Programowanie w powłoce

autorstwa: Radka Przychody□□□□□□

Spis treści

Skrypty powłoki a programy

Jeśli ktoś miał do czynienia z plikami wsadowymi (Batch File) w DOS'ie to może myśleć o skryptach jako o takich plikach. Skrypty są interpretowane przez powłokę. Interpretowanie polega na tym, że w przeciwieństwie do programów zapisanych w postaci binarnej, skrypty są zwykłymi plikami tekstowymi, w których są zapisane polecenia zrozumiałe dla powłoki. Zadaniem powłoki jest przetłumaczenie ich na polecenia systemu. Jako że skrypt jest interpretowany da się go wykonać wpisując po kolei jego zawartość w linii poleceń, np. w następujący sposób w wierszu poleceń możemy zmienić rozszerzenia wszystkich plików *.jpg na *.jp.

```
$ for plik in *.jpg
> do
> mv $plik ${plik%.*}.jp
> done
$
```

Uruchomienie skryptu
Aby móc uruchomić skrypt należy mu nadać prawa do wykonywania (w systemie UNIX'owym rozszerzenie pliku nie decyduje o jego wykonywalności). Prawa te nadajemy komendą **chmod a+x <nazwa_skryptu>**. Po tym możemy uruchomić skrypt tak jak zwykły program wpisując w linii poleceń **./<nazwa_skryptu>**

Dowolny plik wykonywalny można uruchomić w tle pisząc na końcu polecenia znak **&.**
Przykład:

```
$ find / -perm +4000 -xdev >& suid_progs &
```

Powyższe wywołanie zapisze do pliku suid_progs nazwy wszystkich plików z włączonym bitem SUID. Polecenie zostanie wykonane w tle, dodatkowy operator **>&** zostanie wyjaśniony w następnym rozdziale.

Polecenia można wykonywać sekwencyjnie rozdzielając je średnikami. Wtedy następne polecenie wykona się dopiero po zakończeniu poprzedniego, a kodem wyjścia będzie kod ostatniego polecenia.
Przykład

```
$ vi skrypt ; chmod +x skrypt ; ./skrypt
```

W ten sposób uruchomimy edytor 'vi' do edycji pliku skrypt, po jego zakończeniu powłoka przydzieli mu prawa wykonywania i wykona ten skrypt.

Budowa skryptu
Skrypt powinien zaczynać się od linii postaci: #!/bin/powloka
Standardowo komentarze zaczynają się od znaku # i wszystkie linie zaczynające się od tego znaku są ignorowane, jednak w przypadku pierwszej linii skryptu jest inaczej. Jeśli za znakiem # stoi wykrzyknik, to powłoka odrzuca znaki #! i uruchamia */bin/powloka* <*nazwa_skryptu*> używając nazwy skryptu jako parametru dla tej powłoki. W przypadku

braku takiej linii w skrypcie, powłoka bieżąca uruchamia powłokę standardową (w linuxie zwykle /bin/sh). W ogólności /bin/powloka wcale nie musi być powłoką, może być dowolnym poleceniem, .np. następujący skrypt sam się kasuje:

```
#!/bin/rm
Zawartość skryptu nie jest istotna
jakies polecenia
```

Co powłoka rozwinie w **/bin/rm ./nazwa_skryptu**

Polecenie exit
Polecenie exit powoduje natychmiastowe zakończenie skryptu bez względu na miejsce wywołania. Dodatkowy parametr pozwala udostępnić otoczeniu kod wyjścia, który może należeć do przedziału 0 - 125, przy czym zero jest traktowane jako sukces. Kody powyżej 125 są zarezerwowane i mają następujące znaczenie:

Kody wyjścia > 125	
126	plik nie miał atrybutu wykonywalności
127	nie znaleziono pliku o danej nazwie
128 w górę	powodem zakończenia był sygnał (wartość kodu wyjścia równa jest w tym wypadku 128 + numer sygnału)

Do spisu treści

Potoki i przekierowania

Potoki i przekierowania to jedne z najczęściej używanych właściwości powłoki. Pozwalają one zmieniać standardowe wejście, wyjście oraz standardowe wyjście błędów lub je łączyć w potoki. Oczywiście możemy przekierowywać dowolne z deskryptorów, jednak w większości przypadków wykorzystuje się je jedynie na standardowych wejściach i wyjściach. Powłoka interpretuje polecenie od lewej do prawej, tak więc wszystkie potoki i przekierowania są brane pod uwagę właśnie w takiej kolejności

Potoki

Składnia
polecenie1 [| polecenie2] [| polecenie3] ...

Jak więc widać potok jest ciągiem poleceń porozdzielanych znakiem "|", przy czym standardowe wyjście polecenie poprzedzającego jest standardowym wejściem polecenia następnego. Przykład:

```
$ cat /etc/passwd | sort | sed -e 's/root/rut/g'> /etc/passwd
```

Powyższe wywołanie zamieni wszystkie wystąpienia słowa 'root' na 'rut' w pliku /etc/passwd (oczywiście, jeśli mamy prawo pisania do tego pliku). Pierwsze polecenie wyświetla plik /etc/passwd, drugie sortuje te dane, co jest następnie przetwarzane przez edytor strumieniowy 'sed' i przekierowaniem zapisywane do pliku.

Przekierowania

Przed wykonaniem polecenia powłoka sprawdza, czy nie powiązać określonych deskryptorów z innymi, bądź z plikami.

Przekierowanie wejścia **[n] < plik** powoduje otwarcie *pliku* do czytania i powiązanie deskryptora o numerze **n** z zawartością *pliku*. Jeśli pominiemy numer deskryptora powłoka przekieruje standardowe wejście (deskryptor nr 0).
Przykład:

$ **mail jakis@adres.com < wiadomosc.txt**

Przekierowanie wyjścia **[n] > plik** powoduje otwarcie *pliku* do pisania.) i powiązanie go z deskryptorem o numerze **n**. Jeśli plik nie istnieje zostanie stworzony, jeśli istnieje zostanie skrócony do zera (patrz opcja noclobber. Gdy pominiemy numer deskryptora powłoka przekieruje standardowe wejście (deskryptor nr 1).

Przekierowanie wyjścia z dopisywaniem **[n] >> plik** powoduje otwarcie *pliku* do dopisywania (jeśli nie istnieje zostanie stworzony) i powiązanie go z deskryptorem o numerze **n**. Gdy pominiemy numer deskryptora powłoka przekieruje standardowe wejście.

Przekierowanie wyjścia i standardowego wyjścia **&> plik** powoduje otwarcie *pliku* do pisania i powiązanie go ze standardowym wyjściem i standardowym wyjściem błędów.

Powyższe definicje niezbyt jasno tłumaczą różnicę między przekierowaniem wyjścia i wejścia. Na przekierowanie wejścia można patrzeć w ten sposób, że czytając z danego deskryptora będziemy czytali z pliku, a w przypadku przekierowania wyjścia pisząc do danego deskryptora wynik znajdzie się w pliku. Przekierowań wejścia rzadziej się używa, gdyż większość poleceń pozwala podać jako parametr plik wejściowy zamiast standardowego wejścia.

Przykład:

$ **find / -size +500k -xdev >& duze_pliki**

Wpisując powyższą linijkę w pliku *duze_pliki* dostaniemy listę plików bieżącego systemu plików zajmujących conajmniej 500 kilobajtów. W pliku tym zostaną też umieszczone też komunikaty o błędach.

