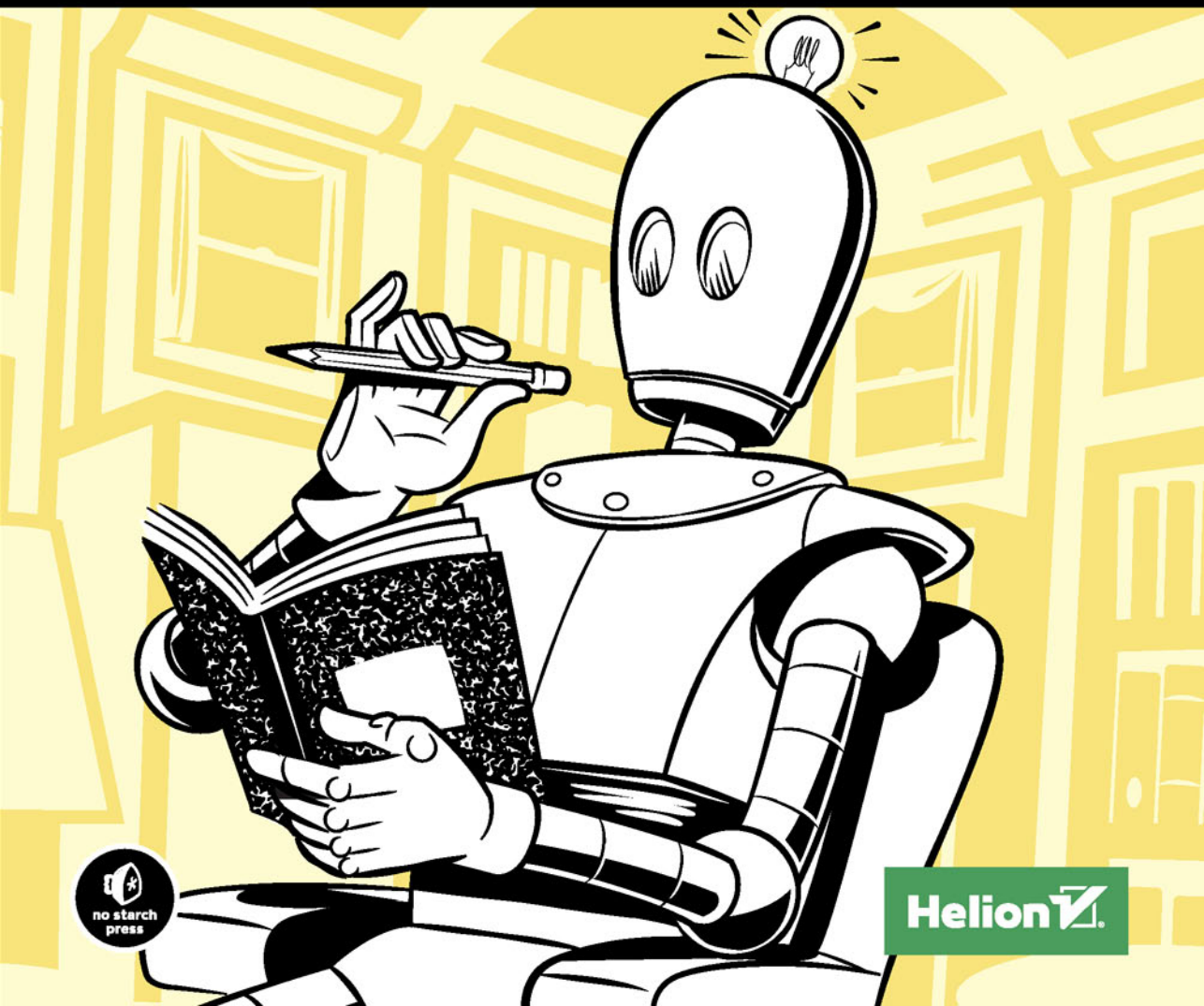
WYDANIE II

GENIALNE SKRYPTY POWŁOKI

PONAD 100 ROZWIĄZAŃ
DLA SYSTEMÓW LINUX,
macOS I UNIX

DAVE TAYLOR, BRANDON PERRY



no starch press

Helion

Tytuł oryginału: Wicked Cool Shell Scripts, 2nd Edition

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-3430-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/geskpo
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

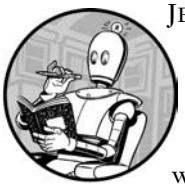
Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/geskpo.zip

Printed in Poland.

Spis treści

1

Brakująca biblioteka kodu



Jedną z największych zalet systemu Unix jest możliwość tworzenia nowych poleceń poprzez wykorzystanie istniejących w inny sposób. Choć system zawiera setki poleceń, które można łączyć na tysiące sposobów, zawsze będziesz spotykał się z przypadkami, gdy żadne z nich nie będzie odpowiednie do wykonania potrzebnej operacji tak jak trzeba. W tym rozdziale opisujemy metody tworzenia lepszych i bardziej zaawansowanych poleceń w świecie skryptów powłoki.

Jest jednak jedna rzecz, o której musimy powiedzieć na samym początku: środowisko do tworzenia skryptów powłoki nie jest tak zaawansowane jak prawdziwe środowisko programistyczne. Języki Perl, Python, Ruby, a nawet C zawierają struktury i biblioteki oferujące różne dodatkowe funkcjonalności. Natomiast w świecie skryptów powłoki obowiązuje zasada „radź sobie sam". Skrypty opisane w tym rozdziale pozwolą Ci odnaleźć własną drogę w tym świecie. Są to elementy konstrukcyjne, za pomocą których będziesz mógł w dalszej części książki tworzyć niesamowite skrypty.

Wiele problemów towarzyszących pisaniu skryptów ma swoje źródło w subtelnych różnicach pomiędzy odmianami systemu Unix oraz dystrybucjami systemu Linux. Choć standardy IEEE POSIX niewątpliwie określają wspólną bazę

funkcjonalności dla implementacji wszystkich systemów, jednak przejście na system macOS po latach używania Red Hat GNU/Linux może być stresujące. Polecenia są inne, gdzie indziej zapisane, pomiędzy ich argumentami występują różnice. Te rozbieżności powodują, że tworzenie skryptów jest trudną sztuką, ale nauczysz się kilku sztuczek, dzięki którym różnice te nie będą takie groźne.

Czym jest POSIX?

Na początku system Unix przypominał Dziki Zachód. Różne firmy ulepszały go i rozwijały w różnych kierunkach, a jednocześnie zapewniały użytkowników, że wszystkie nowe wersje są kompatybilne z poprzednimi i jest to ten sam Unix. W pewnym momencie do gry włączył się instytut **IEEE** (ang. *Institute for Electrical and Electronic Engineers* — Instytut Inżynierów Elektryków i Elektroników) i przy ogromnym wysiłku własnym i wszystkich twórców systemu Unix opracował standard **POSIX** (ang. *Portable Operating System Interface* — przenośny interfejs dla systemu operacyjnego Unix), zgodnie z którym powinny być tworzone wszystkie komercyjne i bezpłatne implementacje systemu Unix. Nie można nabyć systemu operacyjnego POSIX jako takiego, ale system Unix lub GNU/Linux, którego używasz, jest generalnie zgodny ze standardem POSIX (choć pojawiają się głosy, czy standard ten w ogóle jest potrzebny, skoro GNU/Linux sam stał się standardem).

Ale nawet zgodne ze standardem POSIX implementacje systemu Unix mogą różnić się między sobą. Jednym z przykładów, opisanych w dalszej części rozdziału, jest polecenie `echo`. W niektórych wersjach systemu obsługuje ono argument `-n` zapobiegający przejściu kursora do nowego wiersza, co zazwyczaj ma miejsce po wykonaniu polecenia. W innych wersjach systemu w wyświetlanym poleceniem `echo` tekście należy umieścić sekwencję `\c` oznaczają „nie przechodź do nowego wiersza", a w jeszcze innych odmianach systemu nie ma w ogóle możliwości zapobieżenia przejściu kursora do nowego wiersza. Aby było jeszcze ciekawiej, niektóre wersje systemu Unix zawierają powłoki, w których jest wbudowane polecenie `echo` nieobsługujące argumentów `-n` ani `\c`, albo polecenie to jest samodzielnym plikiem binarnym */bin/echo*, obsługującym powyższe argumenty. Wszystko to sprawia, że trudno jest w jednolity sposób implementować żądania wprowadzenia danych przez użytkownika w skryptach, które powinny działać identycznie w możliwie jak największej liczbie odmian systemu Unix. Dlatego gdy chce się tworzyć przydatne skrypty, bardzo ważne jest znormalizowanie polecenia `echo` tak, aby działało ono tak samo we wszystkich systemach. W dalszej części rozdziału, w skrypcie nr 8, dowiesz się, jak wykorzystać polecenie `echo` w skrypcie, aby utworzyć znormalizowaną wersję tego polecenia.

UWAGA *W niektórych skryptach opisanych w tej książce wykorzystane są cechy powłoki bash, które mogą nie być dostępne w innych powłokach zgodnych ze standardem POSIX.*

Ale dość historii, zajmijmy się skryptami, które potem wykorzystasz do utworzenia swojej biblioteki!

Skrypt 1. Wyszukiwanie programów w katalogach zmiennej PATH

W skryptach wykorzystujących zmienne środowiskowe (na przykład `MAILER` lub `PAGER`) kryje się niebezpieczeństwo: niektóre ich ustawienia mogą wskazywać programy, których nie ma. Jeżeli nie miałeś do czynienia z powyższymi zmiennymi, wyjaśniamy, że `MAILER` wskazuje preferowany program pocztowy (na przykład */usr/bin/mailx*), a `PAGER` — program służący do przeglądania długich dokumentów ekran po ekranie (strona po stronie). Jeżeli na przykład zadecydujesz, że w elastycznym skrypcie do wyświetlania długiego wyniku użyjesz programu wskazanego w zmiennej `PAGER` zamiast domyślnego programu systemowego (najczęściej `more` lub `less`), to w jaki sposób możesz wtedy sprawdzić, że powyższa zmienna zawiera poprawną wartość?

Pierwszy skrypt będzie służył do sprawdzania, czy dany program znajduje się w jednym z katalogów zapisanych w zmiennej `PATH`. Jest to jednocześnie dobry przykład zastosowania kilku technik, między innymi funkcji i wyodrębniania fragmentów danych. Listing 1.1 pokazuje, jak można sprawdzić, czy ścieżki są poprawnie wpisane.

Kod

Listing 1.1. Funkcje zawarte w skrypcie inpath

```
#!/bin/bash
# Skrypt inpath sprawdzający, czy dany program jest poprawny sam w sobie,
# czy też znajduje się w katalogach zapisanych w zmiennej PATH.

in_path()
{
    # Funkcja usiłująca odnaleźć program na podstawie jego nazwy i zmiennej PATH.
    # Zwraca 0, jeżeli program istnieje i jest plikiem wykonywalnym, albo 1 w przeciwnym wypadku.
    # Zwróć uwagę, że tymczasowo zmieniany jest separator IFS, ale jego pierwotna wartość jest
    # przywracana na końcu funkcji.

    cmd=$1        ourpath=$2         result=1
    oldIFS=$IFS   IFS=":"

    for directory in "$ourpath"
    do
      if [ -x $directory/$cmd ] ; then
        result=0        # Dojście do tego miejsca oznacza, że program $cmd znajduje się
                        # w katalogu $directory.
      fi
    done

    IFS=$oldIFS
    return $result
 }
```

```
checkForCmdInPath()
{
  var=$1

  if [ "$var" != "" ] ; then
❶   if [ "${var:0:1}" = "/" ] ; then
❷     if [ ! -x $var ] ; then
        return 1
      fi
❸   elif !  in_path $var "$PATH" ; then
      return 2
    fi
  fi
}
```

Jak wspomnieliśmy w rozdziale 0, zalecane jest utworzenie w katalogu domowym nowego katalogu o nazwie *skrypty* i dodanie jego pełnej nazwy do zmiennej `PATH`. Użyj teraz polecenia `echo` do sprawdzenia bieżącej wartości zmiennej, a następnie w odpowiednim skrypcie logowania (*.login*, *.profile*, *.bashrc* lub *.bash_profile* — w zależności od powłoki) wpisz polecenie modyfikujące zmienną `PATH`. Szczegółowe informacje, jak to zrobić, znajdziesz w rozdziale 0, w części „Konfigurowanie skryptu logowania".

UWAGA *Jeżeli do wyświetlania listy plików w terminalu używasz polecenia `ls`, możesz na początku nie zobaczyć specjalnych plików, takich jak* .bashrc *czy* .bash_profile. *Jest tak dlatego, ponieważ pliki o nazwach zaczynających się od kropki są traktowane jako „ukryte". (Na samym początku epoki systemu Unix okazało się, że ta konwencja przysparza pewnych kłopotów). Aby wyświetlić wszystkie pliki w danym katalogu, włącznie z ukrytymi, użyj polecenia `ls` z argumentem `-a`.*

Chcemy jeszcze raz podkreślić, że przyjęliśmy w tej książce założenie, że ze wszystkich skryptów będziesz korzystał w powłoce bash. Zwróć uwagę, że w pierwszym wierszu jawnie wskazana jest ścieżka (shebang) */bin/bash*. W wielu systemach stosowana jest ścieżka */usr/bin/env* określająca środowisko właściwe dla danego skryptu.

UWAGA DO KOMENTARZY

Mieliśmy dylemat, czy powinniśmy szczegółowo opisywać działanie każdego skryptu. W niektórych przypadkach opisujemy w tekście trudniejsze fragmenty kodu, ale generalnie stosujemy w kodzie komentarze zawierające wyjaśnienia, co w danym miejscu się dzieje. Zwracaj uwagę na znaki # i tekst, który po nich następuje.

