Laboratorium Ubuntu Linux.

Programowanie w powłoce bash.

Opracował: *Krzysztof Ropiak*

1. Programowanie w powłoce bash.

1.1. Wprowadzenie

W jakim celu programować w powłoce skoro Ubuntu dostarcza ogromnej ilości gotowych narzędzi oraz daje możliwość pobierania nowych poprzez rozmaite narzędzia zarządzania oprogramowaniem (pakietami)? Oto kilka powodów:

- mimo wielu możliwości dostępnego oprogramowania okazuje się, że do rozwiązania naszego problemu potrzeba czegoś więcej,
- nawet jeżeli oprogramowanie daje nam 100% potrzebnej funkcjonalności to często wymagane jest od nas wykonanie szeregu czynności, które składają się na cały proces rozwiązania problemu,

Co możemy zyskać dzięki programowaniu i skryptom powłoki:

- elastyczność,
- automatyzacja,
- większa wydajność,
- więcej wolnego czasu ☺.

1.2. Skrypty powłoki.

Co to jest skrypt ?

Skrypt to niekompilowany tekstowy plik wykonywalny, zawierający polecenia systemowe oraz instrukcje sterujące jego wykonaniem. Wykonywany jest tylko i wyłącznie przez interpreter (w naszym przypadku przez `/bin/bash`), który tłumaczy polecenia zawarte w skrypcie na język zrozumiały dla procesora.

Jak uruchomić skrypt ?

Po pierwsze należy najpierw ustawić uprawnienia do wykonania dla naszego skryptu:

```
chmod +x skrypt.sh
```

Chociaż nie ma wymogu co do nazewnictwa plików to dobrą praktyką jest nadawanie im rozszerzeń, które będą sugerowały, iż jest to skrypt powłoki, np. `.sh`.

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

```
./skrypt.sh
```

Budowa skryptu

Skrypt powłoki powinien rozpoczynać się od linii:

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

Znak „#” jest znakiem komentarza jednowierszowego, co oznacza, że wszystko co znajduje się po nim w tej samej linii nie będzie brane pod uwagę przez interpreter tylko zostanie potraktowane jako zwykły tekst. Wyjątkiem jest znak shebang (więcej tutaj), który definiuje rodzaj powłoki, w której skrypt ma zostać wykonany.

1.3. Zmienne

Składnia: `zmienna=wartość`

Zmienne, które są wykorzystywane wewnątrz skryptu powinny być zapisywane małymi literami. Stałe, czyli zmienne, których wartość nie ulega zmianie podczas wykonywania skryptu zapisujemy wielkimi literami.

Aby odwołać się do wcześniej zdefiniowanej zmiennej, np. w poleceniu echo nazwę zmiennej poprzedzamy znakiem „$”. Polecenie `echo` wysyła ciąg znaków (bezpośrednio lub będących wynikiem obliczeń) na standardowe wyjście.

KOD:

```
#!/bin/bash
imie=Marek
echo „Mam na imię” $imie
```

Efektem wykonania skryptu będzie tekst:

```
Mam na imię Marek
```

Zmienne mogą przechowywać liczby, znaki i ciągi tekstowe.

KOD:

```
liczba=1
znak=x
ciag=”Ala ma kota”
```

Zmienne mogą przechowywać również wartości będące efektem wykonania poleceń powłoki, np.:

```
GDZIE_JESTEM=`pwd`
echo „Jestem w „$GDZIE_JESTEM
```

Zmienne specjalne

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

`$0`

nazwa bieżącego skryptu lub powłoki

`$1..$9`

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

`$#`

Zawiera liczbę argumentów przekazanych w linii poleceń.

`$_`

Ścieżka z jaką został wywołany skrypt (powłoka)

`$@`

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

`$$`

PID procesu bieżącej powłoki

Zmienne systemowe

Zmienne systemowe, często nazwane także zmiennymi środowiskowymi (ang. environment variables) - są zdefiniowane i przechowywane w środowisku nadrzędnym (powłoce z której uruchamiane są Twoje skrypty). Ich nazwy są zwyczajowo zapisane wielkimi literami. Definiują środowisko użytkownika, dostępne dla wszystkich procesów potomnych.

