘

... or ...

variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

ÖRNEK 12.34 bc ÇAĞIRMANIN BİR BAŞKA YOLU

```
#!/bin/bash
# Invoking 'bc' using command substitution
# in combination with a 'here document'.
var1=`bc << EOF
18.33 * 19.78
EOF
‘
echo $var1          # 362.56
# $( ... ) notation also works.
v1=23.53
v2=17.881
```

```
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2          # 593487.8452

var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Returns the sine of 1.7 radians.
# The "-l" option calls the 'bc' math library.
echo $var3          # .991664810

# Now, try it in a function...
hyp=                # Declare global variable.
hypotenuse ()       # Calculate hypotenuse of a right triangle.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
```

```
# Unfortunately, can't return floating point values from a Bash function.
}
hypotenuse 3.68 7.31
echo "hypotenuse = $hyp"    # 8.184039344

exit 0
```

Çoğu insan **dc** çağırmaktan kaçınır, çünkü sezgiye-dayalı olmayan RPN girdisi gerektirmektedir.

Örnek 12-35. Onaltılık tabandan ondalık sayıya çevirme

```
#!/bin/bash
# hexconvert.sh: Convert a decimal number to hexadecimal.
BASE=16        # Hexadecimal.
if [ -z "$1" ]
then
  echo "Usage: $0 number"
  exit $E_NOARGS
# Need a command line argument.
fi
# Exercise: add argument validity checking.

hexcvt ()
{
if [ -z "$1" ]
then
  echo 0
  return        # "Return" 0 if no arg passed to function.
fi
```

```
echo ""$1" "$BASE" o p" | dc
# "o" sets radix (numerical base) of output.
# "p" prints the top of stack.
# See 'man dc' for other options.
return
}

hexcvt "$1"
exit 0
```

dc için *bilgi* sayfasının incelenmesi bu komutun incelikleri hakkında bazı fikirler verir. Ancak ustalıklarını bu güçlü ama gizli programla göstermekten zevk duyan küçük ve seçilmiş bir grup *dc* meraklısı varlığını sürdürmektedir.

ÖRNEK 12.36 BİR SAYIYI ÇARPANLARINA AYIRMA

```
#!/bin/bash
# factr.sh: Factor a number
MIN=2 # Will not work for number smaller than this.
E_NOARGS=65
E_TOOSMALL=66
if [ -z $1 ]
then
  echo "Usage: $0 number"
  exit $E_NOARGS
fi
if [ "$1" -lt "$MIN" ]
then
```

```
  echo "Number to factor must be $MIN or greater."
  exit $E_TOOSMALL
fi

# Exercise: Add type checking (to reject non-integer arg).
echo "Factors of $1:"
# -------------------------------------------------------------------------
echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2" | dc
# -------------------------------------------------------------------------
# Above line of code written by Michel Charpentier <charpov@cs.unh.edu>.
# Used with permission (thanks).
exit 0
```

awk

Bir komut dosyasında kayan noktalı matematik yapmanın bir başka yolu da, **kabuk sarıcısı** (shell wrapper) içinde **awk**'un yerleşik matematik fonksiyonlarını kullanmaktır.

ÖRNEK 12.37 BİR ÜÇGENİN HİPOTENÜSÜNÜN HESAPLANMASI

```
#!/bin/bash
# hypotenuse.sh: Returns the "hypotenuse" of a right triangle.
# ( square root of sum of squares of the "legs")
ARGS=2                # Script needs sides of triangle passed.
E_BADARGS=65          # Wrong number of arguments.
if [ $# -ne "$ARGS" ] # Test number of arguments to script.
then
  echo "Usage: `basename $0` side_1 side_2"
  exit $E_BADARGS
fi
```

```
AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
# command(s) / parameters passed to awk
echo -n "Hypotenuse of $1 and $2 = "
echo $1 $2 | awk "$AWKSCRIPT"
exit 0
```

12.9 DİĞER BAZI KONULAR

Gözle görülür bir kategoriye alınmayan komutlardır.

jot, seq

Bu komutlar kullanıcı tarafından belirlenen argüman olarak verilen tamsayıları sıralar.

Her tamsayı arasında yer alan ayırıcı normal olarak bir yeni satır karakteridir, ama bu `-s` seçeneği ile değişik karakterler için denenebilir.

```
bash$ seq 5
1
 2
 3
 4
 5

bash$ seq -s : 5
1:2:3:4:5
```

Döngüler için düşünüldüğünde **jot** ve **seq** kullanışlı araçlardır.

ÖRNEK 12.38 DÖNGÜ ARGÜMANLARININ seq KULLANILMASIYLA OLUŞTURULMASI

```
#!/bin/bash

 for a in `seq 80` # or for a in $( seq 80 )

# Same as for a in 1 2 3 4 5 ... 80 (saves much typing!).

# May also use 'jot' (if present on system).

do

  echo -n "$a "

done

# Example of using the output of a command to generate

# the [list] in a "for" loop.

echo; echo

COUNT=80 # Yes, 'seq' may also take a replaceable parameter.


for a in `seq $COUNT` # or for a in $( seq $COUNT )

do

  echo -n "$a "

  done

echo

exit 0
```

getopt

getopt komut satırı seçeneklerini ayrıştırır. Bunun için öncesinde tire (-) koymak gerekir. Bu dış komut, getopts Bash yerleşiğine karşılık gelse de, onun kadar esnek değildir.

ÖRNEK 12.39 KOMUT SATIRI SEÇENEKLERİNİN getopt KULLANILMASIYLA AYRIŞTIRILMASI

```
#!/bin/bash

# Try the following when invoking this script.

# sh ex33a –a

# sh ex33a –abc

# sh ex33a -a -b –c

# sh ex33a –d

# sh ex33a –dXYZ

# sh ex33a -d XYZ

# sh ex33a –abcd

# sh ex33a –abcdZ

# sh ex33a –z

# sh ex33a a

# Explain the results of each of the above.

E_OPTERR=65

if [ "$#" -eq 0 ]

then # Script needs at least one command-line argument.

  echo "Usage $0 -[options a,b,c]"

  exit $E_OPTERR

fi

set -- `getopt "abcd:" "$@"`

# Sets positional parameters to command-line arguments.

# What happens if you use "$*" instead of "$@"?

while [ ! -z "$1" ]

do

  case "$1" in

     -a) echo "Option \"a\"";;

     -b) echo "Option \"b\"";;
```

```
    -c) echo "Option \"c\"";;
    -d) echo "Option \"d\" $2";;
     *) break;;
  esac
  shift
done

# It is better to use the 'getopts' builtin in a script,
#+ rather than 'getopt'.
# See "ex33.sh".
exit 0
```

run-parts

run-parts komutu [1] bir hedef dizinindeki tüm komutları teker teker çalıştırır, bir dizi ASCII-sıralı dosya adlarını alır. Tabii ki, betik dosyalarına verilen izinler arasında çalışma bulunmalıdır.

Bir takım geri plan yordamları tarafından (örneğin, crond) komut dosyalarını çalıştırmak için run_parts çağrılır. Örneğin crond geri plan yordamı `/etc/cron.*` dizinlerindeki betikleri çalıştırmaktadır.

yes

yes komutu varsayılan `stdout`'a `y` karakteri, takiben satır sonu besler. **Kontrol-C** ile bu komut çalışmasını durdurur. **`yes farklı dize`** gibi farklı çıktı dizelerine olanak tanır. Nedeni , komut satırından veya bir komut dosyasından **yes** çıktısı kullanıcı girdisi istenen bir programa yönlendirilebilmesi veya oluk edilebilmesidir.

Sonuç olarak, en yakın benzeri **expect** komutudur.

`yes | fsck /dev/hda1` **`fsck`** etkileşimli olmayan bir şekilde çalışır. (dikkat gerektirir!)

`yes | rm -r dirname` etkisi **`rm -rf dirname`** ile aynıdır. (dikkat gerektirir!)

fsck veya fdisk gibi tehlikeli sistem komutlarına oluk yapmak dikkat gerektirir!

banner

Argümanların görüntüsü, ASCII karakter (varsayılan '#') kullanarak büyük bir dikey afişmiş gibi `stdout` formatında sunulur. Yazıcıya da yönlendirilmesi istenebilir.

printenv

Belirli bir kullanıcı için belirtilen tüm çevre değişkenlerini gösterir.

```
bash$ printenv | grep HOME

HOME=/home/bozo
```

lp

lp ve **lpr** komutları bir yazdırma kuyruğuna gelen tüm dosya (lar) için yazıcıya aktarma işlemlerini gerçekleştirir. [2]

Bu komutlar, adlarını farklı bir teknolojiden almaktadır: nokta-vuruşlu yazıcılar.

`bash$` **`lp file1.txt`** **`veya`** `bash$` **`lp <file1.txt`**

pr komutunun biçimlendirilmiş çıktısı **lp** için oluklandığı zaman fayda sağlayabilir.

```
bash$ pr -options file1.txt | lp
```

groff ve *Ghostscript* gibi biçimlendirme paketleri, **lp** çıktısını direkt ele alır .

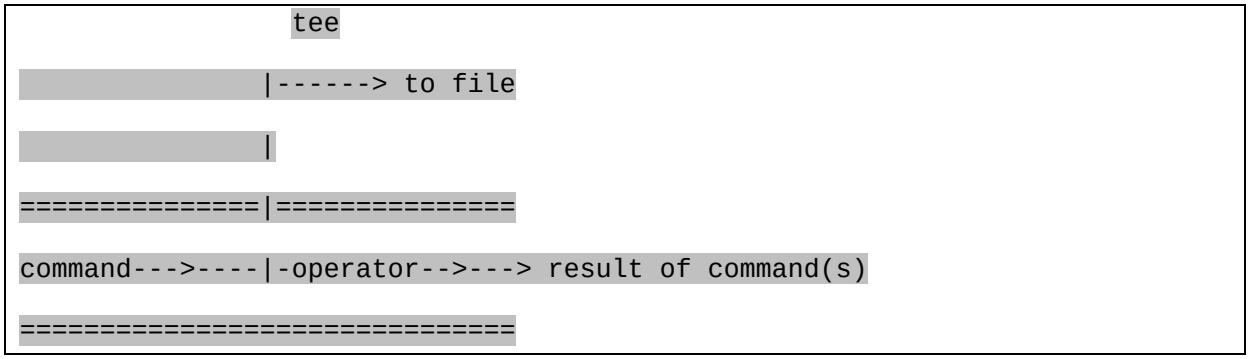
```
bash$ groff -Tascii file.tr | lp
```

```
bash$ gs -options | lp file.ps
```

İlgili komutlar yazdırma kuyruğunu görüntülemek için **lpq**, ve yazdırma kuyruğundan işleri silmek için **lprm** olarak karşımıza çıkmaktadır.

tee

Bu bir yönlendirme operatörüne benzer. Benzer yönü oluk çıktısının içeriği değişmez. Bu bir dosya ya da kağıt için devam eden bir sürece yazdırmak için yararlıdır. Amacı da test ve sistem hatalarının giderilmesidir.




```
cat listfile* | sort | tee check.file | uniq > result.file
```

Yinelenen satırlar **uniq** tarafından kaldırılmadan önce, sıralı "liste dosyalar" halinde `check.file` dosyasında birleştirilir.

mkfifo

Bu komut eski olmasına rağmen, *adlandırılmış bir oluk* yaratmak için kullanılır. Süreçler arasında veri aktarımı sağlar. Tipik olarak, bir süreç *ilk giren-ilk çıkar* kuyruğuna yazarken diğeri oradan okur. Bkz.Örnek A.15.

pathchk

Bu komut bir dosya adının geçerliliğini kontrol eder. Maksimum (255 karakter) uzunluğu aşan dosya adları için, ya da taranamayan dizinler için hata verir, ancak ne yazık ki, hata kodunun komut dosyalarında test amaçlı kullanımına rastlanmaz.

dd

Bu ortak paydası güç bir "veri teksir" komutudur. Halen kullanım alanı olan bu yardımcı program, orijinal olarak UNIX minibilgisayarları ve IBM anabilgisayarları arasındaki manyetik teypler üzerinde veri değişimi sağlamaktadır. **dd** komutu, `(stdin/ stdout)` kopyalama yapar, dosyalarla da çalışma yeteneği sağlanabilir. Olası dönüşümleri, ASCII / EBCDIC [4] olan büyük/küçük harf, girdi ve çıktı arasındaki bayt çiftlerinin takası ve giriş dosyasının başının ve sonunun yok sayılması veya kırpılması gibi çoğaltılabilir.

`dd --help` dönüşüm listeleri ve bu güçlü yardımcı programın aldığı diğer seçenekleri listeler.

```
# Exercising 'dd'.
n=3
p=5
input_file=project.txt
output_file=log.txt
dd if=$input_file of=$output_file bs=1 skip=$((n-1))
# Extracts characters n to p from file $input_file.
```

```
echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null

# Echoes "hello world" vertically.

# Thanks, S.C.
```

dd'nin ne kadar çok yönlü olduğunu göstermek için tuş vuruşlarının anlaşılmasına bakalım.

ÖRNEK 12.40 TUŞ VURUŞLARININ ANLAŞILMASI

```
#!/bin/bash

# Capture keystrokes without needing to press ENTER.

keypresses=4                 # Number of keypresses to capture.

old_tty_setting=$(stty -g) # Save old terminal settings.

echo "Press $keypresses keys."

stty -icanon -echo           # Disable canonical mode.

                             # Disable local echo.

keys=$(dd bs=1 count=$keypresses 2> /dev/null)

# 'dd' uses stdin, if "if" not specified.

stty "$old_tty_setting"    # Restore old terminal settings.

echo "You pressed the \"$keys\" keys."


# Thanks, S.C. for showing the way.

exit 0
```

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc

# The "conv=notrunc" option means that the output file will not be
truncated.

# Thanks, S.C.
```

dd komutu bir veri akışı üzerinde rasgele erişim için de yararlı bir komuttur.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc

# The "conv=notrunc" option means that the output file will not be
truncated.

# Thanks, S.C.
```

dd komutu aygıtlar arasında ham veri ve disk görüntüleri kopyalayabilir. Bu aygıtlar flopi disket ve teyp sürücüleri olabilir (bkz. Örnek A.6). Yaygın olarak önyükleme disketi yapar.

```
dd if=kernel-image of=/dev/fd0H1440
```

dd kullanarak bir flopi disketin tüm içeriğini kopyalamak mümkündür.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

dd diğer uygulamaları başlatırken geçici takas (bkz.Örnek 29.2) ve rasgele erişimli bellek dosyaları (bkz.Örnek 29.3) içerir. Hatta sabit diskin kopyalanması da alt düzeyde mümkündür, fakat genel anlamda önerilmez.

dd komutunu kullanımı daha ilginç örneklerle çeşitlendirebiliriz.

ÖRNEK 12.41 BİR DOSYANIN GÜVENLİ SİLİNMESİ

```
#!/bin/bash

# blotout.sh: Erase all traces of a file.

# This script overwrites a target file alternately

#+ with random bytes, then zeros before finally deleting it.

# After that, even examining the raw disk sectors

#+ will not reveal the original file data.
```

```
PASSES=7          # Number of file-shredding passes.
BLOCKSIZE=1       # I/O with /dev/urandom requires unit block size,
                  #+ otherwise you get weird results.
E_BADARGS=70
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ] # No filename specified.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi
file=$1
if [ ! -e "$file" ]
then
  echo "File \"$file\" not found."
  exit $E_NOT_FOUND
fi
echo;
echo -n "Are you absolutely sure you want to blot out \"$file\" (y/n)? "
read answer
case "$answer" in
[nN]) echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*) echo "Blotting out file \"$file\".";;
esac

flength=$(ls -l "$file" | awk '{print $5}') # Field 5 is file length.
pass_count=1
```

```
echo
while [ "$pass_count" -le "$PASSES" ]
do
  echo "Pass #$pass_count"
  sync          # Flush buffers.
  dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
                # Fill with random bytes.
  sync          # Flush buffers again.
  dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
                # Fill with zeros.
  sync          # Flush buffers yet again.
  let "pass_count += 1"
  echo
done
rm -f $file     # Finally, delete scrambled and shredded file.
sync            # Flush buffers a final time.
# This is a fairly secure, if inefficient and slow method
#+ of thoroughly "shredding" a file. The "shred" command,
#+ part of the GNU "fileutils" package, does the same thing,
#+ but more efficiently.
# The file cannot not be "undeleted" or retrieved by normal methods.
# However...
#+ this simple method will likely *not* withstand forensic analysis.


# Tom Vier's "wipe" file-deletion package does a much more thorough job
#+ of file shredding than this simple script.
#    http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2


# For an in-depth analysis on the topic of file deletion and security,
#+ see Peter Gutmann's paper,
```

```
#+ "Secure Deletion of Data From Magnetic and Solid-State Memory".
#    http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html


exit 0
```

od

od komutu ya da *sekizli taban filtre*si, girdiyi (dosyaları) sekizlik tabana veya diğer başka tabanlara dönüştürür. Bu, ikili veri dosyaları ya da okunması mümkün olmayan sistem aygıt dosyalarının görüntülenmesi veya işlenmesi için yararlıdır, bu tarz örnekler arasında `/dev/urandom`, ve ikili veri için gerçekleştirilmiş bir filtrenin kendisi sayılabilir (bkz. Örnek 9.23 ve Örnek 12.10).

hexdump

İkili bir dosyanın onaltılık, sekizlik, onluk ASCII dökümünü gerçekleştirir. Bu komut yukarıda anlatılan, **od** komutu ile yakın anlamda eşdeğer sayılır, ancak kullanışlılığı daha azdır.

mcookie

Bu komut, bir "sihirli çerez" üretir, 128-bit (32 karakter) sözde rasgele onaltılık sayı, normal olarak, X sunucusu tarafından bir yetkilendirme "imza"sı olarak kullanılır. Bir komut dosyasındaki kullanımı "atıl" bir rasgele sayıyla sonuçlanır.

```
random000=`mcookie | sed -e '2p'`
# Uses 'sed' to strip off extraneous characters.
```

Tabii ki, bir komut dosyası aynı amaçla md5 kullanabilir.

```
# Generate md5 checksum on the script itself.
random001=`md5sum $0 | awk '{print $1}'`
# Uses 'awk' to strip off the filename.
```

m4

m4 sanal bir dil gibi güçlü bir makro işleme filtresidir [5]. Aslında *RatFor* için bir ön-işlemci olarak yazılmış olmasına rağmen, m4 tek başına bir yardımcı araç olarak yararlı oldu. Aslında, **m4**, eval, tr, ve awk işlevselliklerinin bazılarını birleştirmektedir. Çok sayıda makro genişleme olanakları da vardır.

Linux Dergisi Nisan 2002 sayısında m4 kullanım alanlarına inceleme yapan bir makale yayımlamıştır.

ÖRNEK 12.42 m4 KULLANIMI

```
#!/bin/bash
# m4.sh: Using the m4 macro processor
# Strings
string=abcdA01
echo "len($string)" | m4                                # 7
echo "substr($string,4)" | m4                           # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4              # 01Z
# Arithmetic
echo "incr(22)" | m4                                    # 23
echo "eval(99 / 3)" | m4                                # 33
exit 0
```

Notlar

[1] Debian Linux dağıtımına uyarlanan bir betiktir.

[2] *Yazdırma kuyruğu* yazdırılacak "sırada bekleyen" iş grubudur.

[3] Linux Journal dergisinde Andy Vaught tarafından yazılan "Introduction to Named Pipes" adlı makalenin basım tarihi Eylül 1997'dir.

[4] EBCDIC kısaltması genişletilmiş ikili kodlu ondalık değişim kodu (**E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**ode) ifade eden bir veri biçimidir ve IBM tarafından geliştirilmiştir. Örneğin, metin dosyalarını kodlayıcı olarak conv=ebcdic uygulanması "atıl"dır, ve dd'nin güvenli olmasını engeller.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted

# Encode (looks like gibberish).

# Might as well switch bytes (swab), too, for a little extra obscurity.


cat $file_encrypted | dd conv=swab,ascii > $file_plaintext

# Decode.
```

[5] Bir *makro*, tanım gereği parametreler üzerinde hesap yapan işlem topluluğunun sembolik bir sabitle genişletilerek açılmış halidir.

Bölüm 13 SİSTEM ve YÖNETİM KOMUTLARI

`/etc/rc.d` dosyasındaki başlatma ve kapatma komutları, sistem ve yönetim komutlarının çoğunun kullanım (ve kullanışlılığını) göstermektedir. Bunlar genellikle kök tarafından çağrılır ve sistem bakım veya dosya sisteminin acil tamiri için kullanılır. Dikkatli kullanın, bu komutların bazıları yanlış kullanıldığında sisteminize zarar verebilir.

Kullanıcılar ve Gruplar

chown, chgrp

chown komutu, bir dosya veya dosyaların sahipliğini değiştirir. Bu komut *kök*'ün dosya sahipliğini bir kullanıcıdan başka bir kullanıcıya kaydırmak için kullanabileceği yararlı bir yöntemdir. Sıradan bir kullanıcı, dosyaların sahipliğini değiştiremez, kendi dosyalarınınkini bile. [1]

```
root# chown bozo *.txt
```

chgrp komutu bir veya birden fazla dosyanın *grup* sahipliğini değiştirir. Bu işlemi kullanmak için hem dosya(lar)ın sahibi, hem de hedef grubun bir üyesi (veya *kök*) olmalısınız.

```
chgrp --recursive dunderheads *.data

#  The "dunderheads" group will now own all the "*.data" files

#+ all the way down the $PWD directory tree (that's what "recursive"

#+ means).
```

useradd, userdel

useradd yönetim komutu sisteme bir kullanıcı hesabı ekler ve öyle belirtilmişse, o kullanıcı için bir ana dizin oluşturur. Karşılık gelen **userdel** komutu, sistemden bir kullanıcı hesabını ortadan kaldırır [2] ve ilişkili dosyaları siler.

adduser komutu **useradd** ile eş anlamlıdır ve genellikle sembolik bir bağlantıdır.

id

id komutu gerçek ve etkin kullanıcı kimliklerini ve geçerli kullanıcının grup kimliklerini listeler. Bu $UID, $EUID ve $GROUPS iç Bash değişkenlerinin tamamlayıcısıdır.

```
bash$ id

uid=501(bozo) gid=501(bozo)
groups=501(bozo),22(cdrom),80(cdwriter),81(audio)


bash$ echo $UID

501
```

Aynı zamanda, bkz.Örnek 9.5.

who

Sistemde oturum açan tüm kullanıcıları gösterir.

```
bash$ who

bozo tty1 Apr 27 17:45

 bozo pts/0 Apr 27 17:46

 bozo pts/1 Apr 27 17:47

 bozo pts/2 Apr 27 17:49
```

`-m` yalnızca geçerli kullanıcı hakkında ayrıntılı bilgi verir. **who** komutuna herhangi iki parametre gönderilmesi **who -m** ile eşdeğerdir, **who am i** veya **who The Man** örneklerinde olduğu gibi.

```
bash$ who –m

localhost.localdomain!bozo pts/2 Apr 27 17:49
```

whoami, who -m ile benzeşir, ancak sadece kullanıcı adını listeler.

```
bash$ whoami

bozo
```

w

Oturum açmış olan tüm kullanıcıları ve kendilerine ait süreçleri gösterir. Bu, **who** komutunun genişletilmiş versiyonudur. **w** çıktısı belirli bir kullanıcı ve/veya süreci bulmak için **grep** oluğuyla taşınabilir.

```
bash$ w | grep startx

bozo tty1       -               4:22pm    6:41    4.47s    0.45s    startx
```

logname

Geçerli kullanıcının oturum açma adını (`/var/run/utmp` dosyasında bulunduğu üzere) gösterir. Söz konusu işlem, yukarıdaki whoami ile yakın eşdeğerdir.

```
bash$ logname

bozo


bash$ whoami

bozo
```

Ancak...

```
bash$ su

Password: ......

bash# whoami

root

bash# logname

bozo
```

su

Bir program veya komut dosyasını yedek kullanıcı olarak çalıştırır. **su rjones** kullanıcı *rjones* olarak bir kabuk başlatır. Tek başına **su**, *kök* varsayılanına denk gelir. Bkz. Örnek A.15.

sudo

Kök (veya başka bir kullanıcı) olarak bir komutu çalıştırır. Bu bir betikte kullanılabilir, böylece normal bir kullanıcıya, komut dosyasını çalıştırması için izin verir.

```
#!/bin/bash

# Some commands.

sudo cp /root/secretfile /home/bozo/secret

# Some more commands.
```

`/etc/sudoers`, **sudo** çağırma izni olan kullanıcıların adlarını tutar.

users

Oturum açan tüm kullanıcıları gösterir. Bu **who -q** ile yaklaşık olarak eşdeğerdir.

ac

Kullanıcıların oturum açtığı zamanı, `/var/log/wtmp` dosyasından okunduğu üzere gösterir. Bu GNU hesap tutma araçlardan biridir.

```
bash$ ac

        total 68.08
```

last

Son olarak oturum açan kullanıcıları, `/var/log/wtmp` dosyasından okunduğu üzere listeler. Bu komut ayrıca uzaktan girişleri de gösterebilir.

groups

Geçerli kullanıcı ve ona ait grupları listeler. Bu $GROUPS değişkenine karşılık gelir, ancak, sayılar yerine grup adlarını verir.

```
bash$ groups

bozita cdrom cdwriter audio xgrp


bash$ echo $GROUPS

501
```

newgrp

Oturumu kapatmadan kullanıcının grup kimliğini değiştirir. Bu yeni grubun dosyalarına erişime izin verir. Kullanıcılar, aynı anda birden fazla gruba üye olabildikleri için, bu komutun kullanım alanı azdır.

Terminaller

tty

Geçerli kullanıcının terminalinin adını ekrana yazdırır. Unutmayınız ki, her bir ayrı xterm penceresi farklı bir terminal olarak sayılır.

```
bash$ tty
/dev/pts/1
```

stty

Terminal ayarlarını gösterir ve/veya değiştirir. Bu karmaşık komut, bir komut dosyasında kullanıldığında, terminal davranışlarını ve çıktı görüntüleme yolunu kontrol edebilir.

ÖRNEK 13.1 SİLME KARAKTERİNİ AYARLAMA

```
#!/bin/bash
# erase.sh: Using "stty" to set an erase character when reading input.
echo -n "What is your name? "
read name                       # Try to erase characters of input.
                                # Won't work.
```

```
echo "Your name is $name."
stty erase '#'                # Set "hashmark" (#) as erase character.
echo -n "What is your name? "
read name                     # Use # to erase last character typed.
echo "Your name is $name."
exit 0
```

ÖRNEK 13.2 GİZLİ ŞİFRE: TERMİNALİ KAPATMA

```
#!/bin/bash
echo
echo -n "Enter password "
read passwd
echo "password is $passwd"
echo -n "If someone had been looking over your shoulder, "
echo "your password would have been compromised."
echo && echo  # Two line-feeds in an "and list".
stty -echo    # Turns off screen echo.
echo -n "Enter password again "
read passwd
echo
echo "password is $passwd"
echo
stty echo     # Restores screen echo.
exit 0
```

Yaratıcı bir **stty** kullanımı (**ENTER**'a basmadan) kullanıcının tuş vuruşunu bulgulamaktır.

ÖRNEK 13.3 TUŞ VURUŞU BULGULANMASI

```
# /bin/bash

# keypress.sh: Detect a user keypress ("hot keyboard").

echo

old_tty_settings=$(stty -g)         # Save old settings.

stty –icanon

Keypress=$(head -c1)                # or $(dd bs=1 count=1 2> /dev/null)

                                    # on non-GNU systems

echo

echo "Key pressed was \""$Keypress"\"."

echo

stty "$old_tty_settings"            # Restore old settings.

# Thanks, Stephane Chazelas.

exit 0
```

Aynı zamanda bkz.Örnek 9-3.

Terminaller ve modları *(Stephane Chazelas)*

Normalde, bir terminalin çalıştığı mod *kanonik*tir. Bir kullanıcı bir tuşa vurduğunda, ortaya çıkan karakter hemen aslında bu terminalde çalışan programa gitmez. Terminale yerel bir tampon, tuş vuruşlarını saklar. Kullanıcı **ENTER** tuşuna bastığında, bu çalışan programa tüm saklanan tuş vuruşlarını gönderir. Terminal içinde bir temel satır editörü de vardır.

```
bash$ stty –a

speed 9600 baud; rows 36; columns 96; line = 0;

 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 =

 <undef>;

 start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;

 ...

 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop –echoprt
```

Kanonik mod kullanarak, bir yerel terminal satır editörü için özel tuşları yeniden tanımlamak mümkündür.

```
bash $ cat> filexxx

wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>

<ctl-D>

bash$ cat filexxx

hello world

bash$ bash$ wc -c < file

13
```

Kullanıcı 26 tuşa vurduğu halde terminali kontrol eden süreç sadece 13 karakteri alır (12 alfabetik olan, artı 1 yeni satır). Kanonik olmayan ("ham") modda, basılan her tuş (örneğin **ctrl-H** gibi özel karakterler de dahil) kontrol sürecine hemen bir karakter gönderir. Bash istemi hem `icanon` ve hem de `echo` modlarını devre dışı bırakır, çünkü temel terminal satır editörünün yerine daha ayrıntılı programlanabilir. Örneğin, Bash isteminde ctrl-A basıldığında bir terminal tarafından görüntülenen hiçbir ^A yoktur, ama Bash bir \1 karakterini alır, yorumlar, ve imleci satırın başına hareket ettirir.

Stephane Chazelas

tset

Terminal ayarlarını gösterir veya başlatır. Bu **stty**'nin daha az yetenekli bir sürümüdür.

```
bash$ tset -r

Terminal type is xterm-xfree86.

 Kill is control-U (^U).

 Interrupt is control-C (^C).
```

setserial

Seri port parametrelerini ayarlar ya da görüntüler. Bu komutun kök tarafından çalıştırılması gerekir ve genellikle bir sistem kurulum komut dosyası bulunur.

```
# From /etc/pcmcia/serial script:

IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`

setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, agetty

Bir kullanıcı tarafından giriş oturumu açmak için kurulduğunda bir terminal için başlatma işlemi tarafından **getty** veya **agetty** kullanılır. Bu komutlar kullanıcı kabuk betikleri içinde kullanılmaz. **stty** bu komutlara komut dosyasında karşılık gelir.

mesg

Geçerli kullanıcının terminaline yazma erişimi etkinleştirir veya devre dışı bırakır. Erişimi devre dışı bırakma, terminale yazmak isteyen ağdaki başka bir kullanıcının terminale yazmasını önleyecektir. Bu düzenlemekte olduğunuz metin dosyasının ortasında görünebilecek ani bir pizza siparişi hakkındaki mesajın olması çok can sıkıcı olabilir. Çok kullanıcılı bir ağ üzerinde, bu nedenle gerektiğinde terminal yazma erişimini kesintiye uğramaması için devre dışı bırakmak isteyebilirsiniz.

wall

Bu, "tümüne yazmak" için kullanılan bir kısaltmadır, yani, ağ oturumu açmış her terminalindeki bütün kullanıcılar için bir mesaj gönderir. Bu, öncelikle bir sistem yöneticisinin aracıdır, herkesin duyması gereken bir mesaj için yararlıdır, örneğin sistemin

kısa bir süre bir sorun nedeniyle çalışamayacak olmasının herkese uyarı şeklinde duyurulması gibi (bkz.Örnek 17.2).

```
bash$ wall System going down for maintenance in 5 minutes!

Broadcast message from bozo (pts/1) Sun Jul 8 13:53:27 2001...

 System going down for maintenance in 5 minutes!
```

wall yazma erişimi **mesg** ile devre dışı bırakılan belirli bir terminale mesaj gönderemez.

dmesg

`stdout` çıktısı bulunan tüm sistem açılış mesajlarını listeler. Hata ayıklama ve hangi aygıt sürücülerinin kurulu ve hangi sistem kesmelerinin kullanımda olduğundan emin olmak için kullanışlıdır. **dmesg** çıktısı, tabii ki, bir komut dosyası içinde grep, sed veya awk ile ayrıştırılmış olabilir.

Bilgi ve İstatistikler

uname

Sistem özelliklerinin (OS, çekirdek sürümü, vb.) `stdout` çıktısını alır. `-a` seçeneği ile çağrıldığında ayrıntılı sistem bilgisi (bkz.Örnek 12.4) verir. `-s` seçeneği sadece işletim sistemi türünü gösterir.

```
bash$ uname -a

Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000
i686 unknown

bash$ uname -s

Linux
```

arch

Sistem mimarisini gösterir. **uname -m** eşdeğeridir. Bkz. Örnek 10.25.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

`/var/account/pacct` dosyasında saklandığı gibi önceki komutlar hakkında bilgi verir, komut adı ve kullanıcı adı seçenekleri belirtilebilir. Bu GNU muhasebe araçlarından biridir.

lastlog

Tüm sistem kullanıcılarının son oturum açma zamanını listeler. `/var/log/lastlog` dosyasına referans verir.

```
bash$ lastlog
root          tty1             Fri Dec 7 18:43:21 -0700 2001
 bin                              **Never logged in**
 daemon                           **Never logged in**
 ...
 bozo         tty1             Sat Dec 8 21:14:29 -0700 2001

bash$ lastlog | grep root
root          tty1             Fri Dec 7 18:43:21 -0700 2001
```

Çağıran kullanıcının `/var/log/lastlog` dosyası için okuma izni yoksa, bu komut başarısız olur.

lsof

Açık dosyaları listeler. Bu komut, tüm açık olan dosyaların detaylı bir tablo çıkışını sağlar ve onların sahibi, boyutu, bunlarla ilişkili işlemler hakkında bilgi verir. Tabii ki, **lsof** sonuçlarını ayrıştırmak ve analiz etmek için grep ve/veya awk oluğu yaratılabilir.

```
bash$ lsof
COMMAND PID   USER   FD   TYPE   DEVICE   SIZE    NODE   NAME
init      1   root   mem   REG   3,5      30748   30303  /sbin/init
init      1   root   mem   REG   3,5      73120   8069   /lib/ld-2.1.3.so
init      1   root   mem   REG   3,5      931668  8075   /lib/libc-2.1.3.so
cardmgr   213 root   mem   REG   3,5      36956   30357  /sbin/cardmgr
```

strace

Sistem çağrıları ve sinyallerini takip için tanı ve yanlış ayıklama aracıdır. Bunu çağırmanın en basit yolu **strace KOMUTUNU** kullanmaktır.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
uname({sys="Linux", node="bozo.localdomain", ...}) = 0
brk(0) = 0x804f5e4
...
```

Bu **truss** yardımcı programının Linux eşdeğeridir.

free

Tablo şeklindeki bellek ve önbellek kullanımını gösterir. Bu komut, çıktısını grep, awk ya da **Perl** kullanarak ayrıştırmaya borçludur. **procinfo** komutu **free** komutunun gösterdiği her bilgiyi ve daha fazlasını gösterir.

```
bash$ free

             total       used       free     shared    buffers     cached
Mem:         30504      28624       1880      15820       1608      16376
-/+ buffers/cache:      10640      19864
Swap:        68540       3128      65412
```

Kullanılmayan RAM belleği göstermek için:

```
bash$ free | grep Mem | awk '{ print $4 }'

1880
```

procinfo

`/proc` sözde-dosya sisteminden bilgi ve istatistik ayıklar ve listeler. Bu çok kapsamlı ve ayrıntılı bir liste verir.

```
bash$ procinfo | grep Bootup

Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

Aygıtları listeler, yani, yüklü donanım gösterir.

```
bash$ lsdev

Device     DMA    IRQ    I/O Ports

 ------------------------------------------------

cascade      4      2

dma                       0080-008f

dma1                      0000-001f

dma2                      00c0-00df

fpu                       00f0-00ff

ide0               14     01f0-01f7 03f6-03f6

...
```

du

Öz-yineli, (disk) dosyası kullanımını göster. Varsayılan, aksi belirtilmediği sürece, geçerli çalışma dizinidir.

```
bash$ du -ach

1.0k     ./wi.sh

 1.0k     ./tst.sh

 1.0k     ./random.file

 6.0k

 6.0k     total
```

df

Dosya sistemi kullanımını tablo şeklinde gösterir.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5               273262     92607    166547  36% /
/dev/hda8               222525    123951     87085  59% /home
/dev/hda7              1408796   1075744    261488  80% /usr
```

stat

Belirli bir dosya veya dosya seti (bir dizin veya aygıt dosyası da olur) hakkında ayrıntılı *istatistikler* verir.

```
bash$ stat test.cru
  File: "test.cru"
  Size: 49970        Allocated Blocks: 100   Filetype: Regular File
  Mode: (0664/-rw-rw-r--) Uid: ( 501/ bozo)    Gid: ( 501/ bozo)
Device: 3,8 Inode: 18185 Links: 1
Access: Sat Jun 2 16:40:24 2001
Modify: Sat Jun 2 16:40:24 2001
Change: Sat Jun 2 16:40:24 2001
```

Hedef dosya yoksa, **stat** bir hata mesajı döndürür.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

Sanal bellek istatistiklerini görüntüler.

```
bash$ vmstat

   procs                  memory    swap          io  system         cpu

r   b  w  swpd   free   buff   cache si   so    bi    bo    in    cs   us  sy   id

0   0  0     0  11040   2636   38952  0    0    33     7   271    88    8   3   89
```

netstat

Yönlendirme tabloları ve etkin bağlantıları gibi mevcut ağ istatistiklerini ve bilgilerini gösterir, Bu yardımcı program `/proc/net` bilgilerine erişir (Bölüm 28). Bkz.Örnek 28.2.

netstat –r ile route eşdeğerdir.

uptime

İlgili istatistikler ile birlikte sistem aktif çalışma süresini gösterir.

```
bash$ uptime

10:28pm up 1:57, 3 users, load average: 0.17, 0.34, 0.27
```

hostname

Ana bilgisayarın sistem adını listeler. Bu komut `/etc/rc.d` kurulum komut dosyasını (`/etc/rc.d/rc.sysinit` veya benzerleri) çalıştırarak ana bilgisayar adını ayarlar. Bu **uname –n** eşdeğeridir, ve $HOSTNAME iç değişkenine karşılık gelir.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

hostid

Ana makine için 32-bitlik onaltı tabanında bir sayısal tanımlayıcı gösterir.

```
bash$ hostid
7f0100
```

Bahsedildiği üzere, bu komut belirli bir sistem için "tek" seri numarasını getirmektedir. Bazı ürün tescil işlemleri bu benzersiz sayıyı belirli bir kullanıcı lisansını markalamak için kullanmaktadır. Ne yazık ki, hostid makinenin ağ adresini sadece onaltılık tabanda döndürür, bayt çiftlerinin sırası ters çevrilip yer değiştirmiştir. Ağa bağlı olmayan tipik Linux makinesinin ağ adresi, `/etc/hosts` dosyasında bulunur.

```
bash$ cat /etc/hosts
127.0.0.1 localhost.localdomain localhost
```

`127.0.0.1` bayt çiftleri yer değiştiğinde `0.127.1.0` olacağı için onaltılık tabanda bu `007f0100` olarak yazılmalıdır, yukarıda da gösterildiği gibi **hostid** de tam olarak bu sayıyı getirir. Sadece bu aynı *hostid* sahibi, birkaç milyon daha Linux makinesi vardır.

sar

sar (sistem faaliyet raporu) çağrılması sistemi istatistiklerinin çok ayrıntılı bir özetini verir. Bu komut, bazı ticari UNIX sistemleri üzerinde bulunur, ancak temel Linux dağıtımının bir

parçası değildir. Bu, sysstat yardımcı program paketi içinde yer alır, Sebastien Godard tarafından yazılmıştır.

```
bash$ sar

Linux 2.4.7-10 (localhost.localdomain) 12/31/2001

10:30:01 AM CPU %user %nice %system %idle

10:40:00 AM all 1.39 0.00 0.77 97.84

10:50:00 AM all 76.83 0.00 1.45 21.72

11:00:00 AM all 1.32 0.00 0.69 97.99

11:10:00 AM all 1.17 0.00 0.30 98.53

11:20:00 AM all 0.51 0.00 0.30 99.19

06:30:00 PM all 100.00 0.00 100.01 0.00

Average: all 1.39 0.00 0.66 97.95
```

Sistem Kayıtları

logger

Bir kullanıcı tarafından oluşturulan mesajı sistem günlüğüne (`/var/log/messages`) ekler. **logger** kök olmayan kullanıcılar tarafından da çağrılabilir.

```
logger Experiencing instability in network connection at 23:10, 05/21.

# Now, do a 'tail /var/log/messages'.
```

Bir komut dosyası içine **logger** komutunu gömerek, hata ayıklama bilgilerini `/var/log/messages` dosyasına yazmak mümkündür.

```
logger -t $0 -i Logging at line "$LINENO".

# The "-t" option specifies the tag for the logger entry.

# The "-i" option records the process ID.

# tail /var/log/message

# ...

# Jul 7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

Bu yardımcı program, sistem kayıt dosyalarını yönetir, uygun olacak şekilde içeriğini döndürür, sıkıştırır, siler ve/veya e-postalama yapabilir. Genellikle crond düzenli olarak her gün **logrotate** ile birlikte çalışır.

`/etc/logrotate.conf` dosyasına yazılacak uygun bir girdi satırı kişisel ve bunun yanında sistem-çapında olan günlük dosyalarını yönetmeye olanak verir.

İş Denetimi

ps

İstatistik Süreci: Şu anda çalışan süreçlerin, sahip ve PID (işlem kimliği) ile yürütülmesini sağlar. Çağrılış şekli genellikle, `ax` seçenekleri iledir ve özel işlem aramasına yarayan bir grep ya da sed oluğu yaratılabilir (bkz.Örnek 11-9, ve Örnek 28-1).

```
bash$ ps ax | grep sendmail

295 ? S 0:00 sendmail: accepting connections on port 25
```

pstree

Şu anda yürütülen "ağaç" biçiminde süreçleri listeler. `-p` seçeneği hem işlem kimliğini (PID), hem de işlem adlarını gösterir.

top

Sürekli güncellenen en cpu-yoğun süreçlerin ekranda görüntülenmesi. `-b` seçeneği metin modunda görüntüler, böylece çıktı ayrıştırılır veya bir komut dosyasından erişilebilir.

```
bash$ top –b

   8:30pm up 3 min, 3 users, load average: 0.49, 0.32, 0.13

  45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped

 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle

 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,     2352K buff

 Swap: 157208K av,        0K used,  157208K free                   37244K cached


PID  USER      PRI    NI   SIZE   RSS   SHARE STAT   %CPU  %MEM   TIME COMMAND

848  bozo       17    0     996   996   800   R       5.6  1.2    0:00 top

  1  root        8    0     512   512   444   S       0.0  0.6    0:04 init

  2  root        9    0       0     0     0   SW      0.0  0.0    0:00 keventd

...
```

nice

Değişmiş öncelikli bir arka plan iş çalıştırır, 19 - eksi 20 aralığındaki öncelikler düşükten yükseğe doğrudur. Sadece *kök* negatif (yüksek) öncelikleri ayarlayabilir. İlgili komutlar arasında **renice**, **snice** ve **skill** sayılabilir.

nohup

Kullanıcı oturumu kapattıktan sonra bile çalışan komutları saklar. Komutun ardından & karakteri yazılmazsa, ön süreç olarak çalışacaktır. Bir komut dosyası içinde **nohup** kullanıyorsanız, yetim veya zombi süreç oluşturmayı önlemek için wait ile koordineli çalışın.

pidof

Çalışan bir işin *işlem kimliğini (PID)* tanımlar. kill ve **renice** gibi iş kontrol komutları bir sürecin *pid* üzerinde (adı değil) hareket ettiği için bu *pid*'in tanımlanması gerekmektedir. İşte tam bu noktada, **pidof** komutu $PPID iç değişkenine karşılık gelir.

```
bash$ pidof xclock

880
```

ÖRNEK 13.4. SÜREÇ ÖLDÜRMEDE pidof KULLANIMI

```
#!/bin/bash

# kill-process.sh

NOPROCESS=2

process=xxxyyyzzz         # Use nonexistent process.

# For demo purposes only...
# ... don't want to actually kill any actual process with this script.
#
# If, for example, you wanted to use this script to logoff the Internet,
# process=pppd

t=`pidof $process`        # Find pid (process id) of $process.
# The pid is needed by 'kill' (can't 'kill' by program name).
if [ -z "$t" ]            # If process not present, 'pidof' returns null.
then
  echo "Process $process was not running."
  echo "Nothing killed."
  exit $NOPROCESS
fi
```

```
kill $t                   # May need 'kill -9' for stubborn process.
# Need a check here to see if process allowed itself to be killed.
# Perhaps another " t=`pidof $process` ".


# This entire script could be replaced by
# kill $(pidof -x process_name)
# but it would not be as instructive.

exit 0
```

fuser

Seçili bir dosya, dosya seti veya dizine ulaşan süreçlerin kimliğini belirler. -k seçeneği ile çağrılabilir, süreçleri öldürme gerektiği zaman bu seçenek kullanılır. Sistem güvenliği için ilginç etkileri vardır, özellikle yetkisiz kullanıcıların sistem hizmetlerine erişiminin önlenmesinde.

crond

Yönetimsel Program Zamanlayıcısı. Temizlik ve sistem günlük dosyalarını silme gibi görevleri yerine getirir ve slocate veritabanını günceller. Bu, at süper kullanıcısının yaptığı işlerdir (Her kullanıcı **crontab** komutu ile değiştirilebilen kendi crontab dosyasına sahip olabildiği halde) . Bir geri plan yordamı olarak çalışır ve /etc/crontab dosyasından okuduğu zamanı önceden belirlenmiş olan kayıtları yürütür.

