

Hello World!

Utwórz plik, w którym umieszczony zostanie pierwszy skrypt:

```
touch skrypt
```

Następnie za pomocą dowolnego edytora ASCII (mcedit, vi, gedit, kate itp.) wpisz do niego następującą zawartość:

```
#!/bin/bash

#To jest komentarz.

echo "Hello world"
```

Znak # (hasz) oznacza komentarz, wszystko co znajduje się za nim w tej samej linii, jest pomijane przez interpreter. Pierwsza linia skryptu zaczynająca się od znaków: **#!** ma szczególne znaczenie - wskazuje na rodzaj shella w jakim skrypt ma być wykonany, tutaj skrypt zawsze będzie wykonywany przez interpreter poleceń **/bin/bash**, niezależnie od tego jakiego rodzaju powłoki w danej chwili używamy.

```
echo "Hello World"
```

Wydrukuj na standardowym wyjściu (stdout) czyli na ekranie napis: **Hello World**.

Aby móc uruchomić skrypt należy mu jeszcze nadać atrybut wykonywalności za pomocą polecenia:

```
chmod +x skrypt
```

Jeśli katalog bieżący w którym znajduje się skrypt nie jest dopisany do zmiennej **PATH**, to nasz skrypt możemy uruchomić w ten sposób:

```
./naszskrypt
```

podobnie można podać również pełną ścieżkę:

```
/home/user/skrypt
```

Polecenie echo

Polecenie echo

Polecenie echo służy do wydrukowania na standardowym wyjściu (**stdout** - domyślnie jest to ekran) napisu.

Składnia:

```
#!/bin/bash
echo -ne "jakiś napis"
echo "jakiś napis"      #wydrukuj tekst na ekranie
```

Można też pisać do pliku. W tym wypadku **echo** wydrukuj tekst do pliku, ale zmaże całą jego wcześniejszą zawartość, jeśli plik podany na standardowym wyjściu nie istnieje, zostanie utworzony.

```
echo "jakiś napis" > plik
```

Tutaj napis zostanie dopisany na końcu pliku, nie zmaże jego wcześniejszej zawartości.

```
echo "jakiś napis" >> plik
```

Parametry:

- **-n** nie jest wysyłany znak nowej linii
- **-e** włącza interpretację znaków specjalnych takich jak:
 - **\a** czyli alert, usłyszysz dzwonek
 - **\b** backspace
 - **\c** pomija znak kończący nową linię
 - **\f** escape
 - **\n** form feed czyli wysuw strony
 - **\r** znak nowej linii

- \t tabulacja pozioma
- \v tabulacja pionowa
- \\ backslash
- \nnn znak, którego kod ASCII ma wartość ósemkowo
- \xxxx znak, którego kod ASCII ma wartość szesnastkowo

Słowa zastrzeżone

Słowa zastrzeżone (ang. reserved words)

Mają dla powłoki specjalne znaczenie, wtedy gdy nie są cytowane.

Lista słów zastrzeżonych:

- !
- case
- do
- done
- elif
- else
- esac
- fi
- for
- function
- if
- in
- select
- then
- until
- while
- {
- }
- time
- [
-]

Cytowanie

Cytowanie

Znaki cytowania służą do usuwania interpretacji znaków specjalnych przez powłokę.

Wyróżnia się następujące znaki cytowania:

- **cudzysłów** (ang. *double quote*)

" "

Między cudzysłowami umieszcza się tekst, wartości zmiennych zawierające spacje. Cudzysłowy zachowują znaczenie specjalne trzech znaków:

- \$ wskazuje na nazwę zmiennej, umożliwiając podstawienie jej wartości
- \ znak maskujący
- `` odwrotny apostrof, umożliwia zacytowanie polecenia

Przykład:

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

```
x=2
echo "Wartość zmiennej x to $x"      #wydrukuj Wartość zmiennej x to 2
echo -ne "Usłyszysz dzwonek\a"
echo "Polecenie date pokaże: `date`"
```

- **apostrof** (ang. *single quote*)

' '

Wszystko co ujęte w znaki apostrofu traktowane jest jak łańcuch tekstowy, apostrof wyłącza interpretowanie wszystkich znaków specjalnych, traktowane są jak zwykłe znaki.

