2.5 Podstawy programowania w bash'u

Powłoka Linuksa jest programowalna, można używać edytora vi do pisania programów.

Zwykle skrypt powłoki jest zapisany w pliku tekstowym i zaczyna się następującym wierszem (zwanym *shee-bang line*):

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

Po napisaniu, a przed próbą bezpośredniego uruchomienia skryptu konieczne jest nadanie mu atrybutu wykonywalności (plik wykonywalny) poleceniem:

```
chmod u+x skrypt_bash
```

Wewnątrz skryptu można stosować wszystkie polecenia omawiane wcześniej w podręczniku. Polecenie echo możemy stosować do wyświetlania tekstu na ekranie lub żądania zawartości zmiennej.

Przykład:

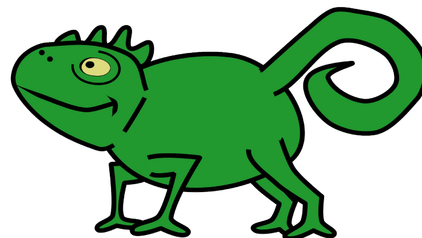
```
#!/bin/bash
echo 'The content of variable $PATH is:' $PATH
```

Zwykle znak **#** jest stosowany do oznaczania tekstu komentarza. Komentarze nie są interpretowane i wykonywane.

Jedynym wyjątkiem od tj zasady jest wiersz *shee-bang*.

Po wykonaniu skryptu, możemy przykładowo otrzymać wyjście w postaci:

```
geeko@da51:~> ./bash_script
The content of variable $PATH is: /home/geeko/bin:/usr/local/bin:/usr/bin:/usr/X
11R6/bin:/bin:/usr/games:/opt/gnome/bin:/opt/kde3/bin:/usr/lib/jvm/jre/bin:/usr/
lib/mit/bin:/usr/lib/mit/sbin
geeko@da51:~>
```



2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

2.5.1 Zmienne w skryptach bash

W ostatnim przykładzie używaliśmy zmiennej $PATH. Można definiować swoje własne zmienne wyłącznie do użytku wewnątrz danego skryptu.

Składnia: nazwa_zmiennej=wartość

Obowiązuje zasada używania małych liter dla nazw zmiennych wewnątrz kodu programu.

Tylko stałe (które nie zmieniają wartości podczas wykonywania programu) są pisane dużymi literami.

```
#!/bin/bash
myvar=13
echo 'The content of variable $myvar is:' $myvar
```

By pobrać zawartość zmiennej (np. poleceniem echo) trzeba dodać znak $ przed nazwą zmiennej.

Zmienne mogą przechowywać liczby, znaki, ciągi tekstowe.

```
#!/bin/bash

a=2
b=x
c=Geeko
d="Geeko and Suzie"

echo '$a is a number:' $a
echo '$b is a character:' $b
echo '$c is a string:' $c
echo '$d is a string with whitespaces:' $d
```

Wynik wykonania skryptu:

```
$a is a number: 2
$b is a character: x
$c is a string: Geeko
$d is a string with whitespaces: Geeko and Suzie
```

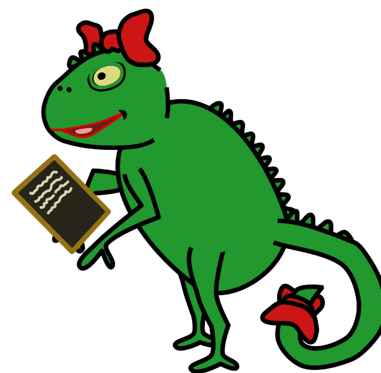
2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Ćwiczenie. Typy zmiennych

Liczby mogą być dodawane, ciągi tekstowe nie. Dla wielu operacji jest to ważne, więc należy odpowiednio używać tych typów wartości zmiennych.

Zapisz poniżej po cztery typowe operacje dla liczb i ciągów znaków.

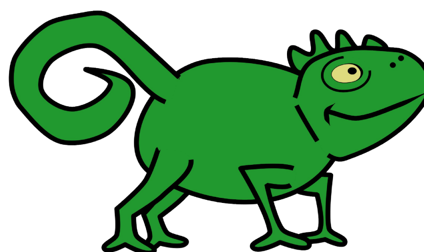
operacje dla liczb:	operacje dla ciągów znaków:
______________________	______________________
______________________	______________________
______________________	______________________
______________________	______________________



2.5.2 Obliczenia numeryczne

Zmienne często są łączone z operatorami matematycznymi, służą do różnych obliczeń. Zmienne liczbowe mogą być:

- dodawane (operator „+”)
- odejmowane (operator „-”)
- mnożone (operator „*”)
- dzielone (operator „/”)
- dzielone modulo (operator „%”)



2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Obliczenia na liczbach całkowitych

W poniższym przykładzie zapiszemy sumę zmiennych $a i $b do nowej zmiennej:

```
#!/bin/bash
a=3
b=2
c=$a+$b
echo '$a+$b='$c
```

Wynikiem może być przykładowo:

```
$a+$b=3+2
```

Jak widać, zmienna C zawiera łańcuch „3 + 2,” a nie sumę „5”! By uzyskać sumę mamy następujące możliwości:

- Określić, że zmienna $c może przechowywać wyłącznie liczby (dane numeryczne).

  By to zrobić, należy użyć do jej deklaracji - funkcji **declare** z opcja **-i :**

```
#!/bin/bash
a=3
b=2
declare -i c=$a+$b
echo '$a+$b='$c
```

- Oznaczyć działanie (formułę) jako wyrażenie, które powinno być wykonane:

  By wymusić obliczenie w wierszu poleceń, należy użyć polecenia **expr**:

```
geeko@da51:~ > expr 3 + 2
5
geeko@da51:~ >
```

Uwaga. W odróżnieniu od metody z funkcją declare, przy funkcji expr muszą być spacje oddzielające operatory od argumentów.

Jeżeli użyjemy funkcji expr w skrypcie powłoki, musimy zamknąć całe wyrażenie w cudzysłowie pojedynczym („' ”) lub poprzedzić znakiem $.

Znaki specjalnego przeznaczenia muszą być maskowane ukośnikiem „\”.

```
#!/bin/bash
a=3
b=2
c=$(expr $a + $b)
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
echo '$a+$b='$c
```

Wyrażenie wewnątrz $(...) jest zastępowane wynikiem działania.

W rezultacie otrzymamy, przykładowo:

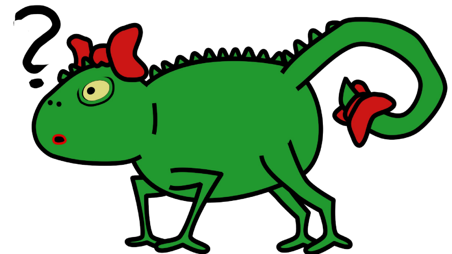
```
$a+$b=5
```

Spróbujmy przeanalizować przykład z dzieleniem zmiennych.

```
#!/bin/bash
a=3
b=2
c=$(expr $a / $b)
echo '$a/$b='$c
```

Wynik jest może być niezgodny z oczekiwanym:

```
$a/$b=1
```



Wynik dzielenia jest liczbą całkowitą – bez miejsc po przecinku. Resztę dzielenia możemy otrzymać jeżeli użyjemy operatora modulo:

```
#!/bin/bash
a=3
b=2
c=$(expr $a / $b)
d=$(expr $a % $b)
echo '$a/$b='$c Rest:$d
```

Zamiast funkcji expr, można użyć składni $((...))

