AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Język skryptowy Linux Bash

Maciej Wielgosz

Wydział Informatyki, Elektroniki i Telekomunikacji

2015, semestr zimowy

Czym jest powłoka?

Powłoka (ang. *shell*)

Program będący pośrednikiem pomiędzy systemem operacyjnym (lub aplikacją) a użytkownikiem. Przyjmuje ona polecenia i wyświetla wyniki działania programów.

Wyróżnić można:

- powłoki tekstowe, np. 4DOS, bash, cmd.exe,
- powłoki graficzne, np. Eksplorator, Nautilus, Unity.

Bash

Bash (ang. *Bourne-Again Shell*):

- wywodzi się od powłoki Bourne'a (sh), jednej z pierwszych powłok systemu UNIX,
- wprowadza idee zawarte w innych powłokach, takich jak ksh i csh,
- domyślna powłoka dla większości dystrybucji systemu GNU/Linux oraz w systemie Mac OS X,
- większość skryptów przeznaczonych dla powłoki sh działa bez zmian w powłoce bash.

Inne powłoki

Wśród innych powłok uniksowych wyróżnić można:

- csh – *C shell*, powłoka o składni wzorowanej na języku C,
- tcsh – ulepszona wersja csh, jest domyślną powłoką we FreeBSD, była także domyślną powłoką wcześniejszych wersji Mac OS X,
- ksh – *Korn shell* – całkowicie kompatybilna wstecz z sh, zawiera wiele elementów zaczerpniętych z csh,
- zsh – *Z shell* – najbardziej przypomina ksh, ale zawiera dalsze ulepszenia.

Geneza bash

Bash był oryginalnie zaprojektowany jako interfejs do komunikacji z użytkownikami, a większe możliwości skryptowe zostały dołożone później. Fakt ten ma duży wpływ na składnię języka.

Podstawowe cechy bash

- bash jest case-sensitive, tzn. wielkość liter ma znaczenie (zmienne `$data` i `$DATA` to dwie różne zmienne),
- białe znaki mają znaczenie – ich niewłaściwe użycie może być przyczyną niedziałania „poprawnego” skryptu,
- polecenia działają dokładnie tak samo wewnątrz skryptu, jak działałyby w linii poleceń,
- aby skrypt bash był wykonywalny potrzebne są prawa do uruchomienia pliku,
- skrypty są uruchamiane jako osobny proces, tzn. najpierw jest tworzony proces i w jego ramach jest uruchamiany skrypt.

Program a proces

Program

Plik wykonywalny, czyli taki, który może być uruchomiony w reakcji na żądanie użytkownika – wpisanie komendy w linii poleceń lub kliknięcie ikony w interfejsie graficznym.

Proces

Działająca instancja danego programu. W systemie może działać równocześnie wiele procesów, niektóre z nich będące instancjami tego samego programu. Każdy proces posiada unikalny identyfikator (PID). Procesy działają niezależnie od siebie, mogą się jednak między sobą komunikować.

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje

Skrypty

Skrypt

Program napisany w języku skryptowym, wykonywany wewnątrz innej aplikacji. Często są to języki interpretowane, stworzone z myślą o interakcji z użytkownikiem.

Oprócz powłok uniksowych skrypty znajdują zastosowanie także m.in. w pakietach biurowych, programach do obróbki i renderowania grafiki, serwerach WWW czy grach komputerowych.

Skrypty bash

Shebang (hashbang)

Sekwencja znaków `#!` stosowana na początku pliku i umożliwiająca uruchomienie go jako skryptu. W systemach uniksowych służy on do wskazania interpretera, który ma być użyty do wykonania zawartego w dalszej części pliku kodu.

Każdy skrypt bash rozpoczyna się od shebang oraz ścieżki do programu bash, najczęściej: `#!/bin/bash`

```
$ which bash
/bin/bash
```

Formatowanie w skryptach

Formatowanie odgrywa znaczącą rolę w wielu miejscach bashowych skryptów. Z tego też powodu analiza białych znaków jest często dobrym punktem startowym w przypadku niedziałającego skryptu.

Na białe znaki warto zwracać uwagę zwłaszcza:

- w sekwencji shebang (tu dodatkowy warunek – musi być w pierwszej linii skryptu),
- przy deklaracji zmiennych,
- w instrukcjach warunkowych.

Hello World

```
$ string="Hello World"
$ echo $string
Hello World
```

Ważny jest brak odstępu pomiędzy wyrażeniami. Poniższy przykład nie zadziała:

```
$ string = "Hello World"
```

Błąd wynika z faktu interpretowania przez bash tej linii jako komendy: nazwy programu i jego argumentów. Jeśli wstawione zostaną odstępy, dla basha oznacza to „uruchom program `string` z argumentami `=` i `Hello World`”.

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje

Pierwszy skrypt

```
1 #!/bin/bash
2 string="Hello World"
3 echo $string
```

Plik zawierający ten skrypt może być zapisany np. jako *hello.sh*.

