6. Skrypty powłoki

Wstęp

Plik tekstowy zawierający listę poleceń dla interpretera nosi nazwę **skryptu**. Skrypty tworzone są w celu ułatwienia użytkownikom pracy z interpreterem. Umożliwiają zapisanie często używanych sekwencji poleceń i zastąpienie ich jednym poleceniem uruchomienia skryptu

W wykładzie 6 opisujemy tworzenie i posługiwanie się skryptami powłoki oraz prezentujemy zestaw wybranych programów, które są często wykorzystywane w skryptach.

6.1. Tworzenie i uruchamianie skryptów powłoki

Plik tekstowy zawierający listę poleceń dla interpretera nosi nazwę **skryptu**. Skrypty tworzone są w celu ułatwienia użytkownikom pracy z interpreterem. Umożliwiają zapisanie często używanych sekwencji poleceń i zastąpienie ich jednym poleceniem uruchomienia skryptu. Są szczególnie przydatne dla administratora systemu, który wielokrotnie wykonuje powtarzające się czynności, takie jak zakładanie kont nowym użytkownikom, tworzenie kopii zapasowych.

Skrypt może zawierać:

- odwołania do zmiennych,
- wbudowane polecenia interpretera,
- pętle sterujące,
- wywołania programów i innych skryptów,
- komentarze.

Wszystkie linie rozpoczynających się od znaku **#**, traktowane są przez powłokę jako komentarz i w związku z tym są ignorowane.

Argumenty wywołania

Argumenty wywołania skryptu przekazywane są do powłoki jako parametry pozycyjne. Parametr **$0** zawsze wskazuje nazwę skryptu, zaś parametry **$1, $2, ...** przechowują kolejne argumenty. Nazwa skryptu nie podlega przesunięciu w wyniku działania polecenia **shift**.

Metody uruchamiania skryptów

Skrypt zawiera listę poleceń, które powinien wykonać interpreter. Istnieje kilka sposobów uruchomienia skryptu.

Naturalnym sposobem jest jawne wywołanie nowej powłoki z pierwszym argumentem w postaci nazwy skryptu:

```
sh skrypt arg ...

bash skrypt arg ...
```

Wygodniej jest jednak posługiwać się wyłącznie nazwą skryptu jako nazwą nowego polecenia. W tym celu należy wcześniej ustawić prawo wykonywania skryptu:

```
chmod +x skrypt

skrypt arg ...
```

W takim przypadku nowa powłoki wywoływana jest niejawnie. Powstaje problem wyboru interpretera, który wykona skrypt. A wybór ten nie jest dowolny, gdyż każda powłoka stosuje inną składnię. Domyślnie uruchamiana jest powłoka **sh**. Użytkownik może wskazać dowolny interpreter umieszczając jego pełną nazwę ścieżkową w pierwszej linii skryptu po znakach **#!**, np.: #!`/bin/bash`.

W obydwu powyższych przypadkach uruchamiany jest nowy proces interpretera, który wykonuje polecenia zawarte w skrypcie, po czym kończy swoje działanie. Sterowanie wraca do procesu interpretera, w którym wydano polecenie. Nie są w nim widoczne zmiany dokonane w skrypcie takie, jak modyfikacja zmiennych czy zmiana bieżącego katalogu.

Skrypt może być również wykonany przez bieżący interpreter i wtedy wszystkie zmiany pozostają widoczne po zakończeniu skryptu. Służy do tego polecenie **.** („kropka”) lub **source** (z wyjątkiem powłoki **sh**):

```
. skrypt arg ...

source skrypt arg ...
```

Typowym zastosowaniem tej metody jest ponowne odczytanie plików konfiguracyjnych powłoki po wprowadzeniu modyfikacji, np:

```
. .bashrc
source .zshrc
```

Skrypt można wreszcie uruchomić zamiast bieżącego interpretera, posługując się poleceniem **exec**:

```
exec skrypt arg ...
```

W tym przypadku proces interpretera, który ma wykonać skrypt, zastępuje bieżący interpreter. Po zakończeniu działania skryptu nie istnieje już proces, w którym wydano polecenie. Sterowanie przechodzi więc do procesu nadrzędnego (macierzystego) lub następuje wylogowanie z systemu.

6.2. Podstawowe konstrukcje języka powłoki

Pętle i polecenia sterujące

Zestaw złożonych poleceń powłoki **bash** obejmuje większość typowych pętli i poleceń sterujących.

Pętla **for** wykonuje się raz dla każdego słowa z listy. Jeśli lista zostanie pominięta, pętla wykonuje się raz dla każdego ustawionego parametru pozycyjnego.

```
for zmienna [in słowo ...]
do lista_poleceń
done
```

Przykład
Wypisanie wszystkich argumentów wywołania.

```
for arg in $*
do
      echo $arg
done

for arg
do
      echo $arg
done
```

Pętla **for** może przyjąć również drugą postać, w której:

- wyrażenie1 określa warunek początkowy i obliczane jest tylko raz przed pierwszą iteracją pętli,
- wyrażenie2 pełni rolę warunku zakończenia pętli i obliczane jest na początku każdej iteracji dopóki nie osiągnie wartość zerowej,
- wyrażenie3 obliczane jest w każdej iteracji pętli po wykonaniu listy poleceń.

```
for ((wyrażenie1; wyrażenie2; wyrażenie3))
do lista_poleceń
done
```

Pętla **while** wykonywana jest dopóki ostatnie polecenie z listy warunków zwraca status zerowy.

```
while lista_warunków
do lista_poleceń
done
```

Pętla **until** wykonywana jest dopóki ostatnie polecenie z listy warunków zwraca status niezerowy.

```
until lista_warunków
do lista_poleceń
done
```