Ponieważ bez wątpienia kiedyś będziesz musiał czytać skrypty utworzone przez innych programistów, dobrą praktyką jest czytanie komentarzy, aby móc zrozumieć, o co, do licha, w tym skrypcie chodzi (nie dotyczy to nas, oczywiście!). Umieszczanie komentarzy jest znakomitym zwyczajem, ponieważ opisują one operacje wykonywane w danym bloku kodu.

Jak to działa?

Aby funkcja `checkForCmdInPath` działała zgodnie z przeznaczeniem, należy przede wszystkim sprawdzić, czy podane dane zawierają tylko nazwę programu (na przykład `echo`), czy też pełną ścieżkę z nazwą pliku (na przykład */bin/echo*). Można to osiągnąć, sprawdzając, czy pierwszy znak wartości zmiennej to ukośnik (/). Dlatego trzeba wyodrębnić z całej wartości zmiennej tylko jej pierwszy znak.

Zwróć uwagę, że składnia wyrażenia wyodrębniającego fragment wartości `${var:0:1}`, użytego w wierszu ❶, umożliwia wydzielenie części ciągu znaków, zaczynającej się od zadanego numeru znaku i zawierającej zadaną liczbę znaków (jeżeli długość części nie zostanie określona, zostanie wyodrębniona część od zadanego znaku do końca ciągu). Na przykład wyrażenie `${var:8}` reprezentuje część wartości zmiennej `$var`, począwszy od 8. znaku do końca ciągu, natomiast wyrażenie `${var:8:6}` reprezentuje fragment ciągu od 8. do 14. znaku włącznie. Możesz to sprawdzić na poniższym przykładzie:

```
$ var="Ciekawe rzeczy tu się dzieją..."
$ echo ${var:8}
rzeczy tu się dzieją...
$ echo ${var:8:6}
rzeczy
$
```

W listingu 1.1 powyższe wyrażenie jest użyte w celu sprawdzenia, czy podana ścieżka do pliku zaczyna się od ukośnika, a następnie czy plik ten faktycznie znajduje się w systemie. Jeżeli pierwszym znakiem jest ukośnik, przyjmowane jest założenie, że zmienna zawiera bezwzględną ścieżkę, po czym za pomocą operatora `-x` sprawdzana jest dostępność pliku w systemie ❷. Jeżeli ukośnika nie ma, wtedy wartość zmiennej jest przekazywana funkcji `in_path` ❸, która sprawdza, czy plik znajduje się w którymś z katalogów zapisanych w zmiennej `PATH`.

Uruchomienie skryptu

Aby móc korzystać ze skryptu tak jak ze zwykłego programu, trzeba najpierw na jego końcu wpisać krótki blok poleceń, które będą wykonywały podstawowe operacje, takie jak pobranie danych wpisanych przez użytkownika i przekazanie ich opisanym wyżej funkcjom:

```
if [ $# -ne 1 ] ; then
  echo "Użycie: $0 polecenie" >&2 ; exit 1
fi

checkForCmdInPath "$1"
case $? in
  0 ) echo "$1 znajduje się w katalogach zmiennej PATH."         ;;
  1 ) echo "$1 nie istnieje i nie jest to plik wykonywalny."     ;;
```

```
  2 ) echo "$1 nie znajduje się w katalogach zmiennej PATH."     ;;
esac

exit 0
```

Po wpisaniu powyższego kodu możesz uruchomić skrypt wprost, w sposób pokazany w części „Wyniki". Pamiętaj, aby później, gdy zakończysz pracę nad tym skryptem i będziesz go chciał umieścić w bibliotece swoich skryptów, usunąć powyższy kod lub zamienić go w komentarz, aby nie wprowadzał niepotrzebnego zamieszania.

Wyniki

Aby przetestować skrypt, użyj polecenia `inpath` z nazwami trzech programów: jednego, który jest zapisany w jednym z katalogów zmiennej `PATH`, drugiego, który istnieje, ale znajduje się w innym katalogu, oraz trzeciego, którego nie ma.

Listing 1.2 zawiera przykładowy test skryptu.

Listing 1.2. Test skryptu inpath

```
$ inpath echo
echo znajduje się w katalogach zmiennej PATH.
$ inpath MrEcho
MrEcho nie znajduje się w katalogach zmiennej PATH.
$ inpath /usr/bin/MrEcho
/usr/bin/MrEcho nie istnieje i nie jest to plik wykonywalny.
$
```

Ostatni blok kodu, który wpisałeś, przekłada wynik zwracany przez funkcję `in_path` na czytelny komunikat. Dzięki temu można łatwo sprawdzić, czy skrypt działa poprawnie w każdym z trzech przypadków.

Rozbudowa skryptu

Jeżeli już przy tworzeniu pierwszego skryptu chcesz poczuć się jak bohater, zastosuj zamiast wyrażenia `${var:0:1}` jego bardziej skomplikowaną wersję: `${var%${var#?}}`. Wyrażenie to wykorzystuje metodę wyodrębniania fragmentu danych określoną w standardzie POSIX. Ten na pozór bezsensowny zapis zawiera w rzeczywistości dwa wyrażenia, z których jedno jest zagnieżdżone w drugim. Wewnętrzne wyrażenie `${var#?}` wyodrębnia ze zmiennej `var` fragment całego ciągu z wyjątkiem pierwszego znaku. Znak `#` oznacza usunięcie pierwszego fragmentu ciągu zgodnego z zadanym wzorcem, natomiast znak `?` jest wyrażeniem regularnym reprezentującym dokładnie jeden znak.

Następnie wyrażenie `${var%wzorzec}` wyodrębnia fragment ciągu zawierający wszystkie znaki pozostałe po usunięciu ze zmiennej `var` wszystkich znaków zgodnych ze wzorcem. W tym przypadku usunięte zostały znaki zwrócone przez wewnętrzne wyrażenie, zatem ostatecznym wynikiem jest pierwszy znak ciągu.

Jeżeli powyższy zapis jest dla Ciebie zbyt zagmatwany, możesz wykorzystać opisane wcześniej wyrażenie `${nazwa_zmiennej:początek:koniec}` wyodrębniające ciąg znaków, dostępne w większości powłok (w tym bash, ksh i zsh).

Jeżeli nie odpowiada Ci żadna z powyższych metod wyodrębniania fragmentu danych, to oczywiście możesz wykorzystać polecenie systemowe `$(echo $var | cut -c1)`. Za pomocą skryptów powłoki dany problem, na przykład wyodrębnienie, przekształcenie czy załadowanie danych, można rozwiązać na kilka sposobów. Pamiętaj, że filozofia „wiązania krawata na różne sposoby" nie oznacza, że jedna metoda jest lepsza od innej.

Ponadto, jeżeli chcesz utworzyć odmianę powyższego lub innego skryptu, który będzie rozpoznawał, czy został uruchomiony w wierszu poleceń, czy za pomocą innego skryptu, wpisz w jednym z pierwszych wierszy następujący kod:

```
if [ "$BASH_SOURCE" = "$0" ]
```

Jako ćwiczenie dla Ciebie, drogi czytelniku, pozostawiamy poeksperymentowanie i napisanie pozostałej części kodu!

UWAGA *Bardzo przydatny jest skrypt nr 47, który jest bardzo podobny do powyższego. Uwzględnia on zarówno katalogi zapisane w zmiennej `PATH`, jak również w zmiennych środowiskowych w skrypcie logowania.*

Skrypt 2. Weryfikacja wprowadzanych danych: tylko litery i cyfry

Użytkownicy często nie zwracają uwagi na wskazówki wyświetlane na ekranie i wprowadzają dane niespójne, niewłaściwie formatowane lub o błędnej składni. Jako twórca skryptów powłoki musisz umieć wyszukiwać i sygnalizować tego rodzaju błędy, zanim staną się przyczyną problemów.

Typowy przykład dotyczy wprowadzania nazw plików lub kluczy w bazie danych. Załóżmy, że Twój skrypt prosi użytkownika o wprowadzenie ciągu zawierającego wyłącznie wielkie i małe litery oraz cyfry, bez znaków interpunkcyjnych, spacji ani znaków specjalnych. Czy użytkownik wpisał poprawne dane? To sprawdza skrypt przedstawiony w listingu 1.3.

Kod

Listing 1.3. Skrypt sprawdzający poprawność danych

```
#!/bin/bash
# Skrypt validalnum sprawdzający, czy wprowadzone dane zawierają wyłącznie litery i cyfry.

validAlphaNum()
{
```

```
    # Funkcja sprawdzająca argument. Zwraca 0, jeżeli składa się on z samych
    # wielkich i małych liter oraz cyfr, a 1 w przeciwnym wypadku.

    # Usunięcie wszystkich niedozwolonych znaków.
❶   validchars="$(echo $1 | sed -e 's/[^[:alnum:]]//g')"

❷   if [ "$validchars" = "$1" ] ; then
      return 0
    else
      return 1
    fi
}

# POCZĄTEK GŁÓWNEJ CZĘŚCI SKRYPTU. USUŃ LUB ZAMIEŃ W KOMENTARZ KOD
# PONIŻEJ TEGO WIERSZA,
# JEŻELI TEN SKRYPT BĘDZIE WYKORZYSTYWANY W INNYCH SKRYPTACH.
# ================
/bin/echo -n "Podaj dane: "
read input

# Sprawdzenie danych.
if ! validAlphaNum "$input" ; then
  echo "Dane mogą zawierać tylko litery lub cyfry." >&2
  exit 1
else
  echo "Dane poprawne."
fi

exit 0
```

Jak to działa?

Algorytm powyższego skryptu jest prosty. Najpierw tworzona jest kopia wprowadzonych danych, przekształconych za pomocą polecenia `sed` usuwającego wszystkie niedopuszczalne znaki ❶. Następnie nowe dane są porównywane z oryginalnymi ❷. Jeżeli nie ma między nimi różnicy, to wszystko jest w porządku. W przeciwnym wypadku różnica oznacza, że dane zawierają niedopuszczalne znaki, usunięte za pomocą polecenia przekształcającego, a więc dane są błędne.

Powyższy algorytm działa poprawnie, ponieważ polecenie `sed` usuwa z ciągu wszystkie znaki określone za pomocą wyrażenia `[:alnum:]`. Jest to wyrażenie regularne zgodne ze standardem POSIX, reprezentujące wszystkie znaki alfanumeryczne. Jeżeli przekształcone dane nie są zgodne z oryginalnymi, oznacza to, że we wprowadzonym ciągu znajdują się znaki inne niż alfanumeryczne, a więc wprowadzone dane są błędne. Funkcja zwraca wtedy wartość różną od zera oznaczającą problem. Pamiętaj, że dane mogą zawierać tylko znaki ASCII.

Uruchomienie skryptu

Skrypt jest samodzielnym programem. Prosi użytkownika o wprowadzenie danych, a następnie wyświetla informację, czy są one poprawne. Jednak bardziej praktycznym przykładem zastosowania tego skryptu jest skopiowanie zawartej

w nim funkcji `validAlphaNum` i wklejenie jej na początku innego skryptu albo umieszczenie w bibliotece i wywołanie w sposób pokazany w skrypcie nr 12.

Powyższy kod jest również dobrym przykładem zastosowania techniki tworzenia skryptów. Powinieneś najpierw tworzyć funkcje, a następnie testować je przed zastosowaniem w większych, bardziej złożonych skryptach. W ten sposób zaoszczędzisz sobie mnóstwa kłopotów.

Wyniki

Korzystanie ze skryptu `validalnum` jest bardzo proste. Użytkownik jest proszony o podanie ciągu znaków do sprawdzenia. Listing 1.4 pokazuje, jak traktowane są błędne i poprawne dane.

Listing 1.4. Test skryptu validalnum

```
$ 02-validalnum
Podaj dane: poprawneDANE1234
Dane poprawne.
$ 02-validalnum
Podaj dane: To są na pewno BŁĘDNE dane, 1234
Dane mogą zawierać tylko litery lub cyfry.
```

Rozbudowa skryptu

Metoda „usuń błędne znaki i sprawdź, co zostało" jest dobra, ponieważ jest elastyczna, szczególnie jeżeli zarówno zmienną zawierającą wprowadzone dane, jak i wzorzec (który może być pusty) umieścisz w cudzysłowach, dzięki czemu unikniesz błędów spowodowanych pustymi zmiennymi. Puste zmienne są częstą przyczyną problemów w skryptach, ponieważ z ich powodu pozornie poprawne wyrażenie warunkowe okazuje się błędne i skutkuje wyświetleniem komunikatu. Musisz zawsze pamiętać, że fraza złożona z cudzysłowów nieobejmujących żadnych znaków różni się od pustej frazy.

Chcesz dopuszczać wprowadzanie wyłącznie wielkich liter, spacji, przecinków i kropek? Wystarczy w tym celu zmienić wzorzec w wierszu ❶ na poniższy:

```
sed 's/[^[:upper:] ,.]//g'
```

Do sprawdzania numerów telefonicznych możesz wykorzystać proste wyrażenie (dopuszczające cyfry, spacje, nawiasy i myślniki, przy czym spacje nie mogą znajdować się na początku ani tworzyć sekwencji w środku ciągu):

```
sed 's/[^- [:digit:]\(\)]//g'
```

Jednak jeżeli chcesz ograniczyć dane tylko do cyfr, pamiętaj o pewnej pułapce. Być może chciałbyś użyć następującego wyrażenia:

```
sed 's/[^[:digit:]]//g'
```

Wyrażenie to działa poprawnie w przypadku liczb dodatnich, ale jak można dopuścić wprowadzanie liczb ujemnych? Jeżeli znak odejmowania umieścisz we wzorcu dopuszczalnych znaków, wówczas ciąg `-3-4` będzie potraktowany jako poprawny, choć oczywiście nie reprezentuje poprawnej liczby ujemnej. Skrypt nr 5 pokazuje, jak obsługiwać liczby ujemne.

