

Programowanie w powłoce Bash



Wprowadzenie – Czym jest powłoka?

- Jest to interfejs zapewniający dostęp do podstawowych operacji jakie możemy wykonać dzięki jądra systemu, np.:
 - Uruchomienie programu
 - Odczytanie zawartości katalogu
- Program pracujący jako interpreter poleceń wydawanych przy pomocy klawiatury w wierszu poleceń
- Z czasem powłoki zostały rozbudowane o możliwość łączenia ze sobą wyników przetwarzania wykonywanych programów

Wprowadzenie - Filozofia

- Podstawową filozofią systemów UNIX jest wykorzystanie ponownie tego samego kodu
- KISS – Keep It Small and Simple
 - Skomplikowane zadania rozbijamy na małe problemy. Gdy mamy gotowe rozwiązania „małych” problemów, możemy je łączyć ze sobą wykonując trudniejsze zadania
- Małe proste programy użytkowe służą jako ogniwo w łańcuchu tworzącym polecenie

\$ ls -al | more

Wprowadzenie – Pierwszy skrypt

- Istnieją dwie metody pisania skryptów powłoki
 - Tryb interaktywny
 - Zapisując te same polecenia do pliku by wykonać go, jak każdy inny program

```
for file in *  
do  
  echo $file  
done
```

Wprowadzenie – Pierwszy skrypt

- Tworzenie pliku z poleceniami
 - Skrypt jest interpretowany w momencie wykonywania, zatem większość błędów jesteśmy w stanie odkryć tylko poprzez uruchomienie skryptu

```
$ touch hello.sh
```

```
$ vim hello.sh
```

```
#!/bin/bash
```

```
echo "hello world!"
```

```
$ chmod 755 hello.sh
```

```
$ chmod +x hello.sh
```

```
$ ./hello.sh
```

Wprowadzenie – Polecenia

- Komendy zewnętrzne oraz wbudowane
 - Zewnętrzne, czyli takie którymi są pliki istniejące w systemie plików, zawarte w jednym z katalogów wskazywanych przez zmienną PATH
 - \$ echo \$PATH
 - Wbudowane, czyli istniejące tylko jako polecenia interpretowane poprawnie wewnątrz powłoki
 - \$ help

Wprowadzenie – Znaki specjalne

- Znaki specjalne

- # komentarz
- ; koniec polecenia
- : pusta instrukcja
 - zawsze zwraca wartość prawda (ang. *true*)
 - przydatna do uproszczenia logiki, np.:
: \${var:=value}
if [-f test.txt] ; then : ; else echo "Nie ma" ; fi
while : ; do echo pik ; done

Wprowadzenie – Znaki specjalne

- Znaki specjalne
 - . wykonuje polecenie w bieżącej powłoce
 - Gdy jest wykonywane zewnętrzne polecenie tworzone jest nowe środowisko
 - Zmiany zmiennych środowiskowych zostaną zachowane
 - Działa podobnie jak dyrektywa #include w C, C++

```
$ cat config.sh
#!/bin/bash
zmienna="wartość"
echo $zmienna
```

```
$ ./config.sh
$ echo $zmienna
$ . config.sh
$ echo $zmienna
```


Wprowadzenie – Podstawowe przekierowania

- Standardowe deskryptory plikowe
 - 0 – strumień danych wejściowych
 - 1 – strumień danych wyjściowych
 - 2 – strumień komunikatów o błędach
- Przykłady prostych przekierowań wyjścia

```
$ ls -al 1> ls.log
```

```
$ ls -al > ls.log
```

```
$ echo dopisuj >> ls.log
```

```
$ ping -c 1 www.google.com >>( cat )
```

Wprowadzenie – Podstawowe przekierowania

- Przykłady prostych przekierowań wyjścia

```
$ ping www.google.com > ping.log
```

```
$ ping www.nicsieniestalo.com > ping.log
```

```
$ ping www.nicsieniestalo.com 2> ping.log
```

```
$ ping www.nicsieniestalo.com > ping.log 2>&1
```

```
$ ping www.nicsieniestalo.com 2> /dev/null > ping.log
```

- Przekierowanie wejścia

```
$ more 0< ping.log
```

```
$ more < ping.log
```

```
$ more <( ping www.google.com )
```

Wprowadzenie – Operacje na deskryptorach

- Kopiowanie deskryptorów wejścia lub wyjścia

$n < &x$ lub $n > &x$

- Kopiuje deskryptor o numerze x , do deskryptora n
- $<$ lub $>$ w zależności czy deskryptor traktujemy jako wejście czy wyjście

- Ważna jest kolejność kopiowania

$\$ ls > ls.log 2 > &1$

$\$ ls 2 > &1 > ls.log$

$&>$ lub $>&$

- Równoważne z wykonaniem „ $> plik 2 > &1$ ”

Wprowadzenie – Operacje na deskryptorach

- Zamknięcie deskryptora o numerze n
 - $n > -1$ lub $n < 0$
- Przeniesienie deskryptora pod inny numer
 - $n < 0$ lub $n > 0$
 - Kopiuje deskryptor o numerze x , do deskryptora n , a następnie zamyka deskryptor x
- Otwarcie pliku do zapisu i odczytu
 - $n < 0$ plik
 - Gdy n nie jest podane, otwierane jest standardowe wejście czyli 0
 - Istniejący plik nie jest przycinany

Wprowadzenie – Potoki

- **Potok**

- łączy wyjście pierwszego procesu z wejściem drugiego procesu
- oba procesy działają równocześnie
- | operator potokowy

- **Przykłady**

`$ ls | grep wzorzec`

`$ du -csh | sort -h`

Wprowadzenie – Procesy w tle

- Procesy uruchamiamy w tle dodając na końcu wykonywanego polecenia znak &

`$ ls &`

`$ echo $!`

`$ while : ; do : ; done &`

- Zarządzanie wieloma procesami

`$ jobs`

`$ fg`

`$ kill %<job_id>`

Wprowadzenie

• Ćwiczenia

- 1) Stwórz skrypt *remote.sh*, który przekopiuje skrypt *remote_exec.sh* po *ssh* lub *scp* na zdalną maszynę, następnie wykona go i przekaże wynik wykonania skryptu *remote_exec.sh*
 - Jako maszynę zdalną użyj *localhost*
 - Skrypt *remote_exec.sh*, niech wykona polecenie *echo*
- 2) Pozbądź się komunikatu błędu z polecenia *ping test*, bez użycia przekierowania do pliku
- 3) Otwórz deskryptor nr 3 do zapisu w pliku *out3*, dla polecenia *echo test*, a następnie skopiuj deskryptor numer 3 do deskryptora numer 1 (stdout)

