Uniksowe filtry i wyrażenia regularne

Witold Paluszyński
Katedra Cybernetyki i Robotyki
Politechnika Wrocławska
http://www.kcir.pwr.edu.pl/~witold/

Uniksowe filtry tekstowe

W systemie Unix zaimplementowano szereg ciekawych programów przetwarzających na różne sposoby ciąg danych zorganizowanych w wiersze, to znaczy rekordy zakończone znakiem NEWLINE (ASCII 10).

Ponieważ programy te wykonują często bardzo proste operacje, często używa się więcej niż jednego na raz, korzystając z mechanizmu potoków:

```
cat plik | prog1 | prog2 | prog3
```

Takie przetwarzanie ma charakter filtrowania strumienia danych, stąd programy wykorzystywane w ten sposób nazywa się **filtrami tekstowymi**.

Typową sytuacją wykorzystania potoków filtrów jest praca interakcyjna, gdzie użytkownik przeszukuje jakieś pliki lub dane dobierając właściwe filtry i ich parametry. Często na początku potoku pojawia się program `cat` czytający dane z pliku. Również typowo na końcu potoku bywa wywoływany program `more` lub `less` pozwalający przeglądać dane ekran po ekranie.

Warto przypomnieć, że **procesy w potoku pracują równolegle**, co daje istotne przyspieszenie w systemach wieloprocesorowych/wielordzeniowych.

Uniksowe filtry tekstowe — cat

Najprostszym z filtrów jest program `cat` czytający dane z wejścia i wypisujący je bez zmian na wyjściu. `Cat` posiada niewiele opcji, a pomimo to jest niezwykle przydatnym programem, często wykorzystywanym w potokach filtrów.

Na przykład, wywołanie cat bez żadnych argumentów na początku potoku pozwala przetwarzać dane pisane z klawiatury. Dzięki temu użytkownik może szybko sam wpisać dane testujące dla jakiejś skomplikowanej filtracji:

```
cat | prog1 | prog2 | prog3
```

Dodatkowe możliwości uzyskujemy dzięki wykorzystaniu w potokach list poleceń, np.:

```
... | ( cat; cat plik ) | ...
... |   cat - plik      | ...
```

W pierwszym przykładzie do danych przesyłanych potokiem drugie wywołanie `cat` dopisuje zawartość pliku, i całość łącznie przesyłana jest dalej. Drugi przykład ilustruje konwencję, na mocy której nazwa pliku w postaci minusa nie jest traktowana jako nazwa pliku dyskowego, tylko oznacza strumień `stdin`.

Uniksowe filtry tekstowe — tee

Program `tee` (ang. trójnik – odwołuje się do analogii hydraulicznej) przekazuje dane z wejścia na wyjście bez zmian, ale dodatkowo zapisuje cały strumień danych na pliku (z opcją `-a` dopisuje). Dzięki temu można łatwo zarejestrować postać pośrednią danych przetwarzanych przez jakiś potok filtracji.

Przykładowym zastosowaniem jest debugowanie skryptów filtrujących:

```
... | filtr1 | tee dbout_f1 | filtr2 | tee dbout_f2 | ...
```

Często przydatne jest dodanie „zdalnego sterowania" tym procesem za pomocą zmiennej eksportowanej do wywoływanego skryptu:

```
... |\
filtr1 | if [ -z "$DEBUG" ]; then cat; else tee dbout_f1; fi |\
filtr2 | if [ -z "$DEBUG" ]; then cat; else tee dbout_f2; fi |\
...
```

Uniksowe filtry tekstowe — cut

`Cut` wycina fragment wiersza. Można określić wycinanie konkretnych znaków, lub pól (słów) wiersza. Niestety, przydatność `cut` jest znacznie ograniczona przez trudność w zdefiniowaniu separatora pól. W odróżnieniu od innych filtrów, separatorem może być tylko pojedynczy, sztywno określony znak, i domyślnie jest to tabulator.

```
# listing katalogu - wybierz pierwszy znak i nazwe pliku
ls -l /etc | cut -c1,51-

# listing katalogu - prawa dostepu i nazwa (PORAZKA)
ls -l /etc | cut -d' ' -f 1,14
```

Jak widać, w wielu przypadkach trudno wyciąć programem `cut` właściwy fragment wiersza, i to ogranicza jego przydatność do wierszy o ściśle zdefiniowanym formacie, np. plików CSV (*Comma Separated Values*).

