Laboratorium 2: Skrypty.

Wstęp: Konfiguracja stanowiska.

Ćwiczenia można przeprowadzić na przykładzie każdego systemu Unix.
Ćwiczenia są przygotowane na przykładzie systemu Knoppix 3.6 uruchomianego z płyty CD.
W przypadku prowadzenia zajęć na innej wersji systemu Unix prowadzący poinformuje o istotnych różnicach w ćwiczeniach.

Wymagania minimalne:
- Komputer klasy PC ,
- Procesor Intel Celeron 366,
- Pamięć RAM 20 MB,
- Napęd CD-ROM.
- BIOS umożliwiający uruchomienie komputera z płyty CD.

UWAGA! Wszelkie zmiany wprowadzone do systemu plików i konfiguracji systemu operacyjnego znikają po restarcie komputera, ponieważ system jest uruchamiany z płyty CD i system plików nie jest zapisywany na dysku.

Przygotowanie komputera:
- Uruchom komputer.
- Włóż do napędu CD-ROM płytę zatytułowaną „SO Laboratorium 1”.
- Zaczekaj na komunikat określający sposób uruchomienia programu konfigurującego BIOS komputera.
- Uruchom program konfigurujący BIOS komputera.
- Odszukaj w konfiguracji komputera parametr umożliwiający uruchomienie systemu operacyjnego (boot from CD-ROM) z płyty CD.
- Ustaw opcję uruchamiania systemu z płyty CD.
- Zapisz konfigurację.

Rysunek 1 Uruchamianie systemu Knoppix.

- Zaczekaj komputer się uruchomi i pojawi się ekran powitalny systemu (Rysunek 1)
- Wpisz knoppix 2
- Naciśnij Enter.

Zmienne powłoki

Jeśli uczysz się programować, wiesz zatem, że programowanie przypomina algebrę. Język programowania i algebra pozwalają bowiem odwołać się do wartości poprzez nazwę; posiadają również wyszukane mechanizmy do manipulacji nazwanymi zmiennymi.
Powłoka jest językiem programowania posiadającym własne prawa. Pozwala odwoływać się do zmiennych zwanych zmiennymi powłoki lub zmiennymi środowiskowymi. Aby przypisać wartość zmiennej powłoki, użyj polecenia, które ma następującą składnię:
`variable=value`
Na przykład:
`DifficultyLevel=1`
przypisuje wartość równą 1 do zmiennej środowiskowej zwanej `DifficultyLevel`. W przeciwieństwie do zmiennych algebraicznych, zmienne powłoki mogą posiadać wartości nienumeryczne. Na przykład:
`Difficulty=medium`
Zmienne środowiskowe są często używane w systemie Unix, ponieważ umożliwiają w prosty sposób przeniesienie wartości z jednego polecenia do drugiego. Programy mogą otrzymywać wartości zmiennych i używać ich do modyfikacji własnych operacji; w podobny sposób korzystają z podawanych przez użytkownika argumentów.
Aby obejrzeć listę zmiennych środowiskowych, użyj polecenia `set`. Ponieważ może się zdarzyć, że będzie ich dużo, możesz użyć potoku i polecenia `more` do oglądania wyjścia polecenia `set` ekran po ekranie:
`set | more`
Wciśnij **Spację**, aby zobaczyć następną stronę lub **Enter**, aby obejrzeć następny wiersz. Wśród tych zmiennych, które zostaną wyświetlone na ekranie znajdziesz pewnie też te, które są przedstawione w tabeli 13.8. Wartości tych zmiennych są generalnie ustawiane przez standardowe skrypty startowe powłoki, o których była wcześniej już mowa.
Tabela 13.8. Ważne zmienne środowiskowe

Zmienna	Funkcja
DISPLAY	Używany serwer X, na przykład: localhost:0.
HOME	Ścieżka bezwzględna katalogu domowego użytkownika.
HOSTNAME	Internetowa nazwa komputera.
LOGNAME	Nazwa logowania użytkownika.

MAIL	Ścieżka bezwzględna katalogu poczty użytkownika.
PATH	Ścieżka przeszukiwań (zobacz w następnej sekcji).
SHELL	Ścieżka absolutna aktualnie używanej powłoki.
TERM	Rodzaj terminalu.
USER	Aktualna nazwa użytkownika, może się różnić od nazwy logowania, jeśli użytkownik wydał polecenie `su`.

