Systemy operacyjne – studia stacjonarne 2022/23
Lab 4. Powłoka Bash - zmienne, aliasy, pliki konfiguracyjne, język skryptowy powłoki.

1. Powłoka bash - zmienne lokalne i środowiskowe, komentarze.

Pracując z daną powłoką można wydawać polecenia, deklarować zmienne, tworzyć i uruchamiać skrypty. Zmienne zdefiniowane w danej powłoce są dla niej lokalne i nie można się do nich odwoływać w innych powłokach. Można jednak zdefiniować zmienne środowiskowe, które są dostępne w każdej nowej powłoce. Sposób ich przekazywania jest analogiczny do przekazywania parametrów przez wartość – nowa powłoka dysponuje kopiami zmiennych środowiskowych. Zmienne środowiskowe są zwykle używane do podawania globalnych ustawień systemu (ścieżka dostępu do poleceń, postać znaku zachęty). Nazwa zmiennej może się składać z dowolnych liter, cyfr oraz ze znaku "_", nie może jednak rozpoczynać się cyfrą.

Lista słów kluczowych /zastrzeżonych/ (mają specjalne znaczenie dla powłoki):

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

Zmienne lokalne można definiować poleceniem: **zmienna=wartosc** (obok operatora przypisania nie mogą znajdować się spacje), a należy się do nich odwoływać poprzez symbol **$zmienna**. Aby usunąć zmienną, stosujemy polecenie **unset zmienna**. Listę wszystkich zdefiniowanych zmiennych można wyświetlić poleceniem **set**.

Cytowania - wartości przypisywane zmiennym mogą zawierać znaki, które mają szczególne znaczenie – jeżeli są cytowane traktowane są jak zwykłe znaki:

"…. " – maskuje znaki specjalne (spacja, *, ?, [,], ., <, >, &, |),

'…. ' – działają jak podwójne cudzysłowy, dodatkowo maskują znak $,

`…..` – wymuszają wykonanie polecenia,

\ – maskuje jeden znak.

Zmienne lokalne powłoki można wyświetlić poleceniem **set**. Aby utworzyć zmienną środowiskową ze zdefiniowanej zmiennej, należy użyć polecenia **export zmienna**. Listę zdefiniowanych zmiennych środowiskowych można wyświetlić poleceniem **env** lub **export.**

Warunkowe ustawienie zmiennej – polecenie **:** - puste-true

: ${zm:=5} – jeżeli zmienna zm nie została zdefiniowana wcześniej przyjmie wartość 5.

Do definicji zmiennej pustej (NULL) używamy poleceń: zmienna= lub zmienna=""

2. Zmienne środowiskowe konfiguracyjne.

Linux definiuje specjalne zmienne powłoki, które są używane do konfigurowania powłoki. Ich nazwy są zarezerwowanymi słowami kluczowymi. Listę zdefiniowanych zmiennych specjalnych można wyświetlić poleceniem **env**. Najważniejsze z nich to:

- **HOME** – ścieżka katalogu macierzystego,
- **USERNAME** – nazwa użytkownika,
- **SHELL** – ścieżka powłoki logowania,
- **PATH** – lista katalogów rozdzielonych dwukropkami, które są przeszukiwane w celu znalezienia polecenia do uruchomienia,

PS1 – monit podstawowy; może zawierać następujące kody:
\! – nr z listy historii,
\d – data,
\s – bieżąca powłoka,
\t – czas,
\u – nazwa użytkownika,
\w – bieżący katalog,

PS2 – monit wtórny (dla drugiego i kolejnych wierszy polecenia),

CDPATH – lista katalogów przeszukiwanych podczas wykonywania komendy cd,

TERM – nazwa terminala,

HISTSIZE – ilość zdarzeń historii,

HISTFILE – nazwa pliku historii zdarzeń,

HISTFILESIZE – rozmiar pliku historii,

MAIL – ścieżka do pliku skrzynki pocztowej,

MAILCHECK – określa odstęp czasu, po jakim użytkownik jest powiadamiany o nowej poczcie,

MAILPATH – nazwy ścieżek plików skrzynek pocztowych.

3. „Przełączniki”.

BASH używa **kilku** dodatkowych zmiennych określających właściwości powłoki, będącymi opcjami, które mogą być włączone lub wyłączone. Są to: **ignoreeof**, **noclobber**, **noglob**. Ustawienie zmiennej dokonuje się poleceniem:

set –o zmienna
set +o zmienna

Znaczenie poszczególnych zmiennych jest następujące:

ignoreeof – blokuje możliwość wylogowania się za pomocą [CTRL+d] (domyślnie do 10 razy). Można nadać jej dowolną wartość (np. ignoreeof=30),

noclobber – zapobiega nadpisaniu plików podczas przekierowania (>),

noglob – blokuje rozwijanie znaków specjalnych.

4. Zmienne specjalne podstawowe - parametryczne.

Zmienne używane w skryptach np.: $0, $1, $2, $#, $$ itd. Opis i zastosowanie patrz punkt. Skrypty.

