Warunki i pętle

Poziom: podstawy **Dotyczy: bash 4+/5+** **Czas: ~20 min**

Skrypt bez warunków i pętli to lista komend wykonywanych zawsze w tej samej kolejności. Ta strona pokazuje, jak podejmujemy decyzje przez `if` i `case`, jak testujemy pliki i napisy, i jak iterujemy — po liście, po plikach i po liniach pliku, właściwie.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
ACL	Access Control List: lista, kto ma jakie prawa do pliku lub zasobu
ERE	Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

TYLKO ODCZYT

Warunek podstawowy: if / then / fi

```
if [[ -d /var/log/NazwaUslugi ]]; then
    echo "katalog logów istnieje"
fi
```

`if` uruchamia podaną komendę (tu: test `[[ ]]`) i wchodzi do bloku `then` dokładnie wtedy, gdy jej kod wyjścia to zero. Mechanizm ten opisujemy szczegółowo na stronie Kod wyjścia, $?, &&, ||, średnik — tutaj korzystamy z niego w praktyce.

TYLKO ODCZYT

Alternatywa: elif i else

```
if [[ -f /etc/redhat-release ]]; then
    echo "rodzina RHEL"
elif [[ -f /etc/debian_version ]]; then
    echo "rodzina Debian"
else
    echo "nierozpoznana dystrybucja"
fi
```

`elif` sprawdzamy tylko wtedy, gdy poprzedni warunek zawiódł — kolejność warunków ma znaczenie, bo pierwszy pasujący blok wygrywa i reszta się nie wykonuje.

RHEL = Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa

ZMIENIA SYSTEM

Wybór wielowariantowy: case

```
case "$1" in
    start|restart)
        echo "uruchamiam usługę"
        ;;
    stop)
        echo "zatrzymuję usługę"
        ;;
    *)
        echo "Użycie: $0 {start|stop|restart}" >&2
        exit 2
        ;;
esac
```

`case` dopasowuje jedną wartość do wielu wzorców naraz, w tym wzorców glob (`*.log`) i alternatyw rozdzielonych `|`. Każda gałąź kończy się `;;`, a `*)` jako ostatnia gałąź łapie wszystko, co nie pasowało wcześniej — odpowiednik `else` w `case`.

exit = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia

ZMIENIA SYSTEM

Test pliku lub katalogu: [[-f]], [[-d]], [[-e]]

```
[[ -e /etc/passwd ]] && echo "ścieżka istnieje"
[[ -f /etc/passwd ]] && echo "to zwykły plik"
[[ -d /etc ]] && echo "to katalog"
```

`-e` sprawdza istnienie czegokolwiek pod tą ścieżką, `-f` zawęża do zwykłych plików, `-d` do katalogów. Rozróżnienie ma znaczenie: `-e` na katalogu też zwróci prawdę, więc sam `-e` nie gwarantuje, że możemy plik odczytać jak plik.

TYLKO ODCZYT

Test uprawnień do pliku: [[-r]], [[-w]], [[-x]]

```
[[ -r /etc/shadow ]] || echo "brak prawa odczytu"
[[ -w /var/log/app.log ]] || echo "brak prawa zapisu"
[[ -x /usr/local/bin/skrypt.sh ]] || echo "brak prawa wykonania"
```

Testy sprawdzają uprawnienia efektywne bieżącego użytkownika, nie tylko bity w `ls -l` — działają poprawnie nawet przy skomplikowanej kombinacji właściciela, grupy i ACL.

ACL = Access Control List: lista, kto ma jakie prawa do pliku lub zasobu

TYLKO ODCZYT

Test rozmiaru pliku: [[-s]]

```
[[ -s /var/log/app.log ]] || echo "plik pusty albo nie istnieje"
```

Prawda, gdy plik istnieje i ma rozmiar większy od zera. Częste zastosowanie: sprawdzenie, czy poprzedni backup faktycznie coś zapisał, zanim uznamy zadanie za udane.