Wartość zmiennej środowiskowej możesz wstawić do polecenia, poprzedzając ją znakiem dolara (`$`). Aby nie zmieszała się ona z otaczającym tekstem (argumentami, opcjami itp.), zamknij ją w nawiasach klamrowych (`{}`). Jest to dobry zwyczaj, aczkolwiek nie wymagany. Na przykład zmieńmy aktualny katalog roboczy na katalog domowy za pomocą zmiennej `HOME`:

```
cd ${HOME}
```

Oczywiście wydanie polecenia `cd` bez żadnych argumentów da nam ten sam efekt. Jednakże załóżmy, że chciałbyś zmienić aktualny katalog roboczy na podkatalog */praca* znajdujący się w katalogu domowym. Poniższe polecenie, pokazuje jak tego dokonać:

```
cd ${HOME}/praca
```

Najłatwiejszym sposobem na wyświetlenie wartości zmiennej środowiskowej jest podanie zmiennej jako argumentu polecenia `echo`. Na przykład sprawdźmy, jaka jest wartość zmiennej `HOME`:

```
echo ${HOME}
```

Aby wartość zmiennej środowiskowej była dostępna nie tylko dla powłoki, ale także dla programów wywoływanych poprzez powłokę, należy tę wartość wyeksportować. Aby tego dokonać, użyj polecenia `export`:

```
export zmienna
```

gdzie *zmienna* oznacza nazwę zmiennej, którą chcesz eksportować. Podczas eksportowania zmiennej możesz również nadać jej nową wartość:

```
export zmienna=wartość
```

Wartość zmiennej możesz usunąć, podając zmiennej pustą wartość (wartość zerową)

```
zmienna=
```

Jednakże zmienna środowiskowa mająca nawet wartość zerową nadal pozostaje zmienną powłoki i pojawia się na wyjściu polecenia `set`. Aby ją stamtąd usunąć, wydaj polecenie:

```
unset zmienna
```

Teraz jej nie zobaczysz po wykonaniu polecenia `set`.

Poznawanie skryptów powłoki

Podrozdział wyjaśnia, jak działają bardziej zaawansowane skrypty powłoki. Informacje są przedstawione po to, abyś szybko i wydajnie mógł pisać własne skrypty. Rozpoczynamy od przetwarzania argumentów skryptów, a następnie przejdziemy do operacji warunkowych i logicznych.

Przetwarzanie argumentów

Z łatwością można napisać program, który będzie przetwarzał argumenty, ponieważ istnieje zestaw specjalnych zmiennych powłoki, który zawiera wartości dla argumentów określanych podczas inicjalizacji skryptu. Tabela 13.10 przedstawia te zmienne.

Oto prosty program, który wyświetla wartość swojego drugiego argumentu:

```
echo Mój drugi argument ma wartość $2
```

Załóżmy, że zapiszesz ten skrypt do pliku o nazwie *drugi*; zmień tryb dostępu tak, aby mógł być wykonywany i wydaj polecenie:

```
./drugi a b c
```

Na wyjściu pojawi się wynik tej operacji:

```
Mój drugi argument ma wartość b
```

Tabela 13.10. Specjalne zmienne powłoki

Zmienna	Znaczenie
`$#`	Liczba argumentów
`$0`	Nazwa polecenia
`$1, $2,...,$9`	Indywidualne argumenty polecenia
`$*`	Cała lista argumentów, traktowana jako pojedyncze słowo
`$@`	Cała lista argumentów, traktowana jako seria słów
`$?`	Status wyjścia poprzedniego polecenia. Wartość 0 oznacza pomyślne zakończenie (wyjście)

`$$`	Id (identyfikator) aktywnego procesu

Zwróć uwagę na fakt, że powłoka pozwala odwołać się bezpośrednio tylko do dziewięciu argumentów. Niemniej jednak istnieje możliwość przywołania większej liczby argumentów. Służy do tego polecenie `shift`, które odrzuca wartość pierwszego argumentu i przesuwa pozostałe wartości o jedną pozycję w dół (lub do góry). Tym sposobem po wykonaniu polecenia `shift` zmienna powłoki `$9` zawiera wartość dziesiątego argumentu. Aby przywołać jedenasty lub każdy następny argument, należy wykonać polecenie `shift` odpowiednią ilość razy.

Kody wyjścia

Zmienna powłoki `$?` utrzymuje numeryczny status wyjścia dla każdego ostatnio wykonanego polecenia. Zasada jest taka, że wartość zerowa tej zmiennej oznacza pomyślne zakończenie programu, tymczasem każda inna wartość oznacza nieprawidłowe wykonanie polecenia.
Kod błędu możesz ustawić w skrypcie, za pomocą polecenia `exit`, które kończy pracę skryptu podając określony status wyjścia. Składnia `exit` jest następująca:

```
exit status
```

gdzie *`status`* jest nieujemną liczbą całkowitą, która określa status wyjścia.