Duplikowanie deskryptorów. Wywołanie [n] &< [m] powoduje, że czytając z deskryptora [n] dostajemy strumień z [m]. Podobnie wywołując [n] >& [m] pisząc do deskryptora [n] piszemy do deskryptora [m].

Przykład:

```
$ find / -size +500k -xdev > duze_pliki 2>&1
```

Wywołanie to jest równoważne wcześniejszemu. Standardowe wyjście jest powiązane z plikiem *duze_pliki*, a standardowe wyjście błędów ze standardowym wyjściem.

Opcja 'noclobber'

Włączenie opcji: **set -o noclobber** lub **set -C**
Po włączeniu tej opcji jeśli w przekierowaniach **&gr; plik** i **>& plik** *plik* istnieje, to zostanie wyświetlony błąd. Zabezpiecza to przed przypadkowym nadpisaniem istniejącego pliku. Aby zapisać do istniejącego pliku należy wyłączyć opcję 'noclobber' (**set +o noclobber** lub **set +C**) lub też użyć przekierowania **>| plik**.

Przykład:

```
$ touch plik
$ set -o noclobber
$ ls > plik
bash: plik: cannot overwrite existing file
$ ls >| plik
$ set +C
$ echo koniec > plik
```

Do spisu treści

Zmienne

W przeciwieństwie do większości języków programowania w skryptach nie deklaruje się zmiennych przed ich zastosowaniem. Ich pierwsze użycie jest jednocześnie ich deklaracją. Nazwa zmiennej może być dowolnym identyfikatorem, przy czym ważna jest wielkość liter i zmienne **VARIABLE** i **variable** są różne. Aby odwołać się do zawartości zmiennych trzeba przed nazwą zmiennej napisać znak **$**. Istotne jest także to, że zmienne są pamiętane przez powłokę jako ciągi znaków, dopiero odpowiednie polecenia interpretują te ciągi w odpowiedni sposób. Przykład:

```
$ ile=5
$ echo ile
ile
$ echo $ile
5
$ ile=5+5    # 5+5 jest ciągiem znaków (patrz: Obliczanie wyrażeń)
$ echo ile
```

```
5+5
$
```

Zmienne środowiskowe

W skrypcie możemy się odwołać do zmiennych środowiskowych w sposób identyczny do zwykłych zmiennych, pisząc znak dolara przed nazwą zmiennej. Aby zobaczyć listę zmiennych zdefiniowanych w systemie wpisz w powłoce **env**. Przykładowe zmienne:

$HOME	katalog domowy użytkownika
$PATH	lista katalogów przeszukiwana przez polecenia
$PS1	znak zgłoszenia powłoki
$IFS	separator pola wejściowego, wszystkie znaki z tej zmiennej używane są do oddzielania słów na wejściu

```
$ IFS=":"
$ read x y z
1:2:3
$ echo x=$x y=$y z=$z
x=1 y=2 z=3
$ read x y z
1 2 3
$ echo x=$x y=$y z=$z
x=1 2 3 y= z=
$ IFS=" :"
$ read x y z
1:2 3
$ echo x=$x y=$y z=$z
x=1 y=2 z=3
$ IFS=":,"
$ read x y z
1,2:3
$ echo x=$x y=$y z=$z
x=1 y=2 z=3
$ read x,y,z    *
1:2:3
bash: read: `x,y,z': not a valid identifier
$ d='x,y,z'
$ read $d    **
1:2:3
$ echo x=$x y=$y z=$z
x=1 y=2 z=3
$ read "$d"    ***
1:2:3
bash: read: `x,y,z': not a valid identifier
```

Kilka ostatnich linijek z pewnością wprowadziło trochę zamieszania i wymaga wytłumaczenia. Powłoka przeszukuje rezultaty rozwinięć parametrycznych, rozwinięć poleceń w apostrofach i rozwinięć arytmetycznych w poszukiwaniu znaków ze zmiennej IFS

{ (*)-nie jest rozwinięciem}. Jeśli takie znajdzie zamienia je na spacje (**). Jeśli jednak całość ujmiemy w cudzysłowy nie dojdzie do zamiany (***).

Polecenie EXPORT

To co przypiszemy zmiennym ginie po jego zakończeniu, a to dlatego, że uruchamiając skrypt dostajemy nowe środowisko i wszelkie przypisania zmiennych dotyczą tego środowiska. Uruchamiając jakiś inny proces z naszego skryptu dziedziczy on nasze środowisko, ale po zakończeniu skryptu wszystko znika. Istnieje jednak polecenie export, któro pozwala zmienić środowisko procesu rodzica. Często to polecenie jest wykorzystywane w pliku konfiguracyjnym powłoki (np. dla bash'a będzie to /etc/bashrc i w katalogu użytkownika .bashrc). **Składnia:**
export zmienna
lub
export zmienna=wartość
Przykład: export PATH=$PATH:$HOME/bin

Zmienne parametryczne

$0	nazwa wykonywanego skryptu
$#	liczba przekazanych parametrów
$$	numer identyfikacyjny procesu skryptu
$*	lista parametrów porozdzielanych pierwszym znakiem ze zmiennej $IFS, pod spodem przykład
$@	lista parametrów porozdzielanych spacjami
$1,$2,...	poszczególne parametry w kolejności podanej przy uruchomieniu skryptu

```
#!/bin/sh
IFS=:
echo "$*"
echo $*
echo $@
echo $1 $3
```

Wykonując skrypt z parametrami: **raz dwa trzy** otrzymamy wynik:
raz:dwa:trzy
raz dwa trzy
raz dwa trzy
raz trzy

Polecenie unset

Polecenie **unset** anuluje przypisania zmiennym wartości usuwając te zmienne ze środowiska. Można również je stosować do funkcji. Przykład:

```
$ zmienna=5
$ echo $zmienna
5
```

$ **unset zmienna**
$ **echo $zmienna**

$

Do spisu treści

Cudzysłowy

Istnieją dwa rodzaje cudzysłowów: **" "** (podwójne) i **' '** (pojedyncze). Istnieją jeszcze ` ` (apostrofy) omówione w następnym rozdziale. Cudzysłowów umożliwiają przypisanie zmiennej ciągu zawierającego spacje. Normalnie powłoka traktuje spacje jako separatory parametrów. Jeśli w poprzednim skrypcie chcielibyśmy przekazać jako parametr ciąg zawierający spacje, musielibyśmy wywołać skrypt następująco:
./nazwa_skryptu raz 'dwa trzy' cztery
co po wykonaniu dałoby nam:
raz:dwa trzy:cztery
raz dwa trzy cztery
raz cztery

Cudzysłowy **" "** i **' '** różnią się tym, że w przypadku tych pierwszych powłoka rozwija wszystkie nazwy zmiennych w wartości tych zmiennych (rozwinięcia parametryczne), rozwija też polecenia w apostrofach (patrz następny rozdział) oraz odpowiednio traktuje pewne znaki poprzedzone backslashem **'\'**. Nie jest tak w przypadku drugich apostrofów. Dodatkowo jeśli chcemy między cudzysłowami **" "** użyć tego samego cudzysłowu, musimy poprzedzić go backslashem \. Nie dotyczy to cudzysłowów pojedynczych, gdyż znak cudzysłowu po backslashu będzie traktowany jako koniec łańcucha. Nic nie stoi jednak na przeszkodzie, by wewnątrz cudzysłowów **" "** używać **' '** i odwrotnie. Aby nie doszło do rozwinięcia zmiennej w cudzysłowach podwójnych możemy znak dolara poprzedzić backslashem. Aby otrzymać backslash musimy napisać go podwójnie. Oczywiście w cudzysłowach pojedynczych nie dojdzie do żadnego rozwinięcia. Przykład:

$ **kat='$HOME'**
$ **echo $kat**
$HOME
$ **echo "\$HOME"**
$HOME
$ **kat="$HOME"**
$ **echo $kat**
/home/radek
$ **kat='Cudzysłów " podwójny'**
$ **echo $kat**
Cudzysłow " podwójny
$ **kat="Cudzysłów \" podwójny"**
Cudzysłów " podwójny

Poniżej wymienione są sekwencje poprzedzone backslashem zamieniane wewnątrz cudzysłowów podwójnych. Pozostałe sekwencje poprzedzone backslashem (wymienione przy poleceniu echo) nie są zamieniane.