Programowanie w powłoce Bash

Zmienne i parametry



Zmienne i parametry

- Powłoka nie wymaga wcześniejszej deklaracji zmiennych
 - Tworzone są przy pierwszym użyciu
 - Zmienne niszczy poleceniem *unset*

```
$ zmienna=„wartosc zmiennej”  
$ echo $zmienna  
$ echo "$zmienna"  
$ echo '$zmienna'  
$ echo \ $zmienna  
$ read zmienna  
$ echo $zmienna  
$ echo ${zmienna}  
$ unset zmienna  
$ echo $zmienna
```

Zmienne i parametry

- Polecenie export

```
$ export zmienna  
$ ./exp.sh  
  
$ zmienna="test_exp"  
$ ./exp.sh  
  
$ export zmienna="test"  
$ ./exp.sh
```

- Zawartość pliku *exp.sh*

```
#!/bin/bash  
echo $zmienna
```

Zmienne i parametry

- Zmienne środowiskowe

- Niektóre zmienne są automatycznie ustawiane podczas uruchamiania powłoki. Wykonaj polecenie *echo* z poniższymi zmiennymi

\$HOME - katalog macierzysty użytkownika

\$PATH - lista katalogów w których wyszukiwane są polecenia

\$PS1 - znak zgłoszenia

\$PS2 - drugi znak zgłoszenia

\$IFS - lista separatorów pola wejściowego, używana przez *read*

\$0 - nazwa skryptu powłoki

\$# - liczba przekazanych parametrów

\$\$ - PID bieżącego procesu powłoki

Zmienne i parametry

- Zmienne parametryczne

- Są tworzone jeżeli do wykonywanego skryptu (lub funkcji wewnątrz skryptu) zostały przekazane parametry
- Można jedynie odczytać ich zawartość
- \$1, \$2, \$3 ... - parametry przekazane do skryptu
- \$* - lista wszystkich parametrów. W momencie ujęcia w podwójny cudzysłów, parametry są rozdzielone pierwszym znakiem zmiennej środowiskowej **IFS**.
- \$@ - lista wszystkich parametrów. "\$@" jest równoznaczne z zapisem "\$1" "\$2" ...

Zmienne i parametry

- Przetestuj poniższy skrypt

```
$ cat params.sh
#!/bin/bash

echo Uzycie '$@'
for p in $@ ; do
    echo "$p"
done
echo

echo Uzycie '$@'
for p in "$@" ; do
    echo "$p"
done
echo
```

Zmienne i parametry

- Przetestuj polecenie read

```
$ cat ifs.sh  
#!/bin/bash
```

```
IFS=","
```

```
read a b c  
echo $a $b $c
```

```
$ echo "ala,beata,ola" | ./ifs.sh
```

- Przetestuj zmienną \$*

```
$ cat ifs2.sh  
#!/bin/bash
```

```
IFS="|"
```

```
echo Uzycie '$*'  
for p in $* ; do  
    echo "$p"  
done  
echo
```

```
echo Uzycie "$*"  
for p in "$*" ; do  
    echo "$p"  
done  
echo
```

```
$ ./ifs2.sh ala "ma kota"
```

Zmienne i parametry

- Zmienna o wartości *NULL*

- Są to zmienne o zerowej długości, czyli zawierające ciąg pusty

```
$ cat null.sh
#!/bin/bash

if [ -z "$1" ]
then
    echo "Wartosc = null"
else
    echo "Wartosc != null"
fi
```

Zmienne i parametry

- Instrukcja *shift*

- Przesuwa wszystkie zmienne parametryczne o podaną ilość pozycji w dół. Zmienne \$# , \$* i \$@ również ulegają zmianie
- Domyślnie przesuwana o jedną pozycję

```
$ cat shift.sh
#!/bin/bash
echo "$#: $1"
shift
echo "$#: $1"

$ ./shift.sh ala ma kota
```

```
$ cat shift2.sh
#!/bin/bash
while [ "$1" != "" ] ; do
    echo "$1"
    shift
done

$ ./shift2.sh ala ma kota
```


Zmienne i parametry

- Instrukcja *set*

- Ustawia zmienne parametryczne dla bieżącej powłoki

```
$ cat set.sh
#!/bin/bash
set $(date +%A)
echo Mamy dzisiaj $1

$ ./set.sh
```

Zmienne i parametry

- Zapisanie wyniku polecenia do zmiennej

```
$ cat set_var.sh
#!/bin/bash
day=$(date +%A)
month=`date +%B`
echo Dzień $day
echo Miesiąc $month

$ ./set_var.sh
```

Zmienne i parametry

- Rozwinięcie parametryczne

- Najprostsza forma rozwinięcia parametrycznego

```
$ zmienna=cos; echo $zmienna
```

- Przeanalizujmy przykłady

```
$ i=1; echo $iteracja; echo ${i}teracja
```

```
$ for i in 1 2 ; do echo $iteracja ; done
```

```
$ for i in 1 2 ; do echo ${i}teracja ; done
```

Zmienne i parametry

- Rozwinięcia parametryczne

- `${param:pozycja:ilosc}`

- `$ zm="ala ma kota"`

- `$ echo ${zm:4:2}`

- `${param/wzór/zamiennik}`

- `$ zm="ala ma kota, kot ma ..."`

- `$ echo ${zm/ma/nie ma}`

- `$ echo ${zm//ma/nie ma}`

Zmienne i parametry

- Rozwinięcia parametryczne (2)

`${param:-domyslna}`

- Jeżeli "param" jest null, zwraca wartość "domyslna"