Logika warunkowa

Skrypty powłoki mogą używać logiki warunkowej, która pozwala im podejmować różne działania w zależności od wartości argumentów, zmiennych środowiskowych lub ich warunków. Na przykład polecenie `test` pozwala określić warunek, który może być albo prawdziwy, albo fałszywy. Wszystkie polecenia warunkowe (włączając `if`, `case`, `while` oraz `until`) używają polecenia `test` do szacowania warunków.

Polecenie test

Tabela 13.11 przedstawia opis niektórych powszechnie stosowanych form argumentów używanych do polecenia `test`. Polecenie sprawdza argumenty i ustawia kod wyjściowy na zero, co oznacza, że dany warunek był prawdziwy lub podaje wartość większą od zera, która mówi, że warunek był nieprawdziwy.

Tabela 13.11. Powszechnie używane formy argumentów polecenia test

Forma	Funkcja
`-d plik`	Określony *`plik`* istnieje i jest katalogiem
`-e plik`	Określony *`plik`* istnieje
`-r plik`	Określony *`plik`* istnieje i jest do odczytu
`-s plik`	Określony *`plik`* istnieje i ma niezerowy rozmiar
`-w plik`	Określony *`plik`* istnieje i jest do zapisu
`-x plik`	Określony *`plik`* istnieje i jest wykonywalny
`-L plik`	Określony *`plik`* istnieje i jest dowiązaniem symbolicznym
`f1 -nt f2`	Plik *`f1`* jest nowszy niż plik *`f2`*
`f1 -ot f2`	Plik *`f1`* jest starszy niż plik *`f2`*
`-n s1`	Łańcuch *`s1`* ma niezerową długość
`-z s1`	Łańcuch *`s1`* ma zerową długość
`s1 = s2`	Łańcuch *`s1`* jest taki sam jak *`s2`*
`s1 =! s2`	Łańcuch *`s1`* nie jest taki sam jak *`s2`*
`n1 -eq n2`	Liczba całkowita *`n1`* jest równa liczbie *`n2`*
`n1 -ge n2`	Liczba całkowita *`n1`* jest większa bądź równa liczbie *`n2`*
`n1 -gt n2`	Liczba całkowita *`n1`* jest większa niż liczba *`n2`*
`n1 -le n2`	Liczba całkowita *`n1`* jest mniejsza bądź równa liczbie całkowitej *`n2`*
`n1 -lt n2`	Liczba całkowita *`n1`* jest mniejsza niz liczba *`n2`*
`n1 -ne n2`	Liczba n1 nie jest równa liczbie *`n2`*
`!`	Operator przeczenia (*not* — nie), który odwraca wartość danego warunku
`-a`	Operator łączności (*and* — i), który łączy dwa warunki. Oba warunki muszą być prawdziwe, aby całkowity rezultat dał prawdziwy wynik
`-o`	Operator alternatywy, który łączy dwa warunki. Jeśli jeden z dwóch warunków jest prawdziwy, wynik jest rownież prawdziwy
`\( ... \)`	Wyrażenia w poleceniu test można grupować zamykając je w `\(` i `\)`

Dla przykładu rozważmy taki skrypt:

```
test -d $1
echo $?
```

Skrypt ten sprawdza, czy pierwszy argument określa katalog i wyświetla status wyjścia, wartość zero lub inną w zależności od wyniku testu.

Załóżmy, że powyższe polecenia są przechowywane w pliku *tester*, który posiada atrybut do odczytu oraz do wykonania. Wykonanie tego skryptu może dawać następujące wyniki:

```
$ ./tester /
0
$ ./tester /missing
1
```

Oznaczają one odpowiednio: `0` — katalog `/` istnieje, a wartość `1` — katalog `/missing` nie istnieje.

Polecenie if

Samo polecenie test nie jest zbyt przydatne, chyba że zostanie użyte w połączeniu z innymi instrukcjami warunkowymi — na przykład z poleceniem `if`. Składnia polecenia `if` jest następująca:

```
if polecenie
then
polecenia
else plecenia
fi
```

Zazwyczaj poleceniem, które występuje bezpośrednio po poleceniu `if` jest `test`. Jednakże, nie jest to warunkiem koniecznym. Polecenie `if` jedynie wykonuje określoną warunki i sprawdza status wyjścia. Jeśli jest on równy 0, wykonywany jest pierwszy zestaw poleceń, jeśli nie wykonywany jest drugi zestaw. Skrócona forma polecenia `if` nie robi nic, jeśli określony warunek jest nieprawdziwy.

```
if polecenie
then
polecenie
fi
```

Kiedy wpisujesz polecenie `if`, zajmuje ono kilka wierszy, a jednak jest uważane za pojedyncze polecenie. Do wprowadzania takich poleceń, powłoka dostarcza specjalnego znaku zachęty (zwanego czasami drugim znakiem zachęty) po wprowadzeniu każdego wiersza. Często skrypty są tworzone za pomocą edytorów tekstowych, wtedy nie są widoczne dodatkowe znaki zachęty powłoki.

