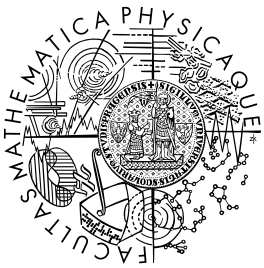
Shell v příkladech

aneb

ABY VÁŠ UNIX SKVĚLE SHELL

Libor Forst



matfyzpress
PRAHA 2010

ISBN 978-80-7378-152-1

OBSAH

ABY VÁŠ UNIX SKVĚLE SHELL... 1

PODROBNÝ OBSAH 3

ČÁST 1 ÚVOD 19

1.1. ZAČNEME ZLEHKA 21
1.2. PŘIDÁME PARAMETRY 22
1.3. POPRVÉ KOMBINUJEME 24
1.4. DALŠÍ PŘESMĚROVÁNÍ 26
1.5. NEPATRNĚ PŘIDÁME 27
1.6. A ZASE UBEREME 28
1.7. ZAMETÁME STOPY 29
1.8. DÁREK K NAROZENINÁM 30
1.9. HVĚZDIČKA 31
1.10. ZÁLUDNÁ JMÉNA 32
1.11. PRVNÍ PROGRAM V SHELLU 34
1.12. POČÍTÁNÍ SOUBORŮ LÉPE 36
1.13. POČÍTÁNÍ PODADRESÁŘŮ 38
1.14. POČÍTÁNÍ DO HLOUBKY 39
1.15. ROZSAH ŘÁDEK 42
1.16. SLOUPEČKY 43
1.17. ZE SLOUPEČKU ŘÁDKA 45
1.18. A NAOPAK 46
1.19. STŘÍHÁNÍ A SLEPOVÁNÍ JINAK 49
1.20. SEZNAM SKUPIN BEZ POMOCI 50
1.21. NA CO NŮŽKY NESTAČÍ 52
DALŠÍ NÁMĚTY NA PROCVIČENÍ 53

ČÁST 2 PRÁCE SE SOUBORY 21

2.1. POŠTOVNÍ SCHRÁNKA 57
2.2. NOVÝ PŘÍKAZ PRO SKLEROTIKY 59
2.3. NOVÝ PŘÍKAZ PRO LENIVÉ 60
2.4. NOVÝ PŘÍKAZ PRO NEROZHODNÉ 62
2.5. KOPÍROVÁNÍ LINKŮ 65
2.6. PŘESUN LINKŮ 67
2.7. JAK VYROBIT NIC 68
2.8. PŘÍKAZ PRO DR. WATSONA 69
2.9. TAKHLE NE, MILÝ WATSONE 70

2.10. KAŽDÉMU, CO MU PATŘÍ ... 72
2.11. KDO JE KDO ... 73
2.12. TŘÍDĚNÍ ... 74
2.13. TŘÍDĚNÍ S KLÍČEM ... 75
2.14. TŘÍDĚNÍ S PROBLÉMY ... 76
2.15. TŘÍDĚNÍ „JINDE“ ... 77
2.16. UNIKÁTNÍ TŘÍDĚNÍ ... 78
2.17. NAJDI DESET ROZDÍLŮ ... 79
2.18. NA NÁVŠTĚVĚ U DATABÁZÍ ... 80
2.19. VSTUP VERSUS PARAMETRY ... 82
DALŠÍ NÁMĚTY NA PROCVIČENÍ ... 84

ČÁST 3 EDITORY ... 59

3.1. DÉJÀ VU ... 87
3.2. ČÍSLA ŘÁDEK ... 89
3.3. SLOŽITĚJŠÍ VÝBĚR ... 90
3.4. PŘEHAZOVÁNÍ SLOUPCŮ ... 92
3.5. HLEDÁNÍ DUPLICIT ... 94
3.6. HLEDÁNÍ ČÍSEL ... 95
3.7. HLEDÁNÍ SLOV ... 97
3.8. STŘELBA NA POHYBLIVÝ CÍL ... 98
3.9. DALŠÍ, PROSÍM ... 100
3.10. CYKLUS ... 102
3.11. PŘIDÁVÁNÍ ŘÁDEK ... 104
3.12. ÚSCHOVNA ZAVAZADEL ... 106
3.13. CO NA NÁDRAŽÍ NESNÁŠEJÍ ... 107
3.14. JEŠTĚ JEDNOU DO ÚSCHOVNY ... 109
3.15. ZALOMENÉ ŘÁDKY ... 110
3.16. PAMĚŤ ... 111
3.17. VERTIKÁLNÍ ZRCADLO ... 113
3.18. LEPORELO ... 115
3.19. EDITACE SOUBORU ... 116
3.20. AKTUÁLNÍ ŘÁDKA ... 118
3.21. GLOBALIZACE ... 119
3.22. HORIZONTÁLNÍ ZRCADLO ... 121
3.23. MAŠINKA ... 123
DALŠÍ NÁMĚTY NA PROCVIČENÍ ... 127

ČÁST 4 SHELL 87

4.1. EXPANZNÍ ZNAKY 131
4.2. PROGRAMOVÁNÍ „ON-LINE“ 132
4.3. POUŽITÍ PROMĚNNÉ 133
4.4. DEFINICE PROMĚNNÉ 134
4.5. JAK NAMALOVAT „OBRÁZEK“ 135
4.6. PROMĚNLIVÁ ŘÁDKA 138
4.7. ODDĚLOVAČ POLÍ 140
4.8. ZPRACOVÁNÍ CHYB 142
4.9. ŠETŘÍME DISKY 143
4.10. ČTENÍ „PER PARTES“ 144
4.11. KDYŽ READ NEČTE 145
4.12. KDYŽ READ PARSUJE 147
4.13. PARAMETRY 148
4.14. HELLO, WORLD 149
4.15. ŠKATULATA, HEJBEJTE SE 150
4.16. DYNAMICKÝ VSTUP 152
4.17. AK, PAK 155
4.18. AK, PAK, INAK 156
4.19. KDYŽ IF NESTAČÍ 159
4.20. PŘÍKAZ CASE NARUBY 161
4.21. CYKLUS S PŘEKVAPENÍM 162
4.22. CYKLUS BEZ PŘEKVAPENÍ 164
4.23. CYKLUS, JAK HO (NE)ZNÁME 166
4.24. POTÍŽE S RELATIVITOU 168
4.25. AMOEBA 171
4.26. JAK NA PARAMETRY 173
4.27. PARAMETRY LÉPE 176
4.28. PARAMETRY DO TŘETICE 177
4.29. POSLEDNÍ PARAMETR 182
4.30. NEZNÁMÝ FORMÁT 184
4.31. INTERAKTIVNÍ MAZÁNÍ 185
4.32. REKURZE A PROMĚNNÉ 188
4.33. REKURZE A PROCESY 191
4.34. PROCESY A PROMĚNNÉ 195
4.35. NĚCO O EFEKTIVITĚ 201
4.36. O EFEKTIVITĚ JEŠTĚ JEDNOU 203
4.37. KDYŽ TEXT NESTAČÍ 207
DALŠÍ NÁMĚTY NA PROCVIČENÍ 211

ČÁST 5 FILTR AWK ... 213

5.1. ZASE OD PÍKY ... 215
5.2. POČITADLO DÉLEK SOUBORŮ ... 216
5.3. NENÍ REGEXP JAKO REGEXP ... 218
5.4. KONEČNÝ AUTOMAT ... 220
5.5. A ZASE SKUPINY ... 223
5.6. SKUPINY S PARAMETREM ... 225
5.7. TRUMFOVÉ ESO AWK ... 228
5.8. UKAŽ, CO MÁŠ ... 231
5.9. INTERAKCE ... 232
5.10. EDITACE ... 235
5.11. FORMÁTOVÁNÍ ... 237
5.12. KDYŽ ŘÁDKA NENÍ ŘÁDKA ... 239
5.13. BLOKY ŘÁDEK ... 241
5.14. PARSER ... 243
5.15. KONFIGURACE PROGRAMU ... 245
5.16. MAMINKA MĚLA PRAVDU ... 249
5.17. JAK NEPODLEHNOUT REKLAMĚ ... 252
5.18. PODSOUVÁNÍ SOUBORŮ ... 255
DALŠÍ NÁMĚTY NA PROCVIČENÍ ... 257

ČÁST 6 PŘÍPADOVÉ STUDIE ... 257

6.1. STAHOVÁNÍ WEBOVÝCH STRÁNEK ... 261
6.2. STAHOVÁNÍ DO HLOUBKY ... 275
6.3. WWW NEWS ... 283
6.4. PROHLEDÁVÁNÍ POŠTY ... 290

ABY VÁŠ UNIX SKVĚLE SHELL... ... 301

PŘÍLOHA A: STRUČNÝ PŘEHLED PŘÍKAZŮ ... 303

A.1 VYBRANÉ UTILITY ... 306
A.2 PŘÍKAZ AWK ... 323
A.3 PŘÍKAZ ED ... 328
A.4 PŘÍKAZ FIND ... 331
A.5 PŘÍKAZ SED ... 333
A.6 PŘÍKAZ SH A JAZYK SHELLU ... 337
A.7 REGULÁRNÍ VÝRAZY ... 346
A.8 TŘÍDY ZNAKŮ ... 347
A.9 FORMÁTOVACÍ DIREKTIVY PRO PRINTF ... 348

PŘÍLOHA B: FORMÁTY POUŽÍVANÝCH ZDROJŮ DAT 349

B.1 POPIS VYBRANÝCH SOUBORŮ 349
B.2 POPIS VYBRANÝCH SOUBORŮ 351

LITERATURA 353

REJSTŘÍK 355

Aby váš UNIX skvěle shell...

...budete se muset naučit hodně triků a fíglů. Mnohé z nich pochytíte na Internetu, od známých a z jiných zdrojů, ale časem vám to jistě nebude stačit a budete si chtít některé z nich upravit anebo dokonce sami napsat. Při tom se budete muset více či méně ponořit do studia nástroje, bez něhož v operačním systému unixového typu neuděláte prakticky nic. Tím nástrojem je *shell*.

Termín *shell* označuje jednak *interpret příkazů* – tedy tu součást (či spíše nadstavbu) systému, která čte a vykonává příkazy zadané uživatelem – a jednak vlastní *jazyk* tohoto interpretu. Tato kniha vám nabídne stručný přehled těch nejdůležitějších vlastností shellu a základních pomocných programů, tzv. *utilit*, které se při práci v shellu používají. To vše se budeme snažit ukázat na praktických příkladech. Pokud si budete zkoušet úlohy řešit sami a půjdete na to jinak, neznamená to, že jste udělali chybu. Cest k cíli vede vždycky více a ty naše sledují především určitý didaktický cíl. Některá řešení proto nepokrývají beze zbytku daný problém – pokud by takové řešení bylo vyčerpávající, vyčerpalo by totiž především čtenáře...

Kniha si rozhodně neklade za cíl postihnout téma nějakým systematickým výkladem. Takových knih popisujících vlastnosti, architekturu či ideové kořeny UNIXu existuje celá řada. Naše kniha se snaží uvést čtenáře do stylu, jakým se v shellu programuje. Vychází tedy z praktické potřeby – z podobné praktické potřeby, z jaké operační systém UNIX před bezmála čtyřiceti lety sám vyvstal.

Pro práci s touto knihou bude dobré mít přístup na nějaký počítač se systémem unixového typu. Jaký konkrétně to bude, je víceméně nezajímavé – budeme se totiž snažit používat jen takové vlastnosti, které vyhovují nejen rozličným normám ale dokonce i existujícím systémům. Ostatně samotné slovo „UNIX“ budeme používat v tom smyslu, jaký mu dali jeho tvůrci dávno před tím, než si ho všimla komerce. Pokud by se někdo domníval, že má na toto slovo výhradní právo (a že už v minulosti takové snahy byly), můžeme zodpovědně říci, že je nám to hluboce líto... Pokud se někde budeme odvolávat na *normu*, budeme mít na mysli dokument [1] (jemuž se také krátce říká „POSIX“) a budeme to činit bez přehnané víry v to, že nás i vás to ochrání od problémů s přenositelností našich programů.

Takže si najděte libovolný UNIX, zařiďte si na něm účet (tedy uživatelské jméno a heslo), přihlaste se na něj a otočte stránku...

Poděkování

Především bych chtěl poděkovat svojí ženě Lence a synům Martinovi, Antonínovi a Albertovi, že to vydrželi...

Neocenitelnou pomoc mi poskytl kolega Dan Lukeš svými upřesněními, radami či poznámkami, i když se všemi opravdu, ale opravdu nemohu souhlasit...

Velký dík patří i kolegům, kteří měli na starosti korekturu a recenze. Nejenom že přispěli k vyšší „štábní“ kultuře, ale odhalili celou řadu chyb, včetně těch nenápadných, které ten, kdo zná dopředu ideu věty, odstavce či programu, má pramalou šanci odhalit.

A konečně chci poděkovat Joelovi Berglundovi za nádherný obrázek perlorodky říční, našeho kriticky ohroženého živočicha, který je, stejně jako já a brzo i vy, životně závislý na shellu...

Podrobný obsah

Část 1 Úvod

1.1 ZAČNEME ZLEHKA 21
Hierarchický souborový systém, pojmy kořen, adresář a cesta
Domovský adresář, aktuální adresář, příkaz **pwd**
Příkazová řádka, prompt

1.2 PŘIDÁME PARAMETRY 22
Typy parametrů, přepínače a operandy, slučování přepínačů
Typy souborů, obyčejný soubor vs. adresář
Příkaz **man**
Příkaz **ls**, dlouhý výpis (**-l**), potlačení výpisu obsahu adresářů (**-d**)
Adresář „.“ a „..“, absolutní a relativní cesta, příkaz **cd**

1.3 POPRVÉ KOMBINUJEME 24
Seznam uživatelů, soubor **/etc/passwd**, textové vs. binární soubory
Příkaz **wc**, výpis počtu řádek (**-l**)
Proudy dat, standardní vstup, standardní výstup, přesměrování vstupu
Terminál, vstup z terminálu, Ctrl+D

1.4 DALŠÍ PŘESMĚROVÁNÍ 26
Přesměrování výstupu (**>**), příkaz **echo**
Metaznaky shellu: operátory přesměrování, mezera, LF
Příkaz **cat**

1.5 NEPATRNĚ PŘIDÁME 27
Přesměrování výstupu (**>>**), rozdíly mezi oběma typy přesměrování
Příkazy **rm** a **date**

1.6 A ZASE UBEREME 28
Příkazy **tail** a **mv**
Přepínače s hodnotou
Současné čtení a zápis souboru, dočasný soubor, adresář **/tmp**

1.7 ZAMETÁME STOPY 29
Datum a čas poslední modifikace souboru, příkaz **touch**

1.8 DÁREK K NAROZENINÁM 30
Rušení významu metaznaků v shellu, apostrof
Příkaz **ls**: třídění výstupu u (**-t**, **-r**)
Formát uložení a výpisu data a času poslední modifikace

1.9 HVĚZDIČKA 31
Expanzní znaky, porovnávací vzory, hvězdička, princip expanze

1.10 ZÁLUDNÁ JMÉNA 32
Obyčejné parametry vs. přepínače, parametr „`--`“
Příkaz **`rm`**: potlačení chyb (**`-f`**)
Příkaz **`ls`**: potlačení třídění (**`-f`**)
Metaznaky ve jménech souborů

1.11 PRVNÍ PROGRAM V SHELLU 34
Roura
Příkaz **`ls`**: odlišnost formátu při výpisu na terminál
Mazání dočasných souborů

1.12 POČÍTÁNÍ SOUBORŮ LÉPE 36
Skryté soubory: výpis pomocí **`ls`** (**`-a`**, **`-A`**), expanze
Příkaz **`tail`**: výpis počínaje konkrétní řádkou od začátku souboru
Standardní chybový výstup, deskriptory
Soubor **`/dev/null`** (jako výstupní), potlačení chybových zpráv
Příkaz **`mkdir`**

1.13 POČÍTÁNÍ PODADRESÁŘŮ 38
Typ souboru ve výpisu **`ls`**
Hledání textu, regulární výrazy, příkaz **`grep`**
Regulární výrazy: ukotvení na začátek výrazu (**`^`**)

1.14 POČÍTÁNÍ DO HLOUBKY 39
Příkaz **`ls`**: rekurzivní výpis obsahu adresářů (**`-R`**)
Příkaz **`grep`**: inverzní výběr (**`-v`**), více vzorů (**`-e`**)
Přípustné znaky pro jména souborů, přípona souboru
Regulární výrazy: ukotvení na konec výrazu (**`$`**), libovolný znak (**`.`**), opakování (**`*`**), rušení zvláštního významu znaků (**`\`**)
Srovnání regulárních výrazů a porovnávacích vzorů
Slučování přepínačů s hodnotou

1.15 ROZSAH ŘÁDEK 42
Volání utilit bez parametrů
Příkaz **`head`**

1.16 SLOUPEČKY 43
Příkazy **`tr`**, **`cut`** a **`paste`**
Náhrada jednoduché databáze
Třídy znaků

1.17 ZE SLOUPEČKU ŘÁDKA 45
Skupiny uživatelů, soubor **/etc/group**
Zápis konce řádky v shellu, escape-sekvence „**\n**“ v příkazu **tr**
Sekundární prompt

1.18 A NAOPAK 46
Příkazy **id** a **tee**
Zpětné lomítko v shellu, pokračovací řádka
Příkaz **tr**: slučování znaků ve výstupu
Příkaz **tr**: opakování znaků v tabulce

1.19 STŘÍHÁNÍ A SLEPOVÁNÍ JINAK 49
Příkaz **split**
Příkaz **paste**: sekvenční spojování řádek (**-s**)
Standardní vstup zadaný jako parametr „**-**“

1.20 SEZNAM SKUPIN BEZ POMOCI 50
Definice členství uživatele ve skupině, primární skupina
Příkaz **grep**: hledání celých slov (**-w**)
Vyjmutí sloupce, přidání sloupce
Sekvenční provádění příkazů, středník

1.21 NA CO NŮŽKY NESTAČÍ 52
Různý pohled na slučování oddělovačů sloupců u různých utilit

Část 2 Práce se soubory

2.1 POŠTOVNÍ SCHRÁNKA 57
Přístupová práva
Příkaz **chmod**

2.2 NOVÝ PŘÍKAZ PRO SKLEROTIKY 59
Příkaz **cp**
Adresář **/bin**
Interní příkazy shellu vs. utility, volání programu s cestou

2.3 NOVÝ PŘÍKAZ PRO LENIVÉ 60
Shellskript, práva pro spouštění skriptu, spouštění pomocí **sh**
Uživatelská maska, příkaz **umask**

2.4 NOVÝ PŘÍKAZ PRO NEROZHODNÉ 62
Indexový uzel (i-node), logický souborový systém
Hardlink, symlink, ekvivalence souborů
Příkazy **ln** a **df**

2.5 KOPÍROVÁNÍ LINKŮ 65
Sémantika a chování příkazů **cp** a **mv**
Adresář **bin**
Příkaz **tar**

2.6 PŘESUN LINKŮ 67
Relativní a absolutní cesta v symlinku
Přejmenování vs. přesun jako důsledek příkazu **mv**

2.7 JAK VYROBIT NIC 68
Soubor **/dev/null** (jako vstupní)
Příkaz **echo** a odřádkování
Příkazy **printf** a **:** (null)
Rozdíl mezi vymazáním a smazáním souboru

2.8 PŘÍKAZ PRO DR. WATSONA 69
Příkaz **find**

2.9 TAKHLE NE, MILÝ WATSONE 70
Expanze v adresářové struktuře
Nevhodné použití **find**
Příkazu **find**: složitější podmínky, logické výrazy, porovnávací vzory, metaznaky shellu jako operátory

2.10 KAŽDÉMU, CO MU PATŘÍ 72
Použití **-exec** pro testování ve **find**
Použití příkazu **grep** pro testování (**-q**)
Příkaz **chmod**: selektivní právo **X**

2.11 KDO JE KDO 73
Příkaz **file**
Rizika provádění příkazů pomocí **-exec** ve **find**
Volání shellu pomocí **sh -c**

2.12 TŘÍDĚNÍ 74
Příkaz **sort**: implicitní a numerické třídění, třídicí atributy
Tabulka znaků ASCII, národní znakové sady
Efektivita utilit vs. řešení v shellu

2.13 TŘÍDĚNÍ S KLÍČEM 75
Příkaz **sort**: třídicí klíč, oddělovač polí, třídění na místě

2.14 TŘÍDĚNÍ S PROBLÉMY 76
Příkaz **sort**: vícenásobný třídicí klíč
Srozumitelnost programů

2.15 TŘÍDĚNÍ „JINDE“ 77
Třídění podle pomocného klíče
Příkaz **grep**: číslování výstupu (**-n**)

2.16 UNIKÁTNÍ TŘÍDĚNÍ 78
Příkaz **sort**: unikátní třídění (**-u**)
Příkaz **uniq**

2.17 NAJDI DESET ROZDÍLŮ... 79
Příkazy **diff** a **comm**
Příkaz **uniq**: výpis duplicitních řádek (**-d**)
Příkaz **cut**: vynechávání neúplných řádek (**-s**)

2.18 NA NÁVŠTĚVĚ U DATABÁZÍ 80
Příkaz **join**

2.19 VSTUP VERSUS PARAMETRY 82
Parametry příkazů vs. standardní vstup
Příkaz **xargs**

Část 3 Editory

3.1 DÉJÀ VU 87
Editace proudu dat: stream editor **sed**
Příkaz *substitute* (**s**), formát a parametry

3.2 ČÍSLA ŘÁDEK 89
Adresní část příkazů, rozsah adres
Číslo řádky jako adresa, poslední řádka
Oddělovač příkazů
Potlačení implicitního výstupu **sed** (**-n**)
Příkazy *delete* (**d**), *print* (**p**) a *quit* (**q**)

3.3 SLOŽITĚJŠÍ VÝBĚR 90
Regulární výrazy: výběr znaků (**[]**), interval a třídy znaků
Změna významu metaznaků mezi hranatými závorkami
Rozsah řetězce pokrytý výrazem obsahujícím hvězdičku

3.4 PŘEHAZOVÁNÍ SLOUPCŮ 92
Regulární výrazy: závorkování, zpětné reference
Metaznak **&** v řetězci náhrady příkazu *substitute*
Rozdíl mezi smazáním (**d**) a vymazáním řádky (**s**)
Podvýraz „**.***“ na krajích regexpu

3.5 HLEDÁNÍ DUPLICIT 94
Regulární výrazy: přesný počet opakování (**\{\}**)
Zpětné reference ve vzoru
Dialekty regulárních výrazů, základní regulární výrazy

3.6 HLEDÁNÍ ČÍSEL 95
Adresa příkazu ve formě regexpu, doplněk adresy (**!**)
Zadání jiného oddělovače regexpu
Zadání skriptu pro **sed** pomocí přepínače **-f**
Příkaz **tr**: komplementární zadání znakové sady (**-c**)

3.7 HLEDÁNÍ SLOV 97
Hledání celých slov pomocí regexpů
Parametry **p** (print) a **w** (write) příkazu *substitute*

3.8 STŘELBA NA POHYBLIVÝ CÍL 98
Pracovní prostor a důsledky postupné modifikace jeho obsahu
Složený příkaz (**{}**)
Příkaz pro konverzi znaků (**y**)
Použití regexpů s ukotvením („**^**“ a „**$**“) v příkazu *substitute*
Komentáře ve skriptu

3.9 DALŠÍ, PROSÍM 100
Parametr **g** (global) příkazu *substitute*, slučování parametrů
Adresní část příkazu ve tvaru rozsahu regexpů
Příkaz *next* (**n**)
Znak konce řádky v řetězci náhrady

3.10 CYKLUS 102
Problémy globálního nahrazování příkazem *substitute*
Řízení toku: návěští (**:**), příkazy *branch* (**b**) a *test* (**t**)
Zadání jiného oddělovače v příkazu *substitute*

3.11 PŘIDÁVÁNÍ ŘÁDEK 104
Příkazy *insert* (**i**), *append* (**a**) a *change* (**c**)

3.12 ÚSCHOVNA ZAVAZADEL 106
Odkládací prostor
Příkazy *hold* (**h**), *get* (**g**) a *exchange* (**x**)

3.13 CO NA NÁDRAŽÍ NESNÁŠEJÍ 107
Příkazy *Hold* (**H**), *Get* (**G**)
Znak konce řádky v řetězcovém vzoru
Příkazy skoku bez udání návěští

3.14 JEŠTĚ JEDNOU DO ÚSCHOVNY 109
Pokročilejší práce s odkládacím prostorem

3.15 ZALOMENÉ ŘÁDKY 110
Přidání nové řádky do pracovního prostoru: příkaz *Next* (**N**)
Zpětná reference uvnitř řetězce obsahujícího číslice

3.16 PAMĚŤ 111
Odkládací prostor jako paměť
Ukotvení a znak konce řádky v pracovním prostoru
Porovnání efektivity implementace regexpů v utilitách **grep** a **sed**

3.17 VERTIKÁLNÍ ZRCADLO 113
Znak konce řádky jako zarážka v pracovním prostoru

3.18 LEPORELO 115
První řádka v pracovním prostoru: příkazy *Delete* (**D**) a *Print* (**P**)

3.19 EDITACE SOUBORU 116
Editace souboru: editor **ed**
Porovnání editorů **ed** a **sed**
Potlačení statistických výpisů editoru **ed** (**-s**)
Formát příkazů, adresa, adresa **0**
Příkazy *append* (**a**), *insert* (**i**), *write* (**w**) a *quit* (**q**)

3.20 AKTUÁLNÍ ŘÁDKA 118
Institut aktuální řádky editoru **ed**
Tvary adresy, posun (offset)
Příkaz *delete* (**d**)

3.21 GLOBALIZACE 119
Opakování příkazů na více řádkách: příkazy *global* (**g**) a *invert* (**v**)
Řetězcové vzory v adrese: rozsah, hledání vpřed (**?**)
Rozdíly v zadání opakovaného provádění příkazů u editorů **ed** a **sed**

3.22 HORIZONTÁLNÍ ZRCADLO 121
Přesun řádek: příkaz *move* (**m**), adresa **0**
Ukončení editace bez uložení: příkaz *Quit* (**Q**)
Implicitní rozsah platnosti pro jednotlivé příkazy
Rozdíly mezi provedením příkazu na „každé“ a na „všech“ řádkách
Editační příkazy v souboru (ed-skript)

3.23 MAŠINKA 123
Výstup příkazu **diff** ve formátu ed-skriptu („**-e**“)
Složitější manipulace s bloky řádek
Ladění skriptů v editoru

Část 4 Shell

4.1 EXPANZNÍ ZNAKY 131
Expanzní znaky: výběr znaků (**[]**), libovolný znak (**?**)
Expanze podle porovnávacích vzorů, třídění výsledku expanze

4.2 PROGRAMOVÁNÍ „ON-LINE“ 132
Spouštění příkazů vygenerovaných on-line
Příkaz **cp**: zachování data a času (**-p**)

4.3 POUŽITÍ PROMĚNNÉ 133
Expanze proměnných, metaznak **$**
Proměnné prostředí
Domovský adresář: **$HOME**, **~**
Typy shellů a přenositelnost

4.4 DEFINICE PROMĚNNÉ 134
Přiřazovací příkaz
Proměnná **PATH**, význam a bezpečnostní souvislosti

4.5 JAK NAMALOVAT „OBRÁZEK 135
Postup zpracování řádky v shellu, oddělovače slov, operátory, expanze
Interní oddělovač polí, proměnná **IFS**
Metaznaky shellu, rušení metavýznamu (quoting), apostrofy, uvozovky, zpětné lomítko
Identifikátory a hodnoty proměnných
Upřesnění práce příkazu **echo**
Příkaz **set**: výpis všech proměnných

4.6 PROMĚNLIVÁ ŘÁDKA 138
Substituce příkazu
Náhrada posloupností bílých znaků mezerou

4.7 ODDĚLOVAČ POLÍ 140
Změna interního oddělovače polí, slučování výskytu oddělovačů
Zadání tabulátoru na příkazové řádce

4.8 ZPRACOVÁNÍ CHYB 142
Duplikace deskriptoru (**>&**), pořadí operátorů přesměrování
Priorita operátorů shellu

4.9 ŠETŘÍME DISKY 143
Složený příkaz (**{}**) a přesměrování výstupu
Vliv vyrovnávacích pamětí (cache)

4.10 ČTENÍ „PER PARTES 144
Složený příkaz (**{}**) a přesměrování vstupu
Příkaz **read**
Interní příkazy shellu
Interakce s uživatelem

4.11 KDYŽ READ NEČTE 145
Procesy, subshell (**()**)
Čtení z roury
Možnost implementace některých příkazů jako interní

4.12 KDYŽ READ PARSUJE 147
Příkaz **read** s více parametry

4.13 PARAMETRY 148
Poziční parametry a jejich počet (**$#**)
Příkaz **set**: nastavení pozičních parametrů
Zápis použití parametru/proměnné ve tvaru **${***jméno***}**

4.14 HELLO, WORLD 149
Jméno skriptu (**$0**)
Příkaz **basename**

4.15 ŠKATULATA, HEJBEJTE SE 150
Expanze všech pozičních parametrů (**$@** vs. **$***)
Příkaz **shift**

4.16 DYNAMICKÝ VSTUP 152
Here-dokument (**<<**) a jeho variace
Podmíněné substituce (**${***jméno***:-***slovo***}** a **${***jméno***:=***slovo***}**)
Čtení z roury: řešení pomocí here-dokumentu

4.17 AK, PAK 155
Návratová hodnota příkazu (**$?**)
Podmíněné spuštění příkazu (**||** a **&&**)

4.18 AK, PAK, INAK 156
Podmíněný příkaz **if**
Příkaz **test** resp. **[**
Negace návratové hodnoty (**!**)
Výpis chybových zpráv

4.19 KDYŽ IF NESTAČÍ 159
Řídicí struktura **case**
Testování řetězců proti vzoru
Příkaz **exit**

4.20 PŘÍKAZ CASE NARUBY 161
Dynamická návěští u příkazu **case**

4.21 CYKLUS S PŘEKVAPENÍM 162
Cyklus **for**
Příkaz **continue**
Editor **ed** a reakce na nenalezení vzoru v adrese

4.22 CYKLUS BEZ PŘEKVAPENÍ 164
Cyklus **while**
Cykly a přesměrování, čtení souboru
Návratová hodnota příkazu **read**
Příkaz **read**: vliv proměnné **IFS**, zpětná lomítka ve vstupu
Nastavení proměnné prostředí pro jeden příkaz
Příkaz **break**

4.23 CYKLUS, JAK HO (NE)ZNÁME 166
Aritmetika v shellu: příkaz **expr**, konstrukce **$(())**
Délka hodnoty proměnné (**${#***jméno***}**)

4.24 POTÍŽE S RELATIVITOU 168
Zpracování relativních cest
Efektivní kombinace řídicích struktur
Používání shellových proměnných ve skriptech pro utility
Komentáře v shellu

4.25 AMOEBA 171
Zpracování parametrů skriptu
Využití textového a interpretačního charakteru shellu
Ladění shellskriptů, přepínače **-x**, **-v** a **-n**

4.26 JAK NA PARAMETRY 173
Zpracování přepínačů, příkaz **getopts**

4.27 PARAMETRY LÉPE 176
Funkce v shellu

4.28 PARAMETRY DO TŘETICE 177
Podrobnější popis postupu zpracování řádky, příkaz **eval**
Funkce v shellu: parametry, volání, příkaz **command**
Pseudosoubory **/dev/tty**, **/dev/stdin**, **/dev/stdout** a **/dev/stderr**

4.29 POSLEDNÍ PARAMETR 182
Nedestruktivní zpracování parametrů
Funkce v shellu: globalita proměnných

4.30 NEZNÁMÝ FORMÁT 184
Další využití textového a interpretačního charakteru shellu
Příkaz **ps**

4.31 INTERAKTIVNÍ MAZÁNÍ 185
Interakce s uživatelem v cyklu s přesměrováním
Příkaz **exit** v cyklu za rourou
Duplikace vstupního deskriptoru (**<&**)
Přenositelný příkaz pro výpis bez odřádkování

4.32 REKURZE A PROMĚNNÉ 188
Prohledávání stromu do hloubky, rekurzivní funkce v shellu
Funkce vs. volání skriptu: lokální proměnné, vracení hodnoty
Příkaz **return**

4.33 REKURZE A PROCESY 191
Vracení hodnoty při volání pomocných skriptů
Číslo procesu (**$$**), číslo rodičovského procesu (**PPID**), volání **$0**
Mazání pomocných souborů, obsluha signálů, příkaz **trap**
Optimalizace volání utilit

4.34 PROCESY A PROMĚNNÉ 195
Exportované proměnné, příkaz **export**, přepínač **-a**
Formátování příkazem **printf**, formátovací direktiva **%s**
Zákaz expanze jmen souborů, přepínač **-f**
Příkaz **set**: nastavování přepínačů
První řádka skriptu ve tvaru **#!**
Věčný cyklus pomocí příkazů **while** a **:**, příkazy **true** a **false**
Vložené volání skriptu příkazem **.** (tečka)

4.35 NĚCO O EFEKTIVITĚ 201
Pole v shellu a jejich náhrada

4.36 O EFEKTIVITĚ JEŠTĚ JEDNOU 203
Čtení z různých zdrojů, otevření deskriptorů, příkaz **exec**
Třídění vstupů pro **join**, správné zadání klíčů pro **sort**

4.37 KDYŽ TEXT NESTAČÍ 207
Zpracování binárních souborů, příkazy **dd** a **od**
Práce se speciálními znaky CR a NUL
Efektivita různých variant volání příkazů **head**, **tail** a **dd**

Část 5 Filtr AWK

5.1 ZASE OD PÍKY 215
Programovatelný filtr **awk**
Pole vstupní řádky, oddělovač polí, odkaz na pole
Příkaz **print**

5.2 POČITADLO DÉLEK SOUBORŮ 216
Struktura programu v **awk**, vzor, akce
Typy vzorů, **BEGIN** a **END**
Regulární výrazy v **awk**
Proměnné v **awk**, jejich hodnota, přiřazovací operátor
Aritmetické operátory, funkce **int**

5.3 NENÍ REGEXP JAKO REGEXP 218
Oddělovač polí, přepínač **-F**
Dialekty regulárních výrazů, rozšířené regulární výrazy (**+**, **?**, **|**)
Porovnávací a logické operátory
Implicitní akce

5.4 KONEČNÝ AUTOMAT 220
Charakter programování v **awk**, konečný automat
Procházení větví programu, příkaz **next**
Příkaz cyklu **for**, inkrement, dekrement
Podmíněný příkaz **if**, složený příkaz
Proměnná **NF**, odkaz na vstupní záznam (**$0**)
Funkce **index** a **substr**
Syntaxe **awk**, konec řádky, středník, zpětné lomítko

5.5 A ZASE SKUPINY 223
Zadání vstupních souborů, proměnná **FILENAME**
Operátor shody s regulárním výrazem (**~**)
Operátor zřetězení řetězců
Verze **awk**
Funkce **match**

5.6 SKUPINY S PARAMETREM 225
Předání hodnot z shellu do **awk**, přiřazení proměnné jako parametr
Rizika používání shellových proměnných v kódu pro **awk**
Kombinovaný vstup, parametr „**-**“
Proměnná **NR**

5.7 TRUMFOVÉ ESO AWK 228
Indexované proměnné, asociativní pole, vícerozměrná pole
Funkce **split**
Vlastní funkce, parametry, lokální proměnné, příkaz **return**

5.8 UKAŽ, CO MÁŠ 231
Procházení pole, příkaz **for in**
Varianty operátorů **++** a **--**

5.9 INTERAKCE 232
Zadání oddělovače polí pomocí proměnné **FS**
Příkaz **printf**, formátovací direktiva **%d**
Příkaz **print** a proměnná **OFS**
Výstupní příkazy a přesměrování
Zápis řetězců, escape-sekvence
Příkaz **exit**
Předávání výsledků volajícímu shellu

5.10 EDITACE 235
Modifikace vstupní řádky
Změna hodnoty proměnné **FS** za běhu, sémantika hodnoty **FS**
Funkce **sub** a **gsub**

5.11 FORMÁTOVÁNÍ 237
Funkce **sprintf**, další modifikátory formátovacích direktiv
Funkce **length**

5.12 KDYŽ ŘÁDKA NENÍ ŘÁDKA 239
Čtení HTML, oddělovač záznamů, proměnná **RS**
Příkaz **print** a proměnná **ORS**
Vzor ve tvaru rozsahu
Regulární výrazy: alternativa

5.13 BLOKY ŘÁDEK 241
Čtení bloků řádek, proměnná **RS** jako prázdný řetězec
Cyklus **while**

5.14 PARSER 243
Složitější konečný automat
Příkaz **exit** a návratová hodnota **awk**
Ladění **awk** programů

5.15 KONFIGURACE PROGRAMU 245
Systémová proměnná **LOGNAME**
Předání více parametrů, sloučení více zdrojů, logika řazení větví
Pole **ENVIRON**

5.16 MAMINKA MĚLA PRAVDU 249
Limit počtu otevřených souborů, řešení
Dělba práce mezi shellem a **awk**
Funkce **close** a **system**

5.17 JAK NEPODLEHNOUT REKLAMĚ 252
Efektivita plynoucí z důkladné analýzy
Roura v **awk**
Příkazy **do-while**, **break** a **continue**
Příkaz **ps** a divergence unixových klonů

5.18 PODSOUVÁNÍ SOUBORŮ 255
Proměnné **ARGC** a **ARGV**
Význam oddělovače polí u funkce **split**

Část 6 Případové studie

6.1 STAHOVÁNÍ WEBOVÝCH STRÁNEK 261
Rozbor předností řešení v shellu, editoru a **awk**
Výpis aktuálního nastavení přepínačů shellu (**$-**)
Příkaz **netcat**
Příkaz **mkdir**: vytvoření plné cesty (**-p**)
Podmíněný výraz v **awk**, skript v souboru (**-f**)

6.2 STAHOVÁNÍ DO HLOUBKY 275
Kompilace řešení, problematika rekurze
Současné čtení a zápis jednoho souboru
Příkaz **tr**: mazání znaků (**-d**)
Příkaz **grep**: hledání pevných řetězců (**-F**) a přesné shody (**-x**)

6.3 WWW NEWS 283
Zpracování elektronického dopisu pomocí **awk**, předání dat shellu
Zamykání paralelního přístupu, noclobber režim shellu (**-C**)
Příkazy **until**, **sleep** a **mailx**

6.4 PROHLEDÁVÁNÍ POŠTY 290
Substituce podřetězců (**${***jméno***#***vzor***}**, **${***jméno***%***vzor***}**)
Rozbor rychlosti řešení v shellu, editoru a **awk**

Část 1 Úvod

V této části se seznámíme se základními zásadami při práci s příkazovou řádkou, voláním příkazů, používáním parametrů, kombinováním příkazů, přesměrováním apod. Poznáme několik důležitých utilit (pomocných programů) a velmi stručně se dotkneme problematiky regulárních výrazů.

1.1. Začneme zlehka

Zadání:

Vypište jméno aktuálního pracovního adresáře.

Souborový systém v Unixu tvoří hierarchickou strukturu. Jednotlivá „patra“ tvoří adresáře (složky), které obsahují soubory a další podadresáře. Adresářová struktura má tak vlastně podobu *stromu* vyrůstajícího z tzv. *kořene*.

Chceme-li vyjádřit umístění nějakého souboru nebo adresáře, musíme zapsat tzv. *cestu*, která říká, přes jaké adresáře se dostaneme k cíli. Názvy jednotlivých adresářů se v cestě oddělují (obyčejným) lomítkem („/“).

Každý běžící program v systému má nějaký adresář nastaven jako svůj pracovní adresář. Podstatou úlohy je vypsat cestu k aktuálnímu pracovnímu adresáři shellu.

Rozbor:

Přesně pro tento účel slouží příkaz **`pwd`** (*print working directory*).

Řešení:

```
sinek:~> pwd
/home/forst
sinek:~>
```

Poznámky:

Po přihlášení do systému bude aktuálním adresářem váš *domovský* adresář – ten, který vám byl vytvořen při založení uživatelského účtu a ve kterém si můžete vytvářet soubory a podadresáře dle chuti. Změnit pracovní adresář můžete příkazem **`cd`** (*change directory*).

Pojem cesty budete možná v dnešní době znát z jiného prostředí, kam se vcelku intuitivně přenesl, a to jako cestu v internetových adresách stránek systému WWW:

```
http://www.yq.cz/trail-o/WTOC2008_results.pdf
```

I zde cesta (zjednodušeně řečeno) vyjadřuje umístění souboru s textem webové stránky.

Můj počítač se jmenuje „`sinek`“. Není to gramatická chyba, jméno má původ v anglickém slovese „think“. V současnosti bývá zvykem, že když vás unixový počítač vyzývá k zadání příkazu (neboli vypisuje tzv. *prompt*), představuje se svým jménem. Za jménem a dvojtečkou vidíte znak „`~`“ (vlnka), jenž znamená, že se právě nacházíme v mém domovském adresáři. Za vlnkou je ještě zobáček a mezera, které oddělují prompt od mnou napsaných příkazů, tzv. *příkazové řádky*. Přesný formát promptu je hodně závislý na nastavení prostředí. Pokud váš počítač vypisuje prompt v jiné podobě, nenechte se tím vůbec zmást.

1.2. Přidáme parametry

Zadání:

Vypište obsah adresáře `/etc`, včetně detailních informací o jednotlivých souborech.

Rozbor:

Pro tento úkol opět existuje správný příkaz (**`ls`**), ale tentokrát budeme muset přidat za název příkazu i nějaké parametry. Kdybychom zavolali příkaz bez nich, vypsal by pouze seznam jmen souborů a podadresářů, a to v aktuálním adresáři.

Přidáme tedy parametr „**`-l`**“ (*long*), který zařídí tzv. „dlouhý“ výpis. Každému souboru v něm přísluší jedna celá řádka s informacemi, jako jsou přístupová práva, jméno vlastníka, velikost, datum a čas poslední modifikace a jméno souboru.

A druhým parametrem příkazu sdělíme, že chceme vypisovat obsah adresáře `/etc`.

Řešení:

```
sinek:~> ls -l /etc
total 586
drwxr-xr-x  2 root  wheel       512 Oct  1 03:30 X11
... atd.
```

Poznámky:

U unixových příkazů se tradičně významově odlišují parametry, které začínají minusem (tzv. *přepínače* nebo optiony) – ty ovlivňují, *co* přesně příkaz dělá (v našem případě dlouhý výpis) – a ostatními parametry (tzv. *operandy*), které určují, *s čím* to dělá (v našem případě jméno adresáře). Všechny parametry se navzájem oddělují mezerami.

Popis parametrů a hlavně všech přepínačů člověk samozřejmě nenosí v hlavě. Návod (tzv. *manuálovou stránku*), jak s příkazem pracovat, zobrazíme příkazem **`man`**:

```
sinek:~> man ls
```

Pokud neznáme název potřebného příkazu, máme možnost název manuálové stránky vyhledat pomocí klíčového slova, pokud zadáme přepínač „**`-k`**“ (*keyword*):

```
sinek:~> man -k directory
```

Jednou ze základních jednotících myšlenek Unixu je zásada, že (skoro) všechno je soubor. Tento princip ulehčuje významně život tvůrcům programů, protože zjednodušuje programátorské rozhraní aplikací (API). Dokonce i na některé informace o běžícím programu se dá nahlížet jako na soubor. Pro nás je ale v této chvíli důležitý jiný důsledek – i adresář je soubor. Budeme tedy od této chvíle důsledně používat termín *obyčejný soubor* (text či data jako posloupnost bytů), pokud by hrozilo, že dojde k záměně s obecným pojmem „soubor“ (ve smyslu „obyčejný soubor i adresář“).

Nyní už si můžeme zpřesnit popis příkazu **`ls`** – vypisuje informaci o souborech, jejichž jména dostal jako parametry, a to tak, že v případě adresáře vypíše jeho obsah (seznam souborů a informace o nich), v případě obyčejného souboru vypíše jen jeho jméno (resp. informace o něm). Vyzkoušejte si třeba příkaz:

```
sinek:~> ls -l /etc/passwd
```

Kdybyste chtěli vidět pouze informace o adresáři `/etc` samotném a nikoliv jeho obsah, museli byste použít další přepínač příkazu **`ls`**, a to „**`-d`**“ (*directory*):

```
sinek:~> ls -l -d /etc
```

Většina unixových příkazů ovšem umožňuje přepínače *slučovat*, tj. napsat jich více za sebou s jediným znakem „**`-`**“. Předchozí ukázku by tedy bylo možné napsat jako:

```
sinek:~> ls -ld /etc
```

Stejně tak je obvyklé, že příkazy dokážou akceptovat více operandů. Pokud bychom chtěli vypisovat současně ještě obsah dalšího adresáře, například `/tmp`, můžeme napsat:

```
sinek:~> ls -l /etc /tmp
```

Všechny doposud uvedené cesty v parametrech, byly cesty *absolutní* – začínaly lomítkem a popisovaly cestu od kořene. Jinou možností, jak popsat umístění souboru, je *relativní* cesta od aktuálního adresáře. Poslední ukázku bychom mohli změnit např. na:

```
sinek:~> cd /
sinek:/> ls -l etc tmp
```

Všimněte si, že po změně adresáře se změnil obsah mého promptu.

Pokud se při psaní relativní cesty potřebujeme odkázat na *aktuální* adresář, můžeme použít soubor se zvláštním jménem „**`.`**“ (tečka). Volání **`ls`** bez parametrů by tedy bylo možné nahradit třeba voláním:

```
sinek:~> ls .
```

A v každém adresáři je ještě jedno zvláštní jméno, které představuje *rodičovský* nebo *nadřazený* adresář, a to „**`..`**“ (dvě tečky). Používá se pro pohyb „vzhůru“ po adresářové struktuře. Můj domovský adresář by bylo možné (zbytečně složitě) vypsat pomocí:

```
sinek:~> ls -l ../../home/forst
```

(neboli: „vyskoč o dvě patra výš a pak sestup do podadresářů `home` a `forst`“).

Pokud v parametru uvedené jméno (cesta) není jménem existujícího souboru, příkaz skončí chybou. Toho se dá například využít pro jednoduchý test existence souboru.

Názvy příkazů v Unixu bývají většinou „logické“ (jen je občas nutné si na tu logiku poněkud zvyknout). Název příkazu **`ls`** patří mezi výjimky potvrzující toto pravidlo. Zkratka pochází ze slov „list segments“ a je reliktem, jakousi zkamenělinou pamatující předchůdce Unixu, systém Multics. Tenhle název si prostě musíte zapamatovat.

1.3. Poprvé kombinujeme

Zadání:

Vypište počet uživatelů systému.

Seznam uživatelů se nachází v souboru **`/etc/passwd`**. Soubor je textový a každému uživateli přísluší jedna řádka (její obsah si probereme v poznámkách):

```
root:*:0:0:Super User:/root:/bin/bash
... atd.
forst:*:1004:1004:Libor Forst:/home/forst:/bin/bash
```

Rozbor:

Tento úkol už nás donutí přemýšlet v souvislostech. Počet uživatelů se rovná počtu řádek souboru `/etc/passwd`, takže by nám stačilo je jen spočítat.

Počítat řádky umí příkaz **`wc`** s přepínačem „**`-l`**“. A abychom k němu řádky souboru dostali, použijeme operátor přesměrování „**`<`**“ s operandem `/etc/passwd`. Shell potom otevře soubor (`/etc/passwd`) a předá ho na vstup příkazu **`wc`**.

Řešení:

```
sinek:~> wc -l < /etc/passwd
      27
```

Poznámky:

Pokud jste zvyklí na jiné operační systémy, možná vás překvapí, že seznam uživatelů je uložen v *textovém* a nikoliv v *binárním* souboru. Textový soubor se člení se na řádky obsahující pouze tisknutelné znaky a ukončené znakem konce řádky (*line feed*, LF). Takový soubor lze číst obyčejným editorem a zpracovávat (namísto jednoúčelových speciálních nástrojů) zcela obecnými programy (*utilitami*). A takových programů má UNIX velké množství. Je to důsledek historického vývoje a dle mého nejhlubšího přesvědčení je to důsledek dobrý.

Jednotlivé sloupce (nebo také pole) řádek v souboru `/etc/passwd` jsou odděleny dvojtečkami a obsahují pro každého uživatele následující atributy:

- jméno nebo též *přihlašovací jméno* (*login*);
- zakódované heslo (dnes už tam obvykle nebývá, ale pole samo v souboru kvůli kompatibilitě zůstalo a bývá vyplněno třeba hvězdičkou);
- číslo uživatele (*UID*);
- číslo (primární) skupiny uživatele (*GID*);
- *plné jméno* uživatele;
- domovský adresář uživatele;
- *login-shell* (název, resp. cestu k shellu, který se uživateli spouští při přihlášení).

Soubor může ještě obsahovat komentářové řádky (začínající znakem „#“). Ty se budeme muset časem naučit ze zpracování vyloučit. Prozatím je budeme ignorovat.

K tomu, aby programy v UNIXu mohly komunikovat se svým okolím, slouží vstupní a výstupní *proudy dat*. Každý program má implicitně otevřen tzv. *standardní výstup* a má ho přiřazen na obrazovku terminálu, na kterém běží. Díky tomu jsme ostatně mohli vidět výsledky všech doposud spouštěných příkazů. Stejně tak má každý program přiřazen i *standardní vstup*. Tím je implicitně klávesnice terminálu, kde program běží. My jsme toto přiřazení zrušili operátorem přesměrování. Kdybychom to neudělali, program **wc** by čekal na to, co naťukáme na klávesnici terminálu, a nakonec by nám to spočítal (tedy, spočítal by nám napsané řádky). Vyzkoušejte si to. Ale nejprve dočtěte odstavec! Až naťukáte celý text, budete totiž muset nějak vyjádřit, že už nic dalšího psát nehodláte. K tomu slouží kombinace kláves Ctrl+D, kterou zmáčknete po odřádkování poslední řádky.

Zastavme se ještě u pojmu *terminál*. Pochází rovněž z prehistorie. Kdysi označoval hardwarové zařízení, které mělo klávesnici a obrazovku (a ještě před tím třeba psací stroj...) a sloužilo pro komunikaci s počítačem. Dnes je jeho pozůstatkem softwarová komponenta, která uživateli poskytuje přesně totéž – klávesnici a obrazovku, ze které budete moci svůj UNIX ovládat.

Nabízela se ještě jedna možnost, jak soubor příkazu předat. Když napíšeme název souboru jako operand na příkazové řádce, nebude **wc** číst vstup, ale zadaný soubor. Tak se ostatně chová většina unixových utilit:

```
sinek:~> wc -l /etc/passwd
      27 /etc/passwd
```

Výsledek ale nesplňuje zadání, protože tam překáží jméno souboru.

Kdo se vydá studovat manuálovou stránku (příkazem „**man wc**“), aby našel přepínač, kterým potlačí výpis jména souboru, jde na věc zcela správně, systémově a prokázal velkou míru pochopení problematiky. Bohužel neuspěje. Příkaz takový přepínač nemá.

Opravdoví programátoři pravděpodobně začnou dumat nad tím, jak z výstupu jméno odfiltrovat nějakým dalším programem. To, pochopitelně, jde. Ale my to zatím neumíme a dokonce i poté, co to umět budeme, nebude to nejlepší řešení.

Nejjednodušší řešení, které jsme také nakonec použili, nám nabídl „selský rozum“. Když program nebude jméno souboru znát, neboť nečte soubor, ale svůj standardní vstup, nemůže žádné jméno vypsat.

Poněkud podezřelé jméno příkazu **wc** pochází z anglických slov *word count* (počítej slova), přestože se příkaz většinou používá pro počítání řádek či znaků. Proto jsme také museli použít přepínač „**-l**“ (*lines*), abychom dali najevo, že nás zajímají jenom řádky.

1.4. Další přesměrování

Zadání:

Vytvořte soubor `datum` obsahující řádku s textem „`datum`“.

Rozbor:

V minulé kapitole jsme poznali operátor přesměrování standardního vstupu. Dá se očekávat, že analogicky bude existovat operátor pro přesměrování standardního výstupu. A dokonce se dá i odhadnout, že oním operátorem bude znak „**>**“.

Kdybychom našli příkaz, kterým potřebný text vypíšeme, a výpis přesměrovali do souboru, máme vyhráno. Hledaným příkazem je **`echo`**, jehož úkolem je pouze opsat svoje parametry na svůj výstup a odřádkovat.

Řešení:

```
sinek:~> echo datum > datum
```

Poznámky:

Operátory přesměrování jsou prvním viditelným případem tzv. *metaznaků*, neboli znaků, které mají v shellu zvláštní význam. Brzo jich poznáme ještě celou řadu. Vlastně pár dalších jsme už viděli. Tedy vlastně spíš neviděli. Metaznakem je totiž i *mezera* – odděluje od sebe název příkazu a jednotlivé parametry – a také *znak konce řádky* (který má na svědomí klávesa Enter) – ten shellu říká, že už nic dalšího nechceme dodat a že má celou řádku zpracovat.

Kdybychom nepoužili operátor přesměrování, výstup příkazu (text „`datum`“) by se objevil na obrazovce terminálu.

Pokud se chcete podívat, jestli soubor obsahuje to, co má, nejsnazší je vypsat si ho příkazem **`cat`**:

```
sinek:~> cat datum
datum
```

Pozor na záměnu příkazů **`echo`** a **`cat`**. Oba mohou mít parametry a oba něco vypisují na výstup (proto se velmi často pletou). Ale **`echo`** svoje parametry pouze opisuje, zatímco **`cat`** má jako parametry jména souborů a vypisuje jejich obsah.

Název příkazu **`cat`** pochází ze slovesa *concatenate* (zřetězit), protože příkaz umí zřetězit na výstup výpis více souborů. A tak, přestože se převážně používá pro vypsání jediného souboru, zůstalo mu toto „podivné“ jméno.

1.5. Nepatrně přidáme

Zadání:

Vytvořte soubor `datum` obsahující dvě řádky, na první je text „`Datum:`“ a na druhé aktuální datum.

Rozbor:

Začneme asi stejně jako minule, opět použijeme příkaz **`echo`** s požadovaným textem.

Text druhé řádky vytvoříme příkazem **`date`**, ale budeme ji nějak muset připsat do našeho souboru. Tady použijeme jinou variantu operátoru přesměrování, která výsledný soubor nepřepisuje, ale zapisuje na jeho konec („**`>>`**“).

Řešení:

```
sinek:~> echo Datum: > datum
sinek:~> date >> datum
```

Poznámky:

Shrňme si do tabulky základní rozdíly v chování obou operátorů přesměrování:

	operátor **`>`**	operátor **`>>`**
soubor neexistuje	shell soubor vytvoří, zapíše nový obsah	shell soubor vytvoří, zapíše nový obsah
soubor existuje	shell vymaže obsah souboru, zapíše nový obsah	obsah souboru se zachová, nový obsah se zapíše na konec

Nic nám nebrání použít v obou příkazech operátor „**`>>`**“ – konečný výsledek by byl stejný. Problém by nastal až při opakovaném provádění naší série příkazů – po druhém provedení by soubor `datum` obsahoval už čtyři řádky. Pokud bychom toto řešení chtěli použít, museli bychom na začátku buď soubor `datum` smazat příkazem **`rm`** (*remove*), nebo vymazat jeho obsah (o to se pokusíme v dalších kapitolách).

Manuálová stránka příkazu **`date`** nabízí ještě další řešení – je totiž možné předepsat *formát* výstupu příkazu. Formátovací řetězec může obsahovat znaky, které příkaz jen opíše na výstup, a kromě nich i zvláštní *direktivy*, které začínají znakem procenta a představují požadavky na vložení určitých komponent aktuálního data a času, ale také například znaku konce řádky („**`%n`**“):

```
sinek:~> date +Datum:%n%d/%m/%Y
Datum:
13/03/2009
```

Direktiva „**`%d`**“ zastupuje pořadové číslo dne v měsíci, „**`%m`**“ číslo měsíce a „**`%Y`**“ rok. Pozor na rozdíly mezi malými a velkými písmeny – zápis „**`%y`**“ by představoval pouze poslední dvojčíslí roku. V UNIXu rozhodně nejde malá a velká písmena zaměňovat.

1.6. A zase ubereme

Zadání:

V souboru `datum` z minulé kapitoly smažte úvodní řádku.

Rozbor:

Přirozeným řešením úlohy je použití nějakého editoru, který by dokázal upravit soubor dle našich potřeb. Takové nástroje v UNIXu samozřejmě existují, ale s těmi ještě chvíli počkáme, protože k vyřešení této úlohy nám postačí jednodušší prostředky.

Pomůže nám příkaz **`tail`**, pomocí nějž vypíšeme poslední řádku do jiného souboru a příkaz **`mv`**, jímž soubor zase přejmenujeme na původní název.

Řešení:

```
sinek:~> tail -n 1 datum > /tmp/datum
sinek:~> mv /tmp/datum datum
```

Poznámky:

Příkaz **`tail`** opisuje na výstup konec souboru. Pro zadání počtu *řádek*, které má vypsat, se používá přepínač „**`-n`**“ (*number*) s povinnou *hodnotou*. Hodnotu (argument) můžeme zapsat buď jako další parametr, nebo ji „slepit“ přímo s přepínačem („**`-n1`**“).

Namísto zadání počtu řádek je možné pomocí přepínače „**`-c`**“ (*char*) nechat vypsat z konce souboru určitý počet *znaků*.

Příkaz **`mv`** (*move*) přesouvá či přejmenovává soubory a adresáře.

Možná vás napadlo zdánlivě jednodušší řešení:

```
sinek:~> tail -n 1 datum > datum
```

A možná vám i funguje. Ale tím se nenechte oklamat. Správně by nemělo! V případě „správného“ chování bude výsledkem prázdný soubor. Vytvoření, resp. přemazání souboru určeného pro výstup pomocí přesměrování dělá totiž shell ještě před spuštěním příkazu. Jakmile se pak program **`tail`** rozběhne, najde soubor `datum` už beznadějně prázdný. Jediné správné řešení je použití dočasného souboru.

V unixovém souborovém systému existuje několik tradičních adresářů s víceméně tradičním obsahem. Příkladem může být třeba právě adresář **`/tmp`**, který je určen pro ukládání dočasných souborů. Je třeba si pouze dát pozor na důvtipnou volbu jmen, aby nedošlo ke kolizi s jiným uživatelem či programem. V tomto ohledu je námi použité jméno přímo ukázkovým příkladem špatné volby. Jak si při vytváření jmen můžeme pomoci technickými prostředky, poznáme později (v kap. 4.33).

1.7. Zametáme stopy

Zadání:

Upravte soubor `datum` z minulé kapitoly zase do původního tvaru (znovu napište původní první řádku), ale zachovejte přitom souboru datum a čas poslední modifikace.

Rozbor:

Připsání první řádky nevypadá na první pohled nijak složitě, ale bez editoru nám nezbude, než celý soubor vlastně vytvořit znova. První řádku opět zapíšeme příkazem **`echo`** a za ni připíšeme původní obsah souboru příkazem **`cat`**. Rozmyslete si, že připsání řádky na konec by bylo výrazně jednodušší...

Náš nový soubor ale bude mít jako čas poslední modifikace uložen aktuální čas. Pro „zametení stop“ použijeme příkaz **`touch`**. Jeho název („dotkni se“) naznačuje, že se příkaz souboru pouze „dotkne“, tj. změní datum a čas poslední modifikace souboru. Implicitně nastavuje aktuální čas, ale je možné zvolit buď nastavení konkrétního časového okamžiku (přepínač „**`-t`**“), nebo převzít čas poslední modifikace jiného, referenčního souboru (přepínač „**`-r`**“). A to přesně potřebujeme.

Řešení:

```
sinek:~> echo Datum: > /tmp/datum
sinek:~> cat datum >> /tmp/datum
sinek:~> touch -r datum /tmp/datum
sinek:~> mv /tmp/datum datum
```

Poznámky:

Datum a čas poslední modifikace je jedním z časových údajů, které se uchovávají pro každý soubor. A je to také časový údaj, který zobrazuje příkaz „**`ls -l`**“. Příkaz **`mv`** nemění obsah souboru, a proto také nemění tento časový údaj.

Aby příkaz **`touch`** mohl změnit datum a čas nějakého souboru, musí takový soubor existovat. Pokud neexistuje, je nutné, aby ho **`touch`** vytvořil. A nemaje žádná data, vytvoří ho prázdný (jeho délka je nula).

Když jsme v kapitole 1.5 popisovali funkci operátoru „**`>>`**“, přemýšleli jsme, jak vyrobit soubor s délkou nula. Možná si v této chvíli řeknete, že toto je hledané řešení. Ale pozor! My jsme hledali, jak délku nula dosáhnout u souboru už existujícího (a tedy vymazat jeho stávající obsah). A to příkaz **`touch`** nezařídí – změní mu jen časové údaje a soubor jinak ponechá beze změny.

1.8. Dárek k narozeninám

Zadání:

Vytvořte soubor se jménem „`darek k narozeninam`“ s časem poslední modifikace shodným s datem vašeho narození.

Rozbor:

Příkaz **`touch`** už známe, musíme pouze nastudovat, jak se zadává přesné datum. A zjistíme, že budeme potřebovat ještě hodiny a minuty. Buď si vzpomenete, kdy jste se narodili, nebo zadáte třeba nuly.

```
sinek:~> touch -t 196302270615 darek k narozeninam
```

Jenže když se podíváme na výsledek, nebudeme spokojeni:

```
sinek:~> ls -ltr
-rw-r--r--  1 forst  forst  0 Feb 27  1963 darek
-rw-r--r--  1 forst  forst  0 Feb 27  1963 k
-rw-r--r--  1 forst  forst  0 Feb 27  1963 narozeninam
```

Shell předal příkazu **`touch`** tři parametry ten vytvořil tři soubory. Tomu musíme zabránit. Musíme shellu říci „vezmi všechna tři slova jako jeden řetězec, mezery nepovažuj za svůj metaznak, nejsou to oddělovače parametrů“. To můžeme udělat několika způsoby, nejsnazší je uzavřít celý text do apostrofů („**`'`**...**`'`**“).

Řešení:

```
sinek:~> touch -t 196302270615 'darek k narozeninam'
```

Poznámky:

Přepínač „**`-t`**“ (*time*) říká příkazu **`ls`**, aby soubory pro výpis setřídil podle data a času poslední modifikace. To je velmi užitečný přepínač. Přepínač „**`-r`**“ (*reverse*) zase říká, aby soubory seřadil pozpátku (od nejstaršího k nejmladšímu).

Povšimněte si, že formát výpisu data je jiný, než jsme doposud viděli. Namísto času (hodin a minut) se vypisuje rok. To se děje u souborů, které se neměnily v posledním půl roce (u nich je rok důležitější než čas). Tento princip má dvě výhody (šetří místo ve výpisu při zachování stejného počtu sloupců) a jednu nevýhodu (norma nedává možnost vypsat jednotný „plný“ formát data v případě potřeby).

Údaj o času je uložen jako celé číslo se znaménkem a jeho hodnotou je počet sekund od začátku „počítačové epochy“ (Nový rok 1970). Moje datum narození bude tedy ve skutečnosti záporné (něco jako novodobé „př. n. l.“). Problémy nás čekají v roce 2038, kdy se vyčerpá kapacita těch čtyř bytů, v nichž je čas uložen...

1.9. Hvězdička

Zadání:

Aktualizujte datum a čas poslední modifikace všem souborům v aktuálním adresáři, jejichž jméno začíná písmenem „`d`“ a končí „`m`“.

Rozbor:

Vše, co potřebujeme k faktickému provedení požadované operace, už umíme. Jediné, co neumíme, je předat příkazu **`touch`** jména všech souborů, aniž bychom je museli opisovat. Zde použijeme další metaznak shellu, *hvězdičku*.

Hvězdička patří mezi tzv. *expanzní znaky* (*wildcards*). Pomocí nich můžeme popsat určitou množinu jmen souborů, které nás zajímají, a shell tato jména místo nás zapíše (*expanduje*) mezi parametry příkazu, než ho spustí. Hvězdičku můžete číst jako „cokoliv“. Řetězci složenému z expanzních znaků shellu (jejich počet není nijak omezen) se říká *porovnávací vzor* a my použijeme porovnávací vzor ve tvaru „**`d*m`**“.

Řešení:

```
sinek:~> touch d*m
```

Poznámky:

Expanzi jmen souborů provádí shell před tím, než parametry předá volanému programu. Program **`touch`** tedy namísto našeho řetězce „**`d*m`**“ už uvidí pouze (abecedně seřazený) seznam všech souborů (v aktuálním adresáři) se jmény začínajícími „`d`“ a končícími „`m`“. Stejně jako kdybychom napsali:

```
sinek:~> touch 'darek k narozeninam' 'datum'
```

Jedinou výjimkou, kdy program expanzní znaky uvidí, je případ, kdy shell žádný vhodný soubor nenajde. Pak programu předá parametr beze změny. Rozmyslete si, co by to v našem případě znamenalo – příkaz by vytvořil soubor se jménem „`d*m`“. S tím by se nám dost obtížně pracovalo...

To ale u nás nenastane. Shell najde v aktuálním adresáři přinejmenším soubory „`darek k narozeninam`“ a „`datum`“. Dokonce zvládne správně ošetřit i to, že náš dárek k narozeninám má ve jméně mezery. Celé takové jméno předá programu jako jeden parametr.

Hvězdičku možná znáte z příkazové řádky MS Windows. Ale tam ji zpracovává každý program sám. V UNIXu má tuto činnost na starosti shell a jednotlivé programy ji už nemusejí řešit.

1.10. Záludná jména

Zadání:

Vytvořte soubor se jménem „`-f`“ a zase ho smažte.

Rozbor:

Vytvoření souboru už nám nečiní žádný problém. Tentokrát to můžeme pro změnu zase vyřešit pomocí přesměrování výstupu.

Ale pokus o smazání selže. Navíc selže záludně – tváří se, že vše je v pořádku:

```
sinek:~> echo > -f
sinek:~> rm -f
sinek:~>
```

Jenomže soubor pořád existuje:

```
sinek:~> ls -l -f
... atd.
-rw-r--r--   1 forst  forst     1 Nov  27 02:05 -f
```

Důvodem neúspěchu je to, že příkaz **`rm`** pochopil parametr „**`-f`**“ jako svůj přepínač a nikoliv jako název souboru. Chtělo by se nám říci: „tohle minus není žádné minus od nějakého přepínače, to je součást jména souboru“. A možná vás napadne použít podobný postup, jakým jsme v minulé kapitole dokázali dostat do „obyčejného“ textu příkazu „neobyčejné“ znaky:

```
sinek:~> rm '-f'
```

Ale výsledek se nezmění! Tentokrát spočívá problém v tom, že apostrofy skrývají (meta)znaky před shellem, ale minus není metaznak shellu! Pro shell je minus úplně obyčejný znak, a tak jeho uzavření do apostrofů postrádá zcela smysl. Parametry, které od shellu dostane příkaz **`rm`**, budou s apostrofy stejné jako bez nich. Je nezbytné dodat patřičné informace přímo příkazu **`rm`** a ne shellu.

Řešení nabízí zvláštní parametr „**`--`**“, který ukončuje seznam přepínačů. Parametry, které následují, už budou chápány jako operandy, byť by začínaly minusem. Stejnou funkci má podle normy u „řádně vychovaných“ příkazů rovněž jakýkoliv parametr, který minusem nezačíná – co je za ním, už není přepínač, ale operand.

Řešení:

```
sinek:~> rm -- -f
```

Poznámky:

Asi bychom si měli říci, co vlastně přepínač „**`-f`**“ (*force*) příkazu **`rm`** dělá. Jeho prvním úkolem je potlačit případné chybové zprávy – např. o neexistenci mazaného souboru. Druhým úkolem je potlačit případné dotazy, které by jinak program obsluze

kladl, kdyby si nebyl jist, zda má opravdu mazat vše, co dostal zadáno. Například cizí soubory ve vašem adresáři máte sice právo smazat (o přístupových právech si budeme povídat v kapitole 2.1), ale **rm** si raději ověří, že to opravdu chcete.

Právě na přepínače je nutné dát pozor, když v adresáři obsahujícím soubory se záludnými jmény použijete shellové expanzní znaky. Řekněme, že chcete v aktuálním adresáři smazat všechny soubory, které vlastníte. Zadáte zdánlivě nevinný příkaz:

```
sinek:~> rm *
```

Shell za hvězdičku sice dosadí jména úplně všech souborů v adresáři, ale případné cizí soubory přece **rm** nesmaže, protože na to by musel mít zadaný přepínač „**-f**“. Jenže přesně to shell udělá! Vyplní do příkazové řádky jména souborů podle abecedy (resp. tabulky znaků). A podle ní bude soubor se jménem „`-f`“ první! Jak víme, program **rm** nebude nic tušit o naší hvězdičce (expanzi provedl shell), a proto název souboru „`-f`“ pochopí jako přepínač „**-f**“ a neštěstí je hotové...

Jednoduchou metodou, jak zjistit, s jakými parametry bude příkaz ve skutečnosti zavolaný, je napsat před něj „**echo --** “ a spustit.

Problém s divným jménem souboru se (méně nápadně) projevil už u příkazu **ls**, jímž jsme kontrolovali existenci souboru. I tam byl parametr „**-f**“ pochopen jako přepínač a příkaz (bez operandů) vypsal celý aktuální adresář, zato (díky přepínači) bez třídění, v pořadí, v jakém jsou soubory uloženy. I zde by pomohl parametr „**--**“:

```
sinek:~> ls -l -- -f
```

Pro vytvoření souboru jsme použili přesměrování zcela záměrně. Kdybychom zkusili použít **touch**, narazili bychom na problém s divným jménem souboru už při vytváření. Příkaz **touch** by pochopil „**-f**“ jako přepínač a skončil by chybou, protože ho nezná.

Co vlastně obsahuje soubor, který jsme vytvořili? Na první pohled by se mohlo zdát, že nic, že bude mít nulovou délku. Ale podívejte se na výstup příkazu **ls** – soubor má délku jedna. Obsahuje tedy jeden znak. Jaký znak to asi bude? Pokud tipujete, že to bude znak konce řádky, máte pravdu – příkaz **echo** přece odřádkoval! Neboli ani tato konstrukce nám nevyřeší starý známý problém vymazání obsahu souboru.

Berte tuto kapitolu jako varování před přílišnou kreativitou při volbě jmen. I my jsme raději náš „`darek k narozeninam`“ pojmenovali bez „nabodeníček“. I když ani ty mezery ve jméně, stejně jako jiné metaznaky, nejsou moc dobrý nápad...

1.11. První program v shellu

Zadání:

Spočítejte soubory v adresáři `/etc`.

Rozbor:

Jednou ze základních myšlenek, na nichž je založen operační systém UNIX, je fakt, že jednoduché věci by se měly dělat jednoduše.

Chceme-li spočítat soubory v adresáři, nejjednodušší je použít příkaz **`ls`**, který vypíše seznam souborů, a příkaz **`wc`**, který umí počítat řádky. Oba příkazy spojíme znakem „|“ („svislítko“), který způsobí, že vše, co první příkaz (*producent*) vypisuje na svůj standardní výstup, druhý příkaz (*konzument*) najde na svém standardním vstupu, a tedy načte a zpracuje.

Řešení:

```
sinek:~> ls /etc | wc -l
      42
```

Poznámky:

Adresář **`/etc`** obsahuje konfigurační soubory (už jsme viděli soubor **`passwd`**). Pokud vám na vašem počítači vyjde jiná odpověď než 42, neděste se. Každý systém se prostě mírně liší.

Použitá metoda komunikace mezi našimi dvěma příkazy se nazývá *roura* (*pipe*). Opravdu si ji můžete představit jako rouru, do níž producent sype text, a konzument si ho z ní odebírá. Je to velmi jednoduchý a pro programátora velmi pohodlný princip. Operační systém se za nás postará o to, aby nám příliš výkonný producent rouru nepřeplnil a naopak aby příliš rychlý konzument dokázal na svoje data počkat. Ani jeden z účastníků se přitom nemusí starat, kdo je na druhé straně roury.

Ovšem náš příkaz **`ls`** to shodou okolností zrovna dělá. Pokud si ho spustíte jen tak, na obrazovku terminálu, snaží se optimalizovat výsledek své činnosti tak, abyste toho co nejvíce viděli, a proto výstup seřadí do několika sloupců. Takový formát by se rozhodně nedal zpracovat tak snadno, jako jsme to udělali my. Jakmile je výstup **`ls`** přesměrován do roury nebo do souboru, příkaz naštěstí změní formát výstupu a výsledek budeme mít v jediném sloupci.

Je velmi důležité pochopit rozdíl mezi operátory „**|**“ a „**>**“. Vyzkoušejte, co udělá

```
sinek:~> ls /etc > wc -l
```

Operátor „**>**“ znamená přesměrování do souboru, v tomto případě do souboru se jménem „**wc**“. Shell tento operátor i s jeho operandem z příkazové řádky vyjme a dál pokračuje v jejím zpracování (obvykle se přesměrování píše až na konec seznamu parametrů, ale povinné to není). Následující řetězec „**-l**“ se tedy použije jako další parametr pro **ls**. Na standardně se chovající implementaci bude chápán jako jméno souboru (protože seznam přepínačů už ukončil operand „**/etc**“). Na systémech, které „přemýšlejí za uživatele“ bude pochopen jako přepínač, protože takhle to přece určitě uživatel chtěl...

Princip roury je jedním ze základních pilířů operačního systému UNIX. Podle hesla „dělej jednu věc a dělej ji dobře“ se místo komplexních, nepřehledných superprogramů se složitým ovládáním v UNIXu vytvářejí jednoduché specializované utility a ty se pak pomocí rour spojují do větších funkčních celků. Tomuto principu se také říká „divide et impera“ („rozděl a panuj“): složitý problém se rozdělí na menší celky, které se snáze řeší, a pak se jen kontroluje jejich součinnost.

Samozřejmě bychom se mohli obejít bez roury a mezivýsledek ukládat do dočasného pomocného souboru a zase ho z něj číst, akorát nám přibude starost o jeho smazání:

```
sinek:~> ls /etc > /tmp/seznam_souboru
sinek:~> wc -l < /tmp/seznam_souboru
      42
sinek:~> rm /tmp/seznam_souboru
```

Pokud chcete namítnout, že v předchozích kapitolách jsme také používali dočasné soubory a o jejich mazání jsme se nestarali, pak sice máte pravdu, ale až dosud jsme vždy dočasným souborem nahradili náš původní soubor, čímž dočasný soubor zmizel. Pokud tomu tak není, jako v tomto případě, musíme se o uklizení nepotřebného dočasného souboru postarat.

Slovo „program“ v nadpisu kapitoly je poněkud nadnesené. Nicméně právě jste vlastní rukou donutili systém, aby spustil dvě utility, koordinoval přesun dat mezi nimi a nakonec vypsal kýžený výsledek. Takže nakonec – proč ne?