Inicjacja zmiennej globalnej:

```
test@host:~$ export JA=”Krzysztof Ropiak”
test@host:~$ echo $JA
Krzysztof Ropiak
```

Listę zdefiniowanych zmiennych systemowych wraz z ich wartościami wyświetlamy poleceniem `printenv`.

1.3.1. 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 indeksowane są przy pomocy liczb całkowitych, zaczynając od 0.

Składnia:

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

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 odwołujemy się za pomocą wskaźników.

Odwołanie do elementów tablicy.

Składnia:

```
${nazwa_zmiennej[wskaźnik]}
```

Wskaźnikami są indeksy elementów tablicy, począwszy od **0** do n oraz **@, ***. Gdy odwołując się do zmiennej nie podamy wskaźnika: `${nazwa_zmiennej}` to nastąpi odwołanie do elementu tablicy o indeksie 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 indeksie 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.

1.4. Obliczenia numeryczne

Operatory matematyczne:

„+” – dodawanie
„-” – odejmowanie
„*” – mnożenie
„/” – dzielenie
„%” – dzielenie z resztą (modulo)

1.4.1. Obliczenia na liczbach całkowitych

Rozpatrzmy poniższy przykład:

```
!#/bin/bash

a=1
b=2
c=$a+$b
echo $c
```

Spodziewamy się, iż powyższy skrypt zwróci wynik w postaci liczby `3`, ale rzeczywistym wynikiem będzie ciąg `1+2`. Jest to domyślne działanie interpretera skryptów powłoki bash.

Aby wykonać obliczenia matematyczne możemy wykonać operacje na kilka sposobów.

1. Deklarujemy zmienną $c jaką zmienną przechowującą dane numeryczne wykorzystując funkcję „declare –i”:

```
#!/bin/bash

a=1
b=2
declare -i c=$a+$b
echo $c
```

Teraz otrzymamy na wyjściu liczbę `3`.

2. Stosujemy funkcję `expr` w celu obliczenia wyrażenia. Należy pamiętać jednak o kilku zasadach, które najlepiej wytłumaczyć na przykładach:

```
test@host:~$ expr 2+3
2+3
```

Mimo zastosowania funkcji `expr` wynikiem działania jest ciąg znaków.

```
test@host:~$ expr 2 + 3
5
```

Stosując polecenie `expr` należy pamiętać o stosowaniu odstępów pomiędzy argumentami i operatorami. Teraz wynik działania jest tym co chcieliśmy osiągnąć.

W dokumentacji można również spotkać zapis, w którym każdy znak specjalny wyrażenia jest poprzedzony backslashem co jak najbardziej jest poprawne:

```
test@host:~$ expr 2 \+ 3
5
test@host:~$  expr 2 \* \( 7 - 1 \)
12
```

W powyższych przykładach wykonywaliśmy obliczenia bezpośrednio z wiersza poleceń. Polecenie `expr` możemy oczywiście wykorzystać w skrypcie:

```
#!/bin/bash

a=1
b=2
c=`expr $a + $b`
echo $c
```

Wyrażenie zawarte pomiędzy znakiem „` ” oraz „ ` ” informuje interpreter o tym, że powinien wykonać te wyrażenie jako polecenie powłoki i w tym konkretnym przypadku jego wynik przypisać do zmiennej c. Wynikiem będzie suma zmiennych a i b czyli liczba 3.

Kolejny sposób na przypisanie wartości wyrażenia do zmiennej z wykorzystaniem polecenia expr:

```
!#/bin/bash

a=1
b=2
c=$(expr $a + $b)
echo $c
```

Zapis $(wyrażenie) jest tożsamy z zapisem `wyrażenie` czyli powoduje obliczenie wartości wyrażenia i w tym przypadku zapisanie tej wartości do zmiennej c.

