

Wyrażenia regularne i filtry tekstowe

Witold Paluszyński
witold.paluszynski@pwr.edu.pl
<http://kcir.pwr.edu.pl/~witold/>

Copyright © 1995–2010 Witold Paluszyński
All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat wyrażeń regularnych i opartych na nich filtrów do przetwarzania danych tekstowych: grep'a, sed'a, awk'a, itp. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Manipulacje na stringach — cut i expr

Wycinanie fragmentów stringów możemy osiągnąć programem `cut`

```
fraza="A mnie jest szkoda lata."
echo $fraza | cut -c3-18          # znaki od 3 do 18
echo $fraza | cut -d" " -f3,4     # trzecie i czwarte slowo
pierwsze_dwa=' echo $fraza | cut -d" " -f1-2 '
```

Poznany już program `expr` posiada operator `:` wykonujący dopasowanie wyrażeń regularnych. Traktuje on drugi argument jako wyrażenie regularne i dopasowuje go do pierwszego argumentu. W najprostszym przypadku `expr` zwraca liczbę dopasowanych znaków.

```
pierwsze_trzy=' echo $fraza | cut -d" " -f1-3 '
dlugosc_trzy='expr "$pierwsze_trzy" : '.*''
od_czwartego='expr $dlugosc_trzy + 2'
reszta='echo $fraza | cut -c${od_czwartego}-'
nowa_fraza=${pierwsze_dwa}" nie "${reszta}
echo $nowa_fraza
==> A mnie nie szkoda lata.
```

Jeśli wyrażenie regularne dane jako drugi argument zawiera operatory `\(...\)` to wynikiem działania operatora dopasowania jest dopasowany string.

```
fraza="Ostateczna ocena: bardzo dobra"
jaka_ocena='expr "$fraza" : '.*ocena.*: \(.*\)${}'
```

Możemy również sprawdzić tylko czy nastąpiło dopasowanie testując status.

```
if expr "$imie" : '.*[aA]${}' > /dev/null
then
    echo Otrzymała Pani ocene: $jaka_ocena
else
    echo Otrzymałaś ocene: $jaka_ocena
fi
```

Wyrażenia regularne (1): podstawy

Jednoznakowe wyrażenia regularne:

- `.` — kropka pasuje do każdego znaku, dokładnie jednego
- `[abcdA-Z]` — ciąg znaków w nawiasach kwadratowych pasuje do każdego znaku z wymienionych, albo należącego do przedziału
- `[^a-zA-Z0-9]` — strzałka na początku w nawiasie kwadratowym oznacza dopełnienie, tu znak niealfanumeryczny
- dowolny znak niespecjalny — pasuje wyłącznie do samego siebie

Powtórzenia:

- $d_1d_2\dots d_n$ — ciąg wyrażeń dopasowuje się do ciągu znaków jeśli każde wyrażenie dopasowuje się do podciągu w sekwencji
- d^* — gwiazdka następująca za jednoznakowym wyrażeniem regularnym d oznacza powtórzenie dopasowania do dowolnej długości (również zerowej) ciągu znaków; każdy znak jest oddzielnie dopasowywany do wyrażenia d

„Kotwice”:

- $\wedge$ — pasuje do zerowego ciągu znaków, ale tylko na początku ciągu
- $\$$ — analogicznie pasuje tylko na końcu łańcucha znaków

Wyrażenia regularne (2): proste przykłady

- `[0-9]` pasuje do pojedynczej cyfry (jak również dowolnego stringa, który taką zawiera)
- `[0-9]*` pasuje do dowolnego ciągu cyfr, w tym również pustego stringa
- `[1-9][0-9]*` pasuje do dowolnego niepustego ciągu cyfr, bez wiodącego zera
- `[A-Z][a-z]*` pasuje do napisu znakowego zaczynającego się od dużej litery
nie pasuje do napisu bez dużych liter
ale pasuje do napisu z więcej niż jedną dużą literą na początku!!
również pasuje do dowolnego zapisu z dużymi literami na końcu!!
- `^[A-Z][a-z]*$` pasuje do dowolnego ciągu małych liter, również pustego, zaczynającego się od jednej dużej litery, i niczego! innego
- `^[A-Z]$[a-z]*` niepoprawne wyrażenie regularne
- `[a-z]* [a-z]*` pasuje do dwóch dowolnych ciągów liter rozdzielonych spacją
w tym również pasuje do dowolnego ciągu dowolnych znaków zawierającego spację

Szybki test: do czego dopasowują się następujące wyrażenia?