\$	$
\`	`
\"	"
\\	\
\n	nowa linia

Do spisu treści

Apostrofy

Apostrofy ` ` mają całkiem odmienne od cudzysłowów znaczenie. Powłoka wykonuje polecenie zawarte między nimi, a wartością wyrażenia jest wynik działania tego polecenia.

```
$ data=`date`
$ echo $data
Sun Sep 24 22:37:46 2000
```

Zamiast dwóch apostrofów można zamiennie użyć konstrukcji: **$( )**, np.:
data=$(date)

Podstawienia komend można zagnieżdżać.

Do spisu treści

Obliczanie wyrażeń

Aby obliczyć wyrażenie możemy skorzystać z zewnętrznego polecenia expr lub użyć wewnętrznej konstrukcji BASH'a, która jest szybsza gdyż nie potrzebuje oddzielnego procesu i jest wygodniejsza w użyciu. Czasami trzeba jednak użyć 'expr', dlatego zostanie później zaprezentowane.

Operatory arytmetyczne	
- , +	minus i plus dwu- i jednoargumentowy
! , ~	logiczna i bitowa negacja
**	potęga
* , / , %	mnożenie, dzielenie całkowite i reszta z dzielenia
<< , >>	przesunięcia bitowe
<= , >= , < , > , == , !=	porównania
& , ^ , |	bitowe AND XOR i OR
&& , ||	logiczne AND i OR
wyr?wyr:wyr	wyrażenie warunkowe (patrz język C)
= , *= , /= , %= , += , -= , <<= , >>= , &= , ^= , |=	przypisania (patrz język C)

Tych operatorów możemy używać w następujących konstrukcjach:

$((wyrażenie))
((wyrażenie))
let wyrażenie

Dwie ostatnie linijki są sobie równoważne. Są one poleceniami w przeciwieństwie do linijki pierwszej, która jest rozwinięciem arytmetycznym i nie może wystąpić samodzielnie. Występuje przeważnie po prawej stronie znaku przypisania, w poleceniu 'echo', itp.

Przykład:

$ **zmienna=0**
$ **echo $((zmienna=zmienna+2))**
2
$ **((zmienna=$zmienna**3));echo $zmienna**
8
$ **let zmienna+=(($zmienna==8)?2:0)**
$ **echo $zmienna**
10

Do spisu treści

Rozwinięcia parametryczne

Załóżmy, że mamy 100 plików o nazwach: plik1.txt, plik2.txt, ... plik100.txt i chcemy te pliki połączyć w jeden o nazwie plik.txt Sprawę załatwi poniższy skrypt:

```
#!/bin/sh
zmienna=0
cat /dev/null > plik.txt    # tworzenie pustego pliku
while [ zmienna -ne 101 ]; do
  zmienna=$(($zmienna+1))
  cat plik$zmienna.txt >> plik.txt
done
```

Jeśli jednak nasze pliki nazywają się np. plik1XX.txt, plik2XX.txt ... plik100XX.txt, to po modyfikacji linijki **cat plik$zmiennaXX.txt >> plik.txt** dostaniemy komunikat o braku pliku. Jak można się domyślić powłoka rozwinęła zmienną **$zmiennaXX**, która jest niezdefiniowana. Aby to zadziałało wystarczy zapisać tę linijkę w następujący sposób: **cat plik${zmienna}XX.txt >> plik.txt.**

W poniższych przykładach **slowo** może być wyrażeniem arytmetycznym, ciągiem znaków, wyrażeniem parametrycznym, poleceniem w odwrotnych apostrofach.

Inne rozwinięcia parametryczne	
${parametr:-slowo}	wartością jest wartość zmiennej *parametr* jeśli jest ona zdefiniowana i nie jest pusta w przeciwnym wypadku wartością jest *slowo* (zmienna parametr się nie zmienia)
${parametr:=slowo	wartością wyrażenia jest wartość zmiennej *parametr* jeśli jest ona zdefiniowana i nie jest pusta, w przeciwnym wypadku wartością wyrażenia i zmiennej parametr staje się *slowo*
${parametr:?slowo}	wartością wyrażenia jest wartość *parametru*, jeśli jest on zdefiniowany i nie pusty, w przeciwnym wypadku wyświetlana jest wiadomość powstała po rozwinięciu *slowo* (jeśli pominiemy *slowo* zostanie wyświetlony komunikat o tym, że zmienna *parametr* jest niezdefiniowana bądź równa null
${parametr:+slowo}	jeśli *parametr* jest niezdefiniowany bądź null nic nie jest podstawiane, w przeciwnym wypadku wartością jest *slowo*

${#parametr}	zwraca długość zmiennej *parametr* w znakach
${parametr#slowo}	zwraca wartość parametru po usunięciu z niego od początku najkrótszego ciągu pasującego do słowa
${parametr##slowo}	zwraca wartość parametru po usunięciu z niego od początku najdłuższego ciągu pasującego do słowa
${parametr%slowo}	zwraca wartość parametru po usunięciu z niego od końca najkrótszego ciągu pasującego do słowa
${parametr%%slowo}	zwraca wartość parametru po usunięciu z niego od końca najdłuższego ciągu pasującego do słowa

Przykład1:

```
#!/bin/sh
plik=/etc/lilo.conf
tmp1=${plik##*/}k
tmp2=${plik#*/}
tmp3=${plik%/*}
echo $plik
echo $tmp1
echo $tmp2
echo $tmp3
```

Wykonanie skryptu spowoduje wypisanie następujących linii:
/etc/lilo.conf
lilo.conf
etc/lilo.conf
/etc

Przykład2:

```
#!/bin/sh
line=`grep ^root /etc/passwd`
echo Shell roota: ${line##*:}
```

Do spisu treści

Polecenie 'echo'

Echo jest poleceniem wyświetlającym na ekranie swoje argumenty. Jeśli uruchomimy je z opcją **-n** nie zostanie dodany znak końca wiersza, np.:

```
$ echo -n Podaj liczbę:
$ read liczba
$ echo Wprowadziłeś $liczba
```

Polecenie to ma jeszcze jedną ciekawą opcje, mianowicie **-e**. Po jej włączeniu znaki poprzedzone backslashem są interpretowane. Są to następujące znaki:

\a	dźwięk speakera
\b	backspace
\c	to samo co z opcją -n
\e	escape
\f	line feed
\r	powrót karetki (\f\r daje to samo co \n)
\t	tabulator poziomy
\v	tabulator pionowy
\n	nowa linia
\\	backslash
\nnn	znak o kodzie ósemkowym nnn
\xnn	znak o kodzie szestnastkowym nn

Jako że w powłoce znak backslash jest traktowany specjalnie, musimy go podwoić, bądź wpisywać ciągi znaków w cudzysłowach.
Przykład:

```
#!/bin/sh echo -e \\a
sleep 1
echo -e "\a"
```

Polecenie 'printf' Pominę wyjaśnienie tego polecenia, dla tych co go nie znają odsyłam do podręczników języka C. Można tu tylko powiedzieć o tym, że nie dopuszcza się konwersji liczb rzeczywistych (w skrypcie takowe nie występują).

