Systemy operacyjne

Laboratorium 1

Podstawy oraz powłoka BASH

Jarosław Rudy
Politechnika Wrocławska

28 lutego 2017

Powłoka `bash` jest podstawową powłoką systemu Linuks, która będzie używana przez większą część kursu. Zrozumienie działania tej powłoki jest niezwykle istotne, ponieważ wiele późniejszych komend i mechanizmów (`find`, `awk`, potoki) korzysta z tej powłoki lub jest uruchamiane z jej poziomu. Laboratorium to ponadto obejmuje ogólne podstawy pracy w systemach typu Linuks (lub Uniks), wliczając w to podstawową obsługę konsoli, edytorów tekstów oraz tworzenie i uruchamianie skryptów `bash`owych.

1 Wymagania ogólne

Poniżej znajduje się lista wymagań odnośnie dozwolonego sposobu pisania skryptów `bash`owych (pierwsze sześć tematów).

1. Wszystkie skrypty powinny być wywoływane jako pliki uruchamialne (np. `./skrypt`) zamiast jawnego wywoływania `bash`a (np. `bash skrypt`).
2. Wszystkie skrypty powinny działać poprawnie niezależnie od tego, z którego miejsca systemu plików zostały wywołane.
3. Definiowanie funkcji nie jest zwykle potrzebne i nie jest zalecane podczas laboratorium (chociaż jest przydatne w praktyce).
4. Użycie tablic jest zwykle zabronione (ich rolę najczęściej przejmują listy).
5. Zaawansowane komendy (jak `xargs`) nie są z reguły potrzebne i nie powinny być stosowane.
6. Użycie zagnieżdżonych pętli `for` jest zabronione, chyba, że zadanie jawnie na to zezwala.

2 Podstawowe informacje

1. Podstawowa znajomość systemu plików: umiejętność tworzenia podstawowych ścieżek do plików. Rozróżnienie pomiędzy dwoma głównymi typami plików: katalogami i plikami zwykłymi (regularnymi). Wiedza na temat katalogu domowego użytkownika i jego skrótu (∼).

   Ścieżki do pliku określane są względem pewnego konkretnego miejsca w systemie plików. Zwykle jest to albo katalog główny systemu plików (root, czyli katalog `"/"`) dla ścieżek bezwzględnych lub nasze obecne położenie (*current/working directory*, CWD) dla ścieżek względnych. Pewnym wyjątkiem są pliki dowiązań symbolicznych (o tym w następnym laboratorium).

   Jeśli plik znajduje się w naszym obecnym katalogu, to zwykle wystarcza odwołanie wprost:

   ```
   nazwa_pliku
   ```

   W przypadku, gdy plik znajduje się w innym katalogu trzeba użyć dłuższej ścieżki. Przykłady ścieżek:

   ```
   katalog/plik
   ../katalog/plik
   /kat1/kat2/plik
   /kat1/kat2
   ./plik
   ```

   Zauważmy, że katalog też jest plikiem. Specjalne nazwy plików `"."` i `".."` oznaczają, odpowiednio, aktualny katalog (CWD) oraz katalog nadrzędny (rodzic dla CWD). Zauważmy też, że dla "normalnego" użycia plików nie ma potrzeby używania `"."`. Przykładowo ścieżki:

   ```
   plik
   ./plik
   ./././plik
   ```

   są równoważne. Pliki `"."` i `".."` zostaną szerzej omówione podczas następnego laboratorium.

   Znak tyldy (∼) na początku ścieżki (i tylko na początku!) jest przez `bash` rozwijany do ścieżki bezwzglednej na katalog domowy użytkownika. Dla użytkownika `user` ścieżki:

   ```
   ~/plik
   /home/user/plik
   ```

Są równoważne (o ile katalogiem domowym jest `/home/user`).

Zauważmy jeszcze, że jeśli ostatnim elementem ścieżki jest katalog, to dodanie po nim znaku ukośnika jest opcjonalne. To znaczy poniższe ścieżki są równoważne:

```
sciezka/do/katalogu
sciezka/do/katalogu/
```

Ponadto dublowanie znaku ukośnika nie jest błędem i wszystkie poniższe ścieżki:

```
katalog/plik
katalog//plik
katalog////plik
```

są "poprawne" (w tym sensie, że system operacyjny potrafi je zredukować do tej pierwszej). Jednakże poniższe dwie ścieżki:

```
katalog/plik
katalogplik
```

NIE są równoważne. Wynika z tego uwaga praktyczna: przy budowie ścieżek należy zwracać uwagę czy w odpowiednich miejscach są ukośniki. Zdublowanie ukośnika dla pewności nie jest błędem (chyba, że ukośnik dodano w złym miejscu). Uwaga ta ma znaczenie głównie w przypadku zadań, gdzie budową ścieżek zajmuje się skrypt.