Istnieje sposób pozwalający na pozbycie się polecenia expr jeżeli chcemy obliczyć wartość wyrażenia. Stosujemy wtedy zapis $((wyrażenie)):

```
#!/bin/bash

a=1
b=2
c=$(($a + $b))
echo $c
```

Warto wspomnieć że stosowanie zapisu $((…)) jest szybsze niż obliczenia z wykorzystaniem `…` gdyż w tym drugim przypadku tworzony jest nowy proces. Dodatkowo zapis $((…) jest interpretowany bezpośrednio przez bash-a. Wadą może być to, że taki zapis może nie działać w powłokach innych niż bash. Wewnątrz nawiasów nie musimy również martwić się o poprzedzanie zmiennych znakiem $ oraz nie musimy stosować backslasha przed operatorami.

W przypadku składni ((…)) można pominąć $ wewnątrz wyrażenia.

```
#!/bin/bash

a=1
b=2
c=$((a + b))
echo $c
```

Dodatkową zaletą jest również to, iż wewnątrz ((...)) możemy korzystać ze znacznie szerszej palety operatorów arytmetycznych - znanych z języka C:

```
((a++))
((--a))
((a+=2))
…
```

1.4.2. Obliczenia na liczbach zmiennoprzecinkowych

Wykonanie poniższego skryptu

```
#!/bin/bash

a=3
b=2
c=$(($a / $b))
echo $c
```

wyświetli na wyjściu wynik 1 – co jak wiemy nie jest zgodne z prawdą. Domyślnie bash nie operuje na liczbach zmiennoprzecinkowych i zwraca wynik w postaci liczby całkowitej. Aby to zmienić możemy wykorzystać wbudowany kalkulator, który dostępny jest poprzez polecenie `bc`.

Zobaczmy jak wyglądałoby powyższe obliczenie w kalkulatorze `bc`:

```
test@host:~$  bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3/2
1
```

Po uruchomieniu kalkulatora wprowadzamy wyrażenie do obliczenia i w celu jego obliczenia wciskamy klawisz Enter. Aby wyjść z kalkulatora wpisujemy `quit` i wciskamy Enter. Jak widać domyślnie kalkulator zwrócił nam taki sam wynik jak skrypt. Aby to zmienić musimy poinformować kalkulator o ilości miejsc po przecinku, na których chcemy operować. Robimy to poprzez ustawienie wartości dla parametru `scale`.

```
test@host:~$  bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
scale=2
3/2
1.50
```

Teraz wynik wyświetla się poprawnie z dokładnością do dwóch miejsc po przecinku.

Aby wykorzystać bc w skrypcie musimy skorzystać z poznanej wcześniej techniki - przekazania wyniku działania jednej funkcji jako dane wejściowe innej funkcji czyli przekazanie jako potok.

```
test@host:~$   echo "2.4+7" | bc
9.4
```

Przykład z parametrem `scale`:

```
test@host:~$   echo "scale=2;3/2" | bc
1.50
```

Przypisanie wyniku działania powyższego wyrażenia do zmiennej:

```
test@host:~$   a=`echo "scale=2;3/2" | bc`
test@host:~$ echo $a
1.50

test@host:~$   a=$(echo "scale=2;3/2" | bc)
test@host:~$ echo $a
1.50
```

1.4.3. Polecenie let

Polecenie let jest wbudowanym poleceniem bash-a. Służy ono do przetwarzania wyrażenia lub wyrażeń.

Składnia:

```
let wyrażenie1 wyrażenie2 ...
```

Powyższy zapis równoważny jest zapisowi:

```
((wyrażenie1))
((wyrażenie2))
...
```

Przykład:

```
test@host:~$ x=0
test@host:~$ let x+=2 "x += 4"
test@host:~$ echo $x
6
```

Należy pamiętać o tym, iż wyrażenia zawierające odstępy muszą być ujęte w cudzysłów.

1.4.4. Porównywanie liczb, łańcuchów i plików.

Polecenie `test` służy do sprawdzania wielu różnych warunków, np. czy plik istnieje, czy jedna wartość jest większa od drugiej lub czy dwa ciągi znaków są jednakowe. Wynikiem takiego porównania jest zawsze jedna z dwóch wartości: 0 – prawda (true) lub 1 – fałsz (false) a wynik porównania przechowywany jest w zmiennej $?.