Do spisu treści

'Here-documents'

'Here documents' - dokumenty miejscowe. Jest to rodzaj przekierowania, które pozwala część treści skryptu traktować jako standardowe wejście.

Składnia:

<<ogranicznik
  here-document
ogranicznik

Przykład:

```
#!/bin/sh
cat << KONIEC
Ten tekst zostanie
wyświetlony na ekranie
ogranicznikiem jest KONIEC
KONIEC
```

Po wykonaniu powyższego skryptu otrzymamy na ekranie wszystko to co zawarliśmy między wyrazami KONIEC. Ten sposób może być używany do wyświetlenia większej ilości informacji na ekran bez konieczności wpisywania echo w każdej linii. W dokumencie miejscowym dokonuje się wszystkich rozwinięć. Ogranicznik końcowy musi znaleźć się sam w jednej linii, dlatego ostatnie słowo KONIEC pojawia się na ekranie. Pierwszy ogranicznik może być dowolnym wyrażeniem, szczegóły w podręczniku systemowym.

Do spisu treści

Polecenie 'test'

Polecenie to ma wiele opcji i jest często wykorzystywane w skryptach do realizowania różnych warunków logicznych, gdyż zawsze zwraca wartość logiczną. Posiada dwa rodzaje składni, np.:

```
$ if test -f /etc/passwd; then
> echo passwd na swoim miejscu
> fi
```

jest równoważne wykonaniu:

```
$ if [ -f /etc/passwd ]; then
> echo passwd na swoim miejscu
> fi
```

Należy pamiętać o istotnym szczególe, mianowicie o pozostawieniu przynajmniej po jednej spacji po nawiasie otwierającym i przed zamykającym.

Opcje które mogą pojawić się po słowie test lub w nawiasach:
Po prawej stronie wypisane są sytuacje, kiedy polecenie zwraca prawdę.

Porównanie ciągów	
lancuch	ciąg nie jest pusty

lancuch1 = lancuch2, lancuch == lancuch2	ciągi są jednakowe
lancuch1 != lancuch2	ciągi są różne
-n lancuch	ciąg nie jest pusty
-z lancuch	ciąg jest pusty
Porównania arytmetyczne	
wyr1 -eq wyr2	wyrażenia są równe
wyr1 -ne wyr2	wyrażenia są różne
wyr1 -gt wyr2	wyr1 jest większe od wyr2
wyr1 -ge wyr2	wyr1 jest większe równe od wyr2
wyr1 -lt wyr2	wyr1 jest mniejsze od wyr2
wyr1 -le wyr2	wyr1 jest mniejsze równe od wyr2
wyr1 -a wyr2	wyrażenia 1 i 2 są prawdziwe (AND)
wyr1 -o wyr2	jedno z wyrażeń jest prawdziwe (OR)
! wyr	wyrażenie jest zerowe
Sprawdzenia plików	
-a plik	plik istnieje
-b plik	plik jest urządzeniem blokowym
-c plik	plik jest urządzeniem znakowym
-d plik	plik jest katalogiem
-e plik	plik istnieje
-f plik	plik jest plikiem regularnym
-g plik	plik z ustawionym bitem SGID
-h plik	plik jest linkiem symbolicznym
-G plik	plik, którego GID właściciela = EGID
-k plik	plik z ustawionym bitem sticky
-L plik	plik jest linkiem symbolicznym
-O plik	plik, którego UID właściciela = EUID
-p plik	plik jest łączem nazwanym (PIPE)

-r plik	plik jest odczytywalny
-s plik	plik ma niezerową wielkość
-S plik	plik jest gniazdem
-u plik	plik z ustawionym bitem SUID
-w plik	plik jest zapisywalny
-x plik	plik jest wykonywalny
plik1 -ef plik2	pliki mają ten sam numer urządzenia i i-węzła
plik1 -nt plik2	plik1 jest młodszy niż plik2
plik2 -ot plik2	plik1 jest starszy niż plik2
Inne	
-o nazwa_opcji	opcja powłoki*nazwa_opcji*jest włączona

Jest jeden szczegół o którym warto pamiętać używając opcji -G i -O. Można nadać skryptowi bit SUID i SGID, ale nie jest on honorowany. Dlatego skrypcie EUID (efektywny ID) jest równy UID (rzeczywisty ID).

Nie zapominajmy o spacjach przed i po znakach równości i różności w warunkach.
Może być kilka niejasności odnośnie niektórych warunków dotyczących sprawdzania plików, ale nie to jest przedmiotem tego kursu i nie zostaną tu wyjaśnione. W większości wypadków wystarczy znać opcje z literami: d, f, r, s, w, x.

Rolę spójników logicznych AND i OR pełnią odpowiednio operatory: **-a** i **-o**, np. [wyr1 -a wyr2 -o wyr3].

Polecenie 'test' jest poleceniem zewnętrznym dla powłoki. '[' jest linkiem symbolicznym do polecenia test. Pisząc więc **[wyrażenie]** zostaje wywołane polecenie test z dodatkowym argumentem ']'. Istnieje jednak wewnętrzna konstrukcja w BASH'u, która pozwala obliczać wyrażenia logiczne. Składnia:

[[wyrażenie]]

Przyjmuje ona taki sam zestaw opcji oprócz spójników logicznych AND i OR, którymi są tu odpowiednio **&&** i ||

Do spisu treści

Polecenie 'expr'

Jak mówi podręcznik systemowy, polecenie **expr** wykonuje obliczenie wyrażenia i zapisuje je na standardowe wyjście. Jako, że jest to polecenie zewnętrzne dla powłoki wykorzystuje się je wpisując w odwrotne apostrofy, bądź w równoważną im konstrukcję **$(*polecenie*)**.
Przykład:

```
$ ile=5
$ ile=`expr $ile + 5`
$ ile=$(expr $ile + 5)
$ echo $ile
15
```

Polecenie przyjmuje w parametrze ciąg operandów i operatorów, które muszą być porozdzielane spacjami. Operandy mogą być liczbami bądź ciągami znaków.