2. Umiejętność wywoływania komend wraz z użyciem argumentów i opcji komend. Normalne wywołanie komendy to ciąg znaków rozdzielonych białymi znakami (zwykle spacją), który występuje na początku linii (ewentualnie poprzedzony białymi znakami). Pierwszy wyraz to komenda do wykonania, a pozostałe wyrazy to argumenty. Przykładowo:

   ```
   komenda arg1 arg2 arg3
   ```

   Nazwy komend są w istocie ścieżkami do plików, które należy wywołać. Jednakże w przypadku wywołania komend normalnie zabronione jest użycie wprost samej nazwy:

   ```
   komenda
   ```

   Takie użycie może spowodować wystąpienie błędu `command not found`. Pierwszym – prostym i zalecanym – rozwiązaniem tego problemu jest wywołanie z użyciem ścieżki, która zawiera więcej niż tylko końcową nazwę pliku:

```
./komenda
sciezka/wzgledna/komenda
/sciezka/bezwzgledna/komenda
```

Drugim wyjściem jest umieszczenie komendy w katalogu, który domyślnie sprawdzany jest w poszukiwaniu komend (jeden z katalogów w treści zmiennej systemowej `PATH`). W ten sposób działają podstawowe komendy (takie jak `echo`, `cat` oraz sam `bash`), ponieważ katalogi, w którym się znajdują (zwykle `/bin` lub `/usr/bin`) znajdują się w zmiennej `PATH`. Dostęp do wspomnianych katalogów z kont studenckich jest zwykle niemożliwy, ale można stworzyć własny katalog skryptów (zwykle `bin` w katalogu domowym użytkownika), a następnie dodać go do zmiennej `PATH`[1].

Opcje to pewne szczególne typy argumentów, zwykle podawane na początku i sterujące daną komendą. Większość komend linuksowych przyjmuje opcje w jednej lub obu z poniższych postaci:

```
-o
--opcja
```

Pierwszy typ to opcja krótka, drugi to opcja długa. Część opcji przyjmuje osobne argumenty (opcjonalne lub wymagane):

```
-oa
--opcja argument
--opcja=argument
```

Komendy, które poprawnie implementują opcje długie pozwalają na używanie skrótów, o ile skróty te są jednoznaczne (w obrębie wszystkich opcji danej komendy):

```
--jakas_opcja
--jakas_opc
--jakas
--j
```

Komendy, które poprawnie implementują opcje krótkie pozwalają na ich łączenie:

```
-a -b -c
-ab -c
-abc
```

Należy jednak uważać, gdy w grę wchodzą opcje z argumentem (opcja `-a` z argumentem `b`, to nie jest to samo co połączenie opcji `-a` i `-b`).

[1] Więcej szczegółów na `http://askubuntu.com/questions/402353/how-to-add-home-username-bin-to-path`.

3. Umiejętność wywoływania skryptów `bash`a, wliczając w to zdolność nadania uprawnień plikom z wykorzystaniem komendy `chmod`. Typowe użycie w celu nadania praw wykonania plikowi ma postać:

   ```
   chmod u+x sciezka_do_pliku
   ```

   Wywołanie skryptu wygląda tak samo jak wywołanie zwykłej komendy:

   ```
   sciezka_do_skryptu lista argumentow
   ```

   Jeśli plik skryptu znajduje się w naszym aktualny katalog roboczym, to użycie sprowadza się do:

   ```
   ./nazwa_skryptu lista argumentow
   ```

   Aby wywołanie skryptu w powyższy sposób się powiodło skrypt musi mieć prawo wykonania. Należy też na początku skryptu (PIERWSZA linijka, przed wszelkimi pustymi liniami i komentarzami) umieścić:

   ```
   #! sciezka/do/interpretera
   ```

   Dla skryptów `bash`owych ścieżką interpretera jest `/bin/bash`. Można też używać starszej powłoki `sh` (ścieżka `/bin/sh`). W późniejszych laboratoriach `bash`a zastąpi `perl` (ścieżka `/usr/bin/perl`). Nie podanie interpretera będzie skutkowało użyciem interpretera domyślnego dla danego użytkownika lub dla danego systemu. W przypadku `bash`a nie jest to duży problem (`bash` jest domyślną powłoką), ale w przypadku `perl`a już tak.