Przykład:

```
#!/bin/bash
echo '$USER'    #nie wypisze twojego loginu
```

- **odwrotny apostrof** (ang. *backquote*)

` `

umożliwia zacytowanie polecenia, bardzo przydatne jeśli chce się podstawić pod zmienną wynik jakiegoś polecenia np:

Przykład:

```
#!/bin/bash
x=`ls -la $PWD`
echo $x                #pokaże rezultat polecenia
```

Alternatywny zapis, który ma takie samo działanie wygląda tak:

```
#!/bin/bash
echo $(ls -la $PWD)
```

- **backslash** czyli znak maskujący

\

Jego działanie najlepiej wyjaśnić na przykładzie: w celu by na ekranie pojawił się napis \$HOME

Przykład:

```
echo "$HOME"          #wydrukuj /home/ja
```

aby wyłączyć interpretację przez powłokę tej zmiennej, należy wpisać:

```
echo \$HOME           #i jest napis $HOME
```

Zmienne programowe

Zmienne programowe (ang. *program variables*)

To zmienne definiowane samodzielnie przez użytkownika.

```
nazwa_zmiennej="wartość"
```

Na przykład:

```
x="napis"
```

Do zmiennej odwołujemy się poprzez podanie jej nazwy poprzedzonej znakiem \$ i tak dla zmiennej x może to wyglądać następująco:

```
echo $x
```

Na co należy uważać? **Nie może być spacji po obu stronach!**

```
x = "napis"
```

ten zapis jest błędny

Pod zmienną możemy podstawić wynik jakiegoś polecenia, można to zrobić na dwa sposoby:

- Poprzez użycie odwrotnych apostrofów:

```
`polecenie`
```

Przykład:

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

```
GDZIE_JESTEM=`pwd`  
echo "Jestem w katalogu $GDZIE_JESTEM"
```

Wartością zmiennej **GDZIE_JESTEM** jest wynik działania polecenia **pwd**, które wypisze nazwę katalogu w jakim się w danej chwili znajdujemy.

- Za pomocą rozwijania zawartości nawiasów:

```
$(polecenie)
```

Przykład:

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

```
GDZIE_JESTEM=$(pwd)  
echo "Jestem w katalogu $GDZIE_JESTEM"
```

Zmienne specjalne

Zmienne specjalne (*ang. special variables, special parameters*)

To najbardziej prywatne zmienne powłoki, są udostępniane użytkownikowi tylko do odczytu (są wyjątki). Kilka przykładów:

- **\$0**

nazwa bieżącego skryptu lub powłoki

Przykład:

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

```
echo "$0"
```

Pokaże nazwę uruchomionego skryptu.

- **\$1..\$9**

Parametry przekazywane do skryptu (wyjątek, użytkownik może modyfikować ten rodzaj \$-ych specjalnych).

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

```
echo "$1 $2 $3"
```

Jeśli wywołany zostanie skrypt z jakimiś parametrami to przypisane zostaną zmiennym: od **\$1** do **\$9**. Zobacz co się stanie jak podasz za małą liczbę parametrów oraz jaki będzie wynik podania za dużej liczby parametrów.

- **\$@**

Pokaże wszystkie parametry przekazywane do skryptu (też wyjątek), równoważne **\$1 \$2 \$3...**, jeśli nie podane są żadne parametry **\$@** interpretowana jest jako nic.

Przykład:

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

```
echo "Skrypt uruchomiono z parametrami: $@"
```

A teraz wywołaj ten skrypt z jakimiś parametrami, mogą być brane z powietrza np.:

```
./plik -a d
```

Efekt będzie wyglądał następująco:

- Skrypt uruchomiono z parametrami -a d
- \$?

kod powrotu ostatnio wykonywanego polecenia

- \$\$

PID procesu bieżącej powłoki

Zmienne środowiskowe

Zmienne środowiskowe (ang. *environment variables*)

Definiują środowisko użytkownika, dostępne dla wszystkich procesów potomnych. Można je podzielić na:

- globalne - widoczne w każdym podshellu
- lokalne - widoczne tylko dla tego shella w którym został ustawione

Aby bardziej uzmysłowić sobie różnicę między nimi zrób mały eksperyment: otwórz xterm (widoczny podshell) i wpisz:

```
x="napis"
echo $x
xterm
```

x="napis" zdefiniowałeś właśnie zmienną **x**, która ma wartość "napis"
echo \$x wyświetli wartość zmiennej **x**
xterm wywołanie podshella

wpisz więc jeszcze raz:

echo \$x nie pokaże nic, bo zmienne lokalne nie są widoczne w podshellach

Możesz teraz zainicjować zmienną globalną:

```
export x="napis"
```

Teraz zmienna **x** będzie widoczna w podshellach, jak widać wyżej służy do tego polecenie **export**, nadaje ono wskazanym zmiennym atrybut zmiennych globalnych. W celu uzyskania listy aktualnie eksportowanych zmiennych należy wpisać **export**, opcjonalnie **export -p**. Na tej liście przed nazwą każdej zmiennej znajduje się zapis:

```
declare-x
```

