VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO ANALÝZU SKRIPTŮ UNIXOVÉHO SHELLU

A TOOL FOR SHELL-SCRIPT ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ HOLOMEK
AUTHOR

VEDOUCÍ PRÁCE Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Bakalářská práce se zabývá analýzou skriptů. V dnešních distribucích Linuxu nebo jiných operačních systémů založených na myšlence Unixu je mnoho nastavení konfigurováno pomocí skriptů. Práce směřuje k vytvoření programu, který takové skripty analyzuje a sdělí uživateli jejich funkci a také bezpečnost. Na základě průzkumu se práce zaměřuje na jazyk Bash.

Abstract

This bachelor's thesis deals with analysis of scripts. In the current distributions of Linux or other operating systems based on the Unix principals, there is made a lot of configuration by using scripts. The main objective of this work is to create a program that analyses the scripts and also reports theirs functionality and safety to a user. According to my research the work is aimed to Bash language.

Klíčová slova

statická analýza, Bash, skript, analýza skriptů

Keywords

static analysis, Bash, script, analysis of scripts

Citace

HOLOMEK, Jiří: Nástroj pro analýzu skriptů unixového shellu. Bakalářská práce. Brno, FIT VUT, 2011.

Nástroj pro analýzu skriptů unixového shellu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Aleše Smrčky, Ph.D.

. . . . . .
Jiří Holomek
18. května 2011

Poděkování

Děkuji vedoucímu své práce, Ing. Aleši Smrčkovi, Ph.D., za odbornou pomoc, kterou mi během mé práce ochotně poskytoval. Dále pak děkuji svým rodičům za podporu a zázemí, které mi ke studiu vytvořili.

© Jiří Holomek, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **5**

2 Statická analýza **6**
2.1 Stavba modelu . . . 6
2.2 Analýza modelu . . . 9
2.3 Prezentace výsledků . . . 9

3 Unixové shelly **10**
3.1 Co to vlastně je shell? . . . 10
3.2 Přehled unixových shellů . . . 10
3.3 Průzkum popularity . . . 11
3.4 Bash . . . 12
3.4.1 Proměnné . . . 13
3.4.2 Uvozovky . . . 14
3.4.3 Aliasy . . . 14
3.4.4 Subshelly . . . 15
3.4.5 Přesměrování . . . 15
3.4.6 Speciální znaky . . . 16
3.4.7 Substituce proměnných . . . 16
3.4.8 Aritmetická expanze . . . 17
3.4.9 Substituce příkazů . . . 17

4 Návrh programu **18**
4.1 Specifikace požadavků . . . 18
4.1.1 Požadavky na program . . . 18
4.1.2 Kontrolované vlastností . . . 19
4.1.3 Vlastnosti programu . . . 21
4.2 Návrh systému . . . 21
4.2.1 Úprava kódu před vlastní analýzou . . . 22
4.2.2 Dotazovací jazyk . . . 22
4.2.3 Řešení dílčích problémů . . . 23
4.2.4 Vlastní řešení příkazů . . . 26

5 Implementace programu **27**
5.1 Použité nástroje . . . 27
5.2 Jednotlivé moduly . . . 28

6 Závěr **30**

A Rozdělení příkazů **32**

B Obsah CD **34**

Seznam obrázků

2.1 Schéma statické analýzy 6
2.2 Syntaktický strom 8

3.1 Přehled hlavních unixových shellů. 11
3.2 Procentuální porovnání v počtu nalezených skriptů 12
3.3 Procentuální porovnání v celkové délce nalezených skriptů 13

5.1 Algoritmus pro zpracování parametrů načítaných ze souboru. 28

Seznam tabulek

2.1 Definice pravidel pro lexémy 7
2.2 Gramatika pro parsování 8

3.1 Výsledky průzkumu 12
3.2 Přehled možností substituce proměnných 16

5.1 Přepínače pro ovládání programu 29

Kapitola 1

Úvod

V dnešních distribucích Linuxu nebo jiných operačních systémů založených na myšlence Unixu je mnoho nastavení konfigurováno pomocí skriptů - ať je to start systému, nastavení prostředí nebo zacházení s jednotlivými službami systému.

Chceme-li se dozvědět, co daný skript vykonává, než ho spustíme, máme dvě možnosti. Buď si daný skript otevřeme a zjistíme „ručně“, co dělá, nebo použijeme nástroje statické analýzy. První možnost funguje dobře pro zkušené uživatele a v případech, kdy daný skript má pouze pár řádků. Je-li skript rozsáhlý, je lepší použít nástroje statické analýzy. Tyto nástroje provedou lexikální, syntaktickou a sémantickou analýzu, sledují tok dat v programu a dokážou uživatele upozornit na nebezpečné konstrukce a příkazy.

Cíl práce

Výsledkem tohoto projektu by měl být program, který umí analyzovat skripty jednoho z unixových shellů a říci uživateli, co vlastně daný skript dělá a jak je pro něj „bezpečný“.

Struktura písemné práce

Práce je rozdělena celkem do šesti kapitol. Tyto kapitoly se věnují seznámení s problematikou statické analýzy, unixovým shellům, návrhu systému, implementaci a zhodnocení dosažených výsledků.

- **Kapitola 1:** Úvod
- **Kapitola 2:** Statická analýza - součásti, postupy
- **Kapitola 3:** Unixové shelly - jejich představení, průzkum popularity, Bash
- **Kapitola 4:** Návrh systému pro statické analyzování skriptů psaných v jazyku Bash
- **Kapitola 5:** Implementace systému - nástroje pro implementaci, vlastní implementace
- **Kapitola 6:** Závěr

Kapitola 2

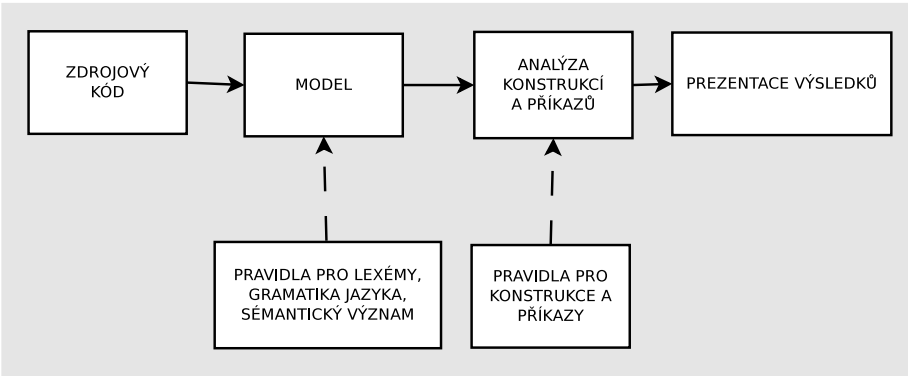
Statická analýza

Tato kapitola zpracovává teoretický základ pro oblast statické analýzy. Text byl zpracován dle [1] a [2]. Průběh statické analýzy popisuje obrázek 2.1.

Nástroje pro statickou analýzu mohou odhalit pouze chyby a problémy, které je možné zjistit ze zdrojového kódu programu. Nejčastěji se provádí statická analýza právě nad zdrojovým kódem. Nelze odchytit problémy, které budou způsobeny uživatelským vstupem do systému za běhu aplikace, nebo samotným během aplikace (analýza prováděná za běhu systému se nazývá dynamická analýza).

Součásti statické analýzy jsou:

- Stavba modelu: zahrnuje lexikální analýzu, syntaktickou analýzu a sémantickou analýzu.
- Analýza modelu: sledování toku dat v programu, nadefinování pravidel pro kontrolu, interpretace jednotlivých částí modelu, prezentace výsledků.



Obrázek 2.1: Schéma statické analýzy

2.1 Stavba modelu

Stavba modelu zahrnuje jednotlivé kroky procesu transformace zdrojového kódu na derivační strom. Tato část se zabývá jejich popisem.