TYLKO ODCZYT

Porównanie dat modyfikacji: [[-nt]]

```
[[ konfiguracja.conf -nt konfiguracja.conf.bak ]] && echo "oryginał nowszy niż kopia"
```

`-nt` (*newer than*) porównuje czas modyfikacji dwóch plików. Używamy tego, żeby uniknąć niepotrzebnej pracy — na przykład przebudowy cache tylko wtedy, gdy plik źródłowy faktycznie się zmienił.

TYLKO ODCZYT

Test napisu pustego lub niepustego: [[-z]], [[-n]]

```
zmienna=""
[[ -z "$zmienna" ]] && echo "zmienna jest pusta"
[[ -n "$zmienna" ]] || echo "zmienna nadal pusta"
```

`-z` to prawda dla pustego napisu, `-n` dla niepustego. Cudzysłów wokół zmiennej jest tu obowiązkowy, żeby pusta wartość nie zniknęła z listy argumentów testu.

TYLKO ODCZYT

Porównanie napisów: =, !=, =~

```
srodowisko="produkcja"
[[ "$srodowisko" == "produkcja" ]] && echo "uwaga, środowisko produkcyjne"
[[ "$srodowisko" != "dev" ]] && echo "to nie jest dev"
[[ "cass01" =~ ^cass[0-9]+$ ]] && echo "nazwa pasuje do wzorca"
```

`==` i `!=` porównują napisy dosłownie (wewnątrz `[[ ]]` prawa strona `==` bez cudzysłowu jest też traktowana jako wzorzec glob). `=~` dopasowuje wyrażenie regularne rozszerzone (ERE) — temat rozwinięty na stronie o wyrażeniach regularnych.

ERE = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem

TYLKO ODCZYT

Test liczbowy: -eq, -lt, -ge

```
liczba_bledow=3
[[ "$liczba_bledow" -eq 0 ]] || echo "są błędy"
[[ "$liczba_bledow" -lt 5 ]] && echo "poniżej progu alarmowego"
[[ "$liczba_bledow" -ge 10 ]] && echo "przekroczono próg krytyczny"
```

Porównania liczbowe wymagają dedykowanych operatorów — `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`. Użycie `==` do porównania liczb zadziała przypadkiem tylko wtedy, gdy obie strony są identycznym napisem; poprawne porównanie liczbowe zawsze idzie przez `-eq` albo przez `(( ))`.

⚠ ZMIENIA SYSTEM

Wyrażenie arytmetyczne jako warunek: (( ))

```
liczba_prob=5
if (( liczba_prob > 3 )); then
    echo "za dużo prób"
fi
```

Wewnątrz `(( ))` piszemy zwykłe wyrażenia w stylu C, zmienne bez `$`. Kod wyjścia `(( ))` to 0 (prawda), gdy wynik wyrażenia jest niezerowy, i 1 w przeciwnym razie — dlatego `(( 0 ))` samo w sobie jest „fałszem" dla `if`.

✓ TYLKO ODCZYT

Pętla po liście wartości: for

```
for srv in cass01 cass02 cass03; do
    echo "sprawdzam $srv"
done
```

Klasyczna pętla `for` w bashu iteruje po liście słów, nie po zakresie liczb jak w innych językach — lista może pochodzić z wyliczenia wprost, z ekspansji nawiasów albo z globa.

✓ TYLKO ODCZYT

Pętla po plikach pasujących do wzorca

```
shopt -s nullglob
for plik in /var/log/*.log; do
    echo "log: $plik"
done
shopt -u nullglob
```

Glob rozwija się w powłoce, zanim pętla wystartuje, więc każda nazwa pliku trafia do zmiennej w całości, ze spacjami czy bez. `nullglob` zapobiega pułapce opisanej niżej, w sekcji „Częste błędy".

shopt = shell options: włącz albo wyłącz dodatkowe opcje zachowania powłoki bash

TYLKO ODCZYT

Pętla C-style po zakresie liczb: for (( ))

```
for (( i = 1; i <= 5; i++ )); do
    echo "próba numer $i"
done
```

Odpowiednik klasycznej pętli `for` ze składni C — trzy wyrażenia arytmetyczne oddzielone średnikami: inicjalizacja, warunek, krok.

