ADMINISTRACJA SYSTEMAMI LINUX

Programowanie w powłoce bash

Andrzej CHMIELEWSKI
Ireneusz MROZEK
Eugenia BUSŁOWSKA

Politechnika Białostocka

ADMINISTRACJA SYSTEMAMI LINUX
PROGRAMOWANIE W POWŁOCE BASH

Andrzej Chmielewski
Ireneusz Mrozek
Eugenia Busłowska

Politechnika Białostocka
OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2023

Recenzent:
dr inż. Andrzej Sawicki

Redaktor naukowy dyscypliny informatyka techniczna i telekomunikacja:
prof. dr hab. Jarosław Stepaniuk

Korekta językowa:
Edyta Chrzanowska

Okładka:
Marcin Dominów

ISBN 978-83-67185-70-7
ISBN 978-83-67185-70-7 (eBook)

Oficyna Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
e-mail: oficyna.wydawnicza@pb.edu.pl
www.pb.edu.pl

Spis treści

Wstęp

W każdym systemie operacyjnym udostępniany jest interfejs, w postaci tekstowej lub graficznej, umożliwiający aplikacjom oraz użytkownikom komunikację z nim. Funkcję takiego pośrednika pełni tzw. powłoka systemowa (ang. *shell*), która służy do zarządzania całym systemem operacyjnym, w tym m.in. do uruchamiania poleceń wbudowanych (systemowych) oraz zewnętrznych (aplikacji), wykonywania operacji w systemie plików, zarządzania usługami etc. W przypadku wielu popularnych systemów operacyjnych, szczególnie tych rozwijanych na licencjach wolnego oprogramowania (np. Linux), użytkownik ma możliwość swobodnego wyboru powłoki spośród wielu dostępnych, kierując się m.in. swoimi preferencjami, udostępnianymi funkcjami, poleceniami oraz ich składnią. W pozostałych systemach użytkownikom zwykle dostarczana jest jedna, oficjalna powłoka, choć warto zauważyć, że ta sytuacja zmienia się w ostatnich latach i najczęściej możliwa jest instalacja dodatkowych powłok.

Ponieważ pierwsze systemy operacyjne miały jedynie interfejs tekstowy, tylko w tej postaci były dostępne również powłoki systemowe. W zależności od systemu nazywane są m.in. terminalem, wierszem poleceń. Umożliwiają one zarówno uruchamianie zewnętrznych aplikacji, jak i same dostarczają często bardzo bogatych zestawów poleceń wbudowanych, zapewniających elastyczną komunikację z systemami operacyjnymi i zarządzanie nimi. Ich niewątpliwą zaletą jest możliwość tworzenia skryptów automatyzujących

wykonywanie nawet bardzo skomplikowanych działań, przez co są niezwykle przydatnym i popularnym narzędziem, szczególnie wśród administratorów.

Wraz z rozwojem systemów operacyjnych z graficznym interfejsem użytkownika ich główne powłoki, w pewnym uproszczeniu, były głównie menadżerami plików oraz panelem konfiguracyjnym. Taki rodzaj komunikacji jest intuicyjny i wygodny z punktu widzenia przeciętnego użytkownika, który niekoniecznie oczekuje dostępu do różnego rodzaju zasobów z poziomu terminala oraz zarządzania systemem, a w wielu przypadkach korzysta z bardzo skomplikowanych składni poleceń. Warto jednak zauważyć, że nawet w tego rodzaju systemach niemal zawsze był dostarczany również interfejs tekstowy, aby zaawansowani użytkownicy mogli w pełni korzystać z systemu, choćby poprzez uruchamianie zestawów poleceń w trybie wsadowym. W ostatnich latach można zauważyć, że zalety powłok tekstowych zostały docenione w wielu nowoczesnych systemach. Można tu wspomnieć m.in. o możliwości instalacji w systemach Windows powłok znanych z systemów Linux.

W systemach Linux dostępne są liczne powłoki tekstowe (np. `bash`, `csh`, `tcsh`, `ksh`, `zsh`) i graficzne. Użytkownicy mogą swobodnie nimi zarządzać, instalując je w zależności od potrzeb. Jednakże najpopularniejszą z nich jest powłoka `bash`, której pierwowzorem była znana z systemów UNIX-owych powłoka `Bourne shell`. Ze względów licencyjnych została ona „przepisana" przez Linusa Torvaldsa (twórcę Linuksa), aby mogła być wydawana na licencji GPL. Powłoka ta oferuje bogaty zestaw poleceń wbudowanych oraz interpretera umożliwiającego wykonywanie nie tylko złożonych operacji na zasobach systemowych, lecz także działań na wartościach zarówno numerycznych, jak i alfanumerycznych.

Możliwości przez nią oferowane są na tyle szerokie, że przekraczają zakres tego skryptu. W publikacji przedstawione zostaną funkcjonalności wystarczające do przygotowania większości skryptów wykorzystywanych w ramach przedmiotu *Administracja systemami Linux*.

Rozdział 1

Wprowadzenie

1.1 Struktura

Skrypty składają się z zestawu poleceń zapisanych w pliku tekstowym, które mogą być również wykonywane z poziomu terminala. W przypadku tych samych zestawów operacji warto je umieścić w pliku, który następnie może być używany wielokrotnie. Uwzględniając dodatkowo możliwość parametryzowania ich wykonania, dostajemy narzędzie bardzo elastyczne, umożliwiające automatyzację wielu czynności administracyjnych.

Na początku skryptu, przed jego pierwszą interpretowalną linią, warto określić ścieżkę do interpretera, która w przypadku powłoki `bash` może mieć postać:

```
#!/bin/bash
```

Dyrektywa ta wskazuje na interpreter, który będzie użyty do uruchomienia skryptu. W przypadku jej braku uruchomi się on w bieżącej powłoce. Wówczas taki skrypt, przygotowany do pracy w `bash`, po uruchomieniu w in-

nej powłoce (np. `tcsh`) spowoduje wygenerowanie błędu przez interpreter np. ze względu na odmienną składnię.

Skrypty składają się z zestawu sekwencyjnie wykonywanych poleceń. Przykładowy ich schemat został przedstawiony na listingu 1.1.

Listing 1.1: Schemat skryptu w powłoce `bash`

```
#!/bin/bash

polecenie_1
polecenie_2
 ...
polecenie_n
```

Jak można zauważyć na powyższym przykładzie, poszczególne polecenia są zwykle zapisywane w oddzielnych liniach. Można je również zapisywać w jednej linii. Wówczas każde z poleceń (oprócz ostatniego) należy oddzielić średnikiem, a ewentualne spacje przed znakiem separatora i po nim nie mają znaczenia (patrz listing 1.2).

Listing 1.2: Przykładowy zapis sekwencji poleceń zapisanych w pojedynczej linii w skrypcie

```
#!/bin/bash

polecenie_1; polecenie_2 ; ... ;polecenie_n
```

W najprostszej postaci skrypt może przedstawiać się tak jak na listingu 1.3, w którym do wypisywania komunikatu na standardowym wyjściu zostało wykorzystane polecenie `echo`.

Listing 1.3: Przykładowy prosty skrypt w `bash`

```
#!/bin/bash

echo "Witaj w powloce bash."
```

Skrypty możemy przygotowywać w dowolnym edytorze tekstowym zarówno w naturalnym środowisku tekstowym, np. `vi`, `vim`, `nano`, `pico`, jak i w środowisku graficznym, np. `gedit`. Każdy z nich dostarcza pewnych funkcjonalności (np. kolorowanie składni), które ułatwiają edycję skryptów. Zaleca się jednak, aby administratorzy potrafili wykonać podstawowe działania na plikach tekstowych z wykorzystaniem edytora `vi` (ewentualnie `vim`), który dostępny jest w każdej dystrybucji, również w przypadku tych służących do naprawy systemu. Tak więc w sytuacjach awaryjnych znajomość obsługi wspomnianych edytorów może się okazać niezbędna, a tworzenie w nich skryptów pozwoli na nabycie tych umiejętności.

1.2 Uruchamianie

Przy tworzeniu nowego pliku (np. za pomocą polecenia `touch`) zwykle nie są mu nadawane prawa do uruchomienia i w konsekwencji próba jego wykonania zakończy się niepowodzeniem, np.:

```
$ touch skrypt.sh
$ ./skrypt.sh
./skrypt.sh: Brak dostępu
```

Dlatego też jednorazowo, przed jego pierwszym uruchomieniem, należy ustawić mu prawa za pomocą polecenia `chmod`, np.:

```
$ chmod +x skrypt.sh
```

Tak przygotowany skrypt można uruchamiać na wiele sposobów, używając zarówno ścieżek względnych, jak i bezwzględnych, np.:

```
$ ./skrypt.sh               # gdy skrypt znajduje się w bieżącym katalogu
$ /home/student/skrypt.sh   # ścieżka bezwzględna
$ ../data/skrypt.sh         # ścieżka względna
```

W przypadku uruchamiania pliku znajdującego się w bieżącym katalogu należy pamiętać, że nazwę skryptu należy poprzedzić sekwencją "`./`". Próba wykonania polecenia:

```
$ skrypt.sh
```

prawdopodobnie zakończy się błędem:

```
skrypt.sh: nie znaleziono polecenia
```

co jest spowodowane faktem, że zmienna środowiskowa `PATH` zwykle nie zawiera ścieżki do katalogu, w którym się on znajduje, i stąd skrypt nie mógł być uruchomiony. Co więcej, zakładając, że zmienna `PATH` zawiera jednak ścieżkę do tego katalogu, skrypt nadal niekoniecznie się uruchomi. Dzieje się tak dlatego, że system operacyjny wyszukuje skrypt (lub aplikację) o określonej nazwie w kolejnych ścieżkach zawartych w zmiennej `PATH`. Tak więc np. w przypadku istnienia dwóch następujących skryptów:

`/usr/bin/skrypt.sh`
`/home/student/skrypt.sh`

oraz przy założeniu, że zmienna `PATH` zawiera zarówno ścieżkę *`/usr/bin`*, jak i *`/home/student`* (dokładnie w takiej kolejności), uruchomiony będzie pierwszy z nich, bez względu na bieżący katalog.

Innym sposobem uruchomienia skryptu, niewymagającym zmiany praw dostępu, jest podanie ścieżki do skryptu jako parametru polecenia `bash`. W tym przypadku, gdy uruchamiany jest skrypt z bieżącego katalogu, ciąg "`./`" można pominąć, np.:

```
$ bash skrypt.sh
$ bash /home/student/skrypt.sh
$ bash ../skrypt.sh
```

Ten sposób jest również często wykorzystywany do uruchamiania skryptów z poziomu powłoki innej niż obecnie używana.

Możliwości uruchamiania skryptów jest bardzo wiele i trudno byłoby je wszystkie wymienić. Niektóre z nich pojawią się w dalszej części podręcznika, m.in. podczas omówienia sposobów przekazywania parametrów, jak również śledzenia ich wykonywania. Można np. użyć różnych opcji polecenia `bash`, z których najpopularniejsza to `-i`, pozwalająca na uruchomienie skryptu w sposób interaktywny:

```
$ bash -i skrypt.sh
```

W powyższym przypadku, zanim wykonany zostanie skrypt, uruchomione zostaną pliki */etc/bash.bashrc* oraz *~/.bashrc*.

Zadania

1. Uzupełnij poniższy skrypt, aby uruchamiał się bez błędów, a na standardowym wyjściu wypisywany był komunikat "`Witaj w powloce bash`".

```
/bin/bash

echo "Witaj w
powloce bash
```

2. Napisz skrypt, który uruchomi sekwencję poleceń: `pwd`, `ls -l` oraz `date`. Przygotuj dwie wersje skryptu, w których wymienione polecenia są zapisane w oddzielnych liniach oraz w pojedynczej linii.

3. Uruchom skrypt z poprzedniego zadania na wiele różnych sposobów, tj. podając ścieżkę bezwzględną, względną oraz z wykorzystaniem polecenia `bash`.

1.3 Przekazywanie parametrów

Podczas uruchamiania skryptów można przekazywać do niego parametry, do których można się odwoływać za pomocą zmiennych `$1`, `$2` etc.

Na przykład wykonanie polecenia:

```
$ ./skrypt.sh jeden dwa 3
```

spowoduje, że zmienna `$1` wewnątrz skryptu przyjmie wartość `jeden`, `$2` – `dwa`, a `$3` – `3`. Warto zaznaczyć, że w przypadku odwoływania się do parametrów od 10 w górę, należy obowiązkowo stosować nawiasy klamrowe (np. `${10}`). W przypadku ich braku przykładowa sekwencja `$10` zostałaby zinterpretowana jako sklejenie zmiennej `$1` oraz znaku `"0"`.

Dostęp do zmiennych możliwy jest również za pomocą zmiennej `$@` (lub `$*`), w której zawarta jest pełna lista wszystkich parametrów przedzielonych spacją. Liczba parametrów przekazanych do skryptu dostępna jest w zmiennej `$#`.

Warto tu zwrócić uwagę na użycie niewspomnianej dotąd zmiennej `$0`, w której zawarte jest polecenie (bez przekazywanych parametrów) użyte do uruchomienia tego skryptu. Może więc być tu zawarta zarówno ścieżka względna (np. `../student/skrypt.sh` lub `./skrypt.sh`), jak i bezwzględna (np. `/home/student/skrypt.sh`). Podsumowanie omówionych w tym rozdziale zmiennych zostało przedstawione w tabeli 1.1.

Tabela 1.1: Wybrane zmienne dostępne z poziomu skryptu, zawierające informacje o przekazanych parametrach

Zmienna	Opis
`$0`	polecenie uruchamiające
`$n` lub `${n}`	n-ty przekazany parametr
`$#`	liczba przekazanych parametrów
`$@` lub `$!`	lista przekazanych parametrów

Listing 1.4: Wypisanie wybranych parametrów przekazanych do skryptu

```
#!/bin/bash

echo "Polecenie: " $0
echo "Nazwa skryptu: " `basename $0`

echo "Liczba parametrow: " $#
echo "Lista parametrow: " $@
echo "Parametr 1: " $1
echo "Parametr 2: " $2
echo "Parametr 10: " ${10}
```

Przykładowy skrypt wykorzystujący przekazane parametry ilustruje listing 1.4. W wyniku jego uruchomienia można uzyskać następujący wynik:

```
$ ./skr_params.sh jeden dwa trzy
Polecenie: ./skr_params.sh
Nazwa skryptu: skr_params.sh
Liczba parametrow: 3
Lista parametrow: jeden dwa trzy
Parametr 1: jeden
Parametr 2: dwa
Parametr 10:
```

Warto zauważyć, że odwoływanie się do zmiennej, która nie istnieje (np. `${10}` w powyższym przykładzie), nie powoduje błędu wykonania skryptu, gdyż jest ona traktowana jako zmienna bez przypisanej wartości. Ponadto w celu uzyskania samej nazwy skryptu (bez ścieżki) ze zmiennej `$0` wykorzystane zostało polecenie `basename`, które w zaprezentowanym przykładzie otoczone zostało symbolem "`" (odwrócony apostrof, ang. *backtick*), wymuszającym wcześniejsze wykonanie zawartego w nim polecenia i przekazanie jego wyniku w tym przypadku do polecenia `echo` (patrz podrozdział 2.3).

Celem uzupełnienia warto wspomnieć o jeszcze jednym sposobie odwoływania się do przekazywanych parametrów, z wykorzystaniem polecenia `shift`,

przesuwającym listę parametrów o jeden krok w lewo, tzn. zmiennej `$1` przypisywana jest wartość `$2`, a tej z kolei wartość `$3` etc. Można użyć innego przeskoku, podając jego wartość jako parametr polecenia, np. `shift 3`. Ten rodzaj analizy parametrów stosuje się przede wszystkim w pętlach, dzięki czemu do kolejnych parametrów można się odwoływać za pomocą zawsze ten samej zmiennej, np. `$1`.

W ten sposób, w przypadku przekazania następujących wartości:

```
$1="jeden", $2="dwa", $3="trzy", $4="cztery"
```

po wykonaniu polecenia `shift` otrzymamy:

```
$1="dwa", $2="trzy", $3="cztery"
```

a zmiennej `$4` zostanie przypisana wartość pusta.

Listing 1.5: Przykładowy skrypt prezentujący wykorzystanie polecenia `shift`

```
#!/bin/bash

echo "Parametr 1: $1"

shift
echo "Parametr 1 (po shift): $1"
echo "Parametr 2 (po shift): $2"

shift 2
echo "Parametr 1 (po shift 2): $1"
echo "Parametr 2 (po shift 2): $2"
```

Polecenie `shift` może być użyte w skrypcie wielokrotnie, w tym również z dodatkowym parametrem określającym o ile pozycji to przesunięcie powinno nastąpić (domyślnie jest to wartość 1). Przykładowy skrypt obrazujący wykorzystanie polecenia `shift` został przedstawiony na listingu 1.5, a w wyniku jego uruchomienia uzyskujemy następujący rezultat:

```
$ ./skr_shift.sh jeden dwa trzy cztery piec
Parametr 1: jeden
Parametr 1 (po shift): dwa
Parametr 2 (po shift): trzy
Parametr 1 (po shift 2): cztery
Parametr 2 (po shift 2): piec
```

Zadania

1. Napisz skrypt, który wypisze: polecenie użyte do jego uruchomienia, liczbę przekazanych parametrów oraz ich listę.

2. Napisz skrypt, który bez użycia polecenia `shift` wypisze na standardowe wyjście wszystkie przekazane do niego parametry parzyste (max. do 12 włącznie). W przypadku przekazania mniejszej liczby parametrów mimo wszystko ich wartości powinny zostać wypisane.

   Przykład:
   ```
   $ ./skr_params.sh jeden dwa 3 4 piec szesc 7 8 9 10
   Parametr 2: dwa
   Parametr 4: 4
   Parametr 6: szesc
   Parametr 8: 8
   Parametr 10: 10
   Parametr 12:
   ```

3. Zmodyfikuj skrypt z poprzedniego zadania tak, aby użyte zostało polecenie `shift`, a tym samym za każdym razem wypisywana była zawartość tej samej zmiennej (np. `$2`).

1.4 Komentarze

Komentarze w skryptach rozpoczynają się od symbolu "#". Można je stosować w każdym miejscu, przyjmując zasadę, że wszystko to, co znajduje się w danej linii za nim, staje się komentarzem. Podobnie jak w większości języków skryptowych, nie ma możliwości tworzenia komentarzy blokowych. Przykłady użycia komentarzy zostały przedstawione na listingu 1.6.