1.12. Počítání souborů lépe

Zadání:

Spočítejte soubory v adresáři, a to včetně souborů skrytých.

Termínem *skryté* se označují soubory, jejichž jméno začíná tečkou.

Rozbor:

Program z minulé kapitoly fungoval jenom proto, že jsme si vybrali vhodný adresář, který skryté soubory (obvykle) neobsahuje. Pokud si vyberete například váš domovský adresář, program bude dávat špatné výsledky. Skryté soubory totiž v systému požívají výsady zvláštního zacházení, a tak třeba příkaz **`ls`** je sám od sebe nevypíše.

Zkusme řešení upravit a využít expanzních znaků shellu. Necháme shell, aby nám do příkazové řádky vložil názvy všech existujících souborů. Samozřejmě přitom nesmíme zapomenout na přepínač „**`-d`**“ (už jsme o tom diskutovali v kapitole 1.2):

```
sinek:~> ls -d * | wc -l
```

Toto řešení má ale ještě stále stejný háček. Skryté soubory totiž ignoruje i shell při nahrazování hvězdičky! Úvodní tečku ve jméně souboru musíme napsat explicitně. Proto budeme muset mezi parametry přidat ještě jeden výraz „**`.*`**“ neboli „všechny soubory, co začínají tečkou a po ní následuje cokoliv (třeba i nic)“.

Zbývá poslední maličkost. Mezi vypsanými soubory se takto objeví i dva poněkud zvláštní „soubory“, a to „`.`“ a „`..`“. Jsou to nám známé formální odkazy na aktuální a nadřazený adresář. Potřebujeme je z výsledku „odečíst“. Nejsprávnější by samozřejmě bylo oba soubory z výstupu vyloučit anebo případně vzít výsledek a odečíst od něj dvojku. To ale zatím neumíme, a tak si pomůžeme poněkud nečistým trikem, který ale ctí zásadu jednoduchosti. Použijeme starý známý příkaz **`tail`**, pomocí nějž dokážeme z proudu dat, který protéká rourou, odebrat první dvě řádky.

Řešení:

```
sinek:~> ls -d * .* | tail -n +3 | wc -l
     186
```

Poznámky:

Hodnota „**`+3`**“ u přepínače „**`-n`**“ příkazu **`tail`** znamená, kterou řádkou má výpis začínat, a nikoliv počet řádek, jež má příkaz z konce souboru vypsat.

Přísně vzato, naše použití příkazu **`tail`** odstraní první dvě řádky, které ale nemusely být zrovna oněmi dvěma, jež jsme chtěli odstranit. Pro účely počítání je to ale jedno.

Pokud by ve zkoumaném adresáři naneštěstí nebyl žádný soubor, shell by jako parametr příkazu **ls** předal námi zadanou hvězdičku nezměněnou. Příkaz **ls** by se pak pokusil vypsat informace o (neexistujícím) souboru se jménem „*“ a výsledkem by bylo chybové hlášení:

```
sinek:~> mkdir empty
sinek:~> cd empty
sinek:~/empty> ls -d * .* | tail -n +3 | wc -l
ls: *: No such file or directory
       0
```

Výsledek by byl sice správný, ale ošklivá chybová zpráva by ho hyzdila. Můžeme si pomoci tím, že přesměrujeme výpis všech chybových zpráv někam mimo terminál (obrazovku). Vhodným adeptem je pseudosoubor **/dev/null**. Ten představuje jakousi „černou díru“ – vše, co se do něj zapíše, nenávratně zmizí v propadlišti dějin.

Při sestavování příkazu je ovšem třeba dát pozor na to, že každý slušný unixový program produkuje dva výstupní proudy dat – *standardní výstup* (to je kýžený seznam souborů, co nám teče do roury) a *standardní chybový výstup* (tam patří nechtěná zpráva). A tyto proudy je třeba odlišit.

Všechny datové proudy (vstupní i výstupní) jsou očíslovány a implicitně jim patří čísla 0 (standardní vstup), 1 (standardní výstup) a 2 (standardní chybový výstup). Přesnější formulace je, že program má k dispozici několik očíslovaných *deskriptorů* (jazykovým puristům se omlouvám, ale překlad „popisovač souboru“ ve mně vyvolává představu tlusté fixky, proto zůstanu u cizího slova). Přesměrování chybového výstupu (tj. deskriptoru číslo 2) provedeme operátorem „**2>**“ místo „**>**“:

```
sinek:~/empty> ls -d * .* 2> /dev/null | tail -n +3 | wc -l
       0
```

Příkaz **mkdir** (*make directory*) v předchozí poznámce, jak jste jistě pochopili, vytváří nový (prázdný) podadresář. Příkaz **cd** se do tohoto nového adresáře „přepne“, tj. změní aktuální adresář (a také cestu v promptu).

Na závěr snad už jen jedna škodolibá poznámka. Když si prostudujete manuálovou stránku příkazu **ls**, najdete „správné“ řešení:

```
sinek:~> ls -a | tail -n +3 | wc -l
```

Přepínač „**-a**“ má přesně tu funkci, kterou potřebujeme (výpis včetně skrytých souborů). V normě i na většině systémů najdeme dokonce ještě šikovnější přepínač „**-A**“, který potlačí i nechtěné „.“ a „..“:

```
sinek:~> ls -A | wc -l
```

1.13. Počítání podadresářů

Zadání:

Spočítejte v adresáři pouze podadresáře (ne obyčejné soubory).

Rozbor:

Umíme vypsat obsah adresáře a pomocí přepínače „**-l**“ i výpis mnoha dalších užitečných atributů souborů. Jedním z nich je *typ* souboru – je to první znak ve výpisu:

```
sinek:~> ls -l
total 11068
-rw-------   1 forst  forst   63227 Apr  9 12:15 1989.jpg
drwx------   2 forst  forst     512 Aug  7 03:05 Desktop
... atd.
```

Pro naše účely stačí vědět, že adresáře mají typ „d“ a obyčejné soubory „-“.

Potřebovali bychom tedy pouze umět filtrovat řádky a vybírat ty, které začínají znakem „d“. Na to máme v UNIXu dva jednoduché nástroje: *regulární výrazy*, pomocí nichž můžeme vyjádřit naše požadavky (řádka začíná písmenem „d“), a příkaz **grep**, který umí podle takových požadavků řádky filtrovat.

Řešení:

```
sinek:~> ls -l | grep ^d | wc -l
      42
```

Poznámky:

Regulární výrazy jsou velmi silným prostředkem. Poznáme je postupně, zatím nám stačí jediný znak se zvláštním významem (*metaznak*) – „**^**“ (stříška) na začátku výrazu. Ta znamená, že text, který následuje („**d**“), se musí nacházet na začátku testované řádky. Bez stříšky by se hledal výskyt podřetězce „d“ kdekoliv na řádce.

Mimochodem, filtrování nám zároveň pomohlo odstranit úvodní řádku s nápisem „`total ...`“, kterou bychom jinak museli ošetřit zvlášť.

Pozornou prohlídkou manuálové stránky příkazu **grep** najdeme přepínač, který nám zjednoduší práci – počítá řádky za nás (resp. za příkaz **wc**):

```
sinek:~> ls -l | grep -c ^d
42
```

A konečně úplně jiná cesta vede přes použití hvězdičky. Zkuste si, co udělá příkaz „**echo */**“...

1.14. Počítání do hloubky

Zadání:

Spočítejte všechny soubory v celém podstromu pod aktuálním adresářem.

Rozbor:

Pokud by zadání znělo pouze na jedno „patro“ stromu (tedy na soubory ve všech podadresářích), asi bychom to nějak zvládli se stávajícími znalostmi (kombinací příkazu **`ls`** a hvězdičky). Tady ale budeme muset znova prozkoumat manuálovou stránku příkazu **`ls`** a zaměřit se na přepínač „**`-R`**“ (rekurzivní výpis). Jeho výstup je ale poněkud složitý, a tak nás čeká trochu více práce se zpracováním:

```
sinek:~> ls -Ra
.
..
.Xauthority
... atd.

./.config:                          ... záhlaví podadresáře
.
..
... atd.
```

Předně budeme muset opět vyloučit řádky „`.`“ a „`..`“, navíc tentokrát i prázdné řádky a úvodní řádku (záhlaví) každého adresáře. Použijeme samozřejmě opět příkaz **`grep`**, který jsme poznali minule. Ovšem vymyslet regulární výraz, který by popsal, jak mají vypadat řádky, jež chceme ve výstupu ponechat, by bylo velmi obtížné, ne-li nemožné. Daleko snazší by bylo popsat, jak mají vypadat řádky, které ve výstupu nechceme. A příkaz **`grep`** opravdu takovou funkci má, stačí přidat přepínač „**`-v`**“ (*invert*). Ještě by se nám hodilo, kdybychom mohli zadat více jednodušších regulárních výrazů (ve smyslu „hledej *výrazA* nebo *výrazB*) namísto jednoho složitého. I to jde. Před každý výraz musíme ale explicitně zapsat přepínač „**`-e`**“ (*expression*) – jen tak je možné zadat více vzorů.

Teď už zbývá jen vymyslet vhodné regulární výrazy.

Čím se vyznačuje řádka záhlaví? Obsahuje lomítko a to jméno žádného souboru nemůže! V UNIXu mohou mít soubory ve jméně libovolný znak vyjma *lomítka* (používá se pro oddělování adresářů) a znaku *NUL* (první znak tabulky ASCII, viz kap. 2.12, ten se v jazyce C používá pro ukončování řetězců). Lomítko tedy pro rozlišení záhlaví perfektně vyhovuje – prvním regulárním výrazem bude „**`/`**“.

Pro vyjádření prázdné řádky budeme muset použít další metaznak regulárních výrazů, a to dolar na konci výrazu, který má obdobnou funkci jako stříška na začátku – znamená požadavek shody s koncem řádky. Požadavek na prázdnou řádku se tedy zapíše „**^$**“ (začátek, konec a nic mezi tím). Uvědomte si, že **grep** při vyhledávání řádek hledá podřetězce, neboli příkaz „**grep ''**“ by vypsal všechny řádky (všechny obsahují prázdný podřetězec).

Zbylé „nevhodné“ řádky vypadají tak, že obsahují právě jednu nebo dvě tečky. To lze snadno spojit s testem prázdné řádky: hledáme řádky, které (od začátku do konce) obsahují 0 až 2 tečky. Takový výraz opravdu lze zapsat, ale my se pro tuto chvíli kvůli jednoduchosti dopustíme malé nepřesnosti a vynecháme všechny řádky, které obsahují pouze tečky. K tomu použijeme další metaznak regulárních výrazů, a to *hvězdičku*, která znamená „opakuj libovolněkrát“ (včetně možnosti „ani jednou“).

Poslední problém představuje znak tečka. Ta je totiž v regulárních výrazech chápána jako metaznak s významem „jakýkoliv znak“. Chceme-li tedy uvést tečku jako *tečku*, musíme před ni zapsat další metaznak, zpětné lomítko (*backslash*, „****“), který ruší metavýznam bezprostředně následujícího znaky, tj. zde tečky.

Dopracovali jsme se k regulárnímu výrazu „**^\.*$**“, který však obsahuje znaky, jež zároveň slouží jako metaznaky shellu (poznali jsme už např. hvězdičku). Musíme je proto před shellem opět „skrýt“ tím, že celý regulární výraz uzavřeme mezi dvojici apostrofů (u prvního výrazu „**/**“ jsme to dělat nemuseli, ale nebyla by to chyba).

Řešení:

```
sinek:~> ls -Ra | grep -cv -e / -e '^\.*$'
11068
```

Poznámky:

Omezení na přesný počet 0 až 2 teček by bylo ještě o malinko složitější, regulární výraz by vypadal „**^\.\{0,2\}$**“ (vysvětlení pro tuto chvíli odložíme). Zjednodušením sice riskujeme, že náš program omylem nezapočítá soubor se jménem „. . .“, ale taková jména se obvykle opravdu nepoužívají. Leda by nám někdo chtěl vědomě škodit.

Případný záškodník nám může ublížit stejně snadno například tím, že vyrobí soubor, který bude mít uprostřed jména znak konce řádky. Příkaz **ls** takové jméno vypíše, vzniknou dvě řádky a příkaz **grep** je započítá jako dva soubory.

Rád bych ale upozornil na to, že samotný fakt, že soubory mohou mít ve jméně roztodivné znaky, ještě neznamená, že je tam mít mají. Neberte si příklad z jiných operačních systémů, které uživatele vedou k volbě jmen jako „Vážení soudruzi“...

Navíc je třeba zdůraznit, že malá a velká písmena jsou různé znaky, takže můžete (máte-li masochistické sklony) pojmenovat dva soubory např. „`soubor`“ a „`SOUBOR`“.

A možná ještě jednu poznámku. V UNIXu se na jméno souboru pohlíží jako na celek. Má-li jméno souboru nějakou *příponu* (např. „`txt`“ ve jméně „`soubor.txt`“), je plnohodnotnou součástí jména, stejně jako tečka před ní. Většinou záleží jen na rozhodnutí uživatele, zda při vytváření souboru zvolí jméno s příponou nebo ne. Operační systém si s vámi a s příponami vašich souborů nehraje na schovávanou, přípony vám ukazuje a na oplátku čeká, že vy je budete psát.

Narazili jsme už na dva „jazyky“ používané pro popis určité množiny řetězců. Jeden jsme si představili jako *expanzní znaky* shellu, protože jsme je poznali jako prostředek pro vyjádření skupiny souborů, jejichž jména shell expanduje do příkazové řádky. Není to ale jejich jediné použití, a tak jim raději budeme říkat *porovnávací vzory* (anglicky *Pattern Matching Notation*). Druhým jazykem jsou *regulární výrazy* (anglicky *Regular Expression*, zkráceně *regexp*), jež známe zatím jako vyhledávací aparát pro příkaz **`grep`**. Oba jazyky ještě v knize podrobněji prozkoumáme, ale už v této chvíli je třeba zdůraznit, že je nutné mezi nimi pečlivě rozlišovat, protože rozhodně nejsou záměnné! Nepříjemné je především to, že existují metaznaky, které jsou zastoupeny v obou dvou jazycích, a to s různým významem. Příkladem je třeba hvězdička – v regulárních výrazech znamená „opakuj“ (to, co jí bezprostředně předchází), zatímco v porovnávacích vzorech značí „libovolný řetězec“. Srovnejme:

požadovaný význam:	v regulárním výrazu:	v porovnávacím vzoru:
libovolný (pod)řetězec, resp. jméno souboru	**`.*`**	**`*`**
(pod)řetězec, resp. jméno začínající znakem `a`	**`^a.*`** nebo **`^a`**	**`a*`**
libovolně dlouhý řetězec znaků `a`	**`a*`**	nelze vyjádřit

Přepínač „**`-e`**“ u příkazu **`grep`** se liší od ostatních přepínačů, které jsme použili, tím že musí mít zadanou hodnotu (konkrétně regulární výraz). Takový přepínač je rovněž možno slučovat s ostatními, přepínač s hodnotou musí být přitom uveden jako poslední. A dokonce je možné připojit k slepeným přepínačům do jediného parametru i hodnotu pro poslední přepínač, ale čitelnost takového zápisu nebude valná. Posuďte sami:

```
sinek:~> ls -Ra | grep -cve/ -e'^\.*$'
```

Přepínač „**`-e`**“ má ještě jednu motivaci: pokud by hledaný výraz začínal znakem minus, příkaz **`grep`** by ho chápal jako přepínač. Hledáme-li číslo „`-1`“, musíme psát:

```
sinek:~> grep -e -1
```

Poměrně složité vymýšlení regulárních výrazů bychom si ušetřili, kdybychom místo příkazu **`ls`** použili příkaz **`find`**. Ale dočkejme času.

1.15. Rozsah řádek

Zadání:

Vypište šestou až desátou řádku ze souboru `/etc/passwd`.

Rozbor:

Obdobou příkazu **`tail`**, který vypisuje *konec* souboru nebo vstupního proudu, je příkaz **`head`**, který naopak vypisuje *začátek*. Úlohu lze snadno vyřešit jejich kombinací.

Řešení:

```
sinek:~> head -n 10 /etc/passwd | tail -n 5
operator:*:2:5:System &:/:/usr/sbin/nologin
bin:*:3:7:Binaries Commands and Source:/:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
games:*:7:13:Games pseudo-user:/usr/games:/usr/sbin/nologin
sinek:~>
```

Poznámky:

Můžete si vyzkoušet nejrozličnější variace:

```
sinek:~> head -n 10 /etc/passwd | tail -n +6
```

nebo:

```
sinek:~> tail -n +6 /etc/passwd | head -n 5
```

Povšimněte si rozdílného volání prvního a druhého příkazu v rouře. První příkaz má kromě přepínačů operand „**`/etc/passwd`**“, takže program čte soubor. Druhý příkaz má pouze přepínače a žádné jméno souboru, a proto čte svůj vstupní proud. To je velmi obvyklý způsob chování unixových příkazů zpracovávajících nějaké soubory. Už jsme to ostatně viděli například u příkazu **`wc`**.

1.16. Sloupečky

Zadání:

Vypište seznam uživatelů ve tvaru „*login_jmeno=JMENO PRIJMENI*“.

Rozbor:

V tomto příkladu poznáme hned několik utilit, které zpracovávají nějaký textový vstup a transformují ho – pro takové utility se používá také termín *filtr*.

Pro převod na velká písmena použijeme utilitu **`tr`** (*translate*). Je to obecný nástroj pro konverzi znaků z jedné znakové sady do druhé. Zdrojová a cílová sada (tabulka) se mu zadávají jako dva samostatné parametry příkazové řádky.

Nejdřív ale musíme soubor `/etc/passwd` „rozstříhat“ po sloupečkách. Na to máme nástroj, který se opravdu jmenuje **`cut`**. Potřebujeme ustřihnout první (login) a pátý (jméno a příjmení) sloupeček (strukturu souboru jsme si ukazovali v kapitole 1.3).

Poté, co pátý sloupeček konvertujeme na velká písmena, musíme sloupečky zase slepit. Na to použijeme inverzní nástroj **`paste`**.

Řešení:

```
sinek:~> cut -d: -f1 /etc/passwd > /tmp/loginy
sinek:~> cut -d: -f5 /etc/passwd | tr a-z A-Z > /tmp/jmena
sinek:~> paste -d= /tmp/loginy /tmp/jmena
root=SUPER USER
... atd.
forst=LIBOR FORST
sinek:~> rm /tmp/loginy /tmp/jmena
```

Poznámky:

Tady se opět ukazuje, jak výhodné je uložení seznamu uživatelů v textovém souboru. Jedná se vlastně o velmi primitivní relační databázi, řádky představují jednotlivé záznamy a utility jako **`cut`** a **`grep`** nám slouží jako jednoduchá náhrada příkazu `SELECT` v databázovém jazyce SQL.

Příkaz **`cut`** stříhá ze souboru sloupečky podle seznamu složeného z pořadových čísel polí. Seznam zadáváme pomocí přepínače „**`-f`**“ (*fields*), přičemž lze přepínačem „**`-d`**“ (*delimiter*) vybrat, který znak slouží jako oddělovač polí. Pokud přepínač neuvedeme, implicitně se použije tabulátor. Alternativně lze požadovat stříhání jednotlivých znaků, seznam pořadových čísel pozic znaků se zadává přepínačem „**`-c`**“ (*chars*).

Rozsah sloupců (nebo pozic) lze zapsat vcelku „logickým“ způsobem pomocí čísel nebo intervalů oddělených čárkami (např. „**`1,7-9,27-`**“ znamená 1., 7. až 9. a od 27. sloupce až do konce). Číslování sloupců, resp. pozic přitom začíná od jedničky.

Pozor na to, že jednotlivé sloupce nejde přehazovat! Ať je v seznamu zapíšete jakýmkoliv způsobem, budou vypsány vždy ve stejném pořadí, v jakém jsou obsaženy ve vstupu. Pokud byste potřebovali pořadí jiné, museli byste použít několik volání příkazů **`cut`** a **`paste`** (podobně, jako jsme to udělali my). Lepší metodu řešení poznáme v kapitole 3.4.

Příkaz **`cut`** je velmi užitečný a je důležité si ho dobře osvojit. A taky neplést s příkazem **`cat`** (už kvůli výslovnosti se to často stává).

Příkaz **`tr`** pracuje tak, že čte standardní vstup (nelze mu zadat soubor jako parametr) a kontroluje každý znak, zda se vyskytuje ve znakové sadě (tabulce) zadané prvním parametrem. Pokud se tam znak nachází, zjistí jeho pořadové číslo v tabulce, podívá se na stejnou pozici do tabulky zadané druhým parametrem a na výstup napíše odpovídající znak z druhé tabulky. Pokud znak v první tabulce není, okopíruje se na výstup.

Znakové sady lze zapsat různými způsoby. My jsme použili interval znaků, ale mohli jsme místo toho zvolit vyjádření pomocí *tříd* znaků:

```
tr '[:lower:]' '[:upper:]'
```

Zápis „**`[:lower:]`**“ značí malá písmena a „**`[:upper:]`**“ velká (ve shodném pořadí). Přehled existujících tříd najdete v příloze A.

Ale pozor, příkaz provádí konverzi znaků, nikoliv náhradu řetězců! Pokud byste měli nahradit v textu všechny výskyty slova „`Libor`“ slovem „`Forst`“ a napsali byste:

```
tr Libor Forst
```

splnili byste sice požadovaný úkol, ale splnili byste ho dokonce na více než 100 %! Všechna slova „`Libor`“ by byla opravdu nahrazena slovem „`Forst`“, ale zároveň by se všechna „`L`“ v textu zaměnila za „`F`“, „`i`“ za „`o`“ atd.

Příkaz **`paste`** má jako implicitní oddělovač polí rovněž tabulátor a lze ho předefinovat stejným přepínačem („**`-d`**“, *delimiter*) jako u **`cut`**. Oddělovačů může být i více, v tom případě je **`paste`** postupně střídá.

Před použitím příkazu **`paste`** je dobré se ujistit, že se nám nepomíchaly nebo nepoztrácely jednotlivé řádky.

Pracovní soubory opět nesmíme zapomenout po sobě smazat. Už to nebudeme v dalších kapitolách připomínat.

1.17. Ze sloupečku řádka

Zadání:

Vypište seznam jmen uživatelských skupin do jedné řádky, jména oddělte mezerami.

Seznam skupin se nachází v souboru **/etc/group**, který má opět textový formát:

```
wheel:*:0:root,forst,kernun
... atd.
```

V prvním poli je jméno skupiny, ve třetím číslo (*GID*) a ve čtvrtém seznam členů.

Rozbor:

První sloupec se jménem skupiny oddělíme pomocí příkazu **cut**.

Dále budeme potřebovat jednotlivé řádky postupně slepit do jediné. Na to lze jít různými cestami. Řešení čistě programátorské (číst postupně v cyklu řádky a vypisovat je bez odřádkování) se časem také naučíme, ale teď se pokusíme zapojit „selský rozum".

Jak vypadá výstup příkazu **cut**? Bude tam vždy jméno skupiny, pak znak konce řádky a pak jméno další skupiny atd. A jaký výstup požaduje zadání? Jméno skupiny, mezeru, jméno další skupiny... Jaký je mezi nimi rozdíl? Pouze znak konce řádky je nahrazen mezerou. A to už umíme. K tomu lze použít opět příkaz **tr**.

Řešení:

```
sinek:~> cut -d: -f1 /etc/group | tr '\n' ' '
wheel daemon kmem ... atd.
```

Poznámky:

Velmi pěkný příklad na to, že je vždycky lepší vzít nejdříve do hrsti rozum a teprve potom klávesnici...

Až se časem naučíme používat editor, pravděpodobně se vaše myšlenky budou také zaobírat možností vyřešit problém pomocí editace. To ale vůbec nepůjde jednoduše. Slepování řádek pomocí příkazu **tr** je dobré si zapamatovat.

Dvojice znaků „**\n**" (tzv. *escape-sekvence*) v zadání první sady příkazu **tr** vyjadřuje znak konce řádky. Některé programy dopřávají uživatelům podobná ulehčení, je třeba si to ale vždy ověřit v manuálu. Zde bychom to však uměli obejít v shellu sami:

```
sinek:~> cut -d: -f1 /etc/group | tr '
> ' ' '
```

Mezi první dva apostrofy jsme uzavřeli znak konce řádky (jediný znak první tabulky). Jen pro pořádek: mezera mezi apostrofy číslo 2 a 3 je metaznak oddělující parametry a mezera mezi apostrofy číslo 3 a 4 je normální mezera, (jediný) znak druhé tabulky.

Zobáček na začátku druhé řádky je tzv. *sekundární* prompt, kterým nám shell dává najevo, že jsme sice odřádkovali, ale on ještě na něco (zde párový apostrof) čeká.

1.18. A naopak

Zadání:

Vypište do sloupečku seznam jmen skupin, do nichž patříte.

Rozbor:

Nejjednodušší cesta vede přes příkaz **id**, který vypisuje všechny potřebné informace:

```
sinek:~> id
uid=1004(forst) gid=1004(forst) groups=1004(forst),0(wheel),
1(daemon),999(kernun),1028(j28)
```

O malinko náročnější bude ale zpracování tohoto „pestrého“ výstupu.

Zajímá nás pouze text za posledním (třetím) rovnítkem (tady pomůže **cut**). Tento text potřebujeme rozlámat na řádky po čárkách (to zvládne **tr**).

Tím by nám zbyly řádky ve tvaru „`1004(forst)`“. Kdyby příkaz **cut** dovoloval definovat dva různé oddělovače, stačilo by říci: vyber druhé pole mezi závorkami. Ale to bohužel nejde. Museli bychom na závěr zavolat **cut** dvakrát:

```
id | cut -d= -f4 | tr , '\n' | cut -d'(' -f2 | cut -d')' -f1
```

Bylo by tedy šikovné zbavit se přebytečné pravé závorky jinak. Pomůže nám přepínač „**-s**“ (*squeeze*) příkazu **tr**. Umí totiž sloučit posloupnost stejných znaků (druhé sady) do jediného. Do první znakové sady umístíme kromě čárky i pravou závorku, oba znaky necháme převádět na znak konce řádky a přepínač „**-s**“ zabezpečí, že ve výsledku nebudou dva znaky konce řádky za sebou (a tedy prázdná řádka mezi nimi).

První volání příkazu **cut** je ovšem poněkud „drsné“. Jeho výsledek je přespříliš závislý na případných, byť i drobných, odchylkách implementace příkazu **id**. A tento příkaz se opravdu na různých systémech chová odlišně – někde budete muset například použít přepínač „**-a**“, aby se ve výstupu vůbec objevila položka „`groups=...`“. Proto výběr přesně čtvrtého pole, který je konstantou zapsán v kódu (tzv. „hardcoded“) určitě není hodný následování. Raději tedy nejprve výstup rozlámeme do řádek a vybereme tu správnou (bude-li se tam taková vyskytovat).

Řešení:

```
sinek:~> id | tr ' ' '\n' | grep ^groups= | \
> tr -s '),' '\n\n' | cut -d'(' -f2
forst
wheel
... atd.
```

Poznámky:

Využití příkazu **id** skrývá drobná úskalí v přenositelnosti. Budeme ho proto v knize používat spíše jako zdroj textu v určitém konkrétním formátu, který budeme posléze dále zpracovávat. Jak zjistit členství ve skupinách, aniž bychom použili příkaz **id**, se naučíme v kapitole 1.20.

Povšimněte si, že jsme celý, relativně složitý program dokázali zapsat na jediné řádce shellu. To je poměrně obvyklý způsob práce – tam, kde to jde.

Přesněji řečeno: kvůli šířce stránky v knize jsme potřebovali řádky dvě. Proto jsme na konci první řádky zapsali znak zpětného lomítka („****“, backslash) těsně před znak konce řádky. Zpětné lomítko už jsme poznali v regulárních výrazech. A stejně jako tam, i v shellu tento znak *ruší zvláštní význam* metaznaků. Konkrétně zde jsme zrušili metavýznam znaku konce řádky, shell tedy neukončil příkaz, naopak vypsal sekundární prompt „`> `“ (už jsme ho viděli v minulé kapitole) a nechal nás příkaz dokončit. Ve výsledném textu příkazu nezbude po znaku konce řádky ani stopa.

Rozdělení příkazové řádky pomocí zpětného lomítka lze použít v jakémkoliv místě příkazové řádky shellu. V našem případě, tj. bezprostředně po symbolu roury, lze beztrestně odřádkovat i bez zpětného lomítka, ale pokud si nejste jisti, raději ho napište.

První řádka, kterou zpracovává příkaz **cut**, obsahuje kromě čísla skupiny ještě text „`groups=`“. To nám ovšem nevadí, protože **cut** jej odstřihne spolu s číslem skupiny. Pokud bychom toto štěstí neměli a potřebovali bychom se ho explicitně zbavit, mohli bychom sadu znaků pro rozlamování řádek ve druhém volání příkazu **tr** rozšířit ještě o rovnítko a pak celou první řádku (s textem „`groups`“) zahodit:

```
sinek:~> id | tr ' ' '\n' | grep ^groups= |
> tr -s '),=' '\n\n\n' | tail -n +2 | cut -d'(' -f2
```

Oba parametry příkazu **tr** musejí být stejně dlouhé (obě znakové sady musejí mít stejný počet prvků). Ovšem opakovaní znaků (v našem případě znaků konce řádky) ve druhém parametru je nepříjemné (a někdy dokonce i téměř nemožné). Představte si, že by v prvním parametru byl třeba interval „**a-z**“... Pro tyto případy se hodí jeden zvláštní konstrukt, který **tr** u druhého parametru povoluje, a to zadání určitého počtu opakování znaku. Náš příkaz bychom mohli psát jako

```
tr -s '),=' '[\n*3]'
```

s významem: ve druhé sadě jsou tři znaky konce řádky. Dokonce bychom mohli psát i:

```
tr -s '),=' '[\n*]'
```

s významem: druhá sada znaků končí tolika znaky konce řádky, kolik je třeba.

Kulaté závorky, které jsme používali jako součást parametrů pro příkazy **tr** nebo **cut**, jsme zavírali do apostrofů. Důvodem je to, že i ony jsou metaznaky shellu (jejich význam si vysvětlíme v kap. 4.11). Abychom nemuseli při psaní příkazů pořád myslet na to, zda některé řidčeji se vyskytující znaky jsou nebo nejsou metaznaky shellu, můžeme „podivné" parametry raději vždy do apostrofů zavírat „pro jistotu". U složitějších příkladů v části o shellu uvidíme, že to tak nejde udělat úplně vždy, ale prozatím toto pravidlo můžeme dodržovat.

U takhle složitého příkazu už může nastat situace, že výsledek neodpovídá našim představám, ale přitom nedokážeme poznat, který z pěti (!) příkazů spojených rourami (takovému spojení se říká *kolona*) to zavinil. Pak se nám může hodit příkaz **tee**. Dostává jako parametr jméno souboru a dělá jakousi rozbočku (název příkazu je opravdu odvozen od písmene „T"): svůj vstup kopíruje beze změny na výstup, ale současně ho zapisuje do souboru. Po skončení příkazu se můžeme do souboru podívat, jaká data přesně daným místem „protekla". Pokud např. napíšeme

```
sinek:~> id | tr ' ' '\n' | tee /tmp/t1 |
> grep ^groups= | tr -s '),' '[\n*]' | tee /tmp/t2 |
> cut -d'(' -f2
```

můžeme si v souborech „`/tmp/t1`" a „`/tmp/t2`" zkontrolovat, zda příkazy **tr** správně rozlámaly svůj vstup do řádek.

1.19. Stříhání a slepování jinak

Zadání:

Upravte výpis příkazu **id** tak, že položku „uid=...“ přesunete na konec.

Rozbor:

Výstup příkazu **id** jsme už zpracovávali minule. Nyní bychom potřebovali prohodit určité části řádky. Nejelegantnější řešení nabízí samozřejmě editor, my si ovšem v této chvíli musíme pomoci jinak. Ukážeme si řešení, které sice není nejefektivnější, ale zato se na něm naučíme něco nového.

Výstup rozlámeme na dvojice řádek (rovnítka a mezery nahradíme znakem konce řádky) a pak použijeme příkaz **split**, který nám text rozdělí do oddělených souborů určené velikosti (2 řádky, přepínač „**-l**“, *lines*). Soubory seřadíme správně za sebe a vyrobíme z nich opět jedinou řádku pomocí příkazu **paste** s přepínačem „**-s**“.

Řešení:

```
sinek:~> id | tr '= ' '[\n*]' | split -l 2 - /tmp/id-
sinek:~> cat /tmp/id-ab /tmp/id-ac /tmp/id-aa |
> paste -s -d'= ' -
gid=1004(forst) groups=1004(forst),0(wheel),1(daemon),
999(kernun),1028(j28) uid=1004(forst)
```

Poznámky:

Jména pro nové soubory sestavuje **split** z prefixu (my jsme si zvolili „/tmp/id-“) a sufixu složeného ze dvou písmenek (první soubor dostane jméno „/tmp/id-aa“, druhý „/tmp/id-ab“ atd.). Každý soubor bude obsahovat právě dvě řádky (název položky a hodnotu). Kdybychom prefix nezadali, **split** by vyrobil jména „xaa“ atd.

První parametr příkazu **split** obsahuje jméno souboru, který chceme rozdělit. Pokud místo toho zadáme „**-**“ (minus), znamená to, že **split** má místo souboru použít svůj standardní vstup. Podobně se chová většina unixových příkazů (použili jsme to třeba hned o kus dál u příkazu **paste**).

Použití příkazu **paste** v tomto příkladu je poněkud odlišné, než jsme viděli doposud. Přepínač „**-s**“ (*serial*) říká, že nechceme spojovat řádky několika souborů k sobě, ale naopak že chceme (sekvenčně) pospojovat všechny řádky jediného souboru do jedné dlouhé řádky. Přepínačem „**-d**“ určíme, čím se mají nahradit znaky konců řádek.

Dokonce je možné těchto oddělovačů nadefinovat více a příkaz **paste** je postupně *střídá*. Toho jsme s výhodou využili. První řádku ukončíme vždy rovnítkem a druhou mezerou. Tím náš výstup dostane tvar shodný s výstupem příkazu **id**.

1.20. Seznam skupin bez pomoci

Zadání:

Vypište seznam čísel skupin, do nichž patříte, ale bez použití příkazu **`id`**.

Rozbor:

Pokud chceme úlohu řešit bez cizí pomoci, musíme začít od definice členství ve skupině. Uživatel je členem:

- svojí *primární skupiny*, jejíž číslo má zapsáno ve čtvrtém poli svého záznamu v souboru `/etc/passwd`;
- všech skupin, u nichž je uveden v seznamu členů (jeho uživatelské jméno se vyskytuje ve čtvrtém poli řádky souboru `/etc/group`).

Nejprve musíme v obou souborech vyhledat řádky, které obsahují naše přihlašovací jméno (v mém případě „`forst`“). To můžeme udělat příkazem **`grep`**, budeme ale muset použít přepínač „**`-w`**“ (*word*, hledej pouze *celá* slova). Jinak by příkaz našel všechny uživatele a skupiny, jejichž jméno obsahuje podřetězec „`forst`“ (našel by třeba uživatele „`forstova`“). Přepínač, bohužel, není v normě, ale zato ve všech rozumných implementacích ano. K hledání slov podle normy se ještě vrátíme (např. v kap. 3.7).

Z nalezených řádek musíme příkazem **`cut`** vybrat správný sloupec. Mírnou komplikací je, že z každého souboru budeme potřebovat sloupec s jiným pořadovým číslem. V souboru `/etc/passwd` je číslo (primární) skupiny ve čtvrtém sloupci, zatímco v `/etc/group` je číslo skupiny ve sloupci třetím. Musíme tedy obě volání příkazu **`cut`** od sebe oddělit a vykonat je v samostatných příkazech.

Řešení:

```
sinek:~> grep -w forst /etc/passwd | cut -d: -f4
1004
sinek:~> grep -w forst /etc/group | cut -d: -f3
0
1
999
1028
```

Poznámky:

Řešení pomocí dvou příkazů je nepříjemné – výstup je rozdělen na dvě části.

Pokud by nám šlo jen o estetickou stránku, můžeme zadat oba příkazy na jedné příkazové řádce a oddělit je *středníkem*. Ten funguje jako oddělovač příkazů (podobně jako znak konce řádky). Shell pak pustí oba příkazy postupně, ale opticky se nám oba výstupy spojí do jednoho.

Výsledek by vypadal asi takto:

```
sinek:~> grep -w forst /etc/passwd | cut -d: -f4; \
> grep -w forst /etc/group | cut -d: -f3
1004
0
1
999
1028
```

Pokud bychom ale chtěli oba výstupy společně zpracovávat (např. je poslat do stejné roury) potřebovali bychom se dozvědět něco více o shellu. Povíme si to v kapitole 4.9.

Ovšem sjednocení výstupu je proveditelné i s našimi současnými znalostmi. Pomoci bychom si mohli opět jedním trikem. Poučnější než trik sám je metoda, jak na něj přijít: Pokuste se soustředit na to, co přesně nám vadí, a odstranit právě to! Co nám zabraňuje spojit naše dva příkazy do jednoho? Rozdílné číslo sloupce v příkazu **cut**. Tak obě čísla sjednoťme!

První možností je „ubrat“ (druhý nebo třetí) sloupec ze souboru `/etc/passwd`. Pomocí jednoho volání **cut** opíšeme všechny sloupce kromě druhého:

```
sinek:~> cut -d: -f1,3- /etc/passwd | cat - /etc/group |
> grep -w forst | cut -d: -f3
```

Příkazu **cat** jsme zadali dva parametry. Prvním z nich je „**-**“, což i pro **cat** (stejně jako u příkazů z minulé kapitoly) znamená, že má v daném místě použít svůj standardní vstup (tedy to, co mu pošle příkaz **cut**). Druhým zřetězeným souborem bude `/etc/group`.

Druhou možností je naopak „přidat“ sloupec do souboru `/etc/group`. Zavoláme příkaz **paste** a necháme spojit soubory `/dev/null` a `/etc/group`. Pseudosoubor **/dev/null** už jsme poznali jako místo, kam lze cokoliv napsat. Ale dá se použít i jako vstupní soubor a pak se chová jako *prázdný* soubor. Proto příkaz **paste** namísto jeho řádek zapisuje na výstup prázdný řetězec, za něj oddělovač a za něj vždy jednu řádku `/etc/group`. Tím nám ve výsledku jeden (prázdný) sloupec přibude:

```
sinek:~> paste -d: /dev/null /etc/group |
> cat /etc/passwd - | grep -w forst | cut -d: -f4
```

Příkaz **cat** bychom mohli vynechat a použít rovnou příkaz **grep** (ovšem jen tehdy, pokud má naše verze implementované použití parametru „**-**“, což norma nezaručuje):

```
sinek:~> paste -d: /dev/null /etc/group |
> grep -w forst /etc/passwd - | cut -d: -f5
```

V závěrečném volání příkazu **cut** jsme v tomto případě museli zvýšit požadované číslo pole o jedničku. Pokud totiž příkaz **grep** čte více souborů, nalezené řádky prefixuje jménem souboru a dvojtečkou. Tím se nám číslování sloupců o jedničku posune.

1.21. Na co nůžky nestačí

Zadání:

Vypište seznam souborů v aktuálním adresáři ve tvaru „*velikost jméno*“.

Rozbor:

Tvar „dlouhého“ výstupu příkazu **ls** už jsme viděli několikrát, velikost souboru je zapsána v pátém a jméno v devátém sloupci (viz třeba popis na str. 352). První nápad je tedy zřejmý – potřebujeme vystřihnout pátý a devátý sloupec:

```
sinek:~> ls -la | cut -d' ' -f5,9
```

Jenomže tohle nefunguje! Příkaz **cut** totiž bude brát každý jednotlivý výskyt mezery jako oddělovač pole. A nejde mu to rozmluvit! Díky různě širokým sloupcům ve výpisu vyplněným různým počtem mezer jsou pořadová čísla polí pro **cut** nepoužitelná.

Pokud nechceme opustit jednoduchá řešení, nezbude nám než příkazu **cut** řádky připravit tak, aby sloupce odděloval jen jediný oddělovač. Na to můžeme použít příkaz **tr**, který ale tentokrát nebude nic „překládat“, pouze slučovat mezery (na to stačí zadat jen jednu tabulku znaků).

Řešení:

```
sinek:~> ls -la | tr -s ' ' | cut -d' ' -f5,9
2560 .
1024 ..
114 .Xauthority
123476 .bash_history
... atd.
```

Poznámky:

Formát výstupu není nijak úchvatný. Mírného zlepšení bychom dosáhli, kdybychom příkazem **tr** nahradili posloupnosti mezer tabulátorem. U následného volání příkazu **cut** bychom pak mohli vynechat zadání oddělovače:

```
sinek:~> ls -la | tr -s ' ' '\t' | cut -f5,9
```

Pokud bychom znali přesné pozice sloupců, mohli bychom použít **cut** s přepínačem „**-c**“. To ale u výstupu příkazu **ls** dopředu neznáme (mění se podle obsahu). Pro výběr polí z takovýchto formátů řádek se příkaz **cut** zkrátka nehodí úplně dobře. Daleko lepší je použít např. filtr **awk**, na nějž si ale budete muset ještě chvíli počkat.

Podobné problémy s tím, zda dva po sobě jdoucí oddělovače vytvářejí mezi sebou prázdné pole anebo se slučují a chápou se jako jeden, budeme ještě řešit častěji. Je dobré vždy pečlivě číst manuálovou stránku příkazu, který se chystáme použít.

Další náměty na procvičení

1. Vypište seznam souborů v adresáři `/etc`, které mají příponu. Poté vypište tentýž seznam, ale tak, že jména vyfiltrujete z výstupu příkazu „**`ls /etc`**“.
2. Vytvořte dva soubory `raz` a `dva`. Napište příkaz, který vypíše jméno staršího z nich (a přitom současně nevypíše rovněž jméno mladšího).
3. V souboru `.bash_history` ve vašem domovském adresáři (pokud používáte jiný shell než **`bash`**, zjistěte si správné jméno v manuálové stránce) je uloženo několik posledních příkazů, které jste zadali (ukládají se tam při odhlášení od systému). Vypište všechny příkazy uložené v historii, v nichž jste použili nějakou rouru.
4. V kapitole 3.16 si budeme povídat o dokumentech RFC a souboru `rfc-index.txt`. Vypište z tohoto souboru řádky, kde se nalézá řetězec „`-H`“.
5. Zjistěte počet vašich jmenovců v systému.
6. Příkaz **`ps`** vypisuje informace o spuštěných programech. Vyzkoušejte si ho a vyberte si jméno nějakého běžícího programu. Zkuste pak napsat příkaz, který z výstupu **`ps`** vyfiltruje pouze řádky s vybraným programem. Spusťte si ho víckrát a možná zjistíte, že se vám ve výstupu objeví i nechtěná řádka s vaším příkazem **`grep`**. Zjistěte, proč se ta řádka objevuje, a zkuste příkaz upravit, aby se to nedělo.
7. Vypište aktuální čas. Můžete použít příkaz **`date`**, ale jen formu bez parametrů.
8. Vypište počet členů skupiny `wheel`, zapsaných v souboru `/etc/group`.
9. Vyřešte příklad z kapitoly 1.19 pouze pomocí příkazů **`cut`** a **`paste`**.
10. V kapitole 1.20 jsme (kvůli přehlednosti) neošetřili případ, že by se hledaný řetězec mohl v řádkách `/etc/passwd` a `/etc/group` nacházet i v některých jiných polích, než potřebujeme. Upravte program tak, aby jméno uživatele hledal opravdu jen tam, kde se jména uživatelů nacházejí (v prvním, resp. čtvrtém sloupci).
11. Nalistujte si ve třetí části knihy kapitoly 3.4 a 3.8 a vyřešte úlohy za pomoci prostředků, které jsme zatím probrali.
12. V příkladu z kapitoly 1.17 zkuste ještě zlepšit celkový dojem tím, že nejprve odstraníte z `/etc/group` komentářové řádky začínající znakem „`#`“.
13. Příkaz **`who`** vypisuje aktuálně přihlášené uživatele a k nim další informace. Vypište seznam přihlášených uživatelů do jedné řádky.

Část 2 Práce se soubory

V této části se podrobněji seznámíme s organizací souborového systému v operačním systému UNIX a s dalšími důležitými utilitami, které se hodí pro nejrůznější operace se soubory (vyhledávání, kopírování, třídění, porovnávání atd.).

2.1. Poštovní schránka

Zadání:

Vytvořte adresář, do nějž může kdokoliv vložit nový soubor, ale nikdo kromě vás si nemůže zjistit jména souborů od ostatních uživatelů.

Rozbor:

Celý princip ochrany souborů v systému UNIX je založen na třech základních přístupových právech. Tato práva jsou – ve shodě s myšlenkou „všechno je soubor“ – definována víceméně bez ohledu na typ souboru. Třeba adresář si pro jednoduchost můžete představit jako textový soubor, ve kterém je zapsáno, jaké soubory a podadresáře obsahuje (až na to, že to není soubor textový, ale binární, to tak doopravdy skoro i je).

Typy práv:

- Právo „**`r`**“ (*read*) znamená, že uživatel smí číst obsah souboru. U obyčejného souboru je význam jasný (soubor lze např. vypsat nebo kopírovat), u adresáře pomůže představa „obsahu“ – seznam souborů a podadresářů v daném adresáři je možné vypsat příkazem **`ls`**, ale také je možné v shellu použít třeba hvězdičku (shell totiž musí také mít právo obsah adresáře přečíst, aby dokázal hvězdičku rozvinout na seznam jmen souborů).
- Právo „**`w`**“ (*write*) znamená, že uživatel smí obsah souboru měnit. U obyčejného souboru je význam opět zřejmý – soubor je možné třeba editovat nebo přepsat operátorem přesměrování. Také u adresáře toto právo umožňuje měnit jeho obsah – v adresáři je tedy možné soubory vytvářet, přejmenovávat, ale i mazat. To vám může připadat poněkud zvláštní, ale k tomu, abychom směli smazat soubor, skutečně nemusíme mít práva k danému souboru, ale jen k tomu adresáři, kde se soubor nachází.
- Význam práva „**`x`**“ (*execute*/*search*) už není možné popsat pomocí stejné šablony. U obyčejného souboru toto právo znamená, že uživatel smí soubor spouštět – smí tedy jméno souboru zapsat jako název příkazu na příkazové řádce. U adresáře už nám představa „obsahu“ příliš nepomůže. Oficiálně se toto právo pro adresáře jmenuje „search“ (hledat), ale ani to nám jeho funkci nijak nepřiblíží. Pro začátek si prostě představte, že adresář toto právo zkrátka mít musí.

Pro každý soubor existuje jeden uživatel (resp. UID) a jedna skupina (resp. GID), kteří jsou jeho *vlastníky*. Z hlediska přístupových práv k tomuto souboru se pak uživatelé systému dělí na tři *množiny*:

- „**`u`**“ – jednoprvková množina obsahující uživatele, jenž je vlastníkem (*user*);
- „**`g`**“ – všichni členové „skupinového“ vlastníka vyjma vlastníka souboru (*group*);
- „**`o`**“ – ostatní uživatelé (*others*).

Adresář s vlastnostmi požadovanými v zadání musí mít pro všechny uživatele práva „**w**“ a „**x**“ („**wx**“) a pro vlastníka navíc „**r**“ („**rwx**“).

Nastavení práv provedeme příkazem **chmod**.

Řešení:

```
sinek:~> mkdir mailbox
sinek:~> chmod u=rwx,go=wx mailbox
```

Poznámky:

Ověření stavu práv provedeme příkazem **ls**:

```
sinek:~> ls -ld mailbox
drwx-wx-wx  2 forst  forst  512 Oct  1 03:30 mailbox
```

Za písmenkem typu souboru („d“) vidíme tři trojice znaků, které odpovídají třem právům („r“, „w“ a „x“) pro tři množiny uživatelů („u“, „g“, resp. „o“).

Příkazu **chmod** (*change file mode*) jsme řekli, které množině uživatelů dáváme jaká práva (neboli *mód* souboru), a pak jsme uvedli seznam souborů, jichž se příkaz týká.

Práva jsme zadali mnemotechnickým způsobem. Parametr jsme složili ze dvou definic oddělených čárkami. V každé jsme nejprve uvedli označení množiny uživatelů („**u**“, „**g**“, „**o**“, případně „**a**“ pro „všem“), pak operátor („**=**“ znamená nastavení práv, „**+**“ přidání práv a „**-**“ odebrání) a potom množinu práv.

Místo konkrétních práv lze také použít označení množiny uživatelů, pak to znamená kopírování práv dané množiny. Fungovalo by tedy třeba (schválně zamotané) zadání „**u=rwx,go=u-r**“ (vlastníkovi dej **rwx**, pro ostatní vezmi práva vlastníka a seber **r**).

Existuje ještě jiný způsob zadávání práv, který je šikovnější pro složitější kombinace nastavovaných práv. Práva k souboru jsou v systému souborů uložena jako číslo, které si lze představit zapsané v osmičkové (oktalové) soustavě. Každá číslice zde představuje jednu množinu uživatelů (v pořadí, v jakém jsme je probírali) a skládá se ze tří bitů, které odpovídají jednotlivým právům:

	Množina **u**	Množina **g**	Množina **o**
Právo **r**	4	4	4
Právo **w**	2	2	2
Právo **x**	1	1	1

Pro náš příklad bychom tedy mohli použít následující příkaz:

```
sinek:~> chmod 733 mailbox
```

Ochrana souborů byla v systému UNIX navržena velmi vizionářským způsobem. Když si uvědomíme, že její koncept vznikal před bezmála čtyřmi desetiletími a dodnes pokrývá většinu požadavků, které administrátor systému potřebuje, musíme smeknout.

2.2. Nový příkaz pro sklerotiky

Zadání:

Zkopírujte soubor `/bin/mv` k sobě do adresáře a přejmenujte ho na `rename`. Změňte mu přístupová práva na nejmenší možná tak, abyste ho mohli spouštět (jako příkaz).

Rozbor:

Použijeme příkaz **`cp`** (*copy*), soubor můžeme přejmenovat rovnou při kopírování.

Jediné právo, které program ke spouštění potřebuje, je „**`x`**“.

Řešení:

```
sinek:~> cp /bin/mv rename
sinek:~> chmod 100 rename
```

Poznámky:

Adresář `/bin` je jedním z adresářů, které typicky obsahují systémové utility. Když zavoláme příkaz **`mv`**, operační systém musí najít soubor s kódem programu (`/bin/mv`), načíst ho do paměti a spustit. Soubor bude obvykle přeložený (binární) program, odtud název adresáře. Podobně fungovala většina příkazů, které jsme zatím poznali. Výjimkou je třeba příkaz **`cd`**, což je *interní příkaz* (též *vestavěný*, *built-in*) – ten shell vykoná sám.

Soubor `rename` nyní bude možné používat úplně stejně – jako příkaz zapsaný na příkazové řádce. Ale je v tom háček. Zkuste napsat:

```
sinek:~> rename a b
-bash: rename: command not found
```

Někde je problém. Spočívá v tom, že pokud jako příkaz napíšeme pouze jméno programu, shell musí vědět, kde se daný soubor nachází. A shell neví, že ho má hledat v našem domovském adresáři. Nejjednodušším způsobem, jak shellu pomoci, je zavolat program nikoliv pouze pomocí jména, ale uvedením (absolutní nebo relativní) cesty:

```
sinek:~> ./rename a b
sinek:~> /home/forst/rename a b
```

V takovém případě shell soubor s programem nemusí hledat, protože my sami jsme mu cestu k souboru kompletně napsali. Připomeňme, že „adresář“ s názvem „`.`“ znamená „tento adresář“. Napsali jsme tedy vlastně „spusť soubor `rename` z aktuálního adresáře“.

Příkaz **`chmod`** jsme použili pouze pro splnění zadání. Kdyby nám stačilo, aby náš program pracoval, nemuseli bychom práva měnit, protože kopírování práva zachová.

Místo numerické formy zadání práv jsme mohli použít mnemotechnickou, třeba:

```
sinek:~> chmod u=x,go= rename
sinek:~> chmod a=,u+x rename
```

2.3. Nový příkaz pro lenivé

Zadání:

Vytvořte shellskript `ll`, který bude provádět příkaz „**`ls -ltr`**“.

Rozbor:

Příkazový soubor (*skript*) pro shell (neboli *shellskript*) je textový soubor obsahující příkazy, které má shell vykonat (tj. náš program).

Na vytvoření tak jednoduchého skriptu, jaký je požadován v zadání, nepotřebujeme žádný editor. Zvládneme to pomocí přesměrování výstupu příkazu **`echo`**.

Stejně jako v minulé kapitole bude náš program potřebovat právo „**`x`**“. Tentokrát mu ho ale budeme muset sami explicitně přidat. Shell nemá důvod mu ho při přesměrování nastavit. Proč by to také sám od sebe dělal, že? Shell se nesnaží věštit z křišťálové koule a rozhodovat se za uživatele, co vlastně chtěl uživatel udělat. Používáme UNIX, nikoliv na nějaký jiný operační systém...

Řešení:

```
sinek:~> echo 'ls -ltr' > ll
sinek:~> chmod +x ll
```

Poznámky:

Kdybychom vůbec nepoužili příkaz **`chmod`**, soubor `ll` by neměl právo „**`x`**“. Pokus o zavolání skriptu by skončil chybou:

```
sinek:~> ./ll
-bash: ./ll: Permission denied
```

Nicméně i tak bychom mohli skript spustit. Museli bychom ale použít příkaz, kterým explicitně zavoláme shell (např. „**`sh`**“, tento soubor leží ve správném adresáři a má práva v pořádku), a název skriptu mu předat jako parametr:

```
sinek:~> sh ll
```

Na rozdíl od minulé kapitoly, kdy souboru s programem stačilo právo „**`x`**“, skript bude potřebovat i právo „**`r`**“. Důvodem je, že shell musí náš skript nejprve přečíst, aby mohl příkazy vykonat. Možná namítnete, že i kód jakéhokoliv jiného programu je nutno před spuštěním načíst, ale to dělá operační systém sám, zatímco zde čte soubor „obyčejný“ program – shell.

Příkaz **`chmod`** jsme tentokrát použili v ještě stručnější variantě. Vynechali jsme určení množiny uživatelů, pro niž právo „**`x`**“ přidáváme. V drtivé většině případů to bude ekvivalentní zadání „**`a+x`**“, ale z vrozené lenosti to uživatelé nejčastěji píšou takhle, a tak si to asi budeme muset vysvětlit. A to bude trochu komplikované...

Každý běžící program v UNIXu má nastavenou tzv. *uživatelskou masku* (*umask*). Ta udává, jaká přístupová práva mají mít soubory vytvářené daným programem. Nebo spíše to, jaká práva mít nemají, protože maska říká, které bity v právech se mají implicitně zamaskovat. Obvykle se pro zadávání i zobrazování masky používá numerická forma přístupových práv. Chtělo by to asi příklad:

aktuální nastavení masky		002		022	
vytváří se		adresář	soubor	adresář	soubor
implicitní práva		777	666	777	666
odečteme masku	-	002	002	022	022
výsledná práva	=	775	664	755	644

Dva příklady hodnot masky, které jsme v tabulce uvedli, jsou opravdu typické. Nejčastěji se maskuje zápisové právo, a to pro množinu „**o**“ a někdy i pro „**g**“.

Shell používá masku jako každý jiný program a na manipulaci s ní má interní příkaz **umask**. Pokud ho zavoláte bez parametrů, vypíše aktuální nastavení masky (oproti tomu, co jsme viděli dosud, ve vypsané hodnotě uvidíme ještě jednu vedoucí nulu navíc). Pokud chcete naopak masku změnit, zadáte parametr s požadovanou hodnotou:

```
sinek:~> umask
0022
sinek:~> echo > pokus1
sinek:~> ls -l pokus1
-rw-r--r--  1 forst  forst  1 Oct  1 03:30 pokus1
sinek:~> umask 002
sinek:~> umask
0002
sinek:~> echo > pokus2
sinek:~> ls -l pokus2
-rw-rw-r--  1 forst  forst  1 Oct  1 03:30 pokus2
```

Ve druhém případě jsme z masky vyřadili bit „**w**“ pro množinu „**g**“ a jako důsledek vidíme, že souboru oproti prvnímu případu skutečně přibylo zápisové právo pro skupinu.

Nastavení masky se týká právě běžící instance shellu a spuštěné programy ji zdědí. Po novém přihlášení k systému bude mít maska zase implicitní hodnotu.

A teď se můžeme konečně vrátit k našemu původnímu tématu. Pokud u příkazu **chmod** nezadáte označení množiny uživatelů, operace se uplatní pro všechny množiny, ale na výsledek se ještě aplikuje právě platná *maska*:

```
sinek:~> umask 023
sinek:~> chmod +x pokus
sinek:~> ls -l pokus
-rwxr-xr--
```

2.4. Nový příkaz pro nerozhodné

Zadání:

Skriptu `ll` z minulé kapitoly přidejte pro jistotu ještě jedno jméno „`ltr`“.

Rozbor (1. varianta):

Souborový systém v UNIXu má jednu zvláštnost: soubory se nijak nejmenují. Každý soubor má na disku svoji datovou strukturu (*indexový uzel* neboli *i-node*), v ní jsou uloženy důležité informace (zhruba ty, které můžete zobrazit příkazem „**`ls -l`**“), ale jméno souboru tam není! Jméno souboru vznikne tak, že se v nějakém adresáři vyrobí záznam „soubor se jménem *soubor* má indexový uzel s číslem *inode*“ (jednotlivé indexové uzly jsou očíslované a jejich čísla můžete vypsat pomocí „**`ls -i`**“).

Tento princip má jednu obrovskou výhodu: je úplně jednoduché přiřadit souboru další jméno. Stačí říct „soubor se jménem `ltr` má stejné číslo i-node jako soubor `ll`“.

Vazbě mezi jménem a indexovým uzlem se říká *link* (nebo také *hardlink*) a nový link vytvoříme příkazem **`ln`**.

Řešení (1. varianta):

```
sinek:~> ln ll ltr
```

Rozbor (2. varianta):

Hardlinky mají dva nedostatky:

- Za prvé se nedají použít pro adresáře, ale jen pro obyčejné soubory. Hardlinky na adresáře si dělá souborový systém sám (jména „`.`“ a „`..`“ nejsou vlastně nic jiného než hardlinky na sebe sama a na nadřazený adresář).
- Za druhé se nedají použít pro vazbu mezi různými logickými souborovými systémy (tzv. *cross-device link*). V rámci jednoho operačního systému totiž může být (a často bývá) do hierarchické struktury adresářů zapojeno více logických systémů souborů. Pro jednoduchost si můžete představit počítač s více pevnými disky – na každém disku obvykle bude alespoň jeden souborový systém. A čísla indexových uzlů přísluší konkrétnímu logickému souborovému systému, takže není možné se pomocí nich odkázat do jiného systému.

Proto existuje ještě jeden typ linku, tzv. *symbolický link* neboli *symlink*. Jeho princip je úplně jiný. Symbolický link je zcela nový soubor, má svůj vlastní i-node, je to soubor úplně nového typu (zatím jsme poznali jen obyčejné soubory a adresáře), pouze namísto opravdových dat obsahuje jen *cestu* ke skutečnému umístění souboru.

Symlink vytvoříme příkazem **`ln`** s přepínačem „**`-s`**“.

Řešení (2. varianta):

```
sinek:~> ln -s ll ltr
```

Poznámky:

Abychom mohli podrobněji prozkoumat chování obou typů linků, vytvoříme si oba dva a pojmenujeme je „`ltr.h`“ a „`ltr.s`“. Jaká bude situace v adresáři? To zjistíme příkazem **`ls`**, ale pro dokonalejší obrázek si kromě dlouhého výpisu vyžádáme ještě zobrazení čísel indexových uzlů pro jednotlivé soubory (přepínač „**`-i`**“):

```
sinek:~> ls -li ll ltr.*
11068 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ll
11068 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ltr.h
11069 lrwxr-xr-x  1 forst  forst  2 Nov 27 02:06 ltr.s -> ll
```

Vidíme, že u hardlinku jsou uvedeny úplně stejné údaje (včetně čísla i-node) jako u originálního souboru. Není divu, jedná se o tentýž soubor. Pojmy „originál“ a „link“ dávají smysl pouze v okamžiku vytváření, poté už jsou obě jména naprosto rovnocenná.

Shodné číslo i-node ovšem ještě není důkaz, že obě jména jsou linky na stejný soubor. Jak už víme, obecně mohou být dva různé soubory prvky různých souborových systémů, a tedy v každém mohou mít (nešťastnou náhodou) stejné číslo i-node. Abychom měli jistotu, museli bychom ještě zavolat příkaz **`df`** (*display filesystem*), který ukáže, na jakém logickém souborovém systému se soubory nacházejí:

```
sinek:~> df ll ltr.h
Filesystem  1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s2d   8605726 6667254 1250014    84%    /home
/dev/ad0s2d   8605726 6667254 1250014    84%    /home
```

Naproti tomu u symlinku vidíme jiné číslo i-node, vidíme i nový typ souboru („`l`“) a konečně za jménem najdeme šipku a cíl odkazu symlinku.

Všechny zobrazené informace o symlinku se týkají jeho samotného. Pokud bychom chtěli vidět naopak atributy cílového souboru, mohli bychom použít přepínač „**`-L`**“ („*sleduj link*“). Příkaz **`ls`** pak (pro symlinky) vypíše sice jméno symlinku, ale atributy cílového souboru:

```
sinek:~> ls -liL ll ltr.*
11068 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ll
11068 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ltr.h
11068 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ltr.s
```

Zde opět vidíte příklad toho, že v UNIXu je rozdíl mezi malými a velkými písmenky důležitý, v jednom příkazu jsme dokonce použili zároveň oba přepínače „**`-l`**“ i „**`-L`**“.

Pokud chceme data souboru číst nebo editovat, je úplně jedno, které jméno (a který hardlink nebo symlink) si vybereme. Při jiných operacích už se jejich chování může lišit. Pokud mažete hardlink, ve skutečnosti pouze odeberete jeden link. Teprve pokud to byl

link poslední, data souboru se opravdu vymažou z disku (pokud se ovšem daný soubor ještě aktuálně někde nepoužívá). Proto je také v indexovém uzlu uložen *počet linků* (hardlinků), které na soubor odkazují (to je to číslo mezi sloupcem s přístupovými právy a jménem uživatele v dlouhém výpisu příkazu **ls**). Naproti tomu smazání symlinku způsobí opravdu jen to, že zmizí samotný symlink. Cílový soubor se nijak nezmění. Pokud naopak smažete z disku cílový soubor (resp. jméno, jež je napsané v symlinku), symlink existuje dál, ale pokus o práci s ním skončí chybou.

Cílový soubor symlinku může být opět symlink. Tak je možné vytvořit celý „řetěz“ odkazů. Ovšem při práci s takovým řetězem je třeba dát pozor na to, že řetěz může být zacyklený. Zkuste si napsat:

```
sinek:~> ln -s linkA linkB
sinek:~> ln -s linkB linkA
sinek:~> ls -l link*
lrwxr-xr-x  1 forst  forst  5 Nov 27 02:05 linkA -> linkB
lrwxr-xr-x  1 forst  forst  5 Nov 27 02:05 linkB -> linkA
sinek:~> cat linkA
cat: linkA: Too many levels of symbolic links
```

Pokud máte problém se sémantikou příkazu **ln**, tj. které jméno je „staré“ a které „nové“, pak jednoduchá pomůcka je: sémantika je stejná jako u příkazu **cp** (nejdříve se píše staré jméno a potom nové). Pokud mají mít obě jména stejnou poslední složku, je možné druhý parametr příkazu **ln** vynechat. Typicky se to dělá, když vyrábíme v aktuálním adresáři link na soubor do jiného adresáře a chceme, aby měly stejné jméno. Ale nic nám nebrání zadat třeba link sama na sebe (vyrobíme tím cyklus délky jedna):

```
sinek:~> ln -s soubor
```

Pokud vám symlinky připomínají „zástupce“ v systému MS Windows, pak to není náhoda. Principem zástupce je opravdu převzatý koncept symlinku.

2.5. Kopírování linků

Zadání:

Zkopírujte soubory `ll`, `ltr.h` a `ltr.s` do adresáře `bin`.

Rozbor:

Tahle úloha nevypadá složitě. Použijeme příkaz **`cp`**:

```
sinek:~> cp ll ltr.* bin
```

Když se ale podíváme na výsledek:

```
sinek:~> ls -li bin
11094 -rwxr-xr-x  1 forst  forst  8 Nov 27 02:05 ll
11087 -rwxr-xr-x  1 forst  forst  8 Nov 27 02:05 ltr.h
11027 -rwxr-xr-x  1 forst  forst  8 Nov 27 02:05 ltr.s
```

příliš nás to neuspokojí. Při kopírování totiž program **`cp`** otevře zdrojový soubor, přečte jeho obsah a uloží ho pod novým jménem. Pro všechny linky bez výjimky nám tedy vytvoří pokaždé úplně nový soubor se shodným obsahem.

Řešení:

Neexistuje.

Přesněji: neexistuje řešení pomocí kopírování. Metodou, jak se kopírování obchází, je zabalení stromu do archivu a jeho rozbalení v novém adresáři:

```
sinek:~> tar cf /tmp/linx.tar ll ltr.*
sinek:~> cd bin
sinek:~/bin> tar xvf /tmp/linx.tar
x ll
x ltr.h
x ltr.s
sinek:~/bin> ls -l
11094 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ll
11094 -rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ltr.h
11087 lrwxr-xr-x  1 forst  forst  2 Nov 27 02:05 ltr.s -> ll
```

Poznámky:

Pokud ve vašem domovském adresáři podadresář `bin` neexistuje, vyrobte si ho.

Představuje jeden ze způsobů uspořádání vlastních programů uživatele tak, aby je mohl snadno volat jako příkazy. V kapitole 2.2 jsme na to už narazili. Příkazy zadané jménem hledá shell „na obvyklých místech“ (v kap. 4.4 to uvedeme na pravou míru) a na většině systémů je takovým místem i podadresář `bin` v domovském adresáři uživatele (a tam, kde tomu tak není, lze to snadno napravit). Proto si uživatel může vlastní příkazy nakopírovat (nebo nalinkovat) do tohoto adresáře a nemusí se o podmínky jejich spouštění dál starat.

Příkazy **cp** a **mv** mají stejnou sémantiku: pokud je posledním parametrem jméno *existujícího* adresáře, všechny soubory zadané předchozími parametry se kopírují, resp. přesouvají do tohoto adresáře. Jinak příkaz musí mít právě *dva* parametry a první soubor se kopíruje, resp. přesouvá a ukládá pod novým jménem.

U příkazu **cp** bývá k mání ještě přepínač „**-r**“ (*recursive*), který způsobí, že příkaz dokáže kopírovat nejen obyčejné soubory, ale i adresáře. V normě byl tento přepínač nahrazen dokonalejší variantou „**-R**“, která korektně kopíruje i symlinky (zachová typ souboru a zkopíruje cestu v odkazu), hardlinky ovšem stejně neřeší. Problém ale spočívá v tom, že přepínač „**-r**“ nezná norma a „**-R**“ zase starší unixy.

Vyzkoušejte si rozdíl v chování **cp** v závislosti na existenci cílového adresáře:

```
sinek:~> cp -R bin novy
sinek:~> cp -R bin novy
```

Jak se budou lišit výsledky prvního a druhého volání příkazu?

Program **tar** je velmi rozšířený archivační nástroj, který používá speciální formát, do nějž dokáže uložit většinu informací obsažených v indexovém uzlu (a také vlastní data, samozřejmě). První parametr popisuje požadovanou funkci. Pokud parametr obsahuje písmeno „**c**“ (*create*), **tar** archiv vyrobí, písmeno „**x**“ (*extract*) znamená naopak rozbalení archivu, „**v**“ (*verbose*) pouze vynutí zobrazování jmen souborů, s nimiž se pracuje. Jméno archivu se zadává jako druhý parametr, do prvního parametru je ale třeba přidat písmenko „**f**“ (*file*). Namísto jména souboru lze napsat hodnotu „**-**“ s obvyklým významem (archiv se zapisuje nebo čte jako standardní výstup, resp. vstup). Další parametry pak vyjadřují jména souborů, které chceme do archivu zařadit, příp. které chceme naopak rozbalit (implicitně se rozbalí celý archiv).

Jedním z míst, kde se formát příkazu **tar** obvykle používá, je distribuce balíků volně dostupného software.

Norma program **tar** až do verze 2 aspoň zmiňovala, ale místo něj doporučovala používat program **pax**. Verze 3 už **tar** nezná. Smutný příběh o tom, jak jednoduchý program, který všichni používají, nahradit složitým, který nikdo nezná...

2.6. Přesun linků

Zadání:

Přesuňte soubory `ltr.h` a `ltr.s` do nového adresáře `linky`.

Rozbor:

I tato úloha je zdánlivě jednoduchá. Pro přesun samozřejmě použijeme příkaz **`mv`**:

```
sinek:~> mkdir linky
sinek:~> mv ltr.* linky
```

Linky se opravdu přesunuly. A hardlink bude dokonce i fungovat. Ale když se pokusíme spustit symlink, narazíme na problém:

```
sinek:~> linky/ltr.s
-bash: linky/ltr.s: No such file or directory
```

Podstata problému spočívá v tom, že cesta, kterou jsme uvedli v odkazu symlinku při jeho vytváření, je relativní, ale my jsme zde přesunuli pouze symlink a nikoliv cílový soubor. Tak se stalo, že relativní odkaz vede do úplně jiného adresáře, než potřebujeme.

```
j28:~> ls -lL linky/ltr.s
lrwxr-xr-x  1 forst  forst  2 Nov 27 02:05 linky/ltr.s -> ll
```

Řešení:

Chceme-li dodržet zadání a linky přesouvat, pak neexistuje.

Ledaže bychom přesunuli i původní skript `ll`.

Poznámky:

Když vytváříme symlink, je třeba pečlivě přemýšlet, zda a jak se může on a jeho cíl po souborovém systému v budoucnosti pohybovat.

Pokud symlink odkazuje na nějaký soubor s pevnou cestou (např. `/bin/mv`), bude asi rozumné použít jako odkaz absolutní cestu:

```
sinek:~> ln -s /bin/mv rename
```

Takový symlink bude fungovat i po přesunu symlinku do jiného místa stromu adresářů.

Pokud symlink odkazuje na nějaký soubor, který je „blízko“ cílovému souboru (je ve stejném nebo sousedním adresáři, v podadresáři apod.), bude asi rozumnější použít odkaz relativní. U tohoto symlinku lze předpokládat, že se případně bude přesouvat společně s cílovým souborem, proto funkčnost po přesunu naopak zaručí relativní cesta.

Až dosud jsme se při použití příkazu **`mv`** nestarali o to, co se ve skutečnosti děje. Pokud se nové umístění souboru nachází ve stejném souborovém systému, jedná se skutečně jen o přejmenování neboli změnu hardlinku, žádná data se nepřesouvají. Pokud je ale nové umístění v jiném souborovém systému, zdánlivě nevinný příkaz může dost dlouho trvat, protože soubor se vlastně celý kopíruje.

2.7. Jak vyrobit nic

Zadání:

Vytvořte nebo upravte soubor `nic` tak, aby byl prázdný (měl délku nula), a to bez ohledu na to, zda soubor doposud existoval nebo ne.

Rozbor:

Jednou možností je použít příkaz pro kopírování souborů a jako zdroj pro kopírování vybrat soubor, ve kterém nic není. Takový „soubor“ už známe, jmenuje se `/dev/null`:

```
sinek:~> cp /dev/null nic
```

Nejjednodušší možností je ale použít operátor přesměrování a zvolit takový příkaz, který na výstup nic nepíše. Hodí se například interní příkaz shellu jménem „dvojtečka“ („**:**“), který nedělá vůbec nic:

```
sinek:~> : > nic
```

Řešení:

Už bylo.

Poznámky:

Vraťme se ještě k jednomu z našich předchozích nápadů – přesměrovat výstup příkazu **`echo`**. Problém spočíval v nadbytečném znaku konce řádky. Napadne vás možná správná otázka, zda nejde příkazu **`echo`** odřádkování rozmluvit. Přirozeně, že ano. Bohužel však existují dvě navzájem nekompatibilní metody, a tak narazíme na problém s přenositelností programů. Varianty jsou:

```
sinek:~> echo -n > nic
sinek:~> echo '\c' > nic
```

Norma se problému elegantně vyhnula tím, že nabádá používat nový příkaz **`printf`**. Tím ovšem s problémem přenositelnosti nijak nepohnula, neboť tento příkaz na starších systémech nenajdete. Podrobněji příkaz probereme později, náš úkol by řešilo volání:

```
sinek:~> printf '' > nic
```

Na potřebu vymazat obsah souboru jsme narazili při studiu operátoru „**>>**“ v kapitole 1.5. Vysvětlili jsme si také, že to většinou nebudeme doopravdy potřebovat a že bude stačit soubor smazat. Nemusí to ale být vždy pravda.

Jeden problém může být v přístupových právech. K tomu, abychom obsah souboru vymazali, potřebujeme mít pouze zápisové právo *k souboru*. Naopak, pro jeho smazání potřebujeme mít zápisové právo *k adresáři*, kde se soubor nachází.

Druhý problém nám mohou způsobit linky. Bude-li daný soubor symlink nebo jeden z několika hardlinků, operace smazání souboru a vymazání obsahu se budou podstatně lišit. Rozmyslete si, proč.

2.8. Příkaz pro dr. Watsona

Zadání:

V kapitole 2.2 jsme kopírovali program **mv**. Vycházeli jsme z toho, že známe jeho umístění. Upravte náš program, aby fungoval bez této znalosti.

Rozbor:

Budeme muset nějak prohledat strom adresářů a soubor najít. Princip jednoduchosti velí zkusit jméno **find** a prostudovat manuálovou stránku tohoto příkazu.

První parametr (může jich být i více a nemusejí to nutně být adresáře) udává místo počátku hledání. Náš příkaz bude hledat od kořene (v celém stromu adresářů).

Další parametry vyjadřují podmínky, které musejí hledané soubory splňovat, a akce, jež se s nimi mají provádět. Názvy těchto parametrů sice začínají minusem, ale nejsou to přepínače. Formálně se dají vytvářet velmi složité logické výrazy, ale nejčastější použití je takové, že uvedeme několik podmínek (splněny musejí být všechny) a jednu akci.

Možných podmínek existuje celá řada a týkají se většiny atributů uchovávaných v indexovém uzlu. My budeme hledat jen podle jména souboru („**-name**“).

Jako akci jsme zvolili „**-exec**“ – ta s každým nalezeným souborem vykoná zadaný příkaz. Jeho text se vytvoří tak, že se vezmou všechny parametry následující za „**-exec**“ až po parametr obsahující samotný středník, a pokud mezi nimi je zvláštní parametr „**{}**“, nahradí se jménem souboru (resp. cestou). V našem případě se tedy po nalezení každého souboru vyhovujícího podmínce (což bude pouze požadovaný `/bin/mv`) spustí příkaz „**cp /bin/mv rename**“.

