FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Osvědčené postupy při skriptování v shellu
Student:	Michal Pešek
Vedoucí:	Ing. Lukáš Bařinka
Studijní program:	Informatika
Studijní obor:	Informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Prostudujte a zdokumentujte osvědčené postupy při skriptování v shellu z pohledu uživatele a programátora. Vytvořte a otestujte skript (sadu skriptů), který ilustruje použití osvědčených postupů v praxi od fáze tvorby skriptu až po jeho testování. Skript bude sloužit jako průvodce (wizard) při vytváření kostry nových skriptů v souladu s osvědčenými postupy.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 23. února 2018

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce, Ing. Lukáši Bařinkovi, za jeho ochotu a cenné rady při vypracování práce. Dále bych rád poděkoval rodině a přátelům za jejich nepodmíněnou trpělivost a morální podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. února 2019 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Michal Pešek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Pešek, Michal. *Osvědčené postupy při skriptování v shellu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce se zabývá využitím osvědčených postupů při skriptování v shellu Bash. Poskytuje podrobnou analýzu vybraných postupů pro mírně pokročilé uživatele a demonstruje jejich použití v praxi při návrhu a implementaci výsledného programu, který generuje kostru nových skriptů dle zadaných parametrů. Analytická část je zároveň doplněna o ukázkové kódy a spustitelnou sadu skriptů.

Klíčová slova návrh a implementace skriptu, generování kostry skriptu, analýza osvědčených postupů skriptování, UNIX, Bash

Abstract

This thesis concerns best practices in Bash shell scripting. It presents detailed analyses of selected practices for lower-intermediate users, and demonstrates their practical application in the design and implementation of the final program, which serves to generate new script templates according to specified parameters. The analytical part is supplemented by sample code excerpts and an executable set of scripts.

Keywords script design and implementation, script template generation, analysis of best practices in scripting, UNIX, Bash

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza existujících řešení **5**
2.1 Rešeršní část . . . 5
2.2 Praktická část . . . 5

3 Analýza osvědčených postupů **7**
3.1 Scriptovací jazyk Shell . . . 7
3.2 Efektivní skriptování . . . 9
3.3 Bezpečné skriptování . . . 14
3.4 Skript . . . 28

4 Návrh **37**
4.1 Popis . . . 37
4.2 Požadavky . . . 37
4.3 Struktura skriptu . . . 38
4.4 Přepínače . . . 39
4.5 Konfigurační soubor . . . 42

5 Implementace a testování **43**
5.1 Výběr technologií . . . 43
5.2 Důležité algoritmy . . . 43
5.3 Testování . . . 47

Závěr **49**

Literatura **51**

A Seznam použitých zkratek **53**

B Obsah přiloženého CD **55**

Seznam obrázků

4.1 Základní struktura skriptu 38

Seznam tabulek

3.1 Zásadní změny v Bash verzích 9

Seznam ukázkových kódů

1 Nebezpečný kód 10
2 Rychlostní převaha awku 11
3 Neefektivní kombinace utilit 12
4 Korektní kombinace utilit 12
5 While cyklus použitý v rouře 12
6 Neefektivní předání vstupu 13
7 Korektní předání vstupu 13
8 Modifikace při expanzi proměnné 13
9 Porovnání řetězců Bashem 14
10 Předání argumentů proměnnou 16
11 Brace expanze s proměnnou 16
12 Nestandardní hodnota IFS 18
13 Word-splitting proměnné v here-stringu 18
14 Korektně nežádoucí quotování 19
15 Quotování v here-dokumentu 19
16 Předání argumentu pouze při jeho existenci 20
17 Proměnná IFS a expanze pole 20
18 Proměnná IFS a builtin read 21
19 Změna proměnných na omezenou dobu 21
20 Nvidia Bumblebee překlep 22
21 Hypertextový odkaz se speciálním znakem 22
22 Shellovská náhrada souborů 23
23 Program find 25
24 Null byte 25
25 Chyba Steam klienta 26
26 Neočekávané ukončení skriptu nastavením errexit 27
27 Předání proměnných externímu programu 27
28 Výhody deskriptivního programování 32
29 Reagování na signály pomocí builtinu trap 35
30 Getopt normalizace 44

31 Duplicitní kombinace přepínačů 45
32 Generování podmínek pro proměnné 46
33 Lokalizování proměnných 46

Úvod

S narůstající oblibou Unixových systémů u široké veřejnosti, roste počet běžných uživatelů jedné z jejich hlavních komponent – interpretu příkazů. Interpret příkazů (neboli shell) umožňuje i méně zkušeným uživatelům využívat systém jednodušeji a efektivněji.

Analytická část práce bude prospěšná pro všechny uživatele Unixových systémů, kterým představí doporučené postupy při používání shellu. Zaměření je převážné na mírně pokročilé uživatele, kteří již znají základy skriptování a struktur jako pole nebo funkce. Praktická část ulehčí tvorbu nových shell skriptů programátorům i běžným uživatelům.

Jeden z nejpoužívanějších, a v mnoha Linux či macOS systémech defaultní interpret, je Unix shell se jménem Bash. Většina materiálů analyzujících osvědčené postupy při skriptování se zaměřuje na velmi specifickou a často velmi pokročilou oblast nevyužitelnou pro běžné uživatele. Oficiální dokumentace interpretu Bash je také velice rozsáhlá a nadměrně podrobná. Podobné zdroje uživatelům nepřináší žádný užitek a nepomáhají v odhalování častých chyb či neočekávaných chování.

Mnoho skriptů, které dodržují nějakou podmnožinu známých osvědčených praktik, řeší podobné, často se opakující problémy, jejichž řešení se dá automatizovat. Současná řešení, která se o toto automatizování snaží, jsou nedostatečná nebo se zabývají pouze specifickou částí. Z uvedených důvodů jsem se rozhodl pro výběr tématu, které tuto problematiku efektivně řeší.

Kapitola 1

Cíl práce

Cílem rešeršní části práce je analýza typických osvědčených postupů při skriptování v shellu. Hlavní zaměření je na Unix shell Bash verze 4.4, který v době psaní práce patřil mezi nejpopulárnější interprety a GNU utility. Ostatními interprety se rešeršní ani praktická část práce do hloubky nezabývá, pouze zmiňuje důležité rozdíly a problémy s tím spojené. Vybrané postupy budou doplněny o ukázkové skripty.

Praktická část práce navazuje na analýzu postupů a jejím hlavním cílem je demonstrovat jejich použití při návrhu a implementaci skriptu či sady skriptů, které generují kostru nového skriptu. Základní kostra je u mnoha skriptů velmi podobná a výsledek praktické části umožňuje uživateli tvorbu této kostry automatizovat. Rešeršní i praktická část práce se zaměřují na mírně pokročilé uživatele.

Kapitola 2

Analýza existujících řešení

Analýzu stávajících řešení rozdělíme na rešeršní a praktickou část.

2.1 Rešeršní část

V současné době je mezi uživateli oblíbený nástroj ShellCheck. Tento nástroj zkontroluje Váš kód a upozorní na části kódu, které se jeví jako chyba. Problém toho nástroje je, že není a z podstaty ani nemůže být stoprocentní. Chování některých shellovských konstrukcí může být v jistých situacích považováno za chybové, za jiných situací to může být chování požadované.

Dalším řešením analytické části jsou různé odborné publikace a oficiální manuál. Přesto, že jsou kvalitními zdroji, zaměřují se na příliš pokročilou problematiku a často jsou pro obyčejného uživatele těžko stravitelné. Ze zmíněných důvodu proto ani jedno řešení nevyhovuje jako náhrada rešeršní části této práce.

2.2 Praktická část

Jedno z důležitých řešení na poli generování skriptů je nástroj Argbash. Jedná se o online a command line nástroj, který pomocí parametrů dokáže generovat zpracování parametrů. Prakticky se toto řešení zabývá podmnožinou našeho problému, ale s jiným přístupem. Řešení Argbash totiž nespoléhá na žádné externí nástroje při zpracování přepínačů, generuje tedy kód nezávislý na nástrojích, což značně zvyšuje přenositelnost.

Výsledná aplikace bude využívat externí nástroje na zpracování přepínačů, převážně kvůli dosti jednodušší implementaci. Nástroj Argbash může být doporučen jako doplněk výsledné aplikace.

Kapitola 3

Analýza osvědčených postupů

3.1 Scriptovací jazyk Shell

Cílem této podkapitoly není sepsání stovky nudných definic a dat, ale předání stručného a obecného přehledu o Unix shellech před samotnou analýzou osvědčených postupů jazyka Bash a GNU utilit.

3.1.1 Co je Shell?

Unix shell je označení pro interpret příkazů a programovací jazyk, který zprostředkovává rozhraní mezi uživatelem a operačním systémem. V podstatě se jedná o makroprocesor, který uživateli umožňuje spouštět a kombinovat sadu programů [1].

3.1.2 Základní pojmy

V této sekci si definujeme důležité základní pojmy používané v textu této práce.

Unix
: Inovativní operační systém vyvinutý v Bellových laboratořích v roce 1969. Portabilita, docílena přepsáním jádra v programovacím jazyce C, byla klíčem komerčního úspěchu Unixu.

BSD
: Verze Unixu vyvinuta Kalifornskou univerzitou v Berkeley

GNU
: Unix-like operační systém obsahující mnoho klíčových GNU programů a nástrojů, například: GNU Bash, GNU Awk apod.

POSIX
: Sada standardů specifikovaných organizací IEEE.

Builtin
Příkaz interně implementovaný shellem.

Metaznaky
Znaky, které mají pro shell speciální význam.

Bílé znaky
Mezera, tabulátor a nový řádek.

Snippet
Označuje kus programového kódu.

3.1.3 Historie

Za existence Unixu vznikl bezpočet shell interpretů. První Unix shell napsal v Bellových laboratořích v roce 1971 Ken Thompson a nesl název „Thompson shell“. Thompson shell byl však velmi minimalistický a měl mnoho omezení, zejména z programovacího hlediska. Neobsahoval například proměnné, jedinou řídící strukturou byl goto label apod. Tyto limitace podnítily vývoj dalších, lepších shellů. Prvním významným následníkem byl „Bourne shell“ [2].

Bourne shell napsal Steven Bourne a poprvé byl zahrnut v Unix verzi 7 vydané roku 1979. Bourne shell byl oproti původnímu shellu velkým zlepšením. Obsahoval běžné řídící struktury, proměnné, náhradu příkazů apod. Řada aktuálně populárních shellů jsou implementacemi Bourne shellu (např. Bash). Psaní skriptů pro Bourne shell je stále populární z důvodu vysoké přenositelnosti. Významnou alternativou byl „C shell“, který napsal Bill Joy pro BSD verzi Unixu. C shell dostal své jméno díky podobnosti syntaxe programovacímu jazyku C. Velkou výhodou byla implementace některých funkcionalit, které Bourne shell postrádal (např. job control) [3].

V neposlední řadě vznikl díky Davidu Kornovi „Korn shell“. Ksh se snažil adresovat nedostatky jak Bourne shellu, tak C shellu. Jednalo se o jakousi střední cestu. Zároveň implementoval mnoho funkcionality navíc. Bash se Korn shellem nechal v mnoha směrech inspirovat [3].

3.1.4 Srovnání shellů

Od dob prvních shellů uplynula dlouhá doba, za kterou se vyvinulo mnoho pokročilých moderních shellů. Aktuálně mezi nejpoužívanější patří:

Bash (Bourne-again shell)
Shell GNU projektu, Bash, je následníkem a z velké části kompatibilní s Bourne shellem. Převážně se řídí standardem POSIX. Implementuje navíc mnoho užitečné funkcionality převzaté z Korn shellu i C shellu. Bash je aktuálně nejrozšířenější shellem a výchozím shellem pro většinu Linux distribucí.

