Úvod do skriptování v shellu

Petr Osička

verze z 13. září 2023

Abstrakt

Text je určen pro samostudium předmětu XVPP1 v zimním semestru 2023. Obsahem je skriptování v unixovém shellu Bash. Předpokládá základní znalosti unixových prostředí a práce se shellem, zhruba v rozsahu předmětu XUNIX. Text je napsán přehledově, strukturou se blíží [1]. Ke studiu důrazně doporučuji i jiné zdroje, například právě [1] nebo [3]. Důležitou součástí studia je psaní vlastních skriptů a exprerimentování s nimi. Za tímto účelem jsou na konci textu vybrané úkoly.

Text není dokončen. Prosím o reportování faktických chyb, nejasných oblastí atd. Na konci dokumentu je seznam změn v jednotlivých verzích.

Obsah

1 Základy

1.1 Vytvoření skriptu a jeho spuštění

Skript je textový soubor, který chceme spustit pomocí nějakého interpretu. Typicky jej lze spustit pomocí

```
> interpret script
```

v našem případě tedy

```
> bash script.sh
```

Alternativou je dát na začátek souboru řádek, jehož první dva symboly jsou `#!` (a tvoří tzv. *shebang*), a zbytek prvního řádku je tvořen příkazem, který daný interpret spustí, typicky přímo cestou k jeho binárce. [1] V našem případě skript používající shebang vypadá například následovně.

```
1 #!/bin/bash
2 #
3 echo "hello sailor!!"
4 echo "second line"
```

Symbol `#` je mimo prvního řádku brán jako začátek komentáře. Skript potom obsahuje dva další řádky, na každém z nich je zapsán jeden příkaz `echo`. Řádky jsou interpretovány postupně seshora, nejdříve je spuštěn příkaz na řádku 3 a poté na řádku 4. Spuštění skriptu si můžeme představit jako spuštění nového shellu a v něm automatické zadávání příkazů z jednotlivých řádků.

Abychom jej mohli spustit nastavíme skriptu příslušné právo, a poté jej spustíme známým způsobem

```
> chmod +x script.sh
> ./script.sh
hello sailor!!
second line
```

Co se stane v případě, že vytvoříme spustitelný skript bez shebangu a spustíme jej, závisí na použitém shellu a operačním systému.

Alternativou k `/bin/bash` v shebangu je nalezení spustitelného souboru shellu pomocí programu `env`. Tento způsob tvoří skript přenositelným na systémy, které `bash` nemají nainstalován v adresáři `/bin`.

```
1 #!/usr/bin/env bash
```

V ukázkách skriptů ve zbytku budeme shebang vynechávát, budeme ale předpokládat, že se v každém skriptu nachází.

[1] Tento mechanismus je poskytován jaderným systémem spouštění programů.

1.2 Operátory řídící běh skriptu

Při provádění skriptu jsou příkazy spouštěny po přečtění znaku nového řádku (nebo konce souboru). Na jeden řádek lze umístit i více příkazů tak, že mezi ně umístíme oddělovač. Použije-li středník, jsou příkazy jsou spuštěny za sebou až v momentě, kdy shell přečtě znak nového řádku.

```
cmd1; cmd2; cmd3;
```

Příkazy lze také oddělovat pomocí `&`. Při vykonávání

```
cmd1 & rest
```

spustí shell `cmd1` na pozadí a přejde ke zpracování `rest`. Operátory `&& a ||` odpovídají líné konjunkci a disjunkci. Při vykonávání

```
cmd1 && rest
```

shell spustí `cmd1` a pokračuje k `rest` pouze pokud `cmd1` skončil úspěšně (viz návratové hodnoty v kapitole 2.1). Naopak při

```
cmd1 || rest
```

shell spustí `cmd1` a pokračuje k `rest` pouze pokud `cmd1` skončí neúspěšně.

Příkazy (jeden i více) lze také seskupovat do skupin pomocí závorek `()`. Příkazy z jedné skupiny pak shell spustí ve své kopii, kterou za tímto účelem vytvoří. Na skupinu se tak shell dívá atomicky (jako na jeden příkaz). Například po

```
> (cmd1; cmd2) & (cmd3; cmd4) &
```

paralelně na pozadí běží dva podshelly, v prvním jsou za sebou vykonány `cmd1` a `cmd2` a ve druhém `cmd3` a `cmd4`.

1.3 Proměnné

Na proměnné se lze dívat tak, že obsahují řetezce, že všechny proměnné existují a mají jako defaultní hodnotu přiřazen prázdný řetězec. Přirozeně ovšem považujeme za existující pouze proměnné, které mají přiřazenu jinou hodnotu.

Proměnnou `X` bash expanduje na její hodnotou pomocí expanzí výrazu `$X` nebo výrazu `${X}`. Například hodnotu proměnné `PATH` zobrazíme

```
> echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:
> echo ${PATH}
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:
```

V textu se budeme držet způsobu se závorkami. Expanze `${}` je pokročilejší mechanismus, kterému se v textu nebudeme plně věnovat, další info lze nalézt v dokumentaci Bash Manuál.

Vypíšeme-li obsah neexistující proměnné, dostaneme prázdný řetězec

```
> echo ${DONOTEXIST}

```

Proměnným přiřazujeme hodnotu pomocí operátoru `=`. Pozor, nesmí okolo něj být mezery.

```
> FOO=uname
> echo ${FOO}
uname
```

Následující spustí utilitku `uname`

```
> ${FOO}
Linux
```

Chceme-li do proměnné přiřadit hodnotu s mezerami, musíme ji napsat do uvozovek (`"`) nebo apostrofů (`'`).

```
> FOO='uname -a'
> echo ${FOO}
uname -a
> ${FOO}
Linux phoenix 4.9.0-19-686-pae #1 SMP Debian 4.9.320-2 (2022-06-30) i686 GNU/Linux
```

V systému obvykle existuje řada systémových proměnných, které lze považovat za součást konfigurace systému a shellu. Například výše uvedená proměnná `PATH` obsahuje dvojtečkami oddělený seznam adresářů, ve kterých se vyhledávájí spustitelné soubory do shellu zadaných programů. Seznam nastavených proměnných lze zobrazit pomocí[2]

```
> set
BASH=/bin/bash
[...]
```

(Výrazem `[...]` značíme, že výpis pokračuje na další řádky)

1.4 Podshell

Výraz

```
1 $(command)
```

vede ke spuštění příkazu `command` a nahrazení výrazu standardní výstupem vzniklým tímto spuštěním. Například

```
> echo date
```

[2] alternativy: `env`, `printenv` zobrazí pouze systémové proměnné

Tabulka 1: Žolíci a jejich expanze.

