Programowanie w

Bashu

czyli jak pisać skrypty w Linuksie

Wersja: 1
Data: 28.02.2011

Artur Pyszczuk

Spis treści

1 Wprowadzenie

1.1 Informacje od autora

Witaj. Na początku każdej książki, tudzież jakiegoś większego poradnika występują informacje od autora, tak więc i ja napiszę coś tutaj. Mam nadzieję, że informacje zawarte w tej małej książeczce sprawią, iż spodoba Ci się operowanie Linuksem za pomocą konsoli. Jeśli jeszcze tego nie lubisz, to może sam fakt umiejętności pisania całkiem użytecznych skryptów, które jak wiadomo znacząco przyspieszają pracę z komputerem – zwłaszcza przy nudnych cyklicznie powtarzających się czynnościach – przekona Cię do tego. Po przerobieniu wszystkich przykładów występujących w kolejnych rozdziałach oraz trenowaniu samemu z różnymi kombinacjami poszczególnych elementów nabierzesz większej wprawy i na pewno zrozumiesz jak fajnie i całkiem łatwo jest pisać skrypty. Informacje zawarte w tej książce nie są jedynym źródłem informacji z których należy korzystać, nie mniej jednak dbałem o to, by czytelnik po przeczytaniu kolejnych rozdziałów wiedział jak najwięcej o danym zagadnieniu. W internecie występuje mnóstwo poradników dotyczących programowania w Bashu. Kilka z nich wymieniono w bibliografii. Starałem się zebrać najistotniejsze informacje w jednym miejscu, by czytając ten poradnik użytkownik nie musiał za każdym razem, gdy czegoś nie rozumie szukać odpowiedzi w innych źródłach. Pomimo wielu starań niektóre aspekty mogą być niejasno wytłumaczone, za co z tego miejsca przepraszam. Jeśli zechcesz mnie o tym fakcie poinformować – będę wdzięczny. Wszelkie informacje odnośnie tego poradnika możesz wysyłać na adres apyszczuk@gmail.com.

 Życzę miłej lektury.

Artur Pyszczuk

1.2 Jak napisana jest ta książka

W poradniku tym występuje duża ilość skryptów, które wystarczy wkleić do edytora tekstu i uruchomić. Nie powinno być większego problemu z tym. Podczas omawiania zasady działania poszczególnych mechanizmów programowania w Bashu wymieniałem szereg poleceń systemowych oraz zmiennych użytych w tych kodach. W niektórych są informacje o klawiszach, które można

naciskać za pośrednictwem klawiatury. W poniższej tabeli zostały wyszczególnione wszystkie elementy, oraz sposób ich zapisu w poradniku.

Opis parametru	Przykład
Listing ze skryptem	`#!/bin/bash`
Polecenie systemowe wykonywane w konsoli	`$ ./skrypt.sh`
Wykaz błędów, zawartość pliku	`./3.sh: line 3: ...`
Nazwy poleceń systemowych, nazwy zmiennych ze skryptu, części instrukcji	echo, cat, zmienna3, if, case
Klawisze z klawiatury, opcja z okienka (dialogi)	ESC, ENTER, OK, Anuluj
Znaki specjalne, nazwy katalogów, nazwy plików	**#**, **$**, **/tmp**, **plik1.txt**

Tabela 1.2.1 Informacje nawigacyjne

1.3 Dla kogo przeznaczona jest ta książka

Prawdę mówiąc poradnik ten jest dla każdego kto chce nauczyć się programować w skryptowym języku programowania – Bash. Jeśli pisałeś już kiedyś skrypty i chcesz przypomnieć sobie jak się to robi, to mam nadzieję, że ten materiał okaże się pomocny. Jeśli nigdy wcześniej tego nie robiłeś, to czytając go od początku mam nadzieję, że się nauczysz. Ja pisząc go wiele się nauczyłem.

2 Podstawowe informacje o Bashu

2.1 Czym jest Bash

Bash (*ang. Bourne-again shell*) jest powłoką systemową używaną w większości nowoczesnych dystrybucji Linuksa. Powłoka systemowa to program, który pośredniczy w wymianie informacji pomiędzy użytkownikiem, a systemem operacyjnym lub aplikacjami. Wpisując polecenia chcemy, aby system operacyjny zrobił coś dla nas. Po wykonaniu lub nie tejże operacji dostajemy komunikat zwrotny o postępach. Po pomyślnym wykonaniu operacji za pośrednictwem poleceń systemowych z reguły nie dostaje się komunikatu typu „Wszystko się udało", a jedynie znak zachęty zostaje wyświetlony w nowej linii oczekując na wpisanie kolejnego polecenia. Aby zademonstrować omówione zachowanie oraz czym jest znak zachęty spójrzmy na poniższy rysunek (rys. 2.1.1).

Rys. 2.1.1 Bash

Jak widać wykonanie polecenia ls (wyświetlanie zawartości katalogu) udało się, lista plików bieżącego katalogu została wyświetlona oraz znak zachęty przeszedł do nowej linii w oczekiwaniu na kolejne polecenia.

Na znak zachęty (*ang. prompt*) składają się wszystkie te informacje, które umieszczone są przed znakiem **$** (lub #) włącznie z nim. W naszym przypadku wyświetlana jest nazwa zalogowanego użytkownika (gruby), nazwa hosta (earth), ścieżka do katalogu, w którym aktualnie się znajdujemy (~/Desktop/pics), oraz znak **$**, informujący o tym, że gruby pracuje z uprawnieniami użytkownika. Znak # byłby wyświetlany jeśli bylibyśmy zalogowaniu na konto administratora systemu (root)[1]. Tak naprawdę informacje wyświetlane w znaku zachęty mogą być dowolnie zmieniane. Znakiem zachęty może być wyświetlany na przykład sam znak **$**. W dalszej części tej książki w miejscach, w których podawane będą polecenia do wpisania w terminalu zaczynać się one będą właśnie od znaku **$** jeśli polecenie wymagać będzie uprawnień użytkownika i znaku # jeśli do wykonania jakiejś operacji potrzebować będziemy uprawnień administratora systemu.

1 $ - dla użytkownika, # - dla administratora są domyślnymi ustawieniami i mogą być zmienione.

2.2 Co odróżnia programowanie w Bashu od programowania w języku C

Różnica jest zasadnicza, ponieważ każdy program napisany w języku C musi zostać skompilowany, tzn przetłumaczony z kodu zrozumiałego dla ludzi na kod maszynowy danego procesora. Po uruchomieniu pliku wykonywalnego procesor realizuje określone zadanie itp. Każda zmiana w kodzie źródłowym wiąże się z ponowną kompilacją programu. Programowanie w Bashu (oraz również w innych językach interpretowanych) cechuje się tym, że kod źródłowy jest interpretowany na bieżąco, krok po kroku. Jeśli stwierdzimy, że chcemy dodać coś do pliku (jakąś funkcję, lub kolejną pętlę) to wystarczy, że dodamy ją i uruchomimy plik, nie trzeba pliku kompilować. Skrypty pisane w Bashu opierają się na poleceniach systemowych, które możemy uruchomić z konsoli. Dodatkowo podstawowe instrukcje takie jak pętle, instrukcje warunkowe i inne mechanizmy (opisane w dalszej części książki) zawarte są w tej powłoce.

Jedną rzeczą o której trzeba pamiętać w momencie, gdy chcemy uruchomić nasz skrypt to informacja o tym, czy mamy uprawnienia do wykonania pliku. Jeśli mamy to skrypt zostanie wykonany.

2.3 Jakie są korzyści z umiejętności pisania skryptów w Bashu

Korzyści są ogromne, mam nadzieje dostrzeżesz to bardzo szybko, bynajmniej postaram się Tobie to wkrótce pokazać. Jeśli mamy pewne operacje powtarzające się i używamy konsoli do wykonywania tego zadania, to czemu nie ułatwić sobie życia i nie napisać skryptu, który zrobi to za nas? Dajmy na to przykład następujący, w katalogu znajduje się 10000 plików (na przykład tekstowych) z rozszerzeniem. Chcemy aby każdy plik trafił do katalogu o takiej samej nazwie jak nazwa pliku, tylko bez rozszerzenia. Analizując po kolei musimy najpierw zobaczyć jak nazywa się plik, następnie utworzyć katalog o takiej nazwie bez rozszerzenia, a ostatnią czynnością jest przeniesienie danego pliku do tego katalogu. Jak myślisz jak długo robiłbyś to sam? A jak myślisz jak szybko zrobi to komputer za nas? W kolejnych rozdziałach dowiesz się jak to zrobić oraz jak długo trwała ta operacja.

2.4 Pisanie w konsoli, czy tworzenie skryptu w pliku

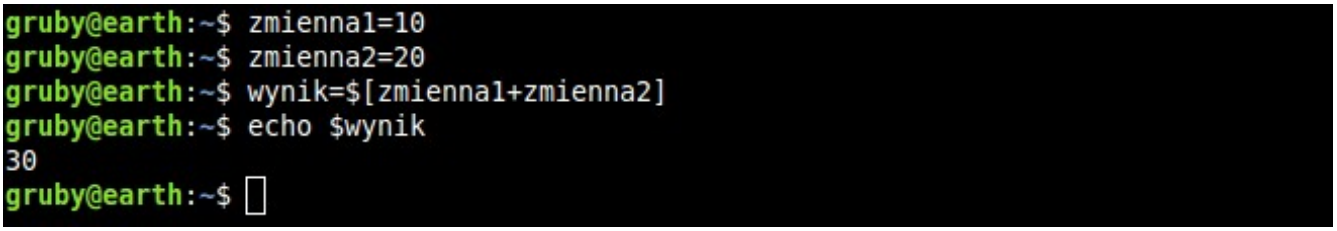
Skrypty możemy pisać w dwojaki sposób. To znaczy składnia i zasady pisania się nie zmieniają, różnica wynika tylko z potrzeby zastosowania danego rozwiązania. Chodzi o to, że wspomniany problem w punkcie poprzednim możemy rozwiązać pisząc dokładnie pięć linijek w konsoli podczas

normalnej pracy, lub wrzucić te pięć linijek do pliku i go uruchomić. Zazwyczaj jednak dłuższe skrypty pisze się w plikach ponieważ w przypadku pomyłki łatwiej jest ją edytować.

W celu pokazania tych sposobów, zostanie pokazany listing z przykładowym skryptem oraz zrzut ekranu z konsoli. Oba skrypty są identyczne.

```
#!/bin/bash

zmienna1=10
zmienna2=20
wynik=$[zmienna1+zmienna2]
echo $wynik
```

Listing 2.4.1 Skrypt pisany w pliku



Rys. 2.4.1 Skrypt pisany w konsoli

3 Podstawy programowania w Bashu

W tym punkcie przedstawione zostaną podstawy pisania skryptów w Linuksie za pomocą Basha. Jak w każdej książce o programowaniu przygodę zaczyna się z nieśmiertelnym „Hello World” tak i w tym miejscu zostanie to pokazane, by w późniejszym czasie pisać już ciekawsze i bardziej praktyczne skrypty.

3.1 Pierwszy skrypt

Na poniższym listingu został przedstawiony pierwszy skrypt wyświetlający informacje dla użytkownika, który powinien w pewnym stopniu przybliżyć sprawę związaną z pisaniem skryptów. W kolejnych rozdziałach będą dodawane kolejne mechanizmy używane w bashu. Tak więc śmiało możesz skopiować poniższy kod, wkleić go do edytora tekstu, zapisać pod dowolną nazwą z rozszerzeniem[1] **.sh**, oraz uruchomić (o uruchomieniu jest punkt 3.2).

```
#!/bin/bash

echo "Hello World"
```

Listing 3.1.1 Pierwszy program – HelloWorld.sh

Pierwsza linia ma szczególnie ważne znaczenie, zaczyna się od znaków **#!** a ścieżka występująca po tych znakach informuje o rodzaju interpretera. W naszym przypadku jest to bash. Jak widać jedynym naszym poleceniem (polecenie systemowe) jest echo, które jako argument przyjmuje nasz powitalny napis. Ogólnie sprawa wygląda tak, że jeśli używamy poleceń systemowych to dokumentacja dostępna jest za pośrednictwem podręcznika systemowego man. Wpisując w konsoli,

```
$ man echo
```

Otrzymamy dostęp do dokumentacji polecenia echo.

3.1.1 Komentarze

Wszystko co znajduje się po znaku # aż do nowego wiersza traktowane jest jako komentarz (poza oczywiście wspomnianym przypadkiem w punkcie powyższym) i omijane przez interpreter. Znak #

1 Rozszerzenia w gruncie rzeczy nie trzeba podawać, aczkolwiek dla ujednolicenia, iż jest to skrypt można.

musi być poprzedzony spacją. Kilka wariantów użycia znaku # zostało przedstawione poniżej.

```
#!/bin/bash

echo "Pewien ciag znakow" # To jest komentarz
echo "Pewien ciag znakow" #To jest komentarz
echo "Pewien ciag znakow"# To nie jest komentarz
echo "Pewien # ciag znakow" # Pomiedzy cudzyslowami sie nie liczy
echo 'Pewien # ciag znakow' # Pomiedzy apostrofami tez sie nie liczy
echo Pewien \# ciag znakow # Poprzedzony \ nie powoduje komentarza
```

Listing 3.1.2 Użycie znaku #

Jak już zauważyłeś, jeśli znak # jest pomiędzy apostrofami, lub cudzysłowami to drukowany jest jako normalny znak – nie powoduje komentarza. Sytuacja jest podobna, jeśli nie używamy wspomnianych znaków, lecz poprzedzimy # znakiem ukośnika. Jeśli pomiędzy # a końcem ciągu znaków nie ma spacji, to znów ten znak drukowany jest jak każdy inny.

3.2 Uruchamianie skryptu

Aby uruchomić skrypt najpierw musimy ustawić (jeśli plik nie ma) możliwość wykonywania. Wykonuje się to za pomocą polecenia chmod. Jeśli nasz plik nazywa się **HelloWorld.sh** to ustawienie uprawnienia do wykonywania odbywa się za pomocą niniejszego polecenia.

```
$ chmod +x HelloWorld.sh
```

Po dodaniu uprawnienia uruchamiania, nasz skrypt wykonuje się następująco.

```
$ ./HelloWorld.sh
```

3.3 Zmienne

Trudno wyobrazić sobie jakikolwiek język programowania bez możliwości używania zmiennych, byłoby to trudnym zadaniem. Dlatego też programując w bashu mamy możliwość operowania na kilku rodzajach zmiennych. W kolejnych podpunktach zostaną wymienione rodzaje zmiennych i pokazany sposób ich używania.

3.3.1 Zmienne programowe

Zmienne programowe, czyli zwykłe zmienne za pomocą których będziemy mogli przechowywać cyfry, znaki, ciągi znaków itp. deklaruje się w bardzo prosty sposób, a mianowicie podajemy nazwę zmiennej po której następuje (bez spacji) znak równości, a po znaku równości (również bez spacji) zawartość zmiennej. Przykładowy listing z użyciem zmiennych lokalnych znajduje się poniżej.

```
#!/bin/bash

wiek=22
powitanie="Witaj"
imie=Artur

echo -n $powitanie $imie
echo -n ". Ilosc Twoich lat to: "
echo $wiek

echo powitanie
echo wiek
```

Listing 3.3.1 Użycie zmiennych

Ciągu znaków nie trzeba ujmować w znaki cudzysłowu (jeśli jest to jeden wyraz), aczkolwiek dla uwypuklenia, iż jest to tekst, oraz dla edytora tekstu (np. VIM), który koloruje składnie będzie to oznaka, że znaki te należy pokolorować. Opcja **-n** polecenia echo użyta jest w celu, by kursor nie przeszedł do nowej linii. O ważnej rzeczy nie wspomniano jeszcze, a mianowicie użycie zmiennych. Przed każdą zmienną, której zawartość chcemy wyświetlić dodajemy znak **$**, wtedy odwołujemy się do wartości. Jeśli tego znaku nie ma, co zresztą już pewnie zauważyłeś, drukowana jest nazwa zmiennej. Wyjątkiem od tej reguły jest sposób użycia zmiennych w pętli (opisany w dalszej części).

Oczywiście wszystko można wpisać w jednej linii jako argument polecenia echo. Tak więc poniższy listing pokazuje, że zostaną wydrukowane te same informacje.

```
#!/bin/bash

wiek=22
powitanie="Witaj"
imie=Artur

echo "$powitanie $imie. Ilosc Twoich lat to: $wiek"
echo '$wiek'
```

Listing 3.3.2 Drukowanie informacji w jednym poleceniu

Dodatkowo dostawiono jedną linijkę, która pokazuje co się dzieje w momencie, gdy próbujemy wydrukować zmienną ujętą w apostrofy. Polecenie echo użyte wraz z apostrofami cytuje dokładną treść, nie podstawia wartości zmiennych, należy o tym pamiętać.