TYLKO ODCZYT

Czytanie pliku linia po linii: while read -r

```
while IFS= read -r linia; do
    echo "linia: $linia"
done < /etc/hosts
```

To jedyny poprawny sposób przetwarzania pliku linia po linii w powłoce. `-r` wyłącza interpretację backslashy jako znaków ucieczki wewnątrz `read`, a `IFS=` przed `read` zapobiega ucinaniu wiodących i końcowych białych znaków z każdej linii.

IFS = Internal Field Separator: zmienna powłoki mówiąca, po jakim znaku dzielimy tekst na słowa, domyślnie spacja, tabulator i nowa linia

ZMIENIA SYSTEM

Pętla wykonywana, dopóki warunek jest fałszywy: until

```
proba=0
until systemctl is-active --quiet NazwaUslugi; do
    proba=$((proba + 1))
    [[ $proba -ge 10 ]] && { echo "usługa nie wstała po 10 próbach" >&2; exit 1; }
    sleep 2
done
```

`until` jest lustrzanym odbiciem `while` — pętla trwa, dopóki warunek zwraca kod niezerowy, i kończy się, gdy warunek w końcu zwróci zero.

sleep = wstrzymaj wykonywanie skryptu na wskazaną liczbę sekund

◫◫ OSTROŻNIE · NIEODWRACALNE

Przerwanie pętli: break

```
for plik in *.tmp; do
    [[ -e "$plik" ]] || break
    rm -- "$plik"
done
```

`break` natychmiast kończy najbliższą otaczającą pętlę, niezależnie od tego, ile jeszcze elementów zostało do przetworzenia.

◫◫ **rm** = remove: usuń (bez kosza, na zawsze)

◫◫ TYLKO ODCZYT

Pominięcie bieżącej iteracji: continue

```
for plik in *; do
    [[ -f "$plik" ]] || continue
    echo "przetwarzam plik: $plik"
done
```

`continue` przerywa bieżący przebieg i przechodzi od razu do następnej iteracji, pomijając resztę ciała pętli dla tego elementu.

◫◫ TYLKO ODCZYT

Wszystkie testy plikowe, napisowe i liczbowe

Test	Znaczenie
`-e SciezkaDoPliku`	ścieżka istnieje
`-f SciezkaDoPliku`	to zwykły plik
`-d SciezkaDoPliku`	to katalog
`-r`, `-w`, `-x`	prawo odczytu, zapisu, wykonania
`-s SciezkaDoPliku`	plik istnieje i ma rozmiar > 0
`plik1 -nt plik2`	`plik1` nowszy niż `plik2`
`-z "$napis"`	napis pusty

Test	Znaczenie
`-n "$napis"`	napis niepusty
`"$a" = "$b"`	napisy równe
`"$a" =~ wzorzec`	dopasowanie do ERE
`-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`	porównania liczbowe

Uważaj

- `[[ ]]` jest rozszerzeniem basha i nie działa w `#!/bin/sh` — tam potrzebny jest pojedynczy `[ ]` albo `test`.
- W `[ ]` porównanie liczbowe `=` zamiast `-eq` przejdzie bez błędu składni, ale porówna napisy, nie liczby — `[ "10" = "010" ]` jest fałszem, `[ 10 -eq 010 ]` też fałszem z innego powodu (wiodące zero to baza ósemkowa w kontekście arytmetycznym).
- `case` bez ostatniego wzorca `*)` po prostu nic nie robi, gdy żaden wzorzec nie pasuje — brak ostrzeżenia.