Lexikální analýza

Jedná se o nástroj, který pracuje nad zdrojovým kódem a transformuje jej na posloupnost tzv. `lexémů` (angl. `tokenů`), což jsou prvky zdrojového jazyka. Většinou ignoruje „bílé znaky“[1] a komentáře. Pravidla pro `lexémy` jsou často definována pomocí regulárních výrazů.

Mějme příklad:

```
if (ret) //komentář
  mat[x][y] = END_VAL;
```

Pokud nadefinujeme pravidla pro lexémy, jaké jsou uvedeny v tabulce 2.1, výsledná sekvence lexémů bude:

IF LPAREN ID(ret) RPAREN ID(mat) LBRACKET ID(x) RBRACKET LBRACKET ID(y) RBRACKET EQUAL ID(END_VAL) SEMI

Pro tvorbu lexikálních analyzátorů je možné používat běžně dostupné automatizované nástroje, jako je např. LEX [3].

Hodnota lexému	**Funkce**
`if`	`{ return IF; }`
`(`	`{ return LPAREN; }`
`)`	`{ return RPAREN; }`
`[`	`{ return LBRACKET; }`
`]`	`{ return RBRACKET; }`
`=`	`{ return EQUAL; }`
`;`	`{ return SEMI; }`
`[ \t\n]+`	`{ Ignoruj bílé znaky }`
`\/\/.*`	`{ Ignoruj komentář }`
`[A-Za-z][A-Za-z0-9]*`	`{ return ID; }`

Tabulka 2.1: Definice pravidel pro lexémy

Syntaktická analýza

Úkolem syntaktického analyzátoru je určit syntaktickou strukturu zdrojového programu. To znamená, že se syntaktický analyzátor snaží sestrojit pro dané slovo lexikálních symbolů (reprezentující zdrojový program) derivační strom.

Syntaktický analyzátor (angl. `parser`) využívá bezkontextovou gramatiku k určení správného pořadí lexému ve zdrojovém kódu.

- **Bezkontextová gramatika** je určena konečnou množinou neterminálů, konečnou množinou terminálů, která nemá společné prvky s množinou neterminálů, počátečním neterminálem a konečnou množinou přepisovacích pravidel typu $X \rightarrow \alpha$, (X přepiš na α), kde X je neterminál a α je řetězec z neterminálů a terminálů.

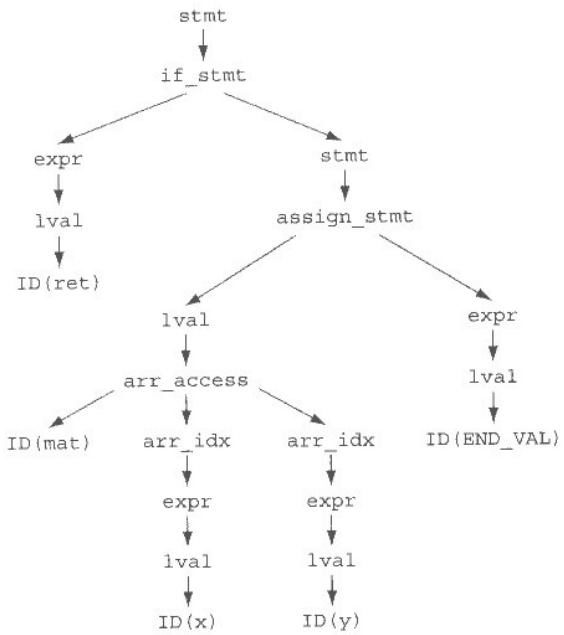
[1] Označení pro mezeru, tabulátor a nový řádek

Pravidla
`stmt := if_stmt	assign_stmt`
`if_stmt := IF LPAREN expr RPAREN stmt`
`expr := lval`
`assign_stmt := lval EQUAL expr SEMI`
`lval := ID	arr_access`
`arr_access := ID arr_idx+`
`arr_idx := LBRACKET expr RBRACKET`

Tabulka 2.2: Gramatika pro parsování

Pro tvorbu syntaktických analyzátorů je možné používat běžně dostupné automatizované nástroje, jako je např. YACC [4].

V tabulce 2.2 jsou definovaná pravidla pro parsování kódu z předchozí sekce. Na obrázku 2.2 pak můžeme pozorovat, jak vypadá syntaktický strom po použití námi nadefinovaných pravidel.



Obrázek 2.2: Syntaktický strom

Sémantická analýza

Sémantická analýza provádí kontrolu nejrůznějších sémantických aspektů programu, jako např. deklarovanost proměnných (u některých jazyků to není nutné). Postupně prochází derivační strom a kontroluje získané symboly či skupiny symbolů.

Můžeme díky ní odhalit chyby, jako je přiřazování textových řetězců do proměnných, které mohou obsahovat pouze číselné hodnoty.

2.2 Analýza modelu

Pro to, abychom mohli interpretovat, co program vykonává, je nutné znát význam všech konstrukcí daného jazyka. Díky tomu je možné správně určit, kde program operuje, s čím zachází, nejsou-li v něm sémantické chyby a co ovlivňuje.

Sledování toku dat

Důležitou součástí statické analýzy je sledování toku dat v programu. Je nutné sledovat obsahy proměnných, je-li to v rámci pouhého analyzování zdrojového kódu možné. Díky udržování aktuálního stavu proměnných je často možné vyhodnotit, které konstrukce kódu se provedou, které ne, jestli skončí program předčasně, nezacyklí se aj.

2.3 Prezentace výsledků

Pokud program narazí na konstrukci, o které není schopný rozhodnout, co přesně vykoná, pak je třeba uživateli o této konstrukci podat zprávu.

U statické analýzy platí, že je lepší uživateli předložit všechny konstrukce, kde program nemůže stoprocentně rozhodnout, co se bude vykonávat, neboť taková místa značí potenciální bezpečnostní rizika a zaslouží si uživatelovu pozornost.

Není-li nástroj schopný rozhodnout, která část větvení se provede, popř. jaký počet iterací, pak je třeba udržovat v kontextu všechny možné varianty.

Mějme takový příklad,

```
if [ "$USER" == "root" ]; then
  rm file1
else
  rm file2
fi
```

pak nástroje statické analýzy nemohou určit obsah proměnné `USER` (proměnná, která je inicializovaná shellem a obsahuje jméno aktuálně přihlášeného uživatele) a je třeba uživatele varovat o všech možných rizicích, tedy že se může z disku mazat buď `file1` nebo `file2` a že výsledek této operace záleží na konstrukci `if [ "$USER" == "root" ]`.

Kapitola 3

Unixové shelly

Tato kapitola zpracovává teoretický základ pro oblast unixových shellů. Je zde uveden přehled nejčastěji používaných shellů v současných unixových distribucích. Dále je rozvedeno, na jaký shell je práce zaměřená, tento je blíže popsán. Kapitola čerpá z [5], [6] a [7].

3.1 Co to vlastně je shell?

Shell je prostředek komunikace uživatele s operačním systémem. Historicky prvním a dodnes nejdůležitějším je Bourne shell, pojmenovaný po svém autorovi A. S. Bournovi, který jej navrhl a naprogramoval na přelomu 60. a 70. let. Principy Bourne shell přejaly všechny následující příkazové interprety, obvykle jde pouze o jiný uživatelský přístup.

Shell vytváří prostředí příkazového řádku, do kterého uživatel zadává názvy příkazů, které chce spustit. Shell tyto příkazy interpretuje, spouští odpovídající programy a umožňuje sledovat jejich výstup. Umožňuje příkazům předávat parametry, seskupovat je, slučovat příkazy do skriptů a podobně.

Příkazy se volají tak, že se napíše jméno příkazu a za něj parametry. Před spuštěním příkazu provádí shell nad příkazovou řádkou několik transformací: nahrazuje proměnné jejich obsahem, rozbaluje *wildcardové*[1] výrazy na seznamy jmen souborů, které jim odpovídají, rozděluje řádky na jednotlivé argumenty a další. Pokud napíše uživatel *mv * dir*, je to shell a nikoliv program *mv*, kdo nahradí hvězdičku seznamem všech souborů v aktuálním adresáři.

