Temat zajęć	Skrypty powłoki
Zakres materiału	Tworzenie i uruchamianie skryptów powłoki `bash`

Interpreter poleceń oraz zmienne środowiskowe

Interpreter poleceń, nazywany inaczej także *powłoką systemową (ang. system shell),* jest pośrednikiem pomiędzy użytkownikiem a funkcjami systemu operacyjnego. Powłoka systemowa pobiera dane i polecenia użytkownika i przekazuje je do wykonania do *jądra* systemu operacyjnego. Dostępnych jest wiele różnych powłok, z których najpopularniejsze wydają się następujące:

- interpreter Korna – uruchamiany poleceniem `ksh`,
- interpreter `tcsh`
- interpreter `zsh`
- najpopularniejszy i najpowszechniej obecnie stosowany `bash` (ang. *Bourne Again Shell*).

Wszystkie dalsze przykłady będą prezentowane właśnie dla powłoki `bash`.

Zmienne środowiskowe to wygodny i uniwersalny sposób konfigurowania i parametryzowania powłok systemowych i uruchamianych przez nie programów. Dostępne zmienne środowiskowe tworzą tzw. *środowisko wykonania procesu* – środowisko to jest kopiowane do wszystkich nowych procesów, a więc modyfikacje zmiennych wykonane w powłoce są widoczne we wszystkich programach uruchomionych przy użyciu tej powłoki. Każdy użytkownik może definiować dowolną liczbę własnych zmiennych oraz przypisywać im dowolne wartości. Aby zdefiniować zmienną środowiskową należy zastosować operator przypisania (znak "`=`") w następujący sposób:

```
ZMIENNA=wartość
```

W tym wypadku ciąg znaków *`ZMIENNA`* to nazwa zmiennej, a *`wartość`* to jej wartość – należy zwrócić uwagę, że pomiędzy nazwą zmiennej, operatorem przypisania i wartością **nie mogą wystąpić żadne białe znaki**. Odwołanie się do wartości zmiennej jest możliwe dzięki specjalnemu znakowi `$`; np. wyświetlenie wartości zmiennej na ekranie jest możliwe z wykorzystaniem polecenia `echo`, które wyprowadzana `stdout` linię tekstu oraz wartości zmiennej pobranej znakiem `$`:

```
$ SYSTEM=Linux
$ echo $SYSTEM
Linux
```

Polecenie powłoki o nazwie `set` pozwala wyświetlić wartości wszystkich zmiennych środowiskowych, a polecenie `unset` usuwa zmienną środowiskową, na przykład:

```
unset SYSTEM
```

Jak wspomniano, środowisko wykonania procesu jest przekazywane do procesów potomnych – jednak nie wszystkie zmienne powłoki muszą być przekazywane do uruchamianych programów. Te zmienne, które są przekazywane do środowiska potomkóœ, nazywa się *zmiennymi eksportowanymi*, a zmienne, które takiemu przekazaniu nie podlegają, nazywa się *zmiennymi lokalnymi*. Z reguły nowo tworzone zmienne są początkowo zmiennymi lokalnymi i niezbędne jest jawne wskazanie, że mają być zmiennymi eksportowanymi. Nazywa się to *eksportowaniem zmiennych* i jest realizowane przez poleceniem:

```
export lista_zmiennych
```

Oto przykład tworzenia zmiennej i jej eksportowania:

```
$ SYSTEM=Unix
$ export SYSTEM
```

Powyższe zlecenia można także zrealizować w jednym kroku:

```
$ export SYSTEM=Unix
```

Domyślnie w systemie jest wstępnie zdefiniowanych wiele zmiennych środowiskowych, oto znaczenie podstawowych z nich:

`HOME`	ścieżka i nazwa katalogu domowego użytkownika;
`USER`	nazwa zalogowanego użytkownika;
`PATH`	ścieżki poszukiwań programów;
`PS1`	postać prompta użytkownika;
`SHELL`	pełna ścieżka do domyślnego interpretatora poleceń użytkownika.