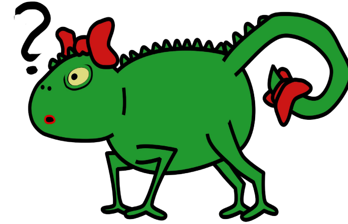
Przykład:

```
#!/bin/bash
a=3
b=2
c=$(($a + $b))
echo '$a+$b='$c
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Ćwiczenie. Obliczanie na liczbach całkowitych

Napisz skrypt bash, który podnosi do kwadratu liczby: 6, 9 oraz 12.



Obliczenia na liczbach zmiennoprzecinkowych

Domyślnie, bash nie może wykonywać obliczeń zmiennoprzecinkowych.

```
#!/bin/bash
a=3
b=2.7
c=$(expr $a + $b)
echo '$a+$b='$c
Ten skrypt da w wyniku:
expr: non-numeric argument
$a+$b=
```

By problem rozwiązać, należy użyć dodatkowego narzędzia powłoki o nazwie bc. Jest to dokładny kalkulator, którego można używać w wierszu poleceń.

```
1  geeko@da51:~> bc
2  bc 1.06
3  Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
4  This is free software with ABSOLUTELY NO WARRANTY.
5  For details type `warranty'.
6  3+4
7  7
8  (10+4)/2
9  7
10 2+0.44
11 2.44
12 7/3
13 2
14 quit
```

Wyjaśnienie. Poleceniem bc uruchamiamy kalkulator iwprowadzamy wyrażenie do obliczenia (wiersze 6,8,10 i 12). By zobaczyć wynik obliczeń, należy nacisnąć Enter na klawiaturze (wiersze 7,9, 11 i 13). Wyrażenia można grupować stosując

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

nawiasy (wiersz 8). Można używać liczb zmiennoprzecinkowych (wiersz 10). Domyślnie, bc używa tego samego przetwarzania przy dzieleniu (bez miejsc po przecinku), co powłoka (wiersze 12+13). By zakończyć pracę z kalkulatorem, należy napisać quit (wiersz 14).

bc ma własny język programowania, ale to tutaj nie ma znaczenia. By używać **bc** wewnątrz skryptu bash, używa się prostego „wybiegu”: potokuje wyrażenie w **bc**.

Polecenie echo i potok „|”:

```
geeko@da51:~> echo "2.4+7" | bc
9.4
```

By zapamiętać wynik obliczeń kalkulatorem bc, wyrażenie umieszcza się w pojedynczym odwrotnym cudzysłowie:

```
geeko@da51:~> a=`echo "2.4+7" | bc`
geeko@da51:~> echo $a
9.4
```

Zamiast cudzysłowia, można użyć składni **$(...)**:

```
geeko@da51:~> a=$(echo "2.4+7" | bc)
geeko@da51:~> echo $a
9.4
```

Nie ma znaczenia, którą składnię wybierzesz. W podręczniku używamy tej drugiej.

By zobaczyć miejsca po przecinku, napisz **scale=*liczba*** przed operacją matematyczną oddzie średnikiem „;”.

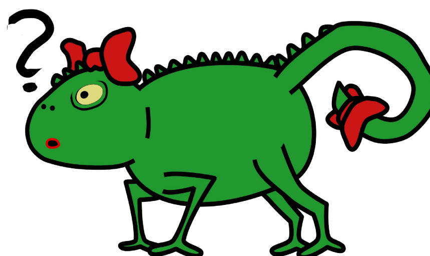
Przykład:

```
geeko@da51:~> echo "scale=2; 9/2" | bc
4.50
```

Ćwiczenie.

Obliczanie na liczbach zmiennoprzecinkowych

Napisz skrypt bash, który podnosi do kwadratu liczby: 1,5; 2,75 oraz 7,2.



2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

2.5.3 Porównywanie plików, łańcuchów i liczb

Do odkrycia, że 3 jest większe niż 2 niekoniecznie trzeba pisać skrypt bash'a ;-).

Jednak w interakcyjnych programach pojawia się wiele pytań i akcja programu zależy od odpowiedzi na nie.

Mogą to być przykładowo pytania następujące:

- Czy istnieje plik *nazwa_pliku*?
- Kiedy plik został zapisany?
- Jak odpowiedział użytkownik na określone pytanie?
- Czy wartość zmiennej jest w określonym, dopuszczalnym, zakresie?
- Ile razy została wykonana dana pętla?

Do sprawdzania odpowiedzi służy polecenie **test**. Wynikiem polecenia **test** jest albo „**0**” (prawda) albo „**1**” (fałsz). Wynik ten jest przechowywany w zmiennej „**$?**”.

Polecenie test ma bardzo dużo parametrów.

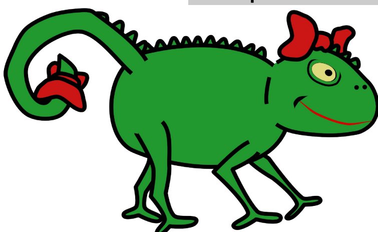
W pierwszej kolejności omówimy parametry dotyczące plików:

- **-d *cel*** sprawdza, czy *cel* jest istniejącym katalogiem:

```
#!/bin/bash

a=/var
b=/etc/profile

test -d $a
echo $a: $?

test -d $b
echo $b: $?
```

Poniżej podano wynik wykonania polecenia :

```
/var: 0
/etc/profile: 1
```



2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

- **-e *cel*** sprawdza, czy *cel* jest istniejącym plikiem:

```
#!/bin/bash

a=/var
b=2

test -e $a
echo $a: $?

test -e $B
echo $b: $?
```

Wynikiem może być:

```
/var: 0
2: 1
```

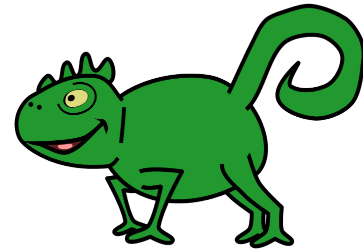
- **-f *cel*** sprawdza, czy *cel* istnieje i jest normalnym plikiem:

```
#!/bin/bash

a=/var
b=/etc/profile

test -f $a
echo $a: $?

test -f $b
echo $b: $?
```

Wynik wygląda mniej więcej tak:

```
/var: 1
/etc/profile: 0
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*



- **-r *cel,* -w *cel,* -x *cel*** sprawdza, czy użytkownik ma określone uprawnienia:

  odczytu (-r),

  zapisu (-w),

  wykonania (-x)

```
#!/bin/bash

a=/etc/profile

test -r $a
echo "Read: $?"

test -w $a
echo "Write: $?"

test -x $a
echo "Execute: $?"
```

Plik /etc/profile ma następujące uprawnienia: rw-r—r-- . Użytkownik Geeko może tylko czytać ten plik.

Rezultat polecenia będzie miał postać:

```
Read: 0
Write: 1
Execute: 1
```

- **-s *cel*** sprawdza, czy plik jest pusty:

```
#!/bin/bash

a=/etc/profile

test -s $a
echo $?
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Rezultatem wykonania polecenia **test** powinno być „0”.

plik1* -nt *plik2 sprawdza, czy plik ***plik1*** jest nowszy od ***plik2***. Porównuje daty ostatniej modyfikacji, a nie utworzenia.