Süreç Kontrol ve Başlatma

init

init komutu tüm süreçlerin üstündedir. Bir önyüklemenin son adımı olarak adlandırılan **init,** sistemin çalışma seviyesini /etc/inittab dosyasından belirler. Öteki adı **telinit** olup sadece kök tarafından çağrılabilir.

telinit

Sembolik olarak **init** komutuna bağlı olup sistemin çalışma seviyesini değiştirmek için bir yöntemdir. Genellikle dosya sistemi acil onarım ve bakım için kullanılır. Sadece kök

tarafından çağrılabilir. Bu komut, tehlikeli olabilir - kullanmadan önce iyi anladığınızdan emin olmanız gereklidir!

runlevel

Mevcut ve son çalışma seviyesini gösterir, sistem durduruldu mu (çalışma seviyesi 0), tek kullanıcı modunda mı (1), çoklu kullanıcı modunda mı (2 veya 3), X Windows (5) içinde mi veya yeniden mi başlatılmış (6)? Bu komut /var/run/utmp dosyasına erişir.

halt, shutdown, reboot

Sistemin yeniden açılana dek güçten (elektrikten) yoksun olmasını sağlar.

Ağ

ifconfig

Ağ arabirimi yapılandırma ve ayarlama programı. En sık olarak, sistem açılışı sırasında arayüzleri kurmak için ya da sistemi yeniden başlatırken arayüz elektriğini yoksun bırakmak için kullanılır.

```
# Code snippets from /etc/rc.d/init.d/network
# ...
# Check that networking is up.
[ ${NETWORKING} = "no" ] && exit 0
[ -x /sbin/ifconfig ] || exit 0
# …
for i in $interfaces ; do
if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
  action "Shutting down interface $i: " ./ifdown $i boot
fi
# The GNU-specific "-q" option to "grep" means "quiet", i.e., producing no
output.
# Redirecting output to /dev/null is therefore not strictly necessary.
# ...
echo "Currently active devices:"
echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
#                             ^^^^^ should be quoted to prevent globbing.
# The following also work.
# echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
# Thanks, S.C., for additional comments.
```

Bkz.Örnek 30.6.

route

Çekirdek yönlendirme tablosuna değişiklik yapar veya sadece bilgi gönderir.

```
bash$ route
Destination      Gateway          Genmask         Flags  MSS Window   irtt  Iface
 pm3-67.bozosisp  *               255.255.255.255  UH      40 0           0  ppp0
 127.0.0.0        *               255.0.0.0        U       40 0           0  lo
 default          pm3-67.bozosisp 0.0.0.0          UG      40 0           0  ppp0
```

chkconfig

Ağ yapılandırmasını kontrol eder. Bu komut önyükleme sırasında başlatılan ve `/etc/rc?.d` dizininde yer alan ağ servislerini listeler ve yönetir. Orijinal olarak IRIX kökenli olup Red Hat Linux'a geçen **chkconfig** Linux temel yüklemelerinin parçası olmak zorunda değildir.

```
bash$ chkconfig –list
atd 0:off 1:off 2:off 3:on 4:on 5:on 6:off
rwhod 0:off 1:off 2:off 3:off 4:off 5:off 6:off
...
```

tcpdump

Ağ paket "dinleyicisi"dir. Bu belirtilen kriterlere uyan paket başlıklarının dökümüyle bir ağ trafiğinin analizi ve sorun giderilmesi amacıyla kullanılan bir araçtır.

bozoville ve *caduceus* ana makineleri arasında ip paket trafiğinin dökümü:

```
bash$ tcpdump ip host bozoville and caduceus
```

Tabii ki, daha önce tartışılan belirli metin işleme araçlarını kullanarak **tcpdump** çıktısı ayrıştırılabilir.

Dosya Sistemi

mount

Genellikle bir disket ya da CDROM gibi dışsal bir cihaza bir dosya sistemini monte eder. `/etc/fstab` dosyası kullanılabilir dosya sistemlerinin, bölüntülerin ve aygıtların kullanışlı bir listesini otomatik olarak ya da elle monte edilebilir seçenekler de dahil olmak üzere sağlar. `/etc/mtab` dosyası şu an için monte edilmiş dosya sistemlerini ve bölüntüleri gösterir (örneğin, `/proc` gibi sanal olanlar dahil).

mount-a `/etc/fstab` dosyasında listelenen tüm dosya sistemlerini ve bölüntüleri, `noauto` seçeneği olanlar dışında kalanları monte eder. Önyükleme sırasında `/etc/rc.d` (`rc.sysinit` veya benzeri) adlı başlangıç komut dosyası her şeyi monte etmek için bunu çağırır.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Mounts CDROM
mount /mnt/cdrom
# Shortcut, if /mnt/cdrom listed in /etc/fstab
```

Bu çok yönlü komut, bir blok aygıtı üzerinde sıradan bir dosyayı bile monte edebilir ve sonrasında dosya bir dosya sistemiymiş gibi hareket edecektir.

mount dosyayı geri dönüş devresine sahip bir döngü aygıtıyla ilişkilendirerek monte etmeyi başarıyla sonuçlandırır. Bunun bir uygulaması olarak CDR üzerine yazmadan önce bir ISO9660 görüntüsünün montajı ve incelenmesidir. [3]

ÖRNEK 13.5 BİR CD GÖRÜNTÜSÜNÜN KONTROL EDİLMESİ

```
# As root...
mkdir /mnt/cdtest # Prepare a mount point, if not already there.
mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest # Mount the image.
# "-o loop" option equivalent to "losetup /dev/loop0"
cd /mnt/cdtest    # Now, check the image.
ls -alR           # List the files in the directory tree there.
                  # And so forth.
```

umount

Monte edilmiş bir dosya sisteminin bağlantısının yok eder. Önceden monte edilmiş disket ya da CDROM diskin fiziksel olarak çıkarılmasından önce, aygıtın bağlantısını **umount** komutuyla kesmek gerekir. Bunun nedeni dosya sisteminde arızalara yol açabilmesidir.

```
umount /mnt/cdrom
# You may now press the eject button and safely remove the disk.
```

Düzgün yüklendiyse, **automount** programı, disket ya da CDROM diskler için erişim ve ortadan kaldırım sırasında mount ve unmount olanağı tanır. Dizüstü bilgisayarlarda bulunan takas disket ve CDROM sürücüleri ile kullanılması sorunsuz geçmeyebilir.

sync

Arabellekte bulunan tüm güncel verilerin sabit diske hemen yazılmasına yol açar, eşzamanlama amaçlı kullanılır. **sync** komutu verinin ani bir elektrik kesintisi sonrasında da korunacağını sistem yöneticisi veya kullanıcına yansıtır. Eskiden, **`sync; sync`** (iki defa üst üste yazılmış olan) bir sistemi yeniden başlatmadan önce faydalı bir önlem olarak karşımıza çıkmaktaydı.

Zaman zaman, bir dosyayı güvenli bir şekilde silmek için (bkz.Örnek 12.41) ya da ışıkları yanıp sönmeye başladığında, hemen bir önbellek yanaştırması yapılabilir.

losetup

Geri dönüş devresine sahip döngü aygıtları kurar ve yapılandırır.

ÖRNEK 13.6 BİR DOSYA SİSTEMİNİ OLUŞTURMA DOSYASI

```
SIZE=1000000 # 1 meg

head -c $SIZE < /dev/zero > file     # Set up file of designated size.

losetup /dev/loop0 file              # Set it up as loopback device.

mke2fs /dev/loop0                    # Create filesystem.

mount -o loop /dev/loop0 /mnt        # Mount it.

# Thanks, S.C.
```

mkswap

Bir takas bölüntüsü veya dosyası oluşturur. Takas alanı daha sonra **swapon** ile etkinleştirilmelidir.

swapon, swapoff

Takas bölüntüsünü veya dosyasını etkinleştirir/devre dışı bırakır. Bu komutlar genellikle önyükleme ve son verme sırasında etkili olur.

mke2fs

Oluşturduğu dosya sistemi Linux ext2 olup kök olarak çağrılması gerekir.

ÖRNEK 13.7 SABİT DİSKE YENİSİNİ EKLEME

```
#!/bin/bash
# Adding a second hard drive to system.
# Software configuration. Assumes hardware already mounted.
# From an article by the author of this document.
# in issue #38 of "Linux Gazette", http://www.linuxgazette.com.
ROOT_UID=0    # This script must be run as root.
E_NOTROOT=67  # Non-root exit error.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

# Use with extreme caution!
# If something goes wrong, you may wipe out your current filesystem.
NEWDISK=/dev/hdb           # Assumes /dev/hdb vacant. Check!
MOUNTPOINT=/mnt/newdisk    # Or choose another mount point.
fdisk $NEWDISK
mke2fs -cv $NEWDISK1   # Check for bad blocks & verbose output.
# Note: /dev/hdb1, *not* /dev/hdb!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT # Makes new drive accessible to all users.

# Now, test...
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# Try creating a directory.
# If it works, umount it, and proceed.
```

```
# Final step:

# Add the following line to /etc/fstab.

# /dev/hdb1 /mnt/newdisk ext2 defaults 1 1

exit 0
```

Bkz.Örnek 13.6 ve Örnek 29.3.

tune2fs

ext2 dosya sistemini ayarlar. Bu maksimum montaj sayısı olarak dosya sistemi parametrelerini değiştirmek için kullanılabilir. Bunun kök olarak çağrılması gerekir.

Bu son derece tehlikeli bir komuttur, kendi sorumluluğunuzda kullanın, yanlışlıkla dosya sisteminizi yok edebilir.

dumpe2fs

Çok ayrıntılı dosya sistemi bilgilerini döker (`stdout` listesi). Bunun kök olarak çağrılması gerekir.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'

dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09

Mount count: 6

Maximum mount count: 20
```

hdparm

Sabit disk parametrelerini listeler veya değiştirir. Bu komutun kök olarak çağrılması gerekir, yanlış kullanıldığında tehlikeli olabilir.

fdisk

Genellikle sabit disk gibi bir depolama aygıtında, bölüntü tablosu oluşturur ya da değiştirir. Bu komutun kök olarak çağrılması gerekir.

Bu komutu çok dikkatli kullanın. Yolunda olmayan bir şeylerle, varolan bir dosya sistemini yok etmek gibi yanlış bir sonuçla karşılaşabilirsiniz.

fsck, e2fsck, debugfs

Dosya sistemi kontrolü, onarım ve hata ayıklama komut setidir.

fsck: UNIX dosya sistemini kontrol etmek için bir ön uçtur (Diğer yardımcı programları çağırması mümkündür). Varsayılan gerçek dosya sistemi genellikle ext2 türüdür.

e2fsck: ext2 dosya sistemi denetleyicisi.

debugfs: ext2 dosya sistemi hata ayıklama programı. Bu çok yönlü, ama tehlikeli komutun kullanım alanları arasında silinmiş dosyaları kurtarmak (girişimi) yer alır. İleri düzeydeki kullanıcılar için uygun bulunmaktadır!

Bunların hepsinin kök olarak çağrılması gerekir ve yanlış kullanıldıklarında dosya sistemine zarar verebilir veya yok edebilir.

badblocks

Bir depolama aygıtı üzerinde kötü bloklar (fiziksel ortam kusurlarını) denetler. Bu komutun kullanım alanları arasında yeni yüklenen bir sabit diski biçimlendirme veya yedekleme ortamı bütünlüğünü test etme sayılabilir. [4] Örneğin, **badblocks /dev/fd0** bir flopi disketi test eder.

badblocks komutunun çağrılması yıkıcı sonuçlara neden olabilir (tüm verilerin üzerine yazabilir) veya salt-okunur modda zararsız sonuçlarla da çağrılabilir. Kök test edilecek aygıtın aynı zamanda sahibi ise, genellikle olduğu gibi, daha sonra kök'ün bu komutu çağırması gerekir.

mkbootdsk

Sistemi göstermek için kullanılabilecek bir önyükleme disketi oluşturur, örneğin, MBR (ana önyükleme kaydı) bozulursa, **mkbootdisk** komutu aslında bir Bash betiğidir. `/sbin` dizininde Erik Troan tarafından yazılmıştır.

chroot

Kök dizinini değiştirir. Normalde komutlar varsayılan kök dizinine (/) göreli olan $PATH değişkeninden getirilir. Bu kök dizini farklı bir dizine değiştirir (çalışma dizinini de aynı dizine ayarlar). Bu güvenlik amaçları için yararlıdır, örneğin sistem yöneticisi belirli kullanıcılara kısıtlamak istediği zaman, dosya sisteminin güvenli kısmına telnet protokolü ile erişim yapılmak istendiği zaman (Bu bazen için bir konuk kullanıcının bir "chroot hapis" ile hapsedilmesi olarak adlandırılır). Unutmayınız ki, sistem ikili dosyalarını için yürütme yolu, chroot komutundan sonra artık geçerli olmayacaktır.

`chroot /opt` çağrılması `/usr/bin` dizinine referans edilenlerin `/opt/usr/bin` dizinine taşınmasını sağlar. Aynı şekilde, `chroot /aaa/bbb/bin/ls` çağrılması gelecekteki **ls** somutlaşan örneklerini ve olgularını `/aaa/bbb` ana dizinine yönlendirecektir. Burada, normal olarak `/` gerekiyordu.

Kullanıcının `~/.bashrc` dizininde **'chroot /aaa/bbb ls'** komutuna verilecek **sözde-ad** olan **XX**, o kişinin "XX" komutunu çalıştırabildiği dosya kısmınını etkili bir şekilde kısıtlar.

Acil önyükleme disketi çalıştığında **chroot** komutu da kullanışlıdır (**chroot** to `/dev/fd0`) ya da bir sistem çökmesi sonrası iyileşme için bir **lilo** seçeneği olarak karşımıza çıkabilir. Diğer kullanım alanları farklı bir dosya sisteminin kurulumudur (bir **rpm** seçeneği), ya da CD ROM'dan salt okunur dosya sistemi çalıştırma, vb. Sadece kök olarak çağırın ve dikkatli kullanın. Normal `$PATH` değişkeni güvenilir olmadığından bu, *chroot edilmiş* bir dizine belirli sistem dosyalarını kopyalamaya yarar.

lockfile

Bu yardımcı, procmail (www.procmail.org) paketinin bir parçasıdır. Bu bir *kilit dosyası* oluşturur, bir dosya, aygıt, veya kaynağa erişimi denetleyen bir işaret flaması (semafor) dosyasıdır. Kilit dosyası belirli bir dosyanın, aygıtın, veya kaynağın, belirli bir işlem tarafından kullanılıyor ("meşgul") olmasını ve bu diğer işlemler için sadece sınırlı erişim izin veren (veya hiçbir erişime izin vermeyen) bir işaretidir.

Kilit dosyaları aynı anda birden fazla kullanıcı tarafından değiştirilmesini önleyen sistem posta klasörlerini koruyan, bir modem bağlantı noktasına erişildiğini belirten, ve Netscape bir somut örneğinin veya olgusunun önbelleği kullandığını gösteren uygulamalarda kullanılır. Bir sürecin çalışıp çalışmadığını kontrol etmek ve belirli bir işlem tarafından o işlemin yürütüldüğünün kontrol edilmesi için oluşturulan komut dosyaları geliştirilmiştir.

Bir komut dosyası zaten var olan bir kilit dosyasını oluşturmaya girişirse oluştururs anız, komut dosyası olasılıkla askıda kalacaktır.

Normalde, uygulamalar `/var/lock` dizininde kilit dosyalarını oluşturur ve kontrol eder. Bir komut dosyasının oluşturulmuş olup olmadığını aşağıdaki gibi bir şey ile test edebilirsiniz.

```
appname=xyzip

# Application "xyzip" created lock file "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock ]

then

  ...
```

mknod

Blok veya karakter aygıt dosyaları oluşturur (Sistem üzerine yeni donanım yüklerken gerekli olabilir).

tmpwatch

Belirli bir süre içinde erişilmemiş dosyaları otomatik olarak siler. Genellikle ilgi ortağı kayıt dosyalarını ortadan kaldırmak için crond tarafından çağrılır.

MAKEDEV

Aygıt dosyalarını oluşturmak için yardımcı programdır. Kök olarak `/dev` dizininde çalıştırılmalıdır.

```
root# ./MAKEDEV
```

Bu gelişmiş bir **mknod** versiyonu türüdür.

Yedekleme

dump, restore

dump komutu genellikle büyük kurulum ve ağlarda kullanılan ayrıntılı bir dosya yedekleme yardımcı programıdır. [5] Bu ham disk bölümlerini okur ve ikili bir biçimde bir yedek dosyası yazar. Yedeklenecek dosyaları disk ve teyp sürücüleri de dahil olmak üzere çeşitli depolama ortamına kaydedilebilir. **restore** komutu, **dump** ile yapılan yedeklemeleri geri yükler.

fdformat

Bir disket üzerinde düşük seviyeli format gerçekleştirir.

Sistem Kaynakları

ulimit

Genellikle `-f` seçeneği ile çağrıldığında, sistem kaynakları üzerinde bir *üst limit* belirler. Dosya boyutu için bir sınır belirlemeye yarar. (**ulimit -f 1000** dosya boyutunu en fazla 1MB olarak belirler.) `-t` seçeneği çekirdek döküm (coredump) boyutunu sınırlar. (**ulimit -c 0** çekirdek dökümü ortadan kaldırır). Normalde, **ulimit** değeri `/etc/profile` ve/veya `~/.bash_profile` içinde ayarlanabilecektir (bkz. Bölüm 27)

umask

Kullanıcılar için dosya oluşturma MASKESİDİR. Belirli bir kullanıcının varsayılan dosya özniteliklerini sınırlar. Bu kullanıcı tarafından oluşturulan tüm dosyalar umask tarafından belirtilen nitelikleri taşımak zorunda değildir. **Umask**'a geçirilen (sekiz tabanlı) değer dosya izinlerini *devre dışı* tanımlamaya olanak verir. Örneğin, **umask 022** yeni dosyalar için en fazla 755 izinlere (777 NAND 022) sahip olmasını sağlar. [6] Tabii ki, kullanıcı daha sonra belirli dosyaların niteliklerini chmod ile değiştirebilir. Her zamanki uygulama `/etc/profile` ve/veya `~/.bash_profile` içinde **umask** değerini ayarlamak olacaktır (Bkz. Bölüm 27).

rdev

Kök aygıt hakkında bilgi alır veya değişiklik yapar, görüntü modunu boşluk modu ve video arasında değiştirir. **rdev** işlevselliği genellikle **lilo** tarafından devir alınmıştır, ama **rdev** bir ram disk kurmak için yararlı olmaya devam etmektedir. Yanlış kullanıldığında, bu, komut da tehlikelidir.

Modüller

lsmod

Yüklü olan çekirdek modüllerini listeler.

```
bash$ lsmod


Module        Size      Used by
 autofs        9456       2 (autoclean)
 OPL3         11376       0
 serial_cs     5456       0 (unused)
 sb           34752       0
 uart401       6384       0 [sb]
 sound        58368       0 [opl3 sb uart401]
 soundlow       464       0 [sound]
 soundcore     2800       6 [sb sound]
 ds            6448       2 [serial_cs]
 i82365       22928       2
 pcmcia_core  45984       0 [serial_cs ds i82365]
```

insmod

Bir çekirdek modulün kurulumu için kuvvet kullanır. Kök olarak çağrılması gerekir.

rmmod

Bir çekirdek modülü boşaltmak için kuvvet kullanır. Kök olarak çağrılması gerekir.

modprobe

Başlangıç betiklerinde normal olarak çağrılan modül yükleyicidir.

depmod

Modül bağımlılık dosyası oluşturur, genellikle başlangıç betiğinden çağrılır.

Diğer Bazı Komutlar

env

Bazı çevresel değişkenleri ayarlı veya değiştirilmiş (genel sistem ortamını değiştirmeden) olarak bir program veya komut dosyasını çalıştırır. `[değişkenadı = xxx]` betik süresince `değişkenadı` çevresel değişkeninin aranmasına izin verir.

Belirtilen herhangi bir seçenek yoksa, bu komut tüm çevresel değişkenlerin ayarlarını listeler.

Bash ve diğer Bourne kabuğu türevleri olarak, tek bir komutla çevre değişkenlerini ayarlamak mümkündür.

```
var1=value1 var2=value2 commandXXX
# $var1 and $var2 set in the environment of 'commandXXX' only.
```

Bir komut dosyasının ilk satırı ("sha-bang" hattı) kabuk veya yorumcu yolu bilinmiyorsa, **env** değişkenini kullanabilir.

```
#! /usr/bin/env perl

print "This Perl script will run,\n";

print "even when I don't know where to find Perl.\n";

# Good for portable cross-platform scripts,

# where the Perl binaries may not be in the expected place.

# Thanks, S.C.
```

ldd

Yürütülebilir bir dosya için paylaşılan lib bağımlılıkları gösterir.

```
bash$ ldd /bin/ls

libc.so.6 => /lib/libc.so.6 (0x4000c000)

/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

strip

Hata ayıklama sembolik başvurularını ortadan kaldırır. Bu, büyüklüğü azaltır, ancak hata ayıklamayı da imkansız hale getirir. Bu komutaya, genellikle ancak nadiren bir kabukta, fakat genellikle bir Makefile içinde rastlamak mümkündür.

nm

Bir araya getirilmiş derlenmiş ikili dosya içindeki sembolleri listeler.

rdisk

Uzaktan dağıtım istemcisi: bir uzak sunucuda bir dosya sistemini eşzamanlı hale getirir, klonlar veya yedekler.

Edindiğimiz yönetimsel komut bilgimiz dahilinde, bir sistem komut dosyasındaki komutları anlamamızın en kısa ve basit yolu **killall** komutunu anlamaktır. Sistemin kapatılması sırasında çalışan süreçleri askıya almak için kullanılır.

ÖRNEK 13.8 /etc/rc.d/init.d DOSYASI İLE killall KULLANIMI

```
#!/bin/sh
# --> Comments added by the author of this document marked by "# -->".
# --> This is part of the 'rc' script package
# --> by Miquel van Smoorenburg, miquels@drinkel.nl.mugnet.org
# --> This particular script seems to be Red Hat specific
# --> (may not be present in other distributions).
# Bring down all unneeded services that are still running (there shouldn't
# be any, so this is just a sanity check)
for i in /var/lock/subsys/*; do
        # --> Standard for/in loop, but since "do" is on same line,
        # --> it is necessary to add ";".
        # Check if the script is there.
        [ ! -f $i ] && continue
        # --> This is a clever use of an "and list", equivalent to:
        # --> if [ ! -f "$i" ]; then continue
        # Get the subsystem name.
        subsys=${i#/var/lock/subsys/}
        # --> Match variable name, which, in this case, is the file name.
        # --> This is the exact equivalent of subsys=`basename $i`.
        # --> It gets it from the lock file name, and since if there
        # --> is a lock file, that's proof the process has been running.
        # --> See the "lockfile" entry, above.

        # Bring the subsystem down.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
```

```
          /etc/rc.d/init.d/$subsys.init stop
        else
          /etc/rc.d/init.d/$subsys stop
        # --> Suspend running jobs and daemons
        # --> using the 'stop' shell builtin.
        fi
done
```

O kadar da kötü olmayan bu betik için söylenebilecek tek şey değişken eşleştirmesi ile ilgili görkemli işin yanı sıra, yeni bir şey yoktur.

Alıştırma 1. **halt** betiğinin analizini yapmak için `/etc/rc.d/init.d`'ye göz atınız. Bu kavram daha uzundur ama **killall**'a benzer. Ana dizininizde bu betiği içeren bir kopya oluşturun ve deneyim kazanın (betiği kök olarak yürütmeyin). Örneğin -vn bayrakları ile yürütülüşünün bir simülasyonunu çalışma olarak yapabilirsiniz **(`sh -vn betikadı`).** Kapsamlı yorum yapmanızda bir sakınca yoktur. "action" komutları "echos" olacak şekilde değiştirin.

Alıştırma 2. Daha karmaşık komut bazı komutlara bakmak için /etc/rc.d/init.d ' ye göz atınız. Bunların parçalarını anlamak mümkün müdür? Bunların analizi için yukarıdaki prosedürü uygulayın. Görüş kazanmak için ayrıca sysvinitfiles dosyasını inceleyebilirsiniz. Dosya, "initscripts" dokümantasyonunun bulunduğu /usr/share/doc/initscripts-?.?? dizininde yer almaktadır.

Notlar

[1] Bu bir Linux makine veya disk kotaları olan bir UNIX sistemindeki durumdur.

[2] Silinen belirli bir kullanıcı hala oturum açtıysa **userdel** komutu başarısız olur.

[3] CDR yakılması ile ilgili daha fazla detay için Alex Withers tarafından yazılan Creating CDs adlı makaleyi Linux Journal dergisinin Ekim 1999 basımında bulabilirsiniz.

[4] mke2fs –c seçeneği ile birlikte kullanıldığında, kötü blokların kontrolü için de çağrı yapar.

[5] Tek kullanıcılı Linux sistemlerinin operatörleri yedekleme için genellikle daha basit şeyleri tercih ederler, **tar** gibi.

[6] NAND mantıksal "değil-ve" operatörüdür. Etkisi biraz çıkarmaya benzer.

BÖLÜM 14 KOMUT DEĞİŞTİRME

Komut değiştirme bir ya da birden çok komutun [1] yeniden çıktı atamasını yapar; komut çıktısını kelimenin tam anlamıyla başka bir bağlam içine oturtur.

Komut değiştirme klasik şekilde ters tırnak işaretini (‘...‘) kullanır. Ters tırnakların içindeki komutlar komut satırı metnini oluşturacaktır.

```
script_name=`basename $0`

echo "The name of this script is $script_name."
```

Komutların çıktısı, bir değişken atama ve hatta bir **for** döngüsü içinde bağımsız değişken listesi oluşturma amacıyla, başka bir komuta argüman olarak verilebilir.

```
rm `cat filename`     # "filename" contains a list of files to delete.

#

# S. C. points out that "arg list too long" error might result.

# Better is              xargs rm -- < filename

# ( -- covers those cases where "filename" begins with a "-" )


textfile_listing=`ls *.txt`

# Variable contains names of all *.txt files in current working directory.

echo $textfile_listing


textfile_listing2=$(ls *.txt) # The alternative form of command
substitution.

echo $textfile_listing

# Same result.

# A possible problem with putting a list of files into a single string

# is that a newline may creep in.

#
```

```
# A safer way to assign a list of files to a parameter is with an array.
#       shopt -s nullglob    # If no match, filename expands to nothing.
#       textfile_listing=( *.txt )
#
# Thanks, S.C.
```

Komut değişimi sözcük bölünmelerine neden olabilir.

```
COMMAND `echo a b`        # 2 args: a and b
COMMAND "`echo a b`"      # 1 arg: "a b"
COMMAND `echo`            # no arg
COMMAND "`echo`"          # one empty arg
# Thanks, S.C.
```

Sözcük bölünmesi var ya da yok olsa da, yeni satır sonları komut değişimiyle ortadan kaldırılabilir.

```
# cd "`pwd`"    # This should always work.
# However...
mkdir 'dir with trailing newline
'
cd 'dir with trailing newline
'
cd "`pwd`"      # Error message:
# bash: cd: /tmp/file with trailing newline: No such file or directory
cd "$PWD"       # Works fine.
```

```
old_tty_setting=$(stty -g)      # Save old terminal setting.
echo "Hit a key "
stty -icanon -echo              # Disable "canonical" mode for terminal.
                                # Also, disable *local* echo.
```

```
key=$(dd bs=1 count=1 2> /dev/null)  # Using 'dd' to get a keypress.

stty "$old_tty_setting"          # Restore old setting.

echo "You hit ${#key} key."      # ${#variable} = number of characters in
$variable

#

# Hit any key except RETURN, and the output is "You hit 1 key."

# Hit RETURN, and it's "You hit 0 key."

# The newline gets eaten in the command substitution.

# Thanks, S.C.
```

echo komutunu kullanarak, *tırnak içine alınmayan* bir değişken setinin çıktısını komut değişimi ile almak, yeniden atanmış komut(lar)ın çıktısından yeni satır karakterlerini ortadan kaldırır. Bu tatsız durumlara neden olabilir.

```
dir_listing=`ls -l`

echo $dir_listing                # unquoted

# Expecting a nicely ordered directory listing.

# However, what you get is:

# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo

# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# The newlines disappeared.


echo "$dir_listing"              # quoted

# -rw-rw-r--         1  bozo       30 May 13 17:15 1.txt

# -rw-rw-r--         1 bozo        51 May 15 20:57 t2.sh

# -rwxr-xr-x         1 bozo       217 Mar 5 21:13 wi.sh
```

Yeniden yönlendirme veya cat komutu kullanılarak, bir dosyanın içeriği komut değişimiyle bir değişkene atanabilir.

```
variable1=`<file1` # Set "variable1" to contents of "file1".
variable2=`cat file2` # Set "variable2" to contents of "file2".
# Be aware that the variables may contain embedded whitespace,
#+ or even (horrors), control characters.

#  Excerpts from system file, /etc/rc.d/rc.sysinit
#+ (on a Red Hat Linux installation)

if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
  sleep 5
  mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
  kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
...
fi
```

Uzun bir metin dosyasının içeriğini, çok iyi bir nedeniniz yoksa, bir değişkene ayarlamayın. Hatta bir ikili dosya içeriğini, şaka olarak bile olsa değişkene ayarlamayın.

ÖRNEK 14.1 BASİT KOMUT DOSYASI HİLELERİ

```
#!/bin/bash

# stupid-script-tricks.sh: Don't try this at home, folks.

dangerous_variable=`cat /boot/vmlinuz`   # The compressed Linux kernel
itself.

echo "string-length of \$dangerous_variable = ${#dangerous_variable}"

# string-length of $dangerous_variable = 794151

# (Does not give same count as 'wc -c /boot/vmlinuz'.)

# echo "$dangerous_variable"

# Don't try this! It would hang the script.

# The document author is aware of no useful applications for

#+ setting a variable to the contents of a binary file.

exit 0
```

Dikkat ediniz ki, bir *arabellek taşması* oluşmaz. Bu, Bash gibi bir yorumlanmış dilin, bir derlenmiş dile göre programcı hatalarından daha fazla koruma sağlamasına bir örnektir.

Komut değişimi, bir döngü çıktısını bir değişkene ayarlamaya izin verir. Bunun için önemli olan, döngü içinde echo komutunun çıktısını yakalamaktır.

ÖRNEK 14.2 BİR DÖNGÜ İÇİNDE BİR DEĞİŞKEN OLUŞTURMA

```
#!/bin/bash

# csubloop.sh: Setting a variable to the output of a loop.

variable1=`for i in 1 2 3 4 5

do
```

```
  echo -n "$i"    # The 'echo' command is critical
done`             #+ to command substitution.

echo "variable1 = $variable1"       # variable1 = 12345

i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"      # Again, the necessary 'echo'.
  let "i += 1"      # Increment.
done`
echo "variable2 = $variable2"         # variable2 = 0123456789
exit 0
```

Komut değişimiyle mevcut Bash araç setinin genişletilmesini mümkündür. Bu sadece, çıktısını `stdout`'a veren bir program veya komut dosyasının yazılı çıktısının bir değişkene atanması meselesidir (Bir UNIX aracının gerektirdiği gibi).

```
#include <stdio.h>
/* "Hello, world." C program */
int main()
{
  printf( "Hello, world." );
  return (0);
}
```

```
bash$ gcc -o hello hello.c
```

```
#!/bin/bash
# hello.sh
greeting=`./hello`
echo $greeting
```

```
bash$ sh hello.sh

Hello, world.
```

$(KOMUT) şeklindeki kullanım ters tırnak formunu kullanmadan da komut değişimi yaptırabilir.

```
output=$(sed -n /"$1"/p $file)

# From "grp.sh" example.
```

Kabuk betiklerinde komut değişimi örnekleri:

1. Örnek 10.7
2. Örnek 10.25
3. Örnek 9.23
4. Örnek 12.2
5. Örnek 12.15
6. Örnek 12.12
7. Örnek 12.38
8. Örnek 10.13
9. Örnek 10.10
10. Örnek 12.24
11. Örnek 16.7
12. Örnek A.17
13. Örnek 28.1
14. Örnek 12.32
15. Örnek 12.33
16. Örnek 12.34

Notlar

[1] *Komut değişimi* amacıyla bir **komut,** dış sistem komutu, iç komut dosyası *yerleşiği*, ve hatta betik fonksiyonu olabilir.

BÖLÜM 15 ARİTMETİK İFADELERİN KULLANIMI

Aritmetik işlemlerin komut dosyalarında gerçekleştirilmesini sağlayan güçlü aritmetik açılım araçları geliştirilebilir. Bu işlemler arasında bir karakter dizisini ters tırnaklar, çift parantez veya let komutunu kullanarak sayısal ifadeye çevirmek sayılabilir.

Değişimleri

Ters tırnaklar ile aritmetik ifadelerin kullanımı (genellikle expr ile birlikte kullanılır)

```
z=`expr $z + 3`     # 'expr' does the expansion.
```

Çift parantez ile **let** ve aritmetik ifadelerin kullanımı

Aritmetik ifadelerin açılımında artık ters tırnak kullanımı değil, çift parantez ya da çok uygun olan **let** yapısı kullanılmaktadır. Şöyle ki: **`$((...))`**

```
z=$(($z+3))
# $((EXPRESSION)) is arithmetic expansion. # Not to be confused with
                                          # command substitution.
let z=z+3
let "z += 3"    # If quotes, then spaces and special operators allowed.
# 'let' is actually arithmetic evaluation, rather than expansion.
```

Tüm yukarıdakiler eşdeğerdir. "Hangisi çanları size çaldıysa" ona seçim yapabilirsiniz.

Aritmetik ifadelerin kullanımının komut dosyalarında kullanımı ile ilgili örnekler:

1. Örnek 12.6
2. Örnek 10.14
3. Örnek 26.1
4. Örnek 26.4
5. Örnek A.17

BÖLÜM 16 G/Ç YENİDEN YÖNLENDİRME

Her zaman açık olan üç varsayılan "dosya" vardır, `stdin` (klavye), `stdout` (ekran), ve `stderr` (hata mesajlarının ekran çıktısı) Bu ve diğer açık dosyalar yeniden yönlendirilebilir. Yönlendirme sadece bir dosyadan, bir komuttan, bir komut dosyasından ve hatta bir komut dosyası içindeki kod bloğundan çıktı yakalamak (bkz Örnek 4.1 ve Örnek 4.2) değil, aynı zamanda bu çıktıyı bir başka dosya, komut, program veya komut dosyasına girdi olarak da yollayan bir kavramdır.

Her açık dosya bir dosya tanıtıcıya atanmalıdır. [1] `stdin`, `stdout` ve `stderr` için dosya tanıtıcıları sırasıyla 0, 1 ve 2 olarak tanımlıdır. Ek dosyalar açmak için varolan tanımlayıcılar 3-9 sayı aralığı arasında yer almalıdır. Bazen, bu dosya tanımlayıcılarının birini `stdin`, `stdout` ve `stderr`'e geçici birer ek bağlantı olarak tanımlamak yararlı olur. [2] Kullanıcıya kolay gibi gelen bu karmaşık yönlendirme sonrasında değişimin normal restorasyonu gerçekleşmiş olur (bkz. Örnek 16.1)

```
COMMAND_OUTPUT >

   # Redirect stdout to a file.

   # Creates the file if not present, otherwise overwrites it.

   ls -lR > dir-tree.list

   # Creates a file containing a listing of the directory tree.

: > filename

   # The > truncates file "filename" to zero length.

   # If file not present, creates zero-length file (same effect as
'touch').

   # The : serves as a dummy placeholder, producing no output.

> filename

   # The > truncates file "filename" to zero length.

   # If file not present, creates zero-length file (same effect as
'touch').

   # (Same result as ": >", above, but this does not work with some
shells.)

COMMAND_OUTPUT >>

   # Redirect stdout to a file.
```

```
   # Creates the file if not present, otherwise appends to it.
   # Single-line redirection commands (affect only the line they are on):
   # --------------------------------------------------------------------
  1>filename
     # Redirect stdout to file "filename".
  1>>filename
     # Redirect and append stdout to file "filename".
  2>filename
     # Redirect stderr to file "filename".
  2>>filename
     # Redirect and append stderr to file "filename".
  &>filename
     # Redirect both stdout and stderr to file "filename".

#==============================================================================
     # Redirecting stdout, one line at a time.
     LOGFILE=script.log
     echo "This statement is sent to the log file, \"$LOGFILE\"."
1>$LOGFILE
     echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
     echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
     echo "This statement is echoed to stdout, and will not appear in
\"$LOGFILE\"."
     # These redirection commands automatically "reset" after each line.


     # Redirecting stderr, one line at a time.
     ERRORFILE=script.errors
     bad_command1 2>$ERRORFILE      # Error message sent to $ERRORFILE.
     bad_command2 2>>$ERRORFILE     # Error message appended to $ERRORFILE.
     bad_command3                   # Error message echoed to stderr,
                                    #+ and does not appear in $ERRORFILE.
```

```
    # These redirection commands also automatically "reset" after each
line.


#============================================================================
==

2>&1

   # Redirects stderr to stdout.

   # Error messages get sent to same place as standard output.

i>&j

   # Redirects file descriptor i to j.

   # All output of file pointed to by i gets sent to file pointed to by j.

>&j

   # Redirects, by default, file descriptor 1 (stdout) to j.

   # All stdout gets sent to file pointed to by j.

0< FILENAME

 < FILENAME

   # Accept input from a file.

   # Companion command to ">", and often used in combination with it.

   #

   # grep search-word <filename

[j]<>filename

   # Open file "filename" for reading and writing, and assign file

   #+ descriptor "j" to it.

   # If "filename" does not exist, create it.

   # If file descriptor "j" is not specified, default to fd 0, stdin.

   #

   # An application of this is writing at a specified place in a file.

   echo 1234567890 > File      # Write string to "File".

   exec 3<> File               # Open "File" and assign fd 3 to it.

   read -n 4 <&3               # Read only 4 characters.

   echo -n . >&3               # Write a decimal point there.

   exec 3>&-                   # Close fd 3.
```

```
  cat File                      # ==> 1234.67890

  # Random access, by golly.

  |
  # Pipe.
  # General purpose process and command chaining tool.
  # Similar to ">", but more general in effect.
  # Useful for chaining commands, scripts, files, and programs together.
  cat *.txt | sort | uniq > result-file
  # Sorts the output of all the .txt files and deletes duplicate lines,
  # finally saves results to "result-file".
```

Birden çok girdi ve çıktı ve/veya oluklar tek bir komut satırı içinde yeniden yönlendirilerek birleştirilebilir.

```
command < input-file > output-file
```

```
command1 | command2 | command3 > output-file
```

Bkz. Örnek 12.23 ve Örnek A.15.

Yeniden yönlendirme bir dosyaya çoklu çıktı iş akış dizileri için de yapılabilir.

```
ls -yz >> command.log 2>&1
# Capture result of illegal options "yz" to "ls" in file "command.log".
# Because stderr redirected to the file, any error messages will also be
#+ there.
```

Dosya Tanımlayıcılarını Kapatma

`n<&-`

n girdi dosya tanımlayıcısını kapatmak için.

`0<&-, <&-`

`stdin`'in kapatılması için.

`n>&-`

n çıktı dosya tanımlayıcısını kapatmak için.

`1>&-, >&-`

`stdout`'un kapatılması için.

Alt süreçler açık dosya tanımlayıcılarından kalıt alır. Oluklar bu yolla çalışır. Bir dosya tanımlayıcısının kalıt olunmasını önlemek için kapatınız.

```
# Redirecting only stderr to a pipe.
exec 3>&1 # Save current "value" of stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&- # Close fd 3 for 'grep' (but not 'ls').
# ^^^^ ^^^^
exec 3>&- # Now close it for the remainder of the script.
# Thanks, S.C.
```

G/Ç yönlendirmesine daha ayrıntılı bir giriş için bkz. Ek D.

Notlar

[1] Bir *dosya tanıtıcı*, işletim sistemi tarafından açık bir dosyayı sadece takip etmek için atanan bir sayıdır. Dosya işaretçisine benzer, ancak daha anlaşılır ve basittir. Bu bir C *dosya sürücüsüne* benzer.

[2] *Dosya tanıtıcı* olarak 5 sayısının kullanılması sorunlara neden olabilir. Bash exec komutuyla yaratılan bir alt süreç dosya tanıtıcı olarak 5 devralır. (bkz. Chet Ramey'in arşiv e-postası https://groups.google.com/forum/?fromgroups=#!topic/gnu.bash.bug/E5Vdqv3tO1w) En doğrusu, bu özel dosya tanıtıcı sayının mümkünse programcı tarafından ele alınmamasıdır.

16.1 exec KULLANIMI

exec <dosya adı komutu, `stdin`'i bir dosyaya yönlendirmeye yarar. Bu noktadan itibaren, tüm `stdin` belirtilen dosyadan gelir, normal kaynağından (genellikle klavye girişi) gelmez. Bir dosyanın satır satır okunması ve büyük olasılıkla sed ve/veya awk kullanılarak her girdi satırının ayrıştırılması için varolan bir yöntemidir.

ÖRNEK 16.1 exec KULLANIMIYLA stdin'e YÖNLENDİRME YAPILMASI

```
#!/bin/bash

# Redirecting stdin using 'exec'.

exec 6<&0          # Link file descriptor #6 with stdin.

                   # Saves stdin.

exec < data-file # stdin replaced by file "data-file"

read a1 # Reads first line of file "data-file".

read a2 # Reads second line of file "data-file."

echo

echo "Following lines read from file."

echo "-------------------------------"

echo $a1

echo $a2

echo; echo; echo

exec 0<&6 6<&-

#  Now restore stdin from fd #6, where it had been saved,

#+ and close fd #6 ( 6<&- ) to free it for other processes to use.

#

# <&6 6<&- also works.

echo -n "Enter data "

read b1 # Now "read" functions as expected, reading from normal stdin.

echo "Input read from stdin."
```

```
echo "----------------------"

echo "b1 = $b1"

echo

exit 0
```

Benzer şekilde, **exec > dosya adı** komutu `stdout`'u belirli bir dosyaya yönlendirir. `stdout`'a gitmesi gereken çıktılar belirtilen dosya için yine aynı dosyaya gönderilir.

ÖRNEK 16.2 exec KULLANIMIYLA stdout'a YÖNLENDİRME YAPILMASI

```
#!/bin/bash

# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1 # Link file descriptor #6 with stdout.