Operatory	
| , &	operatory logiczne OR (gdy wynik nie jest zerem zwracany jest pierwszy niezerowy argument) i AND (gdy wynik nie jest zerem zwracany jest pierwszy z argumentów)
< , <= , = , == , != , >= , >	operatory porównania (operatory: '=' i '==' są równoważne)
+ , - , * , / , %	operatory arytmetyczne (odpowiednio: dodawanie, odejmowanie, mnożenie, dzielenie całkowite, dzielenie modulo)
:	dokonuje porównania wzorców, oba argumenty rzutowane są na napisy przy czym drugi z argumentów może być wyrażeniem regularnym w postaci akceptowalnej przez GREP'a. Jeśli wzorce pasują do siebie zwracana jest długość napisu, jeśli nie, zwracane jest zero. Dodatkowo jeśli w drugim argumencie umieszczona zostanie **jedna**para nawiasów '\(' i '\)', to w przypadku pasowania wzorców zostanie zwrócony napis zawarty między tymi nawiasami lub napis pusty w przypadku przeciwnym.
rozpoznawane słowa kluczowe	
match napis *wyr_regularne*	to samo co 'napis : wyr_regularne'
substr *napis pozycja długość*	zwraca podnapis z podanego napisu o podanej długości i zaczynający się od podanej pozycji

index *napis znaki*	zwraca pozycję wystąpienia tego znaku ze zbioru znaków będącego drugim argumentem, którego zajmuje najniższą pozycję w napisie
length *napis*	zwraca długość napisu

Przykłady:

```
$ napis='Ala ma kota'
$ expr index "$napis" k
8
$ expr index "$napis" A
1
$ expr index "$napis" kA
1
$ expr substr "$napis" 8 3
kot
$ expr match "$napis" 'Ala ma kota'
11
$ expr "$napis" : 'Ala ma psa'
0
$ expr "$napis" : 'Ala .*'   # wyrażenie regularne
11
$ expr "$napis" : 'Ala ma \(kot\)a'
kot
```

Do spisu treści

Konstrukcja 'if'

Jest to instrukcja warunkowa, z którą każdy, kto programował w jakimś języku programowania musiał się zetknąć.
Składnia polecenia

```
if warunek
then
  instrukcje
else
  instrukcje
fi
```

Jako że jesteśmy przyzwyczajeni do pisania **else** w tej samej linii co **if**, możemy tak dalej robić, tylko musimy oddzielić instrukcję **if** *warunek* od **then** średnikiem, gdyż inaczej powłoka traktuje **then** jako dokończenie warunku. Przykład:

```
$ if [ -f /etc/passwd ]; then echo Ufff ; else echo Ups; fi
```

W przypadku złożonych warunków słowa **else if** stojące obok siebie można złączyć w słowo **elif.** Poniższy skrypt drukuje największą z trzech liczb przekazanych jako parametry:

```
#!/bin/sh
if [ $1 -ge $2 ]; then
  if [ $1 -ge $3 ]; then echo $1
  else echo $3
  fi
elif [ $2 -ge $3 ]; then echo $2
else echo $3
fi
```

Wykonajmi poniższe polecenia, niech **imie** będzie zmienną niezainicjowaną.

```
$ if [ $imie = "franek" ]
> then echo Cze franek
> fi
[: =: unary operator expected
```

Ostatni komunikat błędy spowodowany jest tym, że zmienna **imie** jest niezainicjowana, wobec czego warunek w nawiasach staje się warunkiem **[= "franek"]**. Czyli brakuje jednego parametru. Problem rozwiąże następujący zapis: **["$imie" = "franek"]**, a warunek staje się warunkiem **["" = "franek"].**

Do spisu treści

Listy AND i OR

Lista AND
Składnia

```
instrukcja1 && instrukcja2 && instrukcja3 && instrukcja4 ...
```

Wykonywane są instrukcje po kolei aż do momentu określenia wyniku całości wyrażenia, czyli jeśli np. instrukcja1 zwróci FALSE, to instrukcja2 się już nie wykona, a całość wyrażenia przyjmie wartość FALSE. Aby otrzymać TRUE, wszystkie spośród instrukcji muszą zwrócić TRUE, w tym wypadku wszystkie one zostaną wykonane. Zamiast instrukcji może wystąpić dowolne wyrażenie, wtedy zamiast wykonywania instrukcji zostanie obliczona wartość tego wyrażenia. Całość można traktować jako ciąg iloczynów logicznych przetwarzany od lewej do prawej, w którym instrukcje są wykonywane dopóki wartość wyrażenia nie jest jeszcze określona.

Lista OR
Składnia

```
instrukcja1 || instrukcja2 || instrukcja3 || instrukcja4 ...
```

Analogicznie jak w przypadku listy AND instrukcje są wykonywane do momentu określenia całości wyrażenia, tylko w tym przypadku jest odwrotnie, mianowicie jeśli dowolna z instrukcji przyjmie wartość TRUE, to wartość wyrażenia będzie określona jako TRUE i dalsze instrukcje się nie wykonają. Zwrócenie przez instrukcję FALSE powoduje dalsze przetwarzanie wyrażenia. Aby otrzymać FALSE, wszystkie instrukcje muszą zwrócić FALSE. Wyrażenie można traktować jako ciąg sum logicznych.

Można łączyć ze sobą listy AND i OR, by uzyskać bardziej skomplikowane warunki. Żaden z operatorów **&&** i || nie ma wyższego priorytetu od drugiego. Całość wyrażenia jest wykonywana sekwencyjnie od lewej do prawej, jeśli chcemy to zmienić można użyć nawiasów zwykłych.

Przykład 1

```
$ [ -f /etc/passwd ] && echo Ufff || echo Ups
Ups :)
```

Przykład 2

```
#!/bin/sh
[ $1 -ge $2 ] && ( [ $1 -ge $3 ] && echo $1 || echo $3) ||    \
([ $2 -ge $3 ] && echo $2) || echo $3
```

Powyższy skrypt wyświetla na ekran największy z 3 podanych mu parametrów.

Nowy element pojawił się w 2 wierszu w przykładzie 2 na końcu linii - backslash. Postawienie znaku backslash \ na końcu wiersza informuje powłokę, że dana linijka jest kontynuowana w następnym wierszu (normalnie następny wiersz to następna instrukcja). Dzięki temu nie musimy pisać długich wierszy, które przeszkadzają w wygodnym edytowaniu i oglądaniu pliku (nie wszystkie edytory obsługują automatyczne zawijanie).

Do spisu treści

Blok instrukcji

Wszędzie tam, gdzie musimy użyć pojedynczej instrukcji, możemy zastosować blok

instrukcji, czyli ciąg instrukcji ujęty w nawiasy klamrowe lub zwykłe.

Składnia:
{ ciąg poleceń**; }**

lub

(ciąg poleceń**)**

W przypadku nawiasów klamrowych polecenia są wykonywane w bieżącej powłoce i środowisku. Wartością takiego bloku jest wartość ostatniej instrukcji. Dla poleceń w nawiasach zwykłych tworzona jest nowa powłoka. Wartością zwracaną jest kod ostatniej instrukcji bądź wartość zwrócona przez komendę 'exit'.