Řešení:

```
sinek:~> find / -name mv -exec cp {} rename ';'
```

Poznámky:

V knize [2] autor o příkazu **find** píše, že poté, co ho objevil, nedokázal pochopit, jak bez něj mohl do té doby žít. Hledat v hierarchické struktuře souborového systému jiným způsobem soubory určitých vlastností opravdu není jednoduché.

Středník je metaznak shellu (to už víme, slouží k oddělení příkazů, které má shell vykonat po sobě), a proto zde musíme zrušit jeho metavýznam, např. pomocí apostrofů.

V kapitole 1.14 jsme zmiňovali, že pro výpis seznamu souborů ve stromové struktuře je nejvhodnější příkaz **find**. Vyzkoušejte:

```
sinek:~> find .
```

Vynechané podmínky značí, že žádné nemáme, vynechaná akce znamená „vypiš cestu“.

2.9. Takhle ne, milý Watsone

Zadání:

Najděte všechny adresáře se jménem `lib` ležící ve „druhém patře“ souborové hierarchie (tedy v některém z adresářů nacházejících se v kořeni) a vypište o nich „dlouhou“ informaci pomocí příkazu **`ls`**.

Rozbor:

První sloveso zadání samo navádí na použití příkazu **`find`**, třeba:

```
sinek:~> find /* -type d -name lib -exec ls -ld {} ';'
```

Neboli: prohledej všechny podadresáře v kořeni (jejich jména dosadí do příkazu shell díky hvězdičce) a hledej adresáře se jménem `lib`.

Ale věc má několik háčků. Tím hlavním je, že není vůbec snadné omezit hloubku stromu adresářů, kterou bude **`find`** prohledávat. Namísto jednoho (druhého) patra se bude prohledávat celý strom, a tak budeme velmi dlouho čekat na něco, co vlastně vůbec nechceme. A navíc jako výsledek dostaneme i adresář `/lib` (bude v seznamu parametrů a bude vyhovovat podmínce), který rovněž nesplňuje zadání.

Náš návrh je přímo ukázkovým příkladem nesprávného použití příkazu **`find`**. Měl by se totiž používat jen tehdy, když něco opravdu hledáme, tj. neznáme přesnou cestu nebo potřebujeme testovat nějaké jiné atributy než pouze jméno.

My ale známe všechny možné cesty (dají se popsat jako „**`/*/lib`**“), takže stačí jen shellu říct, aby je doplnil do příkazové řádky, a příkaz **`ls`** požadované informace vypíše.

Řešení:

```
sinek:~> ls -ld /*/lib
```

Poznámky:

Tentokrát jsme expanzní znaky použili v poněkud složitější cestě obsahující lomítka. Jak to přesně funguje? Shell za hvězdičku dosadí libovolné jméno, ale pouze v rámci jednoho adresáře. Hvězdička zde zastupuje libovolný řetězec, který neobsahuje lomítko. Našemu výrazu „**`/*/lib`**“ tedy vyhovuje třeba jméno souboru „`/usr/lib`“ ale např. „`/usr/local/lib`“ už nikoli. Tomu by zase odpovídal třeba výraz „**`/*/*/lib`**“.

Naše řešení je v tomto případě asi nejlepší. Ovšem všeho s mírou! Třeba výraz „**`/*/*/*`**“ leckde nebude fungovat, protože reprezentuje příliš mnoho jmen souborů...

Vraťme se ale ještě k příkazu **`find`** z našeho nesprávného řešení. Použili jsme další podmínku pro hledání, a to omezení na typ souboru („**`-type`**“). Z ostatních stojí za zmínku přinejmenším „**`-user`**“ (jméno nebo UID vlastníka), „**`-size`**“ (velikost souboru, přičemž je možno testovat i podmínku „více než“ a „méně než“), „**`-mtime`**“ (datum a čas poslední modifikace) a řada dalších (viz třeba popis na str. 331).

Některé implementace příkazu **find** mají jako rozšíření akci „**-ls**“, která vykonává přesně to, co „**-exec**“ v našem příkladu.

Z rozšíření příkazu **find** je z našeho pohledu zajímavá podmínka „**-maxdepth**“ (maximální hloubka procházení), která, bohužel, v normě také není. Pokud ve vašem systému chybí, můžete zkusit namísto toho zkonstruovat složitější výraz:

```
sinek:~> find /* -path '/*/*' -prune -type d -name lib \
> -exec ls -ld {} ';'
```

Výraz „**-prune**“ se chová jako „podmínka“, která je vždy splněná a způsobí, že **find** už nesestoupí do podstromu pod daným adresářem. Podmínka „**-path**“ představuje test celé cesty k souboru a použitá hodnota vyjadřuje, že musí obsahovat (alespoň) dvě lomítka. Ovšem podmínka „**-path**“ je v normě až od verze 3, a tak i s ní byste mohli narazit. Občas se prostě riziku nevyhnete...

Pokud ve výrazu napíšete více elementárních podmínek, musejí být splněny všechny zároveň. Celá podmínka v poslední ukázce tedy zní: pokud jsou v cestě dvě lomítka, už nesestupuj do hloubky, a zároveň otestuj, zda to je adresář a jmenuje se `lib`. Jistě cítíte, že „ladění“ takových výrazů není úplně triviální.

Pro případy, kdy potřebujete ještě složitější vztahy, existují logické operátory „a zároveň“ („**-a**“, *and*), „nebo“ („**-o**“, *or*), negace („**!**“) a závorky. Přitom se uplatňuje podmíněné vyhodnocování výrazu (podobně jako je tomu v jazyce C) – jakmile je hodnota nějakého podvýrazu jasná (pravda nebo nepravda), v jeho vyhodnocování se dále nepokračuje.

Povšimněte si formátu použitého jako hodnoty parametru „**-path**“. Pokud vám to připomíná použití hvězdičky jako expanzního znaku v shellu, pak máte plnou pravdu. Autoři programu **find** se rozhodli použít ve svém programu pro „**-path**“ a „**-name**“ jazyk *porovnávacích vzorů* shellu. Na rozdíl od expanzních znaků shellu ovšem v příkazu **find** hvězdička zahrnuje i lomítko a tečku na začátku jména. Proto náš výraz znamená ve skutečnosti „alespoň“ dvě lomítka a nikoliv „právě“ dvě.

A když už jsme u teček na začátku jmen souborů – skryté soubory nepožívají žádné zvláštní ochrany, pokud takový soubor vyhovuje podmínce, vypíše se.

Při psaní složitějších výrazů se často neobejdeme bez nutnosti skrývat před shellem pomocí apostrofů či zpětných lomítek jeho metaznaky (středník, hvězdičku, závorky aj.). Museli jsme to udělat třeba u hodnoty „**'/*/*'**“.

Pro rozdělení příkazu do více řádek jsme tentokrát museli použít zpětné lomítko, protože řádku dělíme jinde než za symbolem roury.

2.10. Každému, co mu patří

Zadání:

Nastavte všem souborům a adresářům v určitém podstromě (minimální možná) práva tak, aby je mohli prohlížet uživatelé systému, kteří nejsou vlastníky (others). Použijte třeba podstrom pod vaším domovským adresářem, nemáte-li v něm nějaká citlivá data.

Rozbor:

Výkonnou část úlohy obstará bezesporu příkaz **chmod**. Problém je ale v tom, že práva, která potřebujeme nastavit, se liší pro obyčejné soubory („**r**“) a pro adresáře („**rx**“). Nestačí tedy zavolat příkaz rekurzivně a nechat nastavit všem souborům stejná práva. Potřebujeme selektivně volit jiná práva podle typu souboru.

I zde nám pomůže příkaz **find**, s tím, že podle typu souboru rozhodneme o variantě parametrů příkazu **chmod**.

Řešení:

```
sinek:~> find . '(' -type d -exec chmod o=rx {} ';' ')' -o \
> '(' -type f -exec chmod o=r {} ';' ')'
```

Poznámky:

V tomto řešení jsme zkonstruovali relativně složitý výraz pro **find**, který odporuje našemu původnímu popisu příkazu, protože obsahuje dvě akce. Jak se takový výraz vyhodnocuje? Pokud se akce vykoná, použije se její *výsledek* (úspěch, resp. neúspěch) jako logická hodnota (pravda, resp. nepravda) a pokračuje se ve vyhodnocování výrazu. My jsme zde tento fakt nevyužili, ale pomocí „**-exec**“ lze takto provádět složitější testy souborů (jen je třeba dát pozor na rychlost – každá akce spustí nový program). Například smazání souborů obsahujících text „2008“ by šlo zapsat takto:

```
find . -type f -exec grep -q 2008 {} ';' -exec rm {} ';'
```

Formulaci „výsledek akce“ si v kapitole 4.17 uvedeme na pravou míru (vyhodnocuje se tzv. návratová hodnota příkazu), ale prozatím nám bude stačit intuitivní chápání.

Přepínač „**-q**“ (*quiet*) u příkazu **grep** v předešlé ukázce způsobí, že příkaz nic nevypíše, pouze skončí úspěchem, když hledaný řetězec v souboru nalezne.

Podobného efektu, jako má náš program, bychom dokázali dosáhnout mnohem snáze a rychleji, pokud by nám nevadilo, že příslušná práva získají i některé další soubory. Příkaz **chmod** totiž umí nastavit souborům práva selektivně sám – pokud místo malého „**x**“ použijeme velké „**X**“, znamená to: nastav právo „**x**“ adresářům a spustitelným souborům (tj. těm, jež mají právo „**x**“ alespoň pro jednu další množinu). Zkuste si to:

```
sinek:~> chmod -R o=rX .
```

2.11. Kdo je kdo

Zadání:

Vypište seznam všech skriptů v podstromě pod `/etc`.

Pro rozpoznání skriptů použijte příkaz **`file`**, který vypisuje k zadanému souboru charakter jeho obsahu, např.:

```
sinek:~> file /etc/rc
/etc/rc: Bourne shell script text executable
```

Rozbor:

Základem řešení bude jistě příkaz **`find`**, který nám pomůže prohledat celý podstrom a na každý soubor zavolat příkaz **`file`**. Pak už stačí jen vybrat řádky se správným obsahem a odříznout jméno skriptu.

Řešení:

```
sinek:~> find /etc -exec file {} ';' |
> grep script | cut -d: -f1
```

Poznámky:

Pokud byste program spouštěli na větší části systému souborů, asi byste si mezitím mohli odskočit uvařit a vypít kávu. Náš program totiž bude spouštět příkaz **`file`** na každý soubor, který mu příkaz **`find`** připraví. A to bude chvíli trvat...

Malého zrychlení (na mém počítači při procházení celého systému souborů asi dvojnásobného) jsem dosáhl touto úpravou: příkaz **`file`** budu spouštět jen jednou v každém adresáři, a sice s parametrem „*jméno_adresáře***`/*`**“. Ovšem pozor, řešení

```
find /etc -type d -exec file {}/* ';' | ...
```

fungovat nebude. Příkaz **`find`** totiž spouští program **`file`** sám, přímo, bez pomoci shellu. A my už víme, že náhradu výrazu s hvězdičkou za seznam jmen souborů provádí shell, který se ale v tomto případě nikde nevolá. Musíme tedy shell zavolat explicitně a pomocí přepínače „**`-c`**“ mu předat příkaz, který od něj chceme vykonat:

```
find /etc -type d -exec sh -c 'file {}/*' ';' | ...
```

Ještě většího zrychlení bychom dosáhli, kdybychom zavolali příkaz **`file`** pouze jednou a dali mu jako parametry všechna jména souborů. Tady bychom ale mohli snadno narazit na limit počtu parametrů příkazu (řádově to bývají tisíce).

A navíc by se tady už podstatně větší měrou na celkovém čase podílelo to, že příkaz **`file`** musí každý soubor otevřít a přečíst jeho začátek. Úspora získaná snížením počtu jeho vyvolání by už tedy nebyla příliš efektivní.

2.12. Třídění

Zadání:

Vypište vzestupně setříděný seznam čísel uživatelů (UID).

Rozbor:

Použijeme příkaz s předvídatelným jménem **sort**. Pouze si v manuálové stránce najdeme vhodný přepínač pro *numerické* třídění („**-n**“).

Seznam čísel uživatelů z `/etc/passwd` vystřihneme pomocí příkazu **cut**.

Řešení:

```
sinek:~> cut -d: -f3 /etc/passwd | sort -n
0
0
1
2
... atd.
```

Poznámky:

Příkaz **sort** třídí implicitně celé řádky, a to lexikograficky (abecedně) podle tabulky ASCII. Styl třídění lze ovlivnit jednak pomocí třídicích atributů a jednak nastavením národního prostředí. Kromě numerického třídění patří mezi nejdůležitější třídicí atributy například „**b**“ (*blank*, ignoruj mezery), „**f**“ (*fold*, ignoruj rozdíl mezi malými a velkými písmeny) nebo „**r**“ (*reverse*, třiď v obráceném pořadí).

Trocha osvěty pro čisté svědomí: zkratka *ASCII* znamená *American Standard Code for Information Interchange* a označuje původní tabulku znaků používanou před tím, než se Američanům začali do počítačového světa plést národy, kterým nestačilo 26 písmenek a rozhodly se používat rozličná „nabodeníčka“. Dnes se tak označuje základní znaková sada, v níž třeba české znaky nenajdeme. Každý znak má v této sadě svoje pořadí, tzv. *ordinální hodnotu* – třeba znak NUL má hodnotu 0, znak konce řádky 10, znak „`A`“ 65. Pokud chcete vidět tabulku celou, napište příkaz „**man ascii**“.

Češtinu musíme v UNIXu hledat pod označením ISO-8859-2 (na rozdíl třeba od Windows, kde se pro ni používá – jak jinak – odlišná znaková sada Windows-1250). A to nezabředáváme do takových sad, kde jeden znak může být uložen ve více bytech...

Opravdové programátory jistě zamrzí, že je nenechám třídění programovat. Ale uvědomte si, že shell je interpretovaný jazyk (každá řádka kódu se při provádění znova přečte a znova interpretuje), který navíc prakticky cokoliv provádí za pomoci externích programů, a že tedy kód naprogramovaný v něm je řádově pomalejší, než použití specializovaného programu (je-li takový k dispozici).

2.13. Třídění s klíčem

Zadání:

Vypište seznam jmen uživatelů, setříděný vzestupně podle jejich UID.

Rozbor:

Tentokrát nemůžeme výsledný sloupeček (jméno uživatele) dopředu vystřihnout, protože bychom pak už neměli podle čeho třídit. Musíme tedy v proudu dat nechat oba (příp. všechny) sloupečky, programu **sort** říci, který sloupeček (*třídicí klíč*) má pro třídění použít, a vystřižení požadovaného sloupce nechat na závěr.

Přepínač pro zadání třídicího klíče je „**-k**“ (*key*), jako hodnota se zadávají čísla sloupců (počítáno od jedničky), kde klíč začíná a kde končí, a je možné přidat i další atributy klíče (např. naše známé numerické třídění, „**n**“).

Řešení:

```
sinek:~> sort -t: -k3,3n /etc/passwd | cut -d: -f1
root
toor
daemon
... atd.
```

Poznámky:

Pro určení třídicího klíče musíme samozřejmě také zadat správný oddělovač polí. Na to jsme si už zvykli u příkazu **cut**. Tady si ale budeme muset zvyknout na jiné písmenko přepínače („**-t**“), protože „**-d**“ má jiný význam. Taková překvapení nás ještě potkají vícekrát – staří Římané nás prostě obdařili málo písmenky a o některá je velká tlačenice. A koneckonců člověka to aspoň nutí neusínat na vavřínech a být neustále ve střehu...

Implicitním oddělovačem polí pro příkaz **sort** je posloupnost tzv. *bílých znaků* (mezer a tabulátorů). To je dobrá zpráva. Pokud si vzpomenete na problémy se zpracováním výstupu příkazu „**ls -l**“ příkazem **cut** (v kapitole 1.21), tak zde nás nic podobného nečeká. My v této chvíli ale potřebujeme jiný oddělovač.

Třídicích klíčů je možné zadat více a berou se v pořadí zápisu (tj. nejprve se vezme první klíč, při shodě se kontroluje klíč druhý atd.).

Pozor na to, že příkaz **sort** výsledek třídění implicitně jen *vypisuje*. Kdybychom chtěli nějaký soubor setřídit „na místě“ (resp. uložit pod stejným jménem), museli bychom použít přepínač „**-o**“ (*output*).

2.14. Třídění s problémy

Zadání:

Vypište řádky `/etc/passwd` setříděné podle příjmení a jména uživatele.

Předpokládejte standardní zápis „...:*Jméno Příjmení*:...“ v pátém poli a ignorujte případné odchylky formátu (více křestních jmen, mezery v jiných polích apod.).

Rozbor:

Budeme potřebovat dva třídicí klíče (příjmení a jméno). To příkaz **`sort`** zvládne. Problém je ale v tom, že mají různé oddělovače polí (mezeru a dvojtečku), což už možné není. Tady jednoduché řešení neexistuje. Jedna možná cesta ctí zásadu „soustřeď se na problém a odstraň ho“. Problémem je zde odlišný oddělovač jména a příjmení (mezera). Vzhledem k doplňující podmínce zadání (předpoklad právě jedné mezery) můžeme tuto mezeru nahradit dvojtečkou, soubor setřídit a vrátit do textu zpátky mezeru.

První dva kroky jsou snadné. Horší bude vracení mezery. Potřebujeme totiž nahradit jen jednu, a to právě pátou dvojtečku. Na to by se hodil editor, který jsme ale zatím neprobírali, a tak budeme muset soubor rozdělit a zase slepit pomocí **`cut`** a **`paste`**.

```
sinek:~> tr ' ' : < /etc/passwd |
> sort -t: -k6,6 -k5,5 > /tmp/passwd
sinek:~> cut -d: -f-5 /tmp/passwd > /tmp/passwd.1-5
sinek:~> cut -d: -f6- /tmp/passwd > /tmp/passwd.6-8
sinek:~> paste -d ' ' /tmp/passwd.*
```

Řešení:

Už bylo.

Poznámky:

Povšimněte si, že díky chytré volbě jmen pomocných souborů jsme je u závěrečného volání **`paste`** nemuseli explicitně vypisovat, ale použili jsme zápis „**`/tmp/passwd.*`**“ a shell odvedl práci za nás. Připomínám jen, že u skutečného programu by jména měla být ještě více „specifická“, aby nehrozila kolize s jiným programem či uživatelem.

Řešení, k němuž jsme dospěli, je „správné“. Plní zadaný úkol a každému je jasné, proč a jak to dělá. Následující řešení je „lepší“:

```
tr ... | sort ... | tr : '\n' | paste -s -d':::: :::\n' -
```

Ale jen v uvozovkách. Plní úkol, aniž potřebuje pomocné soubory, a šetří jedno volání programu. Cenou za to ale je, že na první pohled není jasné, co se to ve skutečnosti děje. Nemluvě o tom, že musí být nutně splněna omezující podmínka v zadání, jinak řešení nefunguje. V době, kdy jsem s programováním začínal, bych měl určitě tendenci zkusit toto „lepší“ řešení. Dnes už vím, že lepší je nepřítelem dobrého...

2.15. Třídění „jinde“

Zadání:

Vypište podmnožinu řádek `/etc/passwd` obsahující pět uživatelů s nejvyššími UID.

Ve výstupu se vypsané řádky musejí objevit přesně ve stejném pořadí, v jakém byly v původním souboru.

Rozbor:

První nápad programátora povede k úvahám nad nějakým cyklem, v němž by se probíraly jednotlivé řádky, a v případě, že řádka vyhovuje podmínce, program by ji vypsal. Odhlédněme na chvíli od toho, že jsme cyklus ještě neprobírali, a zastavme se u vyhodnocování podmínky. Poznat, že UID na aktuální řádce patří mezi pět nejvyšších, není totiž vůbec jednoduché. Nejsnazší by asi bylo soubor setřídit a pět nejvyšších UID vybrat. Jenže když už budeme mít řádky setříděné a vybrané, stačilo by je jen vypsat. Ale ke splnění úlohy potřebujeme vybrané řádky vypsat v původním pořadí. A o něj setříděním přijdeme. Kdybychom si nějak dokázali původní pořadí zapamatovat, měli bychom vyhráno. Což kdybychom si řádky nejprve očíslovali? Pak bychom je mohli na závěr setřídit podle těchto čísel (a tedy v původním pořadí) a čísla zase uříznout.

Očíslování řádek nejsnáze provedeme tak, že použijeme přepínač „**`-n`**“ příkazu **`grep`** (vypisuje čísla nalezených řádek) a necháme ho „hledat“ něco, co na každé řádce určitě je. A to je třeba její začátek („**`^`**“). Čísla jsou od řádek oddělena dvojtečkou, proto **`sort`** necháme třídit až podle čtvrtého pole a závěrečný **`cut`** bude první pole odstřihávat.

Řešení:

```
sinek:~> grep -n ^ /etc/passwd | sort -t: -k4,4n |
> tail -n5 | sort -n | cut -d: -f2-
```

Poznámky:

Trik, který jsme při třídění použili – přidat si do každé řádky nějaký pomocný text a pak ho zase vyhodit – se dá použít i v mnoha jiných situacích. Například minulou úlohu jsme mohli vyřešit podobně: přidat si na začátek řádky skutečný třídicí klíč (tedy „*Příjmení Jméno*“), podle něj řádky setřídit a pak ho zase odstranit. Ale konstrukce řídicího klíče se nám daleko lépe bude dělat editorem, a tak ještě chvíli posečkejme a vraťme se k tomuto řešení později.

2.16. Unikátní třídění

Zadání:

Vypište seznam uživatelů ve skupině s číslem nula.

Rozbor:

Zjistit seznam všech uživatelů nějaké skupiny není úplně triviální, to už jsme si vysvětlovali. Potřebujeme jednak vypsat seznam těch uživatelů, kteří ji mají v souboru `/etc/passwd` uvedenu jako svoji primární skupinu (čtvrté pole), a jednak vypsat seznam členů uvedených ve čtvrtém poli patřičné řádky souboru `/etc/group`. První metoda ale povede na seznam oddělený znaky konce řádek, zatímco druhá bude mít jako oddělovače čárky. To bude třeba sjednotit (příkazem **`tr`**).

Na závěr budeme muset vyloučit duplicity. Tady nám pomůže přepínač „**`-u`**“ příkazu **`sort`** (ze všech řádek s identickým klíčem vypisuje právě jeden).

Řešení:

```
sinek:~> grep :0: /etc/passwd | cut -d: -f1 > /tmp/grp0
sinek:~> grep :0: /etc/group | cut -d: -f4 >> /tmp/grp0
sinek:~> tr -s , '\n' < /tmp/grp0 | sort -u
forst
root
toor
```

Poznámky:

První volání příkazu **`grep`** je malinko „odfláknuté“. Řetězec „`:0:`“ by se totiž na řádce mohl vyskytovat i někde jinde, konkrétně jako UID (číslo uživatele ve třetím poli). To by se buď mohlo ošetřit složitějším regulárním výrazem, nebo nejprve ze souboru vystřihnout jen dva nezbytné sloupečky:

```
cut -d: -f1,4 /etc/passwd | grep :0 | cut -d: -f1 ...
```

Pokud by nalezená skupina v `/etc/group` neobsahovala žádné uživatele, druhá dvojice příkazů **`grep`** a **`cut`** by vypsala prázdnou řádku, a ta by se nám objevila ve výstupu. Proto jsme závěrečnému příkazu **`tr`** přidali ještě přepínač „**`-s`**“, aby případné po sobě jdoucí znaky konce řádek sloučil a tím efektivně prázdnou řádku smazal.

Eliminaci shodných klíčů nabízí také příkaz **`uniq`**. Poskytuje některé další zajímavé informace, jako například počty výskytů jednotlivých duplicit. Drobným handicapem je, že potřebuje setříděný vstup. Pokud vstup setříděný nemáme, budeme stejně muset nejprve zavolat příkaz **`sort`** a v tom případě se už objedeme bez **`uniq`** (pokud ovšem nepotřebujeme nějaké dodatečné informace, které **`uniq`** nabízí).

2.17. Najdi deset rozdílů...

Zadání:

Vypište seznam duplicitních UID v systému.

Rozbor:

Pomocí příkazu „**sort -u**“ umíme vyloučit duplicitní UID. Stačilo by tedy porovnat výstup příkazu **sort** bez přepínače „**-u**“ a s ním. Na to se dobře hodí příkaz **diff** (*differences*). Jeho úkolem je porovnat dva soubory a vypsat řádky, ve kterých se liší. Unikátně a neunikátně setříděné kopie `/etc/passwd` se budou lišit právě v řádkách s duplicitními UID. Stačí tedy lišící se řádky vypsat a patřičný sloupec vystřihnout:

```
sinek:~> sort -t: -k3,3 /etc/passwd > /tmp/pw
sinek:~> sort -t: -k3,3 -u /tmp/pw | diff - /tmp/pw |
> cut -s -d: -f3
```

Podobnou službu jako **diff** nám poskytne i příkaz **comm** (*common lines*). Vypisuje do třech sloupců řádky, které našel pouze v prvním, ve druhém, resp. v obou souborech. Pomocí přepínačů řekneme, o které sloupce máme (či spíše, o které nemáme) zájem. Příkaz ovšem vyžaduje setříděný vstup. To nám ale nevadí, stejně už **sort** voláme, jen musíme potřebný sloupec vystřihnout už na začátku, aby celou řádku tvořilo jen UID:

```
sinek:~> cut -d: -f3 /etc/passwd | sort > /tmp/pw
sinek:~> sort -u /tmp/pw | comm -13 - /tmp/pw
```

Na závěr jsme si nechali řešení, které je nejjednodušší, ale je úzce specializované. Pokud budeme pracovat pouze se sloupcem UID, můžeme použít příkaz **uniq**, o němž jsme se zmiňovali v minulé kapitole. Tomu lze na rozdíl od příkazu **sort** říct, že nás naopak zajímají pouze duplicity (přepínačem „**-d**“):

```
sinek:~> cut -d: -f3 /etc/passwd | sort | uniq -d
```

Řešení:

Nestačila by už?

Poznámky:

V prvních dvou řešeních si povšimněte poměrně elegantního volání příkazů **diff**, resp. **comm**. Oba dva potřebují pracovat jednak se souborem `/tmp/pw` a jednak s jeho upravenou verzí (vytvořenou pomocí příkazu „**sort -u**“). Ovšem namísto ukládání modifikovaného textu do jiného souboru se výstup příkazu **sort** jednoduše „pošle“ rourou přímo příkazu **diff** (resp. **comm**) a pomocí parametru „**-**“ se mu sdělí, že má jako jeden z porovnávaných souborů vzít svůj vlastní standardní vstup.

Přepínač „**-s**“ (*suppress*) příkazu **cut** způsobí, že řádky, které nemají správný tvar (dostatečný počet polí) se do výstupu nezahrnou.

2.18. Na návštěvě u databází

Zadání:

Vypište seznam uživatelů systému a ke každému jméno jeho primární skupiny.

Rozbor:

Tuto úlohu by jistě bylo možné naprogramovat pomocí cyklu, který prochází řádky `/etc/passwd` a hledá odpovídající řádky v `/etc/group`.

Daleko efektivnější pomoc nám ale přichází z poněkud neočekávané strany. Pokud budeme na oba soubory, `/etc/passwd` a `/etc/group`, pohlížet jako na jednoduché databázové tabulky, úlohu vyřeší operace *spojení tabulek* na základě rovnosti hodnot v určitém sloupci. Výsledkem operace je, že pro každou dvojici řádek z obou tabulek, jež se v hodnotě spojovací položky shodují, se na výstupu objeví jedna řádka.

Použijeme tedy příkaz **`join`** a jako spojovací pole zvolíme číslo skupiny. Tím se nám každá řádka se jménem uživatele propojí s odpovídající řádkou se jménem jeho skupiny. Současně si můžeme zvolit, že ve výstupu zůstanou pouze pole s názvem uživatele, resp. skupiny (první pole obou tabulek).

Řešení:

```
sinek:~> sort -k 4,4n -t : /etc/passwd > /tmp/pw
sinek:~> sort -k 3,3n -t : /etc/group > /tmp/gr
sinek:~> join -t : -1 4 -2 3 -o 1.1,2.1 /tmp/pw /tmp/gr
root:wheel
toor:wheel
... atd.
```

Poznámky:

Podobně jako u některých dalších příkazů musejí být vstupní soubory pro příkaz **`join`** správně setříděné. Proto jsme si nejprve vytvořili setříděné pracovní kopie.

Navíc se v souborech musejí používat stejné oddělovače, což naštěstí splňujeme.

Pokud se v souborech `/etc/passwd` a `/etc/group` nacházejí komentářové řádky, naše řešení nebude fungovat. Při vyrábění kopií se jich musíme zbavit.

Význam přepínače „**`-t`**“ je zřejmý, udává oddělovač polí (shodný v obou vstupních i ve výstupním souboru). Implicitně ho tvoří libovolně dlouhá posloupnost bílých znaků (toho lze opět s výhodou využít třeba při zpracování výstupu příkazu **`ls`**).

Přepínače „**`-1`**“ a „**`-2`**“ udávají pořadová čísla sloupců, podle jejichž hodnot se budou řádky slučovat (v prvním, resp. druhém souboru). Implicitně by se použil první sloupec. U nás to v prvním souboru bude čtvrtý sloupec a ve druhém souboru třetí.

Pomocí přepínače „**-o**“ můžeme přesně nadefinovat, jak má vypadat výstup příkazu. Implicitní podoba (nejprve spojovací pole a pak seznam zbylých polí v prvním a druhém souboru) vyhovuje málokdy. Formát hodnoty přepínače je čárkami oddělený seznam polí, které se mají ve výstupu objevit. Nula označuje pole, podle něhož se slučování provádělo, ostatní pole mají tvar „*číslo_souboru*.*číslo_pole*“. My chceme vypsat pouze jména (první pole) obou souborů.

V našem případě porovnáváme číslo skupiny (GID). Na výstupu se tedy pro každé GID objeví tolik řádek, kolik uživatelů má danou skupinu jako svoji primární.

Přesněji řečeno, pokud by existovalo více skupin se shodným GID, objeví se pro každého uživatele tolik řádek, kolik takových skupin je. Kdyby třeba v našem systému existovala ještě jedna skupina s nulovým GID, řekněme „`nula`“, vypadal by začátek výstupu takto:

```
root:wheel
root:nula
toor:wheel
toor:nula
... atd.
```

Řádky, které naopak nemají odpovídající párovou řádku v jednom ze souborů, se do výstupu nezařazují. Jejich výpis však lze vynutit pomocí přepínače „**-a**“ (*all*), přesněji „**-a1**“ pro první a „**-a2**“ pro druhý soubor:

```
sinek:~> join -a2 -t: -14 -23 -o1.1,2.1 /tmp/pw /tmp/gr
root:wheel
toor:wheel
daemon:daemon
:kmem
... atd.
```

Analogicky lze pomocí přepínačů „**-v1**“ a „**-v2**“ (*invert*), vynutit výpis pouze nespárovaných řádek:

```
sinek:~> join -v2 -t: -14 -23 -o1.1,2.1 /tmp/pw /tmp/gr
:kmem
... atd.
```

Pozor na to, že příkaz je poněkud „citlivý“ i na neúplné řádky neobsahující spojovací pole (např. už zmiňované komentářové řádky v našich vstupních souborech). Přípravě souborů pro **join** je třeba věnovat trochu péče, ale zato se nám bohatě odmění v rychlosti (tvorby i běhu programu). A je až překvapující, jak často se tato zdánlivě čistě databázová operace dá aplikovat na běžné úlohy úplně jiného charakteru.

2.19. Vstup versus parametry

Zadání:

Smažte nejmladší soubor v adresáři.

Klidně si ho nejprve vytvořte:

```
sinek:~> touch nejmladsi_soubor
```

Rozbor:

Mazání souborů už umíme dávno. Dělá se to příkazem **rm**.

Vyhledání nejmladšího souboru by asi bylo tvrdším oříškem. Budeme si proto muset opět pomoci příkazem **ls** a jeho přepínačem „**-t**“, který vypisované soubory třídí podle data a času poslední modifikace (sestupně). Pokud by nám tedy šlo pouze o vypsání požadovaného jména souboru, měli bychom hotovo.

Nás ale ještě čeká úkol, jak předat nalezené jméno ke zpracování příkazu **rm**. Pokud by vás napadlo vyzkoušet něco jako:

```
sinek:~> ls -t | head -n1 | rm
```

zjistíte, že se mazání nepovedlo. Příkaz **rm** totiž svůj *standardní vstup* zcela ignoruje. Jeho jediným úkolem je smazat soubory, které má *v parametrech*, se vstupním proudem nijak nepracuje.

Problémem tedy je, jak dostat jméno, které se objeví někde ve výstupním proudu do parametrů nějakého příkazu. A v tom nám pomůže příkaz **xargs**. Ten totiž vezme program, jehož název dostane jako parametr, spustí ho a jako parametry mu předá celý obsah svého standardního vstupu.

Řešení:

```
sinek:~> ls -t | head -n1 | xargs -t rm
+ rm nejmladsi_soubor
sinek:~>
```

Poznámky:

Je třeba velmi dobře pochopit rozdíl mezi vstupem programu a jeho parametry. Obojí slouží k předávání informací spouštěnému programu, ale obojí má velmi odlišný charakter. Případy, kdy je programu jedno, zda mu informace zadáte pomocí vstupu nebo pomocí parametrů, jsou vzácnou výjimkou. A zrovna příkaz **rm** takovou výjimkou rozhodně není.

Přepínač „**-t**“ (*trace*) příkazu **xargs** nedělá nic zásadního, pouze vypíše text každého prováděného příkazu na standardní chybový výstup. Použili jsme ho jen proto, abychom demonstrovali činnost našeho řešení.

Při práci s příkazem **xargs** můžeme narazit na dvě omezení, která člověka nemusejí hned napadnout, a to jsou délka výsledného textu příkazu a počet jeho parametrů. Naštěstí příkaz **xargs** si tato omezení hlídá sám a případně si práci rozdělí tak, aby je nepřekročil (vyvolá příkaz vícekrát). To ale zase nemusí zcela souhlasit s vaším záměrem. V takovém případě je možné pomocí dalších přepínačů příkazu předepsat, jak si rozdělení práce představujete – dávky lze stanovit počtem řádek vstupu, počtem výsledných parametrů nebo délkou řádky.

Zajímavou variantou použití příkazu **xargs** je čtení vstupu po jednotlivých řádkách a vykonávání jednoho volání programu pro každou řádku. V této variantě je možné totiž předepsat nejen jméno volaného programu, ale přesný tvar celého prováděného příkazu podobně, jako jsme to dělali u parametru „**-exec**“ příkazu **find** – v textu příkazu napíšeme zástupný symbol, na jehož místo nám **xargs** vloží text právě zpracovávané vstupní řádky. Příklad použití:

```
sinek:~> ls -t | head -n1 | xargs -I{} cp {} /tmp
```

Drobnou komplikací je, že tato varianta je v normě popsána jako rozšíření, proto můžeme narazit na problémy s přenositelností. V příkladu, který jsme uvedli, jsme si tvar zástupného symbolu („**{}**“) mohli sami zvolit, ale museli jsme ho zadat pomocí přepínače „**-I**“, na některých systémech naproti tomu najdeme pro tuto funkci přepínač „**-i**“ a jako implicitní hodnotu právě řetězec „**{}**“. Také je třeba mít na paměti to, že při této formě volání bude příkaz **xargs** pro každou řádku vstupu znovu sestavovat spouštět požadovaný příkaz, což může při velkých objemech dat velmi zdržovat.

Život programátora je zkrátka neustálý kompromis...

Další náměty na procvičení

1. V příkladech z kapitol 2.12 a 2.13 upravte programy tak, aby ignorovaly v souboru `/etc/passwd` komentářové řádky začínající znakem „`#`“.
2. Zkuste setřídit výstup příkazu **`ls`** v adresáři `/bin` podle velikosti souborů.
   Návod: Inspiraci hledejte v kapitole 2.13.
   A zkuste to opravdu naprogramovat, funkci přepínače „**`-S`**“ příkazu **`ls`** si prostudujte až potom...
3. V samém závěru kapitoly 3.4 se vrátíme k úloze z kapitoly 2.14 a budeme ji řešit pomocí externího třídicího klíče přidaného jako první sloupec souboru. To byste ale měli zvládnout už nyní bez editoru.
4. Vyřešte příklad z kapitoly 3.22 za pomoci dosud známých prostředků.
   Návod: I zde pomůže externí třídicí klíč.
5. V adresáři `/etc` najděte všechny soubory s příponou „`conf`“ a zjistěte, který z nich má nejvíce řádek.
6. Vyberte si nějaký nepříliš rozsáhlý podstrom souborového systému (třeba pod vaším domovským adresářem) a najděte v něm největší soubor.
   Pokud vás napadne více způsobů řešení, porovnejte jejich rychlost.
7. Příkaz **`who`** jsme poznali v dodatečných příkladech k první části. Vypisuje seznam přihlášených uživatelů. Pokud je uživatel přihlášen vícekrát, bude ve výstupu také vícekrát vypsán. Vypište uživatele, který má aktuálně nejvyšší počet přihlášení.
   Návod: Klíčovou roli by měl sehrát příkaz **`uniq`**.
8. Vytvořte si dva adresáře `raz` a `dva` a zkopírujte do nich stejnou skupinu souborů. Pak v obou některé (různé) soubory vymažte. Napište skript, který do obou adresářů doplní (zkopíruje) z opačného adresáře smazané soubory.
9. Domovské adresáře uživatelů bývají soustředěné do jednoho nadřazeného adresáře (často to bývá adresář `/home`). Vypište seznam souborů (podadresářů) v tomto adresáři a ke každému adresáři napište UID (nikoliv jméno) jeho vlastníka.
   Návod: Zamyslete se, jak s výhodou využít příkaz **`join`**.
   Řešení s přepínačem „**`-n`**“ příkazu **`ls`** opravdu nemám na mysli.
10. Vyřešte příklad z kapitoly 4.2 za pomoci příkazu **`xargs`**.

Část 3 Editory

Tato část se podrobně věnuje dvěma významnějším a složitějším utilitám, které slouží pro modifikaci obsahu souborů či datových proudů. Jedná se o editory **`ed`** a **`sed`**, které vycházejí ze stejného základu a mnohé vlastnosti mají společné. Díky značné závislosti obou editorů na regulárních výrazech si také v celé části důkladně procvičíme použití tohoto důležitého nástroje.

3.1. Déjà vu

Zadání:

Vypište řádky `/etc/passwd` setříděné podle příjmení a jména uživatele.

Předpokládejte standardní zápis „...:*Jméno Příjmení*:...“ v pátém poli a ignorujte případné odchylky formátu (více křestních jmen, mezery v jiných polích apod.).

Rozbor:

Úlohu už jsme jednou řešili (v kapitole 2.14) a pro třídění jsme si mezeru mezi jménem a příjmením nahradili dvojtečkou. Na závěr jsme zase museli tuto dvojtečku (přesně pátou na řádce) nahradit zpátky mezerou.

Nyní na to zkusíme jít jinak a záměnu mezer a dvojteček udělat pomocí editoru **`sed`** a jeho příkazu *substitute* („**`s`**“). Ten umožňuje v textu nahradit řetězce znaků.

Řešení:

```
sinek:~> sed 's/ /:/' /etc/passwd | sort -t: -k6,6 -k5,5 |
> sed 's/:/ /5'
```

Poznámky:

Pojem „*editor*“ je obecný termín pro aplikaci, sloužící pro úpravu (editaci) textových souborů. Většina čtenářů si možná pod tímto pojmem představí nějaký okénkový program, pomocí něhož může vytvářet různé dokumenty. I takové editory v UNIXu pochopitelně existují. My se ale teď budeme zabývat editory, které můžeme využít při programování.

Prvním z nich bude tzv. *stream editor* **`sed`**. Jeho název je opravdu odvozen od slova „proud“. Bere jednotlivé řádky vstupního proudu, provádí na nich uživatelem definované úpravy a výsledek vypisuje na svůj standardní výstup. Nejedná se tedy o žádnou editaci ve smyslu „otevři soubor, proveď opravy a soubor opět ulož“. Výsledek se vždy jen vypisuje!

Alternativně můžeme (vcelku bez překvapení) programu dát jako parametr jméno souboru (případně i více). Potom program čte místo standardního vstupu zadané soubory. Ale i tak výsledek opět jen vypisuje. Pokud potřebujeme opravdu modifikovat soubor a výsledek zase uložit zpět, určitě nám nepomůže konstrukt:

```
sinek:~> sed ... soubor > soubor
```

To už jsme si vysvětlovali (shell soubor vymaže). Na takovou práci poznáme lepší prostředek později. Jediné možné řešení s využitím editoru **`sed`** je založeno na použití pomocného souboru:

```
sinek:~> sed ... soubor > soubor.new; mv soubor.new soubor
```

Některá rozšíření programu **sed** mají přepínač, kterým výsledek editace uložíme zpět pod stejným jménem. Norma ho ale neuvádí, takže se zbytečně vystavujeme nebezpečí omezené přenositelnosti v situaci, kdy existují jiná řešení.

Editor pracuje na základě vlastních příkazů, s nimiž se postupně seznámíme.

Příkaz, který má editor **sed** vykonat, zadáváme jako jeho první parametr na příkazové řádce. Oba příkazy použité v jednotlivých voláních editoru v našem prvním příkladu jsou velmi triviální, ale i tak jsme si museli pomoci apostrofy, aby shell nepovažoval mezeru v příkazu za svůj metaznak a parametr nám nerozdělil na dva. Je lepší si na tuto praxi zvyknout a příkazy editoru do apostrofů uzavírat, nebrání-li tomu nějaké důvody.

Prvním příkazem editoru, který jsme poznali, je *substitute*. V editoru **sed** se z názvu příkazu píše jen první znak, v tomto případě tedy „**s**“. Význam našeho použití příkazu je poměrně intuitivní. V prvním případě je jeho úkolem nahradit (první) mezeru na každé řádce dvojtečkou. Ve druhém naopak pátou dvojtečku mezerou. Hledaný vzor (*pattern*) a jeho náhrada (*replacement*) se zapisují jako dva parametry příkazu a jsou uzavřeny mezi trojici oddělovačů. První dva oddělovače vymezují vzor, mezi druhý a třetí se píše nový text.

Oddělovačem je první znak zapsaný za názvem příkazu. Tradičně se jako oddělovač používá lomítko, ale je možné zvolit jakýkoliv jiný znak (např. pokud se lomítko vyskytuje ve vzoru nebo v náhradě). Toho ještě někdy využijeme.

Hledaný řetězec se zadává jako regulární výraz. O těch už jsme se také zmiňovali a v následujících kapitolách se jim budeme ještě dále věnovat.

Pokud je řetězec náhrady prázdný, část řádky odpovídající nalezenému vzoru se vymaže.

3.2. Čísla řádek

Zadání:

Vypište (jedním příkazem) pátou řádku souboru `/etc/passwd`.

Rozbor:

Podobnou úlohu už jsme také řešili, ale potřebovali jsme na ni spolupráci dvou utilit, **`head`** a **`tail`**. S editorem to opravdu dokážeme voláním jediného programu.

První možností je použít příkaz *delete* („**`d`**“) a napsat mu, které řádky má vymazat:

```
sinek:~> sed '1,4d;6,$d' /etc/passwd
```

Druhou možností je použít opět příkaz *delete* pro mazání prvních čtyřech řádek a po páté řádce program prostě ukončit příkazem *quit* („**`q`**“):

```
sinek:~> sed '1,4d;5q' /etc/passwd
```

Ve třetí variantě na to půjdeme opačně. Nejprve editoru řekneme (přepínačem „**`-n`**“), že nechceme, aby každou řádku na konci vykonávání svých příkazů vypsal. A pak už jen použijeme příkaz *print* („**`p`**“) a pátou řádku vypíšeme explicitně:

```
sinek:~> sed -n 5p /etc/passwd
```

Řešení:

Pro tuto chvíli stačí.

Poznámky:

První novinkou je zadání více příkazů editoru najednou. Podobně jako u shellu slouží k jejich oddělení středník. Už minule jsme se naučili, že příkaz(y) pro editor se vyplatí uzavírat mezi apostrofy. Díky tomu shell středník nebude sám zpracovávat, celý řetězec i se středníkem bude považovat za první parametr a beze změny ho předá editoru.

Máme-li příkazy mezi apostrofy, můžeme mezi nimi také prostě jen odřádkovat.

V poslední variantě řešení jsme už apostrofy nepotřebovali. V řetězci „**`5p`**“ žádný metaznak shellu není, a tak nebudeme mít problém s dezinterpretací.

Druhou novinkou je použití tzv. *adresní části* (*adresy*) příkazu. Minule před názvem příkazu („**`s`**“) nebylo nic napsáno, editor tedy prováděl příkaz na každé řádce vstupu.

Nyní máme před názvy příkazů („**`d`**“, „**`q`**“ nebo „**`p`**“) buď jedno, nebo dokonce dvě *čísla* řádek. Takové příkazy bude editor provádět jen a pouze na těch řádkách, jejichž pořadové číslo spadá do vymezeného intervalu (příp. jednoprvkového).

Zvláštní formou zápisu „čísla“ řádky je *dolar*, který znamená poslední řádku souboru. Příkaz „**`6,$d`**“ znamená „od šesté řádky až do konce smaž všechny řádky“.

Přepínačem „**`-n`**“ vynutíme, aby editor vypisoval pouze ty řádky, o něž explicitně požádáme (např. právě příkazem *print*), ostatní řádky se ve výstupu neobjeví.

3.3. Složitější výběr

Zadání:

Vypište seznam uživatelů s UID mezi 1000 a 1199.

Rozbor:

Naštěstí s námi měl zadavatel soucit a vybral velmi vhodnou množinu UID, která se dají dobře popsat jako řetězce znaků.

Požadované UID musí začínat jedničkou, na druhém místě musí mít nulu nebo jedničku a potom dvě číslice. Např. rozsah 900 až 1199 by se popisoval mnohem hůře.

Takhle nám budou stačit *regulární výrazy* a příkaz **grep**. S regulárními výrazy už jsme zlehka seznámili v kapitole 1.14, zde budeme ale potřebovat nové znalosti.

Prvním novým metaznakem bude levá hranatá závorka („**[**“), která spolu s párovou pravou závorkou vymezuje *seznam znaků*, které v daném místě zkoumaného řetězce mohou být. Přesněji řečeno, smí tam být právě jeden znak z uvedeného seznamu.

Začneme tedy následujícím návrhem:

```
sinek:~> grep '1[01][0-9][0-9]' /etc/passwd
```

Ten opravdu přesně vyjadřuje požadavek na tvar UID: nejprve jednička, pak 0 nebo 1 a pak dvě libovolné číslice (neboli znaky mezi „0“ a „9“ včetně).

Jenže to nestačí. Víme už, že **grep** hledá na řádce jakýkoliv podřetězec shodující se se zadaným regulárním výrazem. Pro takto zapsaný vzor by nám tedy našel každou řádku, která by obsahovala jakékoliv číslo, pokud by tvarem souhlasilo (například číslo 11068). Naštěstí se ovšem bezprostředně okolo hodnoty UID v souboru `/etc/passwd` nacházejí dvojtečky, a tak první zpřesnění bude jednoduché:

```
sinek:~> grep ':1[01][0-9][0-9]:' /etc/passwd
```

Nyní už máme jistotu, že se námi popsaná čísla budou v řádce opravdu vyskytovat jako celistvá hodnota sloupce a nikoliv jako podřetězec.

Bohužel se ale v `/etc/passwd` nacházejí dvě číselná pole, takže pro přesné splnění úlohy budeme muset ještě zaručit, že shoda bude nalezena v prvním z těchto polí. To už bude o něco složitější, budeme muset popsat celý začátek řádky. Zkusíme tedy výraz:

```
sinek:~> grep '^.*:.*:1[01][0-9][0-9]:' /etc/passwd
```

Tenhle výraz už vypadá docela dobře a často bude v souboru `/etc/passwd` i správně fungovat. Když mu ale zkusíte předložit „šikovně“ vyrobenou řádku, selže:

```
sinek:~> echo x::99:1000: | grep '^.*:.*:1[01][0-9][0-9]:'
x::99:1000:
```

Vysvětlení tentokrát už není tak jednoduché.

Podívejme se, proč **grep** na řádce neočekávaně nějakou shodu našel:

podvýraz regulárního výrazu	^.*	:	.*	:	1[01][0-9][0-9]	:
část řádky s nalezenou shodou	x:	:	99	:	1000	:

Problém spočívá v „nenasytnosti“ hvězdičky. Dokáže do sebe totiž absorbovat vše, co jí stojí v cestě, jen aby našla kýženou shodu. Ačkoliv člověk přirozeně náš regulární výraz bude číst jako „hledej od začátku řádky cokoliv až do (první) dvojtečky, pak cokoliv až do (druhé) dvojtečky a pak požadované číslo“, program to bude chápat jinak – bude větu číst bez upřesňujících závorek. Musíme ještě bádat dál.

Mimochodem, je to sice smůla, ale buďme rádi, že to tak je. S programy, které se samy rozhodují za uživatele, co měl vlastně na mysli, mám jen ty nejhorší zkušenosti.

Věta „vše až do první dvojtečky“ se dá přeformulovat jako „řetězec, kde není dvojtečka“ nebo také „opakuj libovolný znak vyjma dvojtečky“. Potřebovali bychom vyjádřit, že chceme *doplněk* seznamu znaků v hranaté závorce. A to jde. Stačí zapsat na začátek seznamu stříšku („**^**“). Zápis „**[^:]**“ říká: „libovolný znak mimo dvojtečky“, a přidáme-li opakování („**[^:]***“), znamená to opravdu „vše až do první dvojtečky“.

Řešení:

```
sinek:~> grep '^[^:]*:[^:]*:1[01][0-9][0-9]:' /etc/passwd
```

Poznámky:

Nepleťte si použití stříšky („**^**“) v hranatých závorkách s jejím použitím na začátku regulárního výrazu, kde značí požadavek shody se začátkem řádky. Mezi hranatými závorkami běžné metaznaky ztrácejí svůj význam, třeba tečka je zde obyčejná tečka.

Zápis seznamu číslic jsme si usnadnili použitím *intervalu*. To je velmi užitečná pomůcka, ale je třeba s ní zacházet opatrně. Třeba výraz „**[a-Z]**“ je zcela nesprávný: mezi malými a velkými písmeny je v tabulce ASCII ještě šest jiných znaků a hlavně se velká písmena nacházejí před malými, takže celý interval je vlastně neplatný.

Potíže může působit také lokální nastavení. To respektuje i české znaky, a tak třeba interval „**[a-z]**“ zahrnuje všechny znaky od „a“ do „z“ (i ty s diakritikou), ale nikoliv „ž“. Zde by pomohly třídy znaků, o nichž jsme si říkali v kap. 1.16. Náš zápis „**[0-9]**“ neskrývá žádné úskalí, ale mohli bychom místo toho psát „**[[:digit:]]**“.

Pro zápis stříšky, pomlčky a pravé hranaté závorky do seznamu znaků platí přísná pravidla: závorka musí být první, stříška první být nesmí a pomlčka musí být v takovém místě, kde nemůže znamenat interval. Seznam z těchto tří znaků vypadá takto: „**[]^-]**“.

Pozor na to, že hvězdička znamená i „žádný“ výskyt znaku! Pokud budete chtít vypsat řádky obsahující číslo a použijete výraz „**[0-9]***“, výsledek bude obsahovat úplně všechny řádky – v každé je totiž skryt řetězec číslic dlouhý (alespoň) nula znaků.

3.4. Přehazování sloupců

Zadání:

Vypište seznam uživatelů systému ve tvaru „*Příjmení Jméno*“.

Rozbor:

Potřebujeme vystřihnout sloupec se jménem a příjmením a ještě jméno s příjmením prohodit. Už jsme zmínili, že **`cut`** nedokáže měnit pořadí sloupců, bez editoru by nás tedy čekalo zase hraní s pomocnými soubory a voláním příkazů **`cut`** a **`paste`**.

Pomocí editoru to dokážeme mnohem snáze. Použijeme opět příkaz *substitute*, ale v něm se budeme potřebovat odvolávat na části zpracovávané řádky (jednotlivé sloupce), abychom dokázali říct, kde se mají objevit ve výsledném textu. To uděláme tak, že části vzoru uzavřeme do závorek a v řetězci náhrady použijeme odkazy na jednotlivé závorky (tzv. *zpětné reference*). Do patřičných míst výstupní řádky poté editor vloží takovou část vstupní řádky, jež vyhovovala uzávorkované části regulárního výrazu (vzoru).

Řešení:

```
sinek:~> sed 's/.*:\(.*\) \([^:]*\):.*/\2 \1/' /etc/passwd
```

Poznámky:

Metaznaky regulárních výrazů, které jsme poznali před touto kapitolou, tvoří jakýsi „základ“ regulárních výrazů a existují v jakémkoliv jejich dialektu. Počínaje touto kapitolou už to neplatí. Nové metaznaky v této kapitole jsou specifické pro editory.

Odlišíte je snadno – mají jakoby obrácenou logiku. Jejich zvláštní význam se uplatní jen tehdy, předchází-li jim zpětné lomítko, jinak představují obyčejné znaky. Příkladem takového metaznaku jsou třeba právě závorky nebo číslice.

Pokud vás zajímá, jak se přesně „napasuje“ regulární výraz na vstupní řádku a které její části tedy pak odpovídají které závorce, snad pomůže následující schéma:

```
...:Jan X. Nenasytny:...      ⇒     Nenasytny Jan X.
 .*:\(.*\) \([^:]*\):.*       ⇒           \2 \1
```

O nenasytnosti hvězdičky jsme se bavili v minulé kapitole.

Teď se už můžeme vrátit k úloze s tříděním uživatelů podle příjmení a jména (2.14). Umíme totiž vytvořit nový sloupec obsahující třídicí klíč a přidat ho na začátek řádky:

```
sed 's/\(.*:\(.*\) \([^:]*\):.*\)/\3 \2:\1/' /etc/passwd
```

Upravený soubor pak už jen setřídíme a sloupec odstřihneme (to už známe).

Závorky se mohou do sebe vnořovat, pro pořadí je podstatná poloha levé závorky.

Uzávorkování celého regulárního výrazu (celé řádky) se můžeme vyhnout:

```
sed 's/.*:\(.*\) \([^:]*\):.*/\2 \1:&/' /etc/passwd
```

Znak „**&**“ v řetězci náhrady totiž znamená „sem vlož celou část vstupní řádky, která vyhovovala zadanému regulárnímu výrazu“. V našem případě to bude celá řádka.

Znak *ampersand* („**&**“) je velmi záludný. V řetězci náhrady se totiž prakticky jiné metaznaky nevyskytují, a tak člověk často ztratí pozornost, a když potřebuje do řetězce náhrady ampersand zapsat, zapomene před něj přidat zpětné lomítko.

Závorky se v regulárních výrazech také používají pro uzavření nějakého podvýrazu. Například hvězdička znamená opakování pouze bezprostředně předcházející části regulárního výrazu. Výrazu „**ba***“ tedy vyhovuje řetězec „`baaa`“, ale nikoli už „`baba`“. Potřebný vzor pro opakování podřetězce „`ba`“ bychom museli zapsat „**\(ba\)***“.

Pozoroval jsem u svých studentů, že si zpětné reference velmi oblíbili. Dokonce až moc. Používají je často i v situacích, kdy je jejich užití kontraproduktivní. Typickým příkladem je následující úloha: upravte výstup programu **hostname** (vypisuje jméno počítače včetně internetové domény, např. „`sinek.yq.cz`“) tak, aby výsledkem bylo jen jméno počítače bez domény. Značná část jejich řešení by jistě vypadala zhruba takto:

```
sinek:~> hostname | sed 's/\([^.]*\)\..*/\1/'
```

Ne snad, že by to nefungovalo. Ale porovnejte jeho složitost s jiným:

```
sinek:~> hostname | sed 's/\..*//'
```

Když chci něco smazat, nemusím přece nutit chudinku editor pamatovat si naopak to, co má v řádce zachovat! Není vám ho líto? A pokud ne, poznáte to na rychlosti. Na velkých datech může rozdíl činit i několik řádů. Nemluvě o vyšším riziku překlepu.

Nepleťte si operaci vymazání (části) obsahu řádky příkazem *substitute* s operací smazání řádky příkazem *delete*. Čím se liší příkazy „**3d**“ a „**3s/.*//**“?

A ještě poznámka k používání podvýrazu „**.***“ na okrajích regulárního výrazu. Pokud nějaký text pouze hledáme, jsou takové přívěsky zbytečné. Algoritmus hledání shody regulárního výrazu a řádky vstupu totiž vyhledává podřetězce, pokud mu to sami explicitně nezakážeme (pomocí „**^**“ na začátku nebo „**$**“ na konci výrazu). Porovnejte:

```
sinek:~> grep :0: /etc/passwd
sinek:~> grep '.*:0:.*' /etc/passwd
```

V našem případě ale použití „**.***“ na krajích regulárního výrazu nutné bylo. My jsme editoru museli říct, co všechno má ve vstupní řádce nahradit, náš vzor tedy musel nutně pokrývat celou řádku. Dokonce jsme mohli pro jistotu explicitně napsat:

```
s/^.*:\(.*\) \([^:]*\):.*$/\2 \1/
```

3.5. Hledání duplicit

Zadání:

Vypište ty řádky `/etc/passwd`, na nichž se vyskytuje nejvýše trojmístné UID shodné s GID.

Rozbor:

Budeme potřebovat dvě (staro)nové vlastnosti regulárních výrazů.

Jednak to bude omezení počtu opakování – už známe hvězdičku, která znamená libovolný počet výskytů, ale my potřebujeme pouze jeden, dva nebo tři výskyty (číslice). K vyjádření přesného počtu opakování slouží složené závorky.

A jednak bychom se potřebovali v regulárním výrazu odkázat na „něco stejného“. To nám připomíná zpětné reference. Ano, dají se použít nejen v náhradě, ale i v rámci hledaného regulárního výrazu.

Řešení:

```
sinek:~> grep ':\([0-9]\{1,3\}\):\1:' /etc/passwd
root:*:0:0:Super User:/root:/bin/bash
... atd.
kernun:*:999:999:Kernun User:/usr/local/kernun:/nologin
```

Poznámky:

Do složených závorek se zapisuje dolní a horní hranice počtu povolených opakování. Maximum pro horní hranici je 255.

Horní hranici lze vynechat (a psát např. „...**`\{1,\}`**“), pak počet není omezen (resp. je omezen maximem 255).

Pokud je v závorkách uvedeno jen jedno číslo bez čárky, pak je požadován přesný počet opakování.

Zpětná reference v rámci regulárního výrazu se dá přeložit jako „tady musí být totéž, co bylo nalezeno jako shoda v první závorce“.

Porovnejte výraz „**`A\(.\)\1A`**“ a „**`A..A`**“ – první výraz odpovídá slovům jako „`ABBA`“ nebo „`ANNA`“, zatímco druhému vyhovují i slova „`ALFA`“, „`AKTA`“ apod.

Pokud by vás zarazilo, že hovoříme o regulárních výrazech používaných v editorech, ale přitom v řešení máme program **`grep`**, pak svoji zvědavost ještě ukojte, vysvětlení přijde v kapitole 3.22 – **`grep`** je ve skutečnosti vlastně editor.

Abychom dostáli exaktnosti, zmiňme, že „editorový“ dialekt regulárních výrazů se nazývá „*základní regulární výrazy*“.

3.6. Hledání čísel

Zadání:

Vypište ze standardního vstupu všechny podřetězce představující korektní IP adresu.

IP adresa (adresa počítačů v síti TCP/IP) je tvořena čtyřmi čísly (od 0 do 255) oddělenými tečkami.

Rozbor:

Tady už s námi (na rozdíl od kapitoly 3.3) zadavatel soucit neměl. Editoru se nedá říci něco jako „číslo je menší než 256“. Budeme to muset řešit složitěji.

V první řadě bychom se měli zbavit všeho, co by nám komplikovalo život, tj. všech znaků mimo číslice a tečky. Navíc by se hodilo, kdyby se nám jednotlivé posloupnosti „adeptů“ na IP adresy nějak oddělily. K tomu se dá ideálně použít příkaz **`tr`**. Necháme ho nahradit všechny „špatné znaky“ za znak konce řádky.

Máme tedy adepty zapsané na samostatných řádkách a je třeba říct, které řádky se nám „líbí“. To je ale poměrné složité. Daleko snáze řekneme, jaké řádky se nám naopak nelíbí, a těch prostě zbavíme. Budeme špatné případy vylučovat postupně...

Řešení:

```
sinek:~> tr -cs '0-9.' '[\n*]' | sed '
> /^\./d                        ... tečka na začátku
> /\.$/d                        ... tečka na konci
> /\.\./d                       ... dvě tečky za sebou
> /\([0-9]*\.\)\{4\}/d          ... více než tři tečky celkem
> /[0-9]\{4\}/d                 ... alespoň čtyřmístné číslo
> /[3-9][0-9][0-9]/d            ... číslo mezi 300 a 999
> /2[6-9][0-9]/d                ... číslo mezi 260 a 299
> /25[6-9]/d                    ... číslo mezi 256 a 259
> /^0[0-9]/d                    ... číslo s nulou na začátku
> /\.0[0-9]/d                   ... dtto
> /\(.*\.\)\{3\}/ !d'           ... řádka, kde nejsou tři tečky
```

Poznámky:

Přepínač „**`-c`**“ (*complement*) u příkazu **`tr`** znamená, že znaky uvedené v prvním řetězci naopak zaměňovat nemá a má zaměňovat všechny ostatní. Ve druhé sadě máme pouze znak konce řádky. Tím zajistíme, že nám v rouře zůstanou jen číslice, tečky a znaky konce řádek.

Pro jistotu připomeneme, že přepínač „**`-s`**“ způsobí, že ve výstupu nebudou dva znaky konce řádek bezprostředně za sebou. Tím omezíme počet prázdných řádek, které by jinak **`sed`** musel zpracovávat.

V tomto příkladu už bylo zapotřebí takové množství příkazů editoru **sed**, že jsme je pro přehlednost raději místo středníku oddělovali znakem konce řádky, přestože jsme celý program zapsali jako jediný parametr na příkazové řádce.

Takový program už lze bez uzardění označit jako *skript* pro editor **sed** (*sed-skript*). Mohli bychom ho také zapsat do souboru a příkaz volat s přepínačem „**-f**“ (*file*):

```
sinek:~> tr -cs '0-9.' '[\n*]' | sed -f jméno_souboru
```

Přepínač je možné zadat i vícekrát, díky čemuž lze předem vytvořené pomocné skripty pro řešení určitých úkolů (třeba kontrolu čísel) skládat do požadované podoby.

Poznali jsme nový rys příkazů editoru, a to adresu ve tvaru *regulárního výrazu* (*vzoru*). Takový příkaz se vykoná na každé řádce, která vzoru vyhovuje. Všechny naše příkazy (až na poslední) lze tedy chápat jako „vyhovuje-li řádka tomuto vzoru, smaž ji“.

V posledním příkazu jsme potřebovali vyjádřit podmínku „řádka obsahuje nejvýše dvě tečky“. To ale vůbec není jednoduché. Daleko jednodušší je říct podmínku opačnou, tj. „řádka obsahuje (alespoň) tři tečky“. To je častý jev, *negace* regulárního výrazu je obecně velmi obtížná a většinou dokonce nemožná (zkuste popsat regulárním výrazem řádku, jež neobsahuje řetězec „`Libor`“). Proto editor **sed** umožňuje naopak *negovat zadanou adresu*. Pokud (těsně) před název příkazu napíšeme vykřičník, provádí se daný příkaz na všech řádkách, které adrese (zde regulárnímu výrazu) nevyhovují.

Vzor v adrese je nutno omezit z obou stran oddělovačem. I zde se standardně používá lomítko (jako u příkazu *substitute*) a pokud se ve vzoru lomítka vyskytují, musíme před ně psát zpětná lomítka. Další možností je zvolit jako oddělovač jiný znak, pak ovšem musíme před první výskyt oddělovače napsat zpětné lomítko. Smazání všech řádek obsahujících lomítko by se tedy provedlo třeba příkazy „**/\//d**“ nebo „**\@/@d**“.

Příkaz *delete* způsobí, že se řádka vymaže (neobjeví se na výstupu) a krom toho se pro danou řádku přeskočí vykonávání zbytku programu.

Vzhledem k tomu, že jsme vlastně jen vyřazovali řádky odpovídající vzorům, mohli jsme místo editoru použít příkaz **grep** (podobně jako v kapitole 1.14). Získali bychom tím jednu podstatnou výhodu, a to řádově vyšší rychlost, danou lepší implementací regulárních výrazů. Naproti tomu v editoru máme k dispozici pestrou paletu příkazů (kterou ale v tomto příkladu nevyužíváme). Kdybychom potřebovali obojí (rychlost i lepší příkazy), mohli bychom udělat kompromis – nejprve použít příkaz **grep** a probrat „nahrubo“ původní rozsah dat a konečné dořešení detailů už ponechat na pomalejší, ale šikovnější editor, pracující na malém zbytku dat (viz kap. 3.16).

3.7. Hledání slov

Zadání:

Vypište z `/etc/group` seznam skupin, jejichž členem je `root`.

Výstup je požadován ve tvaru „*jméno_skupiny*`:root`“.

Rozbor:

Tohle je opět úloha přesně pro editor. Na každé řádce, která obsahuje slovo „`root`“, potřebujeme vymazat vše kromě názvu skupiny a uživatele. To už bychom asi uměli.

Horší bude vyjádřit, že nám jde o celé slovo! Představte si, že by v systému byl např. uživatel „`kroota`“). Budeme muset slovo „`root`“ z obou stran nějak omezit.

Zleva to ještě půjde jednoduše. Jménu uživatele předchází dvojtečka (je-li v seznamu členů skupiny první) nebo čárka (není-li první). To umíme vyjádřit snadno („**`[:,]`**“).

Horší je to zprava. Tam za jménem uživatele musí být buď čárka, nebo nic (konec řádky). Pokud vás napadlo řešení „**`root[,$]`**“, tak na něj honem rychle zapomeňte! Víme už totiž, že metaznaky regulárních výrazů ztrácejí mezi hranatými závorkami svůj zvláštní význam. V hranatých závorkách je dolar jen obyčejný znak dolar.

Musíme si pomoci fintou. Není-li uživatel poslední v seznamu, musí za jeho jménem být čárka a pak nějaké (vcelku nezajímavé) znaky až do konce řádky. Pokud je poslední, nebude tam až do konce řádky nic. Můžeme tedy říci, že za hledaným slovem musí být nula nebo více výskytů řetězce „čárka a cokoliv“. Ten dokážeme popsat snadno („**`,.*`**“). Zbývá jen říct, že takových řetězců může být nula nebo více.

Řešení:

```
sinek:~> sed -n 's/:.*[:,]root\(,.*\)*$/:root/p' /etc/group
wheel:root
operator:root
```

Poznámky:

Už víme, že při volání s přepínačem „**`-n`**“ editor nevypisuje sám od sebe výsledné řádky po vykonání editačních příkazů. Právě proto jsme za závěrečné lomítko příkazu *substitute* připsali ještě parametr „**`p`**“ (*print*). Ten editoru říká „až řádku upravíš (pokud ji upravíš), vypiš ji“. Nezměněné řádky se vypisovat nebudou.

Kdybyste na přepínač „**`-n`**“ zapomněli, editor by vypisoval všechny řádky. Ty, jež neobsahují slovo „`root`“, by vypsal jednou a ostatní dvakrát (jednou díky parametru „**`p`**“ příkazu *substitute* a jednou sám od sebe na konci zpracování řádky).

Jiným zajímavým parametrem příkazu *substitute* je „**`w`**“ (*write*), který umožňuje řádku po modifikaci vypsat (připsat) do nějakého souboru.

3.8. Střelba na pohyblivý cíl

Zadání:

Najděte v `/etc/group` ty skupiny, jejichž členem je uživatel `root` nebo `forst`. Seznam skupin vypište ve tvaru „*jméno_skupiny*:*číslo_skupiny*:*seznam_členů*“.

Rozbor:

V minulé kapitole jsme řešili podobnou úlohu, přímé aplikaci předchozího řešení ale brání především to, že hledáme dva uživatele. Rozmyslete si, proč nejde napsat:

```
/[:,]root\(,.*\)*$/s/:[^:]*//p
/[:,]forst\(,.*\)*$/s/:[^:]*//p
```

Problém spočívá v tom, že pokud nějaká skupina obsahuje oba uživatele, provedou se oba příkazy *substitute,* a řádka se tak vytiskne dvakrát. Potřebovali bychom editoru říci: pokud najdeš řetězec „`root`“, řádku uprav, vypiš a smaž (víme už, že po *delete* se zbytek programu přeskakuje). Pak bychom mohli v klidu řešit druhý řetězec. Seskupení více příkazů tak, aby se provedly na určité řádce lze vyřešit *složeným příkazem*.

Kdyby v požadovaném tvaru výstupu nebyl seznam členů, žádný problém bychom neměli. Pokud by uspěl první příkaz *substitute*, seznam členů by se z řádky vymazal a hledání druhého řetězce v následujícím příkazu už by neuspělo. Editor totiž během své práce uchovává text řádky vstupu v tzv. *pracovním prostoru*, postupně prochází svoje příkazy a pokouší se je aplikovat na obsah pracovního prostoru. Pokud nějaký příkaz pracovní prostor modifikuje, další příkazy už pracují s jeho novým obsahem.

Na toto chování je třeba dát pozor, ale také toho lze občas šikovně využít. Například náš problém z minulé kapitoly s pravým koncem hledaného slova se dá s touto znalostí elegantně vyřešit. Co odlišuje ony dvě možné varianty výskytu jména uživatele? To, že za posledním uživatelem na řádce není čárka. No tak ji tam přidejme!

Řešení:

```
sinek:~> sed -n '
> s/$/,/                    ... přidání čárky na konec pro jednodušší test
> /[:,]root,/ {             ... test na slovo „root“
>   s/:[^:]*//              ... vymazání druhého pole (i s dvojtečkou)
>   s/,$//p                 ... vymazání přidané čárky, výpis
>   d                       ... ukončení práce s řádkou
> }
> /[:,]forst,/ {            ... test na slovo „forst“ a stejný postup
>   s/:[^:]*//
>   s/,$//p
>   d                       ... tenhle příkaz je zde jen kvůli symetrii
> }' /etc/group
```

Poznámky:

Doplňme pro pořádek popis činnosti editoru o začátek a konec: editor nejprve přečte jednu řádku vstupu do pracovního prostoru, pak postupně pracovní prostor modifikuje pomocí jednotlivých příkazů, a když všechny příkazy probere, celý obsah pracovního prostoru vypíše (pokud nebyl zavolán s přepínačem „**-n**“).

Princip postupné modifikace textu řádky v pracovním prostoru je logický, ale občas velmi záludný. Tipněte si, jak dopadne následující volání:

```
sinek:~> echo a-z | sed 's/a/z/;s/z/a/'
```

Pokud jste tipovali, že výsledkem bude řetězec „z-a“, pak si kapitolu přečtěte ještě jednou! Editor načte řádku („a-z“) do pracovního prostoru a první příkaz *substitute* změní jeho obsah na „z-z“. Druhý příkaz vezme tento řetězec a překlopí ho zase zpátky do původního tvaru! Nahradí totiž pouze první písmeno „z“ za „a“.

Pokud opravdu měl být výsledkem text „z-a“, lepší by bylo použít speciální příkaz editoru „**y**“, který má v podstatě stejnou funkci jako utilita **tr** (náhradu znaku za znak na základě dvou převodních tabulek):

```
sinek:~> echo a-z | sed 'y/az/za/'
```

Možná vás poněkud překvapí konstrukt „**s/$/,/**“. Skutečně, pokud ho budete číst jako „nahraď konec řádky čárkou“, budete zřejmě očekávat, že na řádce „zmizí“ znak konce řádky a místo něj se objeví čárka. Neboli, že dojde ke spojení řádek. Ale editor si do pracovního prostoru ukládá text řádky bez znaku konce řádky. Žádný znak konce řádky v pracovním prostoru nemá. Je třeba si použitý konstrukt přečíst třeba jako „nahraď prázdný řetězec ležící na konci řádky...“

Složený příkaz funguje vcelku očekávaným způsobem. Pokud řádka (přesněji „obsah pracovního prostoru“) vyhovuje adresní části příkazu „**{**“, provedou se všechny příkazy až po koncový příkaz „**}**“. Pozor na to, že „**{**“ a „**}**“ jsou skutečně samostatné příkazy (proto jsou také na samostatných řádkách).

V této kapitole jsme poprvé vytvořili relativně složitou konstrukci příkazů editoru. Pro lepší čitelnost jsme příkazy uvnitř složeného příkazu *odsadili* od začátku řádky (*indentovali*) tak, jak se to dělá běžně ve zdrojových kódech programů v jiných jazycích.

Kromě toho se u složitějších skriptů vyplatí používat komentáře. Komentářová řádka začíná znakem „**#**“ a celý zbytek řádky editor ignoruje. Mohli bychom tedy napsat třeba:

```
> s/$/,/  # pridani carky na konec pro jednodussi test
```

Až poznáme další příkazy editoru, dokážeme tento skript zapsat ještě elegantněji.

3.9. Další, prosím

Zadání:

Vypište do sloupečku seznam čísel skupin, do nichž patříte.

Vycházejte přitom z výstupu příkazu **`id`** rozlámaného jako v kap. 1.19:

```
sinek:~> id | tr ' =' '[\n*]'
…
groups
1004(forst),0(wheel),1(daemon),999(kernun),1028(j28)
```

Rozbor:

Začneme odzadu. Rozlámání řádky se seznamem skupin do více řádek provedeme příkazem *substitute*. Nahradíme (zhruba řečeno) čárky za konce řádek. Jenom to musíme udělat opakovaně a nahradit *všechny* výskyty. Pro tento účel má příkaz *substitute* parametr „**`g`**“ (*globální* náhrada). Globálně také vymažeme obsah závorek.

Horší problém spočívá v tom, že nás nezajímá řádka, jež obsahuje řetězec „`groups`“, ale ta, která po ní následuje! S tím jsme se zatím nesetkali, naše akce byly dosud vázány přímo na řádku, kterou jsme v té chvíli měli „v ruce“ (vlastně v pracovním prostoru).

Jedno řešení využívá možností adresní části příkazu. Už v kapitole 3.2 jsme se setkali s tím, že příkaz měl dvě adresy oddělené čárkou a prováděl se na všech řádkách z bloku daného intervalem čísel. Tady ale čísla řádek neznáme. Víme jen, že blok, který nás zajímá, začíná řádkou obsahující „`groups`“ a končí následující řádkou. Což kdybychom tedy rozsah adres pro náš blok vymezili *řetězcovými vzory*? Takový příkaz se stává platným (poprvé se provádí), jakmile editor narazí na řádku vyhovující prvnímu vzoru, a naposledy se vykoná na první řádce, jež následuje a vyhovuje druhému vzoru. Poté se zase bude čekat na řádku vyhovující vzoru prvnímu (což v našem případě už nenastane).

