

Interpretery komend, skrypty, filtry

Witold Paluszyński

witoldp@pwr.wroc.pl

<http://sequoia.ict.pwr.wroc.pl/~witold/>

Copyright © 1995–2006 Witold Paluszyński

All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat interpreterów komend Unixa, programowania skryptów i wykorzystania typowych filtrów Unixa: 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.

Praca z interpreterem poleceń

Potoki (*pipeline*): równoległe uruchomienie poleceń z połączeniem ich wyjść i wejść

```
who  
who | wc -l  
who | tee save | wc -l
```

Listy: łączenie poleceń (ściślej: potoków) spójnikami: `;`, `&`, `||`, `&&`.

```
cc prog.c ; echo kompilacja zakonczona  
cc prog.c & echo kompilacja uruchomiona  
grep pieniadze msg.txt && echo sa pieniadze  
grep pieniadze msg.txt || echo nie ma pieniedzy
```

Priorytety spójników: `|` — najwyższy, `||`, `&&` — średni, `;`, `&` — najniższy

```
date; who | wc  
(date; who) | wc
```

Polecenie interpretera można wziąć w nawiasy, co zmienia priorytety jego interpretacji. Co więcej jednak, powoduje to wykonanie polecenia w wewnętrznym interpreterze, będącym podprocesem procesu głównego.

Praca z interpreterem poleceń (cd.)

Skierowanie wejścia/wyjścia >, >>, <, <<=...=

```
prog < plik_wejscia > plik_wyjscia
```

Dodatkowo C-shell:

```
progerror 15 >& do_pliku
```

Dodatkowo Bourne shell:

```
progerror 15 > plik_wyjscia 2> plik_bledow
```

```
progerror 15 > plik_razem 2>&1
```

```
progerror 15 2>&1 > plik_wyjscia
```

Znaki powodujące dopasowanie do nazwy plików * ? [a-zA-Z_]

```
wc *.c
```

```
echo obraz?.pgm # wynik: obraz1.pgm obraz2.pgm
```

```
ls -l *.[cho]
```

Dodatkowo C-shell {str1,str2,...} ~[user]

```
ls -l {prog,cwicz}*.c
```

```
xv ~witold/*.pgm
```

Tworzenie skryptów

Chcemy policzyć liczbę użytkowników włączonych do systemu:

```
who | wc -l
```

Możemy utworzyć własną komendę, która będzie to robiła, umieszczając powyższą linię poleceń w pliku, np. `policz` i wywołując go: `sh policz`

```
cat > policz
who | wc -l
^D
sh policz
```

Możemy również nadać plikowi `policz` atrybut `x`, i umieścić go w jednej z kartotek na ścieżce `PATH`. Wtedy można posługiwać się plikiem zawierającym polecenia, zwanym skryptem, jak programem binarnym:

```
chmod +x policz
./policz
mv policz ~witold/bin
rehash                                # hash jesli Bourne shell
policz
```

Argumenty wywołania skryptu

echo pierwszy argument:	\$1
echo drugi argument:	\$2
echo itd. ...	
echo argument zerowy, nazwa skryptu:	\$0
echo wektor argumentow, bez arg. zerowego:	\$*
echo numer procesu interpretera polecen:	\$\$

Dodatkowe mechanizmy dostępu

C-shell:

echo liczba argumentow bez arg. zerowego:	\$#argv
echo przedzial wartosci argumentow:	\$argv[2-]

Bourne shell:

echo liczba argumentow bez arg. zerowego:	\$#
echo zamiennik gdy brak argumentu:	\${1:-innypar}
date	# wynik: czw 11 mar 2004 06:45:23
set 'date'	
echo rok = \$4	# wynik: rok = 2004

Zmienne

Istnieją dwa rodzaje zmiennych: zmienne lokalne interpretera komend, i zmienne globalne, tzw. środowiskowe, dziedziczone przez podprocesy od ich procesów macierzystych. Zatem zmienne globalne są dostępne dla programów uruchamianych przez interpreter komend, w tym również rekurencyjnie uruchamianych interpreterów komend.