žolík	expanze
`*`	0 nebo více libovolných znaků
`?`	jeden libovolný znak
`[chars]`	jeden ze znaků v řetezci chars
`[c1-c2]`	jeden ze znaků v rozsahu `c1-c2`, např. `[0-9]`, `[a-z]`
`{word1,word2,...}`	jedno ze slov
`[!chars]`	jakýkoliv znak, který není v chars

```
date
> echo $(date)
Wed Aug 23 09:44:24 CEST 2023
```

Na prvním řádku `echo` vypíše řetězec `date`. Na druhém řádku ovšem shell spustí program `date` a jeho výstup umístí jako argument `echo`, takže potom vlastně spouštíme

```
> echo Wed Aug 23 09:44:24 CEST 2023
Wed Aug 23 09:44:24 CEST 2023
```

Typické použití ve skriptech je přiřazení výsledku do proměnné

```
> TODAY=$(date +%A)
> echo Today is ${TODAY}
Today is Wednesday
```

Technicky shell spouští nový proces shellu, tzv. *podshell*, ve kterém příkaz spustí. V podshell tak neexistují proměnné vytvořené v rodičovském shellu.[3] Alternativní syntax je vložení `command` mezi zpětné apostrofy (`` ` ``). Tuto syntax nedoporučuji a v textu ji používat nebudeme.

1.5 Expanze

Shell před vlastním provedením příkazu provádí tzv. expanze. Expanzi si lze představit jako nahrazení nějakého výrazu jiným výrazem.

Dva příklady expanzí už jsme viděli v kapitolách 1.3 a 1.4. Další hojně používanou expanzí je expanze cest v souborovém systému. Při specifikaci cesty lze použít tzv. žolíky, které shell nahradí a dostane tak skutečnou cestu nebo seznam cest (anglicky *file globbing*). Často používaní žolíci jsou v tabulce 1.

Potlačit expanzi lze umístěním výrazu do uvozovek nebo apostrofů. Apostrofy potlačí všechny expanze, uvozovky potlačí pouze expanzi cest. Shell je v expandování agresivní a expanduje i výrazy, které jsme dostali jinou expanzí. Předpokládejme například, že že v aktuální adresáři existují soubory `a.txt` a `b.txt` a že spustíme následující.

[3] Toto omezení lze různými způsoby obejít, například pomocí exportování proměnných.

```
> FOO='*.txt'
> echo ${FOO}
a.txt b.txt
> echo "${FOO}"
*.txt
```

První `echo` dostane jako argument `a.txt b.txt`, protože shell expanduje dvakrát, nejdříve `${FOO}` na `*.txt` a tento výraz potom na `a.txt` a `b.txt`. Druhé echo má argument `*.txt`, protože shell expanduje `${FOO}` a ve výrazu `"*.txt"` už neexpanduje. Pro třetí echo shell neexpanduje nic.

```
> FOO='*.txt'
> echo ${FOO}
a.txt b.txt
> echo "${FOO}"
*.txt
> echo '${FOO}'
${FOO}
```

Mechanismus expanze je složitější než to, co jsme popsali. Hezký popis lze najít v kapitole 8 knize [3].

1.6 Přesměrování a roura

Bežící proces má k dispozici standardní proudy: standardní vstup (`stdin`), standardní výstup (`stdout`) a standardní chybový výstup (`stderr`). Ze standardního vstupu může číst, do zbylých dvou zapisovat. Proces neví, kam jsou proudy napojeny, to je zařízeno vně procesu.

Ve výchozím nastavení jsou všechny tři proudy připojeny ke konzoli. Pokud spustíme program, který čte ze standardního vstupu, musíme něco zadat z klávesnice, výstup (i chybový) programu se tiskne do konzole.

Všechny tři proudy můžeme přesměrovat, a to buď do/ze souboru a nebo do/z jiného procesu. Výstup jednoho procesu tak může stát vstupem druhého procesu.

```
1  # stdout je presmerovan do souboru file, pokud existuje, je prepsan
2  command > file
3
4  # stdout je presmerovan do souboru file, pokud existuje, je stdout pripojen na konec
5  command >> file
6
7  # stdin je presmerovan ze souboru file
8  command < file
9
10 # stdin je cten z konzole, ale konci po end
11 command << end
12
13 # presmerovani stdout, stderr do souboru file (prepsani, pripojeni)
```

```
command &> file
command &>> file

# presmerovani stderr do souboru file (prepsani, pripojeni)
command 2> file
command 2>> file

# presmerovani stdout command1 na stdin command2
command1 | command2

# presmerovani stdout a stderr do roury
command1 |& command2
```

Ve skriptech se presměrování `stdin` pomocí `<<` chová jinak než v konzole. Vstup se nečte z klávesnice, ale z řádků samotného skriptu, které se vyskytují za přesměrováním. Tedy například skript

```
cat << EOF
prvni radek
druhy radek
treti radek
EOF
```

vytiskne po spustění

```
prvni radek
druhy radek
treti radek
```

1.7 Aritmetika

Celočíselnou aritmetiku lze provádět pomocí prikazu `expr`, pro který má bash alternativní syntax.

```
$[term]
```

Výraz je expandován vyhodnocením `term` pomocí příkazu `expr`, přičemž uvnitř `term` platí obvyklá pravidla pro expanzi, takže lze používat proměnné.

```
var1=10
var2=20
var3=30
var4=$[${var1} * (${var3} - ${var2})]
var5=$[${var3} / ${var2}]

echo "hodnota var4 je ${var4}"
```

```
8 echo "hodnota var5 je ${var5}"
```

po spuštení skript vypíše

```
hodnota var4 je 100
hodnota var5 je 1
```

Na hodnotě `var5` vidíme, že aritmetika je skutečně celočíselná.

Pro výpočty s celými čísly musíme použít program `bc`. Tato utilita umí ve skutečnosti zpracovat i jednoduché programy s proměnnými, cykly apod. Pro naše účely postačí pouze základní použití.

Typicky se pro výpočet používá podshell a výsledek se uloží do proměnné. Vzor takového použití je

```
1 var=$(echo "scale=x; term" | bc)
```

kde za `x` musíme dosadit přesnost, tj. na kolik desetinných míst chceme, aby `bc` počítalo. V podshellu tedy pomocí roury pošleme `bc` na vstup výraz, jehož výsledek chceme spočítat, `bc` tento výsledek tiskne na standardní výstup, a tento se tak dostane do proměnné `var`. Výhodou je, že ve výrazu `"term"` dochází k expanzi proměnných.

```
1 var2=20
2 var3=30
3
4 var6=$(echo "scale=4; ${var3} / ${var2}" | bc)
5 echo "vysledek je ${var6}"
```

po spuštení skript vypíše

```
vysledek je 1.5000
```

Pro delší výrazy je lepší využít chování přesměrování << ve skriptu, například

```
1  var1=12
2  var2=8
3  var3=30
4
5  var4=$(bc << EOF
6  scale = 4
7  a = ${var1} + ${var2}
8  ${var3} / a
9  EOF
10 )
11 echo "vysledek je ${var4}"
```

Po spuštění skript vytiskne

```
vysledek je 1.5000
```

Proměnná a ve skriptu je proměnná utility `bc`. Je vidět, že díky expanzi shellových proměnných můžem volně použít i proměnné shellu, po expanzi dostane `bc` na vstup

```
scale = 4
a = 12 + 8
30 / a
```

Další alternativou pro práci s aritmetikou je interní příkaz `let`[4].