   Drugim sposobem wywołania skryptu jest jawne wywołanie interpretera ze ścieżką skryptu jako argumentem np.:

   ```
   bash sciezka_do_skryptu
   ```

   Takie rozwiązanie jest mniej restrykcyjne (nie trzeba dodawać skryptowi praw wykonania, nie trzeba jawnie podawać interpretera, można używać prostej ścieżki skryptu), ale mniej autonomiczne. Na laboratorium preferujemy pierwsze rozwiązanie.

4. Znajomość zmiennych w powłoce `bash`, wliczając w to przypisanie wartości zmiennych i ich użycie.

   Zmienne w `bash`u można używać (przypisywać i odczytywać wartości) bez uprzedniej deklaracji. Należy pamiętać, że `bash` (jak każda inna komenda) "dziedziczy" początkowe wartości zmiennych po macierzystej komendzie. W związku z tym część zmiennych może od razu mieć jakąś wartość (w szczególności dotyczy to zmiennych systemowych).

   Przypisanie wartości zmiennej `var` ma postać:

```
var=wartosc
```

Proszę zwrócić uwagę na *BRAK* spacji po obu stronach znaku równości. Użycie spacji (czyli zapis `var = wartosc`) spowodowałoby próbę wywołania komendy `var` z argumentami `"="` i `"wartosc"`.

Przypisana wartość jest normalnie traktowana jako tekst. Wykorzystanie wartości zmiennej wymaga użycia znaku dolara. Przykładowo wypisanie wartości zmiennej `var` (użycie komendy `echo`):

```
echo $var
```

Widzimy przy okazji, że `bash` sam w sobie nie zajmuje się wypisywaniem tekstu, lecz wykorzystuje do tego inne komendy (jak `echo`).

Powłoka `bash` skleja teksty wprost bez żadnego operatora. Przykładowo:

```
echo $var1$var2
```

Spowoduje wypisanie wartości obu zmiennych nierozdzielonych żadnym znakiem. Problem pojawia się, gdy chcemy, przykładowo, odróżnić wartość zmiennej `jakasnazwa` od wartości zmiennej `jakas` z doklejonym tekstem `"nazwa"`. Rozwiązaniem jest użycie nawiasów klamrowych. Przykładowo porównajmy:

```
echo ${jakas}nazwa
echo $jakasnazwa
echo ${jakasnazwa}
```

Dwa ostatnie wywołania `echo` są w tym kontekście równoważne.

5. Przekazywanie argumentów do skryptów, ich weryfikacja i dostęp do nich.

   Powłoka `bash` pozwala na dostęp do argumentów skryptu poprzez szereg specjalnych zmiennych. Zmienne od `$1` do `$9` zawierają kolejne 9 argumentów skryptu. Dalsze argumenty wymagają użycia nawiasów klamrowych np. `${10}`[2]. Zmienna `$0` przechowuje zerowy argument, czyli wywoływaną komendę (ściślej ścieżkę, która została użyta do jej wywołania). Zmienna `$#` przechowuje liczbę podanych argumentów (nie licząc `$0`).

   Zmienne `$@` oraz `$*` zwracają wszystkie argumenty skryptu. Pierwsza z nich zwraca argumenty jako listę wyrazów, druga jako pojedynczy wyraz. Obie zmienne powinny być używane w cudzysłowach.

   Wszystkie skrypty powinni sprawdzać otrzymane argumenty (chyba, że skrypt zakłada całkowity brak argumentów). Zwykle wystarcza (w tej kolejności):

[2] Można też posłużyć się komendą wbudowaną `shift`, ale nie jest to zalecane.

(a) sprawdzenie czy liczba otrzymanych argumentów jest co najmniej taka sama jak wymagana liczba argumentów,

(b) sprawdzenie czy argumenty, które mają być plikami są istniejącymi plikami odpowiednich typów.

Sprawdzenie zwykle odbywa się z użyciem konstrukcji `if` i polecenia `test`. W przypadku błędnych argumentów należy wypisać odpowiednią informację i zakończyć skrypt.