Przykład:

```
test@host:~$ test -d /etc/var
test@host:~$ echo $?
1
```

W powyższym przykładzie sprawdzamy czy argument „`/etc/var`" jest katalogiem (`test -d`). Wynik został zapisany do zmiennej `$?` i wynosi `1` co oznacza, że `/etc/var` nie jest katalogiem lub nie istnieje.

Zapis `test warunek` można zastąpić zapisem `[ warunek ]` – należy pamiętać o zachowaniu odstępów między nawiasami a warunkiem.

Przykład:

```
test@host:~$ [ -d /etc/var ]
test@host:~$ echo $?
1
```

Opcje dla polecenia `test`:

Operacje na systemie plików

-b	plik istnieje i jest blokowym plikiem specjalnym
-c	plik istnieje i jest plikiem znakowym
-d	katalog istnieje
-e	plik istnieje
-f	plik istnieje i jest plikiem zwykłym
-h	plik istnieje i jest linkiem symbolicznym
-r	plik można czytać
-w	plik można zapisywać
-x	plik można wykonywać

plik1 -nt plik2	plik1 jest nowszy od pliku2
plik1 -ot plik2	plik1 jest starszy od pliku2

Operacje arytmetyczne

-eq	równy (equal to)
-ne	różny (not equal to)
-lt	mniejszy niż (less than)
-le	mniejszy lub równy (less than or equal to)
-gt	większe niż (greather than)
-ge	większe lub równe (greather or equal to)

Operatory operujące na łańcuchach znakowych

=	równy
!=	różny
<	pierwszy tekst alfabetycznie przed drugim
>	pierwszy tekst alfabetycznie za drugim
-n	wyrażenie ma długość większą niż 0
-z	wyrażenie ma zerową długość [$x = ""]

Łączenie kilku warunków

Czasem istnieje konieczność sprawdzenia wielu warunków jednocześnie i w tym celu możemy posłużyć się **alternatywą lub koniunkcją logiczną**.

Test warunków i alternatywa logiczna:

```
#!/bin/bash
a=2
test $a –gt 1 –o $a –lt 3
echo $?
```

Jak widać w powyższym przykładzie aby użyć alternatywy logicznej należy między warunkami dodać opcję –o (ang. OR, lub). Parametr –o można zastąpić poprzez „||”:

```
#!/bin/bash
a=2
test $a –gt 1 || test $a –lt 3
echo $?
```

Lub w skróconym zapisie:

```
#!/bin/bash
a=2
[ $a –gt 1 ] || [ $a –lt 3 ]
echo $?
```

W przypadku koniunkcji logicznej (ang. AND) używamy parametru –a lub „&&”:

```
#!/bin/bash
a=2
test $a –gt 1 –a $a –lt 3
echo $?
```

Warto jeszcze wspomnieć o negacji logicznej „!” (ang. NOT), która też jest przydatna podczas pisania skryptów w bash-u.

Przykład:

```
#!/bin/bash
a=7
b=2
test ! $a -gt $b
echo "Result: $?"
```

Wyrażenie `$a -gt $b` sprawdza, czy wartość `$a` jest większa niż wartość `$b`. W powyższym przykładzie jest to prawda. Znak negacji powoduje zmianę wyniku na FAŁSZ.

Zmienna `$?` pełni również inną rolę w środowisku bash. Każdy program kończy się z jakimś statusem, który przechowywany jest właśnie w zmiennej `$?`. Zgodnie z konwencją jeśli polecenie wykonało się z sukcesem, kodem wyjścia jest 0, a jeśli w wyniku wykonania pojawiły się błędy lub polecenie skończyło się porażką, zwracany jest kod różny od zera.

Normalnie własny skrypt kończy się ze statusem wyjścia równym zero. Możemy zakończyć skrypt w dowolnym miejscu z wybranym przez nas statusem wyjścia, stosując polecenie `exit`. Na przykład instrukcja `exit 1` powoduje natychmiastowe zakończenie skryptu z kodem wyjścia 1.