[4] Dokumentace: `help let`

2 Řízení běhu

2.1 Návratové hodnoty programů a příkazů

Příkazy a program po svém skončení vrací shellu návratovou hodnotu (anglicky *exit code*). Pokud program proběhl v pořádku, je návratová hodnota 0, jinak je nenulová (pro určité typy chyb jsou ustálené návratové hodnoty). Návratová hodnota naposled provedeného příkazu je uložena v proměnné `?`.

```
> echo 10
10
> echo ${?}
0
> less aiodfjaoidfja.txt
aiodfjaoidfja.txt: No such file or directory
> echo ${?}
1
```

Jak vracet návratovou hodnotu ze skriptu se dozvíme v kapitole 3

2.2 Větvení

Bash podporuje následující formy příkazu `if`.

```
# bez else
if tst
then
   commands-true
fi

# s else vetvi
if tst
then
   commands-true
else
   commands-false
fi
```

Pokud je na místě `tst` proměnná, je podmínka pravdivá, pokud je proměnná neprázdná. Je-li `tst` příkaz, je spušten. Pokud je jeho návratová hodnota 0, provede se blok `commands-true`. V opačném případě se ve verzi s `else` větví provede `commands-false`. Existuje syntax i pro *žebříkovou* verzi konstrukce, kde je `else if` nahrazeno `elif`

```
# zebrik
if tst
then
   commands
```

```
elif tst
then
   commands
else
   commands
fi
```

Příkladem utility, kterou lze použít na místě `tst` je `grep`. Pokud totiž nalezne aspoň jednu shodu, její návratová hodnota je 0, jinak je nenulová. Následující skript otestuje, jestli je v systému uživatel s vybraným jménem.

```
ouruser='franta'

if grep -q "^${ouruser}:" /etc/passwd
then
    echo "uzivatel ${ouruser} existuje"
else
    echo "uzivatel ${ouruser} neexistuje"
fi
```

Pro ověřování obecnějších podmínek lze použít utilitu `test`, která se používá ve formě[5]

```
[ expression ]
```

Za otevírací závorkou a před uzavírací závorkou musí být mezery. Pokud `test` vyhodnotí `expression` jako pravdivé, skončí s návratovou hodnotou 0, jinak s hodnotou nenulovou. Na místo `expression` lze napsat celou řadu podmínek, užitečná je tady manuálnová stránka programu `test`. Existují podmínky pro celá čísla, řetězce, testy existence, práv a jiných vlastností souborů. Ve výrazu `tst` v konstrukci `if` lze použít logických operátorů pro spojky *a* (`&&`) a *nebo* (`||`). V následujícím kódu jsou příklady použití několika testů.

```
#
# numericke porovnavani
#

# tady si je pouzito && k vytvoreni konjunkce
# n1 -ge n2 je test, jestli n1 >= n2
if [ ${a} -ge ${b} ] && [ ${a} -ge ${c} ]
then
    echo "nejvetsi je a s hodnotou ${a}"
elif [ ${b} -ge ${c} ]
then
    echo "nejvetsi je b s hodnotou ${b}"
else
    echo "nejvetsi je c s hodnotou ${c}"
```

[5] Existuje program se jménem `[`, který se chová jako test.

```
fi

#
# prace s retezci
#
x=cecil
y=josef
z=""

# delky retezcu
if [ -n ${x} ]
then
    echo "x je neprazdny"
fi

if [ -z ${z} ]
then
    echo "z je prazdny"
fi

# rovnost
if [ ${x} != ${y} ]
then
    echo "x a y jsou ruzne"
fi

if [ ${x} = ${y} ]
then
    echo "x a y jsou stejne"
fi

# lexikograficke porovnani
# operator porovnani < je nutne backquotovat, jinak je bran jako presmerovani
if [ ${x} \< ${y} ]
then
    echo "x je mensi nez y"
fi

#
# prace se soubory
#

# vypise obsah adresare, pokud existuje
# dalsi testy existence a vlastnosti souboru a adresaru jsou analogicke
addr=logs
```

```
60 if [ -d  ${addr} ]
61 then
62     ls ${addr}
63 fi
64
65 # porovnani stari souboru
66 if [ ${file1} -ot ${file2} ]
67 then
68     echo "soubor ${file1} je starsi nez ${file2}"
69 fi
```

Bash obsahuje následující rozšíření, která nemusí být podporování v jiných shellech.

- *dvojí obyčejné závorky* na místě `tst` (tj. výraz `(( expression ))`): lze použít některé aritmetické, bitové a logické operátory, které `test` neumí
- *dvojí hranaté závorky* na místě `tst` (tj. výraz `[[ expression ]]`): pro řetězcové porovnávání, obsahuje to co `test`, navíc lze kontrolovat shodu s regulárním výrazem

---

Analogií konstrukce `switch` z jazyka C je v Bashi `case`. Umožňuje porovnat hodnotu s jinými hodnotami (jako řetězce) a v případě shody provést blok příkazů. Syntax příkazu je následující.

```
1  case val in                 # val je hodnota, kterou porovnavame
2    pattern1)                 # hodnoty, se kterymi porovnavame jsou ukonceny )
3        cmd1                  # nasleduje blok prikazu
4        [...]
5        cmdk  ;;              # ktery je ukoncen dvema stredniky
6    pattern2 | pattern3)      # lze uvest vice hodnot oddelenych |, staci shoda s jednou
7        cmds ;;
8    *)                        # * se shoduje se vsim, je to analogie default z C
9        cmds ;;
10 esac
```

Je důležité si uvědomit, že na `val` i `pattern` je aplikována expanze.

Konstrukce se používá na místech, kde bychom jinak použili dlouho `if` konstkci s mnoha `elif` větvemi, jak v následujícím kódu.

```
1 month=June
2
3 case ${month} in
4     January | March | May | July | August | October | December)
5         echo "${month} has 31 days" ;;
6     February)
7         echo "${month} has 28 or 29 days"  ;;
8     April | June | September | November)
```

```
        echo "${month} has 30 days" ;;
    *)
        echo "${month} is not a valid month"
esac
```

Typické použití ve skriptech je při zpracování argumentů, jak uvidíme v kapitole 3.2.

Je důležité si uvědomit, že

2.3 Cykly

```
for var in list
do
   commands
done
```

V každé iteraci je do proměnné `var` přiřazen jeden prvek ze seznamu `list` a poté je provedeno tělo `commands`. List je seznam oddělený mezerami. Ten můžeme vytvořit přímo, nebo jej získáme jinak, například pomocí expanze jmen.

```
# list sestaveny primo
for v in monday tuesday wednesday
do
    echo ${v}
done

# list ziskany expanzi
# pro kazdou (viditelnou) polozku v adresari addr zavolame utilitku file
addr=logs
for f in ${addr}/*
do
    file ${f}
done
```

Při tvorbě seznamu a jeho zpracování je nutné být opatrný, pokud některé položky obsahují mezery. Takové položky lze v ručně vytvořeném seznamu vyznačit pomocí uvozovek nebo apostrofů. Pokud je seznam vytvořen strojově, tuto možnost nemáme. Dělení řetezce na seznam položek se řídí obsahem proměnné `IFS`. Její hodnotou je řetězec znaků, které chápeme jako oddělovače položek. Výchozí oddělovače jsou mezera, nový řádek a tabulátor. V následujícím příkladu změníme tuto proměnnou tak, abychom mohli přistoupit k jednotlivým adresářům v proměnné `PATH`.

```
IFS=:
for dir in ${PATH}
do
    echo -n ${dir}
    echo ": $(ls ${dir} | wc -l) polozek"
```

```
done
```

Při nastavování `IFS` můžeme narazit na problém, jak tam přidat některý z výchozích znaků. K tomu lze použít znakovou expanzi shellu

```
$'znak'
```

Výraz `$'\n'` tak například expanduje na znak nového řádku.