Aby sprawdzić dlaczego znak równości nie może być otoczony spacjami dodaj je i spróbuj uruchomić skrypt. Jeśli pomiędzy nazwą zmiennej, a znakiem równości wstawimy spację otrzymamy następujący komunikat,

```
./3.sh: line 3: wiek: nie znaleziono polecenia
```

./3.sh oznacza nazwę pliku. Następnie mamy numer linii, a w kolejnej części informację o błędzie. W tym przypadku interpreter próbuje odwołać się do polecenia wiek, którego nie ma w systemie. Jeśli natomiast wpiszemy spację pomiędzy znakiem równości, a wartością zmiennej, to otrzymamy następujący komunikat,

```
./3.sh: line 4: Witaj: nie znaleziono polecenia
```

czyli wartość została potraktowana jako polecenie. Oczywiście ma to swoje dobre strony, ponieważ jeśli potrzebujemy wyświetlić na przykład bieżący katalog, ale chcemy, aby ścieżka została zapisana w zmiennej, to możemy to zrobić właśnie w taki sposób jak wyżej opisano, ale można zrobić to w inny, bardziej spójny sposób. Sposoby te przedstawiono na poniższym listingu.

```
#!/bin/bash

biezacy_katalog= pwd
biezacy_katalog2=`pwd`
biezacy_katalog3=$(pwd)

echo $biezacy_katalog
echo $biezacy_katalog2
echo $biezacy_katalog3
echo `pwd`
echo $(pwd)
```

Listing 3.3.3 Drukowanie ścieżki do bieżącego katalogu

Pomiędzy wydrukiem pierwszego, a drugiego następuje spacja, dlatego też lepiej używać drugiej formy przypisywania poleceń do zmiennych. Istnieje jeszcze inny sposób przypisywania wyniku polecenia do zmiennej i jest on pokazany w kolejnej linii, oraz jest tak samo traktowany jak polecenie powyższe. Dwa ostatnie wywołania polecenia echo pokazują, jak można wydrukować wynik polecenia bez

użycia zmiennych.

Porównując listing 3.3.1 z listingiem 3.3.2 nie dostrzeżemy różnicy w wydruku, lecz poniższy listing pokazuje pewną różnicę, o której warto wiedzieć.

```
#!/bin/bash

cyfry="1   2   3   4   5         6

7
8"

echo $cyfry
echo "$cyfry"

echo ${cyfry}
echo "${cyfry}"
```

Listing 3.3.4 Pewne różnice w wydruku

Jeżeli zmienna zawiera napis z białymi znakami (spacje, tabulacje, znaki nowej linii) to w przypadku użycia polecenia echo bez cudzysłowu każda nadmiarowa ilość białych znaków zostanie zredukowana do pojedynczej spacji. Jeśli użyjemy cudzysłowu to białe znaki są drukowane tak jak zostały wpisane. Ostatnie dwa sposoby drukowania są analogiczne do przedstawionych na początku.

O jeszcze jednej rzeczy należy powiedzieć, a mianowicie nazewnictwo zmiennych. Nazwa zmiennej może składać się z liter, cyfr oraz znaku podkreślenia z tym, że cyfra nie może być pierwszym znakiem, podkreślenie może być. Wielkość liter jest rozpoznawana, dlatego tez zmienne wiek i Wiek są dwiema różnymi zmiennymi.

Wszystkie do tej pory przedstawione sposoby deklarowania zmiennych obejmowały zmienne lokalne. Co to znaczy? Jeśli deklarujemy zmienne za pomocą konsoli i tam wykonujemy operacje (jak pokazano na rys. 2.4.1), to tylko w obrębie danej konsoli jesteśmy w stanie używać tych zmiennych. Jeśli chcemy aby nasze zmienne widziane były w innych konsolach również, musimy przed nazwą zmiennej dodać słowo kluczowe export.

Dlaczego do tej pory nie wspomniano nic o typach zmiennych? Ponieważ w bashu nie jest to wymagane. Wszystko zależy od kontekstu. Z grubsza to zawartość zmiennej zawsze jest ciągiem znaków, a jeśli chodzi o wykonywanie operacji arytmetycznych, to interpreter przekształca dany ciąg znaków do liczby, aczkolwiek ciągiem znaków muszą być same cyfry (bardziej szczegółowo

omówione w dalszej części).

Aby jednak zadeklarować zmienną o konkretnym typie używamy polecenia declare z odpowiednią opcją. Opcje te zostały przedstawione w poniższej tabeli, a opis i sposób użycia pod tabelką.

Typ	Opcja
Tylko do odczytu	r
Liczba całkowita	i
Tablica	a
Funkcja	f

Tabela 3.1 Typy zmiennych

```
#!/bin/bash

function fun1
{
echo "Czesc"
}

declare -r var1=10
echo $var1
var1=100
echo $var1

declare -i l1=5
declare -i l2=4
declare -i wynik
wynik=l1+l2
echo $wynik

declare -a tab=(1 3 4)
echo ${tab[0]} ${tab[1]} ${tab[2]}

declare -f fun1
```

Listing 3.3.5 Typy zmiennych

Operacje na liczbach, tablicach oraz używanie funkcji zostanie omówione później, dlatego tutaj nie będę tego opisywać. Jak widać użycie zmiennej tylko do odczytu powoduje błąd w momencie, gdy chcemy zmienić jej zawartość. Opcja **f** nie jest typem, tylko wyświetla ciało funkcji podanej jako argument (fun1), lub ciała wszystkich funkcji, jeśli po tym parametrze nie występuje żadna nazwa funkcji.

3.3.2 Zmienne specjalne

W tym miejscu zajmiemy się zmiennymi specjalnymi, czyli takimi, które w większości przypadków użytkownik może tylko odczytywać i są one przeznaczone do konkretnych z góry ustalonych celów. W poniższej tabeli wypisane są zmienne związane z uruchomieniem skryptu, oraz ich przeznaczenie.

Nazwa zmiennej	Wyświetlanie
0	Nazwa skryptu
1 … 9	Nazwy argumentów
@	Wszystkie argumenty
?	Zwrócona wartość przez ostatnie polecenie
$	PID bieżącej powłoki
#	Ilość argumentów

Tabela 3.2 Zmienne specjalne związane z uruchomieniem programu

```
#!/bin/bash

echo "Nazwa skryptu to: $0"
echo "Pierwszy argument: $1"
echo "Drugi argument: $2"
echo "Wszystkie argumenty: $@"
echo "Zwrocona wartosc przez program echo: $?"
echo "PID biezacej powloki: $$"
echo "Ilosc argumentow: $#"
```

Listing 3.3.6 Użycie zmiennych specjalnych

Przykładowe wywołanie powyższego skryptu wygląda następująco.

```
$ ./1.sh Arg1 Arg2 3 45 Ostatni
```

3.3.3 Zmienne środowiskowe

Zmienna środowiskowe to takie zmienne, które wykorzystywane są przez system do różnego rodzaju operacji. Na przykład zmienną środowiskową może być ścieżka dostępu do katalogu domowego

zapisana pod odpowiednią nazwą, lub ścieżka do poprzedniego katalogu (w którym byliśmy tuż przed tym katalogiem, w którym się aktualnie znajdujemy). W poniższej tabeli znajdują się najczęściej wykorzystywane zmienne środowiskowe, a pod tabelą przykładowy skrypt.

Nazwa zmiennej	Opis
HOME	Ścieżka do katalogu domowego
USER	Nazwa zalogowanego użytkownika
PATH	Ścieżki do katalogu, w którym znajdują się pliki binarne
TERM	Nazwa używanego emulatora konsoli
PWD	Ścieżka do bieżącego katalogu
OLDPWD	Ścieżka do poprzedniego katalogu

Tabela 3.3 Zmienne środowiskowe

```
#!/bin/bash
cd ~
cd $OLDPWD
echo "Sciezka do Twojego katalogu domowego to: $HOME"
echo "Jestes zalogowany jako: $USER"
echo "Lista katalogow z plikami wykonywalnymi: $PATH"
echo "Uzywany emulator terminala: $TERM"
echo "Znajdujesz sie w katalogu: $PWD"
echo "Poprzedni katalog, w ktorym byles to: $OLDPWD"
```

Listing 3.3.7 Użycie zmiennych środowiskowych

W pierwszej linii naszego skryptu przechodzimy do katalogu domowego, następnie przechodzimy do katalogu, w którym byliśmy przed chwilą. Gdyby nie ta operacja, to zmienna OLDPWD byłaby pusta, a wynik ostatniego polecenia echo nie byłby zadowalający.

Aby wyświetlić wszystkie występujące w systemie zmienne środowiskowe możemy skorzystać polecenia systemowego printenv, którego wynik przekażemy do programu more, w celu lepszego odczytu danych (aby wyjść z programu more wciśnij **q**).

```
$ printenv | more
```

3.3.4 Tablice

Czym jest tablica? Tablica jest to zmienna, która przechowuje różne wartości pod tą samą nazwą, lecz aby się odwołać do konkretnej informacji musimy podać odpowiednią wartość indeksu. W odróżnieniu od języka C, w bashu wartości w tablicy mogą być różnego typu (czyli możemy wpisać zarówno cyfry jak i ciągi znaków). Elementy tablicy indeksowane są od 0. Tablicę deklaruje się tak, że po nazwie zmiennej stawiamy znak równości, a po znaku równości w nawiasach okrągłych podajemy elementy tablicy rozdzielone spacjami. Przykład użycia tablicy znajduje się na poniższym listingu.

```
#!/bin/bash

tablica=(1 3 9 "Ciag znakow" Ciag znakow)

echo ${tablica[0]}
echo ${tablica[1]}
echo ${tablica[2]}
echo ${tablica[3]}
echo ${tablica[4]}
echo ${tablica[5]}

tablica[0]=99
echo ${tablica[0]}
```

Listing 3.4.1 Użycie tablicy

Nic nie stoi na przeszkodzie, by ciągi znaków wpisywać bez cudzysłowu, aczkolwiek trzeba mieć na uwadze, że każdy wyraz jest kolejnym elementem tablicy, co nie dzieje się w przypadku użycia cudzysłowu. Do elementów tablicy odwołujemy się tak jak pokazano, czyli po znaku **$** w nawiasach klamrowych podajemy nazwę zmiennej tablicowej oraz jej indeks podany w nawiasach kwadratowych tuż po jej nazwie. Zmieniać wartości elementów tablicy też można za pomocą pokazanego sposobu, czyli wpisujemy nazwę tablicy, jej indeks, a po znaku równości wartość traktując to jako zwykłą zmienną.

Używając tablic możemy skorzystać z pewnych ułatwień. Aby wydrukować wszystkie elementy tablicy, lecz inaczej (łatwiej) niż w poprzednim przypadku, możemy to zrobić podając zamiast numeru indeksu znak *****, lub **@**.

```
#!/bin/bash

tablica=(1 3 9 "Ciag znakow" Ciag znakow)
```

```
echo ${tablica[*]}
echo ${tablica[@]}
```

Listing 3.4.2 Drukowanie wszystkich elementów tablicy w inny sposób

Możemy wydrukować ilość znaków poszczególnych elementów tablicy, jak również możemy dowiedzieć się ile elementów tablica posiada, oba sposoby pokazane zostały poniżej. W pierwszym wywołaniu echo wypisujemy ile znaków ma element znajdujący się pod indeksem trzecim. W kolejnym drukujemy ile elementów ma tablica.

```
#!/bin/bash

tablica=(1 3 9 "Ciag znakow" Ciag znakow)

echo ${#tablica[3]}

echo ${#tablica[@]}
```

Listing 3.4.3 Drukowanie ilości znaków elementów, oraz liczby elementów

3.4 Działania matematyczne

W tym miejscu skupimy się na podstawowych działaniach wykonywanych na zmiennych, tzn dodawanie wartości do zmiennej, odejmowanie od niej itp. Intuicyjny sposób dodawania do zmiennych wartości niestety nie działa tak jakbyśmy tego chcieli, bo de facto wartość do zmiennej zostanie dodana. Na poniższym przykładzie pokazane jest to o czym wspomniałem.

```
#!/bin/bash

a=1
a=a+1

echo $a
```

Listing 3.4.1 Dodawanie wartości

Po wykonaniu tej operacji w zmiennej a będzie wartość a+1, a nie 2, jakby można było pomyśleć. Aby wykonywać operacje matematyczne, należy podejść do tego inaczej. Sposoby są trzy, który wybierzesz zależy od Ciebie. Opis podany jest pod listingiem.

```
#!/bin/bash
a=1
a=$(( a+1 ))
echo $a

a=$[a+10]
echo $a

let a=a-100
echo $a
```

Listing 3.4.2 Poprawnie dodane wartości liczbowe

Pierwszy wymieniony na listingu sposób to użycie podwójnej pary nawiasów okrągłych poprzedzonych znakiem **$**, a w środku operacja, którą chcemy wykonać (nie koniecznie między operacją, a nawiasami musi występować spacja). Zauważ, że pomiędzy nawiasami zmienna a nie jest poprzedzona znakiem **$**, lecz może być. Drugi sposób to użycie operacji matematycznej pomiędzy nawiasami kwadratowymi poprzedzonymi znakiem **$**. Trzeci sposób to użycie instrukcji let, po której następuje nazwa zmiennej, a po niej znak równości i operacja matematyczna.

Operacje wykonywane są na liczbach całkowitych, przez co na poniższym przykładzie wynik dzielenia liczby 1 przez 2 daje 0. Gdy wykonywane jest dzielenie przez 0, dostajemy informację o błędzie.

```
#!/bin/bash

a=1
b=2

echo $(($a+$b))
echo $[$a/b]
wynik=$[b*$b+2*a-8*b]
echo $wynik
echo $(($a + 3))
```

Listing 3.4.3 Operacje matematyczne

Na listingu 3.4.2 pokazano sposób przypisywania wartości za pomocą let, w tym miejscu pokazane zostanie jak bardziej zwięźle napisać przypisanie analogiczne do tamtego.

```
#!/bin/bash

a=10
let a-=100
```

```
echo $a
```

Listing 3.4.4 Inny operator matematyczny

Zapis **a-=100** jest analogiczny z zapisem **a=a-100**, lecz bardziej zwięzły. Zamiast minusa można wstawić każdy inny operator matematyczny używany w bashu, a ponieważ ich do tej pory nie wymieniono, zostaną one przedstawione w poniższej tabelce.

Operator	Działanie matematyczne
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
**	Potęgowanie (operand z prawej strony jest wykładnikiem)

Tabela 3.4.1 Operatory matematyczne

4 Instrukcje warunkowe

Instrukcje warunkowe używane w językach programowania są stosowane w momencie, gdy mamy do wyboru więcej niż jedną drogę, którą program może podążać. W zależności od danego problemu raz lepszym zastosowaniem może być instrukcja if, innym razem case. Obie przedstawione zostały w poniższych punktach.

4.1 Instrukcja if

Instrukcja if jest wykorzystywana, gdy do sprawdzenia mamy pojedynczy warunek, np. czy dany plik istnieje, czy jakaś wartość jest większa od innej i tym podobne rzeczy. Jeśli chcemy to możemy rozbudować instrukcję if o kolejne warunki do sprawdzenia. W tabeli 4.1.1 znajdują się operatory relacji, a w tabeli 4.1.2 inne przydatne operatory do pracy skryptu.

Operator	**Zastosowanie**
-lt	Mniejszy niż
-le	Mniejszy, lub równy
-gt	Większy niż
-ge	Większy lub równy
=	Równy
!=	Różny

Tabela 4.1.1 Operatory relacji

Operator	**Zastosowanie, czy**
-a	Plik istnieje
-b	Plik istnieje i jest blokowym plikiem specjalnym
-	Plik istnieje i jest plikiem znakowym
-e	Plik istnieje
-h	Plik istnieje i jest linkiem symbolicznym
-n	Wyrażenie ma długość większą niż 0
-d	Wyrażenie istnieje i jest katalogiem
-z	Wyrażenie ma zerową długość

-r	Można czytać plik
-w	Można zapisywać do pliku
-x	Można plik wykonać
-f	Plik istnieje i jest plikiem zwykłym
-p	Plik jest łączem nazwanym
-N	Plik istnieje i był zmieniany od czasu jego ostatniego użycia
P1 -nt P2	P1 jest nowszy od P2
P1 -ot P2	P1 jest starszy od P2

Tabela 4.1.2 Inne przydatne operatory

W gruncie rzeczy użycie instrukcji if jest proste, podajemy warunek po słowie kluczowym if. Jeśli warunek jest spełniony to zostają wykonane instrukcje po słowie kluczowym then. Jeśli warunek nie jest spełniony i nie ma słowa kluczowego else, to skrypt przechodzi do dalszych instrukcji. Jeśli występuje instrukcja else, to zostają wykonane instrukcje zawarte po tym słowie. Na poniższych listingach zostało pokazanych parę sposobów użycia instrukcji if.

```
#!/bin/bash

if [ "$#" != 1 ]
then
     echo "Podales zla ilosc argumentow."
     echo "Wywolanie: $0 nazwa_pliku"
     exit 1
fi

if [ -e "$1" ]
then
     echo "Plik istnieje"
else
     echo "Plik nie istnieje"
fi
```

Listing 4.1.1 Nazwa pliku jako argument wywołania

Na powyższym listingu mamy warunek, który sprawdza, czy ilość argumentów wywołania skryptu jest różna od 1. Jeśli tak jest to użytkownik dostaje informacje o tym, że źle użył programu, po czym wyświetla się informacja o sposobie użycia oraz skrypt jest kończony z wartością 1 za pomocą dotąd nie omówionej funkcji exit. Funkcja exit kończy działanie skryptu i zwraca wartość podaną jako jej

argument. W konsoli z której wywołujemy nasz skrypt po „złym” wywołaniu możemy sprawdzić jaka wartość kryje się pod zmienną specjalną **?** (Opisaną w punkcie 3.3.2). Wartością tą będzie 1.

