

# Kurs administrowania systemem Linux

## Zajęcia nr 3: Powłoka systemowa

Instytut Informatyki Uniwersytetu Wrocławskiego

15 marca 2022

# Refleksje z wczorajszej pracowni

- Działanie skierowane na osiągnięcie celu nie jest dobre!
- Przede wszystkim należy zrozumieć i wyjaśnić!
- Nie „kopcie się”, tylko współpracujcie z powłoką!

## Wczoraj w skrypcie

```
program_1 < <(program_2)
```

# Refleksje z wczorajszej pracowni

- Działanie skierowane na osiągnięcie celu nie jest dobre!
- Przede wszystkim należy zrozumieć i wyjaśnić!
- Nie „kopcie się”, tylko współpracujcie z powłoką!

## Wczoraj w skrypcie

```
program_1 < <(program_2)
program_2 | program_1
```

## Jak lepiej?

```
program_1 &| program_2
program_1 2>&1 | program_2
```

## Idiomatyzmy basha

- Środowisko a podprocesy: używanie zmiennych w podprocesach
- Problemy podziału na słowa: spacje w nazwach plików
- Usuwanie pustych słów
- Błędy w trakcie wykonania skryptów
- Globy
- Przekierowania
- *Sourcing*

## Wywoływanie programów

- Opcje programów i getopt
- Dokumentacja: man i info

# Środowisko a podprocesy

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ MSG=World; bash script.sh
```

```
???
```

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ MSG=World; bash script.sh
```

```
Hello !
```

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ export MSG=World; bash script.sh
```

```
???
```

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ export MSG=World; bash script.sh  
Hello World!
```



# Środowisko a podprocesy

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ MSG=World bash script.sh
```

```
???
```

```
script.sh
```

```
echo "Hello $MSG!"
```

```
$ MSG=World bash script.sh
```

```
Hello World!
```

```
script.sh
```

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
```

```
$ bash script.sh
```

```
???
```

```
script.sh
```

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
```

```
$ bash script.sh
```

```
script.sh: line 1: cmd: command not found
```

```
script.sh
```

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
```

```
$ export -f cmd; bash script.sh
```

```
???
```

```
script.sh
```

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
```

```
$ export -f cmd; bash script.sh
```

```
Hello there!
```

```
script.sh
```

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }  
$ export -f cmd; bash script.sh  
Hello there!
```

- Zmienne i funkcje są przekazywane **do** podprocesu tylko wtedy, gdy są eksportowane.
- Nie ma możliwości przekazania zmiennych i funkcji **z** podprocesu.

# Przypisania zmiennych

```
$ DZIEWCZYINKA=Ala  
$ INWENTARZ="$DZIEWCZYINKA ma kota"  
$ echo $INWENTARZ
```



# Przypisania zmiennych

```
$ DZIEWCZYINKA=Ala
$ INWENTARZ="$DZIEWCZYINKA ma kota"
$ echo $INWENTARZ
Ala ma kota
$ DZIEWCZYINKA=Ola
$ echo $INWENTARZ
```

# Przypisania zmiennych

```
$ DZIEWCZYINKA=Ała
$ INWENTARZ="$DZIEWCZYINKA ma kota"
$ echo $INWENTARZ
Ała ma kota
$ DZIEWCZYINKA=Oła
$ echo $INWENTARZ
Ała ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:  
\$ MSG=Run ls; echo \$MSG

# Przypisania zmiennych

```
$ DZIEWCZYINKA=Ała
$ INWENTARZ="$DZIEWCZYINKA ma kota"
$ echo $INWENTARZ
Ała ma kota
$ DZIEWCZYINKA=Oła
$ echo $INWENTARZ
Ała ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:  

```
$ MSG="Run ls"; echo $MSG
```

# Przypisania zmiennych

```
$ DZIEWCZYINKA=Ała
$ INWENTARZ="$DZIEWCZYINKA ma kota"
$ echo $INWENTARZ
Ała ma kota
$ DZIEWCZYINKA=Oła
$ echo $INWENTARZ
Ała ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:  
\$ MSG="Run ls"; echo \$MSG
- Dołączanie na koniec: +=

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ((FAC *= N--))  
done  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
```

```
???
```