`${#param}`

- Zwraca długość parametru w znakach

`${param%slowo}`

- Usuwa od **końca najmniejszą** część pasującą do słowa

`${param%%slowo}`

- Usuwa od **końca największą** część pasującą do słowa

`${param#slowo}`

- Usuwa od **początku najmniejszą** część pasującą do słowa

`${param##slowo}`

- Usuwa od **początku największą** część pasującą do słowa

Zmienne i parametry

- Rozwinięcia parametryczne, przykłady

```
$ zmienna=
```

```
$ echo ${zmienna:-domysl}
```

```
$ zmienna=wartosc
```

```
$ echo ${zmienna:-domysl}
```

```
$ echo ${#zmienna}
```

Zmienne i parametry

- Rozwinięcia parametryczne, przykłady (2)

- * oznacza jeden lub więcej znaków

```
$ zmienna=/dzo/bar/dlu/bar/ga
```

```
$ echo ${zmienna%bar*}
```

```
$ echo ${zmienna%%bar*}
```

```
$ echo ${zmienna#*bar}
```

```
$ echo ${zmienna##*bar}
```

Zmienne i parametry

- Ćwiczenie

- 1) Przypisz wartość „`/usr/local/bin/test.sh`” do zmiennej *file*, a następnie użyj rozwinięcia parametrycznego by:
 - a) Wyłuskać nazwę pliku ze zmiennej *file*
 - b) Wyłuskać ścieżkę katalogów, bez nazwy pliku ze zmiennej *file*

Programowanie w powłocie Bash

Tablice



Tablice

- Bash obsługuje tablice indeksowane oraz asocjacyjne

- Nie ma limitu co do rozmiaru tablic. Indeksowanie rozpoczyna się od zera

- Definicje tablicy indeksowanej i asocjacyjnej

```
$ tablica=(pierwszy drugi trzeci)
```

```
$ declare -A tablica_a=([pierwszy]=a [drugi]=b [trzeci]=c)
```

- Ilość elementów w tablicy

```
$ echo ${#tablica[*]}
```

```
$ echo ${#tablica[@]}
```

Tablice

- Pobranie dowolnego elementu tablicy

```
$ echo ${tablica[0]}
```

```
$ echo ${tablica_a[drugi]}
```

- Pobranie wszystkich elementów tablicy

```
$ echo ${tablica[@]} ${tablica[*]}
```

- Lista kluczy tablicy asocjacyjnej

```
$ echo ${!tablica_a[*]} ${!tablica_a[@]}
```

- Długość elementu trzeciego

```
$ echo ${#tablica[3]}
```

Tablice

- **Nadpisanie lub dodanie elementów**

```
$ tablica[3]=czwarty
```

```
$ tablica_a[drugi]=3
```

- **Usunięcie wartości elementu drugiego. Liczba elementów się zmniejszy, ale indeksy poszczególnych elementów się nie zmienią!**

```
$ unset tablica[1]
```

- **Usunięcie całej tablicy**

```
$ unset tablica
```

```
$ unset tablica[*]
```

Tablice

- Wyświetlenie wszystkich elementów tablicy

```
$ cat array.sh
#!/bin/bash

tablica=(pierwszy drugi trzeci)
tablen=${#tablica[*]}

i=0
for key in ${!tablica[@]} ; do
    echo Element $i : ${tablica[$key]}
    let i++
done

$ ./array.sh
```

- Czemu nie wyświetlamy wartości po indeksie \$i ?

Tablice

- Ćwiczenie

1) Na podstawie przykładu z poprzedniej strony utwórz skrypt *array_asoc.sh* który :

- a) Stworzy tablicę asocjacyjną z minimum trzema elementami
- b) Wyświetli klucze i wartości utworzonej tablicy
- c) Usunie „środkowy” element tablicy, i jeszcze raz wyświetli zawartość tablicy, podobnie jak w poprzednim podpunkcie

Programowanie w powłoce Bash

Instrukcje warunkowe



Instrukcje warunkowe

- Instrukcja *if*

```
if warunek
then
    Instrukcje
else
    Instrukcje
fi
```

```
if true
then
    echo prawda
fi

if ! false
then
    :
else
    echo nie prawda
fi
```


Instrukcje warunkowe

- Instrukcja *if* wraz z *elif*

```
if warunek
then
    Instrukcje
elif warunek
then
    Instrukcje
else
    Instrukcje
fi
```

```
echo tak/nie?
read odp

if [ $odp = "tak" ]
then
    echo Tak tak!
elif [ $odp = "nie" ]
then
    echo Nie nie!
fi
```

Instrukcje warunkowe

- Zagnieżdżanie Instrukcji *if*

```
echo tak/nie?  
read odp  
  
if [ $odp = "tak" ]  
then  
  if true  
  then  
    echo "tak i true"  
  fi  
fi
```

Instrukcje warunkowe

- Instrukcja *case*

- Wzorzec obsługuje takie same symbole wieloznaczne, jak polecenie *ls* przy wyszukiwaniu plików

```
case zmienna in
wzorzec [| wzorzec] )
instrukcje ;;
wzorzec [| wzorzec] )
instrukcje ;;
...
esac
```

```
echo Pada deszcz?
read odp

case "$odp" in
tak)
    echo ":/" ;;
t)
    echo ":/" ;;
nie|n)
    echo ":)" ;;
*)
    echo Nie rozumiem ;;
esac
```

Instrukcje warunkowe

- Instrukcja *case* (2)

```
echo Pada deszcz?  
read odp
```

```
case "$odp" in  
  "tak" | "t" | "TAK")  
    echo ":/" ;;  
  n* | N*)  
    echo ":)" ;;  
  *)  
    echo Nie rozumiem ;;  
esac
```

```
echo Pada deszcz?  
read odp
```

```
case "$odp" in  
  [tT] | [tT][aA][kK])  
    echo ":/" ;;  
  [nN]*)  
    echo ":)" ;;  
  *)  
    echo Nie rozumiem  
    ;;  
esac
```