Polecenie **case** stara się dopasować słowo kolejno do każdego wzorca i wykonuje listę poleceń związaną z pierwszym pasującym wzorcem.

```
case słowo in
wzorzec [|wzorzec] ...) lista_poleceń;;
...
esac
```

W poleceniu **if** wykonywana jest gałąź, dla której ostatnie polecenie z listy warunków zwraca status zerowy.

```
if lista_warunków
then lista_poleceń
[elif lista_warunków
then lista_poleceń]
...
[else lista_poleceń]
fi
```

Pętle **for**, **while** i **until** mogą być przerywane za pomocą poleceń:

```
break [n]
continue [n]
```

Polecenie **break** przerywa wykonywanie pętli. Polecenie **continue** przerywa wykonywanie bieżącej iteracji pętli i wznawia następną iterację. Obydwa polecenia przerywają **n** poziomów zagnieżdżonych pętli, gdy podany jest argument. Domyślnie przyjmowane jest **n**=1.

Zakończenie działania powłoki umożliwia polecenie:

```
exit [status]
```

Argument oznacza status zakończenia zwracany przez powłokę. Domyślnie zwracany jest status zerowy.

Sprawdzanie warunków

Polecenia sterujące **if**, **while** i **until** wykorzystują jako warunek działania status zakończenia innego polecenia. Może to być dowolne polecenie systemu Linux. Najczęściej najbardziej praktyczne okazuje się klasyczne sprawdzenie jakiegoś warunku np. porównanie dwóch liczb lub ciągów znaków. Takich możliwości dostarcza wbudowany w powłokę mechanizm sprawdzania warunków:

```
[[ warunek ]]
```

oraz polecenie **test**, które można wywołać na dwa sposoby;

```
test warunek

[ warunek ]
```

We wszystkich przypadkach składnia specyfikowania warunku jest podobna. Istnieje możliwość sprawdzania atrybutów plików, porównywania ciągów znaków i liczb całkowitych. Liczby rzeczywiste traktowane są jak ciągi znaków.

Testowanie atrybutów pliku wygląda następująco:

Tablica 6.1 Testowanie atrybutów plików

Wyrażenie	Znaczenie
-r plik	Sprawdza, czy użytkownik posiada prawo do czytania,
-w plik	Sprawdza, czy użytkownik posiada prawo do pisania,
-x plik	Sprawdza, czy użytkownik posiada prawo do wykonywania,
-f plik	Sprawdza, czy **plik** to plik zwykły,
-d plik	Sprawdza, czy **plik** to katalog,
-c plik	Sprawdza, czy **plik** to plik specjalny znakowy,
-b plik	Sprawdza, czy **plik** to plik specjalny blokowy,
-p plik	Sprawdza, czy **plik** to plik FIFO,
-h plik	Sprawdza, czy **plik** to dowiązanie symboliczne,
-s plik	Sprawdza, czy **plik** to plik o niezerowej długości.

Przykład
Sprawdzanie, czy użytkownik posiada prawo pisania w katalogu **kat1**:

```
if [ -w kat1 ]
then
      mkdir kat1/kat2
fi
```

Testowanie ciągów znaków wygląda następująco:

Tablica 6.2 Testowanie ciągów znaków

Wyrażenie	Znaczenie
s1	Sprawdza, czy ciąg jest niezerowy.
s1 = s2	Sprawdza, czy ciągi są identyczne.
s1 != s2	Sprawdza, czy ciągi są różne.

Przykład
Wypisywanie kolejno argumentów wywołania skryptu:

```
while [ $1 ]
do
      echo $1
      shift
fi
```

Porównywanie liczb całkowitych umożliwiają operatory:

Tablica 6.3 Testowanie liczb całkowitych

Wyrażenie	Znaczenie
n1 -eq n2	sprawdza czy **n1 = n2**,
n1 -ne n2	sprawdza czy **n1 >< n2**,
n1 -gt n2	sprawdza czy **n1 > n2**,
n1 -ge n2	sprawdza czy **n1 >= n2**,
n1 -lt n2	sprawdza czy **n1 < n2**,
n1 -le n2	sprawdza czy **n1 <= n2.**

Przykład
Poprzedni przykład można również zapisać w następujący sposób:

```
while [ $# -gt 0 ]
do
      echo $1
      shift
done
```

Warunki można ze sobą łączyć lub negować, korzystając z następujących operatorów logicznych:

Tablica 6.4 Łączenie warunków

Wyrażenie	Znaczenie
[warunek1 –a warunek2]	Iloczyn warunków (operator AND)
[warunek1 –o warunek2]	Suma warunków (operator OR)
[! warunek]	Negacja warunku (operator NOT)
( )	Grupowanie warunków

Przykład
Sprawdzenie, czy podany argument wywołania skryptu jest katalogiem i ma ustawione prawo do pisania:

```
if [ ! -d $1 -o ! -x $1 ]
then
      echo ”Plik $1 nie jest katalogiem lub brak uprawnień”
      exit 1
else
      ls -l $1
fi
```

Operacje wejścia/wyjścia

Powłoka **bash** umożliwia wczytywanie danych ze strumienia wejściowego i wypisywanie komunikatów do strumienia wyjściowego.

```
read [opcje] [zmienna ...]
```

Polecenie **read** odczytuje jedną linię ze strumienia wejściowego **stdin**. Następnie dokonuje podziału na wyrazy, stosując separatory zdefiniowane w zmiennej IFS i przypisuje kolejne wyrazy zmiennym podanym na liście argumentów.

```
echo [opcje] [arg ...]
```

Polecenie **echo** przesyła na standardowe wyjście **stdout** argumenty rozdzielone znakiem spacji. Argumentem może być dowolny ciąg znaków, zawierający również znaki specjalne powłoki, np. odwołania do zmiennych.

```
printf format [arg ...]
```

Polecenie **printf** wypisuje argumenty na standardowym wyjściu zgodnie z podanym formatem. Sposób formatowania jest identyczny jak w przypadku funkcji **printf**() języka C.