Rozszerzenie *.sh* nie jest konieczne, ale przyjęto w ten sposób oznaczać skrypty powłok.

Uruchamianie skryptów

Aby skrypt mógł być wykonany, plik który go zawiera musi mieć prawa do uruchomienia. Można to osiągnąć np. wydając polecenie:

```
$ chmod u+x hello.sh
```

Uruchamianie skryptu (jeśli nie jest wpisany w `$PATH`):

```
$ ./hello.sh
```

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
  - Deklaracja, użycie i zasięg zmiennych
  - Zmienne specjalne i środowiskowe
  - Podstawianie komend
  - Eksport zmiennych
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje

Zapis i odczyt zmiennych

Zmienne służą do przechowywania informacji. Mogą one być zapisywane i odczytywane:

- zapis zmiennej realizowany jest **bez** `$`:

```
$ var_to_save="value"
```

- odczytując zmienną poprzedza się ją `$`:

```
$ echo $var_to_read
```

Konwencje nazewnictwa

- przyjęta jest konwencja zapisu nazw zmiennych `malymi_literami`, nie jest to jednak wymagane,
- `WIELKIE_LITERY` zwykle używane są do oznaczania zmiennych środowiskowych,
- nazwy zmiennych powinny być znaczące, np. `total` zamiast `t` – ułatwia to analizę i późniejszą modyfikację skryptu.

Zasięg (widoczność) zmiennych

- deklaracja zmiennej może być umieszczona gdziekolwiek w skrypcie lub linii poleceń, tak długo, jak jest to przed jej pierwszym użyciem. Jest ona podmieniana na wartość przed wykonaniem danego polecenia,
- zmienne domyślnie są widoczne w zakresie w jakim zostały stworzone (proces). Po wyeksportowaniu mogą być dostępne także w innych procesach.

Ciągi znaków a zmienne

- wartość zmiennej zawierająca białe znaki musi być podana w cudzysłowie bądź apostrofach,
- ' (apostrof) oznacza traktowanie ciągu znaków jako literalnej całości (nie pozwala na podmianę zmiennych w jej zawartości),
- " (cudzysłów) pozwala na podmianę zmiennych w zawartości ciągu.

Ciągi znaków a zmienne

```
$ hello='Hello World'
$ echo $hello
Hello World

$ more_hello="More $hello"
$ echo $more_hello
More Hello World

$ even_more_hello='Even $more_hello'
$ echo $even_more_hello
Even $more_hello
```

Zmienne wiersza poleceń

Standardowo w skryptach bash dostępny jest zestaw zmiennych środowiskowych powłoki, parametrów pozycyjnych oraz parametrów specjalnych.

Przykładowo, argumenty wiersza poleceń, z jakimi został uruchomiony skrypt (parametry pozycyjne) można pobrać w ciele skryptu jako `$1`, `$2` itd.:

Przykład skryptu do kopiowania plików:

```
1 #!/bin/bash
2 cp $1 $2
```

Przykładowe dostępne zmienne

- `$0` – nazwa wykonywanego skryptu,
- `$1` – `$9` – pierwszych 9 argumentów linii poleceń,
- `$#` – ilość argumentów przekazana do skryptu,
- `$@` – wszystkie argumenty przekazane do skryptu,
- `$?` – kod wyjścia ostatnio zakończonego procesu,
- `$$` – PID (process ID) bieżącego procesu,
- `$USER` – nazwa użytkownika wykonującego skrypt,
- `$HOSTNAME` – nazwa hosta (komputera) na którym wykonywany jest skrypt,
- `$RANDOM` – zmienna, która zwraca nową liczbę losową za każdym razem, gdy jest odczytana.

Zapamiętywanie wyników poleceń

Mechanizm podstawiania komend pozwala na zapisanie do zmiennej wyniku działania innej komendy bądź programu, jeżeli wynik ten normalnie zostałby wyświetlony na ekranie.

Aby to zrobić, umieszczamy polecenie wewnątrz okrągłych nawiasów, poprzedzonych znakiem `$`.

Zapamiętywanie wyników poleceń

```
1 #!/bin/bash
2 count=$( ls /etc | wc -l )
3 echo There are $count entries in /etc
```

count_etc.sh

```
$ ./count_etc.sh
There are 246 entries in /etc
```

Eksport zmiennych

Standardowo zmienne są widoczne w zakresie (procesie) w jakim zostały stworzone. Czasami jednak z wnętrza skryptu zachodzi potrzeba uruchomienia innego skryptu czy programu, który z zapisanej przez nas zmiennej będzie korzystał. W takiej sytuacji należy wyeksportować zmienną (uczynić ja globalną).

```
$ export var
```

Nazwa eksportowanej zmiennej **nie jest** poprzedzona znakiem `$`.