To wewnętrzne polecenie BASH-a, służące do definiowania zmiennych i nadawania im atrybutów, **-x** to atrybut eksportu czyli jest to, to samo co polecenie **export**. Ale tu uwaga! Polecenie **declare** występuje tylko w BASH-u, nie ma go w innych powłokach, natomiast **export** występuje w **ksh**, **ash** i innych, które korzystają z plików startowych **/etc/profile**. Dlatego też zaleca się stosowanie polecenia **export**.

```
export -n zmienna
```

spowoduje usunięcie atrybutu eksportu dla danej zmiennej

Niektóre przykłady zmiennych środowiskowych:

```
$HOME          #ścieżka do twojego katalogu domowego
$USER          #twój login
$HOSTNAME      #nazwa twojego hosta
$OSTYPE        #rodzaj systemu operacyjnego
```

itp. dostępne zmienne środowiskowe można wyświetlić za pomocą polecenia:

```
printenv | more
```

Zmienne tablicowe

BASH pozwala na stosowanie zmiennych tablicowych jednowymiarowych. Czym jest tablica? To zmienna która przechowuje listę jakichś wartości (rozdzielonych spacjami), w BASH'u nie ma maksymalnego rozmiaru tablic. Kolejne wartości zmiennej tablicowej indexowane są przy pomocy liczb całkowitych, zaczynając od 0.

Składnia

```
zmienna=(wartość1 wartość2 wartość3 wartośn)
```

Przykład:

```
#!/bin/bash

tablica=(element1 element2 element3)

echo ${tablica[0]}
echo ${tablica[1]}
echo ${tablica[2]}
```

Zadeklarowana została zmienna tablicowa o nazwie: **tablica**, zawierająca trzy wartości: **element1 element2 element3**. Natomiast polecenie: **echo \${tablica[0]}** wydrukuje na ekranie pierwszy elementu tablicy. W powyższym przykładzie w ten sposób wypisana zostanie cała zawartość tablicy. Do elementów tablicy **odwołuje się** za pomocą **wskaźników**.

- **Odwołanie do elementów tablicy.**

Składnia:

```
${nazwa_zmiennej[wskaźnik]}
```

Wskaźnikami są indexy elementów tablicy, począwszy od 0 do n oraz @, *. Gdy odwołując się do zmiennej nie poda się wskaźnika: **\${nazwa_zmiennej}** to nastąpi odwołanie do elementu tablicy o indexie 0. Jeśli wskaźnikiem będą: @ lub * to zinterpretowane zostaną jako wszystkie elementy tablicy, w przypadku gdy tablica nie zawiera żadnych elementów to zapisy: **\${nazwa_zmiennej[wskaźnik]}** lub **\${nazwa_zmiennej[wskaźnik]}** są interpretowane jako nic.

Przykład:

Poniższy skrypt robi to samo co wcześniejszy.

```
#!/bin/bash

tablica=(element1 element2 element3)

echo ${tablica[*]}
```

Można też uzyskać długość (liczba znaków) danego elementu tablicy:

```
${#nazwa_zmiennej[wskaźnik]}
```

Przykład:

```
#!/bin/bash

tablica=(element1 element2 element3)

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

Polecenie **echo \${#tablica[0]}** wydrukuje liczbę znaków z jakich składa się pierwszy element tablicy: **element1** wynik to 8. W podobny sposób można otrzymać liczbę wszystkich elementów tablicy, wystarczy jako wskaźnik podać: @ lub *.

Przykład:

```
#!/bin/bash

tablica=(element1 element2 element3)
```

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

Co da wynik: **3**.

- **Dodawanie elementów do tablicy.**

Składnia:

```
nazwa_zmiennej[wskaźnik]=wartość
```

Przykład:

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

```
tablica=(element1 element2 element3)
tablica[3]=element4
```

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

Jak wyżej widać do tablicy został dodany **element4** o indexie **3**. Mechanizm dodawania elementów do tablicy, można wykorzystać do tworzenia tablic, gdy nie istnieje zmienna tablicowa do której dodajemy jakiś element, to BASH automatycznie ją utworzy:

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

```
linux[0]=slackware
linux[1]=debian
```

```
echo ${linux[@]}
```

Utworzona została tablica **linux** zawierająca dwa elementy.

- **Usuwanie elementów tablic i całych tablic.**

Dany element tablicy usuwa się za pomocą polecenia **unset**.

Składnia:

```
unset nazwa_zmiennej[wskaźnik]
```

Przykład:

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

```
tablica=(element1 element2 element3)
echo ${tablica[@]}
unset tablica[2]
echo ${tablica[*]}
```

Usunięty został ostatni element tablicy.

Aby usunąć całą tablicę wystarczy podać jako wskaźnik: **@** lub *****.

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

```
tablica=(element1 element2 element3)
unset tablica[*]
```

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

Zmienna tablicowa o nazwie **tablica** przestała istnieć, polecenie: **echo \${tablica[@]}** nie wyświetli nic.

Strumienie danych

Strumienie danych

Każdy uruchomiony w Linuxie proces skądś pobiera dane, gdzieś wysyła wyniki swojego działania i komunikaty o błędach. Tak więc procesowi przypisane są trzy strumienie danych:

- **stdin** (ang. *standard input*) czyli **standardowe wejście**, skąd proces pobiera dane, domyślnie jest to **klawiatura**
- **stdout** (ang. *standard output*) to **standardowe wyjście**, gdzie wysyłany jest wynik działania procesu, domyślnie to **ekran**
- **stderr** (ang. *standard error*) **standardowe wyjście błędów**, tam trafiają wszystkie komunikaty o błędach, domyślnie **ekran**

Linux wszystko traktuje jako plik, niezależnie od tego czy to jest plik zwykły, katalog, urządzenie blokowe (klawiatura, ekran) itd. Nie inaczej jest ze strumieniami, **BASH** identyfikuje je za pomocą przyporządkowanych im liczb całkowitych (od **0** do **2**) tak zwanych **deskryptorów plików**.
I tak:

- **0** to plik z którego proces pobiera dane **stdin**
- **1** to plik do którego proces pisze wyniki swojego działania **stdout**
- **2** to plik do którego trafiają komunikaty o błędach **stderr**

Za pomocą **operatorów przypisania** można manipulować strumieniami, poprzez przypisanie deskryptorów: **0, 1, 2** innym plikom, niż tym reprezentującym klawiaturę i ekran.

- **Przełączanie standardowego wejścia**

Zamiast klawiatury jako standardowe wejście można otworzyć plik:

```
< plik
```

Przykład:

Najpierw stwórzmy plik **lista** o następującej zawartości:

```
slackware
redhat
debian
caldera
```

Użyjemy polecenia **sort** dla którego standardowym wejściem będzie nasz plik.

```
sort < lista
```

Wynikiem będzie wyświetlenie na ekranie posortowanej zawartość pliku **lista**:

```
caldera
debian
redhat
slackware
```

- **Przełączanie standardowego wyjścia**

Wynik jakiegoś polecenia można wysłać do pliku, a nie na ekran, do tego celu używa się operatora:

```
> plik
```

Przykład:

```
ls -la /usr/bin > ~/wynik
```

Rezultat działania polecenia **ls -la /usr/bin** trafi do pliku o nazwie **wynik**, jeśli wcześniej nie istniał plik o takiej samej nazwie, to zostanie utworzony, jeśli istniał cała jego poprzednia zawartość zostanie nadpisana.

Jeśli chcemy aby dane wyjściowe dopisywane były na końcu pliku, bez wymazywania jego wcześniejszej zawartości, stosujemy operator:

```
>> plik
```

Przykład:

```
free -m >> ~/wynik
```

Wynik polecenia **free -m** (pokazuje wykorzystanie pamięci RAM i swap'a) zostanie dopisany na końcu pliku **wynik**, nie naruszając jego wcześniejszej zawartości.

- **Przelączanie standardowego wyjścia błędów**

Do pliku można też kierować strumień diagnostyczny:

```
2> plik
```

Przykład:

```
#!/bin/bash

echo "Stderr jest skierowane do pliku error"
ls -y 2< ~/error    #błąd
```

W powyższym skrypcie polecenie **ls** jest użyte z błędną opcją **-y**, komunikat o błędzie trafi do pliku **error**.

Za pomocą operatora:

```
<< plik
```

można dopisać do tego samego pliku kilka komunikatów o błędach, dopisanie kolejnego nie spowoduje skasowania wcześniejszej zawartości pliku.

Przykład:

```
#!/bin/bash

echo "Stderr jest skierowane do pliku error"
ls -y 2> ~/error    #błąd
cat /etc/shadow 2> ~/error    #błąd2
```

Jako błąd drugi zostanie potraktowane polecenie **cat /etc/shadow** (zakładając, że zalogowałeś się jako **użytkownik**) ponieważ prawo **odczytu** pliku **/etc/shadow** ma tylko **root**.

Instrukcja if

Instrukcja warunkowa if

Sprawdza czy warunek jest prawdziwy, jeśli tak to wykonane zostanie polecenie lub polecenia znajdujące się po słowie kluczowym **then**. Instrukcja kończy się słowem **fi**.

Składnia:

```
if warunek
then
    polecenie
fi
```

Przykład:

```
#!/bin/bash
if [ -e ~/.bashrc ]
then
    echo "Masz plik .bashrc"
fi
```

Najpierw sprawdzany jest warunek: czy istnieje w twoim katalogu domowym plik **.bashrc**, zapis **~/** oznacza to samo co **/home/twój_login** lub **\$HOME**. Jeśli sprawdzany warunek jest prawdziwy to wyświetlony zostanie napis **Masz plik .bashrc**. W przeciwnym wypadku nic się nie stanie.

W sytuacji gdy test warunku zakończy się wynikiem negatywnym można wykonać inny zestaw poleceń, które umieszczamy po słowie kluczowym **else**:

Składnia:

```
if warunek
then
    polecenie1
```

```
else
    polecenie2
fi
```

Przykład:

```
#!/bin/bash
if [ -e ~/.bashrc ]
then
    echo "Masz plik.bashrc"
else
    echo "Nie masz pliku .bashrc"
fi
```

Jeśli warunek jest fałszywy skrypt poinformuje Cię o tym.

Można też testować dowolną ilość warunków, jeśli pierwszy warunek nie będzie prawdziwy, sprawdzony zostanie następny, kolejne testy warunków umieszczamy po słowie kluczowym **elif**.

Składnia:

```
if warunek
then
    polecenie1
elif warunek
then
    polecenie2
fi
```

Przykład:

```
#!/bin/bash
if [ -x /opt/kde/bin/startkde ]; then
    echo "Masz KDE w katalogu /opt"
elif [ -x /usr/bin/startkde ]; then
    echo "Masz KDE w katalogu /usr"
elif [ -x /usr/local/bin/startkde ]; then
    echo "Masz KDE w katalogu /usr/local"
else
    echo "Nie wiem gdzie masz KDE"
fi
```

Ten skrypt sprawdza gdzie masz zainstalowane **KDE**, sprawdzane są trzy warunki, najpierw czy plik wykonywalny **startkde** znajduje się w katalogu **/opt/kde/bin** jeśli go tam nie ma, szukany jest w **/usr/bin**, gdy i tu nie występuje sprawdzany jest katalog **/usr/local/bin**.

test

Jak się sprawdza warunki?

Służy do tego polecenie **test**. (**Uwaga!** Nie można skryptom nadawać nazwy **test**! Nie będą działać.)

Składnia:

```
test wyrażenie1 operator wyrażenie2
```

lub może być zapisane w postaci nawiasów kwadratowych:

```
[ wyrażenie1 operator wyrażenie2 ]
```

Uwaga! Między nawiasami a treścią warunku muszą być spacje, tak jak powyżej.

w przypadku sprawdzania warunków arytmetycznych:

```
(( wyrażenie1 operator wyrażenie2 ))
```

Polecenie **test** zwraca wartość 0 (true) jeśli warunek jest spełniony i wartość 1 (false) jeśli warunek nie jest spełniony. A gdzie jest umieszczana ta wartość? W zmiennej specjalnej **\$?**.

A to kilka przykładów operatorów polecenia **test**:

- **-a** operator **and**
- **-o** operator **or**
- **-b** plik istnieje i jest blokowym plikiem specjalnym
- **-** plik istnieje i jest plikiem znakowym
- **-e** plik istnieje
- **-h** plik istnieje i jest linkiem symbolicznym
- **=** sprawdza czy wyrażenia są równe
- **!=** sprawdza czy wyrażenia są różne
- **-n** wyrażenie ma długość większą niż 0
- **-d** wyrażenie istnieje i jest katalogiem
- **-z** wyrażenie ma zerową długość
- **-r** można czytać plik
- **-w** można zapisywać do pliku
- **-x** można plik wykonać
- **-f** plik istnieje i jest plikiem zwykłym
- **-p** plik jest łączem nazwanym
- **-N** plik istnieje i był zmieniany od czasu jego ostatniego odczytu
- **plik1 -nt plik2** plik1 jest nowszy od pliku2
- **plik1 -ot plik2** plik1 jest starszy od pliku2
- **-lt** mniejsze niż
- **-gt** większe niż
- **-ge** większe lub równe
- **-le** mniejsze lub równe

Więcej przykładów operatorów w: **man bash**.

Instrukcja case

Instrukcja case

Pozwala na dokonanie wyboru spośród kilku wzorców. Najpierw sprawdzana jest wartość zmiennej po słowie kluczowym **case** i porównywana ze wszystkimi wariantami po kolei. Oczywiście musi być taka sama jak wzorec do którego chcemy się odwołać. Jeśli dopasowanie zakończy się sukcesem wykonane zostanie polecenie lub polecenia przypisane do danego wzorca. W przeciwnym wypadku użyte zostanie polecenie domyślne oznaczone symbolem gwiazdki: ***) polecenie_domyślne**. Co jest dobrym zabezpieczeniem na wypadek błędów popełnionych przez użytkownika naszego skryptu.

Składnia:

```
case zmienna in
    "wzorzec1") polecenie1 ;;
    "wzorzec2") polecenie2 ;;
    "wzorzec3") polecenie3 ;;
    *) polecenie_domyślne
esac
```

Przykład:

```
#!/bin/bash
echo "Podaj cyfrę dnia tygodnia"
read d
case "$d" in
    "1") echo "Poniedziałek" ;;
    "2") echo "Wtorek" ;;
    "3") echo "Środa" ;;
    "4") echo "Czwartek" ;;
    "5") echo "Piątek" ;;
    "6") echo "Sobota" ;;
    "7") echo "Niedziela" ;;
    *) echo "Nic nie wybrałeś"
esac
```

Przy zastosowaniu **;;** po przypasowaniu do warunku i wykonaniu instrukcji wychodzi z konstrukcji **case**. Przy zastosowaniu **&&** wykonuje instrukcje także kolejnego warunku. Jak widać mamy w skrypcie wzorce od **1** do **7** odpowiadające liczbie dni tygodnia, każdemu przypisane jest jakieś polecenie, tutaj ma wydrukować na ekranie nazwę dnia tygodnia. Jeśli podamy 1 polecenie **read** czytające dane ze standardowego wejścia przypisze zmiennej **d** wartość 1 i zostanie wykonany skok do wzorca **1**, na ekranie zostanie wyświetlony napis *Poniedziałek*. W przypadku gdy podamy cyfrę o liczbie

większej niż 7 lub wpisujemy inny znak na przykład literę to wykonany zostanie wariant defaultowy oznaczony gwiazdką:

*) echo "Nic nie wybrałeś".

Pętla for

Pętla for

Wykonuje polecenia zawarte wewnątrz pętli, na każdym składniku listy (iteracja).

Składnia:

```
for zmienna in lista
do
    polecenie
done
```

Przykład:

```
for x in jeden dwa trzy
do
    echo "To jest $x"
done
```

Zmiennej **x** przypisana jest lista, która składa się z trzech elementów: **jeden, dwa, trzy**. Wartością zmiennej **x** staje się po kolei każdy element listy, na wszystkich wykonywane jest polecenie: **echo "To jest \$x"**. Pętla **for** jest bardzo przydatna w sytuacjach, gdy chcemy wykonać jakąś operację na wszystkich plikach w danym katalogu. Na przykład chcemy uzyskać listę wszystkich plików o danym rozszerzeniu znajdujących się w jakimś katalogu, robimy to tak:

```
#!/bin/bash
for x in *.html
do
    echo "To jest plik $x"
done
```

lub jeśli chcemy zmienić nazwy plików pisane *DUŻYMI* literami na nazwy pisane *małymi* literami:

```
#!/bin/bash
for nazwa in *
do
    mv $nazwa `echo $nazwa | tr 'A-Z' 'a-z'`
done
```

Za zmianę *DUŻYCH* liter na *male* (i na odwrót) odpowiedzialne jest polecenie **tr**.

Pętla select

Pętla select

Wygeneruje z listy słów po **in** proste ponumerowane menu, każdej pozycji odpowiada kolejna liczba od 1 wzwyż. Poniżej menu znajduje się znak zachęty PS3 gdzie wpisujemy cyfrę odpowiadającą wybranej przez nas pozycji w menu. Jeśli nic nie wpisujemy i wciśniemy ENTER, menu będzie wyświetlone ponownie. To co wpisaliśmy zachowywane jest w zmiennej **REPLY**. Gdy odczytane zostaje **EOF** (ang. *End Of File*) czyli znak końca pliku (**CTRL+D**) to **select** kończy pracę. Pętla działa dotąd dopóki nie wykonane zostaje polecenie **break** lub **return**.

Składnia:

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

Praktyczny przykład:

```
#!/bin/bash
echo "Co wybierasz?"
select y in X Y Z Quit
do
```

```

case $y in
  "X") echo "Wybrałeś X" ;;
  "Y") echo "Wybrałeś Y" ;;
  "Z") echo "Wybrałeś Z" ;;
  "Quit") exit ;;
  *) echo "Nic nie wybrałeś"
esac
break
done

```

Najpierw zobaczymy proste ponumerowane menu, składające się z czterech elementów: **X**, **Y**, **Z** i **Quit**, teraz wystarczy tylko wpisać numer inetersującej nas opcji, a resztę zrobi instrukcja case. Polecenie **break**, które znajduje się w przedostatniej linii skryptu, kończy pracę pętli. Słowo kluczowe **continue** - przerywa wykonywanie instrukcji w ciele pętli i przechodzi do następnej iteracji.

Pętla while

Pętla while

Najpierw sprawdza warunek czy jest prawdziwy, jeśli tak to wykonane zostanie polecenie lub lista poleceń zawartych wewnątrz pętli, gdy warunek stanie się fałszywy pętla zostanie zakończona.

Składnia:

```

while warunek
do
polecenie
done

```

Przykład:

```

#!/bin/bash
x=1;
while [ $x -le 10 ]; do
echo "Napis pojawił się po raz: $x"
x=$((x + 1))
done

```

Sprawdzany jest warunek czy zmienna **x** o wartości początkowej 1 jest mniejsza lub równa 10, warunek jest prawdziwy w związku z czym wykonywane są polecenia zawarte wewnątrz pętli: **echo "Napis pojawił się po raz: \$x"** oraz **x=\$((x + 1))**, które zwiększa wartość zmiennej **x** o 1. Gdy wartość **x** przekroczy 10, wykonanie pętli zostanie przerwane.

Pętla until

Pętla until

Sprawdza czy warunek jest prawdziwy, gdy jest fałszywy wykonywane jest polecenie lub lista poleceń zawartych wewnątrz pętli, między słowami kluczowymi **do** a **done**. Pętla **until** kończy swoje działanie w momencie gdy warunek stanie się prawdziwy.

Składnia:

```

until warunek
do
polecenie
done

```

Przykład:

```

#!/bin/bash
x=1;
until [ $x -ge 10 ]; do
echo "Napis pojawił się po raz: $x"
x=$((x + 1))
done

```

Mamy zmienną **x**, która przyjmuje wartość 1, następnie sprawdzany jest warunek czy wartość zmiennej **x** jest większa lub równa 10, jeśli nie to wykonywane są polecenia zawarte wewnątrz pętli. W momencie gdy

zmienna **x** osiągnie wartość, 10 pętla zostanie zakończona.

Inne

exit - Zakończ wykonywanie skryptu. Jako argument przyjmuje stan zakończenia. Domyślnie status 0.

Składnia:

```
exit 10
```

Polecenie read

Polecenie read

Czyta ze standardowego wejścia pojedynczy wiersz.

Składnia:

```
read -opcje nazwa_zmiennej
```

Przykład:

```
#!/bin/bash
echo -n "Wpisz coś:\a"

read wpis
echo "$wpis"
```

To co zostało wpisane trafi do zmiennej **wpis**, której to wartość czyta polecenie **read wpis**, zmienna nie musi być wcześniej tworzona, jeśli istniała wcześniej, jej zawartość zostanie zastąpiona tym co wpisaliśmy.

Przykład:

```
#!/bin/bash
echo "Wpisz coś:"

answer="napis"
read
echo "$answer"
```

Wcześniejsza wartość zmiennej **answer** została zastąpiona.

Polecenie **read** pozwala na przypisanie kilku wartości kilku zmiennym.

Przykład:

```
#!/bin/bash
echo "Wpisz cztery wartości:"

read a b c
echo "Wartość zmiennej a to: $a"
echo "Wartość zmiennej b to: $b"
echo "Wartość zmiennej c to: $c"
```

Nie przypadkiem w powyższym przykładzie pojawiło się polecenie wpisania czterech wartości, pierwsza wartość trafi do zmiennej **a**, druga do zmiennej **b**, natomiast trzecia i czwarta oraz rozdzielające je znaki separacji przypisane zostaną zmiennej **c**.

Wybrane opcje:

- **-p**

Pokaże znak zachęty bez kończącego znaku nowej linii.