`[A-Z][a-z][a-z]*[ ][ ]*[A-Z][a-z][a-z]*`
`[_a-z][_a-z0-9]*`
`0[1-9][0-9]-[1-9][0-9][0-9]-[0-9][0-9][0-9][0-9]`

Wyszukiwanie wzorców – grep

```
grep money *
cat * | grep money
grep -n Count *.ch
grep '^From' $MAIL | grep -v 'From szef'
grep -i kowalski spis.telef
ls -l | grep -v '[cho]$'
ls -l | grep '^.....w'
grep '^[^:]*::' /etc/passwd
cat dictionary | grep '^..w.w..e.t$' # ekwiwalent
cat text | grep '\([-A-Za-z][-A-Za-z]*\) [ ]*\1'
egrep 'socket|pipe|msgget|semget|shmget' *.ch
```

pisanie znaków specjalnych grep'a w shell'u, co znaczą poniższe wyrażenia?

grep \\ grep \\ grep \\$ grep \\$ grep \\$	grep '\\$' grep '\\$' grep \"\$" grep '\$' grep "\$"
--	--

Wyrażenia regularne (3): grep i egrep

w kolejności malejącego priorytetu:

z	dowolny znak niespecjalny pasuje do siebie samego
$\backslash z$	kasuje specjalne znaczenie znaku z
$\wedge$	początek linii
$\$$	koniec linii
$\cdot$	dowolny pojedynczy znak
$[abc\dots]$	dowolny znak spośród podanych, też przedziały, np. a-zA-Z
$[\wedge abc\dots]$	dowolny znak spoza podanych, również mogą być przedziały
$\backslash n$	to do czego dopasowało się n -te wyrażenie $\backslash(r\backslash)$ (tylko grep)
r^*	zero lub więcej powtórzeń wyrażenia r
r^+	jedno lub więcej powtórzeń wyrażenia r (tylko egrep)
$r^?$	zero lub jedno wystąpienie wyrażenia r (tylko egrep)
r_1r_2	r_1 i następujące po nim r_2
$r_1 r_2$	r_1 lub r_2 (tylko egrep)
$\backslash(r\backslash)$	zapamiętane wyrażenie regularne r (tylko grep)
(r)	wyrażenie regularne r (tylko egrep)

Krótkie podsumowanie — pytania sprawdzające

1. Jaką rolę w wyrażeniach regularnych pełni gwiazdka?
 2. Jaką rolę w wyrażeniach regularnych pełni znak minus?
 3. Jaką rolę w wyrażeniach regularnych pełni nawiasy kwadratowe?
 4. Czy wyrażenie regularne $'^{\wedge}[AH][BH]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 5. Czy wyrażenie regularne $'^{\wedge}[AH][BH][A-Z]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 6. Czy wyrażenie regularne $'[B-H][B-H]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 7. Czy wyrażenie regularne $'[AH][AH]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 8. Czy wyrażenie regularne $'[AH][AH]^{\$}$ dopasuje się do napisu 'ALHAMBRA' ?
 9. Czy wyrażenie regularne $'^{\wedge}[A-H][A-H]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
-
10. Czy wyrażenie regularne $'^{\wedge}[A-H][A-H]^{\$}$ dopasuje się do napisu 'ALHAMBRA' ?
 11. Czy wyrażenie regularne $'^{\wedge}[A-H]^{\$}$ dopasuje się do napisu 'ALHAMBRA' ?
 12. Czy wyrażenie regularne $'^{\wedge}[A-Z]^{\$}$ dopasuje się do napisu 'ALHAMBRA' ?
 13. Czy wyrażenie regularne $'[B-Z][B-Z]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 14. Czy wyrażenie regularne $'^{\wedge}[B-Z][B-Z]^{'}$ dopasuje się do napisu 'ALHAMBRA' ?
 15. Czy wyrażenie regularne $'[B-Z][B-Z]^{\$}$ dopasuje się do napisu 'ALHAMBRA' ?
 16. Czy wyrażenie regularne $'\([A-L]\)1'$ dopasuje się do napisu 'VALHALLA' ?
 17. Czy wyrażenie regularne $'\([A-H]\)[A-H]^{\$}1'$ dopasuje się do napisu 'VALHALLA' ?
-

