Interpretery komend Unixa

Witold Paluszyński
witold.paluszynski@pwr.wroc.pl
http://kcir.pwr.edu.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat interpreterów komend Unixa i programowania skryptów. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Polecenia uniksowego interpretera poleceń

Poleceniem może być wywołanie zewnętrznego programu, lub wywołanie polecenia wbudowanego albo funkcji interpretera poleceń. Większość czynności realizują programy zewnętrzne. Polecenia mogą mieć argumenty przekazywane w wierszu wywołania (zwanym wektorem argumentów):

```
who
ps -ef
ls -l
set -f
```

Wykonanie programu powoduje utworzenie oddzielnego procesu, który jest podprocesem procesu interpertera poleceń. Wykonanie polecenia wbudowanego lub funkcji odbywa się w ramach procesu interpretera poleceń.

Proces w trakcie wykonywania się ma dostęp do standardowych urządzeń wejścia i wyjścia: *stdin, stdout, stderr*, jak w języku C, i innych. Istnieje również tzw. środowisko, które jest zestawem zmiennych z przypisanymi wartościami (w postaci napisów tekstowych), które są tworzone przed uruchomieniem programu, i potem są dostępne w trakcie jego wykonywania.

Uruchamianie podprocesów

Przy interakcyjnej pracy z interpreterem poleceń możliwa jest manipulacja podprocesami uruchomionymi przez dany interpreter. Można zobaczyć listę podprocesów, i każdy z nich zatrzymać, a także wznowić, zarówno jako zadanie w tle jak i w pierwszym planie. Podprocesy identyfikowane są kolejnymi liczbami, a w odwołaniu do nich piszemy numer podprocesu poprzedzony znakiem procenta.

```
acroread datasheet.pdf &
firefox http://www.google.pl/ &

jobs

fg %n

bg %n

stop %<n>                   # tylko C-shell
```

Globbing — dopasowanie nazw plików

Interpreter poleceń realizuje tzw. *globbing*, polegający na dopasowaniu następujących znaków: `* ? [a-zA-Z_]` do nazw plików:

```
wc *.c
echo obraz?.pgm    # wynik: obraz1.pgm obraz2.pgm
ls -l *.[cho]
```

Ważne jest zrozumienie, że *globbing* jest mechanizmem shella, a nie ogólną konwencją. Na przykład, mechanizm ten może z jakichś powodów nie zadziałać, i wtedy wzorzec nazwy pliku jest tylko zwykłym stringiem, nie mającym nic wspólnego z odpowiadającymi mu nazwami istniejących plików:

```
$ ls try_r*.c
try_readdir.c  try_realloc.c  try_regex.c

$ ls 'try_r*.c'
ls: cannot access try_r*.c: No such file or directory
```

Potoki

Potok (*pipeline*) to równoległe uruchomienie dwóch lub więcej poleceń z szeregowym połączeniem ich wyjść i wejść:

```
who
who | wc -l
who | tee save | wc -l
```

Zapis potoku sprawia skromne wrażenie, lecz w potokach tkwi wielka moc. Bierze się ona po pierwsze z bardzo porządnej implementacji, dzięki której przez potok może przepłynąć nieograniczona ilość danych,[1] i może on pracować przez dowolnie długi czas. W tym czasie poszczególne procesy pracują równolegle, a system synchronizuje ich pracę, usypiając te, które czytają lub zapisują dane zbyt szybko. Następnie budzi je w sposób przezroczysty, gdy pozostałe procesy nadążyły z przetwarzaniem tych danych.

Drugie źródło tej mocy to zestaw narzędzi do przetwarzania tekstów Unixa, działających w trybie input/output, dzięki którym wiele zadań w systemie Unix można wykonać za pomocą tych narzędzi odpowiednio połączonych potokami.

[1] Niezależnie od ich rzeczywistej, zaimplementowanej w systemie pojemności potoku. Standard POSIX wymaga minimum 512 bajtów. Wiele systemów stosuje 5120.

Listy

Lista to połączenie poleceń (ściślej: potoków) spójnikami: `;, &, ||, &&`

```
cc prog.c ; echo kompilacja zakonczona
cc prog.c & echo kompilacja uruchomiona
grep pieniadze msg.txt && echo sa pieniadze
grep pieniadze msg.txt || echo nie ma pieniedzy
```

Priorytety spójników: `|` — najwyższy, `||,&&` — średni, `;,&` — najniższy. Zestaw poleceń można wziąć w nawiasy, aby te priorytety zmienić:

```
 date; who    | wc
(date; who)   | wc
{ date; who;} | wc
```

Użycie nawiasów okrągłych ma dodatkowy efekt w postaci utworzenia dodatkowej kopii interpretera poleceń, będącym podprocesem procesu głównego, który wykonuje listę w nawiasie. Nawiasy klamrowe powodują wykonanie listy w procesie bieżącego interpretera poleceń, jednak wymagają użycia separatorów składniowych.

Status procesu

Każdy proces generuje kod zakończenia (*exit code*) zwany też statusem zakończenia (*exit status*) lub po prostu statusem. W przypadku programu statusem jest normalnie wartość zwracana instrukcją "`return expr;`" w funkcji `main()`, lub funkcją biblioteczną "`exit(expr);`". W przypadku skryptu status można zwrócić wbudowanym poleceniem `exit` lub `return`.