Przykład:

```
$ if true && { false;false;true }
> then echo TRUE
> fi
TRUE
$ if true && (false;false;exit 0)
> then echo TRUE
> fi
TRUE
```

true i **false** są poleceniami zwracającymi odpowiednio prawdę i fałsz. Dodatkowo **false** jest często wpisywane w /etc/passwd w ostatniej kolumnie, by uniemożliwić użytkownikowi wejście na shella. Zamiast polecenia **true** może też wystąpić **:** (pojedynczy dwukropek), któro jest poleceniem pustym zwracającym prawdę, np.

```
$ while :
> do echo -e \\a
> done
```

Do spisu treści

Konstrukcja 'case'

Składnia polecenia:

```
case zmienna in
ciag_wzorca [ | ciag_wzorca ] ... ) ciag_instrukcji ;;
ciag_wzorca [ | ciag_wzorca ] ... ) ciag_instrukcji ;;
...
esac
```

Polecenie to działa w ten sposob, że dopasowuje zmienną po kolei do wzorcow i po udanym dopasowaniu wykonuje ciąg instrukcji przyporządkowany temu wzorcowi. Jeśli zmienna pasuje do kilku wzorcow, wykona się tylko pierwszy. W przypadku, gdy nie pasuje do żadnego wzorca nic się nie wykona. Aby wykonała się jakaś czynność domyślna stosuje się znak * jako wzorzec. Wzorzec ten pasuje do każdej wartości zmiennej, należy więc pamietać, by wpisać go na końcu. Przy stosowaniu polecenia należy się kierować zasadą, by bardziej ogólne wzorce umieszczać dalej. Najlepiej jednak zobaczyć to na przykładzie:

```
#!/bin/sh
echo Czy chcesz kontynuować?
read x
case $x in
"tak" | "TAK" | "T" | "t" ) echo Wpisałeś tak;;
"nie" | "NIE" | "N" | "n" ) echo Wpisałeś nie;;
* ) echo Nie wiem co wpisałeś;;
esac
```

Skracając i uogólniając powyższe warunki otrzymamy poniższy skrypt. Wtedy do pierwszego wzorca pasuje litera T lub t lub napis "tak" w którym dowolna z jego liter może być mała lub duża.

```
#!/bin/sh
echo Czy chcesz kontynuować?
read x
case $x in
[Tt][Aa][Kk] | [Tt] ) echo Wpisałeś tak;;
[Nn][Ii][Ee] | [Nn] ) echo Wpisałeś nie;;
* ) echo Nie wiem co wpisałeś;;
esac
```

Do spisu treści

Pętla 'for'

Składnia polecenia

```
for zmienna in zbiór_wartości do
  instrukcje
done
```

Pętla FOR działa w ten sposób, że dla każdego elementu ze zbioru wartości przypisuje go do zmiennej i wykonuje instrukcje zawarte wewnątrz pętli. Zbiór wartości nie musi być podany jawnie, może być rozwinięty przez powłokę, np.:

```
#!/bin/sh
for zm in *
do
   if [ -u $zm ]
  then echo $zm
   fi
done
```

W tym wypadku powłoka rozwija znak ***** w listę plików z bieżącego katalogu. Powyższy skrypt drukuje na ekranie pliki, które mają ustawiony bit SUID.

Poniższy przykład pokazuje jak wysłać do wszystkich użytkowników systemu posiadających konta shellowe z powłoką /bin/bash. Pamiętajmy o odwróconych cudzysłowach.

```
#!/bin/sh
for user in `cat /etc/passwd | grep /bin/bash | cut -d : -f -1`
do
echo "Cześć mam do sprzedania fortepian" | mail -s "Spam" $user
done
```

Do spisu treści

Pętla 'while'

Składnia polecenia

```
while warunek
do
   instrukcje
done
```

Warunek sprawdzany jest przed wywołaniem instrukcji. Instrukcje będą wykonywane dopóki warunek jest prawdziwy. Przykład:

```
#!/bin/sh
i=1
while [ $i -le 5 ]; do
echo "$i) Opcja $i"
i=$(($i+1))
done
```

Do spisu treści

Pętla 'until'

Składnia polecenia

```
until warunek
do
  instrukcje
done
```

Pętla ta jest podobna do pętli WHILE z tą różnicą, że instrukcje są wykonywane dopóty warunek jest fałszywy. Jeśli warunek staje się prawdziwy pętla jest przerywana. Przykład:

```
#!/bin/sh
until who | grep root > /dev/null
do
  sleep 2
done
echo -e \\a
echo "**** root się zalogował ****"
```

Ten skrypt co 2 sekundy sprawdza, czy do systemu zalogował się root. Jeśli tak, to powiadamia nas i kończy działanie. Jeśli w momencie uruchomienia skryptu root jest już w systemie, instrukcje w pętli UNTIL nie wykonają się ani razu.

Do spisu treści

Pętla 'select'

Składnia polecenia

```
select zmienna [ in zbiór_wartości ]
do
  instrukcje
done
```

Zbiór wartości jest wyświetlany na standardowym wyjściu, każdą pozycję poprzedza numer. Jeśli pominiemy *in zbiór_wartości* wyświetlone zostaną parametry pozycyjne (parametry

przekazane do skryptu bądź parametry przekazane funkcji). Wyświetlany jest następnie znak zachęty systemu (ze zmiennej PS3) i powłoka oczekuje na wprowadzenie numeru. Jeśli wprowadzimy poprawny numer zmienna z polecenia otrzyma wartość odpowiadającą temu numerowi, jeśli nie jest to numer, bądź numer jest błędny zmienna ta otrzyma wartość NULL. Następnie wykonywane są instrukcje wewnątrz bloku. Potem wszystko zaczyna się od początku. Aby wyjść z pętli należy wśród instrukcji wstawić polecenie break lub return lub też wprowadzić znak EOF. Dodatkowo wprowadzona przez nas linia jest pamiętana w zmiennej REPLY.

Przykład:

$ **select plik in ***
> **do**
> **echo Wpisana linia: $REPLY**
> **Wybrałeś $plik**
> **break**
> **done**

Do spisu treści

Polecenia 'break' i 'continue'

Oba polecenia służą do wcześniejszego wyjścia z aktualnego przebiegu pętli, z tą różnicą, że po wykonaniu BREAK wykonywanie pętli kończy się całkowicie podczas, gdy po wykonaniu CONTINUE pomijane są polecenia za CONTINUE i wykonywany jest następny przebieg pętli. Najlepiej zobaczyć to na przykładzie:

```
#!/bin/sh
tymczasowy='names.$$$'
cat /dev/null > $tymczasowy
for zm in *
do
if [ -d $zm ] || [ -x $zm ] || [ $zm = *.tar ] || [ $zm = *.gz ] || [
$zm = $tymczasowy ]
then continue; fi
echo Dodaję $zm do archiwum.
echo $zm >> $tymczasowy
done
tar -T $tymczasowy -cf $1
rm -f $tymczasowy
exit 0
```

Powyższy skrypt pakuje do archiwum o nazwie przekazanej w parametrze wszystkie pliki z bieżącego katalogu, które nie są katalogami, nie mają atrybutu wykonywalności i nie mają rozszerzeń *.tar lub *.gz. Skrypt najpierw zapisuje do pliku tymczasowego nazwy plików, które mają być przetworzone, a następnie wykonuje polecenie tar z odpowiednim przełącznikiem, by czytał nazwy plików do spakowania z pliku tymczasowego. Ostatnie porównanie w liście OR $zm=$tymczasowy służy do tego, by w pliku tymczasowym nie znalazła się nazwa pliku tymczasowego.