Sed: edytor strumieniowy

Edytor strumieniowy **sed** (*stream editor*) wczytuje dane z wejścia wiersz po wierszu, na wczytanym wierszu wykonuje operacje zadane argumentem, i przetworzony wiersz wysyła na wyjście.

Format polecenia:

[adres ₁ [,adres ₂]]	operator	[argumenty[modyfikatory]]
---	----------	---------------------------

Adres w poleceniu **seda** może być liczbą lub wzorcem (wyrażeniem regularnym). Operacja jest wykonywana tylko na wierszu, którego dotyczy adres, albo w przedziale wierszy określonym adresami (jeśli są dwa).

Operatory: **d** wykasuj zawartość bufora (nic nie będzie wysłane na wyjście)
p wyślij na wyjście zawartość bufora (oprócz wyśw.domyślnego)
q zakończ pracę (po przetworzeniu bieżącego wiersza)

```
sed 10q                # przepuszcza 10 pierwszych linii
sed /wzorzec/q          # wyświetla do linii z wzorcem
sed /wzorzec/d           # opuszcza linie z wzorcem (grep -v)
sed '/^$/d'             # opuszcza puste linie
sed -n /wzorzec/p        # wyświetla tylko linie z wzor.(grep)
sed -n '/\\begin{verbatim}/,\\/\\end{verbatim}/p'
```

Oprócz przedstawionych wyżej operatorów **seda**: **d**, **p**, i **q**, najczęściej przydatnym jest operator podmiany stringów **s**. Wymaga on podania dwóch stringów jako argumentów po symbolu operatora. Pierwszym znakiem po **s** jest wybrany znak separatora, a potem dwa argumenty. Normalnie podmiana jest wykonywana jeden raz w wierszu, ale podanie modyfikatora **g** powoduje wykonanie podmiany dowolną liczbę razy.

```
sed 's/marzec/March/g'  # podmiana stringow (wiele razy)
sed 's/^/^I/'           # indentacja (taby na pocz.linii)
sed '/./s/^/^I/'        # ulepszona indentacja
```

Pierwszy argument operatora **s** jest traktowany jako wyrażenie regularne typu **grepa**, tzn. może zawierać operacje zapamiętywania **\(...\)**. Wtedy drugi argument może zawierać odwołania do zapamiętanych stringów **\1**, **\2**, itd. W przypadku wersji Gnu **seda**, możliwe jest również alternatywne stosowanie wyrażeń regularnych **egrepa**. Operacja zapamiętywania jest wtedy niedostępna.

Sed posiada jeszcze kilka bardziej skomplikowanych operatorów, które wraz z sekwencjami pozwalają na pisanie złożonych wyrażeń, które są niekiedy bardzo trudne do zrozumienia i debugowania. Nie zmienia to faktu, że bardzo wiele przydatnych operacji można zrealizować czterema powyższymi operatorami.