Seznam položek lze produkovat i pomocí příkazu. Ma místě `list` v cyklu můžeme dát spuštění nějakého programu v podshellu a v cyklu zpracovat jeho standardní výstup. V následujícím příkladu zpracujeme obsah souboru `/etc/passwd`.

```
# zapamatujeme si puvodni IFS
OLD_IFS=${IFS}

# nova hodnota IFS, soubor prochazime po radcich
IFS=$'\n'

for entry in $(cat /etc/passwd)
do
    # tady nastavime IFS nastavime na dvojtecku
    # nemusime mit strach, ze pokazime IFS pro vnejsi cyklus
    # tam se pamatuje puvodni nastaveni na novy radek
    # tj. vyznam ma pouze hodnota IFS na zacatku cyklu
    IFS=:
    for value in ${entry}
    do
        echo "    ${value}"
    done
done

# tady IFS obsahuje dvojtecku, obnovime puvodni hodnoty
IFS=${OLD_IFS}
```

Dalším typ cyklu připomíná jazyk C, nebudeme tedy blíže komentovat, čtenáři jistě stačí následující příklad

```
for (( i=1; i <= 10; i++ ))
do
   echo ${i}
done
```

Cyklus `while` funguje také podobně jako v jazyku C. Dokud platí podmínka, opakuje se tělo cyklu. Pro podmínky platí podobné zásady jako pro podmínky konstrukce `if`. U cyklu navíc můžeme místo podmínky napsat posloupnost řádků, z nichž pouze poslední je interpretován jako podmínka. Zbylé řádky se spustí vždy na začátku iterace, před vyhodnocením podmínky.

```
# cisla 10...0 se vypisi pred podminkou a uvnitr cyklu,
# cislo -1 se vypise jenom pred podminkou
var=10
while echo "hodnota pred podminkou: ${var}"
      [ ${var} -ge 0 ]
do
    echo "hodnota uvnitr cyklu: ${var}"
    var=$[${var} - 1]
done
```

Konstrukce `until` je analogická `while` s tím, že cyklus končí (nikoliv pokračuje) v momentě, kdy je podmínka pravdivá.

Příkazem `break` lze okamžitě ukončit vykonávání cyklu, v jehož těle se příkaz nachází. Příkazu `break` lze volitelně předat numerický argument (přirozené číslo), a ten potom přeruší některý z vnějších cyklů v kódu obsahujícím vnořené cykly. Argument 1 označuje aktuální cyklus, argument 2 cyklus obsahují aktuální cyklus atd.

```
for v1 in prvni druha treti
do
    for v2 in 10 20 30 40 50
    do
        for v3 in x y z
        do
            break     # prerusi cyklus s promennou v3
            break 1   # prerusi cyklus s promennou v3
            break 2   # prerusi cyklus s promennou v2
            break 3   # prerusi cyklus s promennou v1
        done
    done
done
```

Příkazem `continue` lze přerušit vykonávání aktuální iterace cyklu, jako bychom ve skriptu skočili na konec těla cyklu. Podobně jako `break` lze `continue` předat numerický argument, určující kterého cyklu se příkaz týká.

Standardní výstup cyklu (tedy vše, co se během cyklu vytiskne) lze přesměrovat do souboru nebo rourou. Přesměrování se píše za klíčové slovo `done` na konci těla cyklu.

```
# vypise adresare setrizene podle poctu polozek
IFS=:
for dir in ${PATH}
do
    echo -n ${dir}
    echo " $(ls ${dir} | wc -l)"
done | sort -r -k 2  # tady presmerujeme rourou do prikazu sort
```

3 Interakce s okolím

3.1 Návratová hodnota

Pokud neukončíme předčasně je jeho návratovou hodnotou návratová hodnota posledního provedeného příkazu. Předčasně můžeme skript ukončit pomocí příkazu

```
exit value
```

Za `value` dosadíme číslo v rozsahu 0 až 255 (0 označuje úspěšné ukončení, ostatní chybu). Pokud je `value` větší než 255, vrací se zbytek pod dělení `value` číslem 256.

3.2 Vstupní argumenty

Při spouštění skriptu mu můžeme předat parametry a volby (dohromady je označujeme argumenty) stejně jako libovolnému jinému příkazu.

```
> ./script.sh parameter1 parametr2 [...]
```

Uvnitř scriptu k nim můžeme přistupovat. K jednotlivým argumentům lze přistoupit pomocí `${1}`, `${2}`, ... Hodnota `${0}` je jméno skriptu. Počet argumentů je dostupný pomocí `${#}`,[6] přičemž jméno skriptu se do počtu nepočítá. Výraz `${i}` (i zde je číslo) je prázdný řetězec, pokud je počet argumentů menší než `i`.

Výrazy `$*` a `$@` obsahují všechny argumenty. Liší se pouze jsou-li uvedeny v uvozovkách: výraz `$*` obsahuje všechny argumenty jako jedno slovo tak, jak byli zapsány při spuštění skriptu, kdežto `$@` sice také obsahuje všechny argumenty v jednom řetězci, ale každý z nich bere jako samostatné slovo a lze je tak procházet pomocí `for`. Obě poslední hodnoty se typicky používají v uvozovkách, abychom zabránili expanzi jejich obsahu, čímž by došlo ke rozdělení argumentů s mezerami na samostatná slova. (Bez uvozovek oba cykly v následujícím příkladu vypíší argumenty po podslovech oddělených mezerami.)

```
1  echo "skript ${0}: pocet predanych argumentu ${#}"
2
3  # pomoci if muzeme kontrolovat potrebny pocet argumentu, rekneme, ze chceme aspon 3
4  if [ ${#} -le 2 ]
5  then
6      echo "chyba, chceme aspon 3 argumenty"
7      exit 1
8  fi
9
10 echo 'vypis argumentu pomoci $*'
11 for param in "$*"
12 do
13     echo  ${param}
```

[6] Symbol # tady **není** začátkem komentáře, i když obarvení kódu v naši příkladech ukazuje něco jiného. Je to chyba v algoritmu obarvování kódu.

```
done

echo 'vypis argumentu pomoci $@'
for param in "$@"
do
    echo ${param}
done
```

Pro procházení argumentů lze použít příkaz `shift`. Tento příkaz odstraní první argument, a ostatním argumentům zmenší pořadové číslo o 1 (tj. druhý argument se stane prvním, třetí druhým atd.). Počet argumentů se také zmenší o 1. Argument `${0}` zůstane stejný.

```
# vypiseme vsechny argumenty a jejich poradi

count=1
while [ -n "${1}" ] #proc uvozovky? ${1} chapeme jako jeden retezec, i kdyz obsahuje mezery
do
    echo "parametr ${count} je ${1}"
    count=$[${count} + 1]
    shift
done
```

Argumenty můžeme posunout o více míst, příkazu `shift` lze předat numerický argument, který určuje, o kolik míst chceme argumenty posunout.