Eksport zmiennych

```
1  #!/bin/bash
2  # demonstracja zakresu zmiennych #1
3  var1=foo
4  var2=bar
5  # weryfikacja ustawionych wartości
6  echo $0 :: var1 : $var1, var2 : $var2
7  export var1
8  ./var_scope_2.sh
9  # weryfikacja ustawionych wartości
10 echo $0 :: var1 : $var1, var2 : $var2
```

var_scope_1.sh

Eksport zmiennych

```
1 #!/bin/bash
2 # demonstracja zakresu zmiennych #2
3 # weryfikacja ustawionych wartości
4 echo $0 :: var1 : $var1, var2 : $var2
5 # zmiana wartości
6 var1=flip
7 var2=flap
8 echo $0 :: var1 : $var1, var2 : $var2
```

var_scope_2.sh

Eksport zmiennych

Wynik uruchomienia skryptu:

```
$ ./var_scope_1.sh
./var_scope_1.sh :: var1 : foo, var2 : bar
./var_scope_2.sh :: var1 : foo, var2 :
./var_scope_2.sh :: var1 : flip, var2 : flap
./var_scope_1.sh :: var1 : foo, var2 : bar
```

Eksport zmiennych

Wynik ten może wydawać się nieoczekiwany. Wynika to jednak z faktu, że w momencie uruchamiania nowego procesu (w tym przypadku `./var_scope_2.sh` w pliku *var_scope_1.sh*) wszystkie zmienne globalne są kopiowane i przekazywane do nowego procesu. Tak więc zmienna `var1` w skrypcie *var_scope_1.sh* nie jest tym samym, co zmienna `var1` w skrypcie *var_scope_2.sh*, choć ma ona taką samą nazwę.

Eksport zmiennych

Eksportowanie zmiennych działa tylko w jedną stronę. Proces nadrzędny może przekazać zmienne do procesu potomnego, ale nic co proces podrzędny z nimi zrobi nie będzie miało wpływu na ich wartości w procesie nadrzędnym.

Wprowadzenie do pisania skryptów

Polecenie read

Komenda `read` odczytuje dane ze standardowego wejścia. Jako że zmienne w bash standardowo nie są typowane, wczytany ciąg może być traktowany zarówno jako ciąg znaków, jak i liczba.

```
1 #!/bin/bash
2 # zapytanie o imię
3 echo Hello, who am I talking to?
4 read name
5 echo It\'s nice to meet you $name
```

name.sh

Opcje polecenia read

Zachowanie polecenia read można modyfikować przy pomocy przekazywanych mu opcji. Pełna lista dostępna jest po użyciu polecenia `man`:

```
$ man read
```

Często używane opcje:

- `-p` – pozwala na zdefiniowanie znaku zachęty (tzw. prompt),
- `-s` – definiuje wejście jako ciche, tzn. wpisane znaki nie zostaną wyświetlone użytkownikowi.

Zapytanie o użytkownika i hasło

```
1 #!/bin/bash
2 # zapytanie o nazwę użytkownika i hasło
3 read -p 'Username: ' user
4 read -sp 'Password: ' pass
5 echo
6 echo Thank you $user, we now have your login
    ↪ details
```

user_pass.sh

Odczyt wielu zmiennych

```
1 #!/bin/bash
2 # użycie read do wczytania wielu zmiennych
3 echo What cars do you like?
4 read car1 car2 car3
5 echo Your first car was: $car1
6 echo Your second car was: $car2
7 echo Your third car was: $car3
```

cars.sh

Odczyt wielu zmiennych

```
$ ./cars.sh
What cars do you like?
Jaguar Aston\ Martin Rolls-Royce Phantom
Your first car was: Jaguar
Your second car was: Aston Martin
Your third car was: Rolls-Royce Phantom
```

Odczyt wielu zmiennych

- jeśli podana jest więcej niż jedna zmienna, `read` podzieli wejście używając białych znaków
- każdy kolejny wyraz zostanie przypisany do kolejnej zmiennej
- biały znak można wprowadzić do wartości zmiennej używając znaku ucieczki \
- jeśli słów jest więcej niż zmiennych, wszystkie „nadmiarowe” zostaną przypisane do ostatniej zmiennej
- w odwrotnym przypadku – część zmiennych będzie pusta

Potoki i przekierowania

W GNU/Linux często używa się potoków (ang. *pipe*), by połączyć ze sobą serię prostych, jednozadaniowych poleceń w celu uzyskania bardziej złożonego rozwiązania, a także przekierowań, by np. zachować wyniki pracy programu.

Mechanizm ten może także być wykorzystany w skryptach, co pozwala na np. dopasowane do potrzeb filtrowanie danych.

```
$ ls /etc | wc -l > count.txt
```

STDIN, STDOUT i STDERR

Bash obsługuje potoki i przekierowania używając specjalnych plików. Każdy działający proces otrzymuje swój własny zestaw takich plików, a w momencie użycia potoków lub przekierowań są one odpowiednio ze sobą łączone.