Ćwiczenia 1:

Zdefiniuj zmienne, a następnie sprawdź ich wartości:

1. `a=cos`
   `echo $a`
   `set`
2. `zm1=”wartosc zm. A wynosi $a”`
   `echo $zm1`
   `( ” ” – nie maskują znaku $)`
3. `zm2=’wartosc zm a wynosi $a’`
   `echo $zm2`
   (‘ ‘ – maskują znak $)
4. Nadaj zmiennej wartość odpowiadającą nazwie katalogu i użyj jej w poleceniu kopiowania

```
katalog=~/proba
set
cp plik $katalog
```

5. Usuń zdefiniowaną zmienną i sprawdź, czy faktycznie jej już nie ma
```
unset katalog
echo $katalog
set
```
6. Sprawdź działanie znaków ` czy powodują potraktowanie wartości zmiennej jako polecenia do wykonania:
```
listc=`ls –l *.c`
echo $listc
set
```

5. Historia

W powłoce BASH ostatnio wydane polecenia są pamiętane. Polecenie **history** wyświetla listę ponumerowanych poleceń (zdarzeń). Można się do nich odwoływać używając poleceń z wykrzyknikiem.

Przykłady

!3 – wyświetla i wykonuje polecenie historii nr 3,
!ab – wyświetla i wykonuje ostatnio wydane polecenie rozpoczynające się ciągiem znaków „ab”,
!?ab? – wyświetla i wykonuje ostatnio wydane polecenie zawierające ciąg znaków „ab”,
!-3 – wyświetla i wykonuje trzecie od tyłu wydane polecenie.

Polecenia historii można składać. Jeśli np. polecenie o numerze 100 ma postać „**ls**”, wówczas **„!100 *.c**” jest równoważne poleceniu „**ls *.c**”. Możliwe jest także odwołanie do parametrów ostatnio wydanego polecenia poprzez „**!!***” lub „**!***”.

Przykład

Wydane zostało polecenie:

ls /home/elektr-a/ea029

Wówczas wydanie polecenia:

find !* -name szukaj

powoduje rozpoczęcie poszukiwania pliku o nazwie **szukaj** od katalogu **/home/elektr-a/ea029**.

Liczba pamiętanych zdarzeń historii przechowywana jest w zmiennej systemowej **HISTSIZE**. Aby ją zmienić, należy użyć polecenia np.: **HISTSIZE=100**. Historia przechowywana jest w pliku **.bash_history**, w katalogu domowym użytkownika. Można to zmienić definiując zmienną **HISTFILE** i nadając jej wartość, będącą nazwą innego pliku, np. **HISTFILE=mojahistoria**. Wówczas po wylogowaniu historia wydanych poleceń trafi do pliku o nazwie mojahistoria.

6. Aliasy

Aliasy, czyli utożsamienia, to dodatkowe nazwy dla najczęściej wykonywanych poleceń. Polecenie **alias** bez parametrów wyświetla listę zdefiniowanych aliasów. Tym samym poleceniem definiuje się także nowe aliasy.

Przykłady

alias kto=who – zamiast polecenia who można używać nazwy **kto**,

alias l='ls –li' – **l** staje się drugą nazwą dla polecenia **ls** z opcjami **li**; jeśli w poleceniu występują spacje, należy użyć pojedynczych cudzysłowów,

alias lc='ls –l *.c' – polecenie **lc** może być przydatne przy listowaniu plików źródłowych w języku C z danego katalogu,

alias cp='cp –i' – nazwa polecenia również może być aliasem.

Aby usunąć zdefiniowany alias, należy użyć polecenia **unalias**, np. **unalias kto** – usuwa definicję aliasu **kto**.

Aliasy nie są przekazywane do powłok potomnych, istnieją jedynie w bieżącej powłoce, pamiętane są w pamięci operacyjnej .

7. Pliki startowe powłoki

Definiowanie zmiennych oraz aliasów służy usprawnieniu pracy z powłoką. Dlatego konieczne jest, aby po zalogowaniu użytkownik miał zdefiniowane swoje środowisko według swoich potrzeb. W tym celu definicje aliasów i zmiennych środowiskowych umieszcza się w plikach startowych powłoki. Plik inicjacyjny logowania powłoki BASH to **.bash_profile (.profile)**. Jest on wykonywany automatycznie w czasie logowania użytkownika, powłoką logowania którego (login shell) jest BASH. Plik **.bashrc** wykonywany jest przy każdym uruchomieniu nowego shella (BASH). Oba pliki znajdują się w katalogu domowym użytkownika.

W pliku .bash_profile znajdują się m.in. polecenia ustawiające ścieżki przeszukiwania, typ terminala, znak zachęty, inne zmienne środowiskowe. Nowy shell powoływany jest przy wykonywaniu skryptów lub dowolnych poleceń, które nie są wbudowane w interpreter poleceń (komendy wewnętrzne powłoki). Zawsze wtedy wykonywany jest plik .bashrc. Znajdują się w nim polecenia ustawiające zmienne powłoki oraz definicje aliasów.

