

Linux i sieci: Bash jako narzędzie programowania

Projekt „Matematyka dla Ciekawych Świata”,

Robert Ryszard Paciorek

<rrp@opcode.eu.org>

2023-06-25

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, bash nie posiada wbudowanej obsługi liczb zmiennoprzecinkowych.

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

Zwróć uwagę na brak spacji pomiędzy nazwą zmiennej a znakiem równości w operacji przypisania - jest to wymóg składniowy.

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

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

Jeżeli chcemy aby zmienna była widoczna przez programy uruchamiane z naszej 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` (zwróć uwagę na brak dolara w tym miejscu).

1.2 Podstawowe operacje

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

Dodawanie, mnożenie, odejmowanie zapisuje się i działają one tak jak w normalnej matematyce, dzielenie zapisuje się przy pomocy ukośnika i jest ono zawsze dzieleniem całkowitym:

```
a=12; b=3; x=5; y=6
e=$(( ($a + $b) * 4 - $y ))
c=$(( $x / $y ))
echo $e $c $z
```

Zauważ zachowanie przy odwołaniu do niezainicjalizowanej zmiennej z.

Do operacji arytmetycznych może być też wykorzystywane polecenie `let`. Najczęściej jest stosowane do inkrementacji podanej zmiennej, tak jak w poniższym przykładzie.

```
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 `[]` wynik zwracany jest jako kod powrotu. Należy zwrócić uwagę na escapowanie odwrotnym ukośnikiem nawiasów i na to że spacje mają znaczenie. Negację realizuje `!`, należy pamiętać jednak że wynikiem negacji dowolnej liczby jest `FALSE`.

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

Wartość zmiennej `z` jest wynikiem warunku: ((a większe równe od zera) AND (b mniejsze od dwóch)) OR (z równe 5). Bash stosuje logikę odwróconą 0 oznacza prawdę, coś nie zerowego to fałsz.

Jako operacje podstawowe powinniśmy patrzeć także na 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 `?` (używaliśmy tego już przy obliczaniu wyrażeń logicznych).

```
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, 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 bashu w dużej mierze polega na wywoływaniu innych programów (np. takich jak `sed`, `grep`, `find`, `awk`). Sam bash 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 artrymetyki 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 bashowego często wygodniej jest pisać w pliku tekstowym niż bezpośrednio w linii poleceń. 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, w postaci: `#!/bin/bash`). Może też być wykonany za pomocą wywołania: `bash nazwa_pliku`.

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 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 plikach¹:

```
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

Często używana jest pętla `while` w połączeniu z instrukcją `read`² 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
```

Powyzsza pętla wypisze po kolei wszystkie wiersze pliku `/etc/fstab` przekazanego przez `stdin` (przy pomocy komendy `cat`)³ z pominięciem pierwszego słowa (które wczytywane było do zmiennej `slowo`).

1. Dokładniej: iteracja odbywa się po liście napisów rozdzielanej spacjami - zobacz rezultat `echo /tmp/*`
2. Polecenie `read` można także wykorzystać do wczytania danych podawanych przez użytkownika do jakiejś zmiennej – np. `read -p "wpisz coś >> " xyz` wczyta tekst 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``.
3. Takie rozwiązanie nazywane jest *martwym kotem* i powinno go się 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`.

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=13
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
```

Słowa domyślnie rozdzielane są przy pomocy dowolnego ciągu spacji lub tabulatorów, separator 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 cudzysłowa wokół wypisania zmiennej `c` są istotne – bez nich znak dwukropka mógłby być zmieniony na spacje.

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 `ABC`. 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`⁴.

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

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

```
    echo "coś innego"
fi
```

Zauważ że spacje wokół i wewnątrz nawiasów kwadratowych przy warunku są istotne składniowo, zawartość nawiasów kwadratowych to tak naprawdę argumenty dla komendy test. 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, a wartość nie zerowa fałsz / błąd

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

Istnieje możliwość skróconego zapisu warunków z użyciem łączenia instrukcji przy pomocy && (wykonaj gdy poprzednia zwróciła zero – true) lub || (wykonaj gdy poprzednia zwróciła nie zero – false):