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ((FAC *= N--))  
done  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5  
Factorial of 5 equals 120
```

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ((FAC *= N--))  
done | cat  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
```

```
???
```

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ((FAC *= N--))  
done | cat  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5  
Factorial of 5 equals 1
```



# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ( ((FAC *= N--)) )  
done  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
```

```
???
```

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1  
FAC=1  
while ((N>1))  
do  
    ( ((FAC *= N--)) )  
done  
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
```

*zapętlenie*

# Zmienne a instrukcje wykonywane w podprocesach

```
fac.sh
```

```
N=$1
FAC=1
while ((N>1))
do
    ((FAC *= N--))
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
Factorial of 5 equals 120
```

Zasięg zmiennych w bashu nie jest zasięgiem leksykalnym!

# Względne i bezwzględne ścieżki do poleceń w skryptach

- Polecenie `cmd` w skrypcie zostanie skojarzone z tym, co aktualnie jest związane z tą nazwą (np. zostanie rozwinięty alias `cmd`, a następnie zostanie wykonane podstawione polecenie, np. polecenie wbudowane, funkcja lub program).
- Polecenie `/usr/bin/cmd` w skrypcie wykona zawsze program `/usr/bin/cmd`.
- Polecenie `command cmd` wykona polecenie wbudowane lub program (zgodnie z bieżącym ustawieniem zmiennej `PATH`). Alias nie zostanie wcześniej rozwinięty (bo polecenie `cmd` nie jest pierwszym tokenem w wierszu), a funkcje zostaną pominięte.
- Polecenie `builtin cmd` wykona polecenie wbudowane `cmd`.
- Problem właściwej ścieżki do programu.
- To samo dotyczy `#!`. Tu można zrobić tak:  
`#!/usr/bin/env bash -`
- Polecenie `type -a cmd` wypisze wszystkie możliwe skojarzenia nazwy `cmd`. Pierwsze znalezione to to, które będzie wykonane, jeśli wpiszemy takie polecenie.

# Parametryzowanie skryptu

## Konfiguracja na początku skryptu

```
RSYNC=/usr/bin/rsync
...
if [ -n "$DRY_RUN" ]
then
    CMD="echo rsync"
else
    CMD=$RSYNC
fi
$CMD -av --delete $FROM $TO
```

Dobre również podczas pracy interaktywnej:

## Sprawdź *zanim* wykonasz

```
$ for i in *.md; do echo mv $i $(basename $i .md).txt; done
```

# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
for FILE in ./*.txt  
do  
    echo "$FILE"  
done
```

```
$ bash text.sh
```

```
???
```

# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
for FILE in ./*.txt  
do  
    echo "$FILE"  
done
```

```
$ bash text.sh  
./spacje w nazwie pliku.txt
```

# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
for FILE in $(find . -name '*.txt' -print)
do
    echo "$FILE"
done
```

```
$ bash text.sh
```

```
???
```



# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
for FILE in $(find . -name '*.txt' -print)
do
    echo "$FILE"
done
```

```
$ bash text.sh
./spacje
w
nazwie
pliku.txt
```

# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
IFS=$'\n'  
for FILE in $(find . -name '*.txt' -print)  
do  
    echo "$FILE"  
done
```

```
$ bash text.sh  
./spacje w nazwie pliku.txt
```

# Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

```
text.sh
```

```
IFS=$'\n'  
for FILE in $(find . -name '*.txt' -print)  
do  
    echo "$FILE"  
done
```

```
$ bash text.sh  
./spacje w nazwie pliku.txt
```

Podział na słowa zależy od kontekstu obliczeń i zmiennej IFS.

# Rozwinięcia (przypomnienie)

- rozwinięcia nawiasów wąsatych, np. `file{1,2,3}`, `file{1..10}`,
- rozwinięcia tyldy, np. `~/Downloads/`,
- rozwinięcia zmiennych, np. `$HOME`,
- podstawienia instrukcji, np. `$(cat file.txt)`,
- podstawienia procesów, np. `<(pdftops file.pdf -)`,
- rozwinięcia arytmetyczne, np. `$((N+1))`,
- (powtórny) podział na słowa,
- rozwinięcia nazw plików (*globów*), np. `file?-*.txt`.

# Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

# Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

```
1 2 3 4 5
```

```
$ x=5; echo {1..$x}
```

# Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

```
1 2 3 4 5
```

```
$ x=5; echo {1..$x}
```

```
{1..5}
```

A jak spowodować rozwinięcie `$x`?

# Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

```
1 2 3 4 5
```

```
$ x=5; echo {1..$x}
```

```
{1..5}
```

A jak spowodować rozwinięcie `$x`?

```
$ x=5; eval echo {1..$x}
```



# Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

```
1 2 3 4 5
```

```
$ x=5; echo {1..$x}
```

```
{1..5}
```

A jak spowodować rozwinięcie `$x`?

```
$ x=5; eval echo {1..$x}
```

```
1 2 3 4 5
```

- W powyższym przykładzie widać, czemu przyjęto, że niepoprawne rozwinięcia nawiasów wąsatych, np. `{1..$x}`, pozostają niezmienione — mogą stać się poprawne później, po kolejnych rozwinięciach.

# Uwagi o problemie podziału na słowa

- Znaki oddzielające słowa opisuje zmienna IFS (*Internal Field Separator*). Domyślnie `$' \t\n'`.
- Podział na słowa następuje *po* wykonaniu rozwinięć *z wyjątkiem rozwinięć nazw plików*.
- Opcje `-print0` programu `find` i `-0` programów `xargs` i `parallel`.
- Znak `'\0'` nie działa w `$IFS` — jest terminatorem napisu.
- Opcja `-exec` programu `find` często pozwala wykonać całą pracę podczas wywołania `find`.
- W systemach plików `ext234` nazwa pliku jest ciągiem co najwyżej 255 bajtów różnych od `0x00` i `0x2f` (`'/'`).
- Znak `'\n'` jest dopuszczalny w nazwach plików!
- Nazwy plików w kodowaniu innym niż ASCII.
- Opcja `-b` programu `ls`.
- Konwencja zastępowania spacji znakiem podkreślenia.

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ bash param.sh a b c
```

```
???
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ bash param.sh a b c  
Number of parameters: 3  
Parameter 1: 'a'  
Parameter 2: 'b'  
Parameter 3: 'c'
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ bash param.sh a "" c
```

```
???
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ bash param.sh a "" c  
Number of parameters: 3  
Parameter 1: 'a'  
Parameter 2: ''  
Parameter 3: 'c'
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ X="" bash param.sh a $X c
```

```
???
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ X="" bash param.sh a $X c  
Number of parameters: 2  
Parameter 1: 'a'  
Parameter 2: 'c'
```



# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ X="" bash param.sh a "$X" c  
???
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ X="" bash param.sh a "$X" c
```

```
Number of parameters: 3
```

```
Parameter 1: 'a'
```

```
Parameter 2: ''
```

```
Parameter 3: 'c'
```

# Usuwanie pustych słów

```
param.sh
```

```
echo "Number of parameters: $#"  
i=1  
while (($# > 0))  
do  
    echo "Parameter $((i++)): '$1'"  
    shift  
done
```

```
$ X="" bash param.sh a "$X" c
```

```
Number of parameters: 3
```

```
Parameter 1: 'a'
```

```
Parameter 2: ''
```

```
Parameter 3: 'c'
```

Puste słowa są usuwane, chyba że są ujęte w cudzysłowy.

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh yes  
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh yes
```

```
The answer is affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh no  
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh no
```

```
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh  
???
```



# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh
```

```
answer.sh: line 1: [: =: unary operator expected
```

```
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ "$1" = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh  
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ "$1" = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh
```

```
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh "yes and no"  
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh "yes and no"  
test7.sh: line 1: [: too many arguments  
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh yes  
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [ $1 = yes ]  
then  
    echo "The answer is affirmative"  
else  
    echo "The answer is not affirmative"  
fi
```

```
$ bash answer.sh yes  
test7.sh: line 1: [yes: command not found  
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [[ $1 = yes ]]
then
    echo "The answer is affirmative"
else
    echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
???
```



# Puste argumenty poleceń

```
answer.sh
```

```
if [[ $1 = yes ]]
then
    echo "The answer is affirmative"
else
    echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
test7.sh: line 1: [[yes: command not found
The answer is not affirmative
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [[ $1 = yes ]]
then
    echo "The answer is affirmative"
else
    echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
???
```