Když jako první vzor dáme řetězec „`groups`“ a jako druhý něco, co uspěje kdekoliv (třeba začátek řádky – „**`^`**“), příkaz se bude provádět přesně na našich dvou řádkách.

Pak už stačí jen vymyslet takový příkaz, aby na první řádce neuspěl a na druhé ano. Stačil by třeba *print* s adresou ve tvaru vzoru „alespoň jedna číslice“. Jenže náš příkaz už adresní část má („**`/^groups$/,/^/`**“). Pomůžeme si tedy složeným příkazem:

```
sinek:~> id | tr ' =' '[\n*]' | sed -n '
> s/([^()]*)//g              … smaž obsah (všech) závorek
> s/,/\                      … nahraď (všechny) čárky za konce řádek
> /g
> /^groups$/,/^/ {           … od řádky s „groups“ po následující …
>     /[0-9]/p               … vypisuj řádky obsahující číslici
> }'
```

Výhody předchozího řešení s blokem řádek bychom využili především tehdy, pokud by blok obsahoval více řádek. Náš „blok“ obsahoval jen dvě řádky, zajímala nás vlastně jen jediná „následující“ řádka a v tom případě máme po ruce jednodušší řešení.

Představte si, že máte úlohu naprogramovat v nějakém obyčejném programovacím jazyce. Algoritmus by asi zněl: čti soubor, a jakmile najdeš řádku obsahující „`groups`“, načti další řádku a tu modifikuj. Ale přesně totéž můžeme (pravda, malinko jinými výrazovými prostředky) zapsat i v jazyce editoru. Musíme se jen naučit příkaz *next* („**`n`**“) – ten jednoduše ukončí zpracování aktuální řádky a načte řádku novou. Dělá tedy zhruba totéž, co se děje normálně na konci skriptu, jen s tím rozdílem, že zpracování nové řádky pokračuje od místa, kde byl příkaz *next*, a nikoliv od začátku skriptu.

```
sinek:~> id | tr ' =' '[\n*]' | sed -n '
> /^groups$/ {                  ... narazíš-li na řádku obsahující „groups“ ...
>   n                           ... posuň se na další řádku
>   s/([^()]*)//g               ... smaž obsah (všech) závorek
>   s/,/\                       ... nahraď (všechny) čárky za konce řádek
> /g
>   p                           ... vytiskni řádku
> }'
```

Řešení:

Už dvě byla.

Poznámky:

Pokud chceme znak konce řádky zapsat do řetězce náhrady, napíšeme zpětné lomítko a za ním odřádkujeme. Je to jediný zaručeně přenositelný zápis, speciality typu „**`\n`**“ sice někde fungují, ale zcela bez záruky. Zdůrazňuji, že hovoříme o řetězci náhrady (je to jinak než v regulárním výrazu ve vzoru, tam se naopak skutečně píše „**`\n`**“).

Ještě je třeba upřesnit, že i po rozlámání obsahu pracovního prostoru na více řádek pořád pracujeme s celým pracovním prostorem najednou. Efekt vzniku více řádek se projeví až při výpisu na standardní výstup.

Ve druhém řešení bychom se mohli obejít bez příkazu *print*, kdybychom příkazu *substitute* dali kromě parametru „**`g`**“ (nahraď globálně) ještě parametr „**`p`**“ (po provedení všech náhrad vypiš obsah pracovního prostoru). Jenom bychom museli prohodit oba příkazy *substitute*, protože pokud by uživatel byl pouze v jedné skupině, žádná čárka by na řádce nebyla, substituce by neproběhla a řádka by se nevypsala. Nejprve bychom tedy zaměnili případné čárky a pak současně vymazali všechny závorky a text vypsali:

```
>   s/,/\                       ... nahraď (všechny) čárky za konce řádek
> /g
>   s/([^()]*)//gp              ... smaž obsah (všech) závorek a vytiskni
```

3.10. Cyklus

Zadání:

Napište program, který na standardním vstupu dostává řetězce obsahující (absolutní a korektní) cesty v souborovém systému a „normalizuje“ je.

Příklad úprav:

```
/etc/./passwd              ⇒      /etc/passwd
/usr/local/lib/../src/     ⇒      /usr/local/src/
```

Rozbor:

Základní myšlenka je jednoduchá. Použijeme příkaz *substitute* a vymažeme všechny nežádoucí kombinace. Nejprve potřebujeme vymazat elementy cesty ve tvaru „`./`“ (nebo „`/.`“). Jenže takové posloupnosti znaků by se nám v cestách mohly objevit i jako součást nějakého „normálního“ jména souboru (stačí, aby jeho jméno začínalo či končilo tečkou). Potřebovali bychom tedy spíše nahrazovat posloupnost „`/./`“ lomítkem.

A potřebujeme to provádět opakovaně, pro všechny výskyty na řádce. Ovšem pouhé použití parametru „**`g`**“ příkazu *substitute*, který jsme poznali minule, nám tenhle problém nevyřeší. Globální nahrazování totiž probíhá tak, že po provedení jedné substituce editor pokračuje v prohledávání pracovního prostoru až za nahrazeným řetězcem:

```
/etc/./.passwd
/etc/./passwd
```

⇒

Pokud tedy budou v cestě dva bezprostředně po sobě jdoucí odkazy na aktuální adresář („`/././`“), vymazán bude pouze první z nich.

Pokud vám připadá, že by nám editor v téhle věci mohl vyjít vstříc a pochopit naše globální nahrazování „správně“, zkuste si tipnout, co by, chudák, měl dělat, kdybychom zadali příkaz „**`s/0/00/g`**“. To je zcela korektní příkaz a jeho smyslem je zjevně provést inflaci (zdvojit všechny nuly). Jenže kdyby editor při globálním nahrazování pokračoval ve zpracovávání řádky (v dalším hledání nuly) nikoliv za změněným textem, ale ještě uvnitř něj, vedl by takový korektní příkaz k nekonečné smyčce...

Globální náhrada nám tedy nepomůže a musíme zařídit opakování jiným způsobem. Zachrání nás prostředky pro *řízení toku* editace, které nám umožní vytvořit *cyklus*. Jazyk editoru dovoluje označit si určitá místa kódu *návěštími* a na tato místa během provádění editačních příkazů *skákat*. Příkazy skoku existují jednak *nepodmíněné* (provádějí se vždy, příkaz *branch*, „**`b`**“) a jednak *podmíněné* (příkaz *test*, „**`t`**“). Podmínkou, která rozhodne o provedení skoku, je to, zda byla (od počátku zpracování řádky, resp. od zavolání posledního příkazu *next* či *test*) vykonána nějaká substituce.

Díky tomu dokážeme editoru říci, že má v cyklu opakovat příkaz *substitute* tak dlouho, dokud nevymaže veškeré nežádoucí elementy cest.

Do stejného cyklu můžeme přidat i mazání podřetězců „/*cokoliv*/../“. Ty popíšeme pomocí regulárního výrazu „**/[^/]*/\./**“. Výrazu sice formálně vyhovuje i řetězec „/.../.../“, který určitě vymazat nechceme, ale díky podmínce korektnosti vstupu víme, že takový řetězec by se musel vyskytovat v cestě ve tvaru „/*adr1*/*adr2*/.../.../“ a v ní by náš regulární výraz odpovídal části „/*adr2*/.../“, takže vše je v pořádku.

Řešení:

```
sinek:~> sed '
> : cyklus
>   s:/\./:/:g
>   t cyklus                  ... pokud se řádka změnila, opakuj substituce
>   s:/[^/]*/\.\./:/:g
>   t cyklus                  ... pokud se řádka změnila, opakuj substituce
> '
```

Poznámky:

V příkazu *substitute* jsme jako oddělovač vzoru a řetězce náhrady použili místo lomítka dvojtečku, jak jsme to avizovali v kapitole 3.1. Při použití lomítek bychom všechna vnitřní lomítka museli prefixovat zpětnými lomítky:

```
>   s/\/\.\//\//g
```

Přiznám se, že když jsem se poprvé dozvěděl, že editor **sed** má příkazy skoku, napadlo mě, jestli to náhodou na editor už není trochu „silné kafe“. Ale tenhle příklad z praxe ukazuje, že i relativně jednoduché úlohy nemají bez příkazu skoku jednoduché řešení, zatímco s ním ho mají a je dokonce i docela srozumitelné. Samozřejmě, že se skoky nepoužívají každý den, ale je dobré vědět, že v případě potřeby jsou k dispozici.

Za názvem návěští nesmíte napsat mezeru! Ačkoliv jinde nebývá editor **sed** na mezery tak háklivý, zde si dopřává trochu zhýralosti. A stejně tak není možné v tomto místě použít jako oddělovač příkazů středník a musíte opravdu odřádkovat.

S příkazy skoku už také můžeme poněkud zpřehlednit skript z kapitoly 3.8 tím, že oddělíme výběr řádek od manipulace s nimi:

```
sinek:~> sed '
> s/$/,/                    ... přidání čárky na konec pro jednodušší test
> /[:,]root,/ b ok          ... test na slovo „root“, odskok do druhé části
> /[:,]forst,/ b ok         ... test na slovo „forst“, odskok do druhé části
> d                         ... ukončení práce s nevhodnými řádkami
> : ok
> s/:[^:]*//                ... vymazání druhého pole (i s dvojtečkou)
> s/,$//                    ... vymazání přidané čárky
> ' /etc/group
```

3.11. Přidávání řádek

Zadání:

Vypište seznam administrátorů systému (uživatelů s nulovým UID).

Pro každého vypište dvě řádky ve tvaru:

```
Administrator:                                  ... pevný text
root:*:0:0:Super User:/root:/bin/bash           ... řádka /etc/passwd
```

Rozbor:

Pro tuto úlohu se dobře hodí příkaz *insert* („**i**“), pomocí nějž se dají přidávat celé řádky. Před každou řádku `/etc/passwd` s nulovým UID vložíme novou řádku s textem „`Administrator:`“. Zároveň ale potřebujeme potlačovat výpis řádek, které nulové UID neobsahují. Toho můžeme dosáhnout dvojím způsobem: buď editor zavoláme s přepínačem „**-n**“ a necháme příkazem *print* vypsat jen správné řádky anebo naopak nesprávné řádky explicitně vymažeme příkazem *delete*.

Řešení (*print*):

```
sinek:~> sed -n '/:0:[0-9]/ {
>   i\
> Administrator:
>   p
> }' /etc/passwd
```

Řešení (*delete*):

```
sinek:~> sed '/:0:[0-9]/ !d
> i\
> Administrator:
> ' /etc/passwd
```

Poznámky:

Použitý regulární výraz vyjadřuje, že v řádce musí být pole obsahující nulu a za ním hned následuje pole, které začíná číslicí. Dvě číselná pole za sebou se v `/etc/passwd` nacházejí právě ve sloupcích s číslem uživatele (UID) a s číslem primární skupiny (GID). Kdybychom testovali pouze nulu jako hodnotu pole

```
sinek:~> sed -n '/:0:/ { ... atd.
```

program by vypsal i řádky, které mají nulové GID a nikoliv UID.

Přísně vzato by regulární výraz měl spíš popsat, že se nám jedná právě o třetí pole

```
sinek:~> sed -n '/^[^:]*:[^:]*:0:/ { ... atd.
```

neboli: od začátku řádky řetězec složený ze znaků mimo dvojtečku, pak dvojtečku, pak ještě jeden řetězec bez dvojtečky, druhou dvojtečku a pak nulové pole. Na tomto regulárním výrazu jsou pěkně vedle sebe vidět obě odlišná použití stříšky jako metaznaku, ale jinak je výraz poněkud složitý. Proto jsme se dopustili určitého

zjednodušení a naše verze vychází ze znalosti sémantiky `/etc/passwd` a předpokladu, že plné jméno žádného uživatele nezačíná číslicí. Posouzení oprávněnosti takových zjednodušení je občas problematické, a pokud byste chtěli opravdu „blbuvzdorný“ program, nemohli byste si takové úlevy dovolit.

Pokud bychom potřebovali vložit jedním příkazem *insert* více řádek, zapíšeme je za sebe a na konec každé kromě poslední napíšeme zpětné lomítko. V podstatě je to zde s koncem řádky podobné jako u shellu: příkaz *insert* se zapisuje (včetně textu nově vkládaných řádek) jakoby na jedinou řádku a mezilehlé znaky konce řádek je nutno uvodit zpětným lomítkem.

Pozor na to, že v některých implementacích můžete narazit na problémy s vkládáním prázdných řádek nebo řádek začínajících mezerami, přestože norma žádnou diskusi nad sémantikou nepřipouští.

Vykřičník uvedený před příkazem *delete* ve druhém řešení „neguje“ výběr řádek. Tuto funkčnost už známe. V našem případě by ale potřebná negace regulárního výrazu nebyla zase tak složitá – mohli bychom použít příkaz:

```
/^[^:]*:[^:]*:[1-9]/d
```

Kromě příkazu *insert* (vlož před řádku) umí editor ještě příkaz *append* (vlož za řádku) a *change* (nahraď řádky).

- Příkaz *insert* („**`i`**“) vypíše text nových řádek na výstup a pokračuje ve zpracování programu s nezměněným obsahem pracovního prostoru.
- Příkaz *append* („**`a`**“) pouze uloží obsah vkládaného textu a dál pokračuje ve zpracování skriptu s nezměněným obsahem pracovního prostoru. Teprve před načtením další řádky do pracovního prostoru editor vypíše uložený text.
- Příkaz *change* („**`c`**“) funguje vlastně jako kombinace *insert* a *delete* – vypíše text nových řádek na výstup a pak vymaže dotčenou řádku (resp. řádky, protože na rozdíl od ostatních dvou příkazů může v adrese dostat rozsah řádek). Pak editor skočí na začátek skriptu, načte novou řádku a pokračuje normálně dál.

3.12. Úschovna zavazadel

Zadání:

Vypište seznam administrátorů systému ve tvaru:

```
Administrator root:                          ... text se jménem uživatele
root:*:0:0:Super User:/root:/bin/bash        ... řádka /etc/passwd
```

Rozbor:

Změna oproti minulé úloze spočívá v tom, že vkládaný text není pevný, ale využívá části existující řádky.

Kdyby mělo být pořadí vypisovaných řádek (v rámci každé dvojice) obrácené, bylo by řešení snadné. Nejprve bychom původní řádku příkazem *print* opsali na výstup v nezměněném tvaru a potom bychom její text v pracovním prostoru pomocí editačních příkazů změnili na požadovaný tvar „`Administrator` *jméno*`:`“.

Ovšem má-li se nová řádka ve výstupu objevit dříve, museli bychom nejprve řádku změnit (abychom mohli nový text vypsat), jenže poté už bychom neměli k dispozici původní text řádky. Potřebovali bychom nějaké místo, kam bychom si originální řádku „odložili“. A takové místo opravdu existuje, jmenuje se *odkládací prostor* (hold space) a editor nám nabízí různé druhy přesunů mezi pracovním a odkládacím prostorem.

Postup bude tedy následující: Příkazem *hold* („**h**“) nejprve zkopírujeme každou (správnou) řádku do odkládacího prostoru, poté s jejím textem v pracovním prostoru provedeme potřebné úpravy a výsledek vypíšeme. Pak příkazem *get* („**g**“) zkopírujeme originální řádku zpátky z odkládacího do pracovního prostoru a vytiskneme.

Řešení:

```
sinek:~> sed -n '
> /:0:[0-9]/ {
>   h
>   s/:.*/:/
>   s/^/Administrator /p
>   g
>   p
> }' /etc/passwd
```

Poznámky:

Příkaz *hold* kopíruje obsah pracovního prostoru do odkládacího prostoru a příkaz *get* to dělá opačně (z odkládacího do pracovního). Vedle toho existuje ještě příkaz *exchange* („**x**“), který obsahy prostorů *vymění*.

Nepleťte si příkazy „**p**“ (*print*) a „**g**“ (*get*) s parametry „**p**“ a „**g**“ příkazu *substitute*.

3.13. Co na nádraží nesnášejí

Zadání:

Rozdělte prostý text na vstupu do odstavců pomocí HTML značek „ “ a „ “. Hranici odstavce tvoří jedna nebo více prázdných řádek.

HTML je jazyk, v němž se zapisují texty webových stránek. Je to jazyk *značkovací*, jeho součástí jsou různé značky, kterými autor stránky vyjadřuje svá přání stran formátu či obsahu (blíže viz [9]). Pomocí značky „p“ (*paragraph*) se zapisuje požadavek na vytvoření odstavce. Text odstavce se uzavře mezi dvojici značek „ “ a „ “.

Úloha pochází z knihy [3].

Rozbor:

Nejjednodušší by bylo na začátek textu dát značku „ “, na konec „ “ a každou prázdnou řádku nahradit posloupností „ “. Jenže bychom stejně museli nějak řešit problém více prázdných řádek po sobě. Pojďme zkusit něco jiného.

Rádi bychom poskládali celý odstavec a vypsali ho, jakmile bude kompletní, patřičně obložený správnými značkami. Na to by se nám opět hodila úschovna, ale potřebovali bychom do ní postupně přidávat další řádky. To v úschovně zavazadel na nádraží bytostně nesnášejí. V editoru to ale projde. Namísto příkazu *hold* („**h**“) použijeme příkaz *Hold* („**H**“, velké!), který k obsahu odkládacího prostoru (i kdyby byl zatím prázdný!) přidá znak konce řádky (ve vzoru ho pak budeme zapisovat jako „**\n**“) a za něj *připojí* obsah pracovního prostoru. Tento postup budeme opakovat na neprázdných řádkách. Jakmile naopak narazíme na prázdnou řádku, obsahy prostorů prohodíme, provedeme potřebné substituce (přidání značek) a prázdnou řádku uloženou do odkládacího prostoru opět vrátíme příkazem *Get* („**G**“) na konec pracovního prostoru.

Zbývají nám dva problémy.

Jedním je situace, kdy poslední řádka není prázdná. V tom případě nedojde nikdy k poslední výměně prostorů, doplnění značek a tím ani k výpisu posledního odstavce. Proto musíme tento případ ošetřit zvlášť, a to úplně na začátku skriptu. Je-li poslední řádka neprázdná, přidáme ji normálně do odkládacího prostoru, ale místo příkazu *delete* pracovní prostor pouze vyprázdníme a necháme „řádku“ normálně dál zpracovat. Tím vlastně simulujeme příchod prázdné řádky, která korektně ukončí poslední odstavec.

Druhým problémem je výskyt dvou po sobě jdoucích prázdných řádek. V takovém případě dojde k přesouvání a upravování neexistujícího odstavce, což nedává dobrý výsledek. Chtělo by to nějakou podmínku „je-li v uloženém odstavci to a to...“. Ale to vlastně umíme. Po provedení příkazu *exchange* se obsah odstavce ocitne v pracovním prostoru a ten otestovat umíme. Provedeme skok *bez návěští*, což znamená na konec programu (editor vytiskne obsah pracovního prostoru a pokračuje).

Řešení:

```
sinek:~> sed '
> $ {                  ... na poslední řádce ...
>     /./ H            (1) připojení (neprázdné) řádky do odkládacího prostoru
>     s/.*//           (2) vymazání pracovního prostoru
> }
> /./ {                ... na neprázdné řádce ...
>     H                (3) připojení řádky do odkládacího prostoru
>     d                (4) konec práce s řádkou
> }
> /^$/ {               ... na prázdné řádce ...
>     x                (5) záměna prostorů
>     /^$/ b           (6) skok na konec skriptu, pokud je prostor prázdný
>     s/\n/<p>/        (7) vložení značky místo prvního konce řádky
>     s/$/<\/p>/       (8) vložení značky na konec prostoru
>     G                (9) připojení řádky z odkládacího prostoru
> }'
```

Poznámky:

Možná by bylo dobré pro lepší pochopení ukázat, jak se postupně mění obsahy obou prostorů. Mějme na vstupu řádky „A“, „B“, „“, „“ a „E“:

	pracovní prostor	výstup	odkládací prostor
načtena 1. řádka „A“	A		
provedeny příkazy 3 a 4		ne	\nA
načtena 2. řádka „B“	B		\nA
provedeny příkazy 3 a 4		ne	\nA\nB
načtena 3. řádka „“			\nA\nB
proveden příkaz 5	\nA\nB		
provedeny příkazy 7 a 8	A\nB		
proveden příkaz 9	A\nB \n	ano	
načtena 4. řádka „“			
proveden příkaz 5			
proveden příkaz 6		ano	
načtena 5. řádka „E“	E		
proveden příkaz 1	E		\nE
vymazání řádky příkazem 2			\nE
proveden příkaz 5	\nE		
provedeny příkazy 7 a 8	E		
proveden příkaz 9	E \n	ano	

3.14. Ještě jednou do úschovny

Zadání:

Vypište seznam uživatelů ve tvaru „*login_jmeno=JMENO PRIJMENI*“.

Rozbor:

Ne, že bychom úlohu neuměli řešit jinak nebo že by se nám řešení z kapitoly 1.16 nějak zásadně nelíbilo, ale prostě to chceme zkusit v editoru.

Pro převod na velká písmena by se nám hodil příkaz „**y**“, o němž jsme se zmiňovali už dříve. Problém je, že mu nejde říct, aby pracoval jen s částí řádky. Metodu vedoucí k cíli jsme ale viděli minule – odložíme si řádku do úschovny a budeme si tam jenom chodit pro jednotlivé části řádky.

Nejprve příkazem *hold* každou řádku zkopírujeme do odkládacího prostoru. Poté z ní odstraníme vše, kromě sloupce s plným jménem uživatele. Pak můžeme celý obsah pracovního prostoru (tedy vlastně jen správný sloupec) převést na velká písmena. Následně prohodíme obsahy obou prostorů a příkazem *Get* připojíme obsah odkládacího prostoru na konec pracovního prostoru. Na závěr už jen z pracovního prostoru vymažeme vše nepotřebné.

Řešení:

```
sinek:~> sed '
> h
> s/[^ ]*://
> s/:.*//
> y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
> x
> G
> s/:.*\n/=/' /etc/passwd
```

Poznámky:

Prostudujte si první příkaz *substitute*. Obešli jsme se zcela bez zpětných referencí. Řekli jsme: „vymaž (na začátku řádky) všechna pole, která neobsahují mezeru“.

Příkaz „**y**“ bohužel nenabízí žádné zkratky seznamů znaků (intervaly nebo třídy), a tak je musíme vypsat celé.

3.15. Zalomené řádky

Zadání:

Napište program, který v textu na svém vstupu mění „`Copyright 2000-2008`“ na „`Copyright 2000-2009`“, a to i v případě, že slovo „`Copyright`“ je na konci řádky.

Rozbor:

Celá komplikace spočívá právě ve zpracování zalomených řádek. Pokud budeme hledat pouze řetězec „`Copyright 2000-2008`“, nenahradíme výskyty nalézající se na rozhraní řádek, tj. když první řádka končí slovem „`Copyright`“. Nezbývá nám, než editor donutit, aby v případě nevhodného zalomení obě sousední řádky „slepil“ – umístil do pracovního prostoru obě. K tomu slouží příkaz *Next* („**`N`**“). Jeho úkolem je *připojit* k obsahu pracovního prostoru znak konce řádky a za něj načíst ze vstupu *novou řádku*.

Uvedený postup ale skrývá jedno nebezpečí, které není na první pohled patrné. Pokud totiž bude slovem „`Copyright`“ končit poslední řádka vstupu, příkazu *Next* se nepovede načíst další řádku ze vstupu a celý skript skončí, aniž by vypsal poslední řádku, která „zbyla“ v pracovním prostoru. Na poslední řádce musíme tedy příkazy *Next* a *branch* vynechat (obejdeme je pomocí skoku na návěští „`konec`“).

Řešení:

```
sinek:~> sed '
> : dalsi
>   $ b konec                  ... na poslední řádce Next neprovádíme
>   /Copyright$/ {             ... na špatně zalomené řádce...
>     N                        ... přidáme další řádku do pracovního prostoru
>     b dalsi                  ... a zopakujeme kontrolu
>   }
> : konec                      ... závěrečná substituce
> s/Copyright\([ \n]\)2000-2008/Copyright\12000-2009/g'
```

Poznámky:

Příkaz *Next* představuje další ze způsobů, jak se v pracovním prostoru může objevit znak konce řádky. Víme, že **`sed`** jinak pracuje jen s vnitřkem řádky. Hledání znaku konce řádky proto nikdy neuspěje, s výjimkou podobných případů jako je tento.

Víme i to, že ve vzoru se znak konce řádky zapisuje jako „**`\n`**“. Je to jediná výjimka, kdy zpětné lomítko hraje i mezi hranatými závorkami roli metaznaku.

Abychom nemuseli rozlišovat případ rozdělené a nerozdělené řádky, uzavřeli jsme ve vzoru oddělovač (mezeru nebo „**`\n`**“) do závorek a v řetězci náhrady použijeme zpětnou referenci „**`\1`**“. Kolize zpětné reference s číslem, které následuje („**`2000`**“), se nemusíme bát, protože zpětné reference jsou pouze jednociferné.

3.16. Paměť

Zadání:

Napište program, který vypíše seznam zneplatněných dokumentů RFC.

RFC (Request for Comments) je sada dokumentů, které popisují a definují chování sítě Internet. Jsou k nalezení na většině internetových serverů poskytujících veřejně dostupné dokumenty, např. na `ftp://sunsite.mff.cuni.cz/Network/RFCs`).

Postupem času se samozřejmě starší dokumenty stávají neplatnými. Smyslem úlohy je vyhledat v indexovém souboru (`rfc-index.txt`, stáhněte si ho) čísla těch RFC, které jsou označené jako „`Obsoleted`“. Formát indexového souboru:

```
0172 The File Transfer Protocol. A. Bhushan, B. Braden,
     W. Crowther, E. Harslem, J. Heafner, A. McKenzie,
     J. Melvin, B. Sundberg, D. Watson, J. White. June 1971.
     (Format: TXT=21328 bytes) (Obsoleted by RFC0265)
     (Updates RFC0114) (Updated by RFC0238)
     (Status: UNKNOWN)

0173 ...
```

Každému RFC přísluší jeden záznam (blok řádek) ukončený prázdnou řádkou. Na první řádce je číslo, případný atribut „`Obsoleted`“ najdeme na některé z následujících řádek.

Rozbor:

Klasický nástroj na prohledávání textů, příkaz **`grep`**, nám zde nebude moc platný. Dokáže sice nalézt hledaný text, ale svázání řádky obsahující „`Obsoleted`“ a řádky s číslem RFC, která jí předchází, není vůbec jednoduché.

Problém editoru **`sed`** je, že se v něm nemůžeme „vracet“. Jakmile „opustíme“ úvodní řádku s číslem RFC, už se k této hodnotě nedostaneme. Jedna možnost tedy spočívá v tom, že se do pracovního prostoru postupně kopíruje celý blok řádek odpovídající jednomu RFC (příkazem *Next*) a jakmile je blok kompletní (na jeho konci je prázdná řádka), zkontrolujeme výskyt hledaného řetězce a v případě úspěchu vypíšeme z první řádky hodnotu RFC (vymažeme z bloku vše počínaje první mezerou):

```
sinek:~> sed -n '
> : dalsi
> /\n$/ {
>     /Obsoleted/s/ .*//p
>     b
> }
> /^[0-9]\{4\}/ {
>     N
>     b dalsi
> }' rfc-index.txt
```

Toto řešení je ale dost pomalé, protože neustále modifikuje obsah pracovního prostoru. Rychlejší bude uchovávat si pouze úvodní řádky s číslem RFC a v případě výskytu hledaného textu vypsat poslední uložené RFC. Jedinou možností, kam si lze v editoru něco uložit, je odkládací prostor.

Řešení:

```
sinek:~> sed -n '
> /^[0-9]\{4\}/ h                ... ulož řádku s číslem RFC
> /Obsoleted/ {
>     g                          ... obnov řádku s číslem RFC
>     s/ .*//p                   ... vypiš samotné číslo RFC
> }' rfc-index.txt
```

Poznámky:

Pozor na to, že jakmile máme v pracovním prostoru více řádek, metaznaky „**^**“ a „**$**“ sice nadále představují „začátek“ a „konec“ pracovního prostoru, ale ty se už nekryjí se začátky a konci jednotlivých řádek. Metaznak „**^**“ lze pak přeložit jako „začátek *první* řádky“, metaznak „**$**“ jako „konec *poslední* řádky“ a výraz „**\n**“ představuje zlom mezi řádkami *uvnitř* pracovního prostoru. Konstrukt „**\n$**“ použitý v prvním pokusu o řešení tedy znamená „poslední řádka v pracovním prostoru je prázdná“. A stejně tak regulární výraz „**^[0-9]\{4\}**“ neznamená „aktuální řádka začíná číslem“, ale „v pracovním prostoru je uložen text, jehož první řádka začíná číslem“.

Na začátku analýzy jsme si posteskli, že v tomto příkladu nemůžeme použít příkaz **grep**. Je to škoda i proto, že bývá obvykle podstatně rychlejší než editor **sed**. Ovšem jistou pomoc by nám přece jen mohl poskytnout. Kdybychom nejprve ze souboru vybrali pouze řádky s textem „`Obsoleted`“ a samozřejmě řádky s čísly RFC, zmenšil by se výrazně objem dat, které bude muset **sed** zpracovávat. Navíc bychom na skriptu pro **sed** nemuseli měnit ani čárku!

```
sinek:~> grep -e Obsoleted -e ^[0-9] rfc-index.txt | sed ...
```

Na mém počítači způsobila tato úprava skoro dvojnásobné zrychlení, a to přitom nezpracováváme žádná opravdu veliká data. Konkrétní zrychlení pochopitelně závisí na mnoha faktorech, např. na tom, kolik řádek by filtrace vyřadila. Při opakovaném volání by výsledek mohl být výrazně ovlivněn také tím, zda se v řešení bez filtrace povede operačnímu systému uložit prohledávaný soubor do cache (viz kap. 4.9) a díky tomu ho nebude neustále číst z disku. Záležet bude samozřejmě hodně na tom, jak moc se liší efektivita implementace regulárních výrazů v utilitách **grep** a **sed**. Zvláště se to může projevit při zpracování národních znakových sad – na některých testovaných systémech zapnutí národního prostředí způsobilo snížení rychlosti příkazu **grep** o dva řády!

3.17. Vertikální zrcadlo

Zadání:

Napište program, který v textu převrací obsah řádek.

V některých systémech takový program existuje pod jménem **`rev`**.

Rozbor:

Základní myšlenka je jednoduchá. Budeme v pracovním prostoru znak po znaku převracet obsah řádky – postupně každý znak přesuneme na začátek, takže jako poslední se na začátek řádky dostane opravdu poslední znak původní řádky. Problém spočívá v tom, že musíme umět poznat, kdy už je řádka převrácená celá. Na to bychom si potřebovali do řádky vložit nějakou *značku*. Vybrat znak, který v řádce určitě není. To obecně není vůbec jednoduché. Naštěstí existuje jeden znak, který za těchto podmínek na řádce opravdu nemůže být – znak konce řádky.

Vložíme tedy na začátek řetězce jako zarážku znak konce řádky. Pak budeme v cyklu přesouvat první znak za zarážkou na začátek řádky. Cyklus skončí ve chvíli, kdy za zarážkou už žádný znak nebude, příkaz *substitute* proto neuspěje a podmíněný příkaz skoku *test* se neprovede. V tom okamžiku bude řádka převrácená celá a přidanou zarážku zase smažeme.

Řešení:

```
sinek:~> sed '
> s/^/\
> /                              (1) vložení zarážky na začátek řádky
> : dalsi
>   s/\(.*\n\)\(.\)/\2\1/        (2) přesun znaku za zarážkou na začátek řádky
>   t dalsi                      (3) opakování (2), pokud došlo ke změně řádky
> s/\n//'                        (4) odstranění zarážky
```

Poznámky:

Náš skript je ovšem ve skutečnosti naprogramovaný špatně! Naštěstí se chyba nijak negativně neprojeví, ale pokud bude na vstupu prázdná řádka, budou se provádět příkazy, které vůbec nechceme! Kde je „zakopaný pes“? Náš skript předpokládá, že podmíněný skok (3) proběhne podle toho, zda se vykonal příkaz (2). Jenže příznak úspěšného provedení substituce (který je podmínkou pro vykonání skoku) zůstává neustále v platnosti, dokud se neuskuteční nějaký podmíněný skok nebo se nenačte nová řádka. V našem případě ale tento příznak pro každou řádku poprvé nastaví už příkaz (1)! Neboli od této chvíle se první následující podmíněný příkaz skoku jistě provede, ať mezitím uděláme či neuděláme libovolné substituce. A příznak zruší právě až první vykonání podmíněného skoku, tedy příkaz (3). Naštěstí. Protože díky tomu je další chování skriptu už „očekávané“. Ale jistě si umíte představit, že takové chybné

provedení skoku by v určitých situacích mohlo mít pro skript fatální následky. Občas by se proto hodilo mít možnost „zrušit“ příznak provedení substituce. Ale nic takového neexistuje. Pokud je takový krok opravdu nezbytný, musíme provést podmíněný skok „naprázdno“, například takto:

```
sinek:~> sed '
> s/^/\
> /
> t dalsi                          ... skok „nikam“, zrušení příznaku substituce
> : dalsi
>   s/\(.*\n\)\(.\)/\2\1/
>   t dalsi                        ... skok v závislosti na předchozí substituci
> s/\n//'
```

Rozmyslete si, co se vlastně skrývá za oběma zpětnými referencemi v příkazu 2: „**\1**“ reprezentuje celou doposud převrácenou část řádky (včetně zarážky) a „**\2**“ první z ještě nepřevrácených znaků (za zarážkou).

Trik se zarážkou a s postupným zpracováním řádky okolo ní je v editoru **sed** velmi běžný. Je to jednoduchá metoda, jak obejít fakt, že v editoru nemáme žádné proměnné.

Raději ještě jednou připomeneme, že znak konce řádky se ve vzoru zadává jako „**\n**“ (např. v poslední substituci), zatímco v řetězci náhrady se píše jako zpětné lomítko následované znakem konec řádky (v první substituci).

3.18. Leporelo

Zadání:

Napište program, který rozláme aritmetický výraz zadaný na jediné řádce vstupu tak, že jednotlivé atomy (přirozená čísla, závorky a operátory) budou na zvláštních řádkách.

Rozbor:

Úlohu bychom asi dokázali řešit různými způsoby, ale ukážeme si metodu, která nám může v mnoha případech značně usnadnit život.

Pomocí regulárních výrazů je obvykle možné velmi dobře kontrolovat, co se nachází na nějakém specifickém místě řádky – např. na jejím začátku. Podmínka vztahující se na text někde uprostřed řádky se tvoří daleko hůře. Velmi snadno zformulujeme například myšlenku „je-li na začátku řádky číslo, je to v pořádku a můžeme pokračovat“. Jenže pro pokračování už nebude platit, že další text se nachází na začátku řádky, tudíž formulace se bude tvořit daleko hůře. Kdybychom ale zpracovanou část řádky oddělili (tj. zapsali na výstup a smazali z pracovního prostoru), měli bychom zase „čistý stůl“. Podobá se to skládání dětského leporela.

K práci s úvodní částí prostoru – od začátku do prvního znaku konce řádky – slouží příkazy *Print* („`P`“) a *Delete* („`D`“). Příkaz *Print* úvodní část vypíše a *Delete* ji vymaže.

Řešení:

```
sinek:~> sed '
> s/^ *\([0-9]\{1,\}\)/\1\      ... oddělení čísla znakem konce řádky
> /
> t dalsi
> s/^ *\([-+/*()]\)/\1\         ... oddělení operátoru znakem konce řádky
> /
> t dalsi
> s/^./&\                       ... ošetření nekorektního znaku
> /
> : dalsi
> P                             ... vypsání první řádky (atomu)
> D'                            ... zrušení první řádky a posun na další
```

Poznámky:

Po provedení příkazu *Delete* se editor vrací na začátek skriptu a prochází znova jednotlivé příkazy, přičemž pracuje s novým (zkráceným) obsahem pracovního prostoru. To je rozdíl oproti zdánlivě stejnému příkazu „`s/[^\n]*\n//`“, jehož efekt na obsah pracovního prostoru je shodný, ale zpracování skriptu pokračuje následujícím příkazem.

Ve druhém příkazu *substitute* si všimněte, že mezi hranatými závorkami jsme nijak nemuseli ošetřovat znak lomítko, protože uvnitř závorek se nechápe jako oddělovač.

3.19. Editace souboru

Zadání:

Předpokládejte, že ještě máte soubor `datum` obsahující aktuální datum, který jsme vytvořili v kapitole 1.6. Případně si ho znova vyrobte („**`date > datum`**“).

Vložte do souboru editorem první řádku ve tvaru „`Datum:`“.

Rozbor:

Pro řešení tentokrát nepoužijeme editor **`sed`**, nýbrž jeho kolegu, editor **`ed`**. Hlavním důvodem bude to, že pro operaci, kterou se chystáme provést (*editaci souboru*) je určen právě tento editor, zatímco jeho kolega **`sed`** slouží pro editaci *proudu dat*. Kdybychom chtěli použít **`sed`** (samozřejmě by to šlo), museli bychom výsledek editace uložit do pomocného souboru a ten pak přejmenovat (proč nebude fungovat přesměrování „**`sed`** ... **`datum > datum`**“ jsme si už zdůvodňovali).

Potřebujeme před první řádku vložit jinou řádku. Na to bychom v editoru **`sed`** mohli použít příkaz *insert* s adresou „**`1`**“. V editoru **`ed`** to můžeme udělat úplně stejně. Anebo můžeme použít naopak příkaz *append* s adresou „**`0`**“.

Řešení:

```
sinek:~> ed datum
30.....délka souboru při otevření (vypisuje editor)
0a.....příkaz append
Datum:.....vložená řádka
.....ukončení bloku vkládaných řádek
w.....příkaz write
37.....délka souboru při zápisu (vypisuje editor)
q.....příkaz quit
sinek:~>
```

Poznámky:

Dvě čísla, která se nám připletla do výstupu znamenají velikost souboru v bytech (při načtení a uložení souboru). Jejich výpis se dá potlačit přepínačem „**`-s`**“ (*statistics*).

Porovnání základních vlastností obou editorů je v následující tabulce:

	Editor **`sed`**	Editor **`ed`**
Zdroj dat	standardní vstup (příp. soubor zadaný parametrem)	soubor zadaný parametrem
Výsledek	standardní výstup	
Příkazy	v 1. parametru příkazové řádky	ve vstupním proudu

Podstatný rozdíl mezi oběma editory spočívá v tom, že editor **ed** si na začátku práce udělá dočasnou pracovní kopii editovaného souboru a na ní provede postupně všechny příkazy, které má na svém standardním vstupu (každý příkaz právě jednou). Nakonec je nutné výsledek editace uložit pod původním jménem a k tomu slouží příkaz *write* („**w**“). Ten z editoru **sed** neznáme, ale je tam také (dají se pomocí něj vypisovat vybrané řádky do nějakého souboru). Pokud v editoru **ed** na závěrečný příkaz *write* zapomenete, prostě jste si „jen tak zaeditovali pro radost“...

Je třeba mít na paměti že, u extrémně velkých souborů může úvodní vyrábění kopie dost dlouho trvat.

Tím ale podstatné rozdíly mezi oběma editory končí a jinak se už příliš neliší. Jak vidíte, mají identický formát příkazů, mohli jsme použít rovněž stejné zadání čísla řádky v adrese, příkaz stejného jména (*append*) a jenom jsme museli malinko upravit způsob zadání nově vkládaného textu – na konce řádek se nepíše zpětné lomítko, namísto toho se blok nových řádek ukončí zvláštní řádkou obsahující pouze tečku.

Předjímám šťouravou otázku a odpovídám, že v editoru **ed** skutečně není možné pomocí příkazů *append*, *insert* a *change* vložit řádku obsahující pouze samotnou tečku.

I většina dalších příkazů, které jsou v obou editorech, se chová buď zcela identicky (*delete*, *print* či *substitute*), nebo se liší pouze díky odlišné povaze obou editorů (např. *quit* nemá v editoru **ed** adresu – když na něj editor narazí, skončí).

Jsem proto přesvědčen, že je velmi užitečné (s ohledem na minimální úsilí) osvojit si oba editory a vybírat je podle toho, který se pro konkrétní úkol lépe hodí.

Příkazy editoru **ed** je dobré se naučit ještě z jednoho důvodu. Tento editor (ač se to dnes zdá neuvěřitelné) se kdysi opravdu používal pro interaktivní editaci souborů. Dnes už to naštěstí není obvykle nutné, ale může se vám stát, že znalost editoru oceníte. Díky své jednoduchosti editor nezabere moc místa na disku, a právě proto bývá k dispozici i v těch nejextrémnějších podmínkách (např. po různých haváriích systému). A přesně v takových situacích může několik editačních příkazů ušetřit hodiny marné práce při rozbíhání nouzového režimu operačního systému.

3.20. Aktuální řádka

Zadání:

Uložte výstup příkazu **id** rozlámaný do řádek jako v kap. 1.19 do souboru `id.out`.

```
sinek:~> id | tr ' =' '[\n*]' > id.out
sinek:~> cat id.out
...
gid
1004(forst)
...
```

Poté soubor upravte editorem tak, že vymažete řádku s informací o primární skupině (následuje bezprostředně za řádkou „`gid`“).

Rozbor:

Mazat řádku budeme příkazem *delete* („**d**“). Je třeba jenom přijít na to, jak mu zadat správnou řádku.

Podobný problém, kdy bylo třeba od určité řádky přejít k jiné řádce, jsme už řešili. V editoru **ed** je takové „cestování“ jednodušší. V každém okamžiku práce má jedna řádka souboru status *aktuální řádky* a s tímto „kurzorem“ se dá po souboru pohybovat. Na začátku práce se aktuální řádkou stává poslední řádka souboru, přesun na jinou řádku můžeme vyvolat pomocí adres (zadaných například číslem nebo regulárním výrazem podobně jako v editoru **sed**). Navíc je možné k adrese dodat ještě *offset* – posun dopředu („**+***n*“) či dozadu („**-***n*“) o *n* řádek. Po vykonání nějakého příkazu se aktuální řádkou stává (až na výjimky) poslední řádka, s níž příkaz pracoval (u mazání to pochopitelně neplatí).

Použijeme tedy adresu ve tvaru regulárního výrazu („**/gid/**“) a posun o jednu řádku.

Řešení:

```
sinek:~> ed -s id.out
/gid/ +1 d
w
q
```

Poznámky:

Pokud se při hledání řádky odpovídající vzoru narazí na konec souboru, hledání pokračuje znovu od začátku souboru (pochopitelně jen jednou). Proto náš příkaz opravdu najde správnou řádku, přestože po otevření souboru byla aktuální řádka poslední. Pokud se žádná řádka nenajde, příkaz skončí chybou.

Možností, jak zadat adresy, existuje celá řada, určitě stojí za to si pročíst manuálovou stránku editoru (nebo alespoň náš Stručný přehled na str. 328).

3.21. Globalizace

Zadání:

Ze souboru `rfc-index.txt` (viz kap. 3.16) odstraňte záznamy o zneplatněných dokumentech RFC.

Rozbor:

Začneme od mazání. Předpokládejme, že už jsme našli řádku s textem „`Obsoleted`“ a učinili jsme z ní řádku aktuální. Pak bychom potřebovali vymezit rozsah řádek, který chceme mazat. Bude začínat předcházející řádkou s číslem RFC v záhlaví a končit následující prázdnou řádkou. To obojí lze vyjádřit pomocí adres ve tvaru regulárních výrazů (hledání směrem k začátku souboru se vyznačuje uzavřením mezi *otazníky* namísto lomítek). Potřebný příkaz by tedy zněl:

```
?^[0-9]?,/^$/ d
```

Nyní ale stojíme před problémem, jak editor **`ed`** donutit, aby tento příkaz opakoval. U editoru **`sed`** to bylo jednoduché, jeho náplní práce bylo pro každou řádku vstupu provést všechny zadané příkazy. Jenže **`ed`** otevře soubor a provede každý příkaz právě jednou. K opakování příkazu ho musíme nějak vyzvat.

Naštěstí na to **`ed`** má potřebný nástroj, a to příkaz *global* („**`g`**“). Příkaz dostává dva parametry (regulární výraz a příkaz) a zadaný příkaz vykoná na každé řádce, na níž najde zadaný regulární výraz. Aktuální řádkou se postupně stává vždy ta, kterou právě příkaz *global* zpracovává. Náš problém by tedy vyřešila konstrukce:

```
g/Obsoleted/ ?^[0-9]?,/^$/ d
```

Krom toho ještě potřebujeme, aby příkaz *global* „přeskočil“ úvodní část souboru, kde se nacházejí obecné informace, mezi nimiž se také vyskytuje slovo „`Obsoleted`“. Příkaz *global* má jako většina příkazů svoji adresní část, v níž můžeme zadat, na jakém rozsahu řádek chceme příkaz aplikovat.

Řešení:

```
sinek:~> ed -s rfc-index.txt
/^0001/,$ g/Obsoleted/ ?^[0-9]?,/^$/ d
w
q
```

Poznámky:

Celý příkaz přeložený do češtiny by zněl: od řádky začínající „`0001`“ až do konce souboru najdi řádky obsahující „`Obsoleted`“ a pro každou takovou řádku smaž blok řádek počínaje předchozí řádkou začínající číslicí a konče následující prázdnou řádkou.

Přesnější popis je takový, že editor nejprve projde celý soubor (resp. rozsah zadaný adresou příkazu *global*) a označí si řádky, které vyhovují vzoru. Poté postupně každou z nich nastaví jako aktuální a provede zadaný příkaz. Pokud ale provedení příkazu nějak modifikuje některou z řádek, jež by měl příkaz ještě zpracovávat, editor si u této řádky značku odstraní a pak ji z dalšího zpracování vynechá.

Pokud vzoru nevyhovuje žádná řádka, příkaz nemá žádný efekt.

A ještě jedno upřesnění. Ve skutečnosti se jako parametr příkazu může psát nikoliv jen jeden příkaz, ale *seznam* příkazů. V tom případě se za každý (kromě posledního) napíše zpětné lomítko a znak konce řádky jako vyznačení pokračovací řádky. Například následující příkaz přidá na začátek a konec každé nalezené řádky vykřičník:

```
g/Obsoleted/ s/^/!/\
s/$/!/
```

Pokud příkazu *global* nezadáme adresu v adresní části, provádí se implicitně na celém souboru. Asi nejběžnější použití příkazu *global* je nahrazení všech výskytů nějakého řetězce v celém souboru. Nezapomeňte ovšem v takovém případě ještě na parametr **`g`** příkazu *substitute*.

Vedle příkazu *global* existuje ještě příkaz opačný. Už jsme si říkali, že s negováním regulárních výrazů to není vždy jednoduché, takže jeho existence je velmi užitečná. Jmenuje se *invert*, a protože jeho první písmeno je už použito pro příkaz *insert*, zapisuje se jako „**`v`**“ (stejné písmeno ve stejném významu jsme už viděli u přepínače příkazu **`grep`**) a vykonává zadaný příkaz na všech řádkách, které nevyhovují zadanému regulárnímu výrazu.

Rozdíl v chápání adresy je jedním z mála zásadních rozdílů mezi oběma editory a vychází z odlišného režimu jejich práce. Pokusme se to shrnout do tabulky:

	Editor **`ed`**	Editor **`sed`**
opakování příkazu na všech řádkách	**`1,$`** *příkaz*	*příkaz* **`1,$`** *příkaz*
opakování příkazu na řádkách vyhovujících vzoru	**`g/`***vzor***`/`** *příkaz* **`1,$g/`***vzor***`/`** *příkaz*	**`/`***vzor***`/`** *příkaz*
opakování příkazu na řádkách nevyhovujících vzoru	**`v/`***vzor***`/`** *příkaz* **`1,$v/`***vzor***`/`** *příkaz*	**`/`***vzor***`/!`** *příkaz*
provedení příkazu na následující řádce vyhovující vzoru	**`/`***vzor***`/`** *příkaz*	(nedává smysl)

3.22. Horizontální zrcadlo

Zadání:

Vypište řádky souboru `/etc/passwd` pozpátku (první řádku nakonec).

V některých systémech takový program existuje pod jménem **`tac`**.

Rozbor:

Pokud si řešíte domácí úlohy, máte již příklad vyřešen. Ale protože je zařazen v části pojednávající o editorech, očekáváte jistě nějaké řešení pomocí editoru. Bude to opět editor **`ed`**. Protože pracuje s celým souborem *najednou*, jdou v něm některé úlohy řešit snáze než v editoru **`sed`**. Například přesouvání řádek. Když postupně každou řádku souboru přesuneme na začátek, soubor opravdu bude seřazený pozpátku.

Pak ho stačí jen vytisknout. A tentokrát zařídit, aby se změny neuložily.

Řešení:

```
sinek:~> ed -s /etc/passwd
g/^/ m0
1,$p
Q
kernun:*:999:999:Kernun User:/usr/local/kernun:/nologin
forst:*:1004:1004:Libor Forst:/home/forst:/bin/bash
... atd.
root:*:0:0:Super User:/root:/bin/bash
```

Poznámky:

Příkaz *global* jsme už poznali minule, tentokrát jsme vybrali takový regulární výraz, který jistě bude na každé řádce.

Příkaz *move* („**`m`**“) přesouvá řádky, které má zadané v adresním rozsahu, za řádku zadanou parametrem. My jsme použili jako parametr adresu „**`0`**“, přesouváme tedy za nultou řádku neboli na začátek souboru.

Příkaz *print* („**`p`**“) známe z editoru **`sed`**. Vypisuje text řádek, zadali jsme mu, že je má vypisovat „od začátku do konce“ (neboli „**`1,$`**“).

Příkaz *Quit* říká editoru, že chceme ukončit práci. Na rozdíl od minulých kapitol ale nechceme změněný text ukládat, proto je nutné použít velké „**`Q`**“.

Povšimněte si různého chování různých příkazů při vynechání adresní části.

- Příkaz *quit* (resp. *Quit*) adresní část vůbec *nepoužívá* (pozor, v editoru **`sed`** to neplatí), editor prostě skončí.
- Příkaz *global* adresní část sice může mít, ale pokud ji vynecháme, provádí se *na celém* souboru. Stejně se chová také *write*.

- Příkaz *move* adresní část používá, a když ji vynecháme, provede se pouze pro *aktuální* řádku. Toto je nejběžnější implicitní hodnota adresy, platí pro většinu příkazů. V našem případě – díky použití v rámci příkazu *global* – to například bude znamenat, že se příkaz provede pro každou jednotlivou řádku, kterou *global* vybere.
- Pro úplnost dodejme, že existují některé příkazy, u nichž je implicitní hodnotou *poslední* řádka souboru.

Uvědomte si rozdíl mezi naším použitím příkazu *move* a *print*. U obou jsme chtěli, aby se provedly pro každou řádku. Proč jsme tedy oba volali jinak? Bylo by možné je zadat obráceně? Ano, ale chování by bylo odlišné. Zápis „**`g/^/`**...“ znamená „proveď pro každou řádku jednotlivě“, zatímco zápis „**`1,$`**...“ znamená „proveď pro všechny řádky najednou“. Například u tisku je to úplně jedno, příkaz „**`g/^/p`**“ by fungoval sice zbytečně složitě, ale jinak k naší plné spokojenosti. Zato příkaz „**`1,$m0`**“ by přesunul všechny řádky najednou a nikoliv po jedné, takže jeho efekt by byl nulový.

Je důležité si rovněž uvědomit to, že vzhledem k potřebě vytvoření dočasné kopie souboru a neustálému přesouvání řádek nebude naše implementace z nejrychlejších. Na malých souborech to nepoznáme, ale pokud budeme takto obracet velký soubor, bude řešení pomocí příkazu **`sort`** řádově rychlejší.

Kdybychom tuto úlohu řešili častěji, vyplatí se nám uložit si příkazy pro **`ed`** do zvláštního souboru (*skriptu*, nebo *ed-skriptu*) a vyvolávat **`ed`** s přesměrovaným vstupem.

```
sinek:~> ed -s /etc/passwd < název_skriptu
```

Takový skript si vytvoříme v následující kapitole.

Perlička na okraj: Název programu **`grep`**, který už dlouho používáme, ve vás možná vyvolává představu, že se jedná o nějaké anglické sloveso. Slangově se toto slovo dnes navíc doopravdy jako sloveso používá (a to i přeneseně v češtině). Ale jeho název je odvozen z příkazu editoru `ed`: „**`g/`***re*`/p`“, neboli „*global regular expression print*“...

3.23. Mašinka

Zadání:

Napište skript pro editor **ed**, který upraví normální výstup příkazu **diff** na tvar, který by vypsalo volání s přepínačem „**-e**“.

Příkaz **diff** jsme už poznali, porovnává dva soubory a vypisuje řádky, ve kterých se oba soubory liší. Má několik režimů výstupu. Normální výstup vypadá asi takto:

```
sinek:~> cat old
o-1
o-2
o-3
o-4
o-5
o-6
sinek:~> cat new
o-3
n-x
n-y
o-6
n-z
sinek:~> diff old new
1,2d0     ... ř. 1 až 2 ze souboru old v souboru new chybí, patřily by za ř. 0
< o-1           ... řádka číslo 1 ze souboru old
< o-2           ... řádka číslo 2 ze souboru old
4,5c2,3   ... ř. 4 až 5 z old se liší od odpovídajících ř. 2 až 3 z new
< o-4           ... řádka číslo 4 ze souboru old
< o-5           ... řádka číslo 5 ze souboru old
---       ... oddělovací řádka výpisu
> n-x           ... řádka číslo 2 ze souboru new
> n-y           ... řádka číslo 3 ze souboru new
6a5       ... ř. 5 ze souboru new v souboru old chybí, patřila by za ř. 6
> n-z           ... řádka číslo 5 ze souboru new
```

Výstup se skládá z bloků, které popisují jednotlivé nalezené odlišnosti. Blok začíná řádkou popisující typ změny (jsou to vlastně názvy příkazů editoru **ed**: *delete*, *change* a *append*) a rozsahy čísel řádek (v jednom, resp. druhém souboru), jichž se změna týká. Zbytek bloku obsahuje lišící se řádky obou souborů, směr zobáčku přitom udává, ze kterého souboru konkrétní řádka pochází. Při zpracování se dá dobře využít toho, že každý typ řádky výstupu se rozpozná podle prvního znaku.

Pokud **diff** zavoláme s přepínačem „**-e**“, vypíše sadu příkazů pro editor **ed** – takovou, že když ji aplikujeme na první soubor, výsledkem bude soubor druhý. Je to taková „mašinka“, která se může například opakovaně spouštět na konverzi souborů z jednoho na druhý.

V našem případě by výstup vypadal:

```
sinek:~> diff -e old new
6a
n-z
.
4,5c
n-x
n-y
.
1,2d
```

Rozbor:

Zásadním problémem, s nímž je třeba se při řešení úlohy vypořádat, je to, že příkazy musejí být napsány „pozpátku“ – změny je třeba provádět v pořadí od konce souboru směrem k začátku. Provedení každého příkazu v editoru totiž *změní* čísla řádek od místa editace do konce souboru. Pokud bychom tedy prováděli příkazy popořádku (jako jsou zapsané ve výstupu **diff**), opravovali bychom (s výjimkou prvního příkazu) úplně jiné řádky. Když budeme příkazy provádět „odzadu“, budou se měnit čísla řádek pouze v té části souboru, do níž už zasahovat nebudeme.

Ostatní změny formátu výpisu (úprava znění příkazů, vymazání zobáčků a přidání tečky za vkládaný blok řádek, jak to editor **ed** vyžaduje) jsou už podstatně jednodušší.

Využijeme myšlenku „zrcadla“, která nám pomohla i v minulé kapitole. Jediný rozdíl bude v tom, že nebudeme přesouvat jednotlivé řádky, ale celé bloky řádek (aby uvnitř bloku zůstaly řádky ve správném pořadí).

Napřed si rozmyslíme, pomocí čeho budeme vymezovat blok řádek, který se má přesouvat. Úvodní řádka každého bloku je jasná – je to řádka s čísly řádek a příkazem. Horší je to s určením koncové řádky. Tady by nám výrazně pomohlo, kdybychom si napřed přidali za každý blok řádek nějakou naši vlastní oddělující řádku – pak bychom vymezení intervalu řádek, jež se mají přesouvat, udělali triviálně. Tak si tam takovou zarážku přidejme! Její přesný tvar není tak moc důležitý, jenom se musí snadno odlišit od ostatních.

Jenže úkol „přidej za blok řádek“ má pořád stejný háček – nepoznáme konec bloku. Podstatně lehčí by bylo přidat novou řádku před každý blok (úvodní řádku bloku poznáme, začíná přece číslicí). Tím by se ovšem zarážka ve skutečnosti objevila (skoro)

za každým blokem. Chyběla by pouze jedna na konci souboru a na začátku by byla jedna navíc. Pak by ale stačilo prostě tuto první řádku přesunout na konec:

```
g/^[0-9]/ i\
zarážka\
.
1m$
```

Všimněte si, že jsme u příkazu *insert*, který je parametrem příkazu *global*, museli použít pokračovací řádky, jak jsme se o tom zmiňovali v kapitole 3.21.

Teď už můžeme provést náš známý „kolotoč“:

```
g/^[0-9]/ .,/^zarážka/ m0
```

Příkaz *move* tentokrát má i adresní část, která říká, že se bude přesouvat blok řádek počínaje aktuální řádkou (tedy tou, co *global* vybral) a konče následující řádkou s naší zarážkou. Aktuální řádka se v adrese zapisuje jako tečka.

Řádky souboru máme nyní správně uspořádané a zbývá jen „učesat“ příkazy v nich uložené. Nejprve vymažeme čísla řádek (druhého souboru) na konci řádek s příkazy:

```
g/^[0-9]/ s/[0-9,]*$//
```

Poté smažeme dělící řádky „`---`“ a řádky se „zobáčky“ otočenými na špatnou stranu:

```
g/^[-<]/  d
```

Potom musíme nějak obhospodařit naše zarážky. Úmyslně neříkám „smazat“ – hned uvidíte, proč. Za příkazy *append* a *change* máme totiž připravené bloky vkládaných řádek, které musíme pokaždé ukončit řádkou se samostatnou tečkou. Tu bychom museli za blok řádek zase nějak složitě přidávat. Ale my už tam jednu řádku „navíc“ máme – naši zarážku. Což kdyby tedy měla právě potřebný tvar „`.`“?

Myšlenka je to výborná, ale má jeden háček. Vzhledem k tomu, že naši oddělovací řádku vkládáme i my sami příkazem *insert*, nemůže obsahovat samotnou tečku! To by ji *insert* považoval za konec bloku vkládaných řádek, něco by si o nás pomyslel a nevložil by nic. Musíme proto vymyslet pro zarážku jiný text, ale takový, aby se dal později na samostatnou tečku snadno převést.

Opravdu smazat musíme pouze takové zarážky, jež následují za příkazy *delete* – tam nemají co pohledávat. Najdeme příkazy *delete* a řádky následující za nimi smažeme:

```
g/^[0-9,]*d/ +1 d
```

Poslední úpravou je umazání „zobáčků“ ze začátků řádek. Vlastně není poslední, ještě pořád nemáme naše zarážky upravené do podoby samostatné tečky. Kdybychom ale tvar zarážek zvolili nějak šikovně (třeba „`=  .`“), mohli bychom obě operace spojit a mazání zobáčků udělat najednou s mazáním rovnítek:

```
g/^[=>] / s///
```

Řešení:

```
g/^[0-9]/ i\
= .\
.
1m$
g/^[0-9]/ .,/^=/ m0
g/^[0-9]/ s/[0-9,]*$//
g/^[-<]/  d
g/^[0-9,]*d/ +1 d
g/^[=>] / s///
w
```

Poznámky:

Poslední příkaz *substitute* měl uvedený prázdný regulární výraz. V takovém případě jako vzor chápe poslední použitý regulární výraz – zde konkrétně ten z příkazu *global*. Význam je tedy stejný jako by měl příkaz:

```
g/^[=>] / s/^[=>] //
```

Pokud by se porovnávané soubory lišily v řádce obsahující pouze tečku, nedá se oprava provést příkazem *append* nebo *change*. Je třeba vložit řádku s nějakým jiným textem a posléze ji modifikovat. Ostatně my sami jsme to tak museli v našem programu udělat. Náš program tedy nezvládne svůj úkol, pokud se tento případ vyskytne. Ale asi mu to odpustíme, protože už tak je dost složitý a potřebná úprava by ho zkomplikovala natolik, že by to přestalo být únosné z didaktického hlediska.

Závěrečný příkaz *quit* ve skriptu není zapotřebí, s koncem dat editor skončí sám.

Ladění skriptů pro editory není procházka růžovou zahradou. Zkuste si pro začátek za každý příkaz vložit příkaz „`1,$p`“, ať vidíte, jak se soubor postupně mění.

Další náměty na procvičení

1. V úloze z kapitoly 2.16 vylepšete regulární výrazy tak, aby řetězec „`0`“ vyhledávaly opravdu jen ve sloupci s číslem skupiny.
2. Upravte výstup příkazu „**`wc -l /etc/passwd`**“ tak, aby z něj zbyl jen údaj o počtu řádek (odstraňte jméno souboru).
3. V úloze z kapitoly 1.18 vyřešte zpracování výstupu příkazu **`id`** pomocí editoru **`sed`**.
4. První problém řešený v kapitole 3.10 (vypuštění části cesty obsahující samotnou tečku, resp. posloupnost takových částí) se dá vyřešit i bez cyklu za použití příkazu *substitute* s parametrem „**`g`**“. Zkuste takový příkaz vymyslet.
5. V řešení příkladu v kapitole 3.13 jsme zanedbali možnost výskytu znaků „`<`“, „`>`“ a „`&`“ v původním textu, které se do textu HTML stránky nesmějí dostat a které je třeba nahradit odpovídajícími posloupnostmi znaků „`<`“, „`>`“, resp. „`&`“. Doplňte ošetření těchto znaků.
6. V kapitole 3.16 jsme zpracovávali index dokumentů RFC. Vypište z indexu plný text záznamů popisující RFC s „kulatými“ čísly 1000, 2000, ...
   Návod: O práci s bloky řádek jsme hovořili v kapitole 3.9.
7. Vyřešte úlohu z kapitoly 3.19 pomocí editoru **`sed`**.
8. Tabulka se v jazyce HTML vytváří pomocí značek „`<table>...</table>`“, každá řádka je uzavřena mezi značky „`<tr>...</tr>`“ a každé políčko vyznačují značky „`<td>...</td>`“. Napište program pro převod obsahu `/etc/passwd` do tabulky ve formátu HTML.
9. Vyřešte úlohu z kapitoly 4.10 pomocí editoru **`sed`**, resp. **`ed`**.
10. Podívejte se na způsob zápisu řetězců v jazyce **`awk`** (nebo C) a napište skript, který ze zdrojového textu vypíše všechny řetězce, každý na samostatnou řádku. Dejte pozor na escape-sekvence (hlavně „**`\"`**“ a „**`\\`**“) uvnitř řetězců. Můžete předpokládat, že uvozovky se mimo řetězce nikde nevyskytují.
11. Zkuste napsat posloupnost příkazů editoru **`sed`**, která emuluje (nahrazuje) činnost příkazu *Next*.

12. Napište skript pro editor, který bude kontrolovat, zda je obsah souboru správně uzávorkovaný (tj. že za každou levou závorkou následuje někde v textu její párová pravá závorka a naopak a že se páry závorek nekříží). Pokud najde chybu, vypíše začátek řádky až po nesprávnou závorku. Jako vstup můžete použít (vhodně „pokažené“) hlavičkové soubory z adresáře **`/usr/include`**.

13. Vyřešte úlohu z kapitoly 3.13 způsobem naznačeným v úvodu rozboru, tj. nějak se „poperte“ s problémem posloupností prázdných řádek.

14. Zkraťte v textu na standardním vstupu řádky na délku 80 znaků. Pokud byste přitom měli roztrhnout slovo, vymažte celé slovo.
    Návod: Použijte trik z úlohy v kapitole 3.17.

15. Napište skript, který přeformátuje odstavce ve volném textu (zadaném stejně jako v kap. 3.13) tak, aby maximální délka řádky nepřesáhla 80 znaků. Řádky můžete dělit v místech, kde se nacházejí mezery.

16. Naprogramujte příklad z kapitoly 1.17 pomocí víceřádkového pracovního prostoru v editoru **`sed`**. Pozor na záludnost příkazu *Next*, na kterou jsme narazili v kap. 3.15.

Část 4 Shell

K podrobnějšímu výkladu vlastností a funkcí jádra celé problematiky – shellu – jsme se (možná trochu překvapivě) propracovali až po delší dělostřelecké přípravě. Ale potřebovali jsme nejprve trochu poznat nástroje, které budeme pomocí shellu ovládat. Čtvrtá, nejrozsáhlejší část knihy navazuje na útržkovité informace o shellu, které jsme potřebovali už v předchozích částech, třídí je a doplňuje.

4.1. Expanzní znaky

Zadání:

Zkopírujte z adresáře `/usr/include` do svého domovského adresáře všechny hlavičkové soubory (s příponou „`h`“), v jejichž jméně se vyskytuje číslice.

Rozbor:

Použijeme příkaz **`cp`** a zadáme mu seznam jmen kopírovaných souborů pomocí vzoru z expanzních znaků shellu. Hvězdičku už známe, naučíme se, jak zadat číslici. Používá se stejný konstrukt jako v regulárních výrazech – *hranatá závorka*, resp. *pár hranatých závorek*. Jeho význam je stejný, stejně se zadává seznam znaků, interval, třídy znaků. Jediný rozdíl je v zadání doplňku znaků – místo stříšky se používá vykřičník.

Řešení:

```
sinek:~> cp /usr/include/*[0-9]*.h .
```

Poznámky:

Nezapomeňte přehodit výhybku! Zatímco v předešlé části jsme pitvali regulární výrazy, teď se zase budeme více věnovat porovnávacím vzorům a expanzním znakům v shellu. Oba jazyky pro popisování řetězců se nedají zaměňovat! Editory (a některé další programy) používají pouze regulární výrazy, shell (a třeba také **`find`**) zase naopak porovnávací vzory. Nepříjemné je to, že metaznaky obou jazyků se prolínají (hranaté závorky jsou v obou shodné, hvězdička je v obou, ale znamená pokaždé něco jiného). A podstatný rozdíl je i v tom, že regulární výrazy implicitně popisují shodu vzoru s podřetězcem (shodu na celé délce řetězce musíme explicitně vynutit ukotvením pomocí znaků „**`^`**“ a „**`$`**“), zatímco porovnávací vzory popisují shodu na celé délce (chceme-li testovat pouze podřetězec, musíme okolo vzoru přidat hvězdičky).

Posledním expanzním znakem, který jsme zatím neprobírali, je otazník. Zastupuje právě jeden *libovolný* znak (tj. totéž, co v regulárních výrazech znamená tečka).

Připomeňme, že expanzi (nahrazení porovnávacího vzoru za seznam vhodných jmen souborů) provádí shell sám a k prováděným programům (zde **`cp`**) se namísto jednoho parametru (vzoru) dostane tolik parametrů, kolik souborů vhodného jména existuje.

Kdybyste zapomněli v našem příkazu napsat poslední parametr (cílový adresář „**`.`**“), příkazu **`cp`** by se to nejspíše nelíbilo. Zlobil by se, že poslední parametr není adresář. Ovšem nejhorší případ by nastal, kdyby existovaly právě dva soubory vyhovující vzoru – program **`cp`** by pak dostal dva parametry a druhý soubor by přepsal prvním.

Důležité je rovněž to, že shell jména souborů *abecedně třídí*. Máme-li v adresáři soubory „`a`“ a „`b`“, pak vzor „**`[ba]`**“ na příkazové řádce bude nahrazen za „**`a b`**“...

4.2. Programování „on-line“

Zadání:

Vytvořte kopie všech hlavičkových souborů z aktuálního adresáře – jména pro kopírované soubory vytvořte tak, že příponu „`h`“ nahradíte příponou „`bak`“.

Rozbor:

V systému MS DOS (resp. Windows) bychom zálohování odbyli jediným příkazem:

```
copy *.h *.bak
```

Je to jednoduché a elegantní, ale nechtěl bych být v kůži toho, kdo má vysvětlovat, proč to vlastně dělá to, co to dělá. V UNIXu takový konstrukt fungovat nebude. Hvězdičky (jak už víme) nahrazuje shell, ale žádný soubor se jménem „`*.bak`“ neexistuje, takže shell žádnou expanzi neprovede a poslední parametr, který dostane příkaz **`cp`**, bude právě „**`*.bak`**“. A příkaz buď vyrobí soubor se jménem „`*.bak`“ (pokud bude existovat jediný soubor s příponou „`h`“), nebo úplně selže (pokud jich bude více – už jsme si říkali, že příkaz **`cp`** smí mít více parametrů jen tehdy, pokud posledním z nich je jméno existujícího adresáře).

Takže jinak. Pro kopírování každého jednotlivého souboru budeme potřebovat jeden samostatný příkaz. Na jejich vytvoření asi nejlépe poslouží editor **`sed`**. Jako podklad použijeme seznam souborů, který získáme příkazem **`ls`**. Ze jména souboru si oddělíme příponu a poskládáme potřebný text příkazu („**`cp -p`** *soubor*`.h` *soubor*`.bak`“).

Vypsat vygenerované příkazy už tedy umíme. Zbývá vyřešit, jak je spustit. Tady je nejsnazší řešení překvapivě jednoduché – příkazy prostě pošleme na vstup nové instanci shellu. Shell přece čte příkazy ze svého vstupu a provádí je. A přesně to chceme!

Řešení:

```
sinek:~> ls *.h | sed 's/\(.*\)\.h/cp -p & \1.bak/' | sh
```

Poznámky:

Výroba potřebných příkazů „on-line“ je v shellu velmi oblíbená metoda. Díky tomu, že je to jazyk interpretovaný, dají se v tomto ohledu s kódem shellových skriptů dělat velká kouzla. Jen je třeba znát míru. Náš příklad ji jistě nepřekračuje.

Pro spuštění příkazů vytvořených „na míru“ jsme už poznali jinou metodu (příkaz **`xargs`**) a další poznáme. Je jen otázkou citu a zkušenosti vybrat metodu, která bude nejjednodušší a nejefektivnější.

Přepínač „**`-p`**“ (*preserve*) příkazu **`cp`** zařídí, že kopie budou mít stejné datum a čas poslední modifikace (jinak jim **`cp`** nastaví aktuální čas).

4.3. Použití proměnné

Zadání:

Vypište obsah domovského adresáře nezávisle na aktuálním pracovním adresáři.

Rozbor:

Použijeme samozřejmě opět příkaz **ls**, ale potřebujeme mu jako parametr zadat jméno svého domovského adresáře. Naštěstí mezi *systémovými proměnnými*, které tvoří tzv. *prostředí* (environment) běžícího programu a které systém předvyplní potřebnými hodnotami, najdeme proměnnou **HOME**, jež obsahuje jméno domovského adresáře.

Stačí tedy shellu říci, že hodnotu této proměnné má použít jako součást příkazové řádky. A to se dělá pomocí dalšího metaznaku, *dolaru*.

Řešení:

```
sinek:~> ls $HOME
```

Poznámky:

Krom jiných funkcí je shell také *textový preprocesor*, upravuje text příkazové řádky podle pevných pravidel a nakonec se řádku pokusí interpretovat jako příkaz. Proto musíme v textu řádky nějak odlišit části, kterých si shell nemá všímat (zde třeba název příkazu), od částí, jež naopak potřebujeme změnit (zde nahrazení názvu proměnné její hodnotou). A právě k tomu slouží *metaznaky*. Ty říkají shellu: „na tomto místě je pro tebe nějaká práce“. Jakmile shell metaznak nalezne, provede patřičnou úpravu. Na řádce:

```
sinek:~> ls $HOME/*[0-9]* > $HOME/seznam
```

shell vypíše (*expanduje*) obsah proměnné (dvakrát) a pokračuje v práci (bude přitom respektovat i další metaznaky, hvězdičku, hranaté závorky a většítko).

Zkušenější namítnou, že snazší způsob řešení je použití jiného metaznaku, a to vlnky:

```
sinek:~> ls ~
```

Určitě souhlasím a souhlasí i norma. Ovšem v praxi bude záležet na tom, jaký typ a verzi operačního systému a především typ a verzi shellu budete mít k dispozici.

Shell totiž není vestavěná součást operačního systému, je to zkrátka jen program dodávaný s operačním systémem a v principu si kdokoli může udělat vlastní shell. To je dobrá zpráva. Bohužel to spoustu lidí napadlo a vlastní shell si vyrobili. A to je špatná zpráva. Dnes existuje celá řada shellů a snadno se vám může stát, že to, co odladíte na svém počítači, jinde fungovat nebude. První shell vymyslel pan Bourne a norma ctí tyto základy, avšak přidává k nim některá rozšíření, která už zdaleka nemusíte najít všude. A s rozšířeními dnes asi nejpopulárnějšího shellu **bash** (*Bourne Again Shell*) narazíte ještě častěji. My se v celé knize budeme se snažit vyhnout všem vylepšením, jež by mohla ohrozit přenositelnost našich programů.

4.4. Definice proměnné

Zadání:

Systémová proměnná **PATH** obsahuje dvojtečkami oddělený seznam adresářů (např. „`/bin:/usr/bin:…`“). Přidejte k jejímu obsahu ještě adresář „`.`“.

Rozbor:

V minulé kapitole jsme si ukázali použití proměnné. Teď budeme potřebovat do proměnné také přiřadit novou hodnotu. To lze velmi jednoduše a intuitivně – napíšeme jméno proměnné, rovnítko a hodnotu.

Oproti jazykům, které nejsou textově orientované, nepotřebujeme žádný konstrukt na zřetězení staré hodnoty proměnné a nového řetězce. Shell prostě očekává za rovnítkem text a na tom, jakou má podobu, mu nezáleží. Nahradí hodnotu proměnné a text dosadí.

Řešení:

```
sinek:~> PATH=$PATH:.
```

Poznámky:

Na první pohled vypadá tento příkaz docela přirozeně. Uvědomte si ale, že je to úplně *nový typ* příkazu, který jsme zatím nepotkali. Doposud řádka začínala názvem příkazu, zde začíná identifikátorem proměnné (bez dolaru – nechceme od shellu provést expanzi!) a rovnítkem. Tak shell pozná, že se jedná o přiřazení hodnoty proměnné.

Pokud jste z jiných jazyků zvyklí dělat kolem rovnítka mezery, tak tady to nepůjde. Přečtěte si ještě jednou předchozí odstavec a rozmyslete si, co by znamenal příkaz

```
sinek:~> PATH = $PATH:.
```

(shell by se pokusil zavolat program **PATH** s dvěma parametry: „**=**“ a „`/bin:…:.`“).

