Uniksowe interpretery poleceń

Witold Paluszyński
Katedra Cybernetyki i Robotyki
Politechnika Wrocławska
http://www.kcir.pwr.edu.pl/~witold/

Interpretery poleceń w systemach operacyjnych

Interpreter poleceń jest programem funkcjonującym jako interfejs użytkownika w systemach operacyjnych. Jego rolą jest wykonywanie poleceń użytkownika, których celem jest zwykle uruchomienie jakiegoś programu. Funkcje interpretera poleceń można więc streścić jako: **czytanie poleceń i wykonywanie programów**.

W prehistorycznych systemach operacyjnych interpretery poleceń miały formę szczątkową, zwłaszcza w systemach przeznaczonych do pracy wsadowej, i nie zawsze były odrębnymi programami. Potem jednak, w miarę rozwoju interakcyjnych systemów operacyjnych, uzyskały pełne prawa obywatelskie, i były rozbudowywane o mechanizmy ułatwiające pracę użytkownikom interakcyjnym.

Uniksowe interpretery poleceń zajmują szczególną pozycję, ponieważ ich język poleceń ma wiele konstrukcji typu programistycznego, umożliwiających pisanie zaawansowanych zadań wsadowych — **skryptów**. Jednocześnie mają wiele mechanizmów ułatwiających interakcyjną pracę z systemem. W tradycji systemów uniksowych interpreter poleceń nazywany jest *shell*-em, czyli skorupą izolującą użytkownika od **jądra** systemu, które wykonuje jego właściwe funkcje.

Polecenia uniksowego interpretera poleceń

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

```
who                     # program
ls -l                   # program z argumentem
/bin/ps -ef             # program z pelna sciezka pliku
./oblicz maj.txt        # program z katalogu biezacego
set -x                  # polecenie wbudowane z argumentem
```

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

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.

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

jobs

fg %<n>

bg %<n>

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

Podprocesy identyfikowane są kolejnymi liczbami, a w odwołaniu do nich piszemy numer podprocesu poprzedzony znakiem procenta.

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 system budzi je w sposób przezroczysty, gdy pozostałe procesy nadążyły z przetworzeniem 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.

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.

Przekierowania wejścia/wyjścia

W systemach uniksowych procesy mogą mieć otwarte strumienie danych wejścia/wyjścia, zwane również deskryptorami plików, które są numerowane sekwencyjnie od 0 wzwyż. Dla procesów uruchamianych na terminalu deskryptory 0, 1, i 2 (nazywane odpowiednio `stdin`, `stdout`, i `stderr`) są w czasie inicjalizacji procesu przyporządkowane do klawiatury i ekranu terminala.

Interpreter poleceń posiada mechanizm **przekierowania** tych strumieni w taki sposób, że otwierane są pliki na dysku i dane są przez proces czytane z, lub zapisywane na tych plikach. Podstawowa forma:

```
./prog < plik_wejscia > plik_wyjscia 2> plik_bledow
```

Skierowania strumieni wejścia i wyjścia mają jeszcze szereg innych postaci, z którymi warto się zapoznać. Przykłady:

```
./prog >> plik_wyjscia      # stdout dopis.na kon.pliku
./prog > /dev/null          # stdout calk. ignorowane
./prog > plik_wyjscia 2>&1  # stderr do tego samego pliku
./prog 2>&1 > /dev/null     # stdout ign.,stderr na stdout
```

Skrypty interpretera komend

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

```
who | wc -l
```

Skrypt można uruchomić, wywołując interpreter poleceń (obecny na większości systemów jako `/bin/sh`), z nazwą skryptu jako argumentem:

```
echo 'who | wc -l' > ilu_userow.sh
sh ilu_userow.sh
.  ilu_userow.sh
```

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 `ilu_userow.sh` atrybut wykonywalności `x`. Wtedy można spowodować wykonanie skryptu przez użycie nazwy pliku jako polecenia:

```
chmod +x ilu_userow.sh
./ilu_userow.sh
```

W tym przypadku bieżący interpreter poleceń wywoła **drugi interpreter** dla wykonania tego skryptu. Normalnie będzie to druga instancja tego samego programu, która w systemie będzie **podprocesem** procesu bieżącego. Przy tej formie wywołania nie ma możliwości spowodowania, aby skrypt został wykonany przez nasz macierzysty shell.

Zmienna PATH

W powyższym przykładzie nazwa pliku została podana w postaci względnej ścieżki do pliku, który znajduje się w katalogu bieżącym. Wymusza to wywołanie pliku o podanej nazwie. Nie możemy podać nazwy pliku `ilu_userow.sh` bez podania ścieżki katalogów (przynajmniej szczątkowej). Plik wykonywalny zostanie uruchomiony tylko wtedy, gdy znajduje się w katalogu wymienionym na liście katalogów pamiętanych w zmiennej PATH:

```
./ilu_userow.sh   # poprawne
ilu_userow.sh     # sh: ilu_userow.sh: not found
PATH=$PATH:.      # dopisz biezacy katalog na koniec PATH
ilu_userow.sh     # poprawne
```

Zwróćmy uwagę — można odwołać się do katalogu bieżącego przez symboliczną nazwę kropki. Zwyczajowo jednak, katalogu bieżącego nie umieszcza się na standardowej ścieżce katalogów programowych pamiętanej w zmiennej PATH.

Możnaby to zrobić dla wygody, wpisując polecenie do pliku startowego shella. Jednak jest to uważane za pewne zagrożenie, ponieważ w trakcie pracy można w sposób niezamierzony wywołać jakiś program z bieżącego katalogu. Jeśli jest to konieczne, to kropka powinna znajdować się na końcu ścieżki katalogów.

Inne zmienne systemowe

Jak widzieliśmy, zmienna PATH pełni w uniksowym interpreterze poleceń szczególną rolę. Jest szereg takich zmiennych, wykorzystywanych w różnych rolach przez różne części systemu Unix:

PATH	ścieżka wyszukiwania programów interpretera poleceń
TERM	identyfikator typu terminala, na którym wykonuje się program; przydatny gdy program chce wykonać jakąś operację na ekranie terminala, np. zgaszenie ekranu, lub wyświetlanie tekstu w określonej pozycji
MAIL	ścieżka do pliku z systemową 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`
LANG	nazwa lokalizacji użytkownika, określająca język i konwencje narodowe

I szereg innych.

Najprostsze skrypty

Skrypty pisze się w celu automatycznego, powtarzalnego wykonania pewnych operacji. Ma to sens, gdy ten zestaw operacji jest długi i/lub skomplikowany. Jakkolwiek doświadczeni użytkownicy piszą rozbudowane i złożone skrypty (a z upływem czasu rozbudowują je często do monstrualnych rozmiarów), to największa przydatność skryptów jest w automatyzacji prostych czynności, których nie chcemy każdorazowo pisać po kawałku, a ich napisanie w postaci skryptu jest trywialne.

Przykłady prostych skryptów:

```
echo Do systemu jest zalogowanych
who | wc -l
echo uzytkownikow.

echo W katalogu znajduje sie
ls -1 | wc -l
echo plikow.
```

Polecenia zagnieżdżone

W powyższych przykładach działanie skryptu opierało się na tym, że zarówno programy systemowe, jak i polecenie echo, wyświetlają swoje komunikaty na wyjściu `stdout`, i użytkownik widzi je łącznie.

Można wykorzystać mechanizm **poleceń zagnieżdżonych** aby połączyć teksty wyświetlane przez program z innymi tekstami. Można zagnieździć dowolne polecenie w dowolnym miejscu innego, przez objęcie go znakami **apostrofów wstecznych** ‘ . . . ‘ (*backquote*). Jest ono wykonywane przez drugą instancję interpretera poleceń przed rozpoczęciem wykonywania polecenia głównego. Polecenie zagnieżdżone może wykonać dowolne operacje, a system zbiera wynik jego pracy w postaci wysyłanych na wyjście znaków, a następnie podmienia treść polecenia (razem z apostrofami wstecznymi) otrzymanym ciągiem znaków.

Zmodyfikowana wersja poprzednich skryptów:

```
echo Do systemu jest zalogowanych ‘who|wc -l‘ uzytkownikow
echo W katalogu znajduje sie ‘ls -1 | wc -l‘ plikow.
```

Dodatkową korzyścią tej wersji jest wyświetlanie całości w jednym wierszu, ponieważ mechanizm zagnieżdżania zamienia znaki nowej linii na spacje.

Zmienne shella

Zmienne systemowe typu PATH albo LANG nie są niczym szczególnym. Zmienną można utworzyć w dowolnym momencie, bez deklarowania, przypisując jej jakąś wartość. Można również przypisać wartość istniejącej zmiennej, zastępując poprzednią wartość. Aby obliczyć wartość zmiennej trzeba użyć wyrażenia ze znakiem dolara przed nazwą zmiennej.

```
a=5                  # WAZNE: zadnych spacji przed i za "="
echo zmienna a ma wartosc $a
```

Można użyć zmiennych aby skonstruować kolejną (niekoniecznie lepszą) wersję poprzednich przykładowych skryptów:

```
n_userow=‘who | wc -l‘
echo Do systemu jest zalogowanych $n_userow uzytkownikow.

n_plikow=‘ls -1 | wc -l‘
echo W katalogu znajduje sie $n_plikow plikow.
```

Zmienne interpretera poleceń mają zasadniczo wartości tekstowe. Jeśli program chce odczytać ze zmiennej wartość liczbową, to musi sam ją sobie zdekodować.

Pułapki ze zmiennymi

Zmiennych shella uniksowego nie trzeba (ani nie można) deklarować. Co gorsza jednak, **dopuszczalne jest odwołanie się do nieistniejącej zmiennej**. Nie jest to błąd, tylko daje wartość pustego stringa. Ten mechanizm powoduje, że można paść ofiarą błędu wynikającego z odwołania się do niewłaściwej zmiennej.

Przykład:

```
n_plikow=‘ls -1 | wc -l‘
echo W katalogu znajduje sie $n_pilkow plikow.
```

W powyższym przykładzie zobaczymy komunikat z pustym miejscem zamiast liczby plików, ale w ogólności może to być niewłaściwa wartość (np. poprzednia).

Niestety, jest to **nieuleczalna choroba shella uniksowego**. Co prawda istnieje flaga interpretera poleceń (**-u**) wymuszająca błąd w takich przypadkach (patrz poniżej). Jednak domyślnie flaga ta nie jest ustawiona, zatem można ją ustawiać dowolną liczbę razy w różnych miejscach, a wciąż będziemy mieli do czynienia z sytuacjami, gdzie flaga będzie nieustawiona.

Polecenie read

Polecenie `echo` jest wygodnym narzędziem komunikacji skryptu z użytkownikiem. Aby jednak odczytać odpowiedź użytkownika, potrzebny jest jakiś inny mechanizm, na przykład polecenie `read`. Wczytuje ono jeden wiersz ze standardowego wejścia (normalnie połączonego z klawiaturą użytkownika), i podstawia pod podaną zmienną.

Przykład:

```
echo Podaj adres IP bramy domyslnej w sieci lokalnej
read brama
route add default gw $brama
```

W ogólnym przypadku polecenie `read` czyta cały wiersz z wejścia, ale dekoduje go na „słowa", i podstawia nimi kolejne podane zmienne. Jeśli słów będzie za mało to niektóre zmienne pozostaną niepodstawione, a jeśli słów będzie za dużo, to ostatnia zmienna otrzyma wartość wielosłowową.

```
echo Podaj kilka ulubionych imion:
read pierwsze inne
echo Pewnie $pierwsze bardziej Ci sie podoba niz $inne
```

Zmienne globalne

Nowo utworzone zmienne shella są **lokalne** dla bieżącej instancji interpretera. Programy i skrypty wywoływane w czasie pracy nie mają do nich dostępu. Można zmienną **eksportować** czyniąc ją **globalną**:

```
ZMGLOB=20           # zmienna ZMGLOB ma wartosc 20
polec               # polecenie nie ma dostepu do zmiennej
export ZMGLOB
polec               # teraz polecenie ma dostep do zmiennej
```

Zmienne można tworzyć w dowolnym momencie. jak również unicestwiać poleceniem `unset`. Istnieje również postać wywołania polecenia, w której zmienna jest tworzona i eksportowana tylko na czas wykonywania polecenia:

```
ZMGLOB2=40 polec   # polecenie ma dostep do zmiennej
# ale potem nie ma sladu ani wartosci ani zmiennej
```

Więcej o eksportowaniu zmiennych

Mechanizm „eksportowania" zmiennych jest specyficzny i trzeba dobrze go rozumieć. Polega on na utworzeniu kopii wszystkich zmiennych eksportowanych, wraz z ich wartościami — tzw. **środowisko procesu** — dla każdego tworzonego podprocesu. Podproces może robić ze zmiennymi co zechce, jednak gdy kończy pracę, cały komplet jego zmiennych znika bez śladu.

```
export A
A=25
sh                    # wewn. interpreter dziedziczy zmienna
                      # zmienna A ma wartosc 25
                      A=50
                      # teraz A ma wartosc 50
                      exit
# A ma znow wartosc 25
```

Jak z tego wynika, operacje na zmiennych można wykonywać tylko poleceniami wbudowanymi, a nie można programami ani skryptami, które wykonują się w podprocesach. Oczywiście, polecenie przypisania wartości zmiennej (`A=...`), i polecenie `read` muszą być poleceniami wbudowanymi shella (dlaczego?).

Podstawowy błąd

W teorii, eksportowanie środowiska do podprocesu jest prostą koncepcją, którą każdy może zrozumieć. Często jednak bywa popełniany błąd według schematu:

```
echo Podaj preferowany kolor:
read KOLOR
export KOLOR
exit
```

Jeśli powyższe będzie wywołane jako skrypt, to proces wywołujący nigdy nie dowie się jaki jest preferowany kolor użytkownika, nawet jeśli sam wcześniej wyeksportował powyższe zmienne.

Przypomnijmy, wywołujący proces interpretera poleceń **może wykonać taki skrypt bezpośrednio sam, poleceniem .** (kropka). Wtedy wszystko zadziała jak należy, zmienna zostanie utworzona z odpowiednią wartością w procesie wywołującym. Jednak ten sposób nie jest ogólnie wygodny. Pozwala tylko na wywoływanie skryptów (nie programów), i tylko gdy interpreter poleceń użytkownika jest dokładnie tym samym w jakim został napisany skrypt. Ponadto, skrypty, których efekty zależą od sposobu wywołania są mylące dla użytkowników.

Przekazywanie wartości przez strumienie danych

Rozważmy ponownie powyższy przykład; chcemy napisać skrypt który odpyta użytkownika o preferowany kolor, i przekaże go procesowi nadrzędnemu.

> Pytanie: czy da się napisać skrypt, który będzie w stanie przekazać wynik swej pracy do procesu wywołującego?
>
> Odpowiedź: tak, ale nie przez zmienne tylko przez strumień wejścia/wyjścia.

Zatem, zamiast eksportować zmienne, powinien on wysłać otrzymane wartości na swoje wyjście. Kluczową kwestią jest, żeby przypisanie wartości zmiennych realizował proces macierzysty, bo tylko on ma dostęp do właściwych zmiennych.

```
KOLOR=‘odpytaj_kolor.sh‘
```

Pomimo iż powyższe przypisanie wygląda niewinnie, wymaga nietrywialnej modyfikacji w skrypcie `odpytaj_kolor.sh`. Przyczyną jest **niejawne skierowanie wyjścia, wynikające z mechanizmu zagnieżdżenia**.

Skrypt `odpytaj_kolor.sh` zmodyfikowany na potrzeby przekazania wyniku przez `stdout` ma następującą postać:

```
echo Podaj preferowany kolor: 1>&2
read KOLOR
echo $KOLOR
exit
```

Dialog z użytkownikiem nie może już być zrealizowany przez zwykłe polecenie `echo`, ponieważ standardowe wyjście skryptu zostało przekierowane przez proces wywołujący. W powyższym, wyjście `stdout` polecenia `echo` zostało przekierowane na `stderr`. Ten strumień najczęściej nie jest nigdzie przekierowywany, aby nie zakłócać raportowania błędów (wysłanie zapytania do użytkownika nie zakłóci ewentualnego komunikatu o błędzie jakiegoś polecenia).

Zauważmy, że powyższe przekierowanie wyjścia polecenia `echo` nie zmniejsza jego ogólności. Gdyby proces wywołujący nie przekierował mu wyjścia, zadziałałby tak samo poprawnie. Przy pisaniu skryptów warto brać pod uwagę możliwość, że „moje" wejście lub wyjście będzie przekierowane. Pisane w ten sposób skrypty są bardzo uniwersalne i dają się wywoływać na wiele sposobów.

Lekkie przegięcie

Wyobraźmy sobie, że chcemy wykonać jeszcze jedną modyfikację skryptu `odpytaj_kolor.sh` i przekazać mu kolor domyślny, który zostanie użyty, jeśli użytkownik nie poda swojego (odpowie naciśnięciem klawisza ENTER).

Pomijamy tu fakt, że łatwo byłoby przekazać wartość koloru domyślnego przez argumenty wywołania (o nich patrz niżej). Załóżmy jednak, że z jakiegoś powodu chcemy przekazać ją tak jak wartość wynikową, przez strumień danych. Jak poradzić sobie z przekierowanymi obydwoma strumieniami `stdin` i `stdout`?

```
KOLOR=‘echo PINK | odpytaj_kolor.sh‘
```

Można w tym celu wykorzystać istniejący w systemie plik specjalny terminala, dostępny w katalogu urządzeń jako `/dev/tty`.

```
read DOMYSLNY
echo Podaj preferowany kolor \[$DOMYSLNY\]: > /dev/tty
read PREFEROWANY                             < /dev/tty
if [ -z "$PREFEROWANY" ]
then echo $DOMYSLNY
else echo $PREFEROWANY
fi
```

Dygresja na temat echa

We wcześniejszym przykładzie pojawiła się często stosowana konstrukcja zapytania do użytkownika (tu pomijamy rozważane poprzednio przekierowania):

```
echo Podaj preferowany kolor \[$DOMYSLNY\]:
read PREFEROWANY
```

Polecenie `echo` domyślnie dopisuje NEWLINE do wyświetlanego stringa. Dzięki temu łatwo wyświetlić na wyjściu pusty wiersz wywołując `echo` bez argumentów.

Jednak skutkiem tego w powyższym dialogu jest, że użytkownik odpowiada w następnym wierszu. Dialog wyglądałby lepiej, gdyby użytkownik mógł odpowiedzieć w wierszu pytania. Dlatego też polecenie `echo` posiada **opcjonalną możliwość pomijania NEWLINE-a**.

Niestety, **jest to opcja niestandardowa**. Polecenie `echo` ma wiele wersji — jest i programem wewnętrznym, i poleceniem wbudowanym większości interpreterów poleceń — i różne wersje różnie implementują tę opcję.

Historia wersji echa, i różnic między nimi, jest niemal tak stara jak system Unix. Niestety, **nie ma dobrej metody przenośnego wywołania polecenia echo, jeśli chcemy użyć jednej z jego opcji**. To co należy zrobić?

Zamiast echo można użyć printf

Jeśli chcemy wyświetlić (wysłać na wyjście) komunikat z niestandardowymi opcjami, typowo z pominięciem końcowego NEWLINE-a, lepszą możliwością jest skorzystanie z polecenia **printf, które jest nowsze i bardziej przenośne**.

```
printf "Podaj preferowany kolor [%s]: " $DOMYSLNY
read PREFEROWANY
```

Polecenie `printf` działa podobnie do funkcji biblioteki `stdio` języka C, i podobnie jak ta funkcja nie dopisuje domyślnie końcowego NEWLINE-a.

Zauważ, że poniższe wywołania `printf` są równoważne powyższemu:

```
printf 'Podaj preferowany kolor [%s]: ' $DOMYSLNY
printf "Podaj preferowany kolor [$DOMYSLNY]: "
```

Uwaga: polecenie `printf` ma już własną historię i też może powodować problemy z przenośnością (o tym poniżej). Jednak jego najbardziej podstawowa, przenośna funkcjonalność jest o wiele większa niż polecenia `echo`.

Jeszcze raz echo

Pomimo iż, jak już wiemy, `printf` jest zamiennikiem polecenia `echo`, nie ma powodu nie stosować `echo` do wyświetlania zwykłych komunikatów.

Jest jeden szczególny rodzaj komunikatów — komunikaty o błędach.
W skryptach też zdarza się zakomunikować użytkownikowi wystąpienie błędu. Jednak, gdy skrypt będzie wywołany z przekierowaniem wyjścia, bo normalnie generuje jakieś dane, to komunikat nie pojawi się na wyjściu, i użytkownik go nie zobaczy. Co gorsza, zostanie dopisany do strumienia danych, psując je.

Jest to sytuacja podobna do wcześniejszej, gdy zapytanie do użytkownika zostało przekierowane na `stderr`. Podobnie **komunikaty o błędach dobrze jest kierować na stderr** (który zasadniczo do tego służy):

```
echo Skrypt $0: blad, brak wymaganego pliku $filename 1>&2
```

Zwróćmy uwagę na podanie argumentu 0 (nazwy skryptu) w komunikacie.
W czasie wykonywania skryptu wywołuje się różne programy, a przy użyciu przekierowania (np. potoku) być może również wiele skryptów na raz. Bez tej informacji użytkownik może nie wiedzieć, który skrypt informuje go o błędzie.

Program line

Program `line` czyta jeden wiersz z wejścia `stdin`, i zwraca go w postaci stringa. To znaczy, wyświetla na swoim wyjściu, to co przeczytał na wejściu. Dzięki przekierowaniom, i poleceniom zagnieżdżonym, daje to duże możliwości.

Przykład — realizacja dialogu z użytkownikiem:

```
# za pomocą read
echo Podaj swoj kolor:
read KOLOR
```

```
# za pomoca line
echo Podaj swoj kolor:
KOLOR=`line`
```

Przykład — chcemy otworzyć plik DANE.TXT i przeczytać trzeci wiersz:

```
# bledne rozwiazanie
WIERSZ1=`line < DANE.TXT`
WIERSZ2=`line < DANE.TXT`
WIERSZ3=`line < DANE.TXT`
```

```
# poprawne rozwiazanie
WIERSZ3=`(line >/dev/null; \
          line >/dev/null; \
          line ) < DANE.TXT`
```

Argumenty wywołania skryptu

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

$$\text{nazwaskryptu}_0 \quad \text{arg}_1 \quad \text{arg}_2 \quad \text{arg}_3 \quad \ldots$$

Argumenty wywołania są dostępne wewnątrz skryptu w układzie pozycyjnym:

argument zerowy, nazwa skryptu:	$0
pierwszy argument:	$1
drugi argument:	$2
...	
wektor argumentów bez zerowego, jeden string:	$*
wektor argumentów bez zerowego, oddzielne:	$@

Wartości argumentów są traktowane jako napisy tekstowe, podobnie jak wartości zmiennych.

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:

$$\text{nazwaskryptu}_0 \quad \text{arg}_1 \quad \text{arg}_2 \quad \text{arg}_3$$

po `shift`:

$$\text{nazwaskryptu}_0 \quad \text{arg}_2 \quad \text{arg}_3$$

Można ustawić (nadpisać) cały wektor argumentów (od $1) bieżącego wywołania poleceniem `set` (nowy wektor może być dłuższy lub krótszy):

```
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
```

Inne konstrukcje „dolarowe”

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

numer procesu interpretera poleceń:	`$$`
liczba argumentów, bez argumentu zerowego:	`$#`
wartość domyślna, brana gdy danego argumentu brak:	`${1:-domysl}`

Oraz szereg innych.

Różne interpretery poleceń definiują jeszcze inne, specyficzne konstrukcje dolarowe, dostępne tylko w danym interpreterze. Na uwagę zasługują jednak konstrukcje zdefiniowane przez standard POSIX, o których poniżej.

Przykład: skrypt z argumentami

Argumenty wywołania przydatne są w wielu sytuacjach. Pozwalają np. lepiej zrealizować skrypt odpytania użytkownika o kolor, z poprzednich przykładów. Zakładając, że użytkownik podaje kolor domyślny jako pierwszy argument:

```
DOMYSLNY="$1"
printf "Podaj preferowany kolor [$DOMYSLNY]: " > /dev/tty
read PREFEROWANY                               < /dev/tty
if [ -z "$PREFEROWANY" ]
then echo $DOMYSLNY
else echo $PREFEROWANY
fi
```

Pobranie wartości domyślnej z argumentu zamiast wejścia pozwala łatwo zaimplementować sytuację braku tego argumentu, i użycie wartości wbudowanej:

```
DOMYSLNY=Pistacjowy
if [ $# -gt 0 ]; then DOMYSLNY=$1; fi
printf "Podaj preferowany kolor [$DOMYSLNY]: " > /dev/tty
read PREFEROWANY                               < /dev/tty
...
```

Znaki specjalne: cytowanie stringów

\	odbiera znaczenie specjalne następującego po nim znaku
'...'	sztywny string, brak interpretacji wszelkich znaków specjalnych
"..."	brak interpretacji znaków specjalnych z wyjątkiem $, \, i `...`

Ponadto, znak "\" na końcu wiersza ma znaczenie kontynuacji w następnym wierszu. Znak NEWLINE (\n, ASCII 10), jest dopuszczalny jako zwykły znak wewnątrz napisów cytowanych. Traci też swoje znaczenie specjalne po "\".

Złożone wyrażenia, z wieloma znakami cytowania, które można czasem napotkać, są bardzo trudne do „rozszyfrowania". W praktyce, warto pamiętać następującą zasadę: znaki '...' tracą swoje znaczenie specjalne (stają się zwykłymi znakami), gdy są wewnątrz stringa "...". I na odwrót. Na przykład:

```
echo "$PATH"        # wartosc zmiennej PATH jest obliczana
echo "\$PATH"       # nie jest obliczana
echo "\\$PATH"      # jest obliczana
echo '$PATH'        # nie jest obliczana
echo "'$PATH'"      # jest obliczana
echo '"$PATH"'      # nie jest obliczana
```

Status procesu

Każdy proces generuje kod zakończenia (*exit code*) zwany też statusem zakończenia (*exit status*) lub po prostu statusem. W przypadku skryptu status można zwrócić wbudowanym poleceniem `exit` lub `return`. Wywołanie tego polecenia bez argumentu generuje status równy 0.

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.

Wartość statusu jest normalnie niewidzialna przy interakcyjnym wykonywaniu poleceń. Jest jednak przechwytywana przez interpreter poleceń, i dla poleceń wykonywanych synchronicznie (w pierwszym planie), bezpośrednio po wykonaniu polecenia jest dostępna w zmiennej $?.

W przypadku poleceń złożonych, takich jak: potok, lista, albo skrypt, ich statusem jest status ostatniego wykonanego polecenia.

Wykorzystanie statusu

Wszystkie polecenia generują status, i często sygnalizuje on pewne fakty, zawsze skrupulatnie opisane w podręczniku (man) danego programu/polecenia. Niektóre programy są specjalnie przygotowane do sprawdzania warunków. Za pomocą statusu raportują one jakiś precyzyjnie zdefiniowany warunek, i czasem mają opcję powodującą brak wyświetlania czegokolwiek na wyjściu. Przykładami takich programów są: `grep`, `cmp`, `mail`, i inne. Generowany status można łatwo wykorzystać, za pomocą warunkowych list poleceń `||` i `&&`.

Na przykład, program `ls` generuje niezerowy status, gdy napotka jakiś błąd, zwykle brak pliku o podanej nazwie.

```
ls jakis_plik > /dev/null 2>&1 && echo Jest jakis_plik.
ls jakis_plik > /dev/null 2>&1 || echo Nie ma jakis_plik.
```

Przekierowanie wyjść `stdout` i `stderr` na `/dev/null` ma na celu wyeliminowanie wszelkich komunikatów od programu `ls`, który jest tutaj wykorzystywany tylko jako tester istnienia określonego pliku.

```
(test -e jakis_plik && echo Istnieje.) || echo Nie istnieje.
```

Polecenie warunkowe if

„Etatowym" poleceniem warunkowym systemów uniksowych jest `if` o składni:

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

Zauważmy, że wewnętrzne wywołanie polecenia test można zastąpić wywołaniem dowolnego innego programu lub polecenia wbudowanego. `If` wykonuje je, podobnie jak wykonywane są polecenia zagnieżdżone, i sprawdza jego status.

Zapis polecenia `if` często skraca się pomijając NEWLINE po słowach kluczowych: `then`, `else`, i `fi`. W pozycjach polecenia `if`, gdzie znajdują się polecenia wewnętrzne, można pominąć NEWLINE jedynie pod warunkiem zastąpienia go średnikiem:

```
if [ -r prog.dan ] ; then prog < prog.dan ; else prog ; fi
```

Testowanie warunków programem test

„Etatowym" narzędziem do sprawdzania warunków jest `test`, obsługujący bogaty język specyfikacji warunków.

Przykłady:

```
test -r filespec   # czy plik istnieje i jest dost.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.

Skrócona forma wywołania test

Polecenie test jest niejednoznaczne — jest zarówno poleceniem wbudowanym wielu interpreterów poleceń, jak i programem zewnętrznym na wszystkich systemach uniksowych. Te warianty nieco różnią się od siebie. Zauważmy, że możemy zawsze wymusić wykonanie programu zewnętrznego wywołaniem:

```
/usr/bin/test 1.1 -eq 1 && echo /usr/bin/test zaokragla
```

Istnieje forma skrócona wywołania polecenia test, która zawsze wywołuje jego formę wbudowaną w interpreter poleceń:

```
[ 1.1 -eq 1 ] && echo Wbudowany test zaokragla
```

W przypadku użycia polecenia `if` wywołanie to ma postać

```
if [ 1.1 -eq 1 ] ; then echo Wbudowany test zaokragla; fi
```

Nawias kwadratowy jest w skróconej formie jakby zamiennikiem słowa „test” i **musi po nim wystąpić spacja** aby został poprawnie zinterpretowany.

Problemy ze skróconą formą wywołania test

Skrócona forma wywołania `test` w poleceniu `if` budzi pewne nieporozumienia, bo **sprawia wrażenie, że jest to pewna forma składniowa polecenia if**. Stąd często pisane są błędne wywołania typu:

```
if [ $LOOPS=6 ] ;   then ... fi
if [$LOOPS = 6] ;   then ... fi
if [ $LOOPS = 6 ] ; then ... fi
```

Pierwsza forma jest typowym błędem początkujących, niestety niemożliwym do wykrycia. Polecenie `test` sprawdza wtedy niepustość danego stringa (np. `0=6`) i odpowiada twierdząco, a użytkownik nie widzi gdzie tkwi problem.

Druga forma jest błędna składniowo, ale niestety, **normalnie błąd w skrypcie nie powoduje zatrzymania całego skryptu**. Pojawia się komunikat o błędzie, ale reszta skryptu się wykonuje. Początkujący użytkownicy mają tendencję do ignorowania niezrozumiałych komunikatów, i akceptowania wyniku.

Trzecia forma zasadniczo nie zawiera błędu. Błąd jednak się objawi, jeśli zmienna LOOPS nie będzie miała wartości (lub będzie pustym stringiem). Po interpretacji nie zostanie po niej żaden ślad, i wyrażenie będzie błędne `[ = 6 ]`

Skrócona forma test — wniosek

Analiza błędnych przykładów skróconego wywołania test, i doświadczeń z dużą liczbą błędnych skryptów, prowadzą do następującego zalecenia i wniosku:

> Zalecenie: **stosuj pełną formę test zamiast skróconej!!**
> Zwraca to większą uwagę na to co jest wywoływane, i gdzie szukać błędu.

```
if [ $LOOPS=6 ] ;       then ... fi
if [$LOOPS = 6] ;       then ... fi
if [ $LOOPS = 6 ] ;     then ... fi
if test "$LOOPS" = 6 ; then ... fi
```

Zauważmy, stosując ostatnią formę, kompletnie unikamy błędu z formy drugiej. Łatwiej też spostrzec błąd z formy pierwszej. Pisząc wiele wywołań polecenia `test` łatwiej skojarzyć, że wyrażenie `$LOOPS=6` nie jest podobne do typowych, „rozstrzelonych" wyrażeń polecenia `test`.

Natomiast aby uniknąć błędów z formy trzeciej, warto nabrać nawyku pisania wartości zmiennej w cudzysłowach. Z wyjątkiem rzadkich przypadków, kiedy zależy nam na efekcie „rozpłynięcia" się zmiennej, gdy jej wartość jest pusta, taki sposób zawsze jest poprawny i bezpieczny.

Polecenie warunkowe case

Polecenie case jest innym rodzajem polecenia warunkowego, ale nie sterowanego warunkami logicznymi, tylko wartością wyrażenia:

```
case ‘uname -s‘ in
  "Linux") PATH=$PATH:~/Bin.Linux ;;
  "SunOS") PATH=$PATH:~/Bin.SunOS ;;
        *) echo Unknown system, PATH unchanged. ;;
esac
```

Poza dość specyficzną składnią, to polecenie nie wyróżnia się niczym szczególnym.

Pętla logiczna while

Istnieje polecenie `while` realizujące pętlę sterowaną warunkiem logicznym. Zasada działania jest podobna do polecenia `if`, tylko warunek jest obliczany, i jego status sprawdzany, każdorazowo przed wejściem do pętli.

Przykład — wykonanie pętli n razy, gdzie n jest wartością zmiennej NLOOPS:

```
i=0
while test $i -lt $NLOOPS
do
  echo Tu jakies obliczenia i=$i ...
  i=‘echo $i 1 + p | dc‘
done
```

Do inkrementacji zmiennej i został tu wykorzystany tradycyjnie kalkulator RPN o nazwie `dc`, ponieważ historycznie intrepretery poleceń systemów uniksowych nie mają zdolności obliczeń arytmetycznych. Takie wyrażenia zostały wprowadzone rozszerzeniami standardu POSIX, i będą omówione poniżej.

Pętla wyliczeniowa for

Standardowe interpretery poleceń systemów uniksowych nie posiadają pętli arytmetycznej w stylu `for (i=0;i<N;++i)`. Posiadają natomiast polecenie `for`, które realizuje pętlę wyliczeniową, wykonującą swoją treść kolejno dla wszystkich słów (stringów) podanych w wywołaniu.

Częstym zastosowaniem tego polecenia jest, w połączeniu z mechanizmem *globbing*u, wykonanie pewnych poleceń dla wszystkich zadanych plików, np.:

```
for x in *.c
do
  if test ! -e ${x}~
  then echo Nie istnieje starsza wersja pliku $x

  # teraz wiemy, ze starsza wersja istnieje, porownujemy
  # cmp -s nic nie wyswietla, status informuje o roznicach
  elif cmp -s ${x}~ ${x}
  then echo Istnieje starsza IDENTYCZNA wersja pliku $x
  else echo Istnieje starsza wersja pliku $x
  fi
done
```

Funkcje interpretera poleceń

```
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.

Parametry opcjonalne interpretera poleceń

Interpreter 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.

Magia #!

Większość współczesnych interpreterów poleceń stosuje konwencję polegającą na specjalnym traktowaniu skryptów, których **pierwsze dwa bajty** to #! (fachowa wymowa anglojęzyczna: *sha-bang*). Do wykonania takich skryptów interpreter wywołuje program określony w pierwszym wierszu skryptu, po #!. Dalszy ciąg wiersza traktowany jest jako **wektor argumentów wywołania**.

Zwróćmy jednak uwagę, że ten wektor **nie jest interpretowany**, jak typowy wiersz polecenia, tylko brany dosłownie. Nie ma wyszukiwania według zmiennej PATH, zatem pierwszy argument musi być **ścieżką pliku** (pełną/bezwzględną, lub względną). Można zadać argumenty dla programu, ale nie można stosować żadnych mechanizmów shella: zmiennych, skierowań, zagnieżdżeń, itp.

Te mechanizm pozwala pisać skrypty dla języków interpretowanych, których interpreter jest dostępny w systemie i wywoływalny jako program.

```
#!/usr/bin/perl

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

Mechanizm #! — uwagi

Mechanizm #! bywa często nadużywany. Zauważmy, że zwykłe skrypty interpretera poleceń mogą być wywołane przez nazwę pliku (jeśli plik posiada atrybut „x"), i zostaną zwykle poprawnie wykonane przez standardowy shell uniksowy.

Mechanizm #! **wymusza** wywołanie konkretnego programu, z konkretnego pliku, z konkretnymi argumentami. Jeśli na tym nam zależy, trzeba go użyć.

Natomiast w pozostałych przypadkach, większą przenośność uzyskujemy, pomijając wiersz #!. W ogólności, pisząc skrypt nie mamy pewności czy konkretny interpreter będzie dostępny w danym systemie, oraz jaka dokładnie jest jego ścieżka pliku.

> Bezmyślne wklejanie konstrukcji #! w każdym skrypcie świadczy o niekompetencji programisty.

Uniksowe interpretery poleceń — historia

- Oryginalny interpreter poleceń: Bourne shell (`/bin/sh`)

- Zmodyfikowany interpreter poleceń do pracy interakcyjnej: C-shell (`/bin/csh`) zawiera dodatkowe mechanizmy do pracy interakcyjnej, lecz również zmienioną składnię poleceń złożonych. Powstał paradygmat pisania skryptów Bourne shella, i pracy interakcyjnej w C-shellu.

- Później, w miarę rozwoju systemów uniksowych pojawiło się wiele wersji interpreterów poleceń. Typowo zawierały coraz więcej mechanizmów do pracy interakcyjnej, jak również konstrukcji programistycznych.

  Te programy wpisywały się w grupę kompatybilną z oryginalnym interpreterem Bourne'a, albo w grupę kompatybilną z C-shellem.

  Jednym z najbardziej rozbudowanych w grupie Bourne shella jest bash (Bourne Again Shell) na *opensource* owej licencji Gnu, wprowadzający bardzo dużo rozszerzeń, w tym szereg mechanizmów z grupy C-shella.

- Specyfikacja POSIX interpretera poleceń: wprowadziła standard shella zasadniczo zgodny z Bourne shellem, z szeregiem rozszerzeń.

Uniksowe interpretery poleceń — praktyka

- Współcześnie używane interpretery (tcsh, ksh, zsh, 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 w danej chwili używamy.

- Z punktu widzenia maksymalnej kompatybilności pisanych skryptów, i przenośności na największą liczbę systemów uniksowych, należy brać pod uwagę oryginalny Bourne shell. Nie oznacza to rezygnacji z jakichś ważnych funkcji, a jedynie konieczność pisania pewnych mniej wygodnych konstrukcji.

- Pisząc skrypty pod kątem ich przenośności dla systemów współczesnych, warto brać pod uwagę standard POSIX i używać konstrukcji dobrze zdefiniowanych przez ten standard.

- Mechanizm #! powinien być zarezerwowany dla skryptów napisanych pod kątem konkretnego interpretera (lub wersji), wykorzystującego jego specyficzne konstrukcje. Mechanizm ten nie wynajdzie nam „lepszego" interpretera, gdy np. nie mamy pewności czy standardowy interpreter systemu poprawnie wykona rozszerzone konstrukcje POSIX.

Uwagi na temat basha

- `bash` jest bardzo rozbudowanym uniksowym interpreterem poleceń, zgodnym ze standardem POSIX. Jest produktem typu „open source" na licencji Gnu, i jest z reguły instalowany na systemach linuksowych, oraz na wielu systemach uniksowych, jako interpreter użytkownika (ale nie systemowy).
- W ten sposób `bash` szybko staje się najpopularniejszym [͝] i czasami wręcz jedynym [͡] shellem.
- Jednak `bash` posiada szereg rozszerzeń wykraczających poza standard POSIX. Do tego istnieje niezliczona liczba podręczników typu *„Programowanie w bashu"*, oraz *„Kruczki i sztuczki basha"*, intensywnie eksploatujących te rozszerzenia. Powoduje to tendencję do pisania skryptów typu bash-only, często zupełnie niezamierzenie i niepotrzebnie.
- W rzeczywistości `bash` nie jest jedynym interpreterem, i nie można zakładać ani że jest interpreterem użytkownika, ani systemowym (tzn. wykonującym skrypty administracyjne), ani że w ogóle jest zainstalowany na danym systemie. Skrypty odwołujące się do `basha` (wywołując go jawnie, lub przez mechanizm #!), albo wykorzystujące jego specyficzne konstrukcje, nie będą działać we wszystkich środowiskach uniksowych.

Przykład: nieprzenośne konstrukcje

Jako przykład zagadnienia przenośności, rozważmy polecenie `printf`, w miarę przenośnie pozwalające tworzyć w skryptach napisy niezakończone znakiem NEWLINE. Załóżmy, że chcemy utworzony napis przypisać do zmiennej MSG:

```
MSG=‘printf "Przykladowy komunikat: "‘
```

To polecenie zostanie poprawnie wykonane w każdym interpreterze (grupy Bourne shella), ponieważ korzysta tylko z mechanizmu zagnieżdżenia, i instrukcji przypisania. Wywołane zostanie wbudowane polecenie `printf`, jeśli interpreter takie posiada, lub program zewnętrzny, jeśli tylko istnieje w systemie.

Dla porównania, `bash` ma wbudowane polecenie `printf`, z niestandardową opcją `-v` powodującą od razu przypisanie stringa zmiennej (żaden program nie może obsługiwać takiej formy, z powodów, które zostały wcześniej wyjaśnione):

```
printf -v MSG "Przykladowy komunikat: "
```

Powyższe wywołanie zadziała tylko w bashu. Nie wykonają go poprawnie: `sh`, `ksh`, `dash`, `zsh`, i zapewne wiele innych. Niektóre z nich posiadają wbudowane polecenie `printf`, ale żadne nie obsługuje argumentu `-v`.

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 dodatkowe funkcjonalności:

`${var:-default}`	użyj wartości domyślnej jeśli nie ma wartości lub null
`${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 konstrukcji:

`${zm:+value}`	użyj podanej wartości jeśli zmienna miała już wartość non-null
`${#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: polecenia zagnieżdżone

POSIX wprowadził alternatywną notację dla poleceń zagnieżdżonych:

```
$(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 realizują szereg dodatkowych operacji, które ułatwiają pisanie skryptów. Należą do nich np. operatory arytmetyczne:

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

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 ++. Niektóre interpretery je implementują, ale niestety, powoduje to dwuznaczność interpretacji pewnych wyrażeń:

```
$ 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:]`

Na przykład, dopasowanie plików o nazwach, których część podstawowa kończy się cyfrą, i posiadających tryliterowe rozszerzenia:

```
*[[:digit:]].[[:alpha:]][[:alpha:]][[:alpha:]]
```

Zauważmy, że korzystanie z tych konstrukcji zawsze wymaga pisania podwójnych nawiasów kwadratowych.

POSIX shell: lokalizacja

LANG — domyślna wartość lokalizacji, działa w braku zmiennych LC_*

LC_COLLATE — schemat porządkowania napisów znakowych

LC_CTYPE — schemat typów znakowych

LC_MESSAGES — format i język komunikatów

LC_NUMERIC — zestaw konwencji prezentacji wartości liczbowych

LC_TIME — zestaw konwencji formatowania daty i czasu

LC_MONETARY — format pieniężny

LC_PAPER — rozmiar papieru

LC_NAME — format prezentacji nazw (i nazwisk)

LC_ADDRESS — format prezentacji adresu i lokalizacji

LC_TELEPHONE — format prezentacji numerów telefonicznych

LC_MEASUREMENT — stosowane konwencje miar (np. metryczna)

LC_ALL — wartość przesłaniająca wszystkie pozostałe zmienne LC_*