Istnieją ogólne pliki startowe powłoki, które są wykonywane przed osobistymi. Pliki .bash_profile oraz .bashrc poza swoim standardowym przeznaczeniem mogą być dodatkowo uruchomione w dowolnym momencie przez użytkownika analogicznie jak inne skrypty powłoki poleceniem „**.**" lub „**source**".

*Przykładowy plik **.bash_profile**:*

```
#.bash_profile
PATH=$PATH:$HOME/bin:$HOME:.
CDPATH=$CDPATH:$HOME
HISTSIZE=100
PS1=[\u\s@\w]$
PS2=@
export PATH CDPATH HISTSIZE PS1 PS2
```

*Przykładowy plik **.bashrc**:*

```
#.bashrc
set –o ignoreeof
set –o noclobber
alias rm='rm –i'
alias cp='cp –i'
alias mv='mv –i'
alias l='ls –la'
```

Do plików inicjacyjnych powłoki zaliczany również bywa **.bash_logout**, który jest wykonywany w czasie wylogowywania się użytkownika. Mogą się w nim znaleźć przykładowo polecenia:

```
clear
echo „Do nastepnego razu”
```

8. Skrypty powłoki bash

Powłoka **bash** jest najpopularniejszą a zarazem domyślną powłoką systemu Linux. W porównaniu do podstawowej wersji powłoki Bourne'a **bash**, czyli ***B**ourne-**A**gain **Sh**ell* zawiera wiele zaawansowanych możliwości, zaczerpniętych od innych chętnie stosowanych przez użytkowników unix'a powłok: **csh** oraz powłoki Korne'a (**ksh**). Poza funkcjami interpretera poleceń oraz zapewnienia interfejsu pomiędzy użytkownikiem, a jądrem systemu powłoka jest również bogatym narzędziem programistycznym. Powłoki, a w szczególności **bash** mają możliwości podobne do języków programowania. Można definiować zmienne oraz przypisywać im wartości. Definicje zmiennych, polecenia Linuksa i struktury sterujące można umieszczać w pliku tekstowym (skrypcie), który następnie można uruchomić. Jest on interpretowany przez powłokę.

Sposoby uruchamiania skryptów powłoki bash

Istnieje kilka sposobów uruchamiania skryptu:

- *. skrypt*
- source *skrypt*
- *skrypt*
- bash *skrypt*

Polecenie **.** oraz **source** umożliwiają interpretację skryptu przez tę samą powłokę i nie wymagają ustawienia prawa x do pliku skryptu, a jedynie r. Aby uruchomić skrypt poprzez jego nazwę plik musi mieć ustawione prawo x oraz r. Tak uruchomiony skrypt interpretowany jest przez nową powłokę. Polecenie bash *skrypt* to jawne wywołanie nowej powłoki, a plik skrypt jest argumentem – plikiem zawierającym dane do wykonania.

Komentarze poprzedzone znakiem # w pierwszej kolumnie (zazwyczaj).

#! – specjalny rodzaj komentarza, informuje system, że argument który pojawi się w tej linii jest programem którego należy użyć do wykonania tego pliku (np.: **#!/bin/bash**).

Skrypt kończy polecenie **exit** z kodem wyjścia (exit 0 – zakończenie poprawne, exit 1 – niepoprawne zakończenie)

Ćwiczenie 2:

Prześledź cztery wymienione sposoby uruchamiania skryptów. Zdefiniuj dwie zmienne powłoki: jedną lokalną oraz drugą środowiskową wykonując w powłoce bash polecenia:

```
zmlok=cos
zmsrod=cossrod
export zmsrod
```

Następnie wyedytuj za pomocą edytora plik skrypt, o następującej zawartości:

```
echo ”wartosc zmiennej lokalnej=$zmlok”
echo ”wartosc zmiennej srodowiskowej=$zmsrod”
echo ”PID procesu=$$”
```

Zadaniem skryptu jest wyświetlenie wartości zdefiniowanych zmiennych oraz identyfikatora procesu. Zmienna specjalna $$ jest ustawiana przez powłokę i oznacza identyfikator procesu (PID) bieżącego procesu.

Jaki jest efekt uruchomienia skryptu poleceniami source lub .?

Czym się różni od dwóch pozostałych sposobów?.

Dlaczego PID-y procesów są różne?

11.1. Wczytywanie danych i wypisywanie argumentów zgodnie z formatem.

Do wczytywania wartości zmiennych z klawiatury służy polecenie **read**. Polecenie:

`read` dane są wczytywane do zmiennej o domyślnej nazwie REPLY

`read a` wczytuje linie ze standardowego wejścia do zmiennej a

`read a1 a2 a3` wczytuje ze standardowego wejścia wartości trzech zmiennych: a1, a2, a3.

Do wypisywania argumentów zgodnie z formatem służy polecenie **printf**. Składnia:

```
printf [-v var] format [argumenty]
printf „format” par1 par 2 ...
```

FORMAT jest łańcuchem znakowym zawierającym trzy rodzaje obiektów: zwykłe znaki, które są kopiowane na standardowe wyjście; znaki sekwencji sterujących, które są przekształcane i kopiowane na standardowe wyjście oraz sekwencje formatujące, z których każda powoduje wypisanie kolejnego argumentu.