Dla przykładu spróbujmy usunąć *plik1*, jeśli jest on starszy niż *plik2*. Poniższe plecenie powinno poprawnie rozwiązywać to zadanie:

```
if test plik1 -ot plik2
then
   rm plik1
fi
```

Polecenie możesz dołączyć do skryptu, który akceptuje argumenty określające nazwy plików:

```
if test $1 -ot $2
then
   rm $1
   echo Usunięto stary plik
fi
```

Możesz zapisać polecenie do pliku o dowolnej nazwie np. *usuwacz* i wykonać go:

```
./usuwacz plik1 plik2
```

A skrypt usunie `plik1`, jeśli jest on starszy od `pliku2`.

Polecenie case

Polecenie `case` dostarcza bardziej wyrafinowanej metody przetwarzania warunkowego:

```
case wartość in
  wzorzec1) polecenia ;;
  wzorzec2) polecenia ;;
  ...
esac
```

Polecenie przestępuje do przystępuje do wyszukiwania określonych wartości odpowiadających podanym wzorom (wzorzec1, wzorzec2). Jeśli istnieje odpowiednik pierwszego wzoru, polecenie związane z nim zostanie wykonane. Wzory budowane są przy użyciu znaków i metaznaków, takich jak te używane do określania argumentów poleceń. Przyjrzyjmy się poleceniu `case`, którego zadaniem jest zinterpretowanie wartości pierwszego argumentu skryptu:

```
case $1 in
   -r) echo Wymuś usuwanie bez potwierdzania ;;
   -i) echo Potwierdź przed usunięciem ;;
    *) echo Nieznany argument ;; esac
```

Polecenie wyświetla wiadomość zależnie od wartości pierwszego argumentu skryptu. Dobrym ćwiczeniem jest dołączenie do tego przykładu końcowego wzoru, który odpowiadałby każdej wartości argumentu skryptu.

Polecenie while

Pozwala ono na wykonywanie serii poleceń iteracyjnie, to znaczy wielokrotnie, dopóki sprawdzane warunki będą prawdziwe:

```
while polecenie
do
polecenia
done
```

Oto skrypt, który używa polecenia `while` do wyświetlania własnych argumentów w kolejnych wierszach:

```
echo $1
while shift 2>/dev/null
do
   echo $1
done
```

Część polecenia `while` obejmująca pętlę do może także zawierać polecenia `if`, `case` — a nawet kolejne polecenie `while`. Jednakże w tym ostatnim przypadku, skrypt taki robi się szybko nieczytelny. Powinieneś stosować polecenia warunkowe razem tylko dla osiągnięcia jasnych wyników. Możesz dołączać komentarze wszędzie tam, gdzie wstawiasz zawikłane konstrukcje warunkowe.

Polecenie until

Polecenie działa na niemalże identycznej zasadzie jak `while`, z jednym tylko wyjątkiem — pozwala mianowicie wykonywać serie poleceń iteracyjnie, dopóki sprawdzane warunki są fałszywe.

```
until polecenie
do
polecenia
done
```

Spójrz na skrypt, który używa polecenia `until` do wyświetlania własnych argumentów w kolejnych wierszach, aż nie pojawi się argument o wartości czarny:

```
until test $1 = "czarny"
do
   echo $1
   shift
done
```

Jeśli zapiszemy ten skrypt do pliku o nazwie *kolorki* w katalogu roboczym, to polecenie:

```
./kolorki zielony niebieski żółty czarny czerwony
```

wyświetli nam taki oto wynik:

```
zielony
niebieski
żółty
```

Polecenie for

Polecenie iteruje (powtarza, zapętla) elementy określonej listy:

```
for zmienna in lista
do
polecenia
done
```

Za pomocą poleceń możesz odwołać się do aktualnego elementu listy poprzez zmienną powłoki `$zmienna`. Lista zazwyczaj przyjmuje formę serii argumentów, które mogą zawierać metaznaki. Rozważmy na przykład taką instrukcję:

```
for i in 2 4 6 8
do
   echo $i
done
```

Wyświetla ona wartości `2 4 6 8` w kolejnych wierszach.