Każdy proces otrzymuje trzy pliki:

1. STDIN – */proc/**<processID>**/fd/0* – standardowe wejście
2. STDOUT – */proc/**<processID>**/fd/1* – standardowy wyjście
3. STDERR – */proc/**<processID>**/fd/2* – standardowe wyjście błędów

STDIN, STDOUT i STDERR

By nie trzeba było posługiwać się process ID, system udostępnia pewne skróty:

1. STDIN – */dev/stdin* lub */proc/self/fd/0*
2. STDOUT – */dev/stdout* lub */proc/self/fd/1*
3. STDERR – */dev/stderr* lub */proc/self/fd/2*

Część „fd” w powyższych ścieżkach oznacza deskryptor pliku (ang. *file descriptor*).

Odczyt danych z wejścia

Istnieją trzy podstawowe sposoby na uzyskanie danych od użytkownika:

1. argumenty linii poleceń,
2. dane odczytane w trakcie działania skryptu,
3. dane, które zostały przekierowane poprzez STDIN.

To, która z metod jest najlepsza zależy od sytuacji.

Argumenty linii poleceń – zalety

Zwykle preferuje się argumenty linii poleceń:

- są wygodne dla użytkowników, ponieważ są przechowywane w ich historii poleceń,
- są łatwe w użyciu, jeśli dany skrypt ma być uruchamiany przez inne skrypty i procesy.

Odczyt w trakcie działania skryptu – zalety

Czasami wprowadzane dane powinny pozostać poufne (nie przechowywane w historii poleceń). Dobrym przykładem są szczegóły logowania użytkownika (login i hasło). Tego typu dane najlepiej wczytać w czasie działania skryptu.

Standardowe wejście – zalety

Jeżeli skrypt ma służyć do przetwarzania dużych ilości danych, najprawdopodobniej STDIN jest właściwym rozwiązaniem. W ten sposób dane mogą zostać podane do skryptu przy pomocy potoków bądź przekierowań.

Którą metodę wybrać?

Zwykle okazuje się, że kombinacja powyższych metod jest najwłaściwsza. Użytkownik może mieć np. opcję podania pliku wejściowego z linii poleceń, ale jeśli nie zostanie ona wykorzystana skrypt będzie czytał z STDIN. A może argumenty linii poleceń maja służyć do zdefiniowania ogólnego zachowania skryptu, równocześnie wymagając od użytkownika podania dodatkowych danych.

Którą metodę wybrać?

Podstawowe zagadnienia, które należy wziąć pod uwagę podczas decydowania w jaki sposób użytkownicy powinni podawać dane do skryptu, to:

- łatwość użycia – który ze sposobów pozwoli użytkownikom na najwygodniejsze uruchamiania skryptu?
- bezpieczeństwo – czy wymagane są jakieś poufne dane, które nie powinny być przechowywane w historii?
- solidność i intuicyjność – w jaki sposób można sprawić, by użycie skryptu było intuicyjne i zapobiegało prostym pomyłkom?

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
  - Let
  - Expr
  - $(())
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje

Let

Wbudowana funkcja `let` pozwala na przeprowadzenie operacji arytmetycznych.

```
1 #!/bin/bash
2 # arytmetyka z użyciem let
3 # poniżej wersja podstawowa. Uwaga - jeżeli
    ↪ działanie nie jest otoczone cudzysłowiem,
    ↪ musi być zapisywane bez spacji
4 let a=5+4
5 echo $a # 9
6 let "a = 5 + 4"
7 echo $a # 9
8 let a++
9 echo $a # 10
```

Let

```
10 # znak mnożenia (*) powinien być poprzedzony
       ↪ znakiem ucieczki (\) jeśli uzywamy go bez
       ↪ cudzysłowia
11 let a=4\*5
12 echo $a # 20
13 let "a = 4 * 5"
14 echo $a # 20
15 let "a = $1 + 30"
16 echo $a # 30 + wartość pierwszego argumentu
       ↪ skryptu
```

let.sh

Podstawowe operatory

Najczęściej używane operatory arytmetyczne:

Operator	Operacja
+, -	dodawanie, odejmowanie
*, /, %	mnożenie, dzielenie, modulo (reszta z dzielenia)
var++	inkrementacja (zwiększenie o 1)
var--	dekrementacja (zmniejszenie o 1)

Expr

`expr` jest podobne do `let`, jednak wynik, zamiast być przypisany do zmiennej, jest od razu wypisywany.

W przeciwieństwie do `let`, wyrażenie nie może znaleźć się w cudzysłowie, a spacje pomiędzy operatorem a operandami są konieczne.

```
expr item1 operator item2
```

Często stosuje się także przypisanie wyniku działania `expr` do zmiennej przy pomocy mechanizmu podstawiania komend.

