2 Podstawy Basha

Jak dotąd nasze skrypty składały się jedynie z jednego lub kilku wywołań innych programów. A jednak, Bash jest „pełnoprawnym" językiem programowania. Można w nim tworzyć zmienne, pętle, funkcje itp. Nieraz właśnie przy pomocy tych narzędzi najłatwiej rozwiązać jakieś bardziej skomplikowane zadanie.

Zmienne

Zmienne możemy traktować jak etykiety, które przechowują ciągi znaków. Definiujemy je przy pomocy znaku `=` **bez spacji po obu jego stronach**:
`dog="Azor"`

Do zmiennych w poleceniach odwołujemy się poprzedzając je symbolem `$`.

```
echo $dog       ⟶  Azor
echo $dogek     ⟶
echo ${dog}ek   ⟶  Azorek
```

Okazuje się, że to, czy nazwę zmiennej umieścimy pomiędzy apostrofami, czy też cudzysłowami, ma znaczenie – cudzysłów umożliwia interpretację znaku `$` i ewaluuje zmienne.

```
echo 'Jestem $LOGNAME, a to jest pies $dog.'
⟶  Jestem $LOGNAME, a to jest pies $dog.
echo "Jestem $LOGNAME, a to jest pies $dog."
⟶  Jestem boraks, a to jest pies Azor.
```

Zmienne środowiskowe - `env`

Powyżej pojawiła się zmienna `LOGNAME`, zawierająca nazwę użytkownika – jedna ze *zmiennych środowiskowych*. Są to zmienne, które są dostępne dla wszystkich procesów uruchomionych w powłoce. Zgodnie z konwencją, ich nazwy zapisujemy dużymi literami. Przykłady innych zmiennych środowiskowych:

`HOSTNAME`	nazwa (adres) maszyny, na którą się zalogowaliśmy,
`HOME`	ścieżka do katalogu domowego użytkownika,
`PATH`	ścieżki (oddzielone dwukropkami), w których szukane są programy powłoki,
`PWD`	ścieżka do aktualnego katalogu.

Wszystkie zmienne środowiskowe możemy wypisać komendą `env` (choć nie jest to jej główne przeznaczenie – patrz `man env`).
Możemy zmodyfikować zmienną `PATH` dodając lokalizację z własnymi programami:
`PATH=$PATH:$HOME/scripts`

Jeżeli w katalogu `scripts` znajduje się program o nazwie `my_cmd`, to od tej pory będziemy go mogli uruchomić w dowolnym miejscu wpisując po prostu `my_cmd`.

Możemy to zautomatyzować, dodając linijkę do `~/.bashrc` (plik ładowany przy otwieraniu nowego basha).

Zmienne w skrypcie

Do nazwy skryptu możemy się dostać za pomocą: `"$0"`. Do kolejnych argumentów przekazanych do skryptu za pomocą: `"$1", "$2", "$3"`, itd. Do listy wszystkich parametrów naraz możemy dostać się za pomocą `$@`, zaś do liczby parametrów za pomocą: `$#`.

Ewaluacja wyrażeń arytmetycznych w bashu - `$((...))`

Bash jest językiem skryptowym, nastawionym przede wszystkim na wywoływanie programów i przekazywanie argumentów. Wyrażenia arytmetyczne nie są dostępne wprost:

```
echo 2+2  ⟶  2+2
```

Aby wyrażenie zostało wyliczone, trzeba je odpowiednio oznaczyć. Z historycznych względów jest wiele sposobów by tego dokonać. W starszych skryptach można się spotkać np. z wywołaniami `expr`, `let` lub umieszczeniem wyrażeń wewnątrz `$[ ... ]`. Każde z tych poleceń ma swoje unikalne reguły i możliwości.