Zsh (Z-shell)
Z-shell je v podstatě, stejně jako Bash, rozšířená verze Bourne shellu a adekvátní alternativa Bashe. Navrch má v oblasti interaktivního použití (např. pokročilá náhrada souborů).

Ksh (Korn shell)
Korn shell a Bash sdílí mnoho vlastností. Ksh obsahuje navíc například nativní podporu floating-point aritmetiky, kterou Bash stále neimplementoval a je též adekvátní alternativou Bashe.

Tcsh (Tenex C shell)
Tcsh je vylepšená verze C shellu. V dnešní době v mnoha věcech za ostatními shelly zaostává.

3.1.5 Bash verze

Podstatná část důležité funkcionality je implementována zhruba od verze 3.0. V době psaní této práce byla **nejaktuálnější verze 5.0** vydaná na počátku roku 2019. Nahradila verzi 4.4, jež je rozšířená mezi populárními Linux distribucemi (např. Ubuntu).

Tabulka 3.1 popisuje vybrané verze, které přispěly významnou nebo zajímavou funkcionalitou. Kompletní seznam je k nalezení na oficiálních webových stránkách projektu GNU Bash.

Tabulka 3.1: Zásadní změny v Bash verzích [4]

Funkcionalita	**Verze**
mapfile -d	4.4
jmenné reference	4.3
lastpipe	4.2
globstar, mapfile, ${var,}, asociativní pole	4.0
nocasematch, printf -v var, += operátor	3.1
brace expanze, =~ operátor, pipefail	3.0
printf, read, here-string, extglob, pole, aritmetická expanze ...	< 3.0

3.2 Efektivní skriptování

Cílem této podkapitoly je analyzovat postupy efektivnějšího skriptování. Efektivnějšího skriptu můžeme docílit dodržením jednoduchých principů.

- Zvolit vhodný nástroj
- Neplýtvat prostředky systému

3.2.1 Výběr efektivních nástrojů

Pouhé Bash řešení je často nevhodnou volbou z důvodu rychlosti a přenositelnosti. Využití dostupných GNU utilit či vyššího programovacího jazyka je v mnoha případech lepší volbou. V počáteční designové fázi musí programátor zvážit veškeré faktory a zvolit správné nástroje.

3.2.1.1 Nevýhody Bashe

Bash je interpretovaný jazyk určený převážně pro spouštění a kombinování programů. V praxi se používá hlavně administrátory pro jednoduché úkony, např. manipulace souborů, automatizace apod.

Mezi hlavní nevýhody Bashe oproti ostatním programovacím jazykům patří rychlost. Každé spuštění externího programu vyžaduje systémová volání *fork* a *exec* [5] a samotné interpretování příkazové řádky má též určitou režii. Bash zároveň není stavěný pro rozsáhle manipulace textu, postrádá běžné struktury vyšších programovacích jazyků (např. 2D pole) a běžnou funkcionalitu (např. floating-point aritmetiku).

Syntax Bashe je na druhou stranu triviální. Nevýhodou však je, že chybový kód je shellem často považován za korektní syntax a v mnoha případech neskončí chybou jako u jiných jazyků. Podobné chyby mohou mít okamžité fatální následky nebo skryté následky z dlouhodobého hlediska. Bezpečností se práce zabývá v následující podkapitole.

```
rm -rf /usr /bin/some/path
rm -rf $file
```

Snippet 1: Nebezpečný kód

3.2.1.2 Vyšší programovací jazyky

Oblíbenou alternativou u systémových administrátoru je vyšší programovací jazyk **Perl**, který byl původně navržen, aby podporoval pokročilou manipulaci textu [6]. V dnešní době se Perl používá pro širokou škálu úkonů od systémové administrace až po vývoj grafických prostředí. Perl navíc podporuje objektové programování. Mezi další populární jazyky, které v mnohém předčí Bash a GNU utility, patří **C**, **C++** a **Python**.

3.2.1.3 GNU utility

V průběhu vývoje operačního systému GNU vznikla kromě Bashe řada užitečných a mocných programů, které se za pomocí shellu dají kombinovat. Velká část těchto utilit je napsána v jazyce C a pro větší objem dat doporučena.

V některých případech je jejich použití dokonce nezbytné pro zprostředkování chybějící funkcionality. Příkladem všestranného nástroje je GNU `awk`.

Nehledě na předchozí doporučení tyto utility používat, musíme stále dbát obezřetnosti a zvážit, zejména kdy a kterou z nich správně použít. Utilita `sed` například podporuje rozšířené regulární výrazy, což často vede uživatele k nesprávnému použití `sedu` nebo `awku` pro zpracování složitých struktur jako JSON nebo XML. Pro takový úkol `sed` nikdy nebyl určen. JSON struktura je totiž dosti komplikovaná na to, aby vyžadovala zpracování specializovaným nástrojem či knihovnou, například nástrojem `jq`.

Snippet 2 demonstruje rychlostní převahu `awku` pro relativně velká data oproti čistému Bash řešení. Úkolem je jednoduché vypsání součtu 100 000 jednociferných čísel.

```
# Bash řešení, doba běhu přibližně 1.4s
sum=0
while IFS= read -r num; do
   ((sum+=num))
done < file
printf '%d\n' "$sum"

# Awk řešení, doba běhu přibližně 0.02s
awk 'BEGIN{sum=0}{sum+=$1}END{print sum}' file
```

Snippet 2: Rychlostní převaha awku

3.2.1.4 Efektivní kombinace utilit

Základem Unix filozofie je vytváření programů, které dělají pouze jednu věc a efektivně [7]. Tyto programy následně vzájemně kombinovat za účelem vytváření komplexních řešení.

Většina utilit jsou textovými filtry. Na svém vstupu přijímají text, který následně již zpracovaný produkují na svůj výstup. Kombinování se realizuje pomocí shellovské roury `|`. Roura je operátor, který propojí výstup programu se vstupem následující programu. Každý program v rouře je spuštěn v **separátním procesu**, stejně tak jako každá náhrada příkazu.

Jak již bylo zmíněno, vytvoření nového procesu vyžaduje určitou režii ze strany operačního systému. Efektivní řešení se snaží **nadbytečným** procesům vyvarovat. Argument některých programátorů pro nadbytečné roury je částečně zvýšená přehlednost, která je dle mého názoru nepatrná. Velké procento zbytečných kombinacích vychází z neznalosti všech funkcí používaných utilit.

```
grep 'auto' file | sed 's/A/B/'
cut -f1 file | awk 'NR!=1'
sort file | uniq
wc -l file | awk '{print $1}'
echo "$(cat file)"
```

Snippet 3: Neefektivní kombinace utilit

Ve snippetu 3 pocházejí první dvě redundantní kombinace z neznalosti faktu, že druhá utilita implementuje použitou funkcionalitu první utility. Třetí kombinace opomíná existenci přepínače `-u` utility `sort` a čtvrtá kombinace vychází z neznalosti chování utility `wc`, která jméno souboru nevypíše, pokud je soubor přesměrován na standardní vstup. V poslední kombinaci je použito zbytečné `echo`, jelikož utilita `cat` sama o sobě vypisuje na standardní výstup.

```
sed -n '/auto/{s/A/B/;p}' file
awk 'NR!=1{print $1}'
sort -u file
wc -l < file
cat file
```

Snippet 4: Korektní kombinace utilit

Používání nadbytečných procesů není negativní pouze z pohledu plýtvání prostředků systému, ale také z pohledu někdy neočekávaných a špatně identifikovatelných chyb. Potomek procesu totiž nemůže měnit rodičovský proces a všechny změny provedené v potomkovi jsou po návratu ztraceny. Nejčastějším příkladem tohoto chování je `while` cyklus použitý v rouře.

```
sum=0
cat file | while IFS= read -r line; do
   printf '%s\n' "$line"
   ((sum++))
done
# Hodnota $sum nebyla vůbec změněna
```

Snippet 5: While cyklus použitý v rouře

3.2.2 Předání vstupu

S předchozí sekcí úzce souvisí nevyužívání přesměrování či argumentů na předání vstupu. Programátoři často předávají vstup zbytečně pomocí programů `cat` a `echo`. Převážná většina programů navíc zvládá pracovat i s vícero soubory. Místo spouštění jednoho `grep` skriptu pro každý zpracovávaný soubor,

můžeme spustit `grep` pouze jednou s vícero argumenty. Další užitečnou výhodou přesměrování či argumentů je mimo jiné možnost v souboru seekovat.

```
cat file | sed 's/a/b/'
echo "$var" | tr '[:upper:]' '[:lower:]'
```

Snippet 6: Neefektivní předání vstupu

Jeden z mála populárních programů, který neakceptuje soubor jako jeden ze svých argumentů, je program `tr`. V podobných případech můžeme využít přesměrování souboru pomocí běžného `<` operátoru nebo přesměrování textu pomocí takzvaného here-dokumentu a here-stringu (`<<` a `<<<` operátory).

```
sed 's/a/b/' file
tr '[:upper:]' '[:lower:]' <<< "$var"
```

Snippet 7: Korektní předání vstupu

3.2.3 Bash funkcionalita

V předchozích sekcích jsme si ukázali, že pro velké vstupy a rozsáhlé manipulace dat jsou externí nástroje vhodnější volbou. Pro malé vstupy a jednoduché manipulace se však často nevyplatí vytvářet nové procesy. Pokud například potřebujeme provést jednoduchou transformaci jedné proměnné, spouštění `sedu` představuje zbytečnou režii. Bash implementuje řadu interní funkcionality, kterou se dají nahradit nástroje jako `seq`, `grep` nebo `sed`.

Expanze proměnných
Příkladem této interní funkcionality je možnost provádět modifikace při expandování proměnných. Zvládne běžné úkony jako extraktování přípon souborů nebo odstranění nežádoucích znaků. Tyto modifikace jsou prováděny Bashem interně a jsou velmi efektivní. Snippet 8 demonstruje vybrané modifikace.

```
# Získání podřetězce
awk '{print substr($0,2,3)}' <<< "$var"
printf '%s\n' "${var:1:3}"
# Jednoduchá substituce
sed 's/[0-9]//g' <<< "$var"
printf '%s\n' "${var//[0-9]}"
# Změna kapitalizace písma
tr '[:upper:]' '[:lower:]' <<< "$var"
printf '%s\n' "${var,,}"
```

Snippet 8: Modifikace při expanzi proměnné

Porovnání řetězců
Bash také disponuje nativní podporou porovnání řetězců a regulárních výrazů, které snadno nahradí program `grep`. Klíčové slovo `case` umí porovnávat řetězce za pomocí shellovských vzorů pro náhradu souborů (rozšířené vzory jsou též podporovány). Klíčové slovo `[[` zvládne při použití operátoru `=` stejná porovnání jako `case`. Při použití `=~` operátoru navíc podporuje rozšířené regulární výrazy. Nechybí ani možnost zachytit podvýrazy regulárních výrazů, které Bash ukládá do speciálního pole `$BASH_REMATCH`.

Bash je možno přizpůsobit pomocí builtinu `shopt -s nazev_nastaveni`. Při použití nastavení `nocasematch` porovnávání řetězců ignoruje velikost písma.

```
# Shellovské vzory
case "$var" in
   Abc*) printf 'OK\n' ;;
esac

# Regulární výrazy
if [[ "$var" =~ [0-9]+$ ]]; then
   printf '%s\n' "${BASH_REMATCH[0]}"
fi
```

Snippet 9: Porovnání řetězců Bashem

Existuje mnoho dalších příkladů, kdy programátoři nevyužívají všech možností Bashe. Problémem mnohdy není jen efektivita, ale také složitější použití nebo spoléhání na špatné praktiky jako word-splitting. Vybrané příklady nevyužívání funkcí Bashe zmiňuje následující list.