Jeśli natomiast podamy argument (nazwę pliku) to pierwszy warunek jest fałszywy, ponieważ ilość argumentów wywołania skryptu równa się 1, dlatego też omijane są instrukcje pomiędzy then a fi. Kolejny warunek sprawdza czy dany plik istnieje i w zależności od tego czy znajduje się on na dysku czy nie dostajemy stosowaną informację.

```
#!/bin/bash

if [ "$#" -gt 2 ]
then
     echo "Za duzo argumentow"
     echo "Wywolanie: $0 nazwa_pliku [nazwa_katalogu]"
     exit 1
fi

if [ "$#" = "0" ]
then
     echo "Nie podales nazwy pliku"
     echo "Wywolanie: $0 nazwa_pliku [nazwa_katalogu]"
     exit 2
fi

if [ -n "$2" ]
then
     mkdir "$2"
     cp "$1" "$2"/
else
     mkdir "default"
     cp "$1" default/
fi

if [ "$?" = "0" ]
then
     echo "Skopiowano"
else
     echo "Nie skopiowano"
fi
```

Listing 4.1.2 Kopiowanie pliku do katalogu

W tym skrypcie na początku sprawdzamy czy ilość argumentów wywołania skryptu jest większa niż 2. Jeśli tak jest, to tak jak w poprzednim przypadku stosowna informacja jest wyświetlana, oraz skrypt jest przerywany. Kolejny warunek sprawdza czy podany jest chociaż jeden argument (nazwa pliku).

Jeśli nie jest to i tym razem wyświetlamy komunikat i kończymy skrypt. W następnej kolejności sprawdzamy, czy drugi argument ma długość większą niż zero. Jeśli tak jest to możemy utworzyć katalog o podanej nazwie oraz skopiować plik podany jako pierwszy argument do tego katalogu. W przeciwnym przypadku (jeśli długość drugiego argumentu jest zerowa – nie podaliśmy drugiego argumentu) tworzymy katalog o nazwie **default** i do niego kopiujemy plik. Ostatni warunek sprawdza czy ostatnie polecenie (niezależnie od tego czy podaliśmy nazwę katalogu czy nie, ostatnim poleceniem jest cp) zwróciło wartość 0 (sukces).

Skrypt ten nie jest odporny na drugie takie samo wywołanie, tzn w przypadku, gdy będzie już taki katalog utworzony, nie może on zostać utworzony drugi raz, co skutkuje wyświetleniem błędu. Ale przecież możemy się zabezpieczyć przed tym w bardzo prosty sposób.

```
#!/bin/bash

if [ "$#" -gt 2 ]
then
     echo "Za duzo argumentow"
     echo "Wywolanie: $0 nazwa_pliku [nazwa_katalogu]"
     exit 1
fi

if [ "$#" = "0" ]
then
     echo "Nie podales nazwy pliku"
     echo "Wywolanie: $0 nazwa_pliku [nazwa_katalogu]"
     exit 2
fi

if [ ! -e "$1" ]
then
     echo "Taki plik nie istnieje"
     exit 3
fi

if [ -n "$2" ]
then
     if [ ! -d "$2" ]
     then
          mkdir "$2"
     fi
     cp "$1" "$2"/
else
     if [ ! -d "default" ]
     then
          mkdir "default"
     fi
```

```
     cp "$1" default/
fi

if [ "$?" = "0" ]
then
     echo "Skopiowano"
else
     echo "Nie skopiowano"
fi
```

Listing 4.1.3 Lekka modyfikacja listingu 4.1.2

Na tym listingu występują dwie modyfikacje w porównaniu z poprzednim przykładem. Pierwsza z nich to dodanie zabezpieczenia przed próbą skopiowania pliku, którego nie ma. Skrypt w takim przypadku kończy swoje działanie. Drugą modyfikacją jest dodanie wewnętrznych instrukcji warunkowych, które sprawdzają, czy katalog istnieje. Pierwszy, tuż po then sprawdza czy katalog (nazwa przekazana jako argument) nie istnieje. Opcja **-d** sprawdza czy istnieje, ale wykrzyknik przed neguje działanie. Jeśli nie istnieje to go utwórz, a jeśli istnieje to tylko skopiuj. Analogicznie jest w następnym przypadku, z tym, że tu na sztywno wpisana jest nazwa katalogu.

Za pomocą instrukcji if można sprawdzać wiele warunków. Działa to na takiej zasadzie, że najpierw sprawdzany jest pierwszy, tuż po słowie kluczowym if, a następne wyszczególnione są po słowie kluczowym elif. Jeśli pierwszy nie został spełniony to sterowanie przechodzi do kolejnego. Jeśli żaden nie zostanie spełniony i została zastosowana instrukcja else, to związane z nią akcje zostaną wykonanie, jeśli nie to skrypt przejdzie do kolejnej linii.

```
#!/bin/bash

ilosc_arg="$#"

if [ $ilosc_arg = "0" ]
then
     echo "Nie podales argumentow"
elif [ $ilosc_arg = "1" ]
then
     echo "Podales jeden argument: $1"
elif [ $ilosc_arg = "2" ]
then
     echo "Podales dwa argumenty: $1 $2"
elif [ $ilosc_arg = "3" ]
then
```

```
     echo "Podales trzy argumenty: $1 $2 $3"
else
     echo "Podales wieksza liczbe argumentow"
fi
```

Listing 4.1.4 Przykładowe zastosowanie elif

Przydatne podczas pisania skryptów mogą okazać się operatory logiczne, które zostały zamieszczone w niniejszej tabeli. Z ich pomocą możemy utworzyć dłuższe wyrażenie do sprawdzenia, które bez ich użycia wymagało by zagnieżdżenia kilku instrukcji if.

Operator	Zastosowanie
!	Negacja
||	Logiczne „lub” (OR)
&&	Logiczne „i” (AND)

Tabela 4.1.3 Operatory logiczne

```
#!/bin/bash

numer=10
if [ "$numer" -ge "0" ] && [ "$numer" -le "10" ]
then
     echo "$numer jest z przedzialu: <0; 10>"
else
     echo "$numer nie jest z przedzialu: <0; 10>"
fi

if [ "$numer" -lt "10" ] || [ "$numer" -gt "20" ]
then
     echo "$numer nie jest z przedzialu: <10; 20>"
else
     echo "$numer jest z przedzialu: <10; 20>"
fi
```

Listing 4.1.5 Przykład użycia operatorów logicznych

Zastosowanie operatora negacji zostało pokazane na listingu 4.1.4. W pierwszej kolejności używamy logicznego „i” i sprawdzamy warunek, czy wartość zmiennej numer jest większa lub równa zero, jeśli tak jest, to sprawdzamy drugi warunek, czy liczba jest mniejsza lub równa 10. Jeśli oba warunki są spełnione, to drukowany jest komunikat, że liczba zawiera się w podanym przedziale. Jeśli choćby

jeden jest niespełniony to zostaną wykonane akcje po słowie else. Druga część kodu to operator logiczny „lub”. W tym przypadku wystarczy, że jeden z warunków będzie spełniony i całe wyrażenie jest prawdziwe. W naszym przypadku jeśli liczba będzie mniejsza od 10, lub większa od 20 to dostaniemy informacje, że liczba nie jest z przedziału <10; 20>. Jeśli oba warunku są fałszywe, to liczba jest z danego przedziału i taki komunikat otrzymamy.

4.2 Instrukcja case

Instrukcja case używana jest wtedy, gdy mamy kilka wzorców i chcemy wybrać drogę którą skrypt ma podążać w zależności od tego wzorca. Chyba najczęściej spotykanym przykładem instrukcji case jest drukowanie nazwy miesiąca w zależności od wpisanej liczby. Niech i tym razem ten przykład posłuży nam za wzór.

```
#!/bin/bash

echo -n "Podaj liczbe <1;12>: "
read miesiac

case "$miesiac" in
     "1") echo "Styczen" ;;
     "2") echo "Luty" ;;
     "3") echo "Marzec" ;;
     "4") echo "Kwiecien" ;;
     "5") echo "Maj" ;;
     "6") echo "Czerwiec" ;;
     "7") echo "Lipiec" ;;
     "8") echo "Sierpien" ;;
     "9") echo "Wrzesien" ;;
     "10") echo "Pazdziernik" ;;
     "11") echo "Listopad" ;;
     "12") echo "Grudzien" ;;
     *) echo "Zla liczba"
esac
```

Listing 4.2.1 Zastosowanie instrukcji case

O poleceniu read jeszcze nie mówiliśmy, dlatego w punkcie 4.3 zostanie ono omówiona. Przede wszystkim po słowie kluczowym case wpisujemy zmienną, która przechowuje jakąś wartość. Następnie ta wartość jest porównywana z wzorcami, które wypisane są poniżej. Jeśli wartość kryjąca się pod zmienną miesiac pasuje do określonego wzorca, to akcja związana z tym wzorcem jest

wykonywana. Jeśli żaden wzorzec nie pasuje i jest zdefiniowany wzorzec domyślny („*)”) to te akcje zostaną wykonane, w przeciwnym razie skrypt przechodzi do wykonywania dalszych instrukcji.

4.3 Polecenie read

Do tej pory tylko w punkcie 4.2 użyto tego polecenia, a jednak z punktu widzenia pisania użytecznych skryptów jest ono ważne, ponieważ z jego pomocą możemy pobierać dane od użytkownika. Użycie jest bardzo proste i w sumie pokazane zostało na listingu 4.3.1, aczkolwiek dla ścisłości pokaże tutaj kilka wariantów użycia tego polecenia. Niech posłuży nam do tego poniższy listing.

```
#!/bin/bash

echo -n "Wpisz tekst: "
read tekst
echo "$tekst"

read -p "Wpisz tekst: " tekst
echo "$tekst"

read -p "Wpisz elementy tablicy: " -a tablica
echo ${tablica[@]}

read -p "Wpisz tekst: "
echo "$REPLY"
```

Listing 4.3.1 Polecenie read – różne opcje

Samo wywołanie polecenia read stwarza możliwość wpisania informacji przez użytkownika, ale żaden tekst zachęcający nie jest wydrukowany, dlatego też jeśli nie używamy żadnej opcji polecenia read, to musimy najpierw wydrukować jakiś tekst za pomocą echo. Jeśli użyjemy opcji **-p** to możemy spokojnie pominąć echo, a tekst zachęcający wpisać po tejże opcji. Można by było wpisywać wartości do tablicy po kolei, traktując elementy tablicy jako normalne zmienne, ale jest łatwiejszy sposób, opcja **-a**, po której następuje nazwa tablicy. Każdy wyraz (liczba) zostanie wpisany jako kolejny element tablicy. Jeśli nie podamy nazwy zmiennej do której ma zostać zapisana pobrana wartość, to wartość ta zostanie umieszczona w specjalnej zmiennej REPLY.

5 Pętle

Pętli używa się w momencie, gdy do zrobienia mamy kilka powtarzających się czynności. Bez sensu jest wpisywanie 10 poleceń read do pobrania danych, skoro możemy zrobić to za pomocą pętli. Po pierwsze nie namęczymy się tyle, po drugie kod jest bardziej czytelny i jasny dla innych, którzy mogą go w późniejszym czasie edytować. Nie mniej jednak pętli do wyboru mamy cztery, w zależności od problemu wybieramy tę, która lepiej pasuje.

5.1 Pętla for

Pętla for trochę w składni różni się od tej używanej w języku C, aczkolwiek cechy ma podobne. To znaczy podajemy zmienną, która przyjmuje wartości z określonej listy. Kilka poniższych przykładów powinno przybliżyć sprawę związaną z używaniem tej pętli.

```
#!/bin/bash
tablica=(1 9 2 3 11 23 0)
for i in 0 1 2 3 4 5 6
do
     echo ${tablica[$i]}
done

for i in {0..6}
do
     echo ${tablica[$i]}
done
```

Listing 5.1.1 Drukowanie elementów tablicy – nie najlepsze

Zmienna i jest zmienną, która przyjmuje wartości z listy (0 1 … 6) w każdej następnej iteracji kolejną, zaczynając od pierwszego elementu, który w naszym przypadku jest równy 0. Następnie drukowane są elementy tablicy, ponieważ naszą listą są indeksy. Analogiczny sposób, tylko nie wypisując całej listy, pokazany jest w drugiej pętli. W tym przypadku lista to pierwszy element, oraz ostatni, pomiędzy nimi dwie kropki, a całość ujęta w nawiasy klamrowe. Elementy listy zwiększane są o 1 w tym przypadku.

Skoro możemy wykorzystać wbudowany mechanizm do zwrócenia ilości elementów tablicy to możemy go wykorzystać, tylko jak zamiast listy wartości w tak podany sposób wpisać listę indeksów nie znając jej długości? Z pomocą przychodzi polecenie seq, które wyświetla wszystkie liczby

z podanego zakresu.

```
#!/bin/bash

tablica=(1 9 2 3 11 23 0)
rozmiar=${#tablica[@]}

for i in `seq 0 $[rozmiar-1]`
do
     echo ${tablica[$i]}
done
```

Listing 5.1.2 Wyświetlenie elementów tablicy z użyciem seq

Polecenie seq w miejscu wywołania wstawi wartości od 0 do rozmiar-1, ponieważ elementy tablicy indeksowane są od 0, to ostatni element kryje się właśnie pod wartością o jeden mniejszą niż liczba elementów tablicy. Innym zastosowaniem pętli for może być wyszukanie największej/najmniejszej wartości występującej w tablicy, rzućmy okiem na poniższy przykład.

```
#!/bin/bash

tablica=(1 9 2 3 11 23 0 -3 31 11 93 223 -1312)
rozmiar=${#tablica[@]}

min=${tablica[0]}
max=${tablica[0]}

for i in `seq 0 $[rozmiar-1]`
do
     if [ ${tablica[$i]} -lt $min ]
     then
          min=${tablica[$i]}
     fi
     if [ ${tablica[$i]} -ge $max ]
     then
          max=${tablica[$i]}
     fi
done

echo $min
echo $max
```

Listing 5.1.3 Wyszukiwanie największej oraz najmniejszej wartości

Przed pętlą przypisujemy w ciemno do zmiennej max i min pierwszy element tablicy, po czym w pętli poszukujemy wartości większych i mniejszych. Jeśli takie istnieją to przypisujemy nową wartość pod

konkretną zmienną i szukamy dalej.

Do tej pory tak naprawdę nie wykorzystaliśmy pętli for w celu operowania na plikach, a tak naprawdę podczas pisania skryptów to jest istotne, dlatego też na niniejszych listingach zostanie pokazane kilka praktycznych sposobów wykorzystania pętli for. Na pierwszy ogień pójdzie problem przedstawiony na samym początku książki, w punkcie 2.3. Aby za symulować tę sytuację musimy najpierw utworzyć te 10000 plików, jak się pewnie domyślasz zrobimy to za pomocą pętli.

```
#!/bin/bash

for i in `seq -w 1 10000`
do
     touch "plik$i.txt"
done
```

Listing 5.1.4 Tworzenie 10000 plików

To może zająć trochę czasu, w zależności od szybkości komputera. Utworzyliśmy pliki, o nazwach **plikN.txt**, gdzie **N** jest liczbą od 1 do 10000, poza ostatnim plikiem, każdy plik ma w nazwie liczbę poprzedzoną zerem, za co odpowiada opcja **-w** polecenia seq.

No dobra, to jeśli już mamy tak ogromną liczbę plików, to teraz chcemy, aby każdy plik znalazł się w katalogu o nazwie pliku, bez rozszerzenia.

```
#!/bin/bash

for i in `ls *.txt`
do
     tmp=`basename $i .txt`
     mkdir $tmp
     mv $i $tmp/
done
```

Listing 5.1.5 Przenoszenie plików do katalogów

W pętli for listą będą nazwy wszystkich plików z rozszerzeniem **txt**. W każdej iteracji do zmiennej tmp przypisujemy nazwę bazową (polecenie basename) z aktualnego pliku. Następnie tworzymy katalog o tej nazwie i przenosimy plik do tego katalogu. Widać, że proste. Czas wykonywania tego skryptu na moim komputerze wyniósł jedną minutę.

Chcąc zamienić wielkość liter w nazwach plików z małych na wielkie możemy postąpić analogicznie, używając znów pętli for.

```
#!/bin/bash

ext=$1

for filename in `ls *.$ext`
do
     tmp=`basename $filename $ext`
     newname=`echo $tmp | tr '[a-z]' '[A-Z]'`
     mv $filename $newname$ext
done
```

Listing 5.1.6 Zamiana wielkości znaków w pliku

W tak użytej wersji skryptu nie jest on odporny na drugie wywołanie, tzn jeśli nazwy plików zostały już zamienione na wielkie litery i spróbujemy zrobić to jeszcze raz, to podczas wykonywania polecenia mv dostaniemy informację o tym, że nie można zmienić nazwy pliku na taką samą. Analogiczną informację dostaniemy, jeśli plik będzie składał się tylko z cyfr. Skrypt działa na takiej zasadzie, że wywołując go podajemy jako pierwszy argument rozszerzenie plików, których litery w nazwach mają zostać zamienione z małych na wielkie. W pętli for do zmiennej filename przypisujemy listę plików, którą otrzymujemy z wyniku polecenia ls okrojoną tylko do listy plików o podanym rozszerzeniu. Do zmiennej tmp przypisujemy wywołanie polecenia basename, które z nazwy pliku kryjącego się pod zmienną filename odrzuca rozszerzenie podane jako drugi argument (ext). Kolejna linia jest ciekawsza, bowiem użyliśmy potoku (mechanizm komunikacji między procesowej). W naszym przypadku przekazujemy wynik polecenia echo do polecenia tr, które zamienia wielkość znaków. Po dokonanej zmianie wszystko przypisane jest do zmiennej newname. W następnym kroku za pomocą polecenia mv zmieniamy nazwę początkową na nową nazwę z rozszerzeniem, które było. Wykonujemy te operacje, by rozszerzenie pliku zostało takie jak było.

