Wyrażenia regularne i filtry tekstowe

Witold Paluszyński
witold.paluszynski@pwr.wroc.pl
http://sequoia.ict.pwr.wroc.pl/~witold/

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 Otrzymala Pani ocene: $jaka_ocena
else
    echo Otrzymales 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:

$d_1d_2...d_n$	—	ciąg wyrażeń dopasowuje się do ciągu znaków jeśli każde wyrażenie dopasowuje sie 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":

^	—	pasuje do zerowego ciągu znaków, ale tylko na początku ciągu
$	—	analogicznie pasuje tylko na końcu łańcucha znaków

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 (2): grep i egrep

w kolejności malejącego priorytetu:

z	dowolny znak niespecjalny pasuje do siebie samego
\z	kasuje specjalne znaczenie znaku z
^	początek linii
$	koniec linii
.	dowolny pojedynczy znak
[abc...]	dowolny znak spośród podanych, też przedziały, np. a-zA-Z
[^abc...]	dowolny znak spoza podanych, również mogą być przedziały
\n	to do czego dopasowało się n-te wyrażenie \(r\) (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
$r_1 \mid r_2$	r_1 lub r_2 (tylko egrep)
\(r\)	zapamiętane wyrażenie regularne r (tylko grep)
(r)	wyrażenie regularne r (tylko egrep)
	żadne wyrażenie regularne nie pasuje do znaku nowej linii

Edytor strumieniowy – sed

```
sed 10q              # przepuszcza 10 pierwszych linii
sed '/wzorzec/q'     # wyswietla do linii z wzorcem
sed '/wzorzec/d'     # opuszcza linie z wzorcem (grep -v)
sed '/^$/d'          # opuszcza puste linie
sed -n '/wzorzec/p'  # wyswietla tylko linie z wz.(grep)
sed 's/marzec/March/' # podmiana stringow
sed 's/^/^I/'        # indentacja (taby na pocz.linii)
sed '/./s/^/^I/'     # ulepszona indentacja

sed -n '/begin{verbatim}/,/end{verbatim}/p'

sed -n '/^[ ^I]*$/,$s/^/ > /p' | tail +2l

cat $* \
 | sed -n -e '/^From: /s/^From: \([^<]*\).*/\1 wrote:/p' \
       -e '/^[  ]*$/,$s/^/ > /p' \
 | sed -e '1s/  wrote:/ wrote:/' -e '2s/^ > $//'
```


sed: przykład (1)

```
sierra-90> who
NAME       LINE         TIME          IDLE    PID  COMMENTS
witold   + vt04         Oct 21 04:46  2:45    238
witold   + ttyp0        Oct 21 04:46  2:43    292
witold   + ttyp1        Oct 21 04:46   .      291
witold   + ttyp2        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: przykład (2)

```
s/</\&lt;/g
s/>/\&gt;/g
/^[  ]*$/i\
<p>

/^[  ]*%/s/^[  ]*%\(.*\)$/<!-- \1 -->/
s/\\\\/<br>/g
s#\\verb\(.\)\([^\1]*\)\1#<tt>\2</tt>#g
s#\\underline{\([^}]*\)}#<u>\1</u>#g
s/\\section{\([^}]*\)}/<h1>\1<\/h1>/
s/\\subsection{\([^}]*\)}/<h2>\1<\/h2>/
s#\\begin{enumerate}#<ol>#g
s#\\begin{itemize}#<ul>#g
s#\\begin{description}#<dl>#g
s#\\end{enumerate}#</ol>#g
s#\\end{itemize}#</ul>#g
s#\\end{description}#</dl>#g
s#\\item#<li>#g
s#\\begin{verbatim}#<pre>#g
s#.end{verbatim}#</pre>#g
```


sed: przykład (3)

Poniższy przykładowy program sed'a skraca powtórzenia pustych linii do pojedynczej linii wykorzystując polecenie wczytywania kolejnych wierszy (N) i pętlę zrealizowaną przez skok do etykiety (b):

```
# pierwsza pusta linie jawnie wypuszczamy na wyjscie
/^$/p
:Empty
# nastepnie dodajemy kolejne puste linie usuwajac znaki NEWLINE
/^$/{ N;s/.//;b Empty
}
# mamy wczytana niepusta linie (jesli cokolwiek), wypuszczamy ja
{p;d;}
```

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

sed: operatory

a\	wyprowadź na wyjście kolejne linie do linii nie zakończonej \
b *etyk*	skok do etykiety
c\	zmień linie na następujący tekst, jak dla a
d	skasuj linię
i\	wyprowadź następujące linie przed innym wyjściem
l	wyświetl linię, z wizualizacją znaków specjalnych
p	wyświetl linię
q	zakończ
r *plik*	wczytaj plik, wypuść na wyjście
s/s_1/s_2/z	zastąp stary tekst s_1 nowym s_2; jeden raz gdy brak modyfikatora z, wszystkie gdy z=g, wyświetlaj podstawienia gdy z=p, zapisz na pliku gdy z=w *plik*
t *etyk*	skok do etykiety, gdy w bieżącej linii dokonane podstawienie
w *plik*	zapisz linię na pliku
y/s_1/s_2/	zamień każdy znak z s_1 na odpowiedni znak z s_2
=	wyświetl bieżący numer linii
!*polec*	wykonaj polecenie seda *polec* gdy bieżąca linia nie wybrana
: *etyk*	etykieta dla poleceń b i t
{...}	grupowanie poleceń

Wyrażenia regularne (3): 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 wzajemna niekompatybilność, jej wynikiem jest definicja dwóch języków wyrażeń regularnych: BRE (Basic Regular Expressions) i ERE (Extended Regular Expressions).

W największym skrócie, należy zapamiętać:

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

ERE (zgodne z egrepem) — 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 (4): powtórzenia

$r\backslash\{n,m\backslash\}$	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 (5): 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:]
```


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 moze zawierac tylko warunki
ls -l ~student | awk ' $5 > 100000 '

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



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

# uzycie zmiennych do zapamietania kontekstu miedzy wierszami
awk ' $1 != prev { print; prev = $1 } '

# uzycie zmiennych wbudowanych awka: NF i NR
awk ' NF > 5 { print "linia ",NR," za dluga" } '

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

# mechanizmy ustawiania wartosci poczatkowych zmiennych
awk ' BEGIN { var1=0 } { ... } ' var1=-1

# uzywanie pol wejsciowych 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 ilością 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 zrozumiały rejestr 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:

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

awk: funkcje wbudowane

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

sort	sortowanie wierszy pliku
tr	zamiana znaków
uniq	unikanie powtórzeń
join	bazodanowy operator join
tac	wyświetlaj zawartość plików od końca
rev	wyświetlaj pliki jak cat, ale odwracając kolejność znaków w wierszach
paste	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 w wyrażeniach regularnych pełni gwiazdka?
2. Jaką rolę w wyrażeniach regularnych pełnią nawiasy kwadratowe?
3. Jaką rolę pełnią wyrażenia regularne w programie „sed"?
4. Wymień co najmniej dwie istotne różnice pomiędzy programami „sed" i „awk".
5. Opisz działanie programu join.
6. Opisz działanie programu tr.