Obecnie jednak stosuje się konstrukcję POSIX-ową: `$(( ... ))` która ma najszersze możliwości i w praktyce jest najprostsza w użyciu. Obsługiwane są niemal wszystkie operatory takie jak z języka C.

```
echo $((2+2))   ⟶  4
echo $((11&7))  ⟶  3
```

Wewnątrz `$(( ... ))` każdy napis od razu jest traktowany jako zmienna i nie trzeba stawiać dodatkowego `$`. Wyjątek stanowią zmienne które bez `$` są liczbami (np. `$1`).

```
echo $((i=3**3))  ⟶  27 #potegowanie
echo $((++i))     ⟶  28
```

Niestety wspierane są tylko operacje na liczbach całkowitych. Dla wyrażeń zmiennoprzecinkowych trzeba użyć zewnętrznego programu, takiego jak np. `bc -l`.

```
echo "2+2"| bc     ⟶  4
echo "3+5.5"| bc   ⟶  8.5
echo "2/5"| bc     ⟶  0
echo "2/5"| bc -l  ⟶  .40000000000000000000
```

Używając parametru `scale` możemy określić liczbę miejsce po przecinku (bez użycia `-l`).

```
echo "scale=5; 2/3"| bc  ⟶  .66666
```

Kiedy wyrażenie nie jest częścią innej instrukcji (czyli wtedy gdy nie potrzebujemy wyniku), stosujemy składnię `((...))`, czyli bez `$`. Przykład:

```
echo $((x=3))        ⟶  3
((x=x+5)); ((x++))   ⟶        # x=8, x=9
echo $((x+1))        ⟶  10
```

Kod wyjścia - `$?`

Każdy program jaki wykonujemy w powłoce zwraca tzw. kod wyjścia. Pozwala on określić czy program zakończył się poprawnie czy nie. Kod ostatniego wykonanego polecenia możemy odczytać z `$?`. Status wyjścia ze skryptu zwracamy poprzez `exit <num>`.

Kod `0` oznacza, ze program zadziałał poprawnie, inny niż `0` – że nie (dlatego zwykle piszemy `return 0` na końcu `main`-a w C/C++).

Łączenie komend

Pojęcie poprawnego i błędnego wykonania komendy pozwala na bardzo użyteczną konstrukcję – łączenie komend:

* `cmd1 && cmd2 && ...`
* `cmd1 || cmd2 || ...`

Koniukcja `&&` wykonuje kolejną komendę, jeśli poprzednia zakończyła się poprawnie. Alternatywa `||` wykonuje kolejną komendę, jeśli poprzednia zakończyła błędem (czyli wykonuje komendy aż do pierwszego sukcesu). Przykład:

`mkdir x && cd x`	stwórz folder, a gdy to się uda wejdź do niego
`mkdir x1 || mkdir x2`	załóż `x1`, jeśli się nie powiedzie, załóż `x2`

Testowanie warunków logicznych w bashu - `[ ... ]`, `test`

Wyrażenia arytmetyczne najłatwiej testować za pomocą powyższej konstrukcji `((...))`. Jednakże, często zdarza się, że należy sprawdzić warunek innej natury, np. porównać ciągi znaków bądź sprawdzić właściwości pliku.

Służy do tego konstrukcja: `[ wyrażenie1 operator wyrażenie2 ]`
Należy pamiętać, że po `[` oraz przed `]` spacja jest wymagana.

UWAGA:
jest to tak naprawdę alias do polecenia: `test wyrażenie1 operator wyrażenie2`

Zrozumienie tego, pozwala zrozumieć zachowanie tej konstrukcji, oraz pozwala uniknąć różnych kłopotów – np. gdy argumenty zawierają spacje, lub są ciągiem pustym.

`test "$x" = "$y"`	porównuje zawartość dwóch zmiennych
`test -n "$1"`	sprawdza czy `$1` ma długość większą niż 0
`[ -z "$x" ]`	sprawdza czy `$x` ma długość równą 0
`[ -e "file.tmp" ]`	sprawdza czy isnieje plik o nazwie `file.tmp`
`[ $# -lt 3 ]`	sprawdza czy liczba argumentów jest mniejsza od 3

Jak widać składnia dopuszcza jedno- i dwu-argumentowe operatory, a porównania mogą być różnego typu. Wiedząc, że `[ ... ]` to komenda `test`, możemy sprawdzić jej składnie w `man test`. Oferuje ona dużo możliwości.

Polecenie `test` (konstrukcja `[ ... ]`) sprawdza warunek wyrażenia zgodnie z podanym operatorem. Wynik porównania jest zwrócony jako kod wyjścia (`$?`). Jeśli warunek został spełniony - kod wyjścia to `0`, jeśli nie `test` zwraca `1` (zgodnie z tym, że `0` oznacza sukces).