Argumenty skriptu rozdělujeme na volby a parametry. Volby typicky začínají pomlčkou, pak následuje jeden znak a za ním případně (mezerou oddělená hodnota). Parametry jsou zbytek argumentů. Ve volání

```
> grep -A 3 -E 'printf' soubor1.c soubor2.c
```

Máme volby `-A 3` a `-E`, první volba má hodnotu 3. Zbytek argumentů (`'printf' soubor1.c soubor2.c`) jsou parametry. Konvence při zpracování argumentů je oddělení voleb od parametrů pomocí `--` (dvou pomlček). Argumenty z příkladu výše bychom tedy napsali.

```
-A 3 -E -- 'printf' soubor1.c soubor2.c
```

Ve skriptu pak můžeme volby výhodně zpracovat pomocí `shift` a konstrukce `case`.

```
#predpokladame, ze volby a parametry jsou oddeleny pomoci --

# zpracujeme volby
while [ -n "$1" ]
```

```
do
    case "$1" in
        -a) echo "volba -a" ;;
        -b) echo "volba -b, s hodnotou ${2}"
            shift ;;  # potrebujeme se posunout az za hodnotu
        -c) echo "volba -c" ;;
        --) shift # nasleduje break, musime se posunout za --
            break ;;
        *) echo "nepodporovana  volba ${1}" ;;
    esac
    shift
done

# zpracujeme parametry
count=1
for param in "$@"
do
    echo "parametr ${count} je ${param}"
    count=$[${count} + 1]
    shift
done
```

Ukázka spuštění skriptu

```
> ./script.sh -b 3 -c -d -a -- prvni druhy
volba -b, s hodnotou 3
volba -c
nepodporovana volba -d
volba -a
parametr 1 je prvni
parametr 2 je druhy
```

Pro automatické předzpracování argumentů do požadovaného formátu (rozdělení voleb a parametrů pomocí `--` a rozdělení zhluknutých voleb, kdy pomocí `-ab` myslíme `-a -b` na jednotlivé volby) lze provést pomocí příkazu `getop`. Syntax

```
getop opstring arguments
```

`opstring` se řetězec sestávající se z písmen voleb, pokud má volba hodnotu, píšeme za její písmeno dvojtečku. Program vypíše argumenty ve správném tvaru na standardní výstup.

```
> getop ab:c -ac -b 3 -d prvni druhy
getopt: invalid option -- 'd'
-a -c -b 3 -- prvni druhy
```

Volby, které nejsou specifikované v `opstring` utilita vynechá, ale upozorní na ně chybovou hláškou. Tu lze potlačit s volbou `-q` nebo zapsáním dvojtečky jako prvního symbolu

`optstring`. Pro použití ve skriptu musíme umět nahradit aktuální argumenty pomocí argumentů získaných utilitou `getop`. To se provede řádkem[7]

```
set -- $(getop :ab:c "$@")
```

Poslední možností zpracování voleb příkazem `getops`, který je vylepšením `getop`. Tento příkaz při každém zavolání najde jednu volbu a přiřadí ji do zvolené proměnné (a případnou hodnotu do proměnné `OPTARG`). Pokud volbu úspěšně najde, vrací 0, jinak vrací nenulovou hodnotu. Hodí se tak pro využití přímo v podmínce cyklu `while`. Při úspěšném nalezení volby inkrementuje `getops` proměnnou `OPTIND` o 1. Tato proměnná má výchozí hodnotu 1, po skončení cyklu je tedy obsahuje počet načtených voleb zvětšený o 1. Víme tedy, o kolik se v argumentech musíme posunout, abychom se dostali k parametrům. Při nalezení volby nespecifikované v `opstring` nastaví `getops` proměnnou s načtenou volbou na otazník (`?`).

```
while getopts :ab:c opt
   # opt je promenna, do ktere prirazujeme volby
   # dvojtecka na zacatku optstring potlacuje chybove hlasky
do
    case "$opt" in
        a) echo "volba a" ;;
        b) echo "volba b s hodnotou ${OPTARG}" ;;
        c) echo "volba c" ;;
        *) echo "neznama volba ${opt}" ;;
    esac
done

shift $[${OPTIND} - 1]

count=1
for param in "$@"
do
    echo "parametr ${count} je ${param}"
    count=$[${count} + 1]
    shift
done
```

Příklad spuštění skriptu

```
> ./script.sh -b 3 -c -d -a -- param1 param2
volba b s hodnotou 3
volba c
neznama volba ?
volba a
parametr 1 je param1
parametr 2 je param2
```

[7] `set` vysvětlíme později.

3.3 Vstup pomocí `read`

Příkaz `read` je určen ke čtení standardního vstupu (nebo jiného *file descritoru*, viz příští kapitola). Je-li použit bez dalších argumentů, přečte řádek a uloží jej do proměnné `REPLY` a skončí úspěšně (vrátí 0). Pokud se řádek přečíst nepodaří, skončí neúspěšně (vrátí nenulovou hodnotu). Příkaz akceptuje několik zajímavých argumentů, které lze najít v dokumentaci[8], některé z nich si ukážeme v následujícím příkladu

```
1  # zakladni pouziti
2  # neuspech lze otestovat pomoci Ctrl-d
3  if read
4  then
5     echo "vlozeno: '${REPLY}'"
6  else
7     echo "neuspesny read"
8  fi
9
10 # -p vytiskne prompt
11 if read -p "Vlozte vstup: "
12 then
13    echo "Vlozeno: '${REPLY}'"
14 else
15    echo "Neuspesny read"
16 fi
17
18 # parametry jsou promenne, do ktery se ulozi jednotliva vlozena slova, v
19 # ychozi oddelovac je mezera, lze jej nastavit prepinacem -d
20 # promenna REPLY je prazdna
21 if read -p "Vlozte vstup: " var1 var2
22 then
23     echo "REPLY: ${REPLY}"
24     echo "var1: ${var1}"
25     echo "var2: ${var2}"
26 else
27     echo "neuspesny read"
28 fi
29
30 # prepinac -s zabrani tisteni vstupu pri zadavani (jako u hesel)
31 if read -s -p "Vlozte vstup: " var1 var2
32 then
33     echo ""
34     echo "var1: ${var1}"
35     echo "var2: ${var2}"
36 else
```

[8] `help read`

```
    echo "neuspesny read"
fi

# prepinac '-n nchar' ukonci cteni po nchar znacich (nebo klasickym zpusobem)
if read -n 1 -p "Chcete pokracovat [Y/N]? " var1
then
    case ${var1} in
        Y | y) echo
               echo "pokracujeme" ;;
        N | n) echo
               echo "nepokracujeme" ;;
        *)  echo
            echo "${var1} je nespravne zadani" ;;
    esac
else
    echo "neuspesny read"
fi

# pomoci '-t secs' muzeme dat uzivateli casovy limit na odpoved
if read -t 5 -p "Vstup: " var1
then
    echo ${var1}
else
    echo
    echo "neuspesny read nebo to uzivatel nestihl"
fi
```

V případě, že po uživateli skriptu potřebujeme nějaké rozhodnutí, je příjemnou alternativou příkazu `read` příkaz `select`. Bližší informace lze nalézt v [3] (kapitola 10), případně v manuálu . Bash Manuál.