Przykład:

```
test@host:~$ ls *.txt
test.txt
test@host:~$ echo $?
0
test@host:~$ ls *.nieznane
ls: *.nieznane: Nie ma takiego pliku ani katalogu
bashtest@host:~$ echo $?
2
```

1.5. Wprowadzanie danych wejściowych

Jak powszechnie wiadomo skrypt/funkcja lub program zwraca wynik swego działania na podstawie danych wejściowych, które przetwarza. Dane mogą być zdefiniowane wewnątrz skryptu/funkcji ale czasami istnieje potrzeba wprowadzenia danych zewnętrznych lub dokonania wyboru podczas pracy skryptu/programu. Bash posiada funkcję o nazwie `read`, która pozwala na pobieranie wartości z okna terminala.

Przykład:

```
#!/bin/bash
read a
echo "Wpisałeś: $aA"
```

Jeżeli nie zdefiniujemy zmiennej, której mają być przypisane dane wejściowe będą one przechowywane w zmiennej `$REPLY`.

Przykład:

```
#!/bin/bash
read
echo "Wpisałeś: $REPLY"
```

Za pomocą jednego polecenia `read` możemy wprowadzić wiele zmiennych jednocześnie. Oddzielamy je spacjami. Każdy element to fragment – token.

Przykład:

```
#!/bin/bash
read a b c
echo "Pierwszy token: $a"
echo "Drugi token: $b"
echo "Trzeci token: $c"
```

Jeżeli wprowadzimy większą liczbę „tokenów” niż zmiennych po poleceniu `read`, to wszystkie nadmiarowe dane wraz z ostatnim tokenem trafią do ostatniej zmiennej.

Przykład:

```
test@host:~$  read a b c
Nazywam się Krzysztof Ropiak
test@host:~$  echo $a
Nazywam
test@host:~$  echo $b
się
test@host:~$  echo $c
Krzysztof Ropiak
```

Możemy również wprowadzać dane wejściowe do poleceń poprzez komendę echo:

```
echo "Nasze wejście" | polecenie
```

Jest to całkiem wygodne rozwiązanie pod warunkiem, że nie wprowadzamy kilku linii danych jednocześnie. Wtedy lepszym rozwiązaniem będzie zastosowanie zapisu „<< ETYKIETA".

Przykład:

```
#!/bin/sh
echo "Sortuję liczby rosnąco"
sort -n << LICZBY
120
10
2006
314159
0
LICZBY
```

wynikiem będzie:

```
Sortuję liczby rosnąco
0
10
120
2006
314159
```

Wprowadzanie wartości poprzez argumenty skryptu

W podpunkcie opisującym rodzaje i sposób operowania na zmiennych zostały wspomniane zmienne specjalne. Jednymi z tych zmiennych są zmienne przechowujące wartości argumentów podanych podczas uruchamiania skryptu.

Przykład:

```
#!/bin/bash

echo „Witaj $1 !"


test@host ~$ ./skrypt.sh Krzysztof
Witaj Krzysztof !
```

1.6. Instrukcja warunkowa if

Poznaliśmy już polecenie test, które sprawdza czy określony warunek jest prawdą czy fałszem. Jednak dopiero w połączeniu z instrukcją warunkową pokazuje swoje pełne możliwości. Instrukcja `if` pozwala na sterowanie wykonaniem w zależności od wyniku testu lub spełnienia warunku wyrażenia (czyli mamy tylko dwie możliwości – true oraz false).

Istnieje kilka postaci instrukcji `if`. Najprostsza z nich wygląda tak:

```
if polecenie_warunek; then
  instrukcje
fi
```

Przykład:

```
#!/bin/bash
echo „Wprowadź dowolną cyfrę”
read a
if [ $a -lt 5 ] ; then
echo „Wprowadzona cyfra jest mniejsza niż 5”
fi
```

W powyższym przykładzie wykonujemy czynność tylko wtedy gdy warunek zostanie spełniony. Możemy wykonać różne czynności w zależności czy warunek jest prawdziwy czy nie:

```
if polecenie_warunek; then
  instrukcje1
else
  instrukcje2
fi
```