Tak naprawdę pętlę for można wykorzystywać do wszystkich operacji, jedynym założeniem jest znajomość poleceń systemowych, znacznie ułatwia to nam pracę.

5.2 Pętla while

Pętla while jest użytecznym mechanizmem, który działa na zasadzie następującej, dopóki spełniony

jest warunek podany po słowie kluczowym while dopóty pętla się wykonuje. Jeśli podamy zły warunek to oczywiście pętla może nie wykonać się ani razu, lub wykonywać się w nieskończoność.

```
#!/bin/bash

tablica=(1 3 9 "Ciag znakow" Ciag znakow)
ilosc_elementow=${#tablica[@]}

i=0
while [ $i -lt $ilosc_elementow ];
do
     znaki[$i]=${#tablica[$i]}
     i=$[i+1]
done
echo ${znaki[@]}
```

Listing 5.2.1 Ilość znaków elementów tablicy

Na powyższym listingu mamy utworzoną tablicę znaków, która zawiera różne elementy o różnych długościach. Aby policzyć ilość znaków poszczególnych elementów możemy wykorzystać pętlę while. Przed pętlą przypisujemy do zmiennej sterującej wartość zero (0 jest pierwszym indeksem tablicy) a następnie wpisujemy warunek. Jeśli i jest mniejsze od ilości elementów to wykonuj zadania, jeśli warunek jest fałszywy to wyświetl ilości znaków. W środku pętli do nowej zmiennej – znaki – która jest tablicą przypisujemy na poszczególnych pozycjach wartości ilości znaków z i-tego elementu tablicy tablica. W kolejnej linijce zwiększany jest licznik zmiennej i, by pętla nie wykonywała się w nieskończoność. Ostatnia linijka to drukowanie wszystkich elementów tablicy znaki.

5.3 Pętla until

Jeśli chodzi o sposób działania pętli until to można by powiedzieć, że jest odwrotnością pętli while. To znaczy pętla wykonuje się dopóki warunek jest nie spełniony. W momencie, gdy warunek stanie się prawdziwy pętla kończy swoje działanie. Sposób użycia jest analogiczny, tylko inne słowo kluczowe jest używane.

```
#!/bin/bash

i=0
max=12
until [ $i -gt $max ]
```

```
do
      echo "2^$i = $[2**$i]"
      i=$[i+1]
done
```

Listing 5.3.1 Drukowanie liczb

Dopóki warunek jest nie spełniony, czyli i jest mniejsze niż max dopóty drukowane będą liczby 2 do potęgi i. W momencie, gdy warunek zostanie spełniony pętla przestaje się wykonywać. Oczywiście tak jak w poprzednim przypadku zwiększamy wartość licznika by pętla zakończyła swoje działanie.

5.4 Pętla select

Pętla select różni się trochę od wyżej wymienionych, tzn z jej pomocą oraz pomocą instrukcji case można zrobić menu programu. Zróbmy prosty skrypt, który w zależności od wybranej opcji będzie wyświetlał różne informacje.

```
#!/bin/bash

while [ TRUE ]
do
      echo "Menu programu"
      select option in "Data" "Lista plikow" "Uptime" "Rozmiar plikow" \
            "Koniec"
      do
            case $option in
                  "Data") date ;;
                  "Lista plikow") ls ;;
                  "Uptime") uptime ;;
                  "Rozmiar plikow") du -sh * ;;
                  "Koniec") exit ;;
                  *) echo "Zla opcja"
            esac
            break
      done
      echo -e "\n"
done
```

Listing 5.4.1 Prosty interfejs

Generalnie "**Koniec**" powinno być w jednej linii z innymi opcjami, ale jeśli się nie mieści, to stawiamy znak \ i możemy kontynuować wpisywanie w nowej linii, tak jakbyśmy pisali dalej w tej samej.

W instrukcji select wpisujemy nazwę zmiennej (tutaj option) a w dalszej części listę słów, które będą opcjami w naszym menu. Za pomocą instrukcji case budujemy nasze menu wpisując jako wzorce nasze wartości z select. Następnie wypisano polecenia systemowe, które odpowiadają za odpowiednie akcje. Generalnie można było by wpisać tam nazwy funkcji, by odwołać się do nich i robić inne zamierzone działania, lecz to jest tylko podstawowy przykład. Wszystko otoczone jest w nieskończoną pętle while, która zostanie przerwana jeżeli użytkownik wpisze wartość 5, ponieważ ta wartość wywoła polecenie exit. Jak widać instrukcja select numeruje całą listę wpisaną po słowie kluczowym in od numeru 1 wzwyż.

6 Funkcje

Funkcje używane są w momencie, gdy kawałek kodu ma być wykonany więcej niż jeden raz. Nie opłaca się pisać dwukrotnie (lub nawet większą ilość razy) praktycznie tego samego kodu, lepiej napisać go raz, umieścić w funkcji, a następnie wywołać dwukrotnie (lub większą ilość razy) tę funkcję. Korzyści są ogromne, bowiem jeśli pomylimy się (lub po prostu coś nam nie pasuje w tej funkcji) to zmieniamy zawartość tylko jeden raz, a tak musielibyśmy zmieniać w każdym miejscu. Użycie funkcji jest bardzo proste, mamy do wyboru dwie opcję definiowania funkcji. Obie przedstawione zostały na poniższym listingu.

```
#!/bin/bash

function drukujWartosc {
      x=10
      echo $x
}

nowaFunkcja () {
      x=11
      echo $x
}

drukujWartosc
nowaFunkcja
```

Listing 6.1 Użycie funkcji

Pierwszym sposobem jest użycie słowa kluczowego function, po którym następuje nazwa funkcji, a następnie nawias klamrowy otwierający ciało funkcji. Po skończeniu wpisywania instrukcji będących ciałem funkcji, nawias klamrowy zamykamy. Drugim sposobem używania funkcji jest ten bez słowa kluczowego, podajemy samą nazwę funkcji, po której następuje para nawiasów okrągłych, a po nich tak jak w poprzednim przypadku nawiasy klamrowe i ciało funkcji. Funkcję wywołuje się podając jej nazwę, co zostało pokazane. Widać, że użycie funkcji jest banalnie proste. Funkcja musi być zdefiniowana przed wywołaniem. W przeciwnym razie dostaniemy komunikat, że polecenia (nazwa funkcji) nie znaleziono.

Tak naprawdę funkcje nie przyjmujące argumentów, oraz nie zwracające wartości są mało użyteczne, dlatego też teraz pokaże jak do funkcji przekazać pewną wartość, oraz jak pewną wartość zwrócić. Zacznijmy najpierw od przekazania wartości do funkcji.

```
#!/bin/bash

function nowaFunkcja {
     pierwszyArg=$1
     drugiArg=$2

     echo "$pierwszyArg"
     echo "$drugiArg"
}

nowaFunkcja "Pierwszy ciag znakow" "Drugi ciag znakow"
```

Listing 6.2 Przekazywanie argumentów do funkcji

Jak widać, mechanizm jest taki sam jak w przypadku przekazywania argumentów podczas uruchamiania skryptu, czyli używamy zmiennych specjalnych do wyciągnięcia tych wartości. Funkcję wywołujemy analogicznie, z tym, że po jej nazwie podajemy parametry, które mają zostać do tej funkcji przekazane. Tak jak widać, dwa ciągi znaków zostaną wydrukowane.

Zwracanie wartości jest trochę kłopotliwe, bynajmniej nie tak intuicyjne jak np. w języku C, gdzie używamy instrukcji return. W bashu też występuje instrukcja return, lecz używana jest ona tylko do zwracania statusu, czyli informacji o tym, czy polecenie wykonało się poprawnie, czy nie (odpowiednie kody dla zaistniałych sytuacji dostępne są w podręczniku systemowym man, dla danego polecenia). Podany poniżej przykład pokazuje, że z użyciem return nie dostaniemy poprawnych wyników.

```
#!/bin/bash

function iloczyn {
     arg=$1
     wynik=$[arg*arg]
     return $wynik
}

iloczyn $1
zwroconaWartosc=$?

echo $zwroconaWartosc
```

Listing 6.3 Błędne zwracanie wartości

Dopóki uruchamiamy skrypt z argumentem mniejszym niż 16 wszystko wygląda fajnie i wydaje się, że

sposób ten odpowiada naszym potrzebom, aczkolwiek problem jest w momencie, gdy wartości są większe, wtedy dzieją się „cuda” i wartość zwracana jest zupełnie inna niż ta, której się spodziewaliśmy. Dzieje się tak ponieważ return (jak już wspomniano) zwraca kody błędów i tylko do wartości 255. Po przekroczeniu zakresu licznik idzie od zera. Aby poradzić se z tym problemem rozważmy następujący sposób.

```
#!/bin/bash

function iloczyn {
     arg=$1
     wynik=$[arg*arg]
     echo $wynik
}

poprawnyWynik=$(iloczyn $1)

echo $poprawnyWynik
```

Listing 6.4 Poprawne zwracanie wartości

W tym przykładzie zastosowano pewną sztuczkę, a mianowicie przekazujemy do funkcji wartość w ten sam sposób jak poprzednio, wykonujemy dokładnie taką samą operację mnożenia, lecz drukujemy wynik za pomocą echo już w tej funkcji. Funkcję wywołujemy w trochę inny sposób, tzn przypisujemy ją do zmiennej, do której trafi wartość wydrukowana za pomocą echo. A następnie drukujemy wartość tej zmiennej. Jeśli wywołalibyśmy funkcję iloczyn nie przypisując jej do zmiennej, to wynik też zostałby wyświetlony, lecz później nie moglibyśmy z niego skorzystać.

Ponieważ nie wspomniano o jeszcze jednej rzeczy, to w tym miejscu należą się wyjaśnienia. Zmienna wynik w funkcji iloczyn tak naprawdę jest zmienną globalną, możemy odwołać się do niej poza ciałem funkcji iloczyn i też otrzymamy wartość. Aby zmienne w funkcjach były lokalne, przed jej nazwą dodajemy słowo kluczowe local.

```
#!/bin/bash

g_str1="Globalny napis1"
g_str2="Globalny napis2"

function func {
     str1="Napis 1"
     local str2="Napis 2"
```

```
     echo "$str1"
     echo "$str2"
     echo "$g_str1"
     echo "$g_str2"
}

func
echo "$str1"
echo "$str2"
```

Listing 6.5 Globalne i lokalne zmienne

Jak widać, użycie słowa kluczowego local przed nazwą zmiennej w danej funkcji skutkuje tym, że widziana jest ona tylko wewnątrz tej funkcji.

Funkcje mogą znajdować się w innym pliku, zaoszczędzi nam to trochę miejsca w głównym pliku, oraz nie trzeba raz po raz wjeżdżać na górę i z powrotem na dół w przypadku błędu w funkcji. Użycie funkcji z innego pliku jest analogiczne, tylko funkcje, które są zdefiniowane w innym pliku muszą zostać, że tak powiem dołączone (uruchomione) do skryptu za pomocą jednej linijki. Na przykładzie poniżej pokazano dodanie dwóch plików z funkcjami. Jeden znajduje się bezpośrednio w katalogu ze skryptem, drugi w katalogu podrzędnym – functions.

```
#!/bin/bash

. other_func.sh
. ./functions/funkcje.sh
dodaj 4 5
odejmij 5 6
mnozenie 4 9
```

Listing 6.6 Skrypt do którego dołączamy plik z funkcjami

```
function dodaj {
     wynik=$[$1+$2]
     echo $wynik
}

function odejmij {
     wynik=$[$1-$2]
     echo $wynik
}
```

Listing 6.7 Plik z funkcjami – funkcje.sh

```
function mnozenie {
     wynik=$[$1*$2]
     echo $wynik
}
```

Listing 6.8 Plik z funkcją – other_func.sh

Zawartość katalogu, w którym znajduje się skrypt i folder z funkcjami wygląda następująco.

```
.
|-- 1.sh
|-- functions
|   `-- funkcje.sh
`-- other_func.sh

1 directory, 3 files
```

Wykaz 6.1 Wykaz polecenia tree

Powyższy wykaz został zamieszczony z jednego ważnego powodu. Spójrzmy na listing 6.6, do pliku z funkcjami odwołujemy się za pośrednictwem kropki, po której podajemy nazwę pliku. Pierwszy plik z funkcją – **other_func.sh** – ma jedną kropkę przed nazwą, tak to się robi, gdy plik znajduje się w katalogu, w którym znajduje się skrypt. Jeśli natomiast plik z funkcjami znajduje się w jakimś podrzędnym katalogu, to również wstawiamy kropkę na początku, lecz wymagana jest kolejna kropka, która w tym momencie jest częścią ścieżki dostępu i oznacza bieżący katalog. Jeśli tę kropkę byśmy usunęli, to interpreter szukałby pliku **funkcje.sh** w katalogu **/functions** – zapis ten oznacza, że katalog **functions** jest w najwyżej położonym miejscu w hierarchii danej partycji. Istnieje jeszcze drugi sposób na dołączenie pliku będącego w katalogu podrzędnym, a mianowicie,

```
. functions/funkcje.sh
```

W tym momencie nie potrzebujemy dodatkowej kropki, lecz nie może być znaku / przed nazwą katalogu **functions.** Pliki z funkcjami z oczywistych powodów muszą zostać odczytane, tak więc muszą mieć ustawione uprawnienia do czytania.

7 Przekierowania strumieni

Czym jest przekierowanie strumienia i po co w ogóle się tego używa? Odpowiedź znajdziesz w tym paragrafie. Załóżmy sytuację, w której używamy już dobrze znanego nam polecenia echo, lecz chcemy, aby wynik został wyświetlony w innym miejscu, aniżeli na ekranie monitora. Jak to zrobić? No właśnie za pomocą przekierowania strumienia.

Musimy najpierw rozpocząć od jednej istotnej rzeczy, a mianowicie wszystkie polecenia domyślnie wyświetlają wyniki swoich działań na standardowe wyjście (stdout – standard out), lub standardowe wyjście błędów (stderr – standard error). Istnieje jeszcze trzeci strumień – standardowe wejście (stdin – standard in), za pomocą którego użytkownik może wprowadzać dane. Z tego trzeciego korzysta polecenie read. Dopóki nie przekierujemy żadnego strumienia zachowanie programu jest znane, ponieważ wszystkie rzeczy zostały przedstawione już w poprzednich punktach.

W gruncie rzeczy możemy zrobić następujące czynności.

1. Przekierowanie standardowego wyjścia do pliku
2. Przekierowanie standardowego wyjścia błędu do pliku
3. Przekierowanie standardowego wyjścia i standardowego wyjścia błędów do pliku
4. Przekierowanie standardowego wyjścia na standardowe wyjście błędów
5. Przekierowanie standardowego wyjścia błędów na standardowe wyjście

Przekierowanie standardowego wyjścia do pliku pozwala wydrukować pewien ciąg znaków nie na ekran monitora, a do sprecyzowanego pliku.

```
#!/bin/bash
plik="test.txt"
echo "Ten tekst zostanie zapisany w pliku: $plik" > $plik
```

Listing 7.1.1 Wynik polecenia echo przekierowany do pliku

Jak widać przekierowanie wyjścia do pliku odbywa się za pomocą operatora >. Można to zrobić jeszcze inaczej, to znaczy dodając jedynkę przed znak operatora, czyli **1>**. W tym miejscu należy jeszcze powiedzieć jaką dodatkową rolę pełni operator >. Operator > w przypadku, gdy plik nie istnieje

– tworzy go. W przypadku, gdy plik istnieje, jego zawartość zostaje obcięta do długości 0, a następnie wartość, która została przekierowana zostaje zapisana w tym pliku.

Istnieje jeszcze inny operator, a mianowicie >>, który nie obcina pliku, lecz dołącza przekierowany wynik na sam koniec pliku. Czyli jeśli zamienimy operator z listingu 7.1.1 na operator >>, to po ponownym uruchomieniu skryptu w pliku tekstowym będą dwie linie tekstu.

Czym jest standardowe wyjście błędów? W zasadzie też drukujemy informacje o błędach na monitorze, więc z pozoru nie różnią się niczym, lecz jednak różnica istnieje. To znaczy w momencie, gdy otrzymamy błąd i nie przekierujemy go – dostajemy go na ekranie. Jeśli przekierujemy go do pliku, to błąd ten zostanie zapisany w pliku. Rozważmy następujący przykład.

```
#!/bin/bash

plik="lkjasdn123asd.txt"
ls -l $plik 2> errors.txt
```

Listing 7.2.1 Standardowe wyjście błędów przekierowane do pliku

W moim katalogu, w którym uruchamiam skrypt nie ma pliku, którego nazwa znajduje się pod zmienną plik, dlatego też wykonanie polecenia ls wyświetla błąd informujący o tym, że plik nie istnieje. Lecz informacje o błędzie przekierowane zostały do pliku **errors.txt** i tam się znajdują. Jeśli teraz na przykład utworzymy ten plik i operatorem przekierowania dalej będzie >, to wynik polecenia ls zostanie wyświetlony na ekranie monitora, a plik **errors.txt** zostanie obcięty do długości 0.

```
#!/bin/bash

plik="lkjasdn123asd.txt"
touch $plik
ls -l $plik 2> errors.txt
```

Listing 7.2.2 Standardowe wyjście błędów przekierowane do pliku, część 2

Jak widać obowiązkowo musimy dodać cyfrę 2 przed operatorem przekierowania wyjścia, żeby poinformować interpreter o tym, iż przekierowujemy wyjście błędów. Analogicznie do poprzedniego byłoby, gdyby znakiem przekierowania był >>.