Příkazem `read` můžeme číst i ze souboru (po řádcích v cyklu) tak, že soubor přesměrujeme na standardní vstup cyklu. Čtení není úspěšné, přečte-li `read` konec souboru.

```
# presmerovani pomoci <
count=1
while read line
do
    echo "radek ${count}: ${line}"
    count=$[${count} + 1]
done < soubor.txt

# alternativne lze rourou do while presmerovat vystup cat
count=1
cat soubor.txt | while read line
do
    echo "radek ${count}: ${line}"
```

```
14      count=$[${count} + 1]
15  done
```

3.4 File descriptory a přesměrování

File descriptor (FD) je přirozené číslo unikátně idenfikující otevřený soubor procesu (v případě skriptu je to proces shellu script provádějící).[9] Deskriptory 0, 1 a 2 jsou rezervovány pro standardní vstup, standardní výstup a standardní chybový výstup. Ve skriptu máme ještě přístup k FD s čísly 3 až 8. Abychom mohli FD použít, musíme jej přesměrovat do/ze souboru či jiného FD. Eventuálně ovšem posloupnost přesměrování musí končit v souboru nebo některém standardním FD.

Při přesměrování z/do FD jej identifikujeme jeho číslem, před nějž vložíme znak `&`. Tedy například

```
1  echo "go to the fd" >&3      # presmerujeme vystup do FD 3
2  grep 'pattern' <&4           # grep cte z FD 4
```

Přesměrování FD, které platí po zbytek skriptu provedeme pomocí příkazu `exec`. Při přesměrování do/z souboru je argumentem `exec` výraz, který je analogický běžnému přesměrování, jenom na místě příkazu je FD. Například

```
1  exec 3>test.txt
```

Lze přesměrovat i FD do jiného FD, například

```
1  exec 3>&1     # presmeruje FD 3 na standardni vystup
```

Pokud chceme FD zavřít, přesměrujeme jej do `&-`, například

```
1  exec 3>&-   # zavreme FD 3.
```

Další příklad ukazuje, jak dočasně pro část skriptu přesměrovat `stdout/stdin` do/z souboru, a potom jej zpět obnovit. (Pomáhá dívat se na FD jako na proměnné, kterým přesměrováním přiřazujeme vstupní nebo výstupní místo).

```
1  #
2  # presmerovani stdout
3  #
4  exec 3>&1            # v deskriptoru 3 si "pamatujeme" stdout
5  exec 1>foo.txt       # stdout presmerujeme do souboru
6
7  echo "toto skonci v souboru foo.txt"
8
9  exec 1>&3            # tady "obnovime" puvodni stdout
10
```

[9] Seznam otevřených souborů procesu lze zobrazit pomoci `lsof`.

```
11 echo "toto se objevi v konzoli"
12 #
13 # presmerovani stdin
14 #
15 exec 6<&0             #  uchovame si stdin
16 exec 0<bar.txt        #  presmerujeme do stdin soubor
17
18 # cyklus ted cte ze souboru
19 count=1
20 while read line
21 do
22     echo "radek ${count}: ${line}"
23     count=$[${count} + 1]
24 done
25
26 exec 0<&6   # obnovime stdin z FD 6
27
28 # otestujeme puvodni hodnotu stdin
29 read -p "vstup: " var
30 echo "zadal jsi: ${var}"
```

3.5 Signály

Procesy v běžícím systému mezi sebou mohou komunikovat pomocí signálů. Těch je definováno více než 30 a mají různý význam. Některé už známe: klávesová zkratka `ctrl-c` pošle aktivnímu procesu signál `SIGINT`, zkratka `ctrl-z` signál `SIGSTOP`. Signály obvykle generuje systém nebo běžící procesy. Vetšinu signálů, které process přijme, může tzv. odchytit a nějak na ně reagovat, případně je ignorovat[10] Mezi signály, které nejde zachytit nebo ignorovat patří `SIGSTOP` a `SIGKILL` (sloužící k pozastavení a bezpodmínečnému ukončení procesu). Dokumentaci k signálům a jejich významu lze nalézt na manuálové stránce `signal(7)`[11].

K zachycení signálu ve skriptu slouží příkaz `trap`. Pomocí

```
1 trap commands signals
```

se po přijetí některého ze signálů ze `signals` provedou příkazy `commads`. Část `commands` se typicky píše v uvozovkách, více příkazů za sebou oddělených středníckem, v případě nutnosti více řádků lze použít zpětná lomítka na koncích řádku (podobně jako víceřádková makra v C). V případě delší části commands je ovšem lepší si vytvořit pro reakci na signál funkci (kapitola 4.1) V následujícím příkladu je skript, který odchytává `SIGINT` a po zmáčknutí `ctrl-c` neskončí.

```
1 trap "echo ' I have trapped Ctrl-C'" SIGINT
2
```

[10] Každý signál má definovanou výchozí reakci, například SIGINT vede k terminaci procesu.
[11] man 7 signal

```
count=1
while [ ${count} -le 90 ]
do
    echo "loop: ${count}"
    sleep 1
    count=$[${count} + 1]
done
```

Mimo signálů můžeme pomocí `trap` odchytávat i ukončení skriptu, stačí na místě signálu předat `EXIT`. Typické využití je pro úklid po práci skriptu, například smazání dočasných souborů atd.

Past na signál lze modifikovat opakovým použitím `trap`, kterým se nastaví na signál nová reakce. Pokud místo `commands` uvedeme `--`, odchytání signálu je zrušeno (a skript se vrací k výchozí reakci na signál). Zrušit odchytávání signálu z předchozího příkladu tak můžeme pomocí

```
trap -- SIGINT
```

4 Pokročilejší témata

4.1 Funkce

Funkce je blok kódu ve skriptu, který má přiřazeno jméno a lze jej opakovaně použít.

```
function name {
    commands
}
```

Funkci používáme tak, jakobychom používali příkaz se stejným jménem jako funkce. Funkce musí být před použitím definována, jinak dojde k chybě. Jména funkcí musí být ve skriptu unikátní, dvojí definování funkce stejného jména vede k zapomenutí první definice (bez chybového hlášení). Pokud vytvoříme funkci se stejným jménem jako jiný příkaz, bude tento příkaz ve zbytku skriptu nepřístupný, místo něj se bude volat funkce.

```
# priklad volani funkce, skript vypise dvakrat 'Toto je funkce foo'
function foo {
   echo "Toto je funkce foo"
}

foo
foo
```

Na funkci se v mnoha ohledech dá dívat jako na miniscript:
1. Funkce jako jiný příkaz vrací návratovou hodnotu. Je to buď návratová hodnota posledního příkazu provedeného v těle funkce, nebo hodnota, která je vrácena příkazem `return`. Pro návratové hodnoty funkcí platí stejná pravidla, jako pro návratové hodnoty skriptu.