Porównania na stringach

`=`	równy
`!=`	różny
`"<"`	pierwszy string alfabetycznie przed drugim (uwaga na `<` – przekierowanie do pliku)
`">"`	pierwszy string alfabetycznie za drugim
`-n`	string ma długość większą niż 0
`-z`	string ma zerową długość

```
test abc = def; echo $?
⟶1
```

Przykładowe operacje na plikach

`[ -e PLIK ]`	zwraca `true` jeśli `PLIK` isntieje
`[ -s PLIK ]`	zwraca `true` jeśli `PLIK` isntieje i jest niepusty
`[ -d PLIK ]`	zwraca `true` jeśli `PLIK` jest katalogiem
`[ -r PLIK ]`	zwraca `true` jeśli użytkownik ma prawo do czytania
`[ PLIK1 -nt PLIK2 ]`	zwraca `true` jeśli `PLIK1` jest nowszy niż `PLIK2`

Łączenie testów odbywa się przez `-o` (alternatywa) lub `-a` (koniunkcja).

Konstrukcja `[[ ]]`

Obok komendy `[ ]` bash oferuje komendę `[[ ]]`, która daje więcej możliwości. W szczególności w podwójnych kwadratowych nawiasach możemy umieścić następujące warunki:

`[[ NAPIS = (lub ==) GLOB ]]`	zwraca `true`, jeśli `NAPIS` dopasowuje się do globa `GLOB`
`[[ NAPIS != GLOB ]]`	zwraca `false`, jeśli `NAPIS` *nie* dopasowuje się do globa `GLOB`
`[[ NAPIS =~ REGEX ]]`	zwraca `true`, jeśli `NAPIS` dopasowuje się do wyrażenia regularnego `REGEX` (zapisanego w stylu ERE)

W tym wypadku łączenie wyrażeń odbywa się poprzez alternatywę `||` lub koniunkcję `&&`. Lepiej działa operator `<` (nie powoduje przekierowania) oraz nawiasowanie `()`. Szczegóły komend można sprawdzić poprzez `help [[; help [; help \(\(`.

Testowanie warunków logicznych – zwracanie kodów sukcesu lub błędu – pozwala na tworzenie instrukcji warunkowych w skryptach.

Instrukcje warunkowe - `if`, `case`

W bash-u występują dwie instrukcje warunkowe `if then else` oraz `case`.

```
if warunek1 ; then
    polecenia
elif warunek2 ; then
    polecenia
else
    polecania
fi
```

`warunek` może być zapisany za pomocą komendy, `[ ... ]` lub `[[ ... ]]`. Dla wyrażeń arytmetycznych można też użyć `(( ... ))` (bez znaku `$` na początku). Jeśli kod wyjścia jest `0` wykona się część po `then`, jeśli nie - sterowanie przechodzi do `else` bądź `elif` (odpowiednik `else if (warunek)`, jak to piszemy używając np. *C, C++*).

Klauzule `else` oraz `elif` są opcjonalne. Instrukcja `if` musi być zakończona przez `fi`.

```
if (($1 == 0)) ; then
    echo "$1 jest rowne zeru"
else
    echo "$1 jest rozne od zera"
fi
```

Sprawdzenie czy plik ma rozszerzenie `jpg`:

```
if [[ $1 = *.jpg ]]; then echo "$1 is a jpeg"; fi
```

Sprawdzenie, czy pierwszy argument podany do skryptu jest liczbą.

```
liczba='^[1-9][0-9]*$'
if [[ $1 =~ $liczba ]]; then echo "$1 to liczba"; fi
```

Gdy musimy dla kolejnych wartości zmiennej wykonywać różne akcje, instrukcja `if` staje się niewygodna. Do takich rozwiązań służy instrukcja `case` o następującej składni:

```
case zmienna in
    "wzorzec") polecenie ;;
    "wzorzec") polecenie ;;
    "wzorzec") polecenie ;;
    *) polecenie wykonywane domyslnie
esac
```

W instrukcji `case` ilość wzorców może być dowolna. Przykład wypisujący odpowiednią informację zależną od pierwszego argumentu programu:

```
case $1 in
    "0") echo "Nie rozumiem" ;;
    "1") echo "Jeden" ;;
    "2") echo "Dwa" ;;
    "3") echo "Trzy" ;;
    *) echo "Duzo"
esac
```