C-shell

```
set a = 5
echo a ma wartosc $a
setenv ZMGL 20
echo ZMGL ma wartosc $ZMGL
# program ma dostep do ZMGL
./program ...
unsetenv ZMGL
```

Bourne shell

```
a=5
echo a ma wartosc $a
export ZMGL
ZMGL=20
echo ZMGL ma wartosc $ZMGL
# program ma dostep do ZMGL
./program ...
# tylko chwilowy eksport
ZMGL=20 ./program ...
```

Niektóre programy systemowe używają zmiennych środowiskowych, które zmieniają sposób ich działania:

PATH	używany przez sam interpreter poleceń
TERM	używany przez programy ekranowe, które chcą skorzystać z operacji terminala użytkownika innych niż zwykłe przesuwanie
MAIL	ścieżka do pliku ze skrzynką pocztową użytkownika
PAGER	określenie programu, który należy wywołać w celu wyświetlania tekstu ekran po ekranie, np. <code>man</code>
EDITOR	określenie programu, który należy wywołać w celu edycji pliku zainicjowanej przez niektóre programu, np. <code>crontab</code>

Znaki specjalne: cytowanie

- `\` odbiera znaczenie specjalne następującego po nim znaku
- `'...'` zaniechanie interpretacji wszelkich znaków specjalnych
- `"..."` zaniechanie interpretacji znaków specjalnych z wyjątkiem `$`, `\` (tylko w Bourne shellu), i `'...'`

Ponadto, znak NEWLINE (`\n`, ASCII 10), traci znaczenie specjalne po `\`, ale jest dopuszczalny wewnątrz napisów cytowanych tylko w Bourne shellu.

W teorii to wydaje się proste, ale wyrażenia cytowane można składać, i czasem trzeba dobrze się zastanowić, aby zanalizować, albo skonstruować jakieś wyrażenie, i wtedy powyższe reguły trzeba stosować bardzo uważnie.

```
echo "'proste'" # to jest proste
echo '\ '       # to tez
echo "\""       # poprawne w Bourne shellu ale nie w C-shellu
echo '\ '       # poprawne choc niekompletne w Bourne shellu,
                  # to samo znaczenie, ale niepoprawne w C-shellu
```

Bourne shell: algorytm działania

1. czyta jedną komendę z wejścia
 2. dokonuje interpretacji komendy, np. podstawień wartości zmiennych
 3. traktując pierwsze słowo komendy jako jej nazwę (polecenie), a pozostałe słowa jako argumenty, wykonuje komendę w jeden z następujących sposobów:
 - (a) jako funkcję lub wbudowaną komendę interpretera
 - (b) jako program zawarty w pliku dyskowym, jeśli polecenie ma postać składniową nazwy pliku (względnej lub bezwzględnej), np. `/bin/echo` lub `../prog/prog1`
 - (c) jako program zawarty w pliku dyskowym, jeśli plik o podanej nazwie znajduje się w jednym z katalogów dyskowych zadanych zmienną `PATH`
- program zawarty w pliku dyskowym może być programem binarnym, lub tzw. skryptem (zestawem komend)

Bourne shell: interpretacja komendy

algorytm interpretacji komendy jest dużo bardziej złożony niż algorytm jej wykonania:

1. podstawienia komend zagnieżdżonych (' . . . ')
2. podstawienia parametrów pozycyjnych (argumentów wywołania \$1, \$2) i zmiennych (\$PATH, \$var)
3. podział linii komendy na argumenty (które mogą być puste, np. ' ')
4. zastosowanie skierowania strumieni wejścia i wyjścia
5. podstawienia nazw plików ze znakami specjalnymi *, ?, [. . .]
 - apostrofy (' . . . ') powodują brak jakiejkolwiek interpretacji znaków w nich zawartych, natomiast cudzysłowy (" . . . ") powodują jedynie interpretację zawartych w nich znaków \$, \, i ' . . . '

