Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Comma-shell, interaktivní debugger shellu

Bc. Tomáš Nesrovnal

Vedoucí práce: Ing. Jan Baier

1. května 2017

Poděkování

Děkuji svému vedoucímu Ing. Janu Baierovi za cenné rady a nápady.

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 1. května 2017 .....

Odkaz na tuto práci

Nesrovnal, Tomáš. *Comma-shell, interaktivní debugger shellu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017. Dostupný také z WWW: ⟨`https://github.com/nesro/commash/`⟩.

Abstrakt

Comma-shell je nástroj pro lepší práci v interaktivním shellu, určený především pro uživatele seznamující se shellem. První částí je debugger, který umí spouštět, nebo analyzovat příkazy po částech. Druhou částí je framework pro spouštění uživatelských skriptů před a po spuštění příkazu včetně několika předpřipravených, například statická analýza kódu pomocí nástroje ShellCheck, která umí zabránit spuštění chybného příkazu. Poslední částí je sada přepsaných několika základních příkazů z GNU coreutils, které jsou méně náchylné na chyby uživatele a lze vrátit změny jimi provedené.

Klíčová slova GNU, Bash, shell, debugger

Abstract

Comma-shell is a tool to enhance the work in the interactive shell, primarily for new shell users. The first part is a debugger that can run or analyse commands by parts. The second part is a framework for executing user scripts before and after running commands from the shell. Some of these scripts are already made, for example static analysis done by ShellCheck, which can cancel execution of the bad command. The last part is a set of wrappers of some

basic commands from GNU coreutils, these wrappers are more user friendly and changes they make can be reverted.

Keywords GNU, Bash, shell, debugger

Obsah

Úvod **1**
Motivace 1

1 Definice a pojmy **3**

2 Historie shellu a dnešní využití **5**
2.1 Windows 5
2.2 UNIX 5
2.3 Linux 7
2.4 Shell 7
2.5 cmd.exe 7
2.6 Power Shell 8
2.7 v9 - Thomson Shell 8
2.8 Sh - Bourne Shell 8
2.9 Dash - Debian Almquist Shell 9
2.10 Csh - C Shell 9
2.11 Ksh - Korn Shell 9
2.12 Bash - Bourne Again Shell 9
2.13 Zsh - Z Shell 9
2.14 Práce v příkazové řádce 10
2.15 Nastavení shellu 11
2.16 Terminál 11
2.17 Existující řešení 11
2.18 Bash frameworky pro psaní skriptů 12
2.19 Bash frameworky pro správu doplňků 12
2.20 Bash frameworky pro úpravu promptu 12

3 Analýza a návrh **15**
3.1 Cíl práce 15
3.2 Fungování shellu 16

3.3 Debugovaní shellu 20
3.4 Možnosti debugování v interaktivním shellu 24
3.5 Statická analýza skriptů 27

4 Realizace **31**
4.1 Virtuální stroj 31
4.2 Instalace Comma-shellu 31
4.3 Nespouštění příkazů 32
4.4 Hooks 32
4.5 Implementace debuggeru v interaktivním shellu 35
4.6 Použití a možnosti debuggeru v interaktivním shellu 36
4.7 Implementace debuggeru shell skriptů 38
4.8 Bezpečný mód 40

5 Testování **45**
5.1 Jednotkový test 45
5.2 shUnit2 45
5.3 Bats 45
5.4 Automatizované spouštění příkazů 46

Závěr **49**
Možnost využití 49
Další práce 49

Literatura **51**

A Seznam použitých zkratek **53**

B Obsah přiloženého CD **55**

Seznam obrázků

2.1 Historie unixu 6
2.2 Historie Unixového Shellu 7

3.1 Architektura shellu 16
3.2 Bash diagram 19
3.3 Bash eclipse 25

Úvod

Grafické uživatelské rozhraní (GUI) se jednoduše ovládá, ale ne vždy je k dispozici. To platí zejména při ovládání serverů.

Rozhraní příkazové řádky (CLI) je základní textové prostředí pro komunikaci s operačním systémem. Umožňuje spouštění programů, vkládat vstupní data a sledovat výstupní data v terminálu.

Jedním ze základních bodů UNIXové filosofie je mít jednoduché programy, které dělají pouze jednu věc, ale dělají ji dobře. To platí zejména pro základní příkazy ze sady GNU coreutils, tedy příkazy pro základní manipulaci se soubory, shellem a textem.

Tyto základní příkazy je možné řetězit a tím vytvářet užitečné jednořádkové skripty.

TODO: napsat o tom, ze pro začátečníka to může byt neintuitivní, musí si pamatovat spoustu příkazu. o návratových kódech, o historii prikazu a o logovani

Motivace

Příkazová řádka a základní práce s ní by měly být jedna z prvních věcí, které by se uživatel měl naučit, pokud chce umět používat svůj systém na vyšší úrovni. Stejně jako v programování platí, že počítač provádí dokonale a přesně to, co mu řekneme. Ale i při sebemenší chybě v našem příkazu selže celá sekvence akcí, které se měli provést. Není těžké udělat i nějakou fatální chybu, po které není snadné uvést vše do původního stavu.

Náš projekt, Comma-shell, si dává za úkol vytvořit z příkazové řádky místo, ve kterém je těžší nějaké chyby udělat, případně před chybou varovat, nebo vysvětlit, jak danou věc udělat lépe. Důležitý cíl je také umožnit nastavit toto prostředí tak, aby bylo snadno použitelné, nebylo potřeba externích terminálů, nebo grafických aplikací a aby bylo možné snadno omezit repetitivní výstupy programu. Comma-shell se dělí na tři části, každá pomáhá řešit jiný problém.

Comma-shell hooks, nebo-li části kódu, které lze vykonat před, nebo po vykonání spuštěného příkazu a dokonce zabránit jeho spuštění umožňují ochránit uživatele před vykonáním příkazu, který je v například nějakém smyslu nesprávný, špatný, nebo je v nepořádku. Pokud tedy napíše příkaz, který obsahuje nějaký překlep, nebo chybu, která by způsobila neočekávané chování, může být uživatel varován, než bude takový příkaz proveden. Nejde jen o ochranu uživatele před špatnými příkazy. Díky informacím o spouštěném příkazu a následně o jeho výsledku lze psát různé skripty, které by jinak bylo složité vytvořit.

Comma-shell debugger, tedy ladící program pro příkazy spouštěné z interaktivní příkazové řádky. Pokud uživatel píše složitější program, obsahující například smyčku, nebo roury a příkaz nefunguje tak, jak uživatel očekává, tak nejjednodušší je rozdělit si příkaz na podpříkazy a ujistit se, že každý příkaz dělá to, co od něj uživatel očekává. Comma-shell debugger by měl tento postup automatizovat.

Poslední částí jsou takzvané bezpečné příkazy. Pokud uživatel vykoná příkaz, jehož efekt byl jiný než očekával, chce spustit opačný příkaz, aby napravil svůj omyl, nebo má často mnohem větší problém, pokud šlo například o nesprávné použití programu pro mazání souborů. Cílem bezpečných příkazů je tedy zaprvé umožnit vrátit provedené změny, ale i informovat uživatele o tom, co se stane. Pokud uživatel používá ke specifikování souborů, které mají být cílem nějakého příkazu, je častý omyl vybrat nějaké soubory nechtěně. Nebo například provést příkaz ze špatného adresáře.

Cílem práce je i provést rešerši existujících nástrojů pro statickou analýzu, krokování a hledání chyb v Bash skriptech.

Kapitola 1

Definice a pojmy

Příkazová řádka, anglicky Command Line Interface, je uživatelské rozhraní ovládané příkazy. Příkazy spuštěné z příkazové řádky vypisují výstup také na příkazovou řádku a další interakce se spuštěným programem probíhá zase zadáváním příkazů. Na rozdíl od textového nebo grafického rozhraní uživatel nemůže nijak zasahovat do již vypsané či vykreslené části.

TODO: obrazek prikazove radky, tui a gui

Shellem se obecně myslí uživatelské prostředí pro využívání funkcí jádra operačního systému. V této práci budeme slovem shell označovat právě interpret příkazů v příkazové řádce. Tedy program, ovládaný příkazy, který umí spouštět ostatní programy a zajišťovat jejich vstup a zobrazovat jejich výstup.

Prompt je krátký řetězec znaků, který se objeví na začátku řádky, do které se bude vepisovat příkaz v příkazové řádce. Je používán k zobrazování důležitých informací, jako například, ve kterém adresáři se uživatel nachází či jaké je jméno počítače, na kterém se nachází.

GNU je projekt zabývající se tvorbou a propagací svobodného software. Cílem je vytvořit kompletní svobodný operační systém unixového typu. Protože jádro operačního systému od GNU s názvem Hurd nebylo tak úspěšné, začalo se používat jádro Linux. Kombinace operačního systému GNU a jádra Linux se označuje jako GNU/Linux. GNU vytvořilo i svůj shell, Bash. Sada základních příkazů, jejichž existence je předpokládána v nějaké formě na každém operačním systému, se jmenuje GNU coreutils. Patří do ní příkazy jako ls, cp, rm, atd. Je důležité si uvědomit, že příkazy jako cd, echo jsou vestavěné příkazy shellu Bash.

Terminál byl z historického hlediska elektronické zařízení k základní komunikaci s počítačem. Dnes se v operačních systémech spouští terminálové emulátory, ve kterých lze spustit různé programy. Nejčastěji je to právě shell.

Provádění příkazů můžeme rozdělit na dvě kategorie. Jednou z nich je interaktivní režim, kdy interpret spouští příkazy tak, jak je uživatel píše do příkazové řádky. V tomto režimu může uživatel také používat klávesové zkratky, aliasy, automatické doplňování a další funkce v interaktivním režimu. Druhou

kategorií je dávkový režim, kdy uživatel nejprve příkazy napíše do souboru. Takovému souboru obsahujícímu příkazy se říká skript. Takový soubor může být zavolán interpretem shellu, který postupně vykoná všechny příkazy. Tento postup se hodí pro složitější a opakující se příkazy.

trap todo?

Kapitola 2

Historie shellu a dnešní využití

V této kapitole stručně shrneme historii rodin nepoužívanějších operačních systému a shellů, které se v nich používají.