```
#!/bin/bash

a=/etc/profile
b=/var/log/Xorg.0.log

test $a -nt $b
echo "$a -nt $b:" $?
```

Wynik powinien być równy „1” (fałsz):

```
/etc/profile -nt /var/log/Xorg.0.log: 1
```

plik1* -ot *plik2*,** odwrotnie niż **-nt,** sprawdza, czy plik ***plik1 jest starszy od ***plik2***. Porównuje daty ostatniej modyfikacji, a nie utworzenia.

```
#!/bin/bash

a=/etc/profile
b=/var/log/Xorg.0.log

test $a -ot $b
echo "$a -ot $b:" $?
```

Wynik powinien być „1” (fałsz):

```
/etc/profile -ot /var/log/Xorg.0.log: 0
```

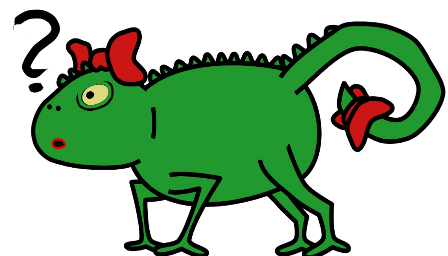
Ćwiczenie. Porównywanie plików, łańcuchów, liczb – część I

Napisz skrypt bash, który...

... sprawdza, czy istnieje plik /var/log/messages,

... czy jest on pusty, oraz

... wyświetla jego prawa dostępu.



2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Test może sprawdzać i porównywać ciągi znaków.

- **-z *łańcuch*** polecenie sprawdza, czy łańcuch znaków jest pusty: wynik jest równy „0” (prawda), gdy ciąg znaków łańcuch jest zerowy:

```
#!/bin/bash

a=geeko

test -z $a
echo "a: $?"

test -z $b
echo "b: $?"
```

Wynik polecenia tym przypadku będzie wygladał następująco:

```
a: 1
b: 0
```

Wyjaśnienie. Ponieważ zmienna $b nie jest zdefiniowana, ma długość =”0” i wynikiem sprawdzenia jest „prawda”.

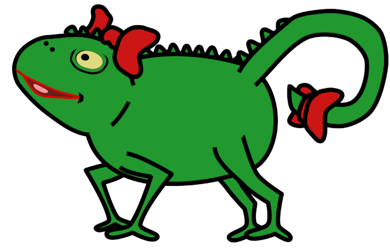
- **-n *łańcuch,*** polecenie odwrotne do „-z”, sprawdza, czy łańcuch znaków jest niepusty: wynik jest równy „0” (prawda), gdy łańcuch zawiera co najmniej jeden znak.

```
#!/bin/bash

a=geeko

test -n $a
echo "a: $?"

test -n $b
echo "b: $?"
```

Wynik działania skryptu:

```
a: 0
b: 1
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*



łańcuch1 == łańcuch2 wynikiem jest „prawda” („0”), gdy ciągi znaków są identyczne.

```
1  #!/bin/bash
2
3  a=geeko
4  b=suzie
5  c=geeko
6
7  test $a == $b
8  echo "$a == $b: $?"
9
10 test $a == $c
11 echo "$a == $c: $?"
```

Wyjaśnienie. Zmienne są definiowane w wierszach 3-5, zmienne $a oraz $c są takie same. w wierszu 7 są porównywane zmienne $a i $b, a w wierszu 10 - porównywane są zmienne $a i $c.

Wynik:

```
geeko == suzie: 1
geeko == geeko: 0
```

łańcuch1* != *łańcuch2 wynikiem jest „prawda” („0”), gdy ciągi znaków są różne.

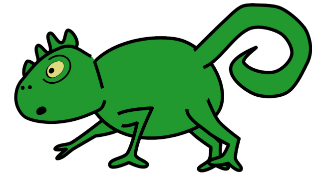
```
#!/bin/bash

a=geeko
b=suzie
c=geeko

test $a != $b
echo "$a != $b: $?"
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
test $a != $c
echo "$a != $c: $?"
```

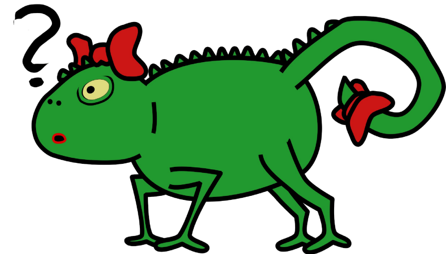
Wynik:

```
geeko != suzie: 0
geeko != geeko: 1
```



Ćwiczenie. Porównywanie plików, łańcuchów, liczb – część II

Napisz skrypt bash, który sprawdza czy łańcuch „Linux” i łańcuch „edukacja” są równe (identyczne).



Jest też kilka operatorów do porównywania liczb.

- ***liczba1* -eq *liczba2*** sprawdza czy obie liczby są równe.

```
#!/bin/bash

a=3
b=2

test $a -eq $b
echo "$a -eq $b: $?"
```

wynik:

```
3 -eq 2: 1
```

- ***liczba1* -ne *liczba2*** sprawdza czy obie liczby są różne.

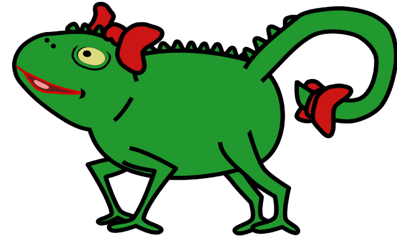
```
#!/bin/bash

a=3
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
b=2

test $a -ne $b
echo "$a -ne $b: $?
```

wynik:

```
3 -ne 2: 0
```



- ***liczba1* -gt *liczba2*** sprawdza czy *liczba1* jest większa niż *liczba2.*

```
#!/bin/bash

a=3
b=2

test $a -gt $b
echo "$a -gt $b: $?"

test $b -gt $a
echo "$b -gt $a: $?"
```

wynik:

```
3 -gt 2: 0
2 -gt 3: 1
```

- ***liczba1* -ge *liczba2*** sprawdza czy *liczba1* jest większa lub równa niż *liczba2.*

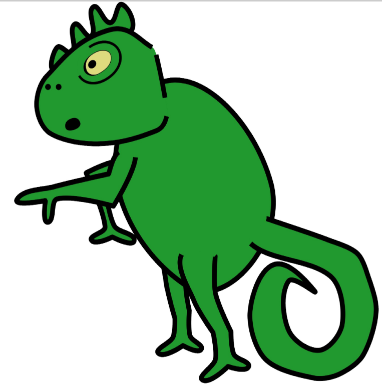
```
#!/bin/bash

a=3
b=2
c=3

test $a -ge $b
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
echo "$a -ge $b: $?"

test $a -ge $c
echo "$a -ge $c: $?"
```

wynik testu:

```
3 -ge 2: 0
3 -ge 3: 0
```



- ***liczba1* -lt *liczba2*** sprawdza czy *liczba 1* jest mniejsza niż *liczba2.*

```
#!/bin/bash
a=3
b=2

test $a -lt $b
echo "$a -lt $b: $?"

test $b -lt $a
echo "$b -lt $a: $?"
```

wynik:

```
3 -lt 2: 1
2 -lt 3: 0
```

- ***liczba1* -le *liczba2*** sprawdza czy *liczba 1* jest mniejsza lub równa niż *liczba2.*

```
#!/bin/bash
a=3
b=2
c=3
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

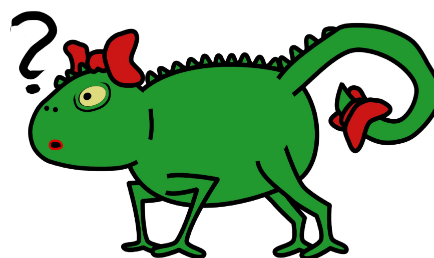
```
test $a -le $b
echo "$a -le $b: $?"

