

Linux i sieci: Bash jako narzędzie programowania

Projekt „Matematyka dla Ciekawych Świata”,

Robert Ryszard Paciorek

<rrp@opcode.eu.org>

2024-10-03

1 Podstawy

1.1 Zmienne

Określanie typów zmiennych w bashu odbywa się na podstawie wartości znajdującej się w zmiennej. Zasadniczo wszystkie zmienne są napisami, a interpretacja typu ma miejsce przy ich użyciu (a nie przy tworzeniu). Obsługiwane są liczby całkowite oraz napisy, ale bash nie posiada wbudowanej obsługi liczb zmiennoprzecinkowych.

```
zmiennaA=-91
zmiennaB="qa  z"
zmiennaC=98.6 # to będzie traktowane jako napis a nie liczba
```

Brak spacji pomiędzy nazwą zmiennej a znakiem równości jest wymogiem składniowym.

Wynika to ze znaczenia spacji w składni powłoki. Spacja oddziela nazwy poleceń i argumenty od siebie, czyli pełni istotną funkcję składniową, odpowiadającą nawiasom okrągłym i przecinkom, używanym do oddzielania nazwy funkcji i argumentów od siebie, z wielu innych języków programowania.

Odwołanie do zmiennej odbywa się z użyciem znaku dolara (\$), po którym występuje nazwa zmiennej. Nazwa może być ujęta w klamry, ale nie musi (jest to przydatne gdy nie chcemy dawać spacji pomiędzy nazwą zmiennej a np. fragmentem napisu). Rozwijaniu ulegają nazwy zmiennych znajdujące się w napisach umieszczonych w podwójnych cudzysłowach.

Umieszczenie odwołania do zmiennej w cudzysłowach zabezpiecza białe znaki (spacje nowe linie) przy przekazywaniu do funkcji i programów (w tym przy przekazywaniu do echo, celem wypisywania).

Użycie apostrofów wyłącza interpretację dolara jako znaku specjalnego (odwołania do zmiennej).

```
echo $zmiennaA ${zmiennaA}AA
echo "$zmiennaA ${zmiennaA}AA"
echo '$zmiennaA ${zmiennaA}AA'
```

1.1.1 Zmienne środowiskowe

Jeżeli chcemy aby zmienna była widoczna przez programy uruchamiane z danej powłoki (w tym przez kolejne instancje bash'a, odpowiedzialne np. za wykonywanie kodu skryptu uruchamianego z pliku) należy ją wyeksportować za pomocą polecenia: `export zmienna`

Do polecenia `export` przekazujemy nazwę zmiennej a nie jej wartość, więc nie używamy znaku dolara.

Taka zmienna jest dostępna jako zmienna środowiskowa dla wszystkich potomków tej powłoki.

Zmienne środowiskowe mogą być także ustawiane bez użycia 'export' dla pojedynczego nowego programu poprzez podanie ich nazw i wartości przed nazwą polecenia:

```
ABCD=678 bash -c 'echo $ABCD'
ABCD=135 EFG=098 bash -c 'echo $ABCD $EFG'
echo $ABCD
```

Są one jedynie widoczne w tak uruchomionym procesie potomnym (nie mogą być użyte w bieżącej powłoce, czy jako jako argumenty w linii poleceń). Dlatego w powyższych przykładach wywoływana jest nowa powłoka, która korzysta z tak ustawionych zmiennych.

Takie ustawianie zmiennych jest szczególnie przydatne gdy chcemy uruchomić pojedyncze polecenie w zmienionym środowisku - np. polecenie `date` w innej strefie czasowej:

1.2 Podstawowe operacje

Aby wykonać działania arytmetyczne należy umieścić je wewnątrz `$(( i ))`

```
a=12; b=3; x=5; y=6

# Dodawanie, mnożenie, odejmowanie zapisuje się i działają one tak jak w normalnej
↪ matematyce:
e=$(( ( $a + $b ) * 4 - $y ))

# Dzielenie zapisuje się przy pomocy ukośnika i jest ono zawsze dzieleniem
↪ całkowitym:
c=$(( $x / $y ))

echo $e $c
```

Do operacji arytmetycznych może być też jest wykorzystywane polecenie `let`. Najczęściej jest stosowane do inkrementacji podanej zmiennej, np.:

```
echo $a
let a++
echo $a
```

Zarówno operator podwójnych nawiasów okrągłych jak i komenda `let` mogą obsługiwać wyrażenia logiczne. Mimo to operacje logiczne najczęściej obsługiwane są komendą `test` lub operatorem `[ ]`, w których wynik zwracany jest jako kod powrotu.

Negację realizuje `!`, ale wynikiem negacją dowolnej liczby jest `FALSE`

Należy zwrócić uwagę na escapowanie odwrotnym ukośnikiem nawiasów i na to że spacje mają znaczenie. Wynika to z faktu, nawiasy kwadratowe wraz z zawartością to argumenty dla komendy `[ ]`.

```
[ \ ( $a -ge 0 -a $b -lt 2 \ ) -o $c -eq 5 ]; z=$?

echo $z
```

Wartość zmiennej `z` jest wynikiem warunku: `((a większe równe od zera) AND (b mniejsze od dwóch)) OR (c równe 5)`. Wynik ten został zwrócony jako `*kod powrotu*`, który jest dostępny (dla ostatnio wykonanego polecenia) poprzez `$?`. Wartość tej zmiennej została przypisana do zmiennej `z`.

Kody powrotu stosują logikę odwróconą 0 oznacza prawdę, coś nie zerowego to fałsz.

Jako operacje podstawowe powinniśmy także traktować wykonanie innych programów i pobieranie ich standardowego wyjścia i/lub kodu powrotu.

Pobieranie standardowego wyjścia możemy realizować za pomocą ujęcia polecenia w *backquotes* (```) lub operatora `$()` (pozwala on na zagnieżdżanie takich operacji).

Natomiast kod powrotu ostatniej komendy znajduje się w zmiennej `$?`.

```
a=`cat /etc/issue`
b=$((cat /etc/issue; cat /etc/resolv.conf))
```

```
echo $a
echo $b
echo "$b"
```

Zwróć uwagę na różnicę w wypisaniu zmiennej zawierającej znaki nowej linii objętej cudzysłowami i nie objętej nimi.

Bash nie obsługuje liczb zmiennoprzecinkowych ani operacji bitowych, nieobsługiwane operacje można wykonać za pomocą innego programu np:

```
a=`echo 'print(3/2)' | python3`
b=$(echo '3/2' | bc -l)
echo $a $b
```

inne polecenia

Programowanie w powłoce w dużej mierze polega na wywoływaniu innych programów (np. takich jak sed, grep, find, awk). Sama powłoka oferuje jedynie podstawowe konstrukcje składniowe, obsługę zmiennych i pewnych podstawowych operacji na nich.