Do spisu treści

Polecenie 'shift'

Podając parametry do skryptu możemy się do nich odwoływać w skrypcie za pomocą nazw $1, $2, ... , $9. Aby uzyskać dostęp do argumentu 10 i powyżej należy wykonać odpowiednią ilość razy polecenie SHIFT. Pojedyncze wywolanie polecenia przesuwa wszystkie argumenty w ten sposób, że $1 zawiera teraz $2, a $9 zawiera 10 argument. Argument, który uprzednio był w $1 jest tracony. Gdy już wyczerpie się lista argumentów przypisywane są ciągi puste. Przykład:

```
#!/bin/sh while [ -n "$1" ]; do
  ile=$(($ile+1))
done
echo -n Wpisałeś $ile parametr
case $ile in
  1 ) echo . ;;
  2 | 3 | 4) echo y. ;;
  * ) echo ów. ;;
esac
```

Do spisu treści

Polecenie 'set'

Polecenie to ustawia zmienne parametryczne powłoki przez co stają się dostępne w taki sam sposób, jak gdyby były podane jako parametry skryptu.

```
#!/bin/sh
set raz dwa trzy
echo $1 $3
```

Wykorzystując dodatkowo zmienną IFS możemy napisac poniższy skrypt, który przetwarza podany mu na wejście plik /etc/passwd wypisując linie zawierające użytkownika i odpowiadający mu katalog domowy. Łącząc go np. z poleceniem grep, możemy wyświetlić tylko tych użytkowników, którzy mają powłokę bash, np. **grep /bin/bash /etc/passwd | nazwa_skryptu**

```
#!/bin/sh
IFS=":"
read x
while [ -n "$x" ]; do
  set $x
  echo $1 $6
  read x
done
```

Do spisu treści

Polecenie 'trap'

Polecenie służy do przechwytywania sygnałów wysłanych do procesu skryptu. Listę sygnałów możemy zobaczyć wpisując w powłoce polecenie **trap -l**
Składnia polecenia: **trap <polecenie> <sygnał>** Pisząc zamiast polecenia myślnik przypisujemy sygnałowi domyślną akcję.
Przykład: (nie zapominajmy o odwrotnych cudzysłowach)

```
#!/bin/sh
trap `echo Nie zamknę się, hehehe` SIGINT
trap `echo No dobra będę już grzeczny; trap - SIGINT`
SIGHUP
while :
do
  echo Naciśnij ^C aby wyjść
  sleep 2
done
```

Pierwsze polecenie trap powoduje, że program na wysłanie mu sygnału SIGINT będzie reagował pierwszym komunikatem. Następna linijka mówi, że wysyłając sygnał SIGHUP skrypt również wypisze komunikat, ale też przypisze standardowe działanie dla sygnału SIGINT. Tak więc początkowo naciskając klawisze ^C nie uda nam się zamknąć programu, ale wystarczy wysłać do procesu sygnał SIGHUP, by było to możliwe. Aby wysłać do dowolnego procesu w systemie jakiś sygnał używamy polecenia **kill** [patrz: man kill]. W naszym przypadku będzie to polecenie **kill -SIGHUP PID**, gdzie **PID** to numer identyfikacyjny naszego procesu, sprawdzimy to za pomocą polecenia **ps x** [patrz: man ps]. PID procesu jest podawany w pierwszej kolumnie, a który to proces znajdziemy po nazwie z ostatniej kolumny.

Do spisu treści

Funkcje, instrukcja 'return'

Składnia:

```
nazwa_funkcji( )  {
  instrukcje
}
```

lub

```
function nazwa_funkcji( )  {
  instrukcje
}
```

Parametry do funkcji przekazujemy pisząc w miejscu wywołania nazwę funkcji i listę parametrów. Wewnątrz funkcji do parametrów możemy się dostać tak, jak w bloku głównym do parametrów przekazanych skryptowi, czyli przez zmienne $1, $2, itd. Odpowiednio się zmieniają także zmienne $#, $*, $@. Po zakończeniu działania funkcji parametrom pozycyjne ($#, $*, $@, $1, ..) przywracane są ich pierwotne wartości. Instrukcja 'return' powoduje przerwanie wykonania funkcji i przetwarzanie następnej linijki od miejsca wywołania. Instrukcję **'return'** można wykorzystać do tego, by funkcja mogła zwrócić jakąś wartość, jednak nie można umieścić nazwy funkcji po prawej stronie znaku przypisania, tak więc nie można pobrać tej wartości. W praktyce zwraca się jedną z dwu wartości: jeden lub zero, a wynik działania funkcji można sprawdzić w instrukcji 'if'. Aby napisać funkcję, która zwraca jakąś wartość trzeba w tej funkcji użyć nazwy zmiennej, która będzie dostępna w miejscu wywołania funkcji. Poniższy przykład pokazuje jak napisać funkcję zwracającą kwadrat liczby przekazanej w parametrze. Wykorzystujemy tu zmienną pomocniczą **kw**, której zostanie przypisany kwadrat wyrażenia.

```
$!/bin/sh
kwadrat() {
kw=$(($1*$1))
}
kwadrat 2
echo $kw
```

W kolejnym przykładzie wykorzystujemy instrukcję 'return' do zwrócenia powodzenia danej funkcji, przy czym wartość równa zeru będzie interpretowana jako powodzenie, a różna od zera jako porażka. Zatem jeśli będziemy chcieli napisać funkcję, która ma zostać użyta w instrukcji IF lub w listach AND i OR, to aby wyrażenie przyjęło wartość prawdy musimy zwrócić zero. Poniższa funkcja oczekuje dwóch parametrów: pliku i znaku mówiącego o prawie dostępu (jeden ze znaków rwx). Gdy użytkownik ma dla danego pliku prawo do operacji przekazanej w parametrze funkcja zwróci zero.

```
#!/bin/sh
czy_masz_prawa() {
case $2 in
"r") if [ -r $1 ]; then return 0; fi ;;
"w") if [ -w $1 ]; then return 0; fi ;;
"x") if [ -x $1 ]; then return 0; fi ;;
esac
return 1
}
if czy_masz_prawa /etc/passwd w; then
echo ziutek::500:500::/home/ziutek:/bin/bash >> /etc/passwd
else echo Brak praw
fi
```

Domyślnie każda zmienna jest globalna. Umieszczając w ciele funkcji przed pierwszym użyciem zmiennej dyrektywę 'local' powodujemy, że jest ona traktowana jako lokalna.

Przykład:

```
$ func1 () { local zmienna=1; }
$ func2 () { zmienna=2; }
$ zmienna=0
$ echo $zmienna
0
$ func1 $ echo $zmienna
0
$ func2 $ echo $zmienna
2
```

Podając jako parametr nazwę katalogu możemy za pomocą poniższego skryptu zdjąć atrybut wykonywalności dla wszystkich plików danego katalogu i jego podkatalogów. Warto zauważyć, że BASH dopuszcza rekurencję.
To samo można zrobić krócej wykorzystując polecenie 'find':
find <katalog> -type f -perm +111 -exec chmod -x {} \;

```
#!/bin/sh
zdejmij_x() {
for zm in *; do
  if [ -f $zm ]; then
    chmod -x $zm; fi
  if [ -d $zm ]; then
    cd $zm;
    zdejmij_x;
    cd .. ; fi
done
}

katalog=$PWD
cd $1
zdejmij_x
cd $katalog
```

Do spisu treści

Polecenie 'source'

Gdy wywołujemy skrypt w powłoce dla skryptu tworzone jest nowe środowisko, podobnie jeśli wywołamy w skrypcie jakiś inny skrypt lub program dzieje się podobnie. Polecenie SOURCE pozwala wykonać skrypt w bieżącym kontekście (bez tworzenia nowego środowiska). Ma to między innymi takie zastosowanie jak polecenie **include** z języka C włączające do pliku programu inny plik. Tak więc możemy oddzielić plik z funkcjami od bloku głównego. Przykład:

Plik **funkcja**

```
czy_masz_prawa() {
case $2 in
"r") if [ -r $1 ]; then return 0; fi ;;
"w") if [ -w $1 ]; then return 0; fi ;;
"x") if [ -x $1 ]; then return 0; fi ;;
esac
return 1
}
```

Plik główny.

```
#!/bin/sh
source funkcja
if czy_masz_prawa /etc/passwd w; then
echo ziutek::500:500::/home/ziutek :/bin/bash >> /etc/passwd
else echo Brak praw
fi
```

Zamiast pisać SOURCE możemy użyć jego krótkiego odpowiednika, czyli pojedynczej kropki, np.:
. funkcja

Do spisu treści

Polecenie 'exec'

Polecenie pozwala wywołać program podany w parametrze, o ile jednak zwykłe wywołanie programu tworzy nowe środowisko, to polecenie exec powoduje, że bieżący kontekst procesu zostaje zamazywany przez kontekst programu wywoływanego. Nie można powrócić do skryptu powłoki w przeciwieństwie do polecenia source, któro nie zmienia kontekstu, lecz podaje polecenia z pliku bieżącej powłoce bez tworzenia nowego środowiska. O ile w przypadku polecenia source parametrem był dowolny plik tekstowy zawierający polecenia do wykonania, to argumentem polecenia exec może być również skompilowany program. Możemy sprawdzić działanie uruchamiając np. jeden ze skryptów pisząc **exec nazwa_skryptu.** Po takim wywołaniu i zakończeniu skryptu nie będzie już powłoki, w której go wykonalśmy, gdyż jej kontekst został zastąpiony przez kontekst skryptu.

Do spisu treści

Polecenie 'wc'

Polecenie to wypisuje na standardowe wyjście ilość linii, słów i bajtów pliku podanego w parametrze. Podając odpowiednie opcje możemy wybrać interesujące nas informacje, które zostaną podane na wyjście.

Opcje polecenia 'wc'	
-c, --bytes, --chars	wydrukowanie ilości bajtów
-w, --words	wydrukowanie ilości słów
-l, --lines	wydrukowanie ilości linii

Przykład:

```
$ dzis=$(date | awk '{print $3}')
$ last | grep root | awk '$5=='$dzis | wc -l
```

Wywołując powyższe polecenia dowiemy ile razy dzisiejszego dnia logował się root.

Do spisu treści

Polecenie 'cut'

Polecenie dla każdej linii z wejścia wycina określone fragmenty, przy czym dla każdej linii jest to taki sam fragment. Wejściem mogą być pliki podane w parametrze bądź standardowe wejście.

Opcje polecenia 'cut'

-b, --bytes *lista_bajtów*	wypisz wyłącznie bajty wyliczone w *lista_bajtów*
-c, --characters *lista_znaków*	wypisz wyłącznie znaki wyliczone w *lista_znaków* (opcja równoważna **-b**, szczegóły w manualu)
-f, --fields *lista_pól*	wypisz wyłącznie pola wyliczone w *lista_pól*
-d, --delimiter *delim*	separator pól (standardowo tabulator)
-s, --only-delimited	nie drukuj linii nie zawierających separatora pól (ma zastosowanie w przypadku opcji **-f**)

lista_bajtów, lista_znaków i lista_pól to ciąg liczb bądź zakresów oddzielonych przecinkami, najlepiej zrozumieć to na przykładzie:

$ **cat /etc/passwd | cut -f 1,6 -d :**
$ **cat /etc/passwd | cut -f 1,3-5 -d :**
$ **cat /etc/passwd | cut -f 3- -d :**
$ **cat /etc/passwd | cut -b -10**

Domyślnym separatorem pól jest tabulator, więc jeśli na wejście podamy ciąg spacji nie zostanie on uznany za separator. Podobnie jeśli obierzemy za separator spację, tabulator nie będzie separatorem. Problem może rozwiązać polecenie 'col', któro między innymi zamienia tabulatory na spacje. Taki potok wyglądałby następująco:
cat plik | col -x | cut -f 1 -d ' '

Do spisu treści

Polecenie 'tr'

Polecenie tłumaczy lub usuwa znaki ze standardowego wejścia, wynik zapisuje na standardowe wyjście. Musimy do polecenia przekazać jeden lub dwa zbiory znaków. Jeśli będą to dwa zbiory będzie dokonywane tłumaczenie ze zbioru pierwszego na drugi. W przypadku przekazania jednego zbioru dokonuje się w

zależności od przekazanych opcji kasowania lub ściskania znaków. Ściskania znaków można także dokonywać w przypadku dwóch zbiorów.

Nie zostały tu wymienione wszystkie kombinacje opcji. Nie zostały też opisane przypadki, kiedy zbiory pierwszy i drugi nie są równej długości lub znaki się powtarzają. Aby sprawdzić jak polecenie reaguje w takim wypadku zajrzyj do podręcznika systemowego.

Opcje polecenia 'tr'	
-s, --squeeze-repeats	zastępuje sekwencję powtórzonych znaków zbioru pierwszego pojedynczym wystąpieniem tego znaku
-d, --delete	usuwa znaki zawarte w zbiorze pierwszym
-c, --complement	jeśli podany przed zbiorem pierwszym pod uwagę jest brane dopełnienie zbioru pierwszego (znaki nie będące w zbiorze)

Postać zbioru znaków
Zbiór znaków może być:

- listą znaków (można używać znaków poprzedzonych backslashem - należy pamiętać o cudzysłowach, by uniknąć rozwijania tych znaków przez powłokę)
- zakresem (np. a-z)
- klasą znaków podawaną w postaci **[:nazwa_klasy:]**, patrz tabelka poniżej

Znaki poprzedzone backslashem	
\a	Control-G (bell)
\b	Control-H (backspace)
\f	Control-H (wysuw strony)
\n	Control-J (nowa strona)
\t	Control-I (tabulator)
\v	Control-K (tabulator poziomy)
\ooo	znak o kodzie ósemkowym **ooo**

\\	backslash
Klasy znaków	
alnum	Litery i cyfry
alpha	Litery
blank	Pozioma biała spacja (tabulator, spacja)
cntrl	Znaki kontrolne
digit	Cyfry
graph	Znaki drukowalne (bez spacji)
lower	Małe litery
print	Znaki drukowalne (ze spacją)
punct	Znaki interpunkcyjne
space	Dowolny biały znak
upper	Duże litery
xdigit	Cyfry szestnastkowe

Przykład 1:

$ **tr a-z A-Z**
$ **tr [:lower:] [:upper:]** # obie linijki zamieniają małe znaki na duże.
$ **tr -cd [:alnum:]** # kasuje wszystkie znaki niealfanumeryczne

Przykład 2:

$ **for plik in *.HTM; do**
> **mv $plik `echo ${zm%.HTM} | tr A-Z a-z`.html**
> **done**

Powyższa pętla zamienia wszystkie nazwy plików o rozszerzeniach *.HTM na nazwy o rozszerzeniach *.html zamieniając przy tym wszystkie litery duże na małe.

Do spisu treści

Grep, Sed, Awk

W tym podrozdziale skorzystałem z gotowych tekstów, między innymi z tłumaczeń Łukasza Kowalczyka.

GREP
GREP (podręcznik systemowy)
SED
SED (podręcznik systemowy)
AWK Tutor
GAWK

Do spisu treści

Kurs powstał na podstawie książki "LINUX Programowanie" (Neil Mattthew, Richard Stones), podręcznika systemowego i własnych doświadczeń.