2. Můžeme přesměrovat standardní výstup z funkce do proměnné pomocí syntaxe podobné subshellu.

```
function foo {
    echo "testovaci vystup"
}

var=$(foo)
echo ${var}   # toto vypise 'testovaci vystup'
```

3. Funkci můžeme předat argumenty a zpracovat je uvnitř jako ve skriptu. (Jméno funkce je uloženo v `$0`, další argumenty jsou `$1` …, atd.)

```
function foo {
   count=1
   while [ -n "${1}" ]
   do
       echo "parametr ${count} je ${1}"
```

```
        count=$[${count} + 1]
        shift
    done
}

echo "prvni volani"
foo -a -b param1 param2
echo "druhe volani"
foo 120 130
```

4. Proměnné ve skriptu jsou globální. Pokud bychom v předchozím kódu nahradili za poslední 4 řádky následující

```
echo "prvni volani"
foo -a -b param1 param2
echo "hodnota count je ${count}"
echo "druhe volani"
foo 120 130
echo "hodnota count je ${count}"
```

vypíše skript nejdříve jako hodnotu `count` číslo 5 a posléze číslo 3. Pracovat uvnitř funkcí s globálními proměnnými není příliš vhodné (z podobných důvodů, jako musíme být s globálními proměnnými opatrní v běžném programování). Lokální proměnné vytváříme pomocí slovíčka `local`, které použijeme při vytvoření proměnné. Ta je tak platná pouze ve funkci, překryje případnou globální proměnnou stejného jména.

```
temp=1

function foo {
    local temp=10             # uvnitr funkce je promenna temp lokalni
    temp=$[${temp} + 2]
}

foo
echo "hodnota temp je ${temp}"  # tady se vypise hodnota 1
```

5. Do funkce lze přesměrovat standardní vstup. Například

```
function foo {
    local line
    while read line
    do
  echo ${line}
    done
}
```

```
8
9 foo < blah.txt
```

vypíše obsah soubor `blah.txt`.

4.2 Pole

Bash podporuje numericky indexovana pole, indexuje se od 1. Nize vidime syntax vytvoreni pole a pristupu k jednotlivym prvkům

```
1 # vytvoreni pole vyctem
2 name=(element1 element2 ...)
3 # vytvoreni pole pomoci explicitni deklarace
4 declare -a anotherArray
5
6 # pristup k prvku na indexu index
7 ${name[index]}
```

Všechna existující pole lze vypsat pomocí[12]

```
1 foo=(ahoj svete blah)
2 declare -a
```

vypíše

```
[...]
declare -a foo=([0]="ahoj" [1]="svete" [3]="blah")
[...]
```

Čtenář může experimenty ověřit, že

- K jednoduché proměnné lze přistupovat jako k poli, hodnota proměnné je na indexu 0, na ostatních indexech je prázdný řetězec; `declare -a` ovšem jednoduché proměnné nevypisuje.
- Pole lze vytvořit i přímým vložením hodnoty na nějaký index, tj. např. `foo[3]=tri` vytvoří pole, které má na indexu 3 hodnotu `foo`; `declare -a` vypisuje pouze hodnoty na indexech, ke kterým bylo přistupováno.

Ke všem prvkům pole najednou lze přistoupit pomocí speciálního indexů `*` nebo `@`. V případě `*` je vytvořen z hodnot řetezec, kde jsou hodnoty jsou odděleny pomocí prvního znaku `IFS`. V případě `@` jsou odděleny mezerami. prvky pole. Je-li ovšem `@` použito v uvozovkách, pamatuje si, že je seznamem položek bez ohledu na oddělovače.

```
1 foo=("jedna jj" dva tri)
2
3 IFS=:
```

[12] builtin `declare` lze použít k deklarování či výpisu proměných vybraných vlastností, dokumentace `help declare`

```
A=(${foo[*]})
B=(${foo[@]})
C=("${foo[*]}")
D=("${foo[@]}")

declare -a
```

Vypíšeme

```
[..]
declare -a A=([0]="jedna jj" [1]="dva" [2]="tri")
declare -a B=([0]="jedna jj" [1]="dva" [2]="tri")
declare -a C=([0]="jedna jj:dva:tri")
declare -a D=([0]="jedna jj" [1]="dva" [2]="tri")
[..]
```

Na první pohled je záhadný řádek, kde bylo vytvořeno `A`. Expanze shellu tady slovo `jedna:dva:tri` rozdělila ve fázi *dělení slov* pomocí znaků `IFS`. Pokud je ovšem slovo v uvozovkách, tato expanze se potlačí a projeví se rozdíl, jako v našem příkladu u `C` a `D`. Upozorníme také, že pokud provedeme

```
foo=("jedna jj" dva tri)

# hodnota IFS je defaultni, obsahuje tedy mezeru
A=(${foo[*]})
B=(${foo[@]})

declare -a
```

dostaneme výše zmíněným expanzním mechanismem

```
[..]
declare -a A=([0]="jedna" [1]="jj" [2]="dva" [3]="tri")
declare -a B=([0]="jedna" [1]="jj" [2]="dva" [3]="tri")
[..]
```

To typicky není vyžadováno, v základu tedy doporučuji používat uvozovky.

Lze pracovat i s asociativnimi poli, kde se indexuje retězci.

```
# asociativnimu poli patri prepinac -A u declare
declare -A bar
bar[ahoj]=10
bar[svete]=20

declare -A
```

s výsledkem

```
[..]
declare -A bar=([ahoj]="10" [svete]="20" )
[..]
```

5 Misc

5.1 `set`

Set je velmi komplikovany builtin (dělá mnoho věcí). Spustíme-li jej bez argumentů, vypíše seznam všech proměnných s hodnotami a funkcí. Jinak `set` umí zapínat a vypínat některé vlastnosti shellu a nastavovat poziční argumenty.

```
set -o vlastnost  # zapnuti
set +o vlastnost  # vypnuti

#pripadne zkracene
set -zkratka # zapnuti
set +zkratka # vypnuti
```

`vlastnost` nahradíme jménem vlastnosti, `zkratka` je jedno písmenou zkratkou vlastnosti. Tabulku všech vlastností (se jménen, zkratkou a funkcí) lze nalézt v dokumentaci a literatuře. Seznam nastavených vlastností je uložen v proměnné `-` (zobrazit tedy lze `echo ${-}`).[13]

Jako příklad uvedeme vlastnost `xtrace` (zkratka x), která je defaultně vypnutá. Po zapnutí shell zobrazí (spolu s promptem `PS4`) přečtené části skriptu, po přečtení a po expanzi.

```
1 set -x
2
3 IFS=:
4 for dir in ${PATH}
5 do
6     echo -n ${dir}
7     echo " $(ls ${dir} | wc -l)"
8 done | sort -r -k 2
```

vede k

```
+ IFS=:
+ sort -r -k 2
+ for dir in ${PATH}
+ echo -n /usr/local/bin
++ wc -l
++ ls /usr/local/bin
+ echo ' 11'
+ for dir in ${PATH}
+ echo -n /usr/bin
++ wc -l
++ ls /usr/bin
+ echo ' 2886'
+ for dir in ${PATH}
```

[13] Další vlastnosti lze měnit pomocí `shopt`.

```
+ echo -n /bin
++ wc -l
++ ls /bin
+ echo ' 166'
+ for dir in ${PATH}
+ echo -n /usr/bin/X11
++ wc -l
++ ls /usr/bin/X11
+ echo ' 2886'
+ for dir in ${PATH}
+ echo -n /usr/games
++ wc -l
++ ls /usr/games
+ echo ' 1'
/usr/bin/X11 2886
/usr/bin 2886
/bin 166
/usr/local/bin 11
/usr/games 1
```

(Čtenář vypozoruje, že zapnuté vlastnosti jsou předány do podshellu a tak příkazy v podshellu pozná podle dvou `+`. Lze vypozorovat i něco o pořadí ve vztahu k přesměrování.)