6.3. Operacje arytmetyczne

Mechanizmy wbudowane w powłokę

Niektóre interpretery, jak **sh** i **csh**, nie mają wbudowanych żadnych operacji arytmetycznych. Powłoki **bash** i **zsh** udostępniają możliwość rozwijania wyrażeń arytmetycznych bezpośrednio w poleceniach:

```
$((wyrażenie))
```

oraz poprzez wbudowane polecenie **let:**

```
let wyrażenie [wyrażenie ...]
```

Każdy argument polecenia **let** traktowany jest jako oddzielne wyrażenie, które należy niezależnie obliczyć. Wyrażenie może zawierać jedynie argumenty całkowite i operatory. Zestaw dostępnych operatorów oraz priorytety poprzedzania są identyczne jak w języku C. Poniżej przedstawiamy wybrane operatory, zaś pełną listę można znaleźć w dokumentacji elektronicznej lub w literaturze. Wybrane operatory przedstawiono w Tabl. 6.5.

Tablica 6.5 Operatory polecenia let

Kategoria operatorów	Operatory
arytmetyczne	**++, --, **, *, / , %, -, +**
logiczne	**!, &&, ||**
bitowe	**~, <<, >>, &, ^, |**
podstawienia	**=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=**
porównania	**<=, >=, <, >, ==, !=**

Przykład
Obliczenia z wykorzystaniem poleceń powłoki **bash**.

```
$ let x=5*6
$ let x=x+1
$ echo $x
31
$ echo $((x=7*8))
56
$ x=$((x+1))
$ echo $x
57
```

Program expr

Do przeprowadzenia operacji obliczeniowych można również wykorzystać program **expr** uruchamiany w następujący sposób:

```
expr wyrażenie

expr arg1 operator arg2 ...
```

Program oblicza wartość podanego wyrażenia i wysyła do **stdout**. Podstawienia można dokonać niezależnie w powłoce:

```
zmienna=`expr wyrażenie`
```

Wszystkie argumenty i operatory w wyrażeniu muszą być rozdzielone znakami spacji. Operatory można podzielić na trzy grupy (Tabl. 6.6).

Tablica 6.6 Operatory polecenia expr

Kategoria operatorów	Operatory
arytmetyczne	**+, -, *, /, %**
porównania	**=, !=, \>, \>=, \<, \<=**
dopasowania	**:**

Niektóre operatory (np. *) są znakami specjalnymi i mogą zostać przedwcześnie zinterpretowane przez powłokę zamiast przez program **expr**. W związku z tym powinny być poprzedzone znakiem ****, aby przekazać je bezpiecznie do programu **expr**. Operatory arytmetyczne działają na liczbach całkowitych. Operatory porównania zapewniają porównanie arytmetyczne liczb całkowitych lub porównanie leksykalne innych argumentów. Operator **:** umożliwia sprawdzenie ile znaków z podanego ciągu pasuje do wzorca:

```
string : wzorzec
```

Wzorzec może zawierać znaki specjalne. Zestaw dostępnych znaków specjalne jest taki jak dla programu **grep**.

Przykład
Wykorzystanie polecenia **expr** do wykonywania obliczeń:

```
x=`expr $x + 1
```

Wykorzystanie operatora dopasowania w poleceniu **expr**:

```
expr $1 : '.*'
expr $1 : '[0-9]*'
```

Przykład
Skrypt do sumowania argumentów

```
#!/bin/sh

if [ $# -eq 0 ]
then
   echo Poprawne wywolanie: $0 arg1 arg2 ...
   exit 1
fi
suma=0
while [ $1 ]
do
  suma=`expr $suma + $1 2 > /dev/null`
  if [ $? -eq 2 ]
  then
     echo "Niewlasciwy argument $1 !"
     exit 2
  fi
  shift
done
echo "Suma argumentow wynosi: $suma";
```

6.4. Funkcje powłoki

Interpreter umożliwia definiowanie własnych funkcji:

```
nazwa () {lista_poleceń;}
```

Odwołanie do funkcji następuje przez nazwę, podobnie jak do wszystkich poleceń. Funkcje są zawsze wykonywane przez bieżącą powłokę. Argumenty wywołania funkcji zostają ustawione jako parametry pozycyjne powłoki wyłącznie na okres wykonywania funkcji. Funkcja zwraca swój strumień wyjściowy stout. Zwracaną wartość można podstawić pod zmienną w następujący sposób:

```
x=`nazwa arg1 arg2`
```

Przykład

Zdefiniowana poniżej funkcja wyświetla informacje o poprawnym wywołaniu skryptu

```
usage()
{
echo "Poprawne wywolanie to : nazwa_skryptu arg1 op1 arg2 ... "
echo "\nWywolania,operatory i realizowane dzialania "
echo "Dodawanie np: 2 + 3 "
echo "Odejmowanie np: 4 - 5 "
echo "Dzielenie np: 4 / 2 "
echo "Mnozenie np: 4 * 4 "
echo "Potegowanie np: 4 \^ 4"
exit
}
```

Funkcja **plus**() sumuje dwa argumenty:

```
plus() {
      echo `expr $1 + $2`
}
```

Funkcję można wywołać w następujący sposób:

```
$ plus 3 5
8
```

lub podstawić wynik pod zmienną:

```
$ x=`plus 2 7`
$ echo $x
9
```

6.5. Filtr grep

Filtry to grupa programów służących do wyszukiwania wzorców i/lub przetwarzania plików tekstowych. Programy te są bardzo często wykorzystywane do tworzenia poleceń i aliasów powłoki, skryptów instalacyjnych oraz skryptów przeznaczonych do wykonywania złożonych zadań administracyjnych. Nie jest to jedyne zastosowanie filtrów, gdyż przydatne mogą być one wszędzie tam gdzie zachodzi konieczność wyszukiwania danych w dużej liczbie plików lub w plikach o bardzo dużej objętości (często okoliczności te występują jednocześnie) - na przykład w zastosowaniach inżynierskich do przetwarzania wyników symulacji do postaci umożliwiającej wykonania żądanych zestawień i wykresów.