Skrypt 3. Normalizacja formatów dat

Jednym z problemów towarzyszących tworzeniu skryptów jest różnorodność formatów dat. Ich normalizacja może być średnio lub bardzo trudna. Formatowanie dat należy do najtrudniejszych zadań, ponieważ daty można zapisywać na wiele różnych sposobów. Nawet jeżeli poprosisz użytkownika o wprowadzenie daty w określonym formacie, na przykład dzień-miesiąc-rok, i tak prawdopodobnie wprowadzi on błędne dane, na przykład zamiast numeru miesiąca wpisze jego nazwę, pełną lub skróconą, albo złożoną z samych wielkich liter. Dlatego bardzo przydatna do tworzenia następnych skryptów, szczególnie skryptu nr 7, będzie funkcja normalizująca daty.

Kod

Skrypt przedstawiony w listingu 1.5 normalizuje formaty dat według dość prostych kryteriów: miesiąc musi być określony za pomocą nazwy lub liczby z zakresu od 1 do 12, a rok musi być czterocyfrową liczbą. Znormalizowana data składa się z numeru dnia, nazwy miesiąca (jej trzyliterowego skrótu) i czterocyfrowego roku.

Listing 1.5. Skrypt normdate

```
#!/bin/bash
# Skrypt normdate zamieniający oznaczenie miesiąca na trzyliterowy skrót nazwy.
# Zawiera funkcję wykorzystaną w skrypcie nr 7, valid-date, zwracającą wartość 0, jeżeli data jest poprawna.

monthNumToName()
{
  # Przypisanie zmiennej 'month' odpowiedniej wartości.
  case $1 in
    1 ) month="sty"    ;;  2 ) month="lut"    ;;
    3 ) month="mar"    ;;  4 ) month="kwi"    ;;
    5 ) month="maj"    ;;  6 ) month="cze"    ;;
    7 ) month="lip"    ;;  8 ) month="sie"    ;;
    9 ) month="wrz"    ;;  10) month="paź"    ;;
    11) month="lis"    ;;  12) month="gru"    ;;
    * ) echo "$0: Błędny numer miesiąca $1" >&2; exit 1
  esac
  return 0
}
```

```
# POCZĄTEK GŁÓWNEJ CZĘŚCI SKRYPTU. USUŃ LUB ZAMIEŃ W KOMENTARZ KOD
# PONIŻEJ TEGO WIERSZA,
# JEŻELI TEN SKRYPT BĘDZIE WYKORZYSTYWANY W INNYCH SKRYPTACH.
# ==================
# Weryfikacja danych
if [ $# -ne 3 ] ; then
  echo "Użycie: $0 dzień miesiąc rok" >&2
  echo "Typowe formaty: '3 sierpnia 2002, '3 8 2002'" >&2
  exit 1
fi
if [ $3 -le 99 ] ; then
  echo "$0: rok musi być liczbą czterocyfrową." >&2; exit 1
fi

# Czy miesiąc jest podany w postaci liczby?
❶ if [ -z $(echo $2|sed 's/[[:digit:]]//g')  ]; then
  monthNumToName $2
else
  # Normalizacja nazwy miesiąca: tylko trzy pierwsze litery.
❷   month="$(echo $2|cut -c1-3 | tr '[:upper:]' '[:lower:]')"
fi

echo $1 $month $3

exit 0
```

Jak to działa?

Zwróć uwagę na trzecią instrukcję warunkową ❶. Usuwa ona z wprowadzonych danych wszystkie cyfry, a następnie sprawdza, czy wynikowy ciąg znaków ma zerową długość (parametr `-z`). Jeżeli tak jest, to znaczy, że wprowadzone dane zawierają tylko cyfry, które można przełożyć na nazwę miesiąca za pomocą funkcji `monthNumToName()`. Funkcja ta jednocześnie sprawdza, czy podana liczba jest poprawnym numerem miesiąca. Jeżeli natomiast wprowadzone dane nie zawierają samych cyfr, wtedy przyjmowane jest założenie, że jest to nazwa miesiąca, która jest następnie normalizowana za pomocą skomplikowanej sekwencji poleceń `cut` i `tr` wywoływanych w podpowłoce (tj. sekwencji poleceń umieszczonych wewnątrz znaków `$( i )`, które powodują zastąpienie poleceń wynikami ich działania).

Polecenia wykonywane w podpowłoce ❷ wyodrębniają z wprowadzonych danych trzy pierwsze znaki (polecenie `cut`) i przekształcają je na małe litery (polecenie `tr`). Zwróć uwagę, że nie jest tu sprawdzana poprawność nazwy miesiąca, jak to ma miejsce w przypadku jego numeru.

Uruchomienie skryptu

Aby móc w przyszłości maksymalnie elastycznie korzystać ze skryptu `normdate` w innych skryptach, należy w nim zaimplementować obsługę trzech argumentów, jak w listingu 1.6.

Wyniki

Listing 1.6. Test skryptu normdate

```
$ normdate 3 8 62
normdate: rok musi być liczbą czterocyfrową.
$ normdate 3 8 1962
3 sie 1962
$ normdate 03 sierpnia 1962
03 sie 1962
```

Zwróć uwagę, że powyższy skrypt normalizuje jedynie oznaczenie miesiąca, a zapis dnia i roku pozostawia bez zmian.

Rozbudowa skryptu

Zanim się nadmiernie podekscytujesz różnorodnymi możliwościami rozbudowywania i komplikowania powyższego skryptu, zapoznaj się ze skryptem nr 7, wykorzystującym skrypt `normdate` do weryfikowania poprawności daty.

Jedną z modyfikacji, którą możesz wprowadzić w tym skrypcie, jest umożliwienie wprowadzania dat w formacie DD/MM/RRRR lub DD-MM-RRRR. W tym celu tuż przed pierwszą instrukcją warunkową wpisz poniższy kod:

```
if [ $# -eq 1 ] ; then # Uwzględnienie formatu ze znakami / i -.
  set -- $(echo $1 | sed 's/[\/\-]/ /g')
fi
```

Po wprowadzeniu powyższej zmiany skrypt normalizuje daty podane w następujących formatach:

```
$ normdate 10-6-2000
10 cze 2000
$ normdate 11-marca-1911
11 mar 1911
$ normdate 3/8/1962
3 sie 1962
```

Jeżeli dokładnie przeanalizujesz kod, zauważysz, że można w nim zaimplementować bardziej zaawansowane metody weryfikacji roku, nie mówiąc już o normalizacji dat zapisanych w różnych międzynarodowych formatach. To zadanie zostawiamy Tobie jako ćwiczenie!

Skrypt 4. Czytelne wyświetlanie dużych liczb

Często popełnianym przez programistów błędem jest wyświetlanie wyników obliczeń bez ich uprzedniego sformatowania. Trudno jest przecież użytkownikowi ocenić, ile milionów reprezentuje liczba `43245435` bez odliczenia cyfr od prawej strony do lewej i wstawienia w pamięci odstępów pomiędzy grupami trzech znaków. Skrypt przedstawiony w listingu 1.7 w czytelny sposób formatuje zadane liczby.

Kod

Listing 1.7. Skrypt nicenumber formatuje duże liczby i wyświetla je w czytelny sposób

```
#!/bin/bash
# Skrypt nicenumber wyświetla zadaną liczbę w czytelnej formie.
# Należy w nim określić wartości zmiennych DD (separator dziesiętny) i TD (separator tysięcy).
# Wynik jest zapisywany w zmiennej nicenum, a jeżeli zostanie podany drugi argument, również jest
# wyświetlany w kanale stdout.

nicenumber()
{
  # Zwróć uwagę, że przyjęte jest założenie, iż separatorem dziesiętnym we wprowadzonej wartości
  # jest przecinek.
  # Separatorem dziesiętnym w wyniku jest również przecinek, chyba że za pomocą argumentu -d
  # zostanie określony inny znak.

❶ integer=$(echo $1 | cut -d, -f1)     # Część liczby po lewej stronie przecinka.
❷ decimal=$(echo $1 | cut -d, -f2)     # Część liczby po prawej stronie przecinka.

  # Sprawdzenie, czy wprowadzona liczba zawiera część ułamkową.
  if [ "$decimal" != "$1" ]; then
    # Liczba zawiera część ułamkową, więc należy ją uwzględnić.
    result="${DD:= ','}$decimal"
  fi

  thousands=$integer

❸ while [ $thousands -gt 999 ]; do
❹   remainder=$(($thousands % 1000))     # Trzy najmniej znaczące cyfry.

    # "Reszta" musi składać się z trzech cyfr. Czy trzeba ją uzupełnić zerami?
    while [ ${#remainder} -lt 3 ] ; do   # Dodanie wiodących zer.
      remainder="0$remainder"
    done

    result="${TD:=" "}${remainder}${result}"     # Przygotowanie wyniku od strony
                                                 # prawej do lewej. ❺
    thousands=$(($thousands / 1000))     # Nowa wartość dla zmiennej remainder. ❻
  done

  nicenum="${thousands}${result}"
  if [ ! -z $2 ] ; then
    echo $nicenum
  fi
```

```
}

DD=","   # Separator dziesiętny oddzielający część całkowitą od ułamkowej.
TD=" "   # Separator tysięcy, rozdzielający grupy trzech cyfr.

# POCZĄTEK GŁÓWNEJ CZĘŚCI SKRYPTU. USUŃ LUB ZAMIEŃ W KOMENTARZ KOD
# PONIŻEJ TEGO WIERSZA,
# JEŻELI TEN SKRYPT BĘDZIE WYKORZYSTYWANY W INNYCH SKRYPTACH.
# ================
while getopts "d:t:" opt; do  ❼
  case $opt in
    d ) DD="$OPTARG"    ;;
    t ) TD="$OPTARG"    ;;
  esac
done
shift $(($OPTIND - 1))

# Weryfikacja poprawności danych
if [ $# -eq 0 ] ; then
  echo "Użycie: $(basename $0) [-d c] [-t c] liczba"
  echo "  -d określa separator dziesiętny (domyślnie przecinek)"
  echo "  -t określa separator tysięcy (domyślnie spacja)"
  exit 0
fi

nicenumber $1 1          # Drugi argument powoduje, że funkcja nicenumber  ❽
                         # wyświetla wynik za pomocą polecenia 'echo'.

exit 0
```

Jak to działa?

Rdzeniem skryptu jest pętla `while` zawarta w funkcji `nicenumber` ❸. Wewnątrz niej z liczby zapisanej w zmiennej `thousands` (tysiące) wyodrębniane są kolejno trzy najmniej znaczące cyfry ❹, które są następnie doklejane do czytelnego wyniku ❺. Wartość zapisana w zmiennej `thousands` jest za każdym razem pomniejszana ❻, po czym pętla wykonuje następny obieg, jeżeli jest to konieczne.

W głównej części skryptu najpierw za pomocą polecenia `getopts` analizowane są zadane argumenty ❼, po czym wywoływana jest funkcja `nicenumber` ❽ z ostatnim argumentem podanym przez użytkownika.

Uruchomienie skryptu

Aby przetestować skrypt, należy w jego argumencie podać po prostu dużą liczbę. Skrypt odpowiednio umieści w niej separatory dziesiętny i tysięczny, czy to w postaci domyślnych znaków, czy określonych za pomocą dodatkowych argumentów.

Wynik działania skryptu można wykorzystać w wyświetlanych komunikatach, na przykład:

```
echo "Naprawdę chcesz zapłacić $(nicenumber $cena) zł?"
```

Wyniki

Skrypt `nicenumber` jest prosty w użyciu, ale posiada kilka zaawansowanych opcji. Listing 1.8 pokazuje zastosowanie skryptu do formatowania liczb.

Listing 1.8. Test skryptu nicenumber

```
$ nicenumber 5894625
5 894 625
$ nicenumber 589462532,433
589 462 532,433
$ nicenumber -d. -t, 589462532,433
589,462,532.433
```

Rozbudowa skryptu

W różnych krajach stosowane są różne separatory dziesiętne i tysięczne, które można określić za pomocą argumentów skryptu. Na przykład w Stanach Zjednoczonych właściwe będą argumenty `-d "." -t ","`, w Niemczech i Włoszech `-d "." -t ","`, a w Szwajcarii, w której używa się czterech języków, będą to argumenty `-d "." -t "'"`. Jest to doskonały przykład sytuacji, w której lepszym rozwiązaniem jest tworzenie skryptów sparametryzowanych niż zawierających wpisane na stałe wartości. Dzięki temu ze skryptu może korzystać możliwie największa rzesza użytkowników.

Zwróć uwagę, że separatorem dziesiętnym wartości podawanej w argumencie skryptu musi być przecinek. Gdybyś więc chciał dostosować skrypt do innego separatora, musiałbyś odpowiednio zmienić argumenty polecenia `cut` w wierszach ❶ i ❷, na przykład podając w nim kropkę. Poniższy kod pokazuje, jak to zrobić:

```
integer=$(echo $1 | cut "-d$DD" -f1) # Część liczby po lewej stronie przecinka.
decimal=$(echo $1 | cut "-d$DD" -f2) # Część liczby po prawej stronie przecinka.
```

Powyższy skrypt działa poprawnie, pod warunkiem że separator dziesiętny w podanej liczbie jest taki sam jak separator dziesiętny określony w argumencie. W przeciwnym wypadku pojawi się komunikat o błędzie. Bardziej zaawansowanym rozwiązaniem byłoby wpisanie przed powyższymi dwoma wierszami kodu sprawdzającego, czy oba separatory są takie same. Można to zrobić, stosując tę samą sztuczkę co w skrypcie nr 2, czyli usuwając z zadanej liczby wszystkie cyfry i sprawdzając, co zostanie:

```
separator="$(echo $1 | sed 's/[[:digit:]]//g')"
if [ ! -z "$separator" -a "$separator" != "$DD" ] ; then
  echo "$0: Podany został błędny separator dziesiętny '$separator'." >&2
  exit 1
fi
```

Skrypt 5. Weryfikacja poprawności liczb całkowitych

Jak już przekonałeś się, tworząc skrypt nr 2, weryfikacja poprawności liczb całkowitych jest dziecinnie prosta, chyba że trzeba również uwzględnić liczby ujemne. Problem polega na tym, że taka liczba może zawierać tylko jeden znak odejmowania, na samym początku. Procedura weryfikacyjna zawarta w listingu 1.9 sprawdza, czy liczba ujemna jest poprawna, a dodatkowo — czy zawiera się w zadanym zakresie.