Na te zewnętrzne polecenia można patrzeć trochę jak na biblioteki w innych językach programowania – komendy gwarantowane przez standard stanowią „bibliotekę standardową” basha, a inne (np. użyty w powyższym przykładzie arytmetyki zmiennoprzecinkowej python) stanowią dodatkowe opcjonalne „biblioteki”, które pozwalają na łatwiejsze i szybsze rozwiązywanie problemów.

W zasadzie podobnie można patrzeć na wywołania zewnętrznych programów w ramach kodu Pythona, C czy innych języków (niekiedy łatwiej jest zrobić np. ‘system("mv plik nowyplik")’ niż zakodować to bezpośrednio w Pythonie czy w C).

1.3 Uruchamianie kodu z pliku

Dłuższe fragmenty kodu shellowego często wygodniej jest pisać w pliku tekstowym niż bezpośrednio w linii poleceń. Plik taki może być wykonany za pomocą wywołania: `bash nazwa_pliku`.

Plik taki może zostać wykonany przy pomocy polecenia: `./nazwa_pliku` pod warunkiem że ma prawo wykonalności. Powinien także zawierać w pierwszej linii komentarz określający program używany do interpretacji tekstowego pliku wykonywalnego – `<em>shebang</em>` (np. `#!/bin/bash`). Jednak wykonywalny plik bez takiego komentarza też zostanie wykonany jako skrypt shellowy, jednak niekoniecznie z użyciem właściwej powłoki.

Przydatną alternatywą dla powyższych metod wykonania kodu zawartego w pliku jest włączenie go do aktualnej sesji basha przy pomocy `./nazwa_pliku`. W odróżnieniu od poprzednich metod kod zostanie wykonany w bieżącej instancji powłoki. Pozwala to na korzystanie z funkcji i zmiennych zdefiniowanych w tym pliku w kolejnych poleceniach.

1.4 Pętle i warunki

1.4.1 Pętla for

W bashu możemy korzystać z kilku wariantów pętli for. Jednym z najczęściej używanych jest przypadek iterowania po liście elementów, najczęściej liście plików:

```
for nazwa in /tmp/* ; do
    echo $nazwa;
done
```

Możliwe jest też iterowanie po wartościach całkowitych zarówno w stylu „shellowym” jak i w stylu C:

```
for i in `seq 0 20`; do
    echo $i;
done

for (( i=0 ; $i<=20 ; i++ )) ; do
    echo $i;
done
```

1.4.2 Pętla while

Pętla while powtarzana jest dopóki podany w niej warunek jest prawdziwy:

```
x=0
while [ $x -le 2 ]; do
    echo $x;
    x=$((x+1));
done
```

konstrukcja while - read Często używana jest pętla while w połączeniu z instrukcją read¹ co umożliwia przetwarzanie jakiegoś wejścia (wyniku komendy lub pliku) linia po linii (także z podziałem linii na słowa):

```
cat /etc/fstab | while read slowo reszta; do
    echo $reszta;
done
```

Powyższa pętla wypisze po kolei wszystkie wiersze pliku /etc/fstab przekazanego przez stdin (przy pomocy komendy cat)² z pominięciem pierwszego słowa (które wczytywane było do zmiennej slowo).

przekierowania strumieni a zmienne

Przekierowanie standardowego wyjścia na standardowe wejście odbywa się między dwoma różnymi procesami. Zatem w konstrukcjach typu while read pętla while uruchamiana może być w procesie potomnym obecnej powłoki. Efektem tego jest iż w niektórych przypadkach wykonywane modyfikacje zmiennych wewnątrz takiej pętli nie będą widoczne poza nią.

Przykładem takiej sytuacji jest poniższy kod (polecenie ps dodano aby pokazać utworzenie procesu potomnego powłoki):

```
zm=0; ps -f
cat /etc/fstab | while read x; do
    [ $zm -lt 1 ] && ps -f
    zm=$((zm+1))
done
```

1. Polecenie read można także wykorzystać do wczytania do jakiejś zmiennej danych podawanych przez użytkownika – np. read -p "wpisz coś >> " xyz wczyta tekst ze standardowego wejścia do zmiennej xyz. read z opcją -e potrafi korzystać z biblioteki readline, jednak np. współdzieli historię z historią basha. Dlatego często wygodniejsze może być zainstalowanie i użycie rlwrap, np: xyz=\$(rlwrap -H historia.txt -S "wpisz coś >> " head -n1).
2. Przekazanie danych do while-read poprzez strumień stdout → stdin z innego programu jest często stosowane np. w celu przefiltrowania lub posortowania danych przekazywanych do tej pętli. Jednak użycie w tym rozwiązaniu cat jest nadmiarowe (nazywane jest *martwym kotem*) i powinno się go unikać. Lepszym rozwiązaniem jest przekazywanie pliku przez przekierowanie strumienia wejściowego przy pomocy < plik, który w tym przypadku powinien znaleźć się za kończącym pętlę słowem kluczowym done. Między innymi oszczędza ono zasoby (i czas wykonania) związany z tworzeniem dodatkowego procesu dla polecenia cat.

```
done
echo $zm
```

Jednak analogiczny kod w którym następuje przekierowanie z pliku zadziała poprawnie:

```
zm=0; ps -f
while read x; do
    [ $zm -lt 1 ] && ps -f
    zm=13
done < /etc/fstab
echo $zm
```

Jeżeli do pętli ma trafić wyjście jakiegoś polecenia możemy użyć składni bash'a pozwalającej na podstawienie wyniku polecenia jako pliku w postaci <(polecenie) wraz z przekierowaniem z pliku, na przykład:

```
zm=0; ps -f
while read x; do
    [ $zm -lt 1 ] && ps -f
    zm=13
done < <$(cat /etc/fstab)
echo $zm
```

Zauważ spację pomiędzy dwoma znakami < i brak spacji pomiędzy drugim < i znakiem dolara). Inną możliwością jest użycie kodu powrotu do odebrania wartości z wnętrza pętli:

```
zm=0; ps -f
my_code() {
    while read x; do
        [ $zm -lt 1 ] && ps -f
        zm=13
    done;
    return $zm;
}
cat /etc/fstab | my_code
zm=$?
echo $zm
```

Została tu zdefiniowana funkcja bash'owa (my_code), o których więcej informacji znajdziesz w odpowiednim rozdziale poniżej.