          # Saves stdout.

exec > $LOGFILE # stdout replaced with file "logfile.txt".

# ----------------------------------------------------------- #

# All output from commands in this block sent to file $LOGFILE.

echo -n "Logfile: "

date

echo "-------------------------------------"

echo

echo "Output of \"ls -al\" command"

echo

ls -al

echo; echo

echo "Output of \"df\" command"

echo

df
```

```
# ------------------------------------------------------------ #
exec 1>&6 6>&- # Restore stdout and close file descriptor #6.
echo
echo "== stdout now restored to default == "
echo
echo
ls -al
echo
exit 0
```

ÖRNEK 16.3 AYNI KOMUT DOSYASI İÇİNDE exec KULLANIMIYLA, HEM stdin HEM DE stdout'a YÖNLENDİRME YAPILMASI

```
#!/bin/bash
# upperconv.sh
# Converts a specified input file to uppercase.
E_FILE_ACCESS=70
E_WRONG_ARGS=71
if [ ! -r "$1" ] # Is specified input file readable?
then
  echo "Can't read from input file!"
  echo "Usage: $0 input-file output-file"
  exit $E_FILE_ACCESS
fi            #  Will exit with same error
              #+ even if input file ($1) not specified.
if [ -z "$2" ]
then
  echo "Need to specify output file."
  echo "Usage: $0 input-file output-file"
```

```
  exit $E_WRONG_ARGS
fi


exec 4<&0
exec < $1       # Will read from input file.
exec 7>&1
exec > $2       # Will write to output file.
          # Assumes output file writable (add check?).
# -----------------------------------------------
cat - | tr a-z A-Z           # Uppercase conversion.
# ^^^^^ # Reads from stdin.
# ^^^^^^^^^^                # Writes to stdout.
# However, both stdin and stdout were redirected.
# -----------------------------------------------
exec 1>&7 7>&-              # Restore stout.
exec 0<&4 4<&-              # Restore stdin.
# After restoration, the following line prints to stdout as expected.
echo "File \"$1\" written to \"$2\" as uppercase conversion."
exit 0
```

16.2 KOD BLOKLARINI YÖNLENDİRME

while, until ve for gibi kod blokları ve hatta if/then test blokları stdin'in yönlendirilmesine mani değildir. Hatta bir fonksiyon bile bu yönlendirme işlevini kullanabilir (bkz. Örnek 23.7) Kod bloğu sonundaki < operatörü bunu gerçekleştirir.

ÖRNEK 16.4 YÖNLENDİRİLEN *while* DÖNGÜSÜ

```
#!/bin/bash
if [ -z "$1" ]
then
  Filename=names.data # Default, if no filename specified.
else
  Filename=$1
fi
#+ Filename=${1:-names.data}
#  can replace the above test (parameter substitution).
count=0
echo
while [ "$name" != Smith ] # Why is variable $name in quotes?
do
  read name # Reads from $Filename, rather than stdin.
  echo $name
  let "count += 1"
done <"$Filename" # Redirects stdin to file $Filename.
# ^^^^^^^^^^^^
echo; echo "$count names read"; echo
#  Note that in some older shell scripting languages,
#+ the redirected loop would run as a subshell.
```

```
#  Therefore, $count would return 0, the initialized value outside the
loop.

#  Bash and ksh avoid starting a subshell whenever possible,

#+ so that this script, for example, runs correctly.

# Thanks to Heiner Steven for pointing this out.


exit 0
```

ÖRNEK 16.5 ALTERNATİF ŞEKİLDE YÖNLENDİRİLEN *while* DÖNGÜSÜ

```
#!/bin/bash

# This is an alternate form of the preceding script.

# Suggested by Heiner Steven

#+ as a workaround in those situations when a redirect loop

#+ runs as a subshell, and therefore variables inside the loop

#+ do not keep their values upon loop termination.


if [ -z "$1" ]

then

  Filename=names.data # Default, if no filename specified.

else

  Filename=$1

fi


exec 3<&0 # Save stdin to file descriptor 3.

exec 0<"$Filename" # Redirect standard input.


count=0

echo
```

```
while [ "$name" != Smith ]
do
  read name # Reads from redirected stdin ($Filename).
  echo $name
  let "count += 1"
done <"$Filename" # Loop reads from file $Filename.
# ^^^^^^^^^^^^
exec 0<&3 # Restore old stdin.
exec 3<&- # Close temporary fd 3.
echo; echo "$count names read"; echoexit
exit 0
```

ÖRNEK 16.6 YÖNLENDİRİLEN *until* DÖNGÜSÜ

```
#!/bin/bash
# Same as previous example, but with "until" loop.
if [ -z "$1" ]
then
  Filename=names.data # Default, if no filename specified.
else
  Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ] # Change != to =.
do
  read name # Reads from $Filename, rather than stdin.
  echo $name
done <"$Filename" # Redirects stdin to file $Filename.
```

```
# ^^^^^^^^^^^^

# Same results as with "while" loop in previous example.

exit 0
```

ÖRNEK 16.7 YÖNLENDİRİLEN *for* DÖNGÜSÜ

```
#!/bin/bash

if [ -z "$1" ]

then

  Filename=names.data # Default, if no filename specified.

else

  Filename=$1

fi


line_count=`wc $Filename | awk '{ print $1 }'`

# Number of lines in target file.

#

#  Very contrived and kludgy, nevertheless shows that

#+ it's possible to redirect stdin within a "for" loop...

#+ if you're clever enough.

#  More concise is line_count=$(wc < "$Filename")

for name in `seq $line_count` # Recall that "seq" prints sequence of
numbers.

# while [ "$name" != Smith ] -- more complicated than a "while" loop --

do

  read name # Reads from $Filename, rather than stdin.

  echo $name

  if [ "$name" = Smith ] # Need all this extra baggage here.
```

```
  then
    break
  fi
done <"$Filename" # Redirects stdin to file $Filename.
# ^^^^^^^^^^^^
exit 0
```

Yukarıdaki örneği, döngünün çıktısını yönlendirmek için de kullanabiliriz.

ÖRNEK 16.8 YÖNLENDİRİLEN *for* DÖNGÜSÜ (HEM stdin HEM stdout YÖNLENDİRMESİYLE)

```
#!/bin/bash
if [ -z "$1" ]
then
  Filename=names.data # Default, if no filename specified.
else
  Filename=$1
fi

Savefile=$Filename.new # Filename to save results in.
FinalName=Jonah # Name to terminate "read" on.

line_count=`wc $Filename | awk '{ print $1 }'` # Number of lines in target
file.

for name in `seq $line_count`
do
  read name
```

```
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile" # Redirects stdin to file $Filename,
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ and saves it to backup file.
exit 0
```

ÖRNEK 16.9 YÖNLENDİRİLEN *if/then* KOŞULU

```
#!/bin/bash
if [ -z "$1" ]
then
  Filename=names.data # Default, if no filename specified.
else
  Filename=$1
fi

TRUE=1
if [ "$TRUE" ] # if true and if : also work.
then
  read name
  echo $name
fi <"$Filename"
# ^^^^^^^^^^^^
# Reads only first line of file.
# An "if/then" test has no way of iterating unless embedded in a loop.

exit 0
```

ÖRNEK 16.10 YUKARIDAKİ ÖRNEKLER İÇİN VERİ DOSYASI "names.data"

```
Aristotle

Belisarius

Capablanca

Euler

Goethe

Hamurabi

Jonah

Laplace

Maroczy

Purcell

Schmidt

Semmelweiss

Smith

Turing

Venn

Wilson

Znosko-Borowski


#  This is a data file for

#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Bir kod bloğu `stdout` yönlendirmesi yoluyla çıktısını dosyaya yazdırmış gibi gösterebilir. Bkz. Örnek 4.2.

Belge dokümanları yönlendirilmiş kod bloklarının özel bir türüdür.

16.3 UYGULAMALAR

G/Ç yönlendirmesi akılsızca kullanılırsa ayrıştırma ve komut çıktılarının bütünlük kazanması güçleşir (bkz. Örnek 11.5). Rapor ve günlük dosyası üretmek, yönlendirme tekniklerini akıllıca kullanmayı gerektirir.

ÖRNEK 16.11 OLAY GÜNLÜĞÜ OLUŞTURMA

```
#!/bin/bash

# logevents.sh, by Stephane Chazelas.

# Event logging to a file.

# Must be run as root (for write access in /var/log).

ROOT_UID=0 # Only users with $UID 0 have root privileges.

E_NOTROOT=67 # Non-root exit error.


if [ "$UID" -ne "$ROOT_UID" ]

then

  echo "Must be root to run this script."

  exit $E_NOTROOT

fi


FD_DEBUG1=3

FD_DEBUG2=4

FD_DEBUG3=5


# Uncomment one of the two lines below to activate script.

# LOG_EVENTS=1

# LOG_VARS=1


log() # Writes time and date to log file.

{
```

```
  echo "$(date) $*" >&7 # This *appends* the date to the file.
}
case $LOG_LEVEL in
   1) exec 3>&2 4> /dev/null 5> /dev/null;;
   2) exec 3>&2 4>&2 5> /dev/null;;
   3) exec 3>&2 4>&2 5>&2;;
   *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null # Bury output.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # Above line will not work in Bash, version 2.04.
  exec 7>> /var/log/event.log # Append to "event.log".
  log                         # Write time and date.
else exec 7> /dev/null # Bury output.
fi
echo "DEBUG3: beginning" >&${FD_DEBUG3}
ls -l >&5 2>&4 # command1 >&5 2>&4
echo "Done" # command2

echo "sending mail" >&${FD_LOGEVENTS} # Writes "sending mail" to fd #7.

exit 0
```

BÖLÜM 17 BELGE DOKÜMANLARI

Bir *belge dokümanı* ftp, telnet, veya ex gibi bir komut veya bir etkileşimli program için komut listesi beslemeye yarayan özel bir G/Ç yönlendirme metodudur. Bir "sınır dizesi"yle bu komut listesi çerçevelenir. Sınır dizesi ise << özel sembolüdür. Bir dosyanın çıktısını bir programa yönlendirmek gibi bir etkisi olduğu bilinir. **etkileşimli program < komut-dosyası** sözdizimi vardır. `komut-dosyası` içinde şunlar bulunur:

```
command #1
command #2
. . .
```

"Belge dokümanı" iskeleti şöyle olabilir:

```
#!/bin/bash
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

Yeterince alışılmadık bir sınır dizesi seçmelisiniz ki bu, komut listesinde karışıklığa neden olmasın.

Unutmayınız ki, *belge dokümanları* etkileşimli olmayan programlar ve komutlar için iyi bazı kullanım alanları bulabilmektedir.

ÖRNEK 17.1 DOLDURMA DOSYA : 2-SATIRLI DOLDURMA DOSYA OLUŞTURUR

```
#!/bin/bash
# Non-interactive use of 'vi' to edit a file.
# Emulates 'sed'.
```

```
E_BADARGS=65
if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

TARGETFILE=$1
# Insert 2 lines in file, then save.
#--------Begin here document-----------#
vi $TARGETFILE <<x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23
#----------End here document-----------#
# Note that ^[ above is a literal escape
#+ typed by Control-V <Esc>.
# Bram Moolenaar points out that this may not work with 'vim',
#+ because of possible problems with terminal interaction.

exit 0
```

Yukarıdaki komut dosyasının uygulamasını yalnızca **vi** ile değil, **ex** ile de yapabilirsiniz. Bir dizi **ex** komutu içeren belge dokümanları, *ex betikleri* olarak bilinen kendi kategorilerini oluşturmaya yeterlidir.

ÖRNEK 17.2 YAYGIN: OTURUM AÇAN HERKESE MESAJ GÖNDERİR

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23

E-mail your noontime orders for pizza to the system administrator.

   (Add an extra dollar for anchovy or mushroom topping.)

# Additional message text goes here.

# Note: Comment lines printed by 'wall'.

zzz23EndOfMessagezzz23

# Could have been done more efficiently by

# wall <message-file

# However, saving a message template in a script saves work.


exit 0
```

ÖRNEK 17.3 cat KOMUTUNU KULLANARAK ÇOKLU-SATIR MESAJ BİLDİRİMİ

```
#!/bin/bash

# 'echo' is fine for printing single line messages,

# but somewhat problematic for for message blocks.

# A 'cat' here document overcomes this limitation.

cat <<End-of-message

-------------------------------------

This is line 1 of the message.

This is line 2 of the message.

This is line 3 of the message.

This is line 4 of the message.

This is the last line of the message.
```

```
------------------------------------

End-of-message

exit 0

#-----------------------------------------

# Code below disabled, due to "exit 0" above.

# S.C. points out that the following also works.

echo "------------------------------------

This is line 1 of the message.

This is line 2 of the message.

This is line 3 of the message.

This is line 4 of the message.

This is the last line of the message.

------------------------------------"

# However, text may not include double quotes unless they are escaped.
```

Belge dokümanının sınır dizisini (**<<-LimitString**) işaretlemeye yarayan – seçeneği, çıktıda sekmeleri (fakat boşlukları değil!) bastırır. Bu komut dosyasını daha okunabilir hale koyar.

ÖRNEK 17.4 SEKMELER BASTIRILMIŞ HALDE ÇOKLU-SATIR MESAJ BİLDİRİMİ

```
#!/bin/bash

# Same as previous example, but...

# The - option to a here document <<-

# suppresses tabs in the body of the document, but *not* spaces.

cat <<-ENDOFMESSAGE

        This is line 1 of the message.

        This is line 2 of the message.

        This is line 3 of the message.

        This is line 4 of the message.
```

```
        This is the last line of the message.
ENDOFMESSAGE
# The output of the script will be flush left.
# Leading tab in each line will not show.
# Above 5 lines of "message" prefaced by a tab, not spaces.
# Spaces not affected by <<- .

exit 0
```

Bir belge dokümanı parametre ve komut değişimini destekler. Belge dokümanının gövdesine değişik parametreleri yollayarak çıktısını değiştirmek, bu nedenle mümkündür.

ÖRNEK 17.5 PARAMETRE DEĞİŞTİRME YAPAN BELGE DOKÜMANLARI

```
#!/bin/bash
# Another 'cat' here document, using parameter substitution.
# Try it with no command line parameters, ./scriptname
# Try it with one command line parameter, ./scriptname Mortimer
# Try it with one two-word quoted command line parameter,
# ./scriptname "Mortimer Jones"
CMDLINEPARAM=1 # Expect at least command line parameter.
if [ $# -ge $CMDLINEPARAM ]
then
  NAME=$1 # If more than one command line param,
  # then just take the first.
else
  NAME="John Doe" # Default, if no command line parameter.
fi
RESPONDENT="the author of this fine script"
cat <<Endofmessage
```

```
Hello, there, $NAME.

Greetings to you, $NAME, from $RESPONDENT.

# This comment shows up in the output (why?).

Endofmessage

# Note that the blank lines show up in the output.

# So does the "comment".


exit 0
```

Bir belge dokümanının başında "sınır dizesini" tırnak işareti içinde veya kaçış karakteriyle kullanmak parametre değiştirmeyi devre dışı bırakarak yararlılığı azaltır.

ÖRNEK 17.6 PARAMETRE DEĞİŞİMİ AKTİF DEĞİL

```
#!/bin/bash

# A 'cat' here document, but with parameter substitution disabled.

NAME="John Doe"

RESPONDENT="the author of this fine script"

cat <<'Endofmessage'

Hello, there, $NAME.

Greetings to you, $NAME, from $RESPONDENT.

Endofmessage

# No parameter substitution when the "limit string" is quoted or escaped.

# Either of the following at the head of the here document would have the
same

# effect.

# cat <<"Endofmessage"

# cat <<\Endofmessage

exit 0
```

Aşağıda parametre değiştirme ile belge dokümanını içeren faydalı bir komut dosyası gösterilmiştir.

ÖRNEK 17.7 upload : "Sunsite" DİZİNİNE BİR DOSYA ÇİFTİ YÜKLER

```
#!/bin/bash
# upload.sh
# Upload file pair (Filename.lsm, Filename.tar.gz)
# to incoming directory at Sunsite (metalab.unc.edu).
E_ARGERROR=65
if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_ARGERROR
fi
Filename=`basename $1` # Strips pathname out of file name.
Server="metalab.unc.edu"
Directory="/incoming/Linux"
# These need not be hard-coded into script,
# but may instead be changed to command line argument.
Password="your.e-mail.address" # Change above to suit.
ftp -n $Server <<End-Of-Session
# -n option disables auto-logon
user anonymous "$Password"
binary
bell     # Ring 'bell' after each file transfer
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
end-Of-Session
exit 0
```

Bir belge dokümanı, bir komut dosyası içindeki fonksiyona girdi sağlayabilir.

ÖRNEK 17.8 BELGE DOKÜMANLARI ve FONKSİYONLARI

```
#!/bin/bash
# here-function.sh
GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # This certainly looks like an interactive function, but...

# Supply input to the above function.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
MD
21226
RECORD001

echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo
exit 0
```

Bir belge dokümanından çıktı kabul edermiş gibi, doldurma komut kullanmak da mümkündür. Bu, yeni bir "anonim" belge doküman etkisi yaratır.

ÖRNEK 17.9 ANONİM BELGE DOKÜMANI

```
#!/bin/bash

: <<TESTVARIABLES

${HOSTNAME?}${USER?}${MAIL?} # Print error message if one of the variables
not set.

TESTVARIABLES


exit 0
```

Yukarıdaki teknikten yola çıkarak, kod bloklarının "yorum halinde örtülmesi"ni de sağlayabilirsiniz.

ÖRNEK 17.10 KOD BLOĞUNUN YORUM HALİNDE ÖRTÜLMESİ

```
#!/bin/bash

# commentblock.sh

: << COMMENTBLOCK

echo "This line will not echo."

This is a comment line missing the "#" prefix.

This is another comment line missing the "#" prefix.

&*@!!++=

The above line will cause no error message,

because the Bash interpreter will ignore it.

COMMENTBLOCK
```

```
echo "Exit value of above \"COMMENTBLOCK\" is $?."    # 0
# No error shown.

#  The above technique also comes in useful for commenting out
#+ a block of working code for debugging purposes.
#  This saves having to put a "#" at the beginning of each line,
#+ then having to go back and delete each "#" later.

: << DEBUGXXX
for file in *
do
  cat "$file
done
DEBUGXXX
exit 0
```

Bu teknik "kendini belgeleyen" komut dosyalarını mümkün kılar.

ÖRNEK 17.11 KENDİ KENDİNİ BELGELEYEN KOMUT DOSYASI

```
#!/bin/bash
# self-document.sh: self-documenting script
# Modification of "colm.sh".
DOC_REQUEST=70
if [ "$1" = "-h" -o "$1" = "--help" ] # Request help.
then
  echo; echo "Usage: $0 [directory-name]"; echo
  cat "$0" | sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATION/p' |
  sed -e '/DOCUMENTATIONXX/d'; exit $DOC_REQUEST; fi
```

```
: << DOCUMENTATIONXX
List the statistics of a specified directory in tabular format.
---------------------------------------------------------------
The command line parameter gives the directory to be listed.
If no directory specified or directory specified cannot be read,
then list the current working directory.

DOCUMENTATIONXX
if [ -z "$1" -o ! -r "$1" ]
then
  directory=.
else
  directory="$1"
fi
echo "Listing of "$directory":"; echo
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l "$directory" | sed 1d) | column –t
exit 0
```

Belge dokümanları geçici dosyalar oluştururlar, ancak bu dosyalar açıldıktan hemen sonra silinir ve başka bir işlem için erişilebilir değildir.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof 1213 bozo 0r REG 3,5 0 30386 /tmp/t1213-0-sh (deleted)
```

Bazı yardımcı programlar *belge dokümanları* içinde çalıştırılamaz.

"Belge dokümanları" için beklenebilecek çok karmaşık görevler için, etkileşimli programlara girdi beslemesi için özel olarak tasarlanmış olan **expect** dilini kullanmayı düşünebilirsiniz.

BÖLÜM 18 BİRAZ ARA VERELİM

Bu tuhaf küçük ara okuyucunun dinlenmesi ve belki biraz gülmesi için bir şans verecektir.

Arkadaşım Linux kullanıcısı, selam! Size şans ve iyilik getirecek bir şey okuyorsunuz. Bu blog'da Advanced Linux başlığı altında bulunan ve Türkçe'ye çevrili bilgi ve kod örneklerinden istediğiniz kadarını 10 kişiye postalamakta serbestsiniz. Bu bir arkadaş zinciri oluşturur. Zinciri kıran insanların başına kötü şeyler geldiği söylenir. İnanın ve siz de zinciri kırmayın! Size gelen zincir listesinin en sonuna yollayacağınız kişileri ekleyin ve zincirin büyümüş halini bırakın onlar büyütsünler.

Bugün on kopyayı göndermekte çekinmeyin. Tüm materyal öğreticidir, ve hiçbir sansüre gerek yoktur.

Yazar: Mendel Leo Cooper

KISIM 4 İLERİ DÜZEY KONULAR

Bölüm 19 KURALLI İFADELER

Kabuk betiklerinin gücünü tam olarak anlamak için, kurallı ifadeler konusunda uzman olmanız gerekir. Bazı komut ve betiklerde yaygın olarak kullanılan expr, sed ve awk gibi bazı yardımcı programlar kurallı ifadeleri yorumlamakta kullanılır.

19.1 KURALLI İFADELERE BAŞLANGIÇ

Karakter dizesi bir ifade olarak adlandırılır. Kendi kelime anlamının üstünde ve ötesinde bir yorumu bulunan karakterlere *metakarakter* denir. Örneğin bir tırnak işaret sembolü, , bir kişi tarafından konuşmasının aynısı, veya takip eden semboller için bir meta-anlam ifade edebilir. Kurallı ifadeler, UNIX'in parçası olan özelliklerle donatılmış birtakım özel karakter setleri ve/veya metakarakterlerdir. [1]

Kurallı ifadeler için ana kullanım alanı metin arama ve dize işleme sayılabilir. Bir kurallı ifade, tek bir karakter veya bir dizi karakter (bir altdize veya bütün bir dize) ile *eşleşir*.

Yıldız işareti `--*--`, karakter dizesinin veya *sıfır dahil* olmak üzere, önceki kurallı ifadenin herhangi bir sayıdaki tekrarı ile eşleşir.

- "1133*" kurallı ifadesi *`11 + bir veya daha fazla 3 + muhtemelen diğer karakterler ile`* eşleşir*: `113, 1133, 111312,`* vb.

- Nokta `--.--` satır sonu dışında, herhangi bir karakterle eşleşir. [2]
  "13." *13 + herhangi en az bir karakter (boşluk da dahil olmak üzere)* ile eşleşir: *1133, 11333* ile eşleşir, fakat *13* ile eşleşmez (ek karakteri eksiktir).

- `--^--`, satırbaşı ile eşleşir, ama bazen, değişen bağlamlara göre, kurallı ifadedeki karakter setinin anlamını ortadan kaldırmaktadır.

- Bir kurallı ifadenin sonundaki dolar işareti -- $ -- satır sonu ile eşleşir.

"^$" boş satırlar ile eşleşir.

- Parantez `--[...]--` bir karakter seti içindeki karakterleri tek bir kurallı ifade içinde eşleştirir.

  "[xyz]" *x, y* veya *z* karakterleri ile eşleşir.

  "[c-n]" *c-n* aralığında herhangi bir karakter(ler) ile eşleşir.

  "[B-Pb-y]" `B-P` ve `b-y` aralığında herhangi bir karakter(ler) ile eşleşir.

  "[a-z0-9]" Herhangi bir küçük harf veya herhangi bir rakamla eşleşir.

- "[^b-d]" *b-d* aralığı *dışındaki* tüm karakterler ile eşleşir. Takip eden kurallı ifadenin anlamını değil'e veya tersine çeviren bir `^` örneğidir (Benzer bir rol alarak farklı bir bağlamda `!` ile benzerliği vardır).

  Parantez içine alınmış karakterlerin sıralı dizileri ortak kelime kalıpları ile eşleşir. "[Ee][Vv][Ee][Tt]" evet, Evet, EVET, eVEt, ve benzeri ile eşleşir.

  "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" 9 basamağa kadar olan bir sayıyla eşleşir.

- Ters eğik çizgi `--\--` özel bir karakterin kaçışı ve karakterin kelime anlamıyla yorumlanacağı anlamına gelir.

  Bir kurallı ifadedeki "\$" satır-sonu anlamının yerine, kendi kelime anlamı olan "$"a geri döner. Benzer şekilde bir "\\" için "\" kelime anlamı vardır.

- Kaçış ile birlikte ifade edilen "köşeli parantez" `--\<...\>--` ile işaretlenerek kelime sınırını çizebilir.

  Kaçış karakteri kullanılmadığı takdirde, parantez içindeki harf dizimi tam kelime anlamına sahip olur.

  "\<onlar\>" kelime anlamı "onlar" ile eşleşir, ama "onlarla", "onlarsız", "var", "öteki", vb. ile eşleşmez.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.
 This is line 4.

bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.

bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

- **Genişletilmiş Kurallı İfadeler**. egrep, awk ve perl'de kullanılır.

- Soru işareti `--?--` öncesindeki bir kurallı ifade ile veya hiçbir şey ile eşleşir. Bu, genellikle tek bir karakter eşleşmesi için kullanılır.

- Artı `--+--` önceki bir ya da daha çok kurallı ifade ile eşleşir. Bu * benzeri bir rol oynar, ancak oluşma sayısı mevcut değilse, eşleştirme *yapmaz*.

```
# GNU versions of sed and awk can use "+",

# but it needs to be escaped.

echo a111b | sed -ne '/a1\+b/p'

echo a111b | grep 'a1\+b'

echo a111b | gawk '/a1+b/'

# All of above are equivalent.

# Thanks, S.C.
```

- "Kıvrık parantez", `--\{\}--` ile işaretlenirse, eşleşme yapılması istenen önceki kurallı ifadelerin oluşma sayısını gösterir.

  Kıvrık parantezlerin kaçma karakteriyle kullanımı gereklidir; çünkü aksi takdirde sadece tam kelime anlamına sahiptir. Bu kullanım teknik olarak, kurallı ifade setinin temel parçası değildir.

  "[0-9]\{5\}" tam beş basamaklı (0 ile 9 aralığında karakter) rakamlar ile eşleşir.

  Kıvrık parantezi awk "klasik" sürümünde bir kurallı ifade olarak kullanılamaz. Fakat, **gawk** için (bir kaçma karakteri olmadan) onlara izin veren `--re-interval` seçeneği vardır.

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

- Parantezler --()-- kurallı ifadeleri gruplar içine alır. Bunlar expr kullanarak altdize çıkarması için ve aşağıdaki "|" operatörü ile birlikte kullanıldığında yararlıdır.

- --|-- "ya da" kurallı ifade operatörü herhangi bir alternatif karakter seti ile eşleşir.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

POSIX Karakter Sınıfları. `[:class:]`

Bu eşleştirme için karakter aralığı belirlemenin alternatif bir yöntemidir.

- **`[:alnum:]`** alfabetik veya sayısal karakterlerle eşleşir. Bu **`[A-Za-z0-9]`** ile eşdeğerdir.
- **`[:alpha:]`** alfabetik karakterlerle eşleşir. Bu **`[A-Za-z]`** ile eşdeğerdir.
- **`[:blank:]`** bir boşluk ya da sekme ile eşleşir.
- **`[:cntrl:]`** kontrol karakterleri ile eşleşir.
- **`[:digit:]`** (ondalık) basamaklar ile eşleşir. Bu **`[0-9]`** ile eşdeğerdir.
- **`[:graph:]`** (grafik yazdırılabilir karakterler). 33 - 126 ASCII aralığındaki karakterler ile eşleşir.
  Bu boşluk karakteri hariç olmak üzere aşağıdaki **`[: print:]`** ile aynıdır.
- **`[:lower:]`** Küçük harfli alfabetik karakterler ile eşleşir. Bu **`[a-z]`** ile eşdeğerdir.
- **`[:print:]`** (Yazdırılabilir karakterler). 32 - 126 ASCII aralığındaki karakterler ile eşleşir. Bu yukarıdaki **`[:graph:]`** ile aynıdır, ancak boşluk karakterini de dahil eder.
- **`[:space:]`** boşluk karakterleri ile (boşluk ve yatay sekme) eşleşir.
- **`[:upper:]`** büyük harfli alfabetik karakterler ile eşleşir. Bu **`[A-Z]`** ile eşdeğerdir.
- **`[:xdigit:]`** on altılık tabandaki basamaklar ile eşleşir. Bu `[0-9A-Fa-f]` ile eşdeğerdir.

POSIX karakter sınıfları genellikle tırnak işareti veya çift parantez gerektirir ([[ ]]).

```
bash$ grep [[:digit:]] test.file
abc=723
```

Bu karakter sınıfları kalıp eşleştirme (globbing) ile de kullanılabilir.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r-- 1 bozo bozo 0 Aug 21 14:47 a33b
```

Komut dosyalarında kullanılan POSIX karakter sınıfları görmek için, bkz. Örnek 12.14 ve Örnek 12.15.

Komut dosyalarında filtre olarak kullanılan, sed, awk ve Perl, dosyaları ve G/Ç akışlarını dönüştürürken veya "gözden" geçirirken kurallı ifadeleri argüman olarak alır. Bunun örnekleri için bkz. Örnek A.12 ve Örnek A.17.

Dougherty ve Robbins tarafından yazılan "Sed & Awk", kurallı ifadelerin bir çok tam ve anlaşılır kullanımını vermektedir (bkz. Kaynakça).

Notlar

[1] En basit kurallı ifade, metakarakter içermeyen ve tam anlamıyla kelime anlamını koruyan bir karakter dizesidir.

[2] Sed, awk ve grep tek satırları işlediği için genellikle yeni satır karakteriyle eşleşmez. Çok-satırlı bir ifadede yeni satır karakterinin olduğu durumlarda, nokta yeni satırla eşleşir.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF # Here Document
line1
line2
EOF
# OUTPUT:
# [line1
# line2]

echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# OUTPUT:
# line
# 1

# Thanks, S.C.
exit 0
```

19.2 KALIP EŞLEŞTİRMELERİ

Bash kendisi kurallı ifadeleri tanıyamaz. Betiklerde, sed ve awk gibi komutlar ve yardımcı programlar kurallı ifadeleri yorumlar.

Bash standart kurallı ifade kümesini kullanarak "kalıp eşleştirmeleri" olarak bilinen genişletmeyi dosya ismi için *açıp büyütemezsiniz*. Bunun yerine, kalıp eşleştirmeleri joker karakterleri tanır ve genişletir. Kalıp eşleştirmeleri standart joker karakterleri, `*` ve `?`, köşeli parantez içindeki karakter listelerini ve diğer bazı özel karakterleri (`^` gibi bir eşleşmeyi tersine çevirmek için) yorumlar. Kalıp eşleştirmede kullanılan joker karakterler için bazı önemli sınırlamalar da vardır. `*` içeren dizeler bir nokta ile başlayan dosyalar ile eşleşmeyecektir, örneğin `.bashrc`'de olduğu gibi. [1] Aynı şekilde,`?` kalıp eşleştirme için bir kurallı ifadenin parçası olduğundan daha farklı bir anlam teşkil eder.

```
bash$ ls –l

total 2

-rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1

 -rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh

 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt


bash$ ls -l t?.sh

-rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh


bash$ ls -l [ab]*

-rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1


bash$ ls -l [a-c]*

-rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1
```

```
bash$ ls -l [^ab]*

-rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1

 -rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh

 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt


bash$ ls -l {b*,c*,*est*}

-rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1

 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1

 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt


bash$ echo *

a.1 b.1 c.1 t2.sh test1.txt


bash$ echo t*

t2.sh test1.txt
```

echo komutu bile dosya isimleri üzerinde jokerleri açıp genişletme özelliğine sahiptir. Bkz. Örnek 10.4.

Notlar

[1] Nokta ile başlayan dosyaları eşleştirmek için dosya ismi genişletilmesi kullanılabilir, ancak bunun için nokta kalıpta açık bir şekilde belirtilmelidir.

```
~/[.]bashrc        # Will not expand to ~/.bashrc

~/?bashrc          # Neither will this.

# Wild cards and metacharacters will not expand to a dot in globbing.

~/.[b]ashrc        # Will expand to ~./bashrc

~/.ba?hrc          # Likewise.

~/.bashr*          # Likewise.

# Setting the "dotglob" option turns this off.

# Thanks, S.C.
```

Bölüm 20 ALT KABUKLAR

Bir kabuk betiğini çalıştırmak, komut işlemcisinin bir başka örneğini başlatır. Komutlarınızın komut satırı isteminde yorumlanmasına benzer şekilde, bir toplu betik dosyasındaki komut listesi işleme konur. Çalışan her kabuk betiği, aslında, ana kabuğun konsolda veya xterm penceresinde size istemini veren bir alt işlemidir.

Bir kabuk betiği alt işlemler de başlatabilir. Bu *alt kabuklar* betiğin paralel işlem yapmasına ve yürürlükte aynı anda birden fazla alt görev yürütmesine izin verir.

Parantez İçindeki Komut Listesi

(komut1; komut2; komut3; ...)

`Parantez` arasında gömülü bir komut listesi bir alt kabuk olarak çalışır.

Bir alt kabuktaki değişkenler alt kabuk kod bloğunun *dışında* görünmez. Onlar alt kabuğu başlatan kabuk ve ana süreç için erişilebilir değildir. Bunlar, aslında, yerel değişkenlerdir.

ÖRNEK 20.1 BİR ALT KABUKTAKİ DEĞİŞKEN KAPSAMI

```
#!/bin/bash
# subshell.sh
echo
outer_variable=Outer
(
inner_variable=Inner
echo "From subshell, \"inner_variable\" = $inner_variable"
```

```
echo "From subshell, \"outer\" = $outer_variable"

)

echo


if [ -z "$inner_variable" ]

then

  echo "inner_variable undefined in main body of shell"

else

  echo "inner_variable defined in main body of shell"

fi


echo "From main body of shell, \"inner_variable\" = $inner_variable"

# $inner_variable will show as uninitialized because

# variables defined in a subshell are "local variables".

echo

exit 0
```

Bkz.Örnek 32-1.

+

Bir alt kabukta yapılan dizin değişiklikleri ana kabuk üzerine taşınmaz.

ÖRNEK 20.2 KULLANICI PROFİLLERİ LİSTESİ

```
#!/bin/bash

# allprofs.sh: print all user profiles

# This script written by Heiner Steven, and modified by the document
author.

FILE=.bashrc    # File containing user profile,
```

```
                #+ was ".profile" in original script.


for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue   # If no home directory, go to next.
  [ -r "$home" ] || continue   # If not readable, go to next.
  (cd $home; [ -e $FILE ] && less $FILE)
done


#  When script terminates, there is no need to 'cd' back to original
directory,
#+ because 'cd $home' takes place in a subshell.
exit 0
```

Bir alt kabuk, bir komut grubu için "özel ortam" kurmak için kullanılabilir.

```
COMMAND1
COMMAND2
COMMAND3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set –C
  shift 5
  COMMAND4
  COMMAND5
  exit 3    # Only exits the subshell.
)
# The parent shell has not been affected, and the environment is preserved.
COMMAND6
COMMAND7
```

Bunun bir uygulaması, bir değişkenin tanımlı olup olmadığını test etmektir.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "Variable is set."
fi
# Could also be written [[ ${variable-x} != x || ${variable-y} != y ]]
# or                    [[ ${variable-x} != x$variable ]]
# or                    [[ ${variable+x} = x ]])
```

Bir başka uygulama, kilit dosyasını kontrol etmektir:

```
if (set -C; : > lock_file) 2> /dev/null
then
  echo "Another user is already running that script."
  exit 65
fi

# Thanks, S.C.
```

Süreçler farklı alt kabuklarda paralel olarak çalışabilir. Bu, karmaşık bir görevin eş zamanlı olarak işlenen alt bileşenler içine bölünmesine olanak verir.

ÖRNEK 20.3 ALT KABUKLARDA PARALEL SÜREÇLERİN ÇALIŞMASI

```
(cat list1 list2 list3 | sort | uniq > list123) &
(cat list4 list5 list6 | sort | uniq > list456) &
# Merges and sorts both sets of lists simultaneously.
# Running in background ensures parallel execution.
```

```
#
# Same effect as
# cat list1 list2 list3 | sort | uniq > list123 &
# cat list4 list5 list6 | sort | uniq > list456 &
wait         # Don't execute the next command until subshells finish.
diff list123 list456
```

Bir alt kabuğa G/Ç yönlendirmesi yapılması için "|" (oluk) operatörü kullanılmalıdır, **ls -al | (komut)** örneğinde olduğu gibi.

Kıvrık parantezler arasındaki bir komut bloğu alt kabuğu *başlatmaz*.

{ komut1; komut2; komut3; ... }

Bölüm 21 KISITLI KABUKLAR

Kısıtlı kabuklarda engellenmiş komutlar

Bir betiğin veya betiğin bir bölümünün *kısıtlı* modda çalışması, normalde kullanılabilir olan belirli komutları devre dışı bırakır. Bu, betik kullanıcısının ayrıcalıklarını sınırlamak ve komut dosyasının çalışmasından doğan olası hasarı en aza indirmek için tasarlanmış bir güvenlik önlemidir.

Çalışma dizinini değiştirmek için `cd` kullanılması.

`$PATH, $SHELL, $BASH_ENV` veya `$ENV` çevresel değişkenlerin değerlerini değiştirme.

`$SHELLOPTS` kabuk çevre seçeneklerini okuma veya değiştirme.

Çıktı yönlendirme.

Bir veya daha fazla `/` içeren komutları çağırma.

Kabuk ile farklı bir sürecin yer değiştirmesi için *exec* çağrılması.

İstenmeyen bir amaç için komut dosyasını yıkmaya çalışacak çeşitli komutlar.

Komut dosyası içindeki kısıtlı moddan çıkmak.

ÖRNEK 21.1 KISITLI MODDA BİR KOMUT DOSYASINI ÇALIŞTIRMAK

```
#!/bin/bash
# Starting the script with "#!/bin/bash -r"
# runs entire script in restricted mode.
echo
echo "Changing directory."
cd /usr/local
echo "Now in `pwd`"
```

```
echo "Coming back home."
cd
echo "Now in `pwd`"
echo
# Everything up to here in normal, unrestricted mode.
set –r
# set --restricted has same effect.
echo "==> Now in restricted mode. <=="
echo
echo
echo "Attempting directory change in restricted mode."
cd ..
echo "Still in `pwd`"
echo
echo
echo "\$SHELL = $SHELL"
echo "Attempting to change shell in restricted mode."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"
echo
echo
echo "Attempting to redirect output in restricted mode."
ls -l /usr/bin > bin.files
ls -l bin.files     # Try to list attempted file creation effort.
echo
exit 0
```

Bölüm 22 İŞLEM ORNATIMI

İşlem ornatımı komut ornatımının karşılığıdır. Komut ormatımı, bir komutun sonucunu bir değişkene ayarlar, **dizin_içeriği='ls -al'** veya **xref=$(grep sözcük veridosyası)** örneklerinde olduğu gibi. İşlem ornatımı, bir işlemin çıktısını başka bir işlem için besler (diğer bir deyişle, bir komutun sonuçlarını başka bir komuta gönderir).

Komut ornatımı şablonu

parantez içindeki komut

>(komut)

<(komut)

Bunlar, işlem ornatımını başlatır. Parantez içinde sürecin sonuçları başka bir işleme gönderilmek üzere `/dev/fd/<n>` dosyalarını kullanır. [1]

"<" veya ">" ve parantez arasında boşluk yoktur. Oradaki herhangi bir boşluk hata mesajı verecektir.

```
bash$ echo >(true)
/dev/fd/63
bash$ echo <(true)
/dev/fd/63
```

Bash, iki dosya tanımlayıcısı olan bir oluk oluşturur, `--fIn` ve `fOut--`. true'nun `stdin`'i `fOut`'a bağlanır, `(dup2(fOut,0))`, sonra Bash **echo** komutuna bir `/dev/fd/fIn` argümanı geçer. `/dev/fd/<n>` dosyaları eksik olan sistemlerde, Bash geçici dosyaları kullanabilir. (Teşekkürler, S.C.)

```
cat <(ls -l)

# Same as      ls -l | cat

catsort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)

# Lists all the files in the 3 main 'bin' directories, and sorts by
filename.

# Note that three (count 'em) distinct commands are fed to 'sort'.


diff <(command1) <(command2)      # Gives difference in command output.

tar cf >(bzip2 -c > file.tar.bz2) $directory_name

# Calls "tar cf /dev/fd/?? $directory_name", and "bzip2 -c > file.tar.bz2".

#

# Because of the /dev/fd/<n> system feature,

# the pipe between both commands does not need to be named.

#

# This can be emulated.

#

bzip2 -c < pipe > file.tar.bz2&

tar cf pipe $directory_name

rm pipe

#        or

exec 3>&1

tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2
3>&-

exec 3>&-

# Thanks, S.C.
```

Bu belgenin bir okuyucusu, işlem ornatımı ile ilgili şu ilginç örneği gönderdi.

```
# Script fragment taken from SuSE distribution:

while read des what mask iface; do
```

```
# Some commands ...

done < <(route -n)


# To test it, let's make it do something.

while read des what mask iface; do

  echo $des $what $mask $iface

done < <(route -n)


# Output:

# Kernel IP routing table

# Destination Gateway Genmask Flags Metric Ref Use Iface

# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo

# As S.C. points out, an easier-to-understand equivalent is:


route -n |

  while read des what mask iface; do     # Variables set from output of
pipe.

  echo $des $what $mask $iface

done     # Same output as above.
```

Notlar

[1] Bu, bir adlandırılmış oluk (geçici dosya) ile aynı etkiye sahiptir, ve gerçekten, adlandırılmış oluklar bir zamanlar işlem ornatımı için kullanılıyordu.

BÖLÜM 23 FONKSİYONLAR

"Gerçek" programlama dilleri gibi, Bash'in biraz sınırlı bir uygulama da olsa, fonksiyonları vardır. Fonksiyon bir alt programdır, bir dizi operasyonu uygulayan bir kod bloğudur, belirli bir görevi yerine getiren bir "kara kutu"dur. Nerede tekrarlayan kod varsa, ne zaman bir görev sadece küçük farklılıklar ile tekrarlıyorsa, o zaman bir fonksiyon kullanmayı düşünün.

function *fonksiyon_adı*

```
{
komut...
}
```

veya

```
fonksiyon_adı() {
komut...
}
```

Bu ikinci şekli C programcıları için daha tanıdıktır (ve daha taşınabilirdir).

C'de olduğu gibi, fonksiyonun açan parantezi isteğe bağlı olarak ikinci satırda görünebilir.

```
fonksiyon_adı()
{
komut...
}
```

Fonksiyonlar sadece isimleri çağrılarak *tetiklenir*.

