UNIX - język poleceń (Skrypty1)

1. Interpreter zleceń

Interpreter zleceń służy do komunikacji użytkownika z systemem. Interpreter taki najczęściej nazywany jest shell-em (pośrednictwem użytkownika). Większość systemów Unix oferuje kilka standardowych pośrednictw:

Bourne Shell

Bourne Shell (**sh**) został napisany przez Stephena Bourne w Bell Laboratories i jest najstarszym shell-em Unixa. Bourne Shell jest również domyślnym shell-em w większości systemów Unix. Cecha ta spowodowała, że **sh** stał się standardem de facto. Bourne shell nie zawiera jednak żadnych dodatkowych udogodnień, ani jeżeli chodzi o edycję linii zlecenia, ani dodatkowych konstrukcji języka skryptów.

C Shell

C Shell (**csh**) został stworzony przez Billa Joya w University of California w Berkeley. Nazwa shell-a wzięła się z podobieństwa składni do języka do języka C. Shell ten zawiera wiele udogodnień w stosunku do Bourne Shell-a takich jak rozbudowanie składni, historia zleceń itp.

Korn Shell

Korn Shell (**ksh**) został napisany przez Davida Korna w Bell Laboratories. Jest to rozszerzona wersja shell-a Bourne'a wyposażona dodatkowo o elementy edycji linii zleceń, historię zleceń itp.

BASH Shell

BASH Shell **(Bourne Again Shell)** został stworzony przez Briana Foxa i Cheta Rameya i udostępniony na zasadach licencji GNU. Jest on powłokąi w wielu systemach linuxowych wykorzystywaną jako domyślna. Jest w pełni kompatybilny z powłoką sh oraz zawiera wiele ciekawych rozwiązań przejętych od powłok Korn i C (ksh i csh).

2. Podstawowe zlecenia systemu UNIX:

`cat` - połączenie, skopiowanie i wydrukowanie zawartości zbioru
`cd` - zmiana bieżącego katalogu
`chgrp` - zmiana grupy zbioru
`chmod` - zmiana praw dostępu do zbioru
`chown` - zmiana właściciela zbioru
`chsh` - zmiana standardowego shell-a użytkownika
`cp` - kopiowanie zawartości zbioru
`date` - wydrukowanie lub ustawienie daty i czasu
`echo` - wydrukowanie tekstu na terminalu
`head` - drukuje początek zbioru
`lnk` - łączenie zbiorów
`ls` - listowanie zawartości kartoteki
`man` - system pomocy systemu UNIX (`man nazwa_polecenia`)
`mkdir` - tworzenie nowego katalogu
`more` - "stronicowanie" wydruków
`mv` - przeniesienie zawartości zbioru
`pg` - "stronicowanie" wydruków
`ps` - wydrukowanie informacji o procesach w systemie
`pwd` - wydrukowanie bieżącej kartoteki
`rm` - usuwanie zbioru
`rmdir` - usuwanie katalogu
`tail` - drukuje końcówkę zbioru
`wc` - zliczanie znaków, słów, linii
`who` - wypisanie listy użytkowników zalogowanych w systemie
`sleep` - "usypia" proces

3. Elementy linii zlecenia shell-a

Najprostsza linia zlecenia ma postać: `zlecenie [opcje] [parametry]`
Przykładem takiego zlecenia może być: `ls -l *.c`. Zlecenie **ls** wydrukuje zawartość kartoteki w "długim" formacie (opcja -l).

Wydrukowane zostaną zbiory kończące się na .c (parametr *.c). Kilka zleceń może być łączone w jedną linię za pomocą znaku średnika ; np:

```
ls *.c ; who ; ps
```

W linii występują trzy zlecenia **ls**, **who** oraz **ps** oddzielone znakiem średnika. Tak połączone zlecenia będą wykonywane sekwencyjnie. Oznacza to, że najpierw wykona się zlecenia **ls**, po jego zakończeniu **who**, a na końcu **ps**. Programy mogą być też wykonywane niesekwencyjnie. W tym przypadku każdy program wykonuje się bez czekania na zakończenie procesu poprzedzającego. Aby wykonać zlecenia w tle należy oddzielać je znakiem & np:

```
ls *.c & who & ps
```

W tym przypadku zlecenie **who** rozpocznie się bez czekania na zakończenie procesu **ls**. Podobnie jest ze zleceniem **ps**.

4. Strumienie danych

Każdy program ma skojarzone co najmniej trzy strumienie standardowe: strumień wejścia (stdin), strumień wyjścia (stdout) oraz strumień błędu (stderr). Predefiniowanym standardowym strumieniem wejściowym jest klawiatura terminala, zaś predefiniowanymi strumieniami wyjścia i błędu jest ekran. Shell-e umożliwiają przekierowania strumieni w linii zlecenia. Przekierowania wejścia dokonuje się podobnie jak w DOS-ie przez znaki **<** lub **<<** i np:

`program < zbior` – program czyta dane ze zbioru o nazwie *zbior*.