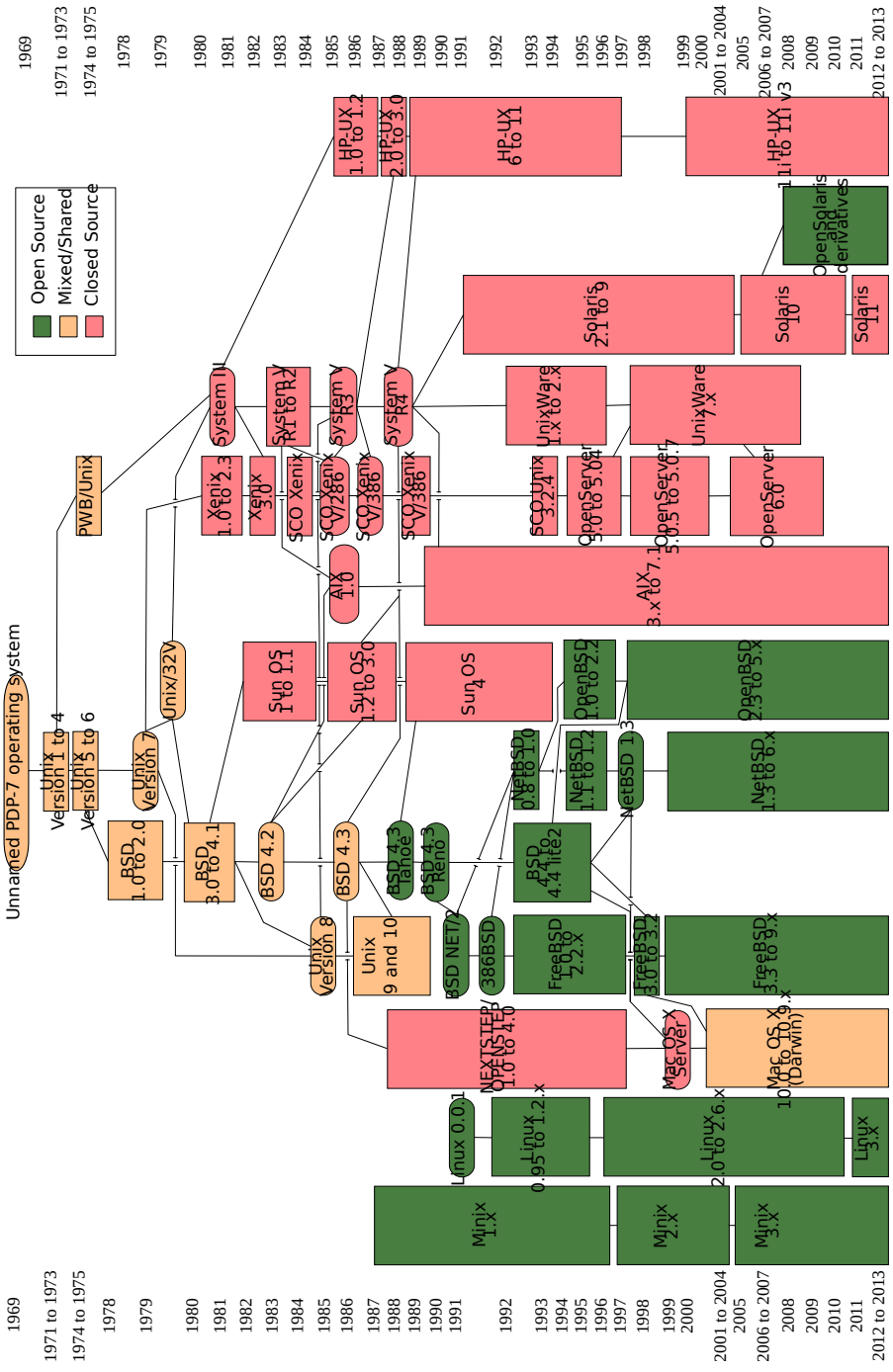
2.1 Windows

Operační systém Windows je obecně uživatelsky přívětivější, protože skoro vše je možné nastavit přes grafické rozhraní. Ovšem ne vždy je grafické prostředí dostupné a příkazová řádka je jediná možnost, jak ovládat počítač.

2.2 UNIX

Unix je rodina operačních systémů, které zvládnou spustit více úkonů najednou a do kterých se v jeden okamžik může připojit více uživatelů. Historie těchto sytému sahá až na konec sedmdesátých let, kdy v Bell laboratořích firmy AT&T byl v jazyce C vytvořen systém UNIX, jehož název byl zaregistrován jako ochranná známka. Časovou osu ukazuje obrázek 2.1. Specifikace "Single UNIX Specification"je souhrn standardu, které operační systém musí splňovat a dodržovat, aby se mohl označovat za UNIX.

Filosofie Unixu je sada doporučení a pravidel, která vznikla postupem času dle zkušeností tvůrců Unixu. Filosofie Unixu zdůrazňuje tvoření jednoduchého, přehledného a hlavně snadno rozšiřitelného a znovupoužitelného softwaru. Váží si programátorův čas více, než čas práce počítače. Nejznámější pravidlo je tvořit programy, které dělají jen jednu věc a dělají jí správně, rychle a bez chyb. Takové programy lze použít jako filtry a skládat je za sebe. To se často používá v příkazové řádce, kdy se pomocí rour spojují výstupy programů se vstupy jiných programů. Zdánlivě složitý úkol tak lze udělat rychle a efektivně bez nutnosti programování v nějakém jazyce.

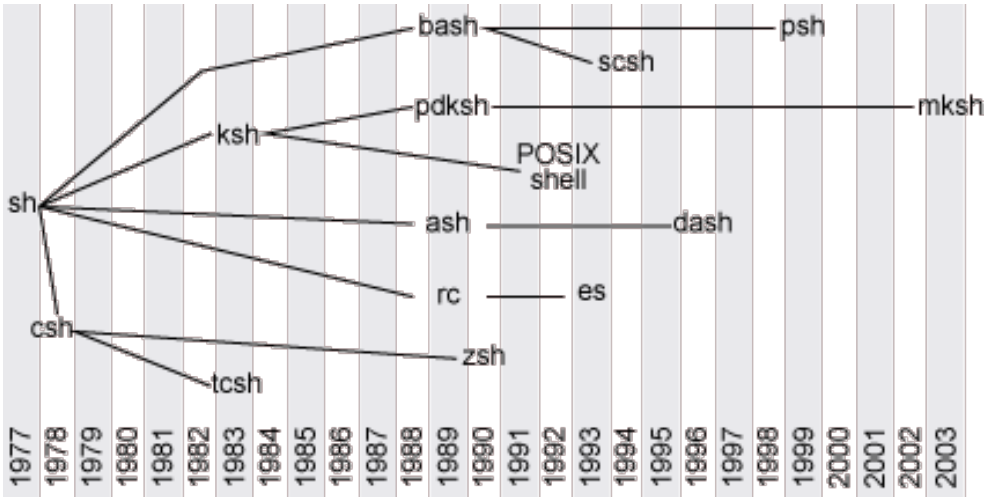


Obrázek 2.1: Historie unixu

2.3 Linux

Linux je otevřený a svobodný Unix-like operační sytém. Linuxem se obecně myslí GNU operační systém s jádrem Linux. Unix-like znamená, že přestože je takový operační systém velice podobný systému Unix, nemá certifikaci "Single UNIX Specification". Linuxová distribuce je GNU/Linuxový systém, který většinou obsahuje balíčkovací systém, kterým je snadné doinstalovat ostatní programy. Linuxové distribuce jsou i předpřipravené, s grafickým prostředím. Například v této práci ukazujeme návod na vytvoření takového připraveného systému, GNU/Linuxové distribuci Debian s předpřipraveným grafickým prostředím LXDE a aplikacemi z rodiny LXDE.

2.4 Shell



Obrázek 2.2: Historie Unixového Shellu

2.5 cmd.exe

Shellem v operačním systému MS-DOS a PC-DOS byl COMMAND.COM, 16-bitový intepretr příkazů. V novějších systémech Windows byl tento shell nahrazen CMD.EXE, 32-bitovým interpretem příkazů. V CMD.EXE byla dostupná emulace COMMAND.COM, pro zpětnou kompatibilitu.

Tato příkazová řádka, shell, již umí přesměrovávat vstupy a výstupy a to jak do souboru tak mezi procesy. Také je schopná vykonávat příkazy ze souborů, které se nazývají dávkové a zpravidla mívají příponu bat.

2.6 Power Shell

Protože je základní příkazová řádka Windows omezená co se funkcionality týče, existují další shelly pro operační systémy Windows, často i s otevřeným zdrojovým kódem. Firma Microsoft chtěla vytvořit i svůj shell, který bude umět více věcí. S jeho vývojem začala v roce 2002 a po 3 letech jej zveřejnila jako Microsoft Command Shell (MSH) s krycím jménem Monad, který byl později přejmenován na Power Shell.

Power Shell je integrován s platformou .NET od Microsoftu. Je tedy možné využívat již vytvořené funkce a nástroje, které usnadní a urychlí vývoj.

Oproti cmd.exe umí Power Shell mnoho věcí. Power Shell umožňuje spouštět příkazy v intervalech, nebo v nějaký přesný čas. Přibyly konstrukce pro větvení. Příkazy je možné spouštět na pozadí, nebo vzdáleně.

2.7 v9 - Thomson Shell

Pro první shell pro systém UNIX napsal Ken Thomson v roce 1971 a byl nazván V9 shell. Jednalo se již o uživatelský program, tedy spuštěný mimo jádro operačního sytému. Shell byl velice jednoduchý a některá základní funkcionalita byla umožněna dalšími programy. Například expanze parametrů byla zajišťována příkazem glob. Základní větvení, if, bylo také jako samostatný program.

V9 shell přinesl syntaxi pro přesměrování a pipelining. Uměl spouštět více oddělených příkazů a dokázal příkazy spouštět na pozadí. Naopak chyběla třeba podpora pro spouštění skriptů.

Díky jednoduchosti a oddělenosti ostatní funkcionality měly zdrojové kódy pouze 900 řádek v jazyce C.

Originalní zdrojové kódy jsou stále dostupné, `https://github.com/yvesnrb/Thompson-Shell`. Projekt Osh, `https://v6shell.org/`, obsahuje jak port tohoto původního shellu, tak i jeho vylepšenou verzi.

2.8 Sh - Bourne Shell

Bourne shell byl vytvořen pro UNIX V7 v roce 1977 Stephenem Bournem v laboratořích Bell a AT&T. Na rozdíl od Thomsonova v9 shellu umí Bourne shell spouštět příkazy ze souborů, skriptů. Šlo tedy psát znovupoužitelné sady příkazů, které zjednodušili a urychlili vývoj různých aplikací. Interaktivní režim je samozřejmě podporován také.

Bourne shell přinesl také podporu smyček, proměnných, signály, subshelly a HERE dokumenty. Skripty v shellu mohly sloužit i jako filtry. Z tohoto shellu poté začaly vznikat další, jak je vidět na obrázku 2.2.

Dnes se Bourne shell stále používá. Je totiž menší, rychlejší a stabilnější než ostatní shelly. Má také minimum závislostí na externí knihovny.

2.9 Dash - Debian Almquist Shell

Dash je moderní implementace Bourne Shellu. Snaží se být stále malý a rychlý, ale přesto třeba i bezpečný a stabilní při chybách systému. Důležitý je fakt, že tento shell splňuje standardy POSIX 1003.2 a 1003.2a, tedy standardy o regulárních výrazech, respektive jak se má chovat shell a jeho základní nástroje.

Tento shell byl nastaven jako shell, který spouští příkazy při bootování i skripty, které mají jako shebang /bin/sh na operačních systémech Ubuntu a Debian.

2.10 Csh - C Shell

C Shell byl vytvořen Bill Joyem v sedmdesátých letech. Hlavní myšlenkou byla syntaxe více podobná jazyku C. Dnes se používá vylepšená verze tcsh.

Přestože není tento shell moc oblíbený v dnešní době, při svém vzniku představil některé nové vlastnosti, které pak začali adaptovat i ostatní shelly. Byla to například historie příkazů, nebo podpora aliasů.

2.11 Ksh - Korn Shell

Korn Shell je Unixový shell, který vytvořil David Korn a jeho kolegové z Bell laboratoří na začátku osmdesátých let. Korn Shell byl vyvíjen z Bourne Shellu a zachovává zpětnou kompatibilitu.

Korn Shell umožnuje upravovat psaný příkaz, jako v editorech Vi, nebo Emacs. Podporu spravování spuštěných programů, historie a aliasy příkazů byly převzaty z C Shellu. Korn Shell dovoluje vytvářet asociativní pole a umí pracovat s desetinnými čísly.

2.12 Bash - Bourne Again Shell

Bash je open source shell projektu GNU. Bash se snaží být zpětně kompatibilní s Bourne shellem, tedy skripty napsané v Bourne shellu pravděpodobně půjdou spustit i v Bashi. Naopak to je problém, který se řeší program checkbashisms, o kterém bude řeč dále.

todo

2.13 Zsh - Z Shell

Z Shell je Unixový shell, pojmenovaný po přihlašovacím jménu profesorovi z Yale, Zhong Shao. Byl v roce 1990 napsán Paulem Falstadem. Ze standardních shellů.

Z Shell nejenom že nabízí nejvíce funkcí mezi používanými Shelly pro Unixové systémy, ale také je kolem něj velká aktivní komunita spravující mnoho pluginů, v čele s Oh My Zsh, o kterém bude řeč dále.

Funkcí navíc oproti ostatním shellům je hodně. Hned po instalaci Z Shellu se uživateli nabídne možnost nastavení, jako například nastavení ukládání historie, nebo automatické doplňování. Z Shell má na rozdíl od Bashe nativní podporu automatického doplňování, která je nejen rychlejší, je více způsobů, jak s ní iteragovat, ale je i snadnější na implementaci do uživatelských skriptů.

Z Shell umožňuje zobrazit hodnotu promptu PS1 i v pravo. Tím se otevírají další možnosti pro zobrazování informací.

Ukládání historie je automaticky sdíleno mezi jednotlivými instancemi Z Shellu. Takové chování je možné nastavit i například v Bashi, ale je to pro to potřeba ukládat každý příkaz do souboru na disk.