Teraz zajmiemy się przekierowaniem standardowego wyjścia oraz standardowego wyjścia błędów do

pliku. Jak myślisz co się stanie? Niezależnie, czy polecenie ls wykona się poprawnie czy z błędem, efekt zostanie zapisany do pliku.

```
#!/bin/bash

plik="tego_pliku_nie_ma.txt"
# touch $plik
ls -l $plik &> errors_output.txt
```

Listing 7.3.1 Przekierowanie obu strumieni wyjścia do pliku

Uruchom skrypt, następnie sprawdź zawartość pliku **errors_output.txt**, po czym usuń znak komentarza sprzed polecenia touch, uruchom ponownie skrypt i sprawdź ponownie zawartość pliku **errors_output.txt**. Jeśli przekierowujemy oba strumienie wyjścia, to przed operatorem musimy wstawić znak **&**.

Przekierowanie standardowego wyjścia i/lub standardowego wyjścia błędów na jeden z wymienionych odbywa się za pomocą już znanego operatora, który tym razem składa się z części informującej, co przekierowujemy, oraz części na co przekierowujemy. Tak więc przekierowanie standardowego wyjścia (1) na standardowe wyjście błędów (2) odbywa się za pomocą następującej instrukcji,

```
1>&2
```

z kolei przekierowanie standardowego wyjścia błędów na standardowe wyjście,

```
2>&1
```

Do tej pory nie powiedziano o standardowym strumieniu wejścia. Możemy przekierować strumień wejściowy w taki sposób, by zamiast pobierać dane od użytkownika (wpisanych na klawiaturze) pobierał je z określonego pliku.

```
#!/bin/bash

plik=filename.txt

read answer < $plik
echo $answer
```

Listing 7.4.1 Przekierowania strumienia wejściowego

Widać, że zawartość pliku (jeśli jest jakaś) **filename.txt** zostanie przypisana do zmiennej answer, która z kolei zostanie wydrukowana później na ekran monitora. Na poniższym przykładzie pokażę pewną czynność, którą też można swobodnie wykonywać. Poniższy listing w zasadzie nie różni się od poprzedniego, a efekt możemy uzyskać taki sam.

```
#!/bin/bash

read answer
echo $answer
```

Listing 7.4.2 Wczytywanie danych – reading.sh

Widać od razu, że po uruchomieniu skryptu użytkownik będzie musiał wpisać pewien ciąg znaków, a ENTER zakończy jego wpisywanie i wiersz ten zostanie wyświetlony. Ok, ale możemy przekierować standardowe wejście, aby dane zamiast z klawiatury wczytane były z pliku, dlatego też uruchom ten skrypt w następujący sposób.

```
$ ./reading.sh < nazwa_pliku
```

Gdzie **nazwa_pliku** to nazwa pliku, którego zawartość chcemy przekazać. Oczywiście wyświetlony zostanie tylko pierwszy wiersz.

Poza domyślnymi strumieniami możemy utworzyć własne. Aby tego dokonać musimy otworzyć plik, otworzyć w tym kontekście nie oznacza otwarcia pliku w edytorze tekstowym, lecz odpowiednie polecenie systemowe. Otwieranie pliku ma na celu możliwość, czytania informacji z niego, zapisywania danych do pliku, lub obie te czynności jednocześnie. Otwarty plik jest powiązany z deskryptorem pliku (strumieniem), za pomocą którego możemy do tego pliku przypisywać dane, lub z niego dane wyciągać. Aby przypisać do deskryptora pliku (fd – file descriptor) konkretny plik, używamy polecenia exec. Na przykładzie poniższym pokazano schemat użycia tego mechanizmu, a pod listingiem opis poszczególnych funkcji.

```
#!/bin/bash

plik1="plik1.txt"
plik2="plik2.txt"
```

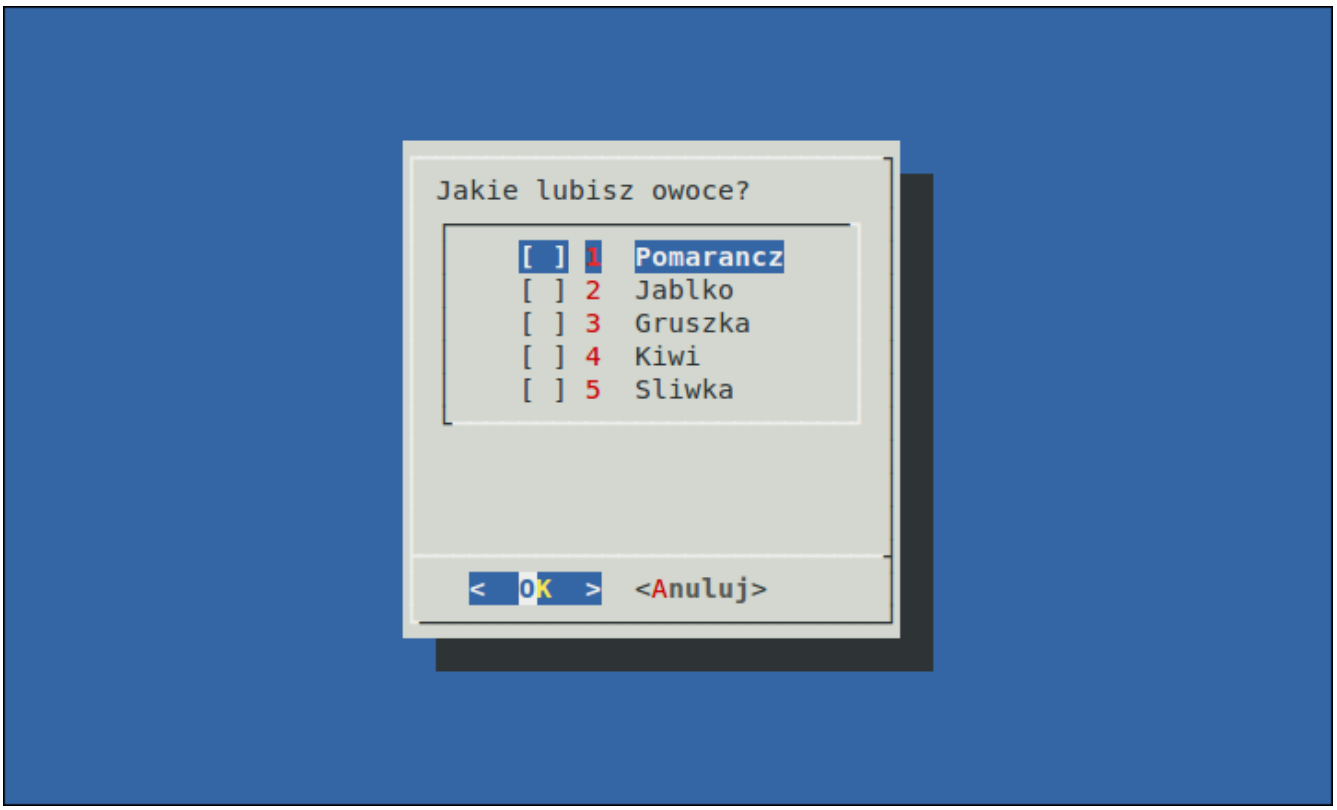
```
plik3="plik3.txt"

exec 3> $plik1
exec 4< $plik2
exec 5<> $plik3

echo "Ten tekst zostanie zapisany w pliku: $plik1" >&3
read wynik <&4
echo "To wpiszemy do: $plik3" >&5
exec 5>&-  # zamkniecie deskryptora

exec 5< $plik3
read zpliku3 <&5

echo $wynik
echo $zpliku3
```

Listing 7.5.1 Użycie własnych deskryptorów plików

A więc tak, po słowie exec podajemy numer deskryptora pliku, a następnie jeden z operatorów odpowiadający za możliwość pobierania danych, wyprowadzania danych, lub pobierania i wyprowadzania danych z pliku (odpowiednio <, >, <>). Po operatorze podajemy nazwę pliku. Od teraz każde przekierowanie na deskryptor numer 3 będzie wiązać się z plikiem **plik1.txt**, do którego możemy tylko zapisywać. Plik **plik2.txt** skojarzony jest z deskryptorem 4, oraz jest tylko do odczytu. Deskryptor 5 pozwala na odczytywanie i zapisywanie do pliku **plik3.txt**. Pierwsza instrukcja echo zapisze pewną wiadomość do **plik1.txt**. Z pliku **plik2.txt** pobieramy pierwszy wiersz i zapisujemy go do zmiennej wynik. Kolejna linijka odpowiada za zapisanie do pliku **plik5.txt**. Pomimo iż **plik3.txt** został otwarty do odczytu i zapisu to jednak nie możemy w oczywisty sposób pobrać danych dopiero co wpisanych. Musimy zamknąć deskryptor formatu, po czym nastąpi faktyczne zapisanie danych do pliku, a następnie po otwarciu pliku do odczytu możemy pobrać (znów pierwszy wiersz) dane i przypisać je do zmiennej zpliku3. Na końcu wyświetlamy wartości znajdujące się pod dwiema zmiennymi.

8 Dialogi

Dialog jest to program, który pozwala wyświetlać informacje w postaci pseudo okienek za pomocą skryptów shell'owskich. Pomimo iż znajomość tego programu nie jest obowiązkowa do pisania skryptów to jednak zachęcam do poznania go, ponieważ skrypty pisane z użyciem tego programu wydają się być bardziej profesjonalne oraz przy okazji dane wyświetlane są w przyjemniejszy sposób. Na rysunku poniższym pokazano przykład okienka wyświetlanego za pośrednictwem programu dialog. Aby wiedzieć wszystko o programie dialog, należy sprawdzić jego dokumentacje za pomocą podręcznika systemowego man. Tutaj pokażę jak zastosować owe polecenie, ale dokumentacja może przydać się w sytuacji, gdy nie wiemy dlaczego jest to w ten, lub inny sposób użyte.



Rys. 8.1 Przykładowy skrypt z użyciem programu dialog

W tym paragrafie będziemy korzystać jedynie z następującego sposobu użycia tego programu.

```
dialog common-options box-options
```

Poniżej zamieszczono listę **common-options**, oraz **box-options**, które wykorzystuje się w programie

dialog. W podręczniku systemowym man (man dialog) znajdziemy opis poszczególnych funkcji, ponieważ nie które zostaną tutaj tylko omówione.

Common options:

[--ascii-lines] [--aspect <ratio>] [--backtitle <backtitle>]
[--begin <y> <x>] [--cancel-label <str>] [--clear] [--colors]
[--column-separator <str>] [--cr-wrap] [--date-format <str>]
[--default-item <str>] [--defaultno] [--exit-label <str>]
[--extra-button] [--extra-label <str>] [--help-button]
[--help-label <str>] [--help-status] [--ignore] [--input-fd <fd>]
[--insecure] [--item-help] [--keep-tite] [--keep-window]
[--max-input <n>] [--no-cancel] [--no-collapse] [--no-kill]
[--no-label <str>] [--no-lines] [--no-ok] [--no-shadow] [--nook]
[--ok-label <str>] [--output-fd <fd>] [--output-separator <str>]
[--print-maxsize] [--print-size] [--print-version] [--quoted]
[--scrollbar] [--separate-output] [--separate-widget <str>] [--shadow]
[--single-quoted] [--size-err] [--sleep <secs>] [--stderr] [--stdout]
[--tab-correct] [--tab-len <n>] [--time-format <str>] [--timeout <secs>]
[--title <title>] [--trace <file>] [--trim] [--version] [--visit-items]
[--yes-label <str>]

Box options:

--calendar <text> <height> <width> <day> <month> <year>
--checklist <text> <height> <width> <list height> <tag1> <item1> <status1>...
--dselect <directory> <height> <width>
--editbox <file> <height> <width>
--form <text> <height> <width> <form height> <label1> <l_y1> <l_x1> <item1> <i_y1> <i_x1> <flen1> <ilen1>...
--fselect <filepath> <height> <width>
--gauge <text> <height> <width> [<percent>]
--infobox <text> <height> <width>
--inputbox <text> <height> <width> [<init>]
--inputmenu <text> <height> <width> <menu height> <tag1> <item1>...

--menu <text> <height> <width> <menu height> <tag1> <item1>...

--mixedform <text> <height> <width> <form height> <label1> <l_y1> <l_x1> <item1> <i_y1> <i_x1> <flen1> <ilen1> <itype>...

--mixedgauge <text> <height> <width> <percent> <tag1> <item1>...

--msgbox <text> <height> <width>

--passwordbox <text> <height> <width> [<init>]

--passwordform <text> <height> <width> <form height> <label1> <l_y1> <l_x1> <item1> <i_y1> <i_x1> <flen1> <ilen1>...

--pause <text> <height> <width> <seconds>

--progressbox <height> <width>

--radiolist <text> <height> <width> <list height> <tag1> <item1> <status1>...

--tailbox <file> <height> <width>

--tailboxbg <file> <height> <width>

--textbox <file> <height> <width>

--timebox <text> <height> <width> <hour> <minute> <second>

--yesno <text> <height> <width>

Zacznijmy od **checklist**'a, którego wynik pokazany jest na rysunku 8.1. Kod tego skryptu ma się następująco.

```
#!/bin/bash

dialog --checklist "Jakie lubisz owoce?" 15 30 5\
       1 Pomarancz off\
       2 Jablko off\
       3 Gruszka off\
       4 Kiwi off\
       5 Sliwka off
```

Listing 8.1 Typ okienka: checklist.

Po słowie dialog opcjonalnie podajemy **common-options,** a w następnej kolejności podajemy **box-options**, czyli jakie to ma być okienko. Na rysunku 8.1 pokazano podstawowe użycie **checklist**'a. Zmodyfikujemy trochę powyższy listing, aby uświadomić sobie pewne właściwości tego programu.

```
#!/bin/bash

dialog --title "Tytul okna" --ascii-lines \
```

```
    --backtitle "Tytul w tle" \
    --checklist "Jakie lubisz owoce?" 15 30 5\
    1 Pomarancz on\
    2 Jablko off\
    3 Gruszka off\
    4 Kiwi off\
    5 Sliwka off\
    6 Mandarynka off\
    7 Brzoskwinia off
```

Listing 8.2 Zmodyfikowany listing 8.1

Rys. 8.2 Wynik wykonania skryptu z listingu 8.2

Widać różnice, przede wszystkim jest tytuł w tle, za który odpowiada opcja **--backtitle**. Tytuł okienka to opcja **--title**. Znaki „–” jako boki ramek, oraz narożniki jako znaki „+” uzyskuje się za pomocą opcji **--ascii-lines**. W sekcji **box options** powyżej mamy wykaz możliwych typów okienek wraz z parametrami jakie przyjmują, tak więc **checklist** przyjmuje jako pierwszy argument ciąg znaków, zazwyczaj pytanie odnośnie poniższych odpowiedzi. Drugi parametr odpowiada za wysokość okienka

(liczony w wierszach), trzeci to szerokość (liczona w kolumnach). Czwarty to wysokość listy, w naszym przypadku wysokość to 5, a lista zawiera większą ilość elementów, dlatego też aby zobaczyć poniższe opcje musimy zjechać kursorem w dół. Kolejne 3 parametry to te odpowiedzialne za wyświetlanie pozycji. Pierwszym z nich jest tag, za pomocą którego będziemy identyfikować, która pozycja została wybrana. Kolejny to nazwa, a ostatni to status (on/off – zaznaczony/odznaczony).