Expr

```
1 #!/bin/bash
2 # arytmetyka z użyciem expr
3 # poniżej wersja podstawowa. Wyrażenie nie jest
    ↪ otoczone cudzysłowiem i zawiera spacje
4 expr 5 + 4 # 9
5 # jeśli otoczymy je cudzysłowiem, nie zostanie
    ↪ ono wyliczone, tylko potraktowane jako
    ↪ całość
6 expr "5 + 4" # 5 + 4
7 # tak samo przy braku spacji
8 expr 5+4 # 5+4
```

Expr

```
9  # mnożenie, tak samo jak w let, musi być
       ↪ poprzedzone znakiem ucieczki
10 expr 5 \* $1 # 5 * pierwszy argument skryptu
11 expr 11 % 2 # 1
12 # aby zapisać wynik działania do zmiennej,
       ↪ używamy podstawienia poleceń
13 a=$( expr 10 - 3 )
14 echo $a # 7
```

expr.sh

$(())

Wynik działania polecenia może łatwo zostać przypisany do zmiennej. Mechanizm ten można także wykorzystać w celu przeprowadzenia podstawowych operacji arytmetycznych, jeśli tylko lekko zmienimy jego składnię. Robi się to stosując podwójne nawiasy:

```
$ $(( expression ))
```

$(())

```
1  #!/bin/bash
2  # arytmetyka z użyciem $(())
3  a=$(( 4 + 5 ))
4  echo $a # 9
5  a=$((3+5))
6  echo $a # 8
7  b=$(( a + 3 ))
8  echo $b # 11
9  b=$(( $a + 4 ))
10 echo $b # 12
11 (( b++ ))
12 echo $b # 13
```

$(())

```
13 (( b += 3 ))
14 echo $b # 16
15 a=$(( 3 * 5 ))
16 echo $a # 15
17 a=$(( 2 * $b ))
18 echo $a # 32
```

double_parentheses.sh

$(())

Podwójne nawiasy są bardzo elastyczne jeśli chodzi o formatowanie wyrażeń, co wpływa na popularność użycia tego rozwiązania. Dodatkowym atutem jest nieznacznie krótszy czas wykonania (co miało większe znaczenie w przeszłości).

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje

Długość zmiennej

Aby w prosty sposób sprawdzić długość zmiennej (ilość znaków, które zawiera), można uzyć następującej konstrukcji:

```
$ ${#variable}
```

Długość zmiennej

```
1 #!/bin/bash
2 # sprawdzanie długości zmiennej
3 a='Hello World'
4 echo ${#a} # 11
5 b=4953
6 echo ${#b} # 4
```

length.sh

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
  - Podstawowa składnia if
  - Podstawowe operatory
  - Testowanie warunków
  - Formatowanie kodu
  - Pozostałe możliwości if
  - Instrukcja case
- Pętle
- Funkcje

If

```
if [ <some test> ]
then
   <commands>
fi
```

If

```
#!/bin/bash
# podstawowy if
if [ $1 -gt 100 ]
then
  echo Hey that\'s a large number.
  pwd
fi
```

basic_if.sh

Test

Kwadratowe nawiasy są w rzeczywistości odniesieniem do komendy `test`, co oznacza, że wszystkie operatory, których użycie jest dozwolone w tej komendzie mogą być zastosowane także w `[]`. Inaczej:

```
if [ <some test> ]
```

to to samo co:

```
if test <some test>
```

Podstawowe operatory

Negacja i ciągi:

Operator	Opis
! EXPRESSION	wyrażenie jest fałszywe
-n STRING	długość ciągu jest większa niż 0
-z STRING	długość ciągu jest 0 (ciąg jest pusty)
STRING1 = STRING2	ciągi są takie same
STRING1 != STRING2	ciągi są różne

Podstawowe operatory

Liczby:

Operator	Opis
INTEGER1 -eq INTEGER2	INTEGER1 jest numerycznie równy INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 jest numerycznie większy od INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 jest numerycznie mniejszy od INTEGER2

Podstawowe operatory

Pliki:

Operator	Opis
-e FILE	plik/katalog o ścieżce FILE istnieje
-d FILE	plik istnieje i jest katalogiem
-r FILE	plik istnieje i można go odczytać
-w FILE	plik istnieje i można go zapisać
-x FILE	plik istnieje i można go uruchomić
-s FILE	plik istnieje i nie jest pusty

Różnica między = a -eq

`[ 001 = 1 ]` zwróci fałsz, ponieważ = porównuje argumenty znak po znaku.

`[ 001 -eq 1 ]` zwróci prawdę, ponieważ ich numeryczna wartość jest taka sama.

Testowanie warunków

Ponieważ `[]` jest referencją do komendy `test`, możemy przeprowadzać eksperymenty w linii poleceń w celu upewnienia się, że dobrze rozumiemy działanie danego warunku.

Występująca w następnym przykładzie zmienna `$?` przechowuje kod wyjścia (*exit status*) ostatnio uruchomionego polecenia. **0 oznacza sukces/prawdę**, natomiast 1 oznacza porażkę/błąd.