test $a -le $c
echo "$a -le $c: $?"
```

wynik testu:

```
3 -le 2: 1
3 -le 3: 0
```

Ćwiczenie. Porównywanie plików, łańcuchów, liczb – część III

Napisz skrypt bash, który sprawdza czy liczba „13” jest większa od liczby „11”.



Można również porównywać liczby zmiennoprzecinkowe.

Składnia jest inna niż typowa składnia powłoki:

- *wyrażenie1 == wyrażenie2*

  Czy *wyrażenie1* jest takie samo jak *wyrażenie2*?
- *wyrażenie1 != wyrażenie2*

  Czy *wyrażenie1* jest różne od *wyrażenie2*?
- *Wyrażenie1 > wyrażenie2*

  Czy *wyrażenie1* jest większe od *wyrażenie2*?
- *wyrażenie1 >= wyrażenie2*

  Czy *wyrażenie1* jest większe lub równe od *wyrażenie2*?
- *wyrażenie1 < wyrażenie2*

  Czy *wyrażenie1* jest mniejsze od *wyrażenie2*?
- *wyrażenie1 <= wyrażenie2*

  Czy *wyrażenie1* jest mniejsze lub równe *wyrażenie2*?

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Wszystkie te operatory pokazano w poniższym skrypcie:

```
#!/bin/bash
a=3.7

b=$(echo "$a == 3.7" | bc)
echo "$a == 3.7: $b"

b=$(echo "$a != 3.7" | bc)
echo "$a != 3.7: $b"

b=$(echo "$a > 3.7" | bc)
echo "$a > 3.7: $b"

b=$(echo "$a >= 3.7" | bc)
echo "$a >= 3.7: $b"

b=$(echo "$a < 3.7" | bc)
echo "$a < 3.7: $b"

b=$(echo "$a <= 3.7" | bc)
echo "$a <= 3.7: $b"
```

Wyniki skryptu:

```
3.7 == 3.7: 1
3.7 != 3.7: 0
3.7 > 3.7: 0
3.7 >= 3.7: 1
3.7 < 3.7: 0
3.7 <= 3.7: 1
```

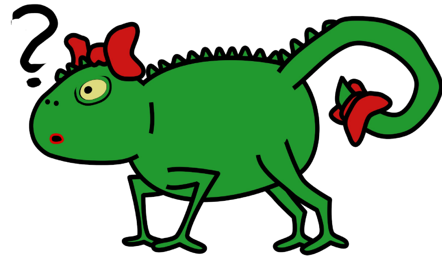
Wynik polecenia **bc** w przypadku testowania wyrażeń zmiennoprzecinkowych ma inną wartość logiczną niż w bash: „0” oznacza fałsz, a „1” to prawda.

Ćwiczenie. Porównywanie plików, łańcuchów, liczb – część IV

Napisz skrypt bash, który sprawdza czy liczba „13,5” jest większa od

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

liczby „22”.



Czasem występuje konieczność kombinacji kilku porównań (testów), na przykład by sprawdzić czy wartość jednej zmiennej jest większa od jednej liczby, a mniejsza od innej.

Są dwie możliwości takich kombinacji:

- **Alternatywa logiczna LUB (OR)**.

Test1 zwraca „0” (prawda) lub Test2 zwraca „0” (prawda)

		Test1	
		PRAWDA	FAŁSZ
Test2	PRAWDA	PRAWDA	PRAWDA
	FAŁSZ	PRAWDA	FAŁSZ

Użyj parametru **-o** by utworzyć połączenie LUB między dwoma testami:

```
#!/bin/bash

a=/var
b=/etc/profile

test -f $a -o -f $b
echo "Result: $?"
```

Wyjaśnienie. Skrypt sprawdza, czy plik /var LUB /etc/profile jest normalnym plikiem. Dla /var wynik jest logicznym fałszem, bo /var jest katalogiem. Ale /etc/profile jest plikiem. Więc wynik alternatywy jest „0” czyli PRAWDA.

- **Iloczyn logiczny I (AND)**.

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Test1 zwraca „0” (prawda) lub Test2 zwraca „0” (prawda)

		Test1	
		PRAWDA	FAŁSZ
Test2	PRAWDA	PRAWDA	FAŁSZ
	FAŁSZ	FAŁSZ	FAŁSZ

Użyj parametru **-a** by utworzyć połączenie I (AND) między dwoma testami:

```
#!/bin/bash

a=7
b=2

test $a -gt $b -a $(expr $a % 2) -eq 0
echo "Result: $?"
```

Wyjaśnienie. Skrypt sprawdza czy wartość $a jest większa niż wartość $b (7>2, prawda) **I** czy $a jest parzysta (fałsz). Ponieważ druga część iloczynu logicznego jest fałszywa - test zwraca wartość logiczną FAŁSZ.

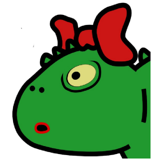
- **logiczna NEGACJA (NOT)**. Negację reprezentuje znak „ **!”**.

```
#!/bin/bash

a=7
b=2

test ! $a -gt $b
echo "Result: $?"
```

Wyjaśnienie. 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.

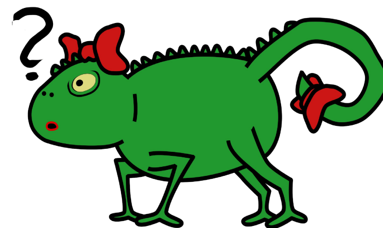
2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

W **bc** iloczyn logiczny AND zapisuje się przez „**&&**”, alternatywę OR w postaci „**||**”. Z kolei negację NOT określa również znak „**!**”.



Ćwiczenie. Porównywanie plików, łańcuchów, liczb – część V

Napisz skrypt bash, który sprawdza czy liczba „13,5” jest większa od liczby „22” lub mniejszy niż „15”



2.5.4 Czytanie z Wejścia

Nie ma większego sensu pisanie ośmiowierszowego programu bash celem porównania dwóch liczb. No i pisanie go na nowo, dla każdych innych liczb...

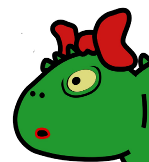
Wygodniej poprosić użytkownika o wprowadzenie danych – wartości zmiennych; natomiast skrypt wykona na nich zdefiniowane operacje.

Polecenia **read** użyjemy do czytania wprowadzanych danych. Program ten czeka, aż użytkownik zakończy wprowadzanie danych naciśnięciem klawisza **Enter**. Można podać poleceniu zmienną, która ma przechowywać wprowadzane informacje.