Separator słów Słowa domyślnie rozdzielane są przy pomocy dowolnego ciągu spacji lub tabulatorów. Separator ten można zmienić za pomocą zmiennej IFS, np:

```
IFS=":"
while read a b c; do echo "$a -- $c"; done < /etc/passwd
unset IFS # przywracamy domyślne zachowanie read poprzez usunięcie zmiennej IFS
```

Należy mieć na uwadze, że cudzysłów wokół wypisania zmiennej c są istotne – bez nich znak dwukropka mógłby być zmieniony na spację.

W niektórych sytuacjach należy mieć na uwadze ograniczenia dotyczących maksymalnej długości listy argumentów (zarówno całej jak i pojedynczego argumentu). Mogą one okazać się problemem na przykład, gdy korzystamy ze znaków uogólniających powłoki i dopasowanych zostanie zbyt wiele plików dopasowanych, lub gdy wykorzystujemy przechwycenie standardowego wyjścia jakiegoś polecenia, aby użyć go jako argumentów innego polecenia. W takich sytuacjach warto skorzystać na przykład z pętli `while-read` działającej na standardowym wyjściu polecenia zwracającego te dane. W przypadku listy plików może to być `ls` bez argumentów lub (gdy potrzebujemy wybrać tylko niektóre pliki) `find`.

Zadanie 1.4.1

Napisz pętlę, która wypisze wszystkie pliki nieukryte z bieżącego katalogu w postaci linków HTML, czyli: dla pliku o nazwie ABC powinna wypisać `<a href="ABC">ABC</a>`. Przedstaw zarówno rozwiązanie z użyciem pętli `for`, jak i pętli `while`.

1.4.3 Instrukcja if

Poznane wcześniej obliczanie wartości wyrażeń logicznych najczęściej stosowane jest w instrukcji warunkowej `if`³.

```
# instrukcja if - else
if [ "$xx" = "kot" -o "$xx" = "pies" ]; then
    echo "kot lub pies";
elif [ "$xx" = "ryba" ]; then
    echo "ryba"
else
    echo "coś innego"
fi
```

Spacje wokół i wewnątrz nawiasów kwadratowych przy warunku są istotne składniowo. Wynika to z faktu, iż `[` jest w istocie komendą, działającą jak `test`, tyle że wymaga `]` jako ostatniego argumentu.

Oprócz typowych warunków logicznych możemy sprawdzać np. istnienie plików, czy też ich typ (link, katalog, etc). Szczegółowy opis dostępnych warunków, które mogą być użyte w tej konstrukcji znajduje się w `man test`.

Jako warunek może wystąpić dowolne polecenie wtedy sprawdzany jest jego kod powrotu: 0 oznacza prawdę / zakończenie sukcesem, wartość nie zerowa fałsz / błąd.

```
if grep '^root:' /etc/passwd > /dev/null; then
    echo /etc/passwd zawiera użytkownika root;
fi
```

Istnieje możliwość skróconego zapisu warunków z użyciem łączenia instrukcji przy pomocy

- `&&` – wykonaj instrukcję występującą po prawej gdy poprzednia zwróciła zero (`true`)
- `||` – wykonaj instrukcję występującą po prawej gdy poprzednia zwróciła nie zero (`false`)

```
[ -f /etc/issue ] && echo "jest plik /etc/issue"

grep '^root:' /etc/passwd > /dev/null && echo /etc/passwd zawiera użytkownika root;
```

3. Może być też stosowane np. w pokazanej wcześniej pętli `while`

1.4.4 Instrukcja case

Instrukcja case służy do rozważania wielu przypadków opartych na równości zmiennej z podanymi napisami.

```
case $xx in
    kot | pies)
        echo "kot lub pies"
        ;;
    ryba)
        echo "ryba"
        ;;
    *)
        echo "cos innego"
        ;;
esac
```

Zadanie 1.4.2

Napisz warunek, który sprawdzi czy /tmp/abc istnieje i jest katalogiem.

1.5 Definiowanie funkcji

W powłoce zgodnej z sh każda funkcja może przyjmować dowolną ilość parametrów pozycyjnych (w identyczny sposób obsługiwane są argumenty linii poleceń dla całego skryptu). Ilość parametrów znajduje się w zmiennej \$#, lista wszystkich parametrów w \$@, a do kolejnych parametrów możemy odwoływać się z użyciem \$1, \$2, itd.

```
f1() {
    # W nawiasach po nazwie funkcji nie podajemy przyjmowanych argumentów,
    # natomiast puste nawiasy te są elementem składniowym i muszą wystąpić.

    echo "wywołano z $# parametrami, parametry to: $@"

    [ $# -lt 2 ] && return;

    echo -e "drugi: $2\npierwszy: $1"

    # albo kolejnych w pętli
    for a in "$@"; do echo $a; done

    # lub z użyciem polecenia shift
    for i in `seq 1 $#`; do
        echo $1
        shift # powoduje zapomnienie $1
               # i przenumerowanie pozostałych argumentów pozycyjnych o 1
               # wpływa na wartości $@ $# itp
    done

    # funkcja może zwracać tylko wartość numeryczną - tzw kod powrotu
    return 83
}
```

```

# Wywołanie funkcji wygląda tak samo tak jak wywołania programów czy poleceń
↪ wbudowanych,
# czyli argumenty od nazwy funkcji i od siebie nawzajem rozdzielane są spacjami
# (nie występują tu nawiasy i przecinki znane z innych języków programowania).
f1 aaa 3 "t t" 56

# Kod powrotu ostatnio wywołanej funkcji uzyskuje się
# (tak samo jak innych poleceń) poprzez $?
echo $?

# Często funkcje (tak jak wiele komend) wynik swojego
# działania zwracają poprzez standardowe wyjście

f2() { echo "Uwolnić mrożone truskawki$1"; }

# Można użyć przekierowań strumieni aby przekazać go do innego programu lub funkcji
# i tak samo można pozyskać go do zmiennej poprzez `` lub $( )
a=`f2 '!!!'`
echo $a

# Zapis z użyciem $( ) może być zagnieżdżony
b=$(f2 "! $(f2 '!!!')")
echo $b

# W jednolinijkowym zapisie definicji funkcji (jak
# miało to miejsce dla f2) spacje po { i przed } są
# obowiązkowe, podobnie jak średniki po instrukcjach

```

1.6 Grupowanie poleceń

Funkcje są przykładem grupowania poleceń – funkcja stanowi nazwany blok kodu, czyli nazwaną grupę poleceń. Powłoka sh (i inne powłoki z nią zgodne) pozwalają na grupowanie bez definiowania funkcji. W tym celu możemy zastosować nawiasy klamrowe (tak jak w definicji funkcji) lub nawiasy okrągłe.