Specjalna forma tego polecenia iteruje argumenty dowolnego skryptu:

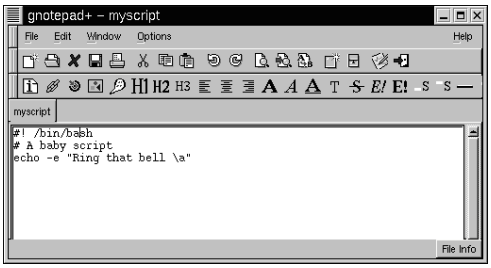
```
for zmienna
do
polecenia
done
```

Na przykład następny skrypt wyświetla własne argumenty w kolejnych wierszach:

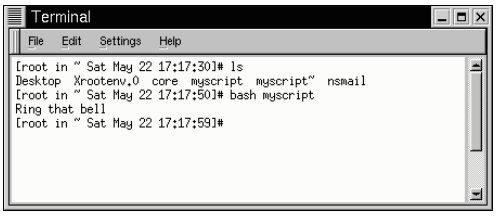
```
for i
do
   echo $i
done
```

Polecenia break oraz continue
Są to proste polecenia i nie przyjmują żadnych argumentów. Kiedy powłoka dostrzeże polecenie `break`, natychmiast wychodzi z pętli (*while*, *until*, *for*). Kiedy natomiast powłoka napotka polecenie `continue`, natychmiast przerywa aktualną iterację pętli. Jeśli warunki pętli na to pozwalają, wykonane mogą zostać inne powtórzenia pętli; w przeciwnym razie następuje wyjście z pętli.

Podstawowe informacje o tworzeniu skryptów powłoki



Rysunek 11.10. *Skrypty powłoki można tworzyć za pomocą dowolnego edytora tekstów (na rysunku edytor Gnotepad+).*



Rysunek 11.11. *Skrypt można uruchomić, podając jego nazwę jako argument polecenia uruchamiającego powłokę.*

Tworzenie skryptu powłoki sprowadza się do utworzenia pliku tekstowego zawierającego odpowiednie polecenia i zapisania go.
Taki skrypt może zostać uruchomiony przez przekazanie jego nazwy jako argumentu polecenia uruchamiającego powłokę lub też jako odrębny plik wykonywalny. Zwykle skrypty są właśnie plikami wykonywalnymi.

Aby utworzyć skrypt powłoki

1. Uruchom edytor tekstów, na przykład vi
2. W pierwszym wierszu skryptu wprowadź tekst **#!/bin/bash**. Powinien on zawierać pełną ścieżkę dostępu do programu, który będzie interpretował skrypt.
3. W następnym wierszu skryptu wprowadź znak # (oznaczający początek komentarza) i krótką informację dotyczącą działania skryptu (jest to powszechnie przyjęta konwencja): **# Malutki skrypt**.
4. Wprowadź polecenia będące właściwą treścią skryptu (w tym przykładzie będzie to polecenie wyświetlające tekst i wydające krótki dźwięk): **echo –e "Ring that bell \a"**
5. Zapisz plik na przykład pod nazwą **myscript**.

Aby uruchomić skrypt za pomocą powłoki

W wierszu poleceń wpisz nazwę powłoki, a po niej nazwę pliku zawierającego skrypt:**bash myscript**
Tekst zostanie wyświetlony na ekranie, a terminal wyda krótki dźwięk
(**rysunek 11.11**).

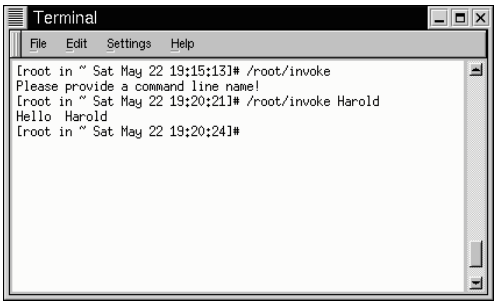
Aby skrypt był wykonywalny

1. Zmień prawa dostępu do pliku zawierającego skrypt, tak aby był on wykonywalny, korzystając z polecenia **chmod** zgodnie ze wskazówkami zawartymi w rozdziale 10., na przykład
   chmod u+x myscript.
2. Upewnij się, że katalog, w którym zapisany jest skrypt, wchodzi w skład ścieżki przeszukiwania, porównując dane zwracane przez polecenia **pwd** i **echo $PATH**. Jeśli nie, dodaj go, postępując zgodnie z informacjami z rozdziału 9.

3. W wierszu poleceń wpisz nazwę skryptu, **myscript**. Tekst zostanie wyświetlony na ekranie, a terminal wyda krótki dźwięk.