Listing 1.6: Przykłady użycia komentarzy w skrypcie

```
#!/bin/bash

#komentarz jednoliniowy
polecenie_1        # komentarz do polecenia
#polecenie_2       --> polecenie zakomentowane
...
#komentarz wieloliniowy
#komentarz wieloliniowy
#komentarz wieloliniowy
polecenie_n
```

Zadania

1. W poniższym skrypcie wstaw symbole komentarza (bez jakichkolwiek innych zmian), tak aby w wyniku jego uruchomienia pojawił się prawidłowy fragment znanego wiersza.

```
#!/bin/bash

echo "Stoi na stacji lokomotywa,"; echo "i udaje, ze jest M"
echo "Ciezka, ogromna i pot z niej splywa."
echo "Tlusta oliwa.";
echo "O ziemie sie huklo"
echo "Biega, krzyczy pan Hilary"
echo "Stoi i sapie, dyszy i dmucha,"
echo "Zar z rozgrzanego jej brzucha bucha."
```

2. Napisz skrypt, który wypisze na standardowe wyjście komunikat "Moj pierwszy skrypt". Na początku pliku umieść w komentarzu, w wielu liniach, informacje o autorze skryptu, tj. imię i nazwisko, kierunek studiów, nazwę uczelni.

1.5 Funkcja `read`

Skrypty są zwykle implementowane w formie nieinteraktywnej, gdyż często są uruchamiane w tle, również z wykorzystaniem narzędzi automatyzujących cykliczne ich uruchamianie (np. poprzez usługę `cron`). Jednakże dzięki zastosowaniu funkcji `read` istnieje możliwość m.in. pobierania danych wprowadzanych przez użytkownika z poziomu konsoli.

Podstawowa składnia polecenia `read` przedstawia się następująco:

```
read zmienna_1 zmienna_2 ... zmienna_n
```

Jak można zauważyć, w jednym poleceniu istnieje możliwość wprowadzenia więcej niż jednej wartości, oddzielając je spacjami. Przykładowy skrypt, w którym wartości zmiennych są pobierane ze standardowego wejścia, został zaprezentowany na listingu 1.7. Użycie klucza `-n` w poleceniu `echo` powoduje, że kursor nie przechodzi do nowej linii, dzięki czemu dane mogą być wprowadzane w tej samej linii co wprowadzający tekst.

Listing 1.7: Skrypt pobierający ze standardowego wejścia podstawowe dane o użytkowniku, takie jak: nazwisko, imię, nr telefonu oraz opis

```
#!/bin/bash

echo -n "Nazwisko i imie: "; read nazwisko imie
echo -n "Nr telefonu: "; read nr_telefonu
echo "Opis:"
read opis
```

W przypadku wprowadzenia zbyt małej liczby danych ostatnim zmiennym przypisywane są wartości puste. Natomiast nadmiarowe dane są dodawane do ostatniej zmiennej. Stąd też w żadnym z powyższych przypadków błędy nie będą generowane. Ewentualną weryfikację liczby wprowadzonych danych, jak i ich poprawności, można wykonać po zakończeniu funkcji `read`, analizując zawartość zmiennych, do których wartości zostały zapisane.

W ten sposób w przypadku poniższego przykładu, jeżeli użytkownik wprowadzi tylko jeden parametr (tj. `nazwisko`), zmiennej `imie` przypisana zostanie wartość pusta, np.:

```
$ echo -n "Nazwisko i imie: "; read nazwisko imie;
  echo "Imie: $imie"; echo "Nazwisko: $nazwisko"
Nazwisko i imie: Kowalski
Imie:
Nazwisko: Kowalski
```

Warto również zwrócić uwagę, że w przypadku wprowadzenia podwójnego imienia (czyli trzech parametrów) uzyskamy następujący wynik:

```
Nazwisko i imie: Kowalski Jan Tadeusz
Imie: Jan Tadeusz
Nazwisko: Kowalski
```

Jak się można domyślić, w przypadku nazwiska składającego się z dwóch części oddzielonych spacjami uzyskany wynik nie będzie taki, jak oczekiwano. W takim przypadku, aby nie było żadnych wątpliwości, użytkownik powinien wszystkie spacje w danej wartości poprzedzić symbolem "\".

```
Nazwisko i imie: vel\ Kowalski Jan
Imie: Jan
Nazwisko: vel Kowalski
```

Innym sposobem rozwiązania tego problemu jest zmiana domyślnego separatora zapisanego w zmiennej `IFS`. W poniższym przypadku poszczególne

parametry podczas wprowadzania powinny być rozdzielone przecinkiem:

```
$ echo -n "Nazwisko i imie: "; IFS=',' read nazwisko imie;
  echo "Imie: $imie"; echo "Nazwisko: $nazwisko"
Nazwisko i imie: vel Kowalski,Jan
Imie: Jan
Nazwisko: vel Kowalski
```

Funkcja `read` oferuje również wiele ciekawych opcji, z którymi można się zapoznać, wykonując polecenie:

```
$ read --help
```

I tak np. opcja `-s` powoduje, że dane wprowadzane przez użytkownika nie są wypisywane na `stdout`. Można to wykorzystać m.in. podczas wprowadzania hasła przez użytkownika:

```
$ echo -n "Podaj haslo: "; read -s haslo; echo;
  echo "Podane haslo: $haslo"
Podaj haslo: <wpisywana wartosc nie jest widoczna>
Podane haslo: topsecret
```

Powyższy fragment można uzupełnić o dodanie opcji `-t` z parametrem określającym maksymalny czas (w sekundach) oczekiwania na wprowadzenie danych, a po jego upływie zmiennej automatycznie zostanie przypisana wartość pusta:

```
$ echo -n "Podaj haslo: "; read -t 3 -s haslo; echo;
  echo "Podane haslo: $haslo"
Podaj haslo: <odczekanie wskazanego czasu bez wprowadzania danych>
Podane haslo:
```

Funkcja `read` może być również użyta do uzupełniania wartości tablicowych, co zostało zaprezentowane w rozdziale 6. Wprowadzone dane są wówczas automatycznie dzielone na wyrazy, które następnie są zapisywane pod kolejnymi indeksami.

Zadania

1. Napisz skrypt, który z odczytanego ze standardowego wejścia kodu pocztowego w formacie `"dd-ddd"` wypisze go bez myślnika. Wykorzystaj w tym celu zmienną `IFS`.

   Przykład:
   ```
   $ ./skr_read_kod.sh
   Podaj kod pocztowy: 15-335
   15335
   ```

2. Napisz skrypt wypisujący zawartość pliku (np. wykorzystując polecenie `cat`), którego ścieżka zostanie odczytana ze standardowego wejścia.

   Przykład:
   ```
   $ ./skr_read_file.sh
   Podaj sciezke po pliku: /etc/resolv.conf
   nameserver 8.8.8.8
   ```

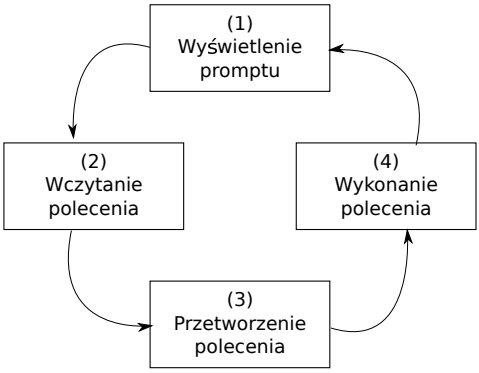
3. Napisz skrypt umożliwiający użytkownikowi wprowadzenie parametrów do jego uwierzytelnienia, tj. login oraz hasło, przy czym w przypadku drugiego z nich wprowadzane znaki nie powinny być widoczne. Na zakończenie wprowadzone dane powinny zostać wypisane wraz z opisem.

   Przykład:
   ```
   $ ./skr_read_login.sh
   Podaj login: student
   Podaj haslo: <niewidoczne>
   login/haslo: student/topsecret
   ```

Rozdział 2

Substytucje

Powłoka jest interfejsem między użytkownikiem a systemem operacyjnym, w tym przypadku systemem `Linux`. Jej podstawowymi funkcjonalnościami są wczytanie, analiza i wykonanie poleceń użytkownika. Działa ona zgodnie ze schematem przedstawionym na rysunku 2.1.



Rysunek 2.1: Schemat działania powłoki `bash`

Krok pierwszy (1) odpowiedzialny jest za poprawne wyświetlenie znaku zachęty (ang. *prompt*). W kroku drugim (2) następuje wczytanie polecenia. Krok trzeci (3) odpowiedzialny jest za analizę, interpretację i ewentualną modyfikację (np. postaci parametrów) wczytanego polecenia. W kroku czwartym (4) następuje jego wykonanie. Polecenie może zostać wykonane wewnętrznie przez powłokę (polecenie wbudowane/wewnętrzne) lub w osobnym procesie utworzonym na bazie wskazanego programu zewnętrznego (polecenie zewnętrzne). W rozdziale tym uwaga skupiona zostanie na punkcie (3) powyższego schematu.

Przed wykonaniem podanego przez użytkownika polecenia powłoka dokonuje jego analizy pod kątem występowania metaznaków. W przypadku ich wystąpienia rozwija je zgodnie z predefiniowanym schematem. Proces rozwinięcia metaznaków określany jest mianem substytucji. W powłoce `bash` realizowane są trzy główne kategorie substytucji:

- substytucja nazw plików,
- substytucja zmiennych środowiskowych,
- substytucja poleceń.

Szczególną uwagę należy zwrócić na fakt, iż wszystkie substytucje realizowane są przez powłokę przed wykonaniem polecenia/skryptu.

2.1 Substytucja nazw plików

Jako parametry do skryptów i programów często przekazywane są nazwy plików. Pod pojęciem tym rozumiemy wszelkie obiekty (pliki, katalogi, urządzenia itp.), które w systemie plików występują pod określoną nazwą. Wielokrotnie zachodzi potrzeba przekazania listy nazw plików jako parametrów. Przykładem może być skasowanie grupy plików. Do polecenia `rm` podajemy wówczas nazwy tych plików, które chcemy wykasować. W przypadku jednego czy dwóch plików nie stwarza to problemu i nazwy kasowanych plików z reguły z łatwością możemy podać jako parametry polecenia `rm`.

W przypadku większej liczby kasowanych plików proces ten może być uciążliwy. Substytucja poleceń w sposób znaczny ułatwia generowanie listy nazw plików i przekazywanie jej jako parametrów poleceń uruchamianych w powłoce. Aby to osiągnąć, należy w miejsce parametrów wykorzystać wzorce nazw plików, które konstruowane są z wykorzystaniem metaznaków – symboli uogólniających będących podzbiorem symboli specjalnych, które powłoka interpretuje w inny sposób niż ich literalne znaczenie. Metaznaki wykorzystywane w procesie substytucji nazw plików (ang. *wildcards*) przedstawia tabela 2.1.

Tabela 2.1: Metaznaki w substytucji nazw plików

Metaznaki	Opis
`*`	ciąg złożony z 0 lub więcej dowolnych znaków
`?`	jeden dowolny znak
`[...]`	jeden ze znaków z podanego zbioru
`[!...]`	dowolny znak spoza podanego zbioru
`[...-...]`	jeden ze znaków z podanego zakresu

Przed wykonaniem polecenia każdy jego element (nazwa polecenia, parametry) zawierający jeden lub więcej z powyższych metaznaków jest interpretowany jako wzorzec nazw plików. Jeżeli w katalogu występują obiekty z pasującymi do niego nazwami, to ich lista, uszeregowana w sposób alfabetyczny, zastępuje wzorzec. Jeżeli w katalogu brak jest obiektów z nazwami pasującymi do wzorca, nie ulega on żadnej zmianie. Zatem substytucja nazw plików polega na dopasowywaniu użytych wzorców do nazw obiektów znajdujących się w katalogu. Jeżeli wzorzec nie będzie poprzedzony ścieżką dostępu, to będzie dopasowywany do nazw plików występujących w katalogu bieżącym.

Przykład: Interpretacje wybranych wzorców nazw plików.

```
$ ls -a
. .. .a a a.txt ab abc b b.txt c dd
```

Pliki z nazwą rozpoczynającą się od litery "a":

```
$ ls a*
a a.txt ab abc
```

Pliki z rozszerzeniem "txt":

```
$ ls *.txt
a.txt b.txt
```

Pliki z nazwą składającą się z jednego znaku:

```
$ ls ?
a b c
```

Pliki z nazwą rozpoczynającą się na literę "a", "b" lub "c":

```
$ ls [a-c]*
a a.txt ab abc b b.txt c
```

Przykładowy skrypt wykorzystujący mechanizm substytucji nazw plików przedstawia listing 2.1. Skrypt z wykorzystaniem pętli `for` sukcesywnie analizuje wszystkie obiekty znajdujące się w katalogu bieżącym. Analiza polega na określeniu typu kolejnych obiektów z wykorzystaniem polecenia `if` i skróconej formy polecenia `test`. Nazwy wszystkich katalogów z wykorzystaniem polecenia `echo` wypisywane są na standardowe wyjście. W skrypcie wykorzystane są instrukcje i struktury opisane w rozdziałach 4 oraz 5.

Listing 2.1: Przykładowy skrypt prezentujący wykorzystanie mechanizmu substytucji nazw plików

```
#!/bin/bash
echo Lista katalogow
echo
for element in *
do
        if [ -d $element ]
        then
                echo $element
        fi
done
```

Wszystkie omawiane substytucje rozwijane są przez powłokę przed wykonaniem polecenia. Fakt ten powinien w sposób szczególny zostać uwzględniony przy uruchamianiu poleceń, które w parametrach przyjmują i w sposób specjalny interpretują wybrane metaznaki. Przykładem takiego polecenia może być polecenie `find`. W takich przypadkach należy „zneutralizować" znaczenie metaznaku symbolem "\" lub odpowiednimi ogranicznikami (patrz podrozdział 2.4). W przypadku, gdy użyte metaznaki nie zostaną odpowiednio zamaskowane, może to prowadzić do trudno wykrywalnych błędów. Ilustruje to poniższy przykład, w którym polecenie `find` wyszukuje w katalogu bieżącym i jego podkatalogach wszystkie pliki rozpoczynające się od litery "a" .

Przykład: Konieczność maskowania metaznaków na przykładzie polecenia `find`.

```
$ tree
.
|-- K1
|   |-- a.txt
|   |-- aa.txt
|-- K2
|   |-- a.txt
|   |-- aa.txt
|-- a.txt
|-- aa.txt

2 directories, 6 files
$ find . -name a\* -print
./a.txt
./K2/a.txt
./K2/aa.txt
./aa.txt
./K1/a.txt
./K1/aa.txt
```

```
$ find . -name a* -print
find: paths must precede expression: 'aa.txt'
find: possible unquoted pattern after predicate '-name'?
```

Otrzymane wyniki potwierdzają, iż w przypadku wykorzystania maskowania znaku specjalnego (w tym przypadku *) polecenie `find` działa poprawnie. Natomiast gdy nie został użyty znak maskujący, zgłoszony został błąd, którego dokładną analizę można przeprowadzić, włączając (za pomocą polecenia `set -x`) tryb debugowania (patrz rozdział 9).

```
$ set -x
$ find . -name a* -print
+ find . -name a.txt aa.txt -print
find: paths must precede expression: 'aa.txt'
find: possible unquoted pattern after predicate '-name'?
```

Otrzymane wyniki wskazują, iż przed wykonaniem polecenia `find` nastąpiło rozwinięcie wzorca "a*", w wyniku czego został on zastąpiony dwiema nazwami plików, które były z nim zgodne, a mianowicie: a.txt i aa.txt. Finalnie wykonane zostało polecenie:

```
find . -name a.txt aa.txt -print
```

które nie było poprawne z punktu widzenia składni (po operatorze `-name` może występować tylko jeden parametr definiujący szukane pliki).

Zadania

1. Co wypisze polecenie "`echo *`"?
2. Z katalogu bieżącego do katalogu `/tmp` skopiuj wszystkie pliki z rozszerzeniem "`.txt`".

3. W katalogu bieżącym utwórz plik o nazwie `$1`.

4. W katalogu bieżącym utwórz pliki `a1.txt`, `a2.txt` i `a3.txt`. Nie zmieniając katalogu, wyszukaj w całym systemie plików wszystkie pliki z rozszerzeniem `".txt"`.

5. Załóżmy, że katalog bieżący zawiera następujące pliki: `prog.c`, `prog1.c`, `prog2.c`, `progabc.c`, `prog`, `p.txt`, `p1.txt`, `p21.txt`, `p22.txt` i `p22.dat`. Do których z nich będzie „pasował” wzorzec `"p12*"`?

2.2 Substytucja zmiennych środowiskowych

Wiele narzędzi w systemie Unix/Linux, włączając w to powłokę, wykorzystuje do poprawnego działania informacje związane z otoczeniem, w którym działają. Przykładowo, edytory potrzebują informacji o typie terminala, z jakim współpracują, a powłoka musi zostać „poinformowana”, gdzie (w jakich katalogach) umieszczone są polecenia systemu (np. `ls`, `cp` itd.). Ponadto bardzo często zdarza się, iż wiele procesów wykorzystuje tę samą informację w swym działaniu (np. ścieżka do katalogu przeznaczonego na pliki tymczasowe czy aktualne kodowanie związane z ustawieniami lokalizacyjnymi). Wszystkie te informacje można przekazywać do uruchamianego programu poprzez parametry wywołania. Aby jednak uniknąć przekazywania nadmiernej liczby parametrów, w wielu systemach operacyjnych (w tym w Linuksie) wykorzystywany jest mechanizm zmiennych środowiskowych (ang. *environment variables*).

W każdym procesie zarezerwowany jest obszar pamięci nazywany „środowiskiem” (ang. *environment*), w którym przechowywane są ciągi znakowe w postaci "klucz=wartość". Klucz jest w tym przypadku nazwą zmiennej środowiskowej. Środowisko (jego zawartość) w nowym procesie dziedziczone jest z procesu macierzystego. Procesy podczas swojego działania mogą zarówno odwoływać się do odziedziczonych zmiennych środowiskowych, jak i definiować nowe zmienne. Wiele procesów wykorzystuje predefiniowane zmien-

ne środowiskowe (o z góry ustalonych w systemie nazwach). Przykładowo, zmienna `TERM` domyślnie precyzuje typ terminala/emulatora, jaki jest używany jako urządzenie wyjściowe, a zmienna `TEMP` wskazuje katalog, w którym procesy mogą umieszczać swoje pliki tymczasowe. W przypadku systemu Linux „rodzicem" większości procesów działających w systemie jest powłoka. Udostępnia ona przyjazny interfejs umożliwiający obsługę zmiennych środowiskowych (m.in. odczyt aktualnych oraz dodawanie nowych).