Další z věcí ovlivňující pohodlnost práce v Z Shellu je takzvaný globbing, který je lepší než v ostatních shellech. Jedná se o popsání jednoho, nebo několik řetězců, zpravidla názvu souborů, kratším zápisem. Například *.txt se zamění za všechny soubory ve složce s příponou txt. V Z Shellu je ale možné popsat cestu k souborů pomocí zkratek a nechat ji doplnit. Například /h/u/t/s se změní na /home/user/test/script.sh.

Velmi užitečnou funkcí je automatická oprava překlepů. V Bashi je velmi limitovaná oprava příkazů, ale v Z Shellu se opravují nejen názvy příkazů, ale také argumentů. Pokud je argumentem cesta k souboru, který neexistuje, Z Shell navrhne cestu k souboru, který existuje a má podobnou cestu.

Přestože Z Shell se zdá být ve všech směrech lepší než Bash, tak není základním shellem ve většině Linuxových distribucích.

`http://zsh.sourceforge.net/FAQ/`

2.14 Práce v příkazové řádce

Práce v příkazové řádce může být mnohem efektivnější na rozdíl od práce v grafickém prostředí. Při normální práci není tolik nutné používat myš. Složitější, nebo méně používané funkce a programy je zpravidla rychlejší napsat, než vyhledat v nějakém kontextovém menu. To samozřejmě platí až tehdy, kdy uživatel ví, jakým příkazem a s jakými parametry se příkazy spouští.

Nový uživatel, který ještě není s prací v příkazové řádce zběhlý, bude ze začátku méně efektivní, ale časem si základní příkazy zapamatuje a jeho efektivita bude růst.

Další výhodou je možnost používání příkazů ve formě filtrů. I složitou věc je často možné vyřešit pomocí spuštění několika příkazů v rouře za sebou. Příkazy v mezikrocích mohou být také uživatelské skripty.

Velice snadné je také automatizace. Pokud uživatel často potřebuje vykonat nějakou sekvenci příkazů, může složené příkazy vyhledat v historii a třeba z nich vytvořit skript.

Nevýhodou na rozdíl od práce v grafickém prostředí je často fakt, že zpravidla ne všechny potřebné informace jsou stále na očích. Tento problém lze vyřešit rozšířeným a vylepšeným promptem, nebo emulátorem terminálům, které mohou vyhradit nějakou část terminálu pro zobrazování informací. I přes to se stává, že uživatel musí často spouštět nějaké příkazy, které vypíšou nějaké informace.

S rychlejší prací přichází také větší zodpovědnost za spuštěné příkazy. Překlep v příkaze, který provádí nějaké změny může způsobit komplikace. Některé programy se uživatele dotazují, zda-li chce operaci opravdu provést, ale není to pravidlem. Nejlepší prevencí je si tedy příkaz před spuštěním překontrolovat.

2.15 Nastavení shellu

GNU/Linuxové distribuce, které nabízejí předpřipravené prostředí, mají pro výchozí shell bash připravené takzvané rc soubory, které nějakým způsobem upravují běžící interaktivní shell. Jednou z nejviditelnějších změn je nastavení proměnné PS1, o které bude řec v pozdější kapitole, která určuje styl promptu. Ve výchozím nastavení bashe, tedy po spuštění bashe bez načtení rc souborů (bash –norc), se v PS1 zobrazuje pouze název shellu, jeho verze a zdali je uživatel root. Po načtení výchozích rc souborů se v PS1 ukazují informace jako je například jméno uživatele, jméno počítače a co je velmi důležité, jméno aktualního adresáře.

Tyto informace jdou snadno získat použitím některých základních příkazů, například whoami, hostname, pwd. Protože tyto informace jsou důležité a potřebujeme je vedět pořád, chceme je mít pořád na očích.

2.16 Terminál

Protože je terminál elektronické zařízení, ukážeme zde pouze emulátory takových terminálů.

Prvním druhem jsou programy, které vytvoří nové okno v desktopovém prostředí operačního systému. Na operačních systémech Linux je to zpravidla X Window System.

Druhým druhem jsou programy, které se spustí už v nějakém terminálu a emulují další terminál. To se hodí například při vzdálené práci. Takový emulátor běží na straně serveru a pokud se přeruší spojení, terminál zůstane neporušen.

2.17 Existující řešení

Mezi seznamem různých zajímavých řešeních awesome (github.com/sindresorhus/awesome) existuje i podsekce awesome-shell (github.com/alebcay/awesome-shell), ve

které lze nalézt spoustu užitečných nástrojů pro práci s příkazovou řádkou, nebo psaním skripů v bashi.

Spousta těchto nástrojů je nad rámec této práce.

2.18 Bash frameworky pro psaní skriptů

Existuje mnoho projektů, jejichž cílem je vytvořit framework v bashi, který má nějakým způsobem zjednodušit vytváření, především větších, skriptů v bashi. Spousta dnešních vývojářů je zvyklá na objektově orientovaný přístup k programovaní a je tedy pro těžké vytvořit větší program, který by zůstal přehledný.

2.18.1 Bash OO framework

Bash OO framework (`github.com/niieani/bash-oo-framework`) je framework napasny v bashi, ktery umoznuje vytvaret tridy, vyjimky a testy. Jeho cílem je vytvořit prostredi pro psani skriptu, kde se bude snadneji psat citelny kod, bez casti, ktere se opakuji.

2.19 Bash frameworky pro správu doplňků

Existuje spousta věcí, které uživatel příkazové řádky potřebuje občas řešit. Řešení pro daný problém je víc. Buď si uživatel vyřeší problém sám, nebo bude hledat řešení na internetu. Shellovské frameworky, které umějí i spravovat doplňky, mohou mít řešení pro daný problém. Instalace a použití pak bude velmi jednoduché, intuitivní a bude zde nějaká záruka o fuknčnosti a kvalitě.

Přestože je tato práce převážně o shellu bash, zsh je v tomto mnohem rozšířenější.

2.19.1 Oh My Zsh

Oh My Zsh (`ohmyz.sh`) je framework pro Zsh.

2.19.2 Bash-it

Bash-it (`http://github.com/Bash-it/bash-it` , `http://itsfoss.com/bash-it-terminal-tool/`) je framework pro Bash.

2.20 Bash frameworky pro úpravu promptu

2.20.1 Liquid Prompt

Projekt liquidpromp `github.com/nojhan/liquidprompt` je adaptivní prompt v interaktivním bashi.

2.20.2 Sexy Bash Prompt

Projekt sexy-bash-prompt `https://github.com/twolfson/sexy-bash-prompt` je další používaný prompt v interaktivním bashi.

Kapitola 3

Analýza a návrh

3.1 Cíl práce

Prvotním cílem práce bylo navrhnout a vytvořit shell, který bude asistovat studentům na začátku jejich studia shellu. Takový shell by je měl upozorňovat na chyby a navrhovat lepší řešení. Přitom mělo být maximálně využito existujících řešení.

Protože nástroj na upozorňování chyb, konkrétně vynikající ShellCheck, již existuje, byla část cíle práce nastavena obecněji na analýzu a ovlivňování příkazů spouštěných z interaktivní příkazové řádky. Tím se v praxi myslí vytvořit nástroj, který dokáže spustit skripty před vykonáním příkazu a být je schopen ovlivnit, nebo zabránit jejich vykonávání.

Protože psaní špatných příkazů, ať je to jejich nesprávné použití, nebo použití nevhodných konstrukcí, může být i jen provedení nechtěné akce, dalším cílem práce je vytvořit sadu upravených základních programů z GNU coreutils, které uživatele nejprve upozorní na změny, které nastanou po spuštění takového příkazu a možnost vrátit stav změněných věcí do původního stavu.

Posledním cílem je vytvořit jednoduchý debugger příkazů spouštěných v interaktivním shellu. Na rozdíl od normálních debuggerů pro shell skripty, například bashdb - BASH Debugger, je cílem takového debuggeru maximální jednoduchost použití. Tím se například myslí spuštění debuggeru s předepsaným posledním příkazem pomocí spuštění jediného příkazu.

Nejdůležitějším cílem práce je vytvořit takový nástroj, který bude neinvazivní při normálním používání příkazové řádky. Nástroj, který bude začátečník intenzivně využívat, zatímco pokročilý uživatel ho bude mít na pozadí a čas od času ho upozorní na nějakou chybu, popřípadě mu pomůže urychlit práci na nějakém nesprávně fungujícím příkazu.

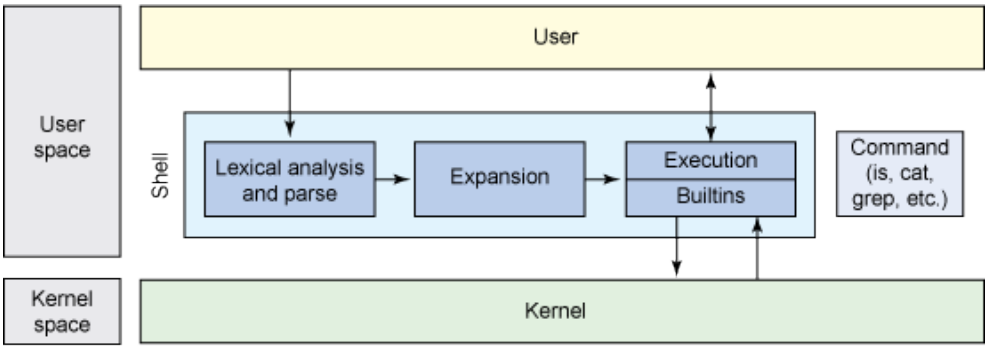
3.2 Fungování shellu

Abychom mohli implementovat jak hooky před a po příkazech, tak přepsat chování základních příkazů z GNU coretuils, je potřeba znát základní procesy, které se dějí při vykonávání příkazů.

Tyto postupy platí pro shell Bash, ale jsou velice podobné i v ostatních Unixových shellech.

3.2.1 Životní cyklus příkazu

V této kapitole vysvětlíme, co všechno všechno důležitého se stane mezi tím, kdy shell přijme příkaz a než je skutečně vykonán.



Obrázek 3.1: Architektura shellu

Prvním krokem je escapovaní, neboli převod znaků převod znaků, které mají ve svém kontextu speciální význam. V shellu lze tento postup provést pomocí znaku \, nebo tím, že řetězec znaků dáme do uvozovek. Příkladem může být znak #, který značí, že znaky za ním jsou pouze komentář. V ukázce 3.1 jsou čtyři příklady zavolání příkazu echo. V prvním případě ale znak # není vyescapován a Bash ho vyhodnotí jako začátek komentáře, proto se vypíše jen část, oproti ostatním voláním.

```
$ echo a # b
a
$ echo a \# b
a # b
$ echo "a # b"
a # b
$ echo 'a # b'
a # b
```

Ukázka 3.1: Escapovaní v shellu

Druhým krokem je odstranění komentářů. V Bashi lze nastavit, aby se v interaktivním shellu komentáře ignorovali. Slouží k tomu proměnná *interactive_comments*, v základním nastavení jsou komentáře zapnuté.

Třetím krokem je rozdělení celého vstupu na příkazy. K příkazům patří přesměrovávání vstupů a výstupů. Jednoduchý příkaz je buď přiřazení proměnné, nebo název příkazu. Co je příkaz bude řečeno dále. Příkazů v jednom vstupu může být více. Příkazy mohou být za sebou v rouře, odděleny logickým and nebo or. Příkazy mohou být také odděleny jednoduše pomocí středníku, nebo v případě spouštění ze skriptu i novou řádkou. Stejně může příkazy rozdělovat ampersand, který značí, že se příkaz spustí na pozadí. Některé konstrukty již z principu obsahují více příkazů, například podmínky nebo smyčky.

Čtvrtým krokem je expanze a substituce. TODO: tohle by mohla byt samostana kaptiola?

Posledním krokem je samotné spuštění příkazu, popřípadě přiřazení do proměnné. O spouštění příkazů pojednává kapitola 3.2.3

3.2.2 Gramatika shellu

Gramatika shellu je oproti ostatním jazykům jednoduchá, ale často nejasná. V mnoha případech záleží na kontextu. Dokonce je možné vytvořit proměnnou s názvem shodujícím se s klíčovým slovem.

Popíšeme si ji zde, jak je napsáno v manuálových stránkách Bashe. Kompletní gramatiku lze nalézt v souboru *parse.y*.