Wybrane formaty:

`%d, %i`	liczba całkowita
`%c`	pojedynczy znak (uwaga: przy zmiennej wypisze tylko pierwszy znak !)
`%s`	ciąg znaków
`%%`	wypisanie znaku %
`%(FORMAT)T`	interpretuję argument liczbowy jako datę i wypisuje ją zgodnie z zadanym formatem (argument -1 oznacza aktualny czas, -2 czas startu powłoki)

Opcja: -v ZMIENNA powoduje przypisanie wyjścia do podanej ZMIENNEJ powłoki zamiast wypisywania na standardowym wyjściu.

```
printf "%s %d\t%s\n" "zajęcia trwają" 90 minut
printf '%(%Y-%m-%d %H:%M:%S %Z (%A))T\n' 0 #czas 0
```

11.2. Polecenie exit

exit n – zakończenie skryptu z kodem wyjścia n, domyślnie kod ostatniego polecenia skryptu.

0 – sukces	
1-125	kody błędów do wykorzystania
126	plik nie był wykonywalny
127	nie znaleziono pliku o podanej nazwie
128 i więcej	pojawił się sygnał , wartość kodu to 128 + numer sygnału

11.3. Wyrażenia arytmetyczne

Często wykonywaną operacją stosowaną w skryptach powłoki bash jest obliczenie wartości wyrażeń. Można to zrobić przy pomocy mechanizmów interpretacji wyrażeń bash'a, polecenia let, albo korzystając zewnętrznego programu expr. Istnieją dwa rodzaje mechanizmów basha:

- `$((wyrażenie))`
- `$[wyrażenie]`

Przykład:

```
a=$(($a+2))
a=$[$a+2]
```

Używając polecenia let: zmienną a można zmodyfikować w następujący sposób:

- `let a=$a+2`

Należy zwrócić uwagę na brak spacji w wyrażeniu. Jeśli zastosuje się spacje należy użyć znaków ”:

```
let a=”$a + 2”
```

Programem, który służy do obliczania wartości wyrażenia jest expr. Pobiera on jako argumenty wyrażenie do obliczenia. Każda liczba i każdy znak muszą być oddzielone znakami spacji. Symbole odwróconego cudzysłowia wymuszają wykonanie polecenia, którego wynik stanie się wartością zmiennej.

```
a=`expr $a +  2`
```

W przypadku mnożenia może być konieczne zamaskowanie znaku * za pomocą symbolu \:

```
a=`expr $a \*  2`
```

Bash używa operatorów arytmetycznych z języka C.

11.4. Parametry powłoki, polecenie set

W skryptach powłoki bash można odwoływać się do zmiennych określających parametry wywołania skryptu. Ich znaczenie jest następujące:

$0	nazwa skryptu
$1,$2,..$9	parametry pozycyjne (przekazane do skryptu) lub ustalone ostatnio wykonanym poleceniem **set**. Odwołanie do parametrów o numerach większych niż 9 ma postać: **${nn}**
$*	lista parametrów $1, $2, ...rozdzielonych separatorem ($IFS)
$@	lista parametrów bez separatorów
$#	liczba parametrów skryptu
$?	status zakończenia ostatnio wykonanego polecenia
$$	identyfikator (PID) procesu interpretującego skryptu
$!	numer (PID) ostatnio uruchomionego procesu drugoplanowego

Polecenie **set** użyte w skrypcie ustawia zwykłe zmienne jako zmienne parametryczne dzięki czemu stają się dostępne w taki sam sposób, jak gdyby były podane jako parametry skryptu.

```
set plik1 plik2 plik3
echo $1 $2 $3
```

11.5. Rozwinięcia parametryczne

Jeżeli fragmentem tekstu/nazwy pliku / ma być parametr /wartość zmiennej/ to w takim przypadku możemy stosować rozwinięcia parametryczne, które zapisujemy w postaci ${…}.

```
for i in 1 2 3 4 5
do
      echo plik${i}A.c
      rm -i plik${i}A.c
done
```

Przykładowe rozwinięcia parametryczne:

`${#parametr}`	podaje długość parametr w znakach
`${parametr%tekst}`	od końca usuwa najmniejszą część parametr pasującą do tekst
`${parametr%%tekst}`	od końca usuwa najdłuższą część parametr pasującą do tekst
`${parametr#tekst}`	od początku usuwa najmniejszą część parametr pasującą do tekst
`${parametr##tekst}`	od początku usuwa najdłuższą część parametr pasującą do tekst
`${parametr:-wartość}`	jeśli parametr ma wartość pustą staje się równy wartość

Przykład:

```
plik=/home/inf1n-21z/nowak.piotr/home/bin
echo $plik
echo ${#plik}
echo ${plik%home*}
```

```
echo ${plik%%home*}
echo ${plik#*/}
echo ${plik##*/}
```