Należy zauważyć, iż powłoka `bash` pozwala również na obsługę zmiennych lokalnych, które nie są przenoszone do nowo tworzonych procesów i są widoczne wyłącznie w powłoce bieżącej. Niemniej jednak interfejs obsługi zmiennych lokalnych, jak i standardowych zmiennych środowiskowych w powłoce `bash` jest identyczny. Definiujemy je w niej, podając nazwę zmiennej, znak równości i bezpośrednio za znakiem równości wartość, jaką chcemy przypisać do zmiennej. Odwołujemy się do zmiennej, poprzedzając jej nazwę znakiem "$". Ilustrują to poniższe polecenia:

```
$ database=/usr/data/factory.dat
$ echo $database
/usr/data/factory.dat
```

Należy podkreślić, iż podobnie jak to miało miejsce w przypadku substytucji nazw plików, za rozwinięcie zmiennej środowiskowej odpowiada powłoka, a nie polecenie. Rozwinięcie znaczenia zmiennej środowiskowej poprzedzonej znakiem "$", podanej jako parametr uruchamianego polecenia nazwy, dokonuje się przed wykonaniem polecenia. W konsekwencji do polecenia jako parametr zostaje przekazana wartość zmiennej.

Przykład: Rozwinięcie zmiennej powłoki przed wykonaniem polecenia.

```
$ set -x
$ database=/data/factory.dat
+ database=/data/factory.dat $ ls -l $database
+ ls -l /data/factory.dat
-rw-r-r- 1 root root 46000 Nov 10 23:10 /data/factory.dat
```

W powyższym przykładzie polecenie `ls` wypisuje informacje o pliku, do którego ścieżka została przypisana zmiennej `database`. Po wczytaniu przez powłokę polecenia:

```
$ ls -l $database
```

jednak jeszcze przed jego uruchomieniem, nastąpiła substytucja, w wyniku której w miejsce nazwy zmiennej wstawiona została jej wartość. W efekcie wykonane zostało polecenie `ls -l /data/factory.dat`.

Blokowanie substytucji, podobnie jak to miało miejsce w przypadku substytucji nazw plików, odbywa się z wykorzystaniem znaku "\" zapisanego przed znakiem "$":

```
$ database=/usr/data/factory.dat
$ ls -l \$database
ls: cannot access '$database': No such file or directory
```

Substytucja zmiennych środowiskowych może również zostać zablokowana z wykorzystaniem mechanizmów cytowania (patrz podrozdział 2.4). Szczegółowe informacje dotyczące zmiennych środowiskowych opisane są w podrozdziale 3.1.

Zadania

1. Wyświetl zawartość zmiennej środowiskowej `PATH`.

2. Co wypisze polecenie `echo` z poniższego ciągu poleceń?

   ```
   $ A=4
   $ B=$A+5
   $ echo $B
   ```

2.3 Substytucja poleceń

Substytucja poleceń to mechanizm powłoki umożliwiający zastąpienie polecenia przez dane wyjściowe wygenerowane przez to polecenie [4]. Substytucja poleceń realizowana jest z wykorzystaniem wyrażeń w postaci:

```
$(polecenie)
```

Powłoka `bash`, interpretując tak podane wyrażenie, wykona polecenie (lub ciąg poleceń) i zastąpi je rezultatem (tekstem) generowanym przez to polecenie na standardowe wyjście. Wyrażenie zostanie zastąpione wygenerowanym rezultatem w całości, tj. począwszy od znaku dolara "$" aż do nawiasu zamykającego ")".

Przykład: Wykorzystanie mechanizmu substytucji poleceń do czytelnego wypisania rozmiaru danych zawartych w katalogu bieżącym i jego podkatalogach.

```
$ echo Rozmiar danych w katalogu biezacym: $(du -sh)
Rozmiar danych w katalogu biezacym: 697M
```

Równoważną konstrukcją do powyższej jest konstrukcja:

```
‘polecenie‘
```

Wykorzystywane są w niej „odwrócone” apostrofy, które najczęściej na klawiaturze można znaleźć w okolicach klawisza `Esc`. Pierwsza konstrukcja wywodzi się z powłoki `Korn shell` i jest preferowana przez powłokę `bash` [4], podczas gdy druga pochodzi z powłok `Bourne shell` i `C-shell` i jest zaimplementowana w powłoce `bash` w celu zachowania kompatybilności wstecznej. Przytoczony wcześniej przykład z wykorzystaniem składni opartej na odwróconych apostrofach został przedstawiony poniżej.

Przykład: Substytucja poleceń z wykorzystaniem składni opartej na „odwróconych” apostrofach.

```
$ echo Rozmiar danych w katalogu biezacym: ‘du -sh‘
Rozmiar danych w katalogu biezacym: 697M
```

Należy zwrócić uwagę, iż polecenia wykonywane w ramach mechanizmu substytucji poleceń uruchamiane są w podpowłoce (ang. *subshell*). Posiadają własne środowisko i jako proces potomny nie mają możliwości oddziaływania na środowisko powłoki macierzystej. Obrazuje to poniższy przykład.

Przykład: Podpowłoka jako środowisko wykonania poleceń w ramach substytucji poleceń.

```
$ zmienna1=abc
$ echo $zmienna1
abc
$ echo $(zmienna1=123; echo $zmienna1)
123
$ echo $zmienna1
abc
```

W powyższym przykładzie do zmiennej `zmienna1` przypisana została wartość "abc". Następnie wykorzystano mechanizm poleceń, w parametrze kolejnego polecenia `echo`, w którym zmodyfikowana została wartość zmiennej `zmienna1`, o czym świadczy wyświetlony wynik "123". Jednak rezultatem ostatniego polecenia `echo $zmienna1`, wypisującego wartość zmiennej `zmienna1`, jest ponownie ciąg znaków "abc". Świadczy to o tym, iż przypisanie wartości "123" do zmiennej `zmienna1` musiało się odbyć w innym procesie niż bieżący proces powłoki. W przeciwnym wypadku zmieniona wartość zmiennej "123" byłaby również rezultatem ostatniego polecenia `echo`.

2.4 Mechanizmy cytowania

Powłoki systemowe, w tym `bash`, w systemie Linux interpretują wybrane znaki w sposób specjalny. Oznacza to, że ich znaczenie jest inne niż literalne. Wśród nich można wymienić m.in.: `* ? [ ] ( ) = | ^ ; < > ‘ $ " ’ \`. Powłoka, napotykając na nie (np. w parametrach uruchamianego polecenia), dokonuje ich rozwinięcia. Sposoby interpretacji i przykłady rozwinięcia przez powłokę wielu znaków specjalnych zostały przybliżone w podrozdziałach 2.1, 2.2 i 2.3 podczas opisu substytucji nazw plików, zmiennych środowiskowych i poleceń. Ponadto również niektóre polecenia uruchamiane z poziomu powłoki traktują wybrane podzbiory znaków (w tym podzbiory znaków specjalnych) w odmienny niż literalny sposób (np. polecenie `find` lub `grep`). W takich przypadkach, aby zapewnić odpowiednią ich interpretację, niezbędny jest mechanizm blokujący proces rozwijania przez powłokę znaków specjalnych tak, aby możliwe było przekazywanie ich do poleceń w formie literalnej. Służą do tego mechanizmy cytowania. Dzięki nim można np. jako element parametru polecenia `find` bezpiecznie przekazać "*", która nie zostanie rozwinięta przez powłokę, ale w postaci literalnej będzie przekazana do polecenia.

W powłoce `bash` są trzy mechanizmy cytowania. Wykorzystują one następujące znaki cytowania: cudzysłów (ang. *double quote*) ("..."), apostrof (ang. *single quote*) ('...') i *backslash* (\). Ten ostatni omówiono przy substytucjach. Znak "\" występujący przed znakiem specjalnym maskuje przed powłoką jego „specjalne" znaczenie. Podczas interpretacji takiego ciągu znaków powłoka usuwa "\", a znak specjalny traktuje w sposób literalny.

Przykład: Maskowanie znaczenia znaku specjalnego z wykorzystanie znaku "\".

```
$ echo \*
*
```

W powyższym przykładzie znak "\" zamaskował specjalne znaczenie "*". Otrzymany wynik działania polecenia `echo` pokazuje, iż powłoka z ciągu znaków "*" będącego parametrem polecenia usunęła "\" i finalnie do polecenia `echo` przekazała "*" w postaci literalnej (nierozwiniętej).

Dodatkowo znak "\" umożliwia pisanie czytelnych poleceń wielowierszowych. Występując na końcu linii, sygnalizuje, iż jest kontynuowana w kolejnym wierszu. Kolejne przykłady pozwolą przeanalizować dwa pozostałe mechanizmy cytowania.

Przykład: Cytowanie oparte na cudzysłowach ("...") w odniesieniu do substytucji nazw plików.

```
$ echo *
a a.txt b b.txt
$ echo ?
a b
$ echo "*"
*
$ echo "?"
?
```

Na podstawie wyników działania powyższych poleceń można stwierdzić, iż cytowanie wykorzystujące cudzysłów blokuje rozwijanie metaznaków używanych w substytucji nazw plików. Zarówno gwiazdka, jak i znak zapytania, które są poprawnie rozwijane poza cudzysłowem, wewnątrz niego traktowane są przez powłokę w sposób literalny.

Przykład: Cytowanie oparte na cudzysłowach ("...") w odniesieniu do substytucji zmiennych środowiskowych.

```
$ a=plik
$ ls -l $plik
-rw-rw-r- 1 dmiasta dmiasta 0 Nov 12 09:58 a.txt
```

```
$ ls -l "$plik"
-rw-rw-r- 1 dmiasta dmiasta 0 Nov 12 09:58 a.txt
```

W kolejnym przykładzie można zaobserwować, iż cytowanie z użyciem cudzysłowu nie wpływa na interpreację przez `bash` elementów związanych z substytucją zmiennych środowiskowych. Zmienna środowiskowa `plik` została prawidłowo rozwinięta, zarówno bez cytowania, jak i będąc cytowana z wykorzystaniem cudzysłowu.

Przykład: Cytowanie oparte na cudzysłowach ("...") w odniesieniu do substytucji poleceń.

```
$ echo ‘ls‘
a a.txt b b.txt
$ echo "‘ls‘"
a a.txt b b.txt
```

Analizując przykład, można zauważyć, iż w obu przypadkach (z cytowaniem i bez cytowania) polecenie `ls` ujęte w odwrotne apostrofy zostało wykonane i jego wynik został poprawnie przekazany do polecenia `echo`. Zatem podobnie jak w przypadku substytucji zmiennych środowiskowych, również cudzysłów nie wpływa na substytucję poleceń.

Podobne przykłady jak powyżej zostaną teraz rozpatrzone dla cytowania opartego na apostrofach.

Przykład: Cytowanie oparte na apostrofach ('...') w odniesieniu do substytucji nazw plików.

```
$ echo *
a a.txt b b.txt
$ echo ?
a b
$ echo '*'
*
```

```
$ echo '?'
?
```

Zarówno gwiazdka (*), jak i znak zapytania (?) nie zostały rozwinięte i zinterpretowano je w sposób literalny. Świadczy to o własności blokującej mechanizmu cytowania opartego na cudzysłowach w odniesieniu do substytucji nazw plików. Podobną własność mechanizm ten ma w odniesieniu do substytucji zmiennych środowiskowych, jak również substytucji poleceń. Ani w pierwszym, ani w drugim przypadku substytucje nie są rozwijane. Wszystkie symbole traktowane są w sposób literalny, co demonstrują zamieszczone poniżej przykłady.

Przykład: Cytowanie oparte na apostrofach ('...') w odniesieniu do substytucji zmiennych środowiskowych.

```
$ a=plik
$ ls -l $plik
-rw-rw-r- 1 dmiasta dmiasta 0 Nov 12 09:58 a.txt
$ ls -l '$plik'
ls: cannot access '$plik': No such file or directory
```

Przykład: Cytowanie oparte na apostrofach ('...') w odniesieniu do substytucji poleceń.

```
$ echo `ls`
a a.txt b b.txt
$ echo ''ls''
`ls`
```

W wynikach działania poszczególnych poleceń `echo` w powyższych przykładach można zauważyć, że wszystkie parametry cytowane z wykorzystaniem apostrofów są interpretowane w sposób literalny.

Podsumowanie mechanizmów cytowania w odniesieniu do podstawowych typów substytucji zawarte jest w tabeli 2.2.

Tabela 2.2: Aktywacja substytucji w zależności od mechanizmu cytowania

Substytucja \ Cytowanie	" ... "	' ... '
Nazw plików	–	–
Zmiennych środowiskowych	+	–
Poleceń	+	–

W tabeli tej plus (+) oznacza, iż przy danym cytowaniu powłoka dokonuje określonej substytucji, a minus (–), iż znaki i wyrażenia specjalne związane z określonym typem substytucji powłoka interpretuje w sposób literalny.

Reasumując, należy stwierdzić, iż tekst zapisany w apostrofach nie podlega żadnej substytucji. Żadne znaki specjalne nie są w tym wypadku rozwijane przez powłokę. Znaczenie specjalne m.in. takich znaków jak "*, $, &" jest ignorowane.

Mechanizm cytowania oparty na cudzysłowie działa w sposób podobny z tą różnicą, iż pewne znaki mimo cytowania są interpretowane przez powłokę w sposób specjalny. Można zauważyć, iż w tekście cytowanym z wykorzystaniem cudzysłowów dokonuje się m.in. substytucja zmiennych środowiskowych i substytucja poleceń. Blokowana jest substytucja nazw plików. Ponadto w sposób specjalny traktowany jest *backslash*, co umożliwia w tekście literalne przytoczenie znaku "$", cudzysłowu czy odwróconych apostrofów [7].

Rozdział 3

Zmienne

W powłoce `bash` typowanie zmiennych nie jest obsługiwane, a wszystkie wartości są domyślnie traktowane jako ciągi znaków. Wykonywanie operacji arytmetycznych jest możliwe, choć w ograniczonym zakresie i wymaga zastosowania odpowiedniej składni.

Rozróżniane są dwa podstawowe typy zmiennych: wartości numeryczne oraz alfanumeryczne, na których można wykonywać zaawansowane operacje, znane z innych popularnych języków programowania. Warto jednak zauważyć, że w przypadku wartości numerycznych jesteśmy ograniczeni tu jedynie do liczb całkowitych. Definiowane zmienne mogą mieć zasięg zarówno globalny (dostępne dla wszystkich użytkowników systemu), jak i lokalny, których widoczność jest ograniczona do danego procesu (np. skryptu).

3.1 Zmienne globalne oraz lokalne

Tworząc skrypty, mamy możliwość korzystania zarówno ze zmiennych środowiskowych globalnych, jak i tych lokalnych, zdefiniowanych wewnątrz skryp-

tu. Konwencja nazewnicza, którą zwykle się przyjmuje, zakłada, że globalne zmienne środowiskowe są zwykle zapisywane wielkimi literami (np. `PATH`), a zmienne lokalne małymi (np. `zmienna`). Stosując się do powyższych zasad, ułatwiamy innym użytkownikom interpretację kodu.

Aktualne zmienne środowiskowe mogą być wypisane za pomocą m.in. poleceń `env`, `set` oraz `printenv`. Poniżej przedstawione są fragmenty wyników uzyskanych za pomocą jednego z nich.

```
$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/XPS-8500
COLORTERM=truecolor
XD_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/student
LOGNAME=student
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=2507
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.7CP8U1
HOME=/home/student
USERNAME=student
LANG=pl_PL.UTF-8
...
```

Zmienne globalne, dostępne z poziomu wszystkich procesów wszystkich użytkowników, tworzone są za pomocą pliku */etc/bash.bashrc*. Natomiast te dostępne dla konkretnego użytkownika są definiowane w *~/.bashrc*.

Zmienne lokalne są tworzone na bieżąco, bezpośrednio w powłoce (lub skrypcie) poprzez przypisanie do niej wartości, np.:

```
$ zmienna=wartosc
$ echo $zmienna
wartosc
```

Tak utworzone zmienne są dostępne jedynie w danym procesie. Ich widoczność można rozszerzyć do procesów potomnych poprzez wyeksportowanie za pomocą polecenia `export`.

Przykład: Dostęp do niewyeksportowanej zmiennej lokalnej `system` w procesie potomnym.

```
$ system=Linux
$ echo $system
Linux
$ bash
$ echo $system
    -> brak wartości
```

Jak można zauważyć, zmienna `system`, po przypisaniu do niej wartości, jest dostępna jedynie w bieżącej powłoce. Po przejściu do procesu potomnego (w tym przypadku jest to nowa powłoka `bash`) jest ona niewidoczna. Identyczna sytuacja miałaby miejsce również w przypadku, gdyby w danej powłoce został uruchomiony skrypt, który także nie miałby do niej dostępu.

Przykład: Dostęp do niewyeksportowanej zmiennej lokalnej `system` w procesie potomnym będącym skryptem.

```
$ system=Linux
$ cat skrypt.sh
#!/bin/bash
echo $system
```

```
$ echo $system
Linux
$ ./skrypt.sh
    -> brak wartości
```

Inaczej sytuacja przedstawia się, gdy zmienna zostanie wyeksportowana. Można to zrobić za pomocą jednego z poniższych poleceń, tj. ustawienia wartości zmiennej, a następnie jej wyeksportowania, lub też za pomocą pojedynczego polecenia:

```
$ system=Linux
$ export system
```
lub
```
$ export system=Linux
```

Przykład: Dostęp do wyeksportowanej zmiennej lokalnej `system` w procesie potomnym będącym skryptem.

```
$ export system=Linux
$ cat skrypt.sh
#!/bin/bash
echo $system
$ echo $system
Linux
$ ./skrypt.sh
Linux
```

Listę wyeksportowanych zmiennych w bieżącej powłoce można wyświetlić za pomocą polecenia `export -p`.

```
$ export -p
...
declare -x XDG_SESSION_CLASS="user"
declare -x XDG_SESSION_DESKTOP="ubuntu"
declare -x XDG_SESSION_TYPE="wayland"
declare -x XMODIFIERS="@im=ibus"
declare -x system="Linux"
...
```

Warto tu również wspomnieć o możliwości przesłaniania zmiennych wewnątrz procesów potomnych. Jeżeli przypiszemy nową wartość zmiennej wewnątrz skryptu, będzie ona w nim obowiązywała. Dotyczy to również zmiennych wyeksportowanych.