Program **grep** poszukuje w standardowym strumieniu wejściowym lub w plikach wejściowych wierszy, które pasują do podanego wzorca. Każdy znaleziony wiersz jest przesyłany w całości na standardowy strumień wyjściowy, o ile nie zastosowano opcji zmieniającej to zachowanie. Wzorzec ma postać dowolnego wyrażenia, w którym mogą wystąpić znaki specjalne. Postać wywołania programu jest następująca:

```
grep [opcje] wzorzec [plik]...
```

gdzie:

wzorzec - wyrażenie opisujące poszukiwany wzorzec,

plik - nazwa pliku wejściowego - jeśli pominięta grep czyta dane ze strumienia wejściowego,

opcje - lista wybranych opcji .

Najczęściej stosowane opcje programu **grep** to:

-c - podaje tylko liczbę dopasowanych linii,

-n - dopisuje numer przed każdą dopasowaną linią,

-v - wypisuje tylko te linie, które **nie pasują** do wzorca.

Znaki specjalne rozpoznawane przez program grep pokazane w tablicy 6.7

Tablica 6.7 Znaki specjalne programu grep

Znak	Znaczenie
^	Początek wiersza.
$	Koniec wiersza.
.	Jeden dowolny znak.
*	Dowolna liczba wystąpień poprzedzającego wyrażenia (w tym zero wystąpień).
[ ]	Jeden dowolny znak z listy zawartej wewnątrz nawiasów.
[-]	Jeden dowolny znak z zakresu podanego wewnątrz nawiasów.
\	Przywraca pierwotne znaczenie następnego znaku.

Jak widać są to znaki, które są również znakami specjalnymi powłoki. Jeśli wyrażenie opisujące wzorzec zawiera takie znaki to powinno zostać umieszczone w apostrofach, np.:

```
ls * | grep '^[Z].[a-d]*graf\.c$'.
```

W ten sposób powłoka nie dokona interpretacji tekstu wzorca i zostanie ono przekazany w oryginalnej postaci programowi grep.

Dostępne są też dwie odmiany programu grep: **fgrep** i **egrep**.

Program **fgrep** dopuszcza stosowanie tylko uproszczonych wzorców, które nie zawierają żadnych znaków specjalnych. Inaczej mówiąc wzorzec jest dokładnie takim tekstem jakiego poszukujemy w danych wejściowych, np.:

```
ps -ef | fgrep inetd
```

W przypadku programu egrep stosować można wzorce o rozszerzonym zakresie znaków specjalnych. Niektóre dodatkowe znaki specjalne pokazano w Tabl. 6.8.

Tablica 6.8 Dodatkowe znaki specjalne programu egrep

Znak	Znaczenie
+	Liczba wystąpień poprzedzającego wyrażenia > **1**.
?	Liczba wystąpień poprzedzającego wyrażenia = **0** lub **1**.
{n}	Poprzedzające wyrażenie powtarza się dokładnie ***n*** razy.

Przykład

Typowym zastosowaniem programu grep jest filtrowanie informacji ze strumienia wyjściowego innego procesu. W pierwszym z poniższych poleceń **grep** wyszukuje w liście procesów proces o nazwie **sendmail**, a w drugim poleceniu – zlicza linie zaczynające się od litery ‘d’, czyli zlicza katalogi.

```
ps –ef | grep sendmail
ls –l | grep –c ”^d”
```

6.6. Edytor strumieniowy sed

Program **sed** służy do edycji strumieni danych. Znajduje zastosowanie przy tworzeniu filtrów do przetwarzania wsadowego, przy edycji dużych plików, które nie mieszczą się w buforze, przy przetwarzaniu potokowym oraz w skryptach shellowych.

Edytor **sed** przesyła strumień wejściowy lub zawartość plików wejściowych do strumienia wyjściowego, wykonując operacje edycyjne kolejno na każdej linii, zgodnie z podanym programem. Program zawiera polecenia, poprzedzone adresami linii wejściowych, do których powinny być zastosowane. Edytor kolejno kopiuje jedną linię do przestrzeni roboczej (do bufora roboczego) i stosuje do niej sekwencję tych poleceń, których adresy wskazują daną linię. Następnie przesyła linię do strumienia wyjściowego oraz czyści bufor roboczy. Niektóre polecenia wykorzystują bufor pomocniczy do zapamiętywania fragmentów tekstu.

Wywołanie programu **sed** może przybrać jedną z następujących postaci:

```
sed [-n] 'program' [plik_danych ...]

sed [-n] -e 'program' [-e 'program']... [-f plik_z_programem]...
[plik_danych...]

sed [-n] -f plik-z-programem [-f plik_z_programem]... [-e 'pro-
gram']... [plik_danych...]
```

gdzie:

program	-	program opisujący operacje edycyjne,
-e 'program'	-	pobranie programu z linii polecenia, znaki **''** zabezpieczają przed interpretacją elementów programu przez powłokę shell,
-f plik_z_programem	-	pobranie programu z pliku plik_z_programem,
plik_danych	-	plik wejściowy - jeśli pominięty to sed czyta dane ze strumienia wejściowego,
-n	-	przesyła na wyjście tylko wskazane w programie linie.