Bourne shell: znaki specjalne

\	\z powoduje wzięcie znaku 'z' dosłownie
#	jeśli na początku słowa to początek komentarza
*	dopasowuje się do dowolnego ciągu znaków w nazwie pliku
?	dopasowuje się do pojedynczego znaku w nazwie pliku
[abc...]	dopasowanie pojedyn.znaku, można podać przedział np. [a-z]
;	sekwencyjne wykonanie poleceń
&	równoległe (asynchroniczne) wykonanie poleceń
	potok, równoległe wykonanie poleceń z przekazyw.danych
&&	sekwencyjne ale warunkowe wykonanie poleceń
< >	skierowanie strumieni danych z i do pliku
(...)	wykonanie ... w wewnętrznym interpreterze poleceń
'...'	wykonanie poleceń w ..., otrzymane wyjście zastępuje '...'
'...'	wzięcie ... dosłownie
"..."	wzięcie ... dosłownie, jednak po interpretacji \$, '...', i \
zm=wart	przypisanie wartości zmiennej
\$zm	wzięcie wartości zmiennej
\${zm}	wzięcie wartości zmiennej, unika dwuznaczności w połączeniu z innym tekstem
\$1, ...	parametry pozycyjne skryptu (\$1 do \$9), nazwa skryptu (\$0)

Bourne shell: wyrażenia warunkowe

```
if test -r prog.dane
then
    prog < prog.dane
else
    prog
fi
```

```
if [ "'tty'" = "/dev/console" ] ; then echo Konsola ; fi
```

```
case $DISPLAY in
    ":0.0") host='uname -n':0.0;;
    ":0") host='uname -n':0.0;;
    *) host=$DISPLAY;;
esac
```

Bourne shell: polecenia pętli

```
for x in *.c
do
    # cmp wyswietla na wyjsciui informacje o roznicach
    if [ "`cmp $x~ $x`" ]
    then
        echo Plik $x zmienil sie, wysylamy koledze...
        mailx -s "nowa wersja pliku $x" kolega < $x
    fi
done
```

```
while true
do
    /bin/echo -n "Podaj nazwe skryptu do wykonania: "
    sh 'line'
done
```

Testowanie warunków — test

Warunki logiczne w instrukcjach `if` i `while` testowane są przez sprawdzenie statusu (kodu zakończenia, ang. *status*, *exit code*) programów. Statusem programu jest liczba całkowita zwracana przez procedurę `main()` instrukcją `return expr;` lub funkcją biblioteczną `exit(expr);`. Status można zwrócić ze skryptu wbudowanym poleceniem `exit` lub `return`.

Wartość statusu 0 oznacza prawdę, a każda inna wartość fałsz.

Wartość statusu jest normalnie niewidzialna przy interakcyjnym wykonywaniu poleceń, ale jest przechwytywana przez interpreter poleceń, i dla poleceń wykonywanych synchronicznie, bezpośrednio po wykonaniu polecenia jest dostępna w zmiennej `$?` (w C-shellu `$status`).

Niektóre programy są specjalnie przygotowane do sprawdzania warunków, np. mają opcję powodującą brak wyświetlania czegokolwiek na wyjściu, a tylko zwracanie wyniku przez status (`grep`, `cmp`, `mail`, i inne). „Etatowym” narzędziem do sprawdzania warunków jest `test`, obsługujący bogaty język specyfikacji warunków.