Pokud chceme předat nějakému příkazu skutečně hvězdičku nebo jiný znak se zvláštním významem, nejjednodušší cestou je dát ji do uvozovek: shell při zpracování příkazové řádky uvozovky odstraní, ale jejich obsah ponechá nezměněný. Přesné chování při zpracování příkazové řádky a seznam znaků se speciálním významem závisí na konkrétním shellu.

3.2 Přehled unixových shellů

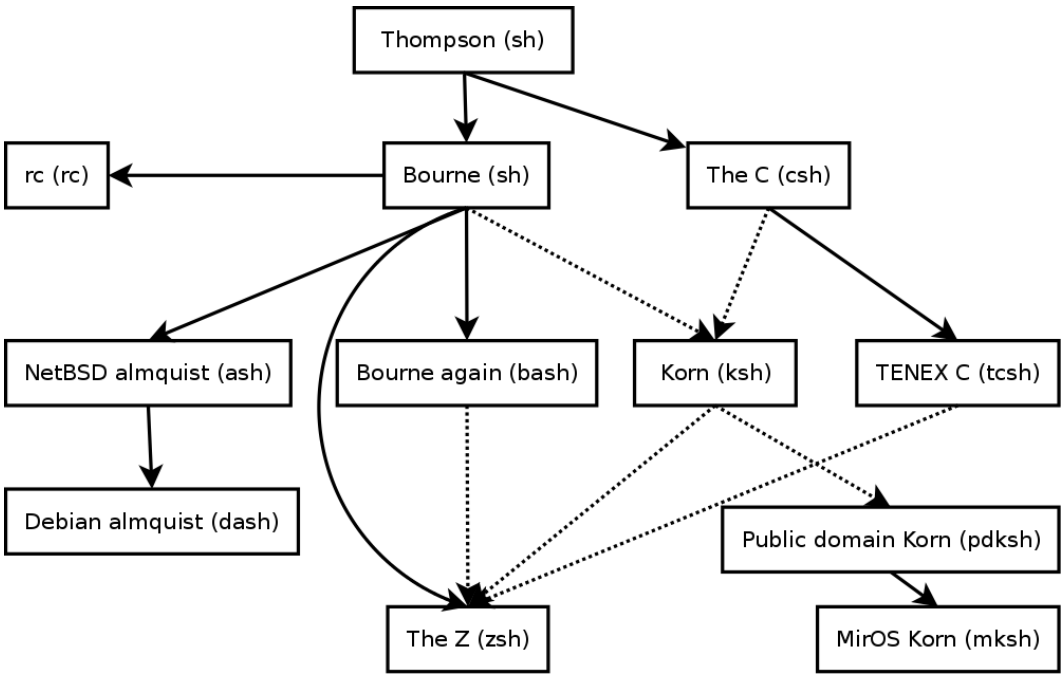
Nejznámějšími variantami shellu jsou (řazeny dle data vzniku):

- `Bourne shell` /bin/sh – historicky nejuznávanější shell, v Unixu je od Verze 7
- `C shell` /bin/csh – jeho syntaxe vychází z jazyka C

[1] Příkazy, v nichž se vyskytuje znak *, který není v uvozovkách, nebo před ním není zpětné lomítko

- `TC shell` /bin/tcsh – rozšíření C shellu o editaci příkazového řádku ve stylu editoru `emacs`
- `Korn shell` /bin/ksh – rozšíření Bourne shellu, zdola kompatibilní, přidává některé užitečné věci z C shellu (práci s procesy, historii a editaci příkazové řádky aj.)
- `Bourne again shell` /bin/bash – obsahuje velmi dobrou podporu interaktivní práce. Více viz kapitola 3.4.

Přehled shellů dle jejich vývoje popisuje obrázek 3.1.



Obrázek 3.1: Přehled hlavních unixových shellů.

3.3 Průzkum popularity

Než jsem se rozhodl pro zaměření na konkrétní unixový shell, provedl jsem malý průzkum, abych zjistil, které shelly se používají a v jakém rozsahu.

Výzkum jsem provedl na sedmi distribucích linuxu: PCLinuxOS, Linux Mint, Mandriva, Debian, openSuse, Fedora a Ubuntu.

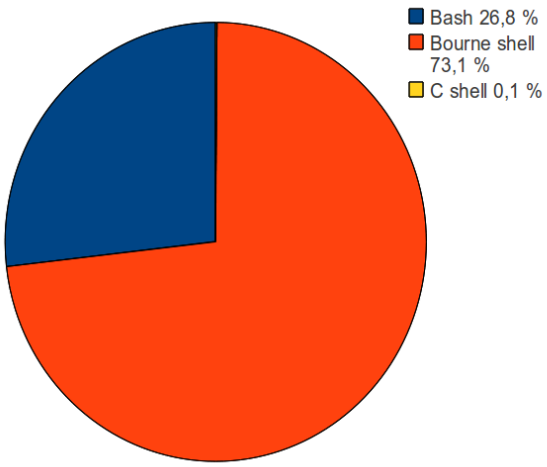
Napsal jsem skript v jazyku `Bash`, který postupně prohledal vybrané složky (zaměřil jsem se na složky /etc a /boot kvůli konfiguračním skriptům), zjistil o jaký typ souboru se jedná a v případě, že skript byl psaný v jazyku `Bash`, `sh`, `ksh`, `csh` nebo `tcsh`, shromáždil statistiky o souboru. Kontrolovanými faktory byl počet skriptů a jejich celková délka.

Výsledky jsou v tabulce 3.1. Vzhledem k tomu, že se v rámci testování neobjevil ani jeden skript shellu `ksh` a `tcsh`, ve výsledcích je neuvádím. Na obrázku 3.3 lze pozorovat porovnání užívání shellů, co se počtu souboru týká, na obrázku 3.2 lze pozorovat porovnání v celkové délce daných skriptů.

Distribuce	Bourne Shell		Bash		C shell	
	Počet skriptů	Celková délka	Počet skriptů	Celková délka	Počet skriptů	Celková délka
Ubuntu	313	600 120	21	1 830	0	0
Debian	495	1 253 672	12	25 726	0	0
PCLinuxOS	302	576 803	196	752 827	1	299
openSUSE	283	984 846	162	1 126 345	0	0
Mandriva	290	558 108	155	653 723	0	0
Linux Mint	304	516 878	15	22 984	0	0
Fedora	227	702 936	251	822 199	2	1432

Tabulka 3.1: Výsledky průzkumu

Je patrné, že shelly `ksh` a `tcsh` se pro konfigurační účely ve výše zmíněných distribucích nepoužívají. `Csh` byl použit ve třech případech, ale v rámci ostatních výsledků je toto číslo zanedbatelné. Nejpoužívanější je stále Bourne shell (`sh`), ale i skriptů jazyka `Bash` se v distribucích nachází hodně.



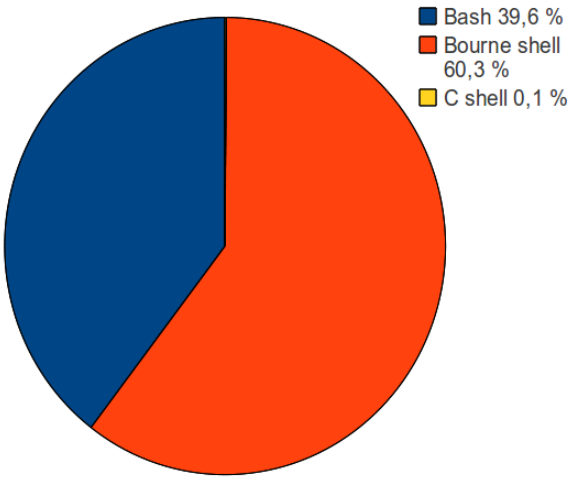
Obrázek 3.2: Procentuální porovnání v počtu nalezených skriptů

Moje práce ze zaměří na jazyk `Bash`, neboť obsahuje mnoho rozšíření v porovnání s klasickým Bourne shellem a navíc je s ním kompatibilní (až na pár výjimek).

Ve všech distribucí v testu byl po instalaci implicitně nastavený `Bash` jako defaultní shell.

