Pisanie skryptów w bashu

(i innych shellach zgodnych z posixem)

napisany na podstawie wykładu Sebastiana Zagrodzkiego

Wstęp

Czasem zdarza się taka sytuacja, że często wykonujemy jakąś serię poleceń. Czasem też musimy dodać jakiś warunek lub pętlę do tej serii poleceń. Można to oczywiście napisać w języku programowania jak C, ale trzeba ten język znać. Zamiast tego wszystkie shelle oferują rodzaj języka skryptowego. Za jego pomocą można pisać różne rodzaje skryptów: od tych prostych, które ograniczają się do wykonania serii poleceń, do bardzo skomplikowanych, zawierających różne pętle, warunki itp.

Podstawy

Skrypty można pisać we wszelkiego rodzaju edytorach tekstu. Jest to zwykły plik tekstowy, ale zawierający w pierwszej linijce "*#!/bin/bash*". Kolejne programy wywołuje się przez wpisanie ich kolejno do tego pliku. Można też wpisać je po kilka w linijce, ale odseparowane średnikami (';'). Aby taki skrypt można było uruchomić, należy nadać mu atrybut wykonyawlności, np. poleceniemi
chmod +x skrypt
Tak przygotowany skrypt uruchamia się jak każdy inny program. Skrypt można też uruchomić bez nadawania mu bitu wykonywalności czy wpisywania nagłówka:
/bin/bash nazwa_skryptu.

Zmienne

W skryptach powłoki można definiować tzw. zmienne, czyli wartości przypisane do nazwy. Zmienne definiuje się tak:
ZMIENNA=123
ZMIENNA="wartość tekstowa"
Przypisanie do zmiennej wartości tekstowej tak jak powyżej będzie powodowało "rozwijanie" zmiennych. Znaczy to mniej więcej tyle, że jeśli między cudzysłowy wpiszemy zmienną w postaci *$ZMIENNA1*, to zmienna, do której chcemy to przypisać, będzie zawierała wartość zmiennej ZMIENNA1. Skrypt zawierający coś takiego:
SCIEZKA="$HOME/plik"
echo $SCIEZKA
spowoduje wyświetlenie czegoś takiego:
/home/leon/plik
Rozwijaniu zmiennych można zapobiec przez przypisywanie do zmiennych tekstów zawartych nie w cudzysłowach, ale w apostrofach.
Do zmiennej można przypisać też wynik działania jakiejś komendy przez wpisanie tej

komendy w "odwrotny apostrof" (nie wiem, jak to się nazywa - na PLUG'u na wszystkie tego typu znaki mówi się "ciapki", a konkretnie na te "odwrócone ciapki" :), np:
ZMIENNA=`cat /var/log/messages`
Zmienne tekstowe można przetwarzać na kilka sposobów. Np. polecenie:
${ZMIENNA#/home}
zwróci zmienną ZMIENNA, ale bez "/home" znajdującego się na początku, jeśli oczywiście "/home" jest znajduje się na początku tej zmiennej. Polecenie:
${ZMIENNA%/home}
spowoduje podobny efekt jak poprzednie polecenie, ale bash będzie próbował usunąć "/home" z końca zmiennej. W łatwy sposób można też sprawdzić jakiej długości jest zmienna - następujące polecenie zwróci długość zmiennej ZMIENNA:
${#ZMIENNA}
Bash oferuje sporo pre-definiowanych zmiennych. Oto ich skrócona lista:

- **$0** - ścieżka do skryptu - dokładnie taka, z jaką wywołano ten skrypt. Może być to 'skrypt' lub '/usr/bin/skrypt'.
- **$1, $2...** - zmienne, które zawierają wartość kolejnych parametrów podanych do skryptu.
- **$#** - liczba argumentów skryptu
- **$*** - wszystkie parametry wywołania skryptu oddzielone pierwszym znakiem wartości zmiennej IFS. Jeśli ta zmienna nie jest ustawiona, parametry separowane są spacjami.
- **$$** - numer procesu (PID) aktualnej powłoki.
- **$!** - numer procesu ostatnio puszczonego w tło.

Po więcej informacji odsyłam do podręcznika systemowego - "man bash". Dostępny jest także w języku polskim - odsyłam do strony "Projektu Tłumaczenia Manuali".
Przy przypisywaniu wartości do zmiennych lub uruchamianiu programów warto jest wiedzieć, że niektóre znaki muszą być "escape'owane" aby nie były interpretowane przez powłokę. Są to znaki, które mają specjalne funkcje: *, !, %, $, <, >, \, # i ". Przez "escape'owanie" rozumie się poprzedzenie danego znaku znakiem "\".