Stosując nawiasy klamrowe należy pamiętać (tak samo jak było to w przypadku funkcji) o spacji po otwierającym nawiasie klamrowym i średniku (lub nowej linii) przed zamykającym. Instrukcje podane w nawiasach klamrowych będą wykonane w bieżącej powłoce basha, czyli mogą modyfikować zmienne (ale w pewnych sytuacjach może się to nie wydarzyć).

Polecenia podane w nawiasach okrągłych będą wykonane w podpowłoce, czyli ustawione lub zmodyfikowane w nich zmienne nie będą widoczne po zakończeniu bloku. Nawiasy okrągłe nie wymagają spacji i ostatniego średnika.

```

a=0;
{ echo abc; a=1; }
echo $a
(echo abc; a=2)
echo $a

```

Grupowanie poleceń jest przydatne na przykład w celu grupowania ich przy używaniu operatorów łączenia poleceń w oparciu o kod powrotu (&& i ||), a także w celu przekazania standardowego wyjścia wielu poleceń w ramach jednego strumienia.

```
a=0;
{ echo AbC; echo abc; echo XyZ; a=1; } | grep b
echo $a
```

W tym wypadku nawiasy klamrowe zachowały się jak nawiasy okrągłe – modyfikacja zmiennej `a` nie jest widoczna po zakończeniu bloku. Wynika to z użycia przekierowania strumienia podobnie jak w sytuacji omawianej przy pętli `while`.

Zadanie 1.6.1

Napisać funkcję przyjmującą dwa argumenty - liczbę i napis; funkcja ma wypisać napis tyle razy ile wynosi podana liczba.

Zadanie 1.6.2

Napisać funkcję przyjmującą jeden argument - liczbę kotów i wypisującą:

- "Ala ma kota" dla ilości kotów równej 1
- "Ala ma x koty" lub "Ala ma x kotów" gdzie dobrana jest poprawna forma, a pod `x` podstawiona podana w argumencie ilość kotów.

Dla uproszczenia należy założyć że podana ilość kotów jest w zakresie od 1 do 9.

2 Przetwarzanie napisów

2.1 Wbudowane przetwarzanie napisów w `bash`'u

Wbudowane przetwarzanie napisów w `bash`u opiera się na odwołaniach do zmiennych w postaci `${}`:

- `${zmienna:-"napis"}` zwróci napis gdy zmienna nie jest zdefiniowana lub jest pusta
- `${zmienna}="napis"` zwróci napis oraz wykona podstawienie `zmienna="napis"` gdy zmienna nie jest zdefiniowana lub jest pusta
- `${zmienna}+"napis"` zwróci napis gdy zmienna jest zdefiniowana i nie pusta
- `${#str}` zwróci długość napisu w zmiennej `str`
- `${str:n}` zwróci pod-napis ze zmiennej `str` od pozycji `n` do końca
- `${str:n:m}` zwróci pod-napis ze zmiennej `str` od pozycji `n` o długości `m`
- `${str/"n1"/"n2"}` zwróci wartość `$str` z zastąpionym pierwszym wystąpieniem `n1` przez `n2`
- `${str//"n1"/"n2"}` zwróci wartość `$str` z zastąpionymi wszystkimi wystąpieniami `n1` przez `n2`
- `${str#"ab"}` zwróci wartość `$str` z obcięciem "ab" z początku
- `${str%"fg"}` zwróci wartość `$str` z obcięciem "fg" z końca

W napisach do obcięcia możliwe jest stosowanie shellowych znaków uogólniających, czyli `*`, `?`, `[abc]`, itd operator `#` i `%` dopasowują minimalny napis do usunięcia, natomiast operatory `##` i `%%` dopasowują maksymalny napis do usunięcia.

Należy pamiętać że wiele z powyższych zapisów jest rozszerzeniami `bash`a niedostępnymi w podstawowej składni `sh`.

Przykład:

```

a=""; b=""; c=""
echo ${a:-"aa"} ${b:="bb"} ${c:+"cc"}
echo $a $b $c

a="x"; b="y"; c="z"
echo ${a:-"aa"} ${b:="bb"} ${c:+"cc"}
echo $a $b $c

x=abcdefg
echo ${#x} ${x:2} ${x:0:3} ${x:0:${#x}-2}}
echo ${x#"abc"} ${x%"efg"}
echo ${x#"ac"} ${x%"eg"}

x=abcd.e.fg
echo ${x##*.} ${x###*.} ${x%.*} ${x%%.*}

y="aa bb cc bb dd bb ee"
echo ${y/"bb"/"XX"}
echo ${y// "bb"/"XX"}

```

```

aa bb
bb
x y cc
x y z
7 cdefg abc abcde
defg abcd
abcdefg abcdefg
e.fg fg abcd.e abcd
aa XX cc bb dd bb ee
aa XX cc XX dd XX ee

```

2.1.1 Wyrażenia regularne

Możliwe jest także korzystanie z wyrażeń regularnych. Polecenie `expr match $x 'wr1\ wr2\ wr3'` zwróci na stdout (wypisze) część `$x` pasującą do wyrażenia regularnego `wr2`, wyrażenia regularne `wr1` i `wr2` pozwalają na określanie części napisu do odrzucenia. Alternatywną składnią jest `expr $x : 'wr1\ wr2\ wr3'`

Przykład użycia jednego i drugiego wariantu składni do podziału napisu na część przed i po znaku równości:

```

z="ab=cd"
expr match $z '^([=]*)\='
expr $z : '^([=]*)=\.*)$'

```

Możliwe jest też sprawdzanie dopasowań wyrażeń regularnych poprzez (uwaga na brak cudzysłowów wokół wyrażenia regularnego):

```

[[ "$z" =~ ^([=]*)= ]] && echo "OK"

```

2.1.2 printf

Możliwe jest także zaawansowane formatowanie napisów, konwertowanie liczb na napisy, w tym wypisywanie w różnych systemach liczbowych przy pomocy `printf` (instrukcja `printf` ma składnię opartą na tej funkcji z C, interpretuje ona także liczby zmiennoprzecinkowe):

```

printf "0o%o %d 0x%x\n" 0xf 010 3

```

```

0o17 8 0x3

```

Zadanie 2.1.1

Napisz polecenie które dla wszystkich plików z rozszerzeniem .TXT w bieżącym katalogu (bez podkatalogów) dokona zmiany ich nazwy zmieniając rozszerzenie na .txt, zachowując podstawową część nazwy bez modyfikacji. W rozwiązaniu nie korzystamy z polecenia rename.