Kod

Listing 1.9. Skrypt validint

```
#!/bin/bash
# Skrypt validint weryfikujący poprawność liczb całkowitych, z ujemnymi włącznie.

validint()
{
  # Sprawdzenie poprawności pierwszego argumentu i porównanie go z minimalną wartością
  # podaną w drugim argumencie i maksymalną w trzecim. Jeżeli zadana wartość
  # nie zawiera się w podanym zakresie lub nie składa się wyłącznie z cyfr, zgłaszany jest błąd.

  number="$1";      min="$2";      max="$3"

❶ if [ -z $number ] ; then
    echo "Podaj liczbę." >&2 ; return 1
  fi

  # Czy pierwszym znakiem jest myślnik?
❷ if [ "${number%${number#?}}" = "-" ] ; then
    testvalue="${number#?}" # Wyodrębnienie do sprawdzenia wszystkich znaków oprócz pierwszego.
  else
    testvalue="$number"
  fi

  # Utworzenie na potrzeby weryfikacji wartości zawierającej wyłącznie cyfry.
❸ nodigits="$(echo $testvalue | sed 's/[[:digit:]]//g')"

  # Sprawdzenie, czy wartość zawiera inne znaki niż cyfry.
  if [ ! -z $nodigits ] ; then
    echo "Błędna wartość! Dozwolone są tylko cyfry, bez przecinka, spacji itp." >&2
    return 1
  fi

❹ if [ ! -z $min ] ; then
    # Czy podana liczba jest mniejsza niż wartość minimalna?
    if [ "$number" -lt "$min" ] ; then
      echo "Liczba $number jest za mała. Minimalna wartość to $min." >&2
      return 1
    fi
  fi
```

```
  if [ ! -z $max ] ; then
    # Czy podana liczba jest większa niż wartość maksymalna?
    if [ "$number" -gt "$max" ] ; then
      echo "Liczba $number jest za duża. Maksymalna wartość to $max." >&2
      return 1
    fi
  fi
  return 0
}
```

Jak to działa?

Weryfikacja poprawności liczby całkowitej jest prosta, ponieważ sprawdzana wartość musi składać się wyłącznie z cyfr (od 0 do 9), z ewentualnym jednym znakiem odejmowania na początku. Jeżeli funkcja `validint()` zostanie wywołana z argumentami określającymi minimalną lub maksymalną wartość, wtedy dodatkowo funkcja sprawdza, czy liczba mieści się w zadanym zakresie.

Funkcja testuje, czy użytkownik w ogóle podał argumenty skryptu ❶ (jest to kolejny przypadek wyrażenia, w którym użycie zmiennej nieujętej w cudzysłowy może spowodować błąd i wyświetlenie komunikatu). Następnie sprawdza, czy na początku podanej wartości znajduje się znak odejmowania ❷, po czym tworzy kopię zadanej wartości z usuniętymi cyframi ❸. Jeżeli nowa wartość nie jest pustym ciągiem znaków, wtedy wynik testu jest negatywny.

Jeżeli użytkownik wprowadził poprawną liczbę, wtedy jest ona porównywana z wartościami minimalną i maksymalną ❹. Jeżeli liczba mieści się w zadanym zakresie, wtedy funkcja zwraca wartość 1, w przeciwnym wypadku zwraca 0.

Uruchomienie skryptu

Cały skrypt jest funkcją, którą można skopiować i umieścić w innym skrypcie lub w bibliotece funkcji. Aby użyć go tak jak polecenia, na jego końcu dopisz kod z listingu 1.10.

Listing 1.10. Dodatkowe wiersze kodu umożliwiające korzystanie ze skryptu jak z polecenia

```
# Weryfikacja poprawności danych
if validint "$1" "$2" "$3" ; then
  echo "Podana wartość jest poprawną liczbą całkowitą zawartą w zadanym
zakresie."
fi
```

Wyniki

Po dopisaniu do skryptu kodu z listingu 1.10 możesz używać skryptu tak, jak pokazuje listing 1.11.

Listing 1.11. Tekst skryptu validint

```
$ validint 1234,3
Błędna wartość! Dozwolone są tylko cyfry, bez przecinka, spacji itp.
$ validint 103 1 100
```

```
Liczba 103 jest za duża. Maksymalna wartość to 100.
$ validint -17 0 25
Liczba -17 jest za mała. Minimalna wartość to 0.
$ validint -17 -20 25
Poddana wartość jest poprawną liczbą całkowitą zawartą w zadanym zakresie.
```

Rozbudowa skryptu

Zwróć uwagę, że wyrażenie w wierszu ❷ sprawdza, czy pierwszym znakiem jest znak odejmowania:

```
if [ "${number%${number#?}}" = "-" ] ; then
```

Jeżeli tak, wtedy zmiennej `testvalue` przypisywana jest numeryczna część zadanej wartości, z której następnie usuwane są wszystkie cyfry. Uzyskana nowa wartość jest dalej sprawdzana.

Być może kusi Cię, aby w niektórych zagnieżdżonych instrukcjach `if` użyć operatora logicznego *oraz* (`-a`), na przykład w następujący sposób:

```
if [ ! -z $min -a "$number" -lt "$min" ] ; then
  echo "Liczba $number jest za mała. Minimalna wartość to $min." >&2
  exit 1
fi
```

Powyższy kod nie będzie jednak działał poprawnie, ponieważ nawet jeżeli pierwszy wyraz w sprawdzanym wyrażeniu będzie miał wartość *fałsz*, to i tak nie będziesz miał pewności, że drugi wyraz nie zostanie sprawdzony (jak to się dzieje w innych językach programowania). Oznacza to, że tego typu zapis może skutkować porównaniem błędnych lub nieoczekiwanych wartości. Nie powinno tak być, ale w ten sposób działają skrypty powłoki.

Skrypt 6. Weryfikacja poprawności liczb zmiennoprzecinkowych

Na pierwszy rzut oka proces sprawdzania poprawności liczb zmiennoprzecinkowych (rzeczywistych) przy wszystkich zawiłościach i niuansach skryptów powłoki może wydawać się czymś bardzo żmudnym. Zauważ jednak, że liczbę zmiennoprzecinkową można potraktować jako dwie liczby całkowite oddzielone przecinkiem. Jeżeli przypomnisz sobie, że zewnętrzne skrypty (np. `validint`) można wykorzystywać w poleceniach, wtedy okaże się, że sprawdzanie poprawności liczb zmiennoprzecinkowych jest zaskakująco prostym zadaniem. W skrypcie przedstawionym w listingu 1.12 przyjęte zostało założenie, że będzie on wywoływany w tym samym katalogu, w którym znajduje się skrypt `validint`.

Kod

Listing 1.12. Skrypt validfloat

```
#!/bin/bash
# Skrypt validfloat sprawdza, czy podana wartość reprezentuje liczbę zmiennoprzecinkową.
# Pamiętaj, że liczba nie może być zapisana w formacie naukowym (np. 1.304e5).

# Aby sprawdzić, czy podana wartość reprezentuje liczbę zmiennoprzecinkową,
# należy podzielić ją na dwie części: całkowitą i ułamkową.
# Następnie należy sprawdzić, czy pierwsza część reprezentuje liczbę całkowitą, a druga — liczbę
# całkowitą nieujemną.
# W ten sposób wartość -30,5 będzie oceniona jako poprawna, a -30,-8 jako błędna liczba.

# Aby w jednym skrypcie wykorzystać inny, należy poprzedzić jego nazwę kropką (.). To proste.

. validint     # Sposób wywoływania skryptu validint w powłoce bash.

validfloat()
{
  fvalue="$1"

  # Sprawdzenie, czy podana wartość zawiera przecinek.
❶ if [ ! -z $(echo $fvalue | sed 's/[^,]//g') ] ; then

    # Wyodrębnienie części wartości po lewej stronie przecinka (np. '3' z liczby '3,14').
❷   decimalPart="$(echo $fvalue | cut -d, -f1)"

    # Wyodrębnienie części wartości po prawej stronie przecinka (np. '14' z '3,14').
❸   fractionalPart="${fvalue#*\,}"

    # Sprawdzenie najpierw części dziesiętnej, czyli znajdującej się po lewej stronie przecinka.

❹   if [ ! -z $decimalPart ] ; then
      # Znak "!" oznacza negację wyniku wyrażenia, więc poniższy zapis oznacza "jeżeli wyrażenie
      # NIE jest liczbą całkowitą".
      if ! validint "$decimalPart" "" "" ; then
        return 1
      fi
    fi

    # Sprawdzenie części ułamkowej (po prawej stronie przecinka). Przede wszystkim na początku
    # nie może być znaku odejmowania, jak np. w wartości 33,-11, więc należy sprawdzić, czy w tej
    # części jest znak '-'.
❺   if [ "${fractionalPart%${fractionalPart#?}}" = "-" ] ; then
      echo "Błędna liczba zmiennoprzecinkowa: po przecinku nie może" \
         "być znaku '-'." >&2       # Zapis >&2 powoduje wysłanie tekstu do kanału stderr.
      return 1
    fi
    if [ "$fractionalPart" != "" ] ; then
      # Jeżeli część ułamkowa NIE jest poprawną liczbą całkowitą...
      if ! validint "$fractionalPart" "0" "" ; then
        return 1
      fi
    fi
```

```
  else
    # Jeżeli cała wartość składa się tylko ze znaku "-", to też nie jest dobrze.
❻   if [ "$fvalue" = "-" ] ; then
      echo "Błędny format liczby zmiennoprzecinkowej." >&2 ; return 1
    fi

    # Na koniec sprawdzamy, czy pozostałe cyfry reprezentują poprawną liczbę całkowitą.
    if ! validint "$fvalue" "" "" ; then
      return 1
    fi
  fi

  return 0
}
```

Jak to działa?

Powyższy skrypt sprawdza najpierw, czy podana wartość zawiera przecinek ❶. Jeżeli nie, oznacza to, że nie jest to liczba zmiennoprzecinkowa. Następnie do dalszej analizy wyodrębniane są części całkowita ❷ i ułamkowa ❸ zadanej wartości, po czym skrypt sprawdza, czy część całkowita (po *lewej* stronie przecinka) reprezentuje poprawną liczbę całkowitą ❹. Dalsza część skryptu jest bardziej skomplikowana, ponieważ najpierw sprawdza, czy wartość nie zawiera dodatkowego znaku odejmowania ❺ (aby zidentyfikować dziwolągi takie jak na przykład 17,-30), a następnie czy część ułamkowa (po *prawej* stronie przecinka) reprezentuje poprawną liczbę całkowitą.

Ostatnia operacja ❻ polega na sprawdzeniu, czy użytkownik nie wprowadził samego znaku odejmowania i przecinka (na pewno zgodzisz się, że byłoby to bardzo osobliwe).

Wszystko w porządku? W takim razie można zwrócić wartość 0 oznaczającą, że podana przez użytkownika wartość jest poprawną liczbą zmiennoprzecinkową.

Uruchomienie skryptu

Jeżeli wywołana funkcja nie wyświetli komunikatu o błędzie, to zwróci wartość 0, oznaczającą, że zadana wartość jest poprawną liczbą zmiennoprzecinkową. Skrypt możesz przetestować, dopisując na jego końcu poniższy kod:

```
if validfloat $1 ; then
  echo "$1 jest poprawną liczbą zmiennoprzecinkową."
fi

exit 0
```

Jeżeli powyższy skrypt wyświetli komunikat o błędzie, sprawdź, czy katalog, w którym zapisany jest skrypt `validint`, znajduje się w zmiennej `PATH`. Ewentualnie skopiuj zawartość tego skryptu i wklej ją bezpośrednio do powyższego skryptu.

Wyniki

Argumentem skryptu `validfloat` jest tylko liczba, której poprawność jest sprawdzana. Listing 1.13 przedstawia przykład weryfikacji poprawności kilku danych.

Listing 1.13. Test skryptu validfloat

```
$ validfloat 1234,56
1234,56 jest poprawną liczbą zmiennoprzecinkową.
$ validfloat -1234,56
-1234,56 jest poprawną liczbą zmiennoprzecinkową.
$ validfloat -,75
-,75 jest poprawną liczbą zmiennoprzecinkową.
$ validfloat -11,-12
Błędna liczba zmiennoprzecinkowa: po przecinku nie może być znaku '-'.
$ validfloat 1,0344e22
Błędna wartość! Dozwolone są tylko cyfry, bez przecinka, spacji itp.
```

Jeżeli oprócz powyższych pojawiają się jakieś inne komunikaty, prawdopodobnie przyczyną jest dodatkowy kod, który dopisałeś do poprzedniego skryptu, ale na koniec zapomniałeś go usunąć. Wróć do skryptu nr 5 i sprawdź, czy usunąłeś z niego lub zamieniłeś w komentarz kilka ostatnich wierszy.