- Používání `expr` místo aritmetické expanze `$((výraz))`.
- Používání `seq` místo brace expanze nebo C-style `for` cyklu.
- Používání `ls` místo náhrady souborů.
- Používání nekompatibilního `echo -e` místo `$'...'`.
- apod.

3.3 Bezpečné skriptování

Cílem této podkapitoly je poukázat na nejčastější chyby a předat postupy, jak se jim vyvarovat. Chyby často vycházejí z toho, jak Bash zpracovává příkazovou řádku.

3.3.1 Zpracování příkazové řádky

Bash zpracovává příkazovou řádku dle následujících kroků [1]:

1. Načtení vstupu ze souboru, z argumentu dodaného při použití `-c` přepínače, nebo z terminálu uživatele.
2. Rozdělení vstupu na slova a operátory, které se řídí pravidly quotování. V tomto kroku se též provádí expanze aliasů.
3. Zpracování tokenů na jednoduché a složené příkazy.
4. Provedení shellovských expanzí.
   - brace expanze
   - náhrada znaku tilda
   - expanze proměnných
   - náhrada příkazů
   - aritmetická expanze
   - word-splitting
   - náhrada souborů
5. Přesměrování vstupu a výstupu.
6. Spuštění programu.
7. Případně čekání na dokončení programu.

Nejproblémovějším krokem je provedení shellovských expanzí, kterým se budou zabývat následující sekce. Již samotné **pořadí kroků** s sebou nese jisté problémy, které vedou nepozorné programátory k chybám.

In-place editování
In-place editování znamená použití vstupního souboru i pro výstup. Přesměrování výstupu se však děje o jeden krok dříve než samotné spuštění programu. Při otevření souboru pro zápis pomocí `>` operátoru se obsah tohoto souboru smaže dříve, než má program možnost soubor začít zpracovávat. Výsledkem je ztráta obsahu souboru. Nejspolehlivějším řešením je použití dočasného souboru. Některé programy in-place editování podporují pomocí přepínačů.

Příkaz v proměnné
Dalším příkladem je uchování příkazu v proměnné. Klíčovým je krok číslo 2, ve kterém je vstup rozdělen na slova a operátory, přičemž se řídí pravidly quotování. Jinak řečeno operátory jsou shellem již identifikovány před expanzí proměnných. Z tohoto důvodu není možné v proměnné uchovávat například přesměrování nebo roury. Řešením je použití funkce.

Proměnné jsou nesprávně používány i pro uchování všech argumentů některého programu. Shell quotování po word-splittingu znovu nezpracovává, a tak se uvozovky v proměnné stanou doslovnými a ne syntaktickými. Řešením je použití správné struktury pro uchování argumentů, pole [8].

```
# Nesprávné předání argumentů
arg='-c "hello world"'
grep $arg file

# Správné předání argumentů pomocí pole
arg=('-c' 'hello world')
grep "${arg[@]}" file
```

Snippet 10: Předání argumentů proměnnou

Proměnná v brace expanzi
Brace expanze i expanze proměnných se provádí ve 4. kroku. Bash ale i tyto expanze provádí v předem určeném pořadí a z tohoto důvodu použití proměnné v brace expanzi, například za účelem generování číselné řady, nebude fungovat. Řešením je použít C-style `for` cyklus nebo ve výjimečných případech použít externí program `seq`.

```
number=10
printf '%s\n' {1..$number}
# Brace expanze nebude rozpoznána a kód vypíše '{1..10}'
```

Snippet 11: Brace expanze s proměnnou

Aliasy
Aliasy jsou expandovány již ve 2. kroku, daleko před spuštěním příkazu. Následkem je dosti matoucí chování aliasů, které jsou použity na stejné řádce jako jejich deklarace nebo použití aliasů uvnitř složených příkazů. Bash manuál výslovně doporučuje aliasy deklarovat na samostatné řádce a nepoužívat je uvnitř složených příkazů. Dalším překvapivým chováním může být fakt, že deklarace funkce je sama o sobě příkaz a alias tedy může přepsat název deklarované funkce!

3.3.2 Quotování

Quotování je způsob, jak odstranit speciální význam některých znaků a slov shellu. Existují tři quotovací mechanismy pro odstranění speciálního významu.

1. obrácené lomítko
2. jednoduché uvozovky

3. dvojité uvozovky

Obrácené lomítko ruší speciální význam **následujícího** znaku. Uvnitř jednoduchých uvozovek ztrácejí svůj speciální význam **všechny** znaky. Jednoduchá uvozovka ale nemůže být uvnitř jednoduchých uvozovek použita. Uvnitř dvojitých uvozovek ztrácí speciální význam **většina** znaků, mimo `$`, `` ` ``, `\`.

Následující podsekce se zabývají **nejčastější** a nejznámější chybou při skriptování v Bashi a to nechráněním řetězců a proměnných proti speciálním znakům.

3.3.2.1 Word-splitting

Word-splitting je mechanismus, který se provádí po expanzích proměnných, náhradách příkazů a aritmetických expanzí, které nebyly uvnitř dvojitých uvozovek. Bash rozdělí tyto expanze na slova použitím hodnot v proměnné `IFS` jako oddělovače. Defaultní hodnota `IFS` je mezera, tabulátor a nový řádek. Po word-splittingu se navíc na výsledných slovech provádí náhrada souborů.

První shelly podporovaly pouze datový typ string. Jedinou možností, jak pracovat s listy položek, bylo uložení položek do stringu za pomocí oddělovačů, které byly později použity pro word-splitting. Dnešní shelly již podporují datový typ pole, a tak se word-splitting stal nepříjemným přebytkem. Například dříve zmíněný shell Zsh, který pole podporoval od svého vzniku, word-splitting již defaultně neprovádí.

Proč a kdy quotovat
Jak již bylo zmíněno, expanze, které nejsou uvnitř dvojitých uvozovek, podléhají word-splittingu a náhradě souborů. Problém nastává v momentě, kdy výsledek naší expanze obsahuje některé znaky z proměnné `IFS` nebo shellovské vzory pro náhradu souborů.

Problém nastává obzvlášť pokud manipulujeme s názvy souborů, které v dnešní době běžně obsahují mezery. Příkladem je použití příkazu `rm` s nechráněnou proměnnou. Pokud taková proměnná obsahuje mezery, `rm` neobdrží pouze jeden soubor (slovo) ke smazání, ale hned několik, což může mít fatální následky.

Obecným doporučením je quotovat **všechny** expanze. Toto doporučení zahrnuje i některé méně zřejmé případy jako quotování **uvnitř** náhrady příkazů. Zároveň však existují situace, kdy quotování není nutné nebo je naopak nežádoucí.

Kdy není quotování nutné
V Bashi existuje několik případů, kdy quotování expanzí opravdu není potřebné. I v těchto případech však stále platí doporučení nepotřebné uvozovky použít. Zachováme tím nejen konzistenci kódu, ale také se vyvarujeme neúmyslnému vynechání uvozovek na špatném místě.

Quotování expanzí není potřebné v následujících případech.

- Přiřazení expanze do proměnné (nutnost quotování uvnitř náhrady příkazu stále platí)
- Uvnitř klíčového slova `[[` (s výjimkou pravé strany při použití operátorů `=` a `=~`)
- Použití proměnné v konstrukci `case $foo in ...`
- Uvnitř aritmetické expanze

Známé hodnoty
Někteří programátoři argumentují, že navíc není potřebné chránit expanze známých hodnot. Jedná se například o návratovou hodnotu programu nebo výsledek aritmetické expanze. Word-splitting se však řídí proměnnou `IFS`, která může jako oddělovač obsahovat i číslice. Bezpečný skript tedy chrání i tyto expanze.

```
IFS=0 number=205

if [ $number -eq 100 ]; then
   printf 'Správný výsledek\n'
fi
# $number se rozdělí na dvě slova a kód skončí chybou
```

Snippet 12: Nestandardní hodnota IFS

Here-string
Od verze Bashe 4.3 bylo v oficiálním manuálu také uvedeno, že ani proměnné použité v here-stringu nepodléhají word-splittingu. I přes toto tvrzení však proměnné v here-stringu podléhaly word-splittingu až do verze 4.4, kde toto chování bylo opraveno. Tento příklad je dalším důvodem, proč chránit všechny expanze a zároveň upozorněním brát i oficiální zdroje s rezervou.

```
foo='a   b'
cat <<< $foo
# Verze 4.3 vypíše 'a b', verze 4.4 a novější vrátí 'a   b'
```

Snippet 13: Word-splitting proměnné v here-stringu

Kdy je quotování nežádoucí
Quotování je korektně nežádoucí hned v několika situacích. Důraz je však kladen na slovo korektně, jelikož nechránění expanzí je často zneužíváno pro nesprávné praktiky.

Porovnání řetězců
První z nich je uložení shellovského vzoru nebo regulárního výrazu do proměnné, která je následně použita s Bash funkcionalitou pro porovnání řetězců. Jinak řečeno při použití klíčového slova `case`, na pravé straně operátorů `=` a `=~` uvnitř klíčového slova `[[` nebo uvnitř manipulací při expanzi proměnných, které shellovské vzory podporují. Toto chování se může na první pohled zdát zbytečné, ale například uložení regulárního výrazu do proměnné, která je následně použita na pravé straně operátoru `=~`, nám ušetří starosti s leckdy složitým ošetřováním speciálních znaků.

Pokud jsou tyto proměnné chráněné uvozovkami, mají doslovný význam a nejsou jako shellovské vzory nebo regulární výrazy rozpoznány.

```
text='abca' reg_vyraz='a$'
# Test vrátí úspěch
[[ "$text" =~ $reg_vyraz ]]
# $reg_vyraz je brán doslovně, test vrátí neúspěch
[[ "$text" =~ "$reg_vyraz" ]]

text='abca' shell_vzor='?'
# Vypíše 'bca, abca'
printf '%s, %s\n' "${foo#$vzor}" "${foo#"$vzor"}"
```

Snippet 14: Korektně nežádoucí quotování

Here-dokument
Druhým případem je použití expanzí uvnitř here-dokumentu, kde se word-splitting ani náhrada souborů neprovádí. Zároveň se neprovádí ani žádné odstranění syntaktických uvozovek a uvozovky okolo proměnných jsou tedy ve výsledku doslovné.

```
# Vypíše '"ab   ca"'
cat << eof
"$(printf 'ab   ca\n')"
eof
```

Snippet 15: Quotování v here-dokumentu

Předání argumentu při jeho existenci
Posledním dosti specifickým případem je předání argumentu programu, pouze pokud byl tento argument nastaven. Použijeme jednu z modifikací při expanzi proměnné, `${promenna+slovo}`. Tato modifikace nic nevrací, pokud je *proměnná* nenastavena. Pokud nastavena je, vrací hodnotu *slova*. Při použití uvozovek by se ale vrácené „nic“ u nenastavené proměnné změnilo na prázdný

řetězec, který by byl programu předán jako argument. Z tohoto důvodu v této specifické situaci quotování nechceme. Nesmíme však opomenout správně quotovat *slovo*.

```
text='abca'; unset arg
f() { printf 'Počet argumentů: %d\n' "$#"; }

# Vypíše 'Počet argumentů: 2'
f ${text+"$text"} ${arg+"$arg"} "${arg+"$arg"}"
```

Snippet 16: Předání argumentu pouze při jeho existenci

Nesprávné použití word-splittingu

Posledním značně nekorektním případem je vyžadování, aby expanze opravdu word-splittingu podléhala. Programátoři této techniky zneužívají v mnoha situacích, například používání listů, uložení řádků výstupu programu do pole, iterování přes řádky souboru, iterování přes výstup programu apod. Důležitým upozorněním je, že tyto postupy **nikdy** nebudou spolehlivé. Bash poskytuje pro podobné úkony lepší a spolehlivá řešení, jako využívání polí, builtinu `mapfile` nebo builtinu `read` a `while` cyklu.