Przykład: Przesłonięcie zmiennej w procesie potomnym.

```
$ cat skrypt.sh
#!/bin/bash
system=Debian
echo $system
```

```
$ export system=Linux
$ echo $system
Linux
$ ./skrypt.sh
Debian
$ echo $system
Linux
```

Istnieje również możliwość przesłaniania zmiennych poprzez przekazanie ich w postaci parametru uruchamianego skryptu. Poniżej przedstawiona jest składnia przesłaniająca dwie wskazane wartości:

```
$ zmienna1=wartosc1 zmienna2=wartosc2 ./skrypt.sh
```

Przykład: Przesłonięcie zmiennej środowiskowej `PATH` przekazanej w postaci parametru uruchamianego skryptu. Warto zauważyć, że po powrocie do procesu nadrzędnego zmienna przyjmuje wartość pierwotną.

```
 $ cat skrypt.sh
#!/bin/bash
echo PATH=$PATH
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
$ PATH="/bin:/sbin" ./skrypt.sh
PATH=/bin:/sbin
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
```

Nieużywane zmienne mogą być usuwane za pomocą polecenia `unset`, np.:

```
$ unset zmienna.
```

Przykład: Usunięcie zmiennej `system`.

```
$ echo $system
Linux
$ unset system
$ echo $system
    -> brak wartości
```

Zadania

1. Wyświetl zmienne środowiskowe, korzystając z poleceń `printenv`, `env` oraz `set`. Do stronicowania wyników użyj polecenia `more`, np.
   `$ printenv | more`

2. Wyświetl wartości wybranych zmiennych środowiskowych, np. `PATH`, `USER`, `TZ`, `HOME` oraz `SHELL`.

3. Ustaw zmienną `uczelnia=PB` i wyświetl jej zawartość. Uruchom nową powłokę za pomocą polecenia `bash` i sprawdź, czy nadal zmienna `uczelnia` jest ustawiona. Wróć do poprzedniej powłoki za pomocą polecenia `exit`.

4. Wyeksportuj zmienną `Uczelnia` oraz uruchom polecenie `export -p`, aby zweryfikować, czy została ona wyeksportowana. Uruchom nową powłokę za pomocą polecenia `bash` i sprawdź, czy tym razem zmienna `Uczelnia` jest ustawiona. Wróć do poprzedniej powłoki za pomocą polecenia `exit`.

5. Za pomocą pojedynczego polecenia ustaw i wyeksportuj zmienną `grupa=PS7`. Wyświetl zawartość tej zmiennej oraz sprawdź, czy pojawia się w wynikach polecenia `export -p`.

6. Usuń zmienne `uczelnia` oraz `grupa`. Wyświetl ich zawartość, aby się upewnić, że nie są przypisane do nich żadne wartości.

7. Porównaj wyniki wyświetlane za pomocą polecenia `ls -al` oraz wykonanego z przesłoniętą zmienną `TZ` ustawioną na wartość "GMT". Powtórz to zadanie również dla polecenia `date`. Czy wyświetlane wartości różnią się w zależności od użytej strefy czasowej?

8. Wyszukaj za pomocą polecenia `which` ścieżkę dostępu do polecenia `ls`. Powtórz to polecenie z przesłoniętą zmienną `PATH=/etc`. Czy ścieżka do polecenia została wyszukana i tym razem?

3.2 Operacje na ciągach znaków

W tabeli 3.1 zostały przedstawione wybrane operacje na ciągach znaków. Jak można zauważyć jest to dosyć bogaty zestaw uwzględniający m.in. dodawanie ciągów, wydzielanie podciągów oraz zamianę fragmentów ciągów. Poniżej przedstawione są przykłady pokazujące wymienione operacje w działaniu.

Tabela 3.1: Wybrane operacje na ciągach znaków

Składnia	Opis
$str1$str2	dodanie (sklejenie) dwóch zmiennych `str1` oraz `str2`
${#str}	długość zmiennej `str`
${str:start}	podciąg zmiennej `str` rozpoczynający się od indeksu `start` do jej końca
${str:start:dlugosc}	podciąg zmiennej `str` rozpoczynający się od indeksu `start` i długości `dlugosc`
${str/ciag}	usunięcie ze zmiennej `str` pierwszego wystąpienia podciągu o wartości `ciag`
${str//ciag}	usunięcie ze zmiennej `str` wszystkich wystąpień podciągów o wartości `ciag`
${str/stary/nowy}	zastąpienie w zmiennej `str` pierwszego wystąpienia ciągu `stary` wartością `nowy`
${str//stary/nowy}	zastąpienie w zmiennej `str` wszystkich wystąpień ciągu `stary` wartością `nowy`
${str#*separator}	usunięcie początku wartości zmiennej `str` do pierwszego wystąpienia ciągu `separator`
${str##*separator}	usunięcie początku wartości zmiennej `str` do ostatniego wystąpienia ciągu `separator`
${str%separator*}	usunięcie części wartości zmiennej `str` od pierwszego wystąpienia ciągu `separator`
${str%%separator*}	usunięcie części wartości zmiennej `str` od ostatniego wystąpienia ciągu `separator`
${str^}	zamiana pierwszej litery na wielką w zmiennej `str`
${str^^}	zamiana na wielkie litery w zmiennej `str`
${str, }	zamiana pierwszej litery na małą w zmiennej `str`
${str,,}	zamiana na małe litery w zmiennej `str`

Przykład: Scalenie wartości zmiennej `imie` oraz `nazwisko` z użyciem separatora, jak i bez niego.

```
$ imie=Jan; nazwisko=Kowalski
$ str=$imie$nazwisko; echo $str
JanKowalski
$ str=$imie" "$nazwisko; echo $str
Jan Kowalski
```

Przykład: Wypisanie długości zmiennej `str`.

```
$ str="Linux Mint"; echo ${#str}
10
```

Przykład: Wypisanie fragmentu zawartości zmiennej `str` od indeksu 2.

```
$ str="Linux Mint"; echo ${str:2}
nux Mint
```

Przykład: Wypisanie fragmentu zawartości zmiennej `str` od indeksu `start` i długości 2.

```
$ str="Linux Mint"; echo ${str:1:4}
inux
```

Przykład: Usunięcie ze zmiennej `str` pierwszego wstąpienia podciągu o wartości `"nt"`.

```
$ str="Ubuntu Mint"; echo ${str/nt}
Ubuu Mint
```

Przykład: Usunięcie ze zmiennej `str` wszystkich wstąpień podciągu o wartości `"nt"`.

```
$ str="Ubuntu Mint"; echo ${str//nt}
Ubuu Mi
```

Przykład: Zastąpienie w zmiennej `str` pierwszego wystąpienia ciągu o wartości `"Linux"` ciągiem `"OS"`.

```
$ str="Linux Mint Linux Debian"; echo ${str/Linux/OS}
OS Mint Linux Debian
```

Przykład: Zastąpienie w zmiennej `str` wszystkich wystąpień ciągów o wartości `"Linux"` ciągiem `"OS"`.

```
$ str="Linux Mint Linux Debian"; echo ${str//Linux/OS}
OS Mint OS Debian
```

Przykład: Wydzielenie ze zmiennej `str="3+5+8"` wybranych podciągów, przy założeniu, że separatorem jest symbol `"+"`.

```
$ str="3+5+8"; echo ${str#*+}
5+8
$ str="3+5+8"; echo ${str##*+}
8
$ str="3+5+8"; echo ${str%+*}
3+5
$ str="3+5+8"; echo ${str%%+*}
3
```

Przykład: Wyświetlenie wartości zmiennej `str` z pierwszą literą jako wielką.

```
$ str="ubuntu mint"; echo ${str^}
Ubuntu mint
```

Przykład: Wypisanie wartości zmiennej `str` wielkimi literami.

```
$ str="ubuntu mint"; echo ${str^^}
UBUNTU MINT
```

Przykład: Wypisanie wartości zmiennej `str` z pierwszą literą jako małą.

```
$ str="Ubuntu Mint"; echo ${str,}
ubuntu Mint
```

Przykład: Wypisanie wartości zmiennej `str` w postaci małych liter.

```
$ str="Ubuntu Mint"; echo ${str,,}
ubuntu mint
```

Przykład: Wypisanie wartości zmiennej `str` z zamianą liter "u" oraz "t" na wielkie.

```
$ str="Ubuntu Mint"; echo ${str^^[ut]}
UbUnTU MinT
```

Zadania

1. Napisz skrypt, do którego przekazane będą następujące 2 parametry: imię oraz nazwisko, zapisane znakami małymi i wielkimi w dowolnej postaci. Na wyjściu imię powinno być wypisane z wielkiej litery (pozostałe małymi), natomiast całe nazwisko wielkimi. Dodatkowo w nawiasach powinna być podana długość przekazanych parametrów.

   Przykład:
   ```
   $ ./skr_imie.sh jAn noWaK
   Imie: Jan (3)
   Nazwisko: NOWAK (5)
   ```

2. Napisz skrypt, do którego przekazana będzie zawartość dowolnej linii z pliku */etc/passwd*. Parametr będzie się więc składał z 7 pól przedzielonych dwukropkiem, z których na standardowe wyjście należy wypisać zawartość pól: 1 (nazwa użytkownika), 6 (katalog domowy) oraz 7 (powłoka).

Przykład:
```
$ ./skr_passwd.sh "student:x:1001:1005:gecos:
/home/student:/bin/bash"
Uzytkownik: student
Katalog domowy: /home/student
Powloka: /bin/bash
```

3. Napisz skrypt, który w przekazanym w postaci parametru ciągu znaków zamieni wszystkie samogłoski na wielkie, a spółgłoski na małe litery.

   Przykład:
   ```
   $ ./skr_litery.sh "Linux Debian"
   lInUx dEbIAn
   ```

4. Napisz skrypt, który parametr podany w postaci 5-cyfrowej wartości przedstawi w formie kodu pocztowego.

   Przykład:
   ```
   $ ./skr_kod.sh "15980"
   15-980
   ```

5. Napisz skrypt, który zmieni format przekazanego numeru telefonu z postaci `"(+YY) XXXXXXXXX"` na `"YY XXX XXX XXX"`.

   Przykład:
   ```
   $ ./skr_kod.sh "(+48)390245177"
   48 390 245 177
   ```

3.3 Operacje arytmetyczne

W powłoce `bash` można również wykonywać podstawowe operacje arytmetyczne (m.in. dodawanie, odejmowanie, mnożenie oraz dzielenie), ale jedynie

na zmiennych o wartościach całkowitych. Kolejność działań jest tu zachowywana.

Można w tym celu używać polecenia `let` lub `expr` oraz zamiennie nawiasów `[]` lub `()`. W przypadku użycia polecenia `let` należy pamiętać, że całość wyrażenia powinna być zapisana bez spacji.

Zupełnie odwrotnie jest w przypadku polecenia `expr`, gdzie spacje po każdej wartości i operatorze są obowiązkowe. Warto jednak zauważyć, że służy ono do wykonywania operacji jedynie na dwóch argumentach. Polecenia można oczywiście zagnieżdżać, ale wówczas należy pamiętać, aby symbole "`" (ang. *backtics*) poprzedzić "\", czyli np. "\`". W podobny sposób należy również postąpić w przypadku operatora mnożenia (tj. "*").

W przypadku ostatniej wspomnianej możliwości, czyli zapisu z użyciem nawiasów, spacje w wyrażeniu nie mają znaczenia.

Przykład: Obliczanie wartości wyrażenia `val=x+10*2` przy użyciu różnych poleceń, przy założeniu, że zmienna `x` została wcześniej zainicjalizowana.

```
$ let val=$x+10*2
$ val=`expr $x + \`expr 10 \* 2 \``
$ val=$(expr $x + `expr 10 \* 2`)
$ val=$[$x + 10 * 2]
$ val=$[$x+10*2]
$ val=$(($x + 10 * 2))
$ val=$(($x+10*2))
```

Po wykonaniu każdej z powyższych instrukcji wynik będzie identyczny. Zakładając, że zmiennej `x` została przypisana wartość `1`, uzyskamy wynik:

```
$ echo $val
21
```

W przypadku operacji dzielenia uzyskiwany wynik nie jest zaokrąglany, ale usuwana jest jego część rzeczywista, np.:

```
$ echo $[19/4]
4
```

Pewnym rozwiązaniem tego problemu może być użycie kalkulatora `bc`, np.:

```
$ echo "3.33*3" | bc
9.99
```

Tabela 3.2: Wybrane operatory arytmetyczne

Operatory	Opis
+, –, *, /	dodawanie, odejmowanie, mnożenie, dzielenie
**	potęgowanie
%	modulo (reszta z dzielenia)
++, —	inkrementacja, dekrementacja

Wybrane operatory arytmetyczne, dostępne w powłoce `bash`, zostały przedstawione w tabeli 3.2. W przypadku operatorów "++" oraz "——" można je stosować zarówno do preinkrementacji/dekrementacji, jak i do postinkrementacji/dekrementacji (np. `++zmienna`, `zmienna++`). Przykłady ich zastosowania przedstawiają poniższe polecenia:

```
$ a=4; let b=a++; echo $a; echo $b
4
4

$ a=4; let b=++a; echo $a; echo $b
4
5
```

Zadania

1. Napisz skrypt, który wypisze sumę oraz różnicę przekazanych parametrów (bez sprawdzania ich poprawności).

   Przykład:
   ```
   $ ./skr_suma.sh 7 5
   Suma: 12
   Roznica: 2
   ```

2. Napisz skrypt, który dla 4 przekazanych parametrów (`a`, `b`, `c` oraz `d`) wykona operację `"a*b+c/b"`, zachowując kolejność wykonywanych działań.

3. Napisz skrypt, który dla 2 przekazanych liczb wypisze wynik dzielenia w postaci liczby całkowitej oraz resztę z tego dzielenia.

   Przykład:
   ```
   $ ./skr_dzielenie.sh 37 5
   5 reszta 2
   ```

4. Napisz skrypt, który wypisze sumę długości 3 pierwszych parametrów.

   Przykład:
   ```
   $ ./skr_dlugosc.sh jeden dwa trzy cztery
   Dlugosc parametrow 1-3: 12
   ```

5. Napisz skrypt, który zsumuje wartości cyfr zawartych w max. 4-cyfrowej liczbie.

   Przykład:
   ```
   $ ./skr_sum_val.sh 4678
   25
   ```

Rozdział 4

Operatory oraz instrukcje warunkowe

Powłoka `bash` udostępnia bardzo szeroki wachlarz różnego rodzaju operatorów, które najczęściej są wykorzystywane do tworzenia instrukcji warunkowych, a także pętli. W rozdziale tym omówione zostaną różne typy operatorów wykorzystywanych do wykonywania operacji na wartościach liczbowych, ciągach znaków, jak również na zasobach plikowych systemu operacyjnego. Ich praktyczne zastosowanie zostanie przedstawione podczas omawiania instrukcji warunkowych `if` oraz `case`, jak też w dalszych rozdziałach w pętlach `for`, `while` oraz `until`.

4.1 Operatory

W powłoce `bash` dostępne są standardowe operatory porównania, zarówno dla wartości numerycznych, jak i alfanumerycznych. Całość uzupełniają operatory do wykonywania operacji na zasobach plikowych.

W przypadku przeprowadzania operacji na wartościach liczbowych dostępne są standardowe operatory porównania, tj. równości, nierówności, większości i mniejszości, które zostały przedstawione w tabeli 4.1.

Tabela 4.1: Operatory porównania wartości numerycznych

Operator	Opis (Prawda, gdy ...)
`arg1 -eq arg2`	`arg1` jest równy `arg2`
`arg1 -ne arg2`	`arg1` nie jest równy `arg2`
`arg1 -lt arg2`	`arg1` jest mniejszy niż `arg2`
`arg1 -le arg2`	`arg1` jest mniejszy lub równy `arg2`
`arg1 -gt arg2`	`arg1` jest większy niż `arg2`
`arg1 -ge arg2`	`arg1` jest większy lub równy `arg2`

Tabela 4.2: Operatory porównania ciągów znaków

Operator	Opis (Prawda, gdy ...)
`-z string`	długość ciągu `string` jest równa 0
`-n string`	długość ciągu `string` jest większa niż 0
`string1 = string2`	ciągi `string` oraz `string2` są identyczne
`string1 != string2`	ciągi `string1` oraz `string2` są różne
`string1 < string2`	w wyniku sortowania leksykograficznego `string1` znajdzie się przed `string2`
`string1 > string2`	w wyniku sortowania leksykograficznego `string1` znajdzie się po `string2`

W tabeli 4.2 przedstawiono operatory porównania dla ciągów znaków. Warto zauważyć, że w przypadku operatorów ">" oraz "<" rezultat jest uzależniony od wyniku sortowania leksykograficznego, w którym istotną rolę mogą odegrać ustawienia systemowe. W tabeli nie uwzględniono operatora "==", który może służyć zarówno do porównywania ciągów alfanumerycznych, jak i do wyszukiwania podciągów opisanych wyrażeniami regularnymi (patrz rozdział 7).

Dostępne są również operatory porównania dla zasobów plikowych, które bardzo często są wykorzystywane w skryptach. Służą one m.in. do weryfikacji, czy dany zasób istnieje, jakiego jest typu oraz jakie ma ustawione prawa dostępu. Wybrane operatory dla zasobów plikowych zostały przedstawione w tabeli 4.3.