Rozbudowa skryptu

Ciekawym rozszerzeniem skryptu byłaby możliwość sprawdzania liczb zapisanych w naukowym formacie, pokazanym na powyższym listingu. Nie powinno to być zbyt trudne. Najpierw należałoby sprawdzić, czy dane zawierają literę `e` lub `E`, a następnie podzielić je na trzy części: całkowitą (zawierającą tylko jedną cyfrę), ułamkową i wykładniczą. Następnie trzeba byłoby sprawdzić, czy każda z tych części reprezentuje poprawną liczbę całkowitą.

Jeżeli chcesz uniemożliwić podawanie liczb bez zera przed przecinkiem, musisz zmienić instrukcję warunkową ❻ w listingu 1.12. Dziwne formaty stosuj jednak z ostrożnością.

Skrypt 7. Weryfikacja poprawności daty

W skryptach, które przetwarzają daty, jedną z najtrudniejszych, ale też najważniejszych funkcjonalności jest sprawdzanie, czy podana data oznacza poprawną datę kalendarzową. Zadanie nie jest takie trudne, jeżeli nie trzeba uwzględniać lat przestępnych, ponieważ liczba dni w każdym miesiącu jest niezmienna każdego roku. Wystarczy w takim przypadku przygotować tabelę zawierającą liczbę dni każdego miesiąca i na jej podstawie sprawdzać liczbę dni w podanej dacie. Jednak aby uwzględnić lata przestępne, trzeba zakodować w skrypcie dodatkowy algorytm, i tu zadanie zaczyna się komplikować.

Zasady określające, czy dany rok jest przestępny, czy nie, są następujące:

- rok, który nie jest podzielny przez 4, *nie* jest rokiem przestępnym;
- rok, który jest podzielny przez 4 i przez 400, *jest* rokiem przestępnym;
- rok, który jest podzielny przez 4, nie jest podzielny przez 400 i jest podzielny przez 100, *nie* jest rokiem przestępnym;
- każdy inny rok podzielny przez 4 *jest* rokiem przestępnym.

Gdy będziesz analizował kod przedstawiony w listingu 1.14, zwróć uwagę, że wykorzystany w nim został skrypt `normdate` do normalizowania formatu daty przed jej dalszym sprawdzaniem.

Kod

Listing 1.14. Skrypt valid-date

```
#!/bin/bash
# Skrypt valid-date sprawdza poprawność daty z uwzględnieniem lat przestępnych.

PATH=.:$PATH

exceedsDaysInMonth()
{
  # Ta funkcja zwraca 0, jeżeli podany w argumencie numer dnia miesiąca jest mniejszy lub równy
  # liczbie dni w danym miesiącu.
  # W przeciwnym wypadku funkcja zwraca 1.
❶ case $(echo $2|tr '[:upper:]' '[:lower:]') in
    sty* ) days=31    ;;  lut* ) days=28    ;;
    mar* ) days=31    ;;  kwi* ) days=30    ;;
    maj* ) days=31    ;;  cze* ) days=30    ;;
    lip* ) days=31    ;;  sie* ) days=31    ;;
    wrz* ) days=30    ;;  paź* ) days=31    ;;
    lis* ) days=30    ;;  gru* ) days=31    ;;
    * ) echo "$0: Błędna nazwa miesiąca $2." >&2; exit 1
  esac
   if [ $1 -lt 1 -o $1 -gt $days ] ; then
     return 1
   else
     return 0    # Numer dnia jest poprawny.
   fi
}

isLeapYear()
{
  # Ta funkcja zwraca 0, jeżeli dany rok jest rokiem przestępnym.
  # W przeciwnym wypadku zwraca 1.
  # Zasady sprawdzania, czy rok jest przestępny, są następujące:
  # 1. rok niepodzielny przez 4 nie jest przestępny
  # 2. rok podzielny przez 4 i 400 jest przestępny
  # 3. rok podzielny przez 4, niepodzielny przez 400 i podzielny przez 100 jest przestępny
  # 4. każdy inny rok podzielny przez 4 jest przestępny
  year=$1
```

```
❷   if [ "$((year % 4))" -ne 0 ] ; then
      return 1 # rok nieprzestępny
    elif [ "$((year % 400))" -eq 0 ] ; then
      return 0 # rok przestępny
    elif [ "$((year % 100))" -eq 0 ] ; then
      return 1
    else
       return 0
    fi
  }

  # POCZĄTEK GŁÓWNEJ CZĘŚCI SKRYPTU. USUŃ LUB ZAMIEŃ W KOMENTARZ KOD
  # PONIŻEJ TEGO WIERSZA,
  # JEŻELI TEN SKRYPT BĘDZIE WYKORZYSTYWANY W INNYCH SKRYPTACH.
  # ==================
  if [ $# -ne 3 ] ; then
    echo "Użycie: $0 dzień miesiąc rok" >&2
    echo "Typowe formaty: '3 sierpnia 2002', '3 8 2002'." >&2
    exit 1
  fi

  # Normalizacja daty i zapisanie wyniku w celu jego dalszego sprawdzenia.
❸ newdate="$(normdate "$@")"
  if [ $? -eq 1 ] ; then
    exit 1          # Błąd zgłoszony przez skrypt normdate.
  fi

  # Podzielenie znormalizowanej daty. Pierwsza część zawiera numer dnia, druga nazwę miesiąca,
  # trzecia numer roku.

  day="$(echo $newdate | cut -d\  -f1)"
  month="$(echo $newdate | cut -d\  -f2)"
  year="$(echo $newdate | cut -d\  -f3)"

  # Po znormalizowaniu daty sprawdzamy, czy numer dnia jest poprawny (np. nie jest to 36 stycznia).
  if ! exceedsDaysInMonth "$1" $month ; then
    if [ "$month" = "lut" -a "$1" -eq "29" ] ; then
      if ! isLeapYear $3 ; then
❹       echo "$0: $3 nie jest rokiem przestępnym, więc luty nie ma 29 dni." >&2
        exit 1
      fi
    else
      echo "$0: błędny numer dnia: $month nie ma $1 dni." >&2
      exit 1
    fi
  fi

  echo "Data $newdate jest poprawna."

  exit 0
```

Jak to działa?

Napisanie takiego skryptu jest nie lada przyjemnością, ponieważ wymaga zastosowania wielu instrukcji warunkowych sprawdzających, czy numer dnia miesiąca jest poprawny, rok jest przestępny itp. Zastosowane reguły nie określają jedynie, że miesiąc musi mieć numer od 1 do 12 albo dzień w miesiącu numer od 1 do 31. W skrypcie zostały zdefiniowane funkcje, dzięki którym łatwiej było napisać uporządkowany i bardziej czytelny kod.

Funkcja `exceedsDaysInMonth()` (numer dnia przekracza liczbę dni miesiąca) bardzo ogólnie sprawdza dzień i miesiąc (na przykład nazwa STYCZEŃ jest kwalifikowana jako poprawna). W instrukcji `case` ❶ nazwa jest zamieniana na małe litery i na jej podstawie określana jest liczba dni w danym miesiącu. Przyjęte jest jednak założenie, że luty ma 28 dni.

W celu uwzględnienia w analizie lat przestępnych została zdefiniowana funkcja `isLeapYear()` (to jest rok przestępny) wykonująca obliczenia matematyczne w celu sprawdzenia, czy w danym roku luty ma 29 dni ❷.

W głównej części skryptu wprowadzone dane są normalizowane za pomocą opisanego wcześniej skryptu `normdate` ❸, a następnie dzielone na trzy części, zapisywane w zmiennych `$day` (dzień), `$month` (miesiąc) i `$year` (rok). Następnie wywoływana jest funkcja `exceedsDaysInMonth()` sprawdzająca, czy numer dnia miesiąca jest poprawny (tzn. czy nie jest to np. 31 września). Osobna instrukcja warunkowa sprawdza, czy wprowadzone dane reprezentują datę 29 lutego. Wykorzystuje ona funkcję `isLeapYear()` ❹ i w razie potrzeby wyświetla odpowiedni komunikat. Jeżeli podana data przejdzie pomyślnie wszystkie testy, oznacza to, że jest poprawna!

Uruchomienie skryptu

Aby przetestować skrypt (jak na listingu 1.15), wpisz w wierszu poleceń jego nazwę i datę w formacie dzień-miesiąc-rok. Miesiąc możesz pisać w postaci trzyliterowego skrótu lub liczby. Rok musi być liczbą czterocyfrową.

Wyniki

Listing 1.15. Test skryptu valid-date

```
$ valid-date 3 sierpnia 1960
Data 3 sie 1960 jest poprawna.
$ valid-date 31 9 2001
valid-date: błędny numer dnia: wrz nie ma 31 dni.
$ valid-date 29 lut 2004
Data 29 lut 2004 jest poprawna.
$ valid-date 29 lut 2014
valid-date: 2014 nie jest rokiem przestępnym, więc luty nie ma 29 dni.
```

Rozbudowa skryptu

Podobny algorytm można zastosować do sprawdzania poprawności zapisu czasu, wykorzystującego format 24-godzinny lub sufiksy am/pm. Wprowadzony czas należy podzielić według dwukropków, następnie sprawdzić, czy liczby minut i sekund (jeżeli zostały określone) mieszczą się w przedziale od 0 do 59 oraz czy godzina zawiera się w przedziale 0 – 23, ewentualnie w przedziale 1 – 12 i zawiera sufiks am/pm.

Zupełnie innym sposobem sprawdzania, czy dany rok jest rokiem przestępnym, jest wykorzystanie polecenia `date`, dostępnego w systemach Unix i GNU/Linux. Sprawdź efekt użycia następującego polecenia:

```
$ date -d 12/31/1996 +%j
```

Jeżeli w systemie zaimplementowana jest nowsza, ulepszona wersja polecenia `date`, wtedy na ekranie pojawi się liczba 366. Starsza wersja spowoduje wyświetlenie komunikatu o błędzie. Zastanów się, czy wynik nowszej wersji polecenia możesz wykorzystać do napisania dwuwierszowej funkcji sprawdzającej, czy dany rok jest rokiem przestępnym!

Ponadto opisany tu skrypt jest dość tolerancyjny pod względem poprawności nazw miesięcy. Dopuszcza nazwę miesiąca „lutowany”, ponieważ sprawdza tylko jej trzy pierwsze litery ❶. Możesz to zmienić, aby skrypt dopuszczał tylko skrócone (cze) i pełne nazwy miesięcy (czerwiec), albo nawet często popełniane błędy (czerwień). Wszystko to można łatwo osiągnąć, jeżeli tylko jest motywacja!

Skrypt 8. Lepsza implementacja polecenia echo

Jak wspomnieliśmy w sekcji „Czym jest POSIX?”, w większości obecnych implementacji systemów Unix i GNU/Linux polecenie `echo` obsługuje argument `-n` zapobiegający przejściu kursora do nowego wiersza. Jednak w niektórych implementacjach tak nie jest. Czasami w celu osiągnięcia powyższego efektu należy stosować specjalny znak `\c`, a czasami po prostu nie da się zapobiec przejściu do nowego wiersza w ogóle.

Sprawdzenie, czy polecenie `echo` jest poprawnie zaimplementowane, jest proste. Wystarczy w tym celu wpisać poniższe polecenia i zobaczyć, jaki jest wynik:

```
$ echo -n "Siała baba mak,"; echo " nie wiedziała jak."
```

Jeżeli polecenie `echo` poprawnie obsługuje argument `-n`, wtedy wynik będzie następujący:

```
Siała baba mak, nie wiedziała jak.
```

W przeciwnym wypadku na ekranie pojawi się:

```
-n Siała baba mak,
nie wiedziała jak.
```

Prezentowanie użytkownikowi wyników działania skryptu w odpowiednim formacie jest bardzo ważne, a jeszcze ważniejsze, gdy skrypty stają się bardziej interaktywne. Dlatego tutaj zaimplementujesz alternatywną wersję polecenia `echo`, o nazwie `echon`, które nigdy nie będzie przenosiło kursora do nowego wiersza. W ten sposób uzyskasz niezawodne polecenie, które będziesz mógł zawsze stosować, gdy będzie potrzebna funkcjonalność polecenia `echo -n`.

Kod

Ten nietypowy problem można rozwiązać na tyle sposobów, ile stron ma ta książka. Nasz ulubiony jest bardzo zwięzły: polega na przefiltrowaniu podanych danych za pomocą poleceń `awk` i `printf`, jak w listingu 1.16.

Listing 1.16. Alternatywne do echo polecenie wykorzystujące polecenia awk i printf

```
echon()
{
  echo "$*" | awk '{ printf "%s", $0 }'
}
```

Być może jednak będziesz chciał uniknąć dodatkowego obciążenia systemu, jakie wprowadza polecenie `awk`. Jeżeli dostępne jest polecenie `printf`, możesz użyć go do filtrowania danych w funkcji `echon`, jak w listingu 1.17.

Listing 1.17. Alternatywne do echo polecenie wykorzystujące tylko polecenie printf

```
 echon()
{
  printf "%s" "$*"
}
```

Ale co robić, gdy polecenie `printf` nie jest dostępne i nie chcesz używać polecenia `awk`? Wtedy do usunięcia ostatniego znaku powodującego przejście do nowego wiersza możesz wykorzystać polecenie `tr`, jak w listingu 1.18.

Listing 1.18. Alternatywne do echo polecenie wykorzystujące polecenie tr

```
echon()
{
  echo "$*" | tr -d '\n'
}
```

Powyższy sposób jest prosty i skuteczny, powinien też być uniwersalny.

Uruchomienie skryptu

Gdy po prostu dodasz do zmiennej `PATH` nazwę katalogu, w którym znajduje się skrypt `echon`, wtedy będziesz mógł wszystkie polecenia `echo` zastąpić poleceniem `echon`. Po wyświetleniu tekstu kursor nie będzie już przenoszony do nowego wiersza.