Pętle - `for, while, until`

W bash-u istnieją kilka rodzajów pętli: `for`, `while`, `until` oraz `select`. Omówimy tutaj pierwsze trzy z nich. Pętla `while` wykonuje swoje działanie dopóki warunek jest spełniony. Pętla `until` różni się tym, że kończy swoje działanie w momencie gdy warunek stanie się prawdziwy.

```
while warunek ; do
    polecenie
done
```

```
until warunek ; do
    polecenie
done
```

Przykład użycia, wypisanie 10 razy zmiennej `i`:

```
i=0;
while ((i<10)) ; do
    echo "$i"
    ((++i))
done
```

```
i=0
until ((i>=10)) do
    echo "$i"
    ((++i))
done
```

Pętla `for`

Najciekawsza z bash-owych pętli, może być używana na dwa sposoby. Po pierwsze możemy jej używać tak jak w `C/C++`, kiedy znamy ograniczenie na ilość iteracji:

```
for ((inicjalizacja zmiennej; warunek zakonczenia; zmiana zmiennej)) ;
do
    polecenia
done
```

Przykład, wypisanie 10 razy wartości zmiennej `i`:

```
#!/bin/bash
for (( i=1; i <= 10; i++ )) ; do
    echo " Iteracja nr: $i"
done
```

Drugie to iterowanie się po pewnej liście zbiorze elementów:

```
for zmienna in lista ; do
    polecenia
done
```

Przykład, wypisanie wszystkich małych liter alfabetu angielskiego:

```
for i in {a..z} ; do echo $i; done
```

Konstrukcja `for in` jest szczególnie przydatna gdy `lista` jest wynikiem działania innego programu. Na przykład, gdy chcemy coś zrobić z wszystkimi plikami podanymi przez `find`, można to zrobić tak:

```
for i in $(find . -name "*.txt"); do
    echo $i
done
```

Aby przerwać działanie pętli używamy komendy `break`. Natomiast aby przeskoczyć do kolejnej iteracji służy polecenie `continue`.

Nawiasowanie - grupowanie komend `()`

W wyrażeniach arytmetycznych nawiasowanie pozwala określać kolejność działań.

W przypadku basha nawiasowanie ma jeszcze jedno, być może ważniejsze znaczenie. Dowolna sekwencja operacji umieszczona w nawiasie jest traktowana jako jedna całość z wszelkimi tego konsekwencjami.

Na przykład, taka grupa może mieć jedno wspólne wejście/wyjście:

```
(echo "a"; echo "b") | xargs cp
ls | (head -n 2; echo "-> wywolano ls")
```

Używanie wyniku innej komendy - `$(...)`

Grupowanie pozwala na wywołanie komendy wewnątrz innej komendy i użycie jej wyniku (outputu). Robi to konstrukcja `$(cmd...)`

```
pliki=$(find . -name "*.c"; find . -name "*.txt")
for p in $pliki; do echo "plik $p ma $(wc -l < $p) linii"; done
```

(uwaga na pliki z białymi znakami w nazwie)

W przypadkach powyżej, wyrażenia wewnątrz nawiasów są wykonywane w *podpowłoce*. Oznacza to, że zmiany dokonane w środku (np. zmiana aktualnego katalogu) nie są widoczne na zewnątrz.

Tablice

Dana zmienna automatycznie staje się tablicą gdy użyjemy operatora kwadratowego, np.

```
names[0]="Bob"
names[1]="Peter"
```

Jeśli znamy wszystkie interesujące nas wartości, można utworzyć tablicę prościej:

```
names=("Bob" "Peter" "$USER" "Big Bad John")
```

Możemy również podać konkretne indeksy, do których chcemy przypisać wartości:

```
names=([0]="Bob" [1]="Peter" [20]="$USER" [21]="Big Bad John")
```

W przypadku, kiedy chcemy wypełnić tablicę nazwami plików, możemy użyć globów:

```
photos=(~/"My Photos"/*.jpg)
```

Możemy też stworzyć tablicę przechwytując output innej komendy:

```
content=($(ls))
numbers=($(echo 1 5 7))
```