V adresářích uvedených v hodnotě proměnné **PATH** hledá shell programy (spustitelné soubory), pokud na příkazové řádce napíšeme pouze název programu (bez zadání cesty). Napíšeme-li například příkaz „**ls**“, shell jen díky správnému nastavení proměnné **PATH** najde, že potřebný program se nachází v souboru „`/bin/ls`“. Tím, že jsme do obsahu proměnné přidali tečku (aktuální adresář), půjdou bez udání cesty spouštět i programy uložené v aktuálním adresáři. Třeba náš příkaz z kapitoly 2.2 by šel nyní zavolat jako:

```
sinek:~> rename a b
```

Tečku jsme schválně přidali na konec seznamu, protože shell hledání skončí u prvního nalezeného programu – kdyby byla tečka na začátku, v některých adresářích by se mohly dít neočekávané a třeba i nebezpečné věci...

Nastavení (jakékoliv) proměnné se odrazí pouze v aktuálně běžící instanci shellu. Až se přihlásíte příště, bude mít proměnná **PATH** zase implicitní hodnotu.

4.5. Jak namalovat „obrázek“

Zadání:

Uložte do proměnné `ctverec` takový řetězec, aby se při vypsání hodnoty proměnné příkazem **`echo`** objevil „obrázek“:

```
+--+
|  |
+--+
```

Rozbor:

Potřebujeme do proměnné přiřadit řetězec obsahující znaky, které se asi shellu nebudou moc líbit. Přinejmenším to jistě víme o znaku konce řádky nebo o mezeře. To už jsme se ale naučili obcházet pomocí apostrofů.

Řešení:

```
sinek:~> ctverec='+--+
> |  |
> +--+'
```

Poznámky:

Pokud jste dospěli až sem, asi budete chtít také vidět důkaz, že jste úlohu vyřešili správně. Jak vypsat obsah proměnné, napovědělo zadání:

```
sinek:~> echo $ctverec
+--+ | | +--+
```

Tohle tedy jako čtverec rozhodně nevypadá! Co se přihodilo? Je špatně nastaven obsah proměnné anebo ji špatně vypisujeme? Správná odpověď je b). Podívejme se, co se přesně odehrálo (tak, jak shell zpracovává řádku v jednotlivých krocích):

1. Shell přečetl příkazovou řádku a našel na ní dvě slova, „**`echo`**“ a „**`$ctverec`**“. *Slovem* (nebo také „polem“) zde přitom míníme řetězec znaků neobsahující mezeru ani tabulátor (a také středník, svislítko, větší a některé další metaznaky shellu – ty, které nazýváme *operátory*).

2. V prvním slově shell nenalezl žádné metaznaky, nechal ho tedy beze změny. Ve druhém slově našel metaznak dolar a za ním název proměnné. Našel tedy v paměti hodnotu proměnné a původní řetězec „**`$ctverec`**“ nahradil touto novou hodnotou – řetězcem „**`+--+`***(znak konce řádky)***`|  |`***(znak konce řádky)***`+--+`**“. Tomuto kroku se říká *expanze* (rozvinutí) nebo také *substituce* (dosazení) hodnoty proměnné a časem poznáme ještě několik jiných druhů substitucí.

3. V dalším kroku se shell podívá na obsah jednotlivých slov na řádce a hledá, zda se v textu, který on sám do řádky dosazoval, neobjevil úsek, který by bylo třeba ještě dále rozdělit na *slova* (pole). V tomto kroku se ale namísto implicitních oddělovačů polí (bílých znaků) používají znaky uložené v systémové proměnné **`IFS`** (*internal field separator*). Implicitně tam sice jsou bílé znaky (mezera, tabulátor a znak konce řádky), ale uživatel si může obsah proměnné libovolně přizpůsobit.

A právě tento krok je zdrojem našich problémů. Po jeho skončení totiž zůstane na příkazové řádce pět slov: „**`echo`**“, „**`+--+`**“, „**`|`**“, „**`|`**“ a „**`+--+`**“, zatímco všechny jejich oddělovače (posloupnosti mezer a znaků konce řádky) už dávno zmizely v propadlišti dějin. Shell tedy bude nakonec volat příkaz **`echo`**, předá mu zbylé čtyři parametry a **`echo`** je jen opíše, přičemž mezi ně vloží po jedné mezeře (jak to má v popisu práce).

Pokud chceme vidět všechny znaky, co v proměnné byly, musíme zabránit shellu, aby provedl ono rozdělení na pole. Musíme mu říct, že celá hodnota proměnné je jedno jediné pole. Možná vás napadlo zkusit znak, který už jsme poznali, apostrof:

```
sinek:~> echo '$ctverec'
$ctverec
```

To je vcelku logická úvaha, háček je v tom, že mezi apostrofy ztrácejí svůj zvláštní význam všechny metaznaky, a tedy i dolar. Proto nám shell obsah proměnné vůbec nedosadí. Potřebovali bychom takové nějaké „polovičaté“ apostrofy. A ony opravdu existují. Akorát nejsou „polovičaté“, ale naopak „dvojnásobné“ – jsou to uvozovky. Mezi nimi totiž ztrácejí zvláštní význam všechny metaznaky s výjimkou dolaru (a ještě několika dalších metaznaků), proměnné se tedy v uvozovkách normálně rozvinou.

Metody skrývání metaznaků před očima shellu (*quoting*) jsou ve skutečnosti tři:

- Prefixování *zpětným lomítkem*.
  Tuto metodu lze použít na všechny metaznaky a obvykle se používá, když je jich na řádce málo – bezprostředně před každý metaznak přidáme zpětné lomítko.
- Uzavření mezi *apostrofy*.
  Platí rovněž na všechny metaznaky (kromě koncového apostrofu).
- Uzavření mezi *uvozovky*. V tomto případě zůstává metaznakem:
  - uvozovka (je zapotřebí jako ukončovací znak);
  - dolar;
  - zpětný apostrof (ten poznáme v další kapitole);
  - zpětné lomítko (prefixují se s ním metaznaky uvedené v tomto seznamu, pokud je chceme zapsat mezi uvozovkami v jejich původním významu).

Správný tvar výpisu obsahu proměnné tedy je:

```
sinek:~> echo "$ctverec"
+--+
|  |
+--+
```

Jinou možností je vypsat obsah (všech) proměnných příkazem **set** (bez parametrů).

Pro pořádek ještě dokončeme popis práce shellu:

4. Poté, co je řádka definitivně rozdělena na pole, shell každé pole prohlédne, zda neobsahuje *expanzní znaky* porovnávacích vzorů (hvězdičku, otazník a hranatou závorku), a pokud ano, nahradí toto pole tolika poli s názvy souborů, kolik jich našel.

5. Závěrečným krokem je *zrušení quotingu*. Shell odstraní všechna zpětná lomítka, apostrofy a uvozovky, takže opět vidí všechny znaky, které jsme před jeho zraky ukryli, ale v jejich „normálním“ významu, nikoliv jako metaznaky.

6. Tím práce shellu s řádkou končí. Nyní se shell podívá, jaký příkaz se na ní nachází, a vykoná ho.

Kroky 3 a 4 se neprovádějí na řetězcích přiřazovaných do proměnných. Pokud do proměnné `hvezdicka` uložíme hvězdičku („**hvezdicka=***“), bude tam opravdu hvězdička a ne seznam souborů (příkazem „**echo $hvezdicka**“ to ale nepoznáme...).

Vzhledem k tomu, že shell je textový preprocesor, proměnné v shellu mají vždy textovou hodnotu, shell nezná pojem *datového typu* proměnné.

Pokud použijeme proměnnou, jejíž hodnota nebyla nastavena, shell dosadí prázdný řetězec. Můžete si představit, že proměnné jsou prázdným řetězcem inicializovány. Ve skutečnosti to tak není, ale z hlediska běžného používání je v tom nepodstatný rozdíl.

Proměnnou jsme nazvali jménem složeným z písmen malé abecedy. Není to náhoda. Podobně jako u jmen souborů, i u jmen proměnných se v UNIXu rozlišují malá a velká písmena. A ve jménech systémových proměnných se zásadně používají velká písmena. Navyknete-li si používat v názvech svých proměnných naopak písmena malá, vyvarujete se nepříjemných zklamání. Z vlastní zkušenosti mohu potvrdit, že když si pro svoji proměnnou obsahující nějakou cestu, zvolíte náhodou jméno **PATH**, budete hezkou chvíli nechápavě kroutit hlavou, proč váš program náhle přestane dělat cokoliv rozumného...

Kromě písmen je ve jméně proměnné možno použít i další *alfanumerické* znaky: podtržítko („_“) a číslice (číslicemi ale název nesmí začínat).

4.6. Proměnlivá řádka

Zadání:

Přesuňte v hodnotě proměnné **PATH** poslední jméno adresáře na první místo.

Rozbor:

Provést rotaci adresářů a vypsat výsledek na terminál zvládneme snadno v editoru:

```
sinek:~> echo $PATH | sed 's/\(.*\):\(.*\)/\2:\1/'
```

Problém ale nastane v okamžiku, kdy budeme chtít dosadit tento text do proměnné. Až dosud jsme do proměnných přiřazovali hodnoty, které jsme už někde měli k dispozici (řetězce a hodnoty jiných proměnných). Teď ale potřebujeme přiřadit něco, co vznikne teprve v průběhu práce shellu. Pokud se pokusíte napsat třeba:

```
sinek:~> PATH=echo $PATH | sed 's/\(.*\):\(.*\)/\2:\1/'
```

shell slovo „echo“ pochopí jenom jako text, který má uložit do proměnné, namísto toho, aby příkaz **echo** spustil. O zavolání příkazu ho musíme explicitně požádat. Pro tento účel slouží další metaznak shellu, zpětný apostrof („`“). Shell vezme řetězec uzavřený mezi pár zpětných apostrofů jako text příkazu, spustí ho a obsah jím vygenerovaného standardního výstupního proudu vloží zpět do příkazové řádky. Pak pokračuje v jejím dalším zpracování. Nám tedy bude stačit pečlivě vymyšlený příkaz uzavřít mezi zpětné apostrofy a zapsat do přiřazovacího příkazu.

Řešení:

```
sinek:~> PATH=`echo $PATH | sed 's/\(.*\):\(.*\)/\2:\1/'`
```

Poznámky:

Tento trik (tzv. *náhrada* nebo *substituce příkazu*) je jedním ze základů programování v shellu! Naučili jsme se ovládat celou řadu nástrojů. Ale pokud jejich výsledky chceme ve skriptech dál používat, musíme je umět přinejmenším ukládat do proměnných.

V novějších shellech (a v normě) najdeme pro tuto funkci ještě jeden konstrukt:

```
sinek:~> PATH=$(echo $PATH | sed 's/\(.*\):\(.*\)/\2:\1/')
```

Jeho význam je zcela stejný a jeho zřejmou výhodou je logičtější vnořování. Představte si, že v příkazu, který chcete mezi zpětné apostrofy napsat, potřebujete také použít zpětné apostrofy. Ze zápisu

```
sinek:~> vypis=`ls `pwd`/bin`
```

není jasné, že se má nejprve spustit příkaz **pwd**. Tedy nám ano, ale shellu ne! Shell k sobě sdruží první pár zpětných apostrofů a zavolá příkaz **ls** bez parametrů, pak sdruží druhý pár a pokusí se zavolat program **/bin**. Chceme-li zapsat vnořené volání, musíme vnitřní zpětné apostrofy pro první okamžik před shellem ukrýt pomocí zpětného lomítka:

```
sinek:~> vypis=`ls \`pwd\`/bin`
```

Logickým důsledkem je, že rovněž samo zpětné lomítko bychom měli mezi zpětnými apostrofy zdvojovat. Nutné je to ovšem pouze tam, kde za zpětným lomítkem následuje další zpětné lomítko. Pokud za zpětným lomítkem následuje jiný znak, shell tuto kombinaci chápe správně.

V novějším způsobu zápisu lze naproti tomu snadno rozpoznat začátek a konec vloženého příkazu, a proto žádná zpětná lomítka zapotřebí nejsou:

```
sinek:~> vypis=$(ls $(pwd)/bin)
```

Ovšem nejjednodušší řešení je zcela přenositelné a výrazně „čitelnější“. Použijeme prostě další proměnnou:

```
sinek:~> adresar=`pwd`
sinek:~> vypis=`ls $adresar/bin`
```

Důležité je rovněž to, že shell z výstupu vygenerovaného příkazem odstraní *poslední* znak konce řádky. Jinak by se nám mezi jménem adresáře vypsaným příkazem **pwd** a následujícím lomítkem znak konce řádky objevil a příkaz by nefungoval.

Ovšem pozor, pro *vnitřní* konce řádek to neplatí. Napíšeme-li

```
sinek:~> dvakrat_adresar=`echo $adresar; echo $adresar`
```

znak konce řádky mezi oběma kopiemi jména adresáře bude.

A když už jsem takovou „ptákovinu“ napsal, musím se u ní zastavit. Napsat totiž mezi zpětné apostrofy jen příkaz **echo**, aniž jeho výstup jakkoli dále zpracováváme, postrádá téměř smysl. Příkaz z předešlého rámečku dělá prakticky totéž jako:

```
sinek:~> dvakrat_adresar="$adresar
> $adresar"
```

Jediný rozdíl je v tom, že případné posloupnosti bílých znaků v hodnotě proměnné se díky příkazu **echo** nahradí jedinou mezerou.

A ještě jedno upřesnění k algoritmu zpracování řádky. V minulé kapitole jsme si říkali, že shell v prvním kroku rozdělí řádku na slova. Řetězce uzavřené do apostrofů či uvozovek přitom považuje za jedno slovo. Zpětné apostrofy představují vlastně další formu quotingu – shell celý řetězec mezi nimi pro první okamžik také chápe jako nedělitelné slovo. U formy „**$(**...**)**“ to dokonce ani nepřekvapí, chová se skutečně téměř stejně jako expanze proměnné. Před vlastní interpretací příkazu mezi zpětnými apostrofy samozřejmě dojde k plnohodnotnému zpracování této části příkazové řádky.

Podobnou funkčnost, jakou poskytují zpětné apostrofy, už jsme viděli u příkazu **xargs**. A stejně jako tam, i zde je třeba varovat před bezmyšlenkovitým používáním. Obsah celého výstupního proudu vygenerovaného spuštěným programem si shell musí někam uložit, takže dřív než napíšete výraz „...**`cat soubor`**...“, zkontrolujte si nejprve velikost souboru...

4.7. Oddělovač polí

Zadání:

Zjistěte, kde leží program, který se zavolá, když napíšete příkaz **sort**.

Rozbor:

Už jsme si vysvětlovali, jak shell hledá programy, jejichž jméno napíšeme jako příkaz na příkazové řádce. A my tento postup teď musíme simulovat. Potřebujeme projít všechny adresáře uvedené v proměnné **PATH** a první z nich, v němž soubor najdeme, máme za úkol vypsat.

Nejsnazší bude použít příkaz **find**, nechat ho hledat ve všech adresářích obsažených v proměnné **$PATH** soubor správného jména a první výskyt vypsat. Je sice pravda, že příkaz bude zbytečně dál hledat i poté, co už nějaký výskyt objeví, ale počet adresářů nebude zase tak veliký. Rovněž zbytečného prohledávání do hloubky se nemusíme bát. Adresáře používané v proměnné **$PATH** nemívají mnoho podadresářů.

Zbývá poslední problém, jak převést seznam adresářů oddělených dvojtečkami na seznam parametrů. A tady s výhodou použijeme možnost změnit nastavení interního oddělovače polí (**IFS**).

Řešení:

```
sinek:~> IFS=:
sinek:~> find $PATH -name sort | head -n1
```

Poznámky:

Pokud byste si na pomocnou ruku, již vám skýtá proměnná **IFS**, nevzpomněli, mohli byste nahradit v hodnotě proměnné **PATH** dvojtečky mezerami a výsledek vložit do příkazové řádky pomocí zpětných apostrofů:

```
sinek:~> find `echo $PATH | tr : ' '` -name sort | head -n1
```

Hodnotu proměnné **IFS** bychom po skončení práce měli zase vrátit do původního stavu. Jinak by se nám mohly některé příkazy začít chovat divně. Nejjednodušší způsob je uložit si starou hodnotu do jiné proměnné:

```
sinek:~> ifs=$IFS
sinek:~> IFS=:
sinek:~> find $PATH -name sort | head -n1
sinek:~> IFS=$ifs
```

Pokud se vám zdá, že by výrazy **$IFS** a **$ifs** měly být zapsány v uvozovkách, pak to svědčí o vaší pozornosti. Důvod, proč to zde není nutné, spočívá v tom, že u přiřazování do proměnných shell neprovádí závěrečné dělení na slova podle **IFS** (a ani expanzi jmen souborů), jak jsme o tom mluvili v kap. 4.5.

Možná byste obsah **IFS** nechtěli ukládat a raději ho posléze sami znova nastavili. Potřebovali byste zapsat něco jako:

```
sinek:~> IFS='(mezera)(tabulátor)
> '
```

Pokud vaším shellem je **bash**, budete mít nejspíše problém, jak na příkazové řádce zapsat znak tabulátoru. V interaktivním režimu totiž tabulátor slouží k tomu, že „napovídá“ jména souborů. Když rozepíšete jméno souboru a zmáčknete tabulátor, shell zjistí, kolik souborů začíná řetězcem, který jste doposud napsali, a pokud je jediný, doplní konec jména. Pokud výběr není jednoznačný, doplní alespoň takovou část, která jednoznačná je. Dvojí zmáčknutí tabulátoru pak vypíše ty varianty, které shell našel, abyste si mohli zvolit vhodné pokračování. Chcete-li do řádky napsat opravdu tabulátor, budete muset těsně před ním stisknout kombinaci kláves Ctrl+V. Ta shellu říká: „to, co teď zmáčknu, do řádky prostě opiš“.

Na tomto místě je třeba upozornit na jednu pozoruhodnou anomálii, která má naštěstí pozitivní důsledky. Pokud jsou v proměnné **IFS** bílé znaky (což jsou všechny, co tam jsou implicitně) a shell je najde při rozdělování řádky na pole, sloučí jejich vícenásobné výskyty v jeden oddělovač polí. Ovšem naopak, pokud jsou tam znaky jiné, shell je neslučuje! Pokud by v našem příkladu byla v proměnné **PATH** hodnota „`/x::/y`“, dostal by příkaz **find** tři parametry („`/x`“, „“ a „`/y`“). Kdybychom zvolili variantu s náhradou dvojteček za mezery, dostal by pouze dva (dvě po sobě následující mezery by shell pochopil jako jediný oddělovač slov).

Volba jména pomocné proměnné (`ifs`) je dost odstrašující. V tomto případě by se to snad dalo ještě „skousnout“ (v tom smyslu, že `ifs` je záloha `IFS`), ale jinak používání různých proměnných, jejichž názvy se liší pouze v malých a velkých písmenech, vřele nedoporučuji.

4.8. Zpracování chyb

Zadání:

Vypište seznam adresářů v systému, k nimž nemáte přístupové právo pro čtení.

Rozbor:

Zjišťovat, zda máme dostatečná práva k adresáři, není úplně jednoduché. Snazší by bylo prostě zkusit každý adresář přečíst (třeba příkazem **find**). Pokud práva nemáme, program vypíše chybovou zprávu („`find: `*adresář*`: Permission denied`“) a tyto zprávy pošleme rourou programu **cut**, který je zpracuje a adresáře zobrazí.

Problém tohoto návrhu je ale v tom, že chybové zprávy se objevují ve standardním chybovém výstupu, zatímco roura (alespoň ve standardním shellu) propojuje „normální“ standardní výstup. Převedeme-li náš požadavek do terminologie deskriptorů, chtěli bychom pracovat s výstupem číslo 2 a nikoliv číslo 1.

Potřebujeme tedy shellu říct, aby výstup č. 2 „přehodil“ do výstupu č. 1, a tím do roury. Na to existuje zvláštní operátor přesměrování „**>&**“.

Mimo to musíme zařídit, aby výstup č. 1 shell zahodil. To už umíme.

Řešení:

```
sinek:~> find / 2>&1 > /dev/null | cut -d: -f2
```

Poznámky:

Pozor na to, v jakém pořadí operátory přesměrování zapíšeme. Kdybychom je totiž napsali obráceně, efekt by byl úplně jiný. Vyplývá to z implementace a není to na první pohled zřejmé. Dokonce se může zdát, že správné pořadí operátorů je „méně logické“. Pokud řádku budete číst: „vezmi výstup č. 2 a přehoď ho do výstupu č. 1, pak vezmi výstup č. 1 (a tedy logicky spolu s ním zároveň i č. 2) a pošli ho do `/dev/null`“, bude to špatně. Shell ve skutečnosti provede *duplikaci* (vytvoří kopii) deskriptoru, takže řádku je třeba číst takto: „nahraď výstup č. 2 kopií výstupu č. 1 (nasměruj ho tamtéž, kam směřuje výstup č. 1, tj. v našem případě do roury), pak vezmi výstup č. 1 (ale už nikoliv č. 2) a pošli ho do `/dev/null`“. Kdybychom operátory napsali obráceně:

```
sinek:~> find / > /dev/null 2>&1 | cut -d: -f2
```

museli bychom to číst: „vezmi výstup č. 1 a pošli ho do `/dev/null`, pak změň výstup č. 2 tak, aby směřoval tam, kam směřuje výstup č. 1 (neboli rovněž do `/dev/null`)“.

Operátory přesměrování mají vyšší prioritu než operátory řídicí (např. roura), což znamená, že patří k tomu příkazu, u něhož jsou zapsány (na stejné „straně“ roury). Nenechte se zmást zdánlivou nelogičností tohoto termínu – shell musí řídicí operátory (s nižší prioritou) při dělení řádky na jednotlivé příkazy vzít v úvahu dříve, než operátory přesměrování (mající vyšší prioritu).

4.9. Šetříme disky

Zadání:

Vypište seznam uživatelů ve skupině s číslem nula bez použití pomocného souboru.

Rozbor:

Problematiku členství ve skupinách a nutnosti zpracovat data ze dvou různých zdrojů (`/etc/group` a `/etc/passwd`) už jsme řešili v kapitole 1.20. Navrhovaným řešením tehdy bylo použití pomocného dočasného souboru:

```
sinek:~> grep :0: /etc/passwd | cut -d: -f1 > /tmp/grp0
sinek:~> grep :0: /etc/group | cut -d: -f4 >> /tmp/grp0
sinek:~> tr , '\n' < /tmp/grp0 | sort -u
```

K řešení bez pomocného souboru bychom potřebovali první dva příkazy spojit a výstup *obou dvou* společně poslat do roury k dalšímu zpracování. K tomu slouží *složený příkaz*. Zapisuje se podobně, jako jsme to viděli u složeného příkazu editoru **`sed`**.

Řešení:

```
sinek:~> {
>   grep :0: /etc/passwd | cut -d: -f1
>   grep :0: /etc/group | cut -d: -f4
> } | tr , '\n' | sort -u
```

Poznámky:

Složené závorky tvořící složený příkaz nejsou operátorem. Dá se na ně do jisté míry pohlížet jako na název příkazu. To má několik důsledků. Tím příjemným je, že složené závorky nejsou metaznakem, a tak není nutno se o ně zvlášť starat, když je potřebujeme na příkazové řádce použít v nějakém jiném významu. Horší je to, že mezi posledním příkazem a koncovou složenou závorkou být oddělovač příkazů (středník nebo konec řádky). Pokud tedy chceme složený příkaz napsat na jednu řádku, musíme před závorku napsat středník. Programátory v jazyce C to nepřekvapí, ale ostatní si na to musejí dávat dobrý pozor. Naopak úvodní složená závorka je zvláštní konstrukt, který může mít bezprostředně za sebou jiný příkaz. Za úvodní závorku tedy středník není nutno psát a stačí tam mezera. Naše série příkazů by se dala zapsat:

```
{ grep :0: /etc/passwd …; grep :0: /etc/group …; } | tr …
```

Úspora systémových zdrojů díky vynechání dočasných souborů je zanedbatelná. Navíc se současné systémy snaží zefektivnit práci programů tím, že omezují přesuny dat mezi operační pamětí a disky a ukládají soubory do vyrovnávacích paměti (*cache*). Proto se dokonce může stát, že program používající dočasný soubor bude rychlejší než stejný program využívající rouru. Spíš si tedy ušetříme problémy s hledáním vhodného volného jména pro dočasný soubor a s jeho mazáním při ukončení skriptu.

4.10. Čtení „per partes“

Zadání:

Upravte výpis „**ls -l**“ tak, že první řádku „total ...“ napíšete až nakonec.

Rozbor:

Potřebujeme první řádku přečíst, někam si ji uschovat a vypsat až na konci.

Pro čtení řádky použijeme, vcelku bez překvapení, příkaz s názvem **read**. Pak potřebujeme opsat zbytek vstupu. Na to se nejlépe hodí nenápadný **cat**. První řádku nakonec vypíšeme příkazem **echo**.

A už potřebujeme jen zařídit, aby příkazy **read** a **cat** četly ze stejného zdroje, a to jsme se naučili minule – použijeme složený příkaz.

Řešení:

```
sinek:~> ls -l | {
>   read total
>   cat
>   echo "$total"
> }
```

Poznámky:

Použili jsme složený příkaz k velmi podobnému účelu, jako v minulé kapitole – opět jde o společné přesměrování, jenže standardního vstupu.

Příkaz **read** přečte ze svého vstupu jednu řádku a její obsah uloží do proměnné, jejíž název dostane jako parametr. Tedy aspoň pro začátek si to takhle můžete představovat. Časem si to nepatrně zpřesníme.

To, že **read** přečte jen jednu řádku, je pro nás podstatné. Kdybyste chtěli první řádku zpracovat třeba nějakým programem, naše řešení by nefungovalo. Zkuste si napsat:

```
sinek:~> ls -l | { head -n1; cat; }
```

Program **head** totiž vstup (buď celý, nebo alespoň část) „zkonzumuje“. Naproti tomu **read** je interní příkaz shellu, a proto dokáže opravdu odebrat ze vstupního proudu jen jedinou řádku a nechat vstup otevřený a připravený pro čtení dalších řádek jinými příkazy, které následují.

Příkaz **read** se mj. používá k interakci s uživatelem:

```
sinek:~> echo -n "Zadej cislo: "; read cislo
```

Nebo chcete-li zvolit jiné riziko:

```
sinek:~> printf "Zadej cislo: "; read cislo
```

4.11. Když `read` nečte

Zadání:

Pomocí příkazu **`read`** načtěte do proměnné počet uživatelů systému vypsaný příkazem „**`wc -l < /etc/passwd`**“ a poté vypište obsah proměnné.

Rozbor:

Žádný problém. Napíšeme

```
sinek:~> wc -l < /etc/passwd | read pocet; echo $pocet
```

a začneme se divit. Výstup bude prázdný. Vysvětlení je jednoduché, ale je třeba malinko nahlédnout pod pokličku operačního systému.

Každý spuštěný program v UNIXu vykonává výpočetní jednotka, tzv. *proces* (příp. jich může být i více). Proces vznikne tak, že nějaký jiný (tzv. *rodičovský*) proces zavolá operaci *fork*, která vytvoří *synovský* proces. Když v shellu napíšeme příkaz **`wc`**, shell musí vytvořit synovský proces, který bude program vykonávat. Naproti tomu **`read`**, jak už víme, je interní příkaz shellu, a tak na jeho vykonání shell nepotřebuje nejen žádný programový soubor, ale také žádný nový proces.

Proměnné jsou vcelku pochopitelně uloženy v paměti běžícího procesu (shellu). To ovšem znamená, že žádný externí program (běžící v synovském procesu) s nimi nemůže manipulovat. O interních příkazech to, naštěstí, neplatí. Proto také **`read`** dokáže načíst proměnnou v té instanci shellu, ve které ho použijeme.

A co se stane, když má shell za úkol vytvořit rouru? V tom případě mu nezbyde nic jiného než vytvořit dva procesy (pro producenta i konzumenta) a mezi nimi (resp. mezi jejich výstupem a vstupem) vybudovat propojení. To už jsme viděli v praxi mockrát a zatím nám to bez problémů vyhovovalo. Tentokrát jsme však narazili, protože příkaz **`read`** proběhne díky rouře v jiném procesu, než je náš shell – v jeho synovském procesu. Náš shell nemá o změně hodnoty proměnné `pocet` v podprocesu ani zdání a vypíše hodnotu svojí vlastní proměnné `pocet` (která bude nejspíše prázdná).

Nejjednodušší řešení je opět vedeno mottem „odstraň problém“. Ten tkví v tom, že díky rouře čte proměnnou jiný proces, než je náš shell. Pokud v programu nebudeme mít rouru, nebude shell pro **`read`** vytvářet nový proces. Stačí tedy výsledek **`wc`** zapsat do dočasného souboru a zase ho z něj přečíst (a soubor po sobě nezapomenout smazat):

```
sinek:~> wc -l < /etc/passwd > /tmp/pocet
sinek:~> read pocet < /tmp/pocet
sinek:~> rm /tmp/pocet
sinek:~> echo $pocet
```

Kdybychom chtěli opět ekologicky šetřit pevné disky (a hlavně se nestarat o mazání souborů), pak řešení není úplně jednoduché. Přesněji řečeno, v tomto našem konkrétním případě to ještě není tak zlé. Stačí zařídit, že proces, který načte proměnnou, ji také vypíše. Na to můžeme s úspěchem použít složený příkaz. Pokud bychom ale takových čtení potřebovali uskutečnit více, program by se stal nečitelným.

Řešení:

```
sinek:~> wc -l < /etc/passwd | {
>   read pocet
>   echo $pocet
> }
```

Poznámky:

Namísto složeného příkazu bychom mohli shellu explicitně říci, aby nám vytvořil nový *podproces* (subshell). To se dělá pomocí (obyčejných, kulatých) závorek.

```
sinek:~> wc -l < /etc/passwd | (read pocet; echo $pocet)
```

Efekt bude úplně stejný, protože shell stejně zvláštní proces musí vytvořit. Takhle je to pouze o malinko viditelnější.

Na rozdíl od závorek složených jsou kulaté závorky *operátorem* a tedy metaznakem shellu. Mají proto opačné přednosti a nevýhody: není třeba kolem nich psát žádné mezery, středníky nebo konce řádek, ale zato si na ně musíme dávat pozor, kdekoliv je potřebujeme zapsat v jiném významu. A to není úplně řídkým jevem, vzpomeňte třeba na použití závorek v regulárních výrazech anebo v podmínkách příkazu **find**.

Podprocesy běžně používáme, když potřebujeme v nastavení shellu něco dočasně změnit a nechceme, aby tato změna ovlivnila aktuální shell. Příkladem může být změna hodnoty proměnné nebo aktuálního adresáře (jeho nastavení je vlastností procesu):

```
sinek:~> (cd adresář; rm *)
```

Možná se vám zdá zadání s omezením na použití příkazu **read** tak trochu nesmyslné. Pokud bychom toto omezení ignorovali, pak by nejlepší samozřejmě bylo napsat:

```
sinek:~> pocet=`wc -l < /etc/passwd`
```

V následující kapitole si ale ukážeme, proč podmínka použití **read** není tak nesmyslná, jak se na první pohled může zdát.

Norma dává velkou volnost v tom, které příkazy mohou být implementovány jako interní. Kromě těch, o nichž jsme se už zmiňovali (**cd**, **umask**, **read** a **:**), bývá obvykle jako interní (kvůli efektivitě) implementován například i příkaz **echo**.

4.12. Když `read` parsuje

Zadání:

Vypište vlastníka, velikost a jméno nejmladšího souboru v `/etc`.

Rozbor:

Najít nejmladší soubor nám pomůže příkaz **`ls`**. A jeho výstup budeme muset rozdělit na jednotlivé sloupce. Už jsme to zkoušeli různě, utilitami **`tr`** a **`cut`** nebo editorem **`sed`**.

S příkazem **`read`** však máme po ruce daleko jednodušší řešení. Příkaz je totiž mnohem chytřejší, než jsme doposud poznali. V případě, že dostane víc parametrů (více jmen proměnných), rozdělí načtenou řádku na slova a postupně je přiřadí jednotlivým proměnným. Používá přitom stejná pravidla jako shell – posloupnost bílých znaků bude považovat za jediný oddělovač.

Řešení:

```
sinek:~> ls -lt | {
>   read total
>   read x x usr x size x x x name
>   echo "$usr $size $name"
> }
```

Poznámky:

První volání **`read`** odstraní ze vstupu řádku „`total`...“, druhé přečte „naši“ řádku. Zbytek výstupu příkazu **`ls`** se zahodí.

Navržené řešení zvládne i soubory s mezerami ve jméně. Pokud se totiž v přečtené řádce nalézá více slov, než má **`read`** parametrů, uloží se do poslední proměnné celý zbytek řádky (to vlastně bylo chování, které jsme viděli dosud v případě zadání jediné proměnné). V proměnné `name` tedy budeme mít správné jméno souboru, dokonce i tehdy, když bude uvnitř obsahovat posloupnosti mezer o délce větší než jedna.

Pokud má příkaz **`read`** naopak více parametrů, než bylo slov na řádce, shell hodnoty nadbytečných proměnných vymaže.

Pokud nás některá slova z řádky nezajímají, nemusíme se trápit s vymýšlením řady pěkných jmen proměnných a použijeme víckrát stejné jméno. Shell si o nás sice asi něco pomyslí, ale je velmi diskrétní a nedá to najevo...

Slovo „parsování“ by se do (skoro) češtiny dalo přeložit jako „provádění syntaktické analýzy“, což je ale krapet těžkopádné. Proto se omlouvám všem jazykovým puristům a setrvám u něj...

4.13. Parametry

Zadání:

Zjistěte počet souborů (jejichž jméno nezačíná tečkou) v aktuálním adresáři.

Pokuste se minimalizovat počet procesů, jež budete potřebovat.

Rozbor:

Úlohu jsme vyřešili už v kapitole 1.11. Využívali jsme přitom několik pomocných programů. Kde bychom mohli ušetřit? Minule nám soubory vypisoval příkaz **`ls`**. Jenže my víme, že procházení adresáře zvládne shell sám. Stačí napsat hvězdičku. Dál musíme vypsané soubory spočítat. Minule jsme použili **`wc`**. Nyní si musíme pomoci jinak. Shell na tom je s aritmetikou dost bídně, to ještě uvidíme. Poziční parametry ale počítat umí.

Poziční parametry slouží primárně pro přístup k parametrům, které uživatel zadal při zavolání našeho skriptu. Toho začneme brzo intenzivně využívat. Mají ovšem i další uplatnění – jejich hodnoty totiž můžeme libovolně sami přenastavit příkazem **`set`**. A my necháme shell, aby do nich dosadil jména souborů.

Počet pozičních parametrů pak zjistíme pomocí speciálního parametru **`$#`**.

Řešení:

```
sinek:~> set -- *; echo $#
```

Poznámky:

Použitá forma příkazu **`set`** vezme své parametry a postupně je přiřadí do pozičních parametrů. Parametr „**`--`**“ jsme přidali jako ochranu proti „divným“ jménům souborů.

K hodnotám jednotlivých pozičních parametrů se můžeme dostat podobně jako k hodnotám proměnných, jen místo jména píšeme pořadové číslo: „**`$1`**“, „**`$2`**“ atd. Po provedení příkazu **`set`** bychom tedy jméno prvního souboru vypsali jako „**`$1`**“.

Příkaz **`set`** je jedinou možností, jak můžeme do pozičního parametru něco přiřadit explicitně. Rozhodně nefunguje něco jako „**`1=`***hodnota*“.

Pokud potřebujeme parametr s číslem větším než devět, musíme použít konstrukt „**`${10}`**“ (zápis „**`$10`**“ totiž znamená „první parametr a za ním znak nula“). Tento konstrukt pro zápis pozičních parametrů ale bohužel nenajdeme u starších verzí shellu. Naštěstí se obvykle k parametrům s takovými čísly přistupuje jinak (viz kapitolu 4.25).

Ale pozor, u obyčejných proměnných je to obráceně! U nich shell neví, jak dlouhý je jejich název, a proto mu v určitých případech musíme složenými závorkami pomoci hranice názvu rozpoznat. Ještě se k tomu vrátíme v kapitole 4.16.

O potřebě psát v shellu efektivně jsme už mluvili. Samozřejmě nejde o jeden dva procesy, ale občas je dobré přemýšlet, zda někde nejde něco ušetřit bez ztráty kytičky...

4.14. Hello, world

Zadání:

Vytvořte skript, který bude mít dvě jména (hardlinky), `Hello` a `Hi` a pozdraví buď „`Hello, world`“, nebo „`Hi, world`“ podle toho, kterým jménem ho uživatel zavolá.

Rozbor:

Vzhledem k tomu, že má jít o dva hardlinky na stejný soubor, budeme muset druh pozdravu rozlišit až za běhu programu. Shell má naštěstí zvláštní parametr **`$0`**, který obsahuje *jméno skriptu* přesně tak, jak bylo zapsáno na příkazové řádce, a tak samotná volba druhu pozdravu bude jednoduchá.

Myšlenka skriptu je v hlavních obrysech jasná, musíme dořešit ještě několik detailů, jak přesně potřebný skript vytvořit a jak ho pak volat.

Pokud vás jako možnost napadl editor, konkrétně editor **`ed`**, máte moje sympatie. Ale ani já bych toto řešení nevolil. Takový triviální skript se snáze vytvoří příkazem **`echo`** a přesměrováním jeho výstupu do souboru. To už jsme koneckonců dělali dávno.

A víme už i to, že shell při přesměrování nevytvoří soubor s takovými právy, jaká bychom potřebovali. Budeme tedy muset sami přidat souboru právo „**`x`**“.

Zbývá otázka, jak přesně se skript bude volat. Pokud totiž nebude uložen v některém z adresářů uvedených v proměnné **`PATH`**, nebudeme ho moci volat pouze jeho jménem. Museli bychom psát místo jména plnou cestu (např. „**`./Hello`**“). Jenže pak by se v **`$0`** (a tím i v našem pozdravu) objevila i tečka s lomítkem a to by nebyl moc pěkný pozdrav. Proto z parametru **`$0`** vybereme pouze část za posledním lomítkem.

K tomu se dá použít program **`basename`**. Já vím, že to už umíme pomocí příkazů **`echo`** a **`sed`**, ale můžeme se zase naučit něco nového. Konečný tvar skriptu tedy bude:

```
echo "`basename $0`, world"
```

Řešení:

```
sinek:~> echo 'echo "`basename $0`, world"' > Hello
sinek:~> ln Hello Hi
sinek:~> chmod +x Hello
sinek:~> ./Hi
Hi, world
```

Poznámky:

Každá správná učebnice programování začíná programem Hello world. V shellu je to ale tak triviální, že jsme si to úmyslně krapet zkomplikovali a příklad použili až mnohem později. Doufáme, že nám to odpustíte.

4.15. Škatulata, hejbejte se

Zadání:

Vytvořte skript, který bude tvořit jen jakousi obálku pro volání příkazu **mv** a předá mu svoje parametry. Na rozdíl od něj bude ale skript očekávat jméno cílového adresáře jako první parametr. Skript ho přesune na konec, ostatní parametry zachová ve stejném pořadí a zavolá **mv**.

Rozbor:

Tentokrát budeme poziční parametry používat v jejich základním významu. Jejich hodnoty shell nastaví na začátku skriptu a my s nimi budeme pracovat.

Jako svoji poslední akci bude náš skript rozhodně provádět volání příkazu **mv**, ale pro něj bude potřebovat už mít připravené všechny parametry na správném místě.

Předně musíme někam uložit první parametr. Na to bude stačit obyčejná proměnná. Pak se ale tohoto prvního parametru budeme muset „zbavit". K tomu zase má shell zvláštní příkaz **shift**, který „posune" poziční parametry. Z druhého parametru se stane první, ze třetího druhý atd. První parametr nenávratně zmizí (proto jsme si ho nejdříve uložili do jiné proměnné).

Nakonec potřebujeme shellu říct „a teď do příkazové řádky opiš všechny poziční parametry, co máš". Na to slouží další zvláštní parametr **$@**.

Řešení:

```
cil=$1
shift
mv "$@" "$cil"
```

Poznámky:

Skript je malinko paranoidní. Pokud ho budou volat rozumní uživatelé s rozumnými jmény souborů, jsou všechny uvozovky zbytečné. Ale trocha opatrnosti občas neškodí. Pokud bychom učinili předpoklad „rozumnosti" uživatelů a uvozovky do skriptu nedali, jistě by se velmi rychle našel uživatel, který by náš předpoklad vyvrátil.

Příkaz **shift** může mít ještě parametr udávající, kolik pozičních parametrů má zmizet (o kolik se mají čísla posunout).

Tento příkaz je základem metody, jak se v shellu zpracovávají parametry, které skript dostal při zavolání na příkazové řádce. Budeme ho ještě často využívat.

Pro pořádek je třeba zdůraznit, že zvláštní parametr **$0** (název skriptu) není poziční parametr a posunu se neúčastní.

Kromě zvláštního parametru **$@** má shell ještě jeden podobný: **$***. Rozdíl mezi nimi asi nejlépe vysvětlíme na příkladu. Mějme v pozičních parametrech hodnoty uvedené v následující tabulce a předpokládejme, že zavoláme příkaz **set**. Jak se hodnoty změní:

		$#	**$1**	**$2**	**$3**	**$4**
aktuální nastavení parametrů		3	„1 2 4“	„X“	„“	
po zavolání	**set -- "$@"**	3	„1 2 4“	„X“	„“	
	set -- "$*"	1	„1 2 4 X “			
	set -- $@	4	„1“	„2“	„4“	„X“
	set -- $*	4	„1“	„2“	„4“	„X“

Konstrukt **"$@"** je tedy nejobecnější – opisuje parametry a tvoří z nich „slova“ přesně tak, jak vypadaly parametry původně. Konstrukt **"$*"** z nich naopak vytváří jediné „slovo“ (oddělí je přitom mezerami). A konečně oba konstrukty bez uvozovek se chovají stejně – opisují obsah parametrů jako jednotlivá slova bez zachování původního členění.

4.16. Dynamický vstup

Zadání:

Vytvořte skript, který ze zadaného souboru vymaže řádku s daným číslem.

Skript dostává dva parametry, prvním je jméno souboru, druhým je číslo řádky. Pokud číslo chybí, smaže se první řádka. Můžete předpokládat, že vstup je korektní (soubor existuje a je dost velký).

Rozbor:

Pro modifikaci souboru se jednoznačně hodí editor **`ed`**. Potřebujeme mu jen předat potřebné příkazy na vstup. Mohli bychom mu je poslat rourou, ale naučíme se lepší postup. Shell umožňuje zapsat obsah vstupu přímo jako součást svého kódu.

Syntakticky se to zapíše jako zvláštní přesměrování pomocí operátoru „**`<<`**“, jehož operandem je slovo, které slouží jako „zarážka“. Veškeré řádky, které v kódu následují, až po řádku obsahující pouze tuto zarážku (nevčetně) se stávají textem standardního vstupu pro příkaz.

V našem případě to budou právě dvě řádky, s příkazy *delete* a *write*.

Pro test existence druhého parametru a dosazení implicitní hodnoty použijeme tzv. podmíněnou substituci parametru.

Řešení:

```
jmeno=$1
radka=${2:-1}
ed "$jmeno" << KONEC
${radka}d
w
KONEC
```

Poznámky:

Podmíněná substituce má tvar „**`${`***jméno***`:-`***slovo***`}`**“ a znamená, že se otestuje obsah proměnné či parametru *jméno* a v případě nenastavené nebo prázdné hodnoty se použije řetězec „*slovo*“, jinak se použije hodnota proměnné, resp. parametru.

Je to efektivní a elegantní, mám pouze drobný problém s mnemotechnikou. Trochu mi pomáhá následující trik: existuje totiž ještě konstrukt „**`${`***jméno***`:=`***slovo***`}`**“, který znamená, že pokud proměnná *jméno* není nastavená nebo má prázdnou hodnotu, pak se do ní text „*slovo*“ přiřadí. Tento konstrukt se skutečně podobá přiřazovacímu příkazu, a to se pamatuje dobře. No a konstrukt „**`${`***jméno***`:-`***slovo***`}`**“ dělá jen „půlku“ práce oproti „**`${`***jméno***`:=`***slovo***`}`**“, a tak má místo dvou jen jednu čárku...

Pokud chcete provést pouze podmíněné přiřazení a novou hodnotu nechcete hned nikam dosadit, můžete využít prázdný příkaz „**`: ${`***jméno***`:=`***slovo***`}`**“.

Textu vloženému operátorem „**<<**“ se říká *here-dokument*. Volně přeloženo je to něco jako „dokument, který vytváříme na místě“, ale překládat to už raději nebudeme a budeme slovní spojení „here-dokument“ chápat jako odborný termín...

Obsah here-dokumentu podléhá textovým substitucím shellu. Chová se (zhruba řečeno) jako text v uvozovkách, shell v něm například expanduje hodnoty proměnných.

Pozor ale na nechtěné důsledky. Kdybychom potřebovali smazat poslední řádku, nemohli bychom do here-dokumentu napsat příkaz „**$d**“, protože shell by se pokusil proměnnou expandovat. Jednou možností by bylo napsat před dolar zpětné lomítko:

```
ed "$jmeno" << KONEC
\$d
w
KONEC
```

Druhou možností je zakázat shellu zasahovat do řádek here-dokumentu. To zařídíme tak, že operand operátoru „**<<**“ (tj. zarážku) zapíšeme s použitím quotingu. Třeba takto:

```
ed "$jmeno" << \KONEC
$d
w
KONEC
```

Pokud se here-dokument nachází v místě kódu, které je odsazené (*indentované*), a budete chtít podobně odsadit i vlastní řádky dokumentu, můžete použít jinou variantu operátoru, a to „**<<-**“:

```
        ed "$jmeno" <<- KONEC
            ${radka}d
            w
            KONEC
```

Způsobí, že shell ze začátku každé řádky odstraní úvodní posloupnost tabulátorů. Pozor, pro případné mezery to neplatí, ty shell neodstraňuje.

Složené závorky kolem jména proměnné jsme použili proto, že výraz „**$radkad**“ by shell vyhodnotil jako substituci proměnné `radkad`. Stejný konstrukt jsme už viděli v kapitole 4.13. Tam se ale jednalo o poziční parametry a u nich je problém opačný – složené závorky jsou nutné pro zápis pozičního parametru s číslem vyšším než devět. U obyčejných proměnných shell neví, jak dlouhý je jejich název, a pokud za místem expanze následují alfanumerické znaky, shell je „přidá“ ke jménu proměnné. Zrádným znakem je zejména podtržítko („_“), které tváří nevinně, ale patří mezi alfanumerické znaky. Naštěstí u proměnných (na rozdíl od pozičních parametrů) jsou složené závorky bez problémů přenositelné.

V tomto konkrétním případě bychom si ovšem mohli pomoci i jinak – mezi adresou a písmenem příkazu v editoru smí být mezera:

```
ed "$jmeno" << KONEC
$radka d
w
KONEC
```

Situace, které vyžadují použití here-dokumentu jsou záludné v tom, že člověk snadno podlehne dojmu, že nic takového nepotřebuje. Přece když totéž píše přímo interaktivně v shellu, vše funguje, jak má:

```
sinek:~> ed "$jmeno"
$d
w
```

Co se ale stane, když ve skriptu bude napsán následující text?

```
ed "$jmeno"
$d
w
```

Shell spustí editor, ale ten, nemaje přesměrovaný vstup, bude očekávat zadávání příkazů z klávesnice. Netuší (jak by mohl), že jsme mu připravili řádky vstupu někam do těla skriptu. Naopak, kdyby se shell nakrásně nějak přes spuštění editoru dostal, pokoušel by se zavolat (shellové) příkazy „**$d**“ a „**w**“.

Nepříjemné na takové chybě je to, že ji hned tak nepoznáte. Skript prostě z vašeho pohledu „poběží nějak dlouho“. Vy si o něm budete myslet, že by si mohl „pospíšit“, a on si zase bude myslet, že už byste na klávesnici nějaký ten příkaz pro editor přece jen mohli zadat, když už na něj takovou dobu čeká...

Here-dokument se dá rovněž použít ke zdánlivě překombinované konstrukci:

```
read pocet x << KONEC
`wc -l /etc/passwd`
KONEC
```

Jako vstup vkládáme text, který ale současně celý vyrábíme spuštěním jiného programu. Pokud si říkáte, proč nepoužít prostě rouru, máte do jisté míry pravdu:

```
wc -l /etc/passwd | read pocet x
```

Pokud by byl text určen ke zpracování v nějakém jiném programu, jistě by to takhle šlo. Ale v případě příkazu **read** už jsme si vysvětlovali, že není tak jednoduché nechat ho číst z roury, protože to znamená čtení v podprocesu a tedy do úplně jiných proměnných. A tento trik s here-dokumentem představuje způsob, jak tuto potíž obejít. Takové volání příkazu **read** čte správně.

4.17. Ak, pak

Zadání:

Vytvořte skript, který smaže soubor zadaný parametrem. O výsledku vypíše zprávu.

Rozbor:

Smazání samotné provedeme příkazem **`rm`**. Pokud při něm dojde k chybě, nebudeme už muset nic dalšího psát, protože příkaz sám vypíše chybovou zprávu. Potřebujeme tedy ošetřit jen případ úspěšného provedení příkazu.

Zkušení programátoři už jistě cítí, že nadešel okamžik pro nějakou řídicí strukturu typu *if-then* (nebo jako kdysi ve slovenské mutaci Pascalu *ak-pak*). Ale já je zklamu. Ne snad, že by takováto struktura v shellu neexistovala, ale naučíme se nejprve o něco jednodušší verzi, která se v některých situacích hodí lépe.

Myšlenkově a syntakticky vychází z podmíněného vyhodnocování logických výrazů v jazyce C. Když v něm napíšete výraz „*A nebo B*“ (v syntaxi jazyka C je to „*A* **`||`** *B*“), nejprve se vyhodnotí podvýraz *A* a k vyhodnocení podvýrazu *B* dojde pouze tehdy, pokud *A nebyl* pravdivý. Pokud *A* už sám byl pravdivý, nemá smysl vyhodnocovat *B*, protože už to celkový výsledek nemůže změnit. Analogicky u výrazu „*A a B*“ (neboli „*A* **`&&`** *B*“) se *B* vyhodnotí pouze tehdy, když *A* uspěje (bude pravdivý). A stejnou syntaxi i logiku přenesli autoři i do shellu. Pokud zapíšeme výraz „*A* **`||`** *B*“, znamená to „spusť příkaz *A*, a pokud neuspěje, spusť i *B*, jinak skonči“. Analogicky „*A* **`&&`** *B*“ znamená „proveď *A* a v případě úspěchu rovněž *B*“.

Na základě čeho se ale rozhoduje o „úspěšnosti“ běhu programu? Každý program předává tomu, kdo ho zavolal, tzv. *návratovou hodnotu* vyjadřující, s jakým výsledkem skončil. Nemůže se moc „rozšoupnout“, k dispozici má jen hodnoty v rozsahu 0 až 255, a to ještě některé z nich používá operační systém sám. Pro základní rozlišení (úspěch či neúspěch) to však stačí. Za úspěch se přitom považuje návratová hodnota *nula*, nenulové hodnoty se obvykle používají pro rozlišení různých důvodů neúspěšného konce.

Řešení:

```
rm "$1" && echo "Soubor $1 smazan"
```

Poznámky:

Pokud byste chtěli po skončení příkazu vidět návratovou hodnotu, je dostupná ve zvláštním parametru **`$?`**.

Pokud pouštíte více příkazů (díky složenému příkazu, rouře, řídicím konstrukcím nebo náhradě výstupu příkazu pomocí `` `...` ``), návratovou hodnotu určuje ten poslední.

Za operátory **`&&`** a **`||`** je možno odřádkovat, podobně jako za operátorem **`|`** (rourou).

4.18. Ak, pak, inak

Zadání:

Napište skript, který se pokusí vytvořit adresář zadaný parametrem. V každém případě oznámí na standardní chybový výstup výsledek („adresář už existuje“, „operace se povedla“ nebo „operace selhala“).

Rozbor:

Teď už opravdu použijeme příkaz **`if`**. Slouží k větvení programů. Programátorům bude jistě připomínat podobné příkazy, jež znají z jiných jazyků. Jenom si budou muset zvyknout na pár odlišností.

Asi tou nejzásadnější je fakt, že tím, co se v příkazu **`if`** vyhodnocuje, není podmínka, ale příkaz! To je na první pohled poněkud zarážející, ale když si uvědomíte, že celý shell je vlastně vymyšlen právě proto, aby se pomocí něj daly spouštět příkazy, logicky to do sebe zapadá. Shell spustí *příkaz* zapsaný za klíčové slovo **`if`** a na základě jeho výsledku (neboli návratové hodnoty) se vydá jednou či druhou větví (podle obvyklé konvence názvů klíčových slov se jim říká **`then`** a **`else`** větev).

Přirozenou otázkou ovšem je, jak se tedy zařídí to, když příkazem **`if`** potřebujeme otestovat nějakou podmínku (třeba jako v našem případě skutečnost, že adresář existuje). A přirozenou odpovědí je, že použijeme příkaz **`test`**. V parametrech se mu zadá podmínka („**`-d`** *jméno*“), příkaz ji zkontroluje a podle výsledku nastaví návratovou hodnotu. Tu pak následně využijeme v příkazu **`if`**, abychom vybrali správnou větev.

Pro zápis do standardního chybového výstupu použijeme obyčejný příkaz **`echo`**. Jeho výstup ale operátorem „**`>&`**“ přesměrujeme do standardního chybového výstupního proudu (tedy proudu č. 2).

Řešení:

```
if test -d "$1"; then
    echo "Adresar '$1' jiz existuje" >&2
elif mkdir "$1" 2> /dev/null; then
    echo "Adresar '$1' byl vytvoren" >&2
else
    echo "Adresar '$1' nejde vytvorit" >&2
fi
```

Poznámky:

Pokud znáte programovací jazyky jako C nebo Pascal, pak si budete muset zvyknout na drobnou odlišnost v návrhu příkazu **`if`** a některých dalších. V jazyce C můžete do každé větve zapsat jen jeden příkaz a chcete-li tam mít více příkazů, musíte je uzavřít do

složeného příkazu. Naopak většina řádkově orientovaných jazyků (třeba preprocesor C nebo MS Visual Basic) dovoluje do větve zapsat libovolně mnoho příkazů a jejich konec se pozná podle nějakého klíčového slova (třeba `#endif` nebo `End If`). Shell patří do této druhé skupiny a klíčovým slovem, které ukončuje celý příkaz **`if`**, je „**`fi`**“ (je to slovo „**`if`**“ napsané pozpátku). Příkazy ve větvi **`then`** ukončuje libovolné z klíčových slov „**`elif`**“, „**`else`**“ nebo „**`fi`**“.

Abychom byli zcela přesní, **`if`** není příkaz (ačkoliv mu tak většina lidí říká), ale *řídicí struktura*, která se zpracovává dokonce ještě dříve než řídicí operátory shellu – proto například součástí příkazu za **`if`** může být roura. Ale když si pro jednoduchost představíte, že **`if`**, **`then`** nebo **`else`** jsou příkazy (pro jejichž používání platí určitá pravidla), zase tak velkou chybu neuděláte. A bude vám třeba jasné, že chcete-li slova **`if`** a **`then`** zapsat na jednu řádku, musíte mezi ně přidat středník.

Klíčové slovo **`elif`** je vlastně pouze zkratkou za dvojici **`else`** a **`if`**. Klidně bychom mohli použít tuto kombinaci, ale na konci bychom pak museli psát dvakrát **`fi`** (pro každý **`if`** jedno).

Žádná větev (**`then`** ani **`else`**, je-li použita) nesmí být prázdná. Nechceme-li v dané větvi nic dělat, musíme použít nějaký neškodný příkaz. Nejlepší je příkaz „**`:`**“.

Pokud příkazy za **`if`** (resp. **`elif`**) produkují nějaký výstup (ať už standardní nebo chybový) a pokud tento výstup nechceme, budeme ho muset explicitně potlačit.

Pokud potřebujeme „obrátit“ výsledek příkazu (úspěch na neúspěch, resp. nulovou návratovou hodnotu na nenulovou a naopak), můžeme před příkaz zapsat vykřičník:

```
if ! mkdir "$1" 2> /dev/null; then
```

Příkaz **`test`** se dá vyvolat pomocí dvou různých jmen. Kromě „**`test`**“ je možno ještě použít i formu „**`[`**“. Obě formy jsou funkčně zcela identické, jediný rozdíl spočívá v tom, že u „závorkové“ formy musíme na konec parametrů ještě přidat jako samostatný parametr „zavírací“ závorku „**`]`**“. První příkaz by tedy mohl mít tvar:

```
if [ -d "$1" ]; then
```

Právě tato, velmi často používaná podoba vede méně zkušené uživatele k pocitu, že i v shellu se za příkazem **`if`** přece jen píše podmínka (akorát se tam používají poněkud atypické závorky). Ale hranatá závorka (ačkoliv tak nevypadá) je opravdu příkaz. Klidně ji můžete zapsat přímo na příkazové řádce, aniž tam máte příkaz **`if`**. A dokonce se to často dělá, a to v kombinaci s řídicími operátory z minulé kapitoly. Například následující sekvence je vlastně ak-pak-inak, ale trochu inak:

```
[ -d "$1" ] && echo ... || echo ...
```

Pokud **`test`** projde, pokračuje se prvním příkazem **`echo`**, ten skončí rovněž úspěchem, a proto se druhé **`echo`** nezavolá. Pokud **`test`** naopak selže, první **`echo`** se nezavolá, návratová hodnota zůstává stejná (tj. „neúspěch“), pročež se vykoná druhé **`echo`**.

Často se setkávám se snahou použít pro **if** nějaký „normální“, výkonný příkaz, ale autorovi se tam zkrátka „připletou“ ty závorky, „co se tam přece vždycky píšou“. Zápis

```
if [ mkdir "$1" 2> /dev/null ]; then
```

shell pochopí jako volání příkazu **test** (resp. „**[**“) se třemi parametry „**mkdir**“, „**$1**“ a „**]**“ a s přesměrovaným chybovým výstupem.

Podmínka vyhodnocovaná příkazem **test** má tvar logického výrazu, který se skládá z elementárních podvýrazů. Tyto podvýrazy obvykle testují nějaký stav systému souborů nebo hodnoty řetězců a jsou buď unární (pak mají obvykle tvar podobný obyčejnému přepínači – jako náš „**-d**“), nebo binární (mají jeden operátor a dva operandy). Přehled těch nejdůležitějších najdete v následující tabulce a podrobněji na str. 318:

Podvýraz:	Testuje:
-e *file*	existenci souboru *file*
-d *file*, **-f** *file*, **-L** *file*	typ souboru *file*
-r *file*, **-w** *file*, **-x** *file*	přístupová práva k souboru *file*
-s *file*	nenulovou délku souboru *file*
-z *str*, **-n** *str*	(ne)nulovou délku řetězce *str*
str1 **=** *str2*, *str1* **!=** *str2*	shodnost řetězců *str1* a *str2*
i1 **-eq** *i2*, *i1* **-ne** *i2*, *i1* **-lt** *i2*, *i1* **-gt** *i2*, *i1* **-le** *i2*, *i1* **-ge** *i2*	vzájemnou velikost čísel *i1* a *i2*

Logické výrazy se tvoří pomocí spojek „a zároveň“ („**-a**“, *and*) a „nebo“ („**-o**“, *or*), negace („**!**“) a závorek. Pozor ale na to, že závorky jsou metaznaky shellu, a proto je musíte pečlivě ošetřovat, například takto:

```
[ -s file -a ! \( -r file -o -w file \) ]
```

Pro zápis složitějšího výrazu lze použít i shellové operátory „**&&**“ a „**||**“:

```
[ -s file ] && ! ( [ -r file ] || [ -w file ] )
```

Navíc tím můžeme dosáhnout podmíněného vyhodnocování výrazu (příkaz **test** totiž vyhodnocuje vždy celou podmínku). Pouze musíte počítat s nárůstem počtu procesů.

Při psaní podmínek příkazu **test** je třeba mít na paměti, že před vyvoláním příkazu ještě vstoupí do hry shell. Pokud tedy testujeme třeba hodnotu nějaké proměnné, je dobré ji uzavřít do uvozovek, aby ji shell chápal a také předal příkazu **test** jako jedno slovo. Pokud to neuděláte a proměnná bude buď prázdná, nebo naopak bude obsahovat více slov, příkaz **test** vašemu záměru jistě neporozumí.

4.19. Když `if` nestačí

Zadání:

Rozšiřte skript z kapitoly 4.16 o kontroly parametrů.

Rozbor:

Co všechno potřebujeme otestovat?

Zadané jméno musí být jménem existujícího obyčejného souboru a musíme k němu mít dostatečná práva. To všechno zvládne příkaz **`test`**, potřebné operátory jsme viděli v minulé kapitole.

Zadané číslo musí být opravdu číslo. To už je horší problém. Na to nám příkaz **`test`** už nebude stačit. Neumí totiž testovat shodu řetězce s nějakým vzorem (jako např. s regulárním výrazem). Mohli bychom si nějak pomoci příkazem **`grep`**, ale naučíme se jednodušší a rychlejší způsob, a to pomocí příkazu (řídicí struktury) **`case`**.

Posledním krokem bude kontrola délky souboru. Zde už se vrátíme opět k příkazu **`test`** a porovnáme zadaný parametr s výsledkem volání příkazu **`wc`**.

Příkaz **`case`** má jako parametr řetězec, ten se rozvine podle pravidel shellu a poté se porovná se seznamem *porovnávacích vzorů*. První vyhovující vzor určí větev, kterou bude skript dále pokračovat. Metaznaky používané v porovnávacích vzorech (hvězdičku, otazník a hranaté závorky) už jsme poznali dříve. Zde však nemají význam expanzních znaků (nezastupují žádná jména souborů), a proto třeba hvězdička zahrnuje i řetězec s tečkou na začátku nebo obsahující lomítko (podobně jako u příkazu **`find`**).

Náš drobný problém spočívá v tom, že se pomocí těchto metaznaků nedá zapsat vzor „řetězec je číslo“. Pomocí regulárních výrazů by to šlo („**`^[0-9][0-9]*$`**“), protože jsou „silnější“, mají větší vyjadřovací sílu. Naštěstí pro nás lze jednoduše napsat *negaci* této podmínky. Ta bude znít „řetězec je prázdný nebo obsahuje alespoň jeden znak různý od číslice“ a to zvládneme jednoduše i pomocí porovnávacích vzorů.

Řešení:

```
case $2 in
'' | *[!0-9]* ) echo "Parametr '$2' neni cislo"; exit 1;;
esac
[ ! -f "$1" ] && echo "Soubor '$1' neexistuje" && exit 2
if [ ! -r "$1" -o ! -w "$1" ]; then
    echo "K souboru '$1' nemame prava"
    exit 3
fi
if [ `wc -l < "$1"` -lt $2 ]; then
    echo "Soubor '$1' je maly"
    exit 4
fi
```

Poznámky:

Návěští, jež obsahují vzory pro jednotlivé větve příkazu **case**, se ukončují pravou (zavírací) kulatou závorkou. Pokud má nějaká větev více vzorů, oddělují se svislítkem.

Syntaxe příkazu **case** je mírně nepovedená. Potíže působí třeba nespárované kulaté závorky (proto norma povoluje před první vzor každé větve zapsat levou závorku navíc). Symbol pro konec větve je zase trochu „perlovský" – autoři se asi upřeně zahleděli na klávesnici chtějíce do ní někam zabodnout prst, ale nic vhodného nenašli, a tak prostě zdvojili středník...

Pořadí zápisu větví struktury **case** je důležité, protože prohledávání vzorů shell ukončí při první shodě. Chceme-li vytvořit *implicitní* větev, která se má použít, pokud všechny ostatní vzory selžou, bude mít vzor „*****" a musí být zapsaná jako poslední.

Pokud selžou úplně všechny vzory, shell nevykoná žádnou větev.

Výraz za klíčovým slovem **case** jsme napsali bez uvozovek. Můžeme si to dovolit, protože shell u tohoto výrazu neprovádí závěrečné dělení podle proměnné **IFS**.

V poslední podmínce jsme také vynechali uvozovky kolem „**$2**". Jenže tam už víme, že poziční parametr obsahuje číslo, takže nedojde k žádným neočekávaným problémům.

Příkaz **test** opravdu neumí ověřit shodu s nějakým vzorem. Často vídávám marné pokusy jako „**[$x = a*]**" ve smyslu „začíná hodnota proměnné x písmenem a?". Bylo by sice možné použít třeba konstrukci „**["`echo "$x" | cut -c1`" = a]**", ale je to oproti použití struktury **case** poněkud těžkopádné.

Pro testování, zda druhý parametr obsahuje pouze číslice, lze použít také příkaz **grep**. Ideální je varianta s přepínačem „**-q**" (*quiet*), která pouze kontroluje shodu se vzorem, nic nevypisuje a vrací odpovídající návratovou hodnotu:

```
if echo "$2" | grep -vq '^[1-9][0-9]*$'; then
```

Pozor ale na záměnu s nesprávným zápisem níže – kde je chyba?

```
if grep -vq '^[1-9][0-9]*$' "$2"; then
```

Příkaz **exit** ukončí provádění aktuálního shellu. Pokud je součástí skriptu, ukončí jeho provádění. Kdybyste ho zadali na příkazové řádce, ukončí vaši relaci.

Jednotlivým voláním příkazu **exit** jsme přidali ještě parametr, jimž rozlišujeme *návratovou hodnotu* celého skriptu (jinak by se použila návratová hodnota posledního příkazu). Různé důvody ukončení našeho skriptu pak lze testovat například takto:

```
náš_skript
case $? in
1 ) ... parametr není číslo;;
2 ) ... soubor neexistuje;;
3 ) ... nedostatečná práva;; ... atd.
```

4.20. Příkaz `case` naruby

Zadání:

Předpokládejte, že máte v proměnné `fakulty` seznam zkratek fakult Univerzity Karlovy ve tvaru „`KTF ... FF PrF MFF ... FHS`“. Napište úsek skriptu, který provádí kontrolu, zda parametr **`$1`** obsahuje korektní zkratku fakulty.

Rozbor:

Vcelku přirozené řešení je použít příkaz **`grep`** a otestovat, že parametr **`$1`** jako vzor bude nalezen v textu vytvořeném rozvinutím hodnoty proměnné `fakulty`. Asi takto:

```
if echo "$fakulty" | grep -wq "$1"; then ...
```

My si však ukážeme jiný přístup, zvláště proto, že demonstruje jeden z příjemných důsledků toho, že shell je interpretovaný jazyk, neboli, že kód programu se může měnit až v průběhu práce. Na rozdíl od zvyklostí v jiných jazycích, mohou totiž v shellu být vzory v jednotlivých návěštích příkazu **`case`** *dynamické*, podléhají substitucím shellu a mohou tedy například obsahovat proměnné. V takovém případě dokonce hodnota textu „testovaného“ příkazem **`case`** může být pevná (náš konstantní seznam fakult) a my budeme ověřovat, zda tento pevný text má tvar „něco...hledaná_zkratka...něco“.

Řešení:

```
case " $fakulty " in
*" $1 "* ) ;;
* ) echo "Neplatna zkratka fakulty: $1"; exit;;
esac
```

Poznámky:

Mezery kolem zápisů „**`$fakulty`**“ a „**`$1`**“ představují jednoduchý trik, jak testovat zkratku jako celé slovo. Místo výskytu řetězce „`FF`“ (který najdeme i uvnitř zkratky „`MFF`“) budeme testovat „ `FF` “, kde záměna nehrozí. Kvůli správnému testování první a poslední zkratky v seznamu musíme mezery přidat i do řetězce „ **`$fakulty`** “.

A oba řetězce musíme kvůli mezerám uzavřít do uvozovek.

Porovnávání vzorů s testovaným výrazem v příkazu **`case`** probíhá na celé jeho délce. To jistě není překvapení, expanzní znaky shellu se chovají stejně. Proto jsme museli na začátek a konec vzoru připsat hvězdičky (to už jsme ostatně dělali i v minulé kapitole). Tím jsme vlastně způsobili, že efektivně testujeme podřetězec.

Naproti tomu v ukázce s příkazem **`grep`** stačilo jako vzor zapsat pouze zkratku fakulty, neboť u regulárních výrazů se testuje shoda podřetězce (to už také dávno víme).

4.21. Cyklus s překvapením

Zadání:

Napište program, který ve všech zdrojových souborech jazyka C (`*.c`) v aktuálním adresáři aktualizuje text „`Copyright` *rok*`-2008`“ na „`Copyright` *rok*`-2009`“ (pro libovolný *rok*). Můžete předpokládat, že text se vyskytuje jen na jedné řádce.

Rozbor:

Máme za úkol editovat soubory, přirozeným řešením tedy bude použít editor **`ed`**. Vymyslet správné editační příkazy a předat je editoru už také umíme.

Zbývá zjistit, jak spustit editor *opakovaně*. A na to má shell cyklus **`for`**. Tělo cyklu se postupně provádí pro každé ze slov zadaných jako součást příkazu. Necháme si tedy od shellu v textu příkazu nahradit výraz „**`*.c`**“ za seznam jmen souborů a pro každé slovo ze seznamu (jeden soubor) provedeme potřebné příkazy v těle cyklu (do proměnné `soubor` se nám přitom postupně dosazují jejich jména).

V těle cyklu nejprve otestujeme, zda se v souboru vůbec hledaný řetězec vyskytuje. Pokud ne, příkazem **`continue`** ukončíme krok cyklu a pokračujeme dalším slovem (souborem). V opačném případě vyvoláme editor a soubor aktualizujeme.

Řešení:

```
for soubor in *.c; do
    grep -q 'Copyright [0-9]*-2008' "$soubor" || continue
    ed -s "$soubor" <<- KONEC
        /\(Copyright [0-9]\{4\}-200\)8/s//\19/
        w
        KONEC
done
```

Poznámky:

Konečně cyklus! A ještě přímo šitý na míru našemu problému. Opravdu, cyklus **`for`** v této podobě je jedním ze základů práce v shellu. Úloha projít všechny soubory nějaké množiny a provést na nich shodnou operaci je naprosto typická. A proto se dokonce dělá často přímo na příkazové řádce při interaktivní práci s shellem...

Přesné chování cyklu je následující: shell provede na řádce potřebné úpravy (v našem případě dosadí jména všech zdrojových souborů), poté vezme text za klíčovým slovem **`in`**, rozdělí ho na jednotlivá slova a pro každé z nich jednou provede tělo cyklu s tím, že aktuální slovo uloží jako hodnotu *řídicí proměnné* cyklu (tj. proměnné `soubor`).

Přitom neplatí omezení týkající se počtu parametrů příkazové řádky, náš skript by tedy fungoval v adresáři s jakýmkoliv počtem souborů. Otázkou je, jak rychle...

Napsat v shellu „klasický“ cyklus (např. „opakuj od jedné do čtyř“) pomocí příkazu **`for`** příliš dobře nejde. Řešení „**`for i in 1 2 3 4`**“ ještě ujde, ale pokud horní hranice bude zadaná proměnnou, pak jednoduché a přenositelné řešení zatím neumíme.

Klíčová slova „**`for`**“, „**`do`**“ a „**`done`**“ opět trochu připomínají samostatné příkazy – zapsat je na jednu řádku smíme jen oddělené středníkem. Ve skutečnosti ovšem tvoří jedinou řídicí strukturu, a proto třeba přesměrování pro celý cyklus se píše až za „**`done`**“.

Občas vídám pokusy psát na konec cyklu slovo „**`od`**“. V programovacím jazyce ALGOL, jednom ze vzorů pro jazyk shellu, tomu tak doopravdy bylo. Koncepčně by asi nejsprávnější bylo „**`rof`**“, jenže to by zase každý cyklus (ještě jich několik poznáme) musel končit jiným klíčovým slovem. Zvolené klíčové slovo „**`done`**“ je tedy jakýmsi kompromisem.

Zastavme se ještě u těla cyklu. Použili jsme nejprve příkaz **`grep`** pro ověření, zda soubor vůbec hledanou řádku obsahuje. To sice znamená, že soubory, které budeme editovat, musí náš program číst dvakrát (jednou příkazem **`grep`**, jednou editorem), ale úspora získaná tím, že část souborů vůbec nebudeme editovat, se nám určitě vyplatí. Víme, že editor si pořizuje kopii souboru, což trvá dlouho, zatímco program **`grep`** je velmi rychlý. Nemluvě o tom, že zbytečně neměníme datum a čas poslední modifikace souborů, kterým neměníme obsah.

Vyvolání editoru na soubory, které hledaný řetězec neobsahují, by způsobilo ještě jeden problém. Příkaz *substitute* v té podobě, jak jsme ho použili, by totiž selhal. Pokud použijeme jako adresu příkazu regulární výraz, musí nějaká vhodná řádka v souboru existovat, jinak editor ohlásí chybu. Této nepříjemnosti bychom se ovšem mohli zbavit i jinak – například tím, že bychom použili příkaz *global*:

```
g/\(Copyright [0-9]*-200\)8/s//\19/
```

Příkaz *global* by sice také žádnou řádku k vykonání příkazu *substitute* nenašel, ale přešel by to v tichosti.

Výsledek příkazu **`grep`** ověříme pomocí operátoru „**`||`**“ – pokud příkaz nic nenajde, zavoláme příkaz **`continue`**. Jeho smyslem je, že shell ukončí vykonávání *těla* cyklu a zahájí novou iteraci (další krok) cyklu. U cyklu **`for`** to znamená, že vezme ze seznamu další slovo a vykoná tělo cyklu.

4.22. Cyklus bez překvapení

Zadání:

Vypište UID a plné jméno uživatele, který byl založen bezprostředně před vámi (jeho řádka v souboru `/etc/passwd` předchází vaší řádce).

Rozbor:

Podobné problémy jsme řešili v části o editorech. I zde potřebujeme postupně číst řádky, uchovávat si údaje z poslední z nich a při nalezení správné řádky údaje vypsat.

Na čtení souboru se ale hodí jiný cyklus, než jsme poznali minule, a to **`while`**. Asi už nepřekvapí, že podobně jako za **`if`** se za klíčové slovo **`while`** píše příkaz, který shell na začátku každého kroku cyklu vykoná, a skončí-li jeho běh úspěšně (tj. s návratovou hodnotou nula), pokračuje prováděním příkazů v těle cyklu.

Pro naše účely se hodí příkaz **`read`**, který se pokusí přečíst řádku ze svého vstupu a návratovou hodnotu nastavuje podle toho, zda se mu to povedlo nebo ne. Jenom mu budeme muset vstup přesměrovat, aby četl řádky ze souboru `/etc/passwd`.