`program << tekst` – program czyta dane z następnych linii zlecenia aż do napotkania linii zawierającej tekst zapisany po znaku << (w tym przypadku słowa *tekst*). Jeżeli po znaku << damy znak - to linie podawane na standardowe wejście będą pozbawiane wiodących znaków tabulacji.

Przekierunkowanie wyjścia ma postać:

program > *zbior* - wyniki są umieszczane w zbiorze. Jeżeli zbiór istniał, to jego poprzednia zawartość będzie zniszczona.

program >> *zbior* - wyniki są dopisywane na końcu zbioru. Jeżeli zbiór nie istniał, zostanie utworzony.

Czasami wykorzystwana jest także konstrukcja postaci:

program <&*cyfra* oraz *program* >&*cyfra*

W tym przypadku wejście lub wyjście jest skierowane do strumienia o podanym numerze. Standardowe wejście ma numer 0, standardowe wyjście 1, zaś strumień błędu 2 np:

`ls >&2` - przekierowuje strumień wyjścia do strumienia błędu. Istotnym jest, że przy przekierunkowywaniu strumieni można wskazać, który strumień mamy na uwadze, np:

`cat 2>err` - przekierunkowanie strumienia błędu do zbioru

Bardzo często wykorzystywanym mechanizmem jest przetwarzanie potokowe. Aby skierować strumień wyjściowy na wejście następnego procesu należy połączyć je znakiem | np:

`who | wc` standardowe wyjście zlecenia who jest podawane na wejście zlecenia ***wc***.

Bardzo istotnym elementem języka shell-a jest mechanizm wzorców służący do specyfikowania zbiorów, które mają być parametrami zlecenia. W skład wzorca mogą wchodzić następujące elementy:

***** - zastępuje dowolny (w tym pusty) ciąg znaków

? - zastępuje dokładnie jeden znak

[...] - zastępuje określone znaki

Przykładem wyrażenia może być np: `rm [a-s]i?abc*.[jm]`

Powyższe zlecenia skasuje wszystkie zbiory, które rozpoczynają się małą literą z zakresu od „a" do „s", następnie występuje litera „i" następnie dowolna jedna litera, następnie człon trójliterowy „abc", następnie dowolny łańcuch znaków, następnie kropka „." i na końcu jedna z liter „j" lub „m".

5. Środowisko

Podobnie jak w DOS-ie, shell w systemie UNIX posiada swoje własne środowisko. Składa się ono ze zmiennych środowiska lub zmiennych shell-a. Zmienne środowiska są dostępne z poziomu shell-a oraz są eksportowane do środowisk procesów potomnych. Zmienne shell-a są dostępne tylko z poziomu samego interpretera i nie są dziedziczone przez procesy potomne. Przypisanie wartości zmiennej shell-a uzyskuje się przez wykonanie zlecenia:

```
ZMIENNA=wartość
```

Aby zmienna shell-a stała się zmienną środowiska musi zostać wyeksportowana. Uzyskuje się to przez zlecenie:

`export ZMIENNA` - wyeksportowanie zmiennej o nazwie `ZMIENNA`

Nazwy zmiennych środowiska/shell-a nie muszą składać się wyłącznie z dużych liter. Podczas logowania do systemu są ustawiane następujące zmienne środowiska:

`HOME` - nazwa kartoteki bazowej użytkownika

`LOGNAME` - nazwa użytkownika

`PATH` - ścieżki poszukiwań programów

`SHELL` - ścieżka do shella

`TERM` - rodzaj terminala, i inne

Wartości zmiennych środowiska/shell-a są dostępne przez konstrukcję: `$ZMIENNA`. Aby przykładowo wyświetlić wartość zmiennej