```
#!/bin/bash

read a
echo "You entered: $aA"
```

Wyjaśnienie. Uruchomiony poleceniem read program czeka na wprowadzenie danych - wejście (input). Wprowadzone dane będą pamiętane w zmiennej $a. Po zakończeniu wprowadzania naciśnięciem klawisza Enter, wprowadzone dane zostaną wyświetlone na ekranie (polecenie echo).

Jeżeli w poleceniu **read** nie podamy zmiennej, w której ma być przechowane „Wejście”, jest one zapamiętane w zmiennej **$REPLY**.



W poleceniu **read** można podać więcej zmiennych. Oddziela się je spacjami,

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

a dane nazywa fragmentami - „*tokenami*”. Pierwszy „token” jest przechowywany w pierwszej zmiennej, drugi – w drugiej, i tak dalej odpowiednio.

```
#!/bin/bash

read a b c
echo "First token: $a"
echo "Second token: $b"
echo "Third token: $c"
```

Pozostały tekst trafia do ostatniej zmiennej. Przykładowo – jeżeli wprowadzimy z konsoli tekst „ This is a short text”, efektem będzie, co następuje:

```
First token: This
Second token: is
Third token: a short text.
```

Jeżeli jest więcej zmiennych niż tokenów tekstowych – pozostałe zmienne są puste.

Ćwiczenie. Czytanie z Wejścia

Napisz skrypt bash, który czyta dwie całkowite liczby z Wejścia i sprawdza czy pierwsza jest większa od drugiej.

2.5.5Sprawdzanie przypadków

bash umożliwia dwa sposoby kontroli wykonywania programów:

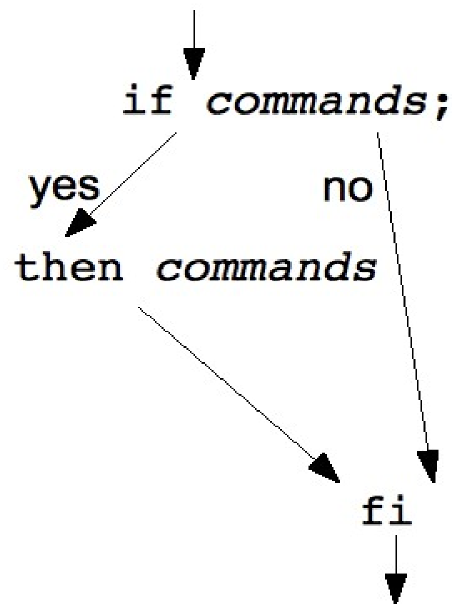
- **if... fi ,**
- **case...esac.**

Jeżeli chcemy rozróżnić między dwoma przypadkami, użyjemy pierwszej możliwości.

Najprościej zrobić to używając następującej składni:

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

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*



W pierwszej kolejności sprawdzane jest polecenie po „if”. Jeżeli wynik jest „0” (prawda), wykonywane są polecenia po „then”. Jeżeli wynikiem polecenia po „if” jest „1” (fałsz), nic się nie dzieje i kontynuowany jest program po etykiecie „fi”.

W większości przypadków, polecenia po „if” sprawdzane są poleceniem test.

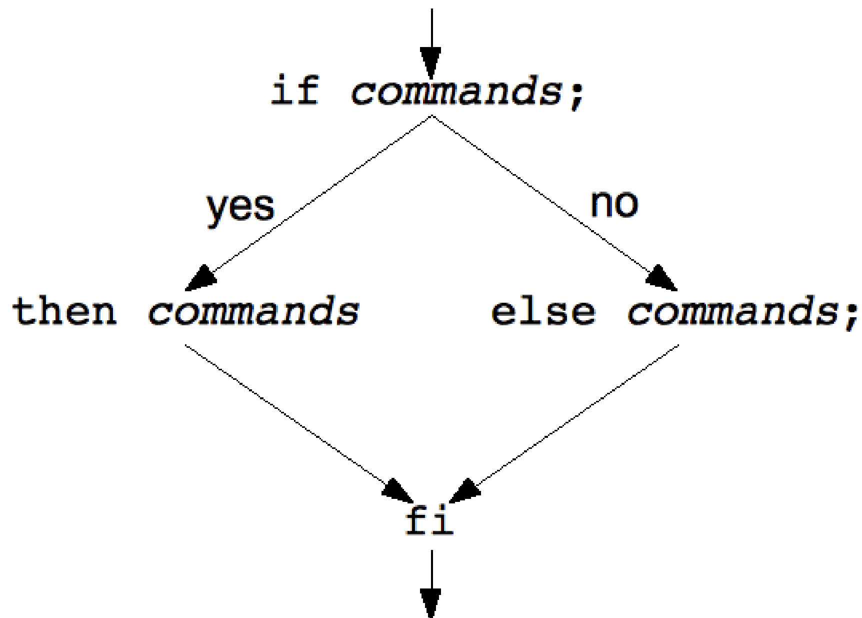
```
1 #!/bin/bash
2
3 echo "Enter something:"
4 read a
5
6 if test -e $a; then
7     echo "$a is an existing file."
8 fi
```

Wyjaśnienie. Program prosi użytkownika o wprowadzenie czegokolwiek z konsoli (wiersze 3-4). następnie sprawdza, czy wprowadzony łańcuch znaków jest istniejącym plikiem (wiersz 6). Jeżeli tak, odpowiedni komunikat jest wyświetlany na ekranie (wiersz 7).

Można dodać wyrażenie „else”, by otrzymać:

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

if *polecenia*; then
polecenia
else
polecenia
fi



Polecenia po „else” są wykonywane, gdy wynik „if” jest fałszem.

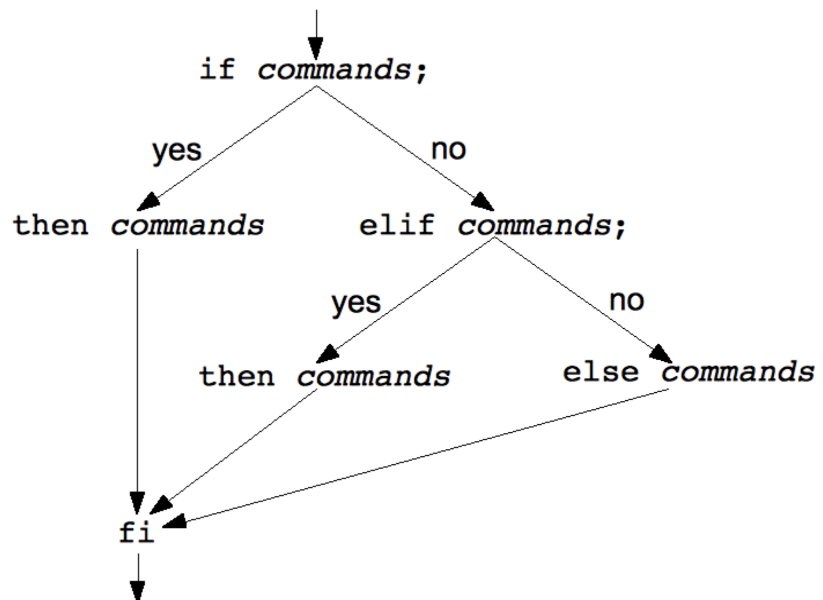
```
#!/bin/bash

echo "Enter something:"
read a

if test -e $a; then
   echo "$a is an existing file."
else
   echo "$a is no file."
fi
```

Wyjaśnienie. Poprzedni skrypt jest rozszerzony o dwa końcowe wiersze (*else...*). W tym przypadku, jeżeli wprowadzone dane nie są nazwą istniejącego pliku, na ekranie jest wyświetlany inny komunikat.

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Jeżeli chcemy sprawdzić więcej niż jeden warunek – użyjemy „elif” między poleceniem „then” a „else”:

if commands; then
commands
elif *commands*; then
commands
else *commands*
commands
fi



W pierwszej kolejności wykonywane są polecenia po „if”. Jeżeli ich wynikiem jest „0” (prawda) – to wykonywane są polecenia po „then”, a jeżeli wynik „if” to „1” (fałsz) – wykonywane są polecenia po „elif”. Jeżeli ani wynik poleceń po „if” ani wynik poleceń po „elif” nie jest prawdą – wykonywane są polecenia po „else”.

Można używać więcej niż jeden warunek „elif” budując długie łańcuchy warunków logicznych.