Instrukcje warunkowe

- Instrukcja *select*
 - Służy do tworzenia prostych menu

```
select zmienna in lista
do
  polecenie
done
```

```
echo Pada deszcz?
select odp in Tak Nie
do
  echo $odp
done
```

```
PS3="Wybierz opcje:"
echo Który dzień?
opcje="Pon Wt Sr Cz Pia"
```

```
select odp in $opcje
do
  echo $odp
done
```

Instrukcje warunkowe

- Polecenie *test*

- Służy do testowania różnorodnych warunków
- Umieszczenie [] sprawia, że składnia jest bardziej przejrzysta
\$ if test -f plik.sh ; then

Jest równoważne z :

\$ if [-f plik.sh] ; then

Instrukcje warunkowe

- Polecenie *test* i sprawdzanie rodzaju i atrybutów plików

- f plik – prawda, gdy plik jest zwykłym plikiem
- d plik – prawda, gdy plik jest katalogiem
- e plik – prawda, gdy plik istnieje
- r plik – prawda, gdy można plik odczytać
- w plik – prawda, gdy można do pliku zapisać
- x plik – prawda, jeżeli jest wykonywalny
- s plik – prawda, gdy plik istnieje i rozmiar większy od zera
- u plik – prawda, gdy plik ma atrybut SUID
- g plik – prawda, gdy plik ma atrybut SGID

Instrukcje warunkowe

- Polecenie *test* i sprawdzanie ciągów tekstowych

`string`

prawda, jeżeli ciąg nie jest pusty

`string1 = string2`

prawda, jeżeli ciągi są jednakowe

`string1 != string2`

prawda, jeżeli ciągi nie są jednakowe

`-n string`

prawda, jeżeli ciąg nie jest pusty (null)

`-z string`

prawda, jeżeli ciąg jest pusty (null)

Instrukcje warunkowe

- Polecenie *test* i operacje arytmetyczne

wyrU -eq wyrT

prawda, gdy wyrażenia są równe

wyrU -ne wyrT

prawda, gdy wyrażenia nie są równe

wyrU -gt wyrT

prawda, gdy $\text{wyrU} > \text{wyrT}$

wyrU -ge wyrT

prawda, gdy $\text{wyrU} \geq \text{wyrT}$

wyrU -lt wyrT

prawda, gdy $\text{wyrU} < \text{wyrT}$

wyrU -le wyrT

prawda, gdy $\text{wyrU} \leq \text{wyrT}$

! wyrażenie

negacja wyrażenia

Instrukcje warunkowe

- Komenda `[[wyrażenie]]`

- Nowsza wersja polecenia `test` lub `[wyrażenie]`, ale niezgodna z POSIX
- Podobnie jak `test` zwraca 0 lub 1 w zależności od wyniku wyrażenia
- Ścieżki do plików nie są rozwijane, a rozwijane łańcuchy nie są dzielone na słowa (zmienne nie muszą być opakowywane w cudzysłów)

```
$ file="./nazwa ze spacja"
```

```
$ [[ -f $file ]]
```

```
$ [ -f $file ]
```

```
-bash: [: too many arguments
```

- Porównywanie łańcuchów znakowych, obsługuje wzorce oraz wyrażenia regularne

Instrukcje warunkowe

- Komenda `[[wyrażenie]]`
 - Automatycznie rozwijane są stałe zapisane w innych systemach liczbowych, podczas działań arytmetycznych

```
dec=15
hex=0x0f

if [[ "$dec" -eq "$hex" ]]
then
    echo Rowne
else
    echo Nie rowne
fi
```

Instrukcje warunkowe

- Ćwiczenie

- 1) Napisz skrypt który sprawdzi czy podane w parametrach dwa ciągi znaków nie są puste, oraz czy są takie same, wyświetlając przy tym stosowne komunikaty

Programowanie w powłoce Bash

Struktury sterujące



Struktury sterujące

- Pętla *for*

- Używamy do tworzenia pętli przechodzących przez pewien zdefiniowany zakres wartości
- Postać klasyczna dla ciągów

```
for zmienna in lista ciągów  
do  
    instrukcje  
done
```

- Postać z rozwinięciem wyrażeń arytmetycznych

```
for (( expr1 ; expr2 ; expr3 ))  
do  
    instrukcje  
done
```

Struktury sterujące

● Przykład 1

```
for zm in 1 2 ala beata  
do  
echo $zm  
done
```

● Przykład 3

```
for zm in {0..5}  
do  
echo $zm  
done
```

● Przykład 2

```
for zm in {a..z}  
do  
echo $zm  
done
```

● Przykład 4

```
for zm in $( seq 1 5 20 )  
do  
echo $zm  
done
```

Struktury sterujące

- Przykład „arytmetycznej” pętli *for*

```
for (( i=0; i < 10; i++ ))  
do  
    echo $i  
done
```

- Zamiast polecenia *seq*

```
for (( i=1; i < 20; i+=5 ))  
do  
    echo $i  
done
```


Struktury sterujące

- **Pętla *while***

- Gdy nie wiemy ile razy pętla będzie musiała się wykonać
- Wykonuj dopóki warunek jest spełniony

```
while warunek  
do  
  instrukcje  
done
```

- **Przykłady**

```
counter=1  
while [ "$counter" -lt 5 ]  
do  
  echo $counter  
  let counter++  
done  
echo $counter
```

```
ls -l | while read f  
do  
  echo Plik $f  
done
```

Struktury sterujące

- *Pętla until*

- Pętla jest wykonywana dopóki warunek **nie** jest spełniony

```
until warunek  
do  
  instrukcje  
done
```

- Przykłady

```
counter=1  
until [ "$counter" -gt 5 ]  
do  
  echo $counter  
  let counter++  
done  
echo $counter
```

```
ls -1 | until ! read f  
do  
  echo Plik $f  
done
```

Struktury sterujące

- Polecenia *break* i *continue*

- *break* – natychmiastowe wyjście z pętli
- *continue* – przerwanie bieżącej iteracji i kontynuacja od następnej

- Możemy je stosować we wszystkich rodzajach pętli

- for
- while
- until

Struktury sterujące

- Polecenia *break* i *continue*

- Przykłady

```
for zm in $( seq 5 )  
do  
  echo $zm  
  continue  
  echo Tego nie zobaczysz  
done
```

```
counter=1  
while [ "$counter" -lt 5 ]  
do  
  echo $counter  
  let counter++  
  break  
done
```

