teksty

Wersja oryginalna: Bash tutorial - pegasus.rutgers.edu/~elflord/unix/bash-tute.html

autor: Donovan Rebbechi (e-mail: elflord@pegasus.rutgers.edu)
tłumaczenie: Łukasz Kowalczyk

Spis treści

Wprowadzenie do tworzenia skryptów dla basł

Prosty skrypt

Do początku strony

Skrypt shellowy to nieco więcej niż lista poleceń do wykonania. Zgodnie z konwencją, skrypt powir od następującej linii:

```
#!/bin/bash
```

Ta linia oznacza, że skrypt powinien być wykonywany przez shell bash niezależnie od tego, jaki she danym momencie. Jest to bardzo ważne, ponieważ składnia rozmaitych shelli może się znacznie róż

Prosty przykład

Oto przykład bardzo prostego skryptu. Służy on do uruchamiania kilku poleceń.

```
#!/bin/bash
echo "Witam. Twoj login to $USER"
echo "Lista plików w bieżącym katalogu, $PWD"
ls   # wypisz listę plików
```

Zauważ, jak wygląda komentarz w czwartej linii. W skrypcie dla bash-a wszystko stojące za znakier ignorowane. Pisząc skrypt powinieneś umieszczać w nim komentarze dla wygody osób, które będą

$USER oraz $PWD to *zmienne*. Są to standardowe zmienne zdefiniowane przez bash-a, więc nie mu definiowane oddzielnie w skrypcie. Podczas wykonywania skryptu nazwy zmiennych poprzedzone z zastępowane przez ich aktualną zawartość. Nazywane jest to *rozwijaniem zmiennych*.

Poniżej znajduje się więcej informacji o zmiennych

Zmienne

Do początku strony

Każdy język programowania potrzebuje zmiennych, w skrypcie bash-a definuje się je w następujący

```
X="hello"
```

a używa się ich w następujący sposób:

```
$X
```

Konkretniej, $X oznacza zawartość zmiennej X. Należy zwrócić uwagę na kilka szczegółów dotycz

- Po obu stronach znaku = nie mogą znajdować się spacje. Na przykład poniższa linia spowodu błędu.

  ```
  X = hello
  ```

- Wprawdzie w przykładach używałem cudzysłowów, ale są one potrzebne tylko, gdy w wartoś znajdują się spacje.

  ```
  X=hello world # błąd
  X="hello world" # OK
  ```

Ten wymóg spowodowany jest tym, że shell interpretuje linię poleceń jako komendę i jej argumenty spacjami. `foo=bar` jest uważane za polecenie. Jeżeli shell będzie musiał zinterpretować linię `foo =` wniosku, że `foo` jest poleceniem. Podobnie, `X=hello world` zostanie zrozumiane przez shell jako p przypisania `X=hello` z dodatkowym argumentem `world`, co nie ma sensu, ponieważ polecenie przyp potrzebuje dodatkowych argumentów.

Apostrofy vs. cudzysłowy

Do początku strony

Zasada jest prosta: wewnątrz cudzysłowów nazwy zmiennych poprzedzone przez $ są zastępowane zawartość, natomiast wewnątrz apostrofów nie. Jeżeli nie zamierzasz odnosić się do zmiennych, użyj ponieważ skutki ich użycia są bardziej przewidywalne.

Przykład

```
#!/bin/bash
echo -n '$USER=' # dzięki opcji -n kursor nie przechodzi do kolejnej linii
echo "$USER"
```

```
echo "\$USER=$USER"  # ten sam efekt, co po pierwszych dwóch liniach
```

Efekty działania skryptu są następujące (zakładając, że twoja nazwa użytkownika to elflord)

```
$USER=elflord

$USER=elflord
```

więc działanie cudzysłowów można ominąć. Cudzysłowy dają większą elastyczność, ale są mniej pr
Stojąc przed wyborem, wybierz raczej apostrofy.