Dostęp do tablicy

`${arr[2]}`	element spod indeksu 2
`${arr[@]}`	wszystkie elementy tablicy
`${#arr[@]}`	liczba elementów
`${!arr[@]}`	lista indeksów używanych w tablicy

Konstrukcja: `${tablica[@]}` jest bardzo wygodna do zastosowania w pętli `for`. Poniższe dwa przykłady dają ten sam efekt.

```
names=("Bob" "Peter" "$USER" "Big Bad John")
for name in "${names[@]}"; do echo "$name"; done
```

```
for name in "Bob" "Peter" "$USER" "Big Bad John"; do echo "$name"; done
```

Kolejny przykład operuje na nazwach plików, które możemy wczytać za pomocą globa.

```
myfiles=(*.{jpeg,jpg})
for file in "${myfiles[@]}"; do
    cp "$file" backups/
done
```

Cudzysłów `"${arr[@]}"` zapewnia poprawną obsługę wpisów z białymi znakami.

Wczytywanie danych ze standardowego wejścia - `read`

Czasami zdarza się, że standardowe wejście nie może być wprost przekierowane do jakiegoś programu lub komendy. Jeśli skrypt ma w jakiś sposób przetworzyć wejście, trzeba wczytać dane do zmiennych.

Wbudowaną komendą czytającą pojedynczą linię wejścia jest
`read -opcje nazwa1 nazwa2 ... reszta`.

Komenda działa następująco:

* wczytywana jest dokładnie jedna, pełna linijka
* pierwsze słowo (sekwencja znaków odseparowana znakiem białym) przypisane jest do zmiennej o nazwie `nazwa1`.

- drugie słowo jest przypisane do `nazwa2`
- i tak dalej do kolejnych zmiennych
- separatory pomiędzy tymi nazwami zostają zapomniane
- dla *ostatniej*, n-tej zmiennej, przypisane jest słowo n i cała następująca po nim reszta tekstu, wraz z wszelkimi białymi znakami.

Przykład – wczytywanie wszystkich linii i łączenie ich w jedną:

```
napis=
while read zmienna
do
  napis=$napis" "$zmienna
done
echo $napis
```

W poniższym przykładzie `plik.in` jest przekierowany do całej pętli `while`. Ostateczny wynik wypisany jest do pliku `plik.out`

```
#!/bin/bash
napis=
while read zmienna
do
  napis=$napis" "$zmienna
done < plik.in
echo $napis > plik.out
```

Funkcje

W każdym języku programowania istnieją konstrukcje pozwalające na zadeklarowanie podprogramów, które umożliwiają nam skrócenie kodu i rozszerzają możliwości języka. W bashu także możemy zdeklarować na dwa sposoby takie funkcje:

```
function nazwa_funkcji {
    polecenia
    ...
}

nazwa_funkcji () {
    polecenia
    ...
}
```

Przykład skryptu z użyciem funkcji:

```
#!/bin/bash
function witaj
{
    echo "Witaj Świecie!"
}

# wywolanie funkcje "witaj"
witaj
```

Przekazywanie argumentów do funkcji jest analogiczne jak przekazywanie ich do skryptu, z wyjątkiem tego że zmienna `$0`, która w skrypcie odpowiada za jego nazwę, nie jest dostępna:

```
nazwa_funkcji argument_1 argument_2 ...
```

Przykład:

```
#!/bin/bash

nazwa_funkcji ()
{
    echo "Argument pierwszy: $1"
    echo "Argument drugi: $2"
}

nazwa_funkcji "przyklad" "przyklad"
```

Funkcje w bashu nie zwracają wartości. Wprawdzie istnieje słowo kluczowe `return` ale służy ono tylko do zwrócenia statusu błędu. Można natomiast zwrócić wartość poprzez standardowe wyjście:

```
#!/bin/bash
suma () {
    echo $(($1+$2))
}
wynik=$(suma 2 2)
echo $wynik
```

Bardzo istotny jest sposób przykrywania zmiennych przez funkcję, spójrzmy na przykładzie jak traktowana jest zmienna `tmp`:

```
#!/bin/bash
function wypisz_tmp {
    tmp=4
    echo $tmp;
}
tmp=3;
wypisz_tmp;
echo $tmp;
```

Wynikiem wykonania powyższego skryptu będzie:

```
4
4
```

Jak widzimy zmienna zmienna globalna `tmp` zostaje nadpisana. Nie jest to rezultat jaki chcielibyśmy osiągnąć pisząc funkcję. Jednym z możliwych jest umieszczenie wywołania funkcji w bloku `()`. Blok ten uruchomi nam je w nowej powłoce gdzie nazwy zmiennych nie będą pamiętane.