Sed: przykład — komedia pomyłek

```
sierra-90> who
NAME          LINE          TIME          IDLE          PID  COMMENTS
witold    + vt04          Oct 21 04:46    2:45          238
witold    + tttyp0        Oct 21 04:46    2:43          292
witold    + tttyp1        Oct 21 04:46    .              291
witold    + tttyp2        Oct 21 04:46    .              290
sierra-91> who | sed 's/ .* / /'
NAME COMMENTS
witold
witold 292
witold 291
witold 290
sierra-92> who | sed 's/ .* [^ ]/ /'
NAME OMMENTS
witold 38
witold 92
witold 91
witold 90
sierra-93> who | sed 's/ .* \([^ ]\) / \1/'
```

Sed: kontynuacja przykładu

Jako wniosek z analizy powyższego przykładu, rozważmy zadanie napisania skryptu `seda`, który, filtrując ciąg wejściowy, wyświetli na wyjściu tylko pierwsze słowo (dla uproszczenia) z każdego wiersza. Rozważ poniższe rozwiązania tego zadania. Które z nich zawierają błędy, a które działają w pełni niezawodnie? Czym różni się działanie tych wersji „niezawodnych” między sobą?

```
sed 's/ .*$/ /'
sed 's/\([^ ]\) .*$/\1/'
sed 's/\([^ ]*\) .*$/\1/'
sed 's/\([a-zA-Z]*\) .*$/\1/'
sed 's/\([a-zA-Z][a-zA-Z]*\) .*$/\1/'
sed 's/\([^ ]*\) .*$/\1/'
sed 's/\([^ ]*\) .*$/\1/'
sed 's/[ ]*\([^ ]*\) .*$/\1/'
```

Dla porównania rozważ możliwość wykorzystania następujących mechanizmów POSIX-owych (patrz poniżej) do realizacji zadania: `\<...>`, `[:alpha:]` i `[:space:]`. Spróbuj napisać dobre rozwiązanie problemu wykorzystując te mechanizmy. Które z nich stanowią istotne ulepszenie wersji nie-POSIX-owej?

Sed: przykład zaawansowany

Poniższy przykładowy skrypt `seda` skracą sekwencje pustych linii do pojedynczej pustej linii wykorzystując polecenie wczytywania kolejnych wierszy (`N`) i pętlę zrealizowaną przez skok do etykiety (`b`):

```
# pierwszy pusty wiersz jawnie wypuszczamy na wyjście
/^$/p
:Empty
# dodajemy kolejne puste wiersze usuwając znaki NEWLINE
/^$/ { N;s/.//;b Empty
}
# mamy wczytany niepusty wiersz, wypuszczamy go
{p;d;}
```

Skrypt w pełni kontroluje co jest wyświetlane na wyjściu i działa tak samo wywołany z opcją `-n` jak i bez niej.

Podobnie jak następujący, zaledwie dziesięcioznakowy skrypt który wyświetla plik wejściowy w odwrotnej kolejności wierszy: `1!G;h;$p;d`

Sed: podstawowe operatory

<code>a\</code>	wyprowadź na wyjście kolejne linie do linii nie zakończonej \
<code>b etyk</code>	skok do etykiety
<code>c\</code>	zmień linie na następujący tekst, jak dla a
<code>d</code>	skasuj linię
<code>i\</code>	wyprowadź następujące linie przed innym wyjściem
<code>l</code>	wyświetl linię, z wizualizacją znaków specjalnych
<code>p</code>	wyświetl linię
<code>q</code>	zakończ
<code>r plik</code>	wczytaj plik, wypuść na wyjście
<code>s/s₁/s₂/z</code>	zastąp stary tekst <i>s₁</i> nowym <i>s₂</i> ; jeden raz gdy brak modyfikatora <i>z</i> , wszystkie gdy <i>z=g</i> , wyświetlaj podstawienia gdy <i>z=p</i> , zapisz na pliku gdy <i>z=w</i> <i>plik</i>
<code>t etyk</code>	skok do etykiety, gdy w bieżącej linii dokonane podstawienie
<code>w plik</code>	zapisz linię na pliku
<code>y/s₁/s₂/</code>	zamień każdy znak z <i>s₁</i> na odpowiedni znak z <i>s₂</i>
<code>=</code>	wyświetl bieżący numer linii
<code>!polec</code>	wykonaj polecenie <code>seda</code> <i>polec</i> gdy bieżąca linia nie wybrana
<code>:etyk</code>	etykieta dla poleceń <code>b</code> i <code>t</code>
<code>{...}</code>	grupowanie poleceń