6. Znajomość cudzysłowów i apostrofów.

   Pewne znaki (m.in. znak zapytania, gwiazdka, nawiasy klamrowe, nawiasy okrągłe, cudzysłowy, apostrofy, tzw. backtick czy znak odwrotnego ukośnika) mają specjalne znaczenie w pewnych kontekstach i muszą zostać "udosłownione" by zostać użyte wprost. W tym celu można poprzedzić je znakiem ucieczki (\, czyli odwrotny ukośnik lub *backslash*). Do "udosłowniania" dłuższych tekstów służą apostrofy:

   ```
   echo 'doslowny $'
   ```

   Powyższa komenda potraktuje znak dolara jako zwykły znak.

   Domyślnie tekst w `bash`u dzielony jest na wyrazy i ulega interpretacji przez `bash`. W celu "zamknięcia" wielu wyrazów w jeden należy wykorzystać apostrofy lub cudzysłowy:

   ```
   komenda 'z jednym dlugim argumentem ze spacjami'
   komenda "z jednym dlugim argumentem ze spacjami"
   ```

   Jednakże powyższe przykłady NIE są równoważne. Wewnątrz apostrofów wszystkie znaki traktowane są dosłownie. Ponadto wewnątrz apostrofów nie mogą wystąpić inne apostrofy, nawet jeśli są poprzedzone backslashem!

   Wewnątrz cudzysłowów znaki `$`, `` ` `` oraz \ (a także `@` i `*`) zachowują swoje specjalne znaczenie. Ponadto wewnątrz cudzysłowów mogą wystąpić apostrofy (ale są wtedy normalnymi znakami) i inne cudzysłowy (o ile są poprzedzone backslashem!).

   Uproszczone zastosowanie apostrofów i cudzysłowów obrazuje poniższa tabela:

Przykład	Znaczenie
`tekst1 tekst2 tekst3`	Trzy wyrazy. Interpretacja: tak.
`"tekst1 tekst2 tekst3"`	Jeden wyraz. Interpretacja: tak.
`'tekst1 tekst2 tekst3'`	Jeden wyraz. Interpretacja: nie.
`'tekst1' 'tekst2' 'tekst3'`	Trzy wyrazy. Interpretacja: nie.

7. Użycie odwrotnych apostrofów tzw. *backtick* (znak `, zwykle ten sam klawisz na klawiaturze co znak tyldy).

   Pomimo podobieństwa odwrotne apostrofy pełnią zupełnie inną rolę niż zwykłe apostrofy lub cudzysłowy. Normalne wywołanie komendy charakteryzują dwie cechy. Po pierwsze, komendę można wywołać jedynie na początku linii. Ewentualnie można wykonać kilka komend w linii, jeśli oddzielone są one średnikiem. Normalnie nie można jednak wykonać komendy w taki sposób:

   ```
   for var in komenda argumenty
   ```

   W kontekście pętli `for` fragment `komenda argumenty` jest zwykłym dwuwyrazowym tekstem, a nie komendą. Aby wywołać w tym miejscu komendę należy użyć odwrotnych apostrofów:

   ```
   for var in `komenda argumenty`
   ```

   Odwrotne apostrofy powodują więc wywołanie komendy w miejscu ich użycia.

   Drugą cechą normalnej komendy, jest fakt, że standardowe wyjście komendy pojawia się na ekranie (jeśli go nie przekierowaliśmy w inny sposób). Odwrotne apostrofy modyfikują to zachowanie, sprawiając, że standardowe wyjście pojawia się tam, gdzie pierwotnie znajdowała się komenda. Innymi słowy kod:

   ```
   var=`cat file`
   ```

   spowoduje, że do zmiennej `var` zostanie przypisany wynik komendy `cat file`, czyli zawartość pliku `file`.

   Uwaga! W przypadku problemów z uzyskaniem znaku ` na klawiaturze można użyć alternatywnej składni `$(komenda)`.

8. Wykorzystanie komendy `test`.

   Powłoka `bash` nie posiada sama w sobie warunków logicznych. Do ich testowania służy komenda `test`. Komenda ta ma dwie alternatywne postaci:

   ```
   test wyrazenie
   [ wyrazenie ]
   ```

   Formy te są praktycznie równoważne[3].