Tabela 4.3: Operatory porównania dla zasobów plikowych

Operator	Opis (Prawda, gdy ...)
`-a filename`	plik `filename` istnieje (może być również pusty)
`-b filename`	plik `filename` istnieje i jest urządzeniem blokowym (np. dysk twardy */dev/sda* lub partycja `/dev/sda1`)
`-c filename`	plik `filename` istnieje i jest urządzeniem znakowym (np. terminal */dev/tty1*)
`-d filename`	zasób `filename` jest katalogiem
`-e filename`	stosowany zamiennie z operatorem `-a`
`-f filename`	zasób `filename` jest zwykłym plikiem (zasób nie może być katalogiem, urządzeniem blokowym ani znakowym, linkiem)
`-g filename`	zasób `filename` istnieje i ma ustawiony bit `SETGID`
`-h filename`	zasób `filename` istnieje i jest linkiem symbolicznym
`-k filename`	zasób `filename` istnieje i ma ustawiony `sticky bit`
`-p filename`	zasób `filename` istnieje i jest kolejką FIFO (nazwanym potokiem (ang. *named pipe*))
`-r filename`	zasób `filename` istnieje i ma ustawione prawa do odczytu
`-s filename`	plik `filename` istnieje i jego rozmiar jest większy niż 0; jeżeli plik jest pusty, zwracana jest wartość `false`
`-t fd`	deskryptor `fd` jest otwarty i wskazuje na terminal
`-u filename`	zasób `filename` istnieje i ma ustawiony bit `SETUID`
`-w filename`	zasób `filename` istnieje i ma ustawione prawa do zapisu
`-x filename`	zasób `filename` istnieje i ma ustawione prawa do uruchomienia
`-G filename`	plik `filename` istnieje i jego właścicielem jest efektywna grupa
`-L filename`	plik `filename` istnieje i jest linkiem symbolicznym
`-N filename`	plik `filename` istnieje i był modyfikowany od czasu ostatniego odczytu

Operator	Opis (Prawda, gdy ...)
`-O filename`	plik `filename` istnieje i jego właścicielem jest efektywny użytkownik
`-S filename`	plik `filename` istnieje i jest gniazdem sieciowym (ang. *socket*))
`file1 -ef file2`	zasoby `file1` oraz `file2` odnoszą się do tego samego *i*-węzła
`file1 -nt file2`	zasób `file1` jest nowszy (na podstawie daty modyfikacji) niż `file2` lub gdy `file1` istnieje i jednocześnie nie istnieje `file2`
`file1 -ot file2`	zasób `file1` jest starszy niż `file2` lub gdy `file2` istnieje i jednocześnie nie istnieje `file1`

4.2 Instrukcje warunkowe

4.2.1 Instrukcja `if`

Powłoka `bash` udostępnia instrukcje warunkowe `if` dostępne w wielu wariantach, które dodatkowo mogą być zapisywane w różnych postaciach. Podstawowa ich składnia ma postać:

```
if [ warunek ]
then
  instrukcja 1
  instrukcja 2
fi
```

Należy jednak pamiętać, że spacje przed nawiasami i po nawiasach otaczających warunek są niezbędne, a ich brak spowoduje, że interpreter wygeneruje błąd składniowy.

Przykład: Sprawdzenie, czy zmienna `val` zawiera wartość `"ok"`.

```
if [ "$val" == "ok" ]
then
  echo "True"
fi
```

Alternatywnie zamiast nawiasów "[ ]" można użyć funkcji `test`.

```
if test warunek
then
  instrukcja 1
  instrukcja 2
fi
```

Przykład: Sprawdzenie, z wykorzystaniem polecenia `test`, czy zmienna `val` zawiera wartość `"ok"`.

```
if test "$val" == "ok"
then
  echo "True"
fi
```

Powyższe instrukcje można również zapisać w pojedynczej linii. Należy wówczas pamiętać, że po warunku, jak również po każdej instrukcji, powinien pojawić się średnik, np.

```
if [ warunek ]; then  instrukcja 1; instrukcja 2; fi
if test warunek; then  instrukcja 1; instrukcja 2; fi
```

Przykład: Warunek zapisany w postaci jednoliniowej sprawdzający, czy zmienna `val` zawiera wartość `ok`.

```
if [ "$val" == "ok" ]; then  echo "OK"; fi
```

Warto zauważyć, że w powyższych przykładach zmienna `val` została otoczona cudzysłowem, co formalnie nie jest wymagane. Zabezpiecza to jednak przed ewentualnymi błędami w przypadku, gdyby zawierała ona np. spację. W ramach testów warto sprawdzić powyższy warunek dla wartości np. `val="Jan Nowak"`. W przypadku braku cudzysłowu podczas wykonywania skryptu pojawiłby się następujący błąd:

```
bash: [: za dużo argumentów.
```

W przypadku instrukcji warunkowych (jak również pętli) można stosować operator negacji (symbol `"!"` przed warunkiem):

```
if [ ! warunek ]
then
  instrukcja 1
  instrukcja 2
fi
```

a w przypadku funkcji `test`:

```
if test ! warunek
then
  instrukcja 1
  instrukcja 2
fi
```

Przykład: Sprawdzenie, czy zmienna `val` nie zawiera wartości `"ok"`.

```
if [ ! "$val" == "ok" ]
then
  echo "False"
fi
```

W pozostałych przykładach składnia będzie prezentowana zwykle z użyciem nawiasów kwadratowych.

Instrukcja warunkowa `if` dostępna jest również w formacie `if-else-fi` oraz `if-elif-else-fi`, np.:

```
if [ warunek1 ]
then
  instrukcja 1
  instrukcja 2
elif [ warunek2 ]
then
  instrukcja 3
  instrukcja 4
else
  instrukcja 5
  instrukcja 6
fi
```

Przykład: Sprawdzenie, czy wartość bezwzględna wartości zmiennej `val` jest większa lub równa 5.

```
if [ $val -ge 5 ]
then
  echo "True"
elif [ $val -le -5 ]
then
  echo "True"
else
  echo "False"
fi
```

Na zakończenie warto przedstawić kilka wybranych warunków operujących na zasobach plikowych (przedstawionych w tabeli 4.3), które dotychczas nie zostały użyte w żadnym z prezentowanych przykładów.

Przykład: Sprawdzenie, czy ścieżka zapisana w zmiennej `path` wskazuje na katalog.

```
path="/var/log"
if [ -d "$path" ]
then
  echo "Podana sciezka wskazuje na katalog"
fi
```

Przykład: Sprawdzenie, czy bieżący użytkownik ma prawa do zapisu (`w`) oraz uruchamiania (`x`) do pliku określonego ścieżką `path`.

```
path="/tmp/skr.sh"
if [ -w "$path" -a -x "$path" ]
then
  echo "Ustawione prawa do zapisu i uruchamiania"
fi
```

Złożenia instrukcji warunkowych

Instrukcje warunkowe mogą składać się z więcej niż jednego warunku. W tym celu można użyć operatorów logicznych `OR` (`||` lub `-o`) oraz `AND` (`&&` lub `-a`).

W przypadku alternatywy instrukcja warunkowa może być zapisana w jednym z poniższych formatów:

```
if [ warunek1 ] || [ warunek2 ]
if [ warunek1 -o warunek2 ]
```

natomiast w przypadku koniunkcji:

```
if [ warunek1 ] && [ warunek2 ]
if [ warunek1 -a warunek2 ]
```

Przykład: Sprawdzenie, czy wartość bezwzględna wartości zmiennej `val` jest większa niż 5.

```
if [ $val -gt 5 ] || [ $val -lt -5 ]
then
  echo "True"
else
  echo "False"
fi
```

Uproszczone postaci instrukcji warunkowych

Często w skryptach możemy się spotkać z uproszczonym zapisem instrukcji `if`, w których główne słowo kluczowe identyfikujące ten typ warunku w ogóle nie występuje.

Ogólna postać składni takich warunków przedstawia się następująco:

```
[ warunek ] && lista operacji, gdy warunek jest spelniony
[ warunek ] || lista operacji, gdy warunek nie jest spelniony
```

Przykład: Weryfikacja, czy do skryptu został przekazany co najmniej 1 parametr. Jeśli nie jest spełniony warunek, powinien zostać wypisany stosowny komunikat oraz zwrócona wartość 1.

```
[ $# -eq 0 ] && echo "Brak argumentow"; exit 1
[ $# -gt 0 ] || echo "Brak argumentow"; exit 1
```

Wspomniana składnia może być również wykorzystana do zastąpienia instrukcji `if-else`. W tym przypadku należy pamiętać, aby zestawy instrukcji wykonywane zarówno przy spełnionym, jak i niespełnionym warunku zostały otoczone nawiasami klamrowymi.

Przykład: Przykład instrukcji do weryfikowania, czy wartość bezwzględna zmiennej `val` jest większa niż 5.

```
[ $val -lt -5 -o $val -gt 5 ] && { echo "wartosc bezwzgledna > 5";
            exit 0; } || { echo "wartosc bezwzgledna <=5"; exit 1; }
```

Zadania

1. Napisz skrypt, który będzie działał jako funkcja `max` dla dwóch przekazanych do skryptu wartości. Skrypt powinien sprawdzać, czy liczba przekazanych parametrów jest prawidłowa.

Przykład:

```
$ ./skr_max.sh 55 66
66

$ ./skr_max.sh 55
Bledna liczba parametrow. Podaj 2 argumenty.
```

2. Napisz skrypt, który będzie pełnił funkcję prostego kalkulatora wykonującego operacje dodawania, odejmowania, dzielenia, mnożenia, potęgowania, reszty z dzielenia dla dwóch wartości. Należy weryfikować m.in. czy liczba przekazanych parametrów jest poprawna oraz czy operator jest prawidłowy.

   Przykład:

```
$ ./skr_kalk.sh 3 + 8
11

$ ./skr_kalk.sh 3 # 8
Blad: nieprawidlowy operator.
```

3. Napisz grę „Papier, kamień, nożyce". Dane powinny być przekazywane w postaci dwóch parametrów. Jeżeli liczba parametrów jest błędna (inna niż 2), wówczas powinna zostać wypisana pomoc do gry. Należy weryfikować poprawność przekazywanych danych.

   Przykład:

```
$ ./gra_pkn.sh
Bledna liczba parametrow. Podaj 2 z 3
nastepujacych argumentow: papier, kamien, nozyce.

$ ./gra_pkn.sh papier kamien
Wygral gracz nr 1.

$ ./gra_pkn.sh nozyce nozyce
Remis.

$ ./gra_pkn.sh nozyce glaz
Blad: Nieznana wartosc "glaz"
```

4. Popraw funkcjonalność gry „Papier, kamień, nożyce” w taki sposób, aby tylko jeden parametr był przekazywany do skryptu (wybór użytkownika). Wartość gracza komputerowego powinna być losowana z wykorzystaniem zmiennej środowiskowej `RANDOM`, której wartość jest z przedziału 0–32767. Można założyć, że jeżeli zostanie wylosowana wartość z przedziału 0–10000 oznacza to opcję "papier", 10001–20000 oznacza "kamien", a powyżej 20000 – wartość "nozyce". Wybór użytkownika i komputera powinien być wypisywany na `stdout`, przy czym wybór komputera powinien być wypisany dodatkowo z wylosowaną wartością liczbową.

   Przykład:

```
$ ./gra_pkn_2.sh
Bledna liczba parametrow. Podaj 1 z 3 nastepujacych
argumentow: papier, kamien, nozyce.

$ ./gra_pkn_2 papier
Wybor uzytkownika: papier
Wybor komputera: kamien (15987)
Wygrales.

$ ./gra_pkn_2 papier
Wybor uzytkownika: papier
Wybor komputera: nozyce (29673)
Wygral komputer.

$ ./gra_pkn_2 nozyce
Wybor uzytkownika: papier
Wybor komputera: nozyce (29673)
Remis.
```

4.2.2 Instrukcja case

Instrukcja `case` występuje w wielu językach programowania i służy do wyboru określonego wariantu. Oczywiście można ją zastąpić konstrukcją instrukcji

warunkowej `if-elif-else-fi`, choć w przypadku licznych wariantów staje się ona mniej czytelna.

Składnia instrukcji `case` przedstawia się następująco:

```
case zmienna in
  "wzorzec_1") instrukcje_1;;
  "wzorzec_2") instrukcje_2;;
  ...
  "wzorzec_n") instrukcje_n;;
            *) instrukcje_domyslne
esac
```

Jak można zauważyć na powyższym przykładzie, składa się ona z kolejnych wzorców (lub wartości), które może przyjmować dana zmienna, zakończonych symbolem `")"`, oraz instrukcji (lub zestawu instrukcji przedzielonych średnikiem), które będą wykonane dla danej wartości. Wartość oznaczona `"*"` jest nieobowiązkowa i określa instrukcje, które powinny zostać wykonane, gdy żaden z wcześniej wymienionych wzorców nie został dopasowany. Każdy z warunków, oprócz ostatniego, musi być zakończony podwójnym średnikiem. Pojedynczy średnik będzie interpretowany jako separator zestawu poleceń, które zostaną wykonane, gdy spełniony zostanie odpowiedni warunek. Warunek kończy się słowem kluczowym `esac`, czyli odwrotnością słowa `case`.

Przykład: Wypisanie nazwy dystrybucji Linux w zależności od wartości zmiennej `linux`. Dodatkowo w przypadku dystrybucji Debian wypisywana jest zawartość pliku */etc/issue*.

```
case $linux in
    "mint") echo "Linux Mint";;
  "debian") echo "Debian"; cat /etc/issue;;
  "redhat") echo "Red Hat";;
         *) echo "Linux"
esac
```

Zadania

1. Napisz skrypt z wykorzystaniem instrukcji `case`, który przetłumaczy nazwę dnia tygodnia podaną w języku polskim na odpowiednik angielski. W przypadku podania nieprawidłowej wartości powinien zostać wygenerowany błąd.
2. Napisz skrypt z wykorzystaniem instrukcji `case` realizujący funkcjonalność prostego kalkulatora opisanego w jednym z zadań z poprzedniego rozdziału.
3. W zadaniu zdefiniowanym w rozdziale 4.2.1, w grze „Papier, kamień, nożyce", zmień sposób zamiany zmiennej `RANDOM` na odpowiedni typ, wykorzystując instrukcję `case`.

4.3 Kody wyjścia

Wprowadzone w tym rozdziale instrukcje warunkowe umożliwiają zakończenie działania skryptów w różnych jego miejscach, niekoniecznie po wykonaniu wszystkich poleceń w nim zawartych. Podobnie jak w przypadku uruchomianych poleceń systemowych, również i skrypty zwracają numeryczne kody wyjścia. Przyjęło się, aby w przypadku poprawnego zakończenia zwracana była wartość 0 (wartość domyślna). Kod wyjścia ostatnio zakończonej operacji przechowywany jest w zmiennej `$?`.

Przykład: Wyświetlenie kodu wyjścia zwróconego przez polecenie systemowe `type`.

```
$ type passwd
passwd jest /usr/bin/passwd
```

```
$ echo $?
0
$ type anyfile 2>/dev/null
$ echo $?
1
```

Jak można zauważyć na powyższym przykładzie, w przypadku pierwszego polecenia została zwrócona wartość 0, co może oznaczać, że zakończyło się ono powodzeniem, gdyż testowane polecenie `passwd` istnieje i można było zweryfikować jego typ. Natomiast drugi z zasobów `anyfile` nie istnieje i stąd został zwrócony kod błędu o wartości 1. Dodatkowo użyte w poleceniu przekierowanie standardowego wyjścia (`stderr`) do pliku */dev/null* zapobiegło wypisaniu komunikatu o błędzie.

Listing 4.1: Przykładowy skrypt weryfikujący liczbę przekazanych parametrów oraz sprawdzający, czy zawierają one wartość dodatnią i dodatkowo są uszeregowane w kolejności rosnącej

```
#!/bin/bash

if [ $# -eq 0 ]; then
  echo "Blad: Brak parametrow"
  exit 100
fi

if [ $1 -lt 0 ]; then
  echo "Blad: Tylko wartosci dodatnie sa akceptowane"
  exit 101
fi

if [ -z "$2" ]; then
  echo "Info: Brak drugiego parametru. Konczymy ..."
  exit 0
fi

if [ $2 -lt $1 ]; then
  echo "Blad: Drugi z parametrow powinien byc wiekszy niz pierwszy"
  exit 102
fi
```

Wewnątrz skryptów do zwracania kodu błędu i jednocześnie zakończenia jego działania w danym miejscu służy polecenie `exit`, którego opcjonalnym parametrem jest kod wyjścia (domyślnie zwracana jest wartość 0), np.:

```
exit 100
```

Na listingu 4.1 przedstawiony jest skrypt, który w zależności od spełnienia pewnych warunków kontynuuje swoje działanie, zwracając przy tym unikalny kod wyjścia. Warto zauważyć, że skrypt nie kończy się poleceniem `exit` (lub `exit 0`), co jest opcjonalne, gdyż wówczas domyślnie będzie zwracana wartość 0. W przypadku uruchomienia powyższego skryptu możemy uzyskać następujące wyniki:

```
$ ./skrypt_exit.sh
Blad: Brak parametrow
$ echo $?
100
$ ./skrypt_exit.sh -6
Blad: Tylko wartosci dodatnie sa akceptowane
$ echo $?
101
$ ./skrypt_exit.sh 7 9
$ echo $?
0
```

Listing 4.2: Przykładowy skrypt, w którym weryfikowany jest kod wyjścia polecenia uruchamianego w jego wnętrzu

```
#!/bin/bash

ls -al /home/student 2>/dev/null
if [ $? -ne 0 ];
then
  echo "Blad podczas wyswietlania zawartosci katalogu"
  exit 100
fi
```

Z kodów wyjścia można również korzystać wewnątrz skryptu weryfikując, czy wykonywane polecenie zakończyło się powodzeniem, co zostało zaprezentowane na listingu 4.2. W przypadku wystąpienia błędu podczas wyświetlenia zawartości wskazanego katalogu skrypt zwróci wartość 100.

Warto tu wspomnieć o bardzo częstych błędach popełnianych przez początkujących programistów w odniesieniu do analizy kodów wyjścia. Wybrane przykłady zostały zaprezentowane na listingu 4.3 i oznaczone komentarzem UWAGA.

Listing 4.3: Przykładowy skrypt ilustrujący często popełniane błędy związane z analizą wartości kodu wyjścia

```
#!/bin/bash

ls -al /home/student 2>/dev/null
echo $?     #UWAGA
if [ $? -ne 0 ];
then
  exit $? #UWAGA
fi
```

W pierwszym przypadku przed warunkiem weryfikującym wartość zmiennej `$?` zostało dodane polecenie wyświetlające jej zawartość. W efekcie w warunku `if` będzie sprawdzany kod wyjścia polecenia `echo` zamiast `ls`.

Drugi z komentarzy został umieszczony w linii, której prawdopodobnym celem było zwrócenie przez skrypt kodu błędu identycznego z wartością zwracaną przez polecenie `ls`. Nawet w przypadku zakomentowania wcześniej wspomnianego polecenia `echo` polecenie `exit` zwróci kod wyjścia warunku `if`, gdyż warunek `if` to po prostu inna postać polecenia `test`.

Przykładowe rozwiązanie tego problemu zostało zaprezentowane na listingu 4.4, w którym ze względu na konieczność wielokrotnego operowania na wartości kodu błędu zwróconego przez polecenie (w tym przypadku `ls`) jego wartość została zapisana w zmiennej `ret`.