o nazwie `ZMIENNA` należy wykonać zlecenie:
`echo $ZMIENNA`
Jeżeli zmienna nie występuje to pojawi się błąd! Zmienna może być usunięta ze środowiska za pomocą zlecenia **unset** np:
`unset ZMIENNA`
Jeżeli zmienna istnieje można wykorzystywać ją do zmiany wartości np:
`PATH=$HOME/bin:$PATH:.`
W tym przypadku do wartości zmiennej środowiskowej PATH dołożono „z przodu" wartość $HOME/bin, a na końcu dołączono katalog bieżący. Jeżeli nazwa zmiennej mogłaby połączyć się z resztą tekstu można użyć znaków {} np:
`echo poczatek${ZMIENNA}koniec.`
Do pracy ze zmiennymi można wykorzystać także inne konstrukcje podstawienia:
`${zmienna:-wartość}` - jeżeli zmienna istnieje i ma przypisaną niepustą wartość wynikiem konstrukcji jest ta wartość. W przeciwnym razie jest podstawiana „wartość".
`${zmienna:=wartość}` - jeżeli zmienna nie istnieje lub jest pusta przypisuje się jej „wartość", która jest także wynikiem podstawienia. Jeżeli zmienna ma przypisaną wartość wynikiem jest ta wartość a przypisanie nie następuje.
`${zmienna:?wartość}` - podobnie jak :-, lecz jeżeli zmienna nie występuje lub jej wartość jest pusta, to występuje błąd.
`${zmienna:+wartość}` - jeżeli zmienna występuje i ma przypisaną niepustą wartość to wynikiem jest „wartość". W przeciwnym wypadku podstawienie nie występuje
Poza zmiennymi środowiska/shell-a występują podobnie jak w DOS-ie, także zmienne parametryczne. Wartości parametrów wywołania skryptu podstawiane są pod kolejne parametry **$1**, **$2**, **$3**, ... **$9**
Parametr **$0** zawiera nazwę skryptu. Można również korzystać ze zmiennych **$*** oraz **$#**. Wartością zmiennej $* jest cała linia zlecenia poza nazwą skryptu ($0). Zmienna **$#** zawiera liczbę parametrów. Jeżeli parametrów jest więcej niż 9 to dostępne są one po wykonaniu operacji `shift` podobnie jak w COMMAND.COM z DOS-a. Występują również inne predefiniowane zmienne np:
$$ - numer procesu aktualnie wykonywanego shell-a
$! - numer ostatniego procesu uruchomionego w tle
$? - wartość kodu powrotu ostatnio zakończonego procesu.

6. Znaki specjalne

Interpreter zleceń ma również szereg innych mechanizmów umożliwiających wykonywanie podstawień. Na początku należy wspomnieć o znaku „\", który odwołuje specjalne znaczenie znaku po nim występującego np: `echo $ZMIENNA` wyświetli wartość zmiennej ZMIENNA zaś `echo \$ZMIENNA` wyświetli tekst `$ZMIENNA` ponieważ znaczenie specjalne znaku $ zostało odwołane. Jeżeli już mowa o znakach specjalnych w linii zleceń to wspomnijmu także o następujących:

Znaki ":

Znaki " " odwołują specjalne znaczenie wszystkich znaków poza \, $, ", ' oraz ` np: `echo "$ZMIENNA rowna sie \"TEKST\""` łączy cały tekst w jeden parametr, w którym jest podstawiana wartość zmiennej. Aby odwołać specjalne znaczenie znaku " zastosowano \.

Znaki ':

Znaki '' odwołują specjalne znaczenie wszystkich znaków poza '.
zlecenie poprzednie musi mieć więc postać:
`echo $ZMIENNA 'rowna sie "TEKST"'`

Znaki `:

Tekst zawarty między znakami ` ` jest traktowany jako zlecenie. Wynikiem takiego podstawienia jest tekst wypisywany przez zlecenie na standardowe wyjście. Przykładowo:
`echo "Uzytkownicy w systemie `who`. "`

7. Polecenie test

Polecenie to odgrywa szczególnie dużą rolę przy budowaniu warunków dla polecenia `if`. Możliwości są bardzo rozbudowane, niestety dostępne opcje nie są oczywiste. Możliwe są dwie postacie polecenia `test`:
`test warunek` na przykład: `test łańcuch1=łańcuch2`
oraz równoważna poprzedniej:
`[warunek ]` na przykład: `[ łańcuch1=łańcuch2 ]`

Podstawowe możliwości porównania to:
- `[ s1 = s2 ]` dwa łańcuchy znaków są identyczne
- `[ s1 != s2 ]` dwa łańcuchy znaków są różne
- `[ s1 ]` łańcuch jest niezerowy

- `[ s1 -eq s2 ]` s1 jest równe s2
- `[ s1 -ne s2 ]` s1 różne od s2
- `[ s1 -gt s2 ]` s1 jet większe niż s2
- `[ s1 -ge s2 ]` s1 jest większe lub równe s2
- `[ s1 -lt s2 ]` s1 jest mniejsze od s2
- `[ s1 -le s2 ]` s1 mniejsze lub równe s2
- `[ !warunek ]` negacja warunku
- `[ -s plik ]` plik nie jest pusty
- `[ -x plik ]` plik wykonywalny
- `[ -r plik ]` plik ma nadane prawo czytania
- `[ -w plik ]` plik ma nadane prawo do pisania

...i wiele innych (patrz man).