Wyniki

Skrypt `echon` ma jeden argument, który wyświetla na ekranie. Skryptu tego można użyć do wyświetlenia użytkownikowi prośby o podanie danych, jak w listingu 1.19.

Listing 1.19. Test skryptu echon

```
$ echon "Podaj współrzędne satelity: "
Podaj współrzędne satelity: 12,34
```

Rozbudowa skryptu

Nie oszukujmy się. Istnienie powłok, w których polecenie `echo` obsługuje argument `-n`, a także innych powłok, w których trzeba na końcu wiersza wpisywać znak `\c`, oraz powłok, w których nie da się uniknąć przeniesienia kursora do nowego wiersza, jest ogromnym problemem dla twórców skryptów. Aby ujednolicić skrypty, można napisać funkcję, która na podstawie wyświetlanego wyniku będzie automatycznie sprawdzała, z jakim przypadkiem ma do czynienia, i odpowiednio modyfikowała swoje działanie. Można na przykład użyć polecenia `echo -n test | wc -c` i sprawdzać, czy wynik składa się z czterech znaków („test"), pięciu („test" i koniec wiersza), siedmiu („-n test"), czy ośmiu („-n test" i koniec wiersza).

Skrypt 9. Zmiennoprzecinkowy kalkulator o konfigurowanej dokładności

Jedną z najczęściej stosowanych w skryptach sekwencji znaków jest `$(( ))`, która umożliwia wykonywanie podstawowych obliczeń matematycznych. Powyższa sekwencja jest bardzo przydatna, umożliwia m.in. zwiększanie zmiennych licznikowych, dodawanie liczb, odejmowanie, mnożenie, dzielenie lub obliczanie reszty z dzielenia (modulo). Nie obsługuje jednak liczb ułamkowych. Zatem poniższe polecenie zwróci wartość `0`, a nie `0.5`:

```
echo $(( 1 / 2 ))
```

Jeżeli więc trzeba obliczyć wynik z większą dokładnością, wtedy pojawia się wyzwanie. Niewiele jest dobrych kalkulatorów, których można używać w wierszu poleceń. Jednym z nich jest `bc`, dziwaczny program, o którego istnieniu wie niewielu użytkowników systemu Unix. Jest to program, którego historia sięga początków systemu Unix, będący rzekomo precyzyjnym kalkulatorem, który jednak wyświetla tajemnicze komunikaty o błędach, nie podaje żadnych podpowiedzi i przyjęte jest w nim założenie, że ten, kto go używa, wie, co robi. Ale to nie jest problem. Można napisać skrypt, przedstawiony w listingu 1.20, dzięki któremu program `bc` jest bardziej przyjazny w użyciu.

Kod

Listing 1.20. Skrypt scriptbc

```
#!/bin/bash
# Skrypt scriptbc wykorzystujący polecenie bc zwracające wynik obliczeń.

❶ if [ "$1" = "-p" ] ; then
    precision=$2
    shift 2
  else
❷   precision=2   # Domyślna dokładność.
  fi

❸ bc -q -l << EOF
  scale=$precision
  $*
  quit
  EOF

  exit 0
```

Jak to działa?

Zapis `<<` ❸ umożliwia potraktowanie treści skryptu tak, jakby została wpisana w wierszu poleceń. W tym przypadku jest to prosty mechanizm przekazywania poleceń programowi `bc` w postaci tzw. dokumentu miejscowego (ang. *here document*). Sekwencja znajdująca się po znakach `<<` definiuje koniec dokumentu. W tym przypadku jest to ciąg **EOF** (ang. *end of file*, koniec pliku).

Powyższy skrypt pokazuje również, jak można wykorzystać argumenty do tworzenia bardziej elastycznych poleceń. Za pomocą argumentu `-p` ❶ można określić dokładność obliczeń. Jeżeli argument ten nie zostanie użyty, zastosowana będzie domyślna dokładność określona poleceniem `scale=2` ❷.

Podczas korzystania z polecenia `bc` bardzo ważna jest znajomość różnic pomiędzy poleceniami `length` (długość) i `scale` (skala). Za pomocą polecenia `length` określa się całkowitą liczbę cyfr w liczbie, natomiast `scale` służy do określania liczby cyfr po przecinku. Na przykład liczba 10.25 ma długość 4 i skalę 2, a liczba 3.14159 długość 6 i skalę 5.

Domyślnie liczby w poleceniu `bc` mają zmienną długość, ale z tego powodu skala jest równa 0. Zatem polecenie to bez wprowadzenia żadnych zmian w jego konfiguracji działa dokładnie tak samo jak sekwencja `$(( ))`. Ale po użyciu opcji `scale` ujawnia się jego ukryta siła, co pokazuje poniższy przykład, w którym obliczana jest liczba tygodni pomiędzy latami 1962 i 2002 (nie uwzględniając lat przestępnych):

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
scale=10
(2002-1962)*365
14600
14600/7
2085.7142857142
quit
```

Aby móc wykorzystać możliwości programu `bc` w wierszu poleceń, skrypt ukrywa wyświetlane informacje o prawach autorskich, niemniej w większości implementacji program `bc` nie wyświetla tych informacji, jeżeli nie jest używany w terminalu (nie wykorzystuje strumienia `stdin`). Skrypt ustawiana rozsądną dokładność obliczeń, przekazuje podane wyrażenie programowi `bc`, a następnie kończy jego działanie poleceniem `quit`.

Uruchomienie skryptu

Aby przetestować skrypt, uruchom go, podając jako argument wyrażenie matematyczne, jak w listingu 1.21.

Wyniki

Listing 1.21. Test skryptu scriptbc

```
$ scriptbc 14600/7
2085.71
$ scriptbc -p 10 14600/7
2085.7142857142
```

Skrypt 10. Blokowanie plików

Każdy skrypt, który odczytuje lub zapisuje dane ze współdzielonego pliku, musi w niezawodny sposób blokować ten plik, aby inne skrypty przypadkowo nie napisały nowych danych. Popularną metodą realizacji tego celu jest tworzenie osobnego **pliku blokady** dla każdego modyfikowanego pliku. Plik blokady pełni

rolę wskaźnika informującego, że główny plik jest niedostępny, ponieważ jest używany przez jakiś skrypt. Inny skrypt może wtedy regularnie sprawdzać, czy blokada została usunięta, co będzie oznaczało, że plik można z powrotem edytować.

Blokowanie plików jest trudną sztuką, ponieważ wiele pozornie niezawodnych rozwiązań nie sprawdza się w praktyce. Na przykład typowym rozwiązaniem jest poniższy kod:

```
while [ -f $lockfile ] ; do
  sleep 1
done
touch $lockfile
```

Kod wydaje się poprawny, prawda? Wykonuje on pętlę dotąd, aż plik blokady zniknie. Następnie tworzy własny plik blokady, aby móc bezpiecznie modyfikować główny plik. Jeżeli inny skrypt z taką samą pętlą wykryje plik blokady, będzie czekał, aż plik zniknie. Jednak w praktyce tak się nie dzieje. Wyobraź sobie, co się stanie, gdy zaraz po zakończeniu wykonywania pętli, ale przed instrukcją `touch` skrypt zostanie usunięty z kolejki procesów, a zanim z powrotem do niej wróci, zostanie uruchomiony inny skrypt.

Jeżeli nie wiesz, o czym piszemy, przypomnij sobie, że komputer tylko pozornie wykonuje jedną operację po drugiej. W rzeczywistości wykonuje wiele programów jednocześnie. Realizuje małą część jednego programu, następnie przełącza się na inny, którego wykonuje małą część, i wraca z powrotem do pierwszego programu. Problem z powyższym skryptem polega na tym, że pomiędzy chwilą, gdy skończy on sprawdzać, czy plik blokady istnieje, a utworzeniem własnego pliku blokady system może przełączyć się na inny skrypt, który może zgodnie z założeniem sprawdzić, czy plik blokady istnieje. Ponieważ tego pliku nie będzie, może utworzyć własny. Następnie system może powrócić do pierwszego skryptu i wznowić jego wykonywanie od polecenia `touch`. W efekcie oba skrypty będą „myślały", że mają wyłączny dostęp do głównego pliku, a więc dojdzie do sytuacji, której właśnie chciałeś zapobiec.

Na szczęście Stephen van den Berg i Philip Guenther, autorzy programu procmail do filtrowania poczty, utworzyli również narzędzie `lockfile` umożliwiające bezpieczne i niezawodne posługiwanie się blokadami plików w skryptach powłoki.

Wiele dystrybucji systemu Unix, z GNU/Linux i OS X włącznie, zawiera zainstalowane już narzędzie `lockfile`. Możesz to sprawdzić, wpisując po prostu polecenie `man 1 lockfile`. Jeżeli pojawi się tekst opisu narzędzia, to masz szczęście! W skrypcie w listingu 1.22 przyjęte jest założenie, że polecenie `lockfile` jest dostępne. Opisane w dalszej części książki skrypty wymagają do swego poprawnego działania niezawodnego mechanizmu blokowania plików, zaimplementowanego w skrypcie 10. Upewnij się więc, że w Twoim systemie dostępne jest polecenie `lockfile`.

Kod

Listing 1.22. Skrypt filelock

```
#!/bin/bash
# Skrypt filelock - elastyczny mechanizm blokowania plików.

retries="10"              # Domyślna liczba prób.
action="lock"             # Domyślna operacja.
nullcmd="`which true`"    # Puste polecenie dla lockfile.

❶ while getopts "lur:" opt; do
    case $opt in
      l ) action="lock"      ;;
      u ) action="unlock"    ;;
      r ) retries="$OPTARG"  ;;
    esac
  done
❷ shift $(($OPTIND - 1))

  if [ $# -eq 0 ] ; then # Wyświetlenie wielowierszowego komunikatu w stdout.
    cat << EOF >&2
  Użycie: $0 [-l|-u] [-r retries] LOCKFILE
  Gdzie -l oznacza żądanie założenia blokady (domyślna operacja),
  -u żądanie zdjęcia blokady, -r X oznacza maksymalną liczbę prób
  przed zgłoszeniem problemu (domyślnie = $retries).
  EOF
    exit 1
  fi

  # Sprawdzenie, czy jest dostępne polecenie lockfile.

❸ if [ -z "$(which lockfile | grep -v '^no ')" ] ; then
    echo "$0 błąd: narzędzie 'lockfile' nie istnieje w katalogach PATH." >&2
    exit 1
  fi
❹ if [ "$action" = "lock" ] ; then
    if ! lockfile -1 -r $retries "$1" 2> /dev/null; then
      echo "$0: błąd: plik blokady nie został utworzony w przewidzianym czasie." >&2
      exit 1
    fi
  else     # Operacja zdjęcia blokady
    if [ ! -f "$1" ] ; then
      echo "$0: uwaga: blokada pliku$1 nie istnieje." >&2
      exit 1
    fi
    rm -f "$1"
  fi

  exit 0
```

Jak to działa?

Jak przystało na dobrze napisany skrypt, połowa jego kodu analizuje argumenty i sprawdza, czy nie ma przeszkód do podjęcia działania. Wreszcie wykonywana jest instrukcja `if` i podejmowana próba wykonania polecenia `lockfile`. Jeżeli polecenie jest dostępne, jest ono wywoływane zadaną liczbę razy i jeżeli nie uda się założyć blokady, wtedy wyświetlany jest odpowiedni komunikat. Co się stanie, jeżeli będziesz chciał zdjąć nieistniejącą blokadę? Pojawi się inny komunikat o błędzie. W przeciwnym wypadku polecenie `lockfile` usunie blokadę i skrypt zakończy działanie.

Mówiąc ściślej, w pierwszej części kodu ❶ zastosowana jest pętla `while` z bardzo przydatnym poleceniem `getopts` analizującym wszystkie możliwe argumenty (`-l`, `-u`, `-r`). Jest to popularny sposób wykorzystania polecenia `getopts`, które nieustannie będzie pojawiać się w tej książce. Zwróć uwagę na polecenie `shift $(($OPTIND - 1))` ❷. Zmienna `OPTIND` zawiera wartość przypisaną przez polecenie `getopts`. Zmienna ta jest wykorzystywana w skrypcie do przekazywania argumentów w lewo (tj. wartość zmiennej `$2` jest przekazywana zmiennej `$1`) dotąd, aż zostaną usunięte argumenty zawierające myślnik na początku.

Ponieważ skrypt wykorzystuje narzędzie systemowe `lockfile`, dobrą praktyką jest sprawdzenie przed jego użyciem, czy jest ono dostępne w jednym z katalogów użytkownika ❸. Jeżeli go nie będzie, wtedy zostanie wyświetlony odpowiedni komunikat. Dalej znajduje się prosta instrukcja warunkowa sprawdzająca ❹, czy operacją do wykonania jest założenie, czy zdjęcie blokady, i odpowiednio wywołująca polecenie `lockfile`.

Uruchomienie skryptu

Ponieważ powyższy skrypt nie działa samodzielnie, do jego przetestowania musisz użyć dwóch okien terminala. W celu utworzenia blokady po prostu uruchom skrypt `filelock`, podając w argumencie nazwę pliku, który chcesz zablokować. Aby zdjąć blokadę, uruchom skrypt ponownie z argumentem `-u`.

Wyniki

Najpierw zgodnie z listingiem 1.23 utwórz plik blokady.

Listing 1.23. Utworzenie pliku blokady za pomocą skryptu filelock

```
$ filelock /tmp/blokada.lck
$ ls -l /tmp/blokada.lck
-r--r--r-- 1 taylor wheel 1 Mar 21 15:35 /tmp/blokada.lck
```

Gdy spróbujesz ponownie zablokować plik, skrypt `filelock` podejmie zadaną liczbę prób (10), po czym wyświetli komunikat o błędzie, jak w listingu 1.24.

Listing 1.24. Nieudana próba zablokowania pliku za pomocą skryptu lockfile