Skrypty i ich argumenty

Skrypty powłoki to pliki tekstowe, które zawierają ciągi poleceń dla powłoki systemowej. Współczesne powłoki pozwalają na to, aby skryty zawierały także typowe konstrukcje programistyczne, takie jak instrukcje warunkowe czy pętle, polecenia pozwalające na interaktywną prace skryptu czy też przekazywanie do skryptów argumentów ich wywołania. W ten sposób skrypty pozwalają kodować i wykonywać bardzo złożone czynności przy użyciu wyłącznie środków udostępnianych przez powłokę i narzędzia systemu, bez konieczności uciekania się do korzystania z języków programowania i ich środowisk.

Skrypty mogą być parametryzowane argumentami ich wywołania – oznacza to, że podczas uruchamiania skryptu można przekazać do niego dowolną liczbę dowolnych danych. Do argumentów wywołania można się odwoływać w skryptach za pomocą tzw. *zmiennych pozycyjnych*, oznaczanych jako cyfry dziesiętne od `1` do `9`. Każda zmienna pozycyjna przechowuje tekst przekazany za pośrednictwem odpowiadającemu jej argumentu wywołania – pierwszy argument dostępny jest w zmiennej pozycyjnej `1`, itd. Oto przykład pierwszego skryptu, który wyświetla wartości pierwszych trzech argumentów jego wywołania:

```
#!/bin/bash
# pierwszy skrypt
echo "argument nr 1: $1"
echo "argument nr 2: $2"
echo "argument nr 3: $3"
```

Takie polecenia należy zapisać do pliku o dowolnej nazwie (zaleca się jednak, aby pliki skryptów miały rozszerzenie `.sh`). Pierwsza linia zawiera tzw. *hash-bang*, który pozwala wskazać, jaki interpreter poleceń ma zostać wykorzystany do jego wykonania – w tym przypadku jest to powłoka `bash`. Druga linia ilustruje sposób zapisu komentarza: jest nim każda linia rozpoczynająca się od znaku `#`. Kolejne trzy linie wyświetlają wartości pierwszych trzech argumentów wywołania skryptu z wykorzystaniem polecenia `echo(1)`. Aby uruchomić taki skrypt wprost w linii poleceń, należy zawierającemu go plikowi nadać prawo wykonywania (`x`). Poniżej przedstawiono przykładowe wywołanie przedstawionego powyżej skryptu oraz jego wynik (przyjęto, że plik ze skryptem nazywa się `skrypt1.sh`):

```
./skrypt1.sh abc xyz 12345
argument nr 1: abc
```

```
argument nr 2: xyz
argument nr 3: 12345
```

Instrukcja warunkowa

Składnia instrukcji warunkowej w powłoce Bash jest następująca:

```
if warunek
then
      polecenie
      :
else
      polecenie
      :
fi
```

Gałąź rozpoczynająca się od `else` jest opcjonalna.

Wcięcia nie są konieczne, ale zaleca się ich stosowanie dla poprawienia czytelności kodu. Możliwe jest również zastąpienie przejścia do nowego wiersza znakiem średnika, co pozwala na zwięzłe zapisywanie prostszych partii kodu.

Warunek może być dowolnym poleceniem i w takim przypadku uznaje się, że jest spełniony, kiedy polecenie zwróciło kod powrotu zero. Zwrócenie niezerowego kodu powrotu powoduje, że warunek uznaje się za niespełniony.

Jeśli warunek ma przybrać postać wyrażenia zbliżonego do składni używanej w tradycyjnych językach programowania, należy posłużyć się poleceniem `test(1)`, które używa znaków "`[`" i "`]`" do wyróżnienia wyrażenia, którego wartość ma zostać wyznaczona. Na przykład:

```
if [ $1 = xyz ]
then
    echo "arg nr 1 = xyz"
else
    echo "arg nr 1 != xyz"
fi
```

Po znaku "`[`" i przed znakiem "`]`" konieczne jest umieszczenie co najmniej jednego znaku spacji.