Použití proměnné IFS

Hodnoty proměnné `IFS` slouží jako oddělovače při word-splittingu. Hlavním doporučením této podkapitoly však bylo word-splitting vůbec nepoužívat. Proměnnou `IFS` se i přesto má cenu zabývat, jelikož se Bashem používá i v jiných případech. Za zmínku stojí její použití v polích a příkazu `read`.

Pole

Pokud expandujeme pole použitím „indexu“ `*`, všechny prvky pole jsou spojeny v jeden řetězec pomocí prvního znaku v proměnné `IFS`. Příkladem použití je vytvoření vizuálně přívětivého listu všech prvků daného pole.

```
declare -a pole=(aaa bbb ccc)

# Vypíše 'aaa,bbb,ccc'
(IFS=,; printf '%s\n' "${pole[*]}")
```

Snippet 17: Proměnná IFS a expanze pole

Příkaz read

`read` je shellovský builtin, který načte jednu řádku vstupu a rozdělí jí podle pravidel word-splittingu (náhrada souborů se neprovádí). Výsledná slova se přiřadí do proměnných s názvy určenými argumenty příkazu `read`. Pokud poskytneme méně názvů proměnných než je výsledných slov, `read` všechny zbývající slova přiřadí do posledního názvu, **včetně oddělovačů!**

`read` můžeme použít jak pro bezpečné načtení jednotlivých položek odděleného listu, tak i pro načtení celých řádků. Načtení celého řádku realizujeme předáním pouze jednoho názvu proměnné. `read` všechny slova přiřadí do této proměnné, včetně oddělovačů. I zde musím dbát obezřetnosti, jelikož chování splittingu je nevyzpytatelné a pokud hodnota `IFS` obsahuje nějaké bílé znaky, budou ze začátku i konce řádku odstraněny. Nejbezpečnější je tedy hodnotu `IFS` nastavit na prázdný řetězec, aby se žádný word-splitting neprováděl.

```
list='111,222'
# Proměnná $a obsahuje '111', Proměnná $b obsahuje '222'
IFS=, read -r a b <<< "$list"

# Proměnná $radek obsahuje první řádek souboru 'soubor'
IFS= read -r radek < soubor
```

Snippet 18: Proměnná IFS a builtin read

Posledním doporučením ohledně `IFS` a obecně jakékoli proměnné nebo příkazu, který mění chování shellu, je měnit toto nastavení na co možná nejkratší dobu. Vyhneme se tak do budoucna problémům s kódem, který s podobným nastavením nepočítá. Programátor může počítat s defaultním nastavením hodnoty `IFS`, jejíž změna může ovlivnit například doposud fungující příkazy `read`. V Bashi existuje možnost nastavit proměnnou do prostředí pouze následujícího programu a to tak, že přiřazení do proměnné použijeme před názvem programu. Další možností, jak nezměnit nastavení aktuálního prostředí, je použití subshellu.

```
# IFS je přenastavena pouze pro builtin read
IFS=, read -r radek <<< 'text'

# Změny IFS provedené v subshellu se po návratu ztratí
(IFS=,; read -r radek <<< 'text')
```

Snippet 19: Změna proměnných na omezenou dobu

3.3.2.2 Metaznaky

Word-splitting se provádí pouze na výsledcích již zmíněných expanzí. Z tohoto důvodu se při použití obyčejných řetězců nemusíme zabývat obsahem proměnné `IFS`. To však neznamená, že řetězce nepodléhají pravidlům quotování. Shell sám o sobě obsahuje celou řadu speciálních znaků, tzv. metaznaků, které se speciálně interpretují. Pokud nechceme, aby shell takové znaky interpretoval, musíme mu to sdělit za pomocí technik quotování.

Obecným doporučením je používat jednoduché uvozovky na **všechny** řetězce, které se mají brát doslovně a dvojité uvozovky na řetězce, kde vyžadujeme náhradu expanzí. Toto doporučení platí striktně i pro řetězce, které na první pohled speciální znaky neobsahují. Jedním příkladem důležitosti tohoto doporučení z praxe je instalační skript Nvidia driveru Bumblebee z roku 2011 [9], který kvůli překlepu mazal `/usr` složky uživatelů. Tento skript obsahoval následující řádek:

```
rm -rf /usr /lib/nvidia-current/xorg/xorg
```

Snippet 20: Nvidia Bumblebee překlep

Přidaná mezera za `/usr` je očividným překlepem, který měl za následek, že příkaz `rm` obdržel dvě složky ke smazání. Pokud by celá cesta byla uvnitř uvozovek, byla by považována za jeden argument a příkaz `rm` by skončil chybou při pokusu odstranit tuto neexistující cestu. Problém nejsou pouze bílé znaky, ale i ostatní znaky jako například `&`, který je běžný v hypertextových odkazech:

```
curl -o output http://example.com/page?fruit=apple&color=green
```

Snippet 21: Hypertextový odkaz se speciálním znakem

3.3.3 Zpracování souborů

Předchozí sekce do hloubky popsala nejčastější chybu při skriptování v shellu. Následující sekce se zabývá druhou nejčastější chybou, a to nesprávným zpracováním souborů.

Běžnou nekorektní praktikou je zpracovávání výstupu příkazu `ls`, který je určen pouze pro zobrazení uživateli, nikoli pro další zpracování. Nejhorší variantou je **iterování** přes náhradu příkazu `ls` nebo `find` pomocí `for` cyklu. Tato metoda s sebou totiž nese i problémy spojené s word-splittingem. Lepší metodou je použití příkazu `ls` nebo `find` pro vypsání seznamu souborů oddělených novými řádky, který následně zpracujeme například pomocí `while` cyklu. Problémem této metody však je, že v Unixu názvy souborů mohou obsahovat všechny znaky (včetně nových řádků) kromě **null bytu**. Naštěstí existuje několik způsobů, jak spolehlivě zpracovávat i takové soubory, které obsahují neobvyklé znaky jako nový řádek.

3.3.3.1 Expanze souborů

Nejzákladnější doporučenou metodou pro zpracování souborů je shellovská náhrada souborů. Výhodou je fakt, že po expanzi souborů se neprovádí žádný

word-splitting, a tak jednotlivá expandovaná slova (soubory) mohou obsahovat i speciální znaky, včetně problémového nového řádku. Vypnout náhradu souborů lze pomocí příkazu `set -f`. Základní shellovské vzory jsou velmi jednoduché a omezené. Rozeznávají pouze tři speciální znaky.

1. `*` – odpovídá libovolnému řetězci
2. `?` – odpovídá jednomu libovolnému znaku
3. `[znaky]` – odpovídá jednomu libovolnému znaku v závorkách

Rozšířená funkcionalita
Jedním z důvodů, proč se někteří uživatelé náhradě souborů vyvarují, je existence mnoha omezení. Obsahuje příliš jednoduché vzory, není rekurzivní, nepracuje se skrytými soubory, neumí zpracovávat metadata, nezvládne ignorovat velikost písma apod. Ani toto však není důvodem pro používání nespolehlivého zpracovávání příkazu `ls`, jelikož Bash poskytuje mnoho nastavení, kterým se dá chování náhrady souborů modifikovat a spoustu omezení tím eliminovat, a proto nadále představuje jednu z doporučených metod.

1. Použití tečky před vzorem způsobí expanzi skrytých souborů, například `/moje/cesta/.*`.
2. Nastavení `shopt -s dotglob` způsobí, že náhrada souborů pracuje se skrytými soubory defaultně.
3. Po použití nastavení `shopt -s globstar` se dva po sobě jdoucí znaky `**` expandují rekurzivně i na všechny podadresáře.
4. Nastavení `shopt -s nocaseglob`, podobně jako `nocasematch`, ignoruje velikost písma při náhradě souborů.
5. `shopt -s extglob` umožní využití rozšířených vzorů pro náhradu souborů.

Ukázka a možné problémy

```
# Příklad 1
shopt -s nullglob
for soubor in ./*.txt; do
   mv "$soubor" adresar
done

# Příklad 2
mv -t adresar -- *.txt
```

Snippet 22: Shellovská náhrada souborů

Prvním problémem je fakt, že shellovský vzor zůstane doslovným pokud neexistují žádné soubory, které by mu odpovídaly. V prvním příkladě by následkem neexistence takových souborů byla jedna iterace `for` cyklu s daným shellovským vzorem v proměnné `$soubor`. Toto chování je očividně nežádoucí a Bash poskytuje dvě možná nastavení. `shopt -s nullgob`, které shellovský vzor odstraní, pokud neodpovídá žádným souborům a nastavení `shopt -s failglob`, které ve stejné situaci vyvolá chybu.

Další problém není specifický pouze pro náhradu souboru a opět se týká nepříjemných znaků v názvech souborů, které mimo jiné mohou začínat znakem `-`. Tento znak mnoho programů interpretuje jako začátek přepínače. Možnostmi, jak se interpretování souboru jako přepínače chránit, jsou použití konvenčního značení `--` pro konec přepínačů (které však nemusí všechny programy implementovat) nebo použití `./` na začátku názvu souboru.

Posledním pozorovaným problémem je náhrada na příliš mnoho souborů. V takovém případě by druhý příklad skončil chybovou hláškou pro příliš dlouhý seznam argumentů. První příklad by chybou neskončil, jelikož shellovské builtiny toto omezení nemají. Řešením tedy může být použití builtinu, předání argumentů v menších dávkách nebo využití programů `find` a `xargs`, které umí automaticky zjistit maximální počet argumentů a volání programu odpovídajícím způsobem optimalizovat.

3.3.3.2 Příkaz find

Příkaz `find` je velice mocný nástroj, který například umí zpracovávat soubory rekurzivně, používat shellovské vzory i regulární výrazy nebo vybírat soubory na základě velkého množství metadat. Speciálně pro zpracování metadat je `find` preferovaným řešením. `find` je bezpečným způsobem, jak zpracovávat soubory, jelikož obsahuje interní funkcionalitu pro spuštění programu s vyhledanými soubory a speciální znaky důsledkem nepředstavují problém. Toto platí i pro soubory obsahující `-` na začátku svého názvu, jelikož `find` defaultně používá doporučenou `./` notaci. Je důležité si však uvědomit různá omezení. Například fakt, že pouze Bash umí používat funkce, a tak není možné, aby `find` sám o sobě bez Bashe zpracovával soubory pomocí funkcí.

Základním použitím je `find -exec program {} ;`, které spustí program na každý jednotlivý soubor. Pokud nepotřebujeme, aby byl program spouštěn na každý soubor zvlášť, je mnohem efektivnější využít možnosti spustit program s vícero soubory. K tomu slouží konstrukce `find -exec program {} +`, která program spustí s maximálním počtem souborů a automaticky použije program vícekrát pouze v případě, že je souborů více jak povolené maximum argumentů, což je oproti běžné náhradě souborů dalším vylepšením.

```
find . -maxdepth 1 -name "*.txt" -exec mv -t slozka {} +
```

Snippet 23: Program find

`find` je velice užitečným a složitým nástrojem. Pokud však soubory potřebujeme zpracovávat jinými programy, neobejdeme se bez použití **null bytu**.

3.3.3.3 Null byte

Null byte je speciální znak, který se v názvu souboru nesmí vyskytovat. To z null bytu dělá výborného kandidáta pro použití jako oddělovače nejen souborů, ale obecně položek, které mohou obsahovat nové řádky. Nemálo GNU utilit umí s null bytem pracovat právě z tohoto důvodu. Příkladem je program `sort -z` nebo užitečný `xargs -0`, který je podobný konstrukci `find -exec`.