Z życia wzięte

Z ŻYCIA WZIĘTE

Środa, 6:00. Ola uruchamia na cass01 skrypt kopiujący zrzuty danych do archiwum, napisany z pętlą `for plik in $(ls /var/lib/cassandra/data)`. Nazwy plików zawierają spacje, więc pętla dzieli każdą z nich na kilka fałszywych przebiegów i kopiuje ułamki nazw zamiast całych plików. Dopiero zamiana na `while IFS= read -r plik; do ... done < <(find /var/lib/cassandra/data -type f)` kończy problem i przywraca kompletność archiwum.

Po co to istnieje

Skrypt, który zawsze robi to samo niezależnie od stanu systemu, jest przydatny tylko w najprostszych przypadkach. W praktyce chcemy: pominąć krok, jeśli usługa już działa; zareagować inaczej na trzy różne dystrybucje; przetworzyć każdy plik logu z katalogu bez wypisywania nazw z góry. Warunki i pętle to jedyny sposób, żeby skrypt reagował na rzeczywisty stan świata, a nie wykonywał ślepo z góry ustaloną sekwencję.

Jak to działa

Rozwiń: jak to działa pod spodem

> **Pojęcie: test.** Komenda, której jedynym zadaniem jest zwrócić kod wyjścia 0 (prawda) albo niezerowy (fałsz), bez żadnego efektu ubocznego. `[`, `[[ ]]` i `test` to trzy różne sposoby wywołania w gruncie rzeczy tego samego mechanizmu.

`[ ]` kontra `[[ ]]` kontra `test`

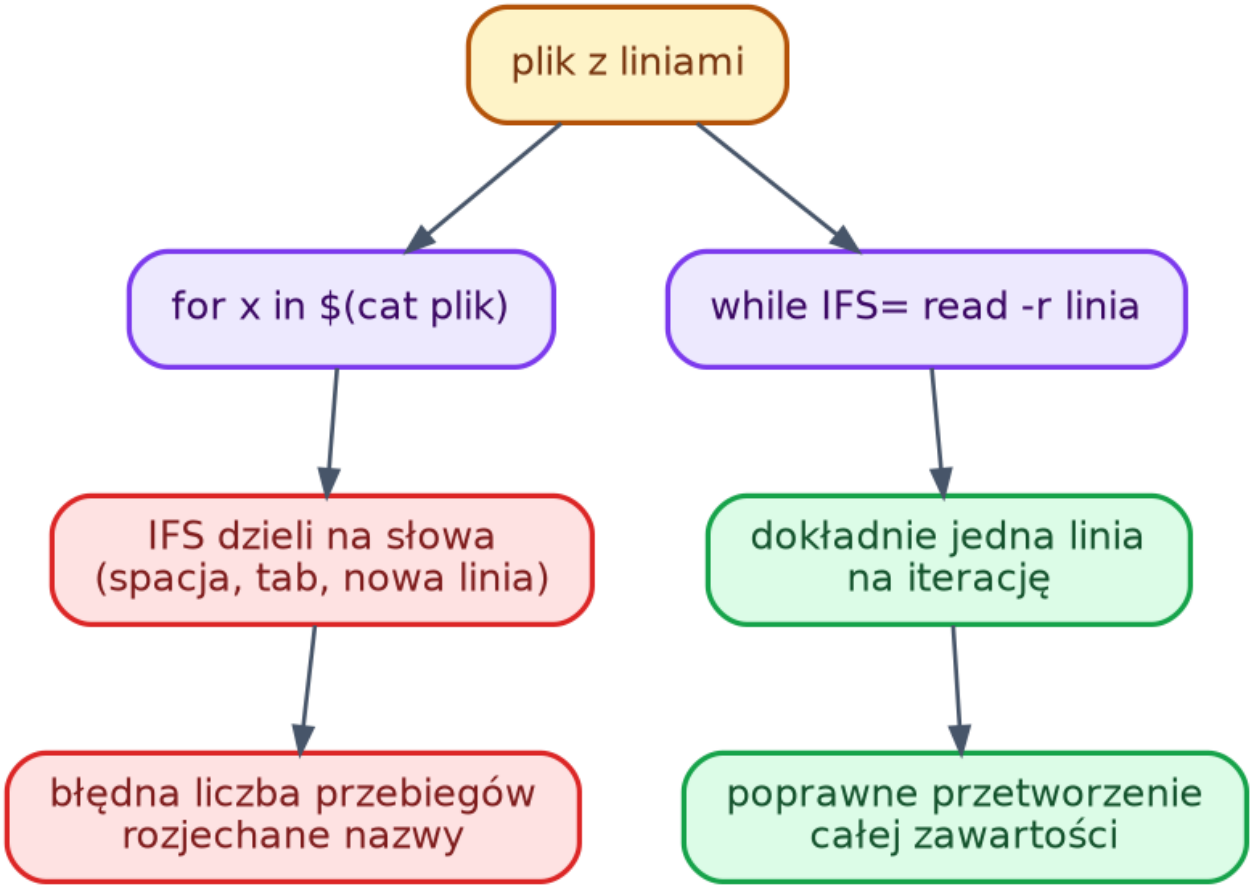
	`test wyrazenie`	`[ wyrazenie ]`	`[[ wyrazenie ]]`
Rodzaj	zewnętrzny program / wbudowana komenda	dokładny odpowiednik `test`, zamknięty `]`	słowo kluczowe powłoki (bash, ksh, zsh)
Cytowanie zmiennych	wymagane	wymagane	zalecane, ale bezpieczniejsze przy braku
Dopasowanie wzorca (`==`, `!=`)	nie	nie	tak, glob po prawej stronie
Dopasowanie regex (`=~`)	nie	nie	tak
Operatory logiczne wewnątrz nawiasów	`-a`, `-o` (przestarzałe)	`-a`, `-o` (przestarzałe)	`&&`, `
Przenośność do `/bin/sh`	tak	tak	nie (bashizm)