Przykłady:

```
test -r filespec      # czy plik istnieje i jest dostępny do odczytu
test -d filespec      # czy plik istnieje i jest katalogiem

test -z string        # czy dany string ma dlugosc zero
test string           # czy dany string jest pusty
test str1 = str2      # czy dane dwa stringi sa identyczne

test n1 -eq n2        # czy dwie liczby calkowite rowne
test 1.1 -eq 1        # daje 0 (prawda) - przez zaokraglenie
test 1+1 -eq 2        # daje 1 (falsz) - nie oblicza wyrazen
test n1 -ge n2        # czy n1 >= n2, analogicznie -gt -le -lt
```

Jak widać, Bourne shell wykonuje pewne operacje liczbowe, np. zaokrąglanie, ale nie wykonuje obliczeń arytmetycznych.

Bourne shell: przykład — skanowanie argumentów

```
while [ $# -ge 1 ]
do
    case $1 in
        -a)    AFLAG=1; shift
                ;;
        -b)    BFLAG=1; shift
                ;;
        -o)    OARG=$2; shift 2
                ;;
        -*)    echo usage: $0 [-a] [-b] [-o oarg] args ...
                exit 1
                ;;
        *)    break
                ;;
    esac
done

echo options: "-a=$AFLAG" "-b=$BFLAG" "-o=$OARG" "args=$@"
```

Bourne shell: przykład — przesyłanie danych pocztą

```
# tarmail: send encoded files by mail

if test $# -lt 3; then
    echo "Usage: tarmail address \"subject\" directory-or-file(s)"
    exit
else
    address=$1
    echo "address = $address"
    shift
    subject="$1"
    echo "subject = $subject"
    shift
    echo files = $*
    tar cvf - $* | compress | btoa | mail -s "$subject" $address
fi
```

Obliczenia arytmetyczne — expr

```
oblicz=5
echo $oblicz          --> 5
oblicz=$((oblicz+1))
echo $oblicz          --> 5+1
```

```
expr 3 + 4            --> 7
expr 2 - 5            --> -3
expr 10 / 3           --> 3
expr 3 \* 8           --> 24
expr 3 + 4 \* 5       --> 23
```

```
expr 3+4              --> 3+4
```

```
oblicz=5
oblicz=$((expr $oblicz + 8))
echo $oblicz          --> 13
```

Możliwości `expr`'a w zakresie obliczeń arytmetycznych nie sięgają głęboko. Kiedy potrzebne są obliczenia zmiennoprzecinkowe, można/należy zastosować program kalkulatora stosowego `dc`, używającego odwrotnej notacji polskiej, a żeby skorzystać z kilku podstawowych funkcji matematycznych niezbędny jest program `bc`, który jest preprocesorem `dc`.

```
oblicz='echo 4k 10 3 / p | dc'
echo $oblicz                                --> 3.3333
```

```
oblicz='echo 4k 2 v p | dc'
echo $oblicz                                --> 1.4142
```

```
# logarytm
oblicz='echo "l(2.73)" | bc -l -c | dc'
echo $oblicz                                --> 1.00430160919686836451
```

```
# sinus
oblicz='echo "s(3.14)" | bc -l -c | dc'
echo $oblicz                                --> .00159265291648695253
```

Bourne shell: skierowanie here document

```
echo Creating /etc/resolv.conf ...
nameserver=$1
cat > /etc/resolv.conf << END-OF-RESOLV-CONF
domain stud.ii
search stud.ii prac.ii ii.uni.wroc.pl
nameserver $nameserver
END-OF-RESOLV-CONF
echo Done /etc/resolv.conf
```

```
echo Fixing /etc/hosts ...
ed /etc/hosts << \END-OF-EDIT-STRING
1,$s/stud.ii/ii.uni.wroc.pl/
1,$s/prac.ii/ii.uni.wroc.pl/
w
q
END-OF-EDIT-STRING
echo Done /etc/hosts
```

Bourne shell: przykład — samorozpakowujące archiwum

```
# bundle: pakujemy wiele plikow do skryptu

echo '# w celu rozpakowania przepusc przez /bin/sh'
for i in $*
do
    echo "echo tworzymy plik $i 1>&2"
    echo "cat >$i <<'Koniec danych pliku $i'"
    cat $i
    echo "Koniec danych pliku $i"
done
```

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 słowo  
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"
jakȃocena='expr "$fraza" : '.*ocena.*: \(.*\)$' '
```

Możemy również sprawdzić tylko czy nastąpiło dopasowanie testując status.