```
[ -f /etc/issue ] && echo "jest plik /etc/issue"
grep '^root:' /etc/passwd > /dev/null && echo /etc/passwd zawiera root-a;
```

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 bashu 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() {
    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 argumentów pozycyjnych o 1
              # wpływa na wartości $@ $# itp
    done

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

Zwróć uwagę że w nawiasach po nazwie funkcji nie podajemy przyjmowanych argumentów, natomiast puste nawiasy te są elementem składniowym i muszą wystąpić. Jeżeli zapisujesz definicję funkcji w jednej linii, np. `abc() { echo "abc"; }` to pamiętaj, że spacja po otwierającym nawiasie klamrowym jest obowiązkowa, podobnie jak średniki występujące po każdej (także ostatniej) instrukcji w ciele funkcji.

Wywołanie funkcji nie różni się niczym od wywołania programów czy instrukcji wbudowanych (możemy używać przekierowań strumieni wejścia, wyjścia, czy też przechwycić wyjście do zmiennej). Powyższą funkcję możemy wywołać np. w następujący sposób: `f1 a "b c" d`

1.6 Grupowanie poleceń

Funkcje są przykładem grupowania poleceń – funkcja stanowi nazwany blok kodu, czyli nazwaną grupę poleceń. Polecenia w bashu możemy też grupować bez definiowania funkcji. W tym celu możemy zastosować nawiasy klamrowe (tak jak w definicji funkcji) lub nawiasy okrągłe.

Stosując nawiasy klamrowe musimy 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.

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
```

Zauważ że 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`
- `${str:n}` zwróci pod-napis zmiennej `str` od `n` do końca
- `${str:n:m}` zwróci pod-napis zmiennej `str` od `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
- `${!x}` zwróci wartość zmiennej, której nazwa jest w zmiennej `x`

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 (zwróć uwagę na brak cytowania 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`⁵:

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

```
0o17 8 0x3
```

5. Instrukcja `printf` ma składnię opartą na tej funkcji z C, interpretuje ona także liczby zmiennoprzecinkowe.

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.

```
# obliczanie długości napisu w znakach, w bajtach i ilości słów w napisie
echo -n "aąbcć 123" | wc -m
echo -n "aąbcć 123" | wc -c
echo -n "aąbcć 123" | 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 tekst którym ma zostać zastąpione, a trzeci gdy jest g to powoduje zastępowanie wszystkich wystąpień a nie tylko pierwszego.

Należy zwrócić uwagę na różnicę w składni wyrażenia regularnego polegającą na poprzedzaniu (,) i + odwrotnym ukośnikiem aby miały znaczenie specjalne (jeżeli nie chcemy tego robić możemy 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 umożliwiający 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 :) 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 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 { komenda }, 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 '{
    # dla każdego pola w rekordzie
    for (i=1; i<=NF; ++i) {
        # jeżeli jego numer jest parzysty
        # to zastąp wszystkie ciągi b pojedynczym B
        if (i%2==0)
            gsub("b+", "B", $i);

        # wyszukaj pozycję pod-napisu B
        ii = index($i, "B")
        # jeżeli znalazł
        # to wypisz pozycję i pod-napis od tej pozycji do końca
        if (ii)
            printf("# %d %s\n", ii, substr($i, ii))
        # zwróć uwagę że w AWK liczy elementy napisy 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=""}
  {slova[$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 slova) printf("%s: %s\n", s, slova[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\nccc" | 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 informację 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 zapisane mamy 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 podanych w napisie w momencie uruchamiania go, itp.

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 możemy 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, .
```

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

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

C=`export x; echo $A | 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 wiążą 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 to 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 (co, dzięki użyciu przekierowania strumieni, tylko pozornie może wydawać się bezsensowne – na przykład wywołanie `cat > plik` pozwala nam wprowadzić treść do wskazanego pliku, przypomnij sobie także przesyłanie strumieniowe plików z użyciem `ssh`).