8. Znaki oddzielające zlecenia:

Konstrukcja &&:
`zlecenie1 && zlecenie2 && ... && zlecenieN`
Konstrukcja służy do warunkowego wykonywania zleceń. Zlecenie i+1 będzie wykonane jeżeli kody powrotu wszystkich zleceń 1..i będą wynosiły 0. Kodem powrotu całej konstrukcji jest kod powrotu ostatnio wykonanego zlecenia. Np:
```
test -d $1 && cd $1
```
Zlecenie cd wykona się jeżeli $1 jest katalogiem.

Konstrukcja ||:
`zlecenie1 || zlecenie2 || ... || zlecenieN`
Konstrukcja i+1 wykona się jeżeli kody powrotu zleceń 1..i zakończą się niepowodzeniem (!= 0). Kodem powrotu całej konstrukcji jest kod powrotu ostatnio wykonanego zlecenia. Np:
```
test -d $1 || echo $1 nie jest katalogiem
```
Zlecenie `echo` wykona się jeżeli $1 nie jest katalogiem. Zlecenia && i || mogą być łączone w jednym zleceniu

Konstrukcja ():
`(lista zleceń)`
Powoduje, że do jej wykonania uruchamiana jest dodatkowa kopia shell-a. Konstrukcję tę używa się najczęściej do wykorzystywania innych konstrukcji lub do wykonywania listy zleceń w tle. Np:
```
( cat zbior | sort | wc -w )&
```
Konstrukcja spowoduje wykonanie całości zlecenia w tle.

Konstrukcja {}
`{ lista zleceń }`
Służy do przekierunkowywania wyjść kilku zleceń. Np:
```
{ echo "Zbiory: "; ls $HOME ; echo " Koniec" } >zbior
```

9. Konstrukcje shella

Shelle posiadają bardzo rozbudowane konstrukcje językowe podobne do konstrukcji spotykanych w językach algolopodobnych.

Konstrukcja IF
Warunek jest prawdziwy, jeżeli kod powrotu listy zleceń wynosi 0
```
if lista_zleceń1
then lista_zleceń2
elif lista_zleceń3
then lista_zleceń4
...
else lista_zleceńN
fi
```
Przykład:
```
if test $# -eq 0
then
 echo "Uzycie: $0 parametr"
fi
```

Konstrukcja FOR
Pętla jest wykonywana dla każdego parametru z listy
```
for parametr [ in lista_parametrów ]
do lista_zleceń
done
```
Jeżeli lista parametrów nie występuje przyjmuje się domyślnie listę parametrów pozycyjnych $*
Przykład:
```
for i in /tmp $HOME/tmp
do
ls $i
done
```

Konstrukcja WHILE
Pętla wykonuje się jeżeli lista zleceń ma wartość prawdy (kod powrotu 0)
```
while lista_zleceń1
do lista_zleceń2
done
```
Przykład:
```
while [ -d $1 ]
do
 ls $1
 shift
done
```

Konstrukcja UNTIL
Pętla jest wykonywana jeżeli lista zleceń ma wartość fałszu (!=0)
```
until lista_zleceń1
do lista_zleceń2
done
```
Przykład:
```
until [ ! -d $1 ]
do
  ls $1
  shift
done
```

Konstrukcja CASE:
Konstrukcja jest rozszerzeniem if i ma postać:
```
case parametr in
wzorzec1 | wzorzec2 .. ) lista_zlecen1;;
wzorzecN | wzorzecN1 ..) lista_zlecen2;;
...
esac
```
Często ostatnim parametrem jest * co odpowieda default z języka C
Przykład:
```
case $1 in
-a | -b ) echo "opcja -a lub -b" ;;
-c      ) echo "opcja -c" ;;
-*      ) echo "inna opcja";;
esac
```

10. Procesy

W systemie UNIX proces można określić jako pojedynczo wykonywaną instancję programu, dysponująca własnymi zasobami, w szczególności własnym obszarem pamięci operacyjnej. Właścicielem procesu jest w zasadzie użytkownik tworzący proces. Należy wyróżnić tu pojęcie użytkownika efektywnego (effective user), którym może być albo użytkownik tworzący proces, albo właściciel pliku zawierającego program. Właściciel pliku staje się użytkownikiem efektywnym gdy plik ma ustawiony bit **s**, np. przez **chmod u+s plik**.
Jeden proces może uruchomić inny, zwany potomnym. Wszystkie procesy w systemie wywodzą się od procesu **init**. Każdy proces ma swój numer identyfikacyjny (PID) - dzięki któremu jest znany systemowi. Każdy proces „zna" też swój proces macierzysty (PPID). Procesy mogą należeć do wspólnej grupy procesowej identyfikowanej przez PGID.
Procesy mogą komunikować się między sobą, co pozwala także na ich synchronizację. Służy do tego szereg mechanizmów: semafory, wspólna pamięć, kolejki komunikatów, łącza, sygnały, potoki oraz oczekiwanie na zakończenie procesu potomnego.