```
if expr "$imie" : '.*[aA]$\ ' > /dev/null
then
    echo Otrzymała Pani ocene: $jakȃocena
else
    echo Otrzymałes ocene: $jakȃocena
fi
```


Wyrażenia regularne — wstęp

Jednoznakowe wyrażenia regularne:

- \cdot — 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 podciagu 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

Bourne shell: uruchamianie skryptów

- opcjonalne argumenty wywołania (flagi) np. `-v`, które nie liczą się jako argumenty pozycyjne `$1`, `$2`, ..., można je podać w wywołaniu skryptu, albo ustawić w czasie pracy shella poleceniem `set`, np. `set -f`
 - v powoduje wyświetlenie wczytanych linii polecenia
 - x powoduje wyświetlenie poleceń przed wykonaniem, po interpretacji
 - n powoduje tylko wyświetlanie poleceń do wykonania, bez wykonania
 - e powoduje zatrzymanie interpretera z błędem jeśli jakiegokolwiek polecenie zwróci niezerowy status
 - u powoduje wygenerowanie błędu przy próbie użycia niepodstawionej zmiennej
 - f powoduje wyłączenie dopasowania nazw plików do znaków specjalnych takich jak `*`, a `set +f` ponownie włącza ten mechanizm
- polecenie puste : przydatne do sprawdzania wyniku interpretacji
- konstrukcja `${zm?}` generuje błąd (status 1) gdy zmienna `zm` jest niepodstawiona (normalnie daje pusty string)
- obsługa sygnałów: polecenie `trap` deklaruje akcję do wykonania po otrzymaniu przez shell sygnału(ów)

Bourne shell: skierowania

Uruchomiony proces może posługiwać się wieloma strumieniami danych (dla których system tworzy struktury zwane deskryptorami plików).

Deskryptory te mają nadawane kolejne numery, począwszy od 0, i po otwarciu pliku proces wykonuje na nich operacje wejścia/wyjścia odwołuje się do nich przez numery deskryptorów.

Normalnie procesy są uruchamiane z trzema otwartymi deskryptorami: 0 (wejście), oraz 1, i 2 (wyjście) i są one skojarzone przez system z terminalem użytkownika (wejścia z klawiaturą a wyjścia z monitorem). Strumień danych 0 i 1 stanowią tzw. standardowe wejście i wyjście, a strumień 2 stanowi standardowe wyjście błędów.

Przy uruchamianiu procesu system może skojarzyć dowolny strumień danych z plikiem dyskowym, co powoduje otwarcie tego pliku i wykonywanie operacji wejścia/wyjścia na tym pliku bez konieczności jawnego otwierania konkretnego pliku w programie.

`<n >n` — skierowanie wejścia (stdin) lub wyjścia (stdout) z/do pliku otwartego dla strumienia danych `n`

`m<n m>n` — skierowanie strumienia wejściowego lub wyjściowego `m` z/do pliku otwartego dla strumienia danych `n`

Bourne shell: funkcje

```
ask_yes_no() {
    answer=X
    while true
    do
        echo The question: $1
        echo Answer yes or no:
        read answer
        case $answer in
            yes|Yes|YES) return 0;;
            no|No|NO)    return 1;;
        esac
        echo Wrong answer.
        echo ""
    done
}
```

```
# przykład wywołania:
if ask_yes_no "Czy kasowac\
               pliki tymczasowe?"
then
    rm -f *.o a.out
fi
```

Można również przechwycić dane wyświetlane przez funkcję na wyjściu, jednak wtedy np. konwersacja z użytkownikiem musiałaby odbywać się przez stderr.

Bourne shell: sterowanie procesami

Przy interakcyjnej pracy z interpreterem poleceń możliwa jest manipulacja podprocesami uruchomionymi przez dany interpreter. Można zobaczyć listę podprocesów, i każdy z nich zatrzymać, a także wznowić, zarówno jako zadanie w tle jak i w pierwszym planie. Podprocesy identyfikowane są kolejnymi liczbami, a w odwołaniu do nich piszemy numer podprocesu poprzedzony znakiem procenta.