Konwencjonalnie, zerowa wartość statusu oznacza poprawne zakończenie procesu, a każda inna wartość oznacza błąd (i często jest kodem błędu). Programy, których wynik ma sens logiczny prawdy lub fałszu (albo sukcesu lub porażki), generują zwykle zerowy status w przypadku sukcesu, a niezerowy wpw.

W systemie Unix status procesu jest przechwytywany przez system, i powinien być odczytany przez proces nadrzędny zakończonego procesu. W przypadku skryptów procesem nadrzędnym jest interpreter poleceń, i on odbiera status wszystkich uruchomionych przez siebie procesów. Dla poleceń wykonywanych synchronicznie, bezpośrednio po wykonaniu polecenia interpreter zapamiętuje status w zmiennej `$?` (w C-shellu `$status`).

W przypadku potoków oraz list, statusem potoku i listy jest status ostatniego wykonanego polecenia.

Skierowania wejścia/wyjścia

Skierowanie wejścia/wyjścia >, >>, <, <<=...=

```
./prog < plik_wejscia > plik_wyjscia
```

Skierowanie <<=...= zwane *here document* będzie omówione później.

Dodatkowo C-shell:

```
./prog 15 >& do_pliku
```

Dodatkowo Bourne shell:

```
./prog 15 > plik_wyjscia 2> plik_bledow
./prog 15 > plik_razem   2>&1
./prog 15 2>&1  > plik_wyjscia
```

Uniksowe konwencje dotyczące skierowań

Ważna metoda pracy w systemach uniksowych polega na pisaniu programów realizujących indywidualne, dobrze zdefiniowane funkcje, i łączeniu ich według potrzeby za pomocą potoków. Programy te często mają minimalny interfejs użytkownika, i ich działanie sterowane jest argumentami wywołania (oraz zmiennymi środowiskowymi). Interakcje z użytkownikiem można dodać w potoku:

```
ls *.txt | wybor | przetwarzanie
Element ‘a.txt’ - wybrac? T
Element ‘b.txt’ - wybrac? N
Element ‘c.txt’ - wybrac? T
```

Pisząc programy w tej konwencji należy uważać na rozdzielenie strumienia danych (stdin, stdout) od wyświetlanych pomocniczych komunikatów, np. o błędach (stderr), oraz interakcji z użytkownikiem (np. /dev/tty).

```
% tar cvf - files | \
 ssh host dd of=/dev/tape
komunikaty tara
user@host's password:
```

```
% ssh host dd if=/dev/tape |\
 tar xvf - files
user@host's password:
komunikaty tara
```

Skrypty interpretera komend

Skryptem nazywamy plik zawierający zestaw poleceń interpretera. Skrypt może zawierać jedno lub więcej poleceń, w jednym lub więcej wierszy. Na przykład, chcemy policzyć liczbę użytkowników włączonych do systemu w następujący (wcale nienajprostszy) sposób:

```
who | wc -l
```

```
cat > policz
who | wc -l
^D
```

Skrypt można uruchomić, jawnie wywołując interpreter poleceń (obecny na większości systemów jako /bin/sh), i podając mu nazwę skryptu jako argument:

```
sh policz
.  policz              # C-shell: source policz
```

Forma z kropką nie wywołuje dodatkowego procesu interpretera komend; skrypt wykonywany jest przez ten sam interpreter, w którym polecenie zostało wpisane.

Pliki wykonywalne

Możemy nadać plikowi policz atrybut wykonywalności x. Wtedy można spowodować wykonanie skryptu przez użycie nazwy pliku jako polecenia:

```
chmod +x policz
./policz
```

W tym przypadku interpreter komend (bieżący) sam wywoła drugi interpreter (który będzie jego podprocesem) dla wykonania skryptu.

Można również skorzystać ze zmiennej PATH zawierającej ścieżkę katalogów automatycznie przeszukiwanych w celu wykonania programów i skryptów. Zakładając, że katalog $LOGNAME/bin, znajdujący się w katalogu domowym danego użytkownika, jest na ścieżce zmiennej PATH, możemy wywołać skrypt z dowolnego katalogu:

```
mv policz $LOGNAME/bin
hash                    # C-shell: rehash
policz
```

Argumenty wywołania skryptu

Skrypt może być wywołany z argumentami (podobnie jak każde polecenie). Argumenty wpisane w wierszu wywołania tworzą tzw. wektor argumentów wywołania. Nazwa skryptu lub programu jest również elementem tego wektora, traktowanym jako element zerowy:

skrypt_0 arg_1 arg_2 arg_3 ...

Argumenty wywołania są dostępne wewnątrz skryptu (lub programu) w układzie pozycyjnym:

```
echo pierwszy argument:                    $1
echo drugi argument:                       $2
echo itd. ...
echo argument zerowy, nazwa skryptu:       $0
echo wektor argumentow, bez arg. zerowego: $*
```

Argumenty są traktowane jako napisy tekstowe, podobnie zresztą jak wszystkie dane przetwarzane przez interpreter komend.