Wyniki:

```
/home/inf1n-21z/nowak.piotr/home/bin
36
/home/inf1n-21z/nowak.piotr/
/
home/inf1n-21z/nowak.piotr/home/bin
bin
```

11.6. Instrukcje warunkowe, polecenie test, operatory relacyjne, logiczne

Powłoka bash umożliwia warunkowe wykonywanie poleceń. Dostępne są warunkowe struktury sterujące if oraz case. W przeciwieństwie do instrukcji sterujących w językach programowania if sprawdza kod wykonania polecenia, a nie wyrażenie. Oto składnia polecenia if-then-else:

```
if warunek
   then
  polecenia1
   else
  polecenia2
fi
```

Jeśli słowo kluczowe **then** występuje w tej samej linii, co **if** musi zostać poprzedzone średnikiem:

```
if warunek; then
    polecenia
fi
```

Możliwa jest konstrukcja wielopoziomowa instrukcji warunkowej: if-elif-else-fi

```
if warunek1 ; then
       polecenia1
elif warunek2 ; then
       polecenia2
elif warunek3 ; then
       polecenia3
else
       polecenia4
fi
```

Poleceniem często występującym bezpośrednio po słowie kluczowym if jest **test** - badające pewien warunek. Jeśli warunek jest spełniony, polecenie test zwraca wartość 0, czyli logiczną prawdę, w przeciwnym wypadku wartość niezerową, czyli fałsz. Zamiast słowa kluczowego test można również używać nawiasów kwadratowych.

Polecenia:

```
test $a –eq 1
[ $a –eq 1 ]
```

są równoważne i sprawdzają, czy wartość zmiennej a wynosi 1. Należy pamiętać o znakach spacji po znaku [i przed znakiem].

W przypadku sprawdzania warunków arytmetycznych można użyć nawiasów okrągłych i operatorów języka C, np.:

```
(( $a == 1 ))
```

Do budowania warunków można używać wiele operatorów.
Najczęściej używane to:

- **operatory logiczne**
  - ! *wyr* – negacja
  - *wyr1* -a *wyr2* - logiczne AND (i)
  - *wyr1* -o *wyr2* - logiczne OR (lub)
- **operatory do porównania łańcuchów znakowych**
  - *ciag1*=*ciag2* - ciągi identyczne
  - *ciag1*!=*ciag2* - ciągi różne
  - -z *ciag* - ciąg pusty
  - -n *ciag* - ciąg nie jest pusty
- **testowanie plików**
  - -f *plik* - plik zwykły
  - -d *plik* - katalog
  - -r *plik* - plik do odczytu
  - -w *plik* - plik do zapisu
  - -s *plik* - plik niepusty
  - -x *plik* - plik wykonywalny
- **porównywanie liczb, wyrażeń arytmetycznych**
  - wyr1 -eq wyr2 (==)
  - wyr1 -ne wyr2 (!=)
  - wyr1 -lt wyr2 (<)
  - wyr1 -le wyr2 (<=)
  - wyr1 -gt wyr2 (>)
  - wyr1 -ge wyr2 (>=)

Należy zwrócić uwagę, że do porównywania ciągów znaków używa się operatora =, natomiast do porównywania liczb operatora –eq.

Ćwiczenie 3:
Sprawdź czy istnieje plik o nazwie będącej pierwszym argumentem wywołania skryptu.

Ćwiczenie 4:
Sprawdź, który z dwóch plików będących argumentem wywołania skryptu jest dłuższy i o ile znaków. Na początku sprawdź liczbę argumentów i ich poprawność.

11.7. Instrukcja warunkowa case

W powłoce bash można korzystać z konstrukcji case, która umożliwia porównanie zmiennej z wieloma wzorcami i w zależności od wyniku dopasowania wykonanie odpowiedniej akcji. Składnia instrukcji warunkowej case jest następująca:

```
case napis in
     wart1)
          polecenia1
       ;;
     wart2)
          polecenia2
       ;;
     ……
     *)
          polecenia
       ;;
esac
```

napis może być odwołaniem do zmiennej – np. $a. Wartość napis porównywana jest kolejno ze wszystkimi wzorcami wart1, wart2, ...i wykonywane jest polecenie przypisane do danej wartości. Jeśli żadne dopasowanie nie zakończy się sukcesem wykonywane są polecenia domyślne, występujące po symbolu *. Należy zwrócić uwagę na podwójny znak średnika kończący każdy potok instrukcji. Jeśli by go zabrakło shell wykonałby kolejne instrukcje.