Program musi być wywołany przynajmniej z jedną opcją **-e** lub **-f.** Pojedynczą opcję **-e** można pominąć, podając jedynie program.

Program dla edytora sed składa się z linii zawierających **adres** oraz **polecenie** z argumentami lub **zestaw poleceń**. Każda linia może przyjąć jedną z następujących postaci:

```
[adres] polecenie [argumenty]

[adres] {
            lista poleceń
        }
```

Polecenia wykonywane są w kolejności występowania w programie. Kolejność tę można zmienić poprzez wykonanie skoku do etykiety.

Polecenie może być poprzedzone 1 lub 2 adresami lub też pozbawione adresu. Polecenie z jednym adresem stosowane jest do linii wskazanych przez adres. Polecenie z dwoma adresami stosowane jest do zakresu ograniczonego liniami wskazanymi przez adresy. Polecenie bez adresu stosowane jest do każdej linii wejściowej.

Adresowanie linii może odbywać się poprzez podanie numeru linii (**adres numeryczny**) lub przez podanie jej kontekstu (**adres kontekstowy**). Adresy numeryczne są podawane w postaci:

```
[nr1[, nr2]]
```

a adresy kontekstowe jako:

```
[/wzorzec[/][, /wzorzec[/]]
```

Adresy numeryczne są więc parą liczb całkowitych dodatnich lub jest to jedna taka liczba. Program sed korzysta z własnego licznika linii, który kumuluje ich liczbę dla wszystkich plików wejściowych łącznie. Znak **$** identyfikuje ostatnią linię w ostatnim pliku.

Adresy kontekstowe mają postać wzorca ograniczonego dwoma identycznymi znakami: **/wzorzec/** lub **\znak wzorzec znak** Wzorzec jest dowolnym wyrażeniem, w którym mogą wystąpić znaki specjalne. Edytor rozpoznaje następujące znaki (tablica 4.9):

Tablica 4.9 Znaki specjalne programu sed

Znak	Znaczenie
^	Początek linii.
$	Koniec linii.
\n	Znak nowej linii z wyjątkiem ostatniego znaku w buforze.
.	Jeden dowolny znak.
*****	Dowolna liczba wystąpień poprzedzającego wyrażenia.
[ ]	Jeden dowolny znak z listy zawartej wewnątrz nawiasów.
[-]	Jeden dowolny znak z zakresu podanego wewnątrz nawiasów.
[^...]	Jeden dowolny znak z wyjątkiem tych, podanych wewnątrz nawiasów oraz kończącego bufor znaku nowej linii.
****	Przywraca pierwotne znaczenie następnego znaku.
\(\)	Określa pole, do którego można odwołać się poprzez numer.
\nr	Odwołanie do pola o numerze **nr** (nr = 0 - 9).

Polecenia edytora sed można podzielić na trzy grupy:

1. polecenia edycyjne,
2. polecenia wejścia/wyjścia,
3. polecenia sterujące.

Polecenia te są zgromadzone w Tabl. 6.10, 6.11 i 6.12.

Tablica 6.10 Polecenia edycyjne programu sed

Polecenie	Opis
[adres]d	Usuwanie wskazanych linii.
[adres]n	Wczytanie do przestrzeni roboczej kolejnej linii. Bieżąca linia jest przesyłana na stdout.
[adres]a \[RETURN] text	Wstawianie podanego tekstu za wskazaną linią (konieczne jest zastosowanie znaku przejścia do nowego wiersza **[RETURN]**).
[adres]i \[RETURN] text	Wstawianie podanego tekstu przed wskazaną linią (konieczne jest zastosowanie znaku przejścia do nowego wiersza **[RETURN]**).
[adres]c \[RETURN] text	Wstawianie podanego tekstu zamiast wskazanych linii (konieczne jest zastosowanie znaku przejścia do nowego wiersza **[RETURN]**).
[adres]s/wzorzec/tekst/flagi	Zamiana fragmentu linii, który pasuje do wzorca na podany tekst (zamiast znaków **//** mogą być użyte dowolne inne).
wzorzec	Wyrażenie specjalne, które może zawierać znaki specjalne.
tekst	Ciąg znaków, w którym można zastosować symbole **& i \d**: • **&** - będzie zawierał napis pasujący do wzorca, • **\d** - będzie zawierał podstring numer **d** pasujący do fragmentu wzorca pomiędzy znakami **\(\).**
flagi	Opcje dla polecenia zamiany: • **n** - zamienia n wystąpień, n = 1 - 512, • **g** - zamienia globalnie wszystkie wystąpienia wzorca, • **p** - przesyła linię na wyjście, jeśli nastąpiła zamiana, • **w *plik*** - przesyła linię do pliku ***plik***, jeśli nastąpiła zamiana.

Tablica 4.11 Polecenia wejścia/wyjścia programu sed

Polecenie	Opis
[adres]p	Przesłanie wskazanych linii na stdout.
[adres]=	Przesłanie numerów wskazanych linii na stdout.
[adres]w plik	Zapisanie wskazanych linii do pliku.
[adres]r plik	Wstawienie zawartości pliku za wskazaną linią.

Tablica 4.12 Polecenia sterujące programu sed

Polecenie	Opis
[adres]! funkcja	Zastosowanie podanej funkcji tylko do linii nie wybranych przez adres.
[adres]{ polecenia }	Grupowanie poleceń.
: etykieta	Ustawienie etykiety.
[adres]b etykieta	Skok do etykiety lub na koniec programu, gdy jej brak.
[adres]t etykieta	Skok do etykiety, jeśli nastąpiły jakieś zmiany w bieżącej linii.
#	Linia komentarza.
#n	Jeśli występuje w pierwszej linii programu, wyłącza standardowe przesyłanie na stdout.
[adres]q	Skok na koniec programu i zakończenie jego działania.