Soubor s gramatikiou bashe, parse.y, ma pres 6000 radek. Chtel bych zde napsat zjednodusenou gramatiku, ktera by se dala snadno pochopit (ono to zas tak slozite neni).

v man bash je gramatika

Popsat jakym zpusobem parsuje gramatiku BASH (yacc) a jakym to delaji parsery bashlex a bashast.

3.2.2.1 Projekty parsující shell

3.2.2.2 Beautysh

Projekt Beautysh si dává za cíl formátovat shell skripty tak, aby byly čitelnější. Jedná se o malý a jednoduchý projekt. Neprobíhá zde žádné parsování do abstraktního syntaktického stromu. Vstup je jen po řádcích rozlišován pomocí regulárních výrazů.

3.2.2.3 Bashlex

Bashlex je knihovna napsaná ve skriptovacím jazyce Python, která zjednodušeně imituje práci vnitřního parseru Bashe. Z velké části se jedná jen o přepsaní zdrojových kódů C.

Existiují rozdíly oproti parseru v Bashi. Za prvé se nespouští žádné příkazy, knihovna umí pouze parsovat. Na rozdíl od pareru Yacc, který používá Bash, je Bashlex reentrantní. Výstupem Bashlexu je kompletní abstraktní syntaktický strom.

TODO jak se pracuje s bashlexem, protoze to pouzivam

```
$ python
>>> import bashlex
>>> parts = bashlex.parse('true && cat <(echo $(echo foo))')
>>> for ast in parts:
...     print ast.dump()
ListNode(pos=(0, 31), parts=[
  CommandNode(pos=(0, 4), parts=[
    WordNode(pos=(0, 4), word='true'),
  ]),
  OperatorNode(op='&&', pos=(5, 7)),
  CommandNode(pos=(8, 31), parts=[
    WordNode(pos=(8, 11), word='cat'),
    WordNode(pos=(12, 31), word='<(echo $(echo foo))', parts=[
      ProcesssubstitutionNode(command=
        CommandNode(pos=(14, 30), parts=[
          WordNode(pos=(14, 18), word='echo'),
          WordNode(pos=(19, 30), word='$(echo foo)', parts=[
            CommandsubstitutionNode(command=
              CommandNode(pos=(21, 29), parts=[
                WordNode(pos=(21, 25), word='echo'),
                WordNode(pos=(26, 29), word='foo'),
              ]), pos=(19, 30)),
          ]),
        ]), pos=(12, 31)),
    ]),
  ]),
])
```

Ukázka 3.2: Výstup z knihovny Bashlex

3.2.2.4 Libbash

Libbash je projekt, který vzniknul v roce 2010 na akci Google Summer of Code. Cílem je jako stejně u knihovny Bashlex vytvořit ze vstup kompletní abstraktní syntaktický strom.

Libbash je napsaný v jazyce C++, ale využívá generátor parseru ANTLR, který je v Javě. Gramatika je popsaná v souboru bashast.g

Projekt se stal i součástí Google Summer of Code v roce 2011. Na stránkách operačního systému Gentoo je Libbashi věnovaná stránka, ale na gitových repozitářích není žádná aktivita. Protože má Libbash velký potenciál, je možné, že se projekt obnoví.

3.2.3 Spouštění příkazů

Shell pracuje v uživatelském adresním prostoru. To znamená, že sám o sobě nemůže pracovat s hardwarem, nebo provádět úkony, jako například vytváření procesů. K tomu, aby takovéto operace mohl provádět, volá systémové volání

tak, aby jádro operačního systému tyto operace provedlo tak, jak potřebuje shell.

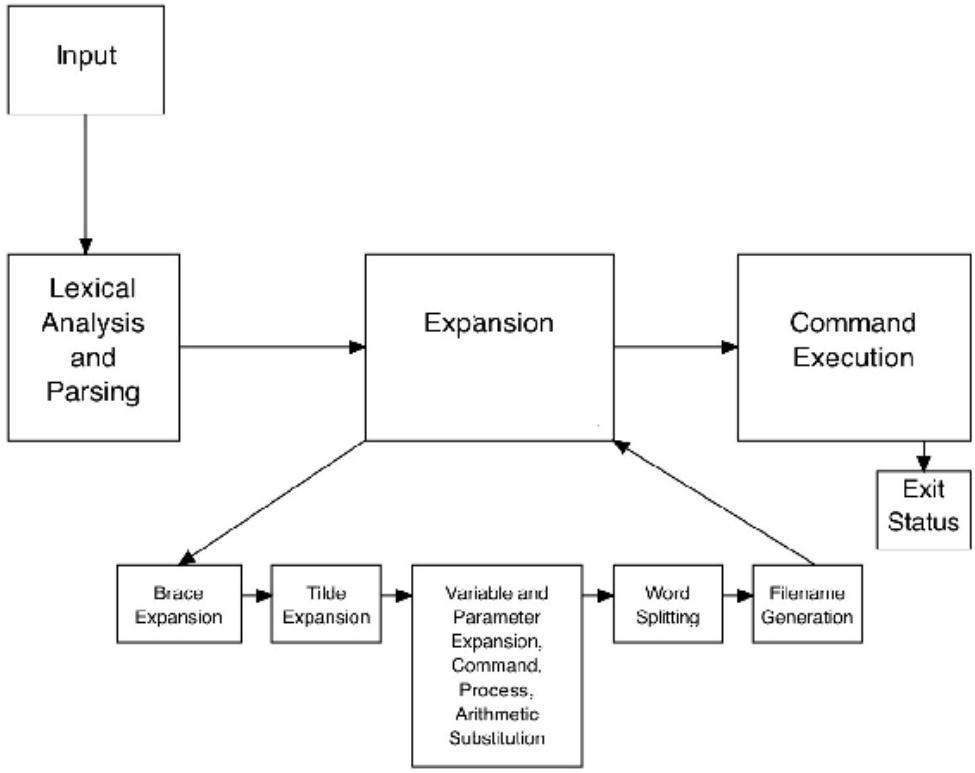
Ke spouštění příkazů se používá systémové volání fork, které vytvoří nový proces tím, že duplikuje proces, jenž fork zavolal.

Pokud je přesměrováván vstup, nebo výstup souborů, je využito systémových volání close pro zavření defaulního výstupu a poté open k otevření výstupu nového.

Pro spouštění příkazů v rouře slouží systémové volání pipe, které v jádře operačního systému vytvoří vyrovnávací paměť pro komunikaci mezi dvěma procesy. Jedná se i o frontu, takže data mohou přicházet rychleji, než je další proces stíhá odebírat.

Důležité je si uvědomit, že ne všechny příkazy potřebují vytváření nových procesů. Například vestavěné příkazy Bashe pracují interně. Takové příkazy fungují mnohem rychleji. Navíc takové příkazy často modifikují nastavení běžícího shellu.

Bash diagram 3.2.



Obrázek 3.2: Bash diagram

3.3 Debugovaní shellu

Cílem této práce je i provést rešerši existujících nástrojů pro statickou analýzu, krokování a hledání chyb v BASH skriptech.

3.3.1 Chyby v Bashi

Pokud se najde takový kód, který vypadá správně, ale při jeho vykonávání nastane něco nečekaného, je dobrou praktikou zmenšit takový kód pouze na tu část, se kterou je něco špatného.

Uživatel pak může chování nahlásit vývojářům Bashe. Pro ověření toho, že něco špatně, může uživatel debuggovat přímo instanci Bashe pomocí debuggeru GNU gdb.

3.3.2 Interní nástroje

Bash sám o sobě obsahuje několik nástrojů, které pomáhají uživateli najít chyby.

Následující nastavení shellu xtrace, verbose, nounset. errexit a functrace se nastavují přes příkaz set, který existuje i v původním Bourne Shellu. Naopak nastavení extdebug se nastavuje příkazem shopt, který v Bourne Shellu není. Detailní informace jsou v manuálových stránkách Bashe, nebo v nápovědě k příkazům set a shopt.

3.3.2.1 xtrace

Xtrace je nastavaní shellu, které po expanzi každého jednoduchého příkazu zobrazí expandovanou hodnotu proměnné PS4. Proměnná PS4 je podobná proměnné PS1, která značí prompt, jen je vypisována právě s výstupem xtrace.

```
bash-4.3$ set -x
bash-4.3$ t=foo
+ t=foo
bash-4.3$ echo $t
+ echo foo
foo
```

Ukázka 3.3: Výstup z knihovny Bashlex

Z ukázky je vidět, že základní hodnota proměnné PS4 je *+*.

3.3.2.2 verbose

Verbose je nastavení shellu, které vypisuje všechny řádky tak, jak jsou čteny.

```
bash-4.3$ set -v
bash-4.3$ t=foo
t=foo
bash-4.3$ echo $t
echo $t
```

```
foo
```

Ukázka 3.4: Výstup z knihovny Bashlex

Nastavení xtrace a verbose lze kombinovat.

```
bash-4.3$ set -xv
bash-4.3$ t=foo
t=foo
+ t=foo
bash-4.3$ echo $t
echo $t
+ echo foo
foo
```

Ukázka 3.5: Výstup z knihovny Bashlex

3.3.2.3 nounset

Nastavení nounset způsobí, že pokud se při expanzi proměnných a nespeciálních parametrů narazí na nějakou nedefinovanou, Bash to bude brát jako chybu.

```
bash-4.3$ set -u
bash-4.3$ echo $t
bash: t: unbound variable
bash-4.3$ echo $?
1
```

Ukázka 3.6: nounset

3.3.2.4 errexit

Nastavení errexit způsobí, že pokud roura, list příkazů, složený příkaz skončí s nenulovou návratovou hodnotou tak se instance Bashe ukončí. Výjimkou jsou místa, která testují návratovou hodnotu, jako jsou podmínky, nebo cykly. Další výjimkou jsou příkazy v seznamu spojené logickým and nebo or, pokud nejsou na jeho konci.

Logické or je právě jeden způsobů, jak i s takovým nastavením spustit příkaz, který má nenulovou návratovou hodnotu. Kterýkoliv příkaz následovaný `|| true` neukončí shell ani s nastavením errexit.

Nevýhodou takového nastavení je často nemožnost testování návratových kódů pomocí proměnné *$?*.

Výhodou je však fakt, že pokud ve skriptu skončí nečekaně nějaký příkaz chybou, často nechceme, aby se pokračovalo ve vykonávání dalších příkazů, protože to může způsobit pouze nepořádek.

Ukázka 3.7 ukazuje, že při nastavení errexit se příkaz echo již neprovede, protože se instance Bashe ukončí.

```
bash-4.3$ bash -c "set -e; false; echo foo"
bash-4.3$ bash -c "false; echo foo"
foo
```

Ukázka 3.7: errexit

3.3.2.5 functrace

Nastavení functrace způsobí, že se DEBUG a RETURN traps dědí do funkcí, substitucí a subshellů. O použití těchto trap bude řeč v TODO.

3.3.2.6 extdebug

Extdebug je nastavení, které v Bashi zapne pokročilý debugovací režim.

První změnou je, že příkaz `declare -F` pro danou funkci řekne, ve kterém souboru a na jakém řádku byla deklarována.

Druhou změnou je chování DEBUG trap. Pokud DEBUG trap vrátí nenulovou hodnotu, další příkaz je přeskočen a není vykonán. Pokud DEBUG trap vrátí jako návratovou hodnotu číslo dva a shell vykonává příkazy ve funkci, nebo zpracovává soubor, je simulován návrat z takové funkce respektive souboru.

Třetí změnou je, že se nastavují proměnné `BASH_ARGC` a `BASH_ARGV`. Tyto proměnné obsahují informace o execution call stack. (TODO: nejak cesky?)

Poslední změnou je, že DEBUG, RETURN a ERR trap jsou děděny do funkcí a subshellů.

3.3.2.7 PS4

PS4 je proměnná, která se vypisuje při zapnutém xtrace. Přestože v základu je nastavena pouze za *+*, existuje mnoho použití, které mohou pomoci vyznat se ve vypisovaném kódu.