Ćwiczenie 5:
Napisz skrypt, który implementuje menu i dla wybranej (podanej z klawiatury) opcji wykonuje określone polecenia:
s- wykonuje polecenie ls –s
l – wykonuje polecenie ls –l
c – wykonuje polecenie ls *.c

11.8. Pętle

W powłoce bash dostępne są trzy typy struktur sterujących tworzących pętle: **for, while** oraz **until**.
Wewnątrz pętli **for, while** i **until** mogą występować dodatkowe polecenia:

break [n] zakończenie wykonywania bieżącej pętli i przejście do pierwszego polecenia po tej pętli, lub - jeżeli wyspecyfikowano n – do polecenia po pętli o n poziomów zewnętrznej

continue [n] zaprzestanie wykonywania dalszych poleceń z zakresu bieżącej pętli i przejście na jej początek, lub - jeżeli wyspecyfikowano n – na początek pętli o n poziomów zewnętrznej

Pętla for powoduje wykonanie ciągu instrukcji dla każdego elementu z listy. Składnia jest następująca:

```
for zmienna in lista_wartosci
     do
          polecenia
     done
```

Powtarzany ciąg poleceń jest zawarty między słowami kluczowymi do a done. Lista wartości może być podana jawnie lub za pomocą metaznaków, jak w przykładzie:

```
for   plik  in   *.c
 do
  echo plik $plik
  cat $plik|more
 done
```

Uruchomienie powyższego skryptu powoduje wyświetlenie nazw i zawartości wszystkich plików źródłowych w języku C z katalogu bieżącego.
Jeśli nie występuje *lista_wartosci* ani słowo kluczowe in polecenia pętli wykonywane są dla wszystkich parametrów skryptu. Tak więc pętla for bez listy (for *zmienna*) jest równoważna pętli:

```
for zmienna in $@
```

Przykładem pętli for bez listy jest poniższy skrypt obliczający sumę wszystkich parametrów wywołania:

```
s=0
for skl
  do
        s=$[$s+$skl]
  done
echo suma=$s
```

W powłoce bash dostępna jest również pętla for składniowo zbliżona do pętli for z języka C. Ilustruje to poniższy przykład:

```
for((i=0;i<5;i++));do echo "---$i---";done
```

Należy zwrócić uwagę na fakt, że wyrażenia sterujące pętlą umieszczone są w podwójnych nawiasach. Podobnie jak w przypadku struktury sterującej if w składni pętli obowiązuje zasada, że jeśli słowo kluczowe do znajduje się w tej samej linii, co słowo kluczowe rozpoczynające pętlę (for, while, until) musi być poprzedzone przecinkiem. Podobnie jest ze słowem kluczowym done kończącym pętlę.
Pętla while oraz until powodują powtarzanie ciągu poleceń w zależności od wartości logicznej wskazanego polecenia (zwykle jest to polecenie test).

```
while warunek
   do
      polecenia
   done
```

```
until warunek
   do
     polecenia
   done
```

Obie pętle wykonują polecenie testujące przed wejściem do pętli. Różnica polega na tym, że pętla while jest wykonywana, gdy *polecenie* zakończy się sukcesem (zwraca kod równy 0), natomiast until powtarza ciąg poleceń przy niezerowym kodzie zakończenia polecenia *polecenie*.

Ćwiczenie 6:
Napisz skrypt, który wyświetli na ekranie określoną ilość razy (będącą pierwszym argumentem wywołania skryptu) tekst „**systemy operacyjne**”.

Ćwiczenie 7:
Napisz skrypt, który sprawdzi, czy użytkownik o podanym loginie - będącym pierwszym argumentem wywołania - zalogował się. Skrypt ma wypisać o której godzinie wskazany użytkownik się zalogował.

Poniższe skrypty ilustrują jak stosując pętle while oraz pętlę until obliczyć sumę wszystkich parametrów wywołania:

```
s=0
until [ ! $1 ]
     do
      s=$[$s+$1]
      shift
     done
echo suma=$s
```

```
s=0
while [ $1 ]
    do
      s=$[$s+$1]
      shift
     done
echo suma=$s
```

Zastosowano w nich polecenie **shift**, które przesuwa zmienne parametryczne o jedną pozycję. Pierwsze wywołanie polecenia shift w skrypcie powoduje, że wartość $0 pozostaje niezmieniona, natomiast $1 przyjmuje wartość drugiego parametru, $2 będzie odpowiadało trzeciemu parametrowi itd. Dzięki zastosowania polecenia shift w pętli, do wszystkich parametrów skryptu, bez względu na ich liczbę można się odwoływać przez: $1.

Innym sposobem na tworzenie pętli jest użycie struktury sterującej **select..do**. Jej składnia jest następująca:

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

W oparciu o zadaną listę tworzy na ekranie ponumerowane menu i wyświetla znak zachęty, określony przez zmienną powłoki PS3. Użytkownik dokonuje wyboru poprzez podanie liczby, odpowiadającej wybranej pozycji i wykonywane są instrukcje pętli (zwykle jest to case). Działanie pętli select … do ilustruje poniższy skrypt.

```
#!/bin/bash
select a in a b c koniec
do
    case $a in
      "a") echo "a"
      ;;
      "b") echo "b"
      ;;
      "c") echo "c"
      ;;
      "koniec") exit
      ;;
      *) echo "zly wybor"
      ;;
     esac
done
```

11.9. Potoki i listy poleceń

Dla skrócenia zapisu w skryptach powłoki bash stosuje się listy poleceń. Lista poleceń jest sekwencją poleceń lub potoków rozdzielonych jednym z symboli: ; & && || i opcjonalnie zakończonych znakiem ; lub &. Symbole: ; i & mają taki sam priorytet, niższy od priorytetów symboli: && i || (równorzędnych).

