Úvod do skriptování v BASH – pro potřeby zkoušky

Proměnné

V `bash` se proměnné deklarují např takto:

```
#!/bin/bash
x=5 echo "Hodnota proměnné x je $x."
```

Deklarace – přiřadí se nějaká hodnota.
K proměnné se přistupuje pomocí znaku symbolu dolaru (**$**) uvedeného před názvem proměnné.

Řídící struktury

If ... else ... elif ... fi

```
if test -f /etc/foo
then #Soubor existuje. Zkopíruj tedy soubor
cp /etc/foo .
echo "Hotovo!"
else
# Soubor neexistuje. Vypiš tedy chybu
echo "Soubor neexistuje!"
exit
fi
```

Tento příklad tedy na základě vyhodnocení příkazem test soubor /etc/foo zkopíruje nebo neexistuje-li soubor, vypíše se chybové hlášení. K otestování existence souboru je použit příkaz `test`. Příkaz na základě přepínače -f určí, zda-li je /etc/foo soubor. Další možnosti přepínače příkazu test shrnuje následující tabulka. Je také možno použít manuálovou stránku:

Možnosti příkazu test	
-d	testuje, jestli soubor je adresář
-e	testuje, jestli soubor existuje
-f	testuje, jedná-li se o regulární soubor
-g	testuje, má-li soubor povolení SIGD
-r	testuje, zda-li jde soubor číst
-s	testuje, zda-li velikost souboru není 0
-u	testuje, má-li soubor povolení SUID
-w	testuje, zda-li se dá do souboru zapisovat
-x	testuje, zda-li je soubor spustitelný

Další možnost

```
if test -f /etc/foo
then
# se dá také zapsat takto:
if [ -f /etc/foo ]; then
```

Hranaté závorky nahrazují příkaz `test`.
Středník ; říká interpretru, že je zde konec řádku.

While ... do ... done

```
#!/bin/bash
while true; do
echo "Press Ctrl-C to quit."
done
```

Příkaz `true` je program, který vždy vrací hodnotu true (pravda). Náš skript tedy bude neustále opakovat vypisování hlášky jak ukončit program a to až do té doby než ho uměle přerušíme (Ctrl-C).

Rychlejší varianta použitím dvojtečky místo true:

```
#!/bin/bash
while :; do
echo "Press Ctrl-C to quit."
done
```

Dvojtečka je totiž interní příkaz interpretru `bash`.
V následujícím příkladu použijeme proměnnou x, která bude při každém průchodu cyklu zvyšovat svojí hodnotu a to až do té doby než dosáhne hodnoty 10.

```
#!/bin/bash
x=0; # inicializuje hodnotu x na 0
while [ "$x" -le 10 ]; do
echo "Aktuální hodnota x: $x"
# zvýšení hodnoty x o 1
x=$(expr $x + 1)
sleep 1
done
```

Následující tabulka shrnuje další možnosti příkazu test.

Možnosti příkazu test **Pro čísla:**	
x -eq y	Testuje, jestli x=y
x -ne y	Testuje, jestli x!=y (nerovná se)
x –qt y	Testjue, jestli x>y
x -lt y	Testuje, jestli x<y
x –ge y	Testuje, jestli x=>y

x –le y	Testuje, jestli x<=y
	Pro řetězce:
x = y	Testuje, jsou-li řetězce shodné
x != y	Testuje, jsou-li řetězce rozdílné
-n x	Vrátí true, není-li řetězec prázdný
-z x	Vrátí true, je-li řetězec prázdný

`expr -` tento příkaz ve skriptu přičte k proměnné x hodnotu 1 a vrátí to do x. Ale co znamená **$(...)**? Tato konstrukce zařídí, že hodnota výsledeku výrazu v závorkách se zapíše jako hodnota do výrazu, který tuto konstrukci obsahuje. Je tedy možné tímto způsobem například zjistit pod jakým jménem jsme přihlášeni do systému.
Příklad:

```
#!/bin/bash
x=$(whoami)
echo "Já jsem $x."
```

Samozřejmě lze také tímto způsobem dostat hodnotu vrácenou příkazem `whoami` přímo do řetězce. A to takto:

```
#!/bin/bash
echo "Já jsem $(whoami)."
```

Until ... do ... done

```
#!/bin/bash
until ls | grep texty.txt>/dev/null; do
done
echo "soubor texty.txt existuje"
```

Příkaz sleep pozastaví běh skriptu na tolik sekund, kolik je uveden parametr (v našem případě pozastaví běh skriptu na 1 sekundu).

