6. Skrypty powłoki

6.1. Interpreter poleceń

Interpreter poleceń, nazywany inaczej także *powłoką systemową* (ang. *system shell*), jest pośrednikiem pomiędzy użytkownikiem a funkcjami systemu operacyjnego. Powłoka systemowa pobiera polecenia użytkownika wraz z argumentami i po sprawdzeniu ich poprawności próbuje je wykonać używając do tego celu różnych usług świadczonych przez jądro systemu operacyjnego. Współcześnie dostępnych jest wiele alternatywnych powłok, z których najpopularniejsze wydają się następujące:

- ***bash*** (angielski akronim wzięty ze słów *Bourne Again Shell*, co wbrew pierwszemu skojarzeniu nie ma żadnego związku z postacią głównego protagonisty cyklu powieści sensacyjnych Roberta Ludluma, a stanowi nawiązanie do historycznej już powłoki napisanej w 1979 roku przez inżyniera Bell Laboratories, Stephena Bourne'a, z przeznaczeniem do wykorzystania w systemie Unix wersji siódmej); w chwili obecnej jest to najpopularniejsza powłoka w systemach linuksowych, instalowana domyślnie w przeważającej większości z nich, a jej język skryptowy stał się *de facto* standardem publikacyjnym
- ***zsh*** (ang. *Z Shell*, od inicjałów i loginu profesora Universytety Yale, Zhong Shao, który to ciąg znaków wydał się autorowi pierwszej wersji powłoki, Paulowi Falstadowi, również pracownikowi tego uniwersytetu, bardzo dobrą

nazwą dla jego dzieła); *zsh* znacząco rozszerza wiele funkcjonalności powłoki *bash* i czyni pracę szybszą i wygodniejszą; jako ciekawostkę dodajmy, że to właśnie *zsh* jest wzorcem, na którym implementowano powłokę systemu macOS oraz że jest domyślną powłoką dystrybucji *Kali Linux*

– ***csh*** (ang. *C Shell*), stabilna i szacowna powłoka, wywodząca się z Uniwersytetu Berkeley i napisana w 1978 roku przez Billa Joya z przeznaczeniem dla systemu BSD Unix drugiej wersji; od powłoki *bash* odróżnia ją szczególnie fakt upodobnienia składni instrukcji do tej znanej z języka C (stąd nazwa)
– ***tcsh*** (ang. *T-csh*, gdzie ***T*** jest pierwszą literą słowa *Tenex* – nazwy systemu operacyjnego napisanego w 1969 roku dla komputera DEC PDP-10, który to system był inspiracją dla autora powłoki, Kena Greera, pracownika Uniwersytetu Carnegie Mellon); pierwsza edycja powłoki *tcsh* została wydana w roku 1981 z przeznaczeniem dla systemu BSD Unix wersji trzeciej; jest znaczącym udoskonaleniem powłoki *csh* i jest z nią wstecznie kompatybilna

W dalszej części naszych rozważań będziemy posługiwać się wyłącznie konwencjami pochodzącymi z powłoki *bash*.

6.2. Zmienne środowiskowe

Zmienne środowiskowe (ang. *environment variables*)to wygodny i uniwersalny sposób konfigurowania i parametryzowania powłok systemowych oraz uruchamianych przez nie programów. Wszystkie znane w danym momencie zmienne środowiskowe tworzą tak zwane *środowisko* (ang. *environment*) wykonania procesu. środowisko to jest kopiowane do wszystkich nowych procesów, a więc modyfikacje zmiennych wykonane w powłoce są widoczne we wszystkich programach uruchomionych przy użyciu tej powłoki po wykonaniu modyfikacji. Każdy użytkownik może definiować dowolną liczbę własnych zmiennych oraz przypisywać im dowolne wartości, chociaż należy pamiętać, że wartością zmiennych jest zawsze ciąg znaków (*string*), nawet jeśli reprezentuje liczbę.

Aby zdefiniować zmienną środowiskową należy zastosować operator przypisania (znak "=") w następujący sposób:

```
ZMIENNA=wartość
```

W tym wypadku ciąg znaków **`ZMIENNA`** to nazwa zmiennej, a ciąg znaków składających się na łańcuch **`wartość`** to przypisywana jej wartość. Należy zwrócić uwagę, że pomiędzy nazwą zmiennej, operatorem przypisania i wartością **nie mogą wystąpić żadne białe znaki**. Jeżeli wartość przypisywana do zmiennej ma zawierać białe znaki, to należy je zacytować w sposób identyczny, jak w przypadku nazw plików.

Takie właśnie zachowanie powłoki spowodowane jest bardzo specyficznym podejściem interpretera do analizy kodu źródłowego, który nie wykonuje żadnej z analiz tradycyjnie przypisywanej interpreterom języków takich jak Perl czy Python, w tym nie przeprowadza analizy leksykalnej, pozwalającej wyodrębnić leksemy z tekstu. Dla powłoki każda analizowana przez nią linia jest **poleceniem do wykonania**. Wstawienie do powyższego podstawienia spacji, które zdaniem nieświadomego programisty pozwolą mu upiększyć tekst, na przykład w poniższy sposób

```
ZMIENNA = wartość
```

spowoduje, że powłoka będzie próbować uruchomić program o nazwie **ZMIENNA**, jednocześnie przekazując mu dwa argumenty, **=** oraz **wartość**, co oczywiście skończy się źle.