Wskazówki

- Możesz wykorzystać zawartość historii sesji opisanej w rozdziale 9., aby zamienić n ostatnio wydanych poleceń w skrypt, po prostu zapisując n ostatnich wierszy pliku historii w nowym pliku, na przykład wydając polecenie **history 10 > myscript**. Takie rozwiązanie pozwala na łatwe rejestrowanie makropoleceń.
- Jeśli plik skryptu nie jest plikiem wykonywalnym, możesz go uruchomić używając operatora **.** (kropka), na przykład **. myscript**.



Rysunek 11.12. *W skryptach powłoki można korzystać z wartości parametrów pozycyjnych.*

Składnia skryptów powłoki

Zmienne używane w skryptach powłoki nie mają typów, co oznacza, że nie trzeba określać, czy zmienna zawierać będzie tekst czy wartości liczbowe (lub cokolwiek innego).
Zmienne deklaruje się równocześnie z przypisywaniem im wartości za pomocą znaku równości, na przykład
licznik = 0
nazw = "Harold Davis"
Jeśli w tekście nie znajdują się spacje, nie trzeba otaczać go cudzysłowem.
Symbol **$** używany jest do przypisywania wartości jednej zmiennej jakiejś innej zmiennej, na przykład
licznik = $mojazm

Korzystanie z argumentów podawanych w wierszu poleceń

Argumenty, z których chcesz korzystać w skrypcie, można przekazać do niego za pomocą wiersza poleceń. Pierwszy z argumentów w skrypcie dostępny jest pod nazwą **$1**, drugi – **$2** i tak dalej.
Poniżej przedstawiamy skrypt ilustrujący to zagadnienie:

```
#Wyswietlanie argumentow z wiersza polecen
if [ $# –eq 0 ]
then
  echo "Please provide a command line name!"
else
  echo "Hello " $1
fi
```

Jeśli zapiszesz taki skrypt i uruchomisz go bez argumentów, wyświetli on tekst „Please provide a command line name!”. W przeciwnym przypadku wyświetlony zostanie identyfikator podany jako argument (**rysunek 11.12**).

Drobna uwaga: **fi** na końcu skryptu to nie literówka – jest to słowo **if** (jeśli) pisane od końca, ponieważ oznacza ono koniec instrukcji **if**. Podobnie słowo **esac** oznacza koniec instrukcji **case**.
W skrypcie występuje również zmienna
$# – jest to zmienna wewnętrzna, oznaczająca liczbę parametrów podanych w wierszu poleceń (patrz **tabela 11.1**). Operator –**eq** służy do porównania wartości zmiennej **$#** z zerem (patrz **tabela 11.2**). Zmienna **$1** zawiera wartość pierwszego argumentu przekazanego w wierszu poleceń.

Aby pozwolić na wprowadzenie danych przez użytkownika

1. Uruchom edytor tekstów.
2. Wpisz następujący skrypt:
   ```
   #! /bin/bash
   # Wprowadzanie danych przez uzytkownika
   echo "Enter your name:"
   ```

read name
echo "Is it time for tea, $name?"

3. Zapisz plik pod nazwą **yrname**.
4. W wierszu poleceń przypisz plikowi prawo wykonywalności:
   chmod +x yrname
5. Uruchom skrypt, wpisując **yrname**. Skrypt poprosi o podanie imienia, a następnie wyświetli je jako część komunikatu (**rysunek 11.13**).

Wskazówki

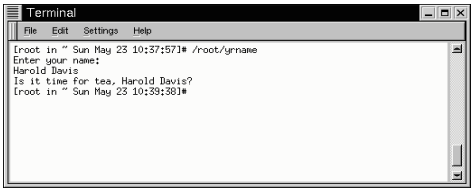
- Umieszczając polecenie **read** w nieskończonych pętlach, można tworzyć struktury menu i aplikacje obsługiwane za pomocą menu.
- Jeśli chcesz tworzyć bardziej skomplikowane programy, powinieneś zamiast ze skryptów powłoki skorzystać z jakiegoś innego języka skryptowego, na przykład Perla, ponieważ w bardziej rozbudowanych skryptach powłoki trudno jest szukać ewentualnych błędów.