```
#!/bin/bash

echo "Enter something:"
read a

if test -e $a; then
   echo "$a is an existing file."
elif test $a == geeko; then
   echo "Entered geeko."
else
   echo "$a is no file and not geeko."
fi
```

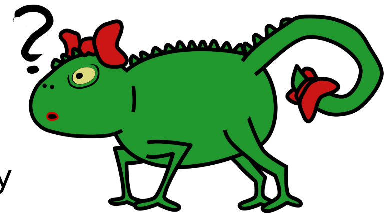
Wyjaśnienie. Poprzedni program został rozszerzony o warunek „elif”. W tym

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

przypadku wprowadzony ciąg jest sprawdzany na zgodność z „geeko” w przypadku, gdy nie jest nazwą istniejącego pliku.

Ćwiczenie. Sprawdzanie warunków – część I

Napisz skrypt bash, który czyta dwie całkowite liczby z wejścia i sprawdza...



1.... czy pierwsza liczba jest większa od drugiej,

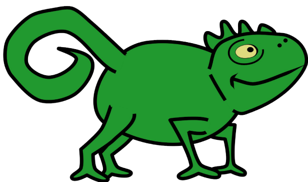
2.... czy pierwsza liczba jest równa drugiej,

3.... pierwsza liczba jest mniejsza od drugiej

W każdym z powyższych przypadków powinien by wyświetlany inny, właściwy dla danego przypadku, komunikat.

Jeżeli chcemy określić dużo różnych warunków, lepiej użyć wyrażenia „**case**” o składni:

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



Warunki są sprawdzane pod kątem dopasowania do wzorców (pattern). Jeżeli pasują do danego wzorca, jego polecenia są wykonywane, jeżeli nie – sprawdzany jest następny wzorzec.

```
#!/bin/bash

echo "Enter something:"
read a

case $a in
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
  [Yy]es|[Tt]rue)
    echo "You entered Yes/True."
    ;;
  [Nn]o|[Ff]alse)
    echo "You entered No/False."
    ;;
esac
```

Wyjaśnienie. Wprowadzany łańcuch znaków jest sprawdzany na zgodność ze słowami „Yes” lub „yes lub „True” lub „true”. Jeżeli nie jest zgodny – sprawdzany jest na zgodność ze słowami „No” lub „no” lub „False” lub „false”.

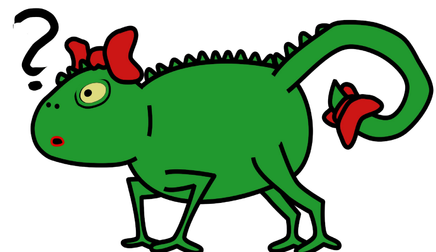
Jeżeli dany wzorzec pasuje i jego polecenia są wykonane, program wychodzi z zakresu ograniczonego słowami „case”-”esac” i kontynuowane jest wykonywanie programu po „esac”.

Jeżeli chcemy by zostały wykonane jakieś polecenia w przypadku, gdy żaden z określonych wzorców warunkówych „case” nie zachodzi, należy dodać wzorzec oznaczony gwiazdką („*”).

```
#!/bin/bash

echo "Enter something:"
read a
case $a in
  [Yy]es|[Tt]rue)
    echo "You entered Yes/True."
    ;;
  [Nn]o|[Ff]alse)
    echo "You entered No/False."
    ;;
  *)
    echo "Wrong input."
esac
```

Ćwiczenie. Sprawdzanie warunków – część II

Napisz skrypt bash, który czyta jedną jednocyfrową liczbę całkowitą z wejścia oraz wyświetla liczbę słownie na ekranie (na przykład: „Jeden” dla „1”).



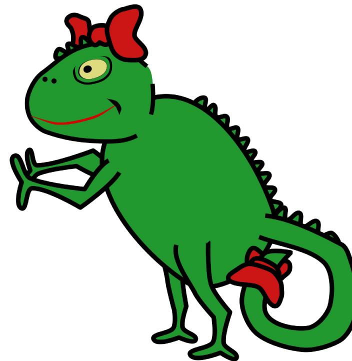
2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

2.5.6 Stosowanie pętli

Stosując pętle powodujemy kilkakrotne wykonanie części programu.

Skrypt bash rozpoznaje trzy typy pętli:

- **while**
- **until**
- **for**

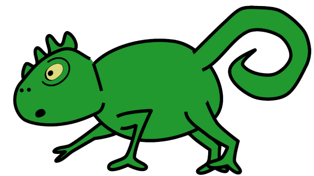


Pętla while

Składnia pętli:

```
while warunek; do
        polecenia
done
```

Zwykle *warunek* jest testem - *polecenia* wewnątrz pętli są wykonywane dopóki *warunek* jest prawdziwy.



```
#!/bin/bash


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

Wyjaśnienie. Zmiennej „a” nadano wartość „1”. Następnie sprawdzane jest czy $a jest mniejsza (lub równa) od „8”. Jeżeli wartość $a jest mniejsza od „8” - polecenia wewnątrz pętli są wykonywane: wyświetlana jest informacja

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

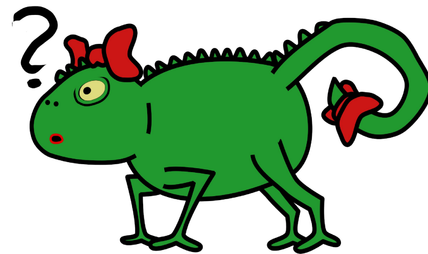
o wartości zmiennej, a następnie wartość zmiennej $a jest zwiększana o 1.

Wyjściem programu jest ciąg:

```
$a=1
$a=2
$a=3
$a=4
$a=5
$a=6
$a=7
$a=8
```

Ćwiczenie. Stosowanie pętli – część I

Napisz skrypt bash, który czyta jedną jednocyfrową liczbę całkowitą z wejścia oraz zlicza od jednego do wprowadzonej liczby. Na ekranie powinna być wyświetlana nazwa liczby, nie cyfra (na przykład: „Jeden” dla „1”). Użyj pętli **while**.



Pętla until

Pętla until jest podobna do pętli while, ma też podobną składnię:

```
until warunek; do
        polecenia
done
```

Różnica między **while** a **until** polega na tym, że przy użyciu **until** polecenia są wykonywane póki *warunek* jest prawdą.