Struktury sterujące

- Listy

- Używamy gdy chcemy połączyć polecenia w serie lub gdy musimy wcześniej sprawdzić kilka warunków
- Możemy również tworzyć zagnieżdżone instrukcje if lecz nie jest to elegancki i efektywny sposób

```
if [ -f plik1 ] ; then
```

```
    if [ -f plik2 ] ; then
```

Struktury sterujące

- Lista AND

- Umożliwia wykonywanie serii poleceń
- Następne polecenie jest wykonywane tylko wtedy, gdy poprzednie zakończy się sukcesem
- Polecenia są wykonywane od strony lewej do prawej
- Cała lista zwróci wartość „prawda” gdy wszystkie polecenia zakończą się sukcesem

```
if [ -f plik1 ] && [ -f plik2 ] ; then
```

```
if [[ -f plik1 && -f plik2 ]]; then
```

Struktury sterujące

- Lista OR

- Następne polecenie jest wykonywane tylko wtedy, gdy poprzednie zakończy się porażką
- Polecenia są wykonywane od strony lewej do prawej, do czasu aż jedno z nich zakończy się sukcesem
- Cała lista zwróci „prawda” gdy przynajmniej jedno z poleceń zakończy się sukcesem

```
if [ -f plik1 ] || [ -f plik2 ] ; then
```

```
if [[ -f plik1 || -f plik2 ]]; then
```

Struktury sterujące

- Listy AND i OR

```
if [ -f plik ] || echo "Hej" || echo " ho" ; then  
    echo OK  
fi
```

```
if [ -f plik ] && echo "Hej" || echo " ho"; then  
    echo OK  
fi
```

```
if [ -f plik ] && ( echo "Hej" || echo " ho" ) ; then  
    echo OK  
fi
```


Struktury sterujące

- Bloki instrukcji

- Stosujemy gdy chcemy użyć wielu instrukcji w miejscu, gdzie normalnie dozwolona jest tylko jedna
- { ... } - wbudowane polecenia zostaną wykonane w bieżącej powłocie
- (...) - polecenia zostaną wykonane w nowej powłocie

```
if [ -f plik ] && { grep -Hn "a" plik; echo druga; }  
then  
    echo Znalazlem \"a\"?  
fi
```

Struktury sterujące

- Funkcje

- Nie musimy dzielić dużego skryptu na mniejsze pliki
- Funkcje wykonują się dużo szybciej
- Łatwiej jest przekazywać rezultaty
- Sprawiają, że mamy czytelniejszy kod
- Trzeba je zawsze najpierw zdefiniować

```
func() {  
    echo "Wykonano sie"  
}  
echo Startujemy  
func  
echo Konczymy
```

Struktury sterujące

```
func() {  
    echo "Chcesz kontynuowac?"  
    select odp in Tak Nie  
    do  
        case $odp in  
            Tak) return 0 ;;  
            Nie) return 1 ;;  
        esac  
    done  
}  
  
while func  
do  
    echo Jeszcze raz  
done
```

Struktury sterujące

- Wywołanie funkcji z parametrami

```
func() {  
    echo Parametry $#: $*  
}  
  
echo Parametry $#: $*  
  
czwarty=trzeci  
func pierwszy drugi $czwarty
```

Struktury sterujące

- Zmienne lokalne w funkcji

- Deklarujemy za pomocą słowa kluczowego local
- Jeżeli zmienna lokalna ma taką samą nazwę jak globalna to przestania ją w obrębie funkcji

```
lokalna="wartosc globalna"  
func() {  
    local lokalna="mam lokalna wartosc"  
    echo $lokalna  
}  
  
func  
echo $lokalna
```

Struktury sterujące

- Przekierowanie wejścia i wyjścia funkcji
 - Sposób wykonywania funkcji nie różni się specjalnie od wykonywania innych poleceń

```
func() {  
  read wej  
  echo Odczytano: $wej  
}
```

```
echo wejscie | func
```

```
func() {  
  echo Wyjscie  
}
```

```
func  
func > /dev/null
```

Struktury sterujące

- **Dokumenty miejscowe**

- Sposób na przekazanie odpowiedniego wejścia bezpośrednio ze skryptu powłoki
- Zaczynają się od znaku << po którym następuje unikalna sekwencja, która musi być powtórzona na końcu dokumentu miejscowego
- Znaki tworzące unikalną sekwencję nie mogą się pojawić w treści dokumentu miejscowego
- W liniach działa rozwijanie zmiennych itd..

```
cat << _EOF_  
Dokument miejscowy  
_EOF_
```

Struktury sterujące

- Ćwiczenie

1) Napisz skrypt *test.sh*, który :

- a) Po podaniu parametru „-f plik” sprawdzi czy plik istnieje i wypisze stosowny komunikat
- b) Po podaniu parametru „-d plik” sprawdzi czy plik jest katalogiem oraz wypisze stosowny komunikat
- c) Parametr „-h” spowoduje wyświetlenie krótkiego tekstu pomocy, podobnie w przypadku podania błędnych parametrów, lub nie podania ich w ogóle
- d) Dodaj długą formę powyższych parametrów, czyli --file, --dir, --help

Kolejność podawania parametrów powinna być dowolna

Programowanie w powłoce Bash

Polecenia wbudowane



Polecenia wbudowane

- Sekwencje specjalne

\\ - znak odwróconego ukośnika

\\" - znak cudzysłowu

\a - alarm (dzwonek lub mignięcie ekranu)

\b - znak cofnięcia

\f - znak przesuwu strony/linii

\n - znak nowej linii

\r - powrót karetki

\t - znak tabulatora

\e - znak ucieczki

\001 – pojedynczy znak o ósemkowej wartości 001

\x0a – pojedynczy znak o szesnastkowej wartości x0a

Polecenia wbudowane

- echo

- Polecenie przestarzałe, ale mimo to nadal chętnie używane i obecne w prawie każdym skrypcie