ÖRNEK 23.1 BASİT BİR FONKSİYON

```
#!/bin/bash
funky ()
{
  echo "This is a funky function."
  echo "Now exiting funky function."
}  # Function declaration must precede call.
   # Now, call the function.
funky
exit 0
```

Fonksiyonun tanımı, ona yapılan ilk çağrıdan önce gelmelidir. C'deki gibi örneğin, fonksiyon "bildirimi" için hiçbir yöntem yoktur.

```
# f1
# Will give an error message, since function "f1" not yet defined.
# However...
f1()
{
  echo "Calling function \"f2\" from within function \"f1\"."
  f2
}
f2()
{
  echo "Function \"f2\"."
}
f1     # Function "f2" is not actually called until this point,
       # although it is referenced before its definition.
       # This is permissable
# Thanks, S.C.
```

Çok yararlı olmasa da, bir fonksiyon içinde (iç içe) bir fonksiyon yazılması da mümkündür.

```
f1()
{
  f2 () # nested
  {
    echo "Function \"f2\", inside \"f1\"."
  }
}

# f2
# Gives an error message.
f1   # Does nothing, since calling "f1" does not automatically call "f2".
f2   # Now, it's all right to call "f2",
     # since its definition has been made visible by calling "f1".
# Thanks, S.C.
```

Fonksiyon tanımlamaları, en olası olmayan yerlerde görünebilir, mesela bir komutun yazılacağı yerde.

```
ls -l | foo() { echo "foo"; } # Permissable, but useless.
if [ "$USER" = bozo ]
then
  bozo_greet ()   # Function definition embedded in an if/then construct.
  {
    echo "Hello, Bozo."
  }
fi
bozo_greet   # Works only for Bozo, and other users get an error.
```

```
# Something like this might be useful in some contexts.

NO_EXIT=1    # Will enable function definition below.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }    # Function definition in an
"and-list".

# If $NO_EXIT is 1, declares "exit ()".

# This disables the "exit" builtin by aliasing it to "true".

exit    # Invokes "exit ()" function, not "exit" builtin.

# Thanks, S.C.
```

23.1 KARMAŞIK FONKSİYONLAR ve FONKSİYONLARIN GÜÇLÜKLERİ

Fonksiyonlar kendilerine geçirilen argümanları işler ve daha fazla işlem için komut dosyasına bir çıkış durumu döndürebilir.

```
fonksiyon_adı $argüman1 $argüman2
```

Fonksiyona geçirilen argümanlar, `$1`, `$2` ve benzeri, konuma göre ifade edilir, (sanki konumsal parametrelermiş gibi).

ÖRNEK 23.2 PARAMETRELERİ KABUL EDEN FONKSİYON

```
#!/bin/bash

func2 () {

  if [ -z "$1" ]           # Checks if parameter #1 is zero length.

  then

    echo "-Parameter #1 is zero length.-" # Also if no parameter is passed.
```

```
  else
    echo "-Param #1 is \"$1\".-"
  fi

  if [ "$2" ]
  then
    echo "-Parameter   #2 is \"$2\".-"
  fi

  return 0
}

echo

echo "Nothing passed."

func2                  # Called with no params
echo

echo "Zero-length parameter passed."
func2 ""               # Called with zero-length param
echo

echo "Null parameter passed."
func2 "$uninitialized_param"      # Called with uninitialized param
echo

echo "One parameter passed."
func2 first            # Called with one param
echo
```

```
echo "Two parameters passed."
func2 first second     # Called with two params
echo
echo "\"\" \"second\" passed."
func2 "" second        # Called with zero-length first parameter
echo                   # and ASCII string as a second one.

exit 0
```

shift komutu fonksiyonlara geçirilen argümanlar üzerinde çalışır (bkz.Örnek 34.6).

Bazı diğer programlama dillerinin aksine, kabuk betikleri normalde sadece değer-parametrelerini fonksiyonlara geçirirler. [1] Değişken adları (aslında işaretçi olan), fonksiyonlara parametre olarak aktarılırsa, dizgi kalıp deyim olarak kabul edilecektir ve geri-referans edilemeyecektir. *Fonksiyonlar argümanlarını kalıp deyim olarak yorumlar.*

Exit ve Return

çıkış durumu

Fonksiyonlar çıkış durumu adı verilen bir değeri geri döndürür. Çıkış durumu açıkça bir **return** deyimi ile belirtilebilir, aksi takdirde fonksiyondaki son komutun çıkış durumudur (0 başarılı olursa, ve sıfır olmayan bir hata kodu, değilse). Bu çıkış durumu, $? ile başvuru yapılarak komut dosyası içinde kullanılabilir. Bu mekanizma, etkili bir şekilde, betik fonksiyonlarının C fonksiyonlarına benzer bir "dönüş değeri"ne sahip olmasına olanak tanır.

return

Bir fonksiyonu sonlandırır. Bir **return** komutu [2] isteğe bağlı olarak, çağıran betiğe fonksiyonun çıkış durumu olarak döndürülen bir *tamsayı* argümanı alır, ve bu çıkış durumu $? değişkenine atanır.

ÖRNEK 23.3 İKİ SAYININ MAKSİMUMU

```
#!/bin/bash

# max.sh: Maximum of two integers.

E_PARAM_ERR=-198    # If less than 2 params passed to function.

EQUAL=-199          # Return value if both params equal.

max2 ()             # Returns larger of two numbers.

{                   # Note: numbers compared must be less than 257.

if [ -z "$2" ]

then

  return $E_PARAM_ERR

fi


if [ "$1" -eq "$2" ]

then

  return $EQUAL

else

  if [ "$1" -gt "$2" ]

  then

    return $1

  else

    return $2

  fi

fi
```

```
}


max2 33 34
return_val=$?
if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "Need to pass two parameters to the function."
elif [ "$return_val" -eq $EQUAL ]
then
  echo "The two numbers are equal."
else
  echo "The larger of the two numbers is $return_val."
fi


exit 0


#  Exercise (easy):
#  ----------------
#  Convert this to an interactive script,
#+ that is, have the script ask for input (two numbers).
```

Bir fonksiyonun dizgi ya da dizilim döndürmesi için özelleştirilmiş bir değişken kullanılmalıdır.

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  # If /etc/passwd is readable, set REPLY to line count.
  # Returns both a parameter value and status information.
}
```

```
if count_lines_in_etc_passwd
then
  echo "There are $REPLY lines in /etc/passwd."
else
  echo "Cannot count lines in /etc/passwd."
fi
# Thanks, S.C.
```

ÖRNEK 23.4 SAYILARIN ROMA RAKAMLARINA DÖNÜŞTÜRÜLMESİ

```
#!/bin/bash
# Arabic number to Roman numeral conversion
# Range: 0 – 200
# It's crude, but it works.
# Extending the range and otherwise improving the script is left as an
#+ exercise.
# Usage: roman number-to-convert
LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66
if [ -z "$1" ]
then
  echo "Usage: `basename $0` number-to-convert"
  exit $E_ARG_ERR
fi
num=$1
if [ "$num" -gt $LIMIT ]
then
  echo "Out of range!"
```

```
  exit $E_OUT_OF_RANGE
fi

to_roman ()     # Must declare function before first call to it.
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ "$remainder" -ge 0 ]
do

  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number
        # Exercise:
        # --------
        # Explain how this function works.
        # Hint: division by successive subtraction.
}

to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
```

```
to_roman $num 40 XL

num=$?

to_roman $num 10 X

num=$?

to_roman $num 9 IX

num=$?

to_roman $num 5 V

num=$?

to_roman $num 4 IV

num=$?

to_roman $num 1 I

echo

exit 0
```

Ayrıca bkz. Örnek 10.27.

Bir fonksiyonun döndürebileceği en büyük pozitif tam sayı 256'dır. Bu özel sınırlamayı oluşturan **return** komutu çıkış durumu kavramıyla yakından bağlantılıdır. Neyse ki, bir fonksiyonun büyük bir tamsayı dönüş değeri gerektiren durumları için çeşitli geçici çözümler vardır.

ÖRNEK 23.5 BİR FONKSİYONDA BÜYÜK DÖNÜŞ DEĞERLERİNİN TEST EDİLMESİ

```
#!/bin/bash

# return-test.sh

# The largest positive value a function can return is 256.

return_test ()         # Returns whatever passed to it.

{

  return $1

}
```

```
return_test 27          # o.k.
echo $?                 # Returns 27.

return_test 256         # Still o.k.
echo $?                 # Returns 256.

return_test 257         # Error!
echo $?                 # Returns 1 (return code for miscellaneous error).

return_test -151896     # However, large negative numbers work.
echo $?                 # Returns -151896.

exit 0
```

Görüldüğü gibi, bir fonksiyon büyük bir negatif değer döndürebilir. Biraz hile kullanarak, büyük pozitif tamsayı döndürmesine de izin verilebilir.

Bunu sağlamanın alternatif bir yöntemi "dönüş değeri" ni sadece global bir değişkene atamaktır.

```
Return_Val= # Global variable to hold oversize return value of function.
alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return            # Returns 0 (success).
}

alt_return_test 1
echo $?                                 # 0
```

```
echo "return value = $Return_Val"    # 1

alt_return_test 256
echo "return value = $Return_Val"    # 256

alt_return_test 257
echo "return value = $Return_Val"    # 257

alt_return_test 25701
echo "return value = $Return_Val"    #25701
```

ÖRNEK 23.6 İKİ BÜYÜK TAMSAYININ KARŞILAŞTIRILMASI

```
#!/bin/bash
# max2.sh: Maximum of two LARGE integers.
# This is the previous "max.sh" example,
# modified to permit comparing large integers.

EQUAL=0                # Return value if both params equal.
MAXRETVAL=256          # Maximum positive return value from a function.
E_PARAM_ERR=-99999     # Parameter error.
E_NPARAM_ERR=99999     # "Normalized" parameter error.

max2 ()                # Returns larger of two numbers.
{
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi
if [ "$1" -eq "$2" ]
```

```
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    retval=$1
  else
    retval=$2
  fi
fi
# ------------------------------------------------------------ #
# This is a workaround to enable returning a large integer
# from this function.
if [ "$retval" -gt "$MAXRETVAL" ]           # If out of range,
then                                        # then
  let "retval = (( 0 - $retval ))"          # adjust to a negative value.
  # (( 0 - $VALUE )) changes the sign of VALUE.
fi
# Large *negative* return values permitted, fortunately.
# ------------------------------------------------------------ #
return $retval
}
max2 33001 33997
return_val=$?
# ------------------------------------------------------------------------#
if [ "$return_val" -lt 0 ]                  # If "adjusted" negative
number,
then                                        # then
  let "return_val = (( 0 - $return_val ))"  # renormalize to positive
fi                                        # "Absolute value" of $return_val.
# ------------------------------------------------------------------------#
```

```
if [ "$return_val" -eq "$E_NPARAM_ERR" ]
then                    # Parameter error "flag" gets sign changed, too.
  echo "Error: Too few parameters."
elif [ "$return_val" -eq "$EQUAL" ]
  then
    echo "The two numbers are equal."
else
    echo "The larger of the two numbers is $return_val."
fi
exit 0
```

Ayrıca bkz. Örnek A.8.

Alıştırma: Öğrendiklerimizin yardımıyla, önceki Roma rakamları örneğini rasgele büyük bir girdi kabul edecek şekilde geliştiriniz.

Yeniden yönlendirme

Bir fonksiyonun stdin'ini yönlendirme

Bir fonksiyon aslında `stdin`'i yönlendirilebilir bir kod bloğudur (bkz. Örnek 4.1).

ÖRNEK 23.7 KULLANICI ADINDAN GERÇEK ADIN BULUNMASI

```
#!/bin/bash
# From username, gets "real name" from /etc/passwd.
ARGCOUNT=1                          # Expect one arg.
E_WRONGARGS=65
```

```
file=/etc/passwd

pattern=$1

if [ $# -ne "$ARGCOUNT" ]

then

  echo "Usage: `basename $0` USERNAME"

  exit $E_WRONGARGS

fi

file_excerpt ()     # Scan file for pattern, the print relevant portion of
line.

{

while read line     # while does not necessarily need "[ condition]"

do

  echo "$line" | grep $1 | awk -F":" '{ print $5 }'   # Have awk use ":"
delimiter.

done

} <$file            # Redirect into function's stdin.


file_excerpt $pattern

# Yes, this entire script could be reduced to

#       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'

# or

#       awk -F: '/PATTERN/ {print $5}'

# or

#       awk -F: '($1 == "username") { print $5 }' # real name from username

# However, it might not be as instructive.

exit 0
```

Bir fonksiyonun `stdin`'ini yönlendirmenin alternatif, ve belki daha az kafa karıştırıcı bir yöntemi daha vardır. Bunun için `stdin`, fonksiyon içine gömülü bir parantez kod bloğu içine yönlendirilir.

```
# Instead of:
Function ()
{
 ...
 } < file

# Try this:
Function ()
{
  {
   ...
  } < file
}

# Similarly,
Function ()      # This works.
{
  {
    echo $*
  } | tr a b
}

Function ()      # This doesn't work.
{
  echo $*
} | tr a b       # A nested code block is mandatory here.

# Thanks, S.C.
```

Notlar

[1] Dolaylı değişken referansları (bkz. Örnek 35.2) değişken işaretçilerini fonksiyonlara geçirmenin bir mekanizmasıdır.

```
#!/bin/bash

ITERATIONS=3    # How many times to get input.

icount=1

my_read () {
  # Called with my_read varname,
  # outputs the previous value between brackets as the default value,
  # then asks for a new value.
  local local_var
  echo -n "Enter a value "
  eval 'echo -n "[$'$1'] "'     # Previous value.
  read local_var
  [ -n "$local_var" ] && eval $1=\$local_var
  # "And-list": if "local_var" then set "$1" to its value.
}
echo


while [ "$icount" -le "$ITERATIONS" ]
do
  my_read var
  echo "Entry #$icount = $var"
  let "icount += 1"
  echo
done
# Thanks to Stephane Chazelas for providing this instructive example.

exit 0
```

[2] **return** komutu bir Bash yerleşiğidir.

23.2 YEREL DEĞİŞKENLER

Bir değişkeni "yerel" yapan nedir?

yerel değişkenler

Yerel olarak bildirilen bir değişken sadece bulunduğu kod bloğu içinde görünür olan bir değişkendir. Yerel "kapsamı" vardır. Bir fonksiyon içindeki yerel değişkenin sadece o fonksiyon bloğu içinde bir anlamı vardır.

ÖRNEK 23.8 YEREL DEĞİŞKEN GÖRÜNÜRLÜĞÜ

```
#!/bin/bash
func () {
  local loc_var=23 # Declared local.
  echo
  echo "\"loc_var\" in function = $loc_var"
  global_var=999 # Not declared local.
  echo "\"global_var\" in function = $global_var"
}
func
# Now, see if local 'a' exists outside function.
echo
echo "\"loc_var\" outside function = $loc_var"
                                    # "loc_var" outside function =
                                    # Nope, $loc_var not visible globally.
echo "\"global_var\" outside function = $global_var"
                                    # "global_var" outside function = 999
                                    # $global_var is visible globally.
echo
exit 0
```

Bir fonksiyon çağrılmadan önce, fonksiyonun içinde bildirilen *tüm* değişkenler, sadece *yerel* olarak bildirilenler değil, fonksiyonun kapsamı dışında görünmezdir.

```
#!/bin/bash

func ()
{
  global_var=37 # Visible only within the function block
                #+ before the function has been called.
                # END OF FUNCTION
}
echo "global_var = $global_var" # global_var =
                # Function "func" has not yet been called,
                #+ so $global_var is not visible here.
func
echo "global_var = $global_var" # global_var = 37
                # Has been set by function call.
```

23.2.1 YEREL DEĞİŞKENLER ÖZYİNELEMEYİ MÜMKÜN KILAR.

Yerel değişkenler özyinelemeye izin verir, [1] ancak bu uygulama genellikle çok işlem yükü içerir ve bir kabuk betiği içinde yapılması kesinlikle tavsiye *edilmez*. [2]

ÖRNEK 23.9 YEREL BİR DEĞİŞKEN KULLANARAK ÖZYİNELEME

```
#!/bin/bash
#              factorial
#              ---------
# Does bash permit recursion?
# Well, yes, but...
# You gotta have rocks in your head to try it.
```

```
MAX_ARG=5
E_WRONG_ARGS=65
E_RANGE_ERR=66

if [ -z "$1" ]
then
  echo "Usage: `basename $0` number"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Out of range (5 is maximum)."
  # Let's get real now.
  # If you want greater range than this,
  # rewrite it in a real programming language.
  exit $E_RANGE_ERR
fi

fact()
{
  local number=$1
  # Variable "number" must be declared as local,
  # otherwise this doesn't work.
  if [ "$number" -eq 0 ]
  then
    factorial=1      # Factorial of 0 = 1.
  else
    let "decrnum = number - 1"
    fact $decrnum    # Recursive function call
```

```
    let "factorial = $number * $?"
  fi
  return $factorial
}

fact $1
echo "Factorial of $1 is $?."
exit 0
```

Bir betik içindeki özyinelemeye örnek için bkz. Örnek A.16. Unutmayınız ki, özyineleme çok kaynak tüketir ve yavaş çalışır, ve bu nedenle bir komut dosyası içinde kullanılması genelde uygun olmaz.

Notlar

[1] Herbert Mayer *özyinelemeyi* şöyle tanımlar: "…bir algoritmayı aynı algoritmanın daha basit bir sürümünü kullanarak ifade etmek…" Bir özyinelemeli fonksiyon kendisini çağıran bir fonksiyondur.

[2] Çok seviyeli özyineleme sırasında bir komut dosyası segfault ile çökebilir.

```
#!/bin/bash
recursive_function ()
{
(( $1 < $2 )) && f $(( $1 + 1 )) $2;
# As long as 1st parameter is less than 2nd,
#+ increment 1st and recurse.
}
recursive_function 1 50000 # Recurse 50,000 levels!
# Segfaults, of course.
# Recursion this deep might cause even a C program to segfault,
#+ by using up all the memory allotted to the stack.
# Thanks, S.C.
exit 0         # This script will not exit normally.
```

BÖLÜM 24 ÖTEKİ ADLAR

Bir Bash *öteki adı* aslında bir klavye kısayolundan başka bir şey değildir, bir kısaltma, uzun bir komut dizisini yazmaktan kaçınmak için bir araçtır. Eğer, örneğin, biz `~/.bashrc` dosyasına, **lm="ls -l | more" öteki adı**nı dahil edersek, daha sonra komut satırına yazdığınız her **`lm`** komutu otomatik olarak **ls -l | more** ile yer değiştirilecektir. Böylece komut satırında büyük bir yazma yükünden kurtulabilirsiniz, ve karmaşık komut ve seçenekler kombinasyonlarını hatırlamak zorunda kalmazsınız. **Öteki ad** olarak, **rm= "rm -i"** ayarı (etkileşimli mod silme) bir çok üzüntüden sizi kurtarabilir, çünkü yanlışlıkla önemli dosyaların silinerek kaybını önleyecektir.

Bir komut dosyasında, öteki adların yararları çok sınırlıdır. Öteki adlar, makro genişleme gibi C önişlemci işlevselliğinin bazılarını varsaysaydı, oldukça güzel olurdu, ama ne yazık ki Bash, öteki ad bünyesinde argümanları genişletme özelliğine sahip değildir. [1] Ayrıca, bir komut dosyası, if/then ifadesi, döngüler ve fonksiyonlar gibi "bileşik yapılar" içinde bir öteki adın kendisini genişletmekte başarısız olur. Buna ek bir sınırlama da, öteki adın özyinelemeli olarak genişletilemeyeceğidir. Hemen hemen her zaman, bir öteki addan yapmasını istediğimiz her şeyi, bir fonksiyon ile çok daha etkin ve başarılı bir şekilde yapabiliriz.

ÖRNEK 24.1 BİR KOMUT DOSYASINDA ÖTEKİ ADLAR

```
#!/bin/bash

# Invoke with command line parameter to exercise last section of this
script.

shopt -s expand_aliases

# Must set this option, else script will not expand aliases.


# First, some fun.

alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy starring
Bob Hope."'

Jesse_James

echo; echo; echo;

alias ll="ls -l"

# May use either single (') or double (") quotes to define an alias.
```

```
echo "Trying aliased \"ll\":"

ll /usr/X11R6/bin/mk*     #* Alias works.

echo


directory=/usr/X11R6/bin/

prefix=mk*     # See if wild-card causes problems.

echo "Variables \"directory\" + \"prefix\" = $directory$prefix"

echo

alias lll="ls -l $directory$prefix"

echo "Trying aliased \"lll\":"

lll            # Long listing of all files in /usr/X11R6/bin stating with
mk.

# Alias handles concatenated variables, including wild-card o.k.

TRUE=1

echo


if [ TRUE ]

then

  alias rr="ls -l"

  echo "Trying aliased \"rr\" within if/then statement:"

  rr /usr/X11R6/bin/mk* #* Error message results!

  # Aliases not expanded within compound statements.

  echo "However, previously expanded alias still recognized:"

  ll /usr/X11R6/bin/mk*

fi

echo

count=0


while [ $count -lt 3 ]

do

  alias rrr="ls -l"
```

```
  echo "Trying aliased \"rrr\" within \"while\" loop:"
  rrr /usr/X11R6/bin/mk* #* Alias will not expand here either.
  let count+=1
done
echo; echo
alias xyz="cat $1"        # Try a positional parameter in an alias.
xyz                       # Assumes you invoke the script
                          #+ with a filename as a parameter.
# This seems to work,
#+ although the Bash documentation suggests that it shouldn't.
#
# However, as Steve Jacobson points out,
#+ the "$1" parameter expands immediately upon declaration of the alias,
#+ so, in the strictest sense, this is not an example
#+ of parameterizing an alias.
exit 0
```

unalias komutu önceden ayarlanmış bir *öteki adı* kaldırır.

ÖRNEK 24.2 unalias: ÖTEKİ ADI AYARLAMA VE ORTADAN KALDIRMA

```
#!/bin/bash
shopt -s expand_aliases          # Enables alias expansion.

alias llm='ls -al | more'
llm
echo
unalias llm                      # Unset alias.
llm
# Error message results, since 'llm' no longer recognized.
exit 0
```

```
bash$ ./unalias.sh

total 6

drwxrwxr-x      2  bozo   bozo         3072  Feb 6 14:04  .

drwxr-xr-x     40  bozo   bozo         2048  Feb 6 14:04  ..

-rwxr-xr-x      1  bozo   bozo         199   Feb 6 14:04  unalias.sh


./unalias.sh: llm: command not found
```

Notlar

[1] Ancak görülmektedir ki, öteki adlar konumsal parametreleri genişletebilirler.

BÖLÜM 25 LİSTE YAPILARI

"ve-listesi" ve "veya-listesi" yapıları bir dizi komutu ardışık olarak işlemek için bir yol sağlar. Bu, karmaşık iç içe **if/then** ve hatta **case** cümlelerinin yerine etkin olarak geçebilir.

Komutların bir arada zincirlenmesi

ve-listesi

```
komut 1 && komut 2 && komut 3 && ... komut n
```

Her komut, önceki komut `true` dönüş değeri (sıfır) vermesi kaydıyla sırayla yürütülür. İlk `false` (sıfırdan farklı bir değer) döndüğünde, komut zinciri sonlanır (`false` dönen ilk komut, son çalıştırılan komut olur).

ÖRNEK 25.1 KOMUT SATIRI ARGÜMANLARINI TEST ETMEK İÇİN BİR "ve-listesi"nin KULLANILMASI

```
#!/bin/bash

# "and list"

if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] && echo
"Argument #2 = $2"

then

  echo "At least 2 arguments passed to script."

  # All the chained commands return true.

else

  echo "Less than 2 arguments passed to script."

  # At least one of the chained commands returns false.

fi
```

```
# Note that "if [ ! -z $1 ]" works, but its supposed equivalent,
# if [ -n $1 ] does not. However, quoting fixes this.
# if [ -n "$1" ] works. Careful!
# It is best to always quote tested variables.
# This accomplishes the same thing, using "pure" if/then statements.
if [ ! -z "$1" ]
then
  echo "Argument #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Argument #2 = $2"
  echo "At least 2 arguments passed to script."
else
  echo "Less than 2 arguments passed to script."
fi
# It's longer and less elegant than using an "and list".
exit 0
```

ÖRNEK 25.2 BİR "ve-listesi"ni KULLANARAK BAŞKA BİR KOMUT SATIRI ARGÜMANI TESTİ

```
#!/bin/bash
ARGS=1              # Number of arguments expected.
E_BADARGS=65        # Exit value if incorrect number of args passed.
test $# -ne $ARGS && echo "Usage: `basename $0` $ARGS argument(s)" && exit
$E_BADARGS
# If condition-1 true (wrong number of args passed to script),
# then the rest of the line executes, and script terminates.

# Line below executes only if the above test fails.
```

```
echo "Correct number of arguments passed to this script."
exit 0

# To check exit value, do a "echo $?" after script termination.
```

veya-listesi

```
komut-1  | |  komut-2  | |  komut-3  | |  ...  komut n
```

Her komut önceki komut `false` döndüğü sürece sırayla yürütülür. İlk `true` dönüş komut zincirini sonlandırır (`true` dönen ilk komut son çalıştırılan komuttur). Bu tabii ki "ve-listesi"nin tersidir.

ÖRNEK 25.3 BİR "ve-listesi" İLE BİRLİKTE "veya-listeleri"nin KULLANILMASI

```
#!/bin/bash
# delete.sh, not-so-cunning file deletion utility.
# Usage: delete filename
E_BADARGS=65
if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS       # No arg? Bail out.
else
  file=$1               # Set filename.
fi
[ ! -f "$file" ] && echo "File \"$file\" not found. \
```

```
Cowardly refusing to delete a nonexistent file."

# AND LIST, to give error message if file not present.

# Note echo message continued on to a second line with an escape.

[ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted.")

# OR LIST, to delete file if present.

# ( command1 ; command2 ) is, in effect, an AND LIST variant.

# Note logic inversion above.

# AND LIST executes on true, OR LIST on false.

exit 0
```

Bir "veya-listesi"nde ilk komut `true` döndürürse, zincir *çalışır*.

Bir `ve-listesinin` veya bir `veya-listesinin` çıkış durumu, son çalıştırılan komutun çıkış durumudur.

"Ve" ve "veya-listeleri"nin akıllı kombinasyonları mümkündür, ama mantık kolayca anlaşılmaz olabilir ve kapsamlı hata ayıklama gerektirir.

```
false && true || echo false            # false

# Same result as

( false && true ) || echo false        # false

# But *not*

false && ( true || echo false ) # (nothing echoed)

# Note left-to-right grouping and evaluation of statements,

# since the logic operators "&&" and "||" have equal precedence.

# It's best to avoid such complexities, unless you know what you're doing.

# Thanks, S.C.
```

Değişkenleri test etmek için **`ve/veya-listesini`** kullanan bir örnek için bkz. Örnek A-8.

BÖLÜM 26 DİZİLİMLER

Yeni Bash sürümleri tek boyutlu dizilimleri destekler. Dizilim elemanları **değişken[xx]** yazımı ile başlatılmış olabilir. Alternatif olarak, bir komut dosyası bir değişkeni açıkça **declare -a değişken** deyimi ile tanıtabilir. Dizilim elemanının içeriğini bulmak için *kıvrık parantez* gösterimini, **${değişken[xx]}**, kullanınız.

ÖRNEK 26.1 BASİT BİR DİZİLİM KULLANIMI

```
#!/bin/bash

area[11]=23

area[13]=37

area[51]=UFOs

# Array members need not be consecutive or contiguous.

# Some members of the array can be left uninitialized.

# Gaps in the array are o.k.

echo -n "area[11] = "

echo ${area[11]}          # {curly brackets} needed

echo -n "area[13] = "

echo ${area[13]}

echo "Contents of area[51] are ${area[51]}."

# Contents of uninitialized array variable print blank.

echo -n "area[43] = "

echo ${area[43]}

echo "(area[43] unassigned)"

echo

# Sum of two array variables assigned to third

area[5]=`expr ${area[11]} + ${area[13]}`

echo "area[5] = area[11] + area[13]"

echo -n "area[5] = "
```

```
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`

echo "area[6] = area[11] + area[51]"

echo -n "area[6] = "

echo ${area[6]}

# This fails because adding an integer to a string is not permitted.

echo; echo; echo

# --------------------------------------------------------------

# Another array, "area2".

# Another way of assigning array variables...

# array_name=( XXX YYY ZZZ ... )

area2=( zero one two three four )

echo -n "area2[0] = "

echo ${area2[0]}

# Aha, zero-based indexing (first element of array is [0], not [1]).

echo -n "area2[1] = "

echo ${area2[1]} # [1] is second element of array.

# --------------------------------------------------------------

echo; echo; echo

# ---------------------------------------------

# Yet another array, "area3".

# Yet another way of assigning array variables...

# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=seventeen [24]=twenty-four)

echo -n "area3[17] = "

echo ${area3[17]}

echo -n "area3[24] = "

echo ${area3[24]}

# ---------------------------------------------

exit 0
```

Dizi değişkenlerinin kendilerine ait bir sözdizimi vardır, ve ayrıca standart Bash komutları ve operatörlerinin dizilim kullanımı için özel uyarlanmış seçenekleri vardır.

```
array=( zero one two three four five )

echo ${array[0]}          # zero

echo ${array:0}           # zero

                          # Parameter expansion of first element.

echo ${array:1}           # ero

                          # Parameter expansion of first element,

                          #+ starting at position #1 (2nd character).

echo ${#array}            # 4

                          # Length of first element of array.
```

Dizilim bağlamında, bazı Bash yerleşiklerinin anlamı biraz değiştirilmiştir. Örneğin unset, dizilim elemanlarını, hatta bütün bir dizilimi siler.

ÖRNEK 26.2 DİZİLİMLERİN BAZI ÖZEL ÖZELLİKLERİ

```
#!/bin/bash

declare -a colors

# Permits declaring an array without specifying its size.

echo "Enter your favorite colors (separated from each other by a space)."

read -a colors       # Enter at least 3 colors to demonstrate features
below.

#  Special option to 'read' command,

#+ allowing assignment of elements in an array.

echo

echoelement_count=${#colors[@]}

# Special syntax to extract number of elements in array.

#     element_count=${#colors[*]} works also.
```

```
#
#  The "@" variable allows word splitting within quotes
#+ (extracts variables separated by whitespace).
index=0


while [ "$index" -lt "$element_count" ]
do    # List all the elements in the array.
  echo ${colors[$index]}
  let "index = $index + 1"
done
# Each array element listed on a separate line.
# If this is not desired, use echo -n "${colors[$index]} "
#
# Doing it with a "for" loop instead:
#   for i in "${colors[@]}"
#   do
#     echo "$i"
#   done
# (Thanks, S.C.)
echo
# Again, list all the elements in the array, but using a more elegant
method.
echo ${colors[@]}        # echo ${colors[*]} also works.
echo
# The "unset" command deletes elements of an array, or entire array.
unset colors[1]          # Remove 2nd element of array.
                         # Same effect as colors[1]=
echo ${colors[@]}        # List array again, missing 2nd element.
unset colors             # Delete entire array.
                         # unset colors[*] and
                         #+ unset colors[@] also work.
```

```
echo; echo -n "Colors gone."

echo ${colors[@]}       # List array again, now empty.

exit 0
```

Daha önceki örnekte görüldüğü gibi, **${dizilim_adı[@]}** veya **${dizilim_adı[*]}** ifadelerinin her ikisi de, dizilimin *bütün* elemanları anlamına gelir. Benzer şekilde, bir dizilimin eleman sayısını elde etmek için, **${#array_name[@]}** veya **${#array_name[*]}** ifadelerini kullanınız. **${#array_name}** ifadesi, dizinin ilk elemanı olan **${array_name[0]}** uzunluğunu (karakter sayısını) verir.

ÖRNEK 26.3 BOŞ DİZİLİMLER ve BOŞ ELEMANLAR

```
#!/bin/bash

# empty-array.sh

# An empty array is not the same as an array with empty elements.

array0=( first second third )

array1=( '' )         # "array1" has one empty element.

array2=( )            # No elements... "array2" is empty.

echo

echo "Elements in array0: ${array0[@]}"

echo "Elements in array1: ${array1[@]}"

echo "Elements in array2: ${array2[@]}"

echo

echo "Length of first element in array0 = ${#array0}"

echo "Length of first element in array1 = ${#array1}"

echo "Length of first element in array2 = ${#array2}"

echo

echo "Number of elements in array0 = ${#array0[*]}"      # 3

echo "Number of elements in array1 = ${#array1[*]}"      # 1 (surprise!)

echo "Number of elements in array2 = ${#array2[*]}"      # 0

echo

exit 0     # Thanks, S.C.
```

${array_name[@]}** ve **${array_name[*]} ilişkisi $@ ve $* arasındaki ilişkiye benzer. Bu güçlü dizilim yazımının birkaç kullanım alanı vardır.

```
# Copying an array.
array2=( "${array1[@]}" )
# or
array2="${array1[@]}"
# Adding an element to an array.
array=( "${array[@]}" "new element" )
# or
array[${#array[*]}]="new element"
# Thanks, S.C.
```

dizilim = (eleman1 eleman2 ... elemanN) başlatma işlemi, komut yer değiştirmesi yardımı ile, bir dizilime bir metin dosyasının içeriğini yüklemeyi mümkün kılar.

```
#!/bin/bash
filename=sample_file
#            cat sample_file
#
#            1 a b c
#            2 d e fg
declare -a array1
array1=( `cat "$filename" | tr '\n' ' '`       # Loads contents
                                               # of $filename into array1.
#         list file to stdout.
#                            change linefeeds in file to spaces.
echo ${array1[@]}             # List the array.
#                               1 a b c 2 d e fg
#
```

```
#  Each whitespace-separated "word" in the file

#+ has been assigned to an element of the array.

element_count=${#array1[*]}

echo $element_count           # 8
```

Dizilimler eski tanıdık algoritmaların, kabuk betiği halinde dağıtımına izin verir. Bunun mutlaka iyi bir fikir olup olmadığının kararı okuyucuya bırakılmıştır.

ÖRNEK 26.4 ESKİ BİR DOST: *KABARCIK SIRALAMA*

```
#!/bin/bash

# bubble.sh: Bubble sort, of sorts.

# Recall the algorithm for a bubble sort. In this particular version...

#  With each successive pass through the array to be sorted,

#+ compare two adjacent elements, and swap them if out of order.

# At the end of the first pass, the "heaviest" element has sunk to bottom.

# At the end of the second pass, the next "heaviest" one has sunk next to
bottom.

# And so forth.

# This means that each successive pass needs to traverse less of the array.

# You will therefore notice a speeding up in the printing of the later
passes.

exchange()

{

  # Swaps two members of the array.

  local temp=${Countries[$1]}       # Temporary storage

                                    #+ for element getting swapped out.

  Countries[$1]=${Countries[$2]}

  Countries[$2]=$temp

  return

}
```

```
declare -a Countries                  # Declare array,
                                      #+ optional here since it's initialized
below.
#  Is it permissable to split an array variable over multiple lines
#+ using an escape (\)?
#  Yes.
Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
Israel Peru Canada Oman Denmark Wales France Kenya \
Xanadu Qatar Liechtenstein Hungary)
#  "Xanadu" is the mythical place where, according to Coleridge,
#+ Kubla Khan did a pleasure dome decree.
clear                         # Clear the screen to start with.
echo "0: ${Countries[*]}"  # List entire array at pass 0.
number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"
count=1 # Pass number.
while [ "$comparisons" -gt 0 ] # Beginning of outer loop
do
  index=0     # Reset index to start of array after each pass.
  while [ "$index" -lt "$comparisons" ]    # Beginning of inner loop
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    # If out of order...
    # Recalling that \> is ASCII comparison operator
    #+ within single brackets.
    #  if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]} ]]
    #+ also works.
    then
      exchange $index `expr $index + 1`     # Swap.
    fi
```

```
    let "index += 1"

  done    # End of inner loop

let "comparisons -= 1"          #  Since "heaviest" element bubbles to
bottom,

                                #+ we need do one less comparison each pass.

echo

echo "$count: ${Countries[@]}" # Print resultant array at end of each pass.

echo

let "count += 1"                # Increment pass count.

done                            # End of outer loop

                                # All done.

exit 0
```

--

Dizilimler, *Eratosthenes'in Eleğinin* bir kabuk betiği sürümü uygulamasını sağlar. Tabii ki, bu tür yoğun-kaynak gerektiren bir uygulama gerçekten, C gibi derlenmiş bir dilde yazılmalıdır. Bir betik olarak bu, dayanılmaz yavaş derecede çalışır.

ÖRNEK 26.5 KARMAŞIK BİR DİZİLİM UYGULAMASI: ERATOSTHENES'İN ELEĞİ

```
#!/bin/bash

# sieve.sh

# Sieve of Eratosthenes

# Ancient algorithm for finding prime numbers.

# This runs a couple of orders of magnitude

# slower than the equivalent C program.

LOWER_LIMIT=1               # Starting with 1.

UPPER_LIMIT=1000            # Up to 1000.
```

```
# (You may set this higher... if you have time on your hands.)
PRIME=1
NON_PRIME=0
let SPLIT=UPPER_LIMIT/2
# Optimization:
# Need to test numbers only halfway to upper limit.
declare -a Primes
# Primes[] is an array.
initialize ()
{
# Initialize the array.
i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
# Assume all array members guilty (prime)
# until proven innocent.
}

print_primes()
{
# Print out the members of the Primes[] array tagged as prime.
i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  if [ "${Primes[i]}" -eq "$PRIME" ]
  then
    printf "%8d" $i
```

```
    # 8 spaces per number gives nice, even columns.
  fi
  let "i += 1"
done
}

sift() # Sift out the non-primes.
{
let i=$LOWER_LIMIT+1
# We know 1 is prime, so let's start with 2.
until [ "$i" -gt "$UPPER_LIMIT" ]
do
if [ "${Primes[i]}" -eq "$PRIME" ]
# Don't bother sieving numbers already sieved (tagged as non-prime).
then
  t=$i
  while [ "$t" -le "$UPPER_LIMIT" ]
  do
    let "t += $i "
    Primes[t]=$NON_PRIME
    # Tag as non-prime all multiples.
  Done
fi
  let "i += 1"
done
}

# Invoke the functions sequentially.

initialize
```

```
sift

print_primes

# This is what they call structured programming.

echo

exit 0

# ---------------------------------------------- #

# Code below line will not execute.

# This improved version of the Sieve, by Stephane Chazelas,

# executes somewhat faster.

# Must invoke with command-line argument (limit of primes).

UPPER_LIMIT=$1                  # From command line.

let SPLIT=UPPER_LIMIT/2         # Halfway to max number.

Primes=( '' $(seq $UPPER_LIMIT) )

i=1

until (( ( i += 1 ) > SPLIT )) # Need check only halfway.

do

  if [[ -n $Primes[i] ]]

  then

    t=$i

    until (( ( t += i ) > UPPER_LIMIT ))

    do

      Primes[t]=

    done

  fi

done

echo ${Primes[*]}

exit 0
```

Dizilim-tabanlı bu asal sayı üretecini, dizilim kullanmayan alternatifi (bkz. Örnek A.16) ile karşılaştırınız.

--

Dizilimler bir dereceye kadar kendilerini, Bash'in yerleşik desteğe sahip olmadığı veri yapılarına öykünmeye borçludurlar.

ÖRNEK 26.6 SON GİREN İLK ÇIKAR YIĞITININ ÖYKÜNÜMÜ

```
#!/bin/bash

# stack.sh: push-down stack simulation

#  Similar to the CPU stack, a push-down stack stores data items

#+ sequentially, but releases them in reverse order, last-in first-out.

BP=100             # Base Pointer of stack array.

                   # Begin at element 100.

SP=$BP             # Stack Pointer.

                   # Initialize it to "base" (bottom) of stack.

Data=              # Contents of stack location.

                   # Must use local variable,

                   #+ because of limitation on function return range.

declare -a stack


stackpush()        # Push item on stack.

{

if [ -z "$1" ]     # Nothing to push?

then

  return

fi

let "SP -= 1"      # Bump stack pointer.

stack[$SP]=$1

return

}
```

```
pop()            # Pop item off stack.

{

Data=            # Empty out data item.

if [ "$SP" -eq "$BP" ]      # Stack empty?

then

  return

fi               # This also keeps SP from getting past 100,

                 #+ i.e., prevents a runaway stack.

Data=${stack[$SP]}

let "SP += 1"    # Bump stack pointer.

return

}


status_report() # Find out what's happening.
{

echo "-----------------------------------"

echo "REPORT"

echo "Stack Pointer = $SP"

echo "Just popped \""$Data"\" off the stack."

echo "-----------------------------------"

echo

}

# ===========================================================

# Now, for some fun.

echo

# See if you can pop anything off empty stack.

pop

status_report

echo

push garbage
```

```
pop

status_report       # Garbage in, garbage out.

value1=23; push $value1

value2=skidoo; push $value2

value3=FINAL; push $value3

pop                 # FINAL

status_report

pop                 # skidoo

status_report

pop                 # 23

status_report       # Last-in, first-out!

#  Notice how the stack pointer decrements with each push,

#+ and increments with each pop.

echo

# ==========================================================

# Exercises:

# ---------

# 1) Modify the "push()" function to permit pushing

# + multiple element on the stack with a single function call.


# 2) Modify the "pop()" function to permit popping

# + multiple element from the stack with a single function call.


# 3) Using this script as a jumping-off point,

# + write a stack-based 4-function calculator.

exit 0

--
```

Dizilim "altsimgeleri"nin değerleriyle oynanması ve üzerlerinde karmaşık çalışmalar yapılması, ara değişkenlerin kullanımını gerektirebilir. Bu tarz projelerde, yine Perl ya da C gibi daha güçlü bir programlama dili kullanmayı düşününüz.

ÖRNEK 26.7 KARMAŞIK BİR DİZİLİM UYGULAMASI: İLGİNÇ BİR MATEMATİKSEL DİZİNİN KEŞFİ

```
#!/bin/bash

# Douglas Hofstadter's notorious "Q-series":

# Q(1) = Q(2) = 1

# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), for n>2

# This is a "chaotic" integer series with strange and unpredictable
behavior.

# The first 20 terms of the series are:

# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

# See Hofstadter's book, "Goedel, Escher, Bach: An Eternal Golden Braid",

# p. 137, ff.


LIMIT=100                           # Number of terms to calculate

LINEWIDTH=20                        # Number of terms printed per line

Q[1]=1                              # First two terms of series are 1.