3.4 Bash

Bash je `POSIX` shell s řadou rozšíření a slouží jako interpret pro příkazový řádek. Naprogramoval ho Brian Fox pro projekt `GNU` a jeho název je akronymem pro Bourne again shell, což poukazuje na jeho základ v dříve nejpoužívanějším unixovém shellu Bourne shell `(sh)`.



Obrázek 3.3: Procentuální porovnání v celkové délce nalezených skriptů

Je používán jako implicitní příkazový interpret v systémech postavených na linuxovém jádře, stejně jako i v Mac OS X. Je možné ho použít i v systému Microsoft Windows za použití subsystému pro unixové aplikace (SUA), nebo emulace `POSIX` pomocí softwaru `Cygwin` a `MSYS`.

Základ syntaxe příkazů v Bashi je převzat z původního Bourne shellu. Také převážná většina `sh` skriptů je v Bashi spustitelná bez jakékoliv modifikace kódu až na pár výjimek, které vznikají jinou interpretací méně používané syntaxe příkazů nebo jinou implementací těchto systémových příkazů. Bash navíc převzal nápady z `Korn shellu` a `C shellu`, například editaci jednotlivých řádků, historii příkazů, proměnné `$RANDOM` a `$PPID` a POSIXové dynamické vkládání příkazů pomocí syntaxe `$(prikaz)`. Při psaní syntaxe nám pak práci nejvíce ulehčí stisk klávesy tabulátor, čímž Bash automaticky doplní konec napsaného příkazu, cesty k souboru nebo proměnné, případně nám ukáže všechny možnosti doplnění.

Na rozdíl od Bourne shellu má Bash mnoho rozšíření i v syntaxi příkazů. Například dokáže provádět celočíselné operace bez volání externích procesů. Za tímto účelem byla vytvořena syntaxe, které se blíže věnuje kapitola 3.4.8. Bash také zjednodušil přesměrování vstupů a výstupů tak, jak to v tradičním Bourn shellu nebylo možné, více v kapitole 3.4.5.

3.4.1 Proměnné

Proměnné jsou obecně pouze jednoho datového typu – řetězec znaků. Některé z nich jsou určeny jen pro čtení, a to buď implicitně nastavením systému, nebo příkazem *readonly*. Lze definovat pomocí vestavěného příkazu *declare* omezení obsahu proměnné (pole, číselná hodnota aj.). Při číselných operacích je rozhodujícím faktorem, jestli obsahuje proměnná pouze číslice. Obsahuje-li proměnná i jiné znaky než číslice, pak se její hodnota v aritmetickém výrazu považuje za nulu.

Proměnné můžeme rozdělit do tří skupin.

- Vnitřní proměnné shellu. O jejich inicializaci se stará shell.

  - např. $HOME, $HOSTNAME, $LANG, $PATH, $SHELL aj.

- Uživatelské proměnné. Stejně jako vnitřní proměnné se skládají pouze z alfanumerických znaků a podtržítka.

- Proměnné speciálního významu, skládají se ze speciálních znaků.

  - $$ - PID[1] shellu
  - $! - PID[1] posledního procesu, který byl spuštěn na pozadí
  - $? - návratová hodnota posledního dokončeného procesu
  - a mnoho dalších

3.4.2 Uvozovky

Jednoduché

Uzavřením sekvence znaků do jednoduchých uvozovek zajistíme, že se žádný ze znaků nebude expandovat. Chceme-li napsat do takového řetězce znak jednoduchých uvozovek, je třeba před něj vložit zpětné lomítko.

Dvojité

Uzavřením sekvence znaků do dvojitých uvozovek zajistíme, že znaky z této sekvence se nebudou expandovat, až na proměnné, viz kapitola 3.4.7.

```
a=5
# Následující příkaz vypíše: Obsah proměnné a je 5
echo "Obsah proměnné a je $a"
```

Zpětné

Zpětné uvozovky znamenají, že se v textu bude celá sekvence nahrazovat výsledkem výrazu, který je umístěný v rámci zpětných uvozovek. Konstrukce příkazu: `‘vyraz‘`

```
# Do proměnné a se uloží výsledek příkazu ls
a=‘ls‘
```

3.4.3 Aliasy

Alias je vestavěný příkaz, který je schopný přepsat defaultní chování příkazů nebo tvořit příkazy vlastní. Používá se pro zjednodušení práce uživatele, aby nemusel často vypisovat stejné konstrukce. Např. `alias ls='ls --color=auto'`. Tento příkaz zajistí, aby výsledek příkazu ls byl defaultně zobrazován barevně.

Stejně tak jako dobrý pomocník, je to i velká bezpečnostní riziko, kterou je nutné hlídat při statické analýze.

[1] Identidikátor procesu, reprezentovaný číslem.

3.4.4 Subshelly

Chceme-li vykonat některé příkazy mimo hlavní běh skriptu, je možné tak učinit s využitím subshellu. Pokud potřebujeme výsledek takových operací použít v běhu hlavního skriptu, pak je třeba použít filedescriptor (český ekvivalent je popisovač souborů) pro přesměrování.

3.4.5 Přesměrování

Je možné zapisovat do souborů, nebo z nich číst. V Bashi jsou vždy otevřené minimálně 3 soubory: `stdin` (standartní vstup), `stdout` (standartní výstup) a `stderr` (standartní chybový výstup). Tyto a další otevřené soubory mohou být přesměrovány. Přesměrováním míníme odchycení výstupu ze souboru, příkazu, programu, skriptu, popř. bloku příkazů (cykly, nebo sekvence příkazů uzavřená do složených závorek) a poslání na vstup jinému souboru, příkazu, programu nebo skriptu.

Bash disponuje velkým množstvím způsobů, jak přesměrovávat vstupy a výstupy. Není to jen obyčejné zapisování do souboru a čtení z něj, nabízí nám i `roury`, `pojmenované roury`, `here documenty` a možnost přesměrování `filedescriptorů`.

Na některé tyto funkce se blíže zaměřím v textu, neboť jejich význam pro statickou analýzu je stěžejní.

Roury

Slouží pro přeposlání výstupu jednoho příkazu na vstup příkazu druhého.

```
# Následující příkaz přepošle výstup příkazu ls na příkaz cat
ls | cat
```

Pojmenované roury

Je to speciální typ souboru existujícího v souborovém systému. Často se používá, pokud chceme roury využívat opakovaně.

Tvoří se příkazem `mkfifo`, následovaným jedním parametrem, jménem roury. Příkazem `ls -l` se můžeme přesvědčit o tom, že se roura skutečně vytvořila a z výpisu je vidět, že se nejedná o klasický soubor, nýbrž o `rouru`, označenou písmenem `p` (z angl. pipe).

Filedescriptor

Filedescriptor je reprezentován číslem a slouží k identifikaci otevřeného souboru. Standartní vstup, standartní výstup a standartní chybový výstup mají označení 0, 1 a 2.

Přesměrování standardního výstupu a standardního chybového výstupu v tu samou chvíli lze udělat pomocí příkazu `&>`. Dále je možné popisovače souborů duplikovat a přesměrovávat (Tyto konstrukce jsou implementované až v Bashi verze 4.0.). Dá se díky nim přeposlat výsledek operací ze subshellu do hlavní části skriptu.

Here document

Here document (nemá český ekvivalent) slouží pro přesměrování víceřádkového textu na vstup daného příkazu. Jsou tři možnosti jak použít here document.

- Tvar zarážky: `<<text`
  V tomto případě se provede v sekvenci here documentu substituce výrazů a proměnných.

- Tvar zarážky: `<<-text`
  Stejné jako v předchozím případě, ale tabulátory na začátku řádků se nebudou brát v potaz při vyhodnocování. Používá se pro lepší přehlednost v kódu.

- Tvar zarážky: `<<"text"`
  V tomto případě se dané aplikaci předá here document čistě jako text.