```
jobs
```

```
fg %n
```

```
bg %n
```

```
stop %n
```

Bourne shell: obsługa przerwań

W Unixie istnieje mechanizm przerwań programowych zwanych sygnałami. Istnieje około 20-30 zdefiniowanych sygnałów, z których większość ma przypisane funkcje związane z funkcjonowaniem sprzętu bądź zachowaniem programu. Jądro systemu „doręcza” sygnały procesom, i otrzymanie sygnału oznacza zwykle, że proces powinien natychmiast zakończyć się. Mówi się, że domyślną reakcją na otrzymanie sygnału jest śmierć. Proces może jednak zmienić tę reakcję:

```
trap "rm tmp_file; echo trapped; exit" 1 2 3 4 5 6 7 8 10 12 13 14 15
```

UWAGA: jakkolwiek obsługa sygnałów może być prostą metodą interakcji z użytkownikiem, niepotrzebne przechwytywanie sygnałów prowadzi często do tworzenia „nieśmiertelnych” procesów, które gdy wygenerują jakiś błąd, mogą być bardzo kłopotliwe. Ogólnie łatwość uśmiercania procesów jest zaletą, podobnie jak akceptowanie sygnałów przysyłanych przez system.

Bourne shell: magia #!

Większość współczesnych unixowych interpreterów poleceń stosuje konwencję polegającą na specjalnym traktowaniu skryptów, których pierwsze dwa bajty to `#!` (fachowa wymowa anglojęzyczna: *sha-bang*). Do wykonania takich skryptów nasz interpreter poleceń wywołuje — jako interpreter do wykonania skryptu — program określony w pierwszym wierszu skryptu, zaczynającym się właśnie od `#!`. Dalszy ciąg wiersza traktowany jest jako nazwa programu do wywołania. Jednak musi to być nazwa programu w postaci pełnej bezwzględnej ścieżki pliku, ponieważ przy wywoływaniu tego programu nie stosuje się normalnego algorytmu wykonania (ani interpretacji) polecenia.

```
#!/usr/bin/perl
```

```
use Config qw(myconfig);  
print myconfig();
```

UWAGA: pisząc skrypt pod określony interpreter najczęściej nie mamy pewności czy taki interpreter jest dostępny w każdym systemie, a nawet jeśli jest, to jaka dokładnie jest jego ścieżka pliku. Jedynym interpreterem, na który zawsze można liczyć jest Bourne shell w pliku `/bin/sh`.

C-shell: modyfikatory ":"

```
lpipascal program.p      #wywołanie kompilatora Pascala
ldpascal program.o       #wywołanie linkera
```

```
#!/bin/csh -f
```

```
echo "lpipascal -longint -o $1:r.o $1"
lpipascal -longint -o $1:r.o $1
if ($status != 0) then
    echo 'Pascal compilation failed, linking not done\!\!'
    exit $status
endif
if ($1 == $1:r) then
    echo "ldpascal -strip -o a.out $1:r.o"
    ldpascal -strip -o a.out $1:r.o
    exit 0
endif
echo "ldpascal -strip -o $1:r $1:r.o"
ldpascal -strip -o $1:r $1:r.o
```

modyfikatory wartości wyrażeń: **:h :r :e :t :s/old/new/ :q :x**

bash: operatory arytmetyczne

Interpretery poleceń zgodne ze standardem POSIX, takie jak bash i ksh realizują szereg dodatkowych operacji, które ułatwiają pisanie skryptów. Należą do nich np. operatory arytmetyczne:

```
echo 2+2= $((2+2))
```

Jednak w wyrażeniach arytmetycznych zapisywanych w podwójnych nawiasach trzeba uważać na operatory porównania, ponieważ zwracają one wartości zgodne z konwencją języków takich jak C, czyli prawda jest reprezentowana przez 1 a fałsz przez 0, odwrotnie niż w konwencji wartości logicznych interpretowanych przez status polecenia.