Powłoka **sh** daje dostęp tylko do trzech ostatnich spośród wymienionych.
Utworzenie nowego procesu przez powłokę następuje w momencie wywołania każdego programu użytkowego poprzez podanie jego nazwy. Jeżeli polecenie uruchamiające program nie jest zakończone znakiem **&**, powłoka zawiesza swoje działanie w oczekiwaniu na zakończenie nowo utworzonego procesu - w przeciwnym wypadku kontynuuje swoje działanie asynchronicznie.
Istnieje przypadek w którym wykonanie programu nie powoduje utworzenia procesu, dzieje się tak gdy program wywołany jest poleceniem **exec**, które powoduje przekazanie sterowania nowemu programowi bez powrotu do poprzedniego.
Jeżeli wywołamy proces w tle możemy zidentyfikować jego PID przez zmienna **$!**. Wykonanie procesu macierzystego może być wstrzymane aż do zakończenia procesu potomnego instrukcją **wait pid**, gdzie **pid** określa proces potomny, jeżeli wykonamy samo **wait** - proces macierzysty wstrzyma działanie aż do wykrycia zakończenia wszystkich procesów potomnych. Proces potomny może zwracać kod powrotu do macierzystego - jest to najprostszy sposób przekazywania informacji między procesami (zmienna **$?** przechowuje kod ostatnio wykonanego procesu pierwszoplanowego).
Procesy nie tworzone synchronicznie i nie połączone relacją potomności mogą komunikować się za pomocą sygnałów. Sygnał w systemie UNIX jest asynchronicznym zdarzeniem mającym źródło, adresata i typ (jest 15 zdefiniowanych i 2 swobodne).
Przechwytywanie sygnałów na poziomie powłoki **sh** umożliwia polecenie **trap** któremu należy podać czynność do wykonania oraz numer sygnału. Oba parametry są opcjonalne, jeżeli polecenie będzie puste to sygnał będzie ignorowany, gdy pominiemy nr sygnału to polecenie zostanie wykonane po wywołaniu dowolnego sygnału. Podanie 0 daje możliwość zdefiniowania czynności, która ma zostać wykonana przed zakończeniem programu (nie można przechwycić sygnału 9). Wysłanie sygnału do konkretnego procesu lub grupy procesów umożliwia polecenie **kill**. Np.:

```
trap 'echo Signal 2 received !' 2
trap 'echo Signal 5 received !!' 5
while (true) do
:
done
```

Jeżeli teraz powyższy skrypt uruchomimy w tle, to wywołanie przez dowolny proces polecenia **kill -2 pid**, lub **kill -5 pid**, gdzie **pid** identyfikuje powyższy skrypt spowoduje wyświetlenie
odpowiednio: „Signal 2 received !" lub „Signal 5 received !!".

11. Poczta

Podstawowym programem w systemach UNIX do obsługi poczty jest program **mail**, może on nieco różnić się w poszczególnych systemach. Zmienna środowiskowa $MAIL określa ścieżkę do skrzynki pocztowej danego użytkownika (najczęściej: /usr/mail/użytkownik) tam wpisywane są wszystkie nadchodzące listy. Wydając polecenie **mail -e** możemy sprawdzić skrzynkę, zwrócona zostanie wartość TRUE jeżeli w skrzynce są jakieś listy. Samo polecenie **mail** odczytuje skrzynkę. Jeżeli do skrzynki wyślemy w kolejności listy: 1, 2, 3, to polecenie **mail** odczyta je w kolejności: 3, 2, 1. Kolejność można zmienić na odwrotną (FIFO) stosując **mail -r**.

12. Wyrażenia regularne

Wyrażenia regularne służą do „abstarkcyjnego" definiowania łańcuchów. Wykorzystywane jest to m. in. w filtrach programowych, omawianych w następnym podpunkcie.