Listing 4.4: Przykładowy skrypt zwracający jako kod wyjścia wartość kodu błędu jednego z wykonywanych poleceń

```
#!/bin/bash

ls -al /home/student 2>/dev/null
ret=$?
if [ $ret -ne 0 ];
then
  exit $ret
fi
```

Zadania

1. Napisz skrypt, który zwróci w postaci symbolicznej (lub numerycznej) prawa dostępu aktualnie zalogowanego użytkownika do określonego zasobu. W przypadku, gdy wskazany zasób nie istnieje, powinien zostać wypisany komunikat i zwrócony kod błędu `100`.

   Przykład:

   ```
   $ ./skr_perm.sh /etc/passwd
   rw-

   $ ./skr_perm.sh /tmp/test
   Brak zasobu: /tmp/test
   $ echo $?
   100
   ```

2. W ramach zadań z tego rozdziału uzupełnij skrypty o zwracanie kodów błędów w przypadku, gdy nie kończą się poprawnie.

Rozdział 5

Pętle

W powłoce `bash` istnieje możliwość stosowania zarówno pętli `for`, jak i `while` oraz `until`. W każdej z nich można stosować operacje `break` oraz `continue`, których znaczenie jest identyczne jak w wielu powszechnie znanych językach, tj. pierwsza z nich przerywa wykonywanie całej pętli, a druga aktualnie wykonywaną iterację.

5.1 Pętla `for`

Schemat pętli `for` przedstawia się następująco:

```
for wartosc_poczatkowa in zakres
do
  instrukcja 1
  instrukcja 2
done
```

Przykłady praktycznego wykorzystania pętli `for` zostały zaprezentowane na listingu 5.1. Jak można zauważyć, zakres wartości, na których będzie wykony-

wana pętla, można definiować na różne sposoby, tj. podając listę konkretnych wartości oraz używając zakresów (m.in. za pomocą nawiasów klamrowych lub polecenia `seq`). Można również, co zostało przedstawione na ostatnim z przykładów, zastosować strukturę znaną z języka C.

Listing 5.1: Przykładowy skrypt prezentujący różne składnie pętli `for` do wypisywania wartości z przedziału od 1 do 5

```
 #!/bin/bash

#Przyklad 1
 for i in 1 2 3 4 5
   do
     echo "Wartosc: $i"
  done

#Przyklad 2
 for i in `seq 1 5`;
   do
     echo Wartosc: $i
 done

#Przyklad 3
 for i in {1..5};
   do
     echo Wartosc: $i
 done

 #Przyklad 4
 for (( i=1; i<=5; i++))
   do
     echo Wartosc: $i
 done
```

Zakresy wartości, na których operują pętle, nie muszą składać się z kolejnych wartości liczb całkowitych. Poniżej zaprezentowane zostały przykłady definiowania zakresów z krokiem innym niż 1, zarówno o wartości dodatniej, jak i ujemnej.

Przykład: Wypisanie co trzeciej wartości z przedziału od 1 do 20 z wykorzystaniem różnych składni.

```
#Przyklad 1
 for ((i=1;i<=20;i+=3)
 do
 done

#Przyklad 2
 for i in {0..20..3}
 do
   echo Wartosc: $i
 done

#Przyklad 3
 for i in `seq 1 3 20`
 do
   echo Wartosc: $i
 done
```

Przykład: Wypisanie w kolejności malejącej co trzeciej wartości z przedziału od 20 do 1 z wykorzystaniem różnych składni.

```
#Przyklad 1
 for ((i=20;i>=1;i-=3)
   do
   echo Wartosc: $i
 done

#Przyklad 2
 for i in {20..0..-3}
   do
   echo Wartosc: $i
 done

#Przyklad 3
 for i in `seq 20 -3 1`;
   do
   echo Wartosc: $i
 done
```

Inny ciekawy przykład wykorzystania pętli `for` został przedstawiony na listingu 5.2, w którym to zamiast iterowania po liczbach całkowitych pętla jest wykonywana dla każdego pliku z rozszerzeniem `".sh"`, który znajduje się w bieżącym katalogu.

Listing 5.2: Wyświetlanie w pętli `for` listy plików z rozszerzeniem `".sh"` z bieżącego katalogu

```
#!/bin/bash

for i in ls *.sh; do
  echo $i;
done
```

5.2 Pętle `while` oraz `until`

Pętle `while` oraz `until` mają podobne zastosowanie i strukturę. Służą do wykonywania wskazanego zestawu operacji, dopóki wskazany warunek jest spełniony (`while`) lub niespełniony (`until`). Jest to zasadnicza różnica w odniesieniu do innych popularnych języków programowania, w których pętle te różnią się kolejnością sprawdzania warunku (przed wykonaniem instrukcji czy po wykonaniu).

Składnie pętli przedstawiają się następująco:

```
while [ warunek ]
do
  instrukcja 1
  instrukcja 2
done

until [ warunek ]
do
  instrukcja 1
  instrukcja 2
done
```

W przypadku zapisania w formie jednoliniowej ich postać przedstawia się następująco:

```
while [ warunek ]; do  instrukcja 1;  instrukcja 2; done
until [ warunek ]; do  instrukcja 1;  instrukcja 2; done
```

Przykładowy skrypt wykorzystujący pętlę `while` do wypisania wartości z zakresu od 0 do 9 został przedstawiony na listingu 5.3. Warto zauważyć, że w przypadku iterowania po wartościach z zadanym krokiem należy pamiętać o zmianie tej wartości wewnątrz pętli.

Listing 5.3: Przykładowy skrypt prezentujący wykorzystanie pętli `while`

```
#!/bin/bash

licznik=0
while [  $licznik -lt 10 ]; do
  echo Licznik: $licznik
  let licznik=$licznik+1
done
```

Zadania

1. Napisz skrypt, który wypisze co drugą wartość z zakresu przekazanego w postaci parametrów. Jeżeli pierwszy parametr jest większy od drugiego, wówczas liczby powinny być wypisywane w kolejności od największej do najmniejszej.

   Przykład:

   ```
   $ ./skr_every_second.sh 11 16
   11
   13
   15
   ```

```
$ ./skr_every_second.sh 10 5
10
8
6
```

2. Napisz skrypt wypisujący z przekazanego jako parametry zakresu wszystkie liczby podzielne przez liczbę podaną jako trzeci parametr. Jeżeli pierwszy parametr jest większy od drugiego, wówczas liczby powinny być wypisywane w kolejności od największej do najmniejszej.

   Przykład:
   ```
   $ ./skr_div_3.sh 3 25 7
   7
   14
   21
   ```

3. Napisz grę, w której użytkownik zgaduje liczbę wylosowaną za pomocą zmiennej środowiskowej `RANDOM`. Po każdej nieudanej próbie wskazanej przez użytkownika skrypt powinien generować komunikat, czy wylosowana liczba jest od niej mniejsza, większa czy też równa. W tym ostatnim przypadku oznacza to koniec gry, a na standardowym wyjściu powinien pojawić się komunikat o liczbie wykonanych prób prowadzących do odgadnięcia wylosowanej wartości.

4. Napisz skrypt, który w pętli `for` będzie sprawdzał, czy wprowadzone z `stdin` (funkcja `read`) hasło jest prawidłowe. Po trzeciej nieudanej próbie program powinien zakończyć się błędem. Prawidłowe hasło powinno być ustawione na stałe wewnątrz skryptu.

   Przykład:
   ```
   $ ./skrypt_passwd.sh
   Wprowadz haslo:
   Haslo prawidlowe.
   ```

   ```
   $ ./skrypt_passwd.sh
   Wprowadz haslo:
   Haslo nieprawidlowe!
   ```

```
Wprowadz haslo:
Haslo nieprawidlowe!
Wprowadz haslo:
Haslo nieprawidlowe!
Nieudana próba uwierzytelnienia uzytkownika.
```

5. Powtórz zadanie z poprzedniego punktu, ale tym razem użyj pętli `while`. Ponadto operacje na hasłach wykonuj w postaci niejawnej, np. po przepuszczeniu przez funkcję `md5sum`. Przykładowo, w celu uzyskania formy niejawnej dla ciągu "haslo" można użyć polecenia:

   ```
   val=‘echo "haslo" | md5sum‘
   ```

6. Napisz skrypt, którego parametry w postaci "klucz=wartosc" zostaną zapisane do tablicy asocjacyjnej, a następnie wypisane na `stdout`. Do wydzielenia klucza i wartości użyj wyrażeń opisanych w podrozdziale 3.2.

   Przykład:
   ```
   $ ./skr.sh imie=Henryk nazwisko=Sienkiewicz tytul=Potop
   Liczba parametrow: 3
   imie: Henryk
   nazwisko: Sienkiewicz
   tytul: Potop
   ```

7. Napisz skrypt, którego parametry w postaci "–klucz wartosc" zostaną zapisane do tablicy asocjacyjnej, a następnie wypisane na `stdout`. Uwaga: nie każdy klucz musi mieć przypisaną wartość.

   Przykład:
   ```
   $ ./skr.sh -rozmiar 100 -d -k test
   Liczba parametrow: 3
   rozmiar: 100
   d:
   k: test
   ```

8. Napisz skrypt imitujący działanie polecenia `id`.

Rozdział 6

Tablice

W powłoce `bash` dane mogą być przechowywane w jednowymiarowych tablicach indeksowanych, a od wersji 4.0 również i asocjacyjnych. W celu sprawdzenia wersji używanej powłoki można użyć polecenia:

```
$ bash --version
```

Podobnie jak w wielu innych językach programowania, powłoka `bash` umożliwia wykonywanie wielu podstawowych operacji na tablicach, w tym dodawanie i usuwanie elementów, oraz m.in. łączenie wielu tablic.

6.1 Tablice indeksowane

W tablicach indeksowanych elementy są numerowane od 0, a ich dostępna maksymalna wielkość limitowana jest zakresem zmiennej typu `integer`. Tablice są deklarowane w sposób analogiczny jak zwykłe zmienne. Podczas ich tworzenia można zainicjalizować je wartościami początkowymi, w razie konieczności (jeżeli np. zawierają spacje) otaczając je cudzysłowem, np.:

```
tablica=(jeden dwa "dwadziescia trzy")
```

Bardziej formalnie można to zrobić z wykorzystaniem polecenia `declare -a`, które w przypadku tablic indeksowanych nie jest obowiązkowe, np.:

```
declare -a tablica=(jeden dwa "dwadziescia trzy")
```

Ten sam efekt można uzyskać, tworząc pustą tablicę (co również nie jest obowiązkowe, ale często stosowane), a następnie przypisując do niej wartości, np.:

```
tablica=()                  #opcjonalnie
tablica[0]=zero
tablica[1]=jeden
tablica[2]="dwadziescia trzy"
```

Elementy można dodawać w dowolnej kolejności, niekoniecznie uzupełniając wszystkie kolejne indeksy. I tak np. w powyższym przykładzie można dodać element o indeksie 4, pomijając 3, np.:

```
tablica[4]=cztery
```

Możliwe jest również dodawanie elementów na końcu tablicy za pomocą operatora `+=` poprzez otaczanie dodawanych wartości nawiasami okrągłymi, np.:

```
tablica+=(piec szesc siedem)
```

Jest to ułatwienie w uzupełnianiu elementów tablicy, gdyż znajomość ostatnio zajętego indeksu nie jest konieczna. W powyższym przypadku pierwszy z dodawanych elementów o wartości `szesc` zostanie zapisany pod indeksem 5, jako że jest to pierwszy wolny po ostatnio zajętym. Element o indeksie 3 nadal będzie więc niewykorzystany.

Odwoływanie się do poszczególnych elementów tablicy jest realizowane za pomocą zmiennej `${tablica[indeks]}`, np.:

```
$ echo "Element tablicy o indeksie 4: " ${tablica[4]}
Element tablicy o indeksie 4: piec
```

W operowaniu na elementach tablicy przydatne są następujące zmienne, wykorzystywane m.in. do wypisywania ich zawartości w pętli:

- `${#tablica[@]}` – liczba elementów w tablicy, a w zasadzie powiększony o 1 indeks ostatniego elementu;
- `${tablica[@]}` – wszystkie elementy tablicy przedzielone separatorem w postaci spacji;
- `${!tablica[@]}` – indeksy wszystkich niepustych elementów tablicy.

W przypadku analizowanej tablicy wspomniane zmienne przyjmują następujące wartości:

```
$ echo "Liczba elementow tablicy: "${#tablica[@]}
Liczba elementow tablicy: 7
$ echo "Lista elementow tablicy: "${tablica[@]}
Lista elementow tablicy: jeden dwa dwadziescia trzy piec szesc
    siedem
$ echo "Lista wykorzystywanych indeksow tablicy: "${!tablica[@]}
Lista wykorzystywanych indeksow tablicy: 0 1 2 4 5 6 7
```

W tablicy można usuwać poszczególne elementy, jak również całą tablicę, np.:

```
$ unset ${#tablica[2]}          #usunięcie elementu o indeksie 2
$ unset ${#tablica[@]}          #usunięcie całej tablicy
```

Warto zauważyć, że przypisanie ciągu pustego do elementu tablicy nie usuwa go, co można sprawdzić, korzystając ze zmiennej `${#tablica[@]}`.

Przykłady różnych metod wypisywania zawartości tablicy indeksowanej zostały przedstawione na listingu 6.1.

Listing 6.1: Wybrane metody wypisywania zawartości tablicy indeksowanej

```
#!/bin/bash

tab=(1 2 3 4)

#Przyklad 1
for val in "${tab[@]}"; do
    echo "$val"
done

#Przyklad 2
for ((i=0; i < ${#tab[@]}; ++i)); do
    echo "${tab[$i]}"
done
```

6.2 Tablice asocjacyjne

W przypadku tworzenia tablic asocjacyjnych użycie polecenia `declare -A` jest obowiązkowe, np.:

```
declare -A tablicaAsoc=([kolor]=bialy [ksztalt]="trojkat
prostokatny" ["dlugosc boku"]=8)
```

Warto zwrócić uwagę na konieczność użycia w powyższym przykładzie cudzysłowu, zarówno w przypadku wartości, jak i nazwy klucza, gdy zawierają one spacje.

Dodawanie elementów do tablicy asocjacyjnej jest analogiczne jak w przypadku tablic indeksowanych, np.:

```
tablicaAsoc+=([kat]=60 [obrot]=90)
```

lub

```
tablicaAsoc[kat]=60
tablicaAsoc[obrot]=90
```

Odwoływanie się do poszczególnych elementów tablicy asocjacyjnej jest realizowane za pomocą zmiennej `${tablicaAsoc[klucz]}`, np.:

```
$ echo "Element tablicy z kluczem 'kat': ${tablicaAsoc[kat]}"
Element tablicy z kluczem 'kat': 60
```

Podobnie jak w przypadku tablic indeksowanych, można usuwać zarówno poszczególne jej elementy, jak i całą tablicę, np.:

```
$ unset tablicaAssoc[obrot]      #usunięcie elementu o kluczu 'obrot'
$ unset tablicaAssoc             #usunięcie całej tablicy
$ unset tablicaAssoc[@]          #usunięcie całej tablicy
```

Dostęp do listy wszystkich elementów tablicy jest realizowany poprzez zmienną `${!tablicaAsoc[@]}`. Natomiast w zmiennej `${!#tablicaAsoc[@]}` zapisana jest liczba elementów tablicy.

Przykłady różnych metod wypisywania zawartości tablicy asocjacyjnej zostały przedstawione na listingu 6.2.

Listing 6.2: Wbrane metody wypisywania zawartości tablicy asocjacyjnej

```
#!/bin/bash

#inicjalizacja tablicy
declare -A tab_asoc=([kolor]=bialy [ksztalt]="trojkat prostokatny" \\
["dlugosc boku"]=8 [kat]=60 [obrot]=90)

# wypisanie tylko wartosci
for str in ${tab_asoc[@]}; do
  echo $str
done

# wypisanie pary "klucz : wartosc"
for klucz in "${!tab_asoc[@]}"; do
  echo "$klucz : ${tab_asoc[$klucz]}"
done
```

Zadania

1. Napisz skrypt, który w oddzielnych liniach wypisze wszystkie parametry przekazane do skryptu oraz ich liczbę (bez wykorzystania zmiennej `$#`).

2. Utwórz tablicę indeksowaną z pięcioma kolejnymi elementami, o wartościach równych poszczególnym indeksom. Wypisz wszystkie jej elementy, ich liczbę oraz listę wykorzystywanych indeksów. Następnie wyzeruj wybrany element, np. o indeksie 2, podstawiając ciąg pusty "", oraz usuń za pomocą polecenia `unset` element o indeksie 3. Porównaj zawartości zmiennych o indeksach 2 oraz 3.

3. Powtórz poprzednie zadanie z wykorzystaniem tablicy asocjacyjnej. Jako klucze użyj słownego zapisu cyfr (np. "jeden", "dwa", ...).

4. Napisz skrypt, którego parametry w postaci "klucz=wartosc" zostaną zapisane do tablicy asocjacyjnej, a następnie wypisane na `stdout`.

   Przykład:
   ```
   $ ./skr.sh imie=Henryk nazwisko=Sienkiewicz tytul=Potop
   Liczba parametrow: 3
   imie: Henryk
   nazwisko: Sienkiewicz
   tytul: Potop
   ```

5. Napisz skrypt, którego parametry w postaci "-klucz wartosc" zostaną zapisane do tablicy asocjacyjnej, a następnie wypisane na standardowe wyjście. Należy również uwzględnić, że nie każdy klucz musi mieć przypisaną wartość.

   Przykład:
   ```
   $ ./skrypt_asoc.sh -rozmiar 100 -d -k test
   Liczba parametrow: 3
   rozmiar: 100
   d:
   k: test
   ```

6. Napisz skrypt, który zapisze poszczególne linie z pliku `/etc/passwd` do oddzielnych elementów tablicy indeksowanej. Wypisz zawartość tablicy.

7. Napisz skrypt, który zapisze poszczególne linie z pliku `/etc/passwd` do oddzielnych elementów tablicy asocjacyjnej, w której kluczem jest identyfikator użytkownika. Wypisz zawartość tablicy.

8. Napisz kalkulator wykonujący operacje arytmetyczne przekazane w postaci parametrów z zachowaniem kolejności ich wykonywania.