`$ echo -n tekst`

- Pomija znak nowej linii

`$ echo -e "pierwsza \t druga \t kolumna"`

- Zapewnia interpretacje znaków specjalnych

Polecenia wbudowane

- **printf**

- Powinno być używane zamiast echo
- `printf "format" param1 param2 param3`
 - `%d` liczba dziesiętna
 - `%c` pojedynczy znak
 - `%s` ciąg znaków
 - `%b` ciąg znaków z interpretacją sekwencji specjalnych
 - `%%` znak %

Polecenia wbudowane

- printf, przykłady

```
$ printf "liczba %d ciag \"%s\"" 110 "dodatkowy napis"
```

```
$ printf "jakis: %s" "\tparam\n"
```

```
$ printf "jakis: %b" "\tparam\n"
```

Polecenia wbudowane

- printf, przykłady

```
$ printf "liczba %d ciag \"%s\"" 110 "dodatkowy napis"
```

```
$ printf "jakis: %s" "\tparam\n"
```

```
$ printf "jakis: %b" "\tparam\n"
```

Struktury sterujące

- **exec**

- Zastępuje bieżącą powłokę innym poleceniem

`$ exec ls -al`

- Gdy nie podamy polecenia, modyfikuje deskryptory plikowe aktualne powłoki

```
#!/bin/bash
exec 1> log
echo przekierowanie wyjscia calego skryptu do pliku
```

```
#!/bin/bash
exec 3>&1 # skopiuj deskryptor nr 1 do deskryptora nr 3
exec 3>&- # zamknij deskryptor nr 3
```

Polecenia wbudowane

- **exit n**

- Kończy skrypt z podanym kodem wyjścia
- Wykonywana jest również „pułapka” *EXIT*

0 sukces

1 – 125 kody błędów

126 plik nie był wykonywalny

127 nie znaleziono polecenia

128 i wyższe, pojawił się sygnał

Polecenia wbudowane

• Przykłady

```
$ cat exit.sh  
#!/bin/bash  
exit 0
```

```
$ ./exit.sh && echo true  
$ ( exit 0 ) && echo true
```

```
$ cat exit2.sh  
#!/bin/bash  
false  
exit $?
```

```
$ ./exit2.sh && echo true  
$ ./exit2.sh || echo false
```

Polecenia wbudowane

- **eval**

- Wykonuje „dodatkowe” rozwinięcie parametrów
- Umożliwia pisanie dynamicznego kodu

```
x=foo  
y=\$$x  
echo $y
```

```
foo=10  
x=foo  
eval y=\$$x  
echo $y
```

Polecenia wbudowane

- **sleep**

- Włącza pauze przez określoną liczbę sekund

- `sleep liczba[s|m|h|d]`

- s – sekundy
- m – minuty
- h – godziny
- d – dni

```
time sleep 1m
```

Polecenia wbudowane

- **pushd, popd, dirs**

- pushd – odkłada bieżący katalog na stos i przechodzi do nowego katalogu

\$ pushd /home

- popd – zdejmuje katalog ze stosu i do niego wchodzi

\$ popd

- **dirs**

- lista katalogów odłożonych na stosie

```
$ pushd /tmp  
$ dirs  
$ popd  
$ pwd
```

Polecenia wbudowane

- **trap**

- Jest stosowane do określenia akcji wykonywanej po otrzymaniu sygnału. Lista sygnałów:

- \$ trap -l

- trap polecenie sygnał

- \$ trap - sygnał

- Przywraca ustawienia domyślne

- \$ trap "" sygnał

- Ignoruje sygnał

```
$ cat trap.sh
#!/bin/bash
trap 'echo Nie dasz rady!' INT
trap 'echo koniec' EXIT
while : ; do sleep 1 ; done
```

Programowanie w powłoce Bash

Debugowanie skryptów



Debugowanie skryptów

- **Typowe błędy w skryptach powłoki i nie tylko**
 - Zawierający niepoprawną składnię, bash zwróci „syntax error”
 - Uruchamia się bezbłędnie ale nie działa tak jak się spodziewamy
 - Działa tak jak się spodziewamy, ale powoduje nie przewidziane szkodliwe efekty

Debugowanie skryptów

- **Numer linii nie zawsze odpowiada dokładnie miejscu wystąpienia błędu**

```
$ cat debug_for.sh
#!/bin/bash

for i in {1..5} ; do
    echo "$i"
# done
exit 0
```


Debugowanie skryptów

- **Debugowanie przy pomocy printf lub echo**

```
$ cat debug_printf.sh
#!/bin/bash

decho() {
  if [ -n "$DEBUG" ] ; then
    printf "+ %s\n" >&2

    printf "+ Trace:\n" >&2
    for key in ${!FUNCNAME[@]} ; do
      printf "+ ${BASH_SOURCE[$key]}:${FUNCNAME[$key]}:\n" >&2
    done
  fi
}
DEBUG=on
decho $LINENO: ala ma kota
DEBUG=
decho $LINENO: ala ma dwa koty
```

Debugowanie skryptów

- **Polecenie set i opcje ułatwiające odpluskwanie**

`$ set -n`

Szuka tylko błędów składni, nie wykonuje poleceń

`$ set -v`

Powtarza polecenia przed ich wykonaniem

`$ set -x`

Powtarza polecenia po przetworzeniu (rozwinieciu zmiennych etc.)