Wyrażenie	Opis
`\t`	Znak tabulacji
`\n`	Nowa linia
`.` (znak kropki)	Dowolny znak
`|`	Operator lub, np.: a|b – zarówno a jak i b
`[ ]`	Lista znaków, można określać przedziałami np.: [ab] zarówno a jak i b; [0-9] – dowolna cyfra
`[^ ]`	wszystkie znaki z wykluczeniem listy znaków np.:[^0-9] – dowolny znak nie będący cyfrą
*	Znak po lewej stronie gwiazdki powinien wystąpić zero lub więcej razy np. *be** oznacza b, be, bee, itd
+	Znak po lewej stronie gwiazdki powinien wystąpić jeden lub więcej razy np. *be** oznacza be, bee, itd ale nie oznacza samego b
?	Znak po lewej stronie pytajnika powinien wystąpić zero lub jeden raz np. *be?* Oznacza b, be, itd ale nie oznacza bee
^	wyrażenie po prawej stronie znaku ^ musi wystąpić na początku wiersza
$	wyrażenie po lewej stronie znaku $ musi wystąpić na końcu wiersza

`()`	Grupowanie wyrażeń – ma realne zastosowanie przy podmienianiu łańcuchów
`\`	Znak „ucieczki" np. * aby uzyskać znak gwiazdki jako znak
`x{m, n}`	Znak x musi wystąpić pomiędzy m do n razy
`x{ ,n}`	Znak x musi wystąpić do n razy
`x{m, }`	Znak x musi wystąpić powyżej m razy

13. Filtry programowe

W literaturze dotyczącej systemu UNIX pojęcie *filtr* oznacza program, który przekształca dane ze strumienia wejściowego w pewien sposób i wypisuje wynik do strumienia wyjściowego. Wiele programów można uznać za filtry, do najpopularniejszych należą: **grep** i **awk**.

Grep przeszukuje zbiór wejściowy (domyślnie standardowe wejście) w poszukiwaniu linii zawierających wskazany wzorzec. Normalnie każda linia zawierająca wskazany wzorzec jest wyprowadzana na standardowe wyjście. Wywołanie ma następujące

```
grep [-E | -F] [-cbilnqsvx] [-e expression [-e expression] ... | -f file] [expression]
[file ...]
```

gdzie:
`-E` – oznacza, że użyte wzorce są rozszerzonymi wyrażeniami regularnymi (ERE),
`-F` – wzorzec może być zdefiniowany jako ciąg łańcuchów oddzielonych znakiem nowej linii lub podany przy pomocy opcji `-e`.
`-b` – każda wyprowadzana linia jest poprzedzona numerem bloku gdzie wzorzec został odnaleziony,
`-c` – wyprowadz tylko ilość linii zawierających wzorzec,
`-f file` – definiuje zbiór ze wzorcami,
`-i` – podczas porównywania ignorowana jest wielkość liter,
`-v` – wyprowadzane są wszystkie linie **oprócz** linii pasujących do wzorca.

Przykłady:

```
cat student|grep so42
```
- wypisze wszystkie linie zezbioru *student* zawierające ciąg *so42*.

```
grep -E'[Dd]ec|[Nn]ov' logfile
```
- wyprowadzi ze zbioru *logfile* wszystkie linie zawierające ciągi *Nov*, *nov*,*Dec* i *dec*. To samo uzyskamy stosując:
```
grep -i 'dec|nov'logfile
```

Awk jest zaawansowanym, programowalnym filtrem. Jego działanie polega na analizowaniu każdego podanego zbioru wejściowego (lub standardowe wejście) w poszukiwaniu linii zawierających wskazany wzorzec. Z każdym wzorcem może być skojarzona określona. Awk traktuje linie zbioru wejściowego jako rekordy, natomiast wyrazy w linii jako pola tego rekordu. Można przedefinować znaki separatora pól rekordu i uzyskać inny podział rekordu (linii) na pola. Do określonego pola rekordu (wyrazu) można się odwołać poprzez zmienne awk: $1, $2 ...$199; $0 oznacza całą analizowaną linię.

Wywołanie programu ma postać:

```
awk [-Ffs] [-v var=value] [prog  | -f file ...] [ file ...]
```

gdzie:
`-F fs` - definuje wyrażenie regularne (RE) określające separator pól w rekordzie, domyslnie znak spacji i tabulacji są traktowane jako separatory pól,
`-f File` - określa zbiór z programem dla awk,
`-v var=value` - pozwala na nadanie wartosci zmiennej (var), do której można odwoływać się w programie dla awk.
`prog` - program dla filtra **awk**

Program dla awk, ogólnie rzecz biorąc, składa się z następującego schematu:

```
wzorzec {akcja}
```

Jeżeli akcja nie jest zdefiniowana, domyślnie oznacza to wyprowadzenie linii, z kolei brak wzorca oznacza dopasowanie zawsze.

Program może zawierać więcej definicji wzorzec-akcja, oddzielamy je wówczas nową linią bądź średnikiem. Jeżeli określona linia pasuje do kilku wzorców, wykonywana jest akcja dla każdego z nich.