W bashu piszemy `[[ ]]`, bo eliminuje całą klasę błędów cytowania: `[[ -n $zmienna ]]` nie wybuchnie nawet bez cudzysłowu, gdy `$zmienna` jest pusta — parser `[[ ]]` zna liczbę argumentów z wyprzedzeniem, zamiast dowiadywać się jej dopiero po ekspansji, jak robi to

zwykły `[`. `[ -n $zmienna ]` przy pustej zmiennej rozwija się do `[ -n ]`, co `test` odczytuje jako sprawdzenie, czy napis `-n` jest niepusty — zawsze prawda, niezależnie od intencji. Tam, gdzie skrypt musi zostać przenośny do `/bin/sh` (na przykład skrypty inicjujące `initramfs`), wracamy do `[ ]` i pilnujemy cytowania ręcznie.

`case` a wzorce glob. Wzorce w `case` to te same reguły dopasowania co przy globbingu plików (`*`, `?`, `[abc]`), a nie wyrażenia regularne. `*.log)` pasuje do każdego napisu kończącego się na `.log`, `[Ss]top)` pasuje do `Stop` i `stop`.

Dlaczego `for line in $(cat plik)` jest błędem. To jest jeden z najczęściej powtarzanych błędów w skryptach pisanych przez osoby uczące się basha:



Ta sama zawartość pliku przetworzona przez `for $(cat plik)` kończy się rozjechanymi słowami, a przez `while read -r` poprawnie, linia po linii. (kliknięcie otwiera pełny rozmiar)

```
for linia in $(cat /etc/hosts); do
    echo "linia: $linia"
done
```

`$(cat plik)` podstawia całą zawartość pliku jako jeden ciąg znaków, a niecytowane podstawienie komendy przechodzi przez podział na słowa według `IFS` — domyślnie spacja, tabulator i **nowa linia jednocześnie**. Efekt: pętla iteruje po pojedynczych słowach, nie po liniach, a każda spacja w treści pliku tworzy dodatkowy, fałszywy przebieg pętli. Dodatkowo glob wewnątrz linii (`*`, `?`) zostanie rozwinięty względem bieżącego katalogu, czego nikt się nie spodziewa.