Navíc by se nám hodilo, aby příkaz **`read`** rozdělil řádky `/etc/passwd` na jednotlivá pole. Víme, že to umí, jenom máme nevhodné oddělovače polí. Naštěstí **`read`** umožňuje oddělovače změnit, a to stejně jako v shellu samotném pomocí proměnné **`IFS`**.

V těle cyklu budeme ukládat naposledy přečtené jméno a UID do dvou proměnných. Jakmile narazíme na řádku se správným jménem uživatele, poslední uložené hodnoty vypíšeme a cyklus ukončíme příkazem **`break`**.

Řešení:

```
while IFS=: read login pwd uid gid full home shell; do
    [ "$login" = forst ] && echo $lastuid $lastfull && break
    lastuid=$uid; lastfull=$full
done < /etc/passwd
```

Poznámky:

Příkaz **`read`** čte standardní vstup a nenulovou návratovou hodnotu vrací, pokud se mu čtení nepovede, třeba když narazí na konec. Díky tomu se skvěle hodí pro přečtení nějakého uceleného vstupu (souboru, proudu dat z roury apod.) v cyklu **`while`**.

Všimněte si, jak jsme vyřešili přesměrování vstupu pro **`read`**. Napsali jsme ho až za klíčové slovo **`done`**. Tím se přesměruje vstup pro celý cyklus. Pokud byste napsali:

```
while read login ... < /etc/passwd; do
```

nebudete s výsledkem asi spokojeni. Tímto způsobem totiž přesměrujete pouze vstup konkrétního příkazu **`read`**, ten přečte první řádku ze souboru a cyklus pokračuje. Když se ale v dalším kroku cyklu opět volá stejný příkaz **`read`**, opět se provede (nové) přesměrování a opět se přečte první řádka souboru.

Pro čtení souboru by se sice teoreticky dal použít i cyklus **for**:

```
for x in `cat /etc/passwd`; do
```

ale je to velmi nepohodlné. Předně bychom do proměnné **IFS** museli nastavit samotný znak konce řádky (jinak by se cyklus opakoval pro každé slovo a ne řádku). Ale hlavně bychom přišli o možnost nechat si řádku od příkazu **read** rozdělit na slova.

A tuto možnost jsme zde dokonale využili. Příkaz **read** rozděluje text přečtené řádky na slova (a tedy do proměnných) podle týchž pravidel jako shell sám – jako oddělovače polí používá znaky uložené v proměnné **IFS**. Díky nastavení hodnoty na dvojtečku dostaneme „zadarmo" přiřazení hodnot polí `/etc/passwd` do jednotlivých proměnných.

Naštěstí se **read** také úplně stejně staví ke slučování oddělovačů. Posloupnost bílých znaků by tvořila jediný oddělovač pole, zatímco u dvojteček tvoří hranici slova každý výskyt. Kdyby tomu tak nebylo a nějaký uživatel by měl prázdné heslo, byly by v jeho řádce dvě dvojtečky za sebou (kolem prázdného hesla), příkaz **read** by je chápal jako jediný oddělovač pole a význam sloupců by se zcela posunul.

Pokud byste naopak řádku dělit na slova nechtěli a potřebovali byste ji dostat přesně tak, jak byla na vstupu, budete si muset hodnotu proměnné **IFS** nastavit na prázdný řetězec. Jinak totiž přijdete o úvodní mezery na řádce.

Příkaz **read** dovoluje v textu načítané řádky vyznačit, že některé znaky nemají mít význam oddělovače polí. Udělá se to konzistentně – před znak se zapíše zpětné lomítko. Potíž je ale v tom, že díky tomu se všechna zpětná lomítka ve vstupu interpretují jako metaznaky a je třeba je tedy zdvojovat. Podobnou úlohu má i zpětné lomítko na konci řádky – příkaz **read** automaticky načte další řádku a text obou řádek spojí, ještě než je začne dělit na slova. Naštěstí lze tuto funkci vypnout přepínačem „**-r**" (*raw*).

V řešení tohoto příkladu jsme použili poněkud jiný způsob nastavení proměnné **IFS**. Přiřazení ve tvaru „*proměnná*=*hodnota příkaz*" způsobí, že proměnná se nastaví pouze jako součást prostředí (environmentu), které se předá zadanému příkazu. V aktuálním shellu se nastavení proměnné vůbec neprojeví. Proto v našem řešení bude sice příkaz **read** pracovat se změněnou hodnotou proměnné **IFS**, ale přitom se vůbec nemusíme starat o „opravu" její hodnoty zpět na původní obsah.

Zkuste si podobně vyřešit nastavení proměnné **IFS** v příkladu z kapitoly 4.7:

```
IFS=: find $PATH -name sort | head -n1
```

To fungovat nebude. Příkaz **find** sice poběží se změněnou hodnotou **IFS**, ale expanzi proměnné **PATH** a následné rozdělení její hodnoty na pole dělá aktuální shell, který má v proměnné **IFS** pořád nezměněnou hodnotu. Možné řešení si ukážeme v kapitole 4.28.

Příkaz **break** ukončí provádění cyklu a shell bude pokračovat za cyklem – prvním příkazem za klíčovým slovem **done**.

4.23. Cyklus, jak ho (ne)známe

Zadání:

Napište skript, který dostane jako parametr číslo *n* a vytvoří dočasné soubory se jmény `/tmp/soubor-`*x*, kde *x* budou čísla od jedné do *n*.

Rozbor:

Tato úloha naráží na problém, který jsme odhalili v kapitole 4.21. Ve většině jazyků by řešením byl cyklus **`for`** (od jedné do *n*), ale v shellu to bude cyklus **`while`**.

Podmínka cyklu je jednoduchá, potřebujeme otestovat, že hodnota řídicí proměnné cyklu je menší nebo rovna `n`. To už umíme pomocí příkazu **`test`**.

Těžší bude zvyšování této hodnoty mezi jednotlivými kroky cyklu. Říkali jsme si, že proměnné v shellu mají pouze textový charakter. V původním konceptu shellu nebyla aritmetika skutečně vůbec zastoupena. Po pravdě řečeno, probíráme spolu v této knize již poměrně dlouho celou řadu úloh a na potřebu aritmetiky jsme zatím téměř nenarazili. Ve shodě s tímto pozorováním byla do shellu aritmetika začleněna jen ve formě utility **`expr`** (*expression*). Ta dostává v parametrech zadaný výraz, spočítá výsledek a vypíše ho na výstup. A ten můžeme pomocí zpětných apostrofů uložit do proměnné.

Skalní staromilci by namítli, že aritmetiku nepotřebujeme ani teď. Shell totiž umí kromě pozičních parametrů počítat ještě další věc, a to *délku hodnoty* proměnné. Má na to zvláštní konstrukt „**`${#`***jméno***`}`**“. Stačilo by tedy do řídicí proměnné cyklu dát řetězec a postupně ho o jeden znak prodlužovat:

Řešení (s aritmetikou):

```
n=$1; x=1
while [ $x -le "$n" ]; do
    touch /tmp/soubor-$x
    x=`expr $x + 1`
done
```

Řešení (s délkou proměnné):

```
n=$1; x=@
while [ ${#x} -le "$n" ]; do
    touch /tmp/soubor-${#x}
    x=@$x
done
```

Poznámky:

Proměnnou `x` jsme v příkazu **`test`** („**`[ $x -le "$n" ]`**“) zapsali bez uvozovek, ačkoliv jsme vás před tím varovali. Ale zde se nemáme čeho obávat, protože víme jistě, že v proměnné je číslo (sami ho tam píšeme).

Jednotlivé složky výrazu pro příkaz **expr** je nutné psát jako samostatné parametry (tj. oddělené mezerami), zápis „**$x+1**“ by příkaz **expr** nepochopil.

Ve výrazech lze používat závorky a následující operátory:

- aritmetické: **+**, **-**, *****, **/**, **%** (zbytek po dělení);
- porovnávací: **<**, **<=**, **>**, **>=**, **=**, **!=**; porovnávání probíhá buď numericky (pokud oba operandy lze chápat jako číslo), nebo lexikograficky;
- řetězcové: **:** (shoda s regulárním výrazem); výraz je vždy ukotven na začátek řetězce (výraz „**$x : a**“ je tedy zcela shodný s výrazem „**$x : ^a**“);
- logické: **|** (nebo), **&** (a zároveň).

Při psaní výrazů je třeba dávat pozor na metaznaky shellu. Co asi udělá příkaz:

```
sinek:~> expr $x * 2 > 27
```

Shell dosadí za hvězdičku seznam souborů, z čehož bude příkaz **expr** řádně zmatený. Navíc jeho standardní výstup shell přesměruje do souboru jménem `27`.

Některé implementace příkazu **expr** mají přepínače. Těm bude vadit i záporné číslo na začátku výrazu. Opět si lze pomoci „selským rozumem“, třeba:

```
sinek:~> expr \( $x \* 2 \> 27 \)
```

Pro používání proměnných platí totéž, co jsme říkali u příkazu **test** – raději všude pište uvozovky. Nicméně ani to nám zde nemusí pomoci, pokud v proměnné bude něco, co příkaz pochopí jako operátor, třeba řetězec „=“. Pak si pomůžeme prastarým trikem – doplníme oba řetězce o nějaký shodný text, abychom kolizi vyloučili:

```
sinek:~> expr "X$x" = "X$y"
```

Příkaz **expr** vždy generuje standardní *výstup* a vrací *návratovou hodnotu*. Je třeba si pečlivě rozmyslet, co z toho potřebujeme. V našem příkladu jsme potřebovali jen výstup a návratová hodnota nás nezajímala. Pokud příkaz **expr** použijeme pro porovnávání, bude nás naopak zajímat spíše návratová hodnota. V tom případě ale nesmíme zapomenout standardní výstup příkazu „zlikvidovat“:

```
sinek:~> if expr "$jmeno" \> "novak" > /dev/null; then ...
```

Navzdory původní koncepci se postupem času ukázalo, že aritmetika se přece jen občas i v shellu hodí a novější verze shellu se snažily tuto díru nějak „zacpat“. V Korn shellu se proto objevil konstrukt aritmetického výrazu „**$((***výraz***))**“, který později pronikl i do normy. Náš příkaz by se pomocí něj zapsal:

```
x=$(($x + 1))
```

Jeho výhodou je vyšší rychlost a také odpadá nutnost ošetřovat metaznaky shellu (výraz v závorkách vyhodnocuje přímo shell, a proto ví, že třeba hvězdička neznamená expanzi souborů ale násobení). Nevýhodou je opět riziko omezené přenositelnosti.

4.24. Potíže s relativitou

Zadání:

Napište skript, který dostane jako parametr jméno symbolického linku a vypíše „dlouhý“ výpis o cílovém souboru (jako příkaz **`ls`** s přepínačem „**`-L`**“, přesněji „**`-ldL`**“).

Použití příkazu **`ls`** je povoleno, pochopitelně ovšem pouze bez přepínače „**`-L`**“.

Ukázka „normálního“ výpisu:

```
-rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ll
lrwxr-xr-x  1 forst  forst  2 Nov 27 02:06 ltr.s -> ll
```

a tvar výpisu s přepínačem „**`-L`**“:

```
-rwxr-xr-x  2 forst  forst  8 Nov 27 02:05 ltr.s
```

Pokud cílový soubor neexistuje, chová se výpis stejně jako bez přepínače „**`-L`**“.

Rozbor:

Řešení se na první pohled může zdát docela jednoduché. Příkazem **`ls`** umíme vypsat cíl symbolického linku a jeho data vypíšeme také pomocí **`ls`**. Ale dva problémy zbývají.

Cíl symbolického linku může být zase symbolický link, takže v hledání skutečného jména cílového souboru budeme muset pokračovat, dokud nedorazíme na konec řetězu odkazů. Na to se bude hodit cyklus **`while`**, v němž budeme příkazem **`test`** kontrolovat typ aktuálního cílového souboru.

Správně by skript měl ještě kontrolovat vzájemné „zacyklení“ linků, jak jsme o něm hovořili v kapitole 2.4. Podobný problém ale budeme řešit v kapitole 6.2, a tak si zde tuto komplikaci odpustíme.

Drobné potíže působí odkazy ve formě relativní cesty. Jejich relativita je vázána na adresář, kde symbolický link leží, takže pro ně musíme nejprve najít skutečné umístění cílového souboru – spojit cestu k adresáři se symbolickým linkem a cestu uloženou jako odkaz do jednoho řetězce. Nejsnazší asi bude po celou dobu pracovat jen s absolutními cestami. Pokud je parametrem skriptu jméno souboru s absolutní cestou, uložíme si ho do proměnné `cesta`. Pokud je parametr relativní, přidáme před něj navíc ještě aktuální adresář. V těle cyklu budeme postupovat podobně, pouze budeme namísto aktuálního používat adresář, kde aktuální symbolický link leží (potřebnou cestu získáme tak, že pomocí editoru **`sed`** z plné cesty k souboru umažeme vše počínaje posledním lomítkem).

V okamžiku, kdy cyklus skončí, a tedy v proměnné `cesta` bude název souboru, který už není symbolický link, vypíšeme informaci o tomto souboru. Ve výpisu pouze nahradíme jméno cílového souboru jménem, které jsme dostali na začátku, tak jak to dělá i **`ls`**. Pokud cílový soubor neexistuje, jen zavoláme **`ls`** s původním parametrem.

Řešení:

```
adr=`pwd`
puvodni=$1
case $1:$adr in      # test parametru a aktualniho adresare
/* ) cesta=$1;;      # parametrem je absolutni cesta
*/ ) cesta=/$1;;     # aktualni adresar je koren
*  ) cesta=$adr/$1;; # slouceni adresare a relativni cesty
esac
while [ -L "$cesta" ]; do
    odkaz=`ls -ld $cesta | sed "s:.* $cesta -> ::"`
    case $odkaz in
    /* ) cesta=$odkaz;;
    * )  cesta=`echo "$cesta" | sed 's:/[^/]*$::'`/$odkaz;;
    esac
done
if [ -e "$cesta" ]; then
    ls -ld "$cesta" | sed "s:$cesta\$:$puvodni:"
else
    ls -ld "$puvodni"
fi
```

Poznámky:

Všimněte si způsobu testování obsahu zadaného parametru a aktuálního adresáře. Oba testy jsme spojili do jediného příkazu **case** namísto testování pomocí několika zanořených příkazů **if**. Čeho potřebujeme dosáhnout? Pokud je zadaná cesta („**$1**“) absolutní, není ji třeba nijak upravovat. Pokud je relativní, záleží na aktuálním adresáři („**$adr**“). Je-li to kořen systému souborů, pak **pwd** vypíše lomítko a lomítko je také třeba před zadanou cestu přidat. Je-li to nějaký jiný adresář, výpis příkazu **pwd** nekončí lomítkem, oba řetězce musíme spojit a přidat mezi ně lomítko.

A přesně to jsme zakódovali do jednotlivých větví **case**. Jako testovaný řetězec jsme spojili oba testované výrazy (**$1** i **$adr**), takže návěští vyjadřují sloučené podmínky pro oba testované výrazy. Vzor v první větvi uspěje, pokud **$1** začíná lomítkem (neboli je absolutní), aktuální adresář je pak už nezajímavý. Druhá větev se použije, když je cesta relativní (mohli bychom to napsat explicitně pomocí „**[!/]*/**“, ale cesty obsahující lomítko začátku nám tady „spolkne“ první větev) a zároveň se nacházíme v kořeni stromu adresářů (jediný případ, kdy **$adr** končí lomítkem). Třetí větev představuje „zbytek“ případů, tedy relativní cestu k souboru a nekořenový aktuální adresář.

Takovéhle triky mohou být užitečné a efektivní, ale pro lepší srozumitelnost se určitě vyplatí je v kódu vysvětlit pomocí *komentáře*. Podobně jako v editoru **sed** se zapisuje pomocí znaku „**#**“ – následující text až do konce řádky shell ignoruje.

V těle cyklu potřebujeme z výpisu příkazu **`ls`** zjišťovat odkaz, kam symbolický link směřuje. Použijeme editor **`sed`** a začátek řádky až po „šipku“ vymažeme. Náš starý známý záškodník by nám zde mohl nadělat problémy, kdyby vyrobil soubory, v jejichž jméně by se v nějaké šikovné kombinaci šipka objevila (např. symbolický link „`a`“ odkazující na soubor „`a -> b`“). Pak by naše substituce selhala. Na ochranu proti takovým případům bychom museli regulární výraz sestavovat mnohem pečlivěji.

V příkazech pro editor **`sed`** jsme potřebovali použít proměnné shellu. Proto jsme příkazy uzavřeli do uvozovek a ne do apostrofů, aby shell proměnné substituoval.

Krom toho jsme museli zvolit jiný oddělovač místo lomítka, protože lomítky se to v používaných proměnných bude jen hemžit. Zrádné je ovšem to, že v zápisu příkazů žádná lomítka vidět nejsou. Dostanou se tam až dosazením hodnot proměnných. Zvolili jsme dvojtečku, což sice není zcela bezpečné, ale riziko chyby je malé.

Stejný problém by nám způsobily i metaznaky regulárních výrazů, pokud by se objevily někde ve jménech souborů. Opět platí, že hranaté závorky, zpětná lomítka nebo hvězdičky rozumní lidé ve jménech souborů nepoužívají. Jiná je situace s tečkou. Ta je rovněž metaznakem regulárních výrazů a ve jménech souborů se jistě vyskytovat bude. Zde to však nevadí, protože regulární výraz používáme vlastně jen pro vymezení části řádky obsahující plnou cestu k souboru. A třeba regulární výraz „**`a/b.c`**“ by sice vyhovoval i jiným jménům souborů než právě „`a/b.c`“, ale ta se v řádce s výstupem příkazu „**`ls -l a/b.c`**“ vyskytovat nebudou.

Pokud bychom se rizika výskytu vybraného oddělovače ve jméně souboru báli, museli bychom řetězce používané v editačních příkazech upravit přidáním zpětného lomítka před každý potenciální problematický znak. Místo proměnné `cesta` bychom použili další proměnnou (třeba `cesta_ed`) a do ní zkopírovali patřičně upravenou hodnotu. Potřebný editační příkaz pro tuto úpravu by vypadal například takto:

```
echo "$cesta" | sed 's:\([/\.[*]\):\\\1:g'
```

Ve vzoru příkazu *substitute* vidíme podvýraz v kulatých závorkách a v něm v hranatých závorkách následující seznam znaků: lomítko, zpětné lomítko, tečku, levou hranatou závorku a hvězdičku. Připomeňme, že uvnitř seznamu znaků v hranatých závorkách žádný z těchto znaků není metaznakem, a proto je stačí jen vyjmenovat, aniž bychom je museli uvozovat zpětným lomítkem. V náhradě vidíme zpětné lomítko (tam ovšem zdvojené být musí) a zpětnou referenci („**`\1`**“).

Ale pozor, abychom výsledek toho příkazu uložili do proměnné, budeme příkaz volat pomocí zpětných apostrofů a mezi nimi je žádoucí opět každé zpětné lomítko zdvojit! Přesněji řečeno, zdvojit musíme pouze ty, za nimiž následuje další zpětné lomítko:

```
cesta_ed=`echo "$cesta" | sed 's:\([/\.[*]\):\\\\\1:g'`
```

V takovém případě je ale opravdu trochu problémem dopočítat se správného počtu zpětných lomítek, a tak je skoro lepší raději zdvojit všechna zpětná lomítka...

```
cesta_ed=`echo "$cesta" | sed 's:\\([/\\.[*]\\):\\\\\\1:g'`
```

4.25. Amoeba

Zadání:

Vytvořte skript se dvěma jmény, `min` a `max`, který bude dostávat jako parametry sérii čísel a spočítá jejich minimum, resp. maximum podle toho, kterým jménem byl zavolán.

Rozbor:

Zpracování parametrů, které dostal váš skript, patří mezi základní úkony, jež je třeba při programování v shellu zvládnout, a zároveň je to dobré cvičení na práci s řídicími strukturami shellu. Vyzkoušíme si dva přístupy.

První variantou je cyklus **`while`**, opakovaný tak dlouho, dokud jsou k mání nějaké parametry (neboli dokud je hodnota **`$#`** kladná). V každém kroku otestuje první parametr a provede **`shift`** (který mj. sníží hodnotu **`$#`**).

Druhou variantou je cyklus **`for`** v jeho dalším typickém nasazení – bez klíčového slova **`in`**. V této podobě cyklus prochází postupně všechny *poziční parametry*.

Pro správné měňavkové fungování budeme potřebovat rozskok podle jména skriptu. Naučíme se jeden zajímavý trik, který vychází z vlastností shellu, konkrétně z toho, že se jedná o textový preprocesor. Rozskok nebudeme dělat před každým porovnáním. Namísto toho upravíme přímo *operátor*, pomocí něhož budeme porovnávat.

Řešení (`while`):

```
[ `basename $0` = min ] && operator=-lt || operator=-gt
minimax=$1; shift
while [ $# -gt 0 ]; do
    [ $1 $operator $minimax ] && minimax=$1
    shift
done
echo $minimax
```

Řešení (`for`):

```
[ `basename $0` = min ] && operator=-lt || operator=-gt
minimax=$1; shift
for cislo; do
    [ $cislo $operator $minimax ] && minimax=$cislo
done
echo $minimax
```

Poznámky:

Cyklus ve druhém řešení by bylo možné zapsat explicitně

```
for cislo in "$@"; do
```

Obě řešení představují standardní způsob, jak zpracovávat parametry skriptu. Proto také není zapotřebí odkazovat se na více než devět pozičních parametrů („**$1**“ až „**$9**“).

Úprava vlastního kódu během práce programu je poměrně silný a velice nebezpečný nástroj. V běžných jazycích máme dnes takové programátorské nástroje, že podobné triky už v nich nejsou zapotřebí a nejsou ani podporované.

Naproti tomu shell jako textový preprocesor nám umožňuje tento trik používat, a to vcelku bezpečným způsobem – máme totiž po ruce poměrně solidní nástroj umožňující sledovat, co zrovna shell dělá, jak přesně vypadá řádka, kterou se chystá provádět.

Uvědomte si, že shell, jako interpretovaný jazyk, je velmi citlivý na velikost kódu. Každá řádka či příkaz navíc může (pravda, nikoliv v tomto případě) mít fatální důsledky pro rychlost. Takže pokud podobné triky pomohou, je dobré je používat. Nezapomeňte ale současně na to, že v programu se nakonec musí někdo vyznat...

Skripty v této kapitole už pomalu začínají dosahovat míry, kdy se vám mohou hodit aspoň stručné informace o ladění shellových skriptů. Už dříve jsme se zmiňovali o jedné variantě spouštění skriptů pomocí explicitního zavolání shellu se jménem skriptu jako parametrem. Tato forma má jednu přednost – je možné takto nastavit shellu nějaké přepínače. Nás bude zajímat zejména přepínač „**-x**“ (*execute trace*), díky němuž bude shell opisovat na standardní chybový výstup každý příkaz, který provádí:

```
sinek:~> sh -x ./max 1 2 3
+ basename ./max
+ [ max = min ]
+ operator=-gt
+ minimax=1
+ shift
+ [ 2 -gt 1 ]
+ minimax=2
+ [ 3 -gt 2 ]
+ minimax=3
+ echo 3
3
```

Podobně může pomoci přepínač „**-v**“ (*verbose*), který pro změnu vypisuje prováděné řádky tak, jak je shell postupně ze skriptu čte. A důležitý je také přepínač „**-n**“ (*noexec*), který způsobí, že shell skript jen přečte a provede syntaktickou kontrolu.

4.26. Jak na parametry

Zadání:

Napište skript, který bude rozeznávat přepínače „`-n`“ a „`-c`“ a operandy příkazu **`tail`**, bude kontrolovat existenci zadaných souborů a pro každý existující soubor skutečně zavolá **`tail`** s danými přepínači.

Rozbor:

Základní metody jsme si vyzkoušeli v minulé kapitole, nyní je třeba jen odlišit různé druhy parametrů – přepínače a operandy (jména souborů).

Pro procházení zadaných přepínačů se hodí cyklus **`while`**, neboť můžeme pružněji pracovat se seznamem parametrů. V každém kroku cyklu otestujeme jeden parametr, a jakmile ho zpracujeme, provedeme příkaz **`shift`**, který parametr „zahodí“. V případě, že to byl přepínač s hodnotou uloženou v následujícím parametru, zahodíme rovnou parametry oba (příkazu **`shift`** řekneme, kolik pozičních parametrů má zahodit). Pokud narazíme na parametr „**`--`**“ nebo na první operand, cyklus ukončíme příkazem **`break`**. Jinak budeme cyklus opakovat, dokud nezpracujeme všechny parametry – podmínku nenulového počtu parametrů nám ověří příkaz **`test`**.

Zpracování parametru závisí na jeho formátu, jenž se dobře rozlišuje pomocí **`case`**:

- Pokud parametrem je některý známý přepínač („`-c`“ nebo „`-n`“), ale nemá přímo připojenou hodnotu, musí tato následovat v dalším parametru. Do proměnné `opt` uložíme přepínač i s hodnotou (spojíme **`$1`** a **`$2`**) a pokusíme se zavolat příkaz **`shift`** s parametrem „**`2`**“, aby posunul seznam pozičních parametrů o dvě místa. Použijeme přitom i výsledek (návratovou hodnotu) tohoto příkazu. Pokud příkaz uspěje, hodnota pro přepínač byla opravdu zadána, zavoláme příkaz **`continue`** a můžeme začít další krok cyklu. Pokud však příkaz selže, hodnota u přepínače chyběla, skript vypíše chybu a ukončí se příkazem **`exit`**.
- Pokud parametrem je některý známý přepínač a má za sebou přímo připojenou hodnotu, uložíme celý parametr do proměnné `opt` a provedeme příkaz **`shift`**.
- Pokud parametrem je sice přepínač, ale není známý, skript vypíše chybu a skončí. Tuto podmínku vyjádříme v **`case`** návěštím „**`-*`**“. To sice zahrnuje jakékoliv pokračování po znaku „`-`“, ale případy „**`-c`**“ a „**`-n`**“ už jsme vyřešili výše.
- Pokud narazíme na zvláštní parametr „**`--`**“, znamená to, že přepínače skončily. Zahodíme parametr a cyklus ukončíme příkazem **`break`**. Větev **`case`** pro tento případ ovšem musí být zapsána před větví „**`-*`**“, jinak by se nikdy neuplatnila.
- Cyklus ukončí také libovolný operand (parametr, který nezačíná minusem).

V této chvíli máme zpracované všechny přepínače.

Zbývající parametry (jména souborů) projdeme druhým cyklem, tentokrát se více hodí cyklus **for** (bez klíčového slova **in**). Jeho tělo už je jednoduché. Otestujeme, zda soubor existuje, a zavoláme příkaz **tail** nebo vypíšeme chybovou zprávu.

Řešení:

```
while [ $# -gt 0 ]; do
    case $1 in
    -[cn]  ) opt="$1$2"
        shift 2 && continue
        echo "Chybi hodnota pro $1" && exit;;
    -[cn]* ) opt="$1"; shift;;
    --     ) shift; break;;
    -*     ) echo "Neznamy prepinac $1"; exit;;
    *      ) break;;
    esac
done
for soubor; do
    if [ -f "$soubor" ]; then
        tail $opt $soubor
    else
        echo "Soubor '$soubor' neexistuje"
    fi
done
```

Poznámky:

Pro pořádek poznamenejme, že pokud skript nedostane žádné přepínače, nebude nastavena hodnota proměnné **opt** a každý příkaz **tail** bude zavolán pouze se jménem souboru. Což ale přesně odpovídá tomu, jak by se měl skript chovat.

Náš skript nekontroluje, zda hodnotou přepínače je číslo, a neumí správně obsloužit volání beze jména souboru. Doplňte si ošetření těchto případů jako cvičení.

Příkaz **continue** v cyklu **while** znamená, že se znova provede příkaz uvedený za **while**, testuje se jeho výsledek a případně se provede další krok cyklu.

Způsob zpracování přepínačů, který jsme zde představili, je ale značně nepohodlný, zvláště když chcete umět i „slučování“ přepínačů (tj. dovolit při volání vašeho skriptu třeba zápisy jako „**ls -lt**“ místo „**ls -l -t**“). Norma proto nabízí pomocnou ruku s příkazem **getopts**. Je zde sice opět riziko přenositelnosti na starší stroje, ale pokud potřebujete zpracovávat opravdu složitou strukturu přepínačů, budete se k příkazu nejspíše muset uchýlit.

První cyklus bychom s pomocí **getopts** mohli napsat takto:

```
while getopts :c:n: name; do
    case $name in
    [cn] ) opt="-$name$OPTARG";;
    \?   ) echo "Neznamy prepinac $OPTARG"; exit;;
    :    ) echo "Chybi hodnota prepinace $OPTARG"; exit;;
    esac
done
shift `expr $OPTIND - 1`
```

Příkaz **getopts** postupně zpracovává přepínače, jež najde v pozičních parametrech podle instrukcí, které dostane jako svůj první parametr. Jeho hodnotou je řetězec obsahující přípustná písmena přepínačů. Za každým písmenem může být ještě dvojtečka na znamení, že přepínač má povinný argument (to v našem případě byla obě písmena „c“ i „n“). Druhým parametrem příkazu je jméno proměnné, do níž **getopts** při každém vyvolání uloží znak právě zpracovaného přepínače. Pokud má přepínač povinný argument, jeho hodnota bude uložena do proměnné **OPTARG**. Jakmile **getopts** projde všechny poziční parametry nebo narazí na první operand, vrací nenulovou návratovou hodnotu a cyklus skončí. Současně příkaz nastaví proměnnou **OPTIND** tak, že bude obsahovat číslo parametru, který se má dále zpracovávat. Obvykle tedy po skončení cyklu následuje posun parametrů (příkazem **shift**) o „**$OPTIND - 1**“ pozic.

Ovlivnit se dá ještě chování příkazu při nalezení neznámého písmena přepínače nebo chybějícího argumentu u přepínače. Zjednodušeně se dá říci, že pokud první parametr **getopts** začíná dvojtečkou, příkaz se chová „tiše“, tj. uloží informace o chybě do proměnných a nechá autora skriptu, aby si sám ošetřil chyby a zpravil o nich uživatele. Pokud první parametr dvojtečkou nezačíná, příkaz vypíše implicitní chybovou zprávu. Obsah důležitých proměnných v případě chyby shrnuje následující tabulka:

	první parametr **getopts**			
	začíná dvojtečkou		nezačíná dvojtečkou	
	hodnota **$name**	hodnota **$OPTARG**	hodnota **$name**	hodnota **$OPTARG**
chybný přepínač	„?“	znak přepínače	„?“	nenastavena
chybějící argument	„:“	znak přepínače	„?“	nenastavena

4.27. Parametry lépe

Zadání:

Upravte skript z minulé kapitoly tak, aby testoval opakování zadaných přepínačů.

Rozbor:

Kontrolu přepínačů jsme minule opravdu „odflákli“. Proměnnou `opt` jsme prostě neustále přepisovali a příkaz zavolali s posledním zadaným přepínačem.

Teď bychom rádi ukládali každý přepínač zvlášť a před tím prováděli kontrolu, zda už stejný nebo rozdílný přepínač nebyl nastaven. Tento kód se ale bude muset opakovat pro oba dva přepínače. Psát identický kód dvakrát je proti všem zásadám správného programování. Ne proto, že programátoři jsou od přírody leniví psát cokoliv zbytečně, ale proto, že údržba takového kódu se snadno dostane mimo kontrolu. V každém slušném jazyce máme pro takovou potřebu *funkce* a shell není výjimkou.

Napíšeme funkci `kontrola`, která bude testovat, zda proměnné `optc`, resp. `optn` už nemají přiřazenou hodnotu. Půjde to snadno, proměnné ve funkcích jsou globální.

Řešení:

```
kontrola() { # definice funkce
    [ -n "$optc" ] && echo "Uz byl zadan prepinac c" && exit
    [ -n "$optn" ] && echo "Uz byl zadan prepinac n" && exit
}
# zacatek hlavniho programu
while getopts :c:n: name; do
    case $name in
    c  ) kontrola; optc="-c$OPTARG";;
    n  ) kontrola; optn="-n$OPTARG";;
    \? ) echo "Neznamy prepinac $OPTARG"; exit;;
    :  ) echo "Chybi hodnota prepinace $OPTARG"; exit;;
    esac
done
shift `expr $OPTIND - 1`
...
        tail $optc $optn $soubor
...
```

Poznámky:

Definice funkce musí předcházet jejímu volání, proto bývá zvykem uvádět je na začátku skriptu. Nenechte se při čtení skriptu zmást tím, že se v místě definice funkce její kód bezprostředně nevykonává.

Syntakticky je volání funkce obyčejný příkaz a její kód vykonává tentýž proces, který ji zavolal. Proto třeba příkaz **`exit`** použitý v naší funkci ukončí běh celého skriptu.

4.28. Parametry do třetice

Zadání:

Upravte skript z minulé kapitoly tak, aby při kontrole přepínačů rozlišoval opakování (dvojí zadání stejného) a kolize (zadání obou).

Rozbor:

Tak pěkně jsme si v minulé kapitole ušetřili ťukání tím, že jsme vyrobili funkci, ale teď se zdá, že funkci už použít nepůjde. Potřebovali bychom totiž kód funkce nějak modifikovat podle toho, pro který přepínač byla vyvolána. Můžeme sice funkci předat jako parametr typ přepínače (resp. odpovídající písmenko), ve funkci si ho uložit do proměnné `znak`, jenže pak ještě potřebujeme na základě této hodnoty pracovat s různými proměnnými (`optc`, resp. `optn`). A neustálé rozeskakování podle hodnoty proměnné by bylo nakonec složitější než zapsání kódu rovnou do odpovídajících větví příkazu **`case`** v „hlavním programu“. Potřebovali bychom něco jako ukazatele (pointery) nebo nepřímé odkazy na proměnné. Nic takového v shellu ovšem nemáme.

Shell je ale interpretovaný jazyk. Do jisté míry si kód programu můžeme za běhu uzpůsobovat. Co kdybychom funkci skutečně přesvědčili, aby pracovala s proměnnou `optc`, resp. `optn` podle toho, jaká bude právě hodnota parametru? Potřebovali bychom zařídit, aby si shell jméno proměnné, s níž právě pracuje, sám za běhu „vyrobil“, tj. aby „jménem“ proměnné byl řetězec „`opt$znak`“. Pokud se s tím budete pokoušet chvíli experimentovat, asi se vám bude zdát, že tudy cesta nevede.

Jakmile se totiž shell dostane k tomu, že substituuje v řetězci „`opt$znak`“ hodnotu proměnné `znak`, už ho nic nedonutí, aby tento nový text („`optc`“ nebo „`optn`“) chápal znova jako jméno proměnné. Musel by se totiž v řádce vracet „zpátky“, aby tam našel dolar a za ním náš text, který by považoval za jméno proměnné.

Řešení však existuje. Jeho základem je interní příkaz shellu **`eval`**, který vezme svoje parametry, sestaví z nich text nové příkazové řádky a tu znova zpracuje. Shell bude číst řádku vlastně dvakrát a při druhém čtení už bude řetězec „`$optc`“, resp. „`$optn`“ interpretovat kýženým způsobem.

Řešení:

```
kontrola() {
    znak=$1
    eval [ -n \"\$opt$znak\" ] &&
        echo "Opakovane zadani prepinace $znak" && exit
    [ -n "$optc$optn" ] &&
        echo "Prepinace c a n se navzajem vylucuji" && exit
    eval opt$znak=\"-$znak\$OPTARG\"
}
```

```
while getopts :c:n: name; do
    case $name in
    [cn] ) kontrola $name;;
    \?   ) echo "Neznamy prepinac $OPTARG"; exit
    esac
done
```

Poznámky:

Pro lepší pochopení činnosti příkazu **`eval`** se podívejme, co se postupně odehrává s našimi dvěma řádkami obsahujícími příkazy **`eval`**. V kapitole 4.5 jsme si už popisovali jednotlivé kroky zpracování řádky, ale od té doby jsme poznali mnohem více prvků jazyka shellu a bude třeba si popis trochu upřesnit.

1. V prvním kroku shell rozdělí řádku na slova (pole), jako oddělovače přitom používá mezery, tabulátory a také operátory shellu. Postupuje zleva doprava a respektuje případný quoting (skrývání metaznaků – například řetězec v uvozovkách považuje za jedno slovo). Slova shell seskupuje k sobě tak, jak tvoří jednotlivé příkazy oddělené navzájem řídicími operátory. Podstatná je přitom jejich priorita: nejvyšší má roura, pak následuje podmíněné spuštění a nakonec středník – příkazová řádka zapsaná jako „**`a;b||c|d`**“ tedy významem odpovídá řádce „**`a; ( b || ( c | d ) )`**“. Shell si každé slovo rovnou klasifikuje, takže už v této fázi rozpoznává klíčová slova řídicích struktur, přiřazení do proměnných nebo operátory přesměrování a jejich operandy.

   V tomto kroku tedy shell oddělí operátor „**`&&`**“ a první příkaz má tvar:
   ```
   eval [ -n \"\$opt$znak\" ]
   ```

2. Druhým krokem je provedení substitucí proměnných a příkazů. Zpracovávají se přitom všechna slova na řádce, tedy i hodnoty dosazované do proměnných nebo operandy přesměrování (jména souborů).

   První dolar na řádce je skryt pomocí quotingu, a proto jedinou náhradou je dosazení hodnoty proměnné `znak`. Řekněme, že obsahuje písmeno „`c`“ a řádka se změní na:
   ```
   eval [ -n \"\$optc\" ]
   ```

3. Ve třetím kroku se výsledky substitucí dělí na pole podle hodnoty proměnné **`IFS`**.

4. Čtvrtým krokem je expanze jmen souborů.

   Oba kroky v našem případě nemají žádný efekt.

   Mimochodem, tyto dva kroky se neprovádějí nejen na řetězcích přiřazovaných do proměnných (to už jsme si říkali v kapitole 4.5), ale také na parametru příkazu **`case`**.

5. Posledním krokem je zrušení quotingu (odkrytí metaznaků).

Řádka má nyní tvar:

```
eval [ -n "$optc" ]
```

V této chvíli se shell podívá, jaký příkaz má spustit, a najde příkaz **eval**. Ten ale udělá pouze to, že svoje parametry sestaví do řádky (oddělené jednou mezerou, stejně jako to dělá příkaz **echo**) a spustí znova jednotlivé kroky algoritmu zpracování řádky:

```
[ -n "$optc" ]
```

Nyní už shell vidí řádku ve tvaru, v jakém jsme ji chtěli mít, a provede tedy potřebnou substituci hodnoty proměnné `optc`. Tím je definitivní tvar řádky hotov a shell spustí příkaz, který se na ní nachází (je to příkaz **test**, resp. „**[**“).

A jak je na tom náš druhý příkaz **eval**?

```
eval opt$znak=\"-$znak\$OPTARG\"
```

V prvním kroku nedojde k žádné změně, po provedení substitucí má řádka tvar:

```
eval optc=\"-c\$OPTARG\"
```

Při pohledu na tuto řádku by se mohlo zdát, že vlastně příkaz **eval** nepotřebujeme. Vždyť odmyslíme-li si zpětná lomítka, řádka už vypadá přesně tak, jak ji chceme mít – je to přiřazení do proměnné. Problém ale spočívá v tom, že rozhodnutí, zda první slovo na řádce představuje přiřazení do proměnné nebo ne, padne ještě v rámci prvního kroku zpracování řádky, a tedy před provedením substitucí. A v té chvíli začíná řádka řetězcem „**opt$znak=**...“, který přiřazení nepředstavuje. Proto je příkaz **eval** nutný i zde.

Zbytek řádky se už zpracuje hladce. Jediná zrada by se mohla ukrývat v proměnné `OPTARG`. Vzhledem k tomu, že se v ní nachází to, co uživatel zadal na příkazové řádce, je nutné očekávat jen to nejhorší. Proto jsme také její expanzi z první fáze zpracování vypustili. Její expanze ještě před provedením příkazu **eval** by mohla být nebezpečná. Představme si, že bychom příkaz napsali takto:

```
eval opt$znak=\"-$znak$OPTARG\"
```

a uživatel zadal parametr „`*příkaz*`“. Pak by řádka po provedení substitucí měla tvar:

```
eval optc=\"-c`příkaz`\"
```

Příkaz **eval** by řádku poslušně sestavil a shell by vykonal příkaz dle zadání uživatele.

Pokud substituci proměnné `OPTARG` odložíme až do druhého průchodu zpracování řádky, nebezpečí už nehrozí, protože zpětné apostrofy se neuplatní jako metaznak.

Druhý důvod pro odložení substituce spočívá v tom, že příkaz **eval** dostává původní řádku už ve tvaru seznamu parametrů, bílé znaky v hodnotě proměnné by se tedy chápaly jako oddělovače parametrů a **eval** by je vůbec nedostal. Budeme-li chtít, aby ve výsledné řádce byla použita přesná hodnota, kterou zadal uživatel (včetně všech bílých znaků), musíme expanzi proměnné nechat až na druhý průchod.

Uveďme ještě jednu typickou aplikaci příkazu **eval** – uložení nějakého operátoru shellu do hodnoty proměnné. Rozpoznávání operátorů probíhá rovněž v prvním kroku,

a tedy dříve než expanze proměnných. Představme si, že chceme ve skriptu nastavením jedné proměnné volit, zda se budou vypisovat ladicí výpisy, a napíšeme:

```
out='> /dev/null'
echo Ladici vypis $out
```

To asi nebude to „pravé ořechové“. V prvním kroku shell žádný operátor přesměrování nevidí. Po dosazení proměnné se tam sice většítko objeví, ale to už ho zase shell nebude považovat za operátor, předá ho jako parametr příkazu **echo** a ten ho prostě vypíše. I zde by **eval** pomohl (i když tato forma zápisu příkazu nebude vypadat nic moc):

```
eval echo Ladici vypis $out
```

Pokud bychom do hodnoty proměnné uložili jenom jméno souboru a vlastní operátor (většítko) zapsali přímo v textu řádky, vše proběhne, jak má:

```
out='/dev/null'
echo Ladici vypis > $out
```

Jenže to budeme mít zase jiný problém v tom, co do hodnoty proměnné `out` uložit, pokud výpis příkazu **echo** budeme chtít vidět. Jednu možnost představuje pseudosoubor **/dev/tty**. Ten zastupuje terminál, na kterém shell právě běží. Výstup se tedy objeví na obrazovce. Ale nepůjde s ním nijak dál pracovat (např. ho nebude možné poslat rourou nějakému dalšímu programu).

Druhou možností je další pseudosoubor z adresáře **/dev**, a to **/dev/stdout**, který zastupuje standardní výstupní proud (podobně jako **/dev/stderr** a **/dev/stdin** zastupují standardní chybový výstup, resp. standardní vstup). Jenže tyto pseudosoubory pro změnu nejsou v normě, takže se opět vystavujeme riziku omezené přenositelnosti.

Nezapomínejte, že to, co za jménem příkazu **eval** následuje, musí být zase příkaz (jinak shell nebude mít co spustit). Porovnejte výsledek těchto příkazů:

```
eval echo \"\$opt$znak\"
echo eval \"\$opt$znak\"
```

V kapitole 4.22 jsme se pokoušeli zjednodušit nastavení proměnné **IFS** před voláním příkazu **find**. Asi takto:

```
IFS=: find $PATH -name sort | head -n1
```

Vysvětlovali jsme si přitom, že to není dobře, protože takto se proměnná **IFS** nastaví pro volaný příkaz (**find**), jenže rozdělení hodnoty proměnné **PATH** na slova dělá shell. Příkaz **eval** by nám zde opravdu mohl pomoci.

Ovšem nikoliv takto:

```
IFS=: eval find $PATH -name sort | head -n1
```

Tady totiž expanzi proměnné **PATH** pořád ještě dělá náš původní shell, a to s původní hodnotou proměnné **IFS**. Správné řešení je:

```
IFS=: eval find \$PATH -name sort | head -n1
```

Zde se expanze provádí až při druhém čtení řádky (po vykonání příkazu **eval**), a to už je proměnná **IFS** nastavena správně.

Test „**[-n "$optc$optn"]**“ je maličko trikový. Správně bychom asi měli napsat „**[-n "$optc" -o -n "$optn"]**“, ale naše verze je ekvivalentní a kratší. Dokonce i její čitelnost je jen o málo horší.

Parametry se funkci předávají jako jakémukoliv jinému příkazu shellu (neboli: žádné závorky okolo nich, žádné čárky mezi nimi). Ve funkci se k nim pak dá přistupovat jako k pozičním parametrům (mj. to znamená, že na rozdíl od proměnných, jež jsou vždy globální, jsou poziční parametry ve funkci lokální).

Vzhledem k tomu, že funkce se volá jako obyčejný příkaz, nastává potenciální kolize. Co když si třeba vytvoříte funkci, která se bude jmenovat **cd**, a pak napíšete příkaz „**cd**“? Zavolá se vaše funkce nebo příkaz? Přednost má funkce. Pro upřednostnění příkazu před funkcí má shell explicitní příkaz **command**:

```
command cd directory
```

4.29. Poslední parametr

Zadání:

Napište skript, který tvoří obálku pro příkaz **mv** – kontroluje, zda poslední parametr je existující adresář, a pak **mv** vykoná.

Rozbor:

Poslední parametr. Kudy na něj?

Potřebovali bychom napsat něco jako „**$***počet_parametrů*“. Ale počet parametrů skriptu přece známe, ten se dá zjistit pomocí speciálního parametru **$#**. Stačilo by shell přesvědčit, aby nejprve dosadil počet parametrů a pak teprve zpracoval celý řetězec. A to je situace velmi podobná problému z minulé kapitoly. Ano, příkaz **eval** představuje jedno z možných řešení:

```
eval cil=\$$#
[ ! -d "$cil" ] && echo "Adresar $cil neexistuje" && exit
mv "$@"
```

Pro počet parametrů menší než 10 to bude řešení vyhovující. Pokud bychom ale chtěli zvládat i počet parametrů větší než devět, museli bychom použít konstrukt:

```
eval cil=\${$#}
```

Jenže ten zase má problémy s přenositelností.

Takže jinak. Kdybychom parametry v cyklu postupně posouvali, až nám zbude jen poslední, můžeme ho jednoduše otestovat. Jenže tím o všechny ostatní parametry přijdeme. Museli bychom si je mezitím někam schovávat, abychom je později mohli zase použít. Třeba do proměnné:

```
while [ $# -gt 1 ]; do
    param="$param $1"
    shift
done
[ ! -d "$1" ] && echo "Adresar $1 neexistuje" && exit
mv $param $1
```

Tohle řešení bude fungovat docela dobře, pokud budou názvy souborů „rozumné“, tj. nebudou obsahovat například mezery, kterými oddělujeme názvy souborů v proměnné `param`. Ty by se totiž v závěrečném příkazu změnily na oddělovače parametrů a název souboru „`a b`“ by se chápal jako dva soubory. Pokud parametry uložíme do proměnné, hranice mezi nimi zmizí. Řešením by mohlo být oddělovat názvy souborů znaky konce řádky (ty opravdu ve jménech souborů nebývají) a do proměnné **IFS** nastavit naopak pouze znak konce řádky.

Pro obecné řešení by to určitě chtělo lepší místo pro uchování parametrů. Jediným spolehlivým místem jsou poziční parametry samy. Kdybychom je postupně posouvali, ale zároveň je přidávali zase na konec seznamu, máme vyhráno. Budeme ale potřebovat jinou proměnnou na počítání kroků, protože **$#** se nám nyní nebude snižovat:

```
i=1
while [ $i -le $# ]; do
    cil=$1
    shift
    set -- "$@" "$cil"
    i=`expr $i + 1`
done
[ ! -d "$cil" ] && echo "Adresar $cil neexistuje" && exit
mv "$@"
```

Další možnost nabízejí funkce. Ty totiž sice používají globální proměnné, ale poziční parametry mají lokální. Proto ve funkci můžeme udělat cyklus s posouváním parametrů, aniž si „zničíme“ poziční parametry našeho vlastního skriptu. Stačí předat funkci náš seznam parametrů:

```
kontroluj_posledni() {
    while [ $# -gt 1 ]; do
        shift
    done
    [ ! -d "$1" ] && echo "Adresar $1 neexistuje" && exit
}
kontroluj_posledni "$@"
mv "$@"
```

Poslední možností je příkaz **for**. Ten projde poziční parametry, aniž by jim uškodil:

```
for cil; do
    :
done
[ ! -d "$cil" ] && echo "Adresar $cil neexistuje" && exit
mv "$@"
```

Řešení:

Ještě spousty variací, ale to už bychom se dost opakovali.

Poznámky:

V posledním řešení jsme opět viděli použití zvláštního příkazu „`:`“ jako syntaktické vaty, neboť v těle cyklu mezi **do** a **done** musí nějaký příkaz být. Nic jiného v těle cyklu dělat nemusíme, poslední parametr nám v proměnné `cil` prostě jen tak zbude.

4.30. Neznámý formát

Zadání:

Napište skript **myps**, který upraví výpis příkazu **ps** tak, že vypíše pouze sloupečky zadané uživatelem v prvním parametru (seznam názvů sloupečků oddělených čárkami).

Předpokládejte, že všechny potřebné sloupečky ve výpisu **ps** jsou, ale neznáte jejich pořadí – to musíte poznat ze záhlaví výpisu.

Příkaz **ps** vypisuje informace o běžících procesech, příklad vstupu skriptu:

```
sinek:~> ps -l
  UID   PID  PPID CPU   VSZ STAT  TT       TIME COMMAND
 1004  9732  6137   0  4300 Ss    p9    0:00.00 bash
 1004 27263  9732   1  1460 R+    p9    0:00.00 ps -l
```

Příklad volání a výstupu:

```
sinek:~> ps -l | myps stat,pid,ppid,command
Ss 9732 6137 bash
R+ 27263 9732 ps -l
```

Rozbor:

Zadržte! Než začnete rozmýšlet složité konstrukce, jak ukládat názvy sloupečků do polí a jak v nich potom vyhledávat, vzpomeňte si na to, že shell je textový preprocesor a text programu je možné (v zájmu mírného pokroku v mezích zákona) za běhu měnit.

První řádka výstupu příkazu **ps**, obsahuje seznam názvů sloupců. Pokud ho ale pojmeme jako seznam jmen proměnných, načteme ho do proměnné (`zahlavi`) a tu pak dáme jako parametr dalšímu příkazu **read**, budou se správné sloupce načítat rovnou do proměnných se správnými jmény (jen pro jistou převedenými na malá písmena kvůli riziku kolizí)! A potom stačí už jen požadované sloupce vypsat – kombinací příkazů **eval** a **echo**, jimž připravíme seznam jmen požadovaných proměnných.

Řešení:

```
seznam=\$`echo $1 | tr A-Z a-z | sed 's/,/ $/g'`
read zahlavi
sloupce=`echo $zahlavi | tr A-Z a-z`
while read $sloupce; do
    eval echo $seznam
done
```

Poznámky:

Převod hodnoty **$1** na seznam proměnných pro příkaz **echo** lze udělat také pomocí příkazu „**s/\([^,]*\),*/ $\1/g**“ (neboli: vezmi každý název a případnou čárku za ním a nahraď je mezerou, dolarem a názvem). Naše řešení je sice trikové, ale dokonce možná i čitelnější: zaměň čárky za mezeru a dolar (a na začátek dej ještě jeden dolar).

4.31. Interaktivní mazání

Zadání:

Vytvořte skript, který bude interaktivně mazat soubory vyhledávané na základě stejných parametrů, jako má příkaz **find**. Skript příkaz zavolá a soubory, které **find** najde, postupně maže. Před smazáním každého souboru se obsluhy zeptá, zda má mazání provést. Možné odpovědi jsou „ne“ (implicitní odpověď), „ano“ a „stop“. Pokud skript nebyl ukončen pomocí „stop“, na konci běhu vypíše počet smazaných souborů.

Rozbor:

Zavoláme **find** a předáme mu parametry, které jsme dostali. Výstup pošleme rourou do cyklu, budeme číst jednotlivé řádky, na každý soubor se zeptáme, přečteme odpověď a podle ní provedeme či neprovedeme mazání. V případě odpovědi „stop“ zavoláme příkaz **exit**. Během cyklu počítáme smazané soubory. A můžeme programovat:

```
n=0
find "$@" | while read soubor; do
    echo -n "Mazat $soubor? "
    read odpoved
    case $odpoved in
    ano     ) n=`expr $n + 1`; rm "$soubor";;
    '' | ne ) ;;
    stop    ) exit;;
    esac
done
echo "Smazano $n souboru"
```

Výsledek vypadá dobře, že? Vypadá, ale nefunguje.

První problém spočívá v tom, že program se nás vůbec na nic neptá. Vypadá to, jako by druhý příkaz **read** vůbec nepracoval. Samozřejmě pracuje, ale čte úplně něco jiného, než jsme očekávali. Podívejte se pozorně na obě volání příkazu **read**. Jaký je mezi nimi rozdíl? První má číst jména souborů z roury, zatímco druhý naše odpovědi z klávesnice terminálu. Jenže to víme jenom my. Shellu jsme to neřekli! Obě volání jsou zapsána zcela stejně. A proto se také shell v obou případech zcela stejně chová – druhý **read** čte „odpovědi“ z roury namísto z klávesnice. Náš skript prostě následující jméno souboru považuje za odpověď na předchozí dotaz. Budeme muset druhému volání příkazu **read** přesměrovat vstup.

Druhý problém je ještě záhadnější. Výsledkem programu je za všech okolností text „Smazano 0 souboru“. Kdybyste si ale vložili dovnitř cyklu ladicí tisky, viděli byste, jak hodnota proměnné poslušně roste. Jako by rostla jiná proměnná. Bingo! Vzpomeňte

si, co jsme si říkali o čtení z roury. Pokud v programu napíšete rouru, shell musí pro každou „stranu“ roury (producenta i konzumenta) spustit nový proces. Naše proměnná se tedy nastaví pouze v tomto synovském procesu (subshellu) zatímco v hlavním procesu (po skončení cyklu) má tamější proměnná hodnotu nezměněnou. Musíme nějak zařídit, aby výpis proměnné byl proveden stejným procesem, který četl z roury. Řešení už známe – necháme cyklus i s následným příkazem **`echo`** provést ve stejném subshellu (uzavřeme je do kulatých závorek).

Poslední problém spočívá v tom, že pokud uživatel ukončí skript zadáním odpovědi „`stop`“, náš program stejně vypíše zprávu o počtu smazaných souborů, což by podle zadání neměl. Jak to, že neproběhl příkaz **`exit`**? Odpověď je stejná jako v minulém případě: Příkaz proběhl, ale v jiném procesu! Příkazy na pravé straně roury se provádějí v subshellu, příkaz **`exit`** ukončí tento subshell, načež program pokračuje za rourou, příkazem **`echo`**. Trik, kterým jsme řešili předchozí problém, zafunguje i zde. Pokud cyklus i následné **`echo`** budou společně uzavřeny do jednoho subshellu, příkaz **`exit`** ukončí celý tento subshell a žádná zpráva se tedy nevypíše.

Řešení:

```
n=0
find "$@" | (
    while read soubor; do
        echo -n "Mazat $soubor? "
        read odpoved < /dev/tty
        case $odpoved in
        ano     ) n=`expr $n + 1`; rm "$soubor";;
        '' | ne ) ;;
        stop    ) exit;;
        esac
    done
    echo "Smazano $n souboru"
)
```

Poznámky:

Pseudosoubor **`/dev/tty`** jsme už poznali jako výstupní zařízení, zde ho použijeme jako vstup – v tom případě představuje klávesnici. Toto řešení má ovšem jednu drobnou nevýhodu. Pokud bychom celý skript pouštěli v rámci nějaké větší úlohy a chtěli mu jeho vstup přesměrovat (předpřipravit „odpovědi“ uživatele), nepůjde to, protože náš příkaz „**`read odpoved < /dev/tty`**“ prostě nečte ze vstupu, ale přímo z terminálu.

Řešením by bylo si „vstup“ (resp. přístup ke vstupnímu proudu) nějak „uchovat“ a i to v shellu lze. Víme, že proudy dat jsou očíslovány, a že těm standardním patří čísla 0, 1 a 2. Už jsme také poznali operátor „**`>&`**“, který provádí duplikaci deskriptoru pro výstupní proudy. A asi není žádným překvapením, že existuje i analogický operátor pro

duplikaci vstupního proudu, a to „**<&**“. Uzavřeme-li celý skript do složeného příkazu a pro něj nadefinujeme duplikaci deskriptoru 0 pod číslem 3, můžeme pak uvnitř cyklu přesměrovat vstup některých příkazů **read** (těch, jež mají číst reakci od obsluhy) tak, aby četly z deskriptoru číslo 3:

```
{
    find "$@" | (
        while read soubor; do ... čtení z roury
            echo -n "Mazat $soubor? "
            read odpoved <&3   ... čtení ze (zkopírovaného) vstupního proudu
            ...
        done
        echo "Smazano $n souboru"
    )
} 3<&0                              ... přesměrování vstupního proudu
```

Malinko nepříjemné na této konstrukci je to, že duplikace deskriptoru, která se musí odehrát na začátku programu, se píše až na konci. Norma (na rozdíl od některých starších implementací) nabízí poněkud logičtější zápis pomocí zvláštního příkazu **exec**. Příkaz normálně slouží k ukončení shellu a předání řízení jinému programu (který dostává jako parametr). Pokud ale není parametr uveden, provedou se pouze operace přesměrování. Mohli bychom tedy napsat:

```
exec 3<&0                       ... zkopíruj deskriptor č. 0 do č. 3
find "$@" | ( ... )
exec 3<&-                       ... zavři deskriptor č. 3
```

Povšimněte si ošetření případu, kdy uživatel stiskne pouze klávesu Enter. Občas vídávám pokusy o test něčeho jako „**\n**“. Ovšem správné řešení lze odvodit prostou logikou: když uživatel napíše „`ano`“ a Enter, testujeme jeho odpověď jako „`ano`“ a nikoliv „`ano\n`“ – takže pokud stiskne pouze Enter, testujeme prázdný řetězec...

Pokud vás trápí, že se stále vyhýbáme přenositelnému řešení problému výpisu textu bez odřádkování, můžete to obejít například takto:

```
if [ "`echo -n; echo x`" = "x" ]; then
    vypis() { echo -n "$@"; }
else
    vypis() { echo "$@\c"; }
fi
...
vypis "Mazat $soubor? "
```

Takové řešení lze doporučit asi více než normou doporučovaný příkaz **printf**. Ten by nám zde posloužil také, ale s přenositelností (na starší stroje) by nám stejně nepomohl a navíc je oproti tomuto řešení pomalejší (což by zrovna v tomto případě nevadilo).

4.32. Rekurze a proměnné

Zadání:

Vytvořte skript, který vyhledá soubor zadaného jména (první parametr) v zadaném adresáři, resp. podstromu (druhý parametr). Skript musí strom prohledávat *do hloubky*, tj. v každém adresáři nejprve prohledá podadresáře a potom teprve zkontroluje jméno adresáře samotného. Skript vypisuje pouze první výskyt daného jména a pak skončí.

Rozbor:

Prohledávání stromu je klasická úloha vedoucí na *rekurzivní* algoritmus. Vytvoříme funkci, jejímž úkolem bude prohledat daný podstrom. Funkce projde adresář a na každý podadresář zavolá sama sebe. Jakmile soubor najde, vypíše výsledek a skončí.

Na procházení obsahu adresáře se hodí cyklus **for** v kombinaci s expanzními znaky shellu („**for x in ***“). Nejprve zavoláme příkaz **cd** a pokusíme se „přepnout“ do prohledávaného adresáře, po skončení cyklu se zase vrátíme zpět:

```
projdi() {
    cd "$1" || return
    for x in *; do
        [ -d "$x" -a ! -L "$x" ] && projdi "$x" && return
        [ "$x" = "$vzor" ] && echo `pwd`/"$x" && return
    done
    cd ..; return 1
}
vzor=$1 projdi "$2"
```

Použité řešení je poněkud nešikovné v tom, že po úspěšném nalezení souboru funkci opustíme pomocí příkazu **return**, takže neproběhne závěrečné „**cd ..**“ a proces bude v jiném adresáři než před zavoláním funkce. V našem jednoduchém příkladu bychom tuto vadu odstranili snadno, ale někdy to může být obtížnější.

Napišme tedy funkci `projdi` trochu jinak: pracovní adresář nebudeme v průběhu práce měnit a cyklus zapíšeme jako „**for x in** *adresář***/***“. Jen si budeme muset při porovnávání jmen dát pozor na to, že „**$x**“ bude nyní obsahovat i cestu:

```
projdi() {
    for x in "$1"/*; do
        [ -d "$x" -a ! -L "$x" ] && projdi "$x" && return
        [ "$x" = "$1/$vzor" ] && echo "$x" && return
    done
    return 1
}
vzor=$1 projdi "$2"
```

Problém obou těchto řešení však spočívá v tom, že proměnné jsou v shellu globální. Hodnota proměnné `x` se proto po návratu z vnitřního volání funkce změní, bude v ní ta hodnota, kterou tam zanechala poslední instance funkce!

Pokud bychom neměli v zadání podmínku procházení do hloubky, obrátili bychom pořadí příkazů ve funkci (nejprve bychom otestovali jméno souboru a až poté zavolali vnořenou funkci) a problém by nenastal, protože bychom proměnnou `x` použili dříve, než nám ji vnořená funkce „zkazí“.

S problémem globálních proměnných nám mohou pomoci poziční parametry, neboť ty jsou ve funkcích lokální. Příkazem **`set`** do pozičních parametrů uložíme seznam souborů v adresáři, a pak jej procházíme pomocí příkazů **`while`** a **`shift`**:

```
projdi() {
    set -- "$1"/*
    while [ $# -gt 0 ]; do
        [ -d "$1" -a ! -L "$1" ] && projdi "$1" && return 0
        case "$1" in
        $vzor | */$vzor ) echo "$1"; return 0;;
        esac
        shift
    done
    return 1
}
vzor=$1 projdi "$2"
```

Jinou cestu nabízí změna logiky použití funkce `projdi`. Budeme ji volat na každé jméno souboru a nikoli jen na adresáře a kontrolu shody jména souboru „odložíme“ až na konec funkce. Pokud při prohledávání adresáře nebudeme používat poziční parametry (použijeme např. stejný cyklus jako ve druhém řešení), jméno souboru nám zůstane zachováno v lokálním pozičním parametru **`$1`**. Ztratíme ovšem informaci o aktuální cestě, takže budeme muset trochu změnit styl kontroly jména:

```
projdi() {
    if [ -d "$1" -a ! -L "$1" ]; then
        for soubor in "$1"/*; do
            projdi "$soubor" && return 0
        done
    fi
    case "$1" in
    */$vzor ) echo "$1"; return 0;;
    *       ) return 1;;
    esac
}
vzor=$1 projdi "$2"
```

Řešení:

Stačí si vybrat.

Poznámky:

Skript v této podobě neumí vyhledávat skryté soubory a procházet skryté adresáře. Doplňte si tuto funkčnost jako cvičení.

Test adresáře jsme doplnili ještě o podmínku „**`-L`**“, aby náš skript neprocházel adresáře „schované“ za symbolickými linky.

Příkaz **`return`** jednak ukončuje práci funkce a jednak může pomocí parametru udávat *návratovou hodnotu*. Návratovou hodnotou může být (stejně jako u každého jiného příkazu) pouze číslo mezi 0 a 255. Nic složitějšího (např. řetězec) takto volajícímu nepředáme! Pokud hodnotu neudáme, použije se návratová hodnota posledního provedeného příkazu. Pro zlepšení čitelnosti programu se ale vyplatí hodnotu vyjadřovat explicitně (známe-li ji). V našich řešeních bychom mohli u příkazů **`return`** parametr vynechávat, ale srozumitelnost kódu by se zhoršila.

My jsme návratovou hodnotu funkce použili na to, abychom detekovali úspěšné nalezení hledaného souboru a způsobili „vynoření“ z rekurzivních volání funkce. V našem konkrétním případě by stačilo zavolat příkaz **`exit`**, protože skript pak už nemá nic jiného na práci, ale kdyby měl skript dál pokračovat, toto řešení by se hodilo.

Některé verze shellu mají možnost deklarovat, že určitá proměnná je ve funkci lokální, ale tento konstrukt není zahrnut v normě.

Problémy s lokalitou proměnných by nám odpadly, kdybychom namísto rekurzivní funkce použili rekurzivní volání celého našeho *skriptu*. Pokud bychom pro zanoření do podadresáře použili znovu náš skript, každé jeho vyvolání by znamenalo nové spuštění (sub)shellu a ten by pochopitelně měl svoje vlastní proměnné.

Řešení pomocí skriptu místo funkce přináší ovšem dvě drobné komplikace. Jednak je pomalejší (spouští se více procesů) a jednak není úplně jednoduché předávat výsledky ze skriptu volaného skriptu volajícímu. Přesně ta výhoda, pro kterou jsme použili novou instanci skriptu (má nezávislé lokální proměnné), se stane nevýhodou, protože výsledky už nepůjde volajícímu předat pomocí proměnných. Obvykle nezbývá, než výsledky poslat třeba rourou anebo je uložit do souboru. V našem příkladu by se tato komplikace naštěstí neprojevila (žádné výsledky nepředáváme), zato hned v následující kapitole budeme muset takový problém řešit.

4.33. Rekurze a procesy

Zadání:

Vytvořte skript, který prochází strom adresářů, jehož kořen dostane jako parametr, a pro každý podstrom postupně spočítá a vypíše součet velikostí všech souborů v něm.

Rozbor:

Spočítat součet velikostí souborů není těžké. Použijeme příkaz **`ls`** a sečteme hodnoty ve správném sloupci. Máme ale dva jiné problémy:

- z rekurzivního volání potřebujeme nějak dostat jeho výsledek;
- během zanoření do podadresáře si musíme někde uchovat mezisoučet.

První problém řeší jednoduše varianta s rekurzivní funkcí – výsledek se prostě uloží do proměnné a volající ho tam najde. Ale funkce nenabízí řešení druhého problému, mezisoučty se nám budou neustále přepisovat. Naopak, rekurzivní skript nemá problém s uchováváním mezisoučtu (má své vlastní proměnné), zato není triviální dostat se k jeho výsledkům.

Zkusíme prošlapat druhou cestu a vymyslet, jak předat volajícímu výsledek skriptu.

Pomocí proměnných to nepůjde, to už víme.

Použití návratové hodnoty by sice přicházelo v úvahu, ale pro naše potřeby má malý rozsah (max. 255). Pro pořádek připomeňme, že uvažovat o předávání výsledků touto cestou by bylo naprosto scestné, pokud by výsledky byly jiného typu (např. řetězce).

Nejsnazší asi bude použít dočasný soubor. Pro každou běžící instanci skriptu však budeme potřebovat vlastní soubor. Jeho jméno bude muset obsahovat nějaký podřetězec, který je pro daný proces jedinečný. Mohli bychom třeba vytvořit jméno s pomocí čísla, které budeme postupně o jedničku zvyšovat. Ale naštěstí existuje jednodušší cesta, jak dosáhnout jedinečného čísla – každý proces má svoje *číslo procesu* (PID). A číslo běžícího procesu shellu lze zjistit pomocí zvláštního parametru „**`$$`**“. Toto číslo (nebo příp. celé jméno souboru) budeme muset volanému skriptu předat, aby věděl, kam nám má výsledek zapsat. U vnořených volání tedy přidáme ještě jeden parametr.

Pomocné soubory ovšem otevírají nový problém, a tím je jejich mazání. Po skončení práce soubor samozřejmě smažeme. Ale pokud by náš skript někdo přerušil násilně, dočasný soubor na disku zůstane a přestane být „dočasný“. Tomu můžeme zabránit tím, že našemu skriptu předepíšeme jinou obsluhu *signálů*. Signály jsou jedním ze základních prostředků meziprocesové komunikace v UNIXu, pomocí nich se oznamuje procesu, že nastala nějaká událost, třeba že uživatel chce, aby program skončil. Signály mají svá jména a čísla – pokud například na klávesnici zmáčknete Ctrl+C, program běžící na terminálu obdrží signál č. 2, neboli „INT“ (*interrupt*). Autor programu může stanovit,

jak se jednotlivé signály obslouží. V shellu se to dělá příkazem **trap** – jeho prvním parametrem je příkaz, který se má provést v případě, že program zachytí některý ze signálů, jejichž jména nebo čísla následují jako další parametry.

Náš skript by měl jako obsluhu signálu provést smazání svého dočasného souboru.

Podívejme se ještě pořádně, co se vlastně stane po zmáčknutí Ctrl+C: Aktuálnímu procesu bude doručen signál č. 2. Aktuálním procesem je poslední spuštěná instance shellu, která vykonává náš skript a prochází nějaký adresář v určité hloubce stromu. Ta skončí. Ale co rodičovský proces? Ten se o tom nedozví! Zjistí sice, že jeho synovský proces skončil, ale nijak ho to netrápí, převezme řízení a pokračuje v procházení svého adresáře (o patro výš). Náš program by nebylo možné tímto způsobem ukončit.

Skript tedy bude muset v případě násilného ukončení vracet nenulovou návratovou hodnotu a také musí kontrolovat, s jakou návratovou hodnotou končí jeho synovské procesy (rekurzivní volání).

Zbývá poslední technický problém, jak sečíst řadu čísel. Zdánlivě nejjednodušší je prostě připočítávat postupně hodnoty k mezisoučtu pomocí příkazu **expr**. Pokud si to ale zkusíte na nějakém větším adresáři, asi se dost načekáte. Spouštění nového procesu pro každý výpočet bude dost zdržovat. Proto nebudeme mezisoučty počítat průběžně, ale vyrobíme (v našem dočasném souboru) jeden dlouhý výraz, který necháme příkazem **expr** spočítat až na konci práce skriptu. Přesněji řečeno, můžeme si do souboru rovnou napsat i název příkazu (**expr**) a pak soubor zavolat jako skript.

Řešení:

```
pomocny=/tmp/pomocny.$$
trap 'rm -f $pomocny; exit 1' 2 3 15
ls -la "$1" | {
    echo -n "expr 0" > $pomocny
    while read typ_prava x x x velikost x x x jmeno; do
        [ "$jmeno" = . -o "$jmeno" = .. ] && continue
        case $typ_prava in
        [-l]* ) echo -n " + $velikost" >> $pomocny;;
        d* )    $0 "$1/$jmeno" $pomocny && continue
                rm -f $pomocny; exit 1;;
        esac
    done
    soucet=`sh $pomocny`
    rm $pomocny
    echo $1:$soucet
    [ -n "$2" ] && echo -n " + $soucet" >> $2
    exit 0
}
```

Poznámky:

Kromě signálu č. 2 jsme si nechali ošetřit zároveň i signál č. 3 („QUIT“), který se vyvolá stiskem Ctrl+\, a signál č. 15 („TERM“, *terminate*). Ten lze programu poslat příkazem **`kill`**, což je další prostředek, jak zastavit běžící program. Tato trojice signálů bývá ve skriptech obvykle obsluhována shodně.

Obslužný příkaz signálu (první parametr příkazu **`trap`**) vykonává při příchodu signálu ten proces shellu, který signál zachytil. Může přitom normálně zacházet se svými proměnnými, volat funkce apod. Jenom je třeba dát pozor, jak tento příkaz zapíšeme. Když ho uzavřeme do uvozovek, budou se případné substituce proměnných provádět už v okamžiku zpracování příkazu **`trap`**, zatímco pokud použijeme apostrofy, expanzi proměnných odložíme až na dobu zachycení signálu a vykonávání obslužného příkazu.

Obsluhu signálu lze zamaskovat (ignorovat příchod signálu) tím, že první parametr zadáme jako prázdný řetězec.

Pokud první parametr příkazu vynecháme, obsluha signálů se vrátí do základního stavu (kterýkoliv z těchto tří signálů by pak skript ukončil).

Příkaz **`trap`** lze použít i pro úklid na konci práce. Pokud bude mezi parametry číslo 0 (resp. jméno „`EXIT`“), je to pokyn shellu, aby obslužný příkaz provedl i tehdy, když skript skončí „sám od sebe“. Výhodou je, že pak mazání nemusíme řešit nadvakrát.

Bez parametrů příkaz vypíše, jaké nastavení obsluhy signálů je aktuálně platné.

Abychom do svých výpočtů zahrnuli i skryté soubory, musíme příkaz **`ls`** volat s přepínačem „**`-a`**“ a při čtení výstupu ignorovat řádky se jmény „`.`“ a „`..`“. Rozlišíme je snadno příkazem **`test`** a ze zpracování je vyřadíme pomocí příkazu **`continue`**. Ten způsobí, že se přeskočí zbytek těla cyklu a shell bude pokračovat pokusem o přečtení nové řádky.

Z výpisu je rovněž třeba vyloučit první řádku („`total ...`“) a řádky pro jiné typy souborů. Udělali jsme to opačně – vybrali jsme typy, pro něž chceme velikost souboru zahrnout a ostatní ignorujeme. Testujeme tedy první znak na řádce pomocí porovnávacího vzoru v návěští příkazu **`case`** („**`[-l]*`**“).

Do vytvářeného pomocného souboru zapíšeme na začátku kromě jména příkazu **`expr`** ještě nulu. Tím obejdeme při výpisu výrazu nepříjemné rozlišování mezi prvním a dalšími sčítanci. Pro každý načtený soubor zapíšeme operátor plus a velikost souboru.

Bude-li počet souborů v nějakém adresáři hodně veliký, můžeme u příkazu **`expr`** narazit na limit počtu parametrů na příkazové řádce. Pak by nám asi opravdu nezbylo nic jiného než s rizikem omezené přenositelnosti na starší systémy zkusit aritmetiku přímo v shellu a provádět přičítání pro každý soubor hned:

```
soucet=$(( $soucet + $velikost ))
```

Norma nabízí ještě další variantu, jak skript může zjistit číslo rodičovského procesu (aby si sám mohl sestavit správné jméno pomocného souboru), a to proměnnou **`PPID`** (*Parent PID*). Zdaleka ne každá implementace však v tomto bodě normu splňuje.

Povšimněte si dalšího příkladu použití speciálního parametru „**`$0`**“. Abychom nemuseli složitě zkoumat, jak přesně zapsat jméno našeho skriptu ve vnořeném volání, použijeme prostě přesně stejný řetězec, s jakým jsme byli zavoláni sami. Nesmíme přitom ale změnit adresář, protože kdyby byl náš skript zavolán pomocí relativní cesty („**`./skript`**“), trik by už nefungoval.

Kdybychom v požadavcích neměli průběžné vypisování pro podadresáře, mohli bychom se obejít bez pomocných souborů. Stačilo by prostě vypsat výsledek na výstup a ve volající instanci si ho přečíst pomocí zpětných apostrofů:

```
ls -la "$1" | {
    vyraz=0
    while read typ_prava x x x velikost x x x jmeno; do
        [ "$jmeno" = . -o "$jmeno" = .. ] && continue
        case $typ_prava in
        [-l]* ) vyraz="$vyraz + $velikost";;
        d*    ) vyraz="$vyraz + `$0 "$1/$jmeno"`";;
        esac
    done
    expr $vyraz
}
```

4.34. Procesy a proměnné

Zadání:

Naprogramujte kalkulačku. Skript bude postupně číst ze vstupu řádky, které mohou obsahovat libovolně mnoho položek ve tvaru: celé číslo, aritmetický operátor nebo závorka. Položky musejí být odděleny alespoň jednou mezerou. Jakmile je na vstupu nalezena položka „=“, skript musí vypsat celý výraz a jeho výsledek.