Wyrażenia regularne (4): BRE i ERE

Specyfikacja POSIX porządkuje i rozszerza oryginalną koncepcję wyrażeń regularnych Unixa. Uwzględnia ona, między innymi, specyfikację powtórzeń, klasy znaków, oraz lokalizacje, tzn. stosowany w danej lokalizacji zestaw znaków i konwencje równoważności i uporządkowania. Stanowi rozszerzenie wyrażeń regularnych grepa i egrepa, ale ze względu na ich wzajemną niekompatybilność, jej wynikiem jest definicja dwóch języków wyrażeń regularnych: BRE (Basic Regular Expressions) i ERE (Extended Regular Expressions).

W największym skrócie, można zapamiętać:

BRE (zgodne z grepem) — wyrażenia regularne z operatorem zapamiętywania "`\( . . . \)`" i odwoływania się do dopasowanych, zapamiętanych stringów `\1`, `\2`, ...

ERE (zgodne z egrephem) — wyrażenia regularne z operatorem alternatywy "`( . . . | . . . )`" gdzie nawiasy nie są obowiązkowe, ale są elementem składni

Dodatkowo język ERE zawiera pomocnicze operatory `?` (opcjonalnego wystąpienia poprzedzającego wyrażenia), oraz `+` (powtórzenia co najmniej jeden raz).

Wyrażenia regularne (5): powtórzenia

$r\{n, m\}$ powtórzenie: n -razy, $n - m$ -razy, lub co najmniej n -razy ([grep](#))
 $r\{n, m\}$ powtórzenie: n -razy, $n - m$ -razy, lub co najmniej n -razy ([egrep](#))

Wyrażenia regularne (6): klasy znaków

Standard POSIX rozszerzył wyrażenie `[]` dopasowujące jeden znak o klasy znaków za pomocą wyrażenia `[[:klasa:]]`, z następującymi klasami znaków:

```
[[:alnum:]]  
[[:lower:]]  
[[:alpha:]]  
[[:print:]]  
[[:blank:]]  
[[:punct:]]  
[[:cntrl:]]  
[[:space:]]  
[[:digit:]]  
[[:upper:]]  
[[:graph:]]  
[[:xdigit:]]
```

Wyrażenia regularne (7): przykłady wyrażeń BRE i ERE

Niektóre wyrażenia mają złożoną składnię i wymagania. Na przykład, adresy email:

```
username@domain-spec
```

Nazwa użytkownika musi być dowolnym niepustym ciągiem liter, cyfr, kropki, podkreślnika (podłogi), minusa i plusa.

Specyfikacja domeny musi składać się z niepustej liczby powtórzeń domen, rozdzielonych kropkami.

Domena musi być niepustym ciągiem liter, cyfr, minusa i podkreślnika (podłogi). Plusy i kropki są wykluczone (ale kropki występują między domenami).

Jako przypadek szczególny, domena główna (ostatni człon) musi składać się wyłącznie z liter, jednak nie mniej niż dwóch i nie więcej niż pięciu.

```
^[a-zA-Z0-9_-\.\+]+@([a-zA-Z0-9_-\.\+)+[a-zA-Z]{2,5}$
```

Zauważmy wygodę wielokrotnego użycia operatorów powtórzenia 1 lub więcej razy `(+)` oraz operatora powtórzenia od 2 do 5 razy `{2,5}`.