Uniksowe filtry tekstowe — head i tail

`Head` i `tail` są przydatnymi i prostymi w użyciu filtrami. `Head` wyświetla na wyjściu początkowy fragment pliku, domyślnie pierwszych 10 wierszy. Analogicznie, `tail` wyświetla końcowy fragment pliku, domyślnie ostatnie 10 wierszy.

Przykład: wyświetl sekcję pliku od wiersza nr 1550 do 1750:

```
head -1750 /opt/csw/apache/logs/access_log | tail +1550
# lub alternatywnie (ktora wersja jest najlepsza?)
head -1750 /opt/csw/apache/logs/access_log | tail -201
tail +1550 /opt/csw/apache/logs/access_log | head -201
```

`Tail` ma jeszcze jedną przydatną opcję (`-f`). Oznacza ona, że po doczytaniu do końca pliku (i wyświetleniu na wyjściu zadanego fragmentu) należy czekać, i ponawiać próby czytania. Gdy na końcu pliku pojawią się dalsze dane, należy je wyświetlać bez żadnych ograniczeń. Pozwala to śledzić pliki, do których dopisują coś na końcu pracujące programy. Przykład:

```
tail -f /opt/csw/apache/logs/access_log
```

Uniksowe filtry tekstowe — sort

Sort sortuje wiersze z wejścia. Domyślne jest sortowanie alfabetyczne całego wiersza, jednak można opcjami wybrać szereg alternatyw, jak sortowanie numeryczne. Można zdefiniować **dowolne pole** (słowo) wiersza jako **klucz sortowania**. Można również zdefiniować klucz drugiego rzędu (sortowanie wierszy z równym kluczem podstawowym), i dla niego oddzielnie wybrać kryterium sortowania, i podobnie klucze dalszych rzędów.

Przykłady:

```
# sortowanie zawartosci katalogu po dlugosci plikow
ls -l | sort -n -k5

# sort.katalogu po pierwszej literze nazwy, potem dlugosci
ls -lt /etc |sort -k8.1,8.2 -k5n

# sortowanie zawartosci archiwum tar po czasie utw.pliku
tar tvf archiw.tar | sort -k7n,7 -k4M,4 -k5n,5 -k6d,6 -k8
```

Sort jest często używanym programem i warto nabrać wprawy w jego użyciu.

Uniksowe filtry tekstowe — uniq

`Uniq` służy do wykrywania i usuwania powtarzających się wierszy w ciągu wejściowym. Jego zastosowanie jest bardziej specjalistyczne, i często w połączeniu z innymi filtrami.

Przykłady

```
# lista nieortograficznych slow z plikow
cat *.tex | ispell -l -t -d polish | sort | uniq

# lista imion uzytkownikow systemu
getent passwd | cut -d: -f5 | cut -d' ' -f1 | sort | uniq

# ile plikow bylo tworzonych w poszczegolnych dniach
ls -l | awk '{print $6}' | sort | uniq -c
```

Jak widać w powyższych przykładach, `uniq` jest często używany łącznie z `sort`.