Przykład

Polecenie wyszukujące w pliku wejściowym linie zaczynające się od dwukrotnego wystąpienia dowolnego wzorca (rozdzielonych spacją):

```
sed -n '/^\(.*\) \1/p' plik
```

Polecenie wypisujące zawartość pliku zamieniając wystąpienia wzorca tekst1 na wzorzec tekst2:

```
sed 's/tekst1/tekst2/g' plik
```

6.7. Procesor tekstu awk

Awk jest językiem programowania przeznaczonym do przetwarzania tekstów. Stosowany jest do generowania raportów, wyszukiwania wzorców, filtrowania i formatowania danych przesyłanych innym programom. Oprócz podstawowej wersji awk istnieją również implementacje nazywane **nawk** i **gawk** o rozszerzonych możliwościach. Szczegółowe informacje o programie **[ng]awk** dostępnym w danym systemie można uzyskać dzięki poleceniu man. W dalszej części lekcji przekazane zostaną podstawowe informacje o posługiwaniu się programem **awk**.

Awk przetwarza zawartość strumienia wejściowego lub kolejnych plików wejściowych zgodnie z podanym programem. Program zawiera **wzorce** oraz związane z nimi **akcje**, które należy wykonać na danych wejściowych pasujących do wzorców. Zawartość każdego pliku wejściowego jest przeglądana jednokrotnie. Wywołanie programu awk może przyjmować dwie formy:

```
awk [-Fseparator_pola] 'program' [[-v] zmienna=wartość]... [plik]...

awk [-Fseparator_pola] -f plik_z_programem [[-v] zmienna=wartość]...
[plik]...
```

gdzie:

program - program zawierający polecenia awk; znaki **''** zapobiegają interpretacji przez powłokę shell,

-f plik_z_programem - pobranie programu z pliku,

-Fseparator_pola - ustawienie znaku ***separator_pola*** jako separatora pola w rekordzie,

plik - plik wejściowy zawierający tekst do przetworzenia,

[-v] zmienna=wartość - nadanie wartości zmiennej, która będzie dostępna w głównym module programu;

opcja **-v** udostępnia zmienną w module inicjującym.

Polecenie awk może być zastosowane na trzy różne sposoby:

1. przez wywołanie awk wraz z programem w linii polecenia,
2. przez wywołanie awk w linii polecenia, program umieszczony w pliku,
3. przez wywołanie awk wraz z programem umieszczone w skrypcie shella.

Pierwszy sposób stosuje się w przypadku krótkich programów (jedna lub dwie linie), które nie będą często wykorzystywane. Programy dłuższe lub przeznaczone do częstego użytkowania umieszcza się na ogół w pliku. Najczęściej stosowany jest trzeci sposób.

Program awk przetwarza dane wejściowe według następującego algorytmu. Dane wejściowe dzielone są na **rekordy** zgodnie z występowaniem **separatorów rekordu**. Standardowo jest to znak nowego wiersza, ale można zmienić to ustawienie, przypisując inne znaki zmiennej **RS**. Awk wczytuje kolejno po jednym rekordzie do przestrzeni roboczej i dokonuje podziału na **pola** zgodnie z rozmieszczeniem **separatorów pola**. Zazwyczaj są to znaki spacji i tabulacji, lub inne ustawione w zmiennej **FS** lub w linii wywołania przez opcję **-Fseparator_pola**. Do poszczególnych pól można się odwoływać poprzez następujące zmienne:

$0 - cały bieżący rekord,

$1, $2 ... - kolejne pola bieżącego rekordu,

$NF - ostatnie pole bieżącego rekordu.

Dodatkowo zdefiniowane są zmienne:

NF - liczba pól bieżącego rekordu,

NR - liczba dotychczas wczytanych rekordów,

FNR - liczba rekordów wczytanych z bieżącego pliku,

Bieżący rekord porównywany jest kolejno ze wszystkimi wzorcami w programie. W przypadku dopasowania wykonywane są akcje związane z danym wzorcem. Po wykonaniu całego programu następuje usunięcie rekordu z przestrzeni roboczej i wczytanie nowego. Program dla awk jest sekwencją instrukcji o następującej postaci:

```
wzorzec { akcja }
```

lub

```
wzorzec { blok akcji }
```

Poszczególne akcje w bloku mogą być oddzielone średnikiem lub znakiem nowego wiersza. Pominięcie wzorca w instrukcji powoduje zastosowanie akcji do wszystkich rekordów wejściowych. Pominięcie akcji powoduje przesłanie wszystkich pasujących rekordów do strumienia wyjściowego. W programie można również umieścić linie komentarza rozpoczynające się znakiem **#**.

Wzorce

Pole **wzorzec** określa, czy związana z nim akcja ma zostać zastosowana do bieżącego rekordu. W programie można stosować trzy typy wzorców:

- wzorce specjalne **BEGIN** i **END**,
- izolowane wyrażenia specjalne,
- wyrażenia wartościowane jako prawda lub fałsz.

W przypadku zastosowania konstrukcji **BEGIN { akcja }** akcja wykonywana jest na początku, jeszcze przed rozpoczęciem przetwarzania danych wejściowych. Daje to możliwość inicjalizacji zmiennych (np. separatorów rekordu **RS** i pola **FS**) lub wypisania nagłówków w raporcie.