# Puste argumenty poleceń

```
answer.sh
```

```
if [[ $1 = yes ]]
then
    echo "The answer is affirmative"
else
    echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
```

```
The answer is not affirmative
```

# Wyrażenia logiczne

- `test` i `[` to polecenia wbudowane odpowiadające programom `/usr/bin/test` i `/usr/bin/``[`. (Oba programy różnią się nieznacznie.)
- Ostatnim argumentem `[` powinien być `]`.
- Analiza składniowa wyrażenia logicznego jest wykonywana przez powyższe programy, a więc *po* wykonaniu rozwinięć przez `bash`.
- `[[...]]` jest konstrukcją składniową `bash`a.
- Analiza wyrażenia logicznego wewnątrz `[[...]]` jest wykonywana przez `bash` *przed* dokonaniem rozwinięć.
- Stare powłoki źle obsługiwały puste słowa.

## Typowy idiom w starych skryptach

```
if [ "x$1" = "xyes" ]  
then  
    ...
```

## Rodzaje błędów

- Wywołany program zwrócił niezerowy kod powrotu.
- Instrukcja basha zwróciła niezerowy kod powrotu.
- Skrypt basha zawiera błąd składniowy.

## W poprzednim przykładzie

- Komunikat o błędzie w wyrażeniu pochodził od programu [.
- Program [ zwrócił niezerowy kod powrotu.
- Z punktu widzenia basha skrypt był poprawny.

# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
false  
echo "Exit code of false: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
???
```

# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
false  
echo "Exit code of false: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh  
The next command will fail  
Exit code of false: 1  
Exit code of true: 0
```

# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
nonexistent  
echo "Exit code of nonexistent: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
???
```



# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
nonexistent  
echo "Exit code of nonexistent: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh  
The next command will fail  
fail.sh: line 2: nonexistent: command not found  
Exit code of nonexistent: 127  
Exit code of true: 0
```

# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
fi  
echo "Exit code of fi: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
???
```

# Błędy a niepowodzenia

```
fail.sh
```

```
echo "The next command will fail"  
fi  
echo "Exit code of fi: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
The next command will fail
```

```
fail.sh: line 2: syntax error near unexpected token 'fi'
```

```
fail.sh: line 2: 'fi'
```

# Błędy a niepowodzenia

```
fail.sh
```

```
set -o errexit  
echo "The next command will fail"  
false  
echo "Exit code of false: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
???
```

# Błędy a niepowodzenia

```
fail.sh
```

```
set -o errexit  
echo "The next command will fail"  
false  
echo "Exit code of false: $?"  
true  
echo "Exit code of true: $?"
```

```
$ bash fail.sh
```

```
The next command will fail
```

# Selektywna obsługa błędów

```
set +o errexit (default)
```

```
if wywołanie programu
```

```
then
```

```
    obsługa błędu, ew. exit nonzero
```

```
fi
```

Kodem powrotu instrukcji warunkowej jest kod powrotu ostatniej wykonanej instrukcji w gałęzi then bądź else.

Idiom (zapożyczony z Perla)

```
wywołanie programu || exit $?
```

Powyższy kod jest równoważny, ale bardziej zwięzły niż:

```
set -o errexit
```

```
wywołanie programu
```

```
set +o errexit
```

## Brak dopasowania w globach

- Domyślnie glob, który nie dopasowuje się do żadnej nazwy pliku rozwija się do siebie samego!
- Uwaga na globy w parametrach programów, np. `find -name *.txt`
- Powyższe polecenie działa dobrze, jeśli w *bieżącym* katalogu *nie ma* plików z rozszerzeniem `.txt`!
- Uwaga na argumenty polecenia `grep`!
- Problemy z testowaniem skryptów.

## Opcje globów

- `shopt [-s|-u] failglob` (powinna być domyślna!), `nullglob`, `nocaseglob` i `dotglob`.
- W kontekście `dotglob` nazwy `.` i `..` nie pasują do wzorca `*`.
- Nazwy `.` i `..` zawsze pasują do wzorca `.*`.
- Problem `rm -r .*` i jego rozwiązanie:  
`rm: refusing to remove '.' or '..' directory: skipping '..'`
- Brudne sztuczki: `.??*`. Niestety nie pasuje do `.a`.  
Lepiej `.[^.]*`, choć nie pasuje do `..a`.