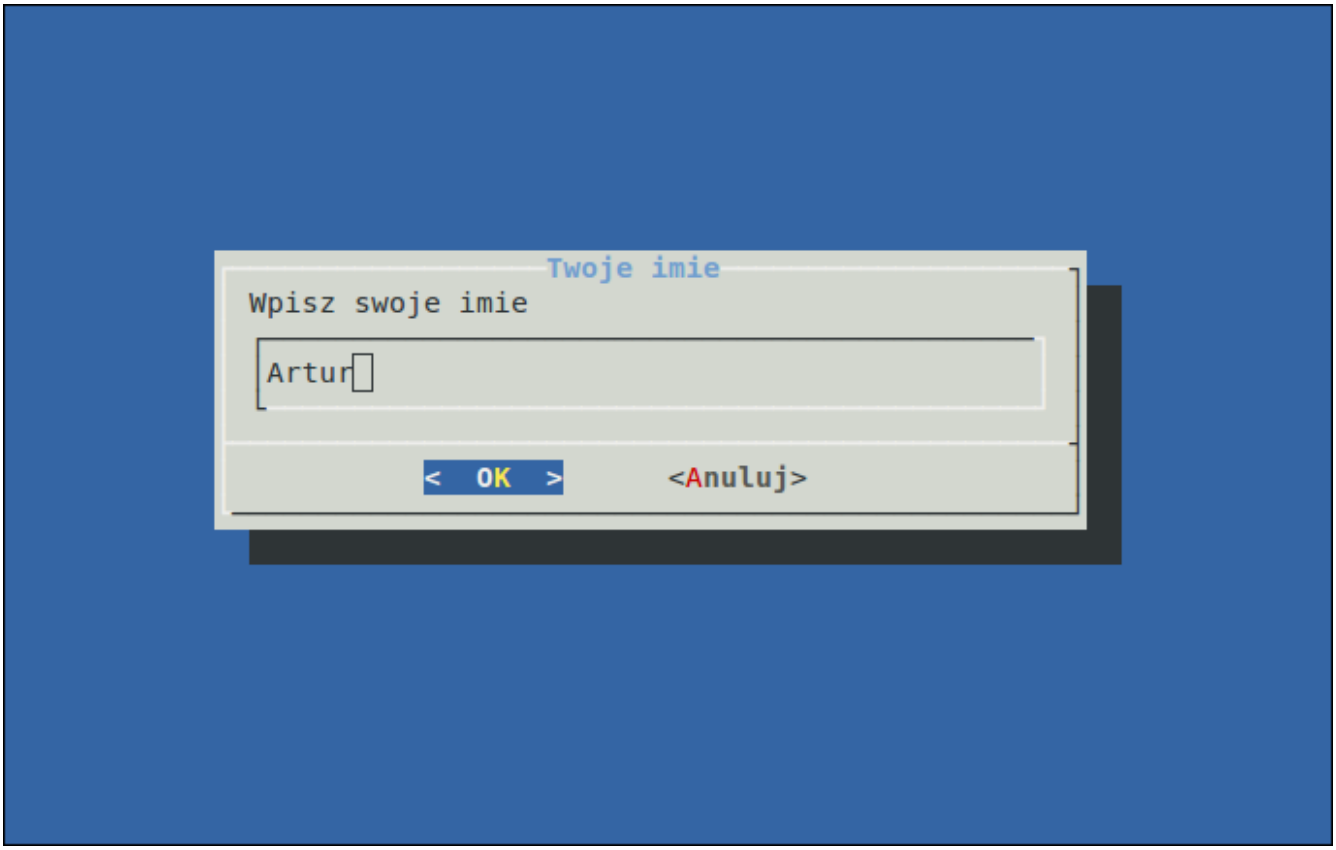
Zaznaczając opcję bez możliwości późniejszego odwołania się do nich (tzn do konkretnej akcji w związku z zaznaczeniem lub wpisaniem danego elementu) mija się z celem, dlatego też teraz pokażę jak za pomocą przykładowego skryptu wyświetlić na ekranie informacje o tym jakie użytkownik wpisał imię.

```
#!/bin/bash
. funkcje.sh

nameFile="/tmp/input.$$"
dialog --title "Twoje imie" --inputbox "Wpisz swoje imie" 8 50 2>\
          $nameFile
dialogreturn=$?
clear

case $dialogreturn in
     "0") wyswietl_imie ;;
     "1") cancel ;;
     "255") esc ;;
esac
```

Listing 8.3.1 Pobieranie od użytkownika jego imienia

```
function wyswietl_imie {
     imie=`cat $nameFile`
     echo "Twoje imie to: $imie"
     rm -f $nameFile
}

function cancel {
     echo "Wcisnieto przycisk Nie, lub Anuluj"
     rm -f $nameFile
     exit 1
}

function esc {
     echo "Wcisnieto klawisz ESC"
     rm -f $nameFile
     exit 2
}
```

Listing 8.3.2 Plik z funkcjami – funkcje.sh



Rys. 8.3 Wynik wykonania skryptu z listingu 8.3.1

Skrypt przygotowany jest na trzy ewentualności, pierwsza z nich to użytkownik wpisze swoje imię i naciśnie OK (domyślnie wybór ustawiony jest na OK, aby wystarczyło wcisnąć ENTER na klawiaturze po wpisaniu imienia). Druga ewentualność, to użytkownik nie wpisze swojego imienia i wciśnie Anuluj (w dół, w prawo, ENTER, ewentualnie kombinacja klawiszy CTRL-C), oraz istnieje trzecia możliwość użytkownik przerwie działanie skryptu za pomocą klawisza ESC. W każdym z trzech przypadków informacja co zrobiliśmy zostanie wyświetlona na ekranie monitora.

Na początku dołączamy plik z funkcjami. Następna czynność to przypisanie do zmiennej konkretnego pliku, plik ten znajduje się w katalogu **/tmp**, a jego nazwa to **input.$$**, gdzie zamiast znaku dolara podstawiony zostanie PID bieżącej powłoki. Oczywiście nazwa pliku może być dowolna, lecz ta będzie się zmieniać w zależności od zmiennej specjalnej. W następnym wierszu używamy programu dialog, podajemy tytuł, oraz typem okienka będzie **inputbox** (okienko do wprowadzania danych), kolejne argumenty określają wysokość i szerokość. Ciekawa rzecz znajduje się na końcu, a mianowicie przekierowujemy standardowy strumień wyjścia błędów na plik określony w zmiennej nameFile. Po

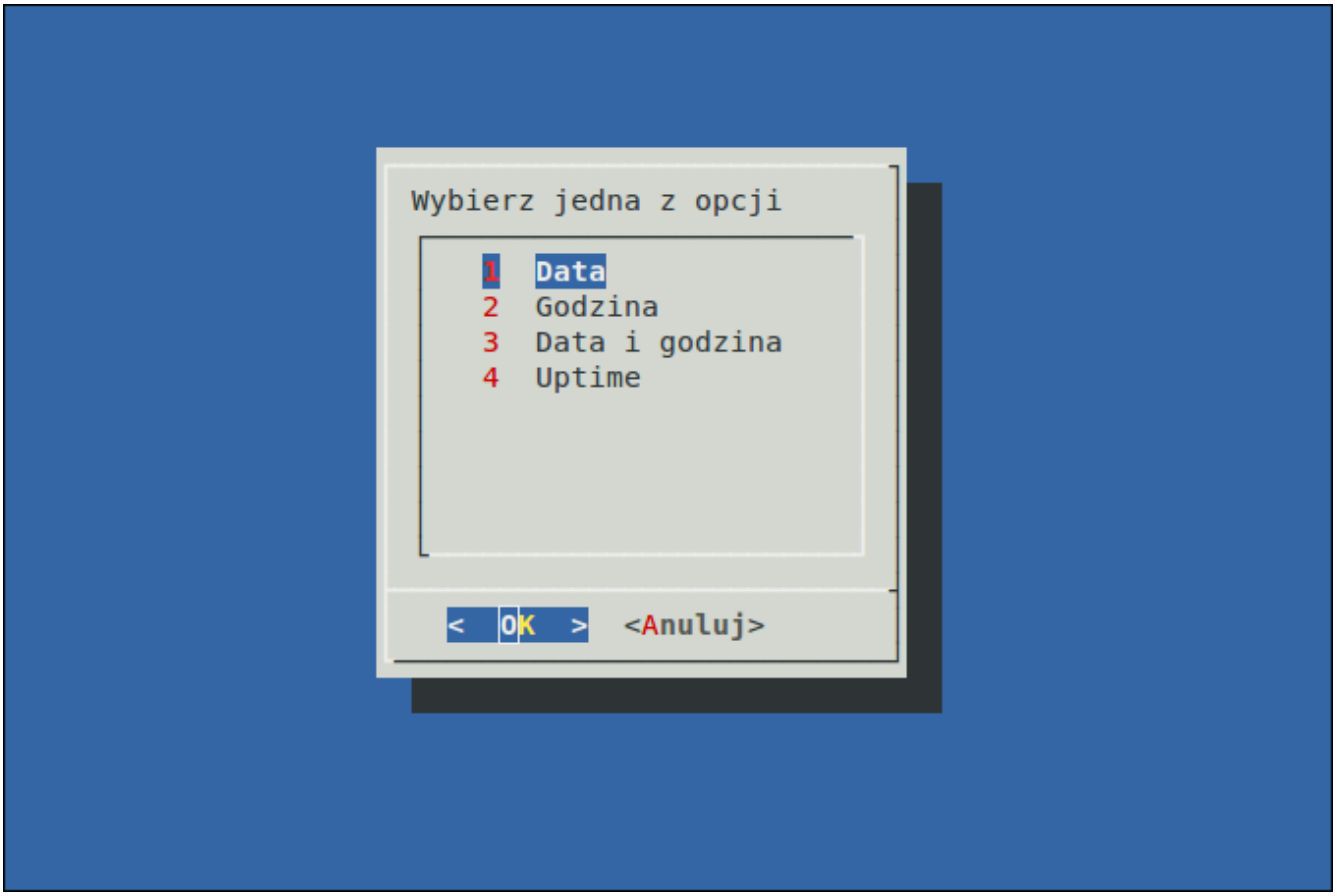
zakończeniu wpisywania, lub przerwaniu za pomocą ESC, lub Anuluj przypisujemy do zmiennej dialogreturn wartość zwróconą przez ostatnie polecenie, czyli przez polecenie dialog. W zależności od tego jaką użytkownik podjął akcję, taka wartość zostanie zwrócona (0 – dialog zamknięty przez OK, 1 – dialog zamknięty przez Anuluj, 255 – dialog zamknięty przez ESC). W kolejnej linii czyścimy ekran, aby nie było widać już niebieskiego tła z okienkiem, tylko „czysty" terminal. Następnie za pomocą case wybieramy która funkcja ma zostać uruchomiona. Ponieważ zmienna nameFile jest globalna, to z funkcji wyswietl_imie możemy się odwołać do niej, czyli przypisujemy do zmiennej imie zawartość pliku (za pomocą polecenia cat) kryjącego się pod tą zmienną, po czym wyświetlamy imię oraz usuwamy plik tymczasowy. Kolejne funkcje wyświetlają odpowiednie informacje, usuwają plik tymczasowy oraz kończą działanie skryptu z odpowiednimi wartościami.

W kolejnym przykładzie typem okienka będzie **menu,** do wyboru będziemy mieli jedną spośród kilku opcji. Podobnie jak w poprzednim przypadku wynik naszego wyboru trafi do tymczasowego pliku, z którego następnie zostanie odczytany i w zależności od jego wartości odpowiednia akcja zostanie wykonana.

```
#!/bin/bash
plik="answer.$$"

function koniec {
     if [ "$result" = "$1" ]
     then
          echo "Nacisneles $2"
          rm -f $plik
          exit $3
     fi
}

dialog --menu "Wybierz jedna z opcji" 15 30 10 \
     1 "Data" \
     2 "Godzina" \
     3 "Data i godzina" \
     4 "Uptime" \
     2>$plik
result=$?
clear

koniec "1" "Anuluj lub Nie" "1"
koniec "255" "ESC" "2"

odpowiedz=`cat $plik`
```

```
case $odpowiedz in
     "1") echo `date +%D` ;;
     "2") echo `date +%R` ;;
     "3") echo `date` ;;
     "4") echo `uptime` ;;
esac
rm -f $plik
```

Listing 8.4 Przykładowe menu



Rys. 8.4 Wynik wykonania skryptu z listingu 8.4

Na początku skryptu znów do zmiennej pomocniczej (przechowującą nazwę pliku) przypisujemy nazwę pliku tymczasowego. Tym razem znajdować się on będzie w katalogu razem ze skryptem. Funkcja koniec zdefiniowana jest po to, by ułatwić przeglądanie naszego skryptu, tzn jak za chwilę zobaczysz, w zależności czy użytkownik przerwał działanie skryptu przez Anuluj, czy przez ESC wykonywane są identyczne akcje, więc po co powielać kod dwukrotnie? Okienko **menu** widać jak na

listingu deklaruje się w taki właśnie sposób. Parametry jakie przyjmuje są wyszczególnione w sekcji **box options**. Znowu pobieramy do zmiennej result wartość jaką zwraca polecenie dialog. Wywołania funkcji koniec różnią się argumentami, o to przecież chodzi, by w zależności od wartości argumentów wyświetlana była poprawna informacja oraz program kończył się z odpowiednią wartością (niezależnie od wartości argumentów plik tymczasowy jest usuwany). Ten sposób jest alternatywą dla robienia kilku funkcji (listing 8.3.2). Następnie przypisujemy do zmiennej odpowiedz zawartość pliku tymczasowego i w zależności od jego wartości wyświetlamy odpowiednie polecenie.

Na początku tego paragrafu pokazano listę wyboru, lecz nie pokazano jak skorzystać z zaznaczonych danych. Zrobione zostało to celowo, by najpierw pokazać ten łatwiejszy sposób używania dialogu (nie oznacza to jednak, że przedstawione poniżej sposoby wymagają jakiejś niezwykłej wiedzy). Najpierw przedstawiony zostanie dialog, w którym mamy formularz, a wypełnione dane zostaną wykorzystane w dalszej części skryptu. Kolejny przykład będzie odnosił się właśnie do **checklist**'y.

```
#!/bin/bash
. funkcje.sh
. global.sh

formularz
result=$?
clear

koniec $result
dane

if [ "$result" = "0" ]
then
     wyswietl_info
fi

rm -f $PLIK
exit 0
```

Listing 8.5.1 Plik główny – 1.sh

```
function formularz {
     dialog --form "Zgloszenie" 25 60 20 "Imie" 1 1 "" 1 20 30 30 \
     "Nazwisko" 2 1 "" 2 20 30 30 \
     "Dzien urodzenia" 3 1 "" 3 20 30 2 \
     "Miesiac urodzenia" 4 1 "" 4 20 30 2 \
     "Rok urodzenia" 5 1 "" 5 20 30 4 \
```

```
     "Kolor oczu" 6 1 "" 6 20 30 30 \
     "Kolor wlosow" 7 1 "" 7 20 30 30 \
     2> $PLIK
}

function dane {
wszystkieDane=`cat $PLIK`

for i in `seq 1 $[ILOSC_WIERSZY]`
do
     szczegoloweDane[$[i-1]]=`echo $wszystkieDane | awk "{print $"$i"}"`
     i=$[i+1]
done
}

function koniec {
     case $1 in
          "255") rm -f $PLIK; exit 1 ;;
          "1") rm -f $PLIK; exit 2 ;;
     esac
}

function wyswietl_info {
     echo "Imie: ${szczegoloweDane[0]}"
     echo "Nazwisko: ${szczegoloweDane[1]}"
     echo "Dzien urodzenia: ${szczegoloweDane[2]}"
     echo "Miesiac urodzenia: ${szczegoloweDane[3]}"
     echo "Rok urodzenia: ${szczegoloweDane[4]}"
     echo "Kolor oczu: ${szczegoloweDane[5]}"
     echo "Kolor wlosow: ${szczegoloweDane[6]}"
}
```

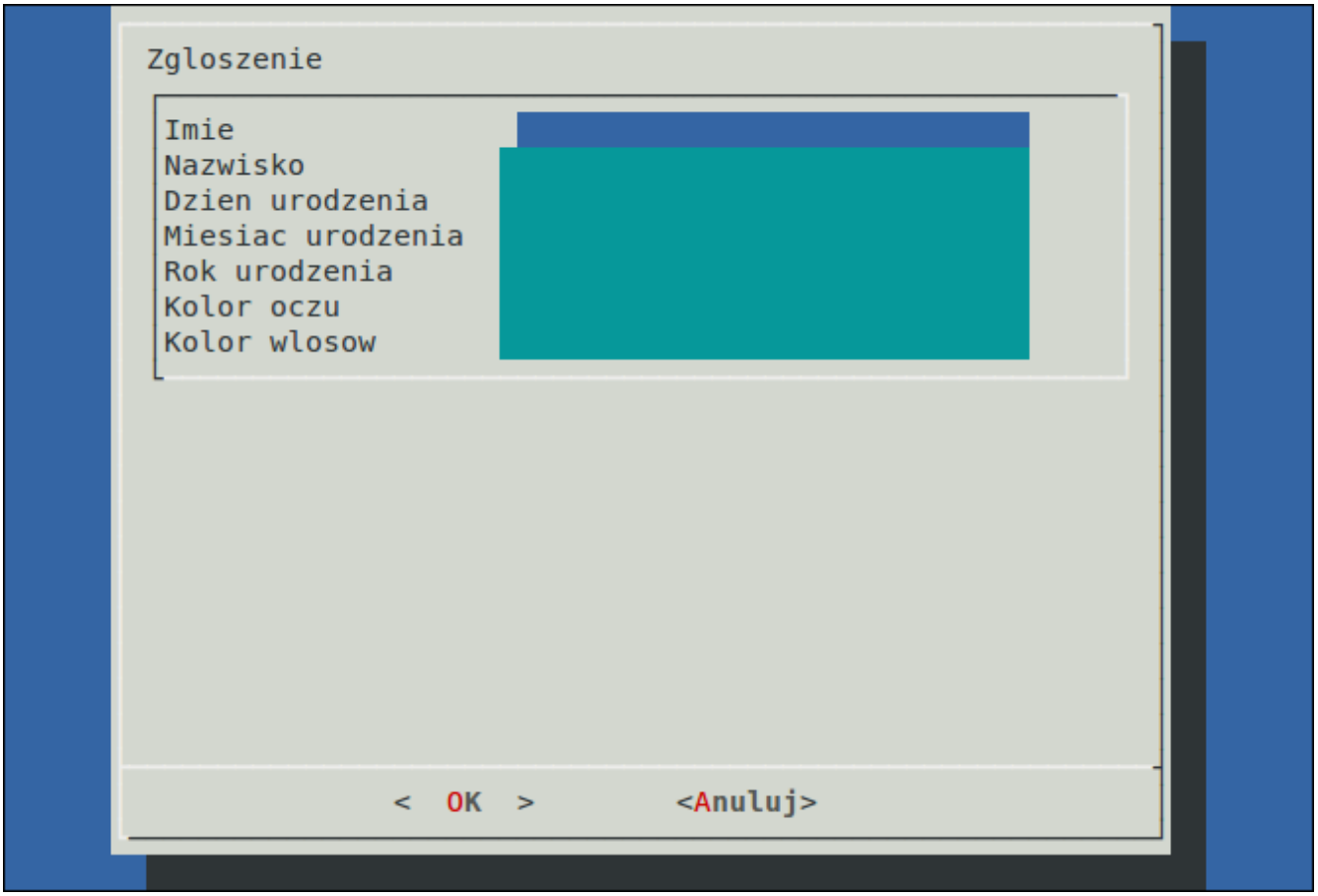
Listing 8.5.2 Plik z funkcjami – funkcje.sh

```
PLIK="file.$$"
ILOSC_WIERSZY="7"
```

Listing 8.5.3 Plik ze zmiennymi globalnymi – global.sh

Widać, że skrypt został rozdzielony na trzy pliki – skrypt główny, plik z funkcjami, oraz plik, w którym występują zmienne globalne (oczywiście można by było zapisać to w głównym skrypcie, lecz taki sposób też istnieje). Na początku głównego skryptu (**1.sh**) dołączane są oba pozostałe pliki, po czym uruchamiana jest funkcja formularz. Zajrzyj do sekcji **box-options** by wiedzieć jakie parametry przyjmuje okienko typu form. Pierwszy trzy są takie same dla wszystkich typów okienek, zostały już opisane, więc skupie się na kolejnych. Czwarty parametr – **form_height** – odpowiada za wysokość

miejsca, w którym znajdują się etykiety oraz miejsca do wprowadzania danych, podobnie jak w przypadku **checklist**'a jeśli wysokość będzie mniejsza niż ilość elementów, to będziemy musieli kursorem zjechać na dół, by zobaczyć niewidoczne na początku opcje. Kolejne trzy parametry oznaczają odpowiednio widoczną nazwę etykiety (label1), położenie względem osi y (l_y1), względem początku formularza, oraz położenie względem osi x (l_x1). Następne trzy odpowiadają za umieszczenie pola do wpisywania danych, podobnie jak dla etykiety pierwszym parametrem jest tekst wpisany w to miejsce domyślnie (item1), kolejne dwa określają położenie względem osi y i x. Dwa ostatnie – flen i ilen – odpowiadają odpowiednio za wyświetlaną długość pola do wprowadzania danych, oraz za ilość możliwych do wpisania znaków. Powracając do naszego przykładu, efektem wykonania tego skryptu będzie wynik jak na rysunku 8.5.