W przypadku użycia konstrukcji **END { akcja }** akcja związanawykonywana jest po zakończeniu przetwarzania danych wejściowych. Umożliwia to np. umieszczenie podsumowania w raporcie.

Jeśki zastosujemy konstrukcję **/wzorzec/ { akcja }** to wzorzec może mieć postać izolowanego wyrażenia ze znakami specjalnymi, umieszczonego pomiędzy znakami **/ /**.

Polecenie **awk** akceptuje następujący zestaw znaków specjalnych (Tabl. 6.13).

Tablica 6.13 Znaki specjalne programu awk

Znak	Znaczenie
^	Początek wiersza.
$	Koniec wiersza.
\n	Znak nowego wiersza z wyjątkiem ostatniego znaku w buforze.
.	Jeden dowolny znak.
*	Zero lub więcej wystąpień poprzedzającego wyrażenia.
+	Jedno lub więcej wystąpień poprzedzającego wyrażenia.
?	Zero lub jedno wystąpienie poprzedzającego wyrażenia.
|	Jeden znak z pary.
[ ]	Jeden dowolny znak z listy zawartej wewnątrz nawiasów.
[-]	Jeden dowolny znak z zakresu podanego wewnątrz nawiasów.
[^...]	Jeden dowolny znak z wyjątkiem tych, podanych wewnątrz nawiasów oraz kończącego bufor znaku nowej linii.
\	Przywraca pierwotne znaczenie następnego znaku.

W ogólnym przypadku wzorzec przyjmuje postać dowolnego wyrażenia, któremu można przypisać wartość logiczną prawda lub fałsz. Wyrażenie może zawierać znaki specjalne oraz dowolne operatory. Najczęściej stosowane są operatory dopasowania wzorców np.:

```
string ~ /wzorzec/ { akcja }
string !~ /wzorzec/ { akcja }
```

oraz operatory relacji np.:

```
string1 == string2 { akcja }
liczba1 > liczba2 { akcja }
```

Akcje

Blok akcji ma postać sekwencji poleceń wejściowych i wyjściowych, poleceń sterujących oraz operacji przypisania wartości zmiennym.

W składni **awk** dostępne są następujące polecenia i pętle sterujące:

```
if (warunek) { blok1 } [ else { blok2 } ]

while (warunek) {blok}

for (wyrażenie; wyrażenie; wyrażenie) { blok }

for (zmienna in tablica) { blok }
```

Wewnątrz bloków stosować można następujące instrukcje zmieniające wykonywanie pętli:

break - przerywa bieżący obieg pętli i rozpoczyna następny,

continue - przerywa bieżący obieg pętli i rozpoczyna go od początku,

next - przerywa wykonywanie programu dla bieżącego rekordu, wczytuje nowy rekord i rozpoczyna przetwarzanie od początku programu,

exit - przerywa wykonywanie programu.

Operacje wejścia /wyjścia mogą być realizowane za pomocą następujących instrukcji:

Tablica 6.14 Operacje wejścia/wyjścia dostępne w programach awk

Polecenie	Znaczenie
print [wyrażenie][[,] wyrażenie]...	Wypisuje kolejne wyrażenia rozdzielone wyjściowym separatorem pola podanym w zmiennej **OFS**; na końcu umieszczany jest wyjściowy separator rekordu pamiętany w zmiennej **ORS.**
printf (format, wyrażenie [, wyrażenie] ...)	Formatuje i wypisuje wyrażenie (składnia jak w języku C).
getline	Wczytuje kolejny rekord z bieżącego pliku wejściowego do przestrzeni roboczej.
getline zmienna	Wczytuje kolejny rekord z bieżącego pliku wejściowego do zmiennej, nie zmieniając zawartości przestrzeni roboczej.
getline < plik	Wczytuje rekord z podanego pliku do przestrzeni roboczej.
getline zmienna < plik	Wczytuje rekord z podanego pliku do zmiennej, nie zmieniając zawartości przestrzeni roboczej.
system (polecenie shella)	Przekazuje polecenie do wykonania w shellu; strumień wyjściowy polecenia jest włączany w strumień wyjściowy awk.

W składni awk dostępne są **zmienne skalarne**, **tablicowe jednowymiarowe** oraz **stałe numeryczne i tekstowe**. Stałe tekstowe mają postać dowolnego napisu objętego znakami " ". Wartość zmiennej może być interpretowana jako tekst lub wartość numeryczna. Wszystkie zmienne są inicjalizowane napisem zerowym (wartość numeryczna 0). Operacja przypisania wartości zmiennej ma ogólną postać:

```
zmienna operator wyrażenie
```

Listę dostępnych operatorów zamieszczono w tablicy 4.14. W przypadku tablic nie deklaruje się ich rozmiaru ani typu elementów. Nowe elementy pojawiają się w momencie pierwszego przypisania wartości. Indeksy tablicy mogą być dowolnymi napisami (w szczególności kolejnymi liczbami całkowitymi).

Wyrażenia występują zarówno we wzorcach jak i w akcjach. Mogą zawierać odwołania do stałych, zmiennych i do pól rekordów, wywołania funkcji oraz dowolne operatory spośród wymienionych poniżej:

Tablica 4.14 Operatory dostępne w programach awk

Typ operatora	Operator
arytmetyczne	**+ , - , * , / , % , (...) , ^**
przypisania	**= , += , -= , *= , /= , %= , ++ , --**
relacji	**< , <= , == , != , > , >=**
logiczne	**|| , && , ! , ?:**
porównywania wzorców	**~ , !~**

W programach awk można posługiwać się również **predefiniowanymi funkcjami do operacji na tekstach** oraz **funkcjami numerycznymi**. Są one pokazane w tablicach 4.15 i 4.16.