`while IFS= read -r linia; do ... done < plik` nie ma żadnej z tych wad: `read` czyta dokładnie jedną linię na iterację, `IFS=` przed `read` zachowuje wiodące i końcowe spacje w treści linii zamiast je obcinać, a `-r` sprawia, że backslash w pliku (na przykład w ścieżce Windows albo w wyrażeniu regularnym) zostaje zachowany dosłownie zamiast być zinterpretowany jako znak ucieczki.

Pułapka pustego globa. Gdy wzorzec w pętli `for plik in *.log` nie pasuje do żadnego pliku, bash domyślnie zostawia go dosłownie:

```
cd /tmp && mkdir -p pusty_katalog && cd pusty_katalog
for plik in *.log; do
    echo "przetwarzam: $plik"
done
```

```
przetwarzam: *.log
```

Pętla wykonuje jeden przebieg z literalnym `*.log`, mimo że żaden plik nie istnieje. `shopt -s nullglob` zmienia to zachowanie: niepasujący wzorzec rozwija się w zero elementów, więc pętla po prostu nie wykona się ani razu. W skryptach iterujących po plikach `shopt -s nullglob` ustawiamy na stałe przed pętlą, a `shopt -u nullglob` po niej, jeśli reszta skryptu ma wrócić do domyślnego zachowania.

Krok po kroku

1. Piszemy funkcję sprawdzającą typ pliku przez `case` na rozszerzeniu.

```
klasyfikuj() {
    case "$1" in
        *.log)  echo "log" ;;
```

```
        *.conf) echo "konfiguracja" ;;
        *)      echo "nieznany" ;;
    esac
}
klasyfikuj app.log
klasyfikuj nginx.conf
```

```
log
konfiguracja
```

2. Iterujemy po plikach katalogu z ochroną przed pustym globem.

```
shopt -s nullglob
for plik in /etc/*.conf; do
    echo "$(klasyfikuj "$plik"): $plik"
done
```

3. Czytamy plik konfiguracyjny linia po linii i pomijamy komentarze.

```
while IFS= read -r linia; do
    [[ "$linia" =~ ^#.*$ || -z "$linia" ]] && continue
    echo "ustawienie: $linia"
done < /etc/hosts
```

Częste błędy

`for line in $(cat plik)` zamiast `while read`.

Opisane wyżej w sekcji „Jak to działa" — pętla dzieli plik po słowach, nie po liniach, i rozwija glob wewnątrz treści. Naprawa: `while IFS= read -r linia; do ... done < plik`.

Zapomniany `nullglob` przy pętli po plikach.

```
for kopia in /srv/backup/*.tar.gz; do
    scp "$kopia" adm_user@cass01:/mnt/archiwum/
done
```

Katalog `/srv/backup` jest pusty. Pętla mimo to wykona jeden przebieg z `$kopia` ustawioną na literalne `*.tar.gz`, a `scp` zgłosi błąd o nieistniejącym pliku — ale tylko wtedy, gdy ktoś czyta logi. Bez `nullglob` błąd bywa cichy w skryptach, które przekierowują stderr donikąd.

`[ $zmienna = "wartosc" ]` bez cudzysłowu przy pustej zmiennej.

```
tryb=""
if [ $tryb = "produkcja" ]; then
    echo "produkcja"
fi
```

```
skrypt.sh: line 3: [: =: unary operator expected
```

Pusta, niecytowana zmienna znika z listy argumentów `[`, więc test dostaje `[ = "produkcja" ]` — o jeden argument za mało. Naprawa: `[[ "$tryb" = "produkcja" ]]`, gdzie `[[ ]]` jest odporne nawet bez cudzysłowu, albo jawne cudzysłowy w `[ ]`.