   Przykład:
   ```
   $ ./skr_kalkulator.sh 2 + 3 * 4
   14
   ```

Rozdział 7

Wyrażenia regularne

7.1 Idea

Skrypty są bardzo często wykorzystywane m.in. do przetwarzania plików tekstowych, w których wyszukiwane są pewne wzorce oraz dodatkowo zwykle przeprowadzana jest weryfikacja poprawności formatu danych (np. numer telefonu, PESEL, kod pocztowy, adres e-mail etc.). Wykonywanie tego rodzaju operacji trudno sobie wyobrazić bez wykorzystania szerokich możliwości wyrażeń regularnych, które są dostępne w wielu językach programowania, w tym również w powłoce `bash`.

Wyrażenie regularne (ang. *regular expressions*), w skrócie regex lub regexp, jest sekwencją znaków definiującą wzorzec, który najczęściej jest używany w algorytmach poszukiwania ciągów znakowych w przypadku funkcji typu "Wyszukaj" lub "Wyszukaj i zamień" czy walidacji danych wejściowych. Przy wyrażeniach regularnych mamy możliwość wykorzystania wielu edytorów tekstowych, programów do przetwarzania tekstu czy wreszcie bezpośrednią sposobność ich zastosowania przy wielu językach programowania.

Wyrażenie regularne to wzorzec, do którego dopasowywane są kolejne fragmenty przetwarzanego tekstu. Przykładowo, wzorzec:

```
TOC
```

składa się z trzech standardowych (nie specjalnych) znaków i pasuje wyłącznie do ciągu znaków, w którym bezpośrednio po sobie występują trzy litery "T", "O" i "C", czyli "TOC". Ciąg znaków "TOC" nie musi być oddzielnym wyrazem, może być dowolnym fragmentem przetwarzanego tekstu. W wyrażeniu regularnym oprócz standardowych znaków mogą również występować znaki specjalne (metaznaki) podlegające odpowiedniemu rozwinięciu.

Program akceptujący wyrażenia regularne (np. `grep`), po sprawdzeniu składni podanego wyrażenia oraz jego rozwinięciu, przetwarza najczęściej tekst wiersz po wierszu. Program porównuje pierwszy znak odczytanego wiersza z pierwszym znakiem wzorca. Jeżeli są one identyczne, to porównywany jest kolejny znak itd. Jeżeli na którejkolwiek pozycji nie będzie zgodności, wówczas następuje próba dopasowania wzorca, poczynając od drugiego znaku odczytanego wiersza. Porównuje się drugi znak wiersza z pierwszym znakiem wzorca itd. Poniżej przedstawione są kolejne kroki dopasowania wzorca "`TOC`" do ciągu znakowego "POLITECHNIKA BIAŁOSTOCKA".

```
P|OLITECHNIKA BIAŁOSTOCKA
T|OC
```

```
P O|LITECHNIKA BIAŁOSTOCKA
  T|OC
```

Brak dopasowania pierwszych znaków wzorca i tekstu, następuje porównanie kolejnych znaków tekstu z pierwszym znakiem wzorca.

```
POLIT|ECHNIKA BIAŁOSTOCKA
     T|OC
```

```
POLIT E|CHNIKA BIAŁOSTOCKA
     T O|C
```

P O L I T E C H N I K A B I A Ł O S T O C K A
T O C

Występuje dopasowanie pierwszego znaku wzorca z piątym znakiem tekstu. Jednak kolejne znaki wzorca nie pasują do kolejnych znaków tekstu. Dlatego następuje próba dopasowania wzorca, poczynając od kolejnego znaku tekstu.

...

P O L I T E C H N I K A B I A Ł O S **T** O C K A
T O C

P O L I T E C H N I K A B I A Ł O S **T** **O** C K A
T **O** C

P O L I T E C H N I K A B I A Ł O S **T** **O** **C** K A
T **O** **C**

Poczynając od 20 znaku tekstu, występuje pełna zgodność ze wzorcem.

Wyrażeń regularnych nie należy mylić z symbolami uogólniającymi (patrz rozdział 2), które w większości przypadków wykorzystywane są przez powłoki i takie programy, jak `find` czy `cpio`. Należy pamiętać, iż symbole uogólniające rozwijane są przez powłokę przed wykonaniem polecenia, a zdefiniowane wzorce wyrażeń regularnych przetwarzane są przez określone polecenia w trakcie wykonania.

7.2 Metaznaki

W definicji wyrażeń regularnych oprócz symboli mających standardowe znaczenie, takich jak litery alfabetu, cyfry itd., wykorzystywane są symbole specjalne (metaznaki), np. "*", ".", "^", "$". Chcąc pominąć specjalne znaczenie metaznaku i uzyskać we wzorcu jego zapis literalny, należy taki symbol specjalny poprzedzić znakiem "\". Wzorzec odpowiadający sekwencji znakowej "`$A`" należy zdefiniować jako "`\$A`".

Istnieje wiele rozszerzeń wyrażeń regularnych. Treści zawarte w kolejnych punktach będą odnosiły się głównie do podstawowego zbioru metaznaków wyrażeń regularnych wywodzącego się ze standardu POSIX, które są prawidłowo interpretowane przez większość „silników" przetwarzających wyrażenia regularne.

POSIX (ang. Portable Operating System Interface for UNIX) jest zbiorem standardów definiujących wybrane funkcjonalności, które musi spełniać system UNIX. Jeden z tych standardów definiuje dwie wersje wyrażeń regularnych: podstawowe (ang. BRE – Basic Regular Expressions) i rozszerzone (ang. ERE – Extended Regular Expressions).

Specyfikatory pozycji

Istnieją dwa podstawowe specyfikatory pozycji: `"^"` i `"$"`. Znak `"^"` oznacza początek wiersza, `"$"` odnosi się zaś do jego końca. Na przykład wyrażenie `"A$"` będzie zgodne ze wszystkimi wierszami kończącymi się literą "A". Specyfikatory pozycji pełnią specjalną funkcję, jeżeli występują tylko odpowiednio na pierwszej i ostatniej pozycji wzorca. Jeżeli specyfikator znajduje się w innym miejscu niż to, na które wskazuje jego funkcja, traktowany jest jak standardowy znak. Grupa przykładów ilustrujących działanie specyfikatorów pozycji zamieszczona jest w tabeli 7.1.

Tabela 7.1: Ilustracje interpretacji specyfikatorów pozycji

Wzorzec	Dopasowanie
`^ABC`	wiersze rozpoczynające się sekwencją znaków "ABC"
`ABC$`	wiersze kończące się ciągiem znaków "ABC"
`^$`	wiersze "puste"
`A^`	wiersze zawierające w dowolnym miejscu sekwencję znaków "A^"
`$A`	wiersze zawierające w dowolnym miejscu sekwencję znaków "$A"

Wyrażenia uogólniające

Podobnie jak w powłoce, również w wyrażeniach regularnych występują pojedyncze symbole uogólniające (metaznaki) czy całe wyrażenia o takim charakterze. W przypadku wyrażeń regularnych do wyrażeń uogólniających należy zaliczyć:

- wyrażenia nawiasowe (ang. *bracket expression*),
- klasy znaków (ang. *character classes*),
- znak kropki '.' (ang. *dot character*).

Wyrażenia nawiasowe

Wyrażenie nawiasowe (ang. *bracket expression*) składa się ze zbioru znaków i odpowiada dowolnemu znakowi należącemu do tego wyrażenia. Zbiór znaków należący do wyrażenia nawiasowego zamyka się nawiasami kwadratowymi, np. `"[ABC]"`. W tym przypadku odpowiada ono jednemu ze znaków: "A", "B" lub "C".

Wyrażenie regularne `"ABC"` odnosi się do trzech występujących po sobie znaków i jest różne od wyrażenia regularnego `"[ABC]"` odnoszącego się do pojedynczego znaku. Pierwszy wzorzec pasuje tylko do wierszy zawierających w sobie ciąg znaków "ABC". W drugim przypadku dopasowanie następuje, gdy w wierszu na dowolnej pozycji występuje jedna z liter "A", "B" lub "C".

Wyrażenie nawiasowe może zostać zdefiniowane poprzez wymienienie poszczególnych znaków należących do tego wyrażenia, np. `"[ABC]"`, poprzez wskazanie zakresu, np. `"[A-C]"`, lub poprzez jedno i drugie, np. `"[A-BC]"`. We wszystkich powyższych przypadkach wyrażenie nawiasowe złożone jest z trzech znaków "A", "B" lub "C". W definicji wyrażenia nawiasowego, oprócz znaku myślnika "–" umożliwiającego definicję zakresu, znakiem specjalnym jest również znak `"^"` oznaczający zaprzeczenie logiczne zbioru znaków danego wyrażenia. Przykładowo, wyrażenie `"[^ABC]"` obejmuje wszystkie możliwe

znaki oprócz "A", "B" lub "C". Znak "^" swoje specjalne znaczenie ma tylko wówczas, jeśli jest pierwszym znakiem wyrażenia nawiasowego. W pozostałych przypadkach traktowany jest jak zwykły składnik tego wyrażenia.

Klasy znaków

Zbiór znaków zdefiniowany przez wyrażenie nawiasowe np. "[a-z]" zdeterminowany jest poprzez kody liczbowe przypisane do poszczególnych znaków. Wartości liczbowe odpowiadające poszczególnym znakom przypisane są na podstawie przyjętego systemu kodowania. Może on szeregować znaki alfabetu na różne sposoby (np. "abc..zABC..Z" czy "aAbB...zZ"). Aby uniknąć ewentualnych nieporozumień, standard POSIX definiuje klasy znaków. Podstawowe z nich przedstawia tabela 7.2.

Wykorzystanie zdefiniowanych przez POSIX klas znaków ograniczone jest do wyrażeń nawiasowych. Wzorzec pasujący do wierszy złożonych wyłącznie z wielkich liter będzie miał zatem postać: "^[[:upper:]]\+$".

Tabela 7.2: Klasy znaków

Klasa	ASCII	Opis
[:alnum:]	[A-Za-z0-9]	znaki alfanumeryczne
[:alpha:]	[A-Za-z]	litery
[:blank:]	[\t]	spacje i tabulatory
[:digit:]	[0-9]	cyfry
[:lower:]	[a-z]	małe litery
[:upper:]	[A-Z]	wielkie litery

Znak kropki

Najbardziej ogólnym metaznakiem w przypadku wyrażeń regularnych jest znak kropki ".". Standardowo symbolizuje on dowolny znak oprócz znaku

nowego wiersza. Taka interpretacja jest w całym wzorcu z wyjątkiem wyrażenia nawiasowego, w którym kropka interpretowana jest literalnie i symbolizuje samą siebie. Tak więc wzorzec `"^A.C$"` pasuje do wszystkich wierszy złożonych z trzech znaków, z których pierwszy i ostatni to odpowiednio "A" i "C", znak środkowy zaś jest znakiem dowolnym. Inaczej jest w przypadku wzorca `"^[A.C]$"`, który pasuje do wszystkich wierszy złożonych z jednego znaku, którym jest "A", "." lub "C".

Przykłady wykorzystania mechanizmów uogólniających w wyrażeniach regularnych zamieszczone są w tabeli 7.3.

Tabela 7.3: Ilustracje interpretacji mechanizmów uogólniających w wyrażeniach regularnych

Wzorzec	Dopasowanie (wiersze ...)
`^[ABC]`	rozpoczynające się od jednej z liter "A", "B" lub "C"
`[0-9]$`	kończące się cyfrą
`A.C`	zawierające trzyznakowy ciąg, gdzie pierwszym znakiem jest litera "A", ostatnim "C", a środkowym dowolny znak
`^[^0-9]`	nierozpoczynające się cyfrą
`[0-9]]`	zawierające cyfrę, po której bezpośrednio występuje znak "]"
`^...$`	złożone dokładnie z trzech znaków
`^[0-9][0-9]$`	złożone z dwóch cyfr
`^[[:upper:]][[:lower:]]$`	złożone z dwóch liter (pierwszej wielkiej i drugiej małej)

Kwantyfikatory powtórzeń

Kwantyfikatory powtórzeń pozwalają wielokrotnie powielić element wzorca regularnego bezpośrednio poprzedzającego kwantyfikator. Podstawowym

kwantyfikatorem liczby powtórzeń jest gwiazdka "∗". Oznacza ona zero lub więcej powtórzeń elementu bezpośrednio poprzedzającego gwiazdkę. Z uwagi na częste niezrozumienie interpretacji gwiazdki w wyrażeniach regularnych przeanalizowanych zostanie kilka przykładów.

Jako pierwszy rozpatrzmy wzorzec "^A∗". Wstępnie analizując podany wzorzec, można błędnie osądzić, iż pozwoli on wyselekcjonować wszystkie wiersze rozpoczynające się sekwencją liter "A". Metaznak "*" dopuszcza jednak możliwość zerowej liczby wystąpień elementu bezpośrednio go poprzedzającego.Oznacza to, że w tym wypadku ze wzorcem będą zgodne wiersze o dowolnej treści. Chcąc osiągnąć zamierzony efekt (wyselekcjonowanie wierszy rozpoczynających się sekwencją liter "A"), wzorzec należy zdefiniować w sposób następujący: "^AA∗" lub też "^A+". Zgodnie z definicją gwiazdka odnosi się tylko do elementu bezpośrednio ją poprzedzającego. Dlatego w tym wypadku mamy zagwarantowane przynajmniej jedno wystąpienie litery "A". Ten sam efekt możemy osiągnąć, wykorzystując znak "+", wymuszający, aby poprzedzający go symbol wystąpił co najmniej raz.

W kolejnym przykładzie zdefiniowany zostanie wzorzec pozwalający wyselekcjonować wiersze, w których jedynym elementem będą sześcioznakowe tagi postaci "<XXXX>", gdzie X jest dowolną wielką literą, np. <HTML>. Dodatkowo przyjmijmy, że występujący w wierszu tag w celu zapewnienia odpowiedniego wcięcia może być poprzedzony dowolną liczbą spacji. Biorąc pod uwagę fakt, iż tag ma być jedynym elementem wiersza, należy wzorzec „zamknąć” znacznikiem początku i końca: "^$". Ograniczniki tagu (nawiasy ostre) są w przypadku wersji BRE znakami zwykłymi, więc we wzorcu możemy umieścić je w sposób literalny: "^<>$". Z poprzedniego punktu wiadomo, że klasa znaków odnosząca się do wielkich liter oznaczana jest jako "[:upper:]". Ponieważ klasy znaków można wykorzystywać wyłącznie w wyrażeniach nawiasowych, postać wzorca odnosząca się do taga będzie następująca:

```
^<[[:upper]][[:upper]][[:upper]][[:upper]]>$
```

Należy jeszcze uwzględnić spacje wiodące, jakie mogą pojawić się przed tagiem. W tym celu wykorzystana zostanie gwiazdka, która pozwala zdefiniować finalną postać wzorca:

```
^ *<[[:upper]][[:upper]][[:upper]][[:upper]]>$.
```

Połączenie metaznaków kropki i gwiazdki pozwala zdefiniować wzorzec interpretowany jako dowolny ciąg znaków. Przykładowo, wzorzec

```
"^[0-9].*[0-9]$"
```

pozwoli wybrać wiersze, które rozpoczynają i kończą się cyfrą, a pomiędzy początkiem i końcem może występować dowolny ciąg znaków.

Rozwijanie wyrażeń regularnych doprowadziło do definicji nowych kwantyfikatorów liczby powtórzeń. Najbardziej popularne kwantyfikatory liczby powtórzeń zamieszczone są w tabeli 7.4

Tabela 7.4: Kwantyfikatory liczby powtórzeń

Kwantyfikator	Znaczenie (wzorzec powtórzony ...)
`{x,y}`	co najmniej x razy, lecz nie więcej niż y razy
`+`	minimum jeden raz
`?`	co najwyżej jeden raz
`*`	zero lub więcej razy

W przypadku wersji BRE system POSIX wymaga, aby nawiasy klamrowe były poprzedzone ukośnikiem wstecznym `"\{\}"`. Metaznaki `"?"` i `"+"` zdefiniowane zostały dopiero w wersji ERE, ale bardzo często implementowane są w programach standardowo akceptujących składnię BRE. W tym przypadku muszą one również być poprzedzone ukośnikiem wstecznym.

Grupa przykładów odnoszących się do kwantyfikatorów liczby powtórzeń zamieszczona jest w tabeli 7.5.

Tabela 7.5: Ilustracje interpretacji kwantyfikatorów liczby powtórzeń

Wzorzec	Dopasowanie (wiersze ...)
`*`	zawierające znak "*"
`^*`	rozpoczynające się znakiem "*"
`AA*B`	zawierające ciąg znaków "A" (przynajmniej jeden), po którym występuje znak "B"
`A\{3\}B`	zawierające ciąg znaków "AAAB"
`A\+$`	kończące się przynajmniej jednym wystąpieniem znaku "A"
`A\{1,3\}B`	zawierające ciąg znaków "AB", "AAB", "AAAB"

Zadania

1. Podaj polecenie wyszukujące w pliku */etc/passwd* wszystkie linie zawierające ciąg "`nologin`".

2. Podaj polecenie wyszukujące w pliku */etc/passwd* wszystkie linie rozpoczynające się od litery "`r`".

3. Podaj polecenie wyszukujące w pliku */etc/passwd* wszystkie linie kończące się ciągiem "`bash`".

4. Podaj polecenie wyszukujące w pliku */etc/passwd* wszystkie linie niezawierające ciągu "`sbin`".

5. Podaj polecenie wyszukujące w katalogu */etc* i jego podkatalogach wszystkie linie z plików zawierające słowo "`shadow`". Strumień błędów należy usunąć z wyników.

7.3 Zastosowania w powłoce `bash`

W przypadku zastosowania wyrażeń regularnych bezpośrednio w powłoce `bash` (np. w instrukcjach warunkowych) należy warunek otoczyć podwójnym

nawiasem kwadratowym oraz użyć operatora porównania "=~". Przykładowa instrukcja `if` przedstawia się wówczas następująco:

```
if [[ zmienna =~ regexp ]];
```

a w przypadku negacji można użyć jednej z poniższych składni:

```
if [[ ! zmienna =~ regexp ]];
if ! [[ zmienna =~ regexp ]];
```

W celu sprawdzenia, czy w zmiennej `numer` przechowywana jest liczba dodatnia, można użyć następującego warunku:

```
if [[ "$numer" =~ ^[0-9]+$ ]];
```

Natomiast to, czy zmienna `haslo` nie jest pusta ani też nie składa się jedynie z małych i wielkich liter, można sprawdzić następująco:

```
if [[ ! "haslo" =~ ^[a-zA-Z]*$ ]];
```

Warto tu wspomnieć, że zastosowanie w warunku operatora == wyłącza interpretację prawej strony jako wyrażenia regularnego i to pomimo umieszczenia go wewnątrz podwójnego nawiasu kwadratowego. Można to sprawdzić, wykonując np. polecenia ze zmienionym operatorem przedstawione wcześniej w tym rozdziale.