Ten sam efekt można uzyskać zapisując kod w zwięźlejszy sposób:

```
if [ $1 = xyz ]; then echo "arg nr 1 = xyz"; else echo "arg nr 1 != xyz"; fi
```

Najczęściej polecenie `test` używa się do wykonania następujących sprawdzeń:

Warunek	*Opis*
`znaki1 = znaki2`	Weryfikacja równości dwóch łańcuchów znaków
`znaki1 != znaki2`	Weryfikacja nierówności dwóch łańcuchów znaków
`-z znaki`	Weryfikacja, czy łańcuch znaków ma zerową długość
`-n znaki`	Weryfikacja, czy łańcuch znaków ma niezerową długość
`liczba1 -eq liczba2`	Weryfikacja równości dwóch liczb
`liczba1 -ne liczba2`	Weryfikacja nierówności dwóch liczb
`liczba1 -gt liczba2`	Weryfikacja, czy `liczba1` jest większa od `liczba2`
`liczba1 -lt liczba2`	Weryfikacja, czy `liczba1` jest mniejsza od `liczba2`
`-e nazwa`	Weryfikacja, czy podany plik istnieje
`-f nazwa`	Weryfikacja, czy podany plik jest plikiem zwykłym
`-d nazwa`	Weryfikacja, czy podany plik jest katalogiem
`-r nazwa` **`-w nazwa`** **`-x nazwa`**	Weryfikacja, czy użytkownik ma prawo, odpowiednio, odczytu, zapisu i wykonywania dla pliku o podanej nazwie
`warunek1`* `-a` *`warunek2`	Iloczyn logiczny warunków
`warunek1`* `-o` *`warunek2`	Suma logiczna warunków
`!` *`warunek1`*	Negacja warunku

Pętle

Skrypty powłoki mogą także zawierać pętle – podstawowe dwie z nich to `for` oraz `while`.

Pętla `for` wykonywana jest z góry określoną liczbę razy, a jej ogólna składnia jest następująca:

```
for zmienna in lista
do
      polecenie
      :
done
```

albo (w wersji zwięzłej):

```
for zmienna in lista; do polecenie; ...; done
```

Wykonanie pętli powoduje przypisywanie zmiennej *zmienna* kolejnych wartości wymienionych na liście *lista*; liczba iteracji jest zatem zależna od długości podanej listy. Jako listę można pętli `for` można podawać wzorce uogólniające powłoki. Poniższy przykład skryptu prezentuje zastosowanie pętli `for` do usunięcia wszystkich plików z rozszerzeniem `.tmp` z katalogu bieżącego:

```
for FILE in *.tmp ; do rm -v $FILE; done
```

Liczba iteracji powyższej pętli będzie zatem determinowana liczbą plików z rozszerzeniem `*.tmp`, które utworzą listę wartości dla zmiennej `FILE`. *******

Realizacja pętli numerycznej z zastosowaniem pętli `for` jest możliwa z użyciem programu `seq(1)`, który wypisuje kolejne liczby, np.:

```
for N in `seq 1 10` ...
```

spowoduje dziesięciokrotne wykonanie pętli.

Pętla `while` pozawala na realizację pętli, dla których liczba iteracji nie jest znana z góry, jej składnia dla skryptów powłoki jest następująca:

```
while warunek
do
   instrukcje do wykonania
done
```

Warunek może być dowolnym poleceniem i najczęściej jest konstruowany – tak jak w przypadku instrukcji warunkowej – z zastosowaniem programu *test*.