Testowanie warunków

```
$ test 001 = 1
$ echo $?
1
$ test 001 -eq 1
$ echo $?
0
$ touch myfile
$ test -s myfile
$ echo $?
1
$ ls /etc > myfile
$ test -s myfile
$ echo $?
0
```

Wcięcia

Bash nie posiada ściśle zdefiniowanych reguł dotyczących wcięć w kodzie - można więc je dowolnie stosować (lub wcale), a skrypty będą działały tak samo.

Odpowiednie stosowanie wcięć zwiększa natomiast czytelność kodu i ułatwia jego późniejszą modyfikację. Służy także jako wizualna wskazówka dotycząca organizacji kodu, co jest szczególnie przydatne w większych skryptach.

Zagnieżdżone wyrażenia warunkowe

W skrypcie może znajdować się dowolna ilość, także zagnieżdżonych, instrukcji warunkowych.

Kolejny przykład ilustruje także wykorzystanie ewaluacji wyrażeń arytmetycznych w roli warunku.

Zagnieżdżone wyrażenia warunkowe

```
1  #!/bin/bash
2  # zagnieżdżony if
3  if [ $1 -gt 100 ]
4  then
5    echo Hey that\'s a large number.
6
7    if (( $1 % 2 == 0 ))
8    then
9        echo And is also an even number.
10   fi
11 fi
```

nested_if.sh

Else

Czasami zachodzi potrzeba wykonania jednego zestawu akcji, jeżeli wyrażenie jest prawdziwe, a innego jeśli jest fałszem. w takich sytuacjach używa sie `else`.

```
if [ <some test> ]
then
  <commands>
else
  <other commands>
fi
```

Else

```
1 #!/bin/bash
2 # przykład if-else
3 value=$( grep -ic $1 /etc/passwd )
4 if [ $value -eq 1 ]
5 then
6   echo "I found $1"
7 else
8   echo "I didn't find $1"
9 fi
```

find_user.sh

Polecenie `grep -ic` zlicza wystąpienia danego ciągu (ignorując wielkość liter). Jest to prosta metoda sprawdzania, czy dany ciąg istnieje w podanym pliku, a jeśli tak – podjęcia jakiejś akcji.

Elif

```
if [ <some test> ]
then
  <commands>
elif [ <some test> ]
then
  <different commands>
else
  <other commands>
fi
```

Elif

```
1  #!/bin/bash
2  # zastosowanie elif
3  if [ $1 -ge 18 ]
4  then
5    echo You may go to the party.
6  elif [ $2 = 'yes' ]
7  then
8    echo You may go to the party but be back before
       ↪ midnight.
9  else
10   echo You may not go to the party.
11 fi
```

party_permission.sh

Operacje logiczne

Czasami zachodzi potrzeba wykonania jakiejś akcji tylko wtedy, gdy kilka (lub przynajmniej jeden z) warunków jest spełnionych. Zachowanie takie uzyskuje się stosując operatory logiczne.

Operator	Operacja
&&	logiczne and
\|\| | logiczne or |

Logiczne AND

Przykładowo, jeśli chcemy, by operacja została wykonana tylko jeśli plik jest odczytywalny i ma rozmiar większy niż zero:

```
1 #!/bin/bash
2 # logiczne and
3 if [ -r $1 ] && [ -s $1 ]
4 then
5   echo This file is useful.
6 fi
```

useful_file.sh

Logiczne OR

Jeśli chcemy, by została podjęta inna akcja dla użytkowników bob oraz andy:

```
1 #!/bin/bash
2 # logiczne OR
3 if [ $USER = 'bob' ] || [ $USER = 'andy' ]
4 then
5   ls -alh
6 else
7   ls
8 fi
```

special_ls.sh

Case

```
case <variable> in
<pattern 1>)
  <commands>
  ;;
<pattern 2>)
  <other commands>
  ;;
esac
```

Case

```
1  #!/bin/bash
2  # użycie case
3  case $1 in
4    start)
5      echo starting
6      ;;
7    stop)
8      echo stoping
9      ;;
10   restart)
11     echo restarting
12     ;;
```

Case

```
13    *)
14       echo don\'t know
15       ;;
16 esac
```

action_case.sh

- `)` oznacza koniec dopasowywanego wzorca,
- koniec listy poleceń dla danej wartości oznacza się `;;`
- `*` oznacza wartość domyślną, dopasowywaną, jeśli żadna wcześniejsza nie pasuje; musi wystąpić jako ostatnia wartość (wszystkie po niej będą ignorowane).

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
  - Pętle while i until
  - Pętla for
  - Polecenia sterujące w pętlach
  - Instrukcja select
- Funkcje

While

Najprostszą formą pętli jest pętla `while`. Będzie ona wykonywana tak długo, jak długo warunek jest spełniony.

```
while [ <some test> ]
do
  <commands>
done
```

Należy zaznaczyć, że podobnie jak w przypadku instrukcji warunkowej `if` formuła testująca umieszczona jest w nawiasach kwadratowych.