Zadania

1. Napisz skrypt sprawdzający, czy przekazana do niego wartość ma format kodu pocztowego w formacie "XX-XXX", np. 15-351.

   Przykład:
   ```
   $ ./skr_kod.sh 15-765
   Prawidlowy kod pocztowy
   ```

2. Napisz skrypt sprawdzający, czy przekazana do niego wartość to: adres MAC, adres IPv4 czy też IPv6. Oprócz stosownego komunikatu należy zwrócić odpowiednio wartość 1, 2, 3 lub wartość >100 w przypadku wystąpienia innych błędów.

   Przykład:
   ```
   $ ./skr_ip.sh 192.168.10.16
   Adres IPv4.
   $ echo $?
   1
   ```

3. Napisz skrypt, który sprawdzi, czy w haśle przekazanym jako parametr zawarte są co najmniej jedna mała i wielka litera, cyfra oraz znak specjalny (w tym ostatnim przypadku można zawęzić ich zbiór do wybranych symboli). Dodatkowo nie powinny być akceptowane hasła krótsze niż 10 znaków. Wynik powinien być raportowany w poniższym formacie:

   ```
   Male litery: OK
   Wielkie litery: OK
   Cyfry: BRAK
   Znaki specjalne: BRAK
   Dlugosc: OK
   ```

   Ponadto w przypadku w pełni poprawnego hasła powinna być zwracana wartość 0, a błędy raportowane z kodami >100.

4. Napisz skrypt, który z pliku */etc/passwd* wypisze nazwy wszystkich użytkowników niemających ustawionej domyślnej powłoki na `false` lub `nologin` (pełne ścieżki do nich w danym systemie powinny być ustawione wewnątrz skryptu).

   Przykład:
   ```
   $ ./skr_shell.sh
   root
   ```

```
student
tester
```

5. Napisz skrypt, który z pliku */etc/group* wypisze nazwy wszystkich grup, do których należy użytkownik wskazany jako parametr skryptu. W przypadku uruchomienia skryptu bez parametrów należy zwrócić informacje o bieżącym użytkowniku.

   Przykład:
   ```
   $ ./skr_gropus.sh student
   student: student cdrom sudo lpadmin
   ```

6. Napisz skrypt, który będzie dodawał do zmiennej środowiskowej `PATH` kolejne ścieżki podane jako parametr uruchomienia skryptu. Wartości te przed dodaniem powinny być weryfikowane, czy istnieje taka ścieżka w systemie, czy jest podana w postaci bezwzględnej (UWAGA: warto sprawdzić, czy np. "../" nie występuje w całym ciągu, a nie tylko na jego początku) oraz czy nie występuje ona już we wspomnianej zmiennej. W przypadku poprawnie dodanej wartości powinna być wypisana zawartość zmiennej `PATH` oraz zwrócony kod 0. Ewentualne błędy powinny być sygnalizowane za pomocą komunikatów oraz zwracanych kodów wyjścia o wartościach >100.

   Przykład:
   ```
   $ ./skr_RegexpPath.sh /bin
   PATH=/usr/sbin:/usr/sbin:/bin
   $ ./skr_RegexpPath.sh /usr/tmp/linux
   Brak wskazanej sciezki w systemie.
   $ ./skr_RegexpPath.sh ../tmp/linux
   Tylko sciezki bezwzgledne sa akceptowane.
   $ ./skr_RegexpPath.sh /bin
   Sciezka juz występuje w zmiennej PATH.
   ```

Rozdział 8

Funkcje

Funkcje mogą być definiowane w dwóch dostępnych formatach:

```
function moja_funkcja {
  instrukcja_1;
  ...
  instrukcja_2;

}

moja_funkcja () {
  instrukcja_1;
  ...
  instrukcja_2;

}
```

W obydwu przypadkach uzyskamy identyczny efekt. Różnica jest jedynie w składni. W pierwszym z nich użyte zostało słowo kluczowe `function`, w drugim nawiasy okrągłe po nazwie.

Do funkcji możemy przekazać dowolną liczbę parametrów, a dostęp do nich jest identyczny jak w przypadku skryptów, tj. `$1`, `$2` etc. Dostępne są również

zmienne określające liczbę przekazanych parametrów, jak i ich listę. Warto zwrócić uwagę, że nawet w przypadku deklaracji funkcji z użyciem nawiasów okrągłych nigdzie nie określamy w sposób jawny jej parametrów.

Listing 8.1: Przykład definicji funkcji ilustrującej przesłanianie zmiennych związanych z przekazywanymi do niej parametrami

```
#!/bin/bash

function test_param {
  echo "Liczba parametrow funkcji: "$#
  echo "Lista parametrow funkcji: "$@
}

echo "Liczba parametrow skryptu (przed funkcja): "$#
echo "Lista parametrow skryptu (przed funkcja): "$@

test_param $1 $2 $3

echo "Liczba parametrow skryptu (po funkcji): "$#
echo "Lista parametrow skryptu (po funkcji): "$@
```

Na listingu 8.1 przedstawiony jest skrypt, który wypisuje liczbę oraz listę przekazywanych parametrów w różnych momentach działania skryptu. Wartości te są wypisywane przed wykonaniem i po wykonaniu funkcji `test_param` oraz w jej wnętrzu, przy czym warto zauważyć, że w prezentowanym przykładzie przekazywane są do niej jedynie pierwsze 3 parametry. W wyniku uruchomienia powyższego skryptu możemy uzyskać następujący wynik:

```
$ ./skr_function.sh 2 4 6 8 10
Liczba parametrow skryptu (przed funkcja): 5
Lista parametrow skryptu (przed funkcja): 2 4 6 8 10
Liczba parametrow funkcji: 3
Lista parametrow funkcji: 2 4 6
Liczba parametrow skryptu (po funkcji): 5
Lista parametrow skryptu (po funkcji): 2 4 6 8 10
```

Jak można zauważyć, wartości poszczególnych parametrów po zakończeniu funkcji mają taką samą wartość jak przed jej wykonaniem. Oznacza to, że wewnątrz niej operujemy na lokalnych kopiach zmiennych o tych samych nazwach i ewentualne zmiany ich wartości nie będą miały wpływu na dalsze działanie skryptu po wyjściu z tej funkcji.

Listing 8.2: Przykład implementacji funkcji `max`

```
#!/bin/bash

function max {

if [ $1 -ge $2 ];
  then
    echo $1
  else
    echo $2
}

max $1 $2
```

Na listingu 8.2 przedstawiony został skrypt z zaimplementowaną funkcją `max` wypisującą większą z dwóch przekazanych do niej wartości. W wyniku jego uruchomienia możemy uzyskać następujące wyniki:

```
$ ./skr_max.sh 4 6
6
$ ./skr_max.sh 3 -6
3
```

Zdefiniowane funkcje, podobnie jak zmienne, są dostępne jedynie w bieżącej powłoce. Aby je wyeksportować, należy użyć polecenia `export -f`.

Przykład: Dostęp do zdefiniowanej funkcji `max` w procesach potomnych (w tym przypadku nowej powłoki `bash`).

```
$ function max {if [ $1 -gt $2 ]; then echo $1; else echo $2; fi}
```

```
$ max 3 6
6
$ bash
$ max 3 6
Nie znaleziono polecenia max.
$ exit
$ export -f max
$ bash
$ max 3 6
6
```

Listę dostępnych funkcji w danej powłoce można uzyskać za pomocą polecenia `declare -F`, np.:

```
$ declare -F
declare -f __all_modules
declare -f __cards
declare -f __expand_tilde_by_ref
declare -f __get_cword_at_cursor_by_ref
...
```

Natomiast pełne definicje funkcji można wyświetlić za pomocą polecenia `declare -f`, np.:

```
$ declare -f
__all_modules ()
{
while read name; do
name=${name%% *};
printf "%s\n" "$name";
done < <(pulseaudio -dump-modules 2> /dev/null)
}
...
```

Nieużywane funkcje można usunąć, podobnie jak zwykłe zmienne, za pomocą polecenia `unset`, np.

```
$ unset max
```

Zadania

1. Napisz skrypt, który będzie odgrywał rolę prostego kalkulatora. Do każdej wykonywanej operacji arytmetycznej zdefiniuj odpowiednią funkcję (np. `dodaj`, `odejmij` etc.). Do skryptu należy przekazać 3 parametry w postaci:
   `wartosc1 operator wartosc2`.

   Przykład:

   ```
   $ ./kalkulator.sh 3 + 5
   8
   ```

2. Napisz skrypt, który zweryfikuje, czy przekazana w postaci parametru wartość jest prawidłowym numerem PESEL. Zaimplementuj w tym celu funkcje, m.in. do sprawdzenia długości parametru, tego, czy składa się jedynie z cyfr oraz czy suma kontrolna jest prawidłowa. W przypadku błędów powinny zostać zwrócone odpowiedni komunikat oraz kod.

   Przykład:

   ```
   $ ./pesel.sh 12345678901
   BLAD: Nieprawidlowa suma kontrolna.
   ```

3. Napisz skrypt, który zweryfikuje, czy przekazana w postaci parametru wartość jest prawidłowym adresem IPv4. Zaimplementuj w tym celu funkcje, m.in. do dzielenia parametru na 4 oktety oraz do sprawdzenia, czy w każdym z nich jest wartość z zakresu od 0 do 255. W przypadku błędów powinny zostać zwrócone odpowiedni komunikat oraz kod.

Przykład:

```
$ ./ip_check.sh 10.11.12.17
Prawidlowy adres IP.
```

4. Napisz skrypt imitujący polecenie `id`. Zaimplementuj w tym celu odpowiednie funkcje, których jednym z parametrów będzie np. identyfikator lub nazwa użytkownika, dla którego wykonywane jest polecenie.

Rozdział 9

Śledzenie wykonywania skryptów

Śledzenie wykonywania (debugowanie) skryptu ma istotne znaczenie w analizie tego procesu oraz ułatwia zidentyfikowanie przyczyny ewentualnych błędów. Umożliwia m.in. sprawdzenie, czy i jakie wartości są przypisane do poszczególnych zmiennych, również w instrukcjach warunkowych, na każdym etapie jego działania.

Dostępne są dwa scenariusze śledzenia wykonywania skryptu, tj. debugowanie całego kodu lub też jego fragmentu. Ponadto istnieje też możliwość weryfikowania składni bez konieczności uruchomienia skryptu oraz zatrzymywania jego działania w przypadku próby wykonania operacji na niezainicjalizowanej zmiennej. Lista wybranych opcji wykorzystywanych do śledzenia wykonywania skryptu została zawarta w tabeli 9.1, a ich praktyczne wykorzystanie zostało przedstawione w dalszej części rozdziału.

W celu śledzenia wykonywania całego skryptu należy go uruchomić z wykorzystaniem opcji `-x`, np.:

```
$ bash -x ./skrypt.sh
```

Tabela 9.1: Wybrane operatory wykorzystywane do śledzenia wykonywania skryptów

Operator	Opis
`-x`	wypisanie wykonywanych poleceń (z rozwiązanymi zmiennymi)
`-v`	wypisanie kodu śledzonej linii
`-n`	weryfikacja składni kodu bez jego uruchamiania
`-u`	zatrzymanie działania skryptu w przypadku wykrycia operacji realizowanej na niezainicjalizowanej zmiennej

Listing 9.1: Przykładowy skrypt prezentujący wykorzystanie poleceń `set` do oznaczania fragmentu poddanego śledzeniu

```
#!/bin/bash

par1=$1

set -x

par2=$2

if [ $par1 -gt $par2 ]; then
  max=$par1
  min=$par2
  echo "Zakres: $par2 --> $par1"
else
  min=$par1
  max=$par2
  echo "Zakres: $par1 --> $par2"
fi

set +x
```

Można również włączyć śledzenie wybranego jego fragmentu, otaczając go parą poleceń `set -x` oraz `set +x`, co zostało zaprezentowane na listingu 9.1. W wyniku jego uruchomienia można uzyskać następujący rezultat:

```
$ ./debug.sh 3 7+ par2=7
+ '[' 3 -gt 7 ']'
+ min=3
+ max=7
+ echo 'Zakres: 3 -> 7'
Zakres: 3 -> 7
+ set +x
```

Powyższy wynik, w przypadku użycia dodatkowej opcji `-v` (czyli `set -xv`), może być rozszerzony o dodatkowe wypisywanie fragmentów śledzonego kodu. Wówczas dla tego samego skryptu na standardowym wyjściu otrzymamy:

```
$ ./debug.sh 3 7+ par2=7
par2=$2
+ par2=7
if [ $par1 -gt $par2 ]; then
max=$par1
min=$par2
echo "Zakres: $par2 -> $par1"
else
min=$par1
max=$par2
echo "Zakres: $par1 -> $par2"
fi
+ '[' 3 -gt 7 ']'
+ min=3
+ max=7
+ echo 'Zakres: 3 -> 7'
Zakres: 3 -> 7
set +xv
+ set +xv
```

W skryptach `bash` może się zdarzyć sytuacja, w której będą przeprowadzane operacje na niezainicjalizowanych zmiennych. W konsekwencji może to prowadzić do powstania błędu składniowego lub do nieprawidłowego działania, co zostało ukazane na poniższych przykładach.

Przykład: Wykonanie operacji dodawania dwóch zmiennych, z których jedna nie została zainicjalizowana.

```
$ x=5; val=‘expr $x + $y‘; echo $val
expr: błąd syntaktyczny: brakujący argument „+”
```

Listing 9.2: Przykładowy skrypt prezentujący możliwość operowania na niezainicjalizowanej zmiennej

```
 #!/bin/bash

login="user"
haslo=$1
login_password="$login/$haslo"

if [ "$login_password" == "user/student" ]
then
  echo "Witaj: $login"
else
  echo "BLAD: Nieprawidlowe dane"
  exit 1
fi
```

Jednakże w przypadku operowania na ciągach znaków błąd nie zawsze będzie generowany. Na listingu 9.2 zaprezentowano przykładowy skrypt, w którym wartość zmiennej `login_password` jest konkatenacją dwóch innych zmiennych. I nawet w przypadku, gdy jedna z nich pozostanie nieustawiona, skrypt będzie się uruchamiał bez żadnych błędów. Taka sytuacja miałaby miejsce, gdyby do powyższego skryptu nie przekazano żadnych parametrów:

```
$ ./debug_unset.sh
login_password: user/
BLAD: Nieprawidlowe dane
```

Listing 9.3: Skrypt usuwający z katalogu domowego bieżącego użytkownika plik lub katalog (wraz z zawartością) podany jako parametr podczas uruchamiania

```
#!/bin/bash

rm -fr $HOME/$1
```

Niestety w niektórych przypadkach niezainicjalizowane zmienne mogą prowadzić do nieplanowanego działania, a nawet do usunięcia danych. Na listingu 9.3 został przedstawiony bardzo prosty skrypt służący do usuwania rekurencyjnego, przekazanego jako parametr zasobu z katalogu domowego bieżącego użytkownika. W przypadku uruchomienia bez parametrów usunięty zostałby rekurencyjnie cały katalog domowy, co prawdopodobnie nie było celem autora skryptu.

Tego rodzaju przypadki, można zdiagnozować, uruchamiając skrypt z parametrem `-u`, który spowoduje zatrzymanie działania skryptu, gdy wykryta zostanie operacja na niezainicjalizowanej zmiennej. Uruchomiając skrypt z listingu 9.3 bez żadnych parametrów, polecenie kasowania `rm -fr $HOME/$1` nie zostałoby wykonane, a interpreter zgłosiłby następujący błąd:

```
$ bash -u debug_rm.sh
debug_rm.sh: linia 4: $1: nieustawiona zmienna
```

Warto również pamiętać, że poprawność składniowa skryptu może być sprawdzona bez konieczności jego uruchamiania. Służy do tego opcja `-n`.

Przykład: Weryfikacja składniowa skryptu z listingu 9.2, w którym zostało usunięte słowo kluczowe `then`.

```
$ bash -n debug_syntax_error.sh
debug_syntax_error.sh: linia 10: błąd składni przy nieoczekiwanym
znaczniku 'else'
debug_syntax_error.sh: linia 10: 'else'
```

Bibliografia

[1] Standard for Information Technology – Portable Operating System Interface (POSIX®). *IEEE Std 1003.1-2001*, strony 1–22, 2000.

[2] R. Brash, G. Naik. *Bash Cookbook: Leverage Bash Scripting to Automate Daily Tasks and Improve Productivity*. Packt Publishing, Limited, Birmingham, 2018.

[3] M. Ebrahim, A. Mallett. *Mastering Linux Shell Scripting: A Practical Guide to Linux Command-Line, Bash Scripting, and Shell Programming, 2nd Edition*. Packt Publishing, Limited, Birmingham, 2018.

[4] Free Software Foundation. *GNU Bash manual*. [Online: https://www.gnu.org/savannah-checkouts/gnu/bash/manual; stan na dzień: 15 lipca 2023].

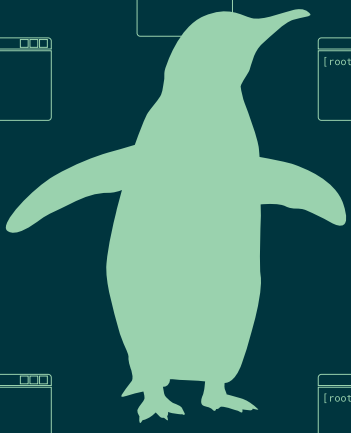
[5] C. F. A. Johnson, J. Varma. *Pro Bash programming: scripting the GNU/Linux shell*. friendsof ED, Apress, New York, wydanie drugie, 2015.

[6] M. Lach. *Bash : praktyczne skrypty*. Helion, Gliwice, 2015.

[7] S. Powers, J. Peek, T. O'Reilly, M. Loukides. *UNIX Power Tools, 3rd Edition*. O'Reilly, USA, 2002.

[8] C. Schroder. *Linux: receptury : najważniejsze umiejętności użytkownika i administratora*. Helion, Gliwice, wydanie 2, 2022.

[9] Wikipedia contributors. Wyrażenie regularne. *Wikipedia, The Free Encyclopedia*, 2004. [Online: https://pl.wikipedia.org/wiki/Wyrażenie_regularne; stan na dzień: 15 lipca 2023].



Politechnika
Białostocka