Tablica 4.15 Funkcje awk do operacji na tekstach

Funkcja	Opis działania
INDEX (s1, s2)	Zwraca indeks pierwszego wystąpienia stringu s2 w stringu s1 lub 0, jeśli dopasowanie nie następuje.
length (string)	Zwraca długość stringu.
match (string, wzorzec)	Zwraca indeks pierwszego wystąpienia wzorca w stringu lub 0, jeśli dopasowanie nie następuje.
split (string, tablica, separator)	Dzieli string na pola (rozdzielone separatorami) oraz przypisuje kolejnym elementom tablicy; funkcja zwraca liczbę pól.
substr (string, pozycja, długość)	Zwraca podstring o podanej długości, rozpoczynający się od wskazanej pozycji.
sprintf (format, wyrażenie [, wyrażenie]...)	Formatuje wyrażenie i zwraca w postaci stringu.
sub(wzorzec, nowy_podstring [, string])	Zamienia pierwsze wystąpienie wzorca w danym stringu na nowy podstring.
gsub(wzorzec, nowy_podstring [, string])	Zamienia wszystkie wystąpienia wzorca w danym stringu na nowy podstring.

Tablica 4.16 Funkcje numeryczne awk

Funkcja	Opis działania
atan2 (y,x)	arcus tangens ***y/x*** w radianach
cos (rad)	cosinus kąta ***rad***
exp (x)	funkcja ***ex***
int (x)	część całkowita ***x***

log (x)	ogarytm naturalny z ***x***
rand ()	generator liczb pseudolosowych z przedziału < 0, 1 >
srand ([seed])	inicjuje generator liczb pseudolosowych
sin (rad)	sinus kąta ***rad***
sqrt (x)	pierwiastek kwadratowy z ***x***

Oprócz funkcji standardowo dostępnych w awk użytkownik ma możliwość zdefiniowania własnych funkcji zgodnie z następującą składnią:

```
function nazwa (arg... ) { ciało funkcji }
```

Wewnątrz ciała fukcji można skorzystać z funkcji:

```
return (wyrażenie)
```

która zwraca wartość podanego wyrażenia.

W przypadku podania treści programu w wywołaniu polecenia awk, program umieszcza się w znakach **' '**. Zapobiega to interpretowaniu znaków specjalnych awk przez powłokę shell. W niektórych przypadkach dopuszczenie do takiej interpretacji jest jednak konieczne, np. w celu udostępnienia argumentów wywołania skryptu w programie awk. Dokonuje się tego poprzez ograniczenie fragmentu programu, przeznaczonego do interpretacji, znakami ' '. Program zostaje w ten sposób "rozcięty" na dwie części zabezpieczone znakami ' ' oraz wskazany fragment pomiędzy nimi. Poniższy przykład ilustruje sposób pobrania wartości pierwszego argumentu wywołania skryptu:

```
awk ' ... '$1' ... '
```

Inną metodą jest rezygnacja z umieszczania całego programu pomiędzy znakami ' ' i zabezpieczanie jedynie wybranych fragmentów. Może to być jednak bardziej pracochłonne.

Przykład
Poniżej podano kilka prostych przykładów zastosowania **awk**. Zachęcamy do ich przeanalizowania oraz praktycznego przetestowania ich działania.

Pierwszy przykład pokazuje zastosowanie awk do "odfiltrowania" zbędnych kolumn w raporcie wypisywanym przez polecenie **ps**:

```
ps -ef | awk 'BEGIN {print "Procesy uzytkownika:"}
           $1 ~ /lab1$/ {print $2, $8}
           END {print "Koniec"} '
```

Następne dwa polecenia wypisują zawartość pliku zamieniając wystąpienia wzorca **tekst1** na wzorzec **tekst2**:

```
awk '{gsub("tekst1", "tekst2", $0); print $0}' plik
awk '{gsub("tekst1", "tekst2"); print}' plik
```

Bibliografia

[1] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 4, 5, 6)

[2] Petersen R.: Arkana Linux, Wydawnictwo RM 1997 (rozdziały: 5, 14, 15)

Zadania do wykładu 6

Zadanie 1

Wykorzystując polecenie expr, napisz skrypt działający jako prosty kalkulator liczb całkowitych z czterema podstawowymi działaniami (+ - * /) oraz potęgowaniem (x ^ n, gdzie: x - liczba całkowita, n - liczba naturalna). Należy uwzględnić priorytet operatorów (^, * /, + -). Operację potęgowania zrealizować w postaci funkcji.

Postać wywołania skryptu: `nazwa_skryptu arg op arg op arg ...`

Skrypt powinien obliczyć i podać wynik końcowy oraz sygnalizować błędy składni:

- niepoprawny typ argumentów (tylko argumenty całkowite),
- niepoprawny operator (poprawne operatory: + - * / ^),
- niepoprawna liczba argumentów,
- inne, np. dzielenie przez 0.

Zadanie 2

Wykorzystując program **sed**, napisz skrypt służący do tworzenia prototypów funkcji znalezionych w tekście programu w języku C. Skrypt poszukuje w plikach wejściowych definicji funkcji i wypisuje na standardowe wyjście prototyp każdej znalezionej funkcji. Ograniczyć się do rozpoznawania typów **char**, **int**, **short**, **long**, **double**, **float**, **void** oraz typów wskaźnikowych do wymienionych typów (np. double **).

Postać wywołania skryptu: `nazwa_skryptu lista_plików`

UWAGA

Wszystkie skrypty powinny zawierać obsługę błędów:

- sygnalizować błędy składni (podając poprawną postać wywołania skryptu),
- sygnalizować użycie niepoprawnego argumentu,
- sygnalizować brak odpowiednich praw dostępu.