Tabela 11.1. *Zmienne wewnętrzne powłoki*

Nazwa zmiennej	Znaczenie
$#	Liczba argumentów przekazanych w wierszu poleceń
$0	Nazwa programu powłoki
$*	Pojedynczy tekst utworzony ze wszystkich argumentów przekazanych w wierszu poleceń

Tabela 11.2. *Operatory porównania.*

Operator	Znaczenie
=	równe (łańcuchy znaków)
!=	nierówne (łańcuchy znaków)
-eq	równe (liczby)
-ge	większe lub równe (liczby)
-le	mniejsze lub równe (liczby)
-ne	różne (liczby)
-gt	większe niż (liczby)
-lt	mniejsze niż (liczby)



Rysunek 11.13. *Aby pozwolić użytkownikowi na wprowadzanie danych do skryptu, skorzystaj z polecenia read.*

Instrukcje iteracyjne i warunkowe

Instrukcje iteracyjne i warunkowe to między innymi:

- instrukcje for…do…done,
- instrukcje while…do…done,
- instrukcje until…do…done,
- instrukcje select element in listaelementow…do…done,
- instrukcje if…elif…else…fi,
- instrukcje case.

Dokładniejsze informacje o składni tych poleceń znajdziesz w dokumentacji powłoki bash

Poniżej przedstawiamy skrypt, który wykorzystuje dane wprowadzane przez użytkownika oraz nieskończoną pętlę, aby obliczyć sumę dowolnej ilości liczb.

```
#! /bin/bash
# Program interaktywny
sum=0
while :
do
  echo -n "Add another number (y/n): "
  read dalej
  if [ $dalej = n ] ; then
    break
  fi
  if [ $dalej != y ] ; then
    echo '"y" or "n" please!';
  continue
  fi
  echo -n "Enter a number to add to the sum: "
  read nowy
  sum=$(($sum + $nowy))
  echo "Sum so far is $sum"
done
echo -e "Your sum is $sum"
echo "Bye"
```

Aby uruchomić skrypt sumujący liczby

1. Zapisz skrypt do pliku o nazwie **sum**.
2. Przypisz mu prawo do wykonywania:
   chmod +x sum.
3. W wierszu poleceń wpisz **sum**.
4. Wciśnij **Enter**. Skrypt zostanie uruchomiony i będzie prosił o wprowadzanie kolejnych liczb (**rysunek 11.14**).

Wskazówki

- Dwukropek po instrukcji **while** w poleceniu **while :** jest argumentem, który ma zawsze wartość prawdy logicznej, co oznacza, że pętla będzie wykonywana bez końca. Pętla jest przerywana wewnątrz instrukcji **if** w przypadku, gdy użytkownik zdecyduje, że nie chce wprowadzać dalszych liczb i wpisze **n**.
- Skrypt nie sprawdza poprawności wprowadzanych danych. Jeśli wprowadzone zostanie cokolwiek innego niż liczba, skrypt zakończy działanie z komunikatem o błędzie składni (syntax error).

Przykłady skryptów:

(symbol # oznacza komentarz)

1. Przykład

```
# Skrypt sprawdza obecność pliku lub katalogu nazwa
# w katalogu biezącym.
if ls | grep nazwa > /dev/null
then
      echo Plik istnieje.
else
      echo Plik nie istnieje.
fi
```

2. Przykład

```
# Skrypt wyświetla zawartość pliku (jego nazwa jest parametrem
# wywołania) lub komunikat, gdy plik nie daje się czytać.
if test -r $1
then
      more $1
else
      echo Plik $1 nie daje się czytac lub nie istnieje
```

```
fi
```

3. Przykład

```
# Skrypt co 15 sekund informuje o obecności
# użytkownika ciwe w systemie.
# Znak & na końcu oznacza uruchomienie w tle.
while who | grep ciwe > /dev/null
do
      echo Uzytkownik ciwe pracuje!
      sleep 15
done &
```

4. Przykład

```
# Skrypt co 15 sekund informuje o nieobecności użytkownika
# nowak w systemie. Znak & oznacza uruchomienie w tle.
until who | grep nowak > /dev/null
do
      echo Uzytkownik nowak nie pracuje!
      sleep 15
done &
```

5. Przykład

```
# Skrypt wyświetla nazwy plików z katalogu bieżącego
# pasujące do wzorca, który jest parametrem wywołania skryptu.
# Na końcu skrypt podaje liczbę takich plików.

licznik=0
for zm in $*
do
      if test -r $zm
      then
            echo Znaleziono plik $zm
            licznik=`expr $licznik + 1`         # uwaga na spacje!
      fi
done
echo Razem: $licznik plikow
```