By uzyskać taki sam wynik, jak w ostatnim przykładzie, trzeba by napisać:

```
#!/bin/bash

a=1

until test $a -gt 8; do
```

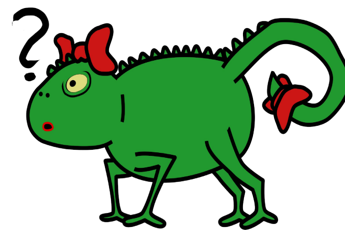
2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
   echo "\$a=$a"
   a=$(expr $a + 1)
done
```

Wyjaśnienie. **while** zastąpiono przez **until**, „-le” zastąpiono przez „-gt”.

Ćwiczenie. Stosowanie pętli – część II

Napisz skrypt bash, który czyta jedną jednocyfrową liczbę całkowitą z wejścia oraz zlicza od jednego do wprowadzonej liczby. Na ekranie powinna być wyświetlana nazwa liczby, nie cyfra (na przykład: „Jeden” dla „1”). Użyj pętli **until**.



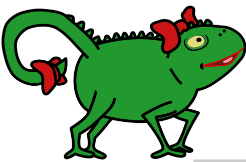
Pętla for

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.



```
#!/bin/bash

for a in Geeko Suzie 100 0 Novell; do
   echo "$a"
done
```

Wyjście programu:

```
Geeko
Suzie
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
100
0
Novell
```

Lista może być tworzona w czasie wykonywania programu:

```
#!/bin/bash

for a in $(ls /bin/b*); do
   echo "$a"
done
```

Wyjaśnienie. Jako lista użyte zostaną wszystkie pliki z katalogu /bin zaczynające się od litery „b”.

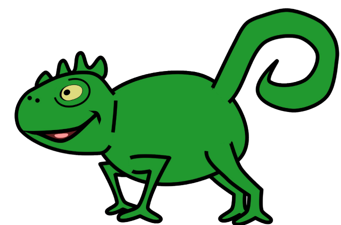
Wyjście z programu:

```
/bin/basename
/bin/bash
/bin/bluepincat
```

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:

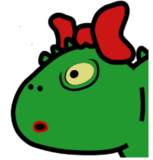
```
#!/bin/bash

for ((a=10; a<=15; a++)); do
   echo "$a"
done
```

Wyjaśnienie. Gdy pierwszy raz jest pętla wykonywana – wartość zmiennej

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

ustawiona jest na „10”. Warunek testuje, czy wartość zmiennej jest mniejsza lub równa od „15”, następnie wyświetlana jest jej wartość. Po wykonaniu polecenia - wartość zmiennej zwiększana jest o „1” i ponownie sprawdzany warunek.

Jeżeli dołączymy podwójny znak „plus” („++”) do zmiennej „a” w poleceniu **for**, wartość zmiennej zwiększana będzie o „1” w każdym obiegu pętli.



Jeżeli dołączymy podwójny minus („--”) - wartość zmiennej będzie zmniejszana o „1”.

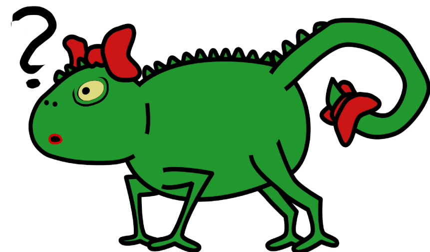
Wyjście programu:

```
10
11
12
13
14
15
```

Nie jest konieczne używane tej samej zmiennej we wszystkich częściach polecenia **for**.

Ćwiczenie. Stosowanie pętli – część III

Napisz skrypt bash, który czyta jedną jednocyfrową liczbę całkowitą z wejścia oraz zlicza od jednego do wprowadzonej liczby. Na ekranie powinna być wyświetlana nazwa liczby, nie cyfra (na przykład: „Jeden” dla „1”). Użyj pętli **for**.



Przerwanie pętli

W specyficznych warunkach można wyjść z pętli zanim zostanie zakończona. By to zrobić, należy użyć polecenia **break**.

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

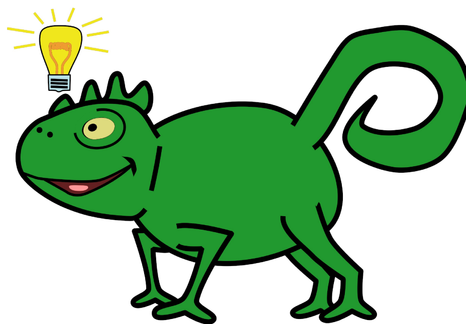
2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
a=1

while true; do
   echo "$a"
   a=$(expr $a + 1)
   if test $a == 8; then
      break
   fi
done
```

Wyjaśnienie. Zmienna $a jest ustawiana na „1”. Warunek while zawsze zwraca wartość „prawda” i pętla mogłaby wykonywać się w nieskończoność, gdyby nie polecenie break, które ją przerywa gdy wartość zmiennej osiągnie „8”.

Wyjście programu:

```
1
2
3
4
5
6
7
```



2.5.7 Stosowanie opcji i parametrów poleceń w programach

Jak już opisywaliśmy w poprzednich rozdziałach, każde polecenie ma zwykle wiele różnych opcji i parametrów. Można ich używać również w programach.

Liczba użytych opcji zapamiętywana jest w zmiennej „$#”.

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

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
echo "$# options passed."
```

Spróbujmy uruchomić krótki program z paroma parametrami:

```
geeko@da51:~> ./script.sh geeko suzie 100 Novell
4 options passed.
```

Polecenie użyte do uruchomienia programu jest zapamiętywane w zmiennej „$0”, pierwszy parametr w zmiennej „$1”, drugi w zmiennej „$2” i tak dalej.

```
#!/bin/bash

echo "\$0=$0"
echo "\$1=$1"
echo "\$2=$2"
echo "\$3=$3"
```

Gdy uruchamiamy program z kilkoma parametrami, wyjście, przykładowo, może wyglądać następująco:

```
geeko@da51:~> ./script.sh geeko suzie 100 Novell
$0=./script.sh
$1=geeko
$2=suzie
$3=100
```

Ta metoda nie jest zbyt wygodna, ponieważ musimy znać liczbę parametrów.

W powyższym przykładzie parametr „Novell” nie jest uwzględniony, ponieważ program oczekuje tylko trzech parametrów.

W ten sposób można uruchomić polecenie z dziewięcioma opcjami.

Poniższa wersja jest stosowana o wiele częściej:

Sprawdź pierwszą opcję. Następnie, usuń pierwszy element listy i drugi stanie się pierwszym.

Wykonuje to polecenie **shift**.

Zwykle, wykonuje się to poleceniami pętli.

Prosty przykład:

```
#!/bin/bash

while test $# -gt 0; do
   echo "Option: $1"
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

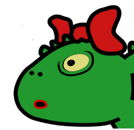
```
   shift
done
```

Wyjaśnienie. Polecenie test sprawdza czy lista opcji ma jeszcze jakieś elementy. Jeżeli tak, pierwszy element jest wyświetlany na ekranie i usuwany z listy. Pętla jest powtarzana póki wszystkie elementy nie zostaną usunięte z listy.