For ... in ... do ... done

Struktura `for` se používá, chcete-li během cyklu použít různé hodnoty pro jednu proměnnou, a to tak, že při každém průběhu cyklu bude mít proměnná jinou hodnotu. Například chcete napsat skript, který desetkrát vytiskne každou sekundu tečku (.) lze to provést takto:

```
#!/bin/bash
echo -n "Zjiš»uji chyby v systému"
for dots in 1 2 3 4 5 6 7 8 9 10; do
echo -n "."
echo "Systém je čistý."
done
```

V příkazu `echo` vidíme parametr -n. Tento parametr určuje, že po vypsání textu se neskočí na další řádku na obrazovce, ale bude se pokračovat přímo za vypsaný text.
příklad

```
#!/bin/bash
echo "Zjiš»uje se existence adresářů v systému"
for x in /dev /etc /bin /root /var /usr /home /mnt; do
if [ -d "$x" ]; then
echo "Adresář $x existuje"
else
echo "Adresář $x neexistuje"
fi
done
```

Tento příklad testuje postupně jestli existují některé vybrané adresáře v systému uvedené jako hodnoty ve struktuře for. V příkladu je zkombinována struktura for se strukturou if.

Case ... in ... esac

Struktura `case` je velmi podobná struktuře `if`. V základu struktura case je určena pro vykonání jednoho z několika kusů kódu a to podle toho, jakou hodnotu má proměnná uvedená za příkazem case. Zde je příklad struktury `case`:

```
#!/bin/bash
x=5; # inicializuje x na hodnotu 5
# nyní se bude testovat hodnota x:
case $x in
0) echo "Hodnota x je 0."
;;
5) echo "Hodnota x je 5."
;;
9) echo "Hodnota x je 9."
;;
*) echo "Hodnta x není ani 0, ani 5, ani 9."
esac
```

Příkaz case tedy vyhodnotí hodnotu řídící proměnné x, podle hodnoty se poté najde příslušná hodnota před závorkou a ta se provede. Nebude-li odpovídat žádné hodnotě ve struktuře, zkusí se nalést ***)** a provede se blok příkazů za ním. Nenalezne-li se hvězdička, žádný příkaz ve struktuře se neprovede. Celá konstrukce by se též dala realizovat strukturou `if`:

```
#!/bin/bash
x=5
if [ "$x" -eq 0 ]; then
echo "Hodnota x je 0."
elif [ "$x" -eq 5 ]; then
echo "Hodota x je 5."
elif [ "$x" -eq 9 ]; then
echo Hodnota x je 9.
else
```

```
echo "Hodnota x je neurčena."
fi
```
Práce struktury case je tedy podle známé struktury if patrná. Nakonec ještě jeden příklad speciální konstrukce pomocí case.
```
#!/bin/bash

echo "Je nyní ráno? Odpovězte \"ano\" nebo \"ne\""
read timeofday

case "$timeofday" in
ano | a | Ano | ANO )
echo "Dobré ráno"
;;
[nN]* )
echo "Dobré odpoledne"
;;
* )
echo "Promiňte, ale nerozuměl jsem"
echo "Odpovězte \"ano\" nebo \"ne\""
exit 1
;;
esac

exit 0
```
Pro ještě lepší a efektivnější kontrolu odpovědi ano můžeme využít kontrukce pomocí žolíkových znaků
```
[aA] | [aA][nN][oO] )
```

Uvozovky

V shellu `bash` můžete psát tři různé typy uvozovek, z nichž každé mají jiný význam. A to jednak dvojité uvozovky: ", přední uvozovka: ' (tzv. devítka nebo apostrofa), a zadní uvozovka: `.
Dvojitá uvozovka se používá hlavně k určení několika slov oddělenými mezerami jako jeden řetězec. Tedy slova ve dvojitých uvozovkách se považují vždy za jeden parametr, a to jako jeden řetězec. Například použijeme-li příkaz:
```
xconsole$ mkdir hello world
xconsole$ ls -F
hello/ world/
```
Uvidíme to, co po použití příkazu ls -F vidíme. To jest vytvořené dva samostatné adresáře. Ale pokud použijeme následně jako parametr "hello world", uvidíme toto:
```
xconsole$ mkdir "hello world"
xconsole$ ls -F
hello world/
```
Vidíme tedy, že chceme-li vytvořit adresář, který obsahuje dvě a více slov oddělenými mezerami, musíme tento název uvést v dvojitých uvozovkách, jinak se nám vytvoří adresářů několik, z nichž každý bude mít název jako jedno ze slov uvedeného jako parametr příkazu mkdir.
Přední uvozovky také uzavírají řetězec podobně jako dvojté uvozovky, ale je zde jeden rozdíl. Řetězec uzavírající přední uvozovky bude zobrazen přesně tak, jak je zadán. Lepší pochopení usnadní následující příklad:
```
#!/bin/bash
x=5
echo "Hodnota proměnné x je $x"
echo 'Hodnota proměnné x je $x'
```
Pro pochopení je vhodné skript spustit a prohlédnout si výsledek. Je z toho patrné, že při použití dvojitých uvozovek se provede nahrazení $x za konkrétní hodnotu, kdežto pokud je výraz uzavřen do předních uvozovek (apostrof), k tomuto nahrazení nedojde.
Poslední typ uvozovek (zadní uvozovky) se používá místo této konstrukce:
```
x=$(expr $x + 1)
```
Jak vidíte x je přiřazena hodnota o jedno větší než byla hodnota x před provedením. Proměnná x se tedy inkrementovala. Zápis lze také provést pomocí uvozovek:
```
x=`expr $x + 1`
```
Který způsob je lepší? To záleží na vás. Já mám raději $(...), ale jak říkám, záleží na vás, jaký způsob vám lépe vyhovuje. Použít lze obojí. I když, někdy je možná estetičtější, obzvláště v řetězcích, použít výraz se závorkami:
```
echo "Já jsem `whoami`."
```