Odwołanie się do wartości zmiennej jest możliwe dzięki operatorowi **$**, który w tym kontekście staje się czymś na kształt *operatora wyłuskania*. Poprzedzenie nazwy zmiennej znakiem **$** sprawi, że powłoka zastąpi nazwę zmiennej jej bieżącą tekstową wartością. Oznacza to także, że użycie nieistniejącej zmiennej w połączeniu z **$** nie wywołuje błędu, a nazwę taką powłoka zamieni na pusty łańcuch.

Wyprowadzenie wartości dowolnej zmiennej na strumień *stdout* jest możliwe poprzez wykorzystanie polecenia **echo**, co ilustruje poniższy przykład (błąd w trzecim z poleceń **echo** jest celowy i służy demonstracji zachowania powłoki w momencie odwołania do nieznanej jej zmiennej środowiskowej):

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

$
```

Polecenie powłoki o nazwie **set** pozwala wyświetlić wartości wszystkich znanych powłoce zmiennych środowiskowych, a samo zestawienie posortowane jest według kolejności alfabetycznej nazw zmienneych (w poniższym przykładzie uwidoczniono tylko 10 pierwszych wierszy wyemitowanych przez polecenie – całe wyjście może zajmować nawet kilkaset wierszy, co daje obraz tego, jak intensywnie

powłoka używa tego mechanizmu).

```
$ set
'!'=0
'#'=0
'$'=16423
'*'=(  )
-=0569BJXZims
0=zsh
'?'=0
@=(  )
ANT_HOME=/usr/share/ant
ARGC=0
CDPATH=''
```

Polecenie postaci

unset *zmienna*

usuwa zmienną środowiskową o wskazanej nazwie, na przykład:

```
$ VAR=123
$ echo $VAR
123
$ unset VAR
$ echo $VAR

$
```

Jak wspomniano, środowisko wykonania procesu jest kopiowane do jego procesów potomnych, jednak nie wszystkie zmienne powłoki muszą podlegać temu procesowi. Te zmienne, które są przekazywane do środowiska potomków, nazywa się zmiennymi *eksportowanymi* (ang. *export variables*), a zmienne, które takiemu przekazaniu nie podlegają, nazywa się zmiennymi *lokalnymi* (ang. *local variables*). Z reguły nowo tworzone zmienne są zmiennymi lokalnymi i niezbędne jest jawne

wskazanie, że mają być zmiennymi eksportowanymi. Proces taki realizowany jest poleceniem:

```
export lista_zmiennych
```

Oto przykład utworzenia zmiennej i jej wyeksportowania:

```
$ COUNTER=1
$ export COUNTER
```

Powyższe dwa zlecenia można także zrealizować w jednym kroku:

```
$ export COUNTER=1
```

Domyślnie powłoka definiuje pokaźną liczbę zmiennych środowiskowych, które można wykorzystywać do rozpoznania bieżącego stanu systemu operacyjnego oraz otoczenia, w jakim wykonuje się skrypt.

Oto niektóre z takich zmiennych.

HOME
kompletna ścieżka prowadząca do katalogu domowego użytkownika, na przykład:

```
$ echo $HOME
/home/slawek
```

USER
nazwa użytkownika, który odwołuje się do wartości zmiennej, na przykład:

```
$ echo $USER
slawek
```

PATH
lista katalogów, które powłoka przeszukuje w celu odnalezienia programu, którego uruchomienia żąda użytkownik; nazwy katalogów są rozdzielane znakiem dwukropka (`:`), na przykład:

```
$ echo $PATH
/usr/local/sbin:/usr/sbin:/usr/bin:/sbin:/bin
```

PS1
postać *prompta*, jaki powłoka wyświetla użytkownikowi, zapisana przy użyciu specyficznej składni, na przykład:

```
$ echo $PS1
\u@\h:\w\$
```

SHELL
pełna ścieżka prowadząca do domyślnej powłoki użytkownika;
uwaga: powłoka używana w danym momencie może być inna niż domyślna, na przykład:

```
$ echo $SHELL
/bin/bash
```

6.3. Skrypty i ich argumenty

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

Skrypty mogą być *parametryzowane* argumentami ich wywołania – oznacza to, że podczas uruchamiania skryptu można przekazać do niego dowolną liczbę dowolnych danych. Do argumentów wywołania można się odwoływać w skryptach za pomocą tak zwanych *parametrów pozycyjnych* (ang. *positional parameters*), oznaczanych cyframi dziesiętnymi od **1** do **9**. Każdy parametr pozycyjny przechowuje

tekst przekazany za pośrednictwem odpowiadającego mu argumentu wywołania – pierwszy argument dostępny jest w parametrze **$1**, drugi w parametrze **$2**, itd. Oto przykład trywialnego skryptu, który wyświetla wartości pierwszych trzech argumentów jego wywołania (użyte w nim polecenie **echo**, służące do wyprowadzania danych na *stdout* zostanie omówione niebawem):

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

W celu uruchomienia skryptu należy powyższy tekst umieścić w pliku o dowolnej nazwie (zaleca się jednak, aby pliki skryptów miały rozszerzenie **.sh**). Przeanalizujmy znaczenie przedstawionego kodu:

- pierwsza linia zawiera element składniowy nazywany zwyczajowo *hash-bang* (**#!**) (chociaż spotyka się również określenie *shebang*), służący do wskazania, który konkretnie interpreter poleceń ma zostać wykorzystany do wykonania zawartości pliku – w tym przypadku jest to powłoka *bash*
- druga linia ilustruje sposób zapisu komentarza: jest nim tekst rozpoczynający się od znaku **#** aż do końca zawierającego go linii
- kolejne trzy linie wyświetlają wartości pierwszych trzech argumentów wywołania skryptu, wykorzystując do tego polecenie **echo**

Aby uruchomić taki skrypt wprost z linii poleceń, należy zawierającemu go plikowi nadać prawo wykonywania (**x**).

Poniżej przedstawiono przykładowe wywołanie przedstawionego powyżej skryptu oraz wynik jego działania (przyjęto, że plik ze skryptem nazywa się **skrypt1.sh**):