Vygenerovat list souborů oddělených null bytem zvládne například konstrukce `find -print0`. Popřípadě můžeme použít běžnou notaci null bytu uvnitř formátovacího řetězce příkazu `printf`.

```
find . -type f -print0 | sort -z
printf '%s\0' "${pole[@]}" | sort -z
```

Snippet 24: Null byte

3.3.4 Validace

Předchozí sekce popisují ty vůbec nejzákladnější principy bezpečného skriptování. Validace vstupů a návratových kódů je však neméně důležitá. Sebelepší quotování neochrání skript před obdržením špatného vstupu od uživatele.

3.3.4.1 Validace dat

Pokud náš kód vyžaduje práci s daty v určitém formátu, je potřeba tato data vždy validovat. Data můžeme rozdělit na dvě skupiny.

1. Data od uživatelů
2. Data od programů

Data od uživatelů jsou na nesprávnost náchylnější. Pokud od uživatele požadujeme jako vstup celé číslo, musíme pomocí například Bash funkcí pro porovnání řetězců ověřit, zda dodaný vstup tato kritéria opravdu splňuje. Pokud požadujeme jako vstup název souboru, který budeme následně zpracovávat, musíme se ujistit, že dodaný název souboru je opravdu existujícím souborem, ke kterému máme přístupová práva.

Dalším důležitým bodem je validovat, zda nějaká data uživatel nebo program vůbec poskytl, případně validovat jejich neprázdnost. Pokud nějaký příkaz skončí chybou, náhrada příkazu se může jednoduše stát prázdným řetězcem. Příkladem tohoto doporučení z praxe byla chyba ve Steam klientu společnosti Valve z roku 2015 [10], který od kořenového adresáře rekurzivně mazal všechna data vlastněná uživatelem:

```
rm -rf "$STEAMROOT/"*
```

Snippet 25: Chyba Steam klienta

Tento řádek se očividně řídí osvědčenými postupy quotování i zpracování souborů. Při inicializaci proměnné `$STEAMROOT` ale vznikla neočekávaná chyba a proměnná obsahovala pouze prázdný řetězec. Jelikož v kódu nikde neproběhla validace této proměnné, skript spustil příkaz `rm -rf /*`, který se pustil do mazání.

set -o unset

Příkaz `set` je dalším builtinem, kterým se dá měnit chování Bashe. `unset` je nastavení, které způsobí ukončení skriptu, pokud použijeme neinicializovanou proměnnou. Jedná se o pokus přiblížit Bash vyšším programovacím jazykům. Jedním z problémů však je vyvolání chyby i u běžně využívaného použití prázdného pole. Toto chování bylo změněno od verze 4.4, přesto i kvůli jiným důvodům nastavení `unset` používat nedoporučuji.

3.3.4.2 Zpracování chyb

Pokud nějaká část kódu jakýmkoli způsobem závisí na úspěšném provedení předchozího programu, je stejně důležité validovat úspěšnost tohoto programu. Běžným příkladem je použití builtinu `cd` na změnu aktuálního adresáře, kde následně náš skript provádí další změny. Pokud však `cd` selže a aktuální adresář nezmění, budeme veškeré změny provádět v původním adresáři, což může mít fatální následky. Kontrolovat úspěšnost programu můžeme například pomocí známého klíčového slova `if`.

set -o errexit

`errexit` je nastavení, které ukončí skript, pokud libovolný příkaz skončí chybou. Opět se jedná o snahu přiblížit Bash vyšším programovacím jazykům. Jelikož ale například nechceme, aby program, který testujeme na úspěšnost manuálně pomocí klíčového slova `if`, ukončil v případě neúspěchu skript, je implementována celá řada pravidel, kdy se toto nastavení neuplatňuje. Tato pravidla mohou být velmi matoucí. Zároveň `errexit` ukončí skript při použití některých běžných konstrukcí. Ze zmíněných důvodu je dopuručeno používat manuální kontrolu a na toto nastavení nespoléhat.

```
set -o errexit
pocet=0
((pocet++))
# Návratový kód ((pocet++)) je 1 a errexit zde ukončí skript
```

Snippet 26: Neočekávané ukončení skriptu nastavením errexit

set -o pipefail
Jako návratový kód programů, které jsou propojeny pomocí rour, je použit návratový kód posledního programu. Pokud poslední program skončí úspěšně, bude komplikované odhalit neúspěch jiného z těchto programů. Bash poskytuje nastavení `pipefail`, při kterém se jako návratový kód použije kód nejposlednějšího selhaného programu, nebo 0, pokud všechny programy proběhly úspěšně. Toto nastavení ulehčuje práci s návratovými kódy a je doporučeno.

3.3.4.3 Podvržení dat

K podvržení dat dochází v případě, že se použitý vstup pro nějaký program nesprávně interpretuje jako část kódu. Hlavním příkladem je použití neošetřeného uživatelského vstupu s builtinem `eval`. `eval` v podstatě spojí obdržené argumenty, které následně interpretuje jako příkaz pomocí kroků zpracování příkazové řádky. Doporučením je se `evalu` snažit úplně vyvarovat, jelikož představuje řadu bezpečnostních rizik. V mnoha případech se dá nahradit použitím správných konstrukcí, např. asociativním polem. Pokud `eval` přesto z nějakého důvodu opravdu použít potřebujeme, je nutné vstup ošetřit například pomocí příkazu `printf %q`. Direktiva `%q` vypíše odpovídající argument ve formátu, který může být znovu použit jako shellovský vstup.

Dalším běžným příkladem je použití shellovské proměnné uvnitř `sedu` nebo `awku`. Pokud proměnnou do těchto programů vložíme přímo, budou se interpretovat jako součást kódu. U `sedu` je jedinou možností ošetřit všechny znaky, které pro `sed` mají speciální význam. Jiné programy poskytují možnosti, jak shellovské proměnné bezpečně předat jako interní proměnnou daného programu. Příkladem je přepínač `-v` programu `awk`.

```
text='abca";system("uname");print "'

# Nesprávně interpretuje vstup a vypíše 'abca\nLinux\n'
awk 'BEGIN{print "'"$text"'"}'

# Vypíše doslovně 'abca";system("uname");print "'
awk -v text="$text" 'BEGIN{print text}'
```

Snippet 27: Předání proměnných externímu programu

3.3.5 Přenositelnost a zastaralá syntax

Tato práce se zabývá programovacím jazykem Bash. Doporučuje tedy využití spousty konstrukcí, které jsou unikátní pouze pro Bash shell. Za zmínku však stojí standard POSIX, kterého se alespoň do jisté míry mnoho shellů snaží držet. Pokud máme nějaký kus kódu, který je bez ztráty funkčnosti nebo jednoduchosti triviální změnit tak, aby odpovídal POSIX definici, zajistíme si snazší potenciální migraci skriptu pro jiný shell interpret. Příkladem je použití POSIX operátoru `=` pro porovnání řetězců místo identického `==`, který je podporovaným rozšířením Bashe. Toto doporučení však neplatí, pokud znamená použití zastaralých konstrukcí. Vybrané zastaralé nebo špatně přenositelné konstrukce zmiňuje následující list.

- Náhrada příkazů `` `příkaz` `` je nepřehledná syntax, která se složitě vnořuje a obsahuje nelogické zpracování zpětných lomítek. Z tohoto důvodu byla nahrazena preferovanou syntaxí `$(příkaz)`.
- Konstrukce `[`, kterou nahradilo pokročilejší klíčové slovo `[[`.
- Příkaz `echo` se může rozdílně a nespolehlivě chovat implementaci od implementace. Z tohoto důvodu je preferován mnohem spolehlivější a mocnější příkaz `printf`.
- V neposlední řadě existuje speciální případ zastaralé syntaxe, která již ani není obsažena v oficiálním manuálu i přesto, že stále funguje. Jedná se o aritmetickou expanzi `$[výraz]`, kterou nahradila preferovaná syntax `$((výraz))`.

3.4 Skript

Předchozí podkapitoly se zabývaly analýzou postupů, jak skriptovat efektivněji a bezpečněji. Tyto postupy jsou aplikovatelné nejen při psaní skriptu, ale i při interaktivním použitím Bashe. V následující podkapitole se budeme zabývat náležitostmi skriptu jako takového.

3.4.1 Uživatelská přívětivost

Jednoduché, přehledné a zároveň adekvátně dokumentované použití našeho skriptu, je pro uživatele stejně důležité jako jeho funkcionalita. Skript i s tou nejlepší funkcionalitou je bezcenný v případě, že je pro uživatele nepoužitelný.

3.4.1.1 Uživatelské rozhraní

Základním rozhodnutím programátora je výběr uživatelského rozhraní. Příklady možných uživatelských rozhraní zmiňuje následující list.

1. Grafická rozhraní GUI a TUI
2. Interaktivní příkaz v terminálu
3. Pouze příkaz s přepínači

Grafická rozhraní jsou vizuálně nejpřívětivější. Neumožňují však, stejně jako interaktivní příkaz, skript automatizovat. Doporučením je psát skripty, které jsou flexibilní a je možno je automatizovat. Pokud přijde požadavek grafické rozhraní přidat, můžeme například jednoduše vytvořit nový skript, který pro náš zmíněný skript poskytuje frontend.

3.4.1.2 Dokumentace

Komplikovanější skripty mohou obsahovat velké množství funkcionality, přepínačů apod. Dokumentace je důležitá například pro seznámí se skriptem nebo v případě, že zapomeneme přesné použití některého z přepínačů. V takovém případě slouží dokumentace jako nejlepší reference. Je tedy klíčové se skriptem dokumentaci v nějaké formě poskytovat. Rozlišujeme tři základní typy dokumentace.

1. Online dokumentace je ideální pro skripty, které jsou dostupné online, rozsáhlé a cílené pro mnoho uživatelů. Dokumentace může být například ve formátu HTML nebo PDF.
2. Typické Unix manuálové stránky jsou offline dokumentací v jednoduchém formátu. Výhodou je jejich dostupnost na strojích bez grafického prostředí. Manuálové stránky jsou konvenční a doporučenou metodou dokumentace.
3. Konvenční přepínače `-h` a `--help` jsou další dokumentací, která je dostupná z příkazové řádky bez grafického prostředí. Většinou obsahuje nejdůležitější podmnožinu manuálové stránky. Tento výtah manuálové stránky nemusí být zobrazen pouze při použití zmíněných přepínačů, ale i jako pomoc uživateli, který například použije neexistující přepínač. Tento typ dokumentace je též doporučeno implementovat.

3.4.1.3 Readline knihovna

Bash využívá pro editování příkazové řádky Readline knihovnu. Editováním příkazové řádky se myslí doplňování slov v interaktivním režimu pomocí tabulátoru, které uživateli výrazně ulehčí a zpříjemní používání skriptu. Doplňování je **programovatelné** za pomoci builtinu `complete`. Můžeme například naprogramovat doplňování názvů implementovaných přepínačů po použití tabulátoru na znaku přepínače `-` či doplňování povolených argumentů, které daný přepínač vyžaduje. Důsledkem je tedy i snížená šance překlepu nebo použití nekorektního argumentu uživatelem.

Běžnou metodou, jak doplňování určitého skriptu naprogramovat, je vytvoření funkce, která do speciálního pole `COMPREPLY` vloží všechna možná doplnění pro aktuálně doplňované slovo. Pokud tuto funkci skriptu přiřadíme pomocí `complete -F funkce skript`, bude se Readline knihovnou volat po každém použití tabulátoru na části příkazové řádky tohoto skriptu.

3.4.1.4 Oznámení stavu

Náš skript má k dispozici dvě základní lokace pro předání výstupu: standardní výstup a **chybový výstup**. Oba typy výstupu mohou být přesměrovány zvlášť.