2.2 grep, cut, sed, ...

Jako że większość operacji wykonywanych w powłoce takiej jak bash wiąże się z uruchamianiem zewnętrznych programów, to także przetwarzanie napisów może być realizowane w ten sposób. Opiera się na tym jedno z podejść do obsługi napisów w bashu, którym jest korzystanie z standardowych komend POSIX, takich jak grep, cut, sed.

```
a="aąbcć 123"

# obliczanie długości napisu w znakach, w bajtach i ilości słów w napisie
echo -n $a | wc -m
echo -n $a | wc -c
echo -n $a | wc -w

# obliczanie ilości linii (dokładniej ilości znaków nowej linii)
wc -l < /etc/passwd

# wypisanie 5 pola (rozdzielanego :) z pliku /etc/passwd z eliminacją
# pustych linii oraz linii złożonych tylko ze spacji i przecinków
cut -f5 -d: /etc/passwd | grep -v '^[ ,]*$'
# komenda cut wybiera wskazane pola, opcja -d określa separator
```

Inną bardzo przydatną komendą jest sed pozwala ona m.in na zastępowanie wyszukiwanego na podstawie wyrażenia regularnego tekstu innym:

```
echo "aa bb cc bb dd bb ee" | sed -e 's#\([bc]\+\) \([bc]\+\)#X-\2-X#g'
```

Sedowe polecenie s przyjmuje 3 argumenty (oddzielane mogą być dowolnym znakiem, który wystąpi za s), pierwszy to wyszukiwane wyrażenie, drugi to tekst którym ma zostać zastąpione, a trzeci to flagi, jeżeli podano tam g to zastępowane będą wszystkie wystąpienia a nie tylko pierwsze.

Należy zwrócić uwagę na różnicę w składni wyrażenia regularnego polegającą na poprzedzaniu (,) i + odwrotnym ukośnikiem aby miały znaczenie specjalne (aby tego uniknąć należy włączyć obsługę ERE w sed poprzez opcję -E).

Innymi przydatnymi komendami przetwarzającymi (specyficznej postaci) napisy są polecenia basename i dirname. Służą one do uzyskania nazwy najgłębszego elementu ścieżki oraz ścieżki bez tego najgłębszego elementu. Zobacz wynik działania:

```
basename /proc/sys/net/core/
dirname /proc/sys/net/core/
```

Zadanie 2.2.1

Wyświetl z /etc/passwd linie w których UID (3 pole) ma wartość ≥ 1000 nie korzystając z AWK. Jeżeli masz pomysł przedstaw więcej niż jedno rozwiązanie.

Zadanie 2.2.2

Napisz funkcję która przyjmuje dwa argumenty - napis wyszukiwany i napis go zastępujący oraz dokonuje rekurencyjnego wyszukania i zamiany tych napisów w wszystkich plikach w bieżącym katalogu.

Wskazówka 1: polecenie **sed** z opcją **-i** i wskazaniem pliku modyfikuje zawartości tego pliku stosownie do poleceń wydanych **sed**'owi

Wskazówka 2: dla uproszczenia możesz przyjąć że napisy te składają się jedynie z liter i cyfr.

2.3 awk

Awk jest interpreterem prostego skryptowego języka umożliwiającym przetwarzanie tekstowych baz danych postaci linia == rekord, gdzie pola oddzielane ustalonym separatorem (można powiedzieć że łączy funkcjonalność komend takich jak **grep**, **cut**, **sed** z prostym językiem programowania).

Wyżej zaprezentowane wypisanie 5 pola (rozdzielanego dwukropkiem) z pliku **/etc/passwd** z eliminacją pustych linii oraz linii złożonych tylko ze spacji i przecinków, realizowane przy użyciu poleceń **cut** i **grep** może być zrealizowane za pomocą samego **awk**:

```
awk -F: '$5 !~ "[ ,]*$" {print $5}' /etc/passwd
```

Awk daje duże możliwości przy przetwarzaniu tego typu tekstowych baz danych: możemy np. wypisywać pierwsze pole w oparciu o warunki nałożone na inne:

```
awk -F: '$5 !~ "[ ,]*$" && $3 >= 1000 {print $1}' /etc/passwd
```

Jak widać w powyższych przykładach do poszczególnych pól odwołujemy się poprzez **\$n**, gdzie **n** jest numerem pola, **\$0** oznacza cały rekord

Program dla każdego rekordu przetwarza kolejne instrukcje postaci warunek { komendy }, instrukcji takich może być wiele w programie (przetwarzane są kolejno), komenda **next** kończy przetwarzanie danego rekordu.

Separator pola ustawiamy opcją **-F** (lub zmienną **FS**), domyślnym separatorem pola jest dowolny ciąg spacji i tabulatorów (w odróżnieniu od **cut** separator może być wieloznakowym napisem lub wyrażeniem regularnym). Domyślnym separatorem rekordu jest znak nowej linii (można go zmienić zmienną **RS**).

Awk jest prostym językiem programowania obsługującym podstawowe pętle i instrukcje warunkowe oraz funkcje wyszukiujące i modyfikujące napisy:

```
echo "aba aab bab baa bba bba" | awk '{
    for (i=1; i<=NF; ++i) {          # dla każdego pola w rekordzie
        if (i%2==0)                  # jeżeli jego numer jest parzysty
            gsub("b+", "B", $i);     # zastąp wszystkie ciągi b
            ↪ pojedynczym B
        ii = index($i, "B")           # wyszukaj pozycję pod-napisu B
        if (ii)                       # jeżeli znalazł to wypisz pozycję
            ↪ i pod-napis
            printf("# %d %s\n", ii, substr($i, ii)) # od tej pozycji
            ↪ do końca
            # UWAGA: AWK liczy elementy w napisie od 1 a nie od 0
        }
    }
    print $0
}'
```

AWK obsługuje także tablice asocjacyjne pozwala to np. policzyć powtórzenia słów:

```
echo "aa bb aa ee dd aa dd" | awk '
BEGIN {RS="[ \t\n]"; FS=""}
{slowa[$0]++}
# może być kilka bloków {} pasujących do rekordu
# jeżeli nie użyjemy next przetworzone zostaną wszystkie
{printf("rekord: %d done\n", NR)}
END {for (s in slowa) printf("%s: %s\n", s, slowa[s])}}
```

Podobny efekt możemy uzyskać stosując "uniq -c" (który wypisuje unikalne wiersze wraz z ich ilością) na odpowiednio przygotowanym napisie (spacje zastąpione nową linią, a linie posortowane):

```
echo "aa bb aa ee dd aa dd" | tr ' ' '\n' | sort | uniq -c
```

Jednak rozwiązanie awk można łatwo zmodyfikować aby wypisywało pierwsze wystąpienie linii bez sortowania pliku.