- **;** oznacza sekwencyjne wykonywanie poleceń
- Potok poprzedzony znakiem **&** jest wykonywany w tle, a kolejne polecenie jest wykonywane natychmiast
- Symbol **&&** oznacza koniunkcję - polecenie występujące po nim zostanie wykonane tylko wtedy, gdy polecenie poprzedzające go zakończy się powodzeniem (kodem 0)
- Symbol **||** oznacza alternatywę - polecenie występujące po nim zostanie wykonana tylko wtedy, gdy polecenie poprzedzające go zostanie zakończone niepowodzeniem (kodem błędu)

Oto przykład dwóch równoważnych skryptów, których zadaniem jest sprawdzenie, czy istnieje plik .profile.

```
if [ -f .profil ]; then
      exit 0
fi
exit 1
```

```
[ -f .profil ] && exit 0 || exit 1
```

Skrypt z kolejnego przykładu sprawdza, czy pierwszy parametr wywołania jest katalogiem z prawem wykonania.

```
if [ -d $1 ] && [ -x $1 ]
then
    echo ”katalog z prawem x”
else
    echo ”nie katalog lub brak praw
x”
fi
```

W powyższym skrypcie zastosowano operator koniunkcji: && dotyczący dwóch poleceń testujących wartość wyrażenia. Skrypt ten można napisać również stosując jeden operator test : [ ] ze złożonym warunkiem, zbudowanym przy pomocy operatora koniunkcji –a, jak w przykładzie poniżej:

```
if [ -d $1 -a -x $1 ]
then
    echo ”katalog z prawem x”
else
    echo ”nie katalog lub brak praw
x”
fi
```

Za pomocą listy poleceń ten sam skrypt można zapisać znacznie krócej, ale za to mniej czytelnie:

```
[ -d $1 ] && [ -x $1 ] && echo ”kat. z x”|| echo ”nie”
```

Do pełnego diagnozowana przypadków elementarnych konieczne jest zastosowanie złożonej struktury if:

```
if [ -d $1 ]
then  if [ -x $1 ]
```

```
      then
       echo ”katalog z prawem x”
      else
       echo ”katalog bez prawa x”
      fi
else
      echo „nie katalog”
fi
```

11.10. Przechwytywanie sygnałów – polecenie trap.

Do przechwytywania sygnałów i innych zdarzeń służy polecenie.

```
trap [-lp] [[arg] sygnał ...]
```

Opcje:

- -l wypisanie listy nazw sygnałów i ich numerów
- -p wypisanie poleceń trap powiązanych z każdym SYGNAŁEM

Gdy powłoka otrzyma podany SYGNAŁ (lub sygnały), odczytywane i uruchamiane jest polecenie podane jako argument ARG.

```
trap 'rm ./tplik$$' SIGINT
```

W razie braku argumentu (i podaniu pojedynczego SYGNAŁU) lub gdy argumentem jest ‘-', każdemu z podanych sygnałów jest przywracane pierwotne zachowanie.

```
trap - SIGINT
```

Jeśli ARG jest pustym łańcuchem, każdy SYGNAŁ jest ignorowany przez powłokę i wywołane przez nią polecenia.

Przykład 1:

```
trap 'rm ./tplik$$' SIGINT
date > ./tplik$$
echo "nacisnij ctrl+c"
while [ -f ./tplik$$ ]; do
     echo plik istnieje przed ctrl+c
     sleep 1
done
trap - SIGINT
date > ./tplik$$
echo "ctrl+c"
while [ -f ./tplik$$ ]; do
     echo plik istnieje po ctrl+c
     sleep 1
done
```

Przykład 2:*

```
#!/bin/bash
trap 'echo "Jestem kuloodporny, strzelaj sobie ..."' SIGINT
trap 'echo "Teraz już nie jestem kulodporny, możesz mnie unicestwić"; trap -
SIGINT' SIGHUP
```

```
while : ; do
  echo "Moj PID to: $$ ; Naciśnij ^C (CTRL+C) aby wyjść"
  sleep 5
done
```

Po uruchomieniu skryptu, naciśnij kilka razy CTRL+C, następnie z drugiej konsoli wydaj poniższe polecenie i ponownie naciśnij CTRL+C (pid procesu sprawdź poleceniem ps)

```
kill -SIGHUP 152356
```

11.11. Definiowanie funkcji

W bash-u także można definiować funkcje - zbiory instrukcji, które często się powtarzają w programie. Funkcję definiuje się w następujący sposób:

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

Słowo kluczowe function może zostać pominięte. Funkcję w skrypcie wywołuje się podając jej nazwę i parametry: ***nazwa_funkcji parametry.*** Wewnątrz funkcji parametry są widoczne jako zmienne $1, $2, $*, itd. przysłaniając parametry wywołania skryptu. Wyjście z funkcji odbywa się po wykonaniu polecenia **return** lub ostatniego polecenia w funkcji. Zwracany jest kod wyspecyfikowany w poleceniu return lub kod zakończenia ostatniego polecenia funkcji. Po zakończeniu funkcji parametry pozycyjne przyjmują poprzednie wartości.