`$ set -u`

Podaje komunikat błędu gdy wykorzystywana jest niezdefiniowana zmienna

`$ set -e`

Zakończ skrypt natychmiast gdy polecenie zwróci nie zerowy kod wyjścia, o ile nie zostało użyte w instrukcji warunkowej

Debugowanie skryptów

- **trap**

- EXIT pułapka wykonywana w momencie kończenia pracy skryptu

trap 'echo krytyczna wartosc = \$zmienna' EXIT

- RETURN wykonywane gdy funkcja lub załączony skrypt (np. „*config.sh*”) zakończy działanie

\$ set -o functrace

powoduje dziedziczenie pułapek *RETURN* oraz *DEBUG* przez funkcje i polecenia wykonywane w podpowłokach

Debugowanie skryptów

- **trap**

- DEBUG pułapka wykonywana przed wykonaniem kolejnej instrukcji

shopt -s extdebug

trap 'echo \$LINENO \${FUNCNAME[0]}' DEBUG

- ERR wykonywana gdy kod wyjścia polecenia jest różny od zera (zakończyło się błędem). Pułapka nie będzie wykonana, gdy polecenie zostało użyte w instrukcji warunkowej, np. *if*

\$ set -o errtrace

włącza dziedziczenie pułapki *ERR* przez funkcje i podpowłoki

Debugowanie skryptów

- **tee**

- Czyta wejście i przesyła na wyjście, zapisując przy okazji dane do pliku

```
$ ls | sort | tee log.txt
```

- Logowanie strumieni *stdout* i *stderr*

```
$ cat debug_tee.sh
#!/bin/bash

LOGFILE=/tmp/${0%.*}.log

exec > >( tee -a "${LOGFILE}" )
exec 2> >( tee -a "${LOGFILE}" >&2 )

echo stdout
echo stderr >/dev/stderr
```

Debugowanie skryptów

- **Ćwiczenia**

1) Stwórz funkcję `assert`, którą można wykonać w taki sposób :

```
assert "warunek" $LINENO
```

2) Stwórz skrypt *debug.sh*, zawierający funkcję *assert*, który dodatkowo

a) Wykorzysta polecenie „*set -x*” by śledzić wykonywanie skryptu

b) Przekieruje przy pomocy zmiennej *BASH_XTRACEFD*, komunikaty śledzenia do pliku

c) Wykorzysta pułapkę `DEBUG`

Programowanie w powłoce Bash

Rozwinięcie arytmetyczne



Rozwinięcie arytmetyczne

- **Rozwinięcie arytmetyczne**

- czyli wykonanie działań arytmetycznych jak dodawanie czy odejmowanie
- Obsługiwane są jedynie liczby całkowite
- Dla nie istniejącej lub pustej zmiennej, przyjmowana jest wartość 0

- **((...)) lub let**

- Rozwinięcie arytmetyczne w powłoce bash
- Różnica polega tylko w sposobie zapisu ((...)) vs *let "..."*

- **expr**

- Polecenie zewnętrzne służące między innymi do wykonywania obliczeń

Rozwinięcie arytmetyczne

- **Równoznaczne przykłady**

```
i=0  
(( i++ ))  
echo $i
```

```
i=0  
echo $(( ++i )) # echo poda obliczoną wartość całego wyrażenia
```

```
i=0  
let i++  
echo $i
```

```
i=0  
i=$(( expr $i + 1 ))  
echo $i
```

Rozwinięcie arytmetyczne

- **Operacje w kolejności od największego priorytetu**

id++ id--

variable post-increment and post-decrement

++id --id

variable pre-increment and pre-decrement

- + unary minus and plus

! ~ logical and bitwise negation

** exponentiation

* / % multiplication, division, remainder

+ - addition, subtraction

<< >> left and right bitwise shifts

<= >= < >

comparison

== != equality and inequality

& bitwise AND

^ bitwise exclusive OR

| bitwise OR

&& logical AND

|| logical OR

Rozwinięcie arytmetyczne

- Operacje w kolejności od największego priorytetu (2)

|| logical OR

expr?expr:expr

conditional operator

= *= /= %= += -= <<= >>= &= ^= |=

assignment

expr1 , expr2

comma

Rozwinięcie arytmetyczne

- **Przykłady**

```
$ x=$(( 3 * 2 + 2 ))
```

```
$ x=$(( 3 * (2 + 2) ))
```

```
$ echo $((1 + 1.1))
```

```
$ (( x = 5 ))
```

```
$ echo $x
```

```
$ if (( x == 5 )) ; then echo Rowne ; fi
```

```
$ if [ $(( x == 5 )) -eq 1 ]; then echo Rowne ; fi
```

```
$ (( x = x < 100 ? 0 : 100 ))
```

```
$ echo $x
```

```
$ (( x += 5 ))
```

Rozwinięcie arytmetyczne

- **Ćwiczenie**

1) Stwórz skrypt *bobert.sh*, który wykona operacje podane jako parametry. Przykład działania :

```
$ ./bobert.sh 2 + 2  
4
```

2) Zabezpiecz skrypt przed niepoprawną operacją, np dzieleniem przez zero

Programowanie w powłoce Bash

Liczby zmiennoprzecinkowe



Liczby zmiennoprzecinkowe

- Wbudowana arytmetyka Bash-a obsługuje tylko liczby całkowite
- Istnieje jednak możliwość wykonywania operacji na liczbach zmiennoprzecinkowych
- Najprościej wykorzystać do tego celu program bc, czyli język stworzony do obliczeń matematycznych o dowolnej dokładności

Liczby zmiennoprzecinkowe

- **Wykorzystanie bc w bash**

- Zmienna „scale” określa liczbę miejsc po przecinku, do jakiej mają być wykonywane obliczenia

- **Wykonywanie operacji arytmetycznych**

```
$ echo $( echo "scale=3; 2.0 * 3.0" | bc -q )
```

- **Wykonanie porównania arytmetycznego**

```
if [ $( echo "4.0 == 4.0" | bc -q ) -eq 1 ]; then  
    echo Rowne  
fi
```


Liczby zmiennoprzecinkowe

- **Ćwiczenie**

1) Stwórz skrypt *fexpr.sh* i opakuj narzędzie `bc` w dwie proste funkcje (`fexpr` i `ftest`) ułatwiające wykonywanie obliczeń zmiennoprzecinkowych

Programowanie w powłoce Bash

Rozwijanie ścieżek i proste wzorce



Rozwijanie ścieżek i proste wzorce

- Po podziale linii z poleceniem na argumenty, bash szuka wśród argumentów, znaków *, ? lub [. Gdy je odnajdzie, uznaje całe słowo za wzorzec
- Za wzorzec podstawiane są nazwy plików które do niego pasują, sortowane w kolejności alfabetycznej
 - \$ shopt -s nocaseglob
 - Wyłącza rozróżnianie małych i dużych liter w wzorcach nazw plików
- Każdy znak wzorca który nie jest znakiem specjalnym, oznacza siebie samego. Znaki specjalne możemy pozbawiać ich specjalnego znaczenia poprzedzając je \
 - Podobny efekt da ujęcie w pojedyncze cudzysłowy