Uniksowe filtry tekstowe — diff

`Diff` nie jest typowym filtrem, ponieważ jego rolą nie jest filtrowanie danych w potoku. Jednak `diff` jest programem niezwykle przydatnym w wielu pracach. Porównuje on dwa pliki tekstowe, wiersz po wierszu, znajduje sekcje różniące te dwa pliki, i wyświetla je w postaci ciągu takich bloków różnic. Pozwala to na szybkie zorientowanie się czym różnią się dwa pliki, pod warunkiem, że stanowią one nieznacznie różniące się od siebie wersje tego samego dokumentu, programu, specyfikacji, czy innego typu pliku.

```
diff /etc/nsswitch.conf_old /etc/nsswitch.conf
11c11
< hosts:   files dns
---
> hosts:   files mdns4_minimal [NOTFOUND=return] dns mdns4
```

Powyższe pliki różnią się tylko wierszem numer 11. Znaki `<` i `>` symbolizują zamianę jakiej należałoby dokonać, aby zrównać pierwszy plik z drugim.

`Diff` ma szereg opcji ułatwiających znajdowanie różnic w różnych sytuacjach, oraz zmieniających sposób ich prezentacji na wyjściu. Często przydatne są opcje ignorowania odstępów (spacji, tabulatorów) w porównywaniu wierszy, np. `-w`

Uniksowe filtry tekstowe — join

Podobnie jak `diff`, `join` nie jest typowym filtrem, ponieważ pracuje na danych z dwóch różnych plików. Łączy on rekordy z dwóch plików w sposób podobny do bazodanowego operatora `join`. Rekordy z obu plików są do siebie dopasowywane według wartości określonego pola rekordów, stanowiącego klucz.

```
# 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 2.1,0,1.5 -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 klucza**.

Uniksowe filtry tekstowe — tr

Program `tr` realizuje transliterację, czyli podmianę jednych liter (znaków) innymi w stringach. Przykłady:

```
# zamiana malych liter na wielkie
echo Ala ma kota. | tr '[a-z]' '[A-Z]'

# zamiana wszystkich nie-liter na podkreslniki
echo 'ABcd1234' | tr -c '[:alpha:]' '[_*]'

# zamiana polskich liter na lacinskie
echo Zażółć gęślą jaźń |\
tr 'ąęłćńóśźżĄĘŁĆŃÓŚŹŻ' 'aelcnoszzAELCNOSZZ'

# konwersja polskich znaków ISO8859-2 na CP1250 \
tr '\261\352\346\263\361\363\266\274\277' \
   '\245\251\206\210\344\242\230\253\276'
```

Wyrażenia regularne (1): podstawowe

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:

$\epsilon_1\epsilon_2...\epsilon_n$	—	ciąg wyrażeń dopasowuje się do ciągu znaków jeśli kolejne wyrażenia dopasowują się do kolejnych podciągów znaków
$\epsilon*$	—	gwiazdka następująca za wyrażeniem regularnym ϵ oznacza wielokrotne (0 lub więcej razy) powtórzenie dopasowania do kolejnych podciągów ciągu znaków; każdy podciąg jest oddzielnie dopasowywany do wyrażenia ϵ

„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]
```

Wyrażenia regularne (3): język grepa i egrepa

Poza podanymi wyżej podstawowymi konstrukcjami wyrażeń regularnych, kilka dalszych konstrukcji również istnieje we wszystkich implementacjach. Jednak z pewnych względów historycznych, znalazły się one w dwóch oddzielnych podzbiorach, które będziemy tu nazywać, ze względu na ich implementacje w dwóch podobnych programach, językami wyrażeń regularnych `grep`a i `egrep`a.

Język wyrażeń regularnych `grep`a — wyrażenia zapamiętane:

$\backslash(\epsilon\backslash)$	—	dopasowanie jak do wyrażenia ϵ, z zapamiętaniem dopasowanego ciągu znaków; można się do niego odwołać konstrukcją `\1` w dalszej części wyrażenia

Język wyrażeń `egrep`a — alternatywy, nawiasy, i niezerowe powtórzenia:

$\epsilon_1\|\epsilon_2$	—	alternatywa, dopasowanie do wyrażenia ϵ_1 lub do wyrażenia ϵ_2
(ϵ)	—	dopasowanie jak dla wyrażenia ϵ
$\epsilon?$	—	dopasowanie jak dla wyrażenia ϵ, lub pasuje do ciągu pustego
$\epsilon+$	—	oznacza powtórzenie podobnie jak `*`, ale co najmniej jednokrotne dopasowanie musi wystąpić

Tak jak można się tego spodziewać, dodatkowe znaki interpretowane specjalnie w wyrażeniach `egrep`a są normalnymi znakami w języku `grep`a, i na odwrót.