Nazwy zmiennych w cudzysłowach

Do początku strony

Niekiedy należy ujmować nazwy zmiennych w cudzysłowy. Jest to istotne, gdy wartość zmiennej (a
(b) jest pustym ciągiem. Na przykład.

```
#!/bin/bash
X=""
if [ -n $X ]; then      # -n testuje, czy argument nie jest pusty
        echo "Zmienna X nie jest pusta"
fi
```

Działanie tego da następujący efekt:

```
Zmienna X nie jest pusta
```

Dlaczego ? Ponieważ shell zamienia $X na pusty ciąg. Wyrażenie [-n] zwraca prawdę (ponieważ n
argumentu). Poprawny skrypt powinien wyglądać następująco:

```
#!/bin/bash
X=""
if [ -n "$X" ]; then    # -n testuje, czy argument nie jest pusty
        echo "Zmienna X nie jest pusta"
fi
```

W tym przykładzie shell rozwinie wyrażenie do postaci [-n ""], co zwraca fałsz, ponieważ ciąg zaw
cudzysłowach jest pusty.

Jak działa rozwijanie zmiennych

Do początku strony

Poniższy przykład ma przekonać czytelnika, że shell naprawdę rozwija zmienne. Żeby przekonać cz
naprawdę rozwija zmienne .

```
#!/bin/bash
LS="ls"
LS_FLAGS="-al"

$LS $LS_FLAGS $HOME
```

Może to wyglądać tajemniczo. Ostatnia linia jest w istocie poleceniem do wykonania przez shell:

```
ls -al /home/elflord
```

(zakładając, że twoim katalogiem domowym jest /home/elflord). Shell po prostu zastępuje zmienne następnie wykonuje polecenie.

Używanie nawiasów klamrowych do ochrony nazw zmiennych

Do początku strony

Oto potencjalna sytuacja. Załóżmy, że chcesz wypisać na ekranie zawartość zmiennej X i bezpośred "abc". Jak to zrobić ? Spróbujmy:

```
#!/bin/bash
X=ABC
echo "$Xabc"
```

Ekran pozostaje pusty; co się stało ? Shell zrozumiał, że chodzi nam o zawartość zmiennej Xabc, kt została zainicjalizowana. Sposób na obejście problemu jest prosty: nazwę zmiennej należy ująć w n aby oddzielić ja od innych znaków. Poniższy kod daje pożądany rezulat.

```
#!/bin/bash
X=ABC
echo "${X}abc"
```

Instrukcje warunkowe

Do początku strony

Niekiedy należy sprawdzić jakiś warunek. Na przykład, czy ciąg ma zerową długość ? Czy istnieje d dowiązaniem symbolicznym, czy prawdziwym plikiem ? Na początku używamy polecenia `if` , aby warunek. Składnia jest następująca:

```
if warunek
then
        wyrażenie1
        wyrażenie2
        ..........
fi
```

Niekiedy możesz chcieć wykonać inny zestaw poleceń, kiedy test warunku kończy się wynikiem neg to osiągnąć w następujący sposób:

```
if warunek
then
    wyrażenie1
    wyrażenie2
    ..........
else
    wyrażenie3
fi
```

Można też sprawdzać inny warunek, kiedy pierwszy nie jest spełniony. Dozwolona jest dowolna ilo

```
if warunek1
then
        wyrażenie1
```

```
        wyrażenie2
        ..........
elsif warunek2
then
        wyrażenie3
        wyrażenie4
        ........
elsif warunek3
then
        wyrażenie5
        wyrażenie6
        ........


fi
```

Polecenia wewnątrz bloku pomiędzy `if/elsif` ,a następnym `elsif` lub `fi` sa wykonywane, jeżeli warunek jest prawdziwy. W miejscu przeznaczonym na warunek może znaleźć się dowolne polecen komend będzie wykonany tylko, jeżeli to polecenie zwróci kod równy 0 (tzn. skończy się "sukcesem wprowadzeniu do testowania warunków będziemy używali tylko polecenia "test" lub "[ ]".