Przykładem zastosowania pętli `while` może być skrypt wypisujący na ekranie wartości wszystkich argumentów wywołania skryptu (niezależnie od ich liczby). Pętla taka będzie wykorzystywała polecenie `shift(1)`, które powoduje przesunięcie argumentów w lewo (tzn. argument numer 2 staje się argumentem nr 1, numer 3 staje się numerem 2, etc) – oto skrypt:

```
while [ -n "$1" ]
do
    echo $1
    shift
done
```

lub (w zapisie równoważnym):

```
while [ -n "$1" ] ; do echo $1 ; shift ; done
```

Warunkiem wykonania pętli jest sprawdzenie, czy pierwszy argument wywołania skryptu ma niezerową długość – jeśli skrypt został uruchomiony bez żadnych argumentów, to pętla nie zostanie wykonana. Jeśli natomiast skrypt został wykonany z argumentami, to w pierwszym wykonaniu pętli zostanie wyświetlona wartość pierwszego argumentu, a następnie nastąpi przesunięcie argumentów w lewo poleceniem `shift` (drugi argument stanie się pierwszym, trzeci drugim itd.). Pętla zakończy się jeśli zostaną wyświetlone i przesunięte wszystkie argumenty (zmienna `$1` będzie miała wówczas zerową długość).

Obie zaprezentowane pętle mogą zostać przerwane poleceniem `break(1)` – oto przykład zastosowania przerywania pętli:

```
#!/bin/bash
for FILE in *.tmp
do
       if [ ! -f $FILE ]
       then
             echo "$FILE nie jest plikiem!"
             break
       fi
       rm -v $FILE
done
```

Pętla **`for`** zostanie przerwana, jeśli nazwa pliku z rozszerzeniem `*.tmp` nie będzie wskazywała na plik zwykły.

Pobieranie wartości do skryptów oraz ich debugowanie

Jeśli skrypt wymaga iteracji z użytkownikiem, to niezbędne staje się pobieranie wartości przekazywanych przez użytkownika. Służy do tego polecenie:

```
read argumenty
```

Argumentami są nazwy zmiennych środowiskowych, które przyjmą wartość odczytana ze standardowego wejścia (do napotkania znaku nowej linii). Jeśli jako argumenty podano kilka zmiennych, to są one inicjowane w ten sposób, że pierwsze słowo trafia do pierwszej zmiennej, drugie do drugiej itd. Polecenie to można przetestować wykonując następujące polecenia:

```
read X Y uzytkownik adam
echo $X uzytkownik
echo $Y adam
```

Skrypty mogą być także wykonywanie w trybie debugowania, np. w celu testowania poprawności działania, warunków, pętli itp. Aby zrealizować wykonanie skryptu z wyświetlaniem informacji kontrolnych, należy zastosować przełącznik `-x` wywołania interpretatora poleceń. Można to zrealizować na dwa sposoby: po pierwsze można dopisać tenże przełącznik w pierwszej linii skryptu:

```
#!/bin/bash -x
```

albo też można uruchomić skrypt wywołując go poprzez wskazanie interpretatora z przełącznikiem i nazwą skryptu jako argumentem; przyjmując, że skrypt posiada nazwę `skrypt.sh` uruchomienie miałoby następującą postać:

```
bash -x skrypt.sh
```

Informacje dodatkowe

Składnia skryptów powłoki umożliwia także wykonywanie operacji arytmetycznych, np.:

```
a=1
a=$[a+1]
```

```
a=$((a+1))
let "a=a+1"
let a=a+1
```

Dopuszcza się użycie tablic jednowymiarowych. Składnia odwołania się do elementu tablicy ma postać:

```
tablica[nr_elementu]=wartość
```

Na przykład:

```
nazwa_tablicy[0]="Jan"
nazwa_tablicy[1]="Ewa"
nazwa_tablicy[2]="Tomek"
nazwa_tablicy[3]="Rysiek"
```

Odczytanie danych elementu nr 2:

```
echo ${nazwa_tablicy[2]}
```

Odczytanie całej tablicy:

```
echo ${ nazwa_tablicy[*]}
```

Odczytanie liczby wszystkich elementów tablicy:

```
echo ${#nazwa_tablicy[*]}
```

Instrukcja `case` pozwala na dokonanie wyboru wielowariantowego. Wartość zmiennej porównywana jest z kolejno podanymi wzorcami. Jeśli wzorzec ma taką samą wartość jak zmienna, wówczas wykonywane są polecenia przypisane do tego wzorca. Jeśli nie uda się znaleźć dopasowania, może zostać wykonane polecenie domyślne, oznaczone symbolem * (gwiazdka). Polecenie domyślne warto umieszczać w instrukcji `case` **zawsze**, co może zabezpieczać przed błędami popełnionymi przez użytkownika.