Wyjście `cat` przekierowane jest do pliku, jednak zwraca tutaj uwagę inny operator - dwa znaki mniejszości (<<). Powoduje on że tekst podawany w kolejnych liniach będzie kierowany na standardowe wejście

komendy po której wystąpił, aż do momentu napotkania linii zawierającej jedynie słowo podane po nim (w tym wypadku klasyczny EOF, ale może to być dowolne inne słowo - np. KONIEC, lub nawet ciąg znaków ze spacjami ujęty w cudzysłowa).

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.

3.2.2 standardowe wejście/wyjście w miejscu ścieżki do pliku

Wiele programów jeżeli w miejscu w którym oczekuje ścieżki do pliku otrzyma myślник 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 <(cat /etc/passwd) <(cat /etc/passwd-)
```

które poleceniu diff jako jeden plik postawia standardowe wyjście pierwszego cat, a jako drugi plik drugiego cat. Oczywiście zaprezentowane zastosowanie tego z poleceniami cat jest bezsensowne (prościej podać ścieżki do plików), ale gdyby występował tam już np. jakiś grep mogłoby to być użyteczne. Jest to rozszerzenie bashowe nie występujące w czystym sh.

3.2.3 polecenia wykonywane na zakończenie

Jak wiemy 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).

4 Wykład wideo⁶

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

6. 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`

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 zadania bez zaglądania do odpowiedzi.

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.

- w nawiasach po nazwie funkcje nie piszemy nic na temat jej argumentów - one są puste
- do argumentów odwołujemy się poprzez dolar numer argumentu
- do n krotnego powtórzenia czynności używamy pętli for która iteruje po liście liczb zwracanej przez seq

Zwróć uwagę że:

```

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

```

Rozwiązanie zadania 1.6.1

- w celu warunkowego wypisania jakiegoś 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:

```

[ -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

- rozwiązania 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 gniazdek (czyli wszystkich plików nieukrytych)
- pętla while listę plików dostaje na standardowe wejście (jedyn 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 podwojnych cudzysłowach (aby móc umieścić w nim zmianę), 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

- 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 składniowo

Rozwiązanie zadania 1.6.2

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

```
koty() {
  if [ $1 -eq 1 ]; then
    echo "Ala ma kota"
  elif [ $1 -lt 5 ]; then
    # moznaby tez elif [ $1 -eq 2 -o $1 -eq 3 -o $1 -eq 4
    ↵ ]; then
    echo "Ala ma $1 koty"
  else
    echo "Ala ma $1 kotów"
  fi
}
```

Rozwiązanie zadania 2.1.1

```
for f in *.TXT; do mv "$f" "${basename $f}.txt"; done

# albo:

for i in *.TXT; do mv "$i" "${basename $i}.txt"; done
```

Do iteracji po plikach używamy pętli 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 | fltrować daną nazwę w każdym obiegu pętli lub użyć potoku ls | grep | while read ...).
W takim przypadku \${f%.TXT}.txt jest lepsze od \${f}/.TXT/.txt lub \${f}//.TXT/.txt} gdyż usuwa tyłko ciąg .TXT występujący na końcu napisu. aa.TXT.xyz.TXT jest poprawną nazwą pliku i zgodna z warunkami zadania powinna być zmieniana na aa.TXT.xyz.txt, a nie na aa.txt.xyz.TXT lub aa.txt.xyz.txt.

Rozwiązanie zadania 2.2.1

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

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


- zignorowanie linii zaczynającej się od "wtmp begins" z użyciem dopasowania wyrażenia regu- larnego i komendy next
- zliczanie wystąpień nazwy użytkownika w tablicy x (indeksowanej tymi nazwami)
- zastosowanie bloku END do wypisania zawartości tablicy, której używaliśmy do zliczania użyt- kowników

```
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

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 większo- ści 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@20g" -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
```

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