Liczba w cudzysłowie z wiodącym zerem w kontekście arytmetycznym.

```
godzina="08"
if (( godzina > 6 )); then
    echo "po szóstej"
fi
```

```
skrypt.sh: line 2: 08: value too great for base (error token is "08")
```

Wewnątrz `(( ))` liczba z wiodącym zerem jest interpretowana jako ósemkowa, a `8` i `9` nie są cyframi ósemkowymi. Naprawa: `(( 10#$godzina > 6 ))` wymusza bazę dziesiątkową, albo pozbywamy się wiodącego zera przy odczycie.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — pułapka `for` i `while read`

```
cd /tmp
printf 'linia jeden\nlinia z odstepami\nlinia trzy\n' > dane.txt

echo "=== for + \$(cat) ==="
for l in $(cat dane.txt); do echo "[$l]"; done
```

```
echo "=== while read ==="
while IFS= read -r l; do echo "[$l]"; done < dane.txt
```

Porównajmy liczbę wypisanych linii w obu wariantach.

Odpowiedź

Pierwszy wariant wypisze jedno słowo w nawiasach na każde słowo w pliku (siedem przebiegów), bo `$(cat dane.txt)` bez cudzysłowu przechodzi przez podział na słowa po spacjach i nowych liniach jednocześnie. Drugi wariant wypisze dokładnie trzy linie, każdą w całości — to jest poprawny sposób.

Ćwiczenie 2 — nullglob

```
mkdir -p /tmp/pusty && cd /tmp/pusty
for p in *.csv; do echo "znaleziono: $p"; done
shopt -s nullglob
for p in *.csv; do echo "znaleziono: $p"; done
shopt -u nullglob
```

Odpowiedź

Pierwsza pętla wypisze `znaleziono: *.csv`, mimo że katalog jest pusty — to jest domyślne, mylące zachowanie basha. Druga pętla nie wypisze nic, bo `nullglob` zamienia niepasujący wzorzec w zero elementów.

Pytania kontrolne

1. Dlaczego w bashu piszemy `[[ ]]` zamiast `[ ]`, skoro oba testują to samo?

Odpowiedź

`[[ ]]` jest słowem kluczowym parsowanym przez powłokę przed ekspansją, więc liczba argumentów jest znana z wyprzedzeniem — pusta czy niecytowana zmienna nie rozsypie testu tak, jak może rozsypać `[ ]`. Dodatkowo `[[ ]]` obsługuje dopasowanie wzorca (`==` z globem) i wyrażeń regularnych (`=~`), których `[ ]` nie ma. Cena to brak przenośności do `/bin/sh`.

2. Mamy pętlę `for plik in *.bak; do rm "$plik"; done` uruchamianą w katalogu, w którym czasem nie ma żadnego pliku `.bak`. Jakie mogą być dwa różne efekty braku `nullglob`?

Odpowiedź

Bez `nullglob`, gdy nie ma dopasowania, `$plik` przyjmuje dosłowną wartość `*.bak`, a `rm` próbuje usunąć plik o tej właśnie nazwie. Jeśli taki plik przypadkiem istnieje (dosłownie nazwany `*.bak`), zostanie skasowany niezamierzenie. Jeśli nie istnieje, `rm` zgłosi błąd „No such file or directory" — nieszkodliwy, ale mylący w logach, bo sugeruje awarię tam, gdzie po prostu nie było nic do zrobienia.

Powiązane: Kod wyjścia, $?, &&, ||, średnik · Cudzysłowy, escapowanie i ekspansja · Zmienne, parametry i argumenty · Funkcje · Wyrażenia regularne

Źródła: `man 1 bash` (sekcje *Compound Commands*, *CONDITIONAL EXPRESSIONS*) · `help test` · `help [[` · POSIX.1-2024

Revision #11
Created 2026-09-25 00:40:47 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:17 UTC by Tomasz Konieczny