   Pierwszą uwagą jest fakt, że obie przedstawione powyżej formy są normalnymi komendami i obowiązują przy ich użyciu reguły takie jak spacje pomiędzy poszczególnymi argumentami. Poniższe wywołania są więc błędne:

[3] Istnieje jeszcze konstrukcja `[[ wyrazenie ]]`, ale nie jest ona tym samym co `[ wyrazenie ]`!

```
[2 = 3 ]
[ 2 =3 ]
```

Pierwsze spowoduje błąd "nieznana komenda `[2`". Drugi zaś nie zostanie zrozumiany przez polecenie `test` (nieprawidłowy operator lub operand).

Komenda test posiada szereg operatorów. Najważniejsze to:

- operatory plikowe. Przykładowo `-f file` zwraca prawdę, gdy ścieżka `file` wskazuje na *istniejący* plik *regularny*.
- operatory liczbowe (dla liczb *całkowitych*!). Używana jest w tym celu nieco nieintuicyjna składnia. Przykładowo, `-eq` oznacza równość, `-gt` oznacza większe, zaś `-ge` oznacza większe lub równe.
- operatory tekstowe. Wśród nich najważniejsze to `A = B` i `A != B`, czyli odpowiednio równość i różność łańcuchów tekstowych. Podwójny znak równości, jako równoważność pojedynczego, zwykle jest wspierany, ale nie jest to gwarantowane!

Poszczególne wyrażenia można łączyć za pomocą operatorów logicznych, negacji czy nawiasów, pamiętając jednak o wymaganych spacjach. Przykładowo:

```
[ \( ! -f file1 \) -a \( $var = "Ala" -o $var2 -gt 3 \) ]
```

Opcje `-a` i `-o` oznaczają odpowiednio logiczną koniunkcję (AND) i alternatywę (OR). Działają one jednak nieco inaczej niż w języku `C/C++`. W przypadku ich użycia oba wyrażenia po ich lewej i prawej stronie zostaną wykonane (zwartościowane) niezależnie od wyniku pierwszego z nich. Przykładowo, w takim przypadku:

```
[ komenda1 -o komenda2 ]
```

`komenda2` zostanie wykonana niezależnie od wyniku wykonania `komenda1`. Aby uzyskać efekt znany z języka `C/C++` można wykorzystać konstrukcje `&&` (dla AND) oraz || (dla OR), pamiętając jednak, że są to konstrukcje `bash`a, a nie `test`a:

```
[ -f file1 -a -f file2 ]      # ok
[ -f file1 ] && [ -f file2 ]  # ok, choć nie równoważnie
[ -f file1 && -f file2 ]      # źle
```

Przy okazji: pojedyncza komenda może być użyta jako wartość logiczna, ale należy pamiętać, że konwencja jest odwrotna niż dla `C/C++` tzn. wynik komendy równy 0 oznacza wartość `TRUE` (komenda wykonana pomyślnie), a wartość różna od 0 oznacza `FALSE` (błąd).

9. Użycie konstrukcji warunkowej `if-else`.

   Powłoka `bash` posiada konstrukcję `if-else`, której ogólna składnia jest następująca:

   ```
   if warunek1
   then
       komendy
   elif warunek2
   then
       komendy
   else
       komendy
   fi
   ```

   Sekcje `elif` oraz sekcja `else` są opcjonalne. Słowa `elif`, `then`, `else` i `fi` są traktowane jak komendy, więc jeśli nie "zaczynają" linii, to muszą być poprzedzone średnikiem:

   ```
   if warunek; then komenda1; else komenda2; fi
   ```

   Warunkiem jest dowolne wyrażenie logiczne. Zwykle jest to komenda `test`:

   ```
   if [ $var -gt 3 ]
   then
       echo zmienna var jest większa od 3
   else
       echo zmienna var nie jest większa od 3
   fi
   ```

   Poszczególne sekcje mogą zawierać wiele komend, ale nie mogą być puste.

10. Rozumienie list. Listą jest ciąg wyrazów oddzielonych białym znakiem. Przykładowo, w zmiennej `$@` jest lista argumentów skryptu. Inny przykład utworzenia listy "na bieżąco":

    ```
    var=Ala
    var=$var" ma"
    var=$var" kota"
    ```

    Po zakończeniu zmienna `var` jest listą `"Ala ma kota"`.

11. Zrozumienie pętli `for`. Pętla ta ma następującą składnię:

    ```
    for zmienna in lista
    do
        komendy
    done
    ```

Słowa `do` i `done` muszą być poprzedzone średnikiem jeśli nie są na początku linii (analogicznie jak w przypadku `if-else`).