While

Przykład wypisywania liczb w kolejności rosnącej od 1 do 10.

```
1 #!/bin/bash
2 # przykład użycia while
3 counter=1
4 while [ $counter -le 10 ]
5 do
6   echo $counter
7   ((counter++))
8 done
```

basic_while.sh

Until

Pętla `until` jest bardzo podobna do `while`. Będzie ona jednak wykonywana dopóki warunek **nie** jest spełniony.

```
until [ <some test> ]
do
  <commands>
done
```

Until

```
1 #!/bin/bash
2 # przykład użycia until
3 counter=1
4 until [ $counter -gt 10 ]
5 do
6   echo $counter
7   ((counter++))
8 done
```

basic_until.sh

Jak widać, jedyne różnice dotyczą zamiany słowa kluczowego z `while` na `until` i odwrócenia warunku.

For

```
for var in <list>
do
   <commands>
done
```

For

```
1 #!/bin/bash
2 # przykład użycia for
3 names='Stan Kyle Cartman'
4
5 for name in $names
6 do
7   echo $name
8 done
```

basic_for.sh

Zakresy

```
1 #!/bin/bash
2 # przykład użycia zakresów w pętli for
3 for value in {1..5}
4 do
5   echo $value
6 done
```

range_for.sh

Zakresy

- Pomiędzy nawiasami klamrowymi nie może pojawić się spacja. Spowodowała by ona traktowanie takiego zapisu nie jako zakresu, a jako listy elementów,
- wartość początkowa i końcowa zakresu są dowolne. Jeżeli pierwsza jest większa, pętla będzie liczyła w dół,
- krok inkrementacji/dekrementacji może zostać ustawiony poprzez dodanie kolejnych `..` (od wersji 4.0 bash).

Zakresy

```
1 #!/bin/bash
2 # przykład użycia zakresów z krokiem
3 for value in {10..0..2}
4 do
5   echo $value
6 done
```

step_for.sh

Przetwarzanie listy plików

Jednym z bardziej przydatnych zastosowań pętli jest przetwarzanie listy plików. Powiedzmy, że chcemy zmienić rozszerzenie kilku plików .html na .php:

```
1 #!/bin/bash
2 # zmiana rozszerzenia plików .html w danym
    ↪ katalogu na .php
3 for value in $1/*.html
4 do
5   mv $value $1/$( basename -s .html $value ).php
6 done
```

html2php.sh

Break

Polecenie `break` wymusza opuszczenie pętli. Jest to przydatne w sytuacjach, gdzie planowo pętla powinna się zakończyć pod jakimś warunkiem, jednakże mogą wystąpić sytuacje nadzwyczajne, które powinny przerwać przetwarzanie. Przykładem może być kopiowanie dużej ilości plików i wyjątek w postaci kończenia się miejsca na dysku.

Break

```
1  #!/bin/bash
2  # przykład użycia break
3  for value in {1..100}
4  do
5    if [ $value -ge $1 ]
6    then
7      break
8    fi
9    echo $value
10 done
```

break.sh

Continue

Polecenie `continue` wymusza przejście pętli do wykonywania następnej iteracji. Przykładowo, możemy przetwarzać listę plików i trafić na taki, do którego nie mamy uprawnień do odczytu.

Continue

```
1  #!/bin/bash
2  # przykład użycia continue
3  for value in {1..100}
4  do
5    if (( $value % $1 == 0 ))
6    then
7      continue
8    fi
9    echo $value
10 done
```

continue.sh

Select

Mechanizm wyboru przy użyciu instrukcji `select` pozwala na stworzenie prostego menu:

```
select var in <list>
do
  <commands>
done
```

Select

Uruchomione polecenie `select` wyświetli użytkownikowi numerowaną listę opcji do wyboru, po jednej dla każdego elementu z oddzielonej spacjami listy elementów. Następnie pojawi się znak zachęty, pozwalający użytkownikowi na wpisanie wybranego numeru. Po zatwierdzeniu wyboru wykonane zostaną polecenia znajdujące się pomiędzy `do` a `done`.

`select` także jest formą pętli – po przetworzeniu poleceń znak zachęty zostanie ponownie wyświetlony użytkownikowi.

Select

- Nie ma sprawdzania poprawności wprowadzanych danych. Jeśli użytkownik wprowadzi coś innego niż numer, bądź numer spoza dopuszczalnego zakresu, zmienna będzie pusta,
- wciśnięcie enter przed dokonaniem wyboru skutkuje ponownym wyświetleniem listy opcji,
- pętla zakończy się, gdy zostanie wprowadzony EOF (Ctrl+D) lub w trakcie przetwarzania pojawi się break,
- znak zachęty może być zmieniony poprzez edycje zmiennej środowiskowej `PS3`.