```
echo '3>2?' $((3>2))
```

Unixowe interpretery komend — historia

- oryginalny interpreter komend: „Bourne shell” (`/bin/sh`)
 - zmodyfikowany interpreter komend do pracy interaktywnej: „C-shell” (`/bin/csh`) zawiera wiele ulepszeń i zmienioną składnię złożonych poleceń programowych
 - ulepszona wersja C-shell (`/bin/tcsh`), zawiera edycję ekranową komend
 - ulepszone wersje Bourne shella: Korn shell (`/bin/ksh`), „Born Again Shell” (`/bin/bash`), ten ostatni zawiera mechanizm historii i kontrolę podprocesów w stylu C-shell, zachowując jednak składnię poleceń programowych Bourne shella, oraz dodając ekranową edycję komend
- obserwacja 1: współcześnie używane interpretery (`tcsh` i `bash`) różnią się minimalnie, głównie składnią poleceń programowych (warunkowych i pętli)
- obserwacja 2: z punktu widzenia kompatybilności i przenośności istnieje tylko jeden interpreter komend — oryginalny Bourne shell

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 grep i egrep

w kolejności malejącego priorytetu:

z	dowolny znak niespecjalny pasuje do siebie samego
$\backslash z$	kasuje specjalne znaczenie znaku z
$^$	początek linii
$\$$	koniec linii
\cdot	dowolny pojedynczy znak
$[abc\dots]$	dowolny znak spośród podanych, można podawać przedziały znaków np. a-zA-Z
$[^abc\dots]$	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)
	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 +21
```

```
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 COMMENTS
```

```
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 wyjście
/^$/p
:Empty
# następnie dodajemy kolejne puste linie usuwając znaki NEWLINE
/^$/ { N;s/././;b Empty
}
# mamy wczytana niepusta linie (jesli cokolwiek), wypuszczamy ją
{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 <i>etyk</i>	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 <i>plik</i>	wczytaj plik, wypuść na wyjście
s/ <i>s</i> ₁ / <i>s</i> ₂ / <i>z</i>	zastąp stary tekst <i>s</i> ₁ nowym <i>s</i> ₂ ; jeden raz gdy brak modyfikatora <i>z</i> , wszystkie gdy <i>z</i> =g, wyświetlaj podstawienia gdy <i>z</i> =p, zapisz na pliku gdy <i>z</i> =w <i>plik</i>
t <i>etyk</i>	skok do etykiety, gdy w bieżącej linii dokonane podstawienie
w <i>plik</i>	zapisz linię na pliku
y/ <i>s</i> ₁ / <i>s</i> ₂ /	zamień każdy znak z <i>s</i> ₁ na odpowiedni znak z <i>s</i> ₂
=	wyświetl bieżący numer linii
! <i>polec</i>	wykonaj polecenie <i>seda polec</i> gdy bieżąca linia nie wybrana
: <i>etyk</i>	etykieta dla poleceń b i t
{...}	grupowanie poleceń

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 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 wejscowego, który 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
<i>nic</i>	konkatenacja napisów
+ -	plus, minus
* / %	mnożenie, dzielenie, reszta
++ --	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 Unixa

sort - sortowanie wierszy

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

uniq - usuwanie powtorzen wierszy

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

tr - zamiana znakow

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

polaczenie roznych filtrow

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

Operator join

```
# 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.