V ukázce 3.8 je jednoduchý skript, který před každým příkazem vypíše číslo řádky, ze které byl spuštěn. Důležité je, že při nastavení PS4 musí použít jednoduché uvozovky, aby došlo k expanzi proměnné LINENO až při samotném výpisu xtrace.

```
bash-4.3$ cat -n ps4.sh
     1  #!/bin/bash
     2
     3  u() {
     4          echo bar
     5  }
     6
     7  t() {
     8          echo foo
     9          u
    10  }
    11
    12
    13  PS4='[line: ${LINENO}] '
    14  set -x
    15
    16  t
bash-4.3$ ./ps4.sh
[line: 16] t
[line: 8] echo foo
foo
[line: 9] u
[line: 4] echo bar
bar
```

Ukázka 3.8: ps4

3.3.2.8 PS0

PS0 je proměnná, která je nová pro Bash ve verzi 4.4. V době psaní této práce je již verze 4.4 vydaná a stabilní, ale stále není jako výchozí v Linuxových operačních systémech.

Stejně jako v případě proměnné PS4, může být obsahem volání funkce ze subshellu. Problémem ale je, že taková funkce nemůže jednoduše ovlivňovat stav mimo svůj subshell, proto se hodí spíše jen pro výpis informací.

Stejné funkcionality se dá dosáhnout v Bashi před verzí 4.4 pomocí kombinace DEBUG trap a PROMPT_COMMAND. To je mimo jiné způsob, na kterém pracuje právě Comma-shell a o kterém bude řeč dále. TODO ref

V ukázce 3.9 je vidět výpis proměnné PS0 před výpisem příkazu.

```
bash-4.4$ PS0="[before]\n"; PS1="[after]\n$ "
[after]
$ echo foo
[before]
foo
[after]
$
```

Ukázka 3.9: ps0

3.3.3 Externí nástroje

3.3.4 Bash Debugger

Bash Debugger je nástroj, který má stejnou syntaxi a chování jako debugger GNU gdb. Bash Debugger umí nastavit breakpoint, tedy místo, na které když se dostane interpretr shellu, tak se vykonávání skriptu zastaví a uživatel může zkoumat stav instance shellu, krokovat do následujících příkazů, nebo si nechat zobrazit zásobník volání.

Bash Debugger má výhodu v tom, že sám je napsaný v Bashi. Je tak velice snadné zasahovat do běžícího skriptu svými příkazy.

Funkčnost debuggeru závisí na nastavení DEBUG trap, která je spuštěna před každým příkazem, V DEBUG trapě se pokaždé zkontroluje, zda-li nemá být skript pozastaven, nebo zda není registrovaná nějaká událost. Také EXIT a INT trapy jsou nastavené, aby debugger běžel i po skončení, respektive přerušení skriptu.

todo: vic ukazek

3.3.5 BashEclipse

BashEclipse je plugin do integrovaného vývojového prostředí Eclipse. Pro svůj běh vyžaduje nainstalovaný editor ShellEd.

BashEclipse funguje také na DEBUG trap. Na začátek každého skriptu, který chceme debugovat musíme přidat speciální soubor `_DEBUG.sh`. V tomto souboru se nastaví spojení s TCP/IP socketem ze souboru /dev/tcp/localhost/33333, ze kterého se čtou příkazy, které se v DEBUG trap vykonají.

Zbytek debuggeru je napsán jako doplněk Eclipse.

Ukázkou grafického prostředí z Eclipse při debugování shell skriptu přes BashEclipse je obrázek 3.3.

3.4 Možnosti debugování v interaktivním shellu

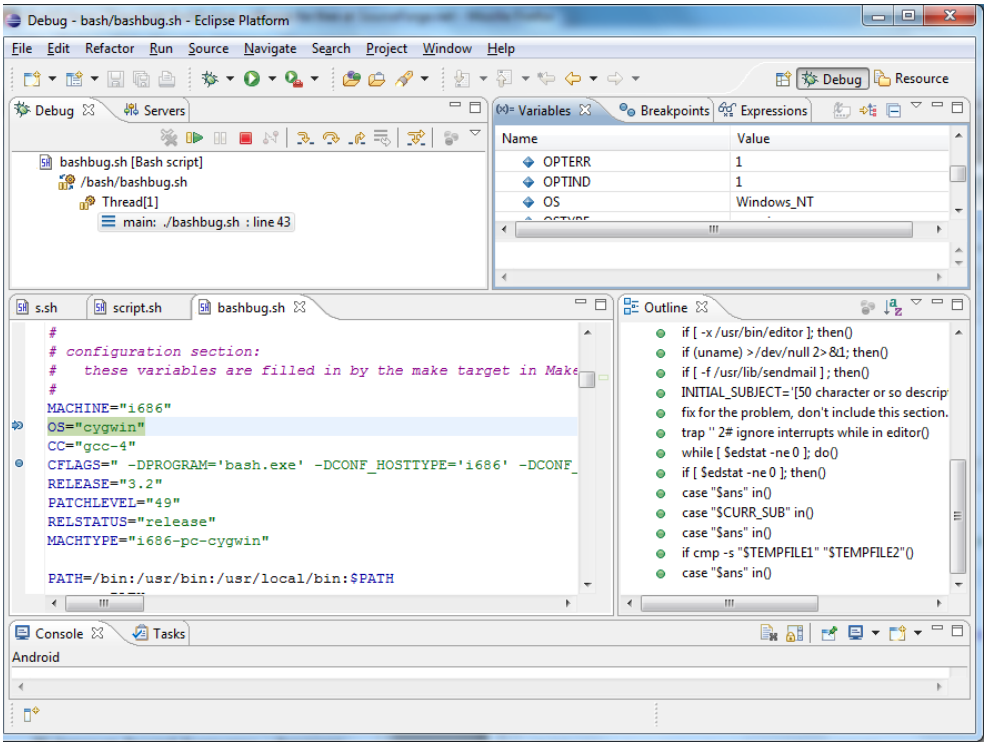
Nebyl nalezen žádný externí nástroj pro debugování interaktivního shellu. Pokud chce uživatel hledat chybu v jednom příkazu co spouští, může mu pomoci nastavení xtrace, aby například viděl, do jakých hodnot expandují proměnné.

Nejčastějším postupem je však spouštění příkazů postupně.

3.4.1 Definice debugování v interaktivním shellu

todo: nema cenu debugovat jednoduche prikazy, ale slozite ano. chceme umet rozkrokovat pipy a subshelly (tedy to co nam to dela ted)

Pro debugování skriptů a programů existuje mnoho nástrojů, které definují, jaké operace a možnosti od takového nástroje očekáváme. Mezi takové základní operací patří možnost zastavit vykonávání skriptu nebo programu na určitém



Obrázek 3.3: Bash eclipse

místě, nebo za určitých podmínek. Po takovém zastavení mít možnost zobrazit si další informace.

V interaktivním shellu se zpravidla nepouštějí příkazy v takovém množství, že by se uživatel chtěl zastavit mezi nimi. V shellu lze příkazy spouštět v rouře a v takovém případě je vhodné vykonávání příkazu pozastavit a nechat si například vypsat výstup, který je následně určen jako vstup do dalšího programu.

I v interaktivním shellu se často setkáme s použitím cyklů, for a while. U for cyklu uživatele může zajímat přes jaké hodnoty iterátoru bude tělo cyklu spuštěno, popřípadě spustit tělo cyklu s vlastním iterátorem.

3.4.2 Možnosti realizování debuggeru interaktivního shellu

Následující kapitoly shrnují možnosti, kterými by šel debugger interaktivního shellu implementovat.

Základní funkcionalita, která je nutná pro vytvoření takového programu je ta, že příkazy napsané do shellu je možné nevykonat a místo toho spustit nějaký vlastní kód.

3.4.3 GNU Readline

GNU Readline umožnuje přemapovat klávesu enter tak, že se před i po řetězec znaků dopíšou nějaké další znaky. Myšlenka byla taková, že se příkaz obalí voláním funkce, ve které se příkaz spustí, popřípadě se zapne nějaký debugger, nebo se vykonají jiné skripty.

Tuto myšlenku se podařilo implementovat, je vidět na ukázce 3.10. V této ukázce už je vidět modifikovaná řádka vstupu, ta se však objeví až po zmáčknutí klávesy enter. Příkaz bind říká, že po zmáčknutí klávesy enter se před vstup má vložit řetězec znaků ", '"a za vstup znak “". Znak ","způsobí volání funkce s názvem ",", tento název byl zvolen tématicky jako je název práce.

Bohužel má tento postup řadu nevýhod, z nichž některé dělají celou myšlenku nepoužitelnou. Největší nevýhodou je, že nelze napsat víceřádkový příkaz, protože při psaní dalšího řádku zapůsobí obalovaní vstupu do funkce. Další nevýhodou je, že se příkaz ke spuštění změní, což může působit velice rušivě. Nevýhodou také je, že takto modifikovaný příkaz se uloží do historie, ale to je věc, která by šla ošetřit.

```
bash-4.3$ ,() { echo "running: \"$@\""; eval "$@"; }
bash-4.3$ bind "RETURN: \"\e[1~, '\e[4~'\n\""
bash-4.3$ , 'echo test'
running: "echo test"
test
```

Ukázka 3.10: Modifikace Readline

3.4.4 Napsání nového REPLu

Další možností, jak ovládat všechny příkazy k vykonání, je napsat celý mechanismus, který se stará o čtení příkazů, jejich vykonání a vypsání dalšího promptu. Takové smyčce se také říká REPL z anglického read-eval-print loop.

Základní implementace by mohla být velice jednoduchá a napsaná v Bashi. Smyčka by mohla být součástí skriptu, který se pustí hned po startu Bashe.

Největší problém toho řešení je fakt, že uživatel není v shellu, ale v našem skriptu. To by mělo za následek například to, že by stavené aliasy v základu nefungovali. Takových případů by se nashromáždilo víc a výsledkem práce by byla omezená verze shellu.

3.4.5 Patch do Bashe

Celý mechanismus pro kontrolu příkazů a jejich nespouštění by mohl být napsán jako patch do zdrojového kódu Bashe. Přestože by takové řešení bylo nejrobustnější, instalace nového shellu je krok, který by mohl odradit nějaké uživatele.

3.4.6 DEBUG trap

DEBUG trap se spustí před vykonáním každého příkazu. Pokud je zapnutý pokročilý debugovací mód pomocí extdebug a návratová hodnota DEBUG trap je nenulová, následující příkaz je přeskočen, tedy nevykonán.

Problém, který zde nastává, je ten, že DEBUG trap se spouští pro každý jednoduchý příkaz a ne pro celý vstup. Na ukázce 3.12 je nastavena DEBUG trap tak, že se vypíše zpracovávaný příkaz, který lze přeskočit.

```
bash-4.3$ trap 'echo "(trap: $BASH_COMMAND)"' DEBUG
bash-4.3$ echo foo | grep oo
(trap: echo foo)
(trap: grep oo)
foo
```

Ukázka 3.11: DEBUG trap

Pokud chceme spustit kód jen jednou pro celý vstup, můžeme použít trik, kdy nebudeme spouštět vůbec nic, tedy DEBUG trap vrátí vždy nenulovou hodnotu a v první DEBUG trap spustit celý vstup pomocí příkazu eval. Tento vstup můžeme získat z historie shellu, protože ta se uloží ještě před vykonáním příkazu.

Protože toho řešení bylo vybráno jako nejvhodnější a realizováno, detailní popis celého postupu je v sekci 4.3.

3.5 Statická analýza skriptů

Statická analýza skriptů znamená prozkoumat, analyzovat skript a hledat v něm například nějaké problémy. Nejdůležitější je, že při této analýze není skript vykonáván.

3.5.1 Vestavěná statická analýza

Při volání skriptu Bashem můžeme použít parametr -n, nebo použít příkaz set -n, kde n značí noexec, tedy mód, ve kterém se nebudou spouštět příkazy.

V takovém módu lze nalézt syntaktické chyby, ale například se nekontroluje existence volaných příkazů. Tato metoda tedy nepomůže proti překlepům.

3.5.2 Check Bashisms

Některé skripty chceme kvůli rychlosti spouštět v Bourne-Shellu. Protože ten je pouze podmnožina Bashe, nástroj Check Bashisms najde výskyty takové syntaxe, která je podporovaná pouze v Bashi a v Bourne-Shellu nemusí fungovat správně.

Check Bashisms je skript napsaný v jazyce Perl a obsahuje sadu regulárních výrazů označující neportabilní syntaxi, které jsou hledané ve vstupním souboru.