Przykład:

```
#!/bin/bash
echo „Wprowadź dowolną cyfrę”
read a
if [ $a -lt 5 ] ; then
echo „Wprowadzona cyfra jest mniejsza niż 5”
else
echo „Wprowadzona cyfra jest większa niż 5”
fi
```

Pełna postać instrukcji `if` wygląda następująco:

```
if warunek1; then
  instrukcje1
elif warunek2; then
  instrukcje2;
...
else
  instrukcje_else;
fi
```

Przykład:

```
#!/bin/bash
echo „Wprowadź dowolną cyfrę”
read a
if [ $a -lt 5 ] ; then
echo „Wprowadzona cyfra jest mniejsza niż 5”
elif [ $a -gt 9 ]; then
echo „Wprowadzona wartość jest za duża”
else
echo „Wprowadzona cyfra jest większa niż 5”
fi
```

1.7. Instrukcja case

Jeżeli mamy wiele warunków do rozpatrzenia lepszym rozwiązaniem niż instrukcja if będzie instrukcja case.

Składnia:

```
case warunek in
pattern|pattern...)
polecenia
;;
pattern|pattern...)
polecenia
;;
Esac
```

Przykład:

```
#!/bin/bash

echo "Wybierz opcję"
echo "1) Opcja-1"
echo "2) Opcja-2"
echo "3) Opcja-3"

read wybor

case $wybor in
1) echo "Wybrałeś 1";;
2) echo "Wybrałeś 2";;
3) echo "Wybrałeś 3";;
*) echo "Wybrałeś nieznaną opcję" ;;
esac
```

Powyższy skrypt wyświetla listę opcji do wyboru, które są obsługiwane przez skrypt a następnie instrukcja `case` porównuje wprowadzoną wartość ze zdefiniowanymi wzorcami. Jeżeli znajdzie pasujący wtedy wykonuje instrukcję do niego przypisaną a w przeciwnym wypadku wykonuje instrukcję „domyślną” oznaczoną `*)`.

Inny przykład:

```
#!/bin/bash
echo „Czy masz ukończone 18 lat ?”
read wiek
case $wiek in
[Tt]ak|[Yy]es|[Tt]rue)
        echo "Możesz wejść.”
        ;;
[Nn]ie|[Nn]o|[Ff]alse)
        echo "Brak dostępu."
        ;;
esac
```

1.8. Pętle

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
    polecenia
done
```

Przykład:

```
#!/bin/bash
a=1
while test $a -le 8; do
    echo "\$a=$a"
    a=$(expr $a + 1)
done
```

Prześledźmy skrypt krok po kroku.

1. Zmienna a otrzymuje wartość 1.
2. Interpreter rozpoczyna wykonanie pętli while i sprawdza warunek $a –le 8 czyli 1 < 8 – wynik to true.
3. Skoro warunek jest spełniony zostaną wykonane instrukcje wewnątrz pętli while. Wypisanie aktualnej wartości zmiennej $a oraz zwiększenie jest wartości o 1. Więc $a = 2.
4. Ponownie sprawdzany jest warunek – 2 < 8 – prawda. Przejdź do punktu 3. Itd. Aż do momentu gdy $a = 8 gdyż wtedy warunek 8 < 8 nie zostanie już spełniony i instrukcje wewnątrz pętli nie wykonają się a interpreter wykona instrukcje za słowem `done` o ile takie istnieją.

Pętle while stosowane są najczęściej wtedy gdy nie wiadomo ile może być jej przebiegów (ile razy będzie się wykonywać) np. gdy wyjście z pętli uwarunkowane jest podaniem przez użytkownika odpowiedniej opcji lub wysłaniem przez skrypt kodu sterującego poleceniem exit.

Przykład:

```
#!/bin/bash

while [ 1 ]
do
echo "Podaj haslo dostepu, q - wychodzi ze skryptu"
echo -n "Haslo: "
read haslo
```