Pomocí

```
set -- arg1 arg2 .. argn
```

jsou pozičním proměnným `1`, `2`, …, až `n` (pro vstupní argumenty skriptu/funkce) nastaveny hodnoty `arg1` až `argn` a proměnné `#` je nastavena hodnota `n`. Funguje i pro argumenty začínající znakem `-`.[14]

5.2 `source`

```
source file args
. file args       # alternativni jmeno
```

vloží obsah souboru `file` na místo spustění `source`. Soubor `file` (pokud není zadán plnou cestou, absolutní nebo relativní) je hledán v adresářích v proměnné `PATH` a poté v aktuálním adresáři. Volitelné argumenty `args` jsou předány souboru `file` jako poziční argumenty.

Příkaz je výhodné použít pro rozdělení skriptu do více souborů. (Rozdíl oproti spuštění souboru `file` jako skriptu je v tom, že se nevytváří podshell. To má důsledky například při práci s proměnnými).

[14] znak `-` je bez použití `--` jako prvního argumentu použit jako oddělovač přepínačů pro set a argumentů, které se mají přiřadit pozičním proměnným.

6 Úkoly

Cílem je napsat skript.

---

1. **[zápočtová úloha]**
   Vypište seznam všech souborů a podadresářů v aktuálním adresáři s informacemi o typu (soubor, adresar, symbolicky odkaz) a právech (čtení, zápis, spusteni) pro spouštějícího uživatele, ve tvaru: `jméno_souboru typ práva` Řešte bez použití programu `ls`.
   Upravte skript tak, aby
   - při zadání argumentu -a, zpracoval i skryté soubory a adresáře;
   - přijímal jako argument adresář, jehož obsah má vypsat. Při zadání neexistující skončí skript s chybou.
2. Naprogramujte "hádání"(celého) čísla, které si uživatel myslí, z intervalu zadaného až dvěma argumenty, pomocí série dotazů "je číslo menší/větší než X?"s odpověďmi ano/ne. Při jednom argumentu je levá mez intervalu 0, při žádném navíc pravá mez 100.
3. Implementujte zjednodušenou verzi programu `seq`: výpis posloupnosti (celých) čísel oddělených mezerou od čísla zadaného jako první argument do čísla zadaného jako třetí argument, obojí včetně, s přírůstkem zadaným jako druhý argument – kladným, pokud je počáteční číslo menší nebo rovno než koncové, jinak záporným. Při dvou argumentech je chybějící přírůstek roven 1, při jednom je i chybějící počáteční číslo rovno 1.
4. **[zápočtová úloha]**
   (a) Do aktuálního adresáře vygenerujte 1000 soubourů se jmény `fooNNNN.txt`, kde `NNNN` nahraďte pořadovými čísly 0001 až 1000. Do každého souboru zapište náhodně vytvořené datum (ne nutně reálně existující, např. 30 únor je OK) ve tvaru `DD/MM/YYYY`. (Dny, měsíce a roky v obvyklém rozsahu). (*Nápověda:* `${RANDOM}`.)
   (b) Uvažujme adresář obsahující soubory ve tvaru `*NNNN.txt` (symbol `*` představuje libovolnou předponu), každý z nich obsahuje datum ve tvaru `DD/MM/YYYY`. Vytvořte skript s tímto adresářem jako argumentem, který přesune soubory do podadresářů adresáře s cestami ve tvaru `YYYY/MM/DD` tak, aby si obsah souboru a jeho umístění odpovídaly. Každý podadresář bude existovat, jen pokud v něm bude alespoň jeden soubor.
5. Vypište řádek ze souboru `/etc/passwd`, který se nachází nad řádkem odpovídajícím uživateli, který skript spustil.
6. Vytvořte skript, který bude obálkou pro pro příkaz `mv`. Bude brát stejné parametry a fungovat jako `mv`, pouze jméno cílového adresáře bude očekávat jako první parametr.
7. Vytvořte skript se dvěma jmény (`min.sh` a `max.sh`), který bude dostávat jako parametry sérii čísel a spočítá jejich minimum nebo maximum, podle toho, se kterým jménem byl zavolán. (*dvě jména = dva soubory se stejným obsahem, případně soubor a link.*, *nápověda:* `basename`)

8. **[zápočtová úloha]**
   Vytvořte skript, jehož prvním argumentem je adresář, zbytek argumentů pak tvoří přepínače pro příkaz `find`. Skript příkaz `find` zavolá a soubory, které najde interaktivně zkopíruje do adresáře předaného skriptu jako první argument. Pře kopií každého souboru se skript uživatele zeptá, zda kopii provést. Možné odpovědi jsou

   - `ano` — soubor je zkopírován,
   - `ne` — implicitní odpověď, soubor není zkopírován,
   - `stop` — soubor není zkopírován a skript skončí.

   Pokud skript neskončí pomocí `stop` (ale projitím všech souborů), vypíše počet zkopírovaných souborů.
9. **[zápočtová úloha]** Vytvořte skript, jehož jediným argumentem bude adresář `A` a soubor `S`, obsahující na každém řádku koncovku jmen souborů (např. `txt`, `jpeg`). Skript v aktuálním adresáři vytvoří archiv souborů z adresáře `A`, které mají koncovky v souboru `S`. Archiv pojmenuje `X-DDMMYYYY-HHMM.zip`, kde `DDMMYYYY` je datum a `MMHH` je čas spuštění skriptu.
10. Vytvořte skript, který zjistí, jestli je aktuální uživatel `root`.

Úkoly jsou převzaty (a případně drobně upraveny) z literatury a z webu `http://outrata.inf.upol.cz/courses/shell/seminars.html`.

Reference

[1] Richard Blum. Christine Bresnahan. *Linux Command Line and Shell Scripting Bible*. Wiley, 2015.

[2] Ellie Quigley. *Unix shells by example*. Prentice Hall, 2015.

[3] Mark G. Sobell. *A practical guide to Linux commands, editors, and shell programming*. Prentice Hall, 2013.

[4] Libor Forst. *Shell v příkladech aneb aby váš unix skvěle shell*. Matfyzpress, 2010.

[5] Bash Manuál

Changelog

- 13. září 2013 - první verze