3.5.3 Explain Shell

Projekt Explain Shell, provádí statickou analýzu kódu, ale nehledá v něm chyby, ale pouze vyparsuje vstupní kód a vyhledá příslušnou dokumentaci v manuálových stránkách. Využívá se zde knihovny bashlex, která zde byla popsána 3.2.2.3.

Manuálové stránky, ze kterých se vybírá jsou z operačního systému Ubuntu.

3.5.4 ShellCheck

ShellCheck je nástroj, který varuje před chybami a dokonce dává návrhy na vylepšení ve skriptech v Bashi, i Bourne Shellu. Nástroj je dostupný jak online, tak jej lze snadno nainstalovat.

Shellcheck si dává za cíl tři základní věci. Za prvé pomoci nezkušeným uživatelům shellu, upozornit je na základní chyby srozumitelně, protože základní hlášení chyb v shellu může působit zmateně.

Druhý cíl je upozorňovat a vysvětlit uživatelům používající středně pokročilé praktiky možné nestandardní chování, nebo na syntaxi, která není intuitivní.

Posledním cílem je upozornit pokročilé uživatele na okrajové případy a nástrahy. Například skript může pracovat dobře, dokud nemá zpracovat soubor obsahující mezeru v názvu.

ShellCheck obsahuje galerii špatného kódu s přehledem základních chyb. Na jeho stránkách je však také podrobně popsán každý případ, který se hledá a na který se upozorňuje.

ShellCheck je napsaný v čistě funkcionálním jazyce Haskell postavený v systému Cabal. Obsahuje parser, který tvoří abstraktní syntaktický strom. Zdroje chyb, varování a doporučení, které ShellCheck hledá jsou kontrolovány jak při tvoření abstraktního syntaktického stromu, tak je zde i soubor pravidel, ve kterých se hledá shoda s příkazy na vstupu.

ShellCheck je velice aktivní projekt a používá se často na testovacích serverech, jako je například Travis CI.

```
bash-4.3$ ~/.cabal/bin/shellcheck ~/.commash/tmp/shellcheck_example.
    sh

In /home/n/.commash/tmp/shellcheck_example.sh line 5:
if (( $n > 3.5 )); then
      ^-- SC2004: $/${} is unnecessary on arithmetic variables.
           ^-- SC2079: (( )) doesn't support decimals. Use bc or awk
               .


In /home/n/.commash/tmp/shellcheck_example.sh line 9:
if [[ $1 == $n ]]; then
            ^-- SC2053: Quote the rhs of == in [[ ]] to prevent glob
                matching.
```

Ukázka 3.12: ShellCheck

Kapitola 4

Realizace

Cílem této práce má být multifunkční nástroj do interaktivní příkazové řádky. První funkcí je umět analyzovat příkazy před jejich vykonáváním, případně je nevykonat. Druhou funkcí je debugger, který umožní krokovat spouštěné příkazy, nebo spouštět pouze jejich část. Poslední funkcí je sada příkazů, která obaluje základní příkazy z GNU coreutils a umožňuje vrátit změny provedené těmito příkazy, nebo rovnou uživateli vysvětlit, co přesně se stane a upozornit ho na jeho chyby.

Vytvořený nástroj se jmenuje Comma-shell a je to otevřený software. Nejaktuálnější zdrojové kódy, ukázky a návod na instalaci jsou na Githubu https://github.com/nesro/comm

4.1 Virtuální stroj

Pokud si chce uživatel vyzkoušet Comma-shell v bezpečném a izolovaném prostředí, je dobrou praxí vytvořit si virtuální stroj pouze pro tento účel.

Instalace byla otestována na virtuálním stroji s operačním systémem `debian-8.7.1-amd64`. Je nutné nastavit minimálně 2GB operační paměti RAM, jinak je možné, že instalace kompilace některých částí, zejména ShellChecku skončí neúspěchem.

Virtuální stroj, který byl použit pro toto testování je VirtualBox verze `5.0.32_Ubuntu r112930`, pod operačním systémem založeným na Ubuntu 16.04 LTS.

4.2 Instalace Comma-shellu

Instalace Comma-shellu byla navrhnuta tak, aby byla co nejjednodušší. V ukázce 4.1 je příklad instalace v operačním systému vycházejícího z Debianu, například Ubuntu.

Nejprve se aktualizují a nainstalují potřebné balíčky. Proč jsou tyto konkrétní balíčky potřeba bude vysvětleno dále. Po nainstalování je dobré stáh-

nout aktuální verzi z Githubu a spustit hlavní soubor. Ten zjistí, že Comma-shell není nainstalovaný a instalaci spustí.

```
sudo apt update
sudo apt install git cabal-install python-pip xdotool -y
sudo pip install bashlex
git clone https://github.com/nesro/commash ~/.commash
source ~/.commash/comma.sh
```

Ukázka 4.1: Instalace Comma-shellu

4.3 Nespouštění příkazů

V možnostech realizování debuggeru interaktivního shellu (todo ref) bylo zmíněno mnoho možností. Pro realizaci byla nakonec vybráno řešení s DEBUG trap.

Celé řešení je postavené na tom, že DEBUG trap stále vrací nenulovou hodnotu, takže žádné příkazy nejsou vykonány.

Bash obsahuje proměnnou PROMPT_COMMAND, která je vykonána před každém vypsání promptu. Comma-shell v této proměnné volá funkci, která nastaví, že následující DEBUG trap je první v pořadí po odeslání příkazu k jeho vykonání.

DEBUG trap, která má vykonat nějaký kód tak učiní pomocí funkce eval. Eval je vestavěná funkce v shellu, která vykoná příkaz, který ji byl předán argumentem.

DEBUG trap má informace pouze o jednoduchém příkazu, který se má vykonat. Příkaz k vykonání se uloží do historie ještě před zavolání první DEBUG trap. Kód, který vykonává funkce eval je tedy vzat jako ten poslední z historie.

TODO: implementovat minimalni verzi a ukazat ji tady?

4.4 Hooks

Hooks mají v adresářové struktuře vyhrazenou vlastní složku, ze které se automaticky načtou všechny soubory. Comma-shell poskytuje funkce, s nimiž lze zaregistrovat hook spouštěný před, nebo po i s jeho prioritou. Priorita slouží s řazení posloupnosti hooků, protože bez nich by byly spouštěny v pořadí abecedního seřazení podle názvů, což není vždy žádané.

Pokud hook spuštěný před příkazem vrátí nenulovou návratovou hodnotu, není následující příkaz vykonán. Vykonáváním příkazu v tomto kontextu se nemyslí zabránění spuštění příkazu z DEBUG trap, ale samotné vykonání příkazu přes eval v první DEBUG trap po zobrazení promptu v interaktivní příkazové řádce.

Díky této modularitě je velice jednoduché vytvořit vlastní kontroly příkazů ke spuštění. Je také možné vytvářet různé statistiky a logy akcí.

4.4.1 Implementace hooků

Při startu Comma-shell se vyvoří dvě pole. Jedno je určené pro hooky před a druhé pro hooky vykonávané po příkazu. Všechny soubory ze složky hooks vykonají pomocí příkazu source. V hook souborech se pak volají příkazy, které přidají vstupní funkce hooků spolu s jejich prioritou do předpřipravených polí.

Pokud se v hlavní DEBUG trap má vykonat příkaz přes eval, jsou před ním zavolány všechny zaregistrované hooky. Pokud některý z nich vrátí nenulovou hodnotu, příkaz se nevykoná. Pokud k vykonání dojde, jsou následně zavolány všechny hooky určené po vykonání příkazu.

Vstupním funkcím hooku jsou předány dva argumenty. Prvním z nich je časová známka, která je stejná pro všechny hooky a s přesností na nanosekundy může sloužit jako jedinečný identifikátor příkazu. Druhým argumentem je jen spouštěný příkaz. Přestože si ho hook může sám přečíst z historie, pokud bude vnitřní fungování v budoucnu změněno, nebude problém se zpětnou kompatibilitou.

4.4.2 Hooky před vykonáním příkazu

Hooky před spuštění příkazů mohou sloužit ke kontrole příkazu, nebo například zaznamenání místa spuštění. Možné je jen vypsat nějaké doporučené používání příkazu, pokud je někde zvolen pracovní kodex.

4.4.3 Hooky po vykonáním příkazu

Jednou z nejdůležitějších věcí v hooku spuštěném po příkazu je přístup k návratovému kódu. Můžeme tak provádět akce závislé například na chybách příkazů.

4.4.4 Předvytvořené hooky

Comma-shell obsahuje v základu několik předvytvořených hooků. Některé z nich si zde popíšeme.

4.4.5 ShellCheck hook

ShellCheck na nástroj na kontrolu skriptů v Bashi nebo Bourne Shellu. ShellCheck hook však vytvoří z příkazu, který se má vykonat skript, který následně ShellCheck zkontroluje. Pokud jsou nalezeny nějaké problémy, varování, nebo doporučení, vypíše se uživateli zpráva, na základě které se může rozhodnout příkaz nespouštět, upravit, nebo vypnout upozornění na danou chybu.

Protože ShellCheck je určen pro skripty, některé případy, na které upozorňuje nejsou v interaktivním shellu moc relevantní. ShellCheck například kontroluje, zda byl příkaz cd, který změní aktuální adresář, ošetřen tak, že pokud selže, skript se přizpůsobí. Například pokud se chce skript přepnout

do nějaké složky a v ní mazat soubory. Pokud příkaz cd selže v interaktivním shellu, uživatel je o situaci informován a může se podle toho zařídit.

```
n@t:~$ cat x | grep y
,: ShellCheck:
cat x | grep y
    ^-- SC2002: Useless cat. Consider 'cmd < file | ..' or 'cmd file
         | ..' instead.
,: Now what? [r]un, [s]top, [i]gnore, [p]retype: s
,: Stopping the command.
```

Ukázka 4.2: ShellCheck

```
n@t:~$ for f in $(ls *.txt); do echo $f; done
,: ShellCheck:
for f in $(ls *.txt); do echo $f; done
         ^-- SC2045: Iterating over ls output is fragile. Use globs.
              ^-- SC2035: Use ./*glob* or -- *glob* so names with
                  dashes won't become options.
                               ^-- SC2086: Double quote to prevent
                                   globbing and word splitting.
,: Now what? [r]un, [s]top, [i]gnore, [p]retype: s
,: Stopping the command.
```

Ukázka 4.3: ShellCheck

4.4.6 Bashlex hook

todo: tohle je bashovska cast debuggeru

4.4.7 Explain RC hook

Explain RC hook je hook vykonávaný po skončení příkazu. Jediné co dělá je, že pokud je návratová hodnota různá od nuly, vypíše hlášku o nestandardní návratové hodnotě.

Ukázka 4.4 ukazuje spuštění příkazu man s neexistujícím příkazem. Přestože příkaz man sám o sobě hlášku vypíše, hook vypíše další informace, které byly získány z manuálové stránky příkazu man.

```
n@t:~$ man noexistingcmd
No manual entry for noexistingcmd
,: return code warning: $? == 16 (At least one of the pages/files/
    keywords didn't exist or wasn't matched.)
```

Ukázka 4.4: Explain RC hook

Ukázka 4.6 zase ukazuje situaci, kdy byl příkaz ukončen klávesovou zkratkou Ctrl-C.

```
n@t:~$ cat
^C
,: return code warning: $? == 130 (Script terminated by Control-C)
```

Ukázka 4.5: Explain RC hook

4.4.8 Notfound hook

Když Bash hledá příkaz k vykonání a nepodaří se mu ho najít, jako poslední možnost zkusí zavolat funkci command_not_found_handle, které jako argument předá příkaz, který nebyl nalezen.

Comma-shell přepíše funkci command_not_found_handle tak, aby nedělala nic, jen vrátila návratový kód 127. V hooku spuštěném po příkazu se otestuje, zda-li je návratová hodnota 127 a pokud ano, zavolá se kód obstarávající logiku, pokud příkaz není nalezen.

```
n@t:~$ le asdf
,notfound: Command not found.
,notfound: [1] choose: "let asdf"
,notfound: [2] choose: "ln asdf"
,notfound: [3] choose: "ls asdf"
,notfound: [a]bort executing
,notfound: [e]dit the wrong command
,notfound: [p]re-type edited
,notfound: [s]uggest package for "le"
,notfound: [r]emove command from hisory
s
,notfound: executing: /usr/lib/command-not-found -- "le"
The program 'le' is currently not installed. You can install it by
    typing:
sudo apt install le
```

Ukázka 4.6: Explain RC hook

V ukázce 4.6 je vidět notfound hook v akci. Uživatel může spustit příkaz, který zamýšlel, zastavit vykonávání

4.5 Implementace debuggeru v interaktivním shellu

V následujících sekcích bude představen debugger určený do interaktivního shellu. Jedná se tedy o takové použití, že uživatel chce krokovat, nebo částečně spouštět příkazy spouštěné v interaktivním režimu shellu.