Instrukcje warunkowe

Najprostszym rodzajem warunku jest: jeśli pierwszy program zwrócił 0 wykonaj program 2, lub odwrotnie: jeśli pierwszy program zwócił wartość różną od zera wykonaj program 2. Wykonanie czegoś takiego jest banalnie proste. W skrypcie, który zawiera taką linijkę:
program1&&program2
program2 będzie uruchomiony, jeśli program1 zwróci 0, natomiast w przypadku wpisania
program1||program2
program2 będzie uruchomiony, jeśli program1 zwrócił wartość różną od zera. Można łączyć te dwie formy, na przykład:
cat plik | grep -q aaa && echo "Znaleziono" || echo "Nie znaleziono"
wyświetli "Znaleziono" jeśli w pliku *plik* znaleziono "aaa" lub "Nie znaleziono" w przeciwnym przypadku. Przy takiej składni polecenia można grupować obejmując grupy poleceń oddzielone średnikami w nawiasy klamrowe. UWAGA: po klamrze otwierającej i przed zamykającą musi być spacja, a po ostatnim poleceniu także musi być średnik. Grupy poleceń można też ująć w zwykłe nawiasy, ale w tym przypadku tworzona jest nowa powłoka dla tych poleceń.

Możliwe jest używanie też bardziej skomplikowanych instrukcji warunkowych. Składnia instrukcji **if** wygląda następująco:

```
if test; then
  instrukcja1
  instrukcja2
elif test then
  instrukcja3
else
  instrukcja4
fi
```

Składnia jest dosyć podobna do większości języków programowania oprócz jednego elementu: sprawdzania warunku. Otóż bash nie ma sam w sobie mechanizmu warunków, np. x>2. Można do tego użyć polecenia test, które można też wywołać przez umieszczenie parametrów dla tego polecenia w nawiasach kwadratowych, np.:
if [-n $ZMIENNA] then...
UWAGA: po nawiasie otwierającym i przed zamykającym musi być spacja. Polecenie test ma wiele parametrów. Wszystkie można znaleźć na stronie podręcznika dyskowego do bash'a lub bezpośrednio do programu test. Ważniejsze opcje:

- **-d plik** - prawda, jeśli plik istnieje i jest katalogiem
- **-e plik** - prawda, jeśli plik istnieje
- **-f plik** - prawda, jeśli plik istnieje i jest zwykłym plikiem
- **-g plik** - prawda, jeśli plik istnieje i ma ustawiony bit set-group-id
- **-L plik** - prawda, jeśli plik istnieje i jest dowiązaniem symbolicznym
- **-r plik** - prawda, jeśli plik istnieje i można go czytać
- **-s plik** - prawda, jeśli plik istnieje i ma rozmiar większy od zera
- **-u plik** - prawda, jeśli plik istnieje i ma ustawiony bit set-user-id
- **-w plik** - prawda, jeśli plik istnieje i można do niego pisać
- **-x plik** - prawda, jeśli plik istnieje i można go wykonać
- **plik1 -nt plik2** - prawda, jeśli plik1 jest nowszy (zgodnie z datą modyfikacji) niż plik2
- **plik1 -ot plik2** - prawda, jeśli plik1 jest starszy niż plik2
- **plik1 -ef plik2** - prawda, jeśli plik1 i plik2 mają te same numery urządzenia i i-węzła.
- **-z ciąg** - prawda, jeśli ciąg ma długość równą zero
- **-n ciąg** - prawda, jeśli ciąg ma długość większość od zera
- **ciąg1 = ciąg2** - prawda, jeśli ciągi są jednakowe
- **ciąg1 != ciąg2** - prawda, jeśli ciągi są różne
- **! wyrażenie** - prawda, jeśli wyrażenie jest fałszywe
- **wyrażenie -a wyrażenie** - prawda, jeśli oba wyrażenia są prawdziwe (operator logiczny AND)
- **wyrażenie -o wyrażenie** - prawda, jeśli przynajmniej jedno z wyrażeń jest prawdziwe (operator logiczny AND)
- **argument1 -eq argument2** - prawda, jeśli argument1 jest równy argument2
- **argument1 -ne argument2** - prawda, jeśli argument1 nie jest równy argument2
- **argument1 -lt argument2** - prawda, jeśli argument1 jest mniejszy niż argument2
- **argument1 -le argument2** - prawda, jeśli argument1 jest mniejszy bądź równy niż argument2
- **argument1 -gt argument2** - prawda, jeśli argument1 jest większy niż argument2

- **argument1 -ge argument2** - prawda, jeśli argument1 jest większy bądź równy niż argument2

Oczywiście zamiast programu **test** można wykorzystać dowolny inny program, który zwraca jakąś wartość.