Rozwijanie ścieżek i proste wzorce

- **Znaki specjalne**

- * dopasowuje jakikolwiek ciąg znaków, w tym pusty

- */ dopasowuje tylko katalogi

- ? jakikolwiek pojedynczy znak

- [...] oznacza którykolwiek ze znaków ujętych w nawias. Rozpoczęcie przy pomocy [^ lub [! oznacza wykluczenie zawartych znaków

- [A-Z] myślnik, oznacza cały przedział znaków, w tym wypadku od A do Z. Myślnik możemy pozbawić specjalnego znaczenia, gdy pojawi się pierwszy albo ostatni, np. [-a] lub [a-]

Rozwijanie ścieżek i proste wzorce

- **[:class:]**
 - Standard POSIX dopuszcza również notację w której możemy zdefiniować całą klasę znaków. Słowo *class* pomiędzy dwukropkami możemy zastąpić jedną z rzeczywistych klas znaków :
 - alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit

Rozwijanie ścieżek i proste wzorce

- **Ćwiczenia**

- Przejdź do katalogu */etc* a następnie wyszukaj:
 - 1) wszystkie katalogi
 - 2) pliki których nazwa zaczyna się od „*cr*”
 - 3) Wyszukaj katalogi mające w nazwie literki „*a*” lub „*b*”
 - 4) Wyszukaj katalogi mające w nazwie literki „*a*” i „*b*”

Rozwijanie ścieżek i proste wzorce

- **Listy wzorców**

- Lista daje nam możliwość zdefiniowania kilku różnych wzorów które mogą dopasować nazwy plików
- Wzorce rozdzielamy znakiem |, co oznacza logiczne *lub*
 - ?(lista) zero lub jedno dopasowanie wzorca
 - *(lista) zero lub więcej dopasowań wzorca
 - +(lista) jedno lub więcej dopasowań wzorca
 - @(lista) dokładnie jedno dopasowanie
 - !(lista) wszystko co nie pasuje do wzorca. Działa pod warunkiem, że włączona jest opcja *extglob*. Sprawdzamy:
\$ shopt extglob

Rozwijanie ścieżek i proste wzorce

- **Listy wzorców, przykłady**

```
$ touch ab22.jpg ab23.jpg ab2.jpg ab3.jpg ab.jpg
$ ls ab?(2).jpg
ab2.jpg ab.jpg
$ ls ab*(2).jpg
ab22.jpg ab2.jpg ab.jpg
$ ls ab+(2).jpg
ab22.jpg ab2.jpg
$ ls ab@(2).jpg
ab2.jpg
$ ls ab*(2|3).jpg
ab22.jpg ab23.jpg ab2.jpg ab3.jpg ab.jpg
```


Rozwijanie ścieżek i proste wzorce

- **Ćwiczenie**

- 1)Przejdź do katalogu `/etc` oraz wyświetl pliki lub katalogi nie zawierające w nazwie litery „a”
- 2)Utwórz pliki `test1.bmp`, `test2.png`, `test3.gif`, `test4.sh`, a następnie wyświetl wszystkie pliki oprócz tych z końcówką `.bmp` lub `.sh`

Programowanie w powłoce Bash

Przetwarzanie tekstu



Przetwarzanie tekstu

- **expand**

- Konwertuje tabulatory na spacje

- **Przykład**

```
$ echo -e "ala\tbeata\tola" | sed "s/ /S/g"
```

```
$ echo -e "ala\tbeata\tola" | expand - | sed "s/ /S/g"
```

Przetwarzanie tekstu

- **unexpand**

- Konwertuje spacje na tabulatory

- **Przykład**

```
$ echo "ala beata ola" | sed "s/ /S/g"
```

```
$ echo "ala beata ola" | unexpand -t 1 - | ↵  
sed "s/ /S/g"
```

Przetwarzanie tekstu

- **head i tail**

- Pokazują początek (head) i koniec (tail) pliku

`head -n <liczba> [plik]`

- Liczba określa ilość lini z początku pliku

`tail -n <liczba> [plik]`

- Liczba określa ilość lini z końca pliku

`tail -f [plik]`

- Wyświetla pojawiające się nowe linie na końcu pliku

Przetwarzanie tekstu

- **tr**

- Podmień lub skasuj określone znaki

```
$ echo abcdefg | tr abcd ABCD
```

```
$ echo abcdefg | tr -d abcd
```

- **paste**

- Łączy wiele linii w jedną

```
$ echo -e "lin1\nlin2\nlin1" | paste -s
```

- Lub łączy linie z jednego pliku z liniami drugiego pliku

```
$ paste <( seq 1 2 10 ) <( seq 2 2 11 )
```

Przetwarzanie tekstu

- **sort**

- Posortuj linie

```
$ echo -e "lin1\nlin2\nlin1" | sort
```

```
$ du -sch /var/* 2>/dev/null | sort -h
```

- **uniq**

- Pomiń zduplikowane linie, które do siebie przylegają

```
$ echo -e "lin1\nlin2\nlin1" | uniq
```

```
$ echo -e "lin1\nlin2\nlin1" | sort | uniq
```

```
$ echo -e "lin1\nlin2\nlin1" | sort -u
```

Przetwarzanie tekstu

- **Ćwiczenie**

1)Policz zużycie pamięci RSS przez uruchomione procesy

```
$ ps -A --no-heading -o rss
```

2)Posortuj listę procesów wygenerowaną przez polecenie „*ps aux*” po kolumnie *rss* (bez użycia opcji *--sort*)

Przetwarzanie tekstu

- **Wyrażenia regularne w bash**

- Jeżeli tekst pasuje do wzorca, zwracane jest 0
- Odwołania wsteczne trafiają do tablicy BASH_REMATCH

```
$ cat test_regexp.sh
#!/bin/bash

regtest="tekst2345"

if [[ $regtest =~ ([a-z]{5})([0-9]{4}) ]] ; then
    echo ${BASH_REMATCH[*]}
fi
```

Programowanie w powłoce Bash

Wyrażenia regularne