Rysunek 8.5 Wynik wykonania skryptu 8.5.1

W naszym przykładzie pierwszy pole z etykieta będzie wyświetlało nazwę **Imie**, które zaczyna się w pierwszym wierszu i pierwszej kolumnie. Miejsce do wprowadzania naszego imienia zaczyna się

w pierwszym wierszu, oraz w dwudziestej kolumnie. Długość miejsca na wprowadzanie znaków jest ustawiona na 30, oraz możemy wpisać maksymalnie 30 znaków. Definicje kolejnych pól formularza są analogiczne. W przypadku dnia, miesiąca i roku urodzenia maksymalna długość na wprowadzenie znaków została ustawiona odpowiednio na 2, 2, oraz 4. Wynik naszego działania przekierowujemy do pliku, w którym to dane zapisywane są każdy w osobnym wierszu. Na przykład plik z wprowadzonymi danymi może wyglądać następująco.

```
Jan
Kowalski
01
12
1967
Zielone
Blond
```

Analogicznie jak w poprzednich przykładach, do zmiennej result przypisujemy wartość zwróconą przez ostatnie polecenie, czyli przez nasz dialog, a następnie czyścimy ekran. Do funkcji koniec przekazujemy wartość kryjącą się pod zmienną result (de facto zmienna ta jest globalna, można by było nie przekazywać wartości i po prostu odwołać się w funkcji koniec do tej zmiennej). W funkcji koniec sprawdzamy, czy przekazana wartość jest jedną z dwóch 1, lub 255. Jeśli tak jest, to usuwamy plik tymczasowy i kończymy skrypt z odpowiednią wartością. Jeśli użytkownik wpisał dane i zakończył za pomocą ENTER, to wartością jest 0, dlatego funkcja koniec nie podejmuje żadnej akcji. Funkcja dane jest może trochę bardziej skomplikowana, ponieważ do tej pory nie użyto polecenia awk, które w tym miejscu przydało się znakomicie. Na początku tej funkcji pobieramy dane z pliku i przypisujemy je do zmiennej wszystkieDane, które przypisane zostaną w następującej postaci.

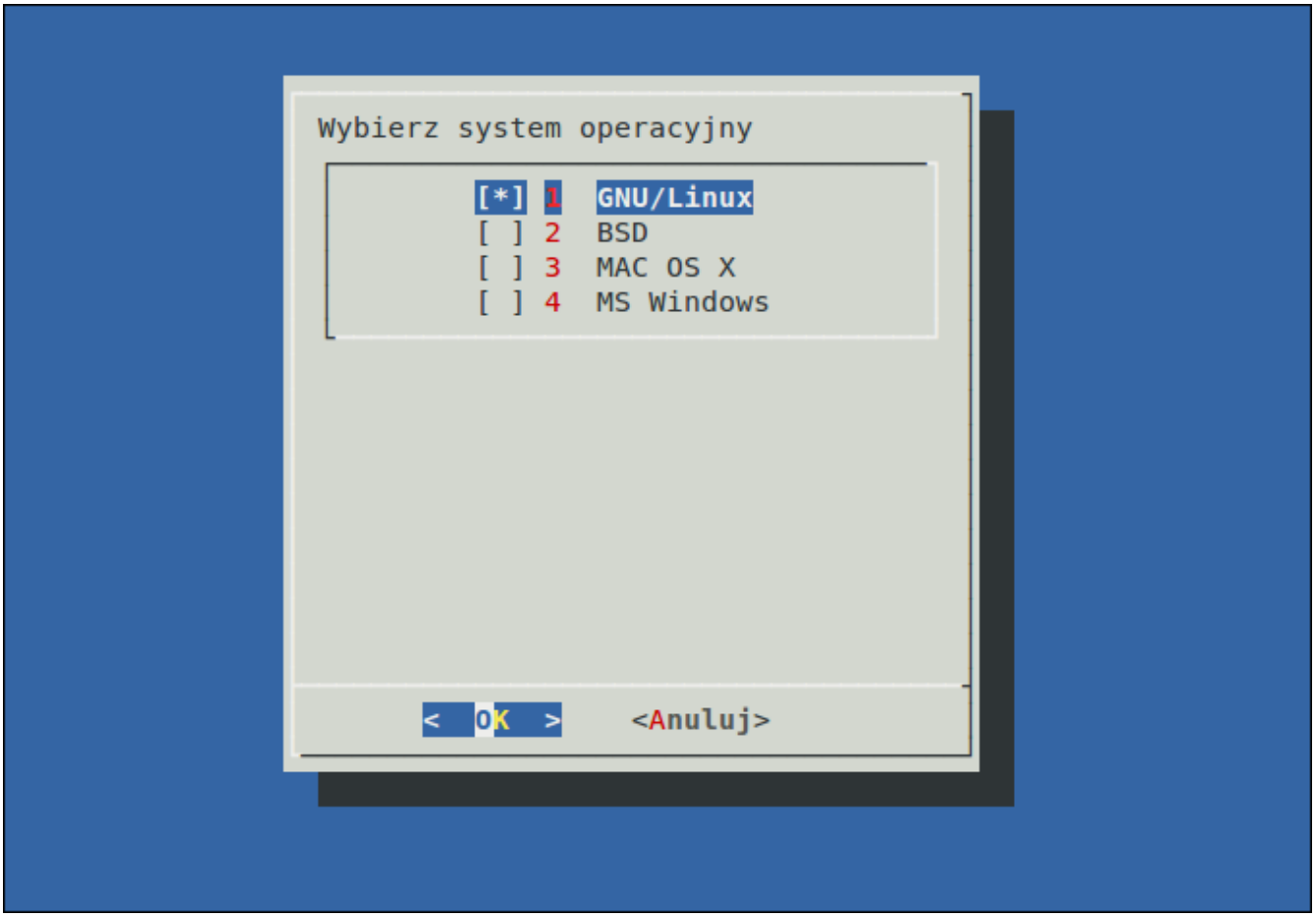
```
Jan Kowalski 01 12 1967 Zielone Blond
```

Następnie za pomocą pętli przypisujemy do tablicy szczegoloweDane pojedyncze słowa. Możesz zastanawiać się jak się to robi, już tłumaczę. Po pierwsze musimy wiedzieć, że używamy potoku, w celu przekazania wyniku polecenia echo $wszystkieDane do programu awk, który te dane obrabia. Jego argumentem jest "{print $"$i"}", który drukuje i-tą kolumnę. Czyli w pierwszej iteracji, na pozycję 0 do tablicy danych przypiszemy pierwszą kolumnę z naszej zmiennej, czyli dla powyższego przykładu pod szczegoloweDane[0] będzie imię Jan. Oczywiście można by było napisać to ręcznie, czyli np. tak.

```
szczegoloweDane[0]=`echo $wszystkieDane | awk '{print $1}'
...
szczegoloweDane[6]=`echo $wszystkieDane | awk '{print $7}'
```

Ale skoro potrafimy już używać pętli, to czemu ich nie wykorzystać? Widać jednak, że w przykładzie bez pętli argument polecenia awk jest zapisany w apostrofach. Ponieważ apostrofy cytują dokładnie to co jest zapisane pomiędzy nimi, czyli nie interpretują zmiennych – nie podstawiają ich wartości – musimy zapisać (w pętli) za pomocą cudzysłowu, by numer kolumny się zmieniał. Tablica szczegoloweDane jest globalna, co w naszym przypadku ułatwia wyświetlanie zawartości. Jeśli zmienna result posiada wartość 0, to wyświetlane są dane za pomocą funkcji wyswietl_info. Następnie usuwamy plik tymczasowy i kończymy skrypt z wartością 0.

W tym miejscu pokazana zostanie lista wyboru po raz kolejny, lecz teraz w trochę bardziej praktyczny sposób, tzn będziemy mogli wykorzystać te dane w jakiś sposób, a nie tylko wybrać je. Poniżej przedstawiono rysunek obrazujący wykonanie niniejszego skryptu.



Rys. 8.6 Praktyczna lista wyboru

```
#!/bin/bash
. funkcje.sh
. ustal_system.sh
PLIK="file$$"
ILOSC_ELEMENTOW=4
dialog_checklist
result=$?

dane=`edytuj_dane`
clear

linux=0; bsd=0; mac=0; win=0

for i in `seq 1 $ILOSC_ELEMENTOW`
do
     wartosc=`echo $dane | awk "{print $"$i"}"`
     case $wartosc in
          "1") linux=1 ;;
          "2") bsd=1 ;;
          "3") mac=1 ;;
          "4") win=1 ;;
     esac
done

ustal_system

rm -f $PLIK
exit 0
```

Listing 8.6.1 Skrypt główny – lista wyboru

```
function dialog_checklist {
dialog --checklist "Wybierz system operacyjny" 20 40 4 \
     "1" "GNU/Linux" on \
     "2" "BSD" off \
     "3" "MAC OS X" off \
     "4" "MS Windows" off \
     2> $PLIK
}

function edytuj_dane {
     local dane=`cat $PLIK`
     echo $dane | sed s/\"//g > $PLIK
     echo `cat $PLIK`
}
```

Listing 8.6.2 Plik z funkcjami – funkcje.sh

```
function ustal_system {
     if [ "$linux" = "1" ]
     then
          echo "Wybrales system: GNU/Linux"
     fi
     if [ "$bsd" = "1" ]
     then
          echo "Wybrales system: BSD"
     fi
     if [ "$mac" = "1" ]
     then
          echo "Wybrales system: MAC OS X"
     fi
     if [ "$win" = "1" ]
     then
          echo "Wybrales system: MS Windows"
     fi

     if [ "$linux" = "0" ] && [ "$bsd" = "0" ] && \
     [ "$mac" = "0" ]  && [ "$win" = "0" ] && \
          [ "$result" = "0" ]
     then
          echo "Nie wybrales zadnego systemu"
     fi
}
```

Listing 8.6.3 Wybór systemu – ustal_system.sh

Początek skryptu jest prawie, że analogiczny do poprzednich przykładów. Zacznijmy więc od funkcji edytuj_dane. Funkcja ta pobiera do zmiennej lokalnej dane zawartość pliku tymczasowego. Następnie znów za pomocą potoku przekazujemy całą zawartość zmiennej dane do programu sed, który edytuje nieco uzyskane w poprzednim kroku dane. Dlaczego to robimy? A więc odpowiedź jest prosta, ponieważ w momencie, gdy użytkownik wybierze elementy z listy, do pliku zostają przekazane wartości **"1" "2"** itp. w zależności od tego ile użytkownik wybrał systemów i jakie. Nas nie interesują wartości razem z cudzysłowem, lecz sama wartość. Polecenie sed w takim wykonaniu,

```
sed s/\"//g
```

zamienia każde wystąpienie cudzysłowu na puste miejsce, a właściwie po prostu usuwa ten znak. Tak przerobione dane trafiają znów do tego pliku nadpisując jego poprzednią zawartość. Trzeci wiersz funkcji edytuj_dane wyświetla zawartość pliku tymczasowego, dlatego też wywołanie funkcji w głównym skrypcie przypisujemy do zmiennej dane. W kolejnych liniach do wymienionych zmiennych przypisujemy wartości początkowe – zera, będzie to dla nas oznaka, że użytkownik nie

wybrał następującego systemu. W pętli for przypisujemy do pomocniczej zmiennej wartosc wywołanie podobne jak w poprzednim przykładzie. Znowu za pomocą awk wybieramy określoną kolumnę w określonej iteracji, a następnie za pomocą case sprawdzamy, czy wartość ta występuje. Jeśli tak jest, to zmieniamy wartość zmiennych odnoszących się do systemów. Należy wspomnieć, że jeżeli użytkownik wybierze mniej niż cztery systemy to polecenie awk w tej nadwyżce zwróci jakiś biały znak, który tak naprawdę nie wpłynie na dalsze nasze działania. Po wykonaniu czterech iteracji sterowanie przechodzi do funkcji ustal_system, która to na podstawie wartości w zmiennych odnoszących się do systemów ustali jaki system (systemy) użytkownik wybrał. W funkcji tej sprawdzane są po kolei wartości tych zmiennych. Ostatnia instrukcja if składa się z kilku warunków. Jeśli zmienne z nazwami systemów są zerami (wszystkie jednocześnie – co oznacza nie wybranie żadnego systemu) to dostajemy informację o tym, że nie wybraliśmy żadnego systemu. Dodatkowo dostawiono jeszcze jeden warunek, który sprawdza, czy kod powrotu polecenia dialog jest zerem. Jeśli tak jest, czyli nie wybrano żadnego systemu i naciśnięto OK, to wyświetl wiadomość. Gdyby tego ostatniego warunku nie było to w momencie, gdy nie wybierzemy żadnego systemu i klikniemy ESC, lub Anuluj dostaniemy też ten komunikat. Na końcu oczywiście usuwany jest plik i skrypt kończony jest z odpowiednią wartością.

Na zakończenie tego rozdziału, oraz generalnie niniejszego podręcznika przygotowałem pewien trochę bardziej rozbudowany skrypt, który pokazuje typowe użycie polecenia dialog. Bardziej zaawansowani programiści powiedzą, że nie ma tam nic specjalnego – może i mają rację, nie mniej jednak niektórym może się on przydać, poza tym dodatkowo dołączono użycie dotąd nie omówionych typów okienek, które pod listingami zostaną omówione, jak zresztą działanie całego programu. Skrypt ten bazuje jedynie na wiedzy, która została przekazana w tym poradniku – zrozumienie jego bez czytania opisu wiąże się z faktem, iż zrozumiałeś przekazywane przeze mnie informacje oraz nauczyłeś się myśleć w trochę inny sposób. Skrypt został podzielony na kilkanaście plików – każda funkcja znajduje się w osobnym pliku o tej samej nazwie z rozszerzeniem **.sh**. Dzięki takiej strukturze wykrycie błędów wydaje się być łatwiejsze, poza tym, dołączenie kolejnych funkcji (funkcjonalność programu) wydaje się banalnie prosta – o czym za chwilę się sam przekonasz. Opis będzie znajdował się „na bieżąco” pod, tudzież nad listingami. Z racji tego, iż skrypt podzielony jest na pliki i katalogi (oczywiście z dostępnych listingów można sobie wywnioskować co gdzie się mieści) niżej zamieszczam wykaz polecenia tree, które pokazuje strukturę katalogów i plików skryptu. Ok, do dzieła.

```
.
|-- funkcje
|   |-- confirm_quit.sh
|   |-- delete_file.sh
|   |-- edit_file.sh
|   |-- koniec.sh
|   |-- logowanie.sh
|   |-- menu.sh
|   |-- naglowki.sh
|   |-- powitanie.sh
|   |-- quit_time.sh
|   |-- show_kernelname.sh
|   |-- show_kernelversion.sh
|   |-- show_ls.sh
|   |-- show_pwd.sh
|   `-- show_uptime.sh
|-- global
|   `-- variables.sh
|-- main.sh
`-- password.txt
```

Wykaz 8.1 Struktura katalogów skryptu

```
#!/bin/bash
. funkcje/naglowki.sh

logowanie
answer=$?

koniec $answer
answer=$?
clear

if [ "$answer" != "$P_OK" ]
then
     exit $E_BADPASSWD
else
     powitanie
     menu
fi
```

Listing 8.7.1 Skrypt główny – main.sh

Pierwszą instrukcją w tym pliku jak widać jest dołączenie innego pliku – **naglowki.h** – plik ten zawiera jak się za chwile okaże same instrukcje dołączające pliki – funkcje – nic więcej. Aby plik **main.sh** był krótki i przejrzysty zastosowano właśnie ten sposób.

```
. global/variables.sh
. funkcje/logowanie.sh
. funkcje/koniec.sh
. funkcje/powitanie.sh
. funkcje/menu.sh
. funkcje/show_kernelname.sh
. funkcje/confirm_quit.sh
. funkcje/show_kernelversion.sh
. funkcje/show_uptime.sh
. funkcje/show_pwd.sh
. funkcje/show_ls.sh
. funkcje/delete_file.sh
. funkcje/edit_file.sh
. funkcje/quit_time.sh
```

Listing 8.7.2 Plik – naglowki.h

W ten sposób dołączamy definicje wszystkich funkcji używanych w skrypcie, w momencie, gdy chcemy dodać nową funkcjonalność dostawiamy w tym pliku kolejną linijkę i mamy problem z głowy – wszystko w jednym miejscu, wiemy gdzie tego szukać.