Implementace debuggeru je rozdělena na dvě části. První část je hook, který se spustí před příkazem a zjišťuje se v něm, zda-li má být debuggování aktivní. Pokud ano, je zavolán skript napsaný v jazyce Python, ve kterém se použije knihovna Bashlex. Výstupem tohoto skriptu je na standardním výstupu informace pro uživatele, které možnosti má. Na chybovém výstupu jsou kódy v shellu odpovídající jednotlivým akcím, které pak hook v Comma-shellu vykoná po tom, co od uživatele přečte jeho požadovanou akci.

Použití knihovny Bashlex je velice jednoduché a intuitivní. Nejprve je celý vstup parsován a je vytvořen abstraktní syntaktický strom. Poté je projde celý strom znovu s naší třídou, která definuje metody pro jednotlivé uzle. Příklad takové metody je visitpipe, která se zavolá, když se v abstraktním syntaktic-

kém stromě narazí na rouru. V takové metodě máme přístup k pozici roury a z této informace můžeme uživateli nabídnout nějaké akce, třeba vykonat všechny příkazy až do té konkretní roury.

4.6 Použití a možnosti debuggeru v interaktivním shellu

V této kapitole budou ukázány některé možnosti použití debuggeru v interaktivním shellu.

V ukázce 4.7 je z příkazové řádky spuštěn příkaz obsahující konstrukt subshellu. Tedy kód, který se vykoná v nové instanci shellu. Demonstrativní příklad nejprve volá vestavěný příkaz echo, jehož argument, který je string obsahuje výstup ze subshellu s voláním příkazu date, pro vypsání aktuálního roku. Pokud by uživatel chtěl zobrazit pouze obsah subshellu, musel by vykopírovat tu část příkazu jemu odpovídající a spustit. Debugger interaktivního shellu však dokáže tuto práci nejen urychlit, ale také ukáže, které součásti lze spustit samostatně.

```
n@t:~$ echo "the year is $(date +%Y)" | grep 2017
,: commash debugger:

echo "the year is $(date +%Y)" | grep 2017
                  ^-- [1] show substituion: $(date +%Y)
                                ^-- [2] show pipe flow: echo "the
                                    year is $(date +%Y)"

,: Select your option, [0] debug whole cmd, [q]uit, [r]un normally 1
,: Executing: "echo "$(date +%Y)""
2017

,: What now? [q]uit, [p]retype and debug just the next command
```

Ukázka 4.7: Debugger interaktivního shellu - subshell

V ukázce 4.8 je stejný příkaz jako v ukázce 4.7, ale uživatel provádí akci, kde si zobrazí výstup před druhou rourou.

```
n@t:~$ echo "the year is $(date +%Y)" | grep 2017
,: commash debugger:

echo "the year is $(date +%Y)" | grep 2017
                  ^-- [1] show substituion: $(date +%Y)
                                ^-- [2] show pipe flow: echo "the
                                    year is $(date +%Y)"

,: Select your option, [0] debug whole cmd, [q]uit, [r]un normally 2
,: Executing: "echo "the year is $(date +%Y)" "
the year is 2017

,: What now? [q]uit, [p]retype and debug just the next command
```

Ukázka 4.8: Debugger interaktivního shellu - subshell

V ukázce 4.9 je příklad debuggování for cyklu. For cyklus se v shellu často používá na provedení operace pro několik souborů, které jsou specifikované v hlavičce for cyklu. Zde je jsou tyto soubory označeny `*.txt`. Comma-shell debugger umožňuje zobrazit hodnoty, přes které bude postupně spuštěno tělo for cyklu.

```
n@t:/tmp/csfor$ for i in *.txt; do echo "$i"; done
,: commash debugger:

for i in *.txt; do echo "$i"; done
    ^-- [1] show values of iterator: i
                  ^-- [2] run with custom iterator: echo "$i";

,: Select your option, [0] debug whole cmd, [q]uit, [r]un normally 1
,: Executing: "csit=0;for i in *.txt; do (( csit++ )); echo "( $csit
    ) i = $i"; done"
( 1 ) i = a.txt
( 2 ) i = b.txt
( 3 ) i = c.txt

,: What now? [q]uit, [p]retype and debug just the next command
```

Ukázka 4.9: Debugger interaktivního shellu - výpis hodnot iterátoru for cyklu

V další ukázce 4.10 na ten samý for cyklus jako v ukázce 4.9 je tělo for cyklu spuštěno s iterátorem, který interaktivně zapsal uživatel.

```
n@t:/tmp/csfor$ for i in *.txt; do echo "$i"; done
,: commash debugger:

for i in *.txt; do echo "$i"; done
    ^-- [1] show values of iterator: i
                  ^-- [2] run with custom iterator: echo "$i";

,: Select your option, [0] debug whole cmd, [q]uit, [r]un normally 2
,: Executing: "read -p "set iterator \"i\" value: " i; echo "$i";"
set iterator "i" value: test
test

,: What now? [q]uit, [p]retype and debug just the next command
```

Ukázka 4.10: Debugger interaktivního shellu - změna iterátoru for cyklu

4.7 Implementace debuggeru shell skriptů

Bash debugger je nejpoužívanější a zaběhlý debugger skriptů napsaných v Bashi. Pracuje se s ním stejně jako s GNU gdb, takže práce s ním může být snadná pro někoho, kdo s GNU gdb má již zkušenosti. Bash debugger je založený na DEBUG trap, je tedy založený čistě na vnitřních nástrojích Bashe pro debuggování.

Comma-shell debugger shell skriptů se pro to snaží jít jinou cestou. Využívá podobných postupů, jako je Comma-shell debugger v interaktivním shellu. Práce v takovém debuggeru by měla být jednoduchá a natolik intuitivní, že není potřeba studovat žádnou manuálovou stránku. Příkazy, které je možné vykonat budou zobrazeny v menu a budou se spouštět zmáčknutím jedné klávesy.

Práce s tímto debuggerem je taková, že je uživateli nabízené menu pro každý příkaz první úrovně, tedy příkaz, který není v žádné funkci, ani součástí smyček, nebo podmínek. V takovém menu, pokud tomu příkaz odpovídá, je možné vstoupit do tohoto příkazu a krokovat příkazy v něm.

V ukázce 4.11 je příklad použití debuggeru na skript, který začíná for cyklem. Uživatel zvolí možnost krokovat cyklem s vlastním iterátorem. Následně krokuje příkazy uvnitř tohoto for cyklu.

```
n@t:~/.commash/script$ /home/n/.commash/script/commash_script.sh
    test.sh
,dbg: Next command:

for i in 1 2
do
        echo "another $i"
        uname
done

,dbg: Choose:
,dbg:    [1] show values of iterator: i
,dbg:    [2] run body with custom iterator: i
,dbg:    [3] step through iterations with custom iterator: i
,dbg:    [4] step in for body: i
,dbg:    [r]un
,dbg:    [q]quit
,dbg: eval begin
  ,dbg: stepping in for cycle
  ,dbg: set iterator "i" value: TEST
  ,dbg: Next command:

echo "another $i"

  ,dbg: Choose:
  ,dbg:    [r]un
  ,dbg:    [q]quit
  ,dbg: eval begin:

another TEST

  ,dbg: eval end
  ,dbg: Next command:

uname

  ,dbg: Choose:
  ,dbg:    [r]un
  ,dbg:    [q]quit
  ,dbg: eval begin:

Linux

  ,dbg: eval end
  ,dbg: input in context has ended
  ,dbg: stepping out for cycle

,dbg: eval end
```

Ukázka 4.11: Debugger shell skriptu

Implementace je podobná jako v Comma-shell debugger v interaktivním shellu. Základem je parsovací knihovna Bashlex, která je spouštěna ve skriptu v jazyce Python. Tento skript je volán ze skriptu v Bashi, pro který jsou

vraceny jak jednotlivé příkazy, tak i položky menu. S každou položkou do menu je vrácen i kód, který je v Bashi potom vykonán pomocí kostruktu eval.

Pokud se vstoupí do nějakého cyklu nebo podmínky, je hlavní část debuggeru zavolána rekurzivně na tělo podmínky nebo cyklu. Nastavování například iterátorů je zapsáno také jako kód v Bashi, který se vykoná ještě před tímto rekurzivním voláním.

4.8 Bezpečný mód

Bezpečný mód umožňuje dvě základní věci. Tou jednodušší je pouze vypsání efektu příkazu, který má nějaké destruktivní následky. Druhá funkcionalita dovoluje vracení do stavu před vykonáním příkazu.

Některé příkazy, jako například rm pro mazání souborů mají zabudovanou nějakou ochranu před nechtěným spuštěním, ale ta je z našeho pohledu buď nedostatečná, nebo je až moc striktní a zabírá příliš času, před samotným vykonáním příkazu.

Při navrhování bezpečných příkazů byl důraz na myšlenku zobrazit uživateli nějaký souhrn akcí co se stanou. Odsouhlasení základního chování by mělo být zmáčknutí pouze jedné klávesy. Při tom by mělo být možné nechat si vypsat nějaké detailnější informace, především pokud uživatel v souhrnu uvidí něco, s čím nepočítal, nebo co jej překvapilo.

Poslední důležitou věcí je nemít přepsané názvy nebezpečných příkazů na ty bezpečné z Comma-shellu. V takovém případě by si uživatel mohl zvyknout na dané chování a nebral by svoje akce jako absolutní. To by pro něj mohl být problém na systému, kde není Comma-shell s bezpečnými příkazy nainstalován. Jak se vidět na ukázce 4.12 V základním nastavení se například název rm přepíše aliasem tak, aby informoval uživatele.

```
n@t:~$ rm test
,: Use ,rm for commash wrapper or /bin/rm for original rm.
```

Ukázka 4.12: Přepsání nebezpečných příkazů

4.8.1 Comma-shell rm

Příkaz rm z balíčku GNU coreutils obsahuje dva přepínače, které mají sloužit proti smazání nechtěných souborů. První přepínač -i, znamená výzvu před každým smazaným souborem. Jak je vidět na ukázce 4.13, je potřeba skutečně odsouhlasit každý mazaný soubor.

```
n@t:/tmp/rmtest$ mkdir -p dir
n@t:/tmp/rmtest$ touch {a..b}.txt dir/{c..d}.txt
n@t:/tmp/rmtest$ /bin/rm -ir ./*
/bin/rm: remove regular empty file './a.txt'? y
/bin/rm: remove regular empty file './b.txt'? y
/bin/rm: descend into directory './dir'? y
/bin/rm: remove regular empty file './dir/c.txt'? y
/bin/rm: remove regular empty file './dir/d.txt'? y
/bin/rm: remove directory './dir'? y
```

Ukázka 4.13: rm

Druhý přepínač je -I. V manuálových stránkách příkazu rm se píše, že je uživatel vyzván jednou, pokud odstraňuje tři, nebo více soubrů, nebo pokud je mazána složka rekurzivně. U tohoto přepínače je také napsáno, že je méně rušivý, než přepínač -i, ale stejně chrání před většinou chyb. V ukázce 4.14 je vidět příklad použití tohoto přepínače. Přestože je uživatel upozorněn o tom, že se bude něco mazat, už není upozorněn o tom, co se bude mazat.

```
n@t:/tmp/rmtest$ mkdir -p dir
n@t:/tmp/rmtest$ touch {a..b}.txt dir/{c..d}.txt
n@t:/tmp/rmtest$ /bin/rm -Ir ./*
/bin/rm: remove 3 arguments recursively? y
n@t:/tmp/rmtest$
```

Ukázka 4.14: rm s

Z našeho pohledu jsou oba tyto přepínače v praxi nepoužitelné. Jeden je moc striktní a druhý je zbytečný. Myšlenkou Comma-shell rm je zobrazit dostatečně komplexní souhrn toho, co se bude mazat tak, aby se uživatel mohl rychle rozhodnout, zda-li chce skutečně tyto soubory smazat, nebo se chce detailněji podívat na to, co všechno se bude mazat, například v podsložkách.