Q[2]=1

Echo

echo "Q-series [$LIMIT terms]:"

echo -n "${Q[1]} "                  # Output first two terms.

echo -n "${Q[2]} "


for ((n=3; n <= $LIMIT; n++))   # C-like loop conditions.

do          # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]] for n>2

# Need to break the expression into intermediate terms,

# since Bash doesn't handle complex array arithmetic very well.

  let "n1 = $n - 1"                 # n-1

  let "n2 = $n - 2"                 # n-2
```

```
  t0=`expr $n - ${Q[n1]}`        # n - Q[n-1]
  t1=`expr $n - ${Q[n2]}`        # n - Q[n-2]

  T0=${Q[t0]}                    # Q[n - Q[n-1]]
  T1=${Q[t1]}                    # Q[n - Q[n-2]]
Q[n]=`expr $T0 + $T1`            # Q[n - Q[n-1]] + Q[n - ![n-2]]
echo -n "${Q[n]} "

if [ `expr $n % $LINEWIDTH` -eq 0 ] # Format output.
then      # mod
  echo # Break lines into neat chunks.
fi
done
echo
exit 0

# This is an iterative implementation of the Q-series.
# The more intuitive recursive implementation is left as an exercise.
# Warning: calculating this series recursively takes a *very* long time.
```

--

Bash, sadece tek-boyutlu dizilimleri destekler, ancak küçük bir hile çok-boyutlu olanların benzetimine de izin verir.

ÖRNEK 26.8 İKİ BOYUTLU BİR DİZİLİMİN BENZETİMİ, SONRA EĞİMİ

```
#!/bin/bash
```

```
# Simulating a two-dimensional array.
# A two-dimensional array stores rows sequentially.
Rows=5
Columns=5
declare -a alpha                # char alpha [Rows] [Columns];
                                # Unnecessary declaration.
load_alpha ()
{
local rc=0
local index
for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
do
  local row=`expr $rc / $Columns`
  local column=`expr $rc % $Rows`
  let "index = $row * $Rows + $column"
  alpha[$index]=$i              # alpha[$row][$column]
  let "rc += 1"
done
# Simpler would be
#   declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
# but this somehow lacks the "flavor" of a two-dimensional array.
}
print_alpha ()
{
local row=0
local index
echo
while [ "$row" -lt "$Rows" ]    # Print out in "row major" order –
do                              # columns vary
                                # while row (outer loop) remains the same.
```

```
  local column=0
  while [ "$column" -lt "$Columns" ]
  do
    let "index = $row * $Rows + $column"
    echo -n "${alpha[index]} "   # alpha[$row][$column]
    let "column += 1"
  done
  let "row += 1"
  echo
done
# The simpler equivalent is
#   echo ${alpha[*]} | xargs -n $Columns
echo
}


filter ()                       # Filter out negative array indices.
{
echo -n " "                     # Provides the tilt.
if [[ "$1" -ge 0 && "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns"
]]
then
  let "index = $1 * $Rows + $2"
  # Now, print it rotated.
  echo -n " ${alpha[index]}"    # alpha[$row][$column]
fi
}


rotate ()                       # Rotate the array 45 degrees
{                               # ("balance" it on its lower lefthand
corner).
local row
```

```
local column


for (( row = Rows; row > -Rows; row-- )) # Step through the array
backwards.
do
  for (( column = 0; column < Columns; column++ ))
  do
    if [ "$row" -ge 0 ]
    then
      let "t1 = $column - $row"
      let "t2 = $column" else
      let "t1 = $column"
      let "t2 = $column + $row"
    fi
  filter $t1 $t2                # Filter out negative array indices.
  done
  echo; echo
done
# Array rotation inspired by examples (pp. 143-146) in
# "Advanced C Programming on the IBM PC", by Herbert Mayer
# (see bibliography).
}
#-------------------------------------------------------
#load_alpha                     # Load the array.
print_alpha                     # Print it out.
rotate                          # Rotate it 45 degrees counterclockwise.
#-------------------------------------------------------##
# This is a rather contrived, not to mention kludgy simulation.
#
# Exercises:
# ---------
```

```
# 1) Rewrite the array loading and printing functions
# + in a more intuitive and elegant fashion.
#
# 2) Figure out how the array rotation functions work.
# Hint: think about the implications of backwards-indexing an array.
exit 0
```

İki-boyutlu bir dizilim, tek-boyutlu bir dizilime denktir, ancak referans için ek adresleme modları gerekir ve "satır" ile "sütun" konumlarına göre elemanların ayrı ayrı değerleri değiştirilmelidir.

İki-boyutlu bir dizinin daha ayrıntılı benzetimi için bkz. Örnek A.10.

BÖLÜM 27 DOSYALAR

başlangıç dosyaları

Bu dosyalar bir kullanıcı kabuğu gibi çalışan Bash ve sistem başlatmanın sonrasında çağrılan tüm Bash betikleri için bulundurulan öteki adları ve çevresel değişkenleri içerir.

`/etc/profile`

Çoğunlukla çevreyi ayarlayan sistem varsayılanlarıdır (tüm Bourne tipi kabukları, sadece Bash değil [1]).

`/etc/bashrc`

Sistem fonksiyonları ve Bash için öteki adlardır.

`$HOME/.bash_profile`

Her kullanıcının ana dizininde bulunan, kullanıcıya özel Bash çevre varsayılan ayarlarıdır

(`/etc/profile`'ın yerel karşılığı).

`$HOME/.bashrc`

Her kullanıcının ana dizininde bulunan kullanıcıya özel Bash başlangıç dosyasıdır. (`/etc/bashrc`'nin yerel karşılığı). Sadece etkileşimli kabuklar ve kullanıcı betikleri bu dosyayı okur. Örnek `.bashrc` dosyası için bkz. Ek G.

çıkış dosyası

`$HOME/.bash_logout`

Her kullanıcının ana dizininde bulunan kullanıcıya özel komut dosyasıdır. Bir oturum açma (Bash) kabuğundan çıkılması üzerine, bu dosyadaki komutlar çalıştırılır.

Notlar

[1] Bu, klasik Bourne kabuğu (**sh**) tarafından türetilmemiş **csh**, **tcsh** ve diğer kabuklar için geçerli değildir.

BÖLÜM 28 /dev VE /proc

Bir Linux ya da UNIX makinesinin genellikle iki özel amaçlı dizini vardır: `/dev` ve `/proc`.

28.1 /dev

`/dev` dizini donanımda mevcut olan veya olmayan fiziksel aygıtlar için girişleri içerir. [1] Mantıksal bağ kurulan dosya sistemlerini içeren sabit disk bölümlerinin `/dev` girişleri vardır, bunu `df` komutuyla gösterebilirsiniz.

```
bash$ df
Filesystem         1k-blocks      Used  Available  Use%  Mounted on
 /dev/hda6             495876    222748     247527   48% /
 /dev/hda1              50755      3887      44248    9% /boot
 /dev/hda8             367013     13262     334803    4% /home
 /dev/hda5            1714416   1123624     503704   70% /usr
```

Diğer şeyler arasında, `/dev` dizini aynı zamanda `/dev/loop0` gibi *geridönüş* aygıtlarını da içerir. Bir geridönüş aygıtı normal bir dosyaya sanki bir blok aygıtıymış gibi erişilmesini sağlayan bir hiledir. [2] Bu tek bir büyük dosya içinde bütün bir dosya sistemine mantıksal bağ kurulmasını sağlar. Bkz. Örnek 13.6 ve Örnek 13.5. `/dev` içindeki `/dev/null`, `/dev/zero` and `/dev/urandom` gibi birkaç sözde-aygıtların diğer özel kullanımları vardır.

28.2 /proc

`/proc` dizini aslında bir sözde-dosya sistemidir. `/proc` dizinindeki dosyalar, şu anda çalışan sistem ve çekirdek *işlemlerinin* aynasıdır ve onlar hakkında bilgi ve istatistik içerir.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia
Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md

mdbash$ cat /proc/interrupts
           CPU0
  0:      84505    XT-PIC  timer
  1:       3375    XT-PIC  keyboard
  2:          0    XT-PIC  cascade
  5:          1    XT-PIC  soundblaster
  8:          1    XT-PIC  rtc
```

```
  12:        4231       XT-PIC  PS/2 Mouse

  14:      109373       XT-PIC  ide0

 NMI:           0

 ERR:           0


bash$ cat /proc/partitions

major  minor #blocks  name   rio rmerge rsect  ruse  wio  wmerge wsect wuse
running use     aveq

3          0 3007872  hda    4472 22260 114520 94240 3551 18703  50384
549710 0 111550 644030

3          1 52416    hda1     27 395    844    960     4     2    14
180 0  800     1140

3          2     1    hda2      0   0     0       0     0     0     0
0 0     0          0

3          4 165280   hda4     10   0    20     210     0     0     0
0 0  210      210

...


bash$ cat /proc/loadavg

0.13 0.42 0.27 2/44 1119
```

Kabuk betikleri `/proc` içindeki belirli dosyalardan veri seçip çıkarabilirler. [3]

```
kernel_version=$( awk '{ print $3 }' /proc/version )



CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )

if [ $CPU = Pentium ]

then

  run_some_commands

  ...

else
```

```
  run_different_commands

  ...

fi
```

/proc dizini sıradışı sayısal isimleri ile bir dizi alt dizin içerir. Bu isimlerin her biri şu anda çalışan bir işlemin süreç kimliği ile eşleşir. Bu alt dizinlerin her birinde, ilgili işlem hakkında yararlı bilgiler tutan bir dizi dosya vardır. stat ve status dosyaları işlem üzerinde çalışan istatistiği tutar, cmdline dosyası, işlem ile birlikte çağrılan komut satırı argümanlarını tutar, ve exe dosyası yürütülen işlemin tam yol adı için bir sembolik bağdır. Birkaç tür başka dosyalar da vardır, ancak bu verilenler bir komut dosyası açısından en ilginçleri gibi görünmektedir.

ÖRNEK 28.1 BİR PID İLE İLİŞKİLİ OLAN İŞLEM SÜRECİNİ BULMAK

```
#!/bin/bash

# pid-identifier.sh: Gives complete path name to process associated with
pid.

ARGNO=1      # Number of arguments the script expects.

E_WRONGARGS=65

E_BADPID=66

E_NOSUCHPROCESS=67

E_NOPERMISSION=68

PROCFILE=exe

if [ $# -ne $ARGNO ]

then

  echo "Usage: `basename $0` PID-number" >&2      # Error message >stderr.

  exit $E_WRONGARGS

fi

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )

# Checks for pid in "ps" listing, field #1.

# Then makes sure it is the actual process, not the process invoked by this
script.
```

```
# The last "grep $1" filters out this possibility.

if [ -z "$pidno" ] # If, after all the filtering, the result is a zero-
length string,

then               # no running process corresponds to the pid given.

  echo "No such process running."

  exit $E_NOSUCHPROCESS

fi

# Alternatively:

# if ! ps $1 > /dev/null 2>&1

# then      # no running process corresponds to the pid given.

#   echo "No such process running."

#   exit $E_NOSUCHPROCESS

# fi

# To simplify the entire process, use "pidof".


if [ ! -r "/proc/$1/$PROCFILE" ]       # Check for read permission.

then

  echo "Process $1 running, but..."

  echo "Can't get read permission on /proc/$1/$PROCFILE."

  exit $E_NOPERMISSION # Ordinary user can't access some files in /proc.

fi


# The last two tests may be replaced by:

#    if ! kill -0 $1 > /dev/null 2>&1       # '0' is not a signal, but

                                            # this will test whether it is
possible

                                            # to send a signal to the
process.

#    then echo "PID doesn't exist or you're not its owner" >&2

#    exit $E_BADPID

#    fi

exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
```

```
# Or exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
# /proc/pid-number/exe is a symbolic link
# to the complete path name of the invoking process.

if [ -e "$exe_file" ]    # If /proc/pid-number/exe exists...
then                     # the corresponding process exists.
  echo "Process #$1 invoked by $exe_file."
else
  echo "No such process running."
fi
# This elaborate script can *almost* be replaced by
# ps ax | grep $1 | awk '{ print $5 }'
# However, this will not work...
# because the fifth field of 'ps' is argv[0] of the process,
# not the executable file path.
#
# However, either of the following would work.
#      find /proc/$1/exe -printf '%l\n'
#      lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
# Additional commentary by Stephane Chazelas.
exit 0
```

ÖRNEK 28.2 ÇEVRİMİÇİ BAĞLANTI DURUMU

```
#!/bin/bash
PROCNAME=pppd             # ppp daemon
PROCFILENAME=status       # Where to look.
NOTCONNECTED=65
INTERVAL=2                # Update every 2 seconds.
```

```
pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME | awk '{
print $1 }' )

# Finding the process number of 'pppd', the 'ppp daemon'.

# Have to filter out the process lines generated by the search itself.

#

#  However, as Oleg Philon points out,

#+ this could have been considerably simplified by using "pidof".

#  pidno=$( pidof $PROCNAME )

#

#  Moral of the story:

#+ When a command sequence gets too complex, look for a shortcut.


if [ -z "$pidno" ]      # If no pid, then process is not running.

then

  echo "Not connected."

  exit $NOTCONNECTED

else

  echo "Connected.";

  echo

fi


while [ true ]          # Endless loop, script can be improved here.

do

  if [ ! -e "/proc/$pidno/$PROCFILENAME" ]

  # While process running, then "status" file exists.

  then

    echo "Disconnected."

    exit $NOTCONNECTED

  fi

netstat -s | grep "packets received"      # Get some connect statistics.

netstat -s | grep "packets delivered"
```

```
  sleep $INTERVAL
  echo; echo
done
exit 0

# As it stands, this script must be terminated with a Control-C.
#    Exercises:
#    ---------
#    Improve the script so it exits on a "q" keystroke.
#    Make the script more user-friendly in other ways.
```

Genel olarak, `/proc` dosyalarına yazmak tehlikelidir, çünkü dosya sisteminin bozulmasına veya makinenizin çökmesine sebep olabilir.

Notlar

[1] `/dev` girişleri fiziksel ve sanal aygıtlar için mantıksal bağlantı noktaları sağlar. Bu girişler çok az disk alanı kullanır.

`/dev/null`, `/dev/zero`, and `/dev/urandom` gibi bazı aygıtlar sanaldır. Onlar gerçek fiziksel aygıtlar değildir ve sadece yazılımda bulunmaktadır.

[2] *Bir blok aygıtı*, karakter birimleri halinde veriye erişen karakter aygıtlarının tersine, veriyi parçalar veya bloklar halinde okur ve/veya yazar. Blok aygıtlarına verilecek örnek bir sabit disk ve CD-ROM sürücüdür. Karakter aygıtına verilecek örnek bir klavyedir.

[3] `procinfo`, `free`, `vmstat`, `lsdev`, ve `uptime` gibi bazı sistem komutları da bunu yapabilir.

BÖLÜM 29 SIFIRLAR ve BOŞLAR

`/dev/zero` ve `/dev/null`

`/dev/null` Kullanımı

`/dev/null`'ı bir "kara delik" olarak düşünün. Bu bir salt-yazılır dosyaya en yakın eşdeğerdir. Kendisine yazılan her şey sonsuza kadar kaybolur. Ondan okumak veya çıktı beklemek girişimi boş sonuç verir. Bununla birlikte, `/dev/null` kullanımı hem komut satırında ve hem de betiklerde oldukça yararlı olabilir.

`stdout`'u bastırmak.

```
cat $filename >/dev/null

# Contents of the file will not list to stdout.
```

`stderr`'i bastırmak. (bkz.Örnek 12.2)

```
rm $badname 2>/dev/null

#           So error messages [stderr] deep-sixed.
```

Hem `stdout` ve *hem de* `stderr` çıktısını bastırmak.

```
cat $filename 2>/dev/null >/dev/null

# If "$filename" does not exist, there will be no error message output

# If "$filename" does exist, the contents of the file will not list to
stdout.

# Therefore, no output at all will result from the above line of code.

#

#  This can be useful in situations where the return code from a command

#+ needs to be tested, but no output is desired.
```

```
#

# cat $filename &>/dev/null

#     also works, as Baris Cicek points out.
```

Dosyanın kendisini ve tüm izinlerini koruyarak bir dosyanın içeriğini silmek (bkz. Örnek 2.1 ve Örnek 2.2):

```
cat /dev/null > /var/log/messages

# : > /var/log/messages has same effect, but does not spawn a new process.

cat /dev/null > /var/log/wtmp
```

Otomatik olarak bir günlük dosyasının içeriğini boşaltmak (ticari Web sitelerinin gönderdiği "çerezler" ile başa çıkmak için özellikle iyidir):

ÖRNEK 29.1 ÇEREZ KAVANOZUNU GİZLEMEK

```
if [ -f ~/.netscape/cookies ] # Remove, if exists.

then

  rm -f ~/.netscape/cookies

fi

ln -s /dev/null ~/.netscape/cookies

# All cookies now get sent to a black hole, rather than saved to disk.
```

`/dev/zero` Kullanımı

`/dev/zero`, `/dev/null` gibi bir sözde-dosyadır, ama aslında boş değerler (sayısal sıfır ASCII türü değil) içerir. Kendisine yazılan çıktı kaybolur, ve aslında `/dev/zero` boş değerlerini okumak oldukça zordur, ama bu bir od veya bir hex editör ile yapılabilir.

`/dev/zero` için başlıca kullanım alanı, belirtilen uzunlukta geçici bir takas dosyası olarak tasarlanan, bir başlatılmış işlevsiz dosya oluşturmaktır.

ÖRNEK 29.2 /dev/zero KULLANARAK BİR GETİR GÖTÜR DOSYASI KURULUMU

```
#!/bin/bash

# Creating a swapfile.

# This script must be run as root.

ROOT_UID=0                # Root has $UID 0.

E_WRONG_USER=65           # Not root?

FILE=/swap

BLOCKSIZE=1024

MINBLOCKS=40

SUCCESS=0

if [ "$UID" -ne "$ROOT_UID" ]

then

  echo; echo "You must be root to run this script."; echo

  exit $E_WRONG_USER

fi

if [ -n "$1" ]

then

  blocks=$1

else

  blocks=$MINBLOCKS       # Set to default of 40 blocks
```

```
fi                          # if nothing specified on command line.

if [ "$blocks" -lt $MINBLOCKS ]

then

  blocks=$MINBLOCKS         # Must be at least 40 blocks long.

fi

echo "Creating swap file of size $blocks blocks (KB)."

dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks      # Zero out file.

mkswap $FILE $blocks        # Designate it a swap file.

swapon $FILE                # Activate swap file.

echo "Swap file created and activated."

exit $SUCCESS
```

`/dev/zero` için bir diğer uygulama da, özel bir amaç için belirli boyutta bir dosyayı "sıfırlamaktır", örneğin bir geridönüş aygıtı üzerinde bir dosya sistemi için mantıksal bağ kurmak (bkz. Örnek 13.6) ya da bir dosyayı güvenli bir şekilde silmek (bkz. Örnek 12.41).

ÖRNEK 29.3 SANAL DİSK OLUŞTURMAK

```
#!/bin/bash

# ramdisk.sh

#  A "ramdisk" is a segment of system RAM memory

#+ that acts as if it were a filesystem.

#  Its advantage is very fast access (read/write time).

#  Disadvantages: volatility, loss of data on reboot or powerdown.

#                 less RAM available to system.
```

```
#

#  What good is a ramdisk?

#  Keeping a large dataset, such as a table or dictionary on ramdisk

#+ speeds up data lookup, since memory access is much faster than disk
access.

E_NON_ROOT_USER=70               # Must run as root.

ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk

SIZE=2000                        # 2K blocks (change as appropriate)

BLOCKSIZE=1024                   # 1K (1024 byte) block size

DEVICE=/dev/ram0                 # First ram device

deviceusername=‘id -nu‘

if [ "$username" != "$ROOTUSER_NAME" ]

then

  echo "Must be root to run \"‘basename $0‘\"."

  exit $E_NON_ROOT_USER

fi

if [ ! -d "$MOUNTPT" ]           #  Test whether mount point already there,

then                             #+ so no error if this script is run

  mkdir $MOUNTPT                 #+ multiple times.

fi

dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE # Zero out RAM device.

mke2fs $DEVICE                   # Create an ext2 filesystem on it.

mount $DEVICE $MOUNTPT           # Mount it.
```

```
chmod 777 $MOUNTPT                  # So ordinary user can access ramdisk.

# However, must be root to unmount it.

echo "\"$MOUNTPT\" now available for use."

# The ramdisk is now accessible for storing files, even by an ordinary
user.

#  Caution, the ramdisk is volatile, and its contents will disappear

#+ on reboot or power loss.

#  Copy anything you want saved to a regular directory.

# After reboot, run this script again to set up ramdisk.

# Remounting /mnt/ramdisk without the other steps will not work.

exit 0
```

BÖLÜM 30 HATA AYIKLAMA

Bash kabuğu ne hata ayıklayıcı, ne de herhangi bir hata ayıklama özel komutu veya yapılarını içermez. Sözdizimi hataları ya da betikteki yazım hataları şifreli hata iletileri üretir, ki bunun işlevsel olmayan bir komut dosyasının hatasını ayıklamada genellikle hiçbir yardımı olmaz.

ÖRNEK 30.1 HATALI BİR KOMUT DOSYASI

```
#!/bin/bash

# ex74.sh

# This is a buggy script.

a=37

if [$a -gt 27 ]

then

  echo $a

fi

exit 0
```

Komut dosyasının çıktısı:

```
./ex74.sh: [37: command not found
```

Yukarıdaki programdaki hata nedir (İpucu: **if** sonrasında) ?

ÖRNEK 30.2 EKSİK ANAHTAR KELİME

```
#!/bin/bash

# missing-keyword.sh: What error message will this generate?

for a in 1 2 3

do

  echo "$a"

# done     # Required keyword 'done' commented out in line 7.

exit 0
```

Komut dosyasının çıktısı:

```
missing-keyword.sh: line 10: syntax error: unexpected end of file
```

Unutmayınız ki, hata mesajı mutlaka hatanın oluştuğu satıra referans olmaz, ama Bash yorumlayıcısının nihayet hatanın farkına vardığı satıra referans verir.

Betik çalışıyorsa, ancak beklendiği gibi çalışmıyorsa ne olur? Bu çok tanıdık mantık hatasıdır.

ÖRNEK 30.3 test24, BİR BAŞKA HATALI KOMUT DOSYASI

```
#!/bin/bash

#  This is supposed to delete all filenames in current directory
#+ containing embedded spaces.
# It doesn't work. Why not?
badname=`ls | grep ' '`
# echo "$badname"
rm "$badname"
exit 0
```

`echo "$badname"` yorum satırını komut satırına çevirerek, Örnek 30.3'teki sorunun ne olduğunu bulmaya çalışın. Echo ifadeleri beklediğinizin gerçekten olup olmadığını görmek için yararlıdır.

Bu özel durumda, **`rm "$badname"`** istenen sonucu vermeyecektir, çünkü `$badname` tırnak içine yazılmamalıdır. Tırnak içine yerleştirmekle **rm** komutuna sadece bir argüman (sadece bir dosya adı ile eşleşecektir) geçirilir. Kısmi düzeltme `$badname` etrafındaki tırnağı kaldırmak ve `$IFS` değişkenini sadece yeni satır karakterini içerecek şekilde sıfırlamaktır, **`IFS=$'\n'.`** Ancak bu işi yapmanın daha basit yolları vardır.

```
# Correct methods of deleting filenames containing spaces.

rm *\ *

rm *" "*

rm *' '*

# Thank you. S.C.
```

Hatalı bir komut dosyasının belirtilerini özetlemek gerekirse,

1. Bir "sözdizimi hatası" mesajı verir ve hiç çalışmaz.
2. Çalışır, ancak beklendiği gibi çalışmaz (mantık hatası).
3. Beklendiği gibi çalışır, ama kötü yan etkileri vardır (mantık bombası).

Çalışmayan komut dosyaları için hata ayıklama araçları aşağıdakileri içerir:

1. Değişkenleri izlemek için betiğin kritik noktalarında `echo` komutunu kullanmak, ve başka neler olup bittiğinin anlık gösterimini vermek.
2. Kritik noktalarda işlemleri veya veri akışını kontrol etmek için **`tee`** filtresi kullanmak.
3. ayar seçeneği bayrakları **`–n –v –x`**
   `sh -n betikadı` betiği gerçekten çalıştırmadan yazım hatalarını kontrol eder. Bu, komut dosyasına **`set –n`** veya **`set -o noexec`** satırını eklemek ile eşdeğerdir. Unutmayınız ki, sözdizimi hatalarının belirli türleri bu onayı atlamış olabilir.

   `sh -v betikadı` her komutu çalıştırmadan önce görüntüler. Bu, komut dosyasına **`set -v`** veya **`set -v verbose`** satırını eklemek ile eşdeğerdir.

   `-n` ve **`-v`** bayrakları birlikte iyi çalışır. **`sh -nv betikadı`** ayrıntılı bir sözdizimi denetimi verir.

   `sh -x betikadı` her komutun sonucunu kısaltılmış bir şekilde görüntüler. Bu, komut dosyasına **`set –x`** veya **`set -o xtrace`** satırını eklemek ile eşdeğerdir.

   Betiğe **`set -u`** veya **`set -o nounset`** eklemek onu çalıştırır, ancak bildirilmemiş bir değişkenin her kullanma girişimi için bir ilişkisiz değişken hata mesajı verir.

4. Betiğin kritik noktalarında bir değişken veya durumu test etmek için "assert" fonksiyonunu kullanmak (Bu fikir C'den ödünç alınmıştır).

ÖRNEK 30.4 "assert" İLE BİR DURUM TESTİ

```
#!/bin/bash
# assert.sh
assert ()                 #  If condition false,
{                         #+ exit from script with error message.
E_PARAM_ERR=98
E_ASSERT_FAILED=99

if [ -z "$2" ]          # Not enough parameters passed.
then
  return $E_PARAM_ERR  # No damage done.
fi
lineno=$2

if [ ! $1 ]
then
  echo "Assertion failed: \"$1\""
  echo "File \"$0\", line $lineno"
  exit $E_ASSERT_FAILED
# else
# return
# and continue executing script.
fi
}

a=5
b=4
condition="$a -lt $b"   # Error message and exit from script.
                        #   Try setting "condition" to something
else,
                        #+ and see what happens.

assert "$condition" $LINENO
# The remainder of the script executes only if the "assert" does not
fail.
# Some commands.
# ...
echo "You will never see this statement echo."
# ...
# Some more commands.

exit 0
```

5. Çıkışta yakalama.

Bir betiğin **exit** komutu sinyal 0'ı tetikleyerek, komut dosyasının kendisi olan süreci sonlandırır. [1] Bu çıkışı, örneğin, değişkenlerin bir "çıktı"sını zorlayarak yakalamak genellikle yararlıdır. Trap (tuzak), betikteki ilk komut olmalıdır.

Yakalama sinyalleri

trap

Bir sinyal alındığındaki eylemi belirtir. Hata ayıklama için de yararlıdır.

Bir *sinyal*, sadece çekirdek ya da başka bir işlem yoluyla, bir işleme bazı belirtilen eylemleri yapması için (genellikle bu bir sonlandırma olabilir) gönderilen bir mesajdır. Örneğin, **Control-C**'ye basmak, çalışan bir programa kullanıcı kesmesi, bir INT sinyali gönderir.

```
trap '' 2
# Ignore interrupt 2 (Control-C), with no action specified.
trap 'echo "Control-C disabled."' 2
# Message when Control-C pressed.
```

ÖRNEK 30.5 ÇIKIŞTA YAKALAMA

```
#!/bin/bash
trap 'echo Variable Listing --- a = $a b = $b' EXIT
# EXIT is the name of the signal generated upon exit from a script.
a=39
b=36
exit 0
# Note that commenting out the 'exit' command makes no difference,
# since the script exits in any case after running out of commands.
```

ÖRNEK 30.6 CONTROL-C SONRASINDA TEMİZLİK

```
#!/bin/bash
# logon.sh: A quick 'n dirty script to check whether you are on-line yet.
TRUE=1
LOGFILE=/var/log/messages
# Note that $LOGFILE must be readable (chmod 644 /var/log/messages).
TEMPFILE=temp.$$
# Create a "unique" temp file name, using process id of the script.
KEYWORD=address
# At logon, the line "remote IP address xxx.xxx.xxx.xxx"
#                 appended to var/log/messages.
ONLINE=22
USER_INTERRUPT=13
CHECK_LINES=100
# How many lines in log file to check.
trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
# Cleans up the temp file if script interrupted by control-c.
echo
while [ $TRUE ]      # Endless loop.
do
  tail -$CHECK_LINES $LOGFILE> $TEMPFILE
  # Saves last 100 lines of system log file as temp file.
  # Necessary, since newer kernels generate many log messages at log on.
  search=`grep $KEYWORD $TEMPFILE`
  # Checks for presence of the "IP address" phrase,
  # indicating a successful logon.
  if [ ! -z "$search" ]     # Quotes necessary because of possible spaces.
  then
    echo "On-line"
    rm -f $TEMPFILE         # Clean up temp file.
```

```
    exit $ONLINE
  else
    echo -n "."              # -n option to echo suppresses newline,
                             # so you get continuous rows of dots.
  fi
  sleep 1
done

# Note: if you change the KEYWORD variable to "Exit",
# this script can be used while on-line to check for an unexpected logoff.
# Exercise: Change the script, as per the above note,
#         and prettify it.

exit 0

# Nick Drage suggests an alternate method:
while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" && exit 0
  echo -n "." # Prints dots (.....) until connected.
  sleep 2
done

# Problem: Hitting Control-C to terminate this process may be insufficient.
#             (Dots may keep on echoing.)
# Exercise: Fix this.

# Stephane Chazelas has yet another alternative:
CHECK_INTERVAL=1
while ! tail -1 "$LOGFILE" | grep -q "$KEYWORD"
do echo -n .
```

```
  sleep $CHECK_INTERVAL
done
echo "On-line"
# Exercise: Discuss the strengths and weaknesses
#           of each of these various approaches.
```

Trap için `DEBUG` argümanı belirtilen eylemin bir komut dosyasında her komutun ardından çalıştırılmasına neden olur. Bu durum, örneğin, değişkenlerin izlenmesine olanak verir.

ÖRNEK 30.7 BİR DEĞİŞKENİ İZLEME

```
#!/bin/bash
trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\""' DEBUG
# Echoes the value of $variable after every command.
variable=29
echo "Just initialized \"\$variable\" to $variable."
let "variable *= 3"
echo "Just multiplied \"\$variable\" by 3."
# The "trap 'commands' DEBUG" construct would be more useful
# in the context of a complex script,
# where placing multiple "echo $variable" statements might be
# clumsy and time-consuming.
# Thanks, Stephane Chazelas for the pointer.
exit 0
```

`trap '' SIGNAL` (iki komşu kesme) komut dosyasının geri kalanı için SIGNAL'i devre dışı bırakır. **`trap SIGNAL`** bir kez daha SIGNAL'in işleyişini geri yükler. Bu bir komut dosyasının kritik bölümünü istenmeyen bir kesmeden korumak için yararlıdır.

```
trap '' 2               # Signal 2 is Control-C, now disabled.
command
command
command
trap 2                  # Reenables Control-C
```

Notlar

[1] Geleneksel olarak, *sinyal 0* çıkmak (exit) için atanır.

BÖLÜM 31 SEÇENEKLER

Seçenekler, kabuk ve/veya komut dosyasının davranışını değiştirmeye yarayan ayarlardır. set komutu bir komut dosyası içinden seçenekleri etkinleştirir. Betikte seçeneklerin etkili olmasını istediğiniz noktada, **set -o seçenek-adı** veya kısaca **set -option-abbrev** komutunu kullanın. Bu iki şekil eşdeğerdir.

```
#!/bin/bash
set -o verbose
# Echoes all commands before executing.
```

```
#!/bin/bash
set -v
# Exact same effect as above.
```

Bir komut dosyası içinde bir seçeneği *devre dışı bırakmak* için**, set +o seçenek-adı** veya **set +option-abbrev** komutunu kullanınız.

```
#!/bin/bash
set -o verbose
# Command echoing on.
command
...
command
set +o verbose
# Command echoing off.
command
# Not echoed.
set -v
# Command echoing on.
command
```

```
...
command
set +v
# Command echoing off.
command
exit 0
```

Bir komut dosyası içinde seçenekleri etkinleştirmenin alternatif bir yöntemi de, hemen *#!* betik başlığını takip ettikten sonra onları belirtmektir.

```
#!/bin/bash -x
#
# Body of script follows.
```

Betik seçeneklerini komut satırından da etkinleştirmek mümkündür. **set** komutu ile çalışmayan bazı seçenekler bu şekilde mevcut hale gelir. Bunların arasında, komut dosyasını etkileşimli çalışmaya zorlayan `-i` seçeneği vardır.

```
bash -v betik-adı
```

```
bash -o verbose betik-adı
```

Aşağıdaki listede bazı kullanışlı seçenekler açıklanmıştır. Bunlar ya kısaltılmış olarak ya da tam adıyla belirtilebilir.

TABLO 31.1 BASH SEÇENEKLERİ

Kısaltma	**Adı**	**Etkisi**
`-C`	noclobber	Dosyaların üzerine yönlendirme ile yazılmasını önler (`>|` tarafından geçersiz kılınabilir)
`-D`	(none)	$ ile başlayan çift tırnak içine alınmış dizeleri listeler, ancak komut dosyası içindeki komutları çalıştırmaz.
`-a`	allexport	Tüm tanımlı değişkenleri dış ortama taşır.
`-b`	notify	Arka planda çalışan işler sonlandığında bildirir (Bir komut dosyasında kullanım alanı çok azdır)
`-c ...`	(none)	… komutlarını okur.
`-f`	noglob	Dosya adı genişlemesini (globbing) devre dışı bırakır.
`-i`	interactive	Komut dosyaları *etkileşimli* modda çalışır.
`-p`	privileged	Komut dosyaları "süper kullanıcı tanıtıcısı" ile çalışır (Dikkat!)
`-r`	restricted	Komut dosyaları *kısıtlı* modda çalışır (bkz. Bölüm 21).
`-u`	nounset	Tanımsız değişkenleri kullanma girişimi hata mesajı verir ve bu, çıkış yapmayı zorlar.
`-v`	verbose	Her komutu çalıştırmadan önce `stdout`'a yazar.
`-x`	xtrace	`-v`'ye benzer, ama komutları açılımıyla yazar.
`-e`	errexit	İlk hatada (sıfır olmayan bir çıkış durumu) komut dosyasını iptal eder.
`-n`	noexec	Komut dosyasındaki komutları okur, ama çalıştırmaz (sözdizimi kontrolü için)
`-s`	stdin	Komutları `stdin`'den okur.
`-t`	(none)	İlk komuttan sonra çıkar.
`-`	(none)	Seçenek sonu bayrağı. Diğer tüm argümanlar konumsal parametrelerdir.
`--`	(none)	Konumsal parametreleri etkinleştirmez. Verilen argümanlar (`-- arg1 arg2`) ise, konumsal parametreler argümanlara ayarlanır.

BÖLÜM 32 SORUNLAR

Ayrılmış sözcük veya karakterlerin değişken adları için atanması.

```
case=value0         # Causes problems.
23skidoo=value1     # Also problems.
# Variable names starting with a digit are reserved by the shell.
# Try _23skidoo=value1. Starting variables with an underscore is o.k.
# However...    using just the underscore will not work.
_=25
echo $_             # $_ is a special variable set to last arg of last
command.
xyz((!*=value2      # Causes severe problems.
```

Bir değişken adı içinde bir tire veya diğer ayrılmış karakterlerin kullanılması.

```
var-1=23
# Use 'var_1' instead.
```

Bir değişken ve bir fonksiyon için aynı adın kullanılması. Bu, komut dosyasının anlaşılmasını zorlaştırabilir.

```
do_something ()
{
  echo "This function does something with \"$1\"."
}
do_something=do_something
do_something do_something
# All this is legal, but highly confusing.
```

Uygunsuz boşluk kullanılması (diğer programlama dillerinin aksine, Bash boşluk hakkında oldukça titiz olabilir).

```
var1 = 23          # 'var1=23' is correct.

# On line above, Bash attempts to execute command "var1"

# with the arguments "=" and "23".

let c = $a - $b   # 'let c=$a-$b' or 'let "c = $a - $b"' are correct.

if [ $a -le 5]    # if [ $a -le 5 ] is correct.

# if [ "$a" -le 5 ] is even better.

# [[ $a -le 5 ]] also works.
```

Başlatılmamış değişkenlerin (kendilerine daha önce bir değer atanmamış değişkenler) değerlerinin "sıfırlandığını" varsaymak. Başlatılmamış bir değişkenin değeri sıfır *değil*, "boş" değeri vardır.

Bir test sırasında = ve -eq operatörlerinin kullanımını karıştırmak. Unutmayın, = kalıp deyim değişkenlerini karşılaştırır, ve -eq tamsayıları karşılaştırma içindir.

```
if [ "$a" = 273 ]             # Is $a an integer or string?

if [ "$a" -eq 273 ]           # If $a is an integer.

# Sometimes you can mix up -eq and = without adverse consequences.

# However...

a=273.0    # Not an integer.

if [ "$a" = 273 ]

then

  echo "Comparison works."

else

  echo "Comparison does not work."

fi        # Comparison does not work.

# Same with a=" 273" and a="0273".

# Likewise, problems trying to use "-eq" with non-integer values.
```

```
if [ "$a" -eq 273.0 ]

then

  echo "a = $a'

fi       # Aborts with an error message.

# test.sh: [: 273.0: integer expression expected
```

Tamsayı ve dize karşılaştırma operatörlerini karıştırmak.

```
#!/bin/bash

# bad-op.sh

number=1

while [ "$number" < 5 ]       # Wrong! Should be while [ "number" -lt 5 ]

do

  echo -n "$number "

  let "number += 1"

done

# Attempt to run this bombs with the error message:

# bad-op.sh: 5: No such file or directory
```

Bazen "test" parantezi (`[]`) içindeki değişkenleri çift tırnak içine almak gerekir. Aksi takdirde beklenmeyen davranışlara neden olabilir. Bkz. Örnek 7.5, Örnek 16.4 ve Örnek 9.6.

Bir komut dosyası içinde verilen komutları çalıştırmak başarısız olabilir, çünkü betik sahibi onları yürütmek için gerekli izinden yoksundur. Eğer bir kullanıcının komut satırından komut çağırması mümkün değilse, betik içine bu komutu koymak da aynı şekilde başarısız olur. Söz konusu komutun özelliklerini değiştirmeyi deneyin, belki de suid bit ayarını (root olarak, elbette) yapabilirsiniz.

Yönlendirme operatörü olarak - kullanma girişimi (bu yanlıştır) genellikle hoş olmayan bir sürprize neden olacaktır.

```
command1 2> - | command2    # Trying to redirect error output of command1
into a pipe...

# ...will not work.

command1 2>& - | command2

# Also futile.Thanks, S.C.
```

Bash sürüm 2+ işlevini kullanmak hata mesajlarını içeren bir kurtarmaya neden olabilir. Eski Linux makinelerinde varsayılan yükleme Bash sürümü 1.xx gibidir.

```
#!/bin/bashminimum_version=2

# Since Chet Ramey is constantly adding features to Bash,

# you may set $minimum_version to 2.XX, or whatever is appropriate.

E_BAD_VERSION=80

if [ "$BASH_VERSION" \< "$minimum_version" ]

then

  echo "This script works only with Bash, version $minimum or greater."

  echo "Upgrade strongly recommended."

  exit $E_BAD_VERSION

fi

...
```

Linux makinede Bash'e özgü işlevleri bir Bourne kabuk betiğinde **(#!/bin/sh)** kullanmak, beklenmeyen davranışlara neden olabilir. Linux sisteminde genellikle **bash** diğer adı olarak **sh** kullanılır, ama bu mutlaka genel bir UNIX makine için geçerli değildir.

DOS-tipi yeni satır (`\r\n`) ile bir komut dosyasını çalıştırmak başarısız olur, çünkü **`#!/bin/bash\r\n`** tanınmıyor; beklenen **`#!/bin/bash\n`** ile aynı *değildir*. Düzeltme için komut dosyası UNIX-tarzı yeni satıra uygun şekilde dönüştürülmelidir.

`#!/bin/sh` başlığı olan bir kabuk betiği tam Bash-uyumluluk modunda çalışmayabilir. Bazı Bash-özel fonksiyonları devre dışı bırakılmış olabilir. Bash'e özgü tüm uzantılar için tam erişim gerektiren komut dosyaları **`#!/bin/bash`** ile başlamalıdır.

Bir komut dosyası dış ortam olarak ana sürece, kabuğa, ya da çevreye değişkenlerini aktarmak zorunda değildir. Biyoloji bilgimize göre, bir çocuk süreç ana süreçten kalıtımı devralır ama tersi doğru değildir.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER
bash$
```

Tabii ki, komut satırında, `$WHATEVER` değeri yoktur.

Değişkenleri altkabukta ayarlayıp manipüle ettikten sonra, aynı değişkenleri altkabuk kapsamı dışında kullanma girişimi hoş olmayan bir sürprize neden olacaktır.

ÖRNEK 32.1 ALTKABUK TUZAKLARI

```
#!/bin/bash
# Pitfalls of variables in a subshell.
outer_variable=outer
echo
echo "outer_variable = $outer_variable"
echo
(
# Begin subshell
echo "outer_variable inside subshell = $outer_variable"
inner_variable=inner      # Set
echo "inner_variable inside subshell = $inner_variable"
outer_variable=inner      # Will value change globally?
echo "outer_variable inside subshell = $outer_variable"
```

```
# End subshell

)

echo

echo "inner_variable outside subshell = $inner_variable"      # Unset.

echo "outer_variable outside subshell = $outer_variable"      # Unchanged.

echo

exit 0
```

echo çıktısının bir read için oluklanması beklenmeyen sonuçlar doğurabilir. Bu senaryoda **read** bir altkabukta çalışıyormuş gibi davranır. Bunun yerine, (Örnek 11.12'deki gibi) set komutunu kullanın.

ÖRNEK 32.2 echo ÇIKTISININ read İÇİN OLUKLANMASI

```
#!/bin/bash

# badread.sh:

#  Attempting to use 'echo and 'read'

#+ to assign variables non-interactively.

a=aaa

b=bbb

c=ccc

echo "one two three" | read a b c

# Try to reassign a, b, and c.

echo "a = $a"      # a = aaa

echo "b = $b"      # b = bbb

echo "c = $c"      # c = ccc

# Reassignment failed.

# ------------------------------

# Try the following alternative.

var=`echo "one two three"`
```

```
set -- $var

a=$1; b=$2; c=$3

echo "-------"

echo "a = $a"      # a = one

echo "b = $b"      # b = two

echo "c = $c"      # c = three

# Reassignment succeeded.

exit 0
```

Komut dosyası içinde "suid" komutlarını kullanmak risklidir, çünkü bu sistem güvenliğini tehlikeye atabilir. [1]

CGI programlama için kabuk komut dosyalarını kullanmak sorunlu olabilir. Betik değişkenleri "tip-güvenli" değildir, ve bu CGI söz konusu olduğunda istenmeyen davranışlara neden olabilir. Ayrıca betikler güvenlik kırıcılara karşı da savunmasız olabilir.

Linux veya BSD sistemleri için yazılmış Bash betiklerinin ticari bir UNIX makinede çalıştırılması için hata düzeltmeleri yapmak gerekebilir. Bu betikler genellikle onların UNIX karşılıklarına göre daha fazla özelliğe sahip olan GNU komutlarını ve filtrelerini içerir. Bu tr gibi bir metin işleme yardımcısında özellikle doğrudur.

Notlar

[1] Betiğin kendisi üzerinde *suid* iznini ayarlamanın bir etkisi yoktur.

BÖLÜM 33 BETİKLER İÇİN STİLLER

Kabuk betiklerini yapılandırılmış ve sistematik bir şekilde yazmayı alışkanlık edinin. Anında ya da zarfın kapağına yazılı betikler için bile olsa, oturmaya ve kodlamaya başlamadan önce düşüncelerinizi planlamak ve organize etmek için birkaç dakika vermeniz yararlı olacaktır.

Bu vesile ile birkaç stil kuralları vardır. Ama bunlar *Kabuk Betikleri İçin Resmi Stiller* olarak tasarlanmamıştır.

33.1 KABUK BETİKLERİ İÇİN RESMİ OLMAYAN STİLLER

- Kodunuza yorumlar ekleyiniz. Bu, kodunuzu başkalarının anlaması (ve takdir etmesi) için daha kolay hale getirir, ve sizin de kodu değiştirmeniz kolaylaşır.

```
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"

# It made perfect sense when you wrote it last year, but now it's a
complete mystery.

# (From Antek Sawicki's "pw.sh" script.)
```

Betiklerinize ve işlevlerine açıklayıcı başlıklar ekleyin.

```
#!/bin/bash

#***********************************************#

# xyz.sh

# written by Bozo Bozeman

# July 05, 2001

# Clean up project files.

#***********************************************#

BADDIR=65                          # No such directory.
```

```
projectdir=/home/bozo/projects    # Directory to clean up.

#-----------------------------------------------------#
# cleanup_pfiles ()
# Removes all files in designated directory.
# Parameter: $target_directory
# Returns: 0 on success, $BADDIR if something went wrong.
#-----------------------------------------------------#
cleanup_pfiles ()
{
  if [ ! -d "$1" ]           # Test if target directory exists.
  then
    echo "$1 is not a directory."
    return $BADDIR
  fi
  rm -f "$1"/*
  return 0     # Success.
}
cleanup_pfiles $projectdir
exit 0
```

#!/bin/bash başlığını herhangi bir yorum satırından önce, betiğin ilk satırı olarak koyduğunuzdan emin olun.

- "Fiziksel bağlantılı" kalıp deyim sabitleri olan "sihirli numaralar" kullanmaktan kaçının, [1] Bunun yerine, anlamlı değişken isimleri kullanın. Bu, komut dosyasının anlaşılmasını kolaylaştırır, ve uygulamayı bozmadan değişiklikler ve güncellemeler yapmanıza izin verir.

```
if [ -f /var/log/messages ]

then

  ...

fi

# A year later, you decide to change the script to check
/var/log/syslog.

# It is now necessary to manually change the script, instance by
instance,

# and hope nothing breaks.

# A better way:

LOGFILE=/var/log/messages      # Only line that needs to be changed.

if [ -f "$LOGFILE" ]

then

  ...

fi
```

- Değişkenler ve fonksiyonlar için açıklayıcı adlar seçin.

```
fl=`ls -al $dirname`               # Cryptic.

file_listing=`ls -al $dirname`     # Better.