Select

```
1  #!/bin/bash
2  # użycie select
3  names='Kyle Cartman Stan Quit'
4  PS3='Select character: '
5
6  select name in $names
7  do
8    if [ $name = 'Quit' ]
9    then
10     break
11   fi
12   echo Hello $name
13 done
```

select.sh

Wprowadzenie do pisania skryptów

- Struktura skryptów
- Uruchamianie skryptów
- Zmienne
- Wprowadzanie danych
- Arytmetyka
- Długość zmiennej
- Instrukcje warunkowe
- Pętle
- Funkcje
  - Podstawy definiowania funkcji
  - Zwracanie wyniku działania funkcji
  - Zasięg zmiennych w funkcjach

Funkcje

Równoważne zapisy:

```
function_name () {
  <commands>
}
```

```
function function_name {
  <commands>
}
```

Funkcje

- `()` pełnią tylko dekoracyjną funkcję, nigdy nie podaje się w nich argumentów,
- definicja funkcji musi pojawić się przed jej pierwszym wywołaniem (analogicznie jak deklaracja zmiennych).

Pierwsza funkcja

```
1 #!/bin/bash
2 # definicja funkcji
3 function hello {
4   echo Hello world!
5 }
6
7 hello
8 hello
```

function.sh

```
$ ./function.sh
Hello world!
Hello world!
```

Funkcje z argumentami

Dostęp do argumentów funkcji odbywa się na tej samej zasadzie, co dostęp do argumentów skryptu – są one dostępne poprzez `$1`, `$2` itd.

Do argumentów powyżej `$9` można dostać się używając notacji `${10}` itp.

Funkcje z argumentami

```
1 #!/bin/bash
2 # definicja funkcji
3 function hello {
4   echo Hello $1! How are you?
5 }
6
7 hello Andy
8 hello Bob
```

args_function.sh

```
$ ./args_function.sh
Hello Andy! How are you?
Hello Bob! How are you?
```

Status wykonania funkcji

W bash nie istnieje bezpośrednio pojęcie wartości zwracanej z funkcji. Funkcje mogą natomiast zwrócić status, używając w tym celu polecenia `return`.

Wynik działania funkcji można sprawdzić tak samo jak wynik działania każdej innej komendy – używając `$?`

Status wykonania funkcji

```
1  #!/bin/bash
2  # status wykonania funkcji
3  function hello {
4    echo Hello $1! How are you?
5    if [ $1 = 'Andy' ]
6    then
7      return 1
8    else
9      return 0
10   fi
11 }
12
13 hello Andy
```

Status wykonania funkcji

```
14 echo $?
15 hello Bob
16 echo $?
```

status_function.sh

```
$ ./status_function.sh
Hello Andy! How are you?
1
Hello Bob! How are you?
0
```

Zazwyczaj kod wyjścia równy 0 oznacza sukces – funkcja wykonała się bez błędów. Cokolwiek innego oznacza, że wystąpił jakiś problem.

Pozyskiwanie wyników działania funkcji

Jeśli funkcja ma wykonać działania, w wyniku których powstaje liczba, można rozważyć zastosowanie w tym celu kodu wyjścia. Nie jest on do tego przeznaczony, ale zadziała.

Pozyskiwanie wyników działania funkcji

Standardowo w celu otrzymania wyników z funkcji, które można przypisać do zmiennej, stosuje się mechanizm podstawienia komend.

W tym celu funkcja musi wypisywać wynik swojego działania (i tylko jego!).

Pozyskiwanie wyników działania funkcji

```
1 #!/bin/bash
2 # pozyskanie wartości z funkcji
3 function num_lines {
4   cat $1 | wc -l
5 }
6
7 num=$( num_lines $1 )
8 echo File $1 has $num lines.
```

return_value.sh

```
$ ./return_value.sh return_value.sh
File return_value.sh has 8 lines.
```

Zasięg zmiennych w funkcji

Domyślnie zmienne są widoczne w całym skrypcie. Aby stworzyć zmienna, której zasięg (możliwość użycia) jest ograniczona do danej funkcji, musimy zadeklarować ją jako lokalną przy pierwszym ustawieniu wartości:

```
local var_name=<var_value>
```

Używanie zmiennych lokalnych należy do dobrych praktyk i zabezpiecza przed przypadkową modyfikacją zmiennych globalnych.

Zasięg zmiennych w funkcji

```
1  #!/bin/bash
2  # zasięg zmiennych
3
4  var1=a
5  var2=b
6
7  function change {
8    local var1=c
9    var2=d
10   echo inside call :: var1 : $var1, var2: $var2
11 }
```

Zasięg zmiennych w funkcji

```
12 echo before call :: var1 : $var1, var2: $var2
13 change
14 echo after call :: var1 : $var1, var2: $var2
```

function_scope.sh

```
$ ./function_scope.sh
before call :: var1 : a, var2: b
inside call :: var1 : c, var2: d
after call :: var1 : a, var2: d
```