3.4.6 Speciální znaky

Až na níže zmíněné znaky `Bash` interpretuje znak sám o sobě. Chceme-li i tyto znaky takto interpretovat, je třeba před ně umístit zpětné lomítko. V některých případech stačí ohraničení uvozovkami, více viz kapitola 3.4.2.

```
| & ; < > ( ) $ ‘ \ " ’ <mezera> <tabulátor> <nový_řádek>
```

3.4.7 Substituce proměnných

Před vlastním vyhodnocení příkazu se provádí substituce proměnných. Jako proměnná je vnímána sekvence znaků, která začíná znakem $ a je následována názvem proměnné, nebo složenými závorkami, ve kterých může být buď název proměnné, nebo výraz pro vyhodnocení viz tabulka 3.2.

	Parametr Inicializovaný Neprázdný	**Parametr Inicializovaný Prázdný**	**Parametr Neinicializovaný**
`${parametr:-word}`	substituce parametru	substituce word	substituce word
`${parametr-word}`	substituce parametru	substituce NULL	substituce word
`${parametr:=word}`	substituce parametru	přiřaď word	přiřaď word
`${parametr=word}`	substituce parametru	substituce NULL	přiřaď word
`${parametr:?word}`	substituce parametru	chyba, konec	chyba, konec
`${parametr?word}`	substituce parametru	substituce NULL	chyba, konec
`${parametr:+word}`	substituce word	substituce NULL	substituce NULL
`${parametr+word}`	substituce word	substituce word	substituce NULL

Tabulka 3.2: Přehled možností substituce proměnných

Složené závorky se často používají, pokud potřebujeme oddělit proměnnou od okolního textu:

```
x=promen
#Vypíše "Substituce promennych"
```

```
echo "Substituce ${x}nych"
#Vypíše "Substituce promen nych"
echo "Substituce $x nych"
```

3.4.8 Aritmetická expanze

Konstrukce pro aritmetickou expanzi vypadá takto: `$((vyraz))`. Před vlastní aritmetickou expanzí se provede substituce proměnných a výrazů v uvozovkách. Je-li ve výrazu obsažen textový řetězec, pak ho `shell` automaticky bere jako proměnnou a hledá její obsah, tzn. že není nutné před proměnné dávat znak `$`.

```
# zopakuje příkaz stokrát
x=100
while [ $x -gt 0 ]
do
    prikaz
    x=$(($x-1))
done
```

3.4.9 Substituce příkazů

Konstrukce pro substituci příkazů vypadá takto: `$(prikaz) nebo ‘prikaz‘`. Uloží se do proměnné výsledek příkazu. Používá se v případech, kdy je potřeba pracovat s výsledkem příkazu vícekrát, a proto nám nestačí použití `roury` pro jednorázové přesměrování výstupu na vstup jiného příkazu.

Kapitola 4

Návrh programu

V této kapitole se věnuji návrhu cílového programu. Jsou kladeny požadavky a specifikuji vlastnosti, které bude výsledný program ověřovat. Dále se věnuji návrhu cílové aplikace a zmiňuji jednotlivé části, které je třeba vzít v úvahu před vlastní implementací.

4.1 Specifikace požadavků

Tato specifikace plyne z požadavků na výsledný program, které byly nastíněny v zadání bakalářské práce.

4.1.1 Požadavky na program

Program by měl podat uživateli komplexní zprávu o tom, s čím daný skript operuje a je-li syntakticky správně napsán.

- program musí být schopný otestovat všechny vlastnosti, specifikované v kapitole 4.1.2
  - měl by umožňovat i výpis dílčích vlastností dle skupin, ne jen komplexní přehled o všech příkazech ve skriptu
- možnost spuštění syntaktické analýzy nad skriptem
  - program by měl uživateli říct, je-li skript syntakticky v pořádku a pokud ne, tak vypíše jméno skriptu, řádek a konstrukci, která je syntakticky špatná
- možnost otestovat i více než jeden skript
  - dávkové zpracování buď automaticky od místa spuštění, nebo dané parametrem cesty k souboru/složce (bude-li zadána složka, popř. automatický test od místa spuštění, pak program bude vyhodnocovat pouze textové soubory)
- dobrá orientace ve výsledcích
  - výpis programu by měl být správně a přehledně formátován, kromě množiny operací, kterou daný skript vykonává, by se u každého příkazu mělo vypsat číslo řádku a jméno skriptu

4.1.2 Kontrolované vlastností

V této kapitole jsou popsány jednotlivé vlastnosti, které bude sledovat výsledný program. Konkrétní rozřazení jednotlivých příkazů do skupin obsahuje příloha A.

- Soubory
  - do kterých se bude zapisovat,
  - ze kterých se bude číst,
  - které se budou kopírovat,
  - které se budou odstraňovat,
  - které se budou porovnávat,
  - jejichž obsah se bude upravovat,
  - které se budou přesouvat,
  - které se budou přejmenovávat,
  - které se budou komprimovat/dekomprimovat,
  - kterým jsou měněny časová razítka,
  - test vlastností souboru.
- Složky
  - nová složka,
  - které se budou odstraňovat,
  - na jejichž obsah se bude nahlížet.
- Síť
  - nastavování rozhraní,
  - posílání dat přes síť,
  - sledování statistik na síti,
  - stahování webu,
  - hledání cíle v síti,
  - přístup na url,
  - test dosažitelnosti,
  - přihlašování přes SSH,
  - dotazy na DNS.
- Práva
  - změna přístupových práv,
  - vužívání root přístupu,
  - spouštění příkazů pod jiným uživatelským účtem.

- Cesty
  - ve kterých skript operuje,
  - do kterých nahlíží,
  - práce se zásobníkem.
- Systém
  - nastavení data,
  - nastavení limitů,
  - nastavení systémového jména,
  - vypnutí/uspání/restart/odhlášení.
- Procesy
  - sledování,
  - zastavení,
  - změna priority.
- Tiskárna
  - tisk souborů,
  - manipulace s tiskovou frontou.
- Uživatelé
  - nový uživatel/skupina,
  - změna hesla,
  - modifikace existujících účtů,
  - sledování zalogovaných uživatelů,
  - změna skupiny/vlastníka u souboru/složky.
- Otevírání
  - souborů,
  - spouštění skriptů,
  - periodické spouštění příkazů.
- Ostatní
  - manipulace s diskem,
  - formát diskety,
  - připojování médií,
  - kompilování,
  - ostatní.

4.1.3 Vlastnosti programu

Co program bude umět:

- Podá uživateli zprávu o příkazech, které skript vykonává, tento výpis dále bude rozdělen na dva oddíly, a to na ty, které se vykonají vždy, a na ty, které se vykonají dle toho, kde se daný skript spustí.
- Bude schopen zjistit, zda skript pomocí příkazu *alias* neschovává jiné příkazy pod jmény defaultních příkazů, nebo nevytváří příkazy vlastní. Mějme např. „alias ls='rm –rf /'“. Pokud potom ve skriptu spustíme příkaz *ls*, tak se nám nevypíše obsah složky, ale smažou se nám všechna data.
- Program pokaždé provede syntaktickou analýzu nad skriptem a řekne uživateli, je-li syntakticky v pořádku. Pro vlastní analýzu je nutné, aby byl skript syntakticky správně napsán.
- Program bude sledovat složku, ve které daný skript operuje, a bude-li to možné, bude ji udržovat v kontextu s ostatními příkazy (znemožnit tuto vlastnost mohou konstrukce, u kterých program staticky nemůže říci, které se vykonají a které ne, a proto uživateli nabídne více možností).
- Program bude schopen uchovávat obsah proměnných, ale pouze v případech, kdy se jejich obsah dá zajistit v rámci statické analýzy. Např. `var=5`.
- Program bude schopen uživateli říci, které procesy spuštěné skriptem mohou běžet dál po jeho ukončení (procesy spouštěné na pozadí).