Dopasowanie wzorców: expr

Program `expr` posiada operator `:` dopasowania wyrażeń regularnych. Traktuje on drugi argument jako wyrażenie regularne i dopasowuje go do pierwszego argumentu. `Expr` sygnalizuje statusem sukces dopasowania, a także wyświetla na wyjściu liczbę dopasowanych znaków stringa danych (w pewnych przypadkach wyświetla dopasowany podstring). UWAGA: **dopasowanie musi obejmować początkowy fragment stringa danych** (lub cały string).

```
fraza="A mnie jest szkoda lata."
expr "$fraza" : "A mnie"      # sukces       stdout: 6
expr "$fraza" : "szkoda"      # porazka      stdout: 0
expr "$fraza" : "."           # sukces       stdout: 1
expr "$fraza" : ".*"          # sukces       stdout: 23
expr "$fraza" : "^"           # porazka(???) stdout: 0
expr "$fraza" : ".*\(lata*\)" # sukces       stdout: lata
expr "$fraza" : ".*\(.*\)"    # porazka(???) stdout: (nic)
```

Jak widać, w przypadkach dopasowania pustego stringa `expr` sygnalizuje brak dopasowania. Szczególnie ostatni przypadek, gdzie oba podwyrażenia `.*` konsumują zerowe podstringi, wydaje się mylący.

Wyszukiwanie wzorców: grep i egrep

Grep znajduje dopasowanie podanego wyrażenia regularnego we wszystkich wierszach strumienia wejściowego, lub zadanych plikach. Spróbuj rozszyfrować znaczenie poniższych przykładów:

```
grep money *
cat * | grep money
grep -n Count *.[ch]
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
grep '^From' $MAIL | grep -v 'From szef'
cat text | grep ' \([-A-Za-z][-A-Za-z]*\) [ ]*\1 '
egrep 'socket|pipe|msgget|semget|shmget' *.[ch]
```

Wyrażenia regularne (4): grep i egrep — zestawienie

Poniższe wyrażenia przedstawione są 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 \$(\epsilon\)$ (tylko `grep`)
$\epsilon*$	zero lub więcej powtórzeń wyrażenia ϵ
$\epsilon+$	jedno lub więcej powtórzeń wyrażenia ϵ (tylko `egrep`)
$\epsilon?$	zero lub jedno wystąpienie wyrażenia ϵ (tylko `egrep`)
$\epsilon_1\epsilon_2$	ϵ_1 i następujące po nim ϵ_2
$\epsilon_1 \mid \epsilon_2$	ϵ_1 lub ϵ_2 (tylko `egrep`)
\$(\epsilon\)$	zapamiętane wyrażenie regularne ϵ (tylko `grep`)
(ϵ)	wyrażenie regularne ϵ (tylko `egrep`)

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 `sed`a 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             # 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 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 (1) komedia pomyłek

```
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: 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 (2) — konwersja Latexa do HTMLa

```
# znaki specjalne HTML'a
s/\\&/\\\&amp;/g       ;       s/</\&lt;/g       ;       s/>/\&gt;/g