Standardní výstup je určený pro předání **vyžadovaných** dat uživateli. Tento výstup je často uložen do souboru nebo předán rourou pro zpracování dalšímu programu. Jelikož se jedná o data, která mohou být dále zpracovávána, standardní výstup není místo pro chybové hlášky, diagnostiku nebo oznámení stavu/postupu. Podobné zprávy patří na chybový výstup.

Vybraná pravidla chybového výstupu:

- Každou chybu okamžitě vypsat na chybový výstup.
- Diagnostické zprávy a oznámení stavu programu vypisovat pouze při zapnutém verbose režimu přepínačem `-v`. Verbose režim může obsahovat vícero úrovní.
- Implementovat přepínač `-s` pro vypnutí veškerého výstupu.

3.4.2 Přehlednost kódu

Přehlednost kódu je stejně důležitá jako jeho korektnost. Nepřehledný kód značně znesnadňuje běžnou údržbu jakéhokoli programu. Na podobném kódu se špatně spolupracuje s jinými programátory, je špatně čitelný, špatně se rozšiřuje a v případě chyb složitě debugguje. Základním postupem je strukturovat náš kód do logických bloků pomocí funkcí. Další osvědčené postupy pro zvýšení přehlednosti existují, avšak jsou v mnoha případech subjektivní. Ať použijeme postupy jakékoli, je nejdůležitější je používat **konzistentně**.

3.4.2.1 Formátování

Každý skript by měl dodržovat základní pravidla formátování. Pokud například dodržujeme správné odsazování kódu, na první pohled vidíme strukturu programu, což může výrazně pomoci při hledání chyb apod. Základními vybranými pravidly jsou:

- Odsazovat kód 3 mezerami.

- Používat řádky o maximální délce okolo 80 znaků.
- Používat prázdné řádky pro oddělení logických částí programu.
- Preferovat strukturovaný víceřádkový zápis složených příkazů jako `for` před zápisem jednořádkovým.
- Vyvarovat se příliš hluboce vnořenému kódu. Takový kód je sám o sobě pravděpodobně nesprávný a lze přepsat efektivněji.

3.4.2.2 Komentáře

Dalším důležitým bodem jsou komentáře v kódu. Zajímavé, složité, nebo na první pohled nejasné části kódu, by se měly adekvátně komentovat. Osvědčeným postupem je také na začátek přidat popis našeho programu a komentovat všechny deklarované funkce. Zároveň je důležité se vyvarovat přehnanému komentování. Pokud v kódu používáme deklaraci proměnné `$pocet_argumentu`, je kontraproduktivní tento řádek komentovat. Zbytečné komentáře přehlednost kódu naopak zhoršují.

3.4.2.3 Jmenné konvence

Základním osvědčeným postupem nejen v Bashi je používání deskriptivních názvů. Již z názvu by na první pohled mělo být snadné určit, co daná proměnná obsahuje či co daná funkce vykonává. Pokud například deklarujeme proměnnou, která obsahuje počet argumentů, je vhodné jí odpovídajícím způsobem pojmenovat jako `$pocet_argumentu`.

Vybrané konvence, z velké části subjektivní, popisuje následující list:

- Používat pouze malá písmena pro názvy proměnných i funkcí. Proměnné složené pouze z velkých písmen mohou kolidovat s proměnnými prostředí nebo interními proměnnými shellu. Příkladem je proměnná `$PATH`.
- Názvy funkcí by neměly přepisovat známe programy jako `read`.
- Používat pouze písmena anglické abecedy a číslice.
- Slova v názvech oddělovat pomocí podtržítek.
- Deklarovat konstanty jako readonly pomocí builtinu `declare -r`.
- Omezit proměnné deklarované ve funkci pomocí builtinu `local`.

Často je také jako doporučení uváděno používat složené závorky pro referencování proměnných, například `${pocet}`. Toto doporučení je dle mého názoru příliš striktní a nedůležité. Výhodou ale může být jednodušší přidání modifikace při expanzi nebo vyvarování se chyb, kdy se část textu za proměnnou stane součástí názvu této proměnné.

3.4.2.4 Deskriptivní programování

Pojmem „deskriptivní programování“ je myšleno programování, které upřednostňuje delší a deskriptivní konstrukce před kratším a často v důsledku chybovým kódem.

Příkladem může být i pouhé doporučení upřednostňovat dlouhé přepínače. V interaktivním režimu je používání krátkých přepínačů opodstatněné, při psaní skriptu je však lepším postupem použít místo přepínače `-t` mnohem přehlednější variantu `--target-directory`. Další ukázkou kratšího a nejasného kódu je operátor `-n` klíčového slova `[[`, který vrací úspěch v případě, že argument není prázdný řetězec. Stejného výsledku dosáhneme sice i kratším kódem bez použití `-n` operátoru, avšak opět za cenu ztráty přehlednosti.

Úplně nejlepším řešením je v podobných případech pro komplexní nebo na první pohled nejasný kód deklarovat funkci s deskriptivním názvem a popisem a to i v případě, že se jedná o jednořádkový kód. Demonstrací tohoto doporučení je následující snippet.

```
# Na první pohled nejasný kód
[[ "$text" ]] || printf 'Text je prázdný\n' >&2

# Preferovaný a na první pohled jasný kód
je_prazdny() {
   [[ -z "$text" ]]
}

if je_prazdny "$text"; then
   printf 'Text je prázdný\n' >&2
fi
```

Snippet 28: Výhody deskriptivního programování

Problémem není pouze ztráta přehlednosti, ale také chybovost. Některé konstrukce, kterými si programátoři snaží ušetřit psaní, jsou často nesprávné. Nejčastějším příkladem je použití zápisu `podminka && prikaz_if || prikaz_else` místo řídící konstrukce `if` a `else`. Tyto dva zápisy nejsou zaměnitelné, jak si mnoho programátorů myslí. V prvním případě se totiž `prikaz_else` nesprávně provede i pokud `podminka` byla úspěšná, ale `prikaz_if` skončil chybou.

3.4.3 Struktura skriptu

Velké procento skriptů se podobá základní strukturou a implementovanou funkcionalitou. Příkladem je zpracování přepínačů nebo uvedení tzv. shebangu. Následující sekce se zabývají vybranými náležitostmi většiny skriptů.

3.4.3.1 Shebang

Než začneme přidávat vlastní kód, je důležité na začátku skriptu uvést tzv. shebang. Shebang je direktiva poskytující systému absolutní cestu k interpretu, který se má pro kód použít. Pokud používáme Bash skript pouze pro spuštění `awk` programu, je efektivnější vytvořit skript obsahující `awk` kód a jako shebang uvést `awk` interpret. Vyhneme se tak zbytečnému „prostředníkovi“.

Příkladem používaného shebangu pro Bash je `#!/bin/bash`. Problémem tohoto příkladu je použití skriptu na systému, kde je daný interpret umístěn na nestandardním místě. Z tohoto důvodu je doporučené používat jako shebang `#!/usr/bin/env bash`, který prohledá adresáře z proměnné `$PATH` a použije první výskyt programu `bash`. V první variantě shebangu navíc můžeme nastavit chování shellu uvedením přepínačů, které akceptuje builtin `set`. Doporučuje se však tato nastavení provádět pomocí `setu` až v samotném kódu, jelikož se shebang v některých případech ignoruje a nastavení by se neuplatnilo. V druhé variantě shebangu tato možnost ani **neexistuje**. Shebang se ignoruje v případě, že je skript spuštěn pomocí příkazu `source` nebo je na příkazové řádce přímo spuštěn pomocí zvoleného interpretu, například `bash nazev_skriptu`.

3.4.3.2 Zpracování přepínačů

Běžné programy poskytují pro uživatele celou řadu přepínačů, kterými uživatel mění chování programu. Existují tři základní postupy, jak přepínače zpracovávat.

1. Manuální zpracování pomocí cyklu poskytuje největší úroveň volnosti a přenositelnosti. Nevyžaduje žádné specializované programy, což na druhou stranu znamená mnohem složitější implementaci. Manuálním zpracováním můžeme podporovat krátké i dlouhé přepínače.

2. Použití builtinu `getopts` je velmi jednoduché na implementaci a ušetří při vytváření skriptu mnoho práce oproti manuálnímu zpracování. Samozřejmostí je rozpoznání konvenčního konce přepínačů `--` nebo seskupování krátkých přepínačů. Největší nevýhodou `getoptsu` je podpora **pouze krátkých přepínačů**.

3. Třetí metodou je použití programu `getopt`. Starší verze `getoptu` měla řadu problémů, a proto tato metoda nebyla doporučována. Dnešní moderní systémy obsahují novou a vylepšenou verzi, která je nyní nejvhodnějším řešením pro zpracování jak krátkých, tak dlouhých přepínačů. Program `getopt` v podstatě přepínače normalizuje a umožní nám jednoduché manuální zpracování pomocí cyklu.

Ať použijeme metodu jakoukoli, je doporučeno implementovat dlouhé přepínače a dodržovat zaběhlé konvence, jako například podporu znaku `--` pro konec přepínačů. Názvy některých přepínačů také často napříč programy zastávají podobnou či přímo identickou funkci. Některé vybrané názvy zmiňuje následující seznam.

- `-v` pro zapnutí verbose režimu.
- `-h` pro zobrazení krátké pomocné dokumentace.
- `-z` pro podporu zpracování null bytu.
- apod.

3.4.3.3 Konfigurační soubor

Konfigurační soubor je obdobou přepínačů a slouží pro úpravu chování programu. Rozlišujeme dva základní způsoby, jak konfigurační soubor předat.

1. Konfigurační soubor je umístěn jako skrytý soubor ve **fixní** lokaci a zároveň s fixním názvem. Jedná se o způsob podobný konfiguračnímu souboru `.bashrc`.
2. Cestu ke konfiguračnímu souboru předáme pomocí implementovaného přepínače.

Samotný formát konfiguračního souboru záleží na programátorovi. Není však doporučeno používat formát, kde jednotlivými řádky jsou přiřazením do proměnných, která se aplikují pomocí příkazu `source soubor`. Tento způsob není bezpečný, jelikož dovoluje podvržení dat. Vhodným způsobem zpracování může být obyčejný `while` cyklus.

3.4.3.4 Dočasné soubory

Pokud skript využívá dočasné soubory, je vhodné pro jejich vytvoření použít program `mktemp`. `mktemp` vytvoří dočasný soubor nebo adresář bezpečně a garantuje unikátní název. Klíčové je udržování listu všech takových dočasných souborů a adresářů. V případě ukončení skriptu je tento list použit pro jejich smazání.

3.4.3.5 Závěr skriptu

Každý skript má při svém ukončení dvě základní povinnosti.

1. Uklidit po sobě vytvořené dočasné soubory a adresáře.
2. Vrátit odpovídající návratový kód.

Reagování na signály
Úklid dočasných souborů je triviální, pokud reagujeme na plánovaný exit, při kterém máme možnost soubory smazat. Program však může skončit i jinými, neplánovanými způsoby. Příkladem je obdržení signálu `SIGINT` použitím řídící sekvence `CTRL+C` od uživatele. Z tohoto důvodu existuje builtin `trap`, který na tyto signály umí reagovat a před ukončením skriptu provést dodaný kód.