Aritmetika v BASH

BASH také podobně jako většina programovacích jazyků dovede provádět matematické výrazy. Jak jste již viděli, aritmetika se provádí pomocí příkazu `expr`. Nicméně stejně jako příkaz true je i příkaz expr pomalý, a proto lze aritmetika dělat pomocí interních příkazů `bash`. Stejně jako je dvojtečka alternativa k příkazu true, je i k příkazu expr alternativa. A tou je uzavření výrazu do kulatých závorek. Výsledek tedy dostaneme pomocí konstrukce `$((...))`. Rozdíl oproti výrazu `$(...)` je v počtu závorek.
příklad:
```
#!/bin/bash
x=8
y=4
z=$(($x + $y))
echo "Součet $x a $y je $z"
```
Bash je schopen vykonat i následující matematické operace:

Možné matematické operace	
Operace	**Znak**
Sčítání	+
Odčítání	-
Násobení	*
Dělení	/
Dělení modulo	%

Dělení modulo je zbytek po dělení. Tedy například 5 % 2 = 1, protože 5 děleno 2 je 2, zbytek **1**. Je také nutno podotknout, že bash dovede pracovat pouze s čísly celými. Nelze tedy používat desetinná čísla.
Zde je příklad skriptu s aritmetickými operacemi:

```
#!/bin/bash
x=5
y=3

add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))

echo "Součet: $add"
echo "Rozdíl: $sub"
echo "Součin: $mul"
echo "Podíl: $div"
echo "Zbytek: $mod"
```

Místo add=$(($x + $y)) je též možné použít add=$(expr $x + $y) nebo add=`expr $x + $y`.

Čtení uživatelského vstupu

read

```
#!/bin/bash
echo -n "Zadej své jméno: "
read jmeno
echo "Zdravím tě $jmeno!"
```

Skript počká, až napíšete vaše jméno a zmáčknete klávesu ENTER. Pak se to, co jste napsali uloží jako hodnota do proměnné `jmeno`. Ta se pak může dále v programu využívat stejně jako jakoukoliv jinou proměnnou. Využití hodnoty zadané uvádí následující příklad:

```
#!/bin/bash
echo -n "Zadej své jméno: "
read jmeno

if [ -z "$jmeno" ]; then
echo "Tys mi neřek tvé jméno!"
exit
fi

echo "Zdravím tě $jmeno!"
```

Toto je typický příklad kontroly vstupních dat. Nemůžeme totiž předpokládat, že uživatel vždy zadá to, co očekáváme a je tedy nutné se proti tomu bránit. Je však vhodné uživatele na špatně zadané data upozorňovat citlivě a ne tak, aby si uživatel z toho vydedukoval, že si o něm myslíme, že je blbec. Jestliže uživatel nezadá žádné jméno, bude podmínka vyhodnocena jako true, vypíše se chybové hlášení a skript se ukončí.

Zachytávání signálů

Signál je událost, kterou generuje systém UNIX (Linux) jako odpověď na nějakou podmínku a po jehož přijetí může proces (skript) provést nějakou akci. Signály generují některé chybové podmínky, například porušení segmentů paměti, chyby procesoru v plovoucí čárce nebo neplatné instrukce. Jsou generovány shellem a ovladači terminálu, které vyvolají přerušení. Může je také jeden proces explicitně poslat druhému procesu a předat mu tak nějakou informaci, případně ovlivnit jeho chování. Ve všech těchto případech je programové rozhraní stejné. Signály mohou být generovány, odchytávány anebo (některé přinejmenším) ignorovány. Názvy signálů jsou definovány v hlavičkovém souboru `signal.h`. Všechny začínají zkratkou "SIG". Výpis signálů systému Linux je uveden níže.
Ve vašich skriptech je možné použít zabudovaný program `trap`, který dovede zachytávat signály ve vašem programu. Je to dobrá cesta, jak uhlazeně ukončit program. Například, máte-li spuštěný program, zmáčknutím kláves Ctrl-C pošlete aplikaci signál přerušení, který zlikviduje váš program. Příkaz `trap` může zachytit signál a dát tak šanci programu v pokračování. Nebo se zeptá uživatele, jestli má skončit. `trap` používá následující syntaxi:
trap *akce signál*
Kde **akce** je to, co bude vykonáno při doručení signálu aplikaci. **Signál** je název signálu, který bude-li doručen, tak se vykoná příslušná akce. Signály, které může aplikace dostat si můžete nechat například vypsat pomocí příkazu `trap -l`. Používáte-li signály ve vašem skriptu, musíte vynechat první tři písmena jeho názvu (obvykle je to **SIG**). Například je-li signál **SIGINT**, bude do skriptu zapsáno pouze **INT**. Můžete také místo názvů signálů používat jejich čísla. Například číselná hodnota **SIGINT** je **2**. Nyní vyzkoušejte následující program:

```
#!/bin/bash
trap sorry INT
sorry()
{
echo "Sorry, to jsem nechtěl!"
sleep 3
}
for i in 10 9 8 7 6 5 4 3 2 1; do
echo "$i sekund do selhání systému."
sleep 1
done echo "Systém selhal!"
```