Co program nebude umět:

- Nemůže zachytit věci, které lze analyzovat za běhu skriptu - neumožňuje dynamickou analýzu.
- Nebude automaticky vyhodnocovat skripty, které jsou volány v rámci analyzovaného skriptu. Uživateli bude pouze dána zpráva o tom, které další skripty se spouští a bude na něm, zda provede analýzu i na těchto skriptech.
- Nelze uživateli říci, zda skript přepisuje obsahy existujících souborů, nebo vytváří soubory nové. Tím je myšleno, že program nebude zjišťovat, zda soubor existuje ještě před tím, než do něj skript zapíše.
- Nelze uživateli říci, zda je skript oprávněn pro některé operace. Např. přistupuje-li skript do složky, kde přihlášený uživatel nemá právo pro čtení.

4.2 Návrh systému

Systém provede na požadovaném skriptu lexikální a syntaktickou analýzu, aby ověřil, že obsah skriptu odpovídá modelu jazyka `Bash`. Výstup si uloží do derivačního stromu.

Pokud uživatel nevyžádal provádění pouze lexikální a syntaktické analýzy, provede se následná analýza toku dat ve skriptu, zjistí se, které konstrukce se provedou, které ne, zda nejsou ve skriptu sémantické chyby a je uživateli podána zpráva o tom, co daný skript provádí.

4.2.1 Úprava kódu před vlastní analýzou

V této části ze zabývám tím, jak a proč je potřeba upravit kód před tím, než podstoupí lexikální a syntaktickou analýzu.

Escapované konce řádků

Escapované konce řádků se upraví na klíčové slovo NEW_LINE, které po nalezení při lexikální analýze zapříčiní, že se počet řádků zvýší o jeden. Je-li klíčové slovo NEW_LINE nalezeno jako součást řetězce, pak se v tomto řetězci substituje za prázdný řetězec, jinak by docházelo ke špatné interpretaci dat.

Psaní zpětného lomítka na konci řádku se používá v případě, že potřebujeme rozdělit větší konstrukci na více řádků. Zdrojový kód je pak přehlednější.

Here document

Pomocí regulárních výrazů vyhledám here document sekvence. Začátek i konec je daný specifikací here document viz 3.4.5. Není-li v souboru konec here document sekvence, pak se vkládá na vstup příkazu celý text až do konce souboru.

Každý here document se uloží do proměnné a ve zdrojovém kódu je nahrazen klíčovým slovem, svým jménem a příslušným počtem prázdných řádků, aby v případě nalezení chyby modulem PLY (viz kapitola 5.1) bylo možné říci, na kterém řádku se daná chyba nachází.

Tato konstrukce:

```
cat <<text
toto je součást here documentu
toto je součást here documentu
text
```

se přepíše na:

```
cat <<HERE_DOCUMENT_text \n \n \n \n
```

Program si do vlastní proměnné uloží here document sekvenci, uchovává o ní tři parametry:

- název
- typ
- obsah

4.2.2 Dotazovací jazyk

Program bude přijímat od uživatele požadavky na dané kategorie pomocí parametrů při spouštění. Středník bude oddělovat hlavní kategorie, seznam dílčích vlastností jednotlivých kategorií bude začínat dvojtečkou a bude oddělován znakem čárka. Pomocí znaku „–“ bude možné odebírat jednotlivé kategorie/vlastnosti, aby uživatel, pokud nechce zahrnout pouze jednu kategorii, nemusel vypisovat všechny ostatní do parametrů.

Př. spuštění

```
./main.py -f nazevskriptu -o "soubor: zapis, mazani; slozka: nova; sit"
```

Vypíší se soubory pro zápis, soubory které se budou mazat, nově vytvářené složky a vše o pracování se sítí.

```
./main.py -f nazevskriptu -o "soubor: -zapis"
```

Vypíší se všechny informace o pracování se soubory kromě těch, do kterých souborů se zapisuje.

```
./main.py -f nazevskriptu -o "all -sit, -ostatni, -uzivatele"
```

Vypíší se všechny vlastnosti kromě kategorií `síť`, `ostatní` a `uživatelé`.

4.2.3 Řešení dílčích problémů

Přesměrování

Je nutné dopředu zjistit, zda není za příkazem či sekvencí příkazů přesměrování.

Mějme:

```
cd /bin
for i in 1 2 3; do
      echo $i
      cd ~
done > file.out
```

Výsledek celého cyklu se uloží do souboru `file.out`, ale ne v domovské složce uživatele, nýbrž do adresáře `/bin`. Důvodem je, že si `Bash` otevře soubor pro zápis ještě před vykonáním prvního příkazu v cyklu. Proto je nutné vědět ještě před kontrolou celé sekvence, zda se bude někam přesměrovávat.

Poziční parametry

Využívá-li v sobě skript parametry, se kterými by mohl být spuštěn, program uživateli předkládá tyto parametry ve formě:

ARGV(1), ARGV(2) ... ARGV(n)

Program bude schopný reagovat i na posouvání mezi parametry pomocí příkazu `shift`.

Proměnné

Program bude schopný interpretovat většinu běžně používaných proměnných v jazyce `Bash`, jako je např. $HOME, $PATH aj. Je třeba zajistit i správné chápání vícečetného přiřazo-

vání do proměnných v rámci jednoho řádku.

O každé proměnné je třeba uchovávat 4 údaje.

- jméno
- obsah
- není-li obsah vymezen příkazem declare
- není-li proměnná zamknutá proti přepisu příkazem *readonly*

Program na základě těchto hodnot zjistí, nepokouší-li se skript zapsat do zamčených proměnných, nebo jestli nepřiřazuje textový řetězec do proměnné, která může obsahovat pouze číselnou hodnotu (v takovém případě nedojde k chybě, ale do proměnné se přiřadí hodnota 0). V údaji `obsah` může být v jeden moment i více hodnot, záleží na tom, je-li program schopný rozhodnout, které příkazy se vykonají a které ne. V druhém případě udržuje obě hodnoty, aby uživatele varoval o všech možných stavech proměnné.

Substituce proměnných

Program bude disponovat pouze rozpoznáváním jednoduchých substitucí, tzn. oddělení proměnné od okolního textu ve složených závorkách. Funkce uvedené v kapitole 3.4.7 implementovány nebudou, ale jsou zmíněny v poslední části dokumentu jako návrh pro rozšíření.

Varování

Program v rámci analýzy skriptu bude vytvářet soupis varování, který bude obsahovat konstrukce, o kterých není možné rozhodnout, příkazy, které poběží i po skončení skriptu, jestli skript neupravuje vnitřní proměnné shellu, neexportuje proměnné mimo právě prováděný skript, nekončí-li předčasně aj.

Varování budou uživateli předkládaná s číslem řádku, aby se lépe orientoval ve zdrojovém kódu v případě ručního procházení. Nebezpečné konstrukce budou navíc obsahovat vlastní číslování – to bude sloužit v rámci vypisování výsledků příkazů. Nebude-li možné správně rozhodnout o vykonaných příkazech, mezi varování se přidá daná konstrukce a u každého příkazu v této konstrukci bude ve výpisu číslo této konstrukce, na které závisí jeho vykonání. Lépe lze ilustrovat příkladem:

Skript pro vyhodnocení

```
if [ "$USER" == "root" ]; then
  rm file1
else
  rm file2
fi
cat /etc/text
```

Výstup analyzátoru

```
Varování                          řádek
1) if [ "$USER" == "root" ]            1
#####################################
```

```
Vyhodnocení                        řádek

Soubory které se budou mazat
[1] file1                             2
[1] file2                             4
Soubory které se budou číst
/etc/text                             6
```

Tato varování lze vypnout pomocí přepínače `-w` popř. `--warning`, ale implicitně budou vypisována.

Návratové kódy

Často se vyskytuje situace, že vykonání příkazů je závislé na návratovém kódu příkazu předchozího. Buď se testuje návratový kód posledního vyvolaného příkazu pomocí speciální proměnné $?, nebo se používají konstrukce, kde se příkazy spojují pomocí `&&` nebo ||.

V rámci navrhovaného systému nebudu implementovat kontroly návratových kódů. U většiny vestavěných příkazů nelze při statické analýze sdělit, jakým návratovým kódem skončí.

Uživateli budou předkládány výsledky tak, že vykonání druhého příkazu závisí na výsledku předchozího (ten bude vypsán ve `varování` a očíslován podobně jako cykly, nebo větvení).