Polecenie test i operatory

Do początku strony

Do testowania warunków używa sie najczęściej polecenia test, które zwraca prawdę lub fałsz (dokła 0 lub różny od zera) zależnie od tego, czy testowany warunek wypadł pozytywnie czy negatywnie. D następująco:

```
test operand1 operator operand2
```

niektóre testy wymagają tylko jednego operandu (drugiego). Zazwyczaj polecenie test jest zapisywa

```
[ operand1 operator operand2 ]
```

Najwyższy czas na kilka przykładów.

```
#!/bin/bash
X=3
Y=4
empty_string=""
if [ $X -lt $Y ]        # czy $X jest mniejsze niż $Y ?
then
        echo "\$X=${X} jest większe niż \$Y=${Y}"
fi

if [ -n "$pusty_ciag" ]; then
        echo "pusty_ciag nie jest pusty"
fi

if [ -e "${HOME}/.fvwmrc" ]; then               # czy istnieje plik !/.fvwm
        echo "masz plik .fvwmrc, "
        if [ -L "${HOME}/.fvwmrc" ]; then       # czy jest dowiązaniem symb
                echo "który jest dowiązaniem symbolicznym"
        elsif [ -f "${HOME}/.fvwmrc" ]; then    # czy zwykłym plikiem
                echo "który jest zwykłym plikiem"
        fi
else
        echo "nie masz pliku .fvwmrc"
fi
```

Niektóre pułapki

Do początku strony

Polecenie test musi mieć postać "operand1<odstęp>operator<odstęp>operand2" lub operator<odstęp mówiąc inaczej, te odstępy są naprawdę potrzebne, ponieważ pierwszy ciąg bez spacji jest interpretc operator (jeżeli zaczyna się od '-') lub operand (jeżeli zaczyna się od czegoś innego). Na przykład,

```
if [ 1=2 ]; then
        echo "hello"
fi
```

Produkuje zły wynik, tzn. wypisuje "hello", ponieważ widzi operand, ale żadnych operatorów.

Kolejną pułapką może się okazać niezabezpieczanie zmiennych cudzysłowami. Podano już przykład ujmować w cudzysłowy wszystkie parametry testu z opcją `-n`. Poza tym, istnieje mnóstwo dobrych p których należy używać cudzysłowów lub apostrofów w niemal każdej sytuacji. Zapominanie o tym m przyczyną bardzo dziwnych błędów. Oto przykład:

```
#!/bin/bash
X="-n"
Y=""
if [ $X = $Y ] ; then
        echo "X=Y"
fi
```

Wynik działania tego skryptu będzie bardzo mylący, ponieważ shell rozwinie nasze wyrażenie do po

```
[ -n = ]
```

a ciąg "=" ma niezerową długość

Krótki opis operatorów polecenia test

Do początku strony

Poniżej znajduje się krótkie omówienie operatorów polecenia test. Nie jest ono w żadnym razie wyc przypuszczalnie tyko tyle powinieneś pamiętać (pozostałe operatory można znaleźć w opisie basha -

operator	zwraca prawdę, jeżeli...	lic op
-n	operand ma niezerową długość	1
-z	operand ma zerową długość	1
-d	istnieje katalog o nazwie *operand*	1
-f	istnieje plik o nazwie *operand*	1
-eq	operandy są równymi sobie liczbami całkowitymi	2
-neq	przeciwnieństwo -eq	2
=	operandy są jednakowymi ciągami znaków	2
!=	przeciwnieństwo =	2
-lt	*operand1* jest mniejszy niż *operand2* (operandy są liczbami całkowitymi)	2
-gt	*operand1* jest większy niż *operand2* (operandy są liczbami całkowitymi)	2

-ge	*operand1* jest równy lub większy niż *operand2* (operandy są liczbami całkowitymi)	2
-le	*operand1* jest równy lub mniejszy niż *operand2* (operandy są liczbami całkowitymi)	2

Pętle

Do początku strony

Pętle pozwalają na wykonywanie iteracji lub wykonanie tych samych działań na kilku parametrach. dostępne są następujące rodzaje pętli:

- petle for
- pętle while

Pętle for

Do początku strony

Składnię tych pętli najlepiej jest zademonstrować na przykładzie.

```
#!/bin/bash
for X in czerwony zielony niebieski
do
        echo $X
done
```

Pętla for wykonuje polecenia zawarte wewnątrz pętli na parametrach rozdzielonych spacjami. Zauw parametry zaierają w sobie spacje, muszą być ujęte w cudzysłowy (apostrofy). Oto przykład:

```
#!/bin/bash
kolor1="czerwony"
kolor2="jasny błękit"
kolor3="ciemna zieleń"
for X in "$kolor1" $kolor2" $kolor3"
do
        echo $X
done
```

Czy zgadłbyś, co się stanie, gdy zapomnimy o cudzysłowach w pętli for ? Powinieneś używać cudzy jesteś pewien, że parametry nie zawierają spacji.

Znaki globalne w pętlach

Do początku strony

Wszystkie parametry zawierające * są zastępowane listą plików, które pasują do wzorca. W szcegól gwiazdka (*) jest zastępowana przez listę wszystkich plików w bieżącym katalogu (z wyjątkiem plił nazwy zaczynją się od kropki ".")

```
echo *
```

wypisuje nazwy wszystkich plików i katalogów w bieżącym katalogu

```
echo *.jpg
```

wypisuje wszystkie pliki jpeg

```
echo ${HOME}/public_html/*.jpg
```

wypisuje nazwy wszystkich plików jpeg w twoim katalogu public_html

Ta właściwość jest bardzo użyteczna w wykonywaniu działań na wszystkich plikach w katalogu, szc
pętlach for. Na przykład:

```
#!/bin/bash
for X in *.html
do
                grep -L '<UL>' "$X"
done
```

Pętle while

Do początku strony

Pętle while działają tak długo, jak długo prawdziwy jest dany warunek. Na przykład:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
        echo $X
        X=$((X+1))
done
```

Nasuwa się naturalne pytanie: dlaczego bash nie pozwala na stosowanie pętli for w stylu języka C

for (X=1,X<10; X++)

powód jest prosty: skrypty basha są interpretowane i z tego powodu dosyć powolne. Dlatego odradz
silnie wykorzystujące iterację.

Podstawianie poleceń

Do początku strony

Podstawianie poleceń jest bardzo wygodnym mechanizmem basha. Pozwala mianowicie na pobrani
polecenie wyprowadza na ekran i traktowanie ich, jak gdyby zostały wprowadzone z klawiatury. Na
chcesz, aby zmiennej X została przypisana treść, którą wyprodukowało jakieś polecenie, sposobem j
poleceń.

Są dwa sposoby podstawiania poleceń: rozwijanie zawartości nawiasów oraz ujmowanie polecenia
apostrofy.

Rozwijanie zawartości nawiasów działa następująco: `$(polecenia)` zostaje zamienione przez to, c
polecenia. Nawiasy mogą być zagnieżdżane, tak więc *polecenia* również mogą zawierać nawiasy.

Rozwijanie we wstecznych apostrofach zamienia `` `polecenia` `` treścią wyprowadzoną przez *polece*

Przykład:

```
#!/bin/bash
pliki="$(ls )"
pliki_html=`ls public_html`
echo $pliki
echo $pliki_html
X=`expr 3 \* 2 + 4` # expr oblicza wyrażenia arytmetyczne
echo $X
```

Zauważ, że chociaż ls wyprowadza listę plików ze znakami nowej linii, zmienne nie zawierają tych basha nie mogą zawierać znaków nowej linii.

Wygodniejsza w użyciu jest metoda z nawiasami, ponieważ łatwo jest je zagnieżdżać. Poza tym, ich większość wersji basha (wszystkie zgodne ze standardem POSIX). Metoda ze wstecznym apostrofa czytelna i dostępna nawet w bardzo prymitywnych shellach (każda wersja /bin/sh).