Akcja jest sekwencją poleceń. Polecenie może być jednym z następujących:

```
if (wyrażenie) polecenie [else polecenie]
while (wyrażenie) polecenie
for (wyrażenie, wyrażenie, wyrażenie) polecenie
for (var in array) polecenie
do polecenie while (wyrażenie)
break
continue
{[polecenie ...]}
wyrażenie                    #przeważnie zmienna=wyrażenie
print [lista-wyrażeń] [> wyrażenie]
printf  format [lista-wyrażeń] [> wyrażenie]
return [wyrażenie]
next                         #opuszcza pozostałe wzorce podczas analizy aktualnej linii
delete array [wyrażenie]     #usuwa wskazany element tablicy
exit [wyrażenie]             #kończy program zwracajac jako status wartość wyrażenia
```

Istnieje możliwość wykonania pewnych operacji przed przystąpieniem przez awk do analizy pierwszej linii zbioru jak i po zakończeniu analizy ostatniej linii - służą temu wzorce BEGIN i END (patrz: przykłady).

Wbudowane funkcje:

`getline[var]` - pobiera do zmiennej *var* (gdy nie jest podana to do $0) następną linię z aktualnie przetważanego zbioru.

`length([s])` - zwraca długośc łańcucha, domyślnie $0 jeżeli *s* nie jest podany,

`rand()` - zwraca liczbę pseudolosową z zakres (0,1),

`srand([expr])` - ustawia i zwraca wartość początkową generatora pseudolosowego, jeżeli *expr* nie jest podane czas systemowy jest używany w zamian.

`int(x)` - obcięcie części ułamkowej do wartości typu całkowitego,

`substr(s, m, [,n])` - zwraca *n* znaków z łańcucha *s* począwszy od pozycji *m*, jeżeli *n* nie jest podane, łańcuch zwracany jest do końca,

`index(s, t)` - zwraca pozycje, na której w łańcuchu *s* wystąpił łąńcuch *t*, w przeciwnym wypadku 0,

`match(s, ERE)` - zwraca pozycje, na której w łańcuchu *s* wystąpiło wyrażenie *ERE*, w przeciwnym wypadku zwraca 0,

`split(s, a [,fs])` - dokonuje podziału łańcucha na elementy i załadowaniu ich do tablicy *a[1]*, *a[2]…a[n]*. Zwracane jest *n*. Podział jest dokonywany na podstwie separatorów zdefiniowanych w wyrażeniu *fs*, gdy wyrażenia tego brak na podstawie tego co zawiera zmienna FS,

`sub(ERE, repl [,in])` - odszukuje pierwsze wystąpienie wzorca *ERE* w łańcuchu *in* (domyślnie $0) i zastępuje go przez *repl,* zwraca ilość wykonanych zastąpień,

`gsub(ERE, repl [,in])` - jak sub, ale zamienia wszystkie wystąpienia *ERE*, zwraca ilość wykonanych zastąpień,

`system(cmd)` - wykonuje polecenie systemowe *cmd* i zwraca status jego wykonania,

`toupper(s)` - zamienia znaki łańcucha *s* na duże litery i zwraca jako wynik,

`tolower(s)` - zamienia znaki łańcucha *s* na małe litery i zwraca jako wynik.

Zmienne wewnętrzne awk:

`FS` - wyrażenie regularne (RE) definiujące separatory rozdzielające pola,

`NF` - liczba pól w aktualnym rekordzie,

`NR` - numer kolejny rekordu (linii),

`FNR` - numer kolejny rekordu (linii) w danym analizowanym pliku,

`FILENAME` - nazwa analizowanego zbioru,

`RS` - rozpoznawany separator rekordów (domyślnie znak nowej linii),

`OFS` - wyprowadzany separator pól (domyślnie znak pusty),

`ORS` - wyprowadzany separator rekordów (domyślnie znak nowej linii),

`OFMT` - wyprowadzany format liczb (domyślnie%.6g),

`ARGC` - liczba podanych argumentów do programu,

`ARGV` - tablica zawierająca argumenty wywołania,

`ENVIRON` - tablica zmiennych środowiskowych, indeksem jest nazwa zmiennej, np.zmienna SHELL= /bin/sh, to ENVIRON["SHELL"] zwróci: /bin/sh.

Przykłady:

```
awk 'length > 80' p*
```
- wypisanie wszystkich lini dłuższych niż 80 znaków w zbiorach zaczynających się na literę **p.**

```
awk '{print $2 $1}'
```
- wypisuje ze wszystkich linii pierwsze dwa słowa w odwrotnym porządku.

```
cat my.cpp | awk '/start/, /stop/'
```
- wyprowadzi wszystkie linie ze zbioru my.cpp pomiędzy wyrażeniami start i stop.

```
awk '{ s+=$1}
END {print "Suma: ", s, "Srednia: ", s/NR}'
```
- dodaje pierwszą klumnę, wyświetla sumę i średnią.