Lexikální a syntaktická analýza

Pro tuto část využiji `lex/yacc`. Konkrétní nástroj je zmíněn v kapitole 5.1. Výstupem této části bude derivační strom.

Při návrhu gramatiky jsem se inspiroval zdrojovými kódy jazyka Bash. V Bashi nelze ignorovat všechny bílé znaky a tak je třeba např. pravidlo pro vyhodnocení podmínek nadefinovat tak, že za levou hranatou a před pravou hranatou závorkou musí být mezera. V případě, že bychom toto pravidlo nenadefinovali, tak by mohlo docházet ke špatnému určení, je-li skript syntakticky správně napsán, neboť Bash pro vyhodnocení podmínek mezeru před a za závorkou potřebuje, interpretuje tuto konstrukci jako příkaz.

```
#Syntakticky správně
[ "$TEMP" == "text" ]
#Syntakticky špatně
["$TEMP" == "text" ]
[ "$TEMP" == "text"]
```

Sémantická analýza

Bude se provádět nad derivačním stromem. V jazyce Bash je nutné kontrolovat sémantický význam jednotlivých příkazů dle stavu systému. Sémantické kontroly se budou provádět až při vlastní analýze příkazů, aby nedocházelo k nesprávné interpretaci např. díky použití aliasů.

Cykly, větvení

Je-li možné staticky zjistit počet iterací cyklu, pak se daný cyklus bude v programu analyzovat právě tolikrát, jaký je jeho počet iterací. Nejde-li vyhodnotit přesný počet iterací, pak se konstrukce vkládá mezi `varování`, aby byl uživatel informován, že počet iterací záleží na běhu skriptu.

Pokud se dá u větvení rozhodnout, které části se provedou, pak se analyzují pouze tyto části. Nejde-li o nich rozhodnout za pomocí statické analýzy, pak se také ukládají mezi `varování` a analyzují se všechny části větvení s odkazem na konstrukci, na které závisí jejich provedení.

4.2.4 Vlastní řešení příkazů

Z derivačního stromu se načítají jednotlivé konstrukce, před vlastním vyhodnocením se provede substituce proměnných.

Příkaz je uložen v takovéto struktuře:

```
['echo', 'Toto', 'je', 'prikaz']
```

Z pole se vezme první prvek a ten se bude testovat:

1. Jedná-li se o přiřazení do proměnné, pak se vytvoří proměnná s daným obsahem a zbytek pole se předá znovu k vyhodnocení. Pokud proměnná již existuje, kontroluje se její vymezení obsahu a omezení proti přepisu. Je-li to možné, pak se obsah přepíše, jinak se mezi `varování` přidá zpráva o nemožnosti zápisu do proměnné. Tímto testováním lze zaručit správnou interpretaci vícečetného přiřazování do proměnných na jednom řádku, popř. přiřazení do proměnné a následného spuštění příkazu. Pokud se nejedná o přiřazení do proměnné, přechází se na bod 2.

2. Nachází-li se tento prvek mezi aliasy, bude nahrazen obsahem aliasu a celé pole se znovu předá hodnotící funkci (pro případ vnořených aliasů, nebo volání funkcí), jinak se přechází na bod 3. Uvažují se pouze aliasy definované v rámci aktuálně kontrolovaného skriptu.

3. Nachází-li se tento prvek mezi nadefinovanými jmény funkcí, pak se zavolá vyhodnocení této funkce a předá se jí zbytek pole jako parametry, jinak se přechází na bod 4.

4. Nachází-li se tento prvek mezi příkazy, které budou implementovány, pak se volá funkce pro vyhodnocení tohoto příkazu. Není-li tomu tak, pak je ve skriptu volán příkaz, který nebude implementovaný v mém systému a tak se přidá mezi varování, aby uživatel byl informován o nestandardním chování.

Kapitola 5

Implementace programu

Kapitola se věnuje nástrojům, které se použily při implementaci. Dále jsou popsány jednotlivé moduly programu a jejich obsah.

5.1 Použité nástroje

Pro implementaci programu jsem zvlolil programovací jazyk Python [8] a využil jsem nástroj PLY [9].

Python

Python je moderní, poměrně nový programovací jazyk, což se odráží v jeho základních vlastnostech. Mezi hlavní znaky patří to, že jde o objektově orientovaný, interpretovaný, na platformě nezávislý jazyk. Tyto základní charakteristiky, spolu s ještě dalšími hlavními vlastnostmi, jako jsou vysoká modularita, jednoduchá syntaxe, možnost integrace s dalšími programovacími jazyky, existence velkého množství knihoven, modulů, podpora tvorby webových aplikací a další, jsou dobrými předpoklady pro použití Pythonu při psaní skriptů, webových aplikací i rozsáhlých projektů.[10]

PLY

PLY (zkratka pro Python Lex-Yacc) je čistě pythonová implementace populárního nástroje pro konstrukci překladačů lex/yacc. Hlavním cílem PLY bylo zůstat věrný způsobu, kterým nástoje lex/yacc fungují.

To zahrnuje podporu LALR parsování, stejně tak jako ověřování vstupu, hlášení chyb a celkovou diagnostiku. Tento nástoj je vyvíjen od roku 2001 a prvotně sloužil jako pomůcka studentům pro tvorbu překladačů.

PLY je rozdělen na dva hlavní moduly:

- lex.py
  Tento modul provádí lexikální analýzu, je třeba mu předat pouze seznam tokenů a jejich definici (pomocí regulárních výrazů). Lze definovat i pravidla pro ignorované konstrukce v textu, jako jsou bílé znaky a komentáře.

- yacc.py
  Tento modul provádí syntaktickou analýzu. Je nutné definovat pravidla bezkontextové

gramatiky. Bohužel tento modul neprovádí automaticky tvorbu derivačního stromu, a tak bylo nutné ke každému pravidlu napsat systém pro jeho generování.

5.2 Jednotlivé moduly

Hlavní program

`main.py`

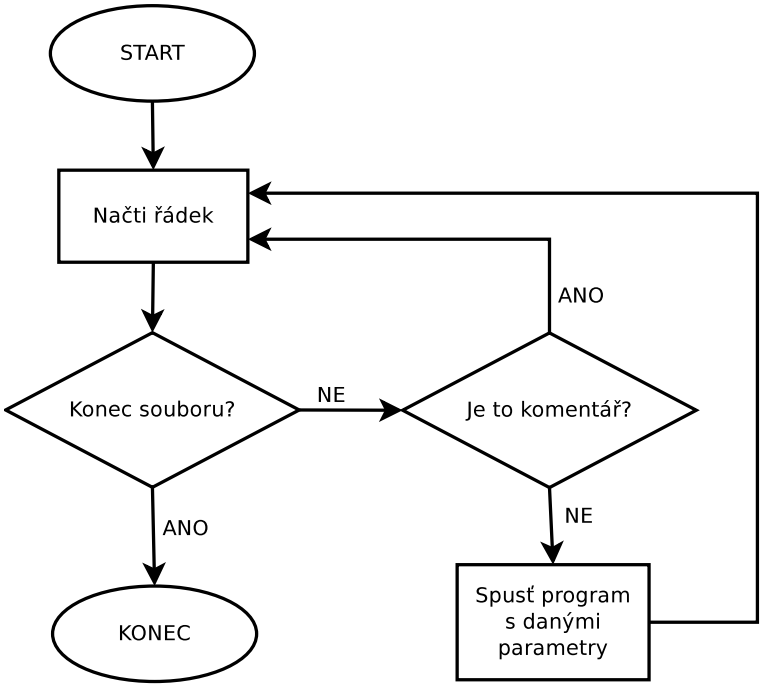
Tento soubor obsahuje hlavní funkci, ze které je voláno postupně vyhodnocení parametrů, lexikální analýza, syntaktická analýza a vlastní vyhodnocení skriptu, při kterém se provádí i sémantická analýza.

Modul pro zpracování parametrů

`funkce/parametry.py`

Jsou implementovány dva režimy, ve kterých program přijímá požadavky na vyhodnocení. Příklady užití viz 4.2.2.