# puste wiersze i komentarze
/^[     ]*$/i\
<p>

/^[     ]*%/s/^[        ]*%\(.*\)$/<!-- \1 -->/

# string cytowany \verb to prawdziwy problem
s#\\verb\(.\)\([^\1]*\)\1#<tt>\2</tt>#g

# jednoznakowe roznosci
s/\\\\/<br\/>/g         ;       s/\\\([#_\$\\]\)/\1/g

# te znaczniki maja swoje odpowiedniki
s#\\underline{\([^}]*\)}#<u>\1</u>#g
s#\\section{\([^}]*\)}#<h1>\1</h1>#
s#\\subsection{\([^}]*\)}#<h2>\1</h2>#
s#\\begin{enumerate}#<ol>#          ;    s#\\end{enumerate}#</ol>#
s#\\begin{itemize}#<ul>#            ;    s#\\end{itemize}#</ul>#
s#\\begin{description}#<dl>#        ;    s#\\end{description}#</dl>#
s#\\item#<li>#
```

Sed: przykłady zaawansowane

Poniższy przykładowy skrypt seda skraca 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 wyjscie
/^$/p
:Empty
# dodajemy kolejne puste wiersze usuwajac znaki NEWLINE
/^$/{ 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

`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 (5): POSIX — BRE i ERE

Specyfikacja POSIX porządkuje i rozszerza oryginalną koncepcję wyrażeń regularnych Uniksa. 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 `grep`a i `egrep`a, 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, warto zapamiętać:

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

ERE (zgodne z `egrep`em) — wyrażenia regularne z operatorem alternatywy `...|...`, wyrażenia w nawiasach `(...)`, wystąpienia opcjonalne `...?`, oraz powtórzenia co najmniej jeden raz `...+`.

Oprócz powyższych, języki BRE i ERE różnią jeszcze szeregiem bardziej subtelnych drobiazgów, które nie będą tu szczegółowo omawiane.

Wyrażenia regularne (6): POSIX — inne konstrukcje

Standard POSIX wprowadził dodatkowo **powtórzenia n-krotne**:

$\epsilon\backslash\{n,m\backslash\}$	powtórzenie: co najmniej n-razy, co najwyżej m-razy (`grep`)
$\epsilon\{n,m\}$	powtórzenie: co najmniej n-razy, co najwyżej m-razy (`egrep`)

Jednej z wartości n lub m można nie podać, co oznacza ograniczenie liczby powtórzeń tylko od dołu lub tylko od góry, o ile przecinek jest obecny. Podana jedna wartość, i brak przecinka, oznacza powtórzenie dopasowania ściśle określoną liczbę razy.

Inną, niezwykle przydatną konstrukcją, wprowadzoną w standardzie POSIX, są operatory `\<...\>` wymuszające **dopasowanie tylko na granicy słowa**.

Standard POSIX rozszerzył też operator `[]` dopasowujący jeden znak o **klasy znaków** za pomocą wyrażenia `[[:klasa:]]`, z następującymi klasami:

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

Wyrażenia regularne (7): przykłady wyrażeń 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 ERE powtórzenia 1 lub więcej razy (`+`) oraz operatora powtórzenia od 2 do 5 razy (`{2,5}`).

Wyrażenia regularne (8): GNU grep

Wersja GNU programu `grep` implementuje całą funkcjonalność `grep`a i `egrep`a. Co więcej, wprowadza rozszerzenia pozwalające łączyć operacje tradycyjnie dostępne tylko dla `grep`a jak i `egrep`a.

Awk: uniwersalny filtr programowalny

`Awk` jest filtrem działającym, podobnie jak `sed`, na kolejnych wierszach. Jednak zamiast prostych operatorów o jednoznakowych nazwach, `awk` ma konstrukcje programowe przypominające język C. Dwukrokowy algorytm działania `awk`a:

1. czyta wiersz z wejścia, dzieli na pola (słowa) dostępne jako: `$1`, `$2`, ... ,
2. wykonuje cały swój program składający się z szeregu par: warunek-akcja.

Uwagi:

- par warunek-akcja może być wiele i w każdej może brakować warunku (domyślnie: prawda) albo akcji (domyślnie: wyświetlenie wiersza na wyjściu),
- w programie można używać zmiennych, które zachowują wartości pomiędzy wywołaniami programu dla kolejnych wierszy,
- zmiennych nie trzeba deklarować ani inicjalizować; w pierwszym użyciu są one inicjalizowane wartością 0 lub pustym stringiem, zależnie od operacji.