MAXVAL=10      # All caps used for a script constant.

while [ "$index" -le "$MAXVAL" ]

...
```

```
E_NOTFOUND=75                              # Uppercase for an errorcode,

                                           # and name begins with "E_".

if [ ! -e "$filename" ]

then

  echo "File $filename not found."

  exit $E_NOTFOUND

fi
```

```
MAIL_DIRECTORY=/var/spool/mail/bozo    # Uppercase for an
environmental variable.

export MAIL_DIRECTORY

GetAnswer ()                           # Mixed case works well for a
function.

{

  prompt=$1

  echo -n $prompt

  read answer

  return $answer

}

GetAnswer "What is your favorite number? "

favorite_number=$?

echo $favorite_number

_uservariable=23                       # Permissable, but not
recommended.

# It's better for user-defined variables not to start with an
underscore.

# Leave that for system variables.
```

- Çıkış kodlarını sistematik ve anlamlı bir şekilde kullanın.

```
E_WRONG_ARGS=65

...

...

exit $E_WRONG_ARGS
```

Aynı zamanda, bkz. Ek C.

- Karmaşık kodları basit modüller şeklinde parçalayın. Uygun olan yerlerde fonksiyon kullanın. Bkz. Örnek 35.4.

- Basit bir yapının yeterli olduğu yerde karmaşık bir yapı kullanmayın.

```
COMMAND
if [ $? -eq 0 ]
...
# Redundant and non-intuitive.

if COMMAND
...
# More concise (if perhaps not quite as legible).
```

Notlar

[1] Bu bağlamda "sihirli sayı"nın, dosya türlerini belirlemek için kullanılan sihirli numaralardan tamamen farklı bir anlamı vardır.

BÖLÜM 34 DİĞER BAZI KOMUTLAR

34.1 ETKİLEŞİMLİ ve ETKİLEŞİMLİ-OLMAYAN KABUK ve KOMUTLAR

Etkileşimli bir kabuk, komutları `tty` üzerinden girilen kullanıcı girdisinden okur. Başka şeylerin yanı sıra, böyle bir kabuk, aktivasyon başlatma dosyalarını okur, bir istem görüntüler ve varsayılan olarak iş denetimi sağlar. Kullanıcı kabuk ile iletişim *kurabilir*.

Bir komut dosyası etkileşimli olmayan bir kabukta çalıştırılır. Ancak, komut dosyası `tty`'a erişebilir. Bir komut dosyası içinde etkileşimli kabuk taklit edilebilir.

```
#!/bin/bash

MY_PROMPT='$ '

while :

do

  echo -n "$MY_PROMPT"

  read line

  eval "$line"

done

exit 0

# This example script, and much of the above explanation supplied by

# Stephane Chazelas (thanks again).
```

Şimdi genellikle `read` komutu ile kullanıcı girişi gerektiren bir etkileşimli komut dosyasını ele alalım (bkz. Örnek 11.2). "Gerçek hayat" aslında biraz daha anlaşılmazdır. Şimdilik, etkileşimli komut dosyasının bir `tty`'a bağlı olduğunu varsayalım. Kullanıcı bu komut dosyasını konsoldan veya `xterm`'den çağırıyor olmalıdır.

`init` ve `startup` komut dosyalarının etkileşimli olmaması zorunludur, çünkü otomatik çalışmalıdır. Birçok idari ve sistem bakım betikleri de aynı şekilde etkileşimli değildir. Değişmeyen tekrarlanan görevler etkileşimli olmayan betikler ile otomasyon içine girerler.

Etkileşimli olmayan komutlar, arka planda çalıştırılabilir, ama etkileşimli olanlar gelmeyen girdiyi bekledikleri zaman askıda kalır. Bir **expect** betiği ya da, arka planda çalışan bir iş olan etkileşimli betiğe girdi olarak beslenen, gömülü belge dokümanına sahipseniz, bu zorluğun üstesinden gelebilirsiniz. En basit durumda, bir **read** komutuna girdi sağlamak için bir dosyayı yönlendiriniz (**read değişken <dosya**). Bu özel geçici çözümler genel amaçlı komut dosyalarının, etkileşimli olan veya olmayan modda çalışmasını mümkün kılar.

Bir komut dosyası, etkileşimli kabukta çalışıyor olup olmadığını test etmek isteyebilir. Bunu sadece hızlı değişken `$PS1` ayarını bularak yapabilir. (Kullanıcı girdisi de istenmişse, komut dosyasının bu istemi görüntülemesi şarttır.)

```
if [ -z $PS1 ] # no prompt?
then
  # non-interactive
  ...
else
  # interactive
  ...
fi
```

Alternatif olarak, komut dosyası `$-` bayrağında `"i"` seçeneğinin olup olmadığını test edebilir.

```
case $- in
*i*)       # interactive shell
;;
*)         # non-interactive shell
;;
# (Courtesy of "UNIX F.A.Q.," 1993)
```

Komut dosyaları `-i` seçeneği ile veya **`#!/bin/bash -i`** başlığı ile etkileşimli modda çalışmak zorunda bırakılabilir. Bunun düzensiz betik davranışlarına neden olabileceğini veya herhangi bir hata mevcut olmasa bile hata mesajları gösterebileceğini unutmayın.

34.2 KABUK SARICILAR

"Sarıcı" işlevini şöyle açıklayabiliriz: Bir sistem komutu veya yardımcı programını kabuk betiğine gömerken, o komut için geçirilen bir dizi parametreyi kaydetmektir. Karmaşık bir işlevselliğin yerine gelmesi, sarıcıların mümkün kıldığı komut satırları ile kolaylaşır. Bu, özellikle sed ve awk için yararlıdır.

Bir **sed** veya **awk** komut dosyası, normal olarak komut satırından **`sed -e`** *`'komutlar'`* veya **`awk`** *`'komutlar'`* ifadesi ile çağrılır. Bir Bash komut dosyası içine böyle bir komut dosyasını gömerek onu daha basit ve "tekrar kullanılabilir" şekilde çağırabilirsiniz. Bir dizi **sed** komutunun çıktısını **awk** için oluklayarak **sed** ve **awk** işlevselliğini birleştirebilirsiniz. Komut satırından yeniden yazılmaksızın yürütülebilir ve kaydedilmiş olan bu dosyayı, tekrar tekrar orijinal veya değiştirilmiş biçiminde çağırabilirsiniz.

ÖRNEK 34.1 KABUK SARICISI

```
#!/bin/bash

# This is a simple script that removes blank lines from a file.

# No argument checking.

# Same as

#       sed -e '/^$/d' filename

# invoked from the command line.

sed -e /^$/d "$1"

# The '-e' means an "editing" command follows (optional here).

# '^' is the beginning of line, '$' is the end.

#  This match lines with nothing between the beginning and the end,
```

```
#+ blank lines.
#  The 'd' is the delete command.
#  Quoting the command-line arg permits
#+ whitespace and special characters in the filename.
exit 0
```

ÖRNEK 34.2 BİRAZ DAHA KARMAŞIK KABUK SARICI

```
#!/bin/bash
# "subst", a script that substitutes one pattern for
# another in a file,
# i.e., "subst Smith Jones letter.txt".
ARGS=3
E_BADARGS=65                # Wrong number of arguments passed to script.

if [ $# -ne "$ARGS" ]
# Test number of arguments to script (always a good idea).
then
  echo "Usage: `basename $0` old-pattern new-pattern filename"
  exit $E_BADARGS
fi
old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
  file_name=$3
else
  echo "File \"$3\" does not exist."
```

```
exit $E_BADARGSfi

# Here is where the heavy work gets done.
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# 's' is, of course, the substitute command in sed,
# and /pattern/ invokes address matching.
# The "g", or global flag causes substitution for *every*
# occurence of $old_pattern on each line, not just the first.
# Read the literature on 'sed' for a more in-depth explanation.
exit 0
# Successful invocation of the script returns 0.
```

ÖRNEK 34.3 BİR awk KOMUT DOSYASI ETRAFINDA KABUK SARICI

```
#!/bin/bash
# Adds up a specified column (of numbers) in the target file.
ARGS=2
E_WRONGARGS=65
if [ $# -ne "$ARGS" ]      # Check for proper no. of command line args.
then
  echo "Usage: `basename $0` filename column-number"
  exit $E_WRONGARGS
fi
filename=$1
column_number=$2
# Passing shell variables to the awk part of the script is a bit tricky.
# See the awk documentation for more details.
# A multi-line awk script is invoked by awk ' ..... '
```

```
# Begin awk script.

# ----------------------------

awk '

{ total += $'"${column_number}"'

} END {

print total

}

' "$filename"

# ----------------------------

# End awk script.

# It may not be safe to pass shell variables to an embedded awk script,

# so Stephane Chazelas proposes the following alternative:

# --------------------------------------

# awk -v column_number="$column_number" '

# { total += $column_number

# }

# END {

# print total

# }' "$filename"

# --------------------------------------

exit 0
```

Komut dosyasında tek seferde her şeyi yapan bir araç, sanki İsviçre çakısı, geliştirilecekse Perl'i önerebiliriz. Perl, **sed** ve **awk** yeteneklerini birleştirir, ve **C**'nin geniş bir alt-kümesini birbirine ekleyerek önyükler. Bu modüler yaklaşımın nesneye dayalı programcılar için kabuk betiklerini yazmayı değiştireceğini söyleyen bazı iddialar bulunmaktadır. Yazar bu konuda şüpheci olsa da, kısa Perl komut dosyalarının, kendilerini kabuk betikleri içine gömerek varolabilecekleri de bir gerçektir.

ÖRNEK 34.4 BASH KOMUT DOSYASINA GÖMÜLÜ PERL

```
#!/bin/bash
# Shell commands may precede the Perl script.
echo "This precedes the embedded Perl script within \"$0\"."
echo "============================================================="

perl -e 'print "This is an embedded Perl script.\n";'
# Like sed, Perl also uses the "-e" option.
echo "============================================================="
echo "However, the script may also contain shell and system commands."
exit 0
```

Bash ve Perl komut dosyalarını aynı dosya içinde birleştirmek mümkündür. Yürütülme sırasında ya Bash ya da Perl komut dosyasından biri çağrılmalıdır.

ÖRNEK 34.5 BİRLEŞTİRİLMİŞ BASH ve PERL KOMUT DOSYALARI

```
#!/bin/bash
# bashandperl.sh
echo "Greetings from the Bash part of the script."
# More Bash commands may follow here.
exit 0
# End of Bash part of the script.
# =======================================================
#!/usr/bin/perl
# This part of the script must be invoked with -x option.
print "Greetings from the Perl part of the script.\n";
# More Perl commands may follow here.
# End of Perl part of the script.
```

```
bash$ bash bashandperl.sh

Greetings from the Bash part of the script.

bash$ perl -x bashandperl.sh

Greetings from the Perl part of the script.
```

34.3 TESTLER ve KARŞILAŞTIRMALAR: ALTERNATİFLER

Testler için, `[[]]` yapısı `[]` yapısından daha uygun olabilir. Aynı şekilde, aritmetik karşılaştırmalar için `(())` yapısı yararlı olabilir.

```
a=8

# All of the comparisons below are equivalent.

test "$a" -lt 16 && echo "yes, $a < 16"   # "and list"

/bin/test "$a" -lt 16 && echo "yes, $a < 16"

[ "$a" -lt 16 ] && echo "yes, $a < 16"

[[ $a -lt 16 ]] && echo "yes, $a < 16"    # Quoting variables within

(( a < 16 )) && echo "yes, $a < 16"       # [[ ]] and (( )) not necessary.

city="New York"

# Again, all of the comparisons below are equivalent.

test "$city" \< Paris && echo "Yes, Paris is greater than $city"   #
Greater ASCII order.

/bin/test "$city" \< Paris && echo "Yes, Paris is greater than $city"

[ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"

[[ $city < Paris ]] && echo "Yes, Paris is greater than $city"     # Need
not quote $city.

# Thank you, S.C.
```

34.4 OPTİMİZASYONLAR

Kabuk betikleri karmaşık sorunlar için düşünülen hızlı ve mükemmel olmayan çözümlerdir. Bu nedenle, betiklerin hızını optimize etmek çoğu zaman bir sorun değildir. Bir komut dosyasının önemli bir görevi yürütmekte olduğunu düşünün, iyi yapıyor, ama çok yavaş çalışıyor. Derlenmiş bir dilde yeniden yazmak kolay olmayabilir. En basit düzeltme, komut dosyasını yavaşlatan unsurları yeniden yazmak olacaktır. Kod optimizasyonunu düzenleyen değişmezleri kabuk betiklerinin daha alt düzeylerine uygulamak mantıklı mıdır?

Komuttaki döngüleri kontrol etmelisiniz. Tekrarlanan işlemler tarafından tüketilen zaman hızlı bir şekilde artar. Mümkünse hiç değilse, döngüler içindeki zaman alıcı işlemleri çıkarın.

Sistem komutları yerine yerleşik komutları kullanın. Yerleşikler daha hızlı yürütülür ve çağrıldığında genellikle altkabuğu başlatmazlar.

Hesaplama-yoğun komutların profili için time ve times araçlarını kullanın. Zaman-kritik kod bölümlerini yeniden yazmak için C dilini, hatta çevirici programı düşünün.

Dosya Girdi/Çıktıyı en aza indirmeye çalışın. Bash, dosya işlemede özellikle verimli değildir, bu nedenle komut dosyası içinde bu amaca daha uygun araçlar kullanmayı, awk ve Perl gibi, düşünün.

Yapılandırılmış, tutarlı şekilde komut yazınız, böylece kod yeniden düzenlenebilir ve gerektiğinde sıkıştırılabilir. Üst düzey diller için geçerli optimizasyon tekniklerinin bazıları komut dosyaları için de çalışabilir, ancak döngüleri çözme gibi bazıları çoğu zaman etkisizdir. Her şeyden önce, sağduyunuzu kullanın.

Optimizasyon bir komut dosyasının yürütme zamanını büyük ölçüde azaltabilir. Bkz. Örnek 12.32.

34.5 BAZI İPUÇLARI

- Belirli bir oturum sırasında veya bir dizi oturumlar boyunca hangi kullanıcı komut dosyalarının kullanıldığının bir kaydını tutmak için takip etmek istediğiniz her komut dosyası için aşağıdaki satırları ekleyin. Bu betik isimlerinin ve çağrılma zamanlarının devamlı bir dosya kaydını tutacaktır.

```
# Append (>>) following to end of each script tracked.

date>> $SAVE_FILE          # Date and time.

echo $0>> $SAVE_FILE       # Script name.

echo>> $SAVE_FILE          # Blank line as separator.

# Of course, SAVE_FILE defined and exported as environmental variable
in ~/.bashrc

# (something like ~/.scripts-run)
```

- >> operatörü bir dosyaya satır ekler. Bir satırı varolan bir dosyanın *önüne* eklemek istiyorsanız?

```
file=data.txt
title="***This is the title line of data text file***"

echo $title | cat - $file >$file.new
# "cat -" concatenates stdout to $file.
# End result is
#+ to write a new file with $title appended at *beginning*.
```

  Tabii ki, sed bunu yapabilir.

- Bir kabuk betiği, başka bir kabuk betiği içine gömülü bir komut olarak hareket edebilir, bu bir *Tcl* hatta *wish* komut dosyası veya bir Makefile olabilir. Bu bir C programında dış kabuk komutu olarak çağrılabilir, `system()` çağrısı, yani `system("betik_adı");` kullanarak.

- En sevdiğiniz ve en yararlı tanımları ve işlevleri içeren dosyaları bir araya koyun. Gerekirse, bu "kütüphane dosyaları"nın bir veya daha fazlasını betik içinde ya dot,(.) ya da source komutu ile "dahil" ediniz.

```
# SCRIPT LIBRARY
# ------ -------
# Note:
# No "#!" here.
# No "live code" either.

# Useful variable definitions
ROOT_UID=0                    # Root has $UID 0.
E_NOTROOT=101                 # Not root user error.
MAXRETVAL=256                 # Maximum (positive) return value of a
function.
SUCCESS=0
FAILURE=-1

# Functions
Usage ()                      # "Usage:" message.
{
  if [ -z "$1" ]              # No arg passed.
  then
    msg=filename
  else
    msg=$@
  fi
  echo "Usage: `basename $0` "$msg""
}

Check_if_root ()              # Check if root running script.
{                             # From "ex39.sh" example.
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Must be root to run this script."
    exit $E_NOTROOT
  fi
}

CreateTempfileName ()         # Creates a "unique" temp filename.
{                             # From "ex51.sh" example.
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}

isalpha2 ()                   # Tests whether *entire string* is
alphabetic.
{                             # From "isalpha.sh" example.
  [ $# -eq 1 ] || return $FAILURE
  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                        # Thanks, S.C.
}

abs ()                        # Absolute value.
```

```
{                             # Caution: Max return value = 256.
  E_ARGERR=-999999
  if [ -z "$1" ]              # Need arg passed.
  then
    return $E_ARGERR          # Obvious error value returned.
  fi

  if [ "$1" -ge 0 ]           # If non-negative,
  then
    absval=$1                 # stays as-is.
  else                        # Otherwise,
    let "absval = (( 0 - $1 ))"   # change sign.
  fi
return $absval
}

tolower ()                    #  Converts string(s) passed as
argument(s)
{                             #+ to lowercase.
  if [ -z "$1" ]              #  If no argument(s) passed,
  then                        #+ send error message
    echo "(null)"             #+ (C-style void-pointer error message)
    return                    #+ and return from function.
  fi

  echo "$@" | tr A-Z a-z
  # Translate all passed arguments ($@).
  return

# Use command substitution to set a variable to function output.
# For example:
#      oldvar="A seT of miXed-caSe LEtTerS"
#      newvar=`tolower "$oldvar"`
#      echo "$newvar"       # a set of mixed-case letters
#
# Exercise: Rewrite this function to change lowercase passed
argument(s)
#           to uppercase ... toupper() [easy].
}
```

- Komut dosyalarında netlik ve okunabilirliği artırmak için özel amaçlı yorum başlıkları kullanın.

```
## Caution.

rm -rf *.zzy          ## The "-rf" options to "rm" are very dangerous,

                         ##+ especially with wildcards.

#+ Line continuation.
```

```
#  This is line 1

#+ of a multi-line comment,

#+ and this is the final line.

#* Note.

#o List item.

#> Another point of view.

while [ "$var1" != "end" ]   #> while test "$var1" != "end"
```

- $? çıkış durumu değişkenini kullanarak, bir komut dosyası bir parametrenin sadece basamak içerip içermediğini test edebilir. Böylece parametre bir tamsayı gibi değerlendirilebilir.

```
#!/bin/bash

SUCCESS=0

E_BADINPUT=65

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null

# An integer is either equal to 0 or not equal to 0.

# 2>/dev/null suppresses error message.

if [ $? -ne "$SUCCESS" ]

then

  echo "Usage: `basename $0` integer-input"

  exit $E_BADINPUT

fi

let "sum = $1 + 25"           # Would give error if $1 not integer.

echo "Sum = $sum"

# Any variable, not just a command line parameter, can be tested this
way.

exit 0
```

- Fonksiyon dönüş değerleri için 0-255 aralığı ciddi bir sınırlamadır. Global değişkenler ve diğer geçici çözümler genellikle sorunludur. Bir fonksiyonun betiğin ana gövdesine

bir değer geçirmesi için alternatif bir yöntem de fonksiyonun `stdout` için bir "dönüş değeri" yazdıktan sonra, bunu bir değişkene atamaktır.

ÖRNEK 34.6 DÖNÜŞ DEĞERİ

```
#!/bin/bash

# multiplication.sh

multiply ()                     # Multiplies params passed.

{                               # Will accept a variable number of
args.

  local product=1

  until [ -z "$1" ]             # Until uses up arguments passed...

  do

    let "product *= $1"

    shift

  done

  echo $product                 # Will not echo to stdout,

}                               #+ since this will be assigned to a
variable.

mult1=15383; mult2=25211

val1=`multiply $mult1 $mult2`

echo "$mult1 X $mult2 = $val1"

                                # 387820813

mult1=25; mult2=5; mult3=20

val2=`multiply $mult1 $mult2 $mult3`

echo "$mult1 X $mult2 X $mult3 = $val2"

                                # 2500mult1=188;

mult2=37; mult3=25; mult4=47

val3=`multiply $mult1 $mult2 $mult3 $mult4`

echo "$mult1 X $mult2 X $mult3 X mult4 = $val3"

                                # 8173300

exit 0
```

Aynı teknik alfa-numerik karakter dizeleri için de çalışır. Bunun anlamı, bir fonksiyon sayısal olmayan bir değer döndürebilir.

```
capitalize_ichar ()             # Capitalizes initial character
{                               #+ of argument string(s) passed.
  string0="$@"                  # Accepts multiple arguments.
  firstchar=${string0:0:1}      # First character.
  string1=${string0:1}          # Rest of string(s).
  FirstChar=`echo "$firstchar" | tr a-z A-Z`
                                # Capitalize first character.
  echo "$FirstChar$string1"     # Output to stdout.
}
newstring=`capitalize_ichar "each sentence should start with a
capital letter."`
echo "$newstring" # Each sentence should start with a capital letter.
```

Bu yöntemle bir fonksiyonun birden çok değer "dönmesi" bile mümkündür.

ÖRNEK 34.7 DAHA FAZLA: DÖNÜŞ DEĞERİ

```
#!/bin/bash
# sum-product.sh
# A function may "return" more than one value.
sum_and_product ()  # Calculates both sum and product of passed args.
{
  echo $(( $1 + $2 )) $(( $1 * $2 ))
# Echoes to stdout each calculated value, separated by space.
}
Echo
echo "Enter first number "
```

```
read first

echo

echo "Enter second number "

read second

echo

retval=`sum_and_product $first $second` # Assigns output of function.

sum=`echo "$retval" | awk '{print $1}'`      # Assigns first field.

product=`echo "$retval" | awk '{print $2}'` # Assigns second field.

echo "$first + $second = $sum"

echo "$first * $second = $product"

echo

exit 0
```

- Çantamızdaki bir sonraki teknik, bir fonksiyona bir dizi geçirme ve daha sonra betiğin ana gövdesine bu diziyi geri "döndürme"dir.

Dizi geçirme, dizinin boşluk ile ayrılmış elemanlarının komut değişimi ile bir değişkene yüklenmesini içerir. Dizinin fonksiyonun "dönüş değeri" olarak geri getirilmesi için, önceden bahsedilen bir strateji olarak, dizi fonksiyon içinde *echo* edildikten sonra, komut değişimini ve (…) operatörü çağrılarak bir diziye atanır.

ÖRNEK 34.8 DİZİLERİN FONKSİYONA GEÇİRİLMESİ ve DÖNÜŞ DEĞERİ OLARAK DİZİ

```
#!/bin/bash

# array-function.sh: Passing an array to a function and...

#                "returning" an array from a function

Pass_Array ()

{

  local passed_array                    # Local variable.

  passed_array=( `echo "$1"` )
```

```
  echo "${passed_array[@]}"
  # List all the elements of the new array
  #+ declared and set within the function.
}

original_array=( element1 element2 element3 element4 element5 )
echo
echo "original_array = ${original_array[@]}"
#                                  List all elements of original array.
# This is the trick that permits passing an array to a function.
# **********************************
argument=`echo ${original_array[@]}`
# **********************************
#  Pack a variable
#+ with all the space-separated elements of the original array.
#
# Note that attempting to just pass the array itself will not work.

# This is the trick that allows grabbing an array as a "return
#+ value".
# ****************************************
returned_array=( `Pass_Array "$argument"` )
# ****************************************
# Assign 'echoed' output of function to array variable.
echo "returned_array = ${returned_array[@]}"
echo "============================================================="
# Now, try it again,
#+ attempting to access (list) the array from outside the function.
Pass_Array "$argument"
#  The function itself lists the array, but...
```

```
#+ accessing the array from outside the function is forbidden.

echo "Passed array (within function) = ${passed_array[@]}"

# NULL VALUE since this is a variable local to the function.

echo

exit 0
```

Fonksiyonlara dizi geçirmenin daha ayrıntılı örneği için bkz. Örnek A.10.

- Çift parantez yapısını kullanarak, değişkenleri ayarlamak ve artırmak için ve for, while döngüleri içinde C-benzeri sözdizimi kullanmak mümkündür. bkz. Örnek 10.12, Örnek 10.17.

- Yararlı bir başka komut dosyası tekniği de, art arda filtrenin çıktısını (oluk ile) farklı argümanlar ve/veya seçenekler ile *tekrar aynı filtreye* beslemektir. Bu özellikle **tr** için uygundur.

```
# From "wstrings.sh" example.
wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

- Kod bloklarını yorum şekline çevirmek için "anonim here belgeleri"ni kullanın. Böylece, her satırı ayrı ayrı # ile yorum şekline çevirmek zorunda kalmazsınız. Bkz. Örnek 17.10.

- Run-parts komutu, özellikle cron ya da at ile beraber kullanıldığında, bir dizi komut dosyasını sırayla çalıştırmak için kullanışlıdır.

- Bir kabuk komut dosyasından X Windows parçacığı çağırmak iyi olurdu. Bunu yapacağını ifade eden birkaç paket vardır, *Xscript*, *Xmenu*, ve *widtools*. Bunlardan ilk ikisi artık sürdürülmüyor. Neyse ki, buradan widtools'u elde etmek mümkündür.

- *Widtools* (Parçacık araçları) paketi *XForms* kütüphanesinin yüklü olmasını gerektirir. Ayrıca, paket tipik bir Linux sistemi üzerine kurulmadan önce Makefile makul şekilde düzenlenmelidir. Son olarak, sunulan altı parçacıktan üçü çalışmaz haldedir (ve aslında, segfault vermektedir).

Parçacıklar ile daha etkin komut dosyaları yazmak için, *Tk* veya *wish* (*Tcl* türevleri), *PerlTk* (*Tk* uzantılı *Perl*), *tksh* (*Tk* uzantılı *ksh*), *XForms4Perl* (*XForms* uzantılı *Perl*), *Gtk-Perl* (*Gtk* uzantılı *Perl*), veya *PyQt* (*Qt* uzantıları ile *Python*) kütüphanelerine başvurunuz.

34.6 DÜŞÜNÜLMESİ GEREKLİ NOKTALAR

Bir komut dosyası özyinelemeli olarak kendisini çağırabilir mi? Gerçekten.

ÖRNEK 34.9 KENDİSİNİ ÇAĞIRAN (İŞE YARAMAZ) BİR KOMUT DOSYASI

```
#!/bin/bash

# recurse.sh

# Can a script recursively call itself?

# Yes, but is this of any practical use?

# (See the following script.)

RANGE=10

MAXVAL=9

i=$RANDOM

let "i %= $RANGE"                # Generate a random number between 0 and
$MAXVAL.

if [ "$i" -lt "$MAXVAL" ]

then

  echo "i = $i"

  ./$0                           # Script recursively spawns a new instance of
itself.

fi                               # Each child script does the same, until

                                 #+ a generated $i equals $MAXVAL.

# Using a "while" loop instead of an "if/then" test causes problems.

# Explain why.

exit 0
```

ÖRNEK 34.10 KENDİSİNİ ÇAĞIRAN (YARARLI) BİR KOMUT DOSYASI

```
#!/bin/bash
# pb.sh: phone book
# Written by Rick Boivie, and used with permission.
# Modifications by document author.
MINARGS=1                       # Script needs at least one argument.
DATAFILE=./phonebook
PROGNAME=$0
E_NOARGS=70                     # No arguments error.
if [ $# -lt $MINARGS ];
then
  echo "Usage: "$PROGNAME" data"
  exit $E_NOARGS
fi
if [ $# -eq $MINARGS ];
then
  grep $1 "$DATAFILE"
else
  ( shift; "$PROGNAME" $* ) | grep $1
  # Script recursively calls itself.
fi
exit 0                          # Script exits here.
                                # It's o.k. to put non-hashmarked comments
                                #+ and data after this point.
# ----------------------------------------------------------------------
# Sample "phonebook" datafile:
John Doe 1555 Main St., Baltimore, MD 21228 (410) 222-3333
Mary Moe 9899 Jones Blvd., Warren, NH 03787 (603) 898-3232
Richard Roe 856 E. 7th St., New York, NY 10009 (212) 333-4567
```

```
Sam Roe 956 E. 8th St., New York, NY 10009 (212) 444-5678

Zoe Zenobia 4481 N. Baker St., San Franciso, SF 94338 (415) 501-1631

#----------------------------------------------------------------------

$bash pb.sh Roe

Richard Roe 856 E. 7th St., New York, NY 10009 (212) 333-4567

Sam Roe 956 E. 8th St., New York, NY 10009 (212) 444-5678


$bash pb.sh Roe Sam

Sam Roe 956 E. 8th St., New York, NY 10009 (212) 444-5678


#  When more than one argument passed to script,

#+ prints *only* the line(s) containing all the arguments.
```

Çok seviyeli bir özyineleme, komut dosyasının yığın alanı tüketerek bölütleme bozukluğuna (segfault) neden olur.

34.7 GÜVENLİK KONULARI

Komut dosyası güvenliği hakkında kısa bir uyarı uygundur. Bir kabuk betiği *solucan*, *trojan*, hatta bir *virüs* içerebilir. Bu nedenle, güvenilir bir kaynaktan elde etmediğiniz sürece ve zararlı hiçbir şey yapmadığından emin olana dek dikkatlice analiz etmedikçe, hiçbir komut dosyasını kök (root) olarak asla çalıştırmayınız (`/etc/rc.d` içinde sistem başlatma betiklerine de eklemeyiniz).

M. Douglas McIlroy, Tom Duff ve Fred Cohen dahil olmak üzere Bell Labs ve diğer sitelerdeki çeşitli araştırmacılar kabuk virüslerinin etkilerini araştırdılar. Onlar, acemi biri için bile bir "yavru komut dosyası" yazmanın çok kolay olduğu sonucuna vardılar. [1]

Komut dosyası yazmayı öğrenmek için bir başka neden daha vardır. Komut dosyalarına bakmak ve anlamak, sisteminize izinsiz girilmesinden veya hasar görmesinden sizi koruyabilir.

Notlar

[1] Bkz. "Unix Shell Script Malware" başlıklı Marius van Oers'in makalesi, ve aynı zamanda kaynakçada Denning referansı.

34.8 TAŞINABİLİRLİK SORUNLARI

Bu kitap, özel olarak Bash komut dosyalarının GNU/Linux sisteminde yazılması ile ilgilenmektedir. **sh** ve **ksh** kullanıcıları da burada, kendilerini ilgilendiren birçok bilgiyi bulabilirler.

İşin ilginç yanı, çok çeşitli kabuklar ve betik dilleri POSIX 1003.2 standardında kesişiyor gibi görünmektedir. `--posix` seçeneği ile Bash'i çağırdığınızda veya bir komut dosyası başında bir **set -o posix** eklediğinizde bu, Bash'in standarda çok yakından uygun hareket etmesine neden olur. Hatta bu özellik eksik olsa bile, birçok Bash betiği **ksh** altında kendisi olduğu gibi çalışır, ve tam tersi de doğru, çünkü Chet Ramey en son Bash sürümlerine **ksh** özelliklerini taşımak ile meşguldür.

Bir ticari UNIX makinada, GNU-belirli özellikler taşıyan standart komutları kullanan betikler çalışmayabilir. Bu son birkaç yıl içinde daha az sorun haline gelmiştir, çünkü GNU programları "büyük demir" UNIX üzerinde bile, patentli, firmaya özel benzerlerini yerinden etmiştir. Orijinal UNIX yardımcı araçlarından birçoğu için en son sürüm kaynak kodlar sadece eğilimi hızlandıracaktır.

34.9 WINDOWS ALTINDA KABUK BETİKLEME

Diğer işletim sistemini çalıştıran kullanıcılar bile, UNIX-benzeri komut dosyalarını çalıştırabilirler ve bu nedenle bu kitabın içindeki birçok dersten yararlanabilirler. Cygnus'tan gelen Cygwin paketi ve Mortice Kern Associates'den gelen MKS yardımcı programları Windows altında kabuk betikleme yeteneklerini eklemişlerdir.

BÖLÜM 35 BASH, SÜRÜM 2

Makineniz üzerinde çalışan Bash'in güncel sürümü aslında versiyon 2.XX.Y'dir.

```
bash$ echo $BASH_VERSION

2.05.8(1)-release
```

Klasik Bash betik dilinin bu güncelleştirmesi, dizi değişkenlerini [1], dize ve parametre genişletmeyi ve daha iyi bir yöntem olarak dolaylı değişken referanslarını eklemiştir.

ÖRNEK 35.1 DİZE GENİŞLETME

```
#!/bin/bash

# String expansion.

# Introduced with version 2 of Bash.

# Strings of the form $'xxx'

# have the standard escaped characters interpreted.

echo $'Ringing bell 3 times \a \a \a'

echo $'Three form feeds \f \f \f'

echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'

exit 0
```

ÖRNEK 35.2 DOLAYLI DEĞİŞKEN REFERANSLARI – YENİ BİR YOL

```
#!/bin/bash

# Indirect variable referencing.

# This has a few of the attributes of references in C++.

a=letter_of_alphabet

letter_of_alphabet=z
```

```
echo "a = $a"           # Direct reference.

echo "Now a = ${!a}"   # Indirect reference.

# The ${!variable} notation is greatly superior to the old "eval
var1=\$$var2"

echo

t=table_cell_3

table_cell_3=24

echo "t = ${!t}"       # t = 24

table_cell_3=387

echo "Value of t changed to ${!t}"    # 387

# This is useful for referencing members of an array or table,

# or for simulating a multi-dimensional array.

# An indexing option would have been nice (sigh).

exit 0
```

ÖRNEK 35.3 DOLAYLI DEĞİŞKEN REFERANSI KULLANARAK BASİT BİR VERİTABANI UYGULAMASI

```
#!/bin/bash

# resistor-inventory.sh

# Simple database application using indirect variable referencing.

# ============================================================== #

# Data

B1723_value=470                         # ohms

B1723_powerdissip=.25                   # watts

B1723_colorcode="yellow-violet-brown" # color bands

B1723_loc=173                           # where they are

B1723_inventory=78                      # how many

B1724_value=1000

B1724_powerdissip=.25
```

```
B1724_colorcode="brown-black-red"

B1724_loc=24N

B1724_inventory=243

B1725_value=10000

B1725_powerdissip=.25

B1725_colorcode="brown-black-orange"

B1725_loc=24N

B1725_inventory=89

# ============================================================= #

echo

PS3='Enter catalog number: '

echo


select catalog_number in "B1723" "B1724" "B1725"

do

  Inv=${catalog_number}_inventory

  Val=${catalog_number}_value

  Pdissip=${catalog_number}_powerdissip

  Loc=${catalog_number}_loc

  Ccode=${catalog_number}_colorcode

  echo

  echo "Catalog number $catalog_number:"

  echo "There are ${!Inv} of [${!Val} ohm / ${!Pdissip} watt] resistors in
stock."

  echo "These are located in bin # ${!Loc}."

  echo "Their color code is \"${!Ccode}\"."

  break

done

echo; echo

# Exercise:

# --------
```

```
# Rewrite this script using arrays, rather than indirect variable
#+ referencing.
# Which method is more straightforward and intuitive?
# Notes:
# -----
#  Shell scripts are inappropriate for anything except the most simple
#+ database applications, and even then it involves workarounds and
#+ kludges.
#  Much better is to use a language with native support for data
#+ structures,
#+ such as C++ or Java (or even Perl).
exit 0
```

ÖRNEK 35.4 DİZİLERİ ve DİĞER ÇEŞİTLİ HİLELERİ KULLANARAK BİR İSKAMBİL DESTESİNDEN RASGELE DÖRT EL ÇEKMEK

```
#!/bin/bash
# May need to be invoked with #!/bin/bash2 on older machines.
# Cards:
# deals four random hands from a deck of cards.
UNPICKED=0
PICKED=1
DUPE_CARD=99
LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUIT=13
CARDS=52

declare -a Deck
```

```
declare -a Suits

declare -a Cards

# It would have been easier and more intuitive

# with a single, 3-dimensional array.

# Perhaps a future version of Bash will support multidimensional arrays.

initialize_Deck ()

{

i=$LOWER_LIMIT

until [ "$i" -gt $UPPER_LIMIT ]

do

  Deck[i]=$UNPICKED      # Set each card of "Deck" as unpicked.

  let "i += 1"

done

echo

}


initialize_Suits ()

{

Suits[0]=C     #Clubs

Suits[1]=D     #Diamonds

Suits[2]=H     #Hearts

Suits[3]=S     #Spades

}


initialize_Cards ()

{

Cards=(2 3 4 5 6 7 8 9 10 J Q K A)

# Alternate method of initializing an array.

}
```

```
pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS"
if [ "${Deck[card_number]}" -eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
  return $card_number
else
  return $DUPE_CARD
fi
}

parse_card ()
{
number=$1
let "suit_number = number / CARDS_IN_SUIT"
suit=${Suits[suit_number]}
echo -n "$suit-"
let "card_no = number % CARDS_IN_SUIT"
Card=${Cards[card_no]}
printf %-4s $Card
# Print cards in neat columns.
}
seed_random ()           # Seed random number generator.
{
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}
```

```
deal_cards ()
{
echo
cards_picked=0
while [ "$cards_picked" -le $UPPER_LIMIT ]
do
  pick_a_card
  t=$?
  if [ "$t" -ne $DUPE_CARD ] then
    parse_card $t
    u=$cards_picked+1
    # Change back to 1-based indexing (temporarily).
    let "u %= $CARDS_IN_SUIT"
    if [ "$u" -eq 0 ]          # Nested if/then condition test.
    then
      echo
      echo
    fi
    # Separate hands.
    let "cards_picked += 1"
  fi
done
echo
return 0
}

# Structured programming:
# entire program logic modularized in functions.
#================
```

```
seed_random

initialize_Deck

initialize_Suits

initialize_Cards

deal_cards

exit 0

#================


# Exercise 1:

# Add comments to thoroughly document this script.

# Exercise 2:

# Revise the script to print out each hand sorted in suits.

# You may add other bells and whistles if you like.

# Exercise 3:

# Simplify and streamline the logic of the script.
```

Notlar

[1] Chet Ramey gelecekteki bir Bash sürümünde, ilişkilendirilebilir dizileri (bir Perl özelliği) vaat ediyor.

KISIM 5 EKLER

Ek A. KATKIDA BULUNAN KOMUT DOSYALARI

Bu komut dosyaları, belgenin metnine uygun olmasa da, bazı ilginç kabuk programlama tekniklerini göstermektedir. Bunlar çok yararlıdır. Analiz etmek ve çalışmaktan keyif alacağınızı düşünüyorum.