14. Edytor *vi*

Edytor *vi* jest chyba najbardziej powszechnym edytorem tekstów w środowisku UNIX. Możemy wyróżnić dwa tryby pracy edytora: tryb wydawania poleceń i wstawiania tekstu. Jeżeli nie jesteśmy pewni w jakim trybie jesteśmy możemy naciskać klawisz ESC, wtedy przechodzimy do trybu wydawania poleceń. Do trybu wstawiania tekstu przechodzimy poleceniem **i** lub **a**, w zależności od tego gdzie chcemy wstawić pierwszy znak tekstu. **Wybrane polecenia**: **x** - usunięcie znaku na pozycji kursora, **u** - anuluje ostatnio wydane polecenie, **dd** - kasowanie wiersza, w którym jest kursor, **d*n*d** - kasowanie *n* wierszy, **ZZ** wyjście z zapisem, **:q!** - wyjscie bez zapisu, **XX** - usuwanie znaku przed kursorem, **b** - przenosi kursor o jedno słowo w lewo, **w** - o jedno słowo w prawo, **o** - wstawia pustą linię pod kursorem, **dw** - usuwa następne słowo, **r** - modyfikuje znak na którym jest kursor, **R** - modyfikuje dowolnie długi tekst. **Wybrane zaawanowane polecenia**: **:***nr_linii* **m** *nowy_nr_linii* - przenoszenie wskazanej linii w nowe miejsce, **:***linia_pocz linia_konc* **m** *nowa_linia* - przenoszenie wskazanego zakresu linii w podane miejsce, **:***nr_linii* **co** *nowy_nr_linii* - kopiowanie wskazanej linii w nowe miejsce, **:***linia_pocz linia_konc* **co** *nowa_linia* - kopiowanie wskazanego zakresu linii w podane miejsce, **?***wzorzec* - poszukiwanie wzorca w kierunku początku pliku (wzorzec to dowolny ciąg znaków), /wzorzec - poszukiwanie wzorca w kierunku końca pliku, **n** - powtarza szukanie tego samego wzorca w tym sammym kierunku, **N** - w przeciwnym kierunku, **:!***zlecenie* - wykonanie zlecenia systemowego podczas pracy z edytorem.

15. Dodatkowe informacje i literatura

[1] Wiecej informacji o możliwościach konkretnego zlecenia lub shell-a można uzyskać korzystając ze zlecenia **man *zlecenie.***
[2] Dokumentacja w internecie, np.: *www.jtz.org.pl* , *www.linuxpl.org* , *www.redhat.com*.
[3] „Unix in a nutshell", Daniel Gilly & the Staff of O'Reilly & Associates; ISBN 1-56592-001-5, 444 pages. Second Edition, June 1992 O'REILLY. Wersja on-line książki w internecie: http://lovecraft.die.udec.cl/orielly/unix/unixnut/index.htm
[4] „sed i awk. Leksykon kieszonkowy", Arnold Robbins, ISBN83-7197-465-5, Hellion 2001.
[5] „sed & awk", Dale Dougherty & Arnold Robbins; ISBN 1-56592-225-5, 432 pages, Second Edition, March 1997 O'REILLY. Wersja on-line książki w internecie: http://lovecraft.die.udec.cl/orielly/unix/sedawk/index.htm
[6] „Learning the vi editor", Linda Lamb; ISBN 0-937175-67-6, 173 pages, Fifth Edition, August 1994 O'REILLY. Wersja on-line książki w internecie: http://lovecraft.die.udec.cl/orielly/unix/vi/index.htm

Przygotowanie do laboratorium:

1. Jakie znasz powłoki (shelle) pośrednictwa znakowego użytkownika w systemach operacyjnych UNIX ?
2. Co to są polecenia wewnętrzne i zewnętrzne ?
4. Zapoznaj się z podstawowymi poleceniami UNIX wymienionymi w p.2 instrukcji (składnia, podstawowe parametry).
5. Zapoznaj się dokładnie z poleceniem *test*. Czym są programy *awk* i *grep* ?
7. Dla wprawy: stworzyć poniższy skrypt (p.8) w edytorze *vi*.
8. Dla wprawy: napisać skrypt, który z podanego jako parametr zbioru tekstowego stworzy zbiór, w którym linie będą w odwrotnej kolejności w stosunku do zbioru wejściowego. Dla ambitnych: uzupełnić skrypt o parametr, którego podanie będzie powodowało odwrócenie kolejność „wyrazów" w każdej linii.