Nyní zkuste program spustit a zmáčknout Ctrl-C. To pošle skriptu signál SIGINT, který by měl přerušit program. Jelikož však signál bude zachycen a dojde ke zpracování funkce sorry(), která však program neukončí, takže bude běžet dál.
Chcete-li však, aby byl signál ignorován úplně, použijte jako akci "''". Chcete-li pak někde v programu již standardní zpracování signálu, použijte jako akci "-". Například:

```
# dojde-li signál, spustí funkci sorry()
trap sorry INT
# resetuje akci (nastavuje standardní chování SIGINT)
trap - INT
#nastavuje signál tak, aby byl ignorován
trap '' INT
```

Seznam signálů, které můžete v Linuxu použít (manuálová stránka signal z sekce 7 manuálových stránek):
```
SIGNAL(7)          Linux - příručka Programátora          SIGNAL(7)
```

AND & OR

|| znamenají nebo (OR, logický součet)
&& znamenají a (AND, logický součin)
Zde je příklad použití:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ] && [ "$y -eq 10 ]; then
echo "Obě podmínky jsou splněny"
else
echo "Nejsou splněny obě podmínky"
fi
```

Podobné použití má i logický součet. Následující příklad to ilustruje:

```
#!/bin/bash
x=3
y=2
if [ "$x" -eq 5 ] || [ "$y -eq 2 ]; then
echo "Alespoň jedna podmínka je splněna"
else
echo "Není splněna žádná podmínka"
fi
```

Další využití konstrukcí AND a OR je v tzv. **seznamech**. Konstrukce seznamu AND umožňuje provádět sadu příkazů tím způsobem, že další příkaz je proveden jen v případě úspěšného provedení všech předchozích příkazů. Má následující syntaxi:

```
příkaz1 && příkaz2 && příkaz3 && ...
```

Začne se zleva, je proveden první příkaz a vrátí-li hodnotu `true`, provede se další příkaz vpravo od něj. Tak to pokračuje, dokud některý z příkazů nevrátí hodnotu `false` a tím provedení seznamu skončí. Konstrukce && testuje hodnotu předchozího příkazu.
V následujícím skriptu provedeme příkaz `touch file_one` (čímž kontrolujeme existenci souboru file_one a pokud neexistuje, tak ho vytvoříme) a potom odstraníme soubor file_two. Potom pomocí seznamu AND otestujeme existenci každého ze souboru a vypíšeme nějaký text.

```
#!/bin/bash

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo "there"
then
echo -e "in if"
else
echo -e "in else"
fi

exit 0
```

Výstup programu bude následující:

```
xconsole$ ./skript.sh
hello
in else
```

Jak to funguje?
Příkazy touhc a rm zajistí požadovaný stav souborů v aktuálním adresáři. Seznam && pak provede příkaz `[ -f file_one ]`, který bude úspěšný, provede se příkaz echo. Ten bude také úspěšný (příkaz echo vždy vrací hodnotu true). Potom je proveden třetí test, `[ -f file_two ]`, který selže, protože tento soubor neexistuje. Jelikož byl poslední příkaz neúspěšný, neprovede se poslední příkaz seznamu echo. Výsledek seznamu && je false, protože jeden z příkazů seznamu selhal, tim pádem se provede větev else příkazu if.
Konstrukce seznamu **OR** umožnuje provádět skupinu příkazů, dokud jeden nevrátí hodnotu true, pak skončí. Syntaxe je následující:

```
příkaz1 || příkaz2 || příkaz3 || ...
```

Příkazy jsou prováděny zleva, jeden po druhém. Tak to pokračuje, dokud některý z příkazů nevrátí hodnotu true a tím provádění seznamu skončí.
Seznam || je podobný seznamu && s tím rozdílem, že nyní musí předchozí příkaz **selhat.**
Následující příklad je pouze modifikací příkladu minulého, liší se pouze v použití seznamu OR místo seznamu AND a také nevytvoří soubor file_one. Průběh však bude díky této konstrukci jiný.