Wyjście z powyższego programu:

```
geeko@da51:~> ./script.sh geeko suzie 100 Novell
Option: geeko
Option: suzie
Option: 100
Option: Novell
```

By usunąć więcej niż jeden element z listy, należy użyć polecenia **shift *liczba***, gdzie *liczba* określa ilość elementów.



Zwykle program używany do sprawdzania opcji i parametrów wygląda jak poniżej:

```
#!/bin/bash
while test $# -gt 0; do
   case $1 in
      -a) echo "Option a"
         shift
         ;;
      -b) echo "Option b"
         shift
         ;;
      -f) echo "Option f with argument $2"
         shift 2
         ;;
      *)  echo "Wrong option: $1"
         break
         ;;
   esac
done
```

Poniżej zamieszczono kilka przykładów testów:

1.

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
geeko@da51:~> ./script.sh -a -f /home/geeko
Option a
Option f with argument /home/geeko
```

2.

```
geeko@da51:~> ./script.sh -f /home/geeko -a
Option f with argument /home/geeko
Option a
```

3.

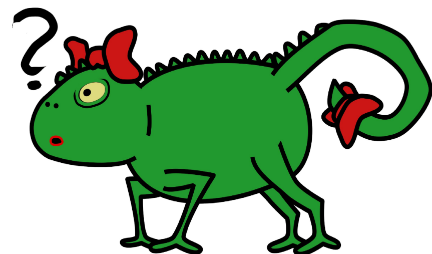
```
geeko@da51:~> ./script.sh -b /home/geeko
Option b
Wrong option: /home/geeko
```

4.

```
geeko@da51:~> ./script.sh /home/geeko -f
Wrong option: /home/geeko
```

Ćwiczenie. Używanie opcji i parametrów w programach

Opisz, co się dzieje w każdym z powyższych czterech przykładów testów.



1.____

2.____

3.____

4.____

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

2.5.8 Tworzenie funkcji 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 funkcji: *name* () { *polecenia* }

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

Przykład:

```
1  #!/bin/bash
2
3  sum() {
4          a=$(expr $1 + $2)
5          echo "$1+$2="$a
6  }
7
8  sum 4 5
9  sum 2 9
10 sum 12 1
11 sum 204 2387
12 sum 5 3
13 sum 21 39
```

Wyjaśnienie. Główny program zaczyna się w wierszu 8. Wywołuje on funkcję „sum” sześciokrotnie, z różnymi parametrami. Funkcja (wiersze 3-6) dodaje podane parametry i wynik wyświetla na ekranie. Parametry określają zmienne „$1” oraz”$2”.

Odwołania do parametrów funkcji realizowne są w ten sam sposób, jak omówione wcześniej odwołania do parametrów programu.

Wyjście progamu wygląda następująco:

```
4+5=9
2+9=11
12+1=13
204+2387=2591
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
5+3=8
21+39=60
```

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

Przykład:

```
1  #!/bin/bash
2  
3  validate() {
4          if [ $1 = yes ]; then
5                  return 1
6          else
7                  return 0
8          fi
9  }
10 
11 a=1
12 while true; do
13         echo $a" -> To continue, enter \"yes\""
14         read b
15         if validate $b; then
16                 break
17         fi
18         a=$(expr $a + 1)
19 done
20 echo "Stop!"
```

Wyjaśnienie. Główny program zaczyna się w wierszu 11. Zawiera nieskończoną pętlę (wiersze 12-19) i prosi użytkownika o wprowadzenie danych (wiersz 14). Pętla jest kończona (przerywana – wiersz 16) wtedy, gdy wartość zwracana funkcją **validate** () jest „prawda” (tj. „0”).

Funkcja pobiera dane od użytkownika jako wartość parametru polecenia. Jeżeli użytkownik wprowadzi „yes”, funkcja zwraca wartość ”1”, we wszystkich innych przypadkach – zwraca wartość „0”.

Jeżeli wprowadzimy „yes” cztery razy, wyjście programu będzie miało postać:

```
1 -> To continue, enter "yes"
yes
2 -> To continue, enter "yes"
```

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

```
yes
3 -> To continue, enter "yes"
yes
4 -> To continue, enter "yes"
no
Stop!
```

Domyślnie, wartość zmiennej obowiązuje w całym programie, Taką zmienną nazywamy ***zmienną globalną***. Możliwe jest jednak tworzenie zmiennych wyłącznie dla danej funkcji. Ten typ zmiennej nazywamy ***zmienną lokalną*** i tworzymy poleceniem **local**.

Przykład:

```
1  #!/bin/bash
2
3  localvar() {
4          local a=1
5          echo "function:" $a
6  }
7
8  a=0
9  echo "main before:" $a
10 localvar
11 echo "main after:" $a
```

Wyjaśnienie. Główny program zaczyna się w wierszu 8. Zmienna „$a” jest tworzona i nadawana jest jej wartość ”0”. W wierszu 10 funkcja jest nazywana. W tej funkcji jest definiowana lokalna zmienna, również o nazwie „a” (wiersz 4).

Wyjście programu:

```
main before: 0
function: 1
main after: 0
```

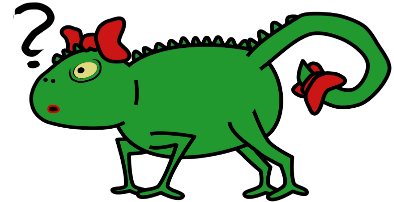
Jeżeli zmienna lokalna ma taką samą nazwę jak zmienna globalna, zmienna globalna jest „nadpisywana” lokalną wewnątrz funkcji. Po wykonaniu funkcji – jest przywracana oryginalną wartość zmiennej globalnej.

Wydaje się łatwiejszym używanie zmiennych globalnych. Jednak w dużych, skomplikowanych programach może to powodować konflikty, trudno zapanować nad setkami zmiennych. Więc lepiej jest deklarować zmienne tylko wtedy, gdy są potrzebne. Jest to cechą dobrego programowania.

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*

Ćwiczenie. Tworzenie funkcji

[...]sz skrypt bash, który czyta liczby z wejścia i oblicza średnią arytmetyczną wprowadzonych liczb. Program powinien się zakończyć, gdy zamiast liczby wprowadzisz słowo „koniec”. Użyj przynajmniej jednej funkcji do sprawdzenia, czy wprowadzone dane to liczba czy słowo „koniec”.



Podstawowe zasady dobrego programowania

- Używaj komentarzy w swoich programach.
- Używaj wcięć w formatowaniu dla większej czytelności twojego programu.
- Używaj „samowyjaśniających się” nazw dla funkcji i zmiennych.
- Dla stałych (nie zmieniają swoich wartości w czasie przebiegu programu), używaj dużych liter.
- Jeżeli potrzebujesz zmiennych dla pętli zliczających (np. dla for) użyj $i, $j lub $k.
- Jeżeli potrzebujesz zmiennej na krótko (np. dla buforowania wartości), użyj $a, $b, $c dla liczb całkowitych

  oraz $x, $y, $z dla liczb zmiennoprzecinkowych.

2008 Novell Inc. *Licencja Creative Commons Attribution-NonCommercial-ShareAlike 2.5. Wolno kopiować, rozprowadzać, przedstawiać i wykonywać utwór oraz opracowane na jego podstawie utwory zależne jedynie dla celów niekomercyjnych, przy zachowaniu autorstwa Novell Inc.*