```
ls -l ~student | awk ' $5 > 100000 '
awk ' {print $2, $1} ' nazwa_pliku
cat /etc/passwd | awk -F: '{print $4,$3}' | sort
```

```
# operator dopasowania stringa do wyrazenia regularnego
awk -F: ' $7 ~ /bash$/ { print $1,$7 }' /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 { printf "Wiersz %d ma %d slow.",NR,NF } '

# obliczanie dlugosci stringa
awk ' { wd+=NF; ch+=length($0)+1 } END { print NR,wd,ch } '

# warunki specjalne do inicjalizacji i finalizacji
awk ' BEGIN { x1=0 } { ... } END {print x1,x2 } ' x2=-1

# uzywanie pol wejsciowych jak zmiennych
awk ' $1 < 0 { $1 = 0 } $1 > 100 { $1 = 100 } { print $0 } '

# polaczenie z mechanizmami shella w wierszu wywolania
awk ' { s += $1 } END { print s } '
awk ' { s += $'$nr_pola' } END { print s } '
```

Awk: użycie tablic

Awk pozwala na użycie tablic, jednak trochę innych niż typowe tablice w językach programowania. Tablice są indeksowane stringami, i nie deklaruje się ich rozmiaru. Z tego powodu nazywa się je **tablicami asocjacyjnymi**.

```
# sumowanie dowolnej liczby pozycji wedlug nazwy
awk ' { sum[$1] += $2 } \
     END { for (name in sum) print name, sum[name] } '

# zliczanie czestosci wystepowania slow w tekscie
awk ' { for (i=1; i<=NF; i++) freq[$i]++ } \
     END { for (word in freq) print word, freq[word] } '
```

Można też używać dwóch lub więcej indeksów tablicy. Warto jednak wiedzieć, że awk używa ich łącznie, jako jednego indeksu składającego się z obu stringów, plus oddzielającego je przecinka.

Awk: uwagi o przenośności

Oryginalny uniksowy awk był dość ograniczonym programem, i wkrótce po jego powstaniu pojawiła się wersja rozszerzona. Niestety, nie mogło się to dokonać w sposób całkowicie przenośny i nowa wersja zaczęła być instalowana pod nazwą nawk równolegle ze starą, do której instalowano link o nazwie oawk. Jednak w duchu utrzymania kompatybilności z wcześniej napisanymi skryptami, które nie mają świadomości nowszych wersji, polecenie awk na wielu systemach uniksowych wywołuje bardzo ograniczonego oryginalnego awka.

Często dobrym sposobem jest **wywołanie awk jako nawk** — na wielu systemach istnieje taki program albo link. Jest to dobra forma przenośnego wywołania awk zapewniająca odcięcie się od wersji najstarszej.

Znacznie bardziej rozbudowany jest Gnu Awk, często instalowany równolegle jako gawk. Jego funkcjonalność i możliwości są daleko większe od obu uniksowych wersji, czyniąc z niego właściwie skryptowy język programowania.

Jednak istnieją jeszcze inne wersje awka, i na systemach linuksowych często instalowany jest program mawk, niestety różniący się drobnymi elementami. Z reguły główna robocza wersja awka na każdym systemie jest instalowana również pod nazwą nawk na potrzeby skryptów napisanych przenośnie zgodnie z powyższą zasadą.

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.

`tac`	wyświetlaj zawartość plików od końca
`rev`	wyświetlaj pliki odwracając kolejność znaków w wierszach
`paste`	łącz i wyświetlaj jako jeden wiersz kolejne wiersze z plików

Łączenie filtrów

Siła filtrów Uniksa leży w prostocie ich funkcjonalności, i łatwości łączenia w bardziej skomplikowane wyrażenia. Ilustracją tego może być poniższy przykład, który w pięciu operacjach znajduje 10 najczęściej występujących słów w dowolnym zbiorze tekstów:

```
cat * | tr -cs "[:alpha:]" "[\012*]" \
      | sort \
      | uniq -c \
      | sort -nr \
      | head
```