```
./skrypt1.sh abc xyz 12345
argument nr 1: abc
argument nr 2: xyz
argument nr 3: 12345
```

6.4. Wyprowadzanie danych na *stdout*

Podstawowym środkiem umożliwiającym skryptowi powłoki wyprowadzanie danych na strumienie wyjściowe jest polecenie **echo** o następującej składni:

```
echo opcja... łańcuch...
```

Polecenie **echo** spowoduje przesłanie do *stdout* wszystkich *łańcuchów* podanych w argumentach.

Możliwe do użycia opcje to:

-n nie przechodź do nowego wiersza po wyprowadzeniu tekstu (domyślnie każde uruchomienie polecenie **echo** wyprowadza do *stdout* osobny wiersz tekstu

-e włącz rozpoznawanie i honorowanie sekwencji sterujących, rozpoczynających się znakiem ****

-E wyłącz rozpoznawanie i honorowanie sekwencji sterujących (zachowanie domyślne)

Jeśli **echo** pracuje w trybie honorowania sekwencji sterujących, rozpoznawane są następujące z nich:

**** znak ****

\a znak kodu ASCII o nazwie *BEL* (alarm), prowokujący terminal do wydania krótkiego dźwięku ostrzegawczego

\b znak kodu ASCII o nazwie *BS*, powodujący cofnięcie kursora o jeden znak

\c przerwij wyprowadzanie na *stdout* bieżącego argumentu i przejdź do następnego

\e znak kodu ASCII o nazwie *ESC*, rozpoczynający sekwencje sterowania terminalem

\f znak kodu ASCII o nazwie *FF*, wymuszający na drukarce przejście do nowej strony, na terminalu bezużyteczny, wyprowadza jedynie jedną pustą linię

\n znak kodu ASCII o nazwie *LF*, wymuszający przejście do nowego wiersza

\r znak kodu ASCII o nazwie *CR*, wymuszający powrót kursora do początku bieżącego wiersza

\t znak kodu ASCII o nazwie *HT*, wymuszający przejście kursora do kolejnej pozycji tabulatora

\0NNN znak kodu ASCII o kodzie wyrażonym ósemkowo jako *NNN* (od 1 do 3 cyfr)

\xHH znak kodu ASCII o kodzie wyrażonym szesnastkowo jako *HH* (od 1 do 2 cyfr)

6.5. Wprowadzanie danych z *stdin*

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

```
read [ -r ] zmienna...
```

Opcja **-r** – jeśli zostanie użyta, informuje polecenie **read**, że znaki odwróconego ukośnika podawane przez użytkownika jako wartości zmiennych mają być traktowane dosłownie, a nie jako znaki rozpoczynające cytowanie.

Argumentami polecenia **read** są nazwy zmiennych środowiskowych, które przyjmą wartości odczytane ze standardowego wejścia (czytanie wejścia trwa aż do napotkania znaku nowej linii). Jeśli podano więcej niż jedną zmienną, to są one inicjowane w ten sposób, że pierwsze słowo (ciąg znaków ograniczony białymi znakami) trafia do pierwszej zmiennej, drugie do drugiej itd. Działanie **read** można przetestować wykonując następującą sekwencję:

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

6.6. Śledzenie wykonania skryptu

Skrypty mogą być także wykonywanie w trybie debugowania, na przykład w celu testowania poprawności działania warunków, pętli i tym podobnych. Aby zrealizować wykonanie skryptu z włączonym mechanizmem śledzenia należy zastosować przełącznik **-x** wywołania interpretatora poleceń. Można to zrealizować na dwa sposoby:

– można dopisać tenże przełącznik w pierwszej linii skryptu, na przykład w taki sposób:

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

– można uruchomić skrypt wywołując go poprzez jawne wskazanie interpretatora z przełącznikiem i nazwą skryptu jako argumentem; przyjmując, że skrypt umieszczony jest w pliku o nazwie *skrypt.sh* uruchomienie miałoby następującą postać:

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

Załóżmy, że w pliku o nazwie *dword.sh* posiadającym ustawiony atrybut **x** umieszczony jest następujący kod skryptu:

```
1 #! /bin/bash
2
3 echo "Podaj słowo:"
4 read VAR
5 VAR=$VAR$VAR
6 echo $VAR
```

Zadaniem skryptu jest zdublowanie ciągu znaków wprowadzonych na strumień *stdin* przez użytkownika. Wykonanie skryptu może w takim przypadku przebiegać następująco:

```
$ ./dword.sh
Podaj słowo:
WORD
WORDWORD
$
```

Wykonanie tego samego skryptu w trybie śledzenia wygląda jak poniżej:

```
$ /bin/bash -x dword.sh
+ echo 'Podaj słowo:'
Podaj słowo:
+ read VAR
WORD
+ VAR=WORDWORD
+ echo WORDWORD
WORDWORD
$
```

Jak widać, w tym przypadku wykonanie każdej linii kodu źródłowego jest poprzedzone wyprowadzeniem na *stdout* jej treści poprzedzonej znakiem **+**.

6.7. Zmienne predefiniowane

Wykonujący się skrypt ma dostęp do licznych zmiennych predefiniowanych, które pozwalają efektywnie rozpoznawać kontekst wykonania kodu. Najczęściej używane zebrano w poniższej tabeli.

Nazwa	*Zawartość*
$0	Nazwa pod jaką uruchomiono skrypt
$*n*	Wartość argumentu *n* (*1..9*)
$@	Wszystkie argumenty skryptu złożone w wektor
$*	Wszystkie argumenty skryptu złożone w łańcuch znaków
$#	Liczba argumentów skryptu
$?	Kod powrotu z ostatnio wykonanego polecenia
$$	PID aktualnie wykonywanej powłoki (tym samym PID skryptu)

6.8. Łańcuchy znaków

Skrypty powłoki Bash rozpoznają trzy rodzaje łańcuchów znaków:

– łańcuchy ujęte w cudzysłowy (**"**), na przykład **"tekst"**
łańcuchy tego rodzaju są przeglądane przez powłokę i jeśli w ich treści znajdują się odwołania do zmiennych, zostaną one zastąpione wartościami tych zmiennych, na przykład:

```
$ VAR=123
$ echo "VAR=$VAR"
VAR=123
```

– łańcuchy ujęte w apostrofy (**'**), na przykład **'tekst'**
łańcuchy tego rodzaju są traktowane dosłownie i nie są poddawane żadnej dodatkowej obróbce, na przykład:

```
$ VAR=123
$ echo 'VAR=$VAR'
VAR=$VAR
```

– łańcuchy ujęte w akcenty (odwrócone apostrofy, ang. *backtick*), na przykład **`tekst`**
łańcuchy tego rodzaju są traktowane jako polecenia powłoki, a ich faktyczną wartością jest całe wyjście, jakie użyte polecenie wyprowadziło na swoje *stdout*, na przykład:

```
$ VAR=‘date -I‘
$ echo $DATA
2022-04-01
```

6.9. Instrukcja warunkowa

Składnia instrukcji warunkowej w powłoce Bash jest następująca:

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

Zauważ, że:
- gałąź rozpoczynająca się od **else** jest opcjonalna
- wcięcia nie są konieczne, ale zaleca się ich stosowanie dla poprawienia czytelności kodu
- możliwe jest zastąpienie przejścia do nowego wiersza znakiem średnika, co pozwala na zwięzłe zapisywanie prostszych partii kodu (przykład takiego zapisu przedstawimy niebawem)

Warunek może być dowolnym poleceniem i w takim przypadku uznaje się, że jest **spełniony** (prawdziwy), kiedy polecenie to zwróciło kod powrotu **zero**. Zwrócenie niezerowego kodu powrotu powoduje, że warunek uznaje się za niespełniony (fałszywy). Ponieważ takie zachowanie może być sprzeczne z intuicjami programisty nienawykłego do programowania skryptów powłoki, wskazana jest pewna ostrożność.

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

```
1 if [ $1 = 1 ]
2 then
3     echo "eq"
4 else
```

```
5       echo "ne"
6   fi
```

W tym miejscu warto zauważyć, że znak **[** nie reprezentuje tu leksemu (co uznalibyśmy za normalne w przypadku klasycznego języka programowania), a jest **poleceniem** (*sic!*), uruchamianym z pliku **/usr/bin/[**. Polecenie to analizuje swoje argumenty aż do momentu napotkania wśród nich znaku **]**, następnie próbuje złożyć z nich wyrażenie logiczne i jeśli jego obliczenie jest możliwe, kończy pracę z kodem powrotu odpowiadającym prawdziwości warunku. Dlatego też po znaku **[** i przed znakiem **]** konieczne jest umieszczenie co najmniej jednego znaku spacji – niespełnienie tego warunku uniemożliwi powłoce prawidłowe wyodrębnienie warunku spośród otaczającego go tekstu.

Taki sam efekt można uzyskać zapisując kod w zwięźlejszy sposób, na przykład tak:

```
1   if [ $1 = 1 ]; then echo "eq"; else echo "ne"; fi
```

Pierwszy ze sposobów jest zalecany w treści skryptów, drugi upraszcza wydawanie poleceń z konsoli.

Najczęściej polecenia **test** używa się do wykonania następujących sprawdzeń:

Warunek	*Opis*
str1 = str2	Weryfikacja równości dwóch łańcuchów znaków
str1 != str2	Weryfikacja nierówności dwóch łańcuchów znaków
-z str	Weryfikacja czy łańcuch znaków ma zerową długość
-n str	Weryfikacja czy łańcuch znaków ma niezerową długość
num1 -eq num2	Weryfikacja równości dwóch liczb
num1 -ne num2	Weryfikacja nierówności dwóch liczb
num1 -gt num2	Weryfikacja, czy liczba **num1** jest większa od **num2**
num1 -ge num2	Weryfikacja, czy liczba **num1** jest większa równa **num2**
num1 -lt num2	Weryfikacja, czy liczba **num1** jest mniejsza od **num2**
num1 -le num2	Weryfikacja, czy liczba **num1** jest mniejsza równa **num2**
-e file	Weryfikacja, czy plik **file** istnieje
-f file	Weryfikacja, czy plik **file** jest plikiem zwykłym
-d file	Weryfikacja, czy plik **file** jest katalogiem
-r file	Weryfikacja, czy użytkownik ma prawo odczytu pliku **file**
-w file	Weryfikacja, czy użytkownik ma prawo zapisu pliku **file**
-x file	Weryfikacja, czy użytkownik ma prawo wykonania pliku **file**
cond1 -a cond2	Iloczyn logiczny (*and*) warunków **cond1** i **cond2**
cond1 -o cond2	Suma logiczna (*or*) warunków **cond1** i **cond2**
! cond	Negacja warunku **cond**

Pełny zestaw możliwych do wykonania sprawdzeń znajdziesz w manualu do polecenia **test**.

Zauważ, że rozważne użycie znaków cudzysłowu ma fundamentalne znaczenie przy kodowaniu warunków z użyciem polecenia **test**. Zilustrujemy to następującym przykładem:

Poniższa sekwencja (mimo pozornej poprawności wynikającej z nieodpowiedzialnego poddania się nawykom nabytym podczas programowania w innych językach) może wywołać błąd, gdy zmienna **VAR** nie ma nadanej wartości:

```
1 if [ -n $VAR ]
2 then
3     echo podano parametr
4 fi
```

W takim przypadku interpreter będzie próbował wykonać kod, który po zastąpieniu zmiennych ich wartościami będzie prezentował się następująco

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

i spowoduje błąd wykonania już w pierwszej linii.
Użycie cudzysłowów uczyni powyższy kod w pełni poprawnym:

```
if [ -n "$VAR" ]
then
    echo podano parametr
fi
```

6.10. Pętle

Skrypty powłoki mogą także zawierać pętle – podstawowe z nich to warianty pętli **for** oraz pętla **while**.

6.10.1. Pętla for iterująca po liście

Pętla taka wykonywana jest z góry określoną liczbę razy, a jej zadaniem jest przejrzenie zawartości pewnej listy. Jej składnia ogólna prezentuje się następująco:

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

albo (w wersji zwięzłej):

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

Wykonanie pętli powoduje przypisywanie zmiennej **zmienna** kolejnych wartości wymienionych na liście **lista**, liczba iteracji jest zatem zależna od długości podanej listy.

Samą listę można skonstruować na kilka sposobów.

– podać ją literalnie jak zbiór rozdzielonych białymi znakami słów, na przykład tak:

```
#! /bin/bash

```

```
3 for zm in ala ma kota
4 do
5   echo $zm
6 done
```

Powyższy skrypt produkuje następujące wyjście:

```
ala
ma
kota
```

– podać ją jako wieloznaczną nazwę pliku – w takim przypadku powłoka skonstruuje listę, wstawiając do niej wszystkie pasujące do wzorca nazwy plików, na przykład tak;

```
1 #! /bin/bash
2
3 for zm in /s*
4 do
5   echo $zm
6 done
```

Powyższy skrypt produkuje następujące wyjście:

```
/sbin
/srv
/sys
```

– użyć w charakterze listy wyjścia z dowolnego programu uruchomionego przez powłokę – w takim przypadku każde słowo odnalezione w strumieniu zostanie użyte jako kolejna wartość zmiennej, na przykład:

```
1 #! /bin/bash
2
3 for zm in `wc /etc/passwd`
4 do
5   echo $zm
6 done
```

Przykładowe wyjście z wykonania powyższego skryptu może wyglądać następująco:

```
65
270
4580
/etc/passwd
```

6.10.2. Pętla `for` ze zmienną licznikową

W dawniejszych implementacjach powłoki Bash zakładano, że jeśli użytkownik zażyczy sobie iterowania po kolejnych wartościach liczb z podanego przedziału, to posłuży się w tym celu poleceniem **`seq`**, produkującym na swoim wyjściu sekwencje wartości liczbowych z zakresu definiowanego argumentami polecenia. Możliwe warianty użycia polecenia **`seq`** prezentują się następująco:

```
seq OSTATNIA
seq PIERWSZA OSTATNIA
seq PIERWSZA PRZYROST OSTATNIA
```

Polecenie zakłada, że pominięcie którejś z wartości *PIERWSZA* i *PRZYROST* spowoduje przyjęcie dla nich wartości 1. Poniżej podajemy kilka przykładów użycia tego polecenia:

```
$ seq 2
1
2
$ seq 2 3
2
3
$ seq 1 2 5
1
3
5
$ seq 5 -2 1
5
3
1
```

Dysponując takim oto *generatorem* sekwencji można zaprząc go do współpracy z pętlą **`for`**, na przykład w poniższy sposób:

```
1 #! /bin/bash
2
3 for n in `seq 3`
```

```
do
  mkdir dir$n
done
```

Kod powyższy spowoduje utworzenie trzech katalogów o nazwach **dir1**, **dir2** i **dir3**.

Ponieważ zapis taki jest w oczywisty sposób niewygodny i nieczytelny, pętla **for** ewoluowała, przyjmując na pewnym etapie rozwoju powłoki taką oto postać:

```
#! /bin/bash

for $n in {1..3}
do
  echo $n
done
```

Kod powyższy spowoduje wypisanie na *stdout* trzech linii tekstu, zawierających kolejno liczby **1**, **2** i **3**.

Na dalszym etapie dodano również możliwość użycia przyrostu różnego od 1, co zapisuje się w nieco nieczytelny sposób:

```
for $n in {1..5..2}
do
  echo $n
done
```

W tym przypadku na strumieniu *stdout* pojawią się kolejno liczby **1**, **3** i **5**.

Współcześnie używane wersje powłoki Bash umożliwiają na szczęście zapis iterowanej pętli **for** w sposób niemalże identyczny z tym, jaki implementują języki programowania wywodzące się z języka C. Oto przykład:

```
#! /bin/bash

for((n=1; n<=3; n++))
do
  rmdir dir$n
done
```

Zwróć uwagę na użycie podwojonych nawiasów okrągłych – jest to niezbędny element takiej pętli. Użycie pojedynczych nawiasów wywoła błąd.

W tym wariancie możliwe jest – dokładnie jak w języku C – zapisanie pętli nieskończonej:

```
#! /bin/bash

for(( ; ; ))
```

```
do
  echo "Pomocy!"
  sleep 1
done
```

6.10.3. Pętla `while`

Analogicznie do klasycznych języków programowania, pętla **`while`** pozwala organizować cykliczne wykonanie partii kodu, dla których liczba iteracji nie jest znana z góry. Jej składnia dla skryptów powłoki Bash jest następująca:

```
while warunek
do
   polecenie
   :
done
```

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

Przykładem zastosowania pętli **`while`** może być skrypt wypisujący na ekranie wartości wszystkich argumentów jego wywołania (niezależnie od ich liczby). Pętla wykorzystuje polecenie **`shift`**, które powoduje przesunięcie argumentów w lewo (tzn. argument numer 2 staje się argumentem nr 1, numer 3 staje się numerem 2, etc). W ten sposób można przezwyciężyć przeszkodę wynikającą z narzuconego przez powłokę Bash ograniczenia nie zezwalającego na użycie zmiennych **`$n`** dla *n* większego od 9.

Oto skrypt:

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

lub w zwięzłym, równoważanym zapisie:

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

Warunkiem wykonania pętli jest tu sprawdzenie, czy pierwszy argument wywołania skryptu (reprezentowany zmienną **`$1`**) ma niezerową długość – jeśli skrypt został uruchomiony bez żadnych argumentów, to pętla nie zostanie wykonana ani ranu. Jeśli natomiast skrypt został wykonany z argumentami, to w pierwszym wykonaniu pętli zostanie wyświetlona wartość pierwszego argumentu, a następnie

nastąpi przesunięcie argumentów w lewo poleceniem **shift** (drugi argument stanie się pierwszym, trzeci drugim, itd.). Pętla zakończy się jeśli zostaną wyświetlone i przesunięte wszystkie argumenty (zmienna **$1** będzie miała wówczas zerową długość).

Zauważ, że otoczenie nazwy zmiennej **$1** cudzysłowami jest w tym przypadku koniecznością – ich pominięcie spowoduje błąd, gdy zmienna ta nie będzie miała przypisanej wartości, co w przypadku powłoki Bash oznacza, że odwołanie do niej zostanie zastąpione pustym łańcuchem znaków.

6.10.4. Sterowanie pętlą

Do przerywania wykonania pętli, podobnie jak w języku C, służy polecenie **break** – oto przykład jego zastosowania:

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

W powyższym przykładzie pętla **for** zostanie przerwana, jeśli nazwa przechowywana w zmiennej **FILE** nie będzie identyfikowała pliku zwykłego. Możliwe jest również posłużenie się poleceniem **continue**, które (podobnie jak w języku C) rozpoczyna kolejny obrót pętli, pomijając resztę jej wnętrza.

Zachowanie polecenia **continue** można prześledzić na poniższym przykładzie:

```
#! /bin/bash

for FILE in *
do
  if [ ! -L "$FILE" ]
  then
    continue
  fi
  echo $FILE
done
```

Powyższy kod powoduje wypisanie na strumień *stdout* nazw wszystkich dowiązań symbolicznych, jakie istnieją w katalogu bieżącym (w celu wykrycia dowiązań użyto przeznaczonego do tego celu operatora **-L**).

Zauważ, że otoczenie cudzysłowami odwołania do zmiennej **FILE** jest niezbędne w celu uniknięcia błędów, jakie nieuchronnie wystąpiłyby, gdyby nazwa

jakiegokolwiek z badanych plików zawierała spację.

6.11. Instrukcja wyboru

Instrukcja **case** pozwala na dokonanie wyboru wielowariantowego i jest bliskim kuzynem znanej z języka C/C++ instrukcji **switch**.

Składnia instrukcji **case** prezentuje się następująco:

```
case zmienna in
"wzorzec_1") polecenie_1 ;;
"wzorzec_2") polecenie_2 ;;
:
"wzorzec_n") polecenie_n ;;
*) polecenie_domyślne
esac
```

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

Poniższy przykład ilustruje użycie instrukcji **case** do weryfikacji poprawności odpowiedzi udzielonej przez użytkownika:

```
1  #!/bin/bash
2
3  echo "Czy mogę usunąć twój katalog domowy? (tak/nie):"
4  read odp
5  case "$odp" in
6    "tak") echo "Dziękuję"
7           echo "rm -rf /home/yourdir" ;;
8    "nie") echo "Jak sobie życzysz."
9           echo "Pora na CSa";;
10   *) echo "Zła odpowiedź"
11 esac
```

6.12. Wektory

Powłoka Bash umożliwia posługiwanie się wektorami, które – tak jak w języku C – indeksowane są począwszy 0. Istotną różnicą jest jednak fakt, że odwołanie się

do nieistniejącego elementu wektora **nie wywołuje błędu**, a jedynie udostępnia w wyniku pusty łańcuch.

Podobnie do zwyczajów panujących w języku Python, indeksowanie wektora indeksami ujemnymi pozwala uzyskać dostęp do elementów od końca wektora, czyli indeks -1 udostępnia ostatni element wektora, -2 przedostatni, itd.

Składnię umożliwiającą utworzenie wektora i manipulowanie nim przestawimy na przykładach.

Utworzenie wektora może zostać wykonane poprzez przypisanie do niego listy (zauważ, że separatorami elementów listy są białe znaki, więc jeśli któryś element listy zawiera na przykład spację, powinien zostać ujęty w cudzysłowy bądź apostrofy):

```
wektor=(file1.txt "file 2.bin" 128)
```

Ten sam efekt można również osiągnąć przypisując elementy wektora po kolei:

```
wektor[0]=file1.txt
wektor[1]="file 2.bin"
wektor[2]=128
```

Dostęp do wartości elementu wektora o wybranym indeksie wymaga użycia specyficznej składni o poniższej postaci:

```
${wektor[indeks]}
```

Poniższa sekwencja instrukcji spowoduje zamianę wartości drugiego i trzeciego elementu wektora:

```
zm=${wektor[1]}
wektor[1]=${wektor[2]}
wektor[2]=$zm
```

Użycie indeksu podanego jako * (gwiazdka) spowoduje udostępnienie łańcucha powstałego z konkatenacji wszystkich elementów listy, na przykład:

```
$ echo ${wektor[*]}
file1.txt 128 file 2.bin
```

Liczbę elementów w wektorze udostępnia niecodzienny operator **#**, a niecodzienność ta wynika w faktu, że **#** użyty w innych kontekstach oznacza otwarcie komentarza.

```
$ echo ${#wektor[*]}
3
```

Iterowanie po elementach tablicy można wykonać posługując się pętlą **for**:

```
$ for((i=0;i<3;i++)); do echo ${wektor[$i]}; done
file1.txt
128
file 2.bin
```

6.13. Operacje arytmetyczne

Skrypty powłoki mogą wykonywać operacje arytmetyczne, jednakże z powodów historycznych w tej części składni panuje pewien nieporządek.

Najstarszym środkiem umożliwiającym wykonanie prostej operacji arytmetycznej jest specjalnie do tego przeznaczone polecenie **let** o następującej składni:

let ***wyrażenie...***

Uwaga: zaleca się ujęcie całego wyrażenia w cudzysłowy, co pozwala na uniknięcie kłopotów z białymi znakami, jakie ewentualnie mogą pojawić się w jego treści.

Zestaw operatorów akceptowanych przez polecenie **let** jest niemal w całości zgodny z możliwościami oferowanymi przez język C i obejmuje:

=	przypisanie
x++, **++x**	post- i pre-inkrementacja zmiennej
x--, **--x**	post- i pre-dekrementacja zmiennej
!	negacja logiczna
∼	negacja bitowa
******	potęgowanie (jak w języku Python)
+, **-**	dodawanie i odejmowanie oraz zachowanie i zmiana znaku
*****, **/**, **%**	mnożenie, dzielenie, reszta z dzielenia
<<, **>>**	przesuwanie bitowe
<, **>**	porównania ostre
<=, **=>**	porównania nieostre
==, **!=**	równość i nierówność
&, **|**, ∼	operacje bitowe: AND, OR, XOR
&&, **||**	operacje logiczne: AND, OR
w1?w2:w3	operator trójargumentowy (jak w C)

Cenną własnością składni wyrażeń polecenia **let** jest możliwość używania nazw zmiennych **wprost**, bez konieczności poprzedzania ich znakiem **$**.

Poniższy przykład ilustruje obliczanie silni dla podanego argumentu (kod jest oczywiście skrajnie naiwny i służy tylko do celów demonstracji działania polecenia **let**):

```
#! /bin/bash

echo -n "Podaj n:"
read n
let "s=1"
for((i = 2; i <= n; i++))
do
  let "s=s*i"
done
echo "$n! = $s"
```

W trakcie rozwoju powłoki Bash udostępniono użytkownikom bardziej zaawansowany sposób konstruowania wyrażeń, który pozwala używać je wprost w kodzie, bez konieczności wcześniejszego wykonania obliczeń i przypisania ich wyników do zmiennych. Techniki te sprowadzają się do umieszczenia wyrażenia o składni zgodnej z używaną przez polecenie **let** wewnątrz specjalnie do tego przeznaczonych nawiasów składniowych o jednej z poniższych postaci:

$[***wyrażenie*** **]**

`$(( wyrażenie ))`

Poniżej przedstawiamy kod z poprzedniego przykładu przepisany w tej konwencji:

```
#! /bin/bash

echo -n "Podaj n:"
read n
s=1
for((i = 2; i <= n; i++))
do
  s=$((s*i))
done
echo "$n! = $s"
```

1. Manual polecenia **`echo`**: `https://man7.org/linux/man-pages/man1/echo.1.html`
2. Manual polecenia **`export`**: `https://man7.org/linux/man-pages/man1/export.1p.html`
3. Manual polecenia **`read`**: `https://man7.org/linux/man-pages/man1/read.1p.html`
4. Manual polecenia **`set`**: `https://man7.org/linux/man-pages/man1/set.1p.html`
5. Manual polecenia **`seq`**: `https://man7.org/linux/man-pages/man1/seq.1.html`
6. Manual polecenia **`test`**: `https://man7.org/linux/man-pages/man1/test.1.html`
7. Manual polecenia **`top`**: `https://man7.org/linux/man-pages/man1/top.1.html`
8. Manual polecenia **`unset`**: `https://man7.org/linux/man-pages/man1/unset.1p.html`
9. Spis słów zarezerwowanych powłoki Bash: `https://www.gnu.org/software/bash/manual/bash.html#Reserved-Word-Index`
10. Spis zmiennych wbudowanych powłoki Bash: `https://www.gnu.org/software/bash/manual/bash.html#Bash-Variables`
11. Strona domowa projektu GNU Bash: `https://www.gnu.org/software/bash/`
12. Strona domowa powłoki zsh: `https://zsh.sourceforge.io/`

6.14. Zadania do samodzielnego wykonania

Napisz skrypt, który:

1. po uruchomieniu będzie czekał na wprowadzenie działania w postaci **`A @ B`**, gdzie **`A`** i **`B`** to liczby, a **`@`** to jeden z operatorów **`+`**, **`-`**, **`/`** lub **`*`**, po czym będzie wyświetlał wynik wprowadzonego działania
2. będzie przyjmował dowolną liczbę argumentów wywołania będących liczbami całkowitymi, sumował je i na końcu wyświetlał wynik
3. będzie przyjmował dowolną liczbę argumentów wywołania i wyświetlał je w odwrotnej kolejności (wskazówka: wykorzystaj polecenie **`tac`**)
4. będzie przyjmował dowolną liczbę argumentów liczbowych i wyświetlał je posortowane malejąco
5. wyświetli najkrótszą i najdłuższą nazwę użytkownika w systemie (wskazówka: nazwa użytkownika jest pierwszym polem w pliku **`/etc/passwd`**);
6. porówna zawartość pliku o nazwie podanej jako pierwszy argument wywołania z plikami, których nazwy znajdują się w kolejnych liniach pliku o nazwie podanej jako drugi parametr wywołania (porównanie wykonaj za pomocą polecenia **`cmp`**)
7. będzie przyjmował dowolną liczbę argumentów będących nazwami katalogów i wyświetlał dla każdego katalogu jego nazwę oraz liczbę znajdujących się w nim plików nieodczytywalnych (bez prawa **`r`**)
8. w katalogu podanym jako jego argument wywołania zmieni nazwy wszystkich plików i katalogów, tak aby litery małe zostały zamienione na wielkie, a wielkie na małe (czyli na przykład z **`RazDwa.txt`** na **`rAZdWA.TXT`**)
9. zamieni rozszerzenia nazw plików w podanym katalogu z podanych na inne (skrypt ma przyjmować 3 argumenty: katalog, stare rozszerzenie, nowe rozszerzenie); należy uwzględnić możliwość pojawienia się w nazwie pliku kropek
10. sprawdzi, czy wybrany użytkownik (login podany jako argument wywołania skryptu) jest aktualnie zalogowany w systemie; w przypadku niepowodzenia skrypt powinien zaczekać na zalogowanie użytkownika i wtedy poinformować o ewentualnym powodzeniu
11. jako argumenty wywołania (dowolna liczba argumentów) będzie przyjmował loginy użytkowników systemu; skrypt ma sprawdzić, czy któryś z podanych użytkowników jest aktualnie zalogowany w systemie więcej niż raz; dla każdego takiego użytkownika system ma wyświetlić jego login oraz liczbę logowań
12. przyjmuje jako argument nazwę programu (na przykład **`nano`**); skrypt powinien monitorować system, wyświetlając co sekundę informację o tym, którzy użytkownicy używają w danej chwili programu o podanej jako argument

nazwie;

13. pobiera jako argumenty wywołania loginy użytkowników (może być podany więcej niż jeden login) i sprawdzający ile instancji edytora **`nano`** poszczególni użytkownicy mają uruchomione; w jednej linii wyświetlamy login i liczbę uruchomionych edytorów;
14. szyfruje plik za pomocą szyfru Cezara (przesunięcie liter alfabetu o stałą wartość); jako argument podajemy przesunięcie (liczba od 0 do 26) oraz nazwę pliku, który ma zostać zaszyfrowany; wynik powinien być zapisany do nowego pliku o takiej samej nazwie jak plik oryginalny, jednak z dodanym rozszerzeniem **`.cezar`** (użyj polecenia **`tr`**);
15. będzie obliczał n-ty element ciągu Fibonacciego (n podawane jako argument wywołania skryptu); skrypt ma działać **rekurencyjnie**, wywołując sam siebie; na ekranie powinien pojawić się jedynie ostateczny wynik (bez wyników pośrednich);
16. jest iteracyjną wersją skryptu z poprzedniego zadania.