W tym momencie przyszła kolej na wywołanie funkcji logowanie (plik **logowanie.sh**). Funkcja ta zwraca pewną wartość, którą przypisujemy do zmiennej answer. Funkcja logowanie odpowiedzialna jest za możliwość logowania się do programu.

```
function logowanie {
     local g_passwd=`cat password.txt`
     local file="pass$$"
     dialog --title "Podaj haslo" \
     --passwordbox "Haslo: " 8 40 2> $file
     local result=$?
     clear

     if [ "$result" != "0" ]
     then
          rm -f $file
          exit $E_ESC
     fi

     local passwd=`cat $file`
     rm -f $file

     if [ "$passwd" = "$g_passwd" ]
     then
          return $P_OK
     else
          return $P_BAD
```

```
      fi
}
```

Listing 8.7.3 Plik – logowanie.sh

Na początku funkcji tworzymy zmienną lokalną, która przechowywać będzie hasło do poprawnego zalogowania się do programu. Plik ten znajduje się w głównym katalogu skryptu, oraz nie jest zaszyfrowany – ponieważ pokazuję tutaj tylko funkcjonalność okienka **passwordbox**, a nie metody zabezpieczeń. W prawdziwym tego typu programie trzeba byłoby wziąć pod uwagę, iż do programu mają dostęp tylko wybrane osoby, w tym momencie, każdy kto podejrzy zawartość pliku **password.txt** ma dostęp do programu. Przykładowa zawartość pliku z hasłem wygląda następująco.

```
LOL123!@#
```

Następnie tworzymy zmienną (też lokalną), która zawiera nazwę pliku tymczasowego. Kolejna linijka odpowiada za utworzenie okienka wspomnianego typu. Parametry wyszczególnione są w sekcji **box options**, w tym rozdziale. Z przyczyn bezpieczeństwa wpisywane dane w tym okienku nie są wyświetlane na ekranie, lecz wartość po wciśnięciu OK wyświetlana jest na ekran, w naszym przypadku przekierowywana do pliku. Podobnie jak w poprzednich przykładach do zmiennej result przypisujemy wartość zwróconą przez program dialog i czyścimy ekran. Jeśli użytkownik nie wcisnął OK, to usuwamy plik tymczasowy i kończymy działanie skryptu z wartością określoną przez zmienną E_ESC, która znajduje się w pliku **variables.sh** w katalogu **global**.

```
#Errors
E_ESC=10   # Podczas wpisywania wcisniety ESC
E_BADPASSWD=7    # Zle haslo

#Status
P_OK=0           # Haslo takie samo
P_BAD=1          # Haslo zle
```

Listing 8.7.4 Plik – variables.sh

Po wpisaniu hasła, do zmiennej passwd przypisujemy to co wpisał użytkownik, oraz usuwamy plik tymczasowy. Jeśli oba słowa są identyczne, to zwracamy wartość P_OK, jeśli różnią się, to P_BAD, obie zdefiniowane w pliku **variables.sh**.

Powracając do naszego skryptu głównego – **main.sh** przekazujemy otrzymaną wartość do funkcji

koniec, która sprawdza, czy podane hasło jest poprawne, czy nie.

```
function koniec {
     if [ "$1" != "$P_OK" ]
     then
          dialog --msgbox "Przykro mi $USER. \
               Podales nie prawidlowe haslo. \
               Uruchom skrypt ponownie \
               i wpisz poprawne haslo." 8 40
          return $P_BAD
     else
          return $P_OK
     fi
}
```

Listing 8.7.5 Plik – koniec.sh

W warunku sprawdzamy czy otrzymana wartość z poprzedniej funkcji jest różna od P_OK, jeśli tak jest to dostajemy komunikat w postaci okienka **msgbox** o złym haśle oraz zachęcie do wprowadzenia poprawnego. Zwracana jest wartość P_BAD. Jeśli użytkownik wpisał dobre hasło, zwracana jest wartość P_OK. Kolejną rzeczą w **main.sh** jest sprawdzenie czy zwrócona wartość jest różna od P_OK, w takim przypadku kończymy skrypt z wartością E_BADPASSWD. W przeciwnym wypadku wywołujemy dwie funkcje, najpierw powitanie, później menu.

```
function powitanie {
     dialog --infobox "Witaj $USER! Za chwile wlaczy sie \
     menu programu. Milego dnia!" 6 30
     sleep 5
}
```

Listing 8.7.6 Plik – powitanie.sh

```
function menu {
     local file="ans$$"

     dialog --title "Menu programu" \
          --menu "Wybierz interesujaca Cie opcje" \
          16 50 10 \
          1 "Nazwa kernela" \
          2 "Wersja kernela" \
          3 "Czas wlaczonego komputera" \
          4 "Nazwa biezacego katalogu" \
          5 "Wyswietl zawartosc katalogu" \
          6 "Wybierz plik do usuniecia" \
```

```
            7 "Wybierz plik do edytowania" \
            8 "Zakoncz dzialanie programu" \
            9 "Zakoncz czasowo" 2>$file
      local result=$?

      if [ "$result" = "0" ]
      then
            local wybor=`cat $file`
            rm -f $file
      else
            rm -f $file
            confirm_quit
      fi

      case $wybor in
            "1") show_kernelname ;;
            "2") show_kernelversion ;;
            "3") show_uptime ;;
            "4") show_pwd ;;
            "5") show_ls ;;
            "6") delete_file ;;
            "7") edit_file ;;
            "8") confirm_quit ;;
            "9") quit_time ;;
      esac
}
```

Listing 8.7.7 Plik – menu.sh

W funkcji powitanie mamy informację witającą użytkownika, po czym jest pięciosekundowa przerwa. Tak naprawdę funkcja menu jest najważniejsza z całego skryptu, to za jej pośrednictwem wykonujemy wszystkie operacje. Tworzymy lokalną zmienną file do przechowywania wyboru użytkownika. Wypisane opcje jak widzisz odpowiadają odpowiednim funkcjom. Reakcja ze strony użytkownika decyduje o tym, czy podjęte zostanie działanie w kierunku wywołania odpowiedniej funkcji, czy zamknięcia programu. Zarówno klawisz ESC jak i przycisk Anuluj zamykają program, lecz przed zamknięciem usuwamy plik tymczasowy, oraz użytkownik dostaje komunikat potwierdzający jego żądanie – funkcja confirm_quit.

```
function confirm_quit {
      dialog --yesno "Czy napewno zakonczyc program?" \
            6 30
      local result=$?
      if [ "$result" = "0" ]
      then
            clear
```

```
            exit $Z_OK
      else
            menu
      fi
}
```

Listing 8.7.8 Plik – confirm_quit.sh

Wyskakujące okienko z potwierdzeniem zamknięcia pytania jest typu **yesno**. Jeśli użytkownik wciśnie OK, to skrypt wyczyści okno terminala i zakończy się z wartością Z_OK. W przeciwnym wypadku wywołujemy jeszcze raz funkcję menu. Jeśli jednak użytkownik wybrał odpowiednią opcję to zostanie ona przypisana do zmiennej lokalnej wybor, a plik tymczasowy zostanie usunięty z dysku. W kolejnym kroku za pomocą instrukcji case wybieramy odpowiednią funkcję. Po nazwach funkcji widać z grubsza co będziemy robić. Pierwsza pozycja pokazuje nazwę kernela. W miejscu, gdzie mamy wstawić tekst, wstawiamy polecenie, którego efekt chcemy wydrukować.

```
function show_kernelname {
      dialog --title "Nazwa Kernela" \
            --msgbox "`uname -s`" 5 20
      menu
}
```

Listing 8.7.9 Plik – show_kernelname.sh

Po wywołaniu okienka, niezależnie od tego, czy użytkownik wciśnie OK, czy naciśnie ESC, wywoływana jest funkcja main. Schemat ten powtarza się w kolejnych trzech funkcjach.

```
function show_kernelversion {
      dialog --title "Wersja Kernela" \
            --msgbox "`uname -r`" 5 30
      menu
}
```

Listing 8.7.10 Plik – show_kernelversion.sh

```
function show_uptime {
      dialog --title "Czas wlaczonego komputera" \
            --msgbox "`uptime | awk '{print $3}' | sed s/,//g`" \
            5 30
      menu
}
```

Listing 8.7.11 Plik – show_uptime.sh

```
function show_pwd {
     dialog --title "Katalog, w ktorym sie znajdujesz" \
          --msgbox "`pwd`" 5 40
     menu
}
```

Listing 8.7.12 Plik – show_pwd.sh

W funkcji show_uptime, w miejscu, gdzie ma zostać wydrukowany tekst wpisano wartość polecenia, które z kolei wynik przekazuje do polecenia awk, w celu wydrukowania tylko i wyłącznie czasu, jaki komputer jest włączony (drukowana jest tylko trzecia kolumna), następnie to co otrzymano przekazywane jest do polecenia sed, które usuwa przecinek (ponieważ trzecia kolumna zawiera wartość razem z przecinkiem, jeśli komputer jest włączony przynajmniej godzinę, jeśli mniej to ilość minut jest bez przecinka, na co polecenie sed nie reaguje).

Wyświetlanie zawartości katalogu osiągamy w podobny sposób, lecz najpierw wybieramy katalog, którego zawartość chcemy obejrzeć za pośrednictwem okienka typu **dselect**.

```
function show_ls {
     local plik="dir$$"
     dialog --title "Wybierz katalog" \
          --dselect ~ 10 30 2>$plik
     local result=$?

     if [ "$result" = "255" ] || [ "$result" = "1" ]
     then
          rm -f $plik
          menu
     else
          local fileName=`cat $plik`
          rm -f $plik
     fi

     dialog --title "Zawartosc katalogu: $fileName" \
          --msgbox "`ls -l $fileName`" 20 70
     menu
}
```

Listing 8.7.13 Plik – show_ls.sh

Podobnie jak w poprzednich przykładach, w których interesował nas wybór (lub wpisane informacje) użytkownika tworzymy zmienną lokalną, do której przypisujemy nazwę pliku. Okienko **dselect** umożliwia wybranie katalogu, po wciśnięciu OK ścieżka do katalogu przekazywana jest do pliku. Jeśli

użytkownik wcisnął ESC, lub Anuluj to usuwamy plik tymczasowy i uruchamiamy ponownie funkcję menu. W tym miejscu taka mała dygresja, która nie nastąpiła wcześniej. Jeśli użytkownik wciśnie ESC, lub Anuluj to plik też zostanie utworzony, lecz będzie pusty. Z tego też powodu w dwóch miejscach wywoływane jest polecenie rm, aby nie zaśmiecać katalogu. Jeśli wyborem użytkownika było OK, to z pliku pobieramy ścieżkę wskazaną przez niego i przypisujemy do zmiennej fileName, oraz usuwamy plik tymczasowy. Kolejnym elementem jest utworzenie nowego okienka, w którym wyświetlamy zawartość wskazanego katalogu.

Aby sprawnie obsługiwać okienka typu **dselect**, **fselect** (omówione za chwile) w tym miejscu poświecę dwa zdania. Ścieżkę można wpisywać ręcznie, lub korzystać z pomocy wyświetlanej w okienku. Pierwsza ważna rzecz, jak jesteśmy w jakimś miejscu i chcemy wejść do danego katalogu, to albo wpisujemy ręcznie jego nazwę i wciskamy klawisz /, przez co zawartość danego katalogu się wyświetli, lub w momencie, gdy mamy część nazwy i katalog jest podświetlony wciskamy SPACJĘ. Druga sprawa, to taka, że możemy strzałkami szukać sobie katalogu, podobnie jak w poprzednim przypadku wejście do wybranego katalogu odbywa się za pomocą SPACJI.

Okienko **fselect** używane do wyboru plików działa na podobnej zasadzie do omówionego przed chwilą **dselect**. Różnica polega na tym, że w tym momencie mamy dwa okienka. Po prawej znajdują się pliki z bieżącego katalogu.

```
function delete_file {
     local file="remFile$$"
     dialog --title "Wybierz plik do usuniecia" \
          --fselect ~ 20 60 2>$file
     local result=$?

     if [ "$result" = "255" ] || [ "$result" = "1" ]
     then
          rm -f $file
          menu
     else

          local file_to_remove=`cat $file`
          rm -f $file
          rm -f "$file_to_remove"
          menu
     fi
}
```

Listing 8.7.14 Plik – delete_file.sh

Funkcja jest analogiczna do poprzedniej, dlatego omówię tylko cześć po else. Do lokalnej zmiennej przypisujemy zawartość pliku uzupełnioną przez okienko dialog w formie ścieżki do pliku. Następnie usuwamy plik tymczasowy oraz ten, na którego wskazuje zmienna file_to_remove. Należy w tym miejscu wspomnieć, że dla lepszego działania wypadałoby dodać zapytanie, czy użytkownik na pewno chce usunąć ten plik, czego w tym miejscu nie uczyniono. Kolejna linia wywołuje funkcję menu.

Z użyciem okienka typu **editbox** możemy edytować plik, niżej przedstawiono jak to zrobić.

```
function edit_file {
     local plik="editFile$$"

     dialog --title "Wybierz plik do edycji" \
          --fselect ~ 20 60 2>$plik
     local result=$?

     if [ "$result" = "255" ] || [ "$result" = "1" ]
     then
          rm -f $plik
          menu
     else
          local file_to_edit=`cat $plik`
          rm -f $plik
     fi

     local tresc="tresc$$"
     dialog --title "Edycja pliku: $file_to_edit" \
          --editbox "$file_to_edit" 20 60 2>$tresc
     result=$?

     if [ "$result" = "0" ]
     then
          cp -f "$tresc" "$file_to_edit"
          rm -f "$tresc"
          menu
     else
          rm -f "$tresc"
          menu
     fi
}
```

Listing 8.7.15 Plik – edit_file.sh

Pierwsza część funkcji edit_file odnosi się do wybrania pliku, gdy już to zrobimy i ścieżka do pliku będzie w zmiennej file_to_edit, możemy przystąpić do edytowania pliku. Zaraz po typie okienka podajemy ścieżkę do pliku. Zawartość pliku zostanie wyświetlona i będziemy mieli możliwość

edytowania go. Po dokonaniu jakiś zmian i zaakceptowaniu ich przez OK, nowa treść zostaje przekierowana do pliku, który zdefiniowaliśmy w zmiennej tresc przed wywołaniem okienka. Jeśli użytkownik kliknął OK, to najpierw kopiujemy plik tymczasowy i podmieniamy go z plikiem bazowym, a następnie usuwamy plik tymczasowy. Można byłoby użyć polecenia mv. Tak czy inaczej funkcja menu zostanie wywołana ponownie.

Pozycja ósma została już omówiona, jest to zapytanie czy na pewno chcemy zakończyć działanie programu – funkcja confirm_quit. Za to na pozycji dziewiątej występuje zamykanie czasowe. Ilość sekund po której zostanie program zamknięty jest wpisana na sztywno, można byłoby zmienić lekko kod, aby wartość ta była wprowadzana przez użytkownika, oraz można byłoby zrobić tak, by w momencie, gdy użytkownik wciśnie ESC, lub Anuluj program się nie zamknął i powrócił do funkcji menu – czynności te były robione przez większość funkcji, więc myślę, że rozbudowa nie będzie problemem.

```
function quit_time {
     dialog --yesno "Czy napewno zakonczyc program?" \
          6 30
     local result=$?
     if [ "$result" = "255" ] || [ "$result" = "1" ]
     then
          menu
     fi

     dialog --title "Zamykanie programu" \
          --pause "Zamykanie programu" 10 35 10
     clear
     exit $Z_OK
}
```

Listing 8.7.16 Plik – quit_time.sh

Jeśli włączając pewną funkcję, nie włącza Ci się ona i wyświetla się ponownie menu, może oznaczać to, że rozmiar Twojego terminala jest za mały. Niektóre okienka, jak na przykład wybór pliku wymaga 60 kolumn do wyświetlenia. Rozszerz terminal i sprawdź ponownie.

Skrypt ten może i mało praktyczny, lecz pokazuje w jaki sposób można używać polecenia dialog w trochę szerszym zakresie. Na pewno nie jest to jedyny sposób na pisanie kodu. Istnieją pewnie inne, bardziej praktyczne sztuczki, ale jak już mówiłem, skrypt ten miał na celu pokazać tylko pewien mechanizm działania.

9 Bibliografia

Poniżej przedstawiono linki do stron z których poniekąd czerpałem inspirację. Pierwsza pozycja zalecana jest jeśli chcesz wiedzieć dosłownie wszystko o całym Bashu, nie tylko programowaniu z użyciem tego interpretera. Pozostałe trzy to poradniki internetowe, kolejność ich decyduje o ilości informacji dostarczonych przez autorów (od największej, do najmniejszej).

1. http://www.gnu.org/software/bash/manual/
2. http://www.tldp.org/LDP/abs/html/index.html
3. http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
4. http://www.republika.pl/dief/main.html

Podczas pisania tego poradnika korzystałem z podręcznika systemowego man w celu dobrania odpowiednich argumentów do poleceń, oraz informacje o niektórych ważnych opcjach danych poleceń.