Innym użytecznym zastosowaniem AWK może być wypisanie pliku bez linii pasujących do wzorca oraz linii poprzednich:

```
echo -e "aa\nbb\nWZORZEC\ncc" | awk '{
# dla linii pasującej do wzorca ustawiamy flagę print_last na zero
# i przechodzimy do następnej linii
/WZORZEC/ {print_last=0; next}

# jeżeli flaga print_last jest nie zero wypisujemy zapamiętaną poprzednią
↪ linię
print_last == 1 {print last}

# zapamiętujemy bieżącą linię do wypisania przy przetwarzaniu kolejnej
# (jeżeli nie będzie pasować do wzorca)
{last=$0; print_last=1}

# jeżeli osiągneliśmy koniec pliku i mamy linię do wypisania to ją
↪ wypisujemy
END {if (print_last == 1) print last}
}'
```

AWK pozwala także na definiowanie funkcji:

```
awk 'function f(x) {return 2*x} { print f($1+$2) }'
```

Zadanie 2.3.1

Korzystając z AWK wyświetl z /etc/passwd linie w których UID (3 pole) ma wartość ≥ 1000 .

Zadanie 2.3.2

Polecenie last wypisuje informacje o ostatnich zalogowaniach w systemie. Napisz polecenie (wykorzystujące last), które wypisze informację jak często logowali się poszczególni użytkownicy.

3 Więcej bash'a

3.1 Napis jako kod bashowy

3.1.1 nazwa polecenia w zmiennej

Jeżeli mamy w zmiennej nazwę polecenia do wykonania to możemy je wykonać uruchamiając po prostu tę zmienną:

```
a="echo"  
$a ABC
```

```
ABC
```

Możemy nawet uruchamiać polecenia które mamy zapisane w zmiennej wraz z opcjami i argumentami:

```
a="echo -e abc\ndef ..."  
$a ABC
```

```
abc  
def ... ABC
```

Metoda ta nie pozwala jednak na podstawianie wartości zmiennych występujących w napisie w momencie uruchamiania go, nie pozwala też na umieszczenie w napisie wielu poleceń rozdzielanych np. średnikami.

3.1.2 eval

Polecenie wbudowane `eval` pozwala na wykonanie przekazanych do niego argumentów jako polecenie shellowe. Zapewnia ono podstawienie występujących w tym napisie zmiennych, itd.

Polecenia tego można użyć np. do z wartości zmiennej, której nazwę mamy w innej zmiennej (zamiast `${!x}`, które nie jest dostępne w czystym sh):

```
A='tekst do wypisania, $SHELL, `ls -d`'  
B="A"  
  
C=$( eval "echo \${$B}" )  
echo $C
```

```
tekst do wypisania, $SHELL, `ls -d`
```

Jeżeli chcielibyśmy aby kod znajdujący się w zmiennej A także został przetworzony (podstawiona wartość zmiennej oraz output komendy ls), możemy użyć `eval` dwukrotnie:

```
A='tekst do wypisania, $SHELL, `ls -d`'  
B="A"  
  
C=$( eval eval "echo \${$B}" )  
echo $C
```

```
tekst do wypisania, /bin/bash, .
```

Jak widać `eval` powoduje wykonywanie poleceń zawartych w napisie zatem należy zachować ostrożność przy jego użyciu na napisach nieznanego pochodzenia/autorstwa.

Dzięki użyciu `eval` możemy też budować kod bashowy (np. fragmenty konstrukcji case) w oparciu o zawartość zmiennych:

```
x=a

LISTA_WYBORU="a) echo AA;; b) echo BB;;"
INNE="e|f"

eval "case $x in
    $LISTA_WYBORU
    c) echo "CCC";;
    $INNE) echo inne;;
esac"
```

```
AA
```

3.1.3 envsubst

Innym poleceniem umożliwiającym podstawienie wartości zmiennych w jakimś napisie jest envsubst. W odróżnieniu od eval nie wykonuje on podanego kodu, a jedynie przetwarza napis podstawiając wartości zmiennych (zatem jest bezpieczniejsze przy przetwarzaniu danych niewiadomego pochodzenia). Jako że jest to zewnętrzny program używane zmienne muszą być dla niego dostępne, czyli muszą być wyeksportowane.

```
x=12
D='tekst, ${x}, $SHELL, $x, $y, `ls -d`'

# y nie jest wyeksportowane więc będzie
↪ zastąpione napisem pustym
C=`export x; echo $D | envsubst`
echo $C
```

```
tekst, 12, /bin/bash, 12, , `ls -d`
```

3.2 Przekierowania, procesy, ...

Poznaliśmy już podstawy przekierowywania standardowych strumieni pomiędzy procesami oraz z użyciem plików. Mówiliśmy też o pewnych ograniczeniach jakie wiąże się z takimi przekierowaniami i programowaniem w bashu. Należy jednak wspomnieć jeszcze o kilku istotnych aspektach takich przekierowań.

3.2.1 cat, tee i <<

Przekierowania takie są często używane do tworzenia zewnętrznych plików z poziomu kodu skryptu bashowego. Wykonywanie takiej operacji z użyciem wielu instrukcji echo niekoniecznie jest wygodne, dlatego często stosowane jest następujące podejście:

```
cat > plik << EOF
wieloliniowa
treść zapisywana
do wskazanego pliku
EOF
```

Widzimy wywołanie komendy cat, z którym mogliśmy się już spotkać. Polecenie cat wywołane z argumentami będącymi nazwami plików wypisze te pliki na standardowe wyjście. Wywołane bez argumentów przeczyta swoje standardowe wejście i wypisze je na standardowe wyjście. Jest to użyteczne w połączeniu z przekierowaniami strumieni – na przykład wywołanie cat > plik pozwala nam wprowadzić treść do wskazanego pliku, zobacz np. przesyłanie strumieniowe plików z użyciem ssh).

Wyjście cat przekierowane jest do pliku, jednak zwraca tutaj uwagę inny operator - dwa znaki mniejszości (<<).

Operator « powoduje że tekst podawany po nim w kolejnych liniach będzie kierowany na standardowe wejście komendy po której wystąpił (w tym wypadku "cat"), aż do momentu napotkania linii zawierającej jedynie słowo podane po nim (w tym wypadku EOF, ale może to być dowolne inne słowo - np. KONIEC lub nawet ciąg znaków ze spacjami ujęty w cudzysłowa).