Pro vlastní výpočet je možno použít příkaz **`expr`**, předpokládejte ovšem, že příkaz neumí pracovat se závorkami – je tedy třeba zpracovávat každý podvýraz v závorkách jedním voláním příkazu. Správné párování závorek je nutno zkontrolovat, pro kontrolu správnosti výrazu v závorkách, stačí ošetřit výsledek volání příkazu **`expr`**. V případě chybného výrazu se vypíše pouze chybová zpráva na chybový výstup.

Rozbor:

Algoritmicky nejjednodušší je opět řešení pomocí rekurze – každou závorku bude řešit nové volání funkce nebo skriptu. Stojíme tedy opět před úplně stejnými problémy jako v předešlých kapitolách. Rekurzivní funkce bude mít problém s absencí lokálních proměnných, pro řešení pomocí skriptu musíme vymyslet vracení výsledků. A tentokrát je výsledků požehnaně:

- Musíme vracet příznak úspěšnosti vykonání skriptu (kvůli reakci na chyby). Na to by se dala normálně použít návratová hodnota skriptu.
- Musíme někam zapisovat postupně zpracovávané položky výrazu (aby bylo možné ho na konci celý vypsat). Kdybychom neměli omezení, že při chybě se nesmí nic vypsat, mohli bychom uvažovat o tom, že budeme položky rovnou psát na standardní výstup, ale toto řešení zde nepůjde použít. Přes proměnné to rovněž nepůjde, to už víme. Budeme muset použít pomocný soubor, ale jeho jméno budeme muset předat všem instancím skriptu.
- Musíme umět vrátit výsledek podvýrazu. Pokud nechceme i zde použít soubor, mohli bychom výsledek vypisovat na výstup a volající by si ho prostě přečetl.
- Musíme rovněž vrátit zbytek nezpracovaného výrazu. Mohlo by se nám totiž stát, že jsme během zpracování závorky načetli novou řádku, jejíž obsah ten, kdo nás zavolal, nezná. Takže další soubor... To je nepříjemné. Zkusme popřemýšlet. Výsledkem závorky je nějaké číslo a za ním následují případně další nedočtené položky výrazu. Volajícímu je ale úplně jedno, zda číslo, které se chystáme vracet, jsme pracně spočítali anebo zda bylo zadáno uživatelem jako zbytek řádky. Tento čtvrtý výsledek rekurzivního volání je tedy možné spojit se třetím a na výstup zapsat nejprve výsledek závorky a za něj zbytek výrazu.

Zdá se, že problémy rekurzivního volání pomocí skriptu se nám podařilo vcelku elegantně vyřešit, zkusme se proto vydat touto cestou.

Volanému skriptu musíme předat jméno pomocného souboru. Na rozdíl od minulé kapitoly ale vytváříme pro celý výraz jen jediný soubor a jeho jméno se nebude mezi jednotlivými „patry“ měnit. Stačilo by tedy předávat jen číslo startovního procesu. Možná vás napadne, že bychom si mohli toto číslo uložit do nějaké proměnné (`start`) a synovské procesy prostě nechat, aby si ho tam přečetli. Pokud si to ale vyzkoušíte, zjistíte, že synovský proces o existenci této proměnné nebude vědět. Jak je to možné? Zavolání skriptu totiž probíhá tak, že se spustí nová instance shellu, která začne celý skript provádět znova od začátku a nemá tedy žádnou informaci o proměnných nastavených v rodičovském procesu. Veškeré předávání informací se musí provádět pomocí prostředků operačního systému, v tomto případě *systémových proměnných*, resp. *proměnných prostředí* (environment variables). Už jsme některé poznali (např. **`PATH`**, **`HOME`**). Shell poněkud stírá rozdíly mezi proměnnými prostředí a svými proměnnými, proto jsme na tento rozdíl dosud nenarazili. V této chvíli ale musíme shell donutit, aby proměnnou `start` přidal do „balíku“ těch proměnných, které bude (jako systémové) předávat do prostředí synovského procesu. Tyto proměnné se také označují jako *exportované* proměnné a zařazení další proměnné na tento seznam dosáhneme příkazem **`export`**. Jinou možností by bylo nastavit přepínač shellu „**`-a`**“ (*allexport*), a pak jsou automaticky exportovány všechny proměnné.

Při psaní výrazu do souboru nechceme mezi jednotlivými položkami odřádkovat. Abychom obešli problémy s používáním příkazu **`echo`**, poslechneme tentokrát normu a použijeme doporučovaný příkaz **`printf`**. Ten vypisuje text, který vznikne na základě tzv. *formátovacího řetězce* (první parametr příkazu), doplněním o hodnoty uvedené jako další parametry. Místa, kam chceme do řetězce hodnoty parametrů doplnit, se označí tzv. *formátovacími direktivami*. Ty tvoří znak procento a písmenko, které říká, jakou hodnotu parametru očekáváme a jaký tvar výpisu hodnoty chceme („**`%s`**“ značí výpis řetězce). Ve formátovacím řetězci lze také používat běžné escape-sekvence jako „**`\n`**“.

Ještě musíme zvážit, jak zpracovat vstupní řádku s několika položkami a jak připravit výraz pro **`expr`**. Výraz by stačilo postupně ukládat do proměnné a nakonec ho předat v parametrech příkazu **`expr`**. Jak ale zpracovávat postupně jednotlivá slova na řádce? Příkaz **`read`** (aspoň podle normy) vždy přečte celou řádku. Nelze tedy vytvořit cyklus, který by v každém kroku příkazem **`read`** přečetl jedno slovo. Na to budeme muset jinak. Nejsnazší bude uložit načtenou řádku do pozičních parametrů a zpracovat je v cyklu **`while`** s postupným voláním příkazu **`shift`**. Problém nastane pouze s hvězdičkou, která by se rozvinula na seznam jmen souborů. Tomu lze asi nejsnáze zabránit tak, že shellu expanzi zakážeme (nastavíme tzv. režim *noglob*) přepínačem „**`-f`**“ (*filenames*).

Řešení:

```
#!/bin/sh -f                    ... nastavení přepínače pro celý skript

zpracuj_podvyraz() {            ... rekurzivní volání a test úspěchu
  vysledek=`$0 $*` || exit 1
}

spocitej_podvyraz() {           ... vyhodnocení podvýrazu a test úspěchu
  vysledek=`expr 0 + $vyraz`
  [ $? -gt 1 ] && exit 1
}

if [ -z "$start" ]; then        ... jestliže skript běží poprvé ...
  start=$$
  export start                  ... exportování proměnné pro subshelly
  rm -f /tmp/$start.vyraz       ... příprava pracovního souboru
  trap "rm -f /tmp/$start.vyraz; exit" 0 2 3 15
  read radka || exit 1          ... načtení první řádky
  set -- $radka                 ... převod vstupu do pozičních parametrů
fi

vyraz=                          ... proměnná pro výpočet podvýrazu
while :; do                     ... hlavní smyčka (čtení řádek)
  while [ $# -gt 0 ]; do        ... vedlejší smyčka (zpracování slov)
    printf '%s ' "$1" >> /tmp/$start.vyraz
    case $1 in
    "(" )
      shift                     ... odstranění závorky z pozičních parametrů
      zpracuj_podvyraz $*
      set -- $vysledek          ... uložení výsledku a zbytku řádky
      vyraz="$vyraz $1"         ... přidání výsledku do výrazu
      shift;;                   ... odstranění výsledku z pozičních parametrů
    ")" )
      if [ $start = $$ ]; then
        echo "Chyba uzavorkovani" >&2
        exit 1
      else
        shift                   ... odstranění závorky z pozičních parametrů
        spocitej_podvyraz
        echo $vysledek $*       ... vypsání výsledku a zbytku řádky
        exit 0
      fi;;
```

```
    "=" )
      if [ $start != $$ ]; then
        echo "Chyba uzavorkovani" >&2
        exit 1
      else
        spocitej_podvyraz      ... závěrečný výpočet a výpis
        cat /tmp/$start.vyraz
        echo $vysledek
        exit 0
      fi;;
    "+" | "-" | "*" | "/" )
      vyraz="$vyraz $1"        ... přidání operátoru do výrazu
      shift;;
    [-0-9]*[!0-9]* | [!-0-9]* )
      echo "Chybna polozka vyrazu: $1" >&2
      exit 1;;
    * )
      vyraz="$vyraz $1"        ... přidání čísla do výrazu
      shift;;
    esac
  done
  read radka || exit 1
  set -- $radka
done
```

Poznámky:

První řádka skriptu může mít tvar „**#!***cesta* [*přepínače*]“. Pomocí ní se dá zvolit konkrétní shell, kterým chceme skript vykonávat, a také se tak dají nastavit přepínače.

Všimněte si ošetření výsledku volání **expr** ve funkci `spocitej_podvyraz`. Pokud je zadaný výraz roven nule, **expr** vrací návratovou hodnotu 1. Ovšem tento stav musíme vyhodnotit jako korektní výsledek a nikoliv jako chybu (jako ostatní nenulové hodnoty). Proto má podmínka pro ukončení skriptu při chybném výrazu tvar „**$? -gt 1**“.

Poznamenejme ještě, že i spuštěný příkaz ve zpětných apostrofech nastavuje návratovou hodnotu. A pokud se jeho výstup už pouze ukládá do proměnné, je tato návratová hodnota určující pro konečnou návratovou hodnotu celého příkazu.

Hlavní smyčka skriptu čte vstupní řádky. Ovšem cyklus „**while read**“ nemůžeme použít, neboť při vnořeném volání už je řádka načtená. Čtení musíme přesunout až na konec těla a jako řídicí příkaz pro příkaz **while**, dát něco, co nezdrží a skončí dobře. Tzv. *věčný cyklus* se v shellu obvykle píše pomocí příkazu „**:**“. Norma sice zná i „čitelnější“ příkazy **true** a **false**, ale u nich opět můžeme narazit s přenositelností.

Kód se na několika místech dělí podle toho, zda se jedná o první nebo další instanci skriptu, tj. zda proměnná `start` obsahuje aktuální PID (**`$$`**). Pokud by kód byl ještě trochu složitější, stálo by za úvahu rozdělení do dvou skriptů. V obou skriptech bychom přitom potřebovali sdílet pomocné funkce. Jenže funkce se do dalších skriptů nedědí. Museli bychom je uložit do třetího souboru a jeho kód sdílet v obou skriptech. K tomu slouží zvláštní příkaz „**`.`** *soubor*“, který v prostředí aktuálně běžícího procesu vykoná příkazy uložené v souboru. Tato forma volání skriptu umožňuje provádět operace jako nastavení proměnných, změna aktuálního adresáře nebo sdílení funkcí. Širšímu použití tohoto konstruktu ale brání mj. to, že se takto nedají volanému skriptu předat parametry.

Doplňme, že přesně takto je při přihlášení k systému vyvolán soubor „`.profile`“ v domovském adresáři uživatele. V něm je proto možné si upravit prostředí shellu (např. nastavení proměnné **`PATH`** nebo tvar promptu) podle vlastních představ. Jenom pozor, pokud budete s tímto skriptem experimentovat, nedávejte na jeho konec příkaz **`exit`**...

Řešení s třemi skripty by vypadalo nějak takto – pomocný skript `funkce.sh`:

```
zpracuj_podvyraz() {
  vysledek=`zavorka.sh $*` || exit 1
}
...
```

Rekurzivní skript `zavorka.sh`:

```
. funkce.sh
vyraz=
while :; do
  ...
done
```

Hlavní skript `kalkulacka.sh`:

```
. funkce.sh
start=$$
...
while :; do
  ...
done
```

Rekurzivní algoritmy jsou dalším typickým příkladem principu „divide et impera“. Úlohu si rozdělíme na menší podúlohy, ty vyřešíme a výsledky zpracujeme. Jenom je třeba vždy správně vyhodnotit, zda pro danou úlohu neexistuje lepší řešení. I v našem případě se lze bez rekurze obejít, ale obě řešení se v efektivitě příliš lišit nebudou.

Jak by řešení bez rekurze vypadalo? Pro ukládání jednotlivých podvýrazů bychom nepoužívali proměnné ale soubory (pro každou úroveň vnoření jeden), jako počitadlo úrovní bychom mohli použít proměnnou `patro`, do níž bychom postupně přidávali třeba znaky „`@`“. Jako pomocné soubory by se tedy používaly `/tmp/$$.@`, `/tmp/$$.@@` atd.

Abychom je pak mohli pohodlně smazat všechny najednou, potřebovali bychom zachovat v shellu možnost expanze hvězdičky. Proto nyní nenastavíme přepínač „**`-f`**“ globálně, ale budeme ho dle potřeby zapínat (příkazem „**`set -f`**“) a vypínat (příkazem „**`set +f`**“). Konečný výsledek by vypadal zhruba takhle (vynechali jsme ošetření chyb vstupu a kontrolu závorek):

```
patro=@
rm -f /tmp/$$.*
trap "set +f; rm -f /tmp/$$.*; exit" 0 2 3 15
set -f
while read radka; do
  for slovo in $radka; do      ... cyklus přes slova na řádce
    printf '%s ' "$slovo" >> /tmp/$$.vyraz
    case $slovo in
    "(" )                      ... zvýšení úrovně (patra)
      patro=@$patro
      rm -f /tmp/$$.$patro;;
    ")" )                      ... snížení patra, výpočet podvýrazu
      patro=`echo $patro | cut -c2-`
      expr 0 + `cat /tmp/$$.@$patro` >> /tmp/$$.$patro
      [ $? -gt 1 ] && exit 1;;
    "=" )                      ... závěrečný výpočet a výpis
      vysledek=`expr 0 + \`cat /tmp/$$.@\``
      [ $? -gt 1 ] && exit 1
      cat /tmp/$$.vyraz
      echo $vysledek
      exit 0;;
    * )                        ... přidání operátoru nebo čísla do výrazu
      echo $slovo >> /tmp/$$.$patro;;
    esac
  done
done
```

Proměnné prostředí představují také další variantu, jak nějakému programu předat dodatečné informace potřebné pro jeho práci. Jako příklad můžeme uvést příkaz **`ls`**. Tomu lze pomocí proměnné **`COLUMNS`** vnutit jinou šířku sloupce, než má ve skutečnosti terminál. Musíme ale proměnnou přidat do jeho proměnných prostředí. Zkuste si:

```
sinek:~> COLUMNS=40 ls
```

4.35. Něco o efektivitě

Zadání:

Napište program, který bude číst výstup příkazu „**`ls -ln`**“ a převede ho na formát, který by vypsalo volání bez přepínače „**`-n`**“.

Přepínač „**`-n`**“ (*number*) říká příkazu **`ls`**, aby nevypisoval jména vlastníků (uživatelů a skupin), ale pouze jejich čísla. Úkolem programu tedy bude nahradit čísla jmény.

Rozbor:

První nápad bude asi směřovat k jednoduchému řešení: budeme číst výstup příkazu a na každé řádce vybereme UID a GID vlastníka a prohledáme soubory `/etc/passwd` a `/etc/group`. Podívejme se, jak rychlé bude toto řešení. Řekněme, že výstup příkazu bude mít *n* řádek, soubor `/etc/passwd` *p* řádek a `/etc/group` *g* řádek. Pro každou z *n* řádek vstupu tedy projdeme (v průměrném případě) *p/2* řádek `/etc/passwd` a *g/2* řádek `/etc/group`. Představme si na chvíli, že všechna tři čísla jsou stejně velká. Pak náš skript bude v průměrném případě číst n^2 řádek. Takovému algoritmu se říká *kvadratický* a jeho nepříjemnou vlastností je, že pro velká *n* velmi rychle roste doba běhu programu. Zvláště v interpretovaných jazycích, jako je shell, je třeba na efektivitu našich programů dávat velký pozor. Pokusme se tedy najít jiné řešení.

Co kdybychom si soubory přečetli dopředu a převody UID a GID na jména si někde uložili? Ale kde? Přirozeným řešením v běžných programovacích jazycích jsou *pole* neboli proměnné, které mohou obsahovat sérii hodnot označených (celočíselnými) indexy. V shellu však pole nemáme. Přesněji řečeno: norma je neuvádí, přestože řada implementací je bude znát.

Pomoci nám může opět příkaz **`eval`**. Díky němu totiž namísto jednoho pole můžeme používat sérii proměnných se jmény `u_`*UID*. Pomocí editoru **`sed`** vytvoříme z řádek `/etc/passwd` správné přiřazovací příkazy pro shell a pomocí **`eval`** je vykonáme. A v místě použití necháme opět pomocí **`eval`** substituovat hodnotu proměnné správného jména. Pro skupiny použijeme stejný postup.

Řešení:

```
eval `sed 's/\([^:]*\):[^:]*:\([^:]*\):.*/u_\2=\1/' \
    /etc/passwd`
eval `sed 's/\(.*\):.*:\(.*\):.*/g_\2=\1;/' \
    /etc/group`
while read prava inody uid gid zbytek; do
    eval echo \$prava \$inody \$u_$uid \$g_$gid \$zbytek
done
```

Poznámky:

Pro správnou funkci se možná opět budete muset zbavit komentářových řádek v obou souborech. Situaci by nám ještě maličko zkomplikovalo, pokud by zadavatel trval na nějakém konkrétním řešení duplicit (uživatelů či skupin se shodným UID, resp. GID).

Výsledek nebude nijak úchvatný. Potřeboval by ještě pořádně naformátovat. Mohli bychom si trochu pomoci, kdybychom řádku nejprve přečetli celou do jedné proměnné, teprve poté ji rozdělili na slova a pak čísla za jména nahradili pomocí editoru **`sed`**:

```
while read line; do
    set -- $line
    eval u=\$u_$3 g=\$g_$4
    echo "$line" | sed "s/ *$3 *$4/ $u $g/"
done
```

Ale ani tak to nebude žádný zázrak. S formátováním by nám asi nejvíce pomohl příkaz **`printf`**. My si ale tento způsob řešení schováme do kapitoly 5.11.

Pole patří mezi rozšíření, po kterých programátor občas pošilhává. Obvykle se však vyplatí si trochu lámat hlavu, protože často lze najít zcela přenositelnou alternativu, která nakonec ani nemusí být výrazně méně srozumitelná nebo efektivní.

Metod, jak nahradit v shellu chybějící pole, existuje celá řada. Pokud byste například programovali nějakou úlohu s kalendářem a potřebovali uložit počty dnů v jednotlivých měsících do prvků pole, můžete si pomoci třeba takto:

```
pocty_dnu="31 28 31 30 31 30 31 31 30 31 30 31"
```

Jak se nyní dostanete k počtu dní v měsíci, jehož číslo je v proměnné `mesic`?

4.36. O efektivitě ještě jednou

Zadání:

Napište program, který vypíše seznam skupin v systému a ke každé napíše seznam uživatelů, kteří mají skupinu jako svoji primární.

Předpokládejte, že soubory `/etc/passwd` a `/etc/group` jsou natolik veliké, že není možno je uložit do paměti.

Rozbor:

Motivací pro tuto úlohu samozřejmě není zpracování souborů `/etc/passwd` a `/etc/group`. Ty by se do paměti jistě vešly. Ale i na nich si můžete vyzkoušet metody, jež byste museli použít, kdybyste potřebovali zpracovat soubory podstatně větší.

První nápad zřejmě povede, stejně jako v minulé kapitole, na kvadratické řešení. Četli bychom `/etc/group` a pro každou řádku (resp. GID) bychom hledali odpovídající řádky v `/etc/passwd`. Znamenalo by to pro každé GID znova přečíst celý soubor `/etc/passwd`. Minule jsme se kvadratickému řešení vyhnuli díky polím (resp. jejich vhodné náhražce). To teď nemůžeme.

Rozbor (dva zdroje):

Kdyby byly oba soubory setříděné (podle GID) a dokázali bychom střídavě číst z obou souborů, měli bychom vyhráno. Pro každé GID bychom četli z `/etc/passwd` jen tak dlouho, dokud bychom nenarazili na vyšší GID, potom bychom přečetli další řádku `/etc/group` a pokračovali v hledání uživatelů s tímto novým GID.

Problémem je, jak v shellu zařídit, aby četl střídavě řádky ze dvou souborů. Číst postupně po řádkách umí sice **`read`**, ale nejde mu přitom říci, ze kterého souboru, protože příkaz „**`read <`** *soubor*“ čte neustále první řádku. Řádku po řádce umí číst jen tehdy, když má soubor otevřený ve svém vstupním proudu. Museli bychom tedy nechat soubory otevřené, ale každý v jiném vstupním proudu. A to jde.

Řešení nabízejí deskriptory souborů. Vybereme si nějaká volná čísla, řekněme 3 a 4. Každý vstupní soubor otevřeme pro čtení a pomocí přesměrování („**`3<`** *soubor*“, resp. „**`4<`** *soubor*“) mu přiřadíme číslo deskriptoru. Příkaz **`read`** potom necháme namísto standardního vstupu číst řádky z proudu dat s konkrétním číslem deskriptoru (použijeme operátor duplikace vstupního deskriptoru „**`<&`**“).

Toto řešení je podobné tomu, jak by se úloha programovala v jiných jazycích.

Řešení (dva zdroje):

```
nastav_gid() {
    [ $g_gid -lt $u_gid ] && gid=$g_gid || gid=$u_gid
}
# zacatek hlavniho programu, priprava pomocnych souboru
grep -v '^#' /etc/passwd | sort -t: -k4,4n > /tmp/passwd.$$
grep -v '^#' /etc/group | sort -t: -k3,3n > /tmp/group.$$
echo :::100000 >> /tmp/passwd.$$
echo ::100000 >> /tmp/group.$$
{ # cely slozeny prikaz ma presmerovane deskriptory 3 a 4
    IFS=:
    # nacteme prvni radky
    read u_jm x x u_gid x <&3
    read g_jm x g_gid x <&4
    nastav_gid
    while [ $gid -lt 100000 ]; do
        if [ $g_gid -ne $gid ]; then
            # nalezena skupina, ktera neni v /etc/group
            echo "Skupina($gid): NEZNAMA"
        fi
        while [ $g_gid -eq $gid ]; do
            # vypis jmen vsech skupin se shodnym GID
            echo "Skupina($g_gid): $g_jm"
            read g_jm x g_gid x <&4
        done
        while [ $u_gid -eq $gid ]; do
            # vypis vsech uzivatelu se shodnym GID
            echo "    $u_jm"
            read u_jm x x u_gid x <&3
        done
        nastav_gid
    done
} 3< /tmp/passwd.$$ 4< /tmp/group.$$
```

Rozbor (slité soubory):

Druhé řešení používá myšlenkově velmi podobný algoritmus, ale jde na to jinak. Namísto čtení ze dvou zdrojů slijeme oba soubory do jednoho proudu dat, necháme si výsledek setřídit podle GID a pak prostě čteme řádky pouze z jediného zdroje. Díky různému formátu řádek přitom umíme rozlišit, ze kterého souboru která řádka pochází, takže nepřicházíme o žádnou informaci a můžeme použít podobný postup rozhodování jako v prvním případě.

Řešení (slité soubory):

```
{
    cut -d: -f4 /etc/passwd |
        sed 's/$/:p/' | paste -d: - /etc/passwd
    cut -d: -f3 /etc/group |
        sed 's/$/:g/' | paste -d: - /etc/group
} | grep -v ':[pg]:#' | sort -n | {
    IFS=:
    skupina=-1
    while read gid typ jm x; do
        if [ $gid -gt $skupina ]; then
            skupina=$gid
            if [ $typ = g ]; then
                echo "Skupina($skupina): $jm"
                continue
            else
                echo "Skupina($skupina): NEZNAMA"
            fi
        fi
        if [ $typ = g ]; then
            echo "Skupina($skupina): $jm"
        else
            echo "    $jm"
        fi
    done
}
```

Rozbor (`join`):

Každý příběh by měl mít svoji pointu. Tou je v tomto případě náš návštěvník ze světa databází, příkaz **`join`**. Zde ho použijeme přesně v jeho původním významu. Naše operace není nic jiného než spojení tabulek s upraveným výstupem.

Sloupcem, přes jehož hodnoty budeme řádky spojovat, bude samozřejmě číslo skupiny (tj. třetí, resp. čtvrté pole). Výstupní řádky pro naše účely musejí obsahovat číslo skupiny (neboli pole s označením 0), jméno skupiny (1.1) a jméno uživatele (2.1). A do výstupu bude třeba zahrnout i nespárované řádky obou souborů (přepínač „**`-a`**“).

Výstup příkazu **`join`** bude mít tedy řádky ve tvaru:

číslo_skupiny:*jméno_skupiny*:*jméno_uživatele*

Pokud nějaká skupina nebude známá, bude pole se jménem skupiny prázdné. Stejně tak bude prázdné pole se jménem uživatele, pokud nějaká skupina nebude mít žádného uživatele. Jinak bude ve výstupu tolik řádek, kolik uživatelů bude nalezeno.

Cyklus na zpracování výstupu příkazu **`join`** můžeme téměř opsat z minulého řešení.

Řešení (`join`):

```
grep -v '^#' /etc/passwd | sort -t: -k4,4 > /tmp/passwd.$$
grep -v '^#' /etc/group | sort -t: -k3,3 > /tmp/group.$$
join -t: -13 -24 -o0,1.1,2.1 -a1 -a2 \
    /tmp/group.$$ /tmp/passwd.$$ | sort -n |
{
    IFS=:
    skupina=-1
    while read gid g_jm u_jm; do
        if [ $gid -gt $skupina ]; then
            skupina=$gid
            echo "Skupina($skupina): ${g_jm:-NEZNAMA}"
        fi
        if [ -n "$u_jm" ]; then
            echo "    $u_jm"
        fi
    done
}
```

Poznámky:

V prvním řešení jsme použili metodu *zarážky*. Když si na konec souboru přidáme řádku s číslem, které určitě nemůže být GID, nemusíme pokaždé kontrolovat výsledek **`read`** a jednoduše počkáme na řádku se zarážkou. Ne vždy to jde. Ale pokud ano, jsou podobné metody výrazně rychlejší, a to jak z hlediska výroby, tak i výkonu programu.

Přiřazení vstupních souborů deskriptorům bychom opět mohli s mírným rizikem horší přenositelnosti udělat příkazem **`exec`**:

```
exec 3< /tmp/passwd.$$ 4< /tmp/group.$$
    read u_jm x x u_gid x <&3
...
```

Ve druhém řešení jsme na začátek každé řádky přidali třídicí klíč (aby bylo možné „smíchané“ soubory třídit) a zároveň příznak „`p`“, resp. „`g`“, který má dokonce dvojí funkci. Za prvé zařídí, že poznáme zdroj, odkud řádka pochází. A za druhé zabezpečí, že řádky z `/etc/group` přijdou na řadu před řádkami z `/etc/passwd`, a tak i ve výstupu budou řádky s názvem skupiny předcházet seznamu uživatelů.

Ve třetím řešení je třeba zohlednit, že příkaz **`join`** používá lexikografické třídění. Pomocné soubory proto musíme setřídit stejně (bez přepínače „**`-n`**“). V tom případě ovšem musíme zadat třídicí klíč ve tvaru „**`-k3,3`**“ („klíčem je třetí pole“), protože klíč ve tvaru „**`-k3`**“ („klíčem je text od třetího pole do konce řádky“) by nefungoval dobře.

4.37. Když text nestačí

Zadání:

Napište program, který dostane jako parametr jméno souboru ve formátu DBF a vypíše seznam sloupců ve tvaru: „*pořadové_číslo šířka_sloupce jméno_sloupce*“.

Formát DBF se používá pro ukládání jednoduchých databází. Obsahuje vždy jednu tabulku a začíná popisem sloupců tabulky. Jistě někde najdete nějaký vhodný příklad na Internetu (např. na webových stránkách knihy [13]).

Soubor ovšem není textový, má binární formát s následující strukturou:

<table>
<tr><td></td><td>0x00</td><td>...</td><td>0x0a</td><td>0x0b</td><td>...</td><td>0x0f</td></tr>
<tr><td>0x00</td><td colspan="6" rowspan="2"></td></tr>
<tr><td>0x10</td></tr>
<tr><td>0x20</td><td colspan="3">jméno 1. sloupce</td><td colspan="3"></td></tr>
<tr><td>0x30</td><td>šířka</td><td colspan="5"></td></tr>
<tr><td>0x40</td><td colspan="3">jméno 2. sloupce</td><td colspan="3"></td></tr>
<tr><td>0x50</td><td>šířka</td><td colspan="5"></td></tr>
<tr><td></td><td colspan="6">... další sloupce</td></tr>
<tr><td>0xn0</td><td>CR</td><td colspan="5">... vlastní data tabulky</td></tr>
</table>

Prvních 32 bytů je pro nás nepodstatných a pak následují bloky dlouhé 32 bytů, kde nás bude zajímat vždy prvních 11 znaků (jméno sloupce) a 17. znak, v němž je zakódovaná šířka sloupce v tabulce. Jméno sloupce má nejvýše 10 znaků a do délky 11 je doplněno znaky NUL. Seznam sloupců je ukončen znakem CR (znak s ordinální hodnotou 13) v místě, kde by jinak začínalo jméno dalšího sloupce.

Rozbor:

Na čtení binárního souboru nám nebudou stačit nástroje, které jsme doposud poznali. Z historických důvodů má UNIX daleko silnější podporu textových formátů (to už jsme ostatně měli možnost poznat), nicméně je možné zpracovávat i formáty binární. Nebude to ale žádná procházka růžovou zahradou.

Klíčovým problémem bude jednak rozpoznání znaku CR, ale hlavně „dekódování“ šířky sloupce z binární hodnoty na celé číslo. Pro tyto úkoly se nám bude hodit užitečný pomocník, utilita **`od`** (*octal dump*). Umí vypsat obsah souboru po jednotlivých bytech a jejich hodnoty umí navzdory svému názvu vypsat nejen oktalově (osmičková soustava byla populární v době vzniku UNIXu), ale též hexadecimálně a dekadicky. Použijeme-li přepínač „**`-t u1`**“ (vypiš jednotlivé znaky v desítkové soustavě jako nezáporná čísla bez znamének), dostaneme výpis souboru ve tvaru:

```
0000000    80 111 114  97 100 105   0   0   0   0   0  78 ...
0000020     4   0   0   0   0   0   0   0   0   0   0   0 ...
```

Tento tvar už snadno zpracujeme známými nástroji. Konec seznamu sloupců poznáme tak, že lichá řádka bude začínat znakem s kódem `13` (na některých systémech bude vypsán jako „`013`“). Šířka sloupce bude zapsána jako hodnota prvního znaku na sudých řádkách. Nenechte se zmást adresami v záhlaví řádek – ty jsou implicitně oktalové.

Druhým pomocníkem nám bude utilita **`dd`**, velmi obecný a univerzální nástroj na kopírování dat. Použijeme jen zlomek její funkčnosti k tomu, abychom ze souboru ve formátu DBF vyjmuli správný blok dat o velikosti 32 bytů. Z něj musíme získat prvních 11 znaků (jméno sloupce). Na to můžeme opět použít příkaz **`dd`**.

Názvy sloupců jsou v souboru doplněny do délky 11 potřebným počtem znaků NUL. Těch se budeme muset zbavit. Mohli bychom si pomocí příkazem „**`tr -d '\0'`**“, ale půjde to ještě snáze. Název sloupce musíme vypsat jako součást výstupní řádky, takže je vhodné si ho nejprve uložit do proměnné pomocí zpětných apostrofů. A znaky NUL se nám přes zpětné apostrofy do proměnné nedostanou.

Nyní už můžeme algoritmus dokončit. První dvě řádky výstupu příkazu **`od`** budeme ignorovat a zbytek budeme v cyklu číst vždy dvěma voláními příkazu **`read`**, čímž do proměnné `znak` načteme při prvním volání první znak názvu sloupce (a budeme ho testovat na hodnotu `13`) a druhým voláním šířku sloupce.

Řešení:

```
od -t u1 $1 | {
    read radka
    read radka
    sloupec=@
    while read adresa znak zbytek && [ $znak -ne 13 ]; do
        read adresa znak zbytek
        nazev=`dd if=$1 bs=32 count=1 skip=${#sloupec} |
            dd bs=11 count=1`
        echo ${#sloupec} $znak $nazev
        sloupec=@$sloupec
    done 2> /dev/null
}
```

Poznámky:

Příkaz **`dd`** se zcela vymyká všemu, co jsme zatím poznali. Jeho název a formát zápisu vycházejí z tradic dalšího z myšlenkových předchůdců UNIXu, a to systému IBM 360. V jeho jazyce JCL (*Job Control Language*) se vyskytoval (a vlastně dodnes vyskytuje) příkaz DD (*Data Definition*) popisující vstupně-výstupní soubory a používající formát parametrů „*klíčové_slovo=hodnota*“. A tento pomník překvapivě nepodlehl zubu času ani snahám o převedení na nějakou podobu bližší jiným unixovým příkazům.

Příkaz čte soubor zadaný parametrem **if** (*input file*), implicitně standardní vstup, a vypisuje výsledná data do souboru zadaného parametrem **of** (*output file*), implicitně standardního výstupu. Dalšími parametry, jež budeme potřebovat, jsou: **bs** (*block size*, velikost bloků, které příkaz bude ze vstupu číst), **count** (počet zpracovávaných bloků) a **skip** (počet bloků, které se mají na začátku přeskočit).

O své práci program vypisuje informace na standardní chybový výstup, proto jsme tento výstup potlačili. Pro jednoduchost jsme to udělali pro celý cyklus najednou (a tedy za klíčovým slovem **done**). Maličko tím riskujeme, že se nedozvíme případné chyby, ke kterým by došlo uvnitř cyklu, ale v našem případě tam vlastně není příkaz, který by mohl skončit chybou.

Ke získání jména sloupce by bylo možné použít i příkaz „**cut -c1-11**“. Jenom bychom mu pro jistotu měli omezit vstup (anebo výstup) na právě jednu řádku, protože mezi binárními znaky následujícími v hlavičce za jménem sloupce by se snadno mohl náhodou vyskytovat znak konce řádky, který by způsobil, že by příkaz **cut** vypsal dvě řádky, první správnou a druhou zcela nesmyslnou.

Vůbec nejjednodušší by bylo použít příkaz „**head -c11**“ (vypiš prvních 11 znaků). Tady ale analogie s příkazem **tail**, který stanovení rozsahu výpisu počtem znaků umí, bohužel selhává. Ve spoustě implementací jistě tento přepínač najdete, ale norma ho nekodifikuje.

Pokud máte obavy, zda nebude časově náročné několikanásobné otevírání souboru, pak máte do jisté míry pravdu. Jenomže spuštění určitého počtu příkazů (procesů) pro každý sloupec DBF souboru se tak či onak nevyhneme a počet sloupců asi nebude tak velký, aby to znamenalo nepřiměřenou zátěž.

Jiná situace by ovšem nastala, pokud bychom soubor nejenom otevírali, ale také ho vždy celý od začátku četli. Toho se ale v našem případě bát nemusíme, protože příkaz **dd** si nejprve spočítá, jak velkou část souboru chceme přeskočit, a začne číst rovnou od správného místa.

Této operaci se říká *seek* a používají ji obvykle i implementace příkazu **tail** při zadání přepínače „**-c**“, takže ani u nich se není třeba obávat zbytečného zpoždění.

Ale pozor – tvrzení platí jen tehdy, pokud příkazy **dd** nebo **tail** čtou soubor z disku a ne třeba vstup z roury. Jaký je rozdíl mezi zdánlivě stejnými kombinacemi příkazů:

```
tail -c +1000000 soubor | head -c 1
head -c 1000000 soubor | tail -c 1
```

Ve výsledku žádný, ale v rychlosti může být až řádový.

Ovšem úplně jiná je situace při použití přepínače „**-n**“. Rychlost příkazů

```
tail -n +100000 soubor | head -n 1
head -n 100000 soubor | tail -n 1
```

se už tolik lišit nebude. Tady si totiž příkaz **tail** nemůže spočítat, kolik bytů od začátku souboru se nachází řádka s konkrétním pořadovým číslem, a nezbude mu nic jiného než celý začátek souboru opravdu přečíst.

Znak CR (*carriage return*, návrat vozíku) se v MS Windows používá (v kombinaci se znakem LF) jako oddělovač konce řádek textových souborů. Díky tomu existuje odpovídající escape-sekvence („**\r**“), kterou zná třeba příkaz **printf**. Mohli bychom si tedy s jeho pomocí znak uložit do zvláštní proměnné `cr` a tu poté používat pro testování obsahu proměnné `nazev`:

```
cr=`printf '\r'`
...
    while ...
        nazev=`...`
        case $nazev in
        $cr* ) break;;
        esac
        ...
```

V této ukázce si všimněte poměrně elegantního řešení testu, zda hodnota proměnné `jmeno` začíná znakem CR. Samozřejmě, mohli byste test provést i jinými způsoby:

```
if echo $jmeno | grep -q ^$cr; then ...
if [ `echo $jmeno | cut -c1` = $cr ]; then ...
```

Tato řešení naprosto vyhovují – tedy až na jednu maličkost: každé totiž představuje několik procesů navíc. V našem případě se pochopitelně spuštění několika procesů na rychlosti neprojeví, ale uvědomte si, jak by se situace dramaticky změnila, kdyby podobný test byl uvnitř nějakého složitějšího cyklu – třeba v nějakém kvadratickém řešení, kde by se vykonával n^2-krát pro n řekněme 10 000...

V takové situaci člověk začne přemýšlet, zda neobětovat přenositelnost a nepoužít třeba konstrukt „**${jmeno:0:1}**“ (tj. z hodnoty proměnné `jmeno` vezmi pouze první znak), který má **bash**:

```
if [ "${jmeno:0:1}" = $cr ]; then ...
```

Jenomže my jsme stejného efektu dosáhli zcela přenositelným a určitě přinejmenším stejně srozumitelným způsobem.

Další náměty na procvičení

1. V kapitole 4.14 jsme použili příkaz **`basename`**. Naprogramujte tento příkaz v shellu.
2. V kapitole 4.21 jsme aktualizovali v souborech rok. Ale neaktuálně. Upravte skript tak, aby text měnil na opravdu aktuální rok.
3. V kapitole 4.28 jsme diskutovali způsoby, jak jednoduše zařídit, aby se ve skriptu pomocí přepínače „**`-v`**“ dalo zapínat, resp. vypínat vypisování ladicích zpráv. Zkuste to bez příkazu **`eval`**.
   Návod: Jaký je rozdíl mezi příkazy „**`echo`**“ a „**`:`**“?
4. V kapitole 5.17 rozebíráme rozdíly v zadávání přepínačů příkazu **`ps`** mezi různými systémy a navrhujeme pro potřeby následných volání ve skriptu zjistit přípustnou kombinaci přepínačů dopředu, abychom už dále volali příkaz **`ps`** správně. Napište takový test a navrhněte, jak se bude příkaz **`ps`** ve skriptu pomocí jeho výsledku volat.
5. Vyřešte příklad z kapitoly 4.7 efektivněji. Program nemusí zbytečně prohledávat další adresáře, jakmile už potřebný soubor nalezne.
6. V řešení kapitoly 4.26 jsme při zpracování přepínačů nekontrolovali, zda hodnotou přepínače je opravdu číslo, a nezabývali jsme se ani případem volání bez jména souboru. Doplňte obě rozšíření.
7. Naše řešení z kapitoly 4.32 neumí hledat skryté soubory. Napravte to.
8. Vyřešte příklad z kapitoly 1.17 pomocí čtení řádek `/etc/group` vhodným cyklem.
9. Některé implementace příkazu **`cat`** mají přepínač „**`-n`**“, který způsobí očíslování vypisovaných řádek. Naprogramujte takový příkaz v shellu.
10. V předchozí části knihy (o editorech) jsme jako námět na procvičení uváděli převod `/etc/passwd` do formátu HTML. Zkuste to přímo v shellu bez editoru.
11. V kapitole 4.35 jsme navrhli možnost, jak uložit počty dnů v jednotlivých měsících do proměnné `pocty_dnu`. Napište funkci `pocet_dnu_v_mesici`, která bude využívat tuto proměnnou, bude mít jako parametr číslo měsíce a vrátí počet dnů. Rozmyslete dobře implementaci i rozhraní (způsob volání, předání parametrů funkci a výsledku zpět volajícímu) tak, aby bylo zapotřebí co nejméně procesů. Funkci můžete také doplnit o správné chování pro přestupné roky.

12. V systému bývá zvykem, že se u logů (souborů se zprávami operačního systému nebo aplikací) provádí tzv. *rotace*. Soubor `xy` se přejmenuje na `xy.0`, `xy.0` na `xy.1` atd. až do nějaké horní hranice, soubory s číslem nad touto hranicí se vymažou. Napište program, který dostává dva parametry – jméno souboru a maximální počet uložených kopií (lze vynechat, implicitní hodnota je 30) – a provede rotaci.

13. Některé shelly obsahují příkaz **`pushd`**, který má efekt příkazu **`cd`**, ale zapamatuje si cestu k původnímu adresáři. Případné další volání **`pushd`** opět uloží starý adresář a opět provede **`cd`**. Následně je možné se vrátit do „předchozího" adresáře příkazem **`popd`**. Této datové struktuře se říká *LIFO* (což není zkratka mého jména, ale výrazu „Last In, First Out" – první se z ní vybírá to, co bylo naposledy uloženo) nebo také *zásobník* (opakem je *fronta* neboli *FIFO*). Rozmyslete si jak zásobník implementovat a napište funkce, které činnost příkazů **`pushd`** a **`popd`** emulují (nahrazují).

14. Implicitní formát výstupu příkazu **`date`** odpovídá zadání formátovacího řetězce ve tvaru „**`+%a %b %e %H:%M:%S %Z %Y`**". Předpokládejte, že máte povoleno použití pouze formy příkazu **`date`** bez parametrů, a s jeho pomocí napište skript, který bude mít jako první parametr formátovací řetězec příkazu **`date`** a vypíše odpovídající výstup. V zadaném formátovacím řetězci se pochopitelně smějí vyskytovat pouze výše uvedené direktivy, ale je v něm třeba ošetřit výskyty „nepříjemných" znaků (jako jsou mezery, lomítka, zpětná lomítka).
    Návod: Z výstupu **`date`** vytvořte sadu příkazů pro editor **`sed`**.

15. Doplňte ještě maličkost – direktivu „**`%%`**". A zkontrolujte funkčnost úpravy třeba na parametru „**`+%%a`**". Jak vidno, úplná maličkost to není...

16. A do třetice můžete přidat i všechny ostatní direktivy – kromě těch, které nejste z implicitního výstupu schopni stanovit (jako **`%j`** nebo **`%U`**).

17. Naprogramujte příklad z kapitoly 5.17 v shellu bez použití filtru **`awk`**. Zkuste přitom implementovat oba diskutované přístupy (jedno volání příkazu **`ps`** pro celý program vs. jedno pro každý proces) a porovnejte jejich rychlosti podobně, jako to v kapitole 5.17 děláme u varianty s filtrem **`awk`**.

18. Naprogramujte příklad z kapitoly 5.18 v shellu bez použití filtru **`awk`**.

19. Naprogramujte příklad z kapitoly 5.13 v shellu bez použití filtru **`awk`**.
    Návod: Pro uchování návazností RFC použijte proměnné `RFC`*nnnn* nebo soubory. Bloky řádek si můžete postupně ukládat do proměnné, než je budete zpracovávat, anebo si data předzpracujte jiným způsobem, abyste program urychlili.

Část 5 Filtr AWK

Programovatelný filtr **awk** je velmi důležitým nástrojem, který posouvá možnosti programování v shellu o stupeň výš, protože vyplňuje některé mezery v programovacích technikách shellu, resp. editorů. Často dokáže řešit problémy, které se při první hrubé analýze zdají jako v shellu neřešitelné, a to při zachování vysoké efektivity vývoje aplikace. Nespornou výhodou (zvláště proti některým svým následovníkům jako třeba programovacímu jazyku Perl) je přitom jasná a přehledná syntaxe a dobrá přenositelnost. Oproti shellu samotnému je navíc rychlejší, zvláště pokud dokážeme využít výhody zpracování vstupu nebo asociativních polí.

5.1. Zase od píky

Zadání:

Napište skript, který vypíše k zadanému souboru jméno skupinového vlastníka.

Rozbor:

Je jasné, že základem řešení bude příkaz „**ls -ld**“. Ovšem starý známý problém spočívá v tom, že potřebujeme vypsat čtvrtý sloupeček, a to bez ohledu na počty mezer okolo. Už jsme viděli řešení pomocí utilit **tr** a **cut**, editoru **sed** nebo příkazu **read**. Nyní si ale ukážeme daleko snazší přístup.

Vstříc nám vychází další užitečná utilita, **awk**. Je to *programovatelný filtr* – podobně jako editor **sed** čte data ze vstupu, na základě svého programu provádí požadované operace a (obvykle) v průběhu nebo na závěr práce něco vypisuje.

Klíčovou vlastností **awk**, kterou zde využijeme, je schopnost rozdělit vstupní řádku na *pole*, a to přesně tak, jak v této chvíli potřebujeme. Implicitním oddělovačem polí je pro filtr **awk** posloupnost bílých znaků. Nás bude tedy zajímat čtvrté pole vstupní řádky a na to se v jazyce **awk** odkážeme pomocí konstruktu „**$4**“. Postupně si tuto informaci budeme zpřesňovat, ale zatím to takto stačí.

Hodnotu pole vypíšeme příkazem **print**. I o něm si časem povíme více, v této chvíli nám stačí vědět, že má-li parametr, opíše ho na standardní výstup a odřádkuje.

Řešení:

```
ls -ld $1 | awk '{ print $4 }'
```

Poznámky:

Vidíme, že podobností mezi programy **awk** a **sed** je více – oba například dostávají sadu příkazů (neboli svůj program) jako první parametr na příkazové řádce.

Pokud si marně lámete hlavu s překladem slova **awk**, tak to nedělejte. Jmenuje se totiž po otcích zakladatelích. Byli to pánové Aho, Weinberger a Kernighan.

Zvláště to poslední jméno by vám mohlo něco připomínat. Ano, je to jeden z autorů jazyka C. Jak se budeme postupně prokousávat syntaxí jazyka **awk**, bude zřejmé, že podoba s jazykem C vůbec není náhodná. Pokud umíte jazyk C nebo některé jiné jazyky z něj odvozené, budete dokonce schopni už v této chvíli řadu programů v jazyce **awk** přečíst a pochopit, aniž byste se cokoliv učili. To je nesporná výhoda **awk**.

Například vidíme, že stejně jako v jazyce C se jednotlivé stavební kameny (atomy) jazyka pro lepší čitelnost oddělují mezerami. Náš program bychom mohli zapsat i takto:

```
ls -ld $1 | awk '{print$4}'
```

5.2. Počitadlo délek souborů

Zadání:

Vypište součet velikostí všech obyčejných souborů v adresáři.

Rozbor:

Základem bude samozřejmě opět příkaz „**ls -l**“, tentokrát potřebujeme vzít páté pole každého záznamu o obyčejném souboru a tato čísla sečíst.

Vlastní program v jazyce **awk** se skládá z dvojic „*vzor* **{** *akce* **}**“ (říkejme jim třeba *větve*). Filtr postupně čte vstupní soubor, pro každou řádku projde všechny větve programu, u každé vyhodnotí, zda je vzor platný, a v takovém případě provede akci. Tady opět vidíme značnou podobnost s editorem **sed**.

Výběr správných řádek (s typem souboru „-“) provedeme pomocí vzoru ve tvaru *regulárního výrazu*. Kromě toho budeme potřebovat ještě inicializační větev (ta má zvláštní vzor „**BEGIN**“) a větev pro závěrečný výpis (se vzorem „**END**“).

Umíme vybrat řádky a umíme z nich vzít páté pole. Pro sčítání budeme potřebovat ještě někam postupně ukládat hodnotu mezisoučtu. Pro tento účel i v **awk** pochopitelně slouží *proměnné*. Použijeme jednoduchou (*skalární*) proměnnou `suma`, do ní přičteme (na každé správné řádce) obsah pátého pole a na závěr hodnotu proměnné vypíšeme.

Řešení:

```
ls -l | awk '
BEGIN { suma = 0 }
/^-/ { suma += $5 }
END { print suma }'
```

Poznámky:

Už jsme viděli tři typy vzorů pro naše tři větve, pojďme si popis vzorů dokončit:

- Prvním vzorem je klíčové slovo **BEGIN**. Tato větev se vykoná právě jednou, a to na úplném začátku běhu programu, ještě než se přečte první řádka. Naopak, posledním vzorem je **END** a tato větev se provede také právě jednou, na úplném konci programu. Je asi zřejmé, k čemu tyto větve slouží – k inicializaci (např. proměnných) a k závěrečným pracím (různé sumarizace, úklid apod.).
- Druhá větev má jako vzor *regulární výraz* (zapsaný mezi lomítka). Jeho význam je asi také zřejmý – pokud vstupní řádka vyhovuje danému regulárnímu výrazu, je vzor platný a akce se provede. Tady vidíme další paralelu s editorem **sed**.

- Vzorem může rovněž být úplně libovolný *logický výraz* v jazyce **awk** (ještě si o nich něco povíme), pak se akce provádí na každé načtené řádce, pokud právě v té chvíli logický výraz platí.
- A konečně je možné vzor úplně vynechat, pak se akce provede vždy. To jsme viděli v minulé kapitole a toto chování opět známe z editoru **sed**.

Proměnné se v jazyce **awk** pojmenovávají podle obvyklých pravidel – identifikátor smí obsahovat pouze alfanumerické znaky a musí začínat písmenem nebo znakem „_“. To známe z jiných jazyků a ostatně i z shellu. I zde je přitom podstatné použití malých a velkých písmen.

Významný rozdíl oproti shellu je ale v zápisu jejich použití. Jazyk **awk** je „normální“ jazyk „algolského“ typu, nikoliv textový preprocesor. Proto se v něm proměnné označují identifikátory, jimž nepředchází žádný speciální znak (jako v shellu dolar). Lexikální analyzátor jazyka totiž ví, kde může a nemůže být proměnná. V shellu naopak můžeme chtít proměnnou substituovat prakticky kdekoliv, a proto je třeba její název uvodit metaznakem.

Je důležité nezaměňovat syntaxi proměnných a polí vstupní řádky. Pole vstupu jsou vlastně jen jiným druhem proměnných, na něž se v jazyce **awk** odkazujeme pomocí dolaru, za kterým následuje číslo požadovaného pole, tedy „**$1**“, „**$2**“ atd.

Proměnné v jazyce **awk** mají stejně jako v shellu *textové* hodnoty, ale pokud jsou použity v kontextu, který vyžaduje číslo, je jejich textová hodnota na číslo převedena. Naše proměnná se tedy bude v celém programu chovat jako normální celočíselná proměnná, i když ve skutečnosti bude její hodnota uložena jako řetězec.

Přesněji řečeno, bude se chovat jako číslo. V našem případě to opravdu bude číslo celé, ale jakmile bychom někde zapsali desetinné číslo nebo ho dostali jako výsledek nějaké operace (např. dělení), hodnota proměnné bude obsahovat i desetinná místa. Pro převod na celá čísla lze použít *vestavěnou funkci* **int**:

```
prumer = int( suma / pocet )
```

Jako aritmetické operátory se požívají standardní symboly, vysvětlení si zaslouží snad jen operátor „**%**“, který znamená „zbytek po dělení“, a „**^**“ (umocnění).

Pro přiřazení se používá jako obvykle symbol „**=**“, ale stejně jako v jazyce C je celé přiřazení *výraz*, jehož hodnotou je to, co se přiřazuje, a je tedy možné s touto hodnotou dále pracovat. Třeba výraz „**i = j = 2**“ přiřadí hodnotu 2 do obou dvou proměnných.

A pro ty, kteří neznají jazyk C, ještě poznámka k operátoru „**+=**“. Znamená přičtení hodnoty na pravé straně do proměnné na levé straně. Je to vlastně pouze zkratka zápisu „**suma = suma + $5**“. Podobným způsobem lze použít všechny aritmetické operátory.

5.3. Není regexp jako regexp

Zadání:

Vypište řádky `/etc/passwd`, kde je nejvýše trojmístné UID shodné s GID.

Rozbor:

Tento příklad jsme řešili pomocí editoru (v kap. 3.5). Jaké bude řešení v jazyce **awk**?

Že filtr **awk** umí rozdělit vstupní řádku na pole, to už víme. A asi nepřekvapí, že bude mít i prostředek, jak říci, co má být *oddělovačem* polí (přepínač „**-F**“, *field separator*).

Shodu třetího (UID) a čtvrtého (GID) pole popíšeme snadno – testem na rovnost.

Požadavek na trojmístné UID jsme v editoru zadávali stanovením přesného počtu opakování podvýrazu v regulárních výrazech. Na tuto vlastnost regulárních výrazů se ale v **awk** bohužel nelze spolehnout. V tomto konkrétním případě nám to nevadí, ale na rozdíly mezi oběma typy regulárních výrazů se budeme muset podrobněji podívat. Pro řešení této úlohy použijeme obyčejné aritmetické porovnání.

Řešení:

```
awk -F: '$3 == $4 && $3 < 1000' < /etc/passwd
```

Poznámky:

V jazyce **awk** by podle normy měly být implementovány tzv. *rozšířené* regulární výrazy. Co nám tento odlišný *dialekt* jazyka regulárních výrazů přináší nového:

výraz	znamená
*z***+**	*alespoň* jeden výskyt znaku (nebo podvýrazu) *z*
*z***?**	*nejvýše* jeden výskyt znaku (nebo podvýrazu) *z*
x|*y*	*výběr* mezi dvěma (příp. více) variantami

Rozšířené regulární výrazy je možné využívat i u příkazu **grep**, pokud použijeme přepínač „**-E**“.

Drobný problém ovšem spočívá v tom, že různí implementátoři si termín „rozšířené“ vykládají různě. Například složené závorky (vyjádření počtu opakování podvýrazu, které jsme poznali u editorů a které jsou v normě uvedeny i u jazyka **awk**) bohužel všude nenajdeme. Pokud tedy opravdu chceme přenositelné řešení pomocí regulárních výrazů, musíme napsat:

```
$3 == $4 && /:([0-9]|[1-9][0-9]|[1-9][0-9][0-9]):/
```

A na předchozím příkladu si můžete všimnout ještě jedné zrady. Část regulárního výrazu se třemi variantami zápisu čísla jsme potřebovali uzavřít do závorek. A závorky

jsme napsali bez uvození zpětnými lomítky! Ano, rozšířené regulární výrazy se v tomto bodě chovají „konzistentněji“ a všechny metaznaky jsou definované „normálně“, tj. napíšeme-li závorku se zpětným lomítkem, je to obyčejná závorka, zatímco zapsaná samostatně představuje metaznak. V základních regulárních výrazech (v editorech) je to, bohužel, obráceně.

Pro jistotu připomeňme, že regulární výrazy i v jazyce **awk** znamenají test na shodu nějakého podřetězce testovaného řetězce. Proto jsme v minulé kapitole museli ve vzoru ukotvit znak minus na začátek řádky pomocí metaznaku „**^**“.

Operátor „**==**“ znamená test na rovnost (čísel nebo řetězců), jeho opakem je „**!=**“. Pozor na záměnu s operátorem „**=**“, který znamená přiřazení do proměnné. Syntakticky je totiž taková konstrukce v pořádku, pouze „dělá něco jiného“. Kdybychom napsali:

```
$3 = $4 && $3 < 1000
```

obsah třetího pole řádky by se přepsal obsahem čtvrtého pole a tato hodnota by byla předmětem testování v podmínce. Jak by to dopadlo? To by záleželo na obsahu pole:

- V případě, že by obsahem textu bylo číslo, chápal by se tento logický podvýraz jako *aritmetický* a jakákoliv nenulová hodnota by znamenala *pravdu*, zatímco nula *nepravdu*.
- V opačném případě by se podvýraz vyhodnocoval jako *řetězcový* a *pravda* by byla reprezentována libovolným neprázdným řetězcem.

Porovnávací operátory „**==**“, „**!=**“, „**<**“, „**<=**“, „**>**“ a „**>=**“ se v jazyce **awk** chovají poměrně očekávaným způsobem: je-li jedním operandem číslo (resp. aritmetický výraz) a druhým operandem číslo nebo řetězec představující číslo, porovnává se velikost čísel, jinak se porovnávají abecedně (lexikograficky) jako řetězce.

Logické výrazy plně kopírují model jazyka C. Jako logické spojky se (podobně jako v shellu) používají operátory „**&&**“ (a zároveň), „**||**“ (nebo) a také zde existuje operátor „**!**“ (negace). Pokud v logickém výrazu vystupuje číslo, je hodnota nula považována za nepravdu a cokoliv jiného za pravdu. Vyhodnocování probíhá podmíněně (viděli jsme to už třeba u příkazu **find**) – jakmile je hodnota nějakého podvýrazu jasná (pravda nebo nepravda), v jeho vyhodnocování se dále nepokračuje.

Jediná větev našeho programu obsahuje jen vzor ve tvaru logického výrazu a žádnou akci. Chybějící příkazová část větve (akce) znamená totéž co „**{ print }**“ a příkaz **print** bez parametrů opíše celou vstupní řádku.

5.4. Konečný automat

Zadání:

Vypište do sloupečku seznam čísel skupin, do nichž patříte.

Vycházejte přitom opět z upraveného výstupu příkazu **id** (jako v kap. 3.9):

```
sinek:~> id | tr ' =' '[\n*]'
...
groups
1004(forst),0(wheel),1(daemon),999(kernun),1028(j28)
```

Rozbor:

Když jsme úlohu řešili v části o editorech, zformulovali jsme algoritmus: Čti soubor, a jakmile najdeš řádku obsahující „`groups`“, načti další řádku a tu zpracuj. Pro **awk** ale potřebujeme trochu jinou logiku. Tady totiž máme „zadarmo“ hlavní cyklus, který načítá řádky vstupu, v našem popisu algoritmu by se tedy vůbec nemělo objevit slovo „načti“.

Pokud se na náš vstup budeme dívat jako na „bloky“ řádek, uvozené řádkou složenou jenom z písmen („`uid`“, „`gid`“, resp. „`groups`“), můžeme algoritmus popsat takto: Pokud jsi uvnitř bloku „`groups`“, zpracuj řádku.

Budeme si tedy do proměnné `blok` ukládat název aktuálního bloku (když budeme na první řádce bloku) a tato hodnota bude určovat, zda další řádky zpracováváme nebo ne.

Řádku se skupinami potřebujeme rozlámat podle čárek. To už umíme, nastavíme oddělovač polí na čárku. Pak jednotlivá pole postupně zpracujeme v cyklu (**for**) od jedné do počtu polí – ten najdeme v předdefinované proměnná **NF** (*number of fields*).

Nakonec potřebujeme vypsat část pole před závorkou. V tom nám pomohou funkce **index** (vrací pozici výskytu jednoho řetězce ve druhém) a **substr** (vrací podřetězec daný pozicí a délkou).

Řešení:

```
id | tr ' =' '[\n*]' | awk -F, '
/^[a-z]*$/ { blok = $0; next }
blok == "groups" {
  for( i = 1; i <= NF; i++ ) {
    pozice = index( $i, "(" )
    if( pozice > 0 )
        print substr( $i, 1, pozice - 1 )
    else
        print $i
  }
}'
```

Poznámky:

Programování v jazyce **awk** se podobá programování *konečných automatů* – na základě nějakého stavu (tj. hodnoty proměnné `blok`) a na základě obsahu vstupu se provede nějaká akce a případně změní stav (neboli proměnná, příp. více proměnných).

V tomto programu už můžeme poprvé spatřit případ více větví, jejichž podmínky (vzory) by se mohly vztahovat na jednu řádku. V takovém případě se opravdu zkoušejí všechny vzory a případně provádějí všechny akce, u nichž je podmínka splněna. To opět připomíná chování editoru **sed**.

- První větev obsahuje podmínku (vyjádřenou pomocí regulárního výrazu), že právě zpracováváme úvodní řádku bloku. Aktuální stav automatu je přitom nezajímavý, protože ho okamžitě změníme – do stavové proměnné `blok` uložíme text celé řádky (neboli „**$0**“).
- Druhá větev testuje naopak už jen stav. Porovnává se obsah stavové proměnné a řetězce „`groups`“. Akce v této větvi se tak bude provádět pro všechny řádky správného bloku. Ale pozor! To by znamenalo, že se zde zpracuje i úvodní řádka bloku. A to nechceme. Proto jsme v první větvi použili příkaz **next**. Ten vlastně říká: „s touto řádkou jsme už vykonali vše, co jsme vykonat chtěli, přestaň pro ni procházet další větve a pokračuj další řádkou od začátku programu“. Pozor na to, že podobnost s příkazem *next* v editoru **sed** je jen zdánlivá. Oba sice načtou další řádku, ale v editoru se po příkazu pokračuje další instrukcí skriptu, zatímco **awk** se vrací na první větev programu.

Cyklus **for** se v jazyce **awk** se píše prakticky stejně jako v jazyce C, což je dobrá zpráva pro ty, kteří jazyk C znají. Pro ostatní budeme muset trochu vysvětlovat. Zápis příkazu **for** je na první pohled poněkud zvláštní. Za klíčové slovo se píše závorka a do ní tři výrazy oddělené navzájem středníky. První z nich (*inicializace*) se vyhodnotí (či lépe „vykoná“, protože hodnota se nikde nepoužije) právě jednou, před začátkem cyklu (my jsme v něm do proměnné `i` přiřadili jedničku). Druhý výraz (*podmínka*) se vyhodnotí na začátku každého kroku cyklu a v případě, že je pravdivý, cyklus pokračuje (my testujeme, zda proměnná `i` již nedosáhla hranice). Třetí výraz (*iterace*) se vyhodnotí (vykoná) naopak na konci každého kroku cyklu, těsně před tím, než se bude znova vyhodnocovat podmínka (my zvyšujeme hodnotu proměnné `i`).

V iteračním výrazu vidíme další konstrukt převzatý z jazyka C: „**i++**“ (*inkrement*). Je to pouze jiný zápis přičtení jedničky k proměnné. Klidně bychom mohli psát „**i=i+1**“ nebo „**i+=1**“. Podobně existuje i opačný konstrukt, *dekrement* („**i--**“).

Tělo cyklu se skládá z více příkazů (přiřazení a **if**), proto jsme je museli uzavřít do složeného příkazu. Stejně jako v jazyce C se zapisuje pomocí složených závorek.

V každém kroku cyklu si potřebujeme „sáhnout“ na hodnotu *i*-tého pole na řádce. To jde v jazyce **awk** naštěstí velmi snadno. Při psaní odkazu na vstupní pole můžeme za dolar napsat libovolný výraz a jeho aritmetická hodnota se použije jako číslo pole. Proto jsme použili zápis „**$i**“. Chceme-li například zjistit hodnotu posledního pole, napíšeme prostě „**$NF**“ (vzpomeňte si, jak složitě jsme podobnou úlohu řešili v shellu). Nepleťte si to se zápisem „**NF**“, který vyjadřuje naopak počet polí. A klidně můžete být ještě kreativnější. Co třeba znamená zápis „**$(NF-1)**“?

Funkce **index** dostává dva parametry (řetězce) a hledá výskyt druhého v prvním. Pokud ho najde, vrací pozici začátku výskytu (číslováno od jedné). Jinak vrací nulu.

Funkce **substr** dostává dva nebo tři parametry, prvním je řetězec, druhým číslo počáteční pozice požadovaného podřetězce a třetím je délka. Funkce vrací vybraný podřetězec. Pokud není třetí parametr uveden, funkce vrací celý zbytek od zvolené pozice až do konce řetězce. Pozice jsou opět číslovány od jedné.

Řetězce se v jazyce **awk** zapisují do uvozovek (stejně jako v jazyce C). Uvědomte si, že na rozdíl od shellu není jazyk **awk** určen pro textový preprocesor, takže musíme jasně říci, co je text (řetězec) a co jsou jiné atomy jazyka (identifikátory, čísla, operátory...).

Příkaz **if** jsme použili pro ošetření případu, kdy pole neobsahuje závorku a funkce **index** vrátí nulu. Podmínka příkazu **if** (musí být zapsána do závorky) testuje hodnotu proměnné `pozice` obsahující výsledek funkce. V případě, že hodnota je kladná, provede se příkaz zapsaný za závorkou, jinak se pokračuje příkazem zapsaným za slovem **else**.

Oba příkazy v první větvi máme na jedné řádce, a proto jsou odděleny *středníkem*.

Akce u druhé větve se nám naopak nevešla na jednu řádku a museli jsme ji rozepsat na více řádek. Je třeba dát ale pozor na to, že v jazyce **awk** je znak konce řádky významový. Není tedy možné program dělit na řádky kdekoli. My jsme ho rozlámali mezi příkazy. To je nejlogičtější místo, ale občas to nestačí. Jaké máme tedy možnosti?

```
for( i = 1;              ... za středníkem v závorce cyklu for
    i <= NF; i++ ) {     ... za „{“
  if( pozice > 0 &&      ... za „&&“ a „||“
      i > 1 )            ... za závorkou příkazu if (a podobných příkazech)
    print substr( $i,    ... za čárkou mezi parametry
      1, pozice - 1 )    ... za libovolným příkazem
  else                   ... za slovy else a do
    print $i
}
```

V každém případě můžete v případě potřeby pro jistotu ukončit řádku zpětným lomítkem (podobně jako v shellu) a pak nemusíte o vhodném místě pro zalomení řádky přemýšlet.

5.5. A zase skupiny

Zadání:

Vypište seznam čísel skupin, do nichž patříte, bez použití příkazu **`id`**.

Rozbor:

Když jsme úlohu řešili v kapitole 1.20, museli jsme vymyslet, jak spojit operace na dvou různých souborech s různým formátem. Tentýž problém musíme řešit i nyní. Zavoláme příkaz **`awk`** s dalšími dvěma parametry obsahujícími názvy souborů, které chceme zpracovávat, a v programu budeme muset oba zdroje rozlišit. Jednou z možností je testovat jméno právě otevřeného souboru, které je uloženo v proměnné **`FILENAME`**.

S hledáním jména uživatele v řádkách souboru `/etc/passwd` nebude problém (stačí porovnat obsah odpovídajícího pole). Hledání v `/etc/group` bude složitější, protože tam musíme najít jméno jako podřetězec (čtvrtého) pole a navíc se tam musí vyskytovat jako celé slovo. Pro hledání podřetězce můžeme použít například operátor shody řetězce s regulárním výrazem („**`~`**“). Pro omezení kontroly na celé slovo použijeme známou fintu – obložíme obsah pole i testované jméno z obou stran čárkami.

Řešení:

```
awk -F: '
FILENAME == "/etc/passwd" && $1 == "forst" {
    print $4
}
FILENAME == "/etc/group" && "," $4 "," ~ /,forst,/ {
    print $3
}' /etc/passwd /etc/group
```

Poznámky:

Program **`awk`** tentokrát vůbec nebude číst standardní vstup, místo toho postupně přečte oba soubory zadané v parametrech.

Jméno právě zpracovávaného souboru (proměnná **`FILENAME`**) nám zde slouží jako indikátor stavu konečného automatu, podobně jako proměnná `blok` minule.

Řetězec, který potřebujeme zapsat ve druhé podmínce, je poněkud složitější. Potřebujeme ho „poskládat“ z čárek a obsahu čtvrtého pole vstupní řádky. Potřebujeme tedy zapsat *řetězcový výraz* složený ze tří zřetězených podvýrazů. Všimněte si, jaký operátor se používá pro zřetězení řetězců. Nic nevidíte? To je správně! Operátorem zřetězení je totiž *mezera*! V jazyce, který jinak mezery téměř ignoruje... Je to velmi pozoruhodné rozhodnutí duchovních otců programu. Soukromě se domnívám, že by se

asi v tomto případě na klávesnici vhodný znak pro zřetězení řetězců našel a ta trocha psaní navíc by čitelnosti programů v jazyce **awk** jen prospěla.

Operátor „**~**“ znamená shodu řetězce na levé straně s regulárním výrazem napravo. Shodou přitom rozumíme (jako obvykle u regulárních výrazů) fakt, že nějaký podřetězec testovaného řetězce vyhovuje popisu danému regulárním výrazem. Opakem operátoru „**~**“ je operátor „**!~**“.

U obou je třeba dát pozor na to, že ve starších verzích **awk** musí být na pravé straně *literál* (neboli regulární výraz uzavřený mezi lomítka). Norma už povoluje na pravé straně i proměnnou, ale při přenosu na starší systémy můžete narazit.

Norma totiž kodifikovala poněkud novější implementaci nazývanou **nawk**, a tak programy využívající všechny vymoženosti normy nemusejí fungovat v systémech, kde se pod názvem **awk** ukrývá „opravdové“ původní **awk**. Ovšem i v takových starších systémech bývá často novější verze k mání, třeba právě pod názvem **nawk**. Přesto se pokusíme, kde to půjde, špatně přenositelným prvkům vyhnout.

A když už jsme jedno rozšíření zmínili, doplníme informaci o rozšíření druhém, a to funkci **match**. Funguje prakticky stejně jako operátor „**~**“, má dva parametry (řetězec a regulární výraz), vrací číslo pozice, kde se nachází podřetězec, který výrazu vyhovuje, a do proměnné **RLENGTH** nastaví jeho délku.

Podmínky ve vzoru větví byly tak dlouhé, že jsme větve opět museli rozdělit na více řádek. Udělali jsme to podobně jako minule. Ale pozor, kdybychom napsali:

```
FILENAME == "/etc/passwd" && $1 == "forst"
    { print $4 }
```

chování by bylo poměrně neočekávané. Byly by to totiž dvě dvojice vzor-akce, první by měla implicitní akci (výpis celé řádky) a druhá implicitní vzor (vždy splněno).

Zápis podmínky jsme mohli zkrátit i pomocí jiné finty. Pro rozlišení zpracovávaného souboru lze použít v tomto případě úplně jinou metodu, a to test na počet polí na řádce (a tedy hodnoty proměnné **NF**). Řádky souboru `/etc/passwd` obsahují totiž polí sedm, zatímco řádky `/etc/group` mají jen čtyři. Srozumitelnost programu tím však spíš utrpí:

```
awk -F: '
NF == 7 && $1 == "forst" { print $4 }
NF == 4 && "," $4 "," ~ /,forst,/ { print $3 }
' /etc/passwd /etc/group
```

5.6. Skupiny s parametrem

Zadání:

Vypište seznam čísel skupin, do nichž patří uživatel, jehož jméno je zadáno v prvním parametru.

Rozbor:

Úloha z minulé kapitoly, ale s parametrem. První potíž spočívá v tom, jak parametr shellu zpřístupnit příkazům jazyka **awk**. Ukážeme si tři řešení.

Jedna možnost je přenechat celý tento úkol shellu. Shell je koneckonců textový preprocesor a my nechceme nic jiného než doplnit hodnotu parametru do textu programu v jazyce **awk**. Namísto pevného řetězce („`forst`“) jako v minulé kapitole, potřebujeme použít jméno uložené v prvním pozičním parametru shellu („**`$1`**“).

Program pro **awk** jsme si ovšem zvykli psát do apostrofů a mezi nimi shell proměnné a poziční parametry neexpanduje. Pokud vás tedy napadlo uzavřít program do uvozovek místo apostrofů, tak je to dost sebevražedný nápad. Uvědomte si, kolik dolarů, uvozovek a zpětných lomítek se v takovém programu může objevit, a ty všechny by před sebou musely mít zpětné lomítko. Lepší nápad je nechat celý program mezi apostrofy a pouze v těch místech, kde od shellu chceme spolupráci, apostrofy na chvíli „přerušit“ a zapsat tam potřebný poziční parametr shellu:

```
awk -F: '
FILENAME == "/etc/passwd" && $1 == "'$1'" {
    print $4
}
FILENAME == "/etc/group" && "," $4 "," ~ /,'$1',/ {
    print $3
}' /etc/passwd /etc/group
```

Druhou možností je poslat (příkazem **echo**) jméno uživatele rourou na vstup **awk**, v programu si ho uložit do proměnné (`jmeno`) a používat obsah této proměnné. Musíme jen přidat známý parametr „**`-`**“ zastupující standardní vstup mezi zpracovávané soubory.

Hledání v `/etc/passwd` uděláme s proměnnou snadno, zato hledání v `/etc/group` budeme muset vymyslet jinak. Už jsme se zmiňovali, že na pravé straně operátoru „**`~`**“ musí být kvůli přenositelnosti literál (regulární výraz zapsaný mezi lomítka) a nikoliv proměnná. A do literálu mezi lomítka obsah proměnné `jmeno` prostě nedostaneme.

Můžeme si opět pomoci funkcí **index**. Hledání výskytu řetězce v jiném řetězci nám bude úplně stačit, ve skutečnosti totiž regulární výraz vůbec nepotřebujeme!

```
echo $1 | awk -F: '
FILENAME == "-" { jmeno = $1 }
FILENAME == "/etc/passwd" && $1 == jmeno { print $4 }
FILENAME == "/etc/group" &&
    index( "," $4 ",", "," jmeno "," ) { print $3 }
' - /etc/passwd /etc/group
```

Třetí možností je zvláštní typ parametru příkazu **awk** ve tvaru „*proměnná*=*hodnota*“, který „zamícháme“ mezi parametry se jmény zpracovávaných souborů. Když filtr **awk** narazí na takový parametr, provede přiřazení hodnoty do proměnné. Následující vstupní soubor se už bude zpracovávat se správně nastavenou proměnnou:

```
awk -F: '
FILENAME == "/etc/passwd" && $1 == jmeno { print $4 }
FILENAME == "/etc/group" &&
    index( "," $4 ",", "," jmeno "," ) { print $3 }
' jmeno=$1 /etc/passwd /etc/group
```

Je to vlastně jen jednodušší (ale hůře přenositelná) alternativa předchozího řešení.

Nastavení hodnot proměnných přes parametry příkazu má ještě alternativu ve formě přepínače: „**-v** *proměnná*=*hodnota*“. Tento způsob se odlišuje tím, že proměnná má správnou hodnotu už během provádění větve **BEGIN**. Zkušenosti s přenositelností jsou ale ještě o něco horší než u varianty bez přepínače, ačkoliv norma zná obě.

Řešení:

Další varianty si jistě najdete sami.

Poznámky:

Modifikace nějakého programu pro **awk** (ale stejně tak třeba pro **sed**) v závislosti na hodnotách proměnných či pozičních parametrů shellu je velmi obvyklou metodou práce. Skrývá v sobě ale některá úskalí. Například v tomto případě jsme se zcela vydali na milost uživateli a spoléháme se na jeho čestnost (což v dnešním světě internetu je zcela iluzorní a naivní představa) a neomylnost. Představte si, že našemu skriptu jako parametr nedá jméno, ale nějaký „nevhodný“ řetězec. Shell nám pak tento řetězec dosadí jako součást našeho programu pro **awk** a ten bude (v lepším případě) syntakticky nesprávný, zato v horším případě může obsahovat nějaký škodlivý kód a náš program ho vykoná.

A nemusí to být ani zlý úmysl. Pokud by náš skript hledal jiné řetězce než zrovna jména, uživatel by mohl v dobré víře hledat třeba řetězec obsahující mezeru nebo lomítko. Na tom není nic špatného, ale náš program by to stejně úplně „vykolejilo“.

Jako ochrana proti případné mezeře v parametru by nám pomohlo uzavřít **$1** ještě do uvozovek. Jenom je třeba pak dát pozor a nezamotat se do změti uvozovek a apostrofů:

```
' ... $1 == "'"$1"'" { ... '     ... původní text zapsaný ve skriptu
' ... $1 == "'                   ... úvodní část programu (v apostrofech)
              "$1"               ... proměnná nebo parametr shellu (v uvozovkách)
                   '" { ... '    ... pokračování programu (v apostrofech)
             "hodnota "          ... tyto uvozovky omezují řetězec v jazyce awk
' ... $1 == "hodnota " { ... '   ... výsledný text programu v jazyce awk
```

První zápis **$1** je uvnitř programu pro **awk** (je v apostrofech) a je to tedy odkaz na první pole vstupní řádky. Druhé **$1** je naopak uvozovkách, takže na toto místo shell dosadí hodnotu pozičního parametru. Kdybychom zde uvozovky nepoužili, případná mezera by nám program pro **awk** „rozetla“ vedví.

Proti veškerým nežádoucím znakům nás to ale stejně neochrání. Naivita některých programů či internetových protokolů je přímo pověstná. A jistě není rozumné tuto rodinu nadále rozšiřovat. Pokud nevíte, odkud pocházejí data, která zpracováváte, vždy je raději zkontrolujte. V našem případě by třeba stačilo na začátku skriptu ověřit, že parametr obsahuje pouze alfanumerické znaky.

Ve druhém řešení jsme zkombinovali čtení standardního vstupu (zadává se pomocí parametru „**-**“) a dvou souborů (`/etc/passwd` a `/etc/group`). Program tedy jako první řádku dostane jméno uživatele, které jsme tam zapsali příkazem **echo**. V kódu jsme čtení této první řádky detekovali dotazem na jméno souboru („**-**“). Mohli jsme ale použít i test na číslo vstupní řádky:

```
NR == 1 { jmeno = $1 }
```

Proměnná **NR** (*number of record*) má v každém okamžiku hodnotu aktuálního čísla řádky. Počítá přitom od jedničky souhrnně přes všechny vstupní soubory, proto bude mít hodnotu 1 pouze v okamžiku, kdy čte výstup našeho příkazu **echo**.

Při testu výsledku funkce **index** v podmínce jsme využili toho, že při nalezení podřetězce vrací nenulovou hodnotu, a tedy „pravdu“. O něco čitelnější by bylo:

```
    index( "," $4 ",", "," jmeno "," ) > 0 { ...
```

Podmínka v poslední větvi už je poměrně složitá, mohli bychom třeba test s voláním funkce **index** přesunout dovnitř závorky akce a použít příkaz **if**:

```
FILENAME == "/etc/group" {
    if( index( "," $4 ",", "," jmeno "," ) )
        print $3
}
```

5.7. Trumfové eso `awk`

Zadání:

Vypište seznam uživatelů ve skupině s číslem nula.

Rozbor:

V kapitole 2.16 jsme se na této úloze naučili třídění. Ve skutečnosti jsme ho ale nepotřebovali, šlo nám jen o vyloučení duplicit. A tak to nyní zkusíme jinak.

Pro vyloučení duplicit bychom si potřebovali pamatovat, kterého uživatele už jsme v průběhu práce viděli. Na to se v běžných programovacích jazycích používají pole. Vytvořili bychom si pole, kam bychom ukládali jednotlivé uživatele (do prvního prvku pole prvního uživatele, do druhého prvku druhého atd.) a vždy před zařazením nového uživatele bychom nejprve pole prohledali, zda už tam tohoto uživatele nemáme.

My na to rovněž použijeme *pole*. Ale pole v jazyce **`awk`** mají jednu zvláštnost, která z nich dělá nesmírně silný nástroj. Už víme, že zde je vlastně jen jediný datový typ – *řetězec*. Není divu, že také indexy polí jsou řetězce (takovým polím se říká *asociativní*). Na rozdíl od řady jiných jazyků, kde indexy polí mohou být pouze ordinální typy (celá čísla), si zde můžeme pole indexovat přímo jménem uživatele. Tím nám odpadá neustálé nepříjemné sekvenční procházení celého pole pokaždé, když chceme zjistit, zda v něm už daný uživatel je. Prostě se pouze podíváme na prvek pole s indexem rovným jménu uživatele (zapíšeme to „**`seznam[clen]`**“) a podle jeho hodnoty (nula nebo jedna) už rovnou víme výsledek! Je jasné, že samo **`awk`** musí správný prvek pole také nějakým způsobem najít, ale dělá to interně a řádově rychleji, než bychom to dělali my.

Kromě toho se ještě seznámíme s funkcí **`split`**. Pomůže nám procházet seznam členů skupiny v řádkách `/etc/group`. Funkce rozdělí zadaný text podle zadaného oddělovače na jednotlivá slova a uloží je postupně do prvků pole indexovaných tentokrát pro změnu „normálními“ čísly. Celkový počet slov funkce vrací jako svůj výsledek. Prvky pole budeme procházet pomocí cyklu **`for`**.

Celý mechanismus testu a vypisování uživatelů musíme aplikovat ve dvou různých situacích – v každém souboru na jiné pole vstupu s jiným formátem. Podobné případy se v běžných jazycích řeší pomocí funkce s nějakým vhodným parametrem.

I v jazyce **`awk`** můžeme vytvořit funkci `pridej`, která zkontroluje a přidá uživatele.

Drobný problém ovšem opět spočívá v přenositelnosti. Starší verze **`awk`** uživatelsky definované funkce neznají. Proto kromě varianty s funkcí ukážeme i jiné řešení, kdy se společný kód umístí do samostatné větve, která se prochází pro oba soubory. Nejprve se ale připraví takové podmínky, aby větev fungovala pro oba soubory správně.

Řešení (s funkcí):

```
awk -F: '
function pridej( clen ) {
    if( ! seznam[clen] ) {
        seznam[clen] = 1
        print clen
    }
}
FILENAME == "/etc/passwd" && $4 == 0 { pridej( $1 ) }
FILENAME == "/etc/group" && $3 == 0 {
    n = split( $4, lidi, "," )
    for( i = 1; i <= n; i++ )
        pridej( lidi[i] )
}' /etc/passwd /etc/group
```

Řešení (bez funkce):

```
awk -F: '
FILENAME == "/etc/passwd" { gid = $4; probrat = $1 }
FILENAME == "/etc/group"  { gid = $3; probrat = $4 }
! gid {
    n = split( probrat, lidi, "," )
    for( i = 1; i <= n; i++ )
        if( ! seznam[lidi[i]] ) {
            seznam[lidi[i]] = 1
            print lidi[i]
        }
}' /etc/passwd /etc/group
```

Poznámky:

Pole `seznam` využíváme k ukládání jmen těch uživatelů, které jsme už vypsali. Je indexováno jménem uživatele a hodnota 1 znamená, že uživatel již byl zpracován. Pro správnou funkci programu budeme potřebovat, aby pole bylo inicializováno. O to se ale naštěstí nemusíme starat. Proměnné a prvky pole, jimž nebyla doposud přiřazena hodnota, se chovají jako prázdný řetězec, resp. nula (podle kontextu).

Až doposud jsme naše proměnné obvykle inicializovali v akci větve **`BEGIN`**. Nyní je zřejmé, že to není nutné. Přesto doporučuji inicializace přesto (v rozumné míře) ve vašich programech raději explicitně psát. Až budete svoje programy číst po uplynutí nějaké doby, ušetří vám to čas.

Činnost funkce **`split`** jsme snad popsali dostatečně. První parametr udává text, který se bude rozdělovat, druhý parametr název indexované proměnné, jež se použije pro uložení slov, a třetí definuje oddělovač slov (polí).

Často se setkávám s otázkou, zda má jazyk **awk** *vícerozměrná* pole. Ta otázka nedává příliš smysl. Je-li indexem polí řetězec, mohu si libovolněrozměrná pole vytvořit. Mohu si např. nadefinovat proměnnou `oddel`, uložit si do ní čárku (nebo cokoliv jiného) a psát „**`a[1 oddel 2]`**“. Mimochodem, norma to dělá stejně, jenom proměnné říká **SUBSEP** a můžete pak psát i „**`a[1,2]`**“. Jen s přenositelností budete mít problém. Zbytečně.

Uživatelsky definované funkce se zapisují na vnější úrovni programu, kamkoliv mezi ostatní větve. Hlavička funkce začíná klíčovým slovem **function**, následuje název funkce a seznam parametrů v závorkách. Na rozdíl od syntakticky poněkud podobných funkcí v shellu mohou funkce v jazyce **awk** vracet hodnotu (příkazem **return**).

Shodné s shellovými funkcemi je u **awk** zacházení s proměnnými. Veškeré proměnné jsou globální, funkce tedy může nastavovat proměnné, které lze po návratu využívat.

Naopak parametry jsou lokální – toho lze s výhodou využít k vytvoření lokálních proměnných: přidáme si ke skutečným parametrům „falešné“ parametry, jejich vstupní hodnoty není třeba při volání zadávat, ve funkci je budeme ignorovat a proměnné můžeme libovolně používat (jako např. proměnná `i` v následujícím rámečku).

Pozor na to, že „obyčejné“ (*skalární*) parametry jsou funkci předávány *hodnotou* (hodnota skutečného parametru se zkopíruje do lokálního parametru funkce), ovšem pole jsou předávána *referencí* (funkce ve skutečnosti pracuje s tím polem, které dostala jako parametr). Následující ukázka tedy sice správně vynuluje prvky `pole`, ale proměnná `velikost` zůstane nezměněna.

```
function vymaz( pp, pn, i ) {
    for( i = 1; i <= pn; i++ )
        pp[i] = 0
    pn = 0
}
{ … pole[1]=1; velikost=1; vymaz( pole, velikost ) … }
```

Funkci jsme ještě přidali falešný parametr `i`, čímž jsme vlastně vytvořili lokální proměnnou. Pokud by existovala globální proměnná `i`, volání funkce by ji nepoškodilo. Kdybychom falešný parametr nepoužili, měnila by funkce hodnotu globální proměnné `i`.

Ve druhém řešení (bez funkce) jsme společný kód zapsali do poslední větve a bude třeba zařídit, aby tato větev pracovala se správnými daty. Nejprve se tedy (pro každý soubor jinak) nastaví hodnoty proměnných `gid` (číslo skupiny) a `probrat` (obsah toho pole, v němž se nacházejí jména uživatelů) a zbytek kódu už může být společný. Poslední větev se provádí pouze pro nulové GID, proměnná `probrat` se rozdělí na jednotlivá jména uživatelů a ta se zpracují. V případě souboru `/etc/passwd` bude pole obsahovat vždy jen jedno jméno a volání funkce **split** je vlastně zbytečné, ale rychlosti to příliš neublíží a čitelnosti to spíše prospěje.

5.8. Ukaž, co máš

Zadání:

Vypište seznam duplicitních UID v systému.

Rozbor:

Použijeme pole indexované číslem UID a do něj budeme ukládat počet uživatelů s tímto UID. Na konci vypíšeme ta UID, která byla zastoupena více než jednou.

Na to, abychom prošli všechny prvky asociativního pole, budeme potřebovat nový konstrukt, cyklus **for-in**. Zadá se v něm řídicí proměnná a název pole, cyklus projde postupně všechny prvky pole a pro každý jednou provede tělo cyklu, s hodnotou indexu prvku nastavenou do řídicí proměnné.

Řešení:

```
awk -F: '{ pocet[$3]++ }
END {
    for( uid in pocet )
        if( pocet[uid] > 1 )
            print uid
}' /etc/passwd
```

Poznámky:

Malá nepříjemnost spočívá v tom, že pořadí procházení indexů není specifikováno. Prvky pole si totiž **awk** neukládá do paměti sekvenčně a cyklus prochází prvky v takovém pořadí, v jakém je má uložené. Nám to teď ale nevadí.

Horší zprávou je to, že index v poli vznikne nejen tak, že danému prvku přiřadíme hodnotu, ale i jen tím, že si na prvek „sáhneme“ (např. testováním v podmínce). Je třeba proto vždy v těle cyklu ověřit, že prvek pole opravdu má neprázdnou hodnotu.

Všimněte si, že vlastně veškerá činnost programu je soustředěna do závěrečné větve **END**. To je častý rys programů v jazyce **awk**. Nejprve se „posbírají“ potřebné údaje a posléze (třeba až na konci programu) se s nimi pracuje. Jenom je třeba myslet na to, že i paměť aplikace **awk** má určitá omezení...

Pokud bychom se chtěli vyhnout použití větve **END**, mohli bychom zvolit jiný přístup. Jakmile počet výskytů pro nějaké UID dosáhne přesně čísla 2, vypíšeme ho:

```
{ if( ++pocet[$3] == 2 ) print $3 }
```

Pro ty, kteří neznají jazyk C, ještě malé vysvětlení. Operátory „**++**“ a „**--**“ mají zvláštní vlastnost. Pokud je zapíšeme *za* proměnnou, výpočet se provede se starou hodnotou proměnné a teprve pak se hodnota změní. Pokud je zapíšeme *před* proměnnou, provede se nejprve změna hodnoty a tato nová hodnota se použije ve výpočtu.

5.9. Interakce

Zadání:

Napište program, který opakovaně vyzývá uživatele k zadání nového uživatelské jména, dokud není zadáno takové jméno, které dosud není v systému použito. Jakmile je jméno v pořádku, vypíše ho a současně navrhne UID – o jedničku vyšší, než nejvyšší doposud použité UID (nižší než 65000, vyšší UID bývají vyhrazena pro speciální účely). Požadovaný tvar výstupu:

```
jmeno=jméno
uid=UID
```

Rozbor:

Program **awk** čte standardní vstup, proto s načítáním návrhů jmen nebude problém.

Naopak u výstupních příkazů budeme muset odlišit texty určené pro uživatele (výzvy k zadání jména) a text závěrečného výstupu. Jednou z možností je směřovat výstup pro uživatele přímo na terminál. Pro tento účel lze použít známý pseudosoubor `/dev/tty`.

Na začátku práce ještě přečteme soubor `/etc/passwd`. Jednak proto, abychom si uložili existující jména do pole a nemuseli neustále soubor procházet při kontrole duplicity, a rovněž proto, abychom zjistili nejvyšší UID a mohli dát návrh na nové UID.

Pro jednodušší tvorbu výstupních textů použijeme nový příkaz **printf**. Poznali jsme stejnojmenný příkaz už v shellu, stačí tedy jen upravit styl volání a přidat nové direktivy.

Řešení:

```
awk -F: '
BEGIN {
    printf "Zadej jmeno uzivatele: " > "/dev/tty"
}
FILENAME == "/etc/passwd" {
    existuje[$1] = 1
    if( $3 < 65000 && $3 > max )
        max = $3
    next
}
existuje[$1] {
    printf "Jmeno %s existuje, zvol jine: ", $1 > "/dev/tty"
    next
}
{
    printf "jmeno=%s\nuid=%d\n", $1, max + 1
    exit
}' /etc/passwd -
```

Poznámky:

Stejně jako v shellu, i zde příkaz **printf** sám neodřádkuje, proto jsme znak konce řádky (tam, kde jsme odřádkovat chtěli) museli do řetězce vložit pomocí „**\n**“. Podobně lze do řetězců v jazyce **awk** psát i další escape-sekvence (např. „**\t**“ jako tabulátor).

Poznali jsme novou direktivu „**%d**“ pro výpis dekadické hodnoty čísla.

Operátor „**>**“ za příkazem **printf** (a stejný lze zapsat i za příkaz **print**) znamená, že výstup bude přesměrován do *souboru*, jehož jméno je uvedeno za operátorem. Ačkoliv je to po krátké úvaze jistě zřejmé, připomínám, že jméno souboru je řetězec, takže ho musíme zapsat do uvozovek. Namísto toho lze uvést i proměnnou, do níž si jméno souboru nejprve dle potřeby přichystáme.

Kromě tohoto existuje ještě operátor „**>>**“, který se liší tím, že při prvním zápisu soubor nevymaže, ale začne zapisovat na jeho konec.

Až doposud naše programy končily svoji práci, jakmile vyčerpaly vstupní data. Nyní potřebujeme ukončit činnost programu na základě dosažení určitého stavu (je vybráno správné jméno). K tomu slouží příkaz **exit**. Ukončí procházení vstupu a začne provádět větev **END** (pokud taková v programu je a pokud příkaz nebyl použit uvnitř této větve).

Oddělovač polí zůstává po celou dobu práce programu nastaven na dvojtečku (jako důsledek čtení souboru `/etc/passwd`). V našem případě to nevadí, jméno uživatele určitě dvojtečku obsahovat nesmí. Pokud by nám to vadilo, mohli bychom oddělovač po přečtení souboru zase změnit. Jeho hodnota je totiž uložena ve speciální proměnné **FS** (*field separator*) a přepínač „**-F**“ vlastně jen umožňuje tuto hodnotu předdefinovat. Začátek našeho programu by klidně mohl vypadat takto:

```
awk 'BEGIN { FS = ":"
    printf "Zadej jmeno uzivatele: " > "/dev/tty"
} ...
```

Pokud hodnotu **FS** změníme v průběhu práce, efekt se projeví až na následující řádce. Z toho vyplývá, že pokud chceme mít oddělovač nastavený správně už pro první řádku, je třeba provést přiřazení do proměnné **FS** ve větvi **BEGIN**. Jenže jak poznat správný okamžik, kdy přepnout hodnotu oddělovače zpátky? Pokud bychom proměnnou **FS** změnili až na základě podmínky, že **FILENAME** je rovno „**-**“, bude už pozdě – první řádka od uživatele bude načtená a rozlámaná podle starého oddělovače.

Pokud bychom přistoupili na mírné riziko s přenositelností, můžeme použít možnost nastavení proměnné **FS** pomocí parametrů:

```
awk '...' FS=: /etc/passwd FS=' ' -
```

Pokud bychom toto riziko podstupovat nechtěli, můžeme si vytvořit zvláštní soubor `zarazka` obsahující jednu jakoukoliv řádku a nastavení proměnné **FS** (a mimochodem také vypsání první výzvy k zadání jména) ošetřit v samostatné větvi:

```
awk -F: '
FILENAME == "/etc/passwd" { … }
FILENAME == "zarazka" {
    FS = " "
    printf "Zadej jmeno uzivatele: " > "/dev/tty"
    next
}
FILENAME == "-" { … }' /etc/passwd zarazka -
```

Z hlediska programu je úplně jedno, zda pro nastavení oddělovače použijete přepínač „**-F**“ nebo proměnnou **FS**. Osobně používám strategii, že u krátkých jednořádkových programů volím přepínač (kvůli kratšímu zápisu), ale u delších programů nastavuji proměnnou. Pokud je totiž celý program závislý na tom, jaký oddělovač polí se použije, je dobré, když nastavení oddělovače je součástí kódu programu a nikoliv způsobu jeho zavolání na příkazové řádce shellu.

Zmiňme ještě jednu variantu výstupu v poslední větvi programu. Pokud totiž příkazu **print** dáme víc parametrů, vypíše je oddělené obsahem proměnné **OFS** (*output field separator*). Implicitně obsahuje mezeru, ale nám by se hodilo použít rovnítko:

```
    OFS = "="
    print "jmeno", $1
    print "uid", max + 1
```

Pozor na záměnu proměnných shellu a **awk**. Tyto dvě množiny proměnných jsou zcela nezávislé. Zvláště u jmen **FS** a **OFS** (**awk**) versus **IFS** (shell) to hodně svádí.

Pokud vám vrtá hlavou, proč jsme zvolili takový divný tvar výstupu, povšimněte si, že jsou to vlastně přiřazovací příkazy. Když výstup tohoto tvaru vezmeme a necháme ho shell vykonat, dostaneme velmi elegantně z programu v jazyce **awk** téměř jakoukoliv sadu výsledků do proměnných shellu. Vypadalo by to asi takto:

```
eval `awk -F: '…' /etc/passwd -`
```

Jenom je třeba v takovém případě pečlivě kontrolovat vstup od uživatele, který se následně stane součástí prováděných příkazů shellu (už jsme to zdůrazňovali v kapitole 4.28). Pokud bychom kontrolu vstupu nechtěli nebo nemohli dělat, bylo by lepší nechat program prostě zapsat svoje výsledky v nějakém vhodném pevném tvaru do pomocného souboru a ten pak v shellu jednoduše přečíst.

5.10. Editace

Zadání:

Napište program, který z výstupu příkazu „**ls -ln** *jméno_adresáře*“ vytvoří výstup ve tvaru „**ls -l** *jméno_adresáře*“ (nahradí čísla uživatelů a skupin jmény).

Rozbor:

Základní ideu řešení jsme už jednou rozmysleli (v kap. 4.35). Program pro **awk** bude pracovat podobně. Opět si vytvoříme převodní tabulky čísel na jména, tentokrát opravdu do indexovaných proměnných (polí).

Zbývá rozmyslet, jak upravit vstupní řádky na požadovaný výstup. Minule jsme to řešili pomocí editoru. Takovou možnost v jazyce (standardního) **awk** nemáme. Ovšem editace, kterou potřebujeme v této úloze, je vcelku triviální. Stačilo by nám nahradit původní vstupní pole novým obsahem. A to opravdu jde.

Řešení:

```
awk '
BEGIN { FS = ":" }
FILENAME == "/etc/passwd" { uid[$3] = $1; next }
FILENAME == "/etc/group"  { gid[$3] = $1; next }
/^total / { FS = " "; print; next }
{
    $3 = uid[$3]
    $4 = gid[$4]
    print
}' /etc/passwd /etc/group -
```

Poznámky:

Řádka „`total ...`“ nám bude sloužit jako impuls pro změnu hodnoty proměnné **FS** zpět na původní hodnotu, aby se nám správně četla pole výstupu příkazu **ls**. Problém jsme diskutovali už v minulé kapitole. Proměnnou je nutno nastavit ještě před načtením řádky, která má být zpracována podle nového oddělovače polí. Potřebovali bychom poznat poslední řádku čtených konfiguračních souborů a to neumíme. Naštěstí vůbec nevadí, když bude řádka „`total ...`“ rozdělena na slova podle špatného oddělovače. Stejně ji jen opíšeme na výstup. A pro další řádky už bude oddělovač nastaven správně.

Na tomto místě je zapotřebí vysvětlit sémantiku hodnoty **FS**. Novější implementace filtru **awk** zavedly zadávání oddělovače jako *regulárního výrazu*, ale starší zvládají jako oddělovač polí pouze *jeden znak*. Norma proto říká, že jednoznakový oddělovač se chová „postaru“ (jako prostý oddělovač), zatímco víceznakový je regulární výraz. Pokud je v proměnné **FS** *prázdný řetězec*, chování není definováno.

Význam jednoznakového oddělovače se navíc liší, podle toho, zda tímto znakem je nebo není mezera. V případě mezery (což je implicitní hodnota) se totiž oddělovačem stává jakákoliv posloupnost bílých znaků, zatímco pro jiné znaky je každý výskyt znaku oddělovačem. Staré verze filtru **awk** tímto chováním napodobovaly práci shellu. Nové implementace i norma toto pravidlo naštěstí dodržují.

Regulární výraz je velmi příjemným prostředkem, jak oddělovač polí definovat. Třeba právě volba, zda každý výskyt oddělovače znamená nové pole nebo zda se mají vícenásobné výskyty slučovat, je velmi jednoduchá a jednoznačná („**:**“ vs. „**:+**“).

Pokud vám možnost měnit pole vstupní řádky připadá poněkud „podivná“, pak u mě najdete podporu. Já s ní mám taktéž drobný „morální“ problém. Vstup mi prostě připadá jako něco, co by mělo zůstat vstupem. Ale na druhé straně je nutno uznat, že tato operace je v mnoha případech (jako je třeba tento) docela užitečná.

Pouze se musíte připravit na to, že v okamžiku provedení modifikace jste definitivně přišli o text původní řádky. Ta se totiž znovu „poskládá“ z jednotlivých polí oddělených jednou mezerou (resp. obsahem proměnné **OFS**). Jestliže nám výsledný tvar výstupní řádky v minulém řešení připadal ošklivý a chystali jsme se na lepší formátování, pak toto řešení nám řádky rozhází ještě podstatně více.

Poznamenejme, že při přiřazení nové hodnoty do **$0** dojde naopak k „přepočítání“ všech polí (**$1**, **$2**, ...). Tímto způsobem se dá obejít problém změny oddělovače polí až poté, co je načtena řádka vstupu:

```
{ FS = "nový oddělovač"; $0 = $0 }
```

Absence editace jako takové je občas poněkud nepříjemná. Programátor pak stojí před rozhodnutím, zda jeho aplikace bude více editovat nebo bude více potřebovat lepší programátorské nástroje (např. výpočetní sílu), jež jsou k dispozici v jazyce **awk**, a podle toho volí **awk** nebo **sed**. Součástí tohoto rozhodování by ale vždycky měla být pořádná analýza problému a případná dekompozice úlohy. Může se totiž ukázat, že díky vhodnému rozdělení úlohy na podúlohy se editační a výpočetní nároky šikovně rozdělí a pak půjde opravdu editaci dělat editorem a výpočty v jazyce **awk**.

A když už opravdu nevidíte jinou cestu, můžete zkusit sáhnout po dalších funkcích, které jsou sice popsány v normě, ale bohužel chybějí ve starších implementacích. Jsou to funkce **sub** (odpovídá zhruba příkazu *substitute* v editoru **sed**) a **gsub** (příkaz *substitute* s parametrem „**g**“):

```
sub( " *" $3 " *" $4, " " uid[$3] " " gid[$4] )
```

Toto volání funkce nahradí ve vstupním záznamu výskyt prvního řetězce (regulárního výrazu) řetězcem druhým.

5.11. Formátování

Zadání:

Napište program, který dělá to, co minule, ale tentokrát i s formátováním.

Rozbor:

Použijeme stejnou základní ideu jako minule. Změníme jenom styl výpisu.

Výstup příkazu **`ls`** je zarovnán do úhledných sloupečků a toho nebude jednoduché dosáhnout. Částečně můžeme využít toho, jak řádky naformátoval příkaz **`ls`** sám (u těch sloupečků, které nebudeme měnit), ale stejně nám nezbude než celý výstup přečíst a zjistit maximální šířku některých sloupečků. Výstup si budeme průběžně ukládat do několika polí:

- `z` Obsahuje začátek řádky (práva a počet linků). Jeho délku stačí určit na první řádce (protože všechny řádky mají stejný formát) a určíme ji tak, že najdeme pozici výskytu pole **`$2`** na řádce (pomocí funkce **`index`**) a přidáme délku samotného pole (tu zjistíme funkcí **`length`**). Na každé řádce už pak jen vybereme potřebný podřetězec pomocí funkce **`substr`**.
- `u` Obsahuje jméno uživatele – vlastníka souboru. Zjistíme ho převedením čísla (**`$3`**) na jméno pomocí připravené převodní tabulky (v poli `uid`). U tohoto sloupečku musíme počítat maximální šířku, protože se jedná o nové údaje, které v původním výstupu vůbec nejsou.
- `g` Obsahuje jméno skupiny. Bude se zpracovávat podobně jako předchozí pole.
- `v` Obsahuje velikost souboru. Vzhledem k budoucímu zarovnávání doprava potřebujeme i u tohoto sloupečku celý výstup projít a zjistit potřebnou šířku.
- `k` Obsahuje konec řádky (od data až po jméno). Ten už zase budeme jen opisovat a stačí nám tedy znát pozici jeho začátku. Tu opět zjistíme na první řádce vyhledáním výskytu pole **`$6`**.

Poté, co získáme veškerá data včetně maximálních šířek sloupečků, budeme moci spustit vypisování uložených dat. Nejprve si ovšem připravíme formátovací řetězec pro příkaz **`printf`** do proměnné `form`. Formátovací direktivy tentokrát budou muset obsahovat další modifikátory, abychom dosáhli správného zarovnání. Zapisují se vždy mezi znak procenta a písmeno formátovací direktivy. Především musíme zadat rozsah, na *kolik znaků* chceme hodnotu zapsat (v případě potřeby se hodnota doplní mezerami). A v případě jmen musíme navíc použít rovněž příznak „**`-`**“, který znamená, že doplnění mezerami se má dělat *zprava* a ne zleva. Představme si, že by všechny nové sloupečky měly mít šířku například osm znaků, pak by výsledný tvar formátovacího řetězce byl:

```
"%s %-8s %-8s %8d %s\n"
```

Řešení:

```
awk '
BEGIN { FS = ":" }
FILENAME == "/etc/passwd" { uid[$3] = $1; next }
FILENAME == "/etc/group"  { gid[$3] = $1; next }
/^total / { FS = " "; print; next }
sirka_z == 0 {
    sirka_z = index( $0, $2 ) - 1 + length( $2 )
    pozice_k = index( $0, $6 )
}
{
    pocet++
    z[pocet] = substr( $0, 1, sirka_z )
    k[pocet] = substr( $0, pozice_k )
    u[pocet] = uid[$3]
    g[pocet] = gid[$4]
    v[pocet] = $5
    if( max_u < length( uid[$3] ) )
        max_u = length( uid[$3] )
    if( max_g < length( gid[$4] ) )
        max_g = length( gid[$4] )
    if( max_v < length( $5 ) )
        max_v = length( $5 )
}
END {
    form = "%s %-" max_u "s %-" max_g "s %" max_v "d %s\n"
    for( i = 1; i <= pocet; i++ )
        printf form, z[i], u[i], g[i], v[i], k[i]
}' /etc/passwd /etc/group -
```

Poznámky:

Princip formátování textu, který zavedli autoři jazyka C ve funkci **`printf`** a který převzali i pro **`awk`**, považuji osobně za jeden z nejpovedenějších, jaké jsem poznal. Jednoduché varianty formátování se dělají jednoduše, kompaktní forma umožňuje velmi snadnou internacionalizaci programů... Nezbytná je jen vnitřní disciplína programátora, aby formátovací direktivy odpovídaly typům parametrů. To je asi kámen úrazu a důvod, proč se spoustě lidí **`printf`** nelíbí. Naštěstí v jazyce **`awk`** se toho nemusíme obávat. Vzhledem k tomu, že **`awk`** je interpret, dokáže nás sám kontrolovat.

Užitečné jsou jistě i další atributy formátovacích direktiv. Můžete je najít buď v manuálových stránkách anebo v našem Stručném přehledu (viz str. 348).

5.12. Když řádka není řádka

Zadání:

Napište program, který vypisuje z textu ve formátu HTML na standardním vstupu všechny bloky uzavřené mezi párové značky „`<a …>`“ a „`</a>`“.

Už jsme si říkali, že HTML je značkovací jazyk, v němž se zapisují texty webových stránek. Nyní se budeme zabývat značkou „`a`“. Pomocí ní se zapisuje *odkaz* na jinou webovou stránku. Její běžný formát je:

```
<a href="adresa_odkazu">text_odkazu</a>
```

Problémem je to, že jazyk HTML má velmi volný formát, nehledí na velikost písmen u klíčových slov ve značkách a dokonce i mezery či znaky konce řádek se mohou vyskytovat téměř kdekoliv. Je třeba tedy řešit nejen to, že více odkazů může být na jediné řádce, ale naopak i to, že odkaz může být zapsán třeba jako:

```
<A
Href
= …
```

Rozbor:

Čtení po řádkách tak, jak jsme ho zatím poznali, by bylo v tomto případě příliš komplikované. Proto využijeme toho, že filtr **`awk`** ve skutečnosti nečte řádky, ale *záznamy* a dovoluje nám nastavit *oddělovač záznamů* pomocí proměnné **`RS`** (*record separator*). Implicitně je jím znak konce řádky, takže implicitně se čte vstup opravdu po řádkách, jak jsme až dosud tvrdili. Nyní si ale oddělovač změníme na znak „**`<`**“. Díky tomu v každém kroku dostaneme jednu značku (a text, který za ní následuje) jako celistvý záznam, bez ohledu na to, na kolika řádkách byl text původně zapsán nebo kolik značek bylo na jedné řádce.

Pokud přitom navíc ponecháme nastaven implicitní oddělovač polí, budeme v **`$1`** mít v podstatě jen „název“ značky a s tím se nám bude poměrně dobře pracovat. U počáteční značky tak bude připadat v úvahu pouze samotné písmeno „`a`“ (resp. „`A`“). U koncové značky to bude trochu komplikovanější – bude tam dvojice znaků „`/a`“ (resp. „`/A`“) následovaná buď znakem „`>`“, nebo koncem pole (to v případě, že v původním textu následovala mezera nebo konec řádky). V rozšířených regulárních výrazech se ale taková alternativa dá popsat snadno.

Pak už bude stačit vypsat text mezi počáteční a koncovou značkou „`a`“. K tomu lze využít varianty zadání *rozsahu* vzorů pro určitou větev. Rozsah vzorů se chová podobně jako rozsah adres v editoru **`sed`** – jakmile byl jednou splněn první vzor, provádí se patřičná akce, dokud nedojde ke splnění druhého vzoru. Odlišit budeme muset pouze poslední záznam, u nějž budeme chtít výpis ukončit hned za závěrečným větším.

Řešení:

```
awk '
BEGIN { RS = "<" }
$1 ~ /^[aA]$/, $1 ~ /^\/[aA](>|$)/ {
    if( $1 ~ /^\/[aA](>|$)/ )
        printf "<%s\n", substr( $0, 1, index( $0, ">" ) )
    else
        printf "<%s", $0
}'
```

Poznámky:

Regulární výraz pro test koncové značky musí pokrýt obě možnosti, „`/a>`…“ i „`/a`“. Přitom test na shodu prvních dvou znaků nestačí, protože by mohl vyhovovat i v případě jiné značky začínající písmenem „`a`“. Oproti řešením, která jsme museli použít dříve, zde máme možnost zadat *výběr* z několika podvýrazů, v nichž lze použít i metaznak dolar. Výraz „**`(>|$)`**“ tedy znamená „zde je větší tko anebo konec řetězce“.

Pro každý záznam v rozsahu mezi počáteční a koncovou značkou vypíšeme jeho celý obsah. Musíme ale přidat úvodní znak „<“, který tvoří oddělovač záznamů, a není proto součástí **`$0`**. Výjimkou je poslední záznam intervalu, z nějž vypíšeme jen část od začátku po znak větší tka (konec značky) a pak odřádkujeme.

Rozsahem vzorů bychom pochopitelně nevyřešili případ, kdyby značky byly „vnořené“ (uvnitř páru „`<a …>`“ a „`</a>`“ byl další pár). Značky „`a`“ naštěstí nemohou být vnořené, ale třeba vyhledávání párů „`<ol>`“ a „`</ol>`“ (jež zanořené být mohou) by se muselo dělat jinak, například takto:

```
awk '
BEGIN { RS = "<"; hloubka = 0 }
$1 ~ /^[oO][lL](>|$)/ { hloubka++ }
$1 ~ /^\/[oO][lL](>|$)/ {
    if( ! --hloubka )
        printf "<%s\n", substr( $0, 1, index( $0, ">" ) )
}
hloubka { printf "<%s", $0 }'
```

Zavedli jsme „hloubkoměr“, abychom měli přehled o hloubce zanoření značek, a při nenulové hloubce vypisujeme text záznamu. Výjimkou je případ, kdy čteme zavírací značku a zároveň jsme dosáhli nulové hloubky – pak se jedná o poslední záznam výpisu.

Vedle proměnné **`RS`** existuje proměnná **`ORS`** (*output record separator*), jejíž hodnotu používá příkaz **`print`** pro „odřádkování“. Zatím jsme říkali, že příkaz na konci práce odřádkuje, ale to platí pouze při implicitním nastavení proměnné **`ORS`**.

5.13. Bloky řádek

Zadání:

Napište program, který pro dané číslo RFC vypíše řetěz jeho předchůdců.

Problematiku RFC jsme už studovali v kapitole 3.16. Pokud je dokument nahrazen nějakým novějším RFC, bude jeho záznam v souboru „`rfc-index.txt`“ obsahovat položku „`(Obsoleted by RFC`*nnnn*`)`“. O starším RFC pak můžeme hovořit jako o *předchůdci* novějšího dokumentu.

Pro zadané RFC je tedy nutno prověřit, zda existuje jeho předchůdce, a pokud ano, je třeba hledat zase jeho předchůdce. Tento postup je třeba opakovat tak dlouho, dokud nebude nalezen dokument, který nemá předchůdce.

Rozbor:

Pokud bychom si nekladli žádné nároky na rychlost, mohli bychom vytvořit (nejlépe v shellu) cyklus, který v indexovém souboru najde předchůdce zadaného RFC, zjistí jeho číslo a postup opakuje. Nepříjemné je, že soubor budeme procházet mnohokrát.

Naproti tomu v jazyce **`awk`** by bylo možné číst indexový soubor pouze jednou, vztahy „byl nahrazen“ si postupně ukládat do pole a na závěr pouze využít informací uložených v tomto poli k vypsání výsledku.

Malou komplikací je nutnost zpracování záznamu o jednom RFC „najednou“, přestože se nachází na více řádkách. V tomto bodě nabízí jazyk **`awk`** jednu zajímavou možnost, a tou je zvláštní nastavení oddělovače záznamů. Pokud je nastaven na *prázdný řetězec*, jako záznamy se chápou *bloky* řádek ukončené *prázdnou* řádkou.

Jak bude tedy náš algoritmus vypadat?

Pokud záznam obsahuje slovo „`Obsoleted`“, probereme jednotlivá pole záznamu a najdeme ho. Přesně o dvě pole dál se bude nacházet číslo následníka. Díky vhodné volbě oddělovače záznamů nám nebude vadit ani případné rozlámání do řádek. Z řetězce „`RFC`*nnnn*“ dostaneme hledané číslo pomocí funkce **`substr`** jako čtyřznakový řetězec začínající na pozici číslo 4. Číslo následníka použijeme jako index do pole `predchudce` a jako hodnotu tam uložíme číslo RFC právě zpracovávaného dokumentu.

Tímto způsobem bychom mohli probrat celý indexový soubor, ale nám bude stačit, když se zastavíme v okamžiku, jakmile číslo nalezeného následníka bude rovno číslu RFC, které bylo původně zadané jako parametr.

V takovém případě projdeme řetěz předchůdců uložený v poli a postupně je budeme vypisovat. Použijeme k tomu ale jiný typ cyklu, a to **`while`**.

Řešení:

```
awk 'BEGIN { RS = ""; rfc = "'$1'" }
/Obsoleted/ {
    for( i = 1; i <= NF && $i != "(Obsoleted"; i++ );
    if( i > NF )
        next
    naslednik = substr( $(i + 2), 4, 4 )
    predchudce[naslednik] = $1
    if( naslednik == rfc ) {
        while( predchudce[rfc] ) {
            rfc = predchudce[rfc]
            print rfc
        }
        exit
    }
}' rfc-index.txt
```

Poznámky:

Práce s bloky řádek není úplně denním chlebem, nicméně je dobré si tuto schopnost filtru **awk** zapamatovat pro případ potřeby (například některé databáze mívají blokový formát s řádkami tvaru „*atribut*=*hodnota*“). I my bychom se v našem příkladu bez této vlastnosti filtru obešli, ale za cenu zbytečně komplikovaného řešení.

Dosazení pozičního parametru ve větvi **BEGIN** jsme zapsali bez uvozovek. Opět to předpokládá kontrolu, že jsme od uživatele dostali číslo, ale jinak je to v pořádku – je-li parametrem číslo, nepotřebujeme uvozovky ani kvůli shellu, ani kvůli jazyku **awk**.

Cyklus na hledání slova „`(Obsoleted`“ má prázdné tělo. Veškerou činnost pokrývá podmínka „dokud nejsi na konci seznamu polí a nenašel jsi hledané slovo“ a iterační výraz „**i++**“. Po skončení cyklu jen pro jistotu zkontrolujeme, že cyklus opravdu skončil díky úspěšnému hledání a nikoliv tak, že probral pole a slovo nenašel.

Seznámili jsme se s novým příkazem, cyklem **while**. Za klíčovým slovem se píše závorka s podmínkou a tělo cyklu. Dokud je podmínka splněna, cyklus pokračuje. Naše podmínka je poněkud „ošizená“. O něco čitelnější by bylo asi napsat

```
        while( predchudce[rfc] > 0 ) { ...
```

Je to obvyklý nešvar programátorů v jazyce C, že si takto „šetří“ práci na úkor čitelnosti. I když v tomto případě nehrozí nějaké nepochopení, protože náš původní zápis můžeme číst jako „dokud předchůdce RFC s číslem `rfc` (existuje), dělej...“. Ale než složitě přemýšlet, zda úspora času při psaní programu nepřinese nesrovnatelně větší ztráty času při každém dalším čtení kódu, je asi lepší dát raději přednost jeho srozumitelnosti...

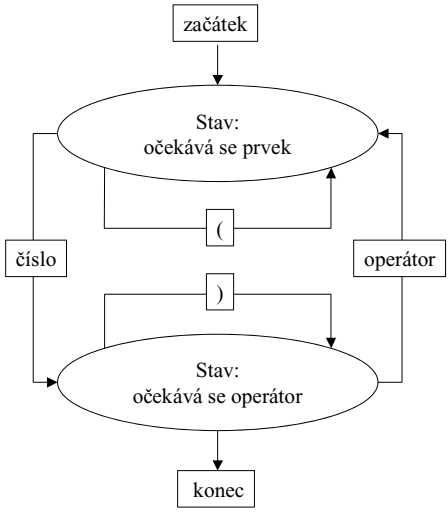
5.14. Parser

Zadání:

Napište program, který kontroluje formální správnost aritmetického výrazu zadaného v parametrech (podobně, jako se zadávají třeba příkazu **`expr`**).

Rozbor:

Toto je klasická úloha, která se nejsnáze řeší pomocí analyzátoru ve formě konečného automatu. Naštěstí to je přesně styl logiky programování v jazyce **`awk`**, takže převedení do kódu bude velmi jednoduché. Pojďme ale nejdříve vzít do ruky tužku a stavy automatu a přechodový diagram si namalujme.



Bude vám to nejspíš připomínat vývojové diagramy, které vás možná někdo někdy během vzdělávacího procesu nutil tvořit, i když vám to možná bylo proti srsti. Je jasné, že malovat si „vývojáky" pro každou sebemenší úlohu je nesmysl. Nicméně existují úlohy, u kterých už to nesmysl není, a jedna počmáraná „A-čtyřka" ušetří hodiny marného ťukání do klávesnice. A možná jedním z největších umění při vývoji programů je právě poznat tu mez, kdy je třeba zahodit klávesnici a vzít do ruky tužku.

Nám bude nyní stačit vyjádřit stavy pomocí stavové proměnné a zapsat jednotlivé přechody jako větve programu. Stavy si pro lepší čitelnost programu označíme pomocí „konstant" `ok` („očekává se operátor, resp. konec výrazu") a `ko` („očekává se operand, výraz nemůže skončit"). Oproti grafu přidáme ještě stav „chyba" (`er`), abychom snáze rozpoznali korektní a nekorektní přechod do větve **`END`**. Budeme navíc kontrolovat také „hloubku" zanoření výrazů v závorkách.

Řešení:

```
printf "$@" | awk 'BEGIN { RS = " "
    ok = 1; ko = 0; er = -1; stav = ko; hloubka = 0
}
stav == ko && $1 == "("   { hloubka++; next }
stav == ko && /^-?[0-9]+$/  { stav = ok; next }
stav == ok && /^[-+*\/]$/ { stav = ko; next }
stav == ok && $1 == ")"   { hloubka--; next }
/./ { print "Neocekavany atom: " $1; stav = er; exit }
END {
    if( stav == er )
        exit 1
    if( stav == ko ) {
        print "Vyraz neuzavren"
        exit 2
    }
    if( hloubka ) {
        print "Nesparovane zavorky"
        exit 3
    }
}'
```

Poznámky:

Mravní ponaučení: zdánlivě velmi obtížnou úlohu jsme díky vývojovému diagramu zvládli prakticky na čtyřech řádkách obsahujících zcela jednoduché příkazy.

Oproti minulým řešením jsme k jednotlivým příkazům **`exit`** přidali ještě hodnotu návratového kódu, kterou poté program **`awk`** předá volajícímu shellu.

Pozor na způsob, jakým předáme parametry na vstup programu **`awk`**. Kdybychom použili volání ve tvaru „**`echo $*`**“ a mezi parametry bylo násobení, hvězdička by se nám v tomto okamžiku rozvinula na seznam souborů.

Ladění programů v jazyce **`awk`** (zvláště těch, které jsou programované jako konečné automaty) je trochu snazší než ladění skriptů pro shell či **`sed`**. Našemu skriptu můžeme třeba přidat přepínač „**`-d`**“, a byl-li použit, vypíšeme v každém kroku stav automatu:

```
LADIT=0
[ "$1" = -d ] && LADIT=1 && shift
printf "$@" | awk '
'$LADIT' { print stav, hloubka, $0 }
...
```

5.15. Konfigurace programu

Zadání:

Napište program, který kontroluje práva přihlášeného uživatele k souborům zadaným na příkazové řádce. První parametr má tvar „`-r`“, „`-w`“ nebo „`-rw`“ podle toho, jaká práva se mají testovat, zbylé parametry jsou jména souborů. Program vypíše „dlouhý výpis“ pro všechny soubory, k nimž má uživatel daná požadovaná práva.

Rozbor:

Samozřejmě implicitně předpokládáme, že máme zakázáno použít příkaz **`test`**, jenž by udělal prakticky veškerou práci za nás. Kdyby tomu tak nebylo, pak by asi bylo nejjednodušší vyřešit úlohu rovnou v shellu, např. takto:

```
case $1 in
-r | -w ) op1=$1; op2=-n; shift;;
-rw     ) op1=-r; op2=-w; shift;;
esac
for soubor; do
    [ $op1 "$soubor" -a $op2 "$soubor" ] && ls -ld "$soubor"
done
```

Případ testu obou práv (čtecího i zápisového) zároveň je zřejmý (příkaz **`test`** bude potřebovat dva elementární výrazy spojené spojkou „**`-a`**“). Ale pro případ testování jediného práva jsme použili trik, kdy druhý, nepotřebný výraz je zastoupen podmínkou „**`-n`**“ (test neprázdného řetězce), která efektivně vlastně nic netestuje.

Ale zpátky k našemu problému. Musíme testování práv udělat sami. Práva umíme vypsat pomocí příkazu **`ls`**, který stejně potřebujeme zavolat. Pro ověření práv je nutné vědět, kdo je vlastníkem a skupinovým vlastníkem souboru (to také ve výpisu příkazu **`ls`** najdeme) a zda je přihlášený uživatel (jeho jméno zjistíme ze systémové proměnné **`LOGNAME`**) členem této skupiny. Podle toho vybereme ze seznamu práv první, druhou nebo třetí trojici (neboli podřetězec začínající na pozici číslo 2, 5 nebo 8) a tu budeme kontrolovat.

Nejtěžší bude zkontrolovat, zda skupinovým vlastníkem souboru je taková skupina, jejímž členem je aktuálně přihlášený uživatel. Postupovat tak, že členství uživatele budeme zkoumat vždy znova pro každou nalezenou skupinu, by bylo příliš zdlouhavé. Jednodušší bude si dopředu připravit seznam „správných“ skupin do asociativního pole – jako index použijeme jméno skupiny a jako hodnotu prvku přiřadíme buď nulu (uživatel není členem), nebo nenulové číslo (uživatel je členem). Pak nám bude stačit pro test členství u každého souboru jen ověřit hodnotu prvku pole.

Zkoumání členství už jsme několikrát dělali, můžeme tedy použít některý známý model (např. ten z kap. 5.6), přidáme jen test primární skupiny.

Pro zkoumání členství potřebujeme číst soubory `/etc/passwd` (z něj vlastně stačí jen jedna správná řádka) a `/etc/group`. V nich jako oddělovač polí použijeme určitě dvojtečku. Naopak při čtení výstupu příkazu **`ls`** by se nám hodil implicitní oddělovač polí (posloupnost mezer). Potřebujeme tedy „přepnout“ oddělovač polí v průběhu práce a to už jsme také dělali. Musíme jen poznat správné místo vstupu, kde máme změnit hodnotu proměnné **`FS`**.

Dalším oříškem bude testování požadované množiny práv. V případě testů na „`-r`“ nebo „`-w`“ by stačilo vyjmout z trojice práv první nebo druhý znak a ověřit, zda má správnou hodnotu. V případě testu „`-rw`“ potřebujeme porovnat oba znaky. Ve všech případech tedy stačí vyjmout vhodný podřetězec a otestovat ho proti „správné“ hodnotě.

To je výborná zpráva (pro čtenáře Koně a psa... :-)! Můžeme totiž připravit vhodné parametry (počáteční pozici v trojici, délku podřetězce a očekávanou hodnotu) už na začátku programu namísto neustálého rozeskakování v jeho průběhu. U interpretovaného jazyka, jakým je **`awk`**, je to důležité.

A navíc je to srozumitelné řešení. Naštěstí. Totiž pokud se opravdu vydáte cestou „musím najít elegantnější řešení“, občas se vám snadno přihodí to, že řešení za cenu elegance natolik zamotáte, že nakonec nebude ani elegantnější, ani rychlejší. Z vlastní zkušenosti vím, o čem mluvím. Proto je nutná trocha sebereflexe a „objektivního“ posouzení výsledku.

Zbývá vyřešit otázku, jak dostat efektivně do programu pro **`awk`** všechny parametry. O možnostech, jak program „nakonfigurovat“ jsme diskutovali v kapitole 5.6, musíme jen vybrat některé řešení. A když už skládáme vstup pro práci **`awk`** z tolika zdrojů, bude nejsnazší přidat na vstup i potřebné parametry a přitom zachováme plnou přenositelnost.

Navíc můžeme řádku s předávanými parametry připravit tak, abychom ji ve skriptu snadno detekovali, a použít ji tak jako zarážku, na níž přepneme oddělovač polí.

Řešení:

```
case $1 in
-r  ) param="0:1:r"; shift;;
-w  ) param="1:1:w"; shift;;
-rw ) param="0:2:rw"; shift;;
* ) echo "Navod: `basename $0` {-r|-w|-rw} soubory"; exit;;
esac
{
    grep "^$LOGNAME:" /etc/passwd
    cat /etc/group
    echo $param
    ls -ld "$@"
} |
```

```
awk '
BEGIN { FS = ":" }
NR == 1 {                       ... řádka /etc/passwd
    jmeno = $1; gid = $4; next }
NF == 3 {                       ... řádka s parametry
    posun = $1; delka = $2; text = $3; FS = " "; next }
FS == ":" {                     ... řádka /etc/group
    if( gid == $3 )
        skupiny[$1] = 1
    else
        skupiny[$1] = index( "," $4 ",", "," jmeno "," )
    next
}
{ zacatek = 8 }                 ... řádky výstupu ls
skupiny[$4] { zacatek = 5 }
$3 == jmeno { zacatek = 2 }
substr( $1, zacatek + posun, delka ) == text { print }'
```

Poznámky:

Rozeberme si podrobněji obsah vstupu pro náš program. Skládá se ze čtyř zdrojů:

- Prvním zdrojem je výstup příkazu **grep**, který vybere ze souboru `/etc/passwd` řádku se jménem uživatele. Taková bude na správně nakonfigurovaném systému právě jedna, a proto ji můžeme detekovat pořadovým číslem a uložíme si z ní jméno uživatele a číslo primární skupiny.
- Jako druhý přijde na řadu soubor `/etc/group`. Jeho řádky můžeme poznávat nejsnáze tak, že oddělovač polí je stále ještě dvojtečka. Na těchto řádkách nastavujeme hodnotu odpovídajícího prvku pole `skupiny` – nenulovou hodnotu nastavíme buď tehdy, když souhlasí číslo aktuální skupiny s číslem primární skupiny uživatele, nebo když nenulovou hodnotu vrátí funkce **index**, která hledá jméno uživatele v seznamu členů skupiny.
- Třetím zdrojem je řádka s konfiguračními parametry pro náš program. Zapsali jsme je pro jednoduchost na jednu řádku oddělené dvojtečkami, abychom teprve na této řádce mohli oddělovač změnit. V programu můžeme tuto řádku detekovat počtem polí (3).
- A konečně čtvrtým zdrojem jsou řádky výstupu příkazu **ls**.

Poslední větev programu obsahuje v podmínce vlastní test práv. Z prvního pole výstupu příkazu **ls** vybereme funkcí **substr** podřetězec potřebné délky začínající na správné pozici a zkontrolujeme ho. V případě shody vypíšeme celou řádku. Délka podřetězce a jeho správný obsah jsou pro celý běh programu stejné. Hodnota počáteční pozice podřetězce se počítá na každé řádce znova a vznikne jako součet pozice začátku

(2, 5 nebo 8 podle toho, která skupina práv je pro daný soubor a uživatele relevantní) a posunu převzatého z parametrů (tj. hodnoty 0 pro testování „`-r`“ a „`-rw`“a hodnoty 1 pro test „`-w`“).

Postup nastavení hodnoty proměnné `zacatek` je „jaksi naopak“. Kdybychom chtěli popsat algoritmus slovy, asi to bude znít zhruba takto: „je-li uživatel vlastníkem, pak je začátek na pozici 2, je-li členem skupinového vlastníka, je to pozice 5, a jinak 8“. Jenže to by se nám hůře programovalo. Kromě první podmínky by každá další musela totiž začínat „není-li dosud pozice stanovena, ...“. My jsme se tomu elegantně vyhnuli. Nejprve (vždycky) nastavíme `zacatek` na hodnotu 8. Pokud žádná další podmínka nebude splněna, tato hodnota v proměnné zůstane. Poté zkontrolujeme skupinového vlastníka a v případě úspěchu změníme hodnotu proměnné na 5. Pokud by uživatel byl přímo vlastníkem souboru, změníme hodnotu proměnné na 2.

Namísto porovnávání vhodně vybraného podřetězce jsme pro odlišení testů na různá práva mohli použít regulární výrazy. Pokud už máme správnou trojici práv, lze čtecí právo otestovat regulárním výrazem „**`r..`**“, zápisové právo výrazem „**`.w.`**“ a obě práva současně výrazem „**`rw.`**“. Namísto nastavování tří proměnných by tedy stačilo předat **`awk`** jeden regulární výraz. Jeho předání přes proměnnou by však nebylo možné, neboť test řetězce proti regulárnímu výrazu uloženému v proměnné, jak víme, není zcela přenositelný. Museli bychom tedy „konfiguraci“ programu nechat na shellu:

```
case $1 in
-r  ) param=/r../; shift;;
-w  ) param=/.w./; shift;;
-rw ) param=/rw./; shift;;
esac
... | awk '
...
substr( $1, zacatek, 3 ) ~ '$param' { print }'
```

Akce v poslední větvi by se také mohla vynechat (je shodná s implicitní akcí), ale dost by to zhoršilo čitelnost programu:

```
...
substr( $1, zacatek, 3 ) ~ '$param
```

K hodnotám proměnných prostředí se dá v novějších verzích **`awk`** přistupovat také přes speciální pole **`ENVIRON`**. Mohli bychom tedy místo nastavování proměnné `jmeno` používat „**`ENVIRON["LOGNAME"]`**“.

5.16. Maminka měla pravdu

Zadání:

Napište program, který vytvoří v aktuálním adresáři soubory pojmenované po jednotlivých skupinách uživatelů a do nich napíše seznam těch uživatelů, kteří mají danou skupinu jako svoji primární skupinu.

Rozbor:

Na první pohled nás nečeká žádná komplikace. Vytvoření mapovací tabulky GID na jména skupin už jsme dělali, teď stačí projít `/etc/passwd` a do správných souborů zapsat správné uživatele.

```
awk -F: '
FILENAME == "/etc/group"  { gid[$3] = $1 }
FILENAME == "/etc/passwd" { print $1 > gid[$4] }
' /etc/group /etc/passwd
```

Pokud si ale zkusíte program takhle napsat, u běžného systému nejspíše narazíte. Běh programu nejspíš skončí chybou „`too many open files`“. Každý zápis (příkazem **`print`** nebo **`printf`**), který použije nové jméno souboru, způsobí, že se otevře soubor s daným jménem. A standardně je dovoleno otevřít nanejvýš 10 souborů. Maminka měla pravdu! Musíme po sobě uklízet! Budeme muset soubory zavírat funkcí **`close`**. Jenže to by znamenalo zavírat je po každé řádce, což rozhodně nepatří k „dobrému vychování“ a u opravdu velkých úloh by mohlo nepříjemně zdržovat.

Lepší bude postup obrátit. Nejprve načíst uživatele a jejich skupiny a pak vypsat pro každou skupinu její členy. Potom každý soubor otevřeme a zavřeme jen jednou.

Řešení:

```
awk -F: '
FILENAME == "/etc/passwd" {
    clenove[$4] = clenove[$4] $1 "\n"
}
FILENAME == "/etc/group" && clenove[$3] != "" {
    printf "%s", clenove[$3] > $1
    close( $1 )
}' /etc/passwd /etc/group
```

Poznámky:

Drobný háček opět spočívá v tom, že funkce **`close`** bohužel není dostupná ve starších verzích **`awk`**. Pokud bychom tedy opravdu chtěli zcela bezpečně přenositelný program, nezbude nám, než zápis do souborů přenechat na práci shellu. To ovšem neznamená, že tu část práce, jež lépe zvládne program pro **`awk`**, na něm nemůžeme

nechat. Rozdíl bude pouze v tom, že pomocí něj jenom připravíme data, která shell jednoduše zpracuje a použije pro tvorbu souborů. Vhodným formátem mohou být řádky se seznamem uživatelů a názvem skupiny jako prvním slovem na řádce. Shell pak řádku přečte, oddělí název skupiny a zbytek řádky zapíše do souboru:

```
awk -F: '
FILENAME == "/etc/passwd" {
    clenove[$4] = clenove[$4] $1 " "
}
FILENAME == "/etc/group" && clenove[$3] != "" {
    print $1, clenove[$3]
}' /etc/passwd /etc/group | while read soubor lidi; do
    echo $lidi | tr ' ' '\n' > $soubor
done
```

Takováto dělba práce mezi filtrem **awk** a shellem je naprosto podle zásady „nechť každý dělá to, co nejlépe umí“. Samozřejmě je i tuto zásadu třeba aplikovat s rozumem a program netříštit na zbytečně malé kousky, ale v principu je to zásada dobrá a je třeba ji mít na paměti, jakmile se nám program začne příliš „zamotávat“. Na druhé straně ale musíme počítat s tím, že každý přenos řízení mezi programy bude znamenat ztrátu času. Zkrátka a dobře – univerzální návod nám nikdo nedá a vždy bude třeba zohlednit všechny aspekty daného případu.

Dělba práce může jít ještě o kousek dál a můžeme shellu dokonce rovnou připravit řádky s příkazy, které shell pouze provede. Vypadaly by asi takto:

```
cat > skupina << KONEC
člen1
člen2
KONEC
```

V tom případě bychom kód programu upravili takto:

```
awk -F: '
FILENAME == "/etc/passwd" {
    clenove[$4] = clenove[$4] $1 "\n"
}
FILENAME == "/etc/group" && clenove[$3] != "" {
    printf "cat > %s << EOF\n%sEOF\n", $1, clenove[$3]
}' /etc/passwd /etc/group | sh
```

Nakonec můžeme shell z celé akce úplně „vynechat“ a provedení shellového příkazu nechat na funkci **system**, která dokáže spustit shell sama (takže jsme ho zase tak moc nevynechali...) a zavolat zadaný příkaz přímo z prostředí **awk**:

```
awk -F: '
FILENAME == "/etc/passwd" {
    clenove[$4] = clenove[$4] $1 "\n"
}
FILENAME == "/etc/group" && clenove[$3] != "" {
    system( "cat > " $1 " << EOF\n" clenove[$3] "EOF\n" )
}' /etc/passwd /etc/group
```

Tím jsme si ale příliš nepomohli, protože funkce **system** patří opět mezi rozšíření. A navíc je toto řešení ze všech nejpomalejší.

Funkce **system** vrací návratovou hodnotu provedeného příkazu, lze tedy posléze testovat jeho úspěšnost. Naproti tomu jeho výstup se normálně vypisuje do standardního výstupního proudu (tj. neřekneme-li jinak, objeví se na obrazovce terminálu), a proto ho není možné v našem programu nijak dále zpracovávat. Způsob, jak toto omezení obejít, uvidíme v následující kapitole.

5.17. Jak nepodlehnout reklamě

Zadání:

Napište program, který dostane jako parametr číslo procesu a vypíše stav tohoto procesu, jeho rodiče, prarodiče atd. až na konec tohoto řetězu. Ten poznáte tak, že poslední proces má uvedeno jako číslo rodiče buď sebe sama, nebo číslo nula.

Rozbor:

S příkazem **ps** jsme už pracovali. Novinkou je, že budeme chtít výpis konkrétních informací („**-o** *seznam*“) o konkrétním procesu („**-p** *PID*“). Seznam požadovaných atributů se zadává podobně jako jsme to měli v zadání v kapitole 4.30. Když do seznamu přidáme číslo rodičovského procesu („**ppid**“), budeme mít veškerá potřebná data.

Máme-li připravit program pro **awk**, budeme v něm muset umět přečíst výstup příkazu **ps**. Funkce **system**, kterou jsme poznali minule, sice zavolá příkaz a vrací hodnotu, ale je to *návratová hodnota* spuštěného příkazu (číslo od 0 do 255). Pokud příkaz vyprodukuje něco na svůj výstup, pak se tento text skutečně na standardní výstup vypíše a my se k němu v programu nijak nedostaneme. Potřebovali bychom *rouru*, stejně jako jsme ji potřebovali v shellu.

Novější verze **awk** a norma rouru opravdu znají. Chceme-li číst výstup nějakého příkazu, nejsnazší je připravit si jeho znění do proměnné a pak napsat jméno proměnné, svislítko a příkaz **getline**. Tím spustíme příkaz a přečteme první řádku výstupu do vstupního záznamu (**$0**). Pokud chceme číst více řádek, můžeme konstrukt s rourou zařadit třeba do cyklu. Po ukončení čtení zavoláme příkaz **close**.

Naši úlohu bychom s pomocí roury mohli vyřešit například následujícím způsobem:

```
awk '
BEGIN {
    dalsi = '$1'
    do {
        proces = dalsi
        prikaz = "ps -opid,ppid,tty,time,comm -p " proces
        prikaz | getline                  ... čte hlavičku výstupu
        if( ! ( prikaz | getline ) )      ... čte řádku výstupu
            break                         ... konec, pokud čtení selhalo
        print
        dalsi = $2                        ... 2. sloupec je číslo rodiče
        close( prikaz )
    } while( dalsi && dalsi != proces )
    exit
}'
```

Celá činnost programu se odehrává v jednom cyklu (cyklus **do-while** je obdobou cyklu **while** s tím rozdílem, že test podmínky se provádí až na konci kroku), v jediné větvi **BEGIN**, protože na vstupu nic nemáme (číslo procesu jako jediný vstupní údaj jsme si přidali do kódu pomocí prostředků shellu) a zbytek informací (především hodnotu parametru, s nímž potřebujeme příkaz **ps** spouštět) se postupně dozvídáme až v průběhu práce. Cyklus ukončíme, když dorazíme na konec řetězu rodičovských procesů.

Použití roury je ale vždy třeba pečlivě rozmyslet. Jednak je roura (podobně jako funkce **system**) rozšířením oproti základní verzi **awk**, ale především představuje určitou komplikaci z hlediska řízení programu, a tak je třeba zvážit, zda je její použití nezbytné.

Což kdybychom si namísto neustálého volání příkazu **ps** pro jednotlivé procesy nechali nejprve jedním voláním **ps** zobrazit veškerá potřebná data, vhodným způsobem bychom si je uložili a posléze je (už bez interakce se systémem) vypsali?

Řešení:

```
ps -A -opid,ppid,tty,time,comm | awk '{
    radka[$1] = $0
    rodic[$1] = $2
}
END { dalsi = '$1'
    do {
        proces = dalsi
        print radka[proces]
        dalsi = rodic[proces]
    } while( radka[dalsi] > "" &&        ... další proces ještě existuje
        dalsi != proces )                ... a nejsme to my sami
}'
```

Poznámky:

Tento příklad je docela pěknou ukázkou toho, jak pečlivá analýza dokáže často odvrátit zdánlivou nutnost použití zbytečně složitých programových konstruktů, a někdy dokonce i tak, že výsledný program je rychlejší. Stejně jako v jiných oblastech i zde je třeba „nepodlehnout reklamě“, neskočit na všechny „bezkonkurenční nabídky“ a raději popřemýšlet, zda nemáme po ruce lepší a jednodušší řešení.

Pokud vás program bude zlobit a příkaz **ps** bude psát cosi o neplatných přepínačích, pak váš systém nejspíše patří mezi starší exempláře, které nepodporují normu. Kořeny problému v tomto případě tkví hluboko v minulosti.

V pradávné historii UNIXu došlo k oddělení vývojových větví na dvě linie (nyní) nazývané *BSD systém* a *Systém V*. Důvody, které k tomu vedly, byly veskrze pozitivní, ale bohužel tehdy nikdo nedohlédl, že se dobrá myšlenka může zvrtnout a že toto

rozdělení bude oním pomyslným smítkem, jež spustí lavinu. Během několika dalších let se různé klony namnožily natolik, že to způsobilo nepřenositelnost programů, se kterou se potýkáme dodnes. UNIX se v té době vzdal myšlenky „dělat věci tak, aby dobře pracovaly“ a přešel na strategii „dělat věci tak, aby dobře vydělávaly“ a k tomu se nepřenositelnost velmi hodila. A cesta zpět je trnitá...

Až doposud jsme vás zatím podobných detailů ušetřili. Jak vidno, skoro celou knihu jsme tento typ problémů nemuseli řešit, takže si můžete odnést poznatek, že pokud se člověk pohybuje v rozsahu základních funkcí, nemusí ho rozdílnost unixových systémů příliš trápit, ale není možné ji zcela pustit z hlavy.

Náš poslední případ je toho ukázkou. Příkaz **`ps`** patří mezi temné výjimky, u nichž prostě není možné zvolit nějaké řešení, které by bylo nezávislé na rodině unixových systémů. U tohoto příkazu teprve člověk ocení snahu normalizátorů, aby se jednotlivé unixové systémy k sobě zase přiblížily. Na systémech odvozených z BSD budete muset místo přepínače „**`-A`**“ (*all*, vypiš všechny procesy) použít kombinaci „**`-ax`**“, zatímco Systém V bude vyžadovat „**`-e`**“. Můžete si zkusit tuto variabilitu dokonce zabudovat do našeho programu (zkusit postupně, zda bez chyby „projde“ volání příkazu s přepínačem „**`-A`**“, resp. „**`-ax`**“, a podle toho připravit přepínače pro správné volání **`ps`**).

Pokud byste chtěli vědět, jakého typu je váš systém, pomůže vám příkaz **`uname`**. Kdybyste ale rozlišení formy přepínačů příkazu **`ps`** chtěli založit na této informaci, museli byste ošetřit každý typ systému zvlášť. Jednodušší je proto vyzkoušet rovnou správné přepínače.

Pro usnadnění možnosti jednotného zpracování svého výstupu příkaz **`ps`** naštěstí umožňuje překlenout rozdíly mezi systémy a nadefinovat vlastní sadu sloupců, které chceme zobrazit (přepínač „**`-o`**“).

Příkazy **`break`** a **`continue`** fungují v jazyce **`awk`** úplně stejně jako v shellu nebo v jazyce C – příkaz **`break`** ukončí provádění cyklu, příkaz **`continue`** ukončí krok cyklu a cyklus dál pokračuje.

Podobně jako vstup z roury lze používat i výstup do roury – za příkazy **`print`** nebo **`printf`** lze napsat svislítko a příkaz. To si také vyzkoušíme později.

V minulé kapitole jsme varovali před omezením počtu otevřených souborů. U rour je situace ještě horší. Otevřená smí být jenom jediná!

5.18. Podsouvání souborů

Zadání:

Napište program, který hledá v systémových hlavičkových souborech definici makra a vypisuje jméno souboru, kde se definice nachází. První parametr obsahuje jméno makra, druhý jméno souboru, kde má program začít hledání – pokud tam definici nenalezne, pokračuje hledáním v dalších vložených souborech.

Hlavičkové soubory slouží programátorům v jazyce C k tomu, aby mohli používat různé systémové funkce či konstanty. Mají příponu „h“ a v UNIXu jsou obvykle uloženy v adresáři **`/usr/include`** a jeho podadresářích. V kódu (programu nebo jiného hlavičkového souboru) se odkaz na vložený hlavičkový soubor vyjadřuje pomocí konstruktu „`#include` <*jméno_souboru*>“ (používá se přitom relativní cesta vůči adresáři `/usr/include`). Makro (konstanta nebo jednoduchá „funkce“) se definuje pomocí konstruktu „`#define` *jméno* ...“ nebo „`#define` *jméno*`(...)` ...“. Všechny uvedené konstrukty musejí být na samostatné řádce a mohou navíc obsahovat libovolné množství bílých znaků.

Rozbor:

Vlastní hledání direktivy `#define` nepředstavuje žádný problém.

Problémem je, že seznam prohledávaných souborů se nám bude dynamicky měnit, jak budeme postupně nacházet další vložené soubory. Vzhledem k tomu, že seznam vstupních souborů dostává **`awk`** v parametrech, znamenalo by to volat ho opakovaně. Jednou z možností, jak řešení pojmout, je spolupráce **`awk`** a shellu. Zatímco program v jazyce **`awk`** bude hledat definice makra a odkazy na vkládané soubory, shell bude pro jeho další instanci připravovat seznam jmen souborů, které mu předá jako parametry.

My si ale ukážeme postup, kterým lze celé řešení realizovat, aniž přitom opustíme jedinou instanci **`awk`**. Filtr totiž sice postupně čte všechny soubory, které dostane zadané na příkazové řádce, ale tyto parametry si sám nejprve uloží do vlastních proměnných **`ARGC`** (počet parametrů) a **`ARGV`** (od jedničky číslované pole se seznamem hodnot parametrů). Pokud za běhu změníme obsah těchto dvou proměnných, filtr to bude respektovat a bude otevírat soubory dle našeho přání!

Toto řešení má ještě jednu velkou přednost. Doposud jsme ve svých úvahách zanedbávali drobnou komplikaci, a to že odkazy mezi jednotlivými vkládanými soubory mohou tvořit cyklus (dva nebo více souborů se může odkazovat na sebe navzájem). Aby se náš program v takovém případě nedostal do nekonečné smyčky, budeme muset před zařazením každého souboru do seznamu k dalšímu zpracování nějak kontrolovat, zda daný soubor už na seznamu není. A tuto kontrolu budeme daleko snáze programovat v jazyce **`awk`** – pomocí asociativního pole `zarazen`.

Řešení:

```
awk '
BEGIN { zarazen[ARGV[1]] = 1 }
/^[ \t]*#[ \t]*define[ \t]*'$1'[ \t(]?/ {
    print FILENAME
    exit
}
/^[ \t]*#[ \t]*include[ \t]*</ {
    n = split( $0, r, "[<>]" );
    soubor = "/usr/include/" r[2];
    if( ! zarazen[soubor]++ )
        ARGV[ARGC++] = soubor;
}' $2
```

Poznámky:

Proměnné **ARGC** a **ARGV** jsou při startu programu naplněny podobně jako parametry funkce **main** v jazyce C. V prvku **ARGV[0]** je uloženo jméno programu („awk“) a tento prvek se započítává do hodnoty **ARGC** (má tedy hodnotu vždy alespoň jedna).

Do seznamu parametrů se ovšem nezařazuje vlastní program pro **awk** a ani případné přepínače, proto je v prvku **ARGV[1]** první následující „skutečný“ parametr (obvykle tedy jméno prvního vstupního souboru).

Proměnné **ARGC** a **ARGV** bohužel nenajdeme ve starších implementacích.

Test hodnoty `zarazen[soubor]` ve druhé větvi je opět maličko trikový. O něco čitelnější by bylo:

```
    if( zarazen[soubor] == 0 ) {
        zarazen[soubor] = 1; ...
```

My jsme si poněkud usnadnili práci a patřičný prvek pole v rámci jednoho příkazu otestujeme a zároveň nastavíme na nenulovou hodnotu. To, že onou hodnotou nebude vždy jednička, ale číslo, které postupně poroste, nás nijak netrápí. Kdybychom podobný trik použili třeba v jazyce C, museli bychom přinejmenším provést úvahu, zda se nám číslo nemůže „přetočit“ (díky omezenému počtu bytů, v nichž je uloženo) zase zpátky do nuly. Kritickou hranicí je obvykle číslo 4 294 967 295, takže by to bylo jistě bezpečné. Ovšem v jazyce **awk** nás to trápit už vůbec nemusí, neboť hodnota proměnné je uložena textově, a proto poroste i přes tuto hranici.

Funkci **split** jsme tentokrát použili s jinou formou zadání oddělovače. Podobně jako u proměnné **FS** musela norma zápasit s různými implementacemi a vyřešila to stejně: pokud má oddělovač délku jedna, je to prostý řetězec, pokud je delší, je to regulární výraz.

Další náměty na procvičení

1. V předchozí části knihy (o shellu) jsme jako námět na procvičení programovali funkci přepínače „**-n**“ příkazu **cat**. Zkuste totéž pomocí **awk**.
2. V závěru kapitoly 4.31 jsme diskutovali možnosti, jak při interakci s uživatelem lze obejít problém přenositelnosti výpisu výzvy bez odřádkování. Napište pomocný program v jazyce **awk**, jenž zobrazí výzvu, přečte odpověď a vhodným způsobem předá volajícímu informaci, kterou z možných odpovědí uživatel vybral.
3. Vyřešte úlohu z kapitoly 3.16 pomocí **awk**.
   Návod: Jednu inspiraci můžete čerpat z řešení v editoru, druhou z kapitoly 5.13.
   Porovnejte rychlost všech řešení za pomoci příkazu **time**.
4. Vyřešte úlohu z kapitoly 3.13 pomocí **awk**.
5. V předchozích dvou částech knihy jsme jako námět na procvičení uváděli převod `/etc/passwd` do formátu HTML. Zkuste to pro změnu pomocí **awk**.
6. Naprogramujte příklad z kapitoly 4.30 pomocí **awk**.
7. Naprogramujte příklad z kapitoly 4.36 pomocí **awk**.
8. V řešení příkladu z kapitoly 5.13 jsme zanedbali případy, kdy jeden dokument