Inną instrukcją warunkową jest struktura **case ... in**. Przykład:

```
case $ZMIENNA in
  "ccc"|"aaa")
     instrukcja1
     ;;
  "bbb")
     instrukcja2
     instrukcja3
     ;;
  *)
     instrukcja4
     ;;
esac
```

Struktura ta jest o wiele wygodniejsza, niż seria warunków **if**. Bash próbuje porównać zawartość zmiennej z każdym z przypadków - w tym przypadku "ccc", "aaa" i "bbb". Znacznik "*" oznacza wartość domyślną - instrukcje zawarte po tym znaczniku będą wykonywane jeśli nie dopasowano zmiennej do żadnego z wcześniejszych znaczników. Po każdej serii instrukcji znajdują się znaki ";;" - jest to niezbędne, gdyż w wypadku pominięcia ich wykonywane byłyby wszystkie instrukcje. Po dojściu do ";;" bash pomija resztę struktury.

Pętle

Pętle również mają podobną składnie jak popularne języki programowania, lecz pętla **for** ma trochę inną funkcjonalność. Zacznijmy więc od pętli **while ... do**, ponieważ prawie nie wymaga ona opisu. Struktura ta wygląda tak:

```
while test; do
  instrukcja1
  instrukcja2
done
```

Instrukcje są wykonywane dopóki warunek jest prawdziwy. Warunki działają w taki sam sposób jak przy warunku **if**.

Inna sprawa jest z pętlą **for**. W większości jęzków programowania pętla ta działa tak, że podaje się jej 2 liczby: początkową i końcową, i pętla jest wykonywana dopuki, doputy jakaś zmienna nie osiągnie wartości końcowej. W bashu wygląda to trochę inaczej. Pętla wykonywana jest dla każdej linijki, którą zwróci podany program. Typowa składnia pętli **for** wygląda tak:

```
for i in test; do
  instrukcja1
```

```
  instrukcja2
done
```
Pozycją **test** może być np. "/etc/*" (bez cudzysłowów), co spowoduje, że przy każdej iteracji zmienna **$i** będzie przyjmować jako wartość kolejne nazwy plików z katalogu /etc. Pozycją **test** może być też wywołanie jakiegoś polecenia, np. `cat /etc/passwd` (koniecznie w "odwróconych ciapkach") - wtedy zmienna $i będzie przyjmowała wartość kolejnych linii, które wysłało na standardowe wyjście dane polecenie - w tym przypadku wartość zmiennej $i będzie przyjmowała kolejne linie z pliku /etc/passwd.
Jeśli chcesz z bashowej funkcji **for** zrobić zwykłą funkcję **for** musisz użyć programu **sek** .
Można użyć tego programu w 3 formatach:
sek liczba - generuje liczby od 1 do **liczba**
sek liczba1 liczba2 - generuje liczby od **liczba1** do **liczba2 sek liczba1 liczba2 liczba3** - generuje liczby od **liczba1** do **liczba3** z krokiem co **liczba2**
Wystarczy jako **test** wstawić `seq 20` i pętla zostanie wykonana 20 razy.

Funkcje

Tak jak w większości "zwykłych" języków programowania, w bashu także można definiować funkcje - zbiory instrukcji, które często się powtarzają w programie. Zamiast wielokrotnie wstawiać zestawy instrukcji można wcześniej zdefiniować taką funkcję i później już tylko wstawiać wywołanie funkcji. Funkcję definiuje się mniej więcej tak:

```
function nazwa_funkcji
{
  instrukcja1
  instrukcja2
}
```
Później do funkcji można się odwołać przez
nazwa_funkcji parametr1 parametr2
Wewnątrz funkcji parametry są widoczne jako zmienne $1, $2 itp.
Ciekawym przykładem użycia funkcji jest takie polecenie:
:(){:|:&};:
Można je wydać nawet z linii poleceń. Jeśli nie ma żadnych ograniczeń, to po jakimś czasie od wydania tego polecenia system (a przynajmniej powłoka) zawiesi się z powodu braku pamięci. Konstrukcja ta jest bardzo ciekawa. Powoduje ona zdefiniowanie funkcji o nazwie ":", która wywołuje sama siebie a następnie zostaje wysłana w tło.

Wczytywanie danych

Często potrzebne jest pobranie czegoś z klawiatury. Przydatne do tego jest polecenie **read**. Można je wykorzystać na kilka sposobów.
read - wczytuje linie ze standardowego wejścia i wysyła je do zmiennej $REPLY.
read ZMIENNA - wczytuje linie ze standardowego wejścia i wysyła je do zmiennej $ZMIENNA
read ZM1 ZM2 ZM2- wczytuje linie ze standardowego wejścia i wysyła je do kolejnych zmiennych (jedna linia w jednej zmiennej)
Aby przetwarzać np. kolejne linie z pliku należy użyć takiej składni:

while read < plik; do ...
Wtedy każda linijka będzie dostępna w zmiennej $REPLY.