Oto przykład prostego skryptu bash wykorzystującego funkcję, w której wyświetlana jest zachęta do podania wartości z klawiatury:

```
#!/bin/bash
function naglowek
{
echo "podaj skladnik"
}

for ((i=0;i<5;i++))
do
     naglowek
     read i
     s=$(($s + $i))
done
echo suma=$s
```

Wygodnie jest umieścić funkcje w innym pliku (np. funkcja). W skrypcie odwołanie do funkcji musi być poprzedzone dołączeniem pliku zawierającego definicje funkcji za pomocą polecenia: . funkcja. Oto przykład skryptu odwołującego się do funkcji nagłówek zdefiniowanej w pliku funkcja oraz przykładowy plik o nazwie funkcja zawierający definicję funkcji:

plik funkcja:

```
#!/bin/bash
function naglowek
{
  echo podaj składnik
}
```

skrypt

```
#!/bin/bash
. funkcja
for ((i=0;i<5;i++))
do
  naglowek
  read ii
  s=$(($s + $ii))
done
echo suma=$s
```

Poniżej przestawiony jest skrypt, który rysuje na ekranie trójkąt prostokątny o zadanej wysokości wypełniony zadanym znakiem. Wykorzystana została funkcja linia, która rysuje pojedyncza linię. Jej parametrami są: liczba początkowych spacji, liczba kolejnych znaków oraz znak.

```
#!/bin/bash
linia()
{
for((i=0;i<$1;i++))
do
echo -n " "
done
for((i=0;i<$2;i++))
do
echo -n $3
done
}
echo -n "wciecie="
read sp
echo -n "znak="
read zn
echo -n "wysokosc="
read r
for((k=1;k<=$r;k++))
do
linia sp k $zn
echo
sp=$[$sp-1]
done
```

Przykładowe wywołanie skryptu:

```
$ bash trojkat
wciecie=20
znak=$
rozmiar=10
                  $
                 $$
                $$$
               $$$$
              $$$$$
             $$$$$$
            $$$$$$$
           $$$$$$$$
          $$$$$$$$$
$
```

11.12. Zmienne tablicowe

W skryptach powłoki bash można definiować tablice. Tablica jest listą wartości oddzielonych spacjami. Oto przykład definicji tablicy o nazwie tab:

```
tab=(kambr ordowik sylur dewon karbon)
```

Kolejne elementy tablicy indeksowane są liczbami całkowitymi zaczynając od 0, a do i-tego elementu tablicy odwołuje się poprzez: ${tab[i]}. ${tab[@]} i ${tab[*]} oznaczają wszystkie elementy tablicy.

Kolejny element tablicy można dodać poleceniem: tab[i]=*nowa_wartosc,* a do usunięcia elementu służy polecenie unset. ${#tab[i]} oznacza długość i-tego elementu tablicy. Poniższy skrypt ilustruje definiowanie i sposoby odwołania się do elementów tablicy:

```
#!/bin/bash
tab=(kambr ordowik sylur dewon karbon)
echo ${tab[@]}
echo ${tab[*]}
for i in 0 1 2 3 4
do
  echo tab[$i]=${tab[i]}
  echo długosc tab[$i] = ${#tab[i]}
done
tab[5]=perm
echo ${tab[@]}
unset tab[0]
echo ${tab[@]}
```

A oto efekt wywołania skryptu:

```
$ bash tablice
kambr ordowik sylur dewon karbon
kambr ordowik sylur dewon karbon
tab[0]=kambr
długosc tab[0] = 5
tab[1]=ordowik
długosc tab[1] = 7
tab[2]=sylur
długosc tab[2] = 5
tab[3]=dewon
długosc tab[3] = 5
tab[4]=karbon
długosc tab[4] = 6
kambr ordowik sylur dewon karbon perm
ordowik sylur dewon karbon perm
$
```

Ćwiczenia 8.

1. Napisz skrypt zamieniający nazwy dwóch plików, będących parametrami wywołania skryptu.
2. Napisz skrypt który wczytuje nazwy plików i sprawdza, czy istnieje taki plik i czy ma ustawione prawa do odczytu. Sprawdzanie powtarza się dla wszystkich wprowadzanych wartości z klawiatury, do podania pustej nazwy. Gdy nie podamy ani jednej wartości, skrypt kończy działanie z kodem błędu 4.
3. Napisz skrypt który oblicza sumę danych, będących parametrami wywołania skryptu (od drugiego parametru) do napotkania wartości równej pierwszemu parametrowi.
4. Napisz skrypt który wyświetla menu zdefiniowane w funkcji i realizuje polecenia w zależności od wybranej opcji.

*Treści oznaczone kursywą pochodzą z różnych źródeł internetowych.