```
#!/bin/bash

rm -f file_one

if [ -f file_one ] || echo "hello" || echo "there"
then
echo -e "in if"
else
echo -e "in else"
fi

exit 0
```

Výstup programu bude následující:

```
xconsole$ ./skript.sh
hello
in if
```

Jak to funguje? První dva řádky jen nastavují soubory pro zbytek skriptu. První příkaz `[ -f file_one ]`, selže, protože daný soubor neexistuje. pak je proveden příkaz echo. Ten vrátí hodnotu true a překvapivě již nejsou žádné další příkazy seznamu || provedeny. Příkaz if bude úspěšný, protože jeden z příkazů seznamu || (echo) vrátil hodnotu true.

Výsledek obou těchto konstrukcí je výsledkem posledního provedeného příkazu.
Seznam ne vyhodnocován podobným způsobem, jakým je v jazyce C vyhodnocováno víc podmínek. K určení výsledku je proveden jen minimální počet příkazů. Příkazy, které již výsledek nemohou ovlivnit, se neprovedou. Tomu říkáme tzv. **zkrácené vyhodnocování.**
Zkombinování těchto dvou konstrukcí dostáváme "ráj logiků". Vyzkoušejte:

```
[ -f file_one ] && příkaz pro true || příkaz pro false
```

Bude-li test úspěšný, provede se první příkaz, v opačném případě druhý. S těmito méně obvyklými seznamy je vždy lépe trošku experimentovat.
Pokud chcete použít víc příkazů na místě, kde je povolen příkaz jeden, například v seznamu AND nebo OR, můžete si pomoci složenými závorkami a vytvořit blok příkazů. V aplikaci, kterou si ukážeme dále, se například setkáte s následujícím kódem:

```
get_confirm && {
grep -v "^${cdcatnum}," $tracks_file > temp_file
mv $temp_file $tracks_file
echo
add_record_tracks
}
```

Použití argumentů

Když píšete do příkazové řádky název nějakého programu, většinou k němu připíšete i nějaké paramtery, které mění chování programu. Parametry také můžete napsat, když spouštíte nějaký skript.
Tyto argumenty jsou uloženy v proměnných. Proměnná `$#` uchovává počet argumentů předaných programu. Jednotlivé argumenty jsou uloženy v proměnných `$0, $1, $2,...` Je tedy vidět, že jsou uloženy v proměnných číslovaných od 0 do počtu parametrů, přičemž v proměnné `$0` je uložen název programu. Argumenty programu předané jsou až v proměnných 1 a více. Parametrů může být celkem 9. Nyní bude následovat příklad, ve kterém se parametry využijí:

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
echo "Použití: $0 argument."
fi

echo "První a jediný argument předaný skriptu je $1"
```

Tento program očekává pouze jeden parametr, aby se spustil program. Jestliže bylo předáno více nebo méně než jeden parametr, vytiskne zprávu použití. Jestliže byl jeden argument předán, bude také vytištěn.
Zde by bylo vhodné si uvést některé proměnné prostředí
Pokud budete chtít přistoupit ke všem parametrům skriptu jako k jediné proměnné, použijte ke čtení parametrů proměnnou $*. Ta vrátí seznam všech parametrů uložených v jedné proměnné a oddělených prvním znakem uvedeným v proměnné IFS. Použijete-li však pro přístup proměnnou $@, vrátí vám totéž co $*, ale nepoužívá proměnnou prostředí IFS. Přesný rozdíl mezi $* a $@ specifikuje X/Open.

Některé proměnné prostředí

Název	Popis
$HOME	Domovský adresář aktuálního uživatele
$PATH	Seznam adresářů oddělený dvojtečkami, ve kterých se mají hledat příkazy
$PS1	Prompt příkazové řádky, obvykle $
$PS2	Druhý prompt, který se využívá při dodatečném vstupu, obvykle >
$IFS	Oddělovač polí. Seznam znaků, které slouží k oddělování slov, když shell čte vstup, obvykle mezera, tabulátor a znak nový řádek
$0	Název skriptu shellu
$#	Počet předaných parametrů
$$	ID procesu skriptu, které se často používá uvnitř shellu ke generování jedinečných názvů dočasných souborů, například /tmp/tmpfile_$$

Přesměrování a roury

Přesměrujeme-li však výstup někam jinam pomocí znaku > (přesměrování) a za to soubor, kam se výstup přesměruje. Příklad přesměrování uvádí následující ukázka:

```
xconsole$ echo "Hello wolrd" > foo.file
xconsole$ cat foo.file
Hello world
```

V ukázce je tedy vypsán řetězec "Hello world" a to do souboru do kterého byl přesměrován výstup. Následující příkaz cat vypíše obsah onoho souboru.
Tento soubor nemusí existovat. S operátorem > je však jeden problém. Jeho použitím se totiž přepíše jakýkoliv soubor do kterého je výstup přesměrován. Proto existuje ještě jeden operátor >>, který výstup přesměruje na konec daného souboru. Podobně, jak když se v jazyce C otevírá soubor příkazem **open** s flagem O_APPEND.
Nakonec si ukážeme roury. Roury umožňují použít výstup programu jako vstup programu jiného. Roury se tvoří operátorem | (nejde o malé L). Tento znak se vytvoří na angl. klávesnici jako SHIFT-\. Následuje příklad roury:

```
xconsole$ cat /etc/passwd | grep xconsole > foo.file
xconsole$ cat foo.file
xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash
```

Příkaz break

Tento příkaz slouží k odskoku z cyklu `for`, `while` nebo `until` ještě před splněním řídicí podmínky. Příkazu break můžete předat číselný parametr, který udává, kolik cyklů má přerušit. Tím ale můžete hodně znepříjemnit čtení skriptů, takže vám jeho použití nedoporučuji. Příkaz break implicitně zruší jednu úroveň cyklu. Zde je příklad:

```
#!/bin/bash

rm -rf fred*
```

```
echo > fred 1
echo > fred 2
mkdir fred3
echo > fred4

for file in fred*
do
if [ -d "$file" ]; then
break;
fi
done

echo "První adresář fred byl $file"

rm -rf fred*

exit 0
```

Příkaz :

Příkaz dvojtečka je nulový příkaz. Příležitostně se používá ke zjednodušení logiky podmínek, kde zastupuje hodnotu `true`. Protože se jedná o zabudovaný příkaz, běží rychleji než příkaz `true`, i když je mnohem méně čitelný.
Můžete se s ním setkat v podmínce cyklů `while`. Zápis `while :` znamená nekonečný cyklus, který se ale běžněji zapisuje jako `while true`.
Kostrukce : je užitečná při podmíněném nastavování proměnných. Například:

```
: ${var:=value}
```

Pokud bychom neuvedli znak:, snažil by se shell vyhodnotit proměnnou `$var` jako příkaz.
V některých, zejména starších skriptech se můžete setkat s použitím dvojtečky na začátku řádku, kde uvádí komentář, ale moderní skripty by měly na začátku řádku komentáře vždy uvádět znak #, protože je to efektivnější.
Na závěr příklad:

```
#!/bin/bash

rm -rf fred
if [ -f fred ]; then
:
else
echo "Soubor fred neexistuje."
fi

exit 0
```

Příkaz continue

Podobně jako stejnojnenný příkaz jazyka C spustí tento příkaz další iteraci (průchod) cyklů `for`, `while` nebo `until`, přičemž je proměnné cyklu přiřazena další hodnota ze seznamu. Příklad:

```
#!/bin/bash

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred 3
echo > fred4

for file in fred*
do
if [ -d "$file" ]; then
echo "přeskakuji adresář $file"
continue
fi
echo soubor je $file
done

rm -rf fred*
exit 0
```

Příkaz `continue` může mít nepovinný parametr, který je číslo cyklu, jímž se má pokračovat, takže můžete částečně vyskočit z vnořených cyklů. Tento parametr je však jen zřídka používán, protože často velice stíží pochopení skriptů. Například skript:

```
#!/bin/bash

for x in 1 2 3
do
echo before $x
continue 1
echo after $x
done

exit 0
```

vrátí následující výstup:

```
before 1
before 2
before 3
```

Příkaz eval

Příkaz `eval` umožňuje vyhodnocovat argumenty. je zabudován do shellu a normálně neexistuje jako samostatný příka. Asi nejlépe jeho použití demonstruje následující příklad, který je přímo ze specifikace X/Open:

```
foo=10
x=foo
y='$'$x
echo $y
```

Tento skript vrátí $foo, ale následující

```
foo=10
x=foo
eval y='$'$x
echo $y
```

již vrátí 10. Tudíž příkaz `eval` funguje trochu jako další znak $ - vrátí hodnotu hodnoty proměnné.
Příkaz `eval` je velmi užitečný, protože umožňuje generování a spouštění kódu za běhu. Komplikuje ladění skriptů, ale umožňuje provádět věci, které bby normálně byly obtížné, ne-li nemožné.

Příkaz exec

Příkaz `exec` má dvě různá použití. Normálně slouží k nahrazování aktuálního shellu jiným programem. Například následující řádek

```
exec wall "Děkuji za všechny ryby"
```

ve skriptu nahradí aktuální shell příkazen `wall`. Řádky, které ve skriptu následují po příkazu exec, nebudou provedeny, protže shell, který skript prováděl, již neexistuje.
Druhý způsob použití příkazu `exec` umožňuje upravit deskriptory souboru:

```
exec 3< soubor
```

Výše uvedený zápis způsobí, že deskriptor tři bude otevřen pro čtení ze souboru `soubor`. Používá se jen zřídka.

Příkaz exit n

Příkaz `exit` způsobí ukončení skriptu s návratovým kódem `n`. Pokud ho použijete v příkazovém řádku kteréhokoliv interaktivního shellu, pak vás tento příkaz odhlásí. Pokud váš skript skončí, aniž by specifikoval nějaký návratový kód, použije se jako návratorý kód stav posledního příkazu provedeného ve skriptu. Vždy je dobré nějaký návratový kód uvést.
Při programování skriptů shellu znamená 0 úspěšné ukončení, 1 až 125 včetně pak chybové kódy, které mohou skripty využívat. Zbylé hodnoty jsou rezervovány: 126 - soubor nebyl spustitelný, 127 - příkaz nenalezen a 128 a vyšší - objevil se nějaký signál.
Toto je jednoduchý příklad, který vrátí nulu, pokud v aktuálním adresáři existuje soubor `.profile`.

```
#!/bin/bash

if [ -f .profile ]; then
exit 0
fi

exit 1
```

Pokud si libujete v hutných skriptech, můžete tento skript přepsat za pomoci seznamů AND a OR.

```
[ =f .profile ] && exit 0 || exit 1
```

Příkaz export

Příkaz `export` zpřístupní proměnnou, kterou mu předáte jako parametr, podřízeným shellům. P
Skript `export2`:

```
#!/bin/bash

echo "$foo"
echo "$bar"
```

Nyní skript `export1`. Na konci skriptu voláme skript `export2`.

```
#!/bin/bash

foo="The first meta-syntactic variable"
export bar="The second meta-syntatic variable"

./export2
```

Spustíme-li skript export1, dostaneme následující výstup:

```
xconsole$ export1

The second meta-syntatic variable
xconsole$
```

První řádek je prázdný, protože proměnná `foo` nebyla ve skriptu export2 dostupná, takže byla vyhodnocena jako prázdná. Předáme-li příkazu `echo` prázdnou proměnnou, vypíše jen znak nový řádek.
Jakmile byla proměnná exportována z shellu, bude exportována do všech skriptů spuštěných z tohoto shellu a také do všech dalších shellů, které z něj budou spuštěny. Pokud by skript export2 volal jiný skript, také on by měl k dispozici hodnotu proměnné `bar`.
Příkazy `set -a` nebo `set -allexport` exportují všechny nově vytvořené proměnné.

Příkaz printf

Příkaz `printf` je dostupný jen v moderních shellech (je dostupný v `bash`). Podle specifikace X/Open byste mu při vytváření formátovaného výstupu měli dávat přednosti před příkazem `echo`. Jeho syntaxe je:

```
printf "formátovací řetězec" parametr1 parametr2 ...
```

Formátovací řetězec je (s určitými omezeními) podobný formátovacímu řetězci v jazyce C či C++. Především nejsou podporovány reálné proměnné, přotože veškerá aritmetika v shellu probíhá v oboru celých čísel. Formátovací řetězec je tvořen libovolnou kombinací písme, řídících sekvencí a konverzních specifikátorů. Všechny jiné znaky než % a \ se na výstupu objeví v původní podobě.
Jsou podporovány následující řídící sekvence:

Řídící sekvence	Popis
\\	Znak zpětné lomítko
\a	Varování (pípnutí)
\b	Znak zpětné lomítko
\b	Znak vysunutí stránky
\n	Znak nový řádek
\r	Návrat vozíku
\t	Znak tabulátor
\v	Znak vertikální tabulátor
\ooo	Znak, jehož osmičková hodnota je *ooo*.

Konverzní specifikátor je o něco složitější, takže zde uvedeme jen běžně používané parametry. Další podrobnosti najdete v manuálu. Konverzní specifikátor je tvořen znakem % následovaným konverzním znakem. Základní konverze jsou tyto:

Konverzní specifikátor	Popis
d	Vypíše desítkové číslo
c	Vypíše znak
s	Vypíše řetězec
%	Vypíše znak %

Formátovací řetězec je pak použit k interpretaci zbylých paramterů a výstupu výsledku. Například:

```
xconsole$ printf "%s\n" hello
hello
xconsole printf "%s %d\t%s" "Hi there" 15 people
Hi There 15 people
```

Příkaz set

Příkaz `set` nastavuje proměnné shellu.

```
#!/bin/bash
echo Datum je $(date)
set $(date)
echo Měsíc je $2
exit 0
```

Tento program nastavuje seznam parametrů podle výstupu příkazu `date` a z pozičních parametrů `$2` pak získá požadovaný měsíc.
Protože výstup příkazu `date` závisí na místním jazyce, ve skutečnosti bychom název měsíce získali pomocí zápisu `date +%B`. Příkaz `date` má mnoho dalších parametrů, o kterých se dozvíte víc v manuálových stránkách.

Příkaz unset

Příkaz `unset` odstraní z prostředí proměnnou nebo funci. Netýká se to ale proměnných určených jen pro čtení, které definoval sám shell, jako například IFS. Příliš často se nepoužívá.
Skript

```
#!/bin/bash

foo="Hello world"
echo $foo

unset foo
echo $foo
```

vypíše nejdříve pozdrav Hello world a potom prázdný řádek.
Zápis `foo=` má podobný efekt jako příkaz `unset` ve výše uvedeném příkladu, ale nastavení `nulové` hodnoty řetězce nemá stejný účinek jako odstranění proměnné `foo` z prostředí.

Příkaz shift

Příkaz `shift` způsobí posun všech pozičních parametrů, takže z pozičního parametru $2 se stane $1, z $3 bude $2 atd. Hodnota $1 zmizí, zatímco hodnota parametru $0 bude zachována. Pokud příkazu `shift` předáte číselný parametr, posunou se poziční parametry o daný počet míst. Ostatní proměnné $*, $@, $# jsou tímto příkazem také ovlivněny.
Příkaz `shift` je vhodný pro procházení parametrů a pokud váš skript vyžaduje deset nebo více parametrů, budete ho potřebovat pro přístup k desátému a vyššímu parametru.
Jako příklad si můžeme ukázat sktipt, který projde všechny poziční parametry:

```
#!/bin/bash

while [ "$1" != "" ]; do
echo "$1"
shift
done
exit 0
```

Expanze parametrů

${parametr:-implicitní} je-li parametr prázdný, použij implicitní hodnotu.
${#parametr } vrací délku parametru.
${parametr%slovo} Od konce odstraní nekratší část parametru, která odpovídá slovu a vrátí zbytek.
${parametr%%slovo} Od konce odstraní nejdelší část parametru, která odpovídá slovu a vrátí zbytek.
${parametr#slovo} Od začátku odstraní nekratší část parametru, která odpovídá slovu a vrátí zbytek
${parametr##slovo} Od začátku odstraní nejdelší část parametru, která odpovídá slovu a vrátí zbytek.
příklad

```
#! /bin/bash
unset neco
echo ${neco:-lokomotiva }
neco=lod
echo ${neco:-lokomotiva }
cesta1=/usr/bin/X11/startx
```

```
echo ${neco#*/}
echo ${neco##*/}
cesta2=/usr/local/etc/local/networks
echo ${cesta2%local*}
echo ${cesta2%%local*}
```

vrátí se
lokomotiva
lod
usr/bin/X11/startx
startx
/usr/local/etc
/usr

Příkaz AWK – vyhledávání a zpracování předloh v souboru

Syntaxe *awk [-v **var=value**] –f **program-file [file-list]***
*awk [-v **var=value**] –f **program-file [file-list]***

Předlohy
<,<=,==,!=,>=,>
&& - AND
|| OR

Akce – obsahuje instrukce, které awk provede v případě, kdy zpracovaný řádek odpovídá předloze.

Proměnné

NR	Pořadové číslo aktuálního záznamu (řádku)
$0	Aktuální záznam
NF	Počet položek v aktuálním záznamu
$1-$n	Položky aktuálního záznamu
FS	Oddělovač vstupních položek (implicitně mezera nebo TAB)
OFS	Oddělovač výstupních položek (implicitně mezera)
RS	Oddělovač vstupních záznamů (implicitně znak nového řádku)
ORS	Oddělovač výstupních záznamů (implicitně znak nového řádku)
FILINAME	Jméno vstupního souboru

Pomocí volby **v** lze přiřazovat vstupní hodnoty

Funkce pro manipulaci s čísly a řetězci

length(str)	Vrací počet znaků řetězce str
int(num)	Vrací celočíselná část čísla num
index(str1,str2)	Vrací pozici řetězce str2 v řetězci str1
split(str,arr,del)	Elementy řetězce str oddělené znakem del uloží do pole proměnných arr[1],…arr[n] a vrací hodnotu n
sprintf(fmt,args)	Naformátuje argumenty args dle formátovací specifikace fmt
substr(str,pos,len)	Vrací podřetězec řetězce str počínaje pozicí pos s délkou len
tolower(str)	Vrací řetězec zadaný jako argument, ve kterém budou všechna velká písmena převedena na malá
toupper(str)	Vrací řetězec zadaný jako argument, ve kterém budou všechna malá písmena převedena na velká

Operátory
,/,%,+,-,=,++,--,+=,-=,=,/=,%=,

Asociativní pole
array[string] = value

for (elem in array) action

Příkaz **printf –** formátování výstupu

Příklady
awk { print }

awk ‘/tatra/‘ auta vybere řádky tatra ze souboru auta
awk ‘{print $3, $1}‘ auta vybere 3 a 1 sloupec ze souboru auta
awk ‘/tatra/{print $3, $1}‘ auta vybere 3 a 1 sloupec ze souboru auta, obsahující slovo tatra
awk ‘/h/‘ auta vybere se řádek, který obsahuje h
awk ‘$1 ~/h/‘ auta vybere se řádek, který obsahuje v první položce slovo h
awk ‘$1 ~/^~h/‘ auta vybere se řádek, který obsahuje h na prvním místě
awk ‘$2 ~/^[ts]/ {print $3, $1,“$“ $5}‘ auta vybere se sloupec 3,2,1,5, který obsahuje v první položce t, nebo s. Před 5. sloupcem bude dolar
awk ‘$3 ==65‘ auta awk ‘$3 <=65‘ auta awk ‘$3 ==“65“ && $5 < “9000“‘ auta
awk ‘/volvo/,/fiat/‘ auta awk ‘$3 ==65‘ auta

RPM
RPM (Redhat Package Manager) se zabyva spravou softwaru pomoci rpm balicku