Jeżeli słowo to jest ujęte w apostrofy bądź cudzysłowy to w przekazywanym tekście nie są dokonywane podstawienia shellowe (np. rozwijane zmienne):

```
cat << KONIEC
HOME: $HOME
KONIEC

cat << 'KONIEC'
HOME: $HOME
KONIEC
```

Innym poleceniem przydatnym przy manipulacji strumieniami jest tee. Podobnie jak cat kopiuje on swoje standardowe wejście na standardowe wyjście. Natomiast jeżeli poda mu się ścieżkę do pliku będzie on zapisywał te dane także do wskazanego pliku.

```
echo ABC | tee /tmp/xxx
cat /tmp/xxx
```

3.2.2 standardowe wejście/wyjście w miejscu ścieżki do pliku

Wiele programów jeżeli w miejscu w którym oczekuje ścieżki do pliku otrzyma myślnik zinterpretuje to jako użycie w tym miejscu standardowego wejścia / wyjścia. Nawet jeżeli program nie wspiera tej konwencji możemy użyć specjalnych urządzeń reprezentujących te strumienie: /dev/stdin, /dev/stdout, /dev/stderr. Na przykład:

```
echo "ABC" > /dev/stderr
```

spowoduje wypisanie komunikatu ABC na standardowym wyjściu błędu.

Bash pozwala także na podstawienie standardowego wyjścia różnych komend w miejsce kilku plików bez potrzeby jawnego tworzenia plików tymczasowych. Na przykład:

```
diff <(sort /etc/passwd) <(sort /etc/passwd-)
```

które poleceniu diff jako jeden plik postawia standardowe wyjście pierwszego sort, a jako drugi plik drugiego sort. Pozwalając porównać posortowane pliki, bez tworzenia plików tymczasowych.

Możliwe jest też działanie w drugą stronę czyli przesłanie danych wpisywanych przez jeden proces do pliku jako standardowego wejścia innego programu również trochę bezsensowny przykład:

```
echo "ABC" | tee >(grep A)
```

<() i >() są rozszerzeniami bashowymi nie występującymi w czystym sh.

3.3 polecenia wykonywane na zakończenie

Działający program może zostać zakończony w sposób niespodziewany, np. na skutek działania sygnałów. Dotyczy to również basha wykonującego jakiś skrypt powłoki.

Możemy nawet samodzielnie zażądać przerwania wykonywania skryptu przy pierwszym błędzie (pierwszym poleceniu które zwróci nieobsłużony niezerowy kod powrotu) wydając polecenie `set -e`.

Niekiedy chcemy móc w takiej sytuacji wykonać jakieś działania. Możliwe jest to z użyciem instrukcji `trap`, która pozwala na zdefiniowanie procedury obsługi sygnałów (oczywiście tych które mamy prawo obsługiwać) i innych zdarzeń. Na przykład:

```
trap '{ echo "to koniec"; }' EXIT
```

Spowoduje wypisanie "to koniec" w momencie kończenia pracy powłoki. Umieszczenie takiej instrukcji w kodzie skryptu spowoduje wypisanie tego komunikatu niezależnie od przyczyny zakończenia (dojście do końca skryptu wywołanie polecenia `exit`, przerwanie Control C, czy też otrzymanie innego przechwytywalnego sygnału).

3.4 zarządzanie procesami w tle

Użycie `&` przy uruchamianiu polecenia nie jest jedynym sposobem aby uzyskać program działający w tle. Z użyciem polecenia `bg` możemy też przenieść w tło polecenie zawieszone za pomocą Control Z (natomiast `fg` użyte w tym momencie przywróci je na pierwszym planie). Powłoka jako rodzic procesów związanych z tak uruchomionymi poleceniami posiada ich listę i pozwala na łatwe zarządzanie nimi. Listę tą można wyświetlić z użyciem polecenia `jobs`, jeżeli dodana zostanie opcja `-l` to wyświetlone będą także identyfikatory procesów.

Korzystając z polecenia `fg` któremu jako argument podamy numer jobu (nie PID) można przenieść takie zadanie na pierwszy plan. Ponowne przeniesienie w tło możliwe będzie na przykład dzięki Control Z i `bg`. Można też przesyłać sygnały do takich zadań z użyciem polecenia `kill` podając albo PID albo numer jobu poprzedzony znakiem procenta. Zadanie można także usunąć z listy jobów pozostawiając je jednak działającym z użyciem `disown`.

4 Wykład wideo⁴

- Wprowadzenie do programowania w bash'u – <http://video.opcode.eu.org/05.01-bash-intro.mkv>
- Pętle, warunki i funkcje w bash'u – http://video.opcode.eu.org/05.02-bash-pętle_warunki_funkcje.mkv
- Bash: obsługa napisów, eval i podsumowanie – http://video.opcode.eu.org/05.03-bash-napisy_i_okolice.mkv
- Język AWK – <http://video.opcode.eu.org/04.02-awk.mkv>

5 Rozwiązania zadań

Treści zadań zamieszczone zostały w odpowiednich rozdziałach skryptu.

Poniżej zamieszczone są przykładowe rozwiązania „głównych” zadań z tego skryptu wraz z komentarzami. Wiemy że zajrzenie do nich już przy pierwszej trudności jest kuszące, mimo to rekomendujemy przynajmniej podjąć ucziwą, co najmniej kilkunastominutową na każde z zadań, próbę rozwiązania tych zadań bez zaglądania do odpowiedzi.

4. Filmy posiadają napisy wgrane do kontenera multimedialnego jako osobny strumień – napisy mogą być włączone lub wyłączone w odtwarzaczu. W wielu filmach dużo dzieje się "na dole ekranu", dlatego polecamy odtwarzać filmy z napisami umieszczonymi poniżej filmu, np. przy pomocy polecenia: `vlc --video-filter='croppadd{paddbottom=120}' --sub-margin=-10 PLIK.mkv`

Pamiętaj!: Samodzielne rozwiązanie problemu (wraz z wszystkimi trudnościami po drodze i popełnionymi błędami) jest dużo bardziej kształcące od nawet wielokrotnego przepisania gotowego rozwiązania, jednak nawet jednokrotne przepisanie rozwiązania jest bardziej kształcące od wielokrotnego przekopiowania go.