```
case $haslo in
"admin") echo "Hasło poprawne, witaj adminie" ; break ;;
"student") echo "Hasło poprawne, witaj studencie" ; break ;;
"q") echo "Wyjscie" ; exit ;;
*)continue ;;
esac
done
```

Pętla until.

Sprawdza czy warunek jest prawdziwy, gdy jest fałszywy wykonywane jest polecenie lub lista poleceń zawartych wewnątrz pętli, miedzy 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.

Pętla for

Pętla for różni się od while i until. Jest stosowana do kilkakrotnego powtórzenia fragmentu kodu. Ilość powtórzeń jest stała i określona z góry.

Pętla for może być używana na dwa różne sposoby.

Najprostsza składnia to:

```
for zmienna in lista; do
polecenia
done
```

Wartość zmiennej jest ustawiana na wartość pierwszej pozycji z listy i wykonywane są polecenia w pętli. Następnie zmienna jest ustawiana na wartość drugiej pozycji z listy i ponownie wykonywane są polecenia w pętli. Wykonywanie pętli kończy się po wyczerpaniu listy.

Przykład:

```
#!/bin/bash
for owoc in Jablko Pomarancza Cytryna
do
echo $owoc
done
```

wynik działania skryptu:

```
Jablko
Pomarancza
Cytryna
```

Przykład:

```
#!/bin/bash
for a in $(ls /home/kropiak); do
    echo "$a"
done
```

Jaki będzie wynik działania powyższego skryptu ?

Inna składnia pętli for:

```
for ((zmienna;warunek;zmiana)); do
polecenia
done
```

- zmienna: początkowa wartość zmiennej,

- warunek: pętla wykonywana jest tak długo, dopóki warunek jest prawdziwy,

- zmiana: na koniec pętli, wartość zmiennej jest zmieniana w sposób tu określony.

Przykład:

```
#!/bin/bash
for ((a=10; a<=15; a++)); do
    echo "$a"
done
```

Ile razy wykona się w/w pętla ?

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 odpowiadająca wybranej przez nas pozycji w menu. Jeśli nic nie wpiszemy 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
```

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 interesującej nas opcji, a resztę zrobi instrukcja `case`. Polecenie `break`, które znajduje się w przedostatniej linii skryptu, kończy pracę pętli.

Przerwanie pętli

Jeżeli istnieje konieczność przerwania pętli przed jej określoną liczbą przebiegów lub przed wystąpieniem warunku wyjścia z pętli wtedy należy posłużyć się poleceniem `break`.

Przykład:

```
#!/bin/bash
for owoc in Jablko Pomarancza Cytryna
do
echo $owoc
if [ "$owoc" = "Pomarancza" ]
then
break
fi
done
```

wynikiem będzie:

```
Jablko
Pomarancza
```

Continue

Polecenie `continue` wymusza przejście do kolejnej iteracji, czyli następnego kroku.

```
#!/bin/bash
for owoc in Jablko Pomarancza Cytryna
do
echo $owoc
if [ "$owoc" = "Pomarancza" ]
then
continue
fi
echo „To nie wykona się po Pomarańczy”
done
```

W napisanym przez nas kodzie zadaniem instrukcji continue jest ukrycie napisu „To nie wykona się po Pomarańczy” podczas przetwarzania „owocu”: Pomarancza.

Jablko
To nie wykona się po Pomarańczy
Pomarancza
Cytryna
To nie wykona się po Pomarańczy

1.9. Definiowanie nowych funkcji w skrypcie powłoki

Funkcje to właściwie gotowe podprogramy – moduły, które można umieszczać w różnych miejscach programu czy programów. Reprezentują często występujące, powtarzające się elementy różnych programów. Kod programu funkcji jest wyodrębniony z programu podstawowego. Funkcja może być wywoływana z głównego programu w dowolnym momencie, w razie potrzeby.

Składnia:

```
nazwa() {

polecenia

}
```

Funkcja wywoływana jest z głównego programu przez swoją nazwę. Można też dodać kilka parametrów.

Przykład:

```
#!/bin/bash
sum() {
        a=$(expr $1 + $2)
        echo "$1+$2="$a
}
sum 4 5
sum 2 9
sum 12 1
```

Parametry pozwalają przekazanie informacji do funkcji. Do przekazania informacji z funkcji do głównego programu użyć można polecenia `return` z odpowiednim numerem.