Przykład:

```
#!/bin/bash
function wypisz_tmp {
    tmp=4;
    echo $tmp;
}

tmp=3;
(wypisz_tmp);
echo $tmp;
```

Wynik:

```
4
3
```

Dużo bardziej naturalne rozwiązanie, skutkujące tym samym rezultatem, używa słowa kluczowego `local`.

```
#!/bin/bash
function wypisz_tmp {
    local tmp=4;
    echo $tmp;
}

tmp=3;
wypisz_tmp;
echo $tmp;
```

Warto też wspomnieć o konstrukcji `declare` która tworzy zmienną z dodatkowymi restrykcjami.

- `-r` Zmienna read-only. Próba zmiany wartości tej zmiennej powoduje błąd.
- `-i` Zmienna przechowująca liczbę. Przypisanie tekstu spowoduje próbę ewaluacji do liczby.
- `-a` Tworzy pustą tablicę.
- `-x` Tworzy zmienną exportową, czyli taką która jest dostępna dla podprocesów i podpowłok

Zmienne stworzone poprzez `declare` wewnątrz funkcji mają zasięg lokalny, podobnie jak przy użyciu `local`

```
#!/bin/bash
function wypisz_tmp {
    declare -i tmp=4;
    echo $tmp;
}
tmp=3;
wypisz_tmp;
echo $tmp;
```

wypisze jak poprzednio:

```
4
3
```

Operacje na stringach

Bash, jak wiele innych języków skryptowych posiada szeroką gamę funkcji wbudowanych w składnię języka operujących na stringach. Oto kilka z nich:

Wyrażenie	Znaczenie
`${#string}`	zwraca długość string
`${string:pos}`	zwraca podnapis zaczynając od pozycji `pos`
`${string:pos:len}`	zwraca podnapis o długości `len` zaczynając od pozycji `pos`
`${string#substr}`	obcina najkrótsze dopasowanie `$substr` od początku `$string`
`${string##substr}`	obcina najdłuższe dopasowanie `$substr` od początku `$string`
`${string%substr}`	obcina najkrótsze dopasowanie `$substr` od końca `$string`
`${string%%substr}`	obcina najdłuższe dopasowanie `$substr` od końca `$string`

`${string/substr/rep}`	zastępuje wystąpienie `$substr` ciągiem `$rep`
`${string//substr/rep}`	zastępuje wszystkie wystąpienia `$substr` ciągiem `$rep`
`${string/#substr/rep}`	jeśli `$substr` zostanie dopasowany do początku `$string`, to podstawia `$rep` za `$substring`
`${string/%substr/rep}`	jeśli `$substr` zostanie dopasowany na końcu `$string`, to podstawia `$rep` za `$substring`
`expr match "$str" '$substr'`	długość dopasowania `$substr*` od początku `$str`
`expr "$str" : '$substr'`	długość dopasowania `$substr*` od początku `$str`
`expr index "$str" $substr`	pozycja pierwszego znaku dopasowania `$substr*` w ciągu `$str` [0 jeśli nie ma dopasowania, pierwszy znak ma pozycję 1]
`expr substr $str $pos $k`	wyciąga `$k` znaków z `$str` zaczynających się na pozycji `$pos` [pierwszy znak ma pozycję 1]
`expr match "$str" '\($sub\)'`	wyciąga `$sub*`, szukany od początku `$str`
`expr "$str" : '\($sub\)'`	wyciąga `$sub*` , szukany od początku `$str`
`expr match "$str" '.*\($sub\)'`	wyciąga `$sub*`, szukany od końca `$str`
`expr "$str" : '.*\($sub)'`	wyciąga `$sub*`, szukany od końca `$str`

*gdzie `$substr` i `$sub` są globami