```
#!/bin/bash

read -p "Pisz:" odp
echo "$odp"
```

- **-a**

Kolejne wartości przypisywane są do kolejnych indeksów zmiennej tablicowej.

Przykład:

```
#!/bin/bash
echo "Podaj elementy zmiennej tablicowej:"

read tablica
echo "${tablica[*]}"
```

- **-e**

Jeśli nie podano żadnej nazwy zmiennej, wiersz trafia do **\$REPLY**.

Przykład:

```
#!/bin/bash
echo "Wpisz coś:"

read -e
echo "$REPLY"
```

- **-t timeout**

czas wygaśnięcia w sekundach

- **-s**

nie wyświetlaj znaków wpisanych przez użytkownika.

```
#!/bin/bash

#Hasło wpisywane bez echa, max. przez 30 sekund
read -p "Password: " -s -t 30 password
echo $password
```

Funkcje

Funkcje

Coś w rodzaju podprogramów. Stosuje się je gdy w naszym skrypcie powtarza się jakaś grupa poleceń, po co pisać je kilka razy, skoro można to wszystko umieścić w funkcjach. Do danej funkcji odwołujemy się podając jej nazwę, a wykonane zostanie wszystko co wpisaliśmy między nawiasy { }, skraca to znacznie długość skryptu.

Składnia:

```
function nazwa_funkcji
{
polecenie1
polecenie2
polecenie3
}
```

lub:

```
function nazwa_funkcji()
{
polecenie1
polecenie2
polecenie3
}
```

Przykład:

```
#!/bin/bash

function napis
{
echo "To jest napis"
}
```

`napis`

Nazwę funkcji umieszczamy po słowie kluczowym **function**, w powyższym przykładzie mamy funkcję o nazwie **napis**, odwołujemy się do niej podając jej nazwę, wykonane zostaną wtedy wszystkie polecenia, jakie jej przypiszemy.

Funkcje mogą się znajdować w innym pliku, co uczyni nasz skrypt bardziej przejrzystym i wygodnym, tworzy się własne pliki nagłówkowe, wywołuje się je tak:

```
. ~/naszplik_z_funkcjami
nazwa_funkcji
```

Trzeba pamiętać o podaniu kropki + spacja przed nazwą pliku

Przykład:

```
#!/bin/bash
function nasza_funkcja
{
echo -e 'Właśnie użyłeś funkcji o nazwie "nasza_funkcja".\a'
}
```

Teraz pozostało jeszcze utworzyć skrypt w którym wywołamy funkcję: **nasza_funkcja**:

```
#!/bin/bash
echo "Test funkcji."
. funkcja
nasza_funkcja
```

Przekazywanie parametrów

Przekazanie parametrów do funkcji następuje dokładnie tak samo jak do każdego polecenia które jest w skrypcie:

```
nazwa_funkcji parametr_1 parametr_2
```

Przykład:

```
#!/bin/bash

funkcja_z_parametrami()
{
    echo "Przekazano $# parametrów"
    echo "Parametr $1"
    echo "Parametr $2"
}

funkcja_z_parametrami "param1" "param2"
```

Zmienna specjalna \$0 przechowująca nazwę skryptu nie jest dostępna!!! Choć na pierwszy rzut oka powinna przechowywać nazwę funkcji.

Obliczanie wyrażeń arytmetycznych

Interpretacja wyrażeń arytmetycznych.

Kiedy zachodzi potrzeba przeprowadzenia jakichś obliczeń można skorzystać z mechanizmu interpretacji wyrażeń arytmetycznych, obliczenia dokonywane są na liczbach całkowitych, nie przeprowadzana jest kontrola przepełnienia (ang. *overflow*).

Składnia:

```
$((wyrażenie)) lub ${wyrażenie}
```

Przykład:

```
#!/bin/bash

echo $((8/2))
```

```
wynik=$((4*5/2))
echo "$wynik"
```

W ten sposób (przykład poniżej) można ponumerować listę:

```
#!/bin/bash

for pliki_html in $(ls *.html)
do
    numer=$((numer+1))
    echo "$numer. "
    echo $pliki_html
done
```

Wynikiem będzie ponumerowana lista wszystkich plików o rozszerzeniu **.html**, znajdujących się w bieżącym katalogu.

Polecenie let

Do przeprowadzenia obliczeń można też skorzystać z polecenia **let**.

Przykład:

```
#!/bin/bash

liczba1=5
liczba2=6

let wynik=liczba1*liczba2
echo $wynik
```

-