Uniwersalny filtr programowalny – awk

- czyta wiersz z wejścia, dzieli na pola (słowa) dostępne jako: \$1, \$2, ...
- wykonuje cały swój program składający się z szeregu par: warunek-akcja
- par warunek-akcja może być wiele i w każdej może brakować warunku (domyślnie: prawda) albo akcji (domyślnie: wyświetlenie wiersza)
- w programie można używać zmiennych, które zachowują wartości pomiędzy kolejnymi wierszami
- zmiennych nie trzeba deklarować ani inicjalizować; są inicjalizowane w pierwszym użyciu wartością 0 lub pustym stringiem, zależnie od operacji

```
# program awk'a może zawierać tylko warunki
ls -l ~student | awk ' $5 > 100000 '
```

```
# może również zawierać tylko akcje
awk '{print $2,$1}' nazwa_pliku
cat /etc/passwd | awk -F: ' { print $4, $3 }'
```

```
# warunki i akcje: tu występuje operator dopasowania
# stringa do wzorca zadanego wyrażeniem regularnym
awk -F: ' $7 ~ /bash$/ { printf "%-s: %-s",$1,$3 }' /etc/passwd
```

```
# użycie zmiennych do zapamiętania kontekstu między wierszami
awk ' $1 != prev { print; prev = $1 } '
```

```
# użycie zmiennych wbudowanych awka: NF i NR
awk ' NF > 5 { print "linia ",NR," za długa" } '
```

```
# przykład funkcji wbudowanej awka
awk ' { wd+=NF; ch+=length($0)+1 } END { print NR,wd,ch } '
```

```
# mechanizmy ustawiania wartości początkowych zmiennych
awk ' BEGIN { var1=0 } { ... } ' var1=-1
```

```
# używanie pól wejściowych jak zmiennych
awk ' $1 < 0 { $1 = 0 } $1 > 100 { $1 = 100 } { print $0 } '
awk ' NF > 8 { print $(NF-2) } '
```

```
# przekazywanie argumentu do skryptu
```

```
awk ' { s += $1 } END { print s }'  
awk ' { s += '$1' } END { print s }'
```

```
# petle
```

```
awk ' BEGIN { x=1;y=1; for (i=1; i<=20; i++) { \  
                                print y;z=x; x=x+y; y=z} } ' < /dev/null
```

```
# tablice asocjacyjne
```

```
awk ' { sum[$1] += $2 } \  
      END { for (name in sum) print name, sum[name] }'  
awk ' { for (i=1; i<=NF; i++) freq[$i]++ } \  
      END { for (word in freq) print word, freq[word] }'
```

awk: przykład z logami spoolera

W dalszym ciągu przedstawiony został przykładowy zestaw skryptów awk'a napisanych w celu podsumowania wykorzystania drukarek przez grupę użytkowników, na potrzeby rozliczeń. Spooler drukarki rejestruje każde drukowane zadanie z dużą liczbą szczegółów, w pliku, którego format — jakkolwiek dość jasny i konsekwentny — nie jest nigdzie formalnie udokumentowany. Potrzebne było narzędzie, które pozwoliłoby na bieżąco podsumowywać wykorzystanie drukarki ze względu na użytkowników, będące jednocześnie elastyczne i łatwe do modyfikacji, gdyby odkryte zostały nieregularności w pliku danych, albo zmieniły się potrzeby.

Zadanie zostało rozwiązane przez zestaw skryptów awk'a, które kolejno: (1) zamieniały nie do końca zrozumiałe rejestry spoolera na proste, jednolinijkowe podsumowania drukowanych zadań, (2) wybierały naturalnie i wygodnie zadany okres rozliczenia, i (3) dokonywały sumowania ze względu na nazwę użytkownika.

Najtrudniejszym zadaniem było przetworzenie logu spoolera lpsched na postać łatwą do dalszej obróbki.