Pętla `for` wykonuje się tyle razy ile jest wyrazów na podanej liście. Z każdym przebiegiem pętli zmienna `var` przyjmie wartość kolejnego wyrazu z listy. Przykładowo, konstrukcja:

```
zdanie="Ala ma kota"
for var in zdanie; do
    echo $var
done
```

powinna dać w wyniku:

```
Ala
ma
kota
```

Do stworzenia listy wykorzystuje się zwykle:

- komendę `cat`, która wypisuje zawartość plik(ów). Pętla wykona się tyle razy ile jest wyrazów w podanym pliku/plikach.
- komendę `ls`, która listuje pliki z podanego katalogu. Pętla wykona się tyle razy ile jest plików (uwaga na spacje w nazwach plików!).
- komendę `seq`, która wypisuje liczby. Przykładowo

  ```
  for var in `seq 1 10`; do
      echo $var
  done
  ```

  Wykona się 10 razy, bo `seq 1 10` zwraca dziesięć liczb od 1 do 10.

W każdym z powyższych przypadków należy pamiętać o odwrotnych apostrofach, w celu wywołania komendy w "nagłówku" pętli.

3 Przydatne komendy

Poniżej znajduje się lista przydatnych i często używanych komend. Część z nich będziemy używać głównie poza skryptem w celach poruszania się po systemie plików lub przygotowania danych testowych.

- `echo` – wypisuje tekst. Posiada przydatne opcje `-n` oraz `-e`.
- `ls` – listuje zawartość katalogów (końcowe nazwy plików, a nie pełne ścieżki!). Posiada bardzo dużą liczbę opcji, ale tylko niektóre (jak `-l` lub `-1`) będą przydatne w czasie laboratorium.

- `rm`, `rmdir`, `mkdir`, `mv`, `cp` – komendy do, kolejno, 1) usuwania plików (domyślnie nie usuwa katalogów), 2) usuwania (pustych) katalogów, 3) tworzenia katalogów, 4) zmieniania nazw plików (ogólniej: przenoszenia plików) oraz 5) kopiowania plików.
- `touch` – "szturchnięcie" które zmienia czasy pliku takie jak czas ostatniego dostępu lub ostatniej modyfikacji. Ma efekt uboczny w postaci tworzenia pliku regularnego, jeśli wskazany plik nie istnieje. Z tego powodu jest używana do "bezpiecznego" tworzenia plików regularnych.
- `head`, `tail`, `cat`. Komendy wypisujące zawartość pliku. Komenda `head` wypisuje początek pliku, zaś `tail` jego koniec. Komenda `cat` [4] łączy podane pliki przed ich wypisaniem.
- `pwd` – wypisuje wartość CWD jako ścieżkę bezwzględną. Rzadko używana (zwykle z powodu nieznajomości lepszych rozwiązań). Wbrew dosyć powszechnej opinii `pwd` nie przyjmuje żadnych argumentów i podaje tylko i wyłącznie ścieżkę do naszego *aktualnego* położenia w systemie plików.
- `cd` – zmiana aktualnego katalogu (zmienia CWD). Użyta bez argumentów przenosi do katalogu domowego. Zaawansowane użycie pozwala przejść do ostatnio odwiedzonego katalogu.
- `expr`, `let` – podstawowe komendy do wykonywania obliczeń arytmetycznych (np. zwiększanie wartości liczników), jako że `bash` sam w sobie nie posiada operatorów arytmetycznych.

4 Synonimy komend

Niektóre komendy i konstrukcje mają swoje alternatywne formy. O części z nich wspomniano wcześniej.

- Komendy `test` i `[` są praktycznie synonimami. Ta druga wymaga jednak obecności zamykającego nawiasu `]`. Konstrukcja `[[ ]]` jest podobna, ale nie równoważna.
- Konstrukcja `` `komenda` `` może zostać zastąpiona przez `$(komenda)`.
- Zamiast `` `expr wyrazenie` `` można użyć składni `$[wyrazenie]` lub `$((wyrazenie))`. Jest to wygodne, ponieważ komenda `expr` wymaga spacji w odpowiednich miejscach i znaku dolara przy wartościach zmiennych. Alternatywne formy nie mają tych ograniczeń, tak więc poniższe przypisania są równoważne:

```
wynik=`expr $zmienna + 3`
wynik=$[zmienna+3]
wynik=$((zmienna+3))
```

[4] Nie mylić z komendą `cut`!