ÖRNEK A.1 manview: BİÇİMLENDİRİLMİŞ man SAYFALARININ GÖRÜNTÜLENMESİ

```
#!/bin/bash



# manview.sh: Formats the source of a man page for viewing.
# This is useful when writing man page source and you want to
# look at the intermediate results on the fly while working on it.
E_WRONGARGS=65
if [ -z "$1" ]
then
  echo "Usage: ‘basename $0‘ [filename]"
  exit $E_WRONGARGS
fi
groff -Tascii -man $1 | less
# From the man page for groff.
# If the man page includes tables and/or equations,
# then the above code will barf.
# The following line can handle such cases.
```

```
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char –man
#
#   Thanks, S.C.
exit 0
```

ÖRNEK A.2 mailformat: BİR e-POSTA İLETİSİNİN BİÇİMLENDİRİLMESİ

```
#!/bin/bash



# mail-format.sh: Format e-mail messages.
# Gets rid of carets, tabs, also fold excessively long lines.
# =================================================================
#                   Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=65
E_NOFILE=66
if [ $# -ne $ARGS ]                          # Correct number of arguments
passed to script?
then
  echo "Usage: ‘basename $0‘ filename"
  exit $E_BADARGS
fi
if [ -f "$1" ]                               # Check if file exists.
then
  file_name=$1
else
  echo "File \"$1\" does not exist."
  exit $E_NOFILE
fi
# =================================================================
```

```
MAXWIDTH=70                                    # Width to fold long lines to.

#  Delete carets and tabs at beginning of lines,

#+ then fold lines to $MAXWIDTH characters.

sed '

s/^>//

s/^ *>//

s/^ *//

s/               *//

' $1 | fold -s --width=$MAXWIDTH

          # -s option to "fold" breaks lines at whitespace, if possible.

#  This script was inspired by an article in a well-known trade journal

#+ extolling a 164K Windows utility with similar functionality.

#

#  An nice set of text processing utilities and an efficient

#+ scripting language makes unnecessary bloated executables.

exit 0
```

ÖRNEK A.3 rn: DOSYA YENİDEN ADLANDIRMA PROGRAMI

Bu komut dosyası Örnek 12.15'in değiştirilmiş bir şeklidir.

```
#! /bin/bash

#

# Very simpleminded filename "rename" utility (based on "lowercase.sh").

#

#  The "ren" utility, by Vladimir Lanin (lanin@csd2.nyu.edu),

#+ does a much better job of this.

ARGS=2
```

```
E_BADARGS=65

ONE=1                                     # For getting singular/plural
right (see below).

if [ $# -ne "$ARGS" ]

then

  echo "Usage: ‘basename $0‘ old-pattern new-pattern"

  # As in "rn gif jpg", which renames all gif files in working directory
  #+ to jpg.

  exit $E_BADARGS

fi

number=0                                  # Keeps track of how many files
actually renamed.

for filename in *$1*                      # Traverse all matching files in
directory.

do

  if [ -f "$filename" ]                   # If finds match...

  then

    fname=‘basename $filename‘             # Strip off path.

    n=‘echo $fname | sed -e "s/$1/$2/"‘    # Substitute new for old in
filename.

    mv $fname $n                          # Rename.

    let "number += 1"

  fi

done



if [ "$number" -eq "$ONE" ]               # For correct grammar.

then

  echo "$number file renamed."

else

  echo "$number files renamed."

fi

exit 0
```

```
# Exercises:
# ---------
# What type of files will this not work on?
# How can this be fixed?
#
#  Rewrite this script to process all the files in a directory
#+ containing spaces in their names, and to rename them,
#+ substituting an underscore for each space.
```

ÖRNEK A.4 BOŞLUKLARI İÇEREN DOSYALARIN YENİDEN ADLANDIRILMASI

Bu, bir önceki komut dosyasından daha basittir.

```
#! /bin/bash
# blank-rename.sh
#
# Substitutes underscores for blanks in all the filenames in a directory.
ONE=1                 # For getting singular/plural right (see below).
number=0              # Keeps track of how many files actually renamed.
FOUND=0               # Successful return value.
for filename in *  # Traverse all files in directory.
do
      echo "$filename" | grep -q " "       #  Check whether filename
      if [ $? -eq $FOUND ]                 #+ contains space(s).
      then
```

```
        fname=$filename                     # Strip off path.
        n=‘echo $fname | sed -e "s/ /_/g"‘ # Substitute underscore for
                                            #+ blank.
        mv "$fname" "$n"                    # Do the actual renaming.
        let "number += 1"
      fi
done
if [ "$number" -eq "$ONE" ]                 # For correct grammar.
then
  echo "$number file renamed."
else
  echo "$number files renamed."
fi
exit 0
```

ÖRNEK A.5 encryptedpw: YEREL OLARAK ŞİFRELİ PAROLA KULLANARAK BİR ftp SİTESİNE YÜKLEME YAPMAK

```
#!/bin/bash
# Example "ex72.sh" modified to use encrypted password.
# Note that this is still somewhat insecure,
#+ since the decrypted password is sent in the clear.
# Use something like "ssh" if this is a concern.
E_BADARGS=65
if [ -z "$1" ]
```

```
then
  echo "Usage: ‘basename $0‘ filename"
  exit $E_BADARGS
fi
Username=bozo                                   # Change to suit.
pword=/home/bozo/secret/password_encrypted.file
# File containing encrypted password.
Filename=‘basename $1‘                          # Strips pathname out of file
nameServer="XXX"
Directory="YYY"                                 # Change above to actual server
                                                #+ name & directory.
Password=‘cruft <$pword‘                        # Decrypt password.
# Uses the author’s own "cruft" file encryption package,
#+ based on the classic "onetime pad" algorithm,
#+ and obtainable from:
#+ Primary-site: ftp://metalab.unc.edu /pub/Linux/utils/file
#+                cruft-0.2.tar.gz [16k]
ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
```

```
bye

End-Of-Session

# -n option to "ftp" disables auto-logon.

# "bell" rings 'bell' after each file transfer.

exit 0
```

ÖRNEK A.6 copy-cd: BİR VERİ CD'si KOPYALAMA

```
#!/bin/bash

# copy-cd.sh: copying a data CD

CDROM=/dev/cdrom                          # CD ROM device

OF=/home/bozo/projects/cdimage.iso        # output file

#       /xxxx/xxxxxxx/                        Change to suit your system.

BLOCKSIZE=2048

SPEED=2                                   # May use higher speed if

                                          #+ supported.

echo; echo "Insert source CD, but do *not* mount it."

echo "Press ENTER when ready. "

read ready                                # Wait for input, $ready not used.

echo; echo "Copying the source CD to $OF."

echo "This may take a while. Please be patient."

dd if=$CDROM of=$OF bs=$BLOCKSIZE         # Raw device copy.

echo; echo "Remove data CD."
```

```
echo "Insert blank CDR."

echo "Press ENTER when ready. "

read ready                                # Wait for input, $ready not used.

echo "Copying $OF to CDR."

cdrecord -v -isosize speed=$SPEED dev=0,0 $OF

# Uses Joerg Schilling's "cdrecord" package (see its docs).

# http://cdrecord.berlios.de/private/cdrecord.html

echo; echo "Done copying $OF to CDR on device $CDROM."

echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.

read answer

case "$answer" in

[yY])  rm -f $OF

       echo "$OF erased."

       ;;

*)    echo "$OF not erased.";;

esac

echo

# Exercise:

# Change the above "case" statement to also accept "yes" and "Yes" as
input.

exit 0
```

ÖRNEK A.7 Collatz SERİSİ

```
#!/bin/bash

#  collatz.sh

#  The notorious "hailstone" or Collatz series.

#  ----------------------------------------
```

```
#  1) Get the integer "seed" from the command line.
#  2) NUMBER <--- seed
#  3) Print NUMBER.
#  4) If NUMBER is even, divide by 2, or
#  5)+ if odd, multiply by 3 and add 1.
#  6) NUMBER <--- result
#  7) Loop back to step 3 (for specified number of iterations).
#
#  The theory is that every sequence,
#+ no matter how large the initial value,
#+ eventually settles down to repeating "4,2,1..." cycles,
#+ even after fluctuating through a wide range of values.
#
#  This is an instance of an "iterate",
#+ an operation that feeds its output back into the input.
#  Sometimes the result is a "chaotic" series.
ARGS=1
E_BADARGS=65
if [ $# -ne $ARGS ]                          # Need a seed number.
then
  echo "Usage: ‘basename $0‘ NUMBER"
  exit $E_BADARGS
fi
```

```
MAX_ITERATIONS=200

# For large seed numbers (>32000), increase MAX_ITERATIONS.

h=$1                                          # Seed

echo

echo "C($1) --- $MAX_ITERATIONS Iterations"

echo

for ((i=1; i<=MAX_ITERATIONS; i++))

do

echo -n "$h     "

#          ^^^^^

#           tab

  let "remainder = h % 2"

  if [ "$remainder" -eq 0 ]                   # Even?

  then

    let "h /= 2"                              # Divide by 2.

  else

    let "h = h*3 + 1"                         # Multiply by 3 and add 1.

  fi

COLUMNS=10                                    # Output 10 values per line.

let "line_break = i % $COLUMNS"

if [ "$line_break" -eq 0 ]

then

  echo
```

```
fi

done

echo

#  For more information on this mathematical function,

#+ see "Computers, Pattern, Chaos, and Beauty", by Pickover, p. 185 ff.,

#+ as listed in the bibliography.

exit 0
```

ÖRNEK A.8 İKİ TARİH ARASINDAKİ GÜN SAYISININ HESAPLANMASI

```
#!/bin/bash

# days-between.sh: Number of days between two dates.

# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY

ARGS=2                      # Two command line parameters expected.

E_PARAM_ERR=65              # Param error.

REFYR=1600                  # Reference year.

CENTURY=100

DIY=365

ADJ_DIY=367                 # Adjusted for leap year + fraction.

MIY=12

DIM=31

LEAPCYCLE=4
```

```
MAXRETVAL=256              # Largest permissable positive
                           #+ return value from a function.
diff=                      # Declare global variable for date difference.
value=                     # Declare global variable for absolute value.
day=                       # Declare globals for day, month, year.
month=
year=


Param_Error ()             # Command line parameters wrong.
{
  echo "Usage: 'basename $0' [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo " (date must be after 1/3/1600)"
  exit $E_PARAM_ERR
}


Parse_Date ()              # Parse date from command line params.
{
  month=${1%%/**}
  dm=${1%/**}              # Day and month.
  day=${dm#*/}
  let "year = 'basename $1'" # Not a filename, but works just the same.
}
```

```
check_date ()             # Checks for invalid date(s) passed.

{

  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -lt
"$REFYR" ] && Param_Error

  # Exit script on bad value(s).

  # Uses "or-list / and-list".

  #

  # Exercise: Implement more rigorous date checking.

}



strip_leading_zero ()     # Better to strip possible leading zero(s)

{                         # from day and/or month

  val=${1#0}              # since otherwise Bash will interpret them

  return $val             # as octal values (POSIX.2, sect 2.9.2.1).

}



day_index ()              # Gauss' Formula:

{                         # Days from Jan. 3, 1600 to date passed as param.

  day=$1

  month=$2

  year=$3

  let "month = $month - 2"

  if [ "$month" -le 0 ]

  then
```

```
    let "month += 12"

    let "year -= 1"

  fi

  let "year -= $REFYR"

  let "indexyr = $year / $CENTURY"

  let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr +
$indexyr/$LEAPCYCLE + $ADJ_DIY*$month/$MIY + $day - $DIM"

  if [ "$Days" -gt "$MAXRETVAL" ]   # If greater than 256,

  then                              # then change to negative value

    let "dindex = 0 - $Days"        # which can be returned from function.

  else let "dindex = $Days"

  fi

  return $dindex

}

calculate_difference ()             # Difference between to day indices.

{

  let "diff = $1 - $2"              # Global variable.

}



abs ()                              # Absolute value

{                                   # Uses global "value" variable.

  if [ "$1" -lt 0 ]                 # If negative

  then                              # then

    let "value = 0 - $1"            # change sign,
```

```
  else                                  # else
    let "value = $1"                    # leave it alone.
  fi
}
if [ $# -ne "$ARGS" ]                   # Require two command line params.
then
  Param_Error
fi


Parse_Date $1
check_date $day $month $year            # See if valid date.
strip_leading_zero $day                 # Remove any leading zeroes
day=$?                                  # on day and/or month.
strip_leading_zero $month
month=$?
day_index $day $month $year
date1=$?
abs $date1                              # Make sure it’s positive
date1=$value                            # by getting absolute value.


Parse_Date $2
check_date $day $month $year
strip_leading_zero $day
```

```
day=$?

strip_leading_zero $month

month=$?

day_index $day $month $year

date2=$?

abs $date2                              # Make sure it’s positive.

date2=$value

calculate_difference $date1 $date2

abs $diff                               # Make sure it’s positive.

diff=$value

echo $diff

exit 0
```

ÖRNEK A.9 BİR "SÖZLÜK" YAPMAK

```
#!/bin/bash

# makedict.sh [make dictionary]

# Modification of /usr/sbin/mkdict script.

# Original script copyright 1993, by Alec Muffett.

#

#  This modified script included in this document in a manner

#+ consistent with the "LICENSE" document of the "Crack" package
```

```
#+ that the original script is a part of.

#  This script processes text files to produce a sorted list

#+ of words found in the files.

#  This may be useful for compiling dictionaries

#+ and for lexicographic research.

E_BADARGS=65

if [ ! -r "$1" ]                  #  Need at least one

then                              #+ valid file argument.

  echo "Usage: $0 files-to-process"

  exit $E_BADARGS

fi

# SORT="sort"                     # No longer necessary to define options

                                   #+ to sort. Changed from original script.

cat $* |                          # Contents of specified files to stdout.

        tr A-Z a-z |              # Convert to uppercase.

        tr ’ ’ ’12’ |             # New: change spaces to newlines.

#       tr -cd ’12[a-z][0-9]’ |   # Get rid of everything non-alphanumeric

                                  #+ (original script).

        tr -c ’12a-z’ ’12’ |      # Rather than deleting

                                  #+ now change non-alpha to newlines.

        sort |                    # $SORT options unnecessary now.

        uniq |                    # Remove duplicates.
```

```
        grep -v '^#' |           # Delete lines beginning with a hashmark.
        grep -v '^$'             # Delete blank lines.
exit 0
```

ÖRNEK A.10 "HAYAT OYUNU"

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"
# ########################################################################## #
# This is the Bash script version of John Conway's "Game of Life".      #
# "Life" is a simple implementation of cellular automata.                #
# ---------------------------------------------------------------------- #
# On a rectangular grid, let each "cell" be either "living" or "dead".  #
# Designate a living cell with a dot, and a dead one with a blank space.#
#  Begin with an arbitrarily drawn dot-and-blank grid,                   #
#+ and let this be the starting generation, "generation 0".             #
# Determine each successive generation by the following rules:           #
# 1) Each cell has 8 neighbors, the adjoining cells                      #
#+   left, right, top, bottom, and the 4 diagonals.                     #
#                      123                                               #
#                      4*5                                               #
#                      678                                               #
```

```
#                                                                           #
# 2) A living cell with either 2 or 3 living neighbors remains alive.   #
# 3) A dead cell with 3 living neighbors becomes alive (a "birth").     #
SURVIVE=2                                                                   #
BIRTH=3                                                                     #
# 4) All other cases result in dead cells.                                  #
# ########################################################################## #
startfile=gen0          # Read the starting generation from the file "gen0".
# Default, if no other file specified when invoking script.
#
if [ -n "$1" ]                    # Specify another "generation 0" file.
then
  if [ -e "$1" ]                  # Check for existence.
  then
    startfile="$1"
  fi
fi
ALIVE1=.
DEAD1=_
                  # Represent living and "dead" cells in the start-up file.
#  This script uses a 10 x 10 grid (may be increased,
#+ but a large grid will will cause very slow execution).
ROWS=10
```

```
COLS=10

GENERATIONS=10                        #  How many generations to cycle through.

#  Adjust this upwards,

#+ if you have time on your hands.

NONE_ALIVE=80                         # Exit status on premature bailout,

#+ if no cells left alive.

TRUE=0

FALSE=1

ALIVE=0

DEAD=1

avar=                                 # Global; holds current generation.

generation=0                          # Initialize generation count.

# ==================================================================

let "cells = $ROWS * $COLS"

# How many cells.

declare -a initial                    # Arrays containing "cells".

declare -a current

display ()

{

alive=0                               # How many cells "alive".

# Initially zero.

declare -a arr

arr=( ‘echo "$1"‘ )                   # Convert passed arg to array.
```

```
element_count=${#arr[*]}

local i

local rowcheck

for ((i=0; i<$element_count; i++))

do

  # Insert newline at end of each row.

  let "rowcheck = $i % ROWS"

  if [ "$rowcheck" -eq 0 ]

  then

    echo                            # Newline.

    echo -n " "                     # Indent.

  fi

  cell=${arr[i]}

  if [ "$cell" = . ]

  then

    let "alive += 1"

  fi

  echo -n "$cell" | sed -e ’s/_/ /g’

  # Print out array and change underscores to spaces.

done

return

}
```

```
IsValid ()                            # Test whether cell coordinate valid.

{

  if [ -z "$1" -o -z "$2" ]           # Mandatory arguments missing?

  then

    return $FALSE

  fi

local row

local lower_limit=0                   # Disallow negative coordinate.

local upper_limit

local left

local right

let "upper_limit = $ROWS * $COLS - 1"  # Total number of cells.

if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]

then

  return $FALSE                       # Out of array bounds.

fi

row=$2

let "left = $row * $ROWS"             # Left limit.

let "right = $left + $COLS - 1"   # Right limit.

if [ "$1" -lt "$left" -o "$1" -gt "$right" ]

then

  return $FALSE                       # Beyond row boundary.

fi
```

```
return $TRUE                          # Valid coordinate.

}



IsAlive ()                            # Test whether cell is alive.

# Takes array, cell number, state of cell as arguments.

{

GetCount "$1" $2                      # Get alive cell count in neighborhood.

local nhbd=$?

if [ "$nhbd" -eq "$BIRTH" ]           # Alive in any case.

then

  return $ALIVE

fi

if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]

then                                  # Alive only if previously alive.

  return $ALIVE

fi

return $DEAD                          # Default.

}
```

```
GetCount ()                          # Count live cells in passed cell’s
neighborhood.

                                     # Two arguments needed:

                                     # $1) variable holding array

                                     # $2) cell number

{

  local cell_number=$2

  local array

  local top

  local center

  local bottom

  local r

  local row

  local i

  local t_top

  local t_cen

  local t_bot

  local count=0

  local ROW_NHBD=3

  array=( ‘echo "$1"‘ )

  let "top = $cell_number - $COLS - 1"      # Set up cell neighborhood.

  let "center = $cell_number - 1"

  let "bottom = $cell_number + $COLS - 1"

  let "r = $cell_number / $ROWS"
```

```
  for ((i=0; i<$ROW_NHBD; i++))   # Traverse from left to right.
  do
    let "t_top = $top + $i"
    let "t_cen = $center + $i"
    let "t_bot = $bottom + $i"
    let "row = $r"                # Count center row of neighborhood.
    IsValid $t_cen $row           # Valid cell position?
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_cen]} = "$ALIVE1" ]   # Is it alive?
      then                        # Yes?
        let "count += 1"          # Increment count.
      fi
    fi
    let "row = $r - 1"            # Count top row.
    IsValid $t_top $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_top]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi
```

```
    let "row = $r + 1"             # Count bottom row.
    IsValid $t_bot $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_bot]} = "$ALIVE1" ]
      then
       let "count += 1"
      fi
    fi
  done
  if [ ${array[$cell_number]} = "$ALIVE1" ]
  then
    let "count -= 1"               # Make sure value of tested cell itself
  fi                               #+ is not counted.
  return $count
}


next_gen ()                        # Update generation array.
{
local array
local i=0
array=( `echo "$1"` )              # Convert passed arg to array.
while [ "$i" -lt "$cells" ]
```

```
do

  IsAlive "$1" $i ${array[$i]} # Is cell alive?

  if [ $? -eq "$ALIVE" ]

  then                         # If alive, then

    array[$i]=.                #+ represent the cell as a period.

  else

    array[$i]="_"              # Otherwise underscore

  fi                           #+ (which will later be converted to space).

  let "i += 1"

done

# let "generation += 1"        # Increment generation count.

# Set variable to pass as parameter to "display" function.

avar=‘echo ${array[@]}‘        # Convert array back to string variable.

display "$avar"                # Display it.

echo; echo

echo "Generation $generation -- $alive alive"

if [ "$alive" -eq 0 ]

then

  echo

  echo "Premature exit: no more cells alive!"

  exit $NONE_ALIVE             # No point in continuing

fi                             #+ if no live cells.

}
```

```
# ============================================================

# main ()

# Load initial array with contents of startup file.

initial=( ‘cat "$startfile" | sed -e ’/#/d’ | tr -d ’\n’ |\

sed -e ’s/\./\. /g’ -e ’s/_/_ /g’‘ )

# Delete lines containing ’#’ comment character.

# Remove linefeeds and insert space between elements.

clear                           # Clear screen.

echo  #         Title

echo "======================="

echo "    $GENERATIONS generations"

echo "            of"

echo "\"Life in the Slow Lane\""

echo "======================="

# -------- Display first generation. --------

Gen0=‘echo ${initial[@]}‘

display "$Gen0"                 # Display only.

echo; echo

echo "Generation $generation -- $alive alive"

# ---------------------------------------

let "generation += 1"           # Increment generation count.

echo

# ------- Display second generation. -------
```

```
Cur=‘echo ${initial[@]}‘

next_gen "$Cur"                # Update & display.

# ---------------------------------------

let "generation += 1"          # Increment generation count.

# ------ Main loop for displaying subsequent generations ------

while [ "$generation" -le "$GENERATIONS" ]

do

  Cur="$avar"

  next_gen "$Cur"

  let "generation += 1"

done

# ================================================================

echo

exit 0

# ----------------------------------------------------------------

# The grid in this script has a "boundary problem".

# The the top, bottom, and sides border on a void of dead cells.

# Exercise: Change the script to have the grid wrap around,

# +         so that the left and right sides will "touch",

# +         as will the top and bottom.
```

ÖRNEK A.11 "HAYAT OYUNU" İÇİN VERİ DOSYASI

```
# This is an example "generation 0" start-up file for "life.sh".

# ---------------------------------------------------------------

#  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,

#+ and an underscore (_) for dead ones. We cannot simply use spaces

#+ for dead cells in this file because of a peculiarity in Bash arrays.

# [Exercise for the reader: explain this.]

#

# Lines beginning with a ’#’ are comments, and the script ignores them

__.__..___

___._._____

____.___..

_._______.

____._____

..__...___

____._____

___...____

__..._..___

_..___..__
```

Yazar Mendel Cooper'ın izni alınarak ticari olmayan amaçla Türkçe'ye çevrilmiştir.

+++

Aşağıdaki iki komut dosyası Toronto Üniversitesi'nden Mark Moraes tarafından yazılmıştır. İzin ve kısıtlamalar için ekteki dosyaya "Moraes-TELİF HAKKI"na bakın.

ÖRNEK A.12 behead: POSTA ve HABER MESAJ BAŞLIKLARINI ÇIKARMA

```
#! /bin/sh

# Strips off the header from a mail/News message i.e. till the first

# empty line

# Mark Moraes, University of Toronto

# ==> These comments added by author of this document.



if [ $# -eq 0 ]; then

# ==> If no command line args present, then works on file redirected to

#+ stdin.

        sed -e '1,/^$/d' -e '/^[        ]*$/d'

        # --> Delete empty lines and all lines until

        # --> first one beginning with white space.

else



# ==> If command line args present, then work on files named.

        for i do

                sed -e '1,/^$/d' -e '/^[       ]*$/d' $i

                # --> Ditto, as above.
```

```
        done

fi

# ==> Exercise: Add error checking and other options.

# ==>

# ==> Note that the small sed script repeats, except for the arg passed.

# ==> Does it make sense to embed it in a function? Why or why not?
```

ÖRNEK A.13 ftpget: ftp İLE DOSYA İNDİRME

```
#! /bin/sh

# $Id: ftpget,v 1.2 91/05/07 21:15:43 moraes Exp $

# Script to perform batch anonymous ftp. Essentially converts a list of

# of command line arguments into input to ftp.

# Simple, and quick - written as a companion to ftplist

# -h specifies the remote host (default prep.ai.mit.edu)

# -d specifies the remote directory to cd to - you can provide a sequence

# of -d options - they will be cd'ed to in turn. If the paths are relative,

# make sure you get the sequence right. Be careful with relative paths –

# there are far too many symlinks nowadays.

# (default is the ftp login directory)

# -v turns on the verbose option of ftp, and shows all responses from the

# ftp server.

# -f remotefile[:localfile] gets the remote file into localfile
```

```
# -m pattern does an mget with the specified pattern. Remember to quote

# shell characters.

# -c does a local cd to the specified directory

# For example,

#       ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \

#                -d ../pub/R3/fixes -c ~/fixes -m 'fix*'

# will get xplaces.shar from ~ftp/contrib on expo.lcs.mit.edu, and put it

# in xplaces.sh in the current working directory, and get all fixes from

# ~ftp/pub/R3/fixes and put them in the ~/fixes directory.

# Obviously, the sequence of the options is important, since the equivalent

# commands are executed by ftp in corresponding order

#

# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989

# ==> Angle brackets changed to parens, so Docbook won't get indigestion.

#

# ==> These comments added by author of this document.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin

# export PATH

# ==> Above 2 lines from original script probably superfluous.

TMPFILE=/tmp/ftp.$$

# ==> Creates temp file, using process id of script ($$)

# ==> to construct filename.

SITE=‘domainname‘.toronto.edu
```

```
# ==> ’domainname’ similar to ’hostname’

# ==> May rewrite this to parameterize this for general use.

usage="Usage: $0 [-h remotehost] [-d remotedirectory]... [-f
remfile:localfile]... \ [-c localdirectory] [-m filepattern] [-v]"

ftpflags="-i -n"

verbflag=

set -f                                          # So we can use globbing in –m

set x ‘getopt vh:d:c:m:f: $*’

if [ $? != 0 ]; then

        echo $usage

        exit 65

fi

shift

trap ’rm -f ${TMPFILE} ; exit’ 0 1 2 3 15

echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"

# ==> Added quotes (recommended in complex echoes).

echo binary >> ${TMPFILE}

for i in $*                                     # ==> Parse command line args.

do

        case $i in

        -v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;

        -h) remhost=$2; shift 2;;

        -d) echo cd $2 >> ${TMPFILE};

            if [ x${verbflag} != x ]; then
```

```
              echo pwd >> ${TMPFILE};

            fi;

            shift 2;;

        -c) echo lcd $2 >> ${TMPFILE}; shift 2;;

        -m) echo mget "$2" >> ${TMPFILE}; shift 2;;

        -f) f1=‘expr "$2" : "\([^:]*\).*"‘; f2=‘expr "$2" :"[^:]*:\(.*\)"‘;

            echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;

        --) shift; break;;

        esac

done

if [ $# -ne 0 ]; then

        echo $usage

        exit 65       # ==> Changed from "exit 2" to conform with standard.

fi

if [ x${verbflag} != x ]; then

        ftpflags="${ftpflags} -v"

fi

if [ x${remhost} = x ]; then

        remhost=prep.ai.mit.edu

        # ==> Rewrite to match your favorite ftp site.

fi

echo quit >> ${TMPFILE}

# ==> All commands saved in tempfile.
```

```
ftp ${ftpflags} ${remhost} < ${TMPFILE}

# ==> Now, tempfile batch processed by ftp.

rm -f ${TMPFILE}

# ==> Finally, tempfile deleted (you may wish to copy it to a logfile).

# ==> Exercises:

# ==> ---------

# ==> 1) Add error checking.

# ==> 2) Add bells & whistles.
```

+

Antek Sawicki, aşağıdaki komut dosyasına katkıda bulunmuştur. Bu komut dosyası, Bölüm 9.3'te ele alınan parametre değiştirme operatörlerin çok akıllıca kullanımını içermektedir.

ÖRNEK A.14 password: RASGELE 8 KARAKTERLİ ŞİFRELER OLUŞTURMA

```
#!/bin/bash

# May need to be invoked with #!/bin/bash2 on older machines.

#

# Random password generator for bash 2.x by Antek Sawicki <tenox@tenox.tc>,

# who generously gave permission to the document author to use it here.

#

# ==> Comments added by document author ==>

MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

LENGTH="8"
```

```
# ==> May change 'LENGTH' for longer password, of course.

while [ "${n:=1}" -le "$LENGTH" ]

# ==> Recall that := is "default substitution" operator.

# ==> So, if 'n' has not been initialized, set it to 1.

do

        PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"

        # ==> Very clever, but tricky.

        # ==> Starting from the innermost nesting...

        # ==> ${#MATRIX} returns length of array MATRIX.

        # ==> $RANDOM%${#MATRIX} returns random number between 1

        # ==> and length of MATRIX - 1.

        # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}

        # ==> returns expansion of MATRIX at random position, by length 1.

        # ==> See {var:pos:len} parameter substitution in Section 3.3.1

        # ==> and following examples.

        # ==> PASS=... simply pastes this result onto previous PASS

        #+ (concatenation).

        # ==> To visualize this more clearly, uncomment the following line

        # ==> echo "$PASS"

        # ==> to see PASS being built up,

        # ==> one character at a time, each iteration of the loop.

        let n+=1

        # ==> Increment 'n' for next pass.
```

```
done

echo "$PASS"                            # ==> Or, redirect to file, as desired.

exit 0
```

+

James R. Van Zandt adlandırılmış olukları kullanan bu komut dosyasına katkıda bulunmuştur ve, kendi sözleriyle, "gerçekten bir alıntı ve değiştirme alıştırmasıdır".

ÖRNEK A.15 fifo: ADLANDIRILMIŞ OLUK KULLANARAK GÜNLÜK YEDEKLEME YAPMA

```
#!/bin/bash

# ==> Script by James R. Van Zandt, and used here with his permission.

# ==> Comments added by author of this document.

  HERE=‘uname -n‘               # ==> hostname

  THERE=bilbo

  echo "starting remote backup to $THERE at ‘date +%r‘"

  # ==> ‘date +%r‘ returns time in 12-hour format, i.e. "08:08:34 PM".

  # make sure /pipe really is a pipe and not a plain file

  rm -rf /pipe

  mkfifo /pipe                  # ==> Create a "named pipe", named "/pipe".

  # ==> ’su xyz’ runs commands as user "xyz".

  # ==> ’ssh’ invokes secure shell (remote login client).
```

```
  su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-daily.tar.gz\" <
/pipe"&

  cd /

  tar -czf - bin boot dev etc home info lib man root sbin share usr var
>/pipe

  # ==> Uses named pipe, /pipe, to communicate between processes:

  # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.

  # ==> The end result is this backs up the main directories, from / on
down.

  # ==> What are the advantages of a "named pipe" in this situation,

  # ==> as opposed to an "anonymous pipe", with |?

  # ==> Will an anonymous pipe even work here?

  exit 0
```

+

Stephane Chazelas asal sayılar üretmek için dizilimlerin gerekmediğini göstermek için aşağıdaki komut dosyasına katkıda bulunmuştur.

ÖRNEK A.16 mod OPERATÖRÜ KULLANILARAK ASAL SAYILAR ÜRETMEK

```
#!/bin/bash

# primes.sh: Generate prime numbers, without using arrays.

# Script contributed by Stephane Chazelas.

#  This does *not* use the classic "Sieve of Eratosthenes" algorithm,

#+ but instead uses the more intuitive method of testing each candidate
number
```

```
#+ for factors (divisors), using the "%" modulo operator.

LIMIT=1000                    # Primes 2 – 1000

Primes()
{
  (( n = $1 + 1 ))            # Bump to next integer.
  shift                       # Next parameter in list.
# echo "_n=$n i=$i_"
  if (( n == LIMIT ))
  then echo $*
    return
  fi
  for i; do                   # "i" gets set to "@", previous values of $n.

# echo "-n=$n i=$i-"
    (( i * i > n )) && break  # Optimization.
    (( n % i )) && continue   # Sift out non-primes using modulo operator.
    Primes $n $@              # Recursion inside loop.
    return
    done
    Primes $n $@ $n         # Recursion outside loop.
                            # Successively accumulate positional parameters.
                            # "$@" is the accumulating list of primes.
}
```

```
Primes 1

exit 0

# Uncomment lines 17 and 25 to help figure out what is going on.

# Compare the speed of this algorithm for generating primes

# with the Sieve of Eratosthenes (ex68.sh).

# Exercise: Rewrite this script without recursion, for faster execution.
```

+

Jordi Sanfeliu kendi yazdığı zarif tree komut dosyasını kullanmamıza izin verdi.

ÖRNEK A.17 tree: BİR DİZİN AĞACININ GÖSTERİLMESİ

```
#!/bin/sh# @(#) tree 1.1 30/11/95 by Jordi Sanfeliu

# email: mikaku@arrakis.es## Initial version: 1.0 30/11/95

# Next version : 1.1 24/02/97 Now, with symbolic links

# Patch by : Ian Kjos, to support unsearchable dirs

# email: beth13@mail.utexas.edu

#

# Tree is a tool for view the directory tree (obvious :-) )

#

# ==> 'Tree' script used here with the permission of its author, Jordi
Sanfeliu.

# ==> Comments added by the author of this document.

# ==> Argument quoting added.

```

```
search ()

{

    for dir in ‘echo *‘

    # ==> ‘echo *‘ lists all the files in current working directory,
without line breaks.

    # ==> Similar effect to for dir in *

    # ==> but "dir in ‘echo *‘" will not handle filenames with blanks.

    do

        if [ -d "$dir" ] ; then          # ==> If it is a directory (-d)...

            zz=0     # ==> Temp variable, keeping track of directory level.

            while [ $zz != $deep ]       # Keep track of inner nested loop.

            do

                echo -n "| "             # ==> Display vertical connector
symbol,

                     # ==> with 2 spaces & no line feed in order to indent.

                zz=‘expr $zz + 1‘        # ==> Increment zz.

            done

            if [ -L "$dir" ] ; then      # ==> If directory is a symbolic
                                         #+ link...

                echo "+---$dir" ‘ls -l $dir | sed ’s/^.*’$dir’ //’‘

             # ==> Display horiz. connector and list directory name, but...

             # ==> delete date/time part of long listing.

            else

                echo "+---$dir"           # ==> Display horizontal
                                          #+ connector symbol...

                                          # ==> and print directory name.

                if cd "$dir" ; then       # ==> If can move to
                                          #+ subdirectory...

                    deep=‘expr $deep + 1‘ # ==> Increment depth.

                    search                # with recursivity ;-)

                                          # ==> Function calls itself.

                    numdirs=‘expr $numdirs + 1‘ # ==> Increment directory
count.
```

```
                fi

            fi

        fi

    done

    cd ..                           # ==> Up one directory level.

    if [ "$deep" ] ; then           # ==> If depth = 0 (returns TRUE)...

        swfi=1                      # ==> set flag showing that search is done.

    fi

    deep=‘expr $deep - 1‘           # ==> Decrement depth.

}

# - Main –

if [ $# = 0 ] ; then

    cd ‘pwd‘   # ==> No args to script, then use current working directory.

else

    cd $1      # ==> Otherwise, move to indicated directory.

fi

echo "Initial directory = ‘pwd‘"

swfi=0         # ==> Search finished flag.

deep=0         # ==> Depth of listing.

numdirs=0

zz=0

while [ "$swfi" != 1 ] # While flag not set...

do

    search     # ==> Call function after initializing variables.

done

echo "Total directories = $numdirs"

exit 0

# ==> Challenge: try to figure out exactly how this script works.
```

Noah Friedman, kendi yazdığı string işlevi komut dosyasını kullanmamız için izin verdi. Bu komut dosyası, aslında C-kütüphanesi dize işleme fonksiyonlarından bazılarını taklit ediyor.

ÖRNEK A.18 string FONKSİYONLARI: C-BENZERİ DİZE FONKSİYONLARI

```
#!/bin/bash

# string.bash --- bash emulation of string(3) library routines

# Author: Noah Friedman friedman@prep.ai.mit.edu

# ==> Used with his kind permission in this document.

# Created: 1992-07-01# Last modified: 1993-09-29

# Public domain

# Conversion to bash v2 syntax done by Chet Ramey

# Commentary:

# Code:

#:docstring strcat:

# Usage: strcat s1 s2

#

# Strcat appends the value of variable s2 to variable s1.

#

# Example:

#    a="foo"

#    b="bar"

#    strcat a b

#    echo $a

#    => foobar

#

#:end docstring:

###;;;autoload           ==> Autoloading of function commented out.

function strcat ()

{

    local s1_val s2_val
```

```
    s1_val=${!1}                                # indirect variable expansion
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Line strcat, but strncat appends a maximum of n characters from the value
# of variable s2. It copies fewer if the value of variable s2 is shorter
# than n characters. Echoes result on stdout.
#
# Example:
#   a=foo
#   b=barbaz
#   strncat a b 3
#   echo $a
#   => foobar
#
#:end docstring:
###;;;autoload

function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val
```

```
    s1_val=${!s1}                        # ==> indirect variable expansion
    s2_val=${!s2}
    if [ ${#s2_val} -gt ${n} ]; then
        s2_val=${s2_val:0:$n}            # ==> substring extraction
    fi
    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}
#:docstring strcmp:
# Usage: strcmp $s1 $s2
#
# Strcmp compares its arguments and returns an integer less than, equal to,
# or greater than zero, depending on whether string s1 is lexicographically
# less than, equal to, or greater than string s2.
#:end docstring:
###;;;autoload

function strcmp ()
{
    [ "$1" = "$2" ] && return 0
    [ "${1}" '<' "${2}" ] > /dev/null && return -1
    return 1
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Like strcmp, but makes the comparison by examining a maximum of n
# characters (n less than or equal to zero yields equality).
#:end docstring:
```

```
###;;;autoload

function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
        return 0
    fi
    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
        strcmp "$1" "$2"
        return $?
    else
        s1=${1:0:$3}
        s2=${2:0:$3}
        strcmp $s1 $s2
        return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:
###;;;autoloadfunction

strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}
```

```
#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:
###;;;autoloadfunction

strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
    local result="${1%%[!${2}]*}"
    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:
###;;;autoloadfunction

strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"
    echo ${#result}
}
```

```
#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr echoes a substring starting at the first occurrence of string s2
# in string s1, or nothing if s2 does not occur in the string. If s2 points
# to a string of zero length, strstr echoes s1.
#:end docstring:
###;;;autoload

function strstr ()
{
    # if s2 points to a string of zero length, strstr echoes s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }
    # strstr echoes nothing if s2 does not occur in s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac
    # use the pattern matching code to strip off the match and everything
    # following it
    first=${1/$2*/}
    # then strip off the first unmatched portion of the string
    echo "${1##$first}"
}
#:docstring strtok:

# Usage: strtok s1 s2
#
# Strtok considers the string s1 to consist of a sequence of zero or more
# text tokens separated by spans of one or more characters from the
# separator string s2. The first call (with a non-empty string s1
```

```
# specified) echoes a string consisting of the first token on stdout. The
# function keeps track of its position in the string s1 between separate
# calls, so that subsequent calls made with the first argument an empty
# string will work through the string immediately following that token. In
# this way subsequent calls will work through the string s1 until no tokens
# remain. The separator string s2 may be different from call to call.
# When no token remains in s1, an empty value is echoed on stdout.
#:end docstring:
###;;;autoload

function strtok ()
{
  :
}

#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Used by many functions like strncmp to truncate arguments for comparison.
# Echoes the first n characters of each string s1 s2 ... on stdout.
#:end docstring:
###;;;autoload

function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}
```

```
# provide string

# string.bash ends here

#
================================================================================
#

# ==> Everything below here added by the document author.

# ==> Suggested use of this script is to delete everything below here,

# ==> and "source" this file into your own scripts.

# strcat

string0=one

string1=two

echo

echo "Testing \"strcat\" function:"

echo "Original \"string0\" = $string0"

echo "\"string1\" = $string1"

strcat string0 string1

echo "New \"string0\" = $string0"

echo

# strlen

echo

echo "Testing \"strlen\" function:"

str=123456789

echo "\"str\" = $str"

echo -n "Length of \"str\" = "

strlen str

echo
```

```
# Exercise:

# --------

# Add code to test all the other string functions above.

exit 0
```

Stephane Chazelas bir Bash komut dosyasında nesne yönelimli programlamayı gösteriyor.

ÖRNEK A.19 NESNE YÖNELİMLİ VERİTABANI

```
#!/bin/bash

# obj-oriented.sh: Object-oriented programming in a shell script.

# Script by Stephane Chazelas.

person.new()               # Looks almost like a class declaration in C++.

{

  local obj_name=$1 name=$2 firstname=$3 birthdate=$4

  eval "$obj_name.set_name() {

          eval \"$obj_name.get_name() {

                    echo \$1

                }\"

        }"

  eval "$obj_name.set_firstname() {

          eval \"$obj_name.get_firstname() {

                    echo \$1

                }\"

        }"

  eval "$obj_name.set_birthdate() {

          eval \"$obj_name.get_birthdate() {

            echo \$1

          }\"

          eval \"$obj_name.show_birthdate() {
```

```
            echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")

          }\"

          eval \"$obj_name.get_age() {

            echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))

          }\"

        }"

  $obj_name.set_name $name

  $obj_name.set_firstname $firstname

  $obj_name.set_birthdate $birthdate

}

echo

person.new self Bozeman Bozo 101272413

# Create an instance of "person.new" (actually passing args to the

#+ function).

self.get_firstname                          # Bozo

self.get_name                               # Bozeman

self.get_age                                # 28

self.get_birthdate                          # 101272413

self.show_birthdate                         # Sat Mar 17 20:13:33 MST 1973

echo

# typeset –f

# to see the created functions (careful, it scrolls off the page).

exit 0
```

Ek B. Sed ve Awk ALFABESİ

Bu bölüm **sed** ve **awk** metin işleme programlarına çok kısa bir giriş yapmaktadır. Biz, burada sadece birkaç temel komut ile ilgileneceğiz ama bu, kabuk betikleri içinde basit sed ve awk yapılarını anlamamız için yeterli olacaktır.

sed: Etkileşimli-olmayan bir metin dosyası editörü

awk: C-benzeri sözdizimi ile bir alan-odaklı kalıp işleme dili

Tüm farklılıklarına rağmen, iki yardımcı program, benzer bir çağırma sözdizimini paylaşırlar, her ikisi de kurallı ifadeleri kullanırlar, her ikisi de varsayılan olarak stdin'den girdi okurlar, her ikisi de stdout'a çıktı gönderirler. Bu iki UNIX aracı birlikte çok iyi çalışırlar. Birinin çıktısı diğerine yöneltilebilir ve birleştirilmiş yetenekleri ile bir miktar Perl gücü kabuk betiklerine verilebilir.

Araçlar arasındaki önemli fark şudur ki; kabuk betikleri sed'e kolayca argüman geçebilir ise de, awk için bu daha karmaşıktır (bkz. Örnek 34.3 ve Örnek 9.20).

B.1 Sed

Sed etkileşimli-olmayan bir satır editörüdür. stdin veya bir dosyadan metin girdisi alır, girdinin belirtilen satırlarında her seferinde bir satır olmak üzere belirli işlemleri gerçekleştirir, sonra stdout veya bir dosyaya çıktıyı gönderir. Bir kabuk betiği içinde sed, genellikle bir oluk içindeki birkaç araç bileşeninden biridir.

sed hangi satırlar üzerinde çalışacağını, girdi olarak kendisine iletilen *adres aralığından* belirler. [1] Bu adres aralığı satır numarasına göre, veya bir eşleşme kalıbı ile belirtilir. Örneğin, 3d girdisi sed'e 3. satırı silme sinyali gönderir, ve */windows/d* girdisi içinde "windows" geçen tüm satırları siler.

sed araç setinin içindeki tüm operasyonların içinden biz öncelikle en sık kullanılan üçü üzerinde odaklanacağız. Bu üç operasyon, çıktı baskısı (stdout), silme ve değiştirmedir.

Tablo B.1 TEMEL sed OPERATÖRLERİ

Operatör	Adı	Etkisi
`[adres-aralığı]/p`	print	Belirtilen adres aralığını yazar.
`[adres-aralığı]/d`	delete	Belirtilen adres aralığını siler.
`s/kalıp1/kalıp2/`	substitute	Belirtilen satırda, kalıp1'in ilk örneğini kalıp2 ile değiştirir.
`[adres-aralığı]/s/kalıp1/kalıp2/`	substitute	Belirtilen *adres aralığı* boyunca, kalıp1'i kalıp2 ile değiştirir.
`[adres-aralığı]/y/kalıp1/kalıp2/`	transform	Belirtilen *adres aralığı* boyunca, kalıp1'in her karakterini kalıp2'de karşılık gelen karakter ile değiştirir.(**tr** ile eşdeğerdir)
`g`	global	Eşleşen her girdi satırı içindeki kalıp eşleşmesi üzerinde çalışır.

g (global) operatörü *substitute* komutuna eklenmediği sürece, değiştirme her satırdaki sadece ilk kalıp eşleşmesi örneği üzerinde çalışır.

Komut satırından ve kabuk betiğinden çalıştırılan bir sed işlemi referans ve belirli seçenekleri gerektirebilir.

```
sed -e '/^$/d' $filename

# The -e option causes the next string to be interpreted as an editing
instruction.

# (If passing only a single instruction to "sed", the "-e" is optional.)

# The "strong" quotes ('') protect the RE characters in the instruction

#+ from reinterpretation as special characters by the body of the script.

# (This reserves RE expansion of the instruction for sed.)

#

# Operates on the text contained in file $filename.

```

sed izleyen dizenin komut veya komut seti olduğunu belirtmek için `-e` seçeneğini kullanır. Dizge içinde tek bir komut bulunuyorsa, o zaman bu seçenek atlanabilir.

```
sed -n '/xzy/p' $filename

# The -n option tells sed to print only those lines matching the pattern.

# Otherwise all input lines would print.

# The -e option not necessary here since there is only a single editing
instruction.
```

TABLO B.2 ÖRNEKLER

Gösterim	Etkisi
`8d`	Girdinin 8. satırını siler.
`/^$/d`	Tüm boş satırları siler.
`1,/^$/d`	Girdinin başından, ilk boş satır da dahil olmak üzere siler.
`/Beyza/p`	"Beyza" içeren satırları yazdırır (-n seçeneği ile).
`s/Windows/Linux/`	Her girdi satırında bulunan ilk "Windows" örneğini "Linux" ile değiştirir.
`s/BSOD/kararlılık/g`	Her girdi satırında bulunan her "BSOD" örneğini "kararlılık" ile değiştirir.
`s/ *$//`	Her satırın sonundaki tüm boşlukları siler.
`s/00*/0/g`	Ardışık sıfır dizelerini tek bir sıfır olarak sıkıştırır.
`/GUI/d`	"GUI" içeren tüm satırları siler.
`s/GUI//g`	Her satır için, satırın kalanına dokunmadan, tüm "GUI" örneklerini siler.

Bir dizeyi sıfır uzunluklu bir dize ile değiştirme, girdi satırı içinde bu dizeyi silme ile eşdeğerdir. Bu işlem satırın kalanını aynı bırakır. Aşağıdaki satıra **`s/GUI//`** uygulayarak,

```
The most important parts of any application are its GUI and sound effects
```

şu satır elde edilir:

```
The most important parts of any application are its and sound effects
```

Ters eğik çizgi bir değişiklik karakteri olarak *yeni satırı* gösterir. Bu özel durumda, yer değiştirme ifadesi bir sonraki satırda devam eder.

```
s/^ */\
/g
```

Bu ifade, satır-başındaki boşlukları yeni satır ile yer değiştirir. Net sonuç paragraf girintilerini paragraflar arasında boş bir satır ile yer değiştirmektir.

Bir adres aralığı ardından bir veya daha fazla işlem takibi, uygun yeni satırlar ile açık ve kapalı küme parantezlerini gerektirebilir.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
}
```

Bu ifade, her ardışık boş satır setinin yalnızca ilk satırını siler. Bunun yararı, paragraflar arasındaki boş satır(lar)ı koruyarak, tek aralıklı bir metin dosyası elde etmek olabilir.

Bir metin dosyasını çift-aralıklı hale dönüştürmenin çabuk yolu **`sed G dosya-adı`** ile mümkündür.

Kabuk betikleri içindeki açıklayıcı sed örnekleri için bkz:

1. Örnek 34.1
2. Örnek 34.2
3. Örnek 12.2
4. Örnek A.3
5. Örnek 12.12
6. Örnek 12.20
7. Örnek A.12
8. Örnek A.17
9. Örnek 12.24
10. Örnek 10.9
11. Örnek 12.33
12. Örnek A.2
13. Örnek 12.10
14. Örnek 12.8
15. Örnek A.10
16. Örnek 17-11

Daha kapsamlı sed açıklamaları için, Kaynakça'daki uygun referanslara başvurunuz.

Notlar

[1] Hiçbir adres aralığı belirtilmemişse, varsayılan *tüm* satırlardır.

B.2 Awk

Awk, **C** sözdizimini anımsatan tam özellikli bir metin işleme dilidir. Geniş bir dizi operatör setine ve yeteneklere sahip olmasına rağmen, burada biz bunların sadece birkaçını kapsayacağız - kabuk betikleme için en yararlı olanlarını.