Doporučeným způsobem je deklarování speciální úklidové funkce, kterou bude definovaný `trap` volat. Ne na všechny signály je možno reagovat. Příkladem je signál `SIGKILL`. Doporučenými signály, na které je vhodné reagovat, jsou `EXIT`, `SIGINT` a `SIGTERM`.

```
uklid() {
   local navratovy_kod="$?"
   # Mazání dočasných souborů a adresářů
   exit "$navratovy_kod"
}

trap 'uklid' EXIT SIGINT SIGTERM
```

Snippet 29: Reagování na signály pomocí builtinu trap

Návratový kód
Skript by při svém ukončení měl navrátit odpovídající návratový kód. Návratový kód je číslo od 0 do 255, které určuje, zdali daný program skončil úspěšně či neúspěšně. Návratový kód 0 signalizuje úspěch, kterékoli jiné číslo signalizuje neúspěch. Doporučením je tyto konvence dodržovat a uživateli oznámit úspěch pomocí návratového kódu 0, neúspěch pomocí návratového kódu 1 až 125. Kódy 126 a výše mohou totiž být použity samotným Bashem a není doporučeno je v našem skriptu používat.

Kapitola 4

Návrh

Následující kapitola se zabývá návrhem výsledného skriptu praktické části této práce. Kapitola shrnuje požadavky a základní strukturu skriptu.

4.1 Popis

Hlavním účelem výsledného skriptu je generování koster nových skriptů dle zadaných požadavků a zároveň demonstrování zdokumentovaných osvědčených postupů.

Některé prvky skriptů, jako například zpracování přepínačů, se často opakují a jejich tvorbu lze automatizovat. Pokud uživatel chce například zpracovávat přepínače, může požadované přepínače jednou definovat a program vygeneruje jejich zpracování, zároveň je přidá do usage funkce apod.

4.2 Požadavky

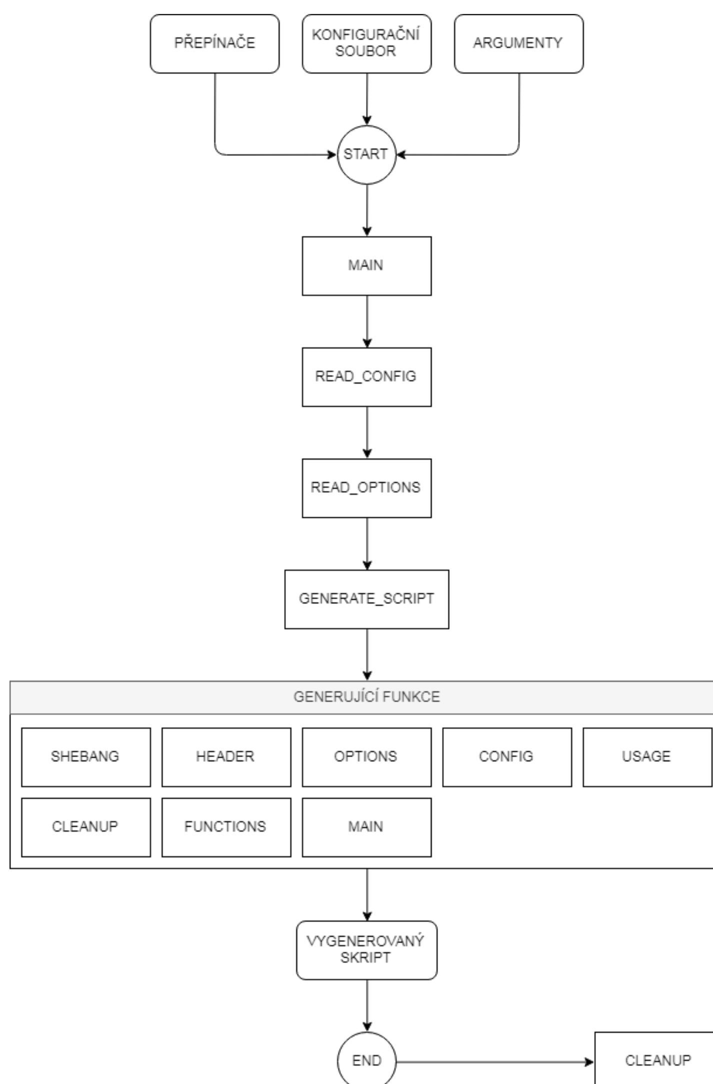
V této sekci si popíšeme základní funkční požadavky výsledného programu. Především se jedná o části skriptu, které by měl program zvládat generovat.

1. Možnost generovat shebang.
2. Možnost generovat hlavičku s popisem.
3. Možnost generovat nastavení shellu.
4. Možnost generovat zpracování přepínačů.
5. Možnost generovat zpracování konfiguračního souboru.
6. Možnost generovat „usage“ funkci.
7. Možnost generovat prostředky pro vytvoření dočasných souborů.

8. Možnost generovat funkcionalitu pro úklid dočasných souborů apod.
9. Možnost generovat balíčky pomocných funkcí.
10. Korektně lokalizovat použité proměnné a pole.
11. Možnost konfigurovat informace o skriptu, např. název skriptu, popis skriptu nebo jméno autora.

4.3 Struktura skriptu

Následující diagram znázorňuje základní strukturu skriptu.



Obrázek 4.1: Základní struktura skriptu

Skript, po uvážení, bude pouze příkaz bez grafického prostředí. Bude monolitický a nebude obsahovat funkcionalitu, která vyžaduje dodatečné soubory nebo instalaci, jako například manuálovou stránku, PDF dokumentaci nebo programmable completion. Účelem je co nejjednodušší přenositelnost a použití.

Skript na svém vstupu akceptuje nepovinné přepínače, konfigurační soubor a argumenty. Konfigurační soubor je zpracován jako první, jelikož přepínače mají prioritu a mohou následně nastavení konfiguračním souborem přepsat. Jakýkoli přepínač či konfigurační direktiva mohou být použity mnohonásobně. Zbylé argumenty na příkazové řádce jsou všechny adresáře, kam má být vygenerovaný skript uložen. Jejich zpracování se řídí následujícími pravidly:

- Pokud adresář neexistuje, skript se pokusí tento adresář vytvořit.
- Pokud adresář není zapisovatelný nebo nastane jiná neočekávaná chyba, skript tento adresář ignoruje.
- Pokud v adresáři existuje soubor se zvoleným jménem, skript k názvu přidá unikátní číselný suffix.
- Pokud není předán žádný validní adresář, skript vypíše kostru na standardní výstup.

Po zpracování konfiguračního souboru a přepínačů, skript spustí hlavní generující funkci, která následně volá jednotlivé funkce, které generují určitou část výsledné kostry. Shebang, usage, cleanup, základní nastavení a main se generují automaticky za předpokladu, že jejich generování není uživatelem potlačeno. Výstup je uložen podle pravidel viz výše. V neposlední řadě po sobě skript před ukončením uklidí dočasné soubory.

4.4 Přepínače

Program obsahuje řadu přepínačů, kterými může uživatel chování skriptu modifikovat. Důležité přepínače jsou ve formátu jak krátkých přepínačů, tak přepínačů dlouhých. Méně důležité přepínače jsou pouze ve formátu dlouhých přepínačů. Jedná se zejména o přepínače, které potlačují generování důležitých částí skriptu.

- **-o, --options, direktiva options**

  Způsobuje generování zpracování přepínačů. Pokud je tento přepínač použit, musí být nadefinován alespoň jeden validní přepínač, jinak skript skončí chybou.

- **-c, --config, direktiva config**

  Způsobuje generování zpracování konfiguračního souboru. Pokud je tento přepínač použit, musí být nadefinována alespoň jedna validní direktiva, jinak skript skončí chybou.

- **-f, --config-file**

  Umožňuje zvolit cestu ke konfiguračnímu souboru. Pokud je tento přepínač použit, defaultní konfigurační soubor se nezpracovává.

- **-p, --param, direktiva param**

  Přidání definice přepínačů a konfiguračních direktiv, které jsou použity při generování. Tato definice se skládá z následujících položek oddělených `:`.

  1. krátký přepínač
  2. dlouhý přepínač
  3. direktiva
  4. proměnná

     Pokud proměnná není vyplněna, přepínače a direktivy nemají argument. Pokud vyplněna je, argument je podporován a přiřazen do dané proměnné.
  5. popis

     Popis je využit při generování funkce „usage“ a generování hlavičky.
  6. atribut,atribut ...

     Pokud jsou vyplněny nějaké atributy, argument musí tyto podmínky splňovat, jinak není přiřazen. Povolené atributy:

     a) int – hodnota je celé číslo
     b) float – hodnota je desetinné číslo
     c) non-empty – hodnota není prázdná
     d) file-exists – hodnota je existující soubor
     e) non-empty-file - hodnota je neprázdný soubor
     f) file – hodnota je existující obyčejný soubor
     g) dir – hodnota je existující složka
     h) readable – hodnota je soubor se čtecími právy
     i) writable – hodnota je soubor se zapisovacími právy
     j) executable – hodnota je spustitelný soubor

  Alespoň jeden z identifikátorů musí být vyplněn a validní, jinak je definice ignorována. Příklad definice:

  `f:file:file-conf:file_var:Path to a file:readable,writable`

- **-k, --header, direktiva header**

  Způsobuje generování hlavičky s popisem.

- **-b, --function, direktiva function**

  Způsobuje generování balíčků pomocných funkcí. Dostupné balíčky jsou: math, variables, strings, files.

- **-a, --arguments, direktiva arguments**

  Definuje v generovaném skriptu pole s nezpracovanými argumenty.

- **--option-else, direktiva option-else**

  Pokud vygenerovaný přepínač nebo direktiva obsahují atributy, jsou pro tyto podmínky vygenerovány také else větve.

- **--output-config-delimiter, direktiva output-config-delimiter**

  Oddělovač direktivy a hodnoty ve vygenerovaném zpracování konfiguračního souboru. Defaultní hodnotou je mezera.

- **--output-config-file, direktiva output-config-file**

  Cesta ke konfiguračnímu souboru, který je zpracováván ve vygenerovaném kódu.

- **--extra-settings, direktiva extra-settings**

  Přidá do výsledného kódu populární, avšak nedoporučované nastavení shellu `set -e` a `set -u`.

- **-t, --temp-files, direktiva temp-files**

  Přidá prostředky pro jednoduché generování dočasných souborů.

- **-n, --name, direktiva name**

  Definuje název skriptu. Defaultní hodnotou je „script“.

- **-d, --description, direktiva description**

  Definuje krátký popis skriptu. Defaultní hodnotou je „Description“.

- **-u, --user, direktiva user**

  Definuje jméno autora skriptu.

- **-v, --verbose, direktiva verbose**

  Zvyšuje hodnotu verbose úrovně o 1.

- **-h, --help**

  Ignoruje ostatní přepínače a zobrazí usage.

- **--suppress-settings, direktiva suppress-settings**
  Potlačuje generování nastavení shellu.
- **--suppress-shebang, direktiva suppress-shebang**
  Potlačuje generování shebangu.
- **--suppress-usage, direktiva suppress-usage**
  Potlačuje generování funkce usage.
- **--suppress-rc**
  Potlačuje zpracovávání defaultního konfiguračního souboru.
- **--suppress-cleanup, direktiva suppress-cleanup**
  Potlačuje generování funkcionality pro úklid dočasných souborů.

Použití chybného přepínače skončí chybou a vypsáním usage.

4.5 Konfigurační soubor

Existují dva způsoby, jak skriptu předat konfigurační soubor.

1. Pomocí zmíněných přepínačů.
2. Uložením konfiguračního souboru v předem dané lokaci.

Pokud není konfigurační soubor předán přepínači, používá se soubor s názvem `.createsrc` v domovském adresáři uživatele. Při neexistenci tohoto souboru skript nevyvolá chybu. Pokud je však konfigurační soubor předán přepínačem, neúspěch tento soubor přečíst chybu vyvolá. Zároveň se při použití přepínače ignoruje defaultní konfigurační soubor.

Formát konfiguračního souboru je následující.

- Bílé znaky na začátku řádku se ignorují.
- Prázdné řádky nebo řádky složené pouze z bílých znaků se ignorují.
- Řádky začínající znakem komentáře se ignorují.
- Na jednom řádku může být maximálně jedna direktiva.
- Direktivy mají právě jednu hodnotu.
- Direktivy a hodnoty jsou odděleny přesně jednou mezerou.