Przykład z logami spoolera — dane wejściowe

```
= hp5_q-636, uid 71, gid 0, size 48121, Mon Oct 27 09:10:42 CET 2003
y /etc/lp/interfaces/hp5_q
z hp5_q
C 1
D hp5_q
F /var/spool/lp/tmp/rab/636-1
O nobanner flist='(lpr_filter)'
P 20
T 636-1
t postscript
U kreczmer@rab
s 0x0010
v 0
= hp5_q-168, uid 71, gid 0, size 47960, Mon Oct 27 09:23:43 CET 2003
z hp5_q
C 1
D hp5_q
F /var/spool/lp/tmp/rab/168-1
O nobanner flist='(lpr_filter)'
P 20
T 168-1
t simple
U mw@rab
s 0x0010
v 0
```

Przykład z logami spoolera — przetwarzanie danych

```
# Copyright 1993 Witold Paluszynski
```

```
# All rights reserved.
```

```
# NAME: lpsumrequests -- summarize lp print jobs by users
```

```
# SYNOPSIS: lpsumrequests
```

```
# DESCRIPTION: wybiera ze strumienia wejsciowego, ktory musi
```

```
#      miec format rejestru /usr/spool/lp/logs/requests,
```

```
#      informacje o wykonanych zadaniach drukowania i
```

```
#      wypuszcza zwiezly skrot, po 1 linijce
```

```
awk ' ,
```

```
    BEGIN { jobid = "" ; user = "unknown" ; filecount = 0 }
```

```
    $1 == "U" { user = $2 }
```

```
    $1 == "F" { filecount++
```

```
                filenames[filecount] = $2 }
```

```
    $1 == "=" && jobid != "" ) {
```

```
        printf "%s %s %s",user,size,date
```

```
        for (i = 1 ; i <= filecount ; i++)
```

```
            printf " %s",filenames[i]
```

```
        printf "\n"
```

```
    }
```

```
    $1 == "=" {
```

```
        filecount = 0
```

```
        user = "unknown"
```

```
        jobid = $2
```

```
        size = substr($8,1,length($8)-1)
```

```
        date = $9 " " $10 " " $11 " " $12 " " $13
```

```
    }
```

```
END {
```

```
    if ( jobid != "" ) {
```

```
        printf "%s %s %s",user,size,date
```

```
        for (i = 1 ; i <= filecount ; i++)
```

```
            printf " %s",filenames[i]
```

```
        printf "\n"
```

```
    }
```

```
    } ,
```

Przykład z logami spoolera — wybór i sumowanie

```
# Copyright 1993 Witold Paluszynski
# All rights reserved.

# NAME: lptotalsum -- total up print job sizes by users,years,months
# SYNOPSIS: lptotalsum [year [month]]
# DESCRIPTION: sumuje ilosc wydrukow wedlug uzytkownikow na
#              podstawie zestawienia wyprodukowanego przez script
#              lpsumrequests, przy czym jesli dane sa parametry $1 i $2
#              to tylko w danym roku i miesiacu

awk ' $7 ~ /^'$1'/ && $4 ~ /^'$2'/ ' | awk '
BEGIN { year = "'$1'" ; month = "'$2'"
        if (year == "") year = "ALL"
        if (month == "") month = "ALL"
      }
      { jobsizes[$1] += $2 ; jobcount[$1]++ }
END {
  printf "Print job summary for year: %s, month: %s\n\n",year,month
  for (user in jobsizes)
    printf "User %s, print job count %s, total size %d\n", \
          user,jobcount[user],jobsizes[user]
}
```

awk: zmienne wbudowane

FILENAME	nazwa bieżącego pliku wejściowego
FS	znak podziału pól (domyślnie spacja i tab)
NF	liczba pól w bieżącym rekordzie
NR	numer kolejny bieżącego rekordu
OFMT	format wyświetlania liczb (domyślnie %g)
OFS	napis rozdzielający pola na wyjściu (domyślnie spacja)
ORS	napis rozdzielający rekordy na wyjściu (domyślnie linefeed)
RS	napis rozdzielający rekordy na wejściu (domyślnie linefeed)

awk: operatory

w kolejności rosnącego priorytetu:

<code>= += -= *= /= %=</code>	operatory przypisania podobne jak w C
<code> </code>	alternatywa logiczna typu „short-circuit”
<code>&&</code>	koniunkcja logiczna typu „short-circuit”
<code>!</code>	negacja wartości wyrażenia
<code>> >= < <= == !=</code>	operatory porównania
<code>~ !~</code>	(nie)dopasowanie wyrażeń regularnych do napisów
<code>nic</code>	konkatenacja napisów
<code>+ -</code>	plus, minus
<code>* / %</code>	mnożenie, dzielenie, reszta
<code>++ --</code>	inkrement, dekrement (prefix lub postfix)

awk: funkcje wbudowane

<code>cos(expr)</code>	kosinus, argument w radianach
<code>exp(expr)</code>	e^{expr}
<code>getline()</code>	czyta następną linię z wejścia
<code>index(s1,s2)</code>	pozycja napisu s2 w s1; zwraca 0 jeśli nie ma
<code>int(expr)</code>	część całkowita
<code>length(s)</code>	długość napisu znakowego
<code>log(expr)</code>	logarytm naturalny
<code>sin(expr)</code>	sinus, argument w radianach
<code>split(s,a,c)</code>	podziel napis s względem c na części do tablicy a
<code>sprintf(fmt,...)</code>	formatowanie napisu
<code>substr(s,m,n)</code>	n-znakowy podciąg s począwszy od pozycji m

Inne przydatne filtry Uniksa

Warto znać podstawowy zestaw filtrów tekstowych Uniksa, ponieważ realizują one bardzo proste algorytmy, które łatwo zapamiętać i ich używać. Jednocześnie łączenie tych filtrów pozwala czasem zaimplementować całkiem zaawansowane funkcje.

<code>sort</code>	sortowanie wierszy pliku
<code>tr</code>	zamiana znaków
<code>uniq</code>	unikanie powtórzeń
<code>join</code>	bazodanowy operator join
<code>tac</code>	wyświetlaj zawartość plików od końca
<code>rev</code>	wyświetlaj pliki jak cat, ale odwracając kolejność znaków w wierszach
<code>paste</code>	łącz i wyświetlaj jako jeden wiersz kolejne wiersze z wielu plików

`sort` — sortowanie wierszy

```
awk -F: '{print $5}' /etc/passwd | sort +1 -2 +0
```

uniq — usuwanie powtórzeń wierszy

```
awk -F: '{print $5}' /etc/passwd|awk '{print $1}'|sort|uniq -c
```

tr — zamiana znaków

```
alias 8859-2-to-windows tr \  
'\261\352\346\263\361\363\266\274\277' \  
'\245\251\206\210\344\242\230\253\276'
```

join — łączenie rekordów z różnych plików

```
# lista uzytkownikow z symbolicznymi nazwami grup
sort -t: -k4 /etc/passwd > /tmp/passwd
sort -t: -k3 /etc/group > /tmp/group
join -j1 4 -j2 3 -o 1.1 2.1 1.6 -t: /tmp/passwd /tmp/group

# polaczenie dwoch list numerow telefonow
cat /tmp/phone                                cat /tmp/fax
!Name    Phone Number                        !Name    Fax Number
Don       +1 123-456-7890                     Don       +1 123-456-7899
Hal       +1 234-567-8901                     Keith     +1 456-789-0122
Yasushi   +2 345-678-9012                     Yasushi   +2 345-678-9011

join -t"<tab>" -a 1 -a 2 -e '(unknown)' -o 0,1.2,2.2 \
                                           /tmp/phone /tmp/fax
```

WAŻNE: oba pliki wejściowe muszą być posortowane według pola, na którym wykonywane jest połączenie.

Łączenie filtrów

```
cat * | tr -cs "[A-Z][a-z]" "[\012*]" \
      | sort \
      | uniq -c \
      | sort -nr \
      | head
```

Krótkie podsumowanie — pytania sprawdzające

1. Jaką rolę pełnią wyrażenia regularne w programie `sed`?
 2. Wymień co najmniej dwie istotne różnice pomiędzy programami `sed` i `awk`.
 3. Opisz działanie programu `join`.
 4. Opisz działanie programu `tr`.
-