Další funkcionalitou Comma-shell rm je schopnost obnovovat mazané soubory. Tato funkcionalita je zajištěna použitím koše ze specifikace FreeDestkop. Pokud Comma-shell rm smaže nějaké soubory tak, že je přesune do koše, vytvoří soubor s informací kdy, odkud a co bylo smazáno. Uživatel tedy má možnost nejen obnovit z koše soubory, které potřebuje, ale je možné obnovit všechny soubory smazané jedním příkazem rm.

Implementace koše podle specifikace FreeDestkop není úplně triviální, proto Comma-shell rm používá externí nástroj pro práci s tímto košem, trash-cli.

V ukázce 4.15 je příklad podobný 4.14 a 4.13. Uživatel je informován o tom, které soubory se budou mazat a po potvrzení trvalého smazání je spuštěn originální příkaz rm.

```
n@t:/tmp/rmtest$ mkdir -p dir3
n@t:/tmp/rmtest$ touch {a..b}.txt dir3/{c..d}.txt f
n@t:/tmp/rmtest$ ,rm -r *
,rm: Files to remove:
,rm:    Directory: "dir3" (with 2 files)
,rm:    Removing file: "/tmp/rmtest/f"
,rm:    2 files with extension "txt"
,rm: Choose:
,rm:    [r]emove files
,rm:    [q]uit
,rm:    [t]rash files
,rm:    [s]how more files
r
,rm: /bin/rm -r a.txt b.txt dir3 f
```

Ukázka 4.15: rm s

V ukázce 4.16 se maže složitější adresářová struktura a uživatel se rozhodl vypsat si více souborů ke smazání. Comma-shell rm vypisuje pouze obsah adresářů ze složky, ve které se začíná mazat, pro úplný seznam si už uživatel musí vypsat obsah adresáře sám.

```
n@t:/tmp/rmtest$ mkdir -p dir3/dir4
n@t:/tmp/rmtest$ touch {a..b}.txt dir3/{c..d}.txt dir3/dir4/{e..f}.
    txt g
n@t:/tmp/rmtest$ ,rm -r *
,rm: Files to remove:
,rm:    Directory: "dir3" (with 3 files)
,rm:    Removing file: "/tmp/rmtest/g"
,rm:    2 files with extension "txt"
,rm: Choose:
,rm:    [r]emove files
,rm:    [q]uit
,rm:    [t]rash files
,rm:    [s]how more files
s
,rm: all files:
    a.txt
    b.txt
    dir3 (,: directory with 3 files)
        ./dir3/c.txt
        ./dir3/dir4 (,: directory with 2 files)
        ./dir3/d.txt
    g
,rm: Choose:
,rm:    [r]emove files
,rm:    [q]uit
,rm:    [t]rash files
,rm:    [s]how all files
q
```

Ukázka 4.16: rm s

V ukázce 4.17 je použití Comma-shell rm pro přesunutí souborů do koše. Soubory se pomocí trash-put mažou postupně, včetně složek, aby bylo snad-

nější upozorňovat na chyby.

```
n@t:/tmp/rmtest$ mkdir -p dir3/dir4
n@t:/tmp/rmtest$ touch {a..b}.txt dir3/{c..d}.txt dir3/dir4/{e..f}.
    txt g
n@t:/tmp/rmtest$ ,rm -r *
,rm: Files to remove:
,rm:    Directory: "dir3" (with 3 files)
,rm:    Removing file: "/tmp/rmtest/g"
,rm:    2 files with extension "txt"
,rm: Choose:
,rm:    [r]emove files
,rm:    [q]uit
,rm:    [t]rash files
,rm:    [s]how all files
t
,rm trash: trying to trash: "/tmp/rmtest/a.txt" ok
,rm trash: trying to trash: "/tmp/rmtest/b.txt" ok
,rm trash: trying to trash: "/tmp/rmtest/dir3" ok
,rm trash: trying to trash: "/tmp/rmtest/g" ok
,rm trash: Done. Use ,t or ,trash handle trashed bundles
```

Ukázka 4.17: ,rm - přesouvání do koše

V ukázce 4.18 je obnova souborů z předchozí ukázky 4.17. Po zavolání příkazu pro zobrazení balíčků smazaných souborů a vybrání jednoho podle času, místa a seznamu souborů si uživatel mimo jiné může nechat zobrazit detailní výpis souborů přesunutých do koše. Soubory může z koše pernamentně odstranit, nebo je obnovit. Pro odstranění se používá příkaz trashcli-rm. Pro obnovu souborů se používá příkaz trashcli-restore.

```
n@t:/tmp/rmtest$ ,t
,rm: This is a wrapper. Real trash is located at ~/.local/share/
    Trash
,rm: Choose the bundle:
    [1] ",rm -r *" from: /tmp/rmtest at 2017.04.28 20:06:38
    [q]uit
,rm: Chosen bundle 2017-04-28-20-06-38-461442374.
    [a]nother - select different bundle
    [s]how all trashed files
    [r]estore
    [d]iscard from the trash
    [q]uit
Showing all deleted files:
    -rw-rw-r-- 1 n n 0 dub 28 20:06 /tmp/rmtest/a.txt
    -rw-rw-r-- 1 n n 0 dub 28 20:06 /tmp/rmtest/b.txt
    drwxrwxr-x 3 n n 4096 dub 28 19:21 /tmp/rmtest/dir3
    -rw-rw-r-- 1 n n 0 dub 28 20:06 /tmp/rmtest/g
,rm: Chosen bundle 2017-04-28-20-06-38-461442374.
    [a]nother - select different bundle
    [s]how all trashed files
    [r]estore
    [d]iscard from the trash
    [q]uit
restore
,rm: restoring /tmp/rmtest/a.txt
,rm: restoring /tmp/rmtest/b.txt
,rm: restoring /tmp/rmtest/dir3
,rm: restoring /tmp/rmtest/g
,rm: bundle restored.
```

Ukázka 4.18: ,rm - obnova souborů

4.8.2 Další bezpečné příkazy

Cílem práce bylo implementovat základní příkazy ze sady GNU coreutils, podobně jako právě Comma-shell rm. Tyto příkazy zde budou popsány pouze stručně a bez ukázek, protože jsou velice podobné Comma-shell rm.

Za prvé jsou to příkazy chgrp, chmod a chown. todo

Za druhé jsou to příkazy cp, ln, mv, mkdir. todo: ochrana proti přepsání souboru.

Kapitola 5

Testování

5.1 Jednotkový test

Jednotkový test, anglicky unit test, je metoda software testování, při které jsou kontrolovány individuální jednotky zdrojového kódu testovány na funkčnost.

5.1.1 Testování shell skriptů

Ukážeme si zde dva způsoby testování shell skriptů.

Tím prvním jsou klasické jednotkové testy, kdy se skripty testují přes jejich rozhraní, které nabízí. Jsou to buď jednotlivé funkce, nebo celé skripty.

Druhým způsobem je testování, které simuluje uživatele v příkazové řádce a následně se testuje výstup pro určité řetězce. Tento postup je vhodný, pokud je hlavním cílem testování správná interakce v příkazové řádce a nikoli jen spouštění skriptu. Příkladem takového testování může být připojení se na vzdálený počítač a spuštění nějakých příkazů.

5.2 shUnit2

shUnit je framework na unit testy, tedy jednotkové testy, z rodiny frameworků xUnit. Framework nabízí funkce na testování podmínek, jako například test, zdali se dvě hodnoty rovnají. Tyto funkce potom uživatel může použít ve svých funkcí pro zjištění, jestli jeho kód funguje správně.

Framework spouští funkce které napsal uživatel. Vyhledává funkce pro jednorázové nastavení a závěrečné čistění, ale také nastavení a čištění před každou uživatelskou funkcí.

5.3 Bats

Bats je framework pro Bash, který splňuje specifikaci TAP, Test Anything Protocol.

Příklad testu Bats je v ukázce 5.1. Jednotlivé testy jsou v test blocích uvozených příkazem @test a popiskem testu. V takovém bloku lze volat externí skripty nebo funkce příkazem run. Bats při tom naplní obsah proměnných, obsahují návratové kódy, výstup jako celek, nebo pole jednotlivých řádek, pomocí kterých může uživatel kontrolovat, zdali byl jeho skript nebo funkce úspěšná.

```
@test "A test I don't want to execute for now" {
  skip
  run foo
  [ "$status" -eq 0 ]
}
```

Ukázka 5.1: Bats

5.4 Automatizované spouštění příkazů

Popsat jak se dají automaticky spouštět příkazy. At uz lokalne, nebo vzdalene. Popsat jak rekonstrukci z typescriptu, tak treba Tcl, Expect.

5.4.1 Expect

Expect je rozšířením jazyka Tcl. Hlavním účelem je automatizace interakcí s programy, které nabízejí nějakou formu textového terminálového uživatelského rozhraní.

Expect byl napsán Donem Libesem pro operační systémy z rodiny Unix, ale později se dostal i na systémy Microsoft Windows a další.

Základní práce v jazyce Expect je spouštění programů, například Bash, nebo i třeba ssh. Do takovýchto instancí je možné posílat vstup, stejně tak, jako by ho uživatel psal na klávesnici. Expect potom obsahuje konstrukce pro očekávání nějakého výstupu, podle kterého se může větvit zbytek skriptu.

5.4.2 Testování Comma-shell

Pro testování byl zvolen skriptovací jazyk Expect. Lze tedy velice snadno otestovat chování Comma-shellu a jeho částí pro různé vstupy.

Testy se skládají ze vstupů a z řetězců, které se očekávají na výstupu. Přes Expect se potom vytvoří nová instance Bashe, ve které se automaticky načte Comma-shell a postupně se posílají vstupy a sledují se výstupy pro očekáváné řetězce.

Testy jsou rozdělené na menší části, kde každá testuje jednu konkrétní věc. Přes skript lze pustit sérii všech testů.

V ukázce 5.2 je část testu v Expectu, která testuje, zdali se pro příkaz zavolá ShellCheck a příkaz se po zmáčknutí klávesy r skutečně spustí.

```
spawn bash

send "var=content\n"
send "echo itis\$var\n"

expect {
        timeout {
                        puts "CS_EXPECT_TIMEOUT 1
                            test_shellcheck.tcl"
                        exp_continue
        }
        ",: ShellCheck:" {
                puts "\nPASS 1"
        }
}

send "r"

expect {
        timeout {
                        puts "CS_EXPECT_TIMEOUT 2
                            test_shellcheck.tcl"
                        exp_continue
        }
        "itiscontent" {
                puts "\nPASS 2"
                exit 0
        }
}
```

Ukázka 5.2: Expect test

Závěr

Byl vytvořen nástroj Comma-shell, který umožňuje 1 2 3 todo

Comma-shell vytváří nástroje, které dávají nové možnosti pro implementaci užitečných do interaktivního shellu. Existují nástroje pro zobrazování informací v shellu, ale chyběl zde nástroj možnost komplexní analýzy příkazu ještě před jeho vykonáním.

Možnost využití

Comma-shell využije nejen začátečník, ale i zkušený uživatel. Přestože to v takto rané fázi projektu zřejmě nebude, tak v budoucnu by například bylo možné, aby se firmách, nebo školících zařízeních připravil Comma-shell tak, aby zajistil bezpečnost a informoval uživatele.

Další práce

Základem Comma-shell frameworku pro hooky je připravit prostředí pro další vývoj. Toto prostředí funguje dobře a až na opravu případných chyb jej není potřeba příliš rozšiřovat. Je možné, že se v budoucích verzích Bashe objeví nové debuggovací nástroje tak, že bude možné jádro Comma-shellu přepsat.

Comma-shell debugger a sada bezpečných příkazů mají přirozeně místo pro další vývoj a vylepšování. Především to bude reagování na podněty uživatelů.

Literatura

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

Příloha **B**

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
exe ..... adresář se spustitelnou formou implementace
src
    impl ..... zdrojové kódy implementace
    thesis ..... zdrojová forma práce ve formátu LaTeX
text ..... text práce
    thesis.pdf ..... text práce ve formátu PDF
    thesis.ps ..... text práce ve formátu PS
```