Rozwiązanie zadania 1.6.2

- w nawiasach po nazwie funkcji nie piszemy nic na temat jej argumentów - one są puste
- do argumentów odwołujemy się poprzez dolar numer argumentu
- do n-którego powtórzenia używamy pętli for która iteruje po liście liczb zwracanej przez seq
- seq objęta jest znakami ' oznaczającymi że należy wykonać podany w nich kod i podstawić w to miejsce jego standardowe wyjście, nie należy ich mylić z apostrofami używanymi do napisów
- spacja po oraz średnik (lub nowa linia) przed są istotne składnikowo

Zwróć uwagę że:

```
f() { for i in `seq 1 $1`; do echo $2; done; }
```

Rozwiązanie zadania 1.6.1

- w celu warunkowego wypisania jakiejś informacji możemy użyć zarówno konstrukcji if, jak też łączenia poleceń, jednak w przypadku większej ilości poleceń objętych warunkiem konstrukcja z if jest bardziej czytelna
- sprawdzenie czy podana ścieżka istnieje i jest katalogiem odbywa się przy pomocy opcji -d, informacja ta celowo nie była podana w treści skryptu - należało to sprawdzić w dokumentacji systemowej (man test). **Czytanie dokumentacji jest ważne!**

Zwróć uwagę że:

```
if [ -d /tmp/abc ] && echo "jest katalogiem";
```

lub krócej:

```
if [ -d /tmp/abc ] ; then echo "jest katalogiem"; else echo "nie"; fi
```

Rozwiązanie zadania 1.4.2

- rozwiązań te różnią się jedynie sposobem uzyskania listy plików dla której mają wypisać linki
- pętla for w każdym obiegu podstawia pod f kolejny element z listy nazw dopasowanych do gwiazdki (czyli wszystkich plików nieukrytych)
- pętla while listę plików dostaje na standardowe wejście (jeden plik na linie) i wczytuje każdą kolejną linie (czyli kolejną nazwę pliku) stosując komendę read - jest to bardzo standardowe rozwiązanie do przetwarzania standardowego wejścia linia po linii
- w obu wypadkach używamy takiego echo z napisem w podwójnych cudzysłowach (aby móc umieścić w nim znienam), cudzysłowa które mają być wypisane zabezpieczamy odwrotnym ukośnikiem
- wypisywanie można rozwiązać na kilka innych sposobów np.: `echo "<a href=\"$f\">\"$f\"</a>\"` - zadziała tak samo, ale wydaje się być to mniej czytelne

Zwróć uwagę że:

```
for f in *; do echo "<a href=\"$f\">\"$f\"</a>"; done
ls | while read f; do echo "<a href=\"$f\">\"$f\"</a>"; done
```

Rozwiązanie zadania 1.4.1

```

done
[ $uid -ge 1000 ] && echo $!linia;
↪ tabulatorze)
# następnie czytamy go w petli while-read jako dwa pola (przed i po
cut -f3 -d: /etc/passwd | paste /dev/stdin /etc/passwd | while read uid linia; do
# używając cut i paste tworzymy plik złożony z pola 3 tabulator cała linia

```

```

IFS=":"; while read p1 p2 p3 px; do
if [ $p3 -ge 1000 ]; then echo "$p1:$p2:$p3:$px"; fi
done < /etc/passwd; unset IFS

```

```

while read l; do
x=$(echo $l | cut -f3 -d:);
[ $x -ge 1000 ] && echo $l;
done < /etc/passwd

```

Rozwiązanie zadania 2.2.1

Do iteracji po plikach używamy petli for, możemy jej od razu podać wyrażenie określające po jakich plikach ma przebiegać. Nie potrzebujemy tutaj konstrukcji typu for f in `ls \*.TXT`, aczkolwiek w bardziej złożonych przypadkach może ona być użyteczna. Na przykład gdy chcemy pominąć pliki w nazwie których występuje XYZ: for f in `ls \*.TXT | grep -v XYZ` (pod warunkiem że nie mamy plików ze spacjami w nazwie - wtedy lepiej zrobić for f in `\*.TXT` i filtrować daną nazwę w każdym obiegu petli lub użyć potoku ls | grep | while read ...).

W takim przypadku \${f%.TXT}.txt jest lepsze od \${f}/.TXT/.txt lub \${f%.TXT}/.TXT/.txt gdyż usuwa tylko ciąg .TXT występujący na końcu napisu. aa.TXT.xyz.TXT jest poprawną nazwą pliku i zgod- nie z warunkami zadania powinna być zmieniana na aa.TXT.xyz.txt, a nie na aa.txt.txt.TXT lub aa.txt.txt.xyz.txt.

```

# albo:
for f in *.TXT; do mv "$f" "${f%.TXT}.txt"; done

for i in *.TXT; do mv "$i" "$(basename $i .TXT).txt"; done

```

Rozwiązanie zadania 2.1.1

```

koty() {
if [ $1 -eq 1 ]; then
echo "Ala ma kota"
elif [ $1 -lt 5 ]; then
↪ ]; then
echo "Ala ma $1 koty"
else
echo "Ala ma $1 kotów"
fi
}

```

```

koty() {
case $1 in
1) echo "Ala ma kota";;
2|3|4) echo "Ala ma $1 koty";;
*) echo "Ala ma $1 kotów";;
esac
}

```

- zignorowanie linii zaczynającej się od "wtmp begins" z użyciem dopasowania wyrażenia regularnego i komendy next
 - zliczanie wystąpień nazwy użytkownika w tablicy x (indeksowanej tymi nazwami)
 - zastosowanie bloku END do wypisania zawartości tablicy, której używaliśmy do zliczania użytkowników
- Zwróć uwagę na:

```
last | awk '/wtmp begins/{next} $1 != "" {x[$1]++;} END{for (u in x) print u, x[u]}'
```

Rozwiązanie zadania 2.3.2

- ustawienie separatora pól przy pomocy opcji -F na dwukropki
 - wykorzystanie charakterystyki AWK polegającej na wykonywaniu bloku dla każdej pasującej linii
 - warunek na wartość 3 pola, awk sam zadba o konwersję napisu przeczytanego z pliku na liczbę celem wykonania tego porównania
- Zwróć uwagę na:

```
awk -F: '$3>=1000 {print $0}' /etc/passwd
```

Rozwiązanie zadania 2.3.1

Zamiast `grep -lR "$1"` można by napisać `grep -R "$1" . | cut -f 1 -d: | uniq`, w większości przypadków zadziała tak samo, ale jest bardziej skomplikowane, wymaga więcej zasobów i będzie miało problem z nazwami plików zawierającymi dwukropki.

```
replace() {
    grep -lR "$1" . | while read f; do
        sed -e "s@$1@$2@g" -i $f;
    done;
}
```

Rozwiązanie zadania 2.2.2

Warto zwrócić uwagę że rozwiązanie z `awk (awk -F: '$3>=1000 {print $0}' /etc/passwd)` jest znacznie prostsze.

```
egrep '~([[:*]]{2}[0-9]{4})' /etc/passwd
```