Inne konstrukcje „dolarowe”

Interpreter komend posiada szereg dalszych konstrukcji pozwalających obliczać różne wartości w sposób podobny do brania wartości argumentów wywołania, np.:

```
echo numer procesu interpretera polecen:      $$
```

Dalsze mechanizmy są różne w różnych interpreterach poleceń, np. w C-shellu mamy:

```
echo liczba argumentow, bez arg. zerowego:    $#argv
echo przedzial wartosci argumentow:           $argv[2-]
```

W Bourne shellu dostępne są inne konstrukcje dolarowe:

```
echo liczba argumentow, bez arg. zerowego:    $#
echo uzycie wart.domyslnej gdy brak argumentu: ${1:-domysl}
```

Operacje na wektorze argumentów

Wektor argumentów wywołania można przesuwać w lewo operacją `shift`. Powoduje ona zastąpienie argumentu pierwszego drugim, drugiego trzecim, itp., efektywnie skracając wektor argumentów wywołania. Argument zerowy (nazwa skryptu) nie podlega przesuwaniu operacją `shift` i pozostaje niezmieniony:

skrypt_0 arg_1 arg_2 arg_3

po `shift`:

skrypt_0 arg_2 arg_3

W Bourne shellu istnieje też polecenie `set` pozwalające ustawić (nadpisać) cały wektor argumentów (od $1) bieżącego wywołania:

```
set jeden dwa trzy
echo $*             # wynik: jeden dwa trzy

date                # wynik: czw 11 mar 2004 06:45:23
set `date`
echo czas = $5      # wynik: czas = 06:45:23
```

Zmienne

Istnieją dwa rodzaje zmiennych: zmienne lokalne interpretera komend, i zmienne globalne, tzw. środowiskowe, dziedziczone przez podprocesy od ich procesów macierzystych. Zatem zmienne globalne są dostępne dla programów uruchamianych przez interpreter komend, w tym również rekurencyjnie uruchamianych interpreterów komend.

C-shell

```
set a = 5
echo a ma wartosc $a
setenv ZMGL 20
echo ZMGL ma wartosc $ZMGL
# program ma dostep do ZMGL
./program ...
unsetenv ZMGL
```

Bourne shell

```
a=5
echo a ma wartosc $a
export ZMGL
ZMGL=20
echo ZMGL ma wartosc $ZMGL
# program ma dostep do ZMGL
./program ...
# tylko chwilowy eksport
ZMGL=20 ./program ...
```

Zmienne systemowe

Niektóre programy systemowe używają zmiennych środowiskowych, które określają lub zmieniają sposób ich działania:

PATH	używany przez sam interpreter poleceń
TERM	używany przez programy ekranowe, które chcą skorzystać z operacji terminala użytkownika innych niż zwykłe przesuwanie
MAIL	ścieżka do pliku ze skrzynką pocztową użytkownika
PAGER	określenie programu, który należy wywołać w celu wyświetlania tekstu ekran po ekranie, np. `man`
EDITOR	określenie programu, który należy wywołać w celu edycji pliku zainicjowanej przez niektóre programu, np. `crontab`

Znaki specjalne: cytowanie

\	odbiera znaczenie specjalne następującego po nim znaku
'...'	zaniechanie interpretacji wszelkich znaków specjalnych
"..."	zaniechanie interpretacji znaków specjalnych z wyjątkiem $, \ (tylko w Bourne shellu), i `...`

Ponadto, znak NEWLINE (\n, ASCII 10), traci znaczenie specjalne po \, ale jest dopuszczalny wewnątrz napisów cytowanych tylko w Bourne shellu.

W teorii to wydaje się proste, ale wyrażenia cytowane można składać, i czasem trzeba dobrze się zastanowić, aby zanalizować, albo skonstruować jakieś wyrażenie, i wtedy powyższe reguły trzeba stosować bardzo uważnie.

```
echo "'proste'" # to jest proste
echo '\'        # to tez
echo "\""       # poprawne w Bourne shellu ale nie w C-shellu
echo '\''       # poprawne choc niekompletne w Bourne shellu,
                # to samo znaczenie, ale niepoprawne w C-shellu
```

Polecenia zagnieżdżone

Polecenie zagnieżdżone może być umieszczone w dowolnym miejscu każdego polecenia i objęte znakami apostrofów wstecznych `...` (*backquote*). Jest ono wykonywane przez dodatkową kopię interpretera poleceń przed rozpoczęciem wykonywania polecenia oryginalnego. Polecenie zagnieżdżone może wykonywać dowolne operacje, i system zbiera wynik jego pracy w postaci stringa wysyłanych na wyjście znaków, a następnie podmienia treść polecenia (razem z apostrofami wstecznymi) otrzymanym ciągiem znaków.

Przykłady:

```
scp -p *.c remotesystem:`pwd`     # kopia do ident.katalogu

string_length=`echo $str|wc -c`   # echo generuje NEWLINE

case `uname -s` in
  "Linux") runprog=prog.linexe ;;
  "SunOS") runprog=prog.sunexe ;;
        *) echo Unknown system. ;;
esac
```

Bourne shell: algorytm działania

1. czyta jedną komendę z wejścia

2. dokonuje interpretacji komendy, np. podstawień wartości zmiennych

3. traktując pierwsze słowo komendy jako jej nazwę (polecenie), a pozostałe słowa jako argumenty, wykonuje komendę w jeden z następujących sposobów:

   (a) jako funkcję lub wbudowaną komendę interpretera
   (b) jako program zawarty w pliku dyskowym, jeśli polecenie ma postać składniową nazwy pliku (względnej lub bezwzględnej), np. `/bin/echo` lub `../prog/prog1`
   (c) jako program zawarty w pliku dyskowym, jeśli plik o podanej nazwie znajduje się w jednym z katalogów dyskowych zadanych zmienną `PATH`

→ program zawarty w pliku dyskowym może być programem binarnym, lub tzw. skryptem (zestawem komend)

Bourne shell: interpretacja komendy

algorytm interpretacji komendy jest dużo bardziej złożony niż algorytm jej wykonania:

1. podstawienia komend zagnieżdżonych (`‘...‘`)
2. podstawienia parametrów pozycyjnych (argumentów wywołania `$1, $2)` i~zmiennych `($PATH, $var)`
3. podział linii komendy na argumenty (które mogą być puste, np. `’’`)
4. zastosowanie skierowania strumieni wejścia i wyjścia
5. podstawienia nazw plików ze znakami specjalnymi `*, ?, [...]`

- apostrofy (`’...’`) powodują brak jakiejkolwiek interpretacji znaków w nich zawartych, natomiast cudzysłowy (`"..."`) powodują jedynie interpretację zawartych w nich znaków $, \, i `‘...‘`

Bourne shell: wyrażenia warunkowe

```
if test -r prog.dane
then
  prog < prog.dane
else
  prog
fi


if [ "‘tty‘" = "/dev/console" ] ; then echo Konsola ; fi


case $DISPLAY in
        ":0.0") host=‘uname -n‘:0.0;;
          ":0") host=‘uname -n‘:0.0;;
             *) host=$DISPLAY;;
esac
```

Bourne shell: znaki specjalne

\	\z powoduje wzięcie znaku ’z’ dosłownie
#	jeśli na początku słowa to początek komentarza
*	dopasowuje się do dowolnego ciągu znaków w nazwie pliku
?	dopasowuje się do pojedynczego znaku w nazwie pliku
[abc...]	dopasowanie pojedyn.znaku, można podać przedział np. [a-z]
;	sekwencyjne wykonanie poleceń
&	równoległe (asynchroniczne) wykonanie poleceń
|	potok, równoległe wykonanie poleceń z przekazyw.danych
&& ||	sekwencyjne ale warunkowe wykonanie poleceń
< >	skierowanie strumieni danych z i do pliku
(...)	wykonanie ... w wewnętrznym interpreterze poleceń
‘...‘	wykonanie poleceń w ..., otrzymane wyjście zastępuje ‘...‘
’...’	wzięcie ... dosłownie
"..."	wzięcie ... dosłownie, jednak po interpretacji $, ‘...‘, i \
zm=wart	przypisanie wartości zmiennej
$zm	wzięcie wartości zmiennej
${zm}	wzięcie wartości zmiennej, unika dwuznaczności w połączeniu z innym tekstem
$1, ...	parametry pozycyjne skryptu ($1 do $9), nazwa skryptu ($0)

Bourne shell: polecenia pętli

```
for x in *.c
do
  # cmp wyswietla na wyjsciu informacje o roznicach
  if [ "‘cmp $x~ $x‘" ]
  then
    echo Plik $x zmienil sie, wysylamy koledze...
    mailx -s "nowa wersja pliku $x" kolega < $x
  fi
done


while true
do
  /bin/echo -n "Podaj nazwe skryptu do wykonania: "
  sh ‘line‘
done
```

Testowanie warunków — `test`

Warunki logiczne w instrukcjach `if` i `while` testowane są przez sprawdzenie statusu (kodu zakończenia, ang. *status, exit code*) programów. Statusem programu jest liczba całkowita zwracana przez procedurę `main()` instrukcją `return expr;` lub funkcją biblioteczną `exit(expr);`. Status można zwrócić ze skryptu wbudowanym poleceniem `exit` lub `return`.

Wartość statusu 0 oznacza prawdę, a każda inna wartość fałsz.

Wartość statusu jest normalnie niewidzialna przy interakcyjnym wykonywaniu poleceń, ale jest przechwytywana przez interpreter poleceń, i dla poleceń wykonywanych synchronicznie, bezpośrednio po wykonaniu polecenia jest dostępna w zmiennej `$?` (w C-shellu `$status`).

Niektóre programy są specjalnie przygotowane do sprawdzania warunków, np. mają opcję powodującą brak wyświetlania czegokolwiek na wyjściu, a tylko zwracanie wyniku przez status (`grep`, `cmp`, `mail`, i inne). „Etatowym" narzędziem do sprawdzania warunków jest `test`, obsługujący bogaty język specyfikacji warunków.

Przykłady:

```
test -r filespec   # czy plik istnieje i jest dostepny do odczytu
test -d filespec   # czy plik istnieje i jest katalogiem

test -z string     # czy dany string ma dlugosc zero
test string        # czy dany string jest pusty
test str1 = str2   # czy dane dwa stringi sa identyczne

test n1 -eq n2     # czy dwie liczby calkowite rowne
test 1.1 -eq 1     # daje 0 (prawda) - przez zaokraglenie
test 1+1 -eq 2     # daje 1 (falsz) - nie oblicza wyrazen
test n1 -ge n2     # czy n1 >= n2, analogicznie -gt -le -lt
```

Jak widać, program `test` wykonuje pewne operacje liczbowe, np. zaokrąglanie, ale nie wykonuje obliczeń arytmetycznych.

Sprawdzanie innych warunków — `expr`

```
oblicz=5
echo $oblicz                    --> 5
oblicz=$oblicz+1
echo $oblicz                    --> 5+1

expr 3 + 4                      --> 7
expr 2 - 5                      --> -3
expr 10 / 3                     --> 3
expr 3 \* 8                     --> 24
expr 3 + 4 \* 5                 --> 23

expr 3+4                        --> 3+4

oblicz=5
oblicz=‘expr $oblicz + 8‘
echo $oblicz                    --> 13
```

Możliwości `expr`'a w zakresie obliczeń arytmetycznych nie sięgają głęboko. Kiedy potrzebne są obliczenia zmiennoprzecinkowe, można/należy zastosować program kalkulatora stosowego `dc`, używającego odwrotnej notacji polskiej, a żeby skorzystać z kilku podstawowych funkcji matematycznych niezbędny jest program `bc`, który jest preprocesorem `dc`.

```
oblicz=‘echo 4k 10 3 / p | dc‘
echo $oblicz                    --> 3.3333

oblicz=‘echo 4k 2 v p | dc‘
echo $oblicz                    --> 1.4142

# logarytm
oblicz=‘echo "l(2.73)" | bc -l -c | dc‘
echo $oblicz                    --> 1.00430160919686836451

# sinus
oblicz=‘echo "s(3.14)" | bc -l -c | dc‘
echo $oblicz                    --> .00159265291648695253
```

Bourne shell: przykład — przesyłanie danych pocztą

Celem tego skryptu, dawniej często używanego w Internecie, jest wywołanie kompletu programów niezbędnych do wysłania plików binarnych pocztą elektroniczną, a wcześniej pobranie od użytkownika niezbędnych parametrów:

```
# tarmail: send encoded files by mail

if test $# -lt 3; then
  echo "Usage: tarmail address \"subject\" directory-or-file(s)"
  exit
else
  address=$1
  echo "address = $address"
  shift
  subject="$1"
  echo "subject = $subject"
  shift
  echo files = $*
  tar cvf - $* | compress | btoa | mail -s "$subject" $address
fi
```

Bourne shell: przykład — skanowanie argumentów

Przykład pokazuje jak można dokonać przeglądu listy argumentów opcjonalnych skryptu, z zapamiętaniem ich w zmiennych, po czym w wektorze argumentów pozostają jedynie argumenty pozycyjne, stanowiące rzeczywiste operandy:

```
USAGE="usage: $0 [-a] [-b] [-o oarg] arg_1 arg_2 ..."
while [ $# -ge 1 ]
do
    case $1 in
    -a)    AFLAG=1; shift ;;
    -b)    BFLAG=1; shift ;;
    -o)    OARG=$2; shift 2 ;;
    -*)    echo $USAGE; exit 1 ;;
    *)     break ;;
    esac
done

echo arguments: "-a=$AFLAG" "-b=$BFLAG" "-o=$OARG" "args=$*"
```

Konwencja dla argumentów wywołania

Istnieje konwencja dla argumentów wywołania programów i skryptów, aby odróżnić argumenty opcjonalne (flagi), które powinny być zapisane pojedynczą literą z minusem, od argumentów pozycyjnych, które mogą mieć dowolną postać. Argumenty opcjonalne mogą posiadać wartość, zapisywaną w kolejnym argumencie, a te, które nie mają wartości, można grupować, z pojedynczym minusem dla całej grupy. Dodatkowo, argumenty opcjonalne, z wartościami lub bez, powinny wystąpić w wektorze wywołania przed argumentami pozycyjnymi, i mogą być oddzielone od argumentów pozycyjnych dodatkowym argumentem w postaci dwóch minusów, który pełni tylko rolę separatora.

Na przykład, skrypt, który posiada dwa argumenty opcjonalne -a i -b, oraz trzeci argument opcjonalny -o z wartością, i dowolną liczbę argumentów pozycyjnych, może być wywołany w następujące, równoważne sposoby:

```
skrypt  -a  -b  -o  wart_o  pozyc_1  pozyc_2  pozyc_3
skrypt  -ab  -o  wart_o  pozyc_1  pozyc_2  pozyc_3
skrypt  -ab  -o  wart_o  --  pozyc_1  pozyc_2  pozyc_3
```

Niestety, nie wszystkie programy systemu Unix stosują się do tej konwencji (np. tar, find, dd, i inne).

Bourne shell — getopt

Wspomniana konwencja dla argumentów wywołania upraszcza pisanie kodu do analizy tych argumentów w skrypcie lub programie. Parsowanie argumentów opcjonalnych można wykonać według schematu przedstawionego w przykładzie wyżej. Jednak możliwość grupowania opcji jest z tym schematem niezgodna. Istnieje narzędzie `getopt` porządkujące wektor argumentów przez rozdzielenie zgrupowanych opcji, po czym można zastosować powyższy schemat:

```
set -- `getopt abo: $*`
if [ $? != 0 ] ; then echo $USAGE; exit 2; fi
while [ $# -ge 1 ]
do
    case $1 in
    -a)    AFLAG=1; shift ;;
    -b)    BFLAG=1; shift ;;
    -o)    OARG=$2; shift 2 ;;
    --)    shift; break ;;
    -*)    echo $USAGE; exit 1 ;;
    *)     break ;;
    esac
done
```

Bourne shell: skierowanie *here document*

```
echo Creating /etc/resolv.conf ...
nameserver=$1
cat > /etc/resolv.conf << END-OF-RESOLV-CONF
domain stud.ii
search stud.ii prac.ii ii.uni.wroc.pl
nameserver $nameserver
END-OF-RESOLV-CONF
echo Done /etc/resolv.conf

echo Fixing /etc/hosts ...
ed /etc/hosts << \END-OF-EDIT-STRING
1,$s/stud.ii/ii.uni.wroc.pl/
1,$s/prac.ii/ii.uni.wroc.pl/
w
q
END-OF-EDIT-STRING
echo Done /etc/hosts
```

Bourne shell: funkcje

```
ask_yes_no() {
  answer=X
  while true
  do
    echo The question: $1
    echo Answer yes or no:
    read answer
    case $answer in
      yes|Yes|YES) return 0;;
      no|No|NO)    return 1;;
    esac
    echo Wrong answer.
    echo ""
  done
}
```

```
# przyklad wywolania:
if ask_yes_no "Czy kasowac\
         pliki tymczasowe?"
then
  rm -f *.o a.out
fi
```

Można również przechwycić dane wyświetlane przez funkcję na wyjściu, jednak wtedy np. konwersacja z użytkownikiem musiałaby odbywać się przez stderr.

Bourne shell: przykład — samorozpakowujące archiwum

```
# bundle: pakujemy wiele plikow do skryptu

echo '# w celu rozpakowania przepusc przez /bin/sh'
for i in $*
do
  echo "echo tworzymy plik $i 1>&2"
  echo "cat >$i <<'Koniec danych pliku $i'"
  cat $i
  echo "Koniec danych pliku $i"
done
```

Bourne shell: parametry opcjonalne

Bourne shell posiada opcjonalne argumenty wywołania (flagi), które w wektorze argumentów nie liczą się jako argumenty pozycyjne $1, $2, ... Można je podać w wywołaniu skryptu, albo ustawić w czasie pracy poleceniem `set`, np. `set -f`:

-v powoduje wyświetlenie wczytanych linii polecenia

-x powoduje wyświetlenie poleceń przed wykonaniem, po interpretacji

-n powoduje tylko wyświetlanie poleceń do wykonania, bez wykonania

-e powoduje zatrzymanie interpretera z błędem jeśli jakiekolwiek polecenie zwróci niezerowy status

-u powoduje wygenerowanie błędu przy próbie użycia niepodstawionej zmiennej

-f powoduje wyłączenie dopasowania nazw plików do znaków specjalnych takich jak $*$

Niezależnie od ustawienia flagi -u dostępna jest konstrukcja ${zm?} generująca błąd (status 1) gdy zmienna `zm` jest niepodstawiona.

Po ustawieniu danej flagi, można ją ponownie wyłączyć zamieniając minus na plus, np. `set +f` ponownie włącza mechanizm dopasowania nazw plików.

Bourne shell: obsługa przerwań

W Unixie istnieje mechanizm przerwań programowych zwanych sygnałami. Istnieje około 20-30 zdefiniowanych sygnałów, z których większość ma przypisane funkcje związane z funkcjonowaniem sprzętu bądź zachowaniem programu. Jądro systemu „doręcza" sygnały procesom, i otrzymanie sygnału oznacza zwykle, że proces powinien natychmiast zakończyć się. Mówi się, że domyślną reakcją na otrzymanie sygnału jest śmierć. Proces może jednak zmienić tę reakcję:

```
trap "rm tmp_file; echo trapped; exit" 1 2 3 4 5 6 7 8 10 12 13 14 15
```

UWAGA: jakkolwiek obsługa sygnałów może być prostą metodą interakcji z użytkownikiem, niepotrzebne przechwytywanie sygnałów prowadzi często do tworzenia „nieśmiertelnych" procesów, które gdy wygenerują jakiś błąd, mogą być bardzo kłopotliwe. Ogólnie łatwość uśmiercania procesów jest zaletą, podobnie jak akceptowanie sygnałów przysyłanych przez system.

Bourne shell: skierowania — full story

Uruchomiony proces może posługiwać się wieloma strumieniami danych (dla których system tworzy struktury zwane deskryptorami plików).

Deskryptory te mają nadawane kolejne numery, począwszy od 0, i po otwarciu pliku proces wykonuje na nich operacje wejścia/wyjścia odwołuje się do nich przez numery deskryptorów.

Normalnie procesy są uruchamiane z trzema otwartymi deskryptorami: 0 (wejście), oraz 1, i 2 (wyjście) i są one skojarzone przez system z terminalem użytkownika (wejścia z klawiaturą, a wyjścia z monitorem). Strumienie danych 0 i 1 stanowią tzw. standardowe wejście i wyjście, a strumień 2 stanowi standardowe wyjście błędów.

Przy uruchamianiu procesu system może skojarzyć dowolny strumień danych z plikiem dyskowym, co powoduje otwarcie tego pliku i wykonywanie operacji wejścia/wyjścia na tym pliku bez konieczności jawnego otwierania konkretnego pliku w programie.

`<&n >&n` — skierowanie wejścia (stdin) lub wyjścia (stdout) z/do pliku otwartego dla strumienia danych n

`m<&n m>&n` — skierowanie strumienia wejściowego lub wyjściowego m z/do pliku otwartego dla strumienia danych n

---

Bourne shell: magia #!

Większość współczesnych unixowych interpreterów poleceń stosuje konwencję polegającą na specjalnym traktowaniu skryptów, których pierwsze dwa bajty to `#!` (fachowa wymowa anglojęzyczna: *sha-bang*). Do wykonania takich skryptów nasz interpreter poleceń wywołuje — jako interpreter do wykonania skryptu — program określony w pierwszym wierszu skryptu, zaczynającym się właśnie od `#!`. Dalszy ciąg wiersza traktowany jest jako nazwa programu do wywołania. Jednak musi to być nazwa programu w postaci pełnej bezwzględnej ścieżki pliku, ponieważ przy wywoływaniu tego programu nie stosuje się normalnego algorytmu wykonania (ani interpretacji) polecenia.

```
#!/usr/bin/perl

use Config qw(myconfig);
print myconfig();
```

UWAGA: pisząc skrypt pod określony interpreter najczęściej nie mamy pewności czy taki interpreter jest dostępny w każdym systemie, a nawet jeśli jest, to jaka dokładnie jest jego ścieżka pliku. Jedynym interpreterem, na który zawsze można liczyć jest Bourne shell w pliku `/bin/sh`.

Bourne shell: :, eval i exec

: — powoduje tylko interpretację argumentów, bez wykonania żadnego polecenia

```
: ${par1:?blad1} ${par2:=podstaw2}
```

eval — powoduje wykonanie polecenia otrzymanego z interpretacji argumentów:

Mamy listę plików w `/tmp/lista`:

```
$ cat /tmp/lista
130_0423/imgp987?.jpg \
132_0520/imgp994[2-5].jpg \
134_0617/imgp99{88,89,90}.jpg
$
```

Porównaj:

```
$ echo "‘cat /tmp/lista‘"
$ echo ‘cat /tmp/lista‘
$ eval echo ‘cat /tmp/lista‘
```

exec — powoduje normalne wykonanie polecenia, ale utworzony proces zastępuje proces interpretera komend, który „wyparowuje":

```
$ exec dlugiprogram
$ dlugiprogram ; exit
```

Zauważmy, że drugie wywołanie, na pozór podobne do pierwszego, może jednak zostać przerwane (`^C`), co powoduje powrót do shella. W pierwszym wywołaniu interpreter komend znika od razu.

C-shell: dodatkowe znaki specjalne

Interpretery poleceń z rodziny C-shell mają pewne dodatkowe metaznaki, które są rozwijane na etapie interpretacji polecenia: `{str1,str2,...}` `~[user]`

```
ls -l {prog,cwicz}*.c
xv ~witold/*.pgm
```

Zwróćmy uwagę, że pomimo iż te mechanizmy są często używane w nazwach plików, to ich działanie nie jest uzależnione od istnienia plików o danych nazwach, np.:

```
echo ~witold/{nie,ma,takich,plików}.JPEG

WYNIK:
/home/witold/nie.JPEG /home/witold/ma.JPEG
/home/witold/takich.JPEG /home/witold/plików.JPEG
```

Unixowe interpretery komend — historia

- oryginalny interpreter komend: Bourne shell (`/bin/sh`)
- zmodyfikowany interpreter komend do pracy interakcyjnej: C-shell (`/bin/csh`) zawiera wiele ulepszeń i zmienioną składnię złożonych poleceń programowych
- ulepszona wersja C-shella (`/bin/tcsh`), zawiera edycję ekranową wiersza komendy
- ulepszone wersje Bourne shella: Korn shell (`/bin/ksh`), Bourne Again Shell (`/bin/bash`); ten ostatni zawiera mechanizm historii i kontrolę podprocesów w stylu C-shella, zachowując jednak składnię poleceń programowych Bourne shella, oraz dodając ekranową edycję komend
- specyfikacja POSIX interpretera komend: wprowadza standard zasadniczo zgodny z Bourne shellem, uwzględniająca wiele rozszerzeń Korn shella i basha

C-shell: modyfikatory ":"

```
    lpipascal program.p    #wywolanie kompilatora Pascala
    ldpascal program.o     #wywolanie linkera


#!/bin/csh -f

echo "lpipascal -longint -o $1:r.o $1"
lpipascal -longint -o $1:r.o $1
if ($status != 0) then
        echo 'Pascal compilation failed, linking not done\!\!'
        exit $status
endif
if ($1 == $1:r) then
        echo "ldpascal -strip -o a.out $1:r.o"
        ldpascal -strip -o a.out $1:r.o
        exit 0
endif
echo "ldpascal -strip -o $1:r $1:r.o"
ldpascal -strip -o $1:r $1:r.o
```

modyfikatory wartości wyrażeń: **:h :r :e :t :s/old/new/ :q :x**

Unixowe interpretery komend — praktyka

- Współcześnie używane interpretery (tcsh, ksh, i bash) różnią się minimalnie, głównie składnią poleceń programowych (warunkowych i pętli). W pracy interakcyjnej, gdzie te polecenia wykorzystuje się rzadko, można nie zorientować się nawet jakiego interpretera używamy w danej chwili.
- Z punktu widzenia pisania skryptów i ich maksymalnej kompatybilności wstecznej, i przenośności na największą liczbę systemów uniksowych, należy brać pod uwagę wyłącznie oryginalny Bourne shell. Ograniczenie się do Bourne shella nie oznacza rezygnacji z jakichś ważnych funkcji, a jedynie mniejszą wygodę pisania skryptów.
- Pisząc skrypty pod kątem ich przenośności dla systemów współczesnych, warto brać pod uwagę standard POSIX i używać konstrukcji dobrze zdefiniowanych przez ten standard. Takie przenośne skrypty nie powinny jednak odwoływać się do konkretnego interpretera mechanizmem `#!`

POSIX shell: obliczanie wartości zmiennych

W Bourne shellu, poza podstawową postacią odwołania się do wartości zmiennej `$var`, albo jej ogólniejszą postacią `${var}` istnieje szereg dodatkowych postaci składniowych uruchamiających dotatkowe funkcjonalności:

`${var:-default}`	użyj wartości domyślnej jeśli nie ma ona wartości lub null
`${var:=default}`	użyj wartości domyślnej j.w. i jednocześnie podstaw zmienną; nie można w ten sposób podstawić parametrów pozycyjnych
`${var:?msg}`	użyj wartości zmiennej jeśli istnieje i jest non-null, w.p.w. wyświetl komunikat i zakończ skrypt z błędem

Standard POSIX dodatkowo wprowadził kilka dalszych podobnych operatorów:

`${zm:+value}`	użyj podanej wartości jeśli zmienna miała już wartość non-null
`${#zm}`	oblicz długość wartości (stringa)
`${zm%suf}`	usuń najkrótszy przyrostek
`${zm%%suf}`	usuń najdłuższy przyrostek
`${zm#pref}`	usuń najkrótszy przedrostek
`${zm##pref}`	usuń najdłuższy przedrostek

POSIX shell: alternatywna składnia komend zagnieżdżonych

POSIX wprowadził również alternatywną notację dla poleceń zagnieżdżonych, oryginalnie zapisywanych apostrofami wstecznymi:

```
$(polecenie)
```

co jest równoważne tradycyjnej składni poleceń zagnieżdżonych Bourne shella:

```
‘polecenie‘
```

Alternatywna składnia pozwala w sensowny sposób zagnieżdżać w sobie wiele poleceń, co w oryginalnym Bourne shellu wymagało karkołomnej ekwilibrystyki.

POSIX shell: operatory arytmetyczne

Interpretery poleceń zgodne ze standardem POSIX, takie jak `bash` i `ksh` realizują szereg dodatkowych operacji, które ułatwiają pisanie skryptów. Należą do nich np. operatory arytmetyczne:

```
echo 2+2= $((2+2))
```

Jednak w wyrażeniach arytmetycznych zapisywanych w podwójnych nawiasach trzeba uważać na operatory porównania, ponieważ zwracają one wartości zgodne z konwencją języków takich jak C, czyli prawda jest reprezentowana przez 1 a fałsz przez 0, odwrotnie niż w konwencji wartości logicznych interpretowanych przez status polecenia.

```
echo '3>2?' $((3>2))
```

POSIX shell: operatory arytmetyczne (cd.)

Standard POSIX pozostawia jednak pewną dowolność w implementacji operatorów arytmetycznych, np. nie wymaga implementacji operatorów -- ani ++. Niestety, powoduje to dwuznaczność interpretacji:

```
$ bash -c 'b=5; echo $((--b)); echo $((--b))'
4
3
$ zsh -c 'b=5; echo $((--b)); echo $((--b))'
4
3
$ ksh -c 'b=5; echo $((--b)); echo $((--b))'
5
5
$ dash -c 'b=5; echo $((--b)); echo $((--b))'
5
5
```

W tym przypadku dash i ksh zinterpretowały podwójny minus jako podwójne przeczenie i obliczyły poprawny wynik.

POSIX shell: dopasowanie nazw plików

Standard POSIX rozszerzył mechanizm *globbing* dopasowania metaznaków `*`, `?`, `[...]` do nazw plików o klasy znaków za pomocą wyrażenia `[[:klasa:]]`, z następującymi klasami znaków:

- `[:digit:]`
- `[:alpha:]`
- `[:lower:]`
- `[:upper:]`
- `[:punct:]`

Uwagi na temat basha

- `bash` jest prawdopodobnie najbardziej rozbudowanym interpreterem komend, który jest zarazem bardzo mocno zgodny ze standardem POSIX, a poza tym jest produktem typu „open source," na skutek czego jest często instalowany na systemach linuksowych jako domyślny interpreter.
- W ten sposób `bash` szybko staje się najpopularniejszym ☺ i zarazem jedynym ☹ shellem.
- Jednak `bash` posiada szereg rozszerzeń w stosunku do standardu POSIX; przy czym istnieje niezliczona liczba podręczników z serii *„kruczki i sztuczki basha"* intensywnie eksploatujących te rozszerzenia.
- W rzeczywistości `bash` nie jest jedynym interpreterem, i nie można zakładać ani że jest interpreterem użytkownika, ani że w ogóle jest zainstalowany na danym systemie. Skrypty odwołujące się do `basha` (wywołując go jawnie, lub przez mechanizm `#!`), albo wykorzystujące specyficzne konstrukcje `basha`, nie będą działać we wszystkich środowiskach uniksowych.
- Na przykład, jako produkt „open source" `bash` nie może być stosowany w środowiskach i zastosowaniach komercyjnych pozostających w sprzeczności z warunkami licencji GPL.

POSIX shell: lokalizacja

LC_COLLATE Określenie schematu porządkowania napisów znakowych

LC_CTYPE Określenie schematu typów znakowych.

LC_MESSAGES Określenie języka komunikatów.

LC_NUMERIC Określenie zestawu konwencji prezentacji wartości liczbowych.

LC_TIME Określenie zestawu konwencji formatowania daty i czasu.

LC_ALL Wartość przesłaniająca wszystkie pozostałe zmienne LC_*

LANG Domyślna wartość dla wszystkich nieustawionych zmiennych LC_*

Krótkie podsumowanie — pytania sprawdzające

1. Jaka jest rola i znaczenie potoków?
2. Jaką rolę pełni interpretacja polecenia?
3. Co to jest „globbing"?
4. Co to są polecenia zagnieżdżone?
5. Co to jest status wykonanego polecenia?
6. Z jakich elementów składa się polecenie warunkowe „if"?
   (Chodzi o elementy znaczeniowe, nie o składnię polecenia.)
7. Jaka jest rola skierowania „here document" w skryptach?
8. Jak działa polecenie „exec"?