Język Awk

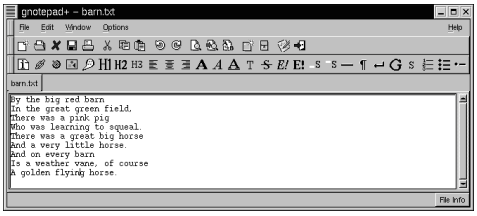
Język GNU awk – lub gawk – jest wersją języka służącego do przetwarzania tekstów. Jego autorami byli Alfred V. Aho, Peter J. Weinberger oraz Brian W. Kernighan (nazwa *awk* pochodzi od pierwszych liter ich nazwisk). Awk najlepiej nadaje się do przetwarzania plików tekstowych, ponieważ „myśli" on w kategoriach pól i rekordów, którymi zwykle są wyrazy i wiersze takich plików. Z tego powodu niezbyt dobrze nadaje się on do przetwarzania plików binarnych. Typowym zastosowaniem tego języka jest tworzenie sformatowanych raportów na podstawie dostarczonych danych.
Mnóstwo informacji na temat programowania w tym języku znajdziesz w katalogu **/usr/doc/gawk-3.0.6**.
Gawk może być używany bezpośrednio z wiersza poleceń; w takim przypadku polecenia języka gawk należy otoczyć pojedynczym cudzysłowem, aby powłoka nie próbowała ich interpretować. Programy w języku Gawk dłuższe niż jedno czy dwa polecenia należy zapisywać w plikach.
W języku Gawk zmienna NF jest zmienną predefiniowaną zawierającą liczbę pól każdego z rekordów (czyli wyrazów w wierszu). Zmienna $0 zawiera cały rekord (wiersz),
$1 – pierwsze pole wiersza (czyli pierwszy wyraz), $2 – drugie pole wiersza (czyli drugi wyraz) itd.

Aby wypisać liczbę wyrazów znajdujących się w każdym wierszu pliku tekstowego

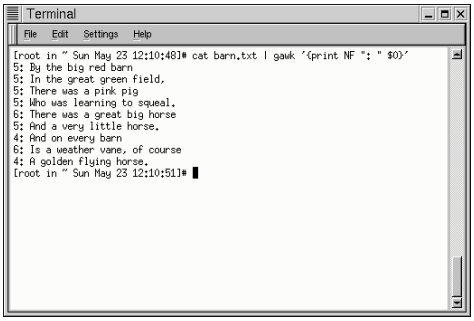
1. Utwórz plik tekstowy (**rysunek 11.21**).
2. Zapisz plik pod nazwą **barn.txt**.
3. W wierszu poleceń wpisz
   cat barn.txt | gawk '{print NF ": " $0}'
   Wykorzystaliśmy tu polecenie **cat** i mechanizm potoków, za pomocą którego przekazaliśmy plik tekstowy na wejście programu gawk.
4. Wciśnij **Enter**. Wyświetlona zostanie liczba wyrazów w każdym z wierszy wraz z ich tekstem (**rysunek 11.22**).

Wskazówka

- Skrypt w języku gawk ma w tym przypadku postać **{print NF ": " $0}**.



Rysunek 11.21. *Język Gawk najlepiej nadaje się do przetwarzania plików tekstowych o określonej strukturze.*



Rysunek 11.22. *Gawk może zostać uruchomiony z wiersza poleceń lub po zapisaniu odpowiednich poleceń do pliku; w tym przypadku wyświetla on liczbę wyrazów wchodzących w skład każdego wiersza pliku.*

Język Perl

Perl jest językiem wysokiego poziomu opartym na języku C, językach skryptowych takich jak awk, oraz językach skryptowych powłoki. Choć dostępne są kompilatory programów w nim napisanych, programy zwykle kompilowane są tuż przed ich uruchomieniem. Perl jest bardzo przenośny – program w tym języku może być bez większych zmian uruchomiony w prawie każdym systemie operacyjnym.
Perl znakomicie radzi sobie z obsługą procesów, plików i tekstu. Jest wykorzystywany do obsługi danych wprowadzanych przez użytkowników na stronach WWW za pomocą mechanizmów CGI. Warto go poznać choćby ze względu na popularność na stronach WWW.
Poza tym Perl jest często używany do automatyzacji zadań administracyjnych, ponieważ posiada on dostęp do powłoki.
Zmienne w języku Perl nie posiadają typów, ale w języku tym występuje kilka ich rodzajów:

- **Zmienne skalarne** to liczby lub tekst, w zależności od kontekstu.
- **Tablice** mogą zawierać elementy dostępne za pomocą indeksowania.
- **Tablice asocjacyjne** mogą zawierać elementy dostępne za pośrednictwem kluczy.

Więcej informacji na temat języka Perl znajdziesz w książce *Perl and CGI: Visual QuickStart Guide*, wydanej przez Peachpit Press. Zajrzyj też na stronę **http://www.perl.com** poświęconą w całości temu językowi.