Awk her girdi satırını *alanlara* böler. Varsayılan olarak, bir alan, boşluk ile ayrılmış ardışık karakter dizesidir. Seçenekler yardımıyla sınırlayıcı (boşluk) değiştirilebilir. Awk her biri alanı ayrı ayrı ayrıştırır ve üzerinde faaliyet gösterir. Bu nedenle, yapılandırılmış metin dosyalarını ve özellikle tabloları, satır ve sütun olarak tutarlı parçalar halinde düzenlenmiş verileri işlemek için awk idealdir.

Güçlü alıntı (tek tırnak) ve kıvrık parantez bir kabuk betiği içinde awk kod kesimlerini içine alır.

```
awk '{print $3}' $filename

# Prints field #3 of file $filename to stdout.

awk '{print $1 $5 $6}' $filename

# Prints fields #1, #5, and #6 of file $filename.
```

Az önce, awk **print** komutunu gördük. Burada başa çıkmamız gereken diğer bir awk özelliği de değişkenlerdir. Awk biraz daha esnek olarak, kabuk betiklerine benzer şekilde değişkenleri yönetir.

```
{ toplam += ${sütun_no} }
```

Bu yürümekte olan "toplam" değişkenine *sütun_no* değerini ekler. Son olarak, "toplam"ı yazdırmak için, bir **END** komut bloğu vardır. Komut dosyası tüm girdileri işledikten sonra bu blok çalıştırılır.

```
END { print total }
```

END'e karşılık gelen ve awk girdileri işlemeye başlamadan önce yapılması gereken bir **BEGIN** kod bloğu vardır.

Kabuk betikleri içindeki awk örnekleri için bkz:

1. Örnek 11.9
2. Örnek 16.7
3. Örnek 12.24
4. Örnek 34.3
5. Örnek 9.20
6. Örnek 11.14
7. Örnek 28.1
8. Örnek 28.2
9. Örnek 10.3
10. Örnek 12.41
11. Örnek 9.23
12. Örnek 12.3
13. Örnek 9.11
14. Örnek 34.7
15. Örnek 10.8

EK C. ÖZEL ANLAMLARI OLAN ÇIKIŞ KODLARI

TABLO C.1 "ÖZEL" ÇIKIŞ KODLARI

Çıkış Kodu Numarası	Anlamı	Örnek	Yorum
1	genel hataların hepsi için	let "var1=1/0"	"sıfıra bölme" gibi çeşitli hatalar
2	Bash dokümantasyonuna göre kabuk yerleşiklerinin yanlış kullanımı		nadiren görülür, genellikle çıkış kodu 1 varsayılır
126	çağrılan komut yürütülemiyor		izin sorunu veya komut çalıştırılabilir değil
127	"komut bulunamadı"		$PATH veya bir yazım hatası olası sorunu
128	exit komutu için geçersiz argüman	exit 3.14159	**exit** 0-255 arası sadece tamsayı argüman alır.
128+n	ölümcül hata sinyali "n"	**kill -9 betiğe_ait_$PPID**	**$?** 137 (128+9) döndürür.
130	komut dosyası Kontrol-C tarafından sonlandırıldı		Kontrol-C ölümcül hata sinyali 2'dir (130=128+2)
255*	aralık dışı çıkış kodu	exit -1	**exit** 0-255 arası sadece tamsayı argüman alır.

Tabloya göre, 1-2, 126-165, ve 255 [1] çıkış kodlarının özel anlamları vardır ve bu nedenle kullanıcı tarafından belirtilen çıkış parametreleri olarak kullanımından kaçınılmalıdır. **Çıkış 127** kodu ile bir betiği sonlandırmak sorun giderirken kesinlikle karışıklığa neden olur (Hata bir "komut bulunamadı" mıdır veya bir kullanıcı tanımlı bir hata mıdır?). Ancak, birçok komut dosyası hata üzerinde genel bir kurtarış yapmak için **çıkış 1** kodu kullanır. Çıkış kodu 1 pek çok olası hataları gösterdiği için, bu, herhangi bir ek belirsizlik getirmez, ancak, diğer taraftan, muhtemelen çok bilgilendirici de olmaz.

Çıkış durumu numaralarını sistematize etmek için bir girişim olmuştur (bkz. `/usr/include/sysexits.h`) ama bu C ve C + + programcıları için tasarlanmıştır. Komut dosyaları için de benzer bir standart uygun olabilir. Bu belgenin yazarı C / C + + standardına uyan, 64 – 113 aralığında kısıtlı kullanıcı tanımlı çıkış kodlarını önermektedir (ek olarak, başarı için 0). Burada 50 geçerli kod tahsis edilecektir ve bu, komut dosyalarında sorun gidermeyi daha kolay hale getirecektir.

Tüm bu belgeye ekli örneklerdeki kullanıcı tanımlı çıkış kodları, Örnek 9.2 gibi bazı durumlar dışında, şimdi bu standarda uygun hale gelmiştir.

Bir kabuk betiğinden çıktıktan sonra komut satırından $? çalıştırılması, sadece Bash veya *sh* isteminden yukarıdaki tablo ile tutarlı sonuçlar verir. C-kabuğu veya *tcsh* çalışıyorsa farklı değerler verebilir.

Notlar

[1] Aralık dışı çıkış değerleri öngörülemeyen çıkış kodlarına neden olabilir. Örneğin, **exit 3809**, 225 çıkış kodu verir.

EK D. G/Ç ve G/Ç YENİDEN YÖNLENDİRMESİNE AYRINTILI BİR GİRİŞ

Stephane Chazelas tarafından yazılmış ve belge yazarı tarafından gözden geçirilmiştir.

Bir komut ilk üç dosya tanımlayıcısının kullanılabilir olmasını bekler. Birincisi, *fd 0* (standart giriş, `stdin`), okuma içindir. Diğer ikisi (*fd 1*, `stdout` ve *fd 2*, `stderr`) yazma içindir.

Her komut ile ilişkili bir stdin, stdout, ve stderr vardır. **`ls 2>&1`** geçici olarak **ls** komutunun `stderr`'ini aynı "kaynak" için kabuk `stdout`'una bağlamak anlamına gelir.

Geleneksel olarak, bir komut, fd 0 (`stdin`)'dan girdi okur, normal çıktısını fd 1 (`stdout`)'a yazar, ve hata çıktısını fd 2 (`stderr`)'a yazar. Bu üç fd'den en az biri açık değilse, sorunlarla karşılaşabilirsiniz:

```
bash$ cat /etc/passwd >&-

cat: standard output: Bad file descriptor
```

Örneğin, **xterm** çalıştığında, ilk kendisini başlatır. Kullanıcının kabuğu çalıştırılmadan önce, **xterm** terminal aygıtı (/dev/pts/<n> veya benzeri) üç defa açılır.

Bu noktada, Bash bu üç dosya tanımlayıcılarını devralır, ve Bash tarafından çalıştırılan her komut (çocuk süreç), komut yönlendirme dışında, sırayla onları devralır. Yeniden yönlendirme dosya tanımlayıcılarından birini başka bir dosya için (ya da bir oluk, ya da izin verilen başka bir şey için) yeniden atama anlamına gelir. Dosya tanımlayıcıları (bir komut, bir komut grubu, bir alt kabuk, bir while, if, case, veya for döngüsü için) yerel olarak yeniden atanabildiği gibi, kabuğun geri kalanı için de (exec kullanarak) global olarak yeniden atanabilir.

`ls > /dev/null` komutunun anlamı, **ls**'in fd 1'i `/dev/null`'a bağlı olarak çalışmaktadır.

```
bash$ lsof -a -p $$ -d0,1,2

COMMAND     PID     USER     FD     TYPE     DEVICE     SIZE     NODE NAME
bash        363     bozo     0u     CHR       136,1      3       /dev/pts/1
bash        363     bozo     1u     CHR       136,1      3       /dev/pts/1
bash        363     bozo     2u     CHR       136,1      3       /dev/pts/1



bash$ exec 2> /dev/null

bash$ lsof -a -p $$ -d0,1,2
```

```
COMMAND     PID     USER     FD     TYPE     DEVICE     SIZE     NODE NAME
bash        371     bozo     0u     CHR      136,1      3        /dev/pts/1
bash        371     bozo     1u     CHR      136,1      3        /dev/pts/1
bash        371     bozo     2w     CHR      1,3        120      /dev/null

bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat

COMMAND     PID     USER     FD     TYPE     DEVICE     SIZE     NODE NAME
lsof        379     root     0u     CHR      136,1      3        /dev/pts/1
lsof        379     root     1w     FIFO     0,0        7118     pipe
lsof        379     root     2u     CHR      136,1      3        /dev/pts/1

bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"

COMMAND     PID     USER     FD     TYPE     DEVICE     SIZE     NODE NAME
lsof        426     root     0u     CHR      136,1      3        /dev/pts/1
lsof        426     root     1w     FIFO     0,0        7520     pipe
lsof        426     root     2w     FIFO     0,0        7520     pipe
```

Bu, yönlendirmenin farklı türleri için çalışır.

Alıştırma: Aşağıdaki komut dosyasını analiz edin.

```
#! /usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2

while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 &

exec 7> /tmp/fifo1

exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1

(

  (

    (

      while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee
/dev/stderr | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 &

      exec 3> /tmp/fifo2

      echo 1st, to stdout

      sleep 1

      echo 2nd, to stderr >&2
```

```
      sleep 1
      echo 3rd, to fd 3 >&3
      sleep 1
      echo 4th, to fd 4 >&4
      sleep 1
      echo 5th, to fd 5 >&5
      sleep 1
      echo 6th, through a pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
      sleep 1
      echo 7th, to fd 6 >&6
      sleep 1
      echo 8th, to fd 7 >&7
      sleep 1
      echo 9th, to fd 8 >&8
    ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
  ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-

rm -f /tmp/fifo1 /tmp/fifo2
# For each command and subshell, figure out which fd points to what.
exit 0
```

EK E. YERELLEŞTİRME

Yerelleştirme belgelenmemiş bir Bash özelliğidir. Yerelleştirilmiş kabuk betiği metin çıktısını, sistemde yerel olarak tanımlanan dilde gösterir. Berlin, Almanya'daki bir Linux kullanıcısı, komut dosyasının çıktısını Almanca olarak alır. Maryland'deki kuzeni ise aynı çıktıyı İngilizce dilinde alacaktır.

Yerelleştirilmiş bir komut dosyası oluştururken, kullanıcıya tüm mesajları (hata mesajları, istemler, vb) yazmak için aşağıdaki şablonu kullanın.

```
#!/bin/bash

# localized.sh

E_CDERROR=65

error()

{

  printf "$@" >&2

  exit $E_CDERROR

}

cd $var || error $"Can't cd to %s." "$var"

read -p $"Enter the value: " var

# ...
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
 "Enter the value: "
```

Bu, tüm yerelleştirilmiş metni listeleyecektir. (`-D` seçeneği, komut dosyasını çalıştırmaksızın $ ile başlayan çift tırnak dizelerini listeler.)

```
bash$ bash --dump-po-strings localized.sh

#: a:6

msgid "Can't cd to %s."

msgstr "" #: a:7

msgid "Enter the value: "

msgstr ""
```

`--dump-po-strings` Bash seçeneği `-D` seçeneğine benzer, ama gettext "po" biçimini kullanır.

Şimdi, komut dosyasının tercüme edileceği her dil için *msgstr*'in belirtildiği bir `language.po` dosyasını oluşturun. Örnek olarak:

fr.po:

```
#: a:6

msgid "Can't cd to %s."

msgstr "Impossible de se positionner dans le répertoire %s."

#: a:7

msgid "Enter the value: "

msgstr "Entrez la valeur : "
```

Daha sonra, **msgfmt** çalıştırın.

```
msgfmt -o localized.sh.mo fr.po
```

Çıkan `localized.sh.mo` dosyasını `/usr/local/share/locale/fr/LC_MESSAGES` dizinine yerleştirin, ve komut dosyasının başında aşağıdaki satırları ekleyin:

```
TEXTDOMAINDIR=/usr/local/share/locale

TEXTDOMAIN=localized.sh
```

Bir Fransız kullanıcı, sistemde komut dosyasını çalıştırırsa, Fransızca mesajlar alacaktır.

Bash veya diğer kabukların eski sürümleri ile yerelleştirme, `-s` seçeneğini kullanarak, gettext gerektirir.

Bu durumda, komut dosyası şu şekilde olur:

```
#!/bin/bash

# localized.sh

E_CDERROR=65

error()

{

  local format=$1
```

```
  shift
  printf "$(gettext -s "$format")" "$@" >&2
  exit $E_CDERROR
}
cd $var || error "Can't cd to %s." "$var"
read -p "$(gettext -s "Enter the value: ")" var
# ...
```

`TEXTDOMAIN` ve `TEXTDOMAINDIR` değişkenlerinin çevreye aktarılması gerekir.

Bu ek Stephane Chazelas tarafından yazılmıştır.

EK F. TARİH KOMUTLARI

Bash kabuğu, kullanıcının komut geçmişini düzenleme ve işlemek için komut satırı araçları sağlar. Bu öncelikle bir kolaylık, tuş vuruşlarından tasarruf sağlayan bir araçtır.

Bash tarih komutları:

1. **history**
2. **fc**

```
bash$ history
   1 mount /mnt/cdrom
   2 cd /mnt/cdrom
   3 ls
   ...
```

Bash tarih komutları ile ilişkili iç değişkenler:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. !!
8. !$
9. !#
10. !N
11. !-N
12. !STRING
13. !?STRING?
14. ^STRING^string^

Ne yazık ki, Bash tarih araçları komut dosyalarında hiçbir kullanım alanı bulamaz.

```
#!/bin/bash
# history.sh
# Attempt to use 'history' command in a script.

history

# Script produces no output.
# History commands do not work within a script.
```

```
bash$ ./history.sh
(no output)
```

EK G. ÖRNEK .bashrc DOSYASI

~/.bashrc dosyası etkileşimli kabukların davranışını belirler. Bu dosyayı iyice incelemek Bash'in daha iyi anlaşılmasına yol açacaktır.

Emmanuel Rouat bir Linux sistemi için yazılmış olan aşağıdaki çok ayrıntılı .bashrc dosyasına katkıda bulunmuştur. Okuyucun geribildirimini beklemektedir.

Dosyayı dikkatli bir şekilde çalışınız ve oradaki kod parçacıkları ve işlevlerini kendi .bashrc dosyanızda veya kendi komut dosyalarınızda yeniden kullanmayı deneyiniz.

ÖRNEK G.1 ÖRNEK .bashrc DOSYASI

```
#=============================================================
#
# PERSONAL $HOME/.bashrc FILE for bash-2.05 (or later)
#
# This file is read (normally) by interactive shells only.
# Here is the place to define your aliases, functions and
# other interactive features like your prompt.
#
# This file was designed (originally) for Solaris.
# --> Modified for Linux.
# This bashrc file is a bit overcrowded - remember it is just
# just an example. Tailor it to your needs
#
#=============================================================
# --> Comments added by HOWTO author.
#-----------------------------------
# Source global definitions (if any)
#-----------------------------------
if [ -f /etc/bashrc ]; then
    . /etc/bashrc                # --> Read /etc/bashrc, if present.
fi
#--------------------------------------------------------------
# Automatic setting of $DISPLAY (if not set already)
# This works for linux and solaris - your mileage may vary....
#--------------------------------------------------------------
if [ -z ${DISPLAY:=""} ]; then
    DISPLAY=$(who am i)
    DISPLAY=${DISPLAY%%\!*}
    if [ -n "$DISPLAY" ]; then
        export DISPLAY=$DISPLAY:0.0
    else
        export DISPLAY=":0.0"     # fallback
    fi
fi
#---------------
# Some settings
#---------------
set -o notify
set -o noclobber
set -o ignoreeof
```

Yazar Mendel Cooper'ın izni alınarak ticari olmayan amaçla Türkçe'ye çevrilmiştir.

```
set -o nounset
#set -o xtrace           # useful for debuging

shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s histappend histreedit
shopt -s extglob               # useful for programmable completion
#-----------------------
# Greeting, motd etc...
#-----------------------
# Define some colors first:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'                     # No Color
# --> Nice. Has the same effect as using "ansi.sys" in DOS.
# Looks best on a black background.....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on
${RED}$DISPLAY${NC}\n"
date

if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s        # makes our day a bit more fun.... :-)
fi

function _exit()               # function to run upon exit of shell
{
    echo -e "${RED}Hasta la vista, baby${NC}"
}
trap _exit 0
#---------------
# Shell prompt
#---------------
function fastprompt()
{
    unset PROMPT_COMMAND
    case $TERM in
        *term | rxvt )
            PS1="[\h] \W > \[33]0;[\u@\h] \w07\]" ;;
        *)
            PS1="[\h] \W > " ;;
    esac
}

function powerprompt()
{
    _powerprompt()
    {
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
        TIME=$(date +%H:%M)
    }
    PROMPT_COMMAND=_powerprompt
    case $TERM in
        *term | rxvt )
            PS1="${cyan}[\$TIME \$LOAD]$NC\n[\h \#] \W > \[33]0;[\u@\h] \w07\]" ;;
        linux )
```

Yazar Mendel Cooper'ın izni alınarak ticari olmayan amaçla Türkçe'ye çevrilmiştir.

```
            PS1="${cyan}[\$TIME - \$LOAD]$NC\n[\h \#] \w > " ;;
        * )
            PS1="[\$TIME - \$LOAD]\n[\h \#] \w > " ;;
    esac
}

powerprompt            # this is the default prompt - might be slow
                       # If too slow, use fastprompt instead....
#============================================================
#
# ALIASES AND FUNCTIONS
#
# Arguably, some functions defined here are quite big
# (ie 'lowercase') but my workstation has 512Meg of RAM, so .....
# If you want to make this file smaller, these functions can
# be converted into scripts.
#
# Many functions were taken (almost) straight from the bash-2.04
# examples.
#
#============================================================
#-------------------
# Personnal Aliases
#-------------------
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# -> Prevents accidentally clobbering files.
alias h='history'
alias j='jobs -l'
alias r='rlogin'
alias which='type -all'
alias ..='cd ..'
alias path='echo -e ${PATH//:/\\n}'
alias print='/usr/bin/lp -o nobanner -d $LPDEST'  # Assumes LPDEST is defined
alias pjet='enscript -h -G -fCourier9 -d $LPDEST' # Pretty-print using enscript
alias background='xv -root -quit -max -rmode 5'   # put a picture in the background
alias vi='vim'
alias du='du -h'
alias df='df -kh'

# The 'ls' family (this assumes you use the GNU ls)
alias ls='ls -hF --color'        # add colors for filetype recognition
alias lx='ls -lXB'               # sort by extension
alias lk='ls -lSr'               # sort by size
alias la='ls -Al'                # show hidden files
alias lr='ls -lR'                # recursice ls
alias lt='ls -ltr'               # sort by date
alias lm='ls -al |more'          # pipe through 'more'
alias tree='tree -Cs'            # nice alternative to 'ls'

# tailoring 'less'
alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-'  # Use this if lesspipe.sh exists
export LESS='-i -N -w -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# spelling typos - highly personnal :-)
alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
```

```
alias kk='ll'

#----------------
# a few fun ones
#----------------
function xtitle ()
{
    case $TERM in
        *term | rxvt)
            echo -n -e "33]0;$*07" ;;
        *) ;;
    esac
}

# aliases...
alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'
alias ncftp="xtitle ncFTP ; ncftp"

# .. and functions
function man ()
{
    xtitle The $(basename $1|tr -d .[:digit:]) manual
    man -a "$*"
}

function ll() { ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -v "^d|total "; }
function xemacs() { { command xemacs -private $* 2>&- & } && disown ;}
function te()     # wrapper around xemacs/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( xemacs "$@" & );
    Fi
}

#-----------------------------------
# File & strings related functions:
#-----------------------------------
function ff() { find . -name '*'$1'*' ; }                # find a file
function fe() { find . -name '*'$1'*' -exec $2 {} \; ; } # find a file and run $2
on it
function fstr()             # find a string in a set of files
{
    if [ "$#" -gt 2 ]; then
        echo "Usage: fstr \"pattern\" [files] "
        return;
    fi
    SMSO=$(tput smso)
    RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print | xargs grep -sin "$1" | \
sed "s/$1/$SMSO$1$RMSO/gI"
}

function cuttail()         # cut last n lines in file, 10 by default
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase()       # move filenames to lowercase
{
    for file ; do
```

```
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()                 # swap 2 filenames around
{
    local TMPFILE=tmp.$$
    mv $1 $TMPFILE
    mv $2 $1
    mv $TMPFILE $2
}

#-------------------------------------
# Process/system related functions:
#-------------------------------------
function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }
# This function is roughly the same as 'killall' on Linux
# but has no equivalent (that I know of) on Solaris

function killps()                   # kill by process name
{
    local pid pname sig="-TERM"    # default signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Usage: killps [-SIGNAL] pattern"
        return;
    fi

    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
        pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
        if ask "Kill process $pid  with signal $sig?"
            then kill $sig $pid
        fi
    done
}

function my_ip()                    # get IP adresses
{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -e s/P-t-P://)
}

function ii()                       # get current host related info
{
    echo -e "\nYou are logged on ${RED}$HOST"
    echo -e "\nAdditionnal information:$NC " ; uname -a
    echo -e "\n${RED}Users logged on:$NC " ; w -h
    echo -e "\n${RED}Current date :$NC " ; date
    echo -e "\n${RED}Machine stats :$NC " ; uptime
    echo -e "\n${RED}Memory stats :$NC " ; free my_ip 2>&- ;
    echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-"Not connected"}
```

```
    echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
    echo
}

# Misc utilities:
function repeat()                   # repeat n times command
{
    local i
    max max=$1; shift;
    for ((i=1; i  C-like syntax
        eval "$@";
    done
}

function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#============================================================================
#
# PROGRAMMABLE COMPLETION - ONLY SINCE BASH-2.04
# (Most are taken from the bash 2.05 documentation)
# You will in fact need bash-2.05 for some features
#
#============================================================================

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
    echo "You will need to upgrade to version 2.05 for programmable completion"
    return
fi
shopt -s extglob          # necessary
set +o nounset            # otherwise some completions will fail
complete -A hostname rsh rcp telnet rlogin r ftp ping disk
complete -A command nohup exec eval trace gdb
complete -A command command type which
complete -A export printenv
complete -A variable export local readonly unset
complete -A enabled builtin
complete -A alias alias unalias
complete -A function function
complete -A user su mail finger
complete -A helptopic help             # currently same as builtins
complete -A shopt shopt
complete -A stopped -P '%' bg
complete -A job -P '%' fg jobs disown
complete -A directory mkdir rmdir
complete -A directory -o default cd
complete -f -d -X '*.gz' gzip
complete -f -d -X '*.bz2' bzip2
complete -f -o default -X '!*.gz' gunzip
complete -f -o default -X '!*.bz2' bunzip2
complete -f -o default -X '!*.pl' perl perl5
complete -f -o default -X '!*.ps' gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.+(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
```

```
complete -f -o default -X '!*.+(jpg|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.mp3' mpg123
complete -f -o default -X '!*.ogg' ogg123

# This is a 'universal' completion function - it works when commands have
# a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'
_universal_func ()
{
    case "$2" in
        -*) ;;
        *) return ;;
    esac

    case "$1" in
        \~*) eval cmd=$1 ;;
        *) cmd="$1" ;;
    esac
    COMPREPLY=( $("$cmd" --help | sed -e '/--/!d' -e 's/.*--\([^ ]*\).*/--\1/'| \
grep ^"$2" |sort -u) )
}

complete -o default -F _universal_func ldd wget bash id info

_make_targets ()
{
    local mdef makef gcmd cur prev i
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}
    # if prev argument is -f, return possible filename completions.
    # we could be a little smarter here and return matches against
    # `makefile Makefile *.mk', whatever exists
    case "$prev" in
        -*f)  COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # if we want an option, return the possible posix options
    case "$cur" in
        -) COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac
    # make reads `makefile' before `Makefile'
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk           # local convention
    fi
    # before we scan for targets, see if a makefile name was specified
    # with -f
    for (( i=0; i /dev/null | awk 'BEGIN {FS=":"} /^[^.# ][^=]*:/ {print $1}' | tr
-s ' ' '12' | sort -u | eval $gcmd ) )
}

complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake

_configure_func ()
{
    case "$2" in
        -*) ;;
        *) return ;;
    esac
```

```
    case "$1" in
        \~*) eval cmd=$1 ;;
        *) cmd="$1" ;;
    esac

    COMPREPLY=( $("$cmd" --help | awk '{if ($1 ~ /--.*/) print $1}' | grep ^"$2" |
sort -u) )
}

complete -F _configure_func configure

# cvs(1) completion
_cvs ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}
    if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
        COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
        export history import log rdiff release remove rtag status \
        tag update' $cur ))
    else
        COMPREPLY=( $( compgen -f $cur ))
    fi
    return 0
}

complete -F _cvs cvs

_killall ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    # get a list of processes (the first sed evaluation
    # takes care of swapped out processes, the second
    # takes care of getting the basename of the process)
    COMPREPLY=( $( /usr/bin/ps -u $USER -o comm | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))
    return 0
}

complete -F _killall killall killps
# Local Variables:
# mode:shell-script
# sh-shell:bash
# End:
```

EK H. DOS TOPLU DOSYALARINI KABUK BETİKLERİNE DÖNÜŞTÜRME

Pek çok programcı betiklemeyi DOS çalıştıran bir bilgisayarda öğrendi. DOS toplu iş dosyası dili bile genellikle geçici çözümler gerektiren bazı oldukça güçlü komut ve uygulamaları yazmaya izin verdi. Hala bazen, UNIX kabuğu için eski DOS toplu iş dosyasını dönüştürme gereksinimi ortaya çıkmaktadır. Bu genellikle zor değildir, çünkü DOS toplu iş dosyası operatörleri kendilerine eşdeğer kabuk komut dosyası operatörlerinin sadece sınırlı bir alt kümesidir.

TABLO H.1 TOPLU DOSYALARIN ANAHTAR KELİME/DEĞİŞKEN/OPERATÖRLERİ, ve ONLARIN KABUK EŞDEĞERLERİ

Toplu Dosya Operatörü	Kabuk Betiği Eşdeğeri	Anlamı
`%`	`$`	komut satırı parametresi ön-eki
`/`	`-`	komut seçeneği bayrağı
`\`	`/`	dizin yolunu ayırıcı
`==`	`=`	(eşit-için) dize karşılaştırma testi
`!==!`	`!=`	(eşit-değil-için) dize karşılaştırma testi
`|`	`|`	oluk
`@`	`set +V`	geçerli komutu göstermemek
`*`	`*`	"joker" dosya adı
`>`	`>`	dosya yeniden yönlendirmesi (üzerine yazar.)
`>>`	`>>`	dosya yeniden yönlendirmesi (sonuna ekler.)
`<`	`<`	`stdin` yönlendirmesi
`%VAR%`	`$VAR`	çevre değişkeni
`REM`	`#`	yorum
`NOT`	`!`	izleyen testin olumsuzu

NUL	/dev/null	komut çıktısını gömmek için "kara delik"
ECHO	echo	echo (Bash'te daha çok seçenek vardır.)
ECHO.	echo	echo (boş satır)
ECHO OFF	set +V	izleyen komutları göstermemek
FOR %%VAR IN (LIST) DO	for var in [list]; do	"for" döngüsü
:LABEL	hiçbiri (gereksiz)	label
GOTO	hiçbiri (fonksiyon kullanın)	betikte başka bir yere atlamak
PAUSE	sleep	duraklamak veya bir aralık beklemek
CHOICE	case veya select	menü seçimi
IF	İf	if-testi
IF EXIST FILENAME	if [-e filename]	dosyanın var olduğu testi
IF !%N==!	if [-z "$N"]	değiştirilebilir parametre "N" mevcut değilse
CALL	kaynak veya nokta (.) işleci	başka bir komut dosyasını "dahil et"
COMMAND /C	kaynak veya nokta (.) işleci	başka bir komut dosyasını "dahil et" (CALL ile aynı)
SET	export	bir çevre değişkenini ayarla
SHIFT	shift	komut satırı argüman listesini sola kaydır
SGN	-lt veya –gt	(tamsayı) işareti
ERRORLEVEL	$?	çıkış durumu
CON	stdin	"konsol" stdin
PRN	/dev/lp0	(genel) yazıcı aygıtı
PRN	/dev/lp0	birinci yazıcı aygıtı
COM1	/dev/ttyS0	birinci seri port

Toplu iş dosyaları genellikle DOS komutları içerir. Bir toplu iş dosyasını, kabuk betiğine dönüştürmek için bunlar, UNIX eşdeğerlerine tercüme edilmelidir.

TABLO H.2 DOS KOMUTLARI ve BUNLARIN UNIX EŞDEĞERLERİ

DOS Komutu	UNIX Eşdeğeri	Etkisi
ASSIGN	ln	Dosya ya da dizini bağlar.
ATTRIB	chmod	Dosya izinlerini değiştirir.
CD	cd	Dizin değiştirir.
CHDIR	cd	Dizin değiştirir.
CLS	clear	Ekran görüntüsünü temizler.
COMP	diff, comm, cmp	Dosyaları karşılaştırır.
COPY	cp	Dosyayı kopyalar.
Ctl-C	Ctl-C	break (sinyal)
Ctl-Z	Ctl-D	EOF (dosya-sonu)
DEL	rm	Dosya(lar)ı siler.
DELTREE	rm –rf	Dizinleri ardışık olarak siler.
DIR	ls –l	Dizinleri listeler.
ERASE	rm	Dosya(lar)ı siler.
EXIT	exit	Geçerli işlemden çıkar.
FC	comm, cmp	Dosyaları karşılaştırır.
FIND	grep	Dosyalarda dizeleri bulur.
MD	mkdir	Dizin oluşturur.
MKDIR	mkdir	Dizin oluşturur.
MORE	more	Metin dosyasını sayfa sayfa gösterir.
MOVE	mv	Dosya(lar)ı taşır.

PATH	$PATH	Yürütülebilirlerin bulunduğu yol
REN	mv	Dosyayı yeniden adlandırır (taşır).
RD	rmdir	Dizini siler (kaldırır).
SORT	sort	Dosyayı sıralar.
TIME	date	Sistem saatini görüntüler.
TYPE	cat	Çıktı dosyasını stdout'a yazar.
XCOPY	cp	Dosya kopyalar.(genişletilmiş)

Hemen hemen tüm UNIX ve kabuk operatörleri ve komutlarının DOS ve toplu iş dosyası benzerlerine göre daha birçok seçeneği ve geliştirmeleri vardır. Birçok DOS toplu iş dosyaları read'in benzeri olan **ask.com** gibi yardımcı araçlara güvenmektedir.

DOS sadece * ve? karakterlerini tanıyarak, dosya adı joker genişlemesinin çok sınırlı ve uyumsuz bir alt kümesini destekler.

Bir DOS toplu iş dosyasını kabuk betiğine dönüştürmek genellikle basittir, ve sonuç çoğu kez orijinalden daha iyi okunur.

ÖRNEK H.1 VIEWDATA.BAT: DOS TOPLU DOSYASI

```
REM VIEWDATA
REM INSPIRED BY AN EXAMPLE IN "DOS POWERTOOLS"
REM BY PAUL SOMERSON

@ECHO OFF
IF !%1==! GOTO VIEWDATA
REM IF NO COMMAND-LINE ARG...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM PRINT LINE WITH STRING MATCH, THEN EXIT.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM SHOW ENTIRE FILE, 1 PAGE AT A TIME.

:EXIT0
```

Komut dosyası dönüşümünün geliştirmesi şöyledir:

ÖRNEK H.2 viewdata.sh: viewdata.bat İÇİN ÖNERİLEN KABUK BETİĞİ DÖNÜŞÜMÜ

```
#!/bin/bash
# Conversion of VIEWDATA.BAT to shell script.
DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1
# @ECHO OFF Command unnecessary here.
if [ $# -lt "$ARGNO" ]         # IF !%1==! GOTO VIEWDATA
then
  less $DATAFILE               # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $DATAFILE          # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi
exit 0                         # :EXIT0
# GOTOs, labels, smoke-and-mirrors, and flimflam unnecessary.
# The converted script is short, sweet, and clean,
# which is more than can be said for the original.
```

Ted Davis tarafından yazılan PC Üzerinde Kabuk Betikleme toplu iş dosyası programlamanın eski moda sanatı hakkında kapsamlı bir dizi öğretici içermektedir. Onun ustaca tekniklerinin bazıları kabuk betikleri için makul ve önemli olabilir.

EK I. ALIŞTIRMALAR

I.1 KOMUT DOSYASI ANALİZİ

Aşağıdaki komut dosyasını inceleyiniz. Çalıştırınız, sonra ne yaptığını anlatınız. Komut dosyasına açıklamalar ekleyiniz, sonra daha küçük ve zarif bir şekilde yeniden yazınız.

```
#!/bin/bash
MAX=10000
  for((nr=1; nr<$MAX; nr++))
  do
    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
      continue
    fi

  break # What heppens when you comment out this line? Why?
  done
  echo "Number = $nr"
exit 0
```

Bir okuyucum aşağıdaki kod parçasını gönderdi.

```
while read LINE
do
  echo $LINE
done < `tail -f /var/log/messages`
```

Okuyucum sistem günlük dosyasına, `/var/log/messages`, yapılan değişiklikleri izlemek için komut dosyası yazmak istiyordu. Ne yazık ki, yukarıdaki kod bloğu asılı kalıyor ve yararlı hiçbir şey yapmıyor. Neden? İşi yapacak şekilde kodu değiştiriniz (ipucu: döngünün `stdin`'ini yönlendirme yerine, bir oluk deneyiniz).

Örnek A.10'u analiz ediniz, ve basitleştirilmiş ve daha mantıklı tarzda yeniden düzenleyiniz. Kaç tane değişkenin ortadan kaldırılabileceğini görünüz ve yürütme zamanı hızlandırmak için komut dosyasını optimize etmeyi deneyiniz.

Birinci "nesil" girdisi olarak sıradan bir ASCII metin dosyasını kabul edecek şekilde komut dosyasını değiştiriniz. Komut, ilk `$SATIR*$SÜTUN` karakterleri okuyacak ve ünlüleri "yaşayan" hücreler olarak atayacaktır. İpucu: Giriş dosyasındaki boşlukları alt çizgi karakteri olarak çevirdiğinizden emin olunuz.

I.2 KOMUT DOSYASI YAZMA

Aşağıdaki görevlerden her birini yürütmek için bir komut dosyası yazınız.

Basit

Ana Dizin Listeleme

Kullanıcının ev dizini üzerinde bir özyinelemeli dizin listesi yapınız ve bir dosyaya bilgileri kaydediniz. Dosyayı sıkıştırınız, komut dosyasının kullanıcıdan bir flopi disket girmesini, ve sonra ENTER tuşuna basmasını istemesini sağlayınız. Son olarak, dosyayı diskete kaydediniz.

For döngülerini while ve until döngülerine çevirme

Örnek 10.1'deki *for* döngülerini *while* döngülerine çevirin. İpucu: verileri bir dizilimde saklayınız ve dizilim elemanları üzerinde adım adım ilerleyiniz.

Bu "ağır iş"i yaptıktan sonra, şimdi örnekteki döngüleri *until* döngülerine çeviriniz.

Bir metin dosyasının satır aralığı değiştirme

Bir hedef dosyanın her satırını okuyan, sonra satırın ardından gelen ve fazladan bir boş satırı stdout'a geri yazan bir komut dosyası yazınız. Bunun etkisi, dosyayı çift satır aralıklı yapmaktır.

Komut dosyasının komut satırından gerekli argümanı (bir dosya adı) alıp almadığını ve belirtilen dosyanın var olup olmadığını kontrol etmek için tüm gerekli kodu ekleyiniz.

Komut dosyası düzgün çalıştığında, hedef dosyayı üç satır aralıklı yapmak için gerekli değişiklikleri yapınız.

Son olarak, hedef dosyadan tüm boş satırları kaldırarak onu tek aralıklı yapan bir komut dosyası yazınız.

Ters Liste

stdout'a kendini geriye doğru yazan bir komut dosyası yazınız.

Otomatik Dekompresyon Dosyaları

Girdi olarak bir dosya listesi girildiğinde, bu komut dosyası hedef dosyaların her birini (file komutunun çıktısını ayrıştırarak) kullanılan sıkıştırma türü için sorgular. Bundan sonra komut dosyası otomatik olarak (**gunzip**, **bunzip2**, **unzip**, **uncompress**, ya da her neyse) uygun dekompresyon komutunu çağırır. Bir hedef dosya sıkıştırılmış değilse, komut dosyası bir uyarı mesajı verir, ama o dosya üzerinde başka hiçbir harekete geçmez.

Tek Sistem Kimliği

Bilgisayarınız için "benzersiz" bir 6 basamaklı onaltılık tanımlayıcı oluşturunuz. Kusurlu hostid komutunu *kullanmayınız.* İpucu: md5sum `/etc/passwd` çıktısının ilk 6 hanesini seçiniz.

Yedekleme

Ana dizininiz altında (`/home/adınız`) son 24 saat içinde değiştirilmiş tüm dosyaları (`*.tar.gz` dosyası olarak) arşivleyiniz. İpucu: find kullanınız.

Asal Sayılar

60000 ve 63000 arasındaki tüm asal sayıları (`stdout`'a) yazdırınız. Çıktı sütunları güzelce biçimlendirilmiş olmalıdır. İpucu: printf kullanınız.

Piyango Numaraları

Birinci piyango tipi 1-50 arasında beş farklı numara çekmeyi içeriyor. Bu aralıkta beş rasgele sayı üreten bir komut dosyası yazın. Sayıların hepsi birbirinden *farklı* olmalıdır. Komut dosyası numaraları, oluşturuldukları tarih ve saat bilgileri ile birlikte `stdout`'a veya bir dosyaya yazma seçeneği sunmalıdır.

Orta Derecede Zor

Disk Alanı Yönetimi

Her seferinde bir tane olmak üzere, `/home/username` dizin ağacı altındaki 100K'dan daha büyük tüm dosyaları listeleyiniz. Kullanıcıya dosyayı silme veya sıkıştırma seçeneği sununuz, sonra bir sonraki dosyayı göstermek için devam ediniz. Bir günlük dosyasına tüm silinmiş dosyaları ve silme zamanlarını yazınız.

Güvenli Silme

Bir komut dosyası olarak, "güvenli" bir silme komutu, `srm.sh` yazınız. Bu komut için komut satırı argümanları olarak geçirilen dosya adları silinmez, bunun yerine sıkıştırılmış değilse gzip'lenir (kontrol etmek için file kullanabilirsiniz), sonra `/home/username/trash` dizinine taşınır. Çağırma sırasında, komut dosyası "çöp" dizini kontrol eder ve 48 saat'ten daha eski olan dosyaları siler.

İkinci Dereceden Denklemler

Ax^2 + Bx + C = 0 şeklindeki "ikinci" dereceden bir denklemi çözünüz. Bir komut dosyası, argüman olarak **A**, **B**, ve **C** katsayılarını alsın, ve dört ondalık basamağa kadar çözümler döndürsün.

İpucu: *x = (-B +/- sqrt(B^2 - 4AC)) / 2A* formülünü kullanarak katsayılar için bc'ye oluk yapınız.

Eşleşen Sayıların Toplamı

10000 - 99999 aralığında olan ve { 4, 5, 6 } basamaklarından *tam olarak iki* tanesini içeren tüm beş-basamaklı sayıların toplamını bulunuz. Bunlar, aynı sayıda içinde tekrarlayabilir, ve eğer öyleyse, her bir oluş için bir defa olacak şekilde sayılmalıdır.

Eşleşen bazı numara örnekleri arasında 42057, 74638 ve 89515 bulunmaktadır.

Şanslı Numaralar

Bir "şanslı numara" basamaklarının art arda toplamı 7 olan sayılardır. Örneğin, 62431 bir "şanslı numara"dır (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). 1000-10000 arasındaki tüm "şanslı numaraları" bulun.

Bir Dizinin Alfabetik Okunuşu

Komut satırından girilen rastgele bir diziyi (ASCII sırasına göre) alfabetik olarak okuyunuz.

Ayrıştırma

`/etc/passwd` dosyasını ayrıştırınız, ve içeriğinin çıktısını güzel, kolay okunur tablo şeklinde gösteriniz.

Bir Veri Dosyasının Güzel-Baskısı

Bazı veritabanı ve elektronik tablo paketleri *virgülle ayrılmış değerler* (CSV'ler) ile kayıt dosyaları kullanırlar. Diğer uygulamalar genellikle bu dosyaları ayrıştırmaya gereksinim duyarlar. Aşağıdaki gibi virgülle ayrılmış alanları verilen bir veri dosyası :

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989

Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612

...
```

Veriyi yeniden biçimlendirerek, etiketli ve eşit aralıklı sütunlar şeklinde `stdout`'a çıktısını yazdırınız.

Şifreler

[0-9], [A-Z], [a-z] aralığındaki karakterleri kullanarak, 8-harfli rastgele şifreler oluşturunuz. Her şifrenin en az iki hanesi sayı basamağı olmalıdır.

Zor

Dosya Erişimlerini Günlüğe İşleme

Bir gün boyunca `/etc` dizini altındaki dosyalara yapılan tüm erişimleri günlüğe işleyiniz. Bu bilgiler dosya adı, kullanıcı adı ve erişim süresini içermelidir. Dosyalarda herhangi bir değişiklik gerçekleşecek ise, bayraklarla bu belirtilmelidir. Bu veriyi düzgün biçimlendirilmiş kayıtlar halinde bir günlük dosyasına yazınız.

Yorum Silme

Adı komut satırında belirtilen kabuk komut dosyası içindeki tüm yorumları siliniz. Unutmayınız ki, "#! satırı" silinmemelidir.

HTML Dönüştürme

Belirli bir metin dosyasını HTML'e dönüştürün. Bu etkileşimli-olmayan komut dosyası otomatik olarak tüm uygun HTML etiketlerini argüman olarak belirtilen bir dosyaya ekler.

HTML Etiketlerini Silme

Belirli bir HTML dosyası içindeki tüm HTML etiketleri siliniz, sonra 60 ve 75 karakter uzunluğu arasındaki satırlar içine yeniden biçimlendiriniz. Uygun olacak şekilde, paragraf ve blok aralığını sıfırlayınız, ve HTML tablolarını yaklaşık metin eşdeğerlerine dönüştürünüz.

XML Dönüşümü

Bir XML dosyasını hem HTML ve hem de metin biçimine dönüştürünüz.

Mors Alfabesi

Bir metin dosyasını Mors koduna dönüştürünüz. Metin dosyasının her karakteri bir sonraki gelenden boşluk ile ayrılmış olarak noktalar ve çizgilerin (çizgi) karşılık geldiği bir Mors kodu grubu olarak temsil edilecektir. Örneğin, "script" ===> "... _._. ._. .. .__. _".

Hex Dökümü

Argüman olarak belirtilen bir ikili dosya üzerinde hex (adecimal) döküm yapınız. Çıktı düzgün sekmeli alanlarda olmalıdır, birinci alan adresi gösterirken, sonraki 8 alan 4-byte hex sayı ve son alan da önceki 8 alanın ASCII-eşdeğeri olmalıdır.

Bir Kaydıran Yazmacın Taklidi

İlham kaynağı olarak Örnek 26.6'yı kullanarak, 64-bit kaydıran yazmacı bir dizilim olarak taklit eden bir komut dosyası yazınız. Yazmacı *yükleyen*, *sola kaydıran* ve *sağa kaydıran* fonksiyonları uygulayınız. Son olarak, sekiz 8-bit ASCII karakter olarak yazmaç içeriğini yorumlayan bir fonksiyon yazınız.

Belirteç

4 x 4 belirtecini çözünüz.

Gizli Kelimeler

Bir "kelime bulma" bulmacası üreteci, bir başka deyişle rasgele harflerin saklandığı bir 10 x 10 matris içinde 10 giriş kelimesini gizleyen bir komut dosyası yazınız. Kelimeler yatay, dikey, ya da çapraz yönde gizlenmiş olabilir.