Składnia:

```
case zmienna in
```

```
  "wzorzec1") polecenie1 ;;
  "wzorzec2") polecenie2 ;;
  "wzorzec3") polecenie3 ;;
  *) polecenie_domyślne
esac
```

Przykład:

```
#!/bin/bash
echo "Podaj cyfre (1 lub 2):"
read d
case "$d" in
  "1") echo "Cyfra jeden" ;;
  "2") echo "Cyfra dwa" ;;
  *) echo "Błędna dana"
esac
```

` (akcent lub odwrotny apostrof)

Znaki akcentu umożliwiają wykonanie polecenia i wstawienie wyników jego działania. Znak akcentu znajduje sie na klawiaturze na lewo od znaku '!'. Na przykład, polecenie

```
cat /etc/passwd>`hostname`.txt
```

spowoduje najpierw wykonanie polecenia `hostname` (ujętego w akcenty), zastąpienie napisu ujętego w akcenty wynikiem działania polecenia i w efekcie wykonanie polecenia

```
cat /etc/passwd>superserver.com.pl.txt
```

' (apostrof)

Znaki apostrof służą do dosłownego cytowania tekstu. Cytowanie napisu zawierającego wiele spacji przy użyciu znaku '\' (*backslash*) jest niewygodne. Stosując apostrofy możemy pisać

```
echo 'Ala ma kota'
```

Wszystkie znaki cytowane w apostrofach pozostają niezmienione

Przykład:

```
#porownaj ponizsze instrukcje echo
KAT=`pwd`
echo '$KAT'
echo $KAT
```

" (cudzysłów)

Cudzysłowy, podobnie jak apostrofy, służą do cytowania tekstu. Różnica w stosunku do apostrofów polega na tym, że w cytowanym napisie nazwy zmiennych zostaną zamienione ich wartościami:

```
DZIS=`date`
echo "Dzisiaj jest: $DZIS"
echo 'Dzisiaj jest: $DZIS'
```

Znaki cudzysłowu mają istotne znaczenie, gdy piszemy instrukcje warunkowe lub iteracje.

Instrukcja

```
if [ $1 ]
then
    echo podano parametr
fi
```

jest błędna. Jeśli parametr `$1` nie został podany, wówczas interpretator zgłosi błąd składniowy, gdyż (po podstawieniu pustej zmiennej) będzie próbował wykonać następujący kod:

```
if [ ]
then
    echo podano parametr
fi
```

Cudzysłów zabezpiecza nas przed takim błędem, gdyż nawet jeśli parametr nie został podany, to instrukcja

```
if [ "$1" ]
then
    echo podano parametr
fi
```

po podstawieniu wartości przyjmie postać:

```
if [ "" ]
then
  echo podano parametr
fi
```

co jest konstrukcją w pełni poprawną.

Zmienne używane w skryptach wykorzystujących parametry

`$0`	nazwa wywołanego skryptu
`$1`	pierwszy parametr
`.....`	
`$9`	dziewiąty parametr
`$@`	lista wszystkich parametrów
`$*`	wszystkie parametry złączone w jeden łańcuch znaków
`$#`	liczba parametrów
`$?`	stan ostatnio wykonanego polecenia (0 - poprawnie, nie 0 - błędnie)
`$$`	numer procesu aktualnie wykonywanej powłoki
`$!`	numer procesu ostatnio wykonywanego w tle

Zadania do wykonania w czasie zajęć

Napisz skrypt, który:

1. po uruchomieniu będzie czekał na wprowadzenie działania **`A @ B`**, gdzie **`A`** i **`B`** to liczby, a **`@`** to operator: +, -, / lub *, po czym będzie wyświetlał wynik wprowadzonego działania;

2. będzie przyjmował dowolną liczbę argumentów wywołania będących liczbami całkowitymi, sumował je i na końcu wyświetlał wynik;
3. będzie przyjmował dowolną liczbę argumentów wywołania i wyświetlał je w odwrotnej kolejności (wykorzystaj polecenie `tac`);
4. będzie przyjmował dowolną liczbę argumentów liczbowych i wyświetlał je posortowane;
5. wyświetli najkrótszą i najdłuższą nazwę użytkownika w systemie (nazwa użytkownika jest pierwszym polem w pliku `/etc/passwd`);
6. porówna zawartość pliku o nazwie podanej jako pierwszy argument wywołania z plikami, których nazwy znajdują się w kolejnych liniach pliku o nazwie podanej jako drugi parametr wywołania (porównanie wykonać za pomocą polecenia `cmp`);
7. będzie przyjmował dowolną liczbę argumentów będących nazwami katalogów i wyświetlał dla każdego katalogu jego nazwę oraz liczbę znajdujących się w nim plików nieodczytywalnych (bez praw odczytu);
8. w katalogu podanym jako jego argument wywołania zmieni nazwy wszystkich plików i katalogów, tak aby litery małe zostały zamienione na wielkie, a wielkie na małe (czyli np. z `RazDwa.txt` na `rAZdWA.TXT`);
9. zamieni rozszerzenia nazw plików w podanym katalogu z podanych na inne (3 argumenty: katalog, stare rozszerzenie, nowe rozszerzenie). Uwzględnić możliwość pojawienia się w nazwie pliku kropek (w zadaniu można wykorzystać polecenie `rev`);
10. sprawdzi, czy wybrany użytkownik (login podany jako argument wywołania skryptu) jest aktualnie zalogowany w systemie; w przypadku niepowodzenia skrypt powinien zaczekać na zalogowanie użytkownika i wtedy poinformować o ewentualnym powodzeniu;
11. jako argumenty wywołania (dowolna liczba argumentów) będzie przyjmował loginy użytkowników systemu; skrypt ma sprawdzić, czy któryś z podanych użytkowników jest aktualnie zalogowany w systemie więcej niż raz; dla każdego takiego użytkownika system ma wyświetlić jego login oraz liczbę logowań;
12. przyjmuje jako argument nazwę programu (np. `nano`); skrypt powinien monitorować system, wyświetlając co sekundę informację o tym, którzy użytkownicy używają w danej chwili programu o podanej jako argument nazwie;
13. pobiera jako argumenty wywołania loginy użytkowników (może być podany więcej niż jeden login) i sprawdzający ile instancji edytora `nano` poszczególni użytkownicy mają uruchomione; w jednej linii wyświetlamy login i liczbę uruchomionych edytorów;
14. szyfruje plik za pomocą szyfru Cezara (przesunięcie liter alfabetu o stałą wartość); jako argument podajemy przesunięcie (liczba od 0 do 26) oraz nazwę pliku, który ma zostać

zaszyfrowany; wynik powinien być zapisany do nowego pliku o takiej samej nazwie jak plik oryginalny jednak z dodanym rozszerzeniem `.sec` (użyć polecenia `tr`);

15. będzie obliczał *n-ty* element ciągu Fibonacciego (*n* podawane jako argument wywołania skryptu):

    $f_0 = 0;$

    $f_1 = 1;$

    $f_n = f_{n-2} + f_{n-1} \quad dla \quad n > 1;$

    *s*krypt ma działać **rekurencyjnie**, odwołując się do samego siebie; na ekranie powinien pojawić się jedynie ostateczny wynik (bez wyników pośrednich);
16. jest iteracyjną wersją skryptu z poprzedniego zadania.