```
$ filelock /tmp/blokada.lck
filelock : błąd: plik blokady nie został utworzony w przewidzianym czasie.
```

Gdy pierwszy proces zakończy przetwarzanie pliku, może zdjąć blokadę zgodnie z listingiem 1.25.

Listing 1.25. Zdjęcie blokady za pomocą skryptu filelock

```
$ filelock -u /tmp/blokada.lck
```

Aby przekonać się, jak skrypt `filelock` działa w dwóch terminalach, w jednym z nich załóż blokadę i obserwuj, jak w drugim oknie skrypt usiłuje założyć własną.

Rozbudowa skryptu

Ponieważ powyższy skrypt opiera swoje działanie na istnieniu pliku blokady, warto byłoby rozbudować go o dodatkowy argument określający maksymalny czas, przez który blokada może być założona. Gdy upłynie okres oczekiwania na zdjęcie blokady, wtedy powinien być sprawdzany czas ostatniej modyfikacji pliku. Jeżeli będzie on późniejszy niż podana w argumencie wartość, wtedy plik blokady będzie mógł zostać bezpiecznie potraktowany jako zapomniana blokada i usunięty, a jednocześnie wyświetlony zostanie odpowiedni komunikat ostrzegawczy.

Prawdopodobnie nie będzie to miało dla Ciebie znaczenia, ale polecenie `lockfile` nie działa poprawnie w systemie plików **NFS** (ang. *network file system* — sieciowy system plików), ponieważ mechanizm blokowania plików w tym systemie jest dość skomplikowany. Rozwiązaniem tego problemu może być tworzenie plików blokady tylko na lokalnych dyskach lub rozbudowanie skryptu o możliwość wykrywania sieci i zarządzania blokadami w różnych systemach.

Skrypt 11. Sekwencje kolorów ANSI

Możesz o tym nie wiedzieć, ale większość terminali wykorzystuje do wyświetlania tekstu różne style. Na przykład niektóre słowa w skrypcie mogą być wyróżnione pogrubioną lub czerwoną czcionką na żółtym tle. Jednak stosowanie sekwencji kolorów **ANSI** (ang. *American National Standards Institute* — Amerykański Państwowy Instytut Standaryzacyjny) jest uciążliwe, ponieważ nie jest ona przyjazna dla użytkownika. Aby rozwiązać ten problem, za pomocą listingu 1.26 możesz utworzyć zestaw zmiennych zawierających kody ANSI. Zmienne te możesz następnie wykorzystać do zmieniania różnych kolorów oraz do włączania lub wyłączania opcji formatujących tekst.

Kod

Listing 1.26. Skrypt initializeANSI

```
#!/bin/bash
# Kolory ANSI. Użyj poniższych zmiennych do wyświetlania tekstu w różnych kolorach i formatach.
# Kolory, których nazwy kończą się na literę "f", dotyczą pierwszego planu, natomiast na literę "b"— tła.

initializeANSI()
{
  esc="\033"    # Jeżeli ten kod nie działa, wpisz bezpośrednio znak ESC.

  # Kolory pierwszego planu (na końcu każdej nazwy znajduje się litera "f").
  blackf="${esc}[30m";   redf="${esc}[31m";    greenf="${esc}[32m"
  yellowf="${esc}[33m"   bluef="${esc}[34m";   purplef="${esc}[35m"
  cyanf="${esc}[36m";    whitef="${esc}[37m"

  # Kolory tła
  blackb="${esc}[40m";   redb="${esc}[41m";    greenb="${esc}[42m"
  yellowb="${esc}[43m"   blueb="${esc}[44m";   purpleb="${esc}[45m"
  cyanb="${esc}[46m";    whiteb="${esc}[47m"

  # Przełączniki czcionki pogrubionej, pochyłej i negatywowej.
  boldon="${esc}[1m";    boldoff="${esc}[22m"
  italicson="${esc}[3m"; italicsoff="${esc}[23m"
  ulon="${esc}[4m";      uloff="${esc}[24m"
  invon="${esc}[7m";     invoff="${esc}[27m"

  reset="${esc}[0m"
}
```

Jak to działa?

Jeżeli znasz kod HTML, może zdziwić Cię sposób stosowania powyższych sekwencji. W kodzie HTML znaczniki zamykające umieszcza się w odwrotnej kolejności niż znaczniki otwierające, a ponadto każdy znacznik otwierający musi mieć odpowiadający mu znacznik zamykający. Aby więc wyświetlić pogrubioną czcionką zdanie, w którym część wyrazów jest wyróżniona czcionką pochyłą, należy użyć następującego kodu:

```
<b>To jest pogrubiona czcionka, <i>a to czcionka pochyła</i> wewnątrz
pogrubionej.</b>
```

Umieszczenie znacznika wyłączającego pogrubienie i pominięcie znacznika wyłączającego pochylenie czcionki wprowadza bałagan, w którym czasami gubią się twórcy stron WWW. Jednak w przypadku sekwencji kolorów ANSI niektóre modyfikatory zastępują inne. Ponadto jest dostępna sekwencja, która anuluje wszystkie pozostałe. Po użyciu określonych sekwencji kolorów trzeba zawsze stosować sekwencję resetującą, jak również wyłączać włączone wcześniej opcje. W przypadku zastosowania powyższego skryptu zdanie z poprzedniego przykładu można wyświetlić w następujący sposób:

```
${boldon}To jest pogrubiona czcionka, ${italicson}a to czcionka
pochyła${italicsoff} wewnątrz pogrubionej.${reset}
```

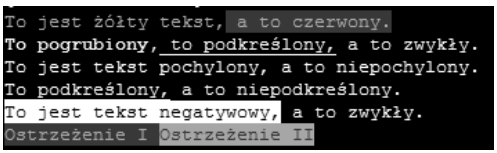
Uruchomienie skryptu

W celu przetestowania skryptu najpierw wywołaj funkcję inicjującą `initializeANSI()`, a następnie kilka poleceń `echo` z różnymi kombinacjami sekwencji kolorów i stylów:

```
initializeANSI
cat << EOF
${yellowf}To jest żółty tekst,${redb} a to czerwony.${reset}
${boldon}To pogrubiony,${ulon} to podkreślony,${reset} a to zwykły.
${italicson}To jest tekst pochylony,${italicsoff} a to niepochylony.
${ulon}To podkreślony,${uloff} a to niepodkreślony.
${invon}To jest tekst negatywowy,${invoff} a to zwykły.
${yellowf}${redb}Ostrzeżenie I ${yellowb}${redf}Ostrzeżenie II${reset}
EOF
```

Wyniki

Wyniki przedstawione na rysunku 1.1 nie wyglądają w druku efektownie, ale w terminalu, który obsługuje sekwencje kolorów, na pewno będą cieszyć Twoje oko.



Rysunek 1.1. Tekst wyświetlony po uruchomieniu skryptu z listingu 1.26

Rozbudowa skryptu

Gdy uruchomisz powyższy skrypt, możesz uzyskać następujący wynik:

```
\033[33m\033[41mOstrzeżenie I \033[43m\033[31mOstrzeżenie II\033[0m
```

Przyczyną problemu może być terminal, który nie obsługuje sekwencji kolorów ANSI lub nie obsługuje kodu `\33` najważniejszego znaku `esc`. W tym drugim przypadku otwórz skrypt w edytorze vi lub innym, usuń sekwencję `\33` i zamiast niej naciśnij klawisze *Ctrl+V*, a następnie *Esc*. Jeżeli uzyskasz zapis `esc="^[`", to problem będzie rozwiązany.

Jeżeli natomiast Twój terminal nie obsługuje sekwencji ANSI, zmień go na inny, w którym będziesz mógł stosować w swoich skryptach różne style wyników. Zanim jednak porzucisz swój terminal, sprawdź jego opcje. Niektóre z nich oferują ustawienia umożliwiające pełną obsługę sekwencji ANSI.

Skrypt 12. Tworzenie biblioteki skryptów powłoki

Wiele skryptów opisanych w tym rozdziale zostało przygotowanych w postaci funkcji, a nie niezależnych skryptów. Dzięki temu będzie można je łatwo wykorzystać w innych skryptach bez wprowadzania dodatkowego obciążenia systemu. Choć w skryptach powłoki nie można stosować instrukcji dołączającej, jak np. `#include` w języku C, dostępna jest jednak niezwykle ważna funkcjonalność umożliwiająca umieszczanie jednych skryptów wewnątrz innych, tak jakby były one funkcjami biblioteki.

Aby przekonać się, jak to jest ważne, rozważmy następujący przypadek. Gdy wywołujesz jeden skrypt wewnątrz drugiego, domyślnie jest on uruchamiany w osobnej podpowłoce. Możesz to sprawdzić w następujący sposób:

```
$ echo "test=2" >> małyskrypt.sh
$ chmod +x małyskrypt.sh
$ test=1
$ ./małyskrypt.sh
$ echo $test
1
```

Skrypt `małyskrypt.sh` zmienia wartość zmiennej `test`, ale tylko w obrębie podpowłoki, w której został uruchomiony, dlatego wartość zmiennej w powłoce nadrzędnej pozostanie niezmieniona. Jeżeli natomiast uruchomisz skrypt, umieszczając przed nim kropkę (`.`), wtedy każde zawarte w nim polecenie zostanie wykonane tak, jakby zostało wpisane w bieżącej powłoce:

```
$ . małyskrypt.sh
$ echo $test
2
```

Jak się zapewne domyślasz, jeżeli uruchomisz w powyższy sposób skrypt zawierający polecenie `exit 0`, spowoduje on wyjście z powłoki i zamknięcie okna terminala, ponieważ skrypt ten będzie działał jak proces podstawowy. Gdyby skrypt był uruchomiony w osobnej podpowłoce, wtedy zakończyłby swoje działanie bez przerywania działanie głównego skryptu. Na tym polega główna zaleta uruchamiania skryptu za pomocą kropki, polecenia `source` lub `exec` (które opiszemy później). W powłoce bash kropka i polecenie `source` pełnią taką samą funkcję. My będziemy używać kropki, ponieważ jest ona obsługiwana w różnych powłokach POSIX.

Kod

Aby umieścić w bibliotece opisane w tym rozdziale funkcje, jak również zmienne globalne (czyli wykorzystywane przez różne funkcje tablice) i móc je wykorzystywać w innych skryptach, należy je wyodrębnić i umieścić w jednym dużym pliku.

Jeżeli plikowi temu nadasz nazwę *library.sh* (biblioteka), będziesz mógł wykorzystać skrypt przedstawiony w listingu 1.27 do przetestowania wszystkich funkcji opisanych w tym rozdziale i sprawdzić, czy działają poprawnie.

Listing 1.27. Umieszczenie w bibliotece wszystkich dotychczas utworzonych funkcji i wywołanie ich

```
#!/bin/bash
# Skrypt testujący bibliotekę.
# Najpierw wykonywany jest plik 'library.sh'.

❶ . library.sh

initializeANSI  # Inicjalizacja wszystkich sekwencji ANSI.

# Sprawdzenie funkcji validint.
echon "Przede wszystkim, czy w zmiennej PATH jest polecenie echo? (1=tak, 2=nie) "
read answer
while ! validint $answer 1 2 ; do
  echon "${boldon}Spróbuj jeszcze raz${boldoff}."
  echon "Czy polecenie echo znajduje się w zmiennej PATH? (1=tak, 2=nie) "
  read answer
done

# Czy działa funkcja sprawdzająca, czy dane polecenie znajduje się w zmiennej PATH?
if ! checkForCmdInPath "echo" ; then
  echo "Nie mogę znaleźć polecenia echo."
else
  echo "Polecenie echo znajduje się w ścieżce PATH."
fi

echo ""
echon "Wpisz rok, który według ciebie jest rokiem przestępnym: "
read year

# Sprawdzenie, czy rok zawiera się w przedziale od 1 do 9999
# za pomocą funkcji validint z argumentami określającymi wartość minimalną i maksymalną.
while ! validint $year 1 9999 ; do
  echon "Wpisz rok w ${boldon}poprawnym${boldoff} formacie: "
  read year
done

# Teraz sprawdzamy, czy rok faktycznie jest przestępny.
if isLeapYear $year ; then
  echo "${greenf}Dobrze! Rok $year jest przestępny.${reset}"
else
  echo "${redf}Nie, to nie jest rok przestępny.${reset}"
fi

exit 0
```

Jak to działa?

Zwróć uwagę, że cała biblioteka ze wszystkimi funkcjami jest ładowana do środowiska uruchomieniowego skryptu za pomocą jednego wiersza ❶.

Ten przydatny sposób korzystania z różnych skryptów opisanych w tej książce można stosować wielokrotnie. Należy jedynie sprawdzić, czy dołączana biblioteka znajduje się w jednym z katalogów w zmiennej `PATH`, dzięki czemu polecenie `.` (kropka) będzie mogło ją odnaleźć.

Uruchomienie skryptu

Uruchom skrypt testowy w wierszu poleceń, jak w listingu 1.28.

Wyniki

Listing 1.28. Uruchomienie skryptu library-test

```
$ library-test
Przede wszystkim, czy w zmiennej PATH jest polecenie echo? (1=tak, 2=nie) 1
Polecenie echo znajduje się w ścieżce PATH.