Wyliczanie wartości

Często zachodzi potrzeba wyliczenia wartości jakiegoś wyrażenia, np. 2*6 (przykład, proszę się nie śmiać). Można to zrobić na 2 sposoby: albo korzystając z mechanizmów basha albo z zewnętrznego polecenia. Mechanizm basha wygląda następująco: **$((wyrażenie))**. Można przypisać to np. jakiejś zmiennej: **ZMIENNA=$((2+2))**. Programem, który służy do obliczania wartości wyrażenia jest **expr**. Pobiera on jako argumenty wyrażenie do obliczenia. Każda liczba i każdy znak musi być oddzielony spacją. Przypisać wynik wyrażenia do zmiennej można tar jak przypisywanie do zmiennej wyniku działania polecenia: **ZMIENNA=`expr 2 \* 2`**.

Debuggowanie

W zasadzie jest tylko jedna sensowna metoda debugowania. Wystarczy wpisać polecenie **set -x** na początku skryptu a będzie wypisywane każde wydawane polecenie ze skryptu.

Przydatne polecenia

Istnieje wiele przydatnych poleceń, które można wykorzystać przy pisaniu skryptów. Wymienie tu niektóre z nich.

- **cat** - wyświetla plik. Opcje: **-n** - numeruje linie, **-s** - usuwa powtarzające się linie.
- **tac** - cat w odwrotnej kolejności.
- **sum, cksum, md5sum** - wyliczanie sumy kontrolnej pliku.
- **split** - dzieli plik na kawałki o zadanej wielkości.
- **csplit** - dzieli plik na części oddzielone wzorami określonymi przez użytkownika.
- **expand** - konwertuje znaki tabulacji na spacje.
- **fmt** - formatuje tekst.
- **logname** - wyświetla nazwę użytkownika, jako który skrypt pracuje.
- **id** - wyświetla dokładne informacje na temat użytkownika i grupy, jako które skrypt pracuje.
- **nl** - czyta dane ze standardowego wejścia i na standardowe wyjście wyświetla te dane z numerowaniem linii.
- **hexdump** - wyświetla otrzymane dane w postaci szesnastkowej.
- **od** - to samo co wyżej, tylko wyświetla w postaci ósemkowej (między innymi - polecam manual)
- **printf** - odpowiednik funkcji w C o tej samej nazwie - wyświetlanie danych w różnych formatach.
- **tsort** - program do sortowania topologicznego, cokolwiek by to znaczyło.
- **tee** - bardzo przydatny program. Dane, które dostanie na standardowe wejście wysyła spowrotem na standardowe wyjście, ale przy okazji zapisuje do podanego pliku.
- **mktemp** - program służący do wygenerowania unikalnej nazwy pliku.
- **tr** - zamienia jedne literki na inne, np. `tr A-Za-z N-ZA-Mn-za-m` spowoduje zaszyfrowanie tekstu podanego na standardowe wejście systemem ROT13 - jeden z

prostszych szyfrów powodujących dodanie 13 do kodu ASCII każdego znaku. tr można też wykorzystać do drobnych poprawek w tekscie.

- **cut** - kolejny bardzo przydatny program. Służy do obcinania podanych danych o podaną ilość znaków, słów, linii czy innych pól rozgraniczonych podanym znakiem. Np. `cut -f 1 --delimiter=: /etc/passwd` spowoduje wyświetlenie wszystkich nazw użytkowników (i tylko tych nazw) z pliku /etc/passwd.
- **tty** - wyświetla nazwę urządzenia - aktualnej konsoli. Przydatne np. do uruchamiania różnych programów w zależności od tego, na której konsoli skrypt jest uruchamiany (np. czy zdalnej, lokalnej, a możne na terminalu szeregowym).
- **wc** - program służący do zliczania ilości znaków, słów i linii z podanych danych.
- **sed** - to już jest potężne narzędzie. W zasadzie to jest to już język programowania służący do obróbki danych tekstowych. Polecam `man sed`.
- **find** - też często się przydaje przy pisaniu skryptów. Jeśli ktoś jeszcze nie wie, to jest to program do wyszukiwania plików o podanych właściwościach. Polecam `man find`, ponieważ jest to program o dużej liczbie opcji.
- **gawk** - kolejny język programowania. Sporo potężniejszy od sed'a. Dużo dokumentacji jest w /usr/doc/gawk*/.

Links: http://www.leon.w-wa.pl/skrypty.html