1. Program načítá požadavky z parametrů příkazové řádky.
2. Program načítá požadavky ze vstupního souboru (skript), do kterého si uživatel může uložit často používáné příkazy. O vyhodnocení, či nevyhodnocení aktuální řádky rozhoduje, je-li první znak na řádku `#`. Schéma vyhodnocení viz obrázek 5.1.



Obrázek 5.1: Algoritmus pro zpracování parametrů načítaných ze souboru.

Přehled přepínačů obsahuje tabulka 5.1.

Krátká verze	Dlouhá verze	Význam
-h	- -help	Vypíše nápovědu
-s	- -syntax	Bude se provádět pouze syntaktická analýza
-o	- -option	Specifikace analyzovaných vlastností, viz 4.2.2
-f	- -file	Určení jednoho skriptu, či složky se skripty
-l	- -lines	Vypne vypisování čísel řádků ve výsledcích
-w	- -warnings	Vypne varovná hlášení

Tabulka 5.1: Přepínače pro ovládání programu

Modul pro lexikální a syntaktickou analýzu

`parser/LexYacc.py`

Tento modul provede lexikální a syntaktickou analýzu nad právě analyzovaným skriptem. Pokud se vyskytne lexikální nebo syntaktická chyba, pak je vypsána uživateli zpráva o konstrukci a řádku, kde se tato chyba vyskytuje. Nad syntakticky špatně napsaným skriptem nelze provádět konečnou analýzu, neboť by docházelo k chybným výsledkům.

Modul sémantické analýzy a analýzy dat

`funkce/analyza.py`

Modul tvoří hlavní funkce `Solve`, která je implementací návrhu z kapitoly 4.2.4, funkce pro jednotlivé příkazy (pro jejich správnou interpretaci a sémantickou kontrolu) a několik dalších funkcí (podpora aritmetické expanze, expanze příkazů aj.).

Kapitola 6

Závěr

Zhodnocení práce

Úkolem práce bylo navrhnout a implementovat systém, který by prováděl statickou analýzu nad skripty jednoho z jazyků unixového shellu. Provedení malého průzkumu vedlo k zaměření na jazyk Bash i přesto, že se ještě stále pro konfigurační skripty v prostředí linuxu používá více klasický Bourne shell.

Implementoval jsem lexikální a syntaktický analyzátor za pomocí nástroje PLY od Davida Beazlyho. Implementovaný systém odpovídá návrhu a specifikaci požadavků. Byla vytvořena sada testů pro ověření funkčnosti jednotlivých částí, tato sada je umístěna na přiloženém cd.

Nalezené chyby

Po dobu testování byla nalezena pouze jedna větší chyba, a to, že implementovaná aplikace není schopná reagovat na „vnořený“ here document. Tuto chybu lze odstranit pomocí propracovanějšího vyhledávače sekvencí here document.

Tento příklad:

```
cat <<text1 ; cat <<text2
Zde je prvni vypis
text1
Zde je druhy vypis
text2
```

program upraví na: `cat <<text1 ; cat <<HERE_DOCUMENT_text2 \n \n \n \n` a poté skončí s chybovým hlášením při syntaktické analýze.

Možnost rozšíření

Současný stav práce určuje další logický krok - doplnění programu o správné chápání všech konstrukcí jazyka Bash: rozšířené pracování se substitucí parametrů, přesměrování pomocí filedescriptorů atd.

Po doplnění výše zmíněných možností by bylo vhodné projekt rozšířit o nástroje dynamické analýzy, což bych si rád zvolil pro téma diplomové práce.

Literatura

[1] Chess, B.: *Secure programming with static analysis*. Pearson Education, 2007, ISBN 0-321-42477-8.

[2] Meduna, A.,Roman, L.: *Formální jazyky a překladače – studijní opora*. Brno, VUT v Brně, 2006.

[3] Lesk, M. E., Schmidt, E.: Lex - A Lexical Analyzer Generator [online]. `http://dinosaur.compilertools.net/lex/index.html`, [cit. 2011-05-16].

[4] Johnson, S. C.: Yacc: Yet Another Compiler-Compiler [online]. `http://dinosaur.compilertools.net/yacc/index.html`, [cit. 2011-05-16].

[5] WWW stránky: Bash [online]. `http://cs.wikipedia.org/wiki/Bash`, [cit. 2011-05-16].

[6] WWW stránky: Unixový shell [online]. `http://cs.wikipedia.org/wiki/Unixov%C3%BD_shell`, [cit. 2011-05-16].

[7] WWW stránky: Shelly UNIXu [online]. `http://milux.php5.cz/skoleni/skoleni-unix-7.html`, [cit. 2011-05-16].

[8] Python tutorial [online]. `http://docs.python.org/tutorial/`, [cit. 2011-05-10].

[9] Beazly, D.: PLY [online]. `http://www.dabeaz.com/ply/`, [cit. 2011-05-16].

[10] Sklenička, R.: *Objektově orientovaný jazyk Python*. Praha, ČVUT, 2005.

Příloha A

Rozdělení příkazů

Skupina	Podskupina	Příkazy ve skupině
Soubory	Zápis	>, >>, tee
	Čtení	cat, egrep, fgrep, grep, head, tail, look, paste, nl
	Kopírování	cp, dd, install,
	Odstraňování	rm
	Porovnávání	cmp, comm, diff, diff3
	Upravování	csplit, cut, awk, gawk, fmt, fold, join, rev, sdiff, sed, sort, split, tr, tsort
	Přesouvání	mmv, mv
	Přejmenování	mmv, mv, rename,
	Komprese	bzip2, gzip, tar
	Časová razítka	touch
	Test vlastností	test, [ ]
Složky	Vytváření	mkdir
	Odstraňování	rmdir
	Nahlížení	ls
Síť	Nastavení rozhraní	ifconfig, ifdowdn, ifup, ethtool
	Posílání dat	scp, ftp, rcp, sftp
	Sledování statistik	mtr, netstat
	Stahování webu	wget
	Hledání cíle	traceroute
	Přístup na url	xdg-open
	Test dosažitelnosti	ping
	Přihlašování přes SSH	ssh
	Dotazy na DNS	dig, nslookup
Práva	Změna	chmod
	Root přístup	sudo, chroot
	Spouštění příkazů pod jiným účtem	op, su

Skupina	Podskupina	Příkazy ve skupině
Cesty	Kde skript operuje	cd
	Na které se nahlíží	pwd, ls
	Práce se zásobníkem	popd, pushd
Systém	Nastavení data	date
	Nastavení limitů	ulimit
	Nastavení systémového jména	hostname
	Vypnutí, uspání, restart, odhlášení	shutdown, sleep, reboot, logout
Procesy	Sledování	ps, jobs
	Zastavení	kill, killall, pkill
	Změna priority	nice
Tiskárna	Tisk souborů	lprint
	Manipulace s tiskovou frontou	lprintd, lprm, lpr
Uživatelé	Nový uživatel nebo skupina	useradd, groupadd, adduser
	Změna hesla	passwd
	Modifikace účtů	usermod
	Sledování zalogovaných uživatelů	users
	Změna skupiny / vlastníka u souboru / složky	chgrp, chown
Otevírání	Souborů	open
	Spouštění skriptů	., source
	Periodické spouštění příkazů	watch
	Vykonání příkazů	nohup, xargs, exec, eval
Ostatní	Manipulace s diskem	cfdisk, quota, quotacheck, fdisk,
	Formát diskety	fdformat
	Připojování médií	mount
	Kompilování	make
	Ostatní	alias, function, enable, import, getopts, set, export

Příloha B

Obsah CD

Níže uvedené přílohy jsou umístěné na přiloženém CD

- Zdrojové kódy aplikace
  (složka **src**)
- Potřebné moduly pro běh aplikace s návodem k instalaci
  (složka **install**)
- Elektronická verze tohoto dokumentu
  (soubor **bp.pdf**)
- Zdrojové soubory elektronické verze tohoto dokumentu
  (složka **tex**)