Wpisz rok, który według ciebie jest rokiem przestępnym: 432432
Liczba 432432 jest za duża. Maksymalna wartość to 9999.
Wpisz rok w poprawnym formacie: 432
Dobrze! Rok 432 jest przestępny.
```

Jeżeli liczba oznaczająca rok będzie za duża, wówczas pojawi się komunikat, w którym słowo „poprawnym" będzie wyróżnione pogrubioną czcionką. Ponadto jeżeli podany rok będzie rokiem przestępnym, stosowny komunikat zostanie wyświetlony zieloną czcionką.

W rzeczywistości rok 432 nie był rokiem przestępnym, ponieważ lata przestępne pojawiły się w kalendarzach dopiero po roku 1752. Ale my się tu zajmujemy skryptami, a nie kalendarzami, więc możemy przymknąć na to oko.

Skrypt 13. Diagnostyka skryptów powłoki

Choć niniejsza sekcja nie zawiera skryptu, chcemy jednak kilka stron poświęcić podstawowym metodom diagnozowania skryptów powłoki, ponieważ na pewno do Twoich skryptów będą zakradały się różne pomyłki.

Z naszego doświadczenia wynika, że najlepszą metodą unikania problemów jest tworzenie skryptów stopniowo. Niektórzy programiści są nastawieni bardzo optymistycznie do swojej pracy i zakładają, że wszystko będzie od razu działać poprawnie. Jednak posuwanie się małymi krokami do przodu może naprawdę ułatwić pracę. Oprócz tego powinieneś często stosować polecenie `echo` do śledzenia wartości zmiennych oraz jawnie wywoływać skrypty poleceniem `bash -x`, aby na ekranie pojawiały się informacje diagnostyczne, np.:

```
$ bash -x mójskrypt.sh
```

Ewentualnie możesz od razu włączyć tryb diagnostyczny, wpisując polecenie `set -x`, a na koniec wyłączyć go poleceniem `set +x`, jak niżej:

```
$ set -x
$ ./mójskrypt.sh
$ set +x
```

Aby zobaczyć efekty zastosowania poleceń `set -x` i `set +x`, zdiagnozuj prostą grę-zgadywankę, pokazaną w listingu 1.29.

Kod

Listing 1.29. Skrypt hilow, w którym są błędy wymagające zdiagnozowania

```
#!/bin/bash
# Skrypt hilow – prosta zgadywanka liczbowa.

biggest=100                     # Maksymalna dopuszczalna liczba.
guess=0                         # Liczba podana przez gracza.
guesses=0                       # Liczba prób odgadnięcia.
❶ number=$(( $$ % $biggest ))   # Losowa liczba pomiędzy 1 a $biggest.
echo "Odgadnij liczbę z przedziału od 1 do $biggest"

while [ "$guess" -ne $number ] ; do
❷   /bin/echo -n "Jaka to liczba? " ; read answer
    if [ "$guess" -lt $number ] ; then
❸     echo "... większa!"
    elif [ "$guess" -gt $number ] ; then
❹     echo "... mniejsza!
    fi
    guesses=$(( $guesses + 1 ))
done

echo "Dobrze! Odgadłeś liczbę $number za $guesses. razem."

exit 0
```

Jak to działa?

Aby zrozumieć, jak działa polecenie generujące losową liczbę ❶, musisz wiedzieć, że sekwencja `$$` oznacza identyfikator procesu (PID) powłoki, w której działa skrypt. Zazwyczaj jest to 5- lub 6-cyfrowa liczba. Przy każdorazowym uruchomieniu skryptu identyfikator jest inny. Operacja `% $biggest` powoduje podzielenie identyfikatora przez maksymalną dopuszczalną liczbę i obliczenie reszty. Na przykład działanie 5 % 4 daje resztę 1, podobnie jak działanie 41 % 4. Jest to prosty sposób generowania pseudolosowych liczb w zakresie od 1 do `$biggest`.

Uruchomienie skryptu

Pierwszą czynnością diagnostyczną jest sprawdzenie, czy generowane liczby rzeczywiście są losowe. Liczba jest generowana poprzez odczytanie za pomocą wyrażenia $$ identyfikatora PID procesu powłoki, w której działa skrypt, a następnie zmniejszenie go do dopuszczalnego zakresu za pomocą operatora % ❶. Aby sprawdzić działanie tego wyrażenia, możesz wpisać je bezpośrednio w wierszu poleceń, jak niżej:

```
$ echo $(( $$ % 100 ))
5
$ echo $(( $$ % 100 ))
5
$ echo $(( $$ % 100 ))
5
```

Wyrażenie jest poprawne, ale nie generuje losowych liczb. Po chwili zastanowienia wiadomo dlaczego: jeżeli polecenie jest wykonywane bezpośrednio w wierszu poleceń, wtedy identyfikator PID jest zawsze taki sam. Natomiast gdy polecenie jest wykonywane w skrypcie, wtedy za każdym razem jest otwierana nowa podpowłoka i identyfikator się zmienia.

Innym sposobem generowania liczb losowych jest odwoływanie się do zmiennej środowiskowej `$RANDOM`. To jest magiczna zmienna! Przy każdym odwołaniu zawiera inną wartość. Aby wygenerować liczbę z przedziału od 1 do `$biggest`, można w wierszu ❶ zastosować następujące wyrażenie: `$(( $RANDOM % $biggest + 1 ))`.

Następna część kodu to prosty algorytm gry. Najpierw generowana jest liczba z przedziału od 1 do 100 ❶, a następnie użytkownik próbuje ją odgadnąć ❷. Po każdej próbie pojawia się komunikat, czy podana liczba jest za duża ❸, czy za mała ❹. Po wpisaniu kodu czas go uruchomić i sprawdzić, jak działa. Poniżej przedstawione jest działanie kodu z listingu 1.29 ze wszystkimi jego mankamentami:

```
$ hilow
./hilow: line 20: unexpected EOF while looking for matching '"'
./hilow: line 23: syntax error: unexpected end of file
```

Ujawniła się zmora każdego programisty powłoki: błąd nieoczekiwanego końca pliku (EOF). Choć komunikat zawiera informację, że błąd jest w wierszu 20., nie oznacza to, że faktycznie tam jest. W rzeczywistości wiersz 20. jest poprawny:

```
$ sed -n 20p hilow
echo "Dobrze! Odgadłeś liczbę $number za $guesses. razem."
```

Aby zrozumieć, co tu się dzieje, przypomnij sobie, że ciągi znaków wewnątrz cudzysłowów mogą zawierać znaki końca wiersza. Oznacza to, że jeżeli powłoka napotka ciąg bez cudzysłowu zamykającego, wtedy odczytuje pozostałą część kodu, szukając tego cudzysłowu. W tym przypadku zatrzyma się dopiero przy ostatnim cudzysłowie i stwierdzi, że czegoś tu brakuje.

Zatem problem pojawia się wcześniej. W powyższym komunikacie przydatna jest jedynie informacja o brakującym znaku. Dzięki temu można wyodrębnić za pomocą instrukcji `grep` wszystkie wiersze skryptu zawierające cudzysłów, a następnie odrzucić z nich te, które zawierają dokładnie dwa cudzysłowy. Służy do tego celu poniższe polecenie:

```
$ grep '"' hilow | egrep -v '.*".*".*'
echo "... mniejsza!
```

Jest! Brakuje cudzysłowu zamykającego w wierszu wyświetlającym komunikat, że użytkownik musi podać mniejszą liczbę ❹. Wpisz brakujący cudzysłów i uruchom skrypt jeszcze raz:

```
$ hilow
./hilow: line 7: unexpected EOF while looking for matching ')'
./hilow: line 23: syntax error: unexpected end of file
```

Nic z tego. Pojawił się inny problem z brakiem nawiasu zamykającego. Ponieważ w skrypcie bardzo mało jest wyrażeń z nawiasami, wystarczy wytężyć wzrok i stwierdzić brak nawiasu w wyrażeniu pomniejszającym wygenerowaną losowo liczbę:

```
number=$(( $$ % $biggest )      # Losowa liczba pomiędzy 1 a $biggest.
```

Popraw błąd, dopisując brakujący nawias na końcu wiersza, ale przed znakiem komentarza. Czy teraz gra będzie działać? Sprawdź to:

```
$ hilow
Odgadnij liczbę z przedziału od 1 do 100
Jaka to liczba? 33
... większa!
Jaka to liczba? 66
... większa!
Jaka to liczba? 99
... większa!
Jaka to liczba? 100
... większa!
Jaka to liczba? ^C
```

Prawie dobrze. Ponieważ 100 jest największą możliwą wartością, wygląda na to, że algorytm zawiera błędy. Tego rodzaju błędy trudno znaleźć, ponieważ nie ma przemyślnego polecenia `grep` lub `sed`, które pozwoliłoby rozwiązać problem. Przyjrzyjmy się kodowi jeszcze raz i sprawdźmy, co w nim jest nie tak.

Aby zdiagnozować problem, wpisz dodatkowe polecenie `echo` wyświetlające liczbę wpisaną przez użytkownika, a następnie sprawdź, czy jest ona porównywana z wygenerowaną liczbą. Porównywanie jest wykonywane w wierszu ❷. Poniżej przedstawiony jest ten fragment jeszcze raz:

```
/bin/echo -n "Jaka to liczba? " ; read answer
if [ "$guess" -lt $number ] ; then
```

Faktycznie, po zmianie polecenia `echo` i bliższym przyjrzeniu się kodowi widać błąd: liczba podawana przez użytkownika jest zapisywana w zmiennej `answer`, a porównywana jest zmienna `guess`. Banalny błąd, ale nierzadko popełniany (szczególnie w przypadku zmiennych o skomplikowanych nazwach). Aby go poprawić, zmień polecenie `read answer` na `read guess`.

Wyniki

Wreszcie kod działa zgodnie z oczekiwaniami, jak pokazuje listing 1.30.

Listing 1.30. Perfekcyjnie działający skrypt zgadywanki

```
$ hilow
Odgadnij liczbę z przedziału od 1 do 100
Jaka to liczba? 50
... większa!
Jaka to liczba? 75
... większa!
Jaka to liczba? 88
... mniejsza!
Jaka to liczba? 83
... mniejsza!
Jaka to liczba? 80
... mniejsza!
Jaka to liczba? 77
... większa!
Jaka to liczba? 79
Dobrze! Odgadłeś liczbę 79 za 7. razem.
```

Rozbudowa skryptu

Najpoważniejszym mankamentem powyższego skryptu jest brak weryfikacji poprawności wprowadzanych liczb. Jeżeli wpiszesz cokolwiek innego niż liczbę całkowitą, skrypt wyświetli komunikat o błędzie i przerwie działanie. Zaimplementowanie prostego testu sprawdzającego, czy w ogóle została podana jakakolwiek wartość, jest łatwe i polega jedynie na wpisaniu wewnątrz pętli `while` następującego kodu:

```
if [ -z "$guess" ] ; then
  echo "Podaj poprawną liczbę. Naciśnij ^C, aby zakończyć."; continue;
fi
```

Problem jednak polega na tym, że niezerowy ciąg znaków nie musi być liczbą i po wpisaniu na przykład „cześć” znów pojawi się komunikat o błędzie. Aby go poprawić, wykorzystaj funkcję `validint` ze skryptu nr 5.

Skorowidz

N

O

P

Q

R

S

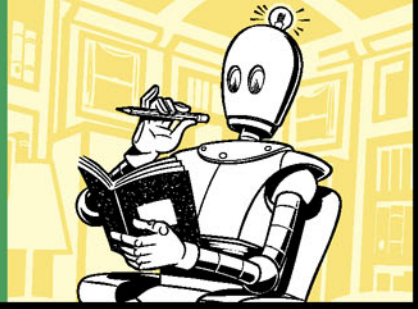
Ś

T

U

W

Z



SKRYPTY POWŁOKI: ODKRYJ JE NA NOWO!

Systemy uniksowe rozkwitają. W ciągu ostatnich kilkunastu lat pojawiły się przeróżne, stosunkowo proste w obsłudze dystrybucje Linuksa, firma Apple stworzyła OS X, łatwiejsze stało się również administrowanie systemem Unix. Olbrzymią karierę robi system Android — pod jego kontrolą pracują miliardy przeróżnych urządzeń. Istnieje wspólny element każdego z nich: powłoka Bourne, czyli *bash*. Okazuje się, że w czasach finezyjnych GUI umiejętność wykorzystania całej mocy skryptów powłoki jest bezcenna.

Niniejsza książka posłuży administratorom i użytkownikom systemów uniksowych, przyda się też programistom i analitykom danych. Znalazło się tu ponad sto świetnych skryptów, z których każdy można modyfikować według potrzeb. Omówiono również główne koncepcje skryptów, jednak nie zamieszczono analizy ich działania wiersz po wierszu. Zamiast tego pokazano mnóstwo niezwykle przydatnych sposobów automatyzacji różnych czynności związanych z zarządzaniem systemem. W każdym rozdziale opisano po kilka nowych funkcji skryptów i przykładów ich zastosowania. Dodatkowo omówiony został proces instalacji powłoki *bash* w systemie Windows 10!

W tej książce między innymi:

- zwięzłe wprowadzenie do składni i zastosowań skryptów powłoki
- narzędzia dla webmasterów i administratorów serwerów WWW
- skrypty dla macOS i ich możliwości
- gry, praca w chmurze i przetwarzanie plików graficznych

Dave Taylor — zajmuje się informatyką od 1980 roku. Jest jednym z twórców systemu BSD 4.4 Unix — wszystkie najważniejsze dystrybucje Linuksa zawierają programy jego autorstwa. Jest autorem licznych książek i artykułów w branżowych wydawnictwach informatycznych. Często bierze udział w seminariach.

Brandon Perry — jest programistą. Jego przygoda z tworzeniem oprogramowania zaczęła się na początku XXI wieku, gdy opublikowano otwartą implementację projektu .NET Mono. W wolnym czasie tworzy moduły dla platformy Metasploit, analizuje pliki binarne i rozkłada różne aplikacje na czynniki pierwsze.

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
● http://helion.pl/promocje
Książki najchętniej czytane:
● http://helion.pl/bestsellery
Zamów informacje o nowościach:
● http://helion.pl/nowosci

ISBN 978-83-283-3430-4
9 788328 334304
cena: 67,00 zł

no starch press