Použití chybné direktivy vyvolá chybu.

Kapitola **5**

Implementace a testování

Následující kapitola se zabývá výběrem technologií a důležitými algoritmy, které byly při tvorbě skriptu použity. Na závěr pojednává o testování výsledné aplikace.

5.1 Výběr technologií

Práce se zabývá skriptováním v Bash shellu a výběr technologií je v důsledku omezený. Srovnání Bashe s ostatními prostředky bylo zmíněno v předchozích kapitolách. V aplikaci je tedy využíván především Bash a na Linux systémech běžně dostupné GNU a jiné utility. Není použit jiný programovací jazyk jako Python, zejména z důvodu jednoho z hlavních účelů výsledné aplikace, a to demonstrování osvědčených postupů při skriptování v shellu. V době psaní práce byla populární Bash verze 4.4 a implementování i testování je na tuto verzi zaměřeno.

5.2 Důležité algoritmy

Následující sekce popíše implementaci vybraných algoritmů, jejich možné problémy a vylepšení.

5.2.1 Funkce read_options() a read_config()

Jedná se o hlavní funkce zpracovávající uživatelský vstup. Tyto funkce spolu úzce souvisí, jelikož přepínače mají prioritu před konfiguračním souborem. Z tohoto důvodu se funkce `read_config()` provádí před funkcí `read_options()`. Implementace přepínače pro předání ne-defaultního konfiguračního souboru však situaci komplikuje, protože nemůžeme zpracovávat konfigurační soubor předaný přepínačem, pokud se konfigurační soubor zpracovává před přepínači. Řešením je přesunutí zpracování některých přepínačů do funkce `read_config()`.

Samotné zpracování přepínačů je implementováno pomocí utility `getopt`, která příkazovou řádku normalizuje a připraví pro následné jednoduché zpracování pomocí, například `while` cyklu. Původní argumenty jsou normalizovanými argumenty nahrazeny. Tato normalizace musí neintuitivně proběhnout již ve funkci pro čtení konfiguračního souboru (ne ve funkci pro čtení přepínačů), jelikož některé přepínače musí být zpracovány už tam.

```
read_config() {
   # Normalizace argumentů
   if ! _script_option_string="$(getopt ... )"

   # Nahrazení původních argumentů normalizovanými
   eval set -- "$_script_option_string"

   # Zpracování argumentů
   while true; do
      case "$1" in
         -f|--config-file)
         ...
      esac
   done
}
```

Snippet 30: Getopt normalizace

Funkce `read_options()` již normalizovat argumenty nemusí a pouze před zpracováním použije normalizované argumenty od funkce `read_config()`.

5.2.2 Příprava uživatelských přepínačů a direktiv

Pokud si uživatel zvolil generování zpracování přepínačů nebo konfiguračního souboru, proběhne příprava definovaných přepínačů a direktiv pomocí funkce `parse_user_parameters()`. Funkce nejprve zvaliduje identifikátory v obdržených definicích a přepínače i direktivy uloží do příslušných polí.

Problém nastává v momentě, kdy skript musí řešit duplicitní definice. Řešení pro direktivy je triviální, bereme v potaz pouze nejposlednější definici dané direktivy. Situace pro přepínače je však komplikovanější, jelikož se v generovaném kódu kombinují krátké a dlouhé přepínače. Uvažujme definici, která definuje kombinaci krátkého přepínače `-a` a dlouhého přepínače `--aaa`. Jak se skript zachová, pokud je na vstupu nová definice, která definuje kombinaci `-a` a `--ccc`? Jednoduché řešení by bylo původní definici zcela vyřadit. Lepší řešení, které bylo implementováno, respektuje důležitost nejnovějších definic, ale odstraňuje z předchozí kombinace jen tu část, která je v nové definice použita. V našem případě tedy dostaneme původní přepínač `--aaa` s původními

atributy a novou kombinaci přepínačů `-a` a `--ccc` s novými atributy. Jinak řečeno, pouze přepínač `-a` byl z první definice odstraněn.

Tento klíčový algoritmus udržuje pole kombinací, které po přečtení všech definic projde v opačném pořadí, od nejnovějších kombinací po nejstarší. Pokud kombinace obsahuje ještě nevygenerový přepínač, je vložen do pole pro finální kombinace přepínačů a označen jako vygenerovaný.

```
parse_user_parameters() {
   index=$(( "${#comb_opts[@]}" ))

   # Iteruj pokud existují nevygenerované přepínače
   # a stále máme kombinace
   while (( "${#short_tmp[@]}" + "${#long_tmp[@]}" > 0
            && index > 0 ))
   do
      result_param=''
      IFS=':' read -r short long <<< "${comb_opts[--index]}"

      # Je v této kombinaci nevygenerovaný krátký přepínač?
      if is_set "short_tmp[$short]"; then
         result_param="${short_tmp["$short"]}"
         # Označit jako vygenerovaný
         unset "short_tmp[$short]"
      fi
   ...
   done
}
```

Snippet 31: Duplicitní kombinace přepínačů

5.2.3 Generování zpracování přepínačů a konfiguračního souboru

Generování zpracování přepínačů a konfiguračního souboru zajišťují funkce `generate_user_options()` a `generate_user_config()`. Obě funkce mají podobnou strukturu a využívají připravené přepínače a direktivy od předchozí funkce. Struktura je následující.

1. Vygeneruj statickou hlavičku.
2. Vygeneruj proměnné zpracování přepínačů/direktiv pomocí `case` vzorů.
3. Vygeneruj podmínky uvnitř `case` výrazů na základě atributů.
4. Vygeneruj statické záhlaví.

Body 1 a 4 jsou triviální. Bod 2 využívá připravená data od předchozí funkce. Generování podmínek (bod 3) má na starost funkce `generate_if_statement()`. Tato funkce jako parametr obdrží string, který obsahuje atributy aktuálně zpracovávané proměnné. Tyto atributy se mapují pomocí asociativního pole na reálné názvy funkcí, které jsou použity při generování `if` podmínek.

```
generate_if_statement() {
   # Mapa funkcí a atributů
   function_map=(
      [int]='is_integer'
      [readable]='is_readable'
   )
   # Zpracuj všechny atributy
   for attribute in "${attributes[@]}"; do
      # Generuj podmínku, pokud atribut existuje
      if ! is_empty "${function_map[$attribute]}"; then
         conditions+=("${function_map[$attribute]}")
         functions["${function_map[$attribute]#! }"]='used'
         # Odstranit mapování, ať nejsou duplicitní podmínky
         unset "function_map[$attribute]"
      fi
   done
   ...
}
```

Snippet 32: Generování podmínek pro proměnné

5.2.4 Použité proměnné a funkce

Skript si udržuje tři asociativní pole použitých proměnných, polí a funkcí. Použité proměnné a pole jsou v kódu korektně lokalizovány a použité funkce vygenerovány. Generování funkcí má na starost funkce `generate_functions()`. Ta prvotně přeloží možné balíčky funkcí zvolených uživatelem na reálné funkce a následně zavolá funkci, která jednoduše vygeneruje definice všech vybraných funkcí. Funkce jsou v kódu tzv. natvrdo. Druhou možností bylo využití příkazu `declare -f funkce` pro získání definic. Problém však je, že tento příkaz nerespektuje původní formátování a je důsledkem nevhodný.

```
# Pokud byly použity proměnné, lokalizuj je
if ! is_empty_array scalars; then
   printf '%s\n' "local $(join_elements ' ' "${!scalars[@]}")"
fi
```

Snippet 33: Lokalizování proměnných

5.3 Testování

Testování proběhlo pomocí testovacího skriptu `run_tests`, který spouští sadu testů zaměřených na různé kritické části skriptu. Jedná se například o testování nekorektních definic přepínačů od uživatele apod.

Testování též proběhlo pomocí nástroje ShellCheck. Ve výsledku byl skript doladěn do podoby, kdy ShellCheck zobrazuje pouze dvě varování. Tato varování jsou však důsledkem neintuitivní implementace zpracování přepínačů a nepředstavují problém.

Závěr

Cílem práce bylo vypracovat studii zaměřenou na osvědčené postupy při skriptování v shellu a následně navrhnout a implementovat skript, který automatizuje vytváření kostry nového skriptu. Implementace i generovaná kostra se řídí principy popsanými v rešeršní části práce.

Rešeršní část práce pokrývá zásadní principy při skriptování v shellu, zejména interpretu Bash. Převážně se tato práce zaměřuje na mírně pokročilé uživatele. Výsledná aplikace, vzniklá za pomocí analyzovaných postupů, se zaměřuje na generování těchto hlavních částí: zpracování parametrů, zpracování konfiguračních souborů, vytvoření užitečných defaultních funkcí, nastavení prostředí skriptu, logické strukturování kódu a v neposlední řadě, zachycování signálů.

Vytvořená studie neobsahuje pokročilé postupy a do hloubky nepojednává o jiném shellu než Bash. Do budoucna by bylo vhodné zmínit postupy doporučené pro jiné populární interprety, například Zsh. S možností vzestupu jiného z shellů by mohla v budoucnu vzniknout potřeba skript pozměnit, aby byl stále pro cílovou skupinu aktuální. Tedy aby generoval skripty v jazyce požadovaném širokou komunitou. Výsledný skript lze rozšířit o další funkcionalitu jako například pokročilejší zpracování parametrů, které nespoléhá na externí nástroje. V dnešním GUI světě by bylo také vhodné rozšířit skript o kvalitní grafické rozhraní.

Literatura

[1] GNU Project: *Bash Reference Manual [online].* [cit. 2018-12-05]. Dostupné z: https://www.gnu.org/software/bash/manual/bash.html

[2] Thompson shell. *Wikipedia, The Free Encyclopedia [online]*, prosinec 2018, [cit. 2019-01-02]. Dostupné z: https://en.wikipedia.org/wiki/Thompson_shell

[3] Robbins, A.; Rosenblatt, B.: *Learning the Korn Shell.* O'Reilly Media, druhé vydání, ISBN 978-0596001957.

[4] Bash changes. Červenec 2018, [cit. 2019-01-04]. Dostupné z: https://wiki.bash-hackers.org/scripting/bashchanges

[5] Cesati, M.; Bovet, D. P.: *Understanding the Linux Kernel.* O'Reilly Media, třetí vydání, ISBN 978-0596000028.

[6] The Perl Foundation: *Perl Programming Documentation [online].* [cit. 2019-01-08]. Dostupné z: http://perldoc.perl.org/perlintro.html

[7] Raymond, E. S.: *The Art of UNIX Programming.* Addison-Wesley, první vydání, ISBN 978-0131429017.

[8] BASH Frequently Asked Questions. Květen 2017, [cit. 2019-01-14]. Dostupné z: https://mywiki.wooledge.org/BashFAQ/050

[9] Bumblebee. Květen 2011, [cit. 2019-02-01]. Dostupné z: https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123

[10] Steam for Linux Client. Leden 2015, [cit. 2019-02-02]. Dostupné z: https://github.com/valvesoftware/steam-for-linux/issues/3671

Příloha A

Seznam použitých zkratek

GNU GNU's Not Unix

BSD Berkeley Software Distribution

IEEE Institute of Electrical and Electronics Engineers

POSIX Portable Operating System Interface for Unix

BASH Bourne-again shell

ZSH Z-shell

KSH Korn shell

TCSH Tenex C shell

JSON JavaScript Object Notation

IFS Internal Field Separator

TUI Text-based User Interface

HTML HyperText Markup Language

PDF Portable Document Format

GUI Graphical user interface

XML Extensible markup language

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
snippets .......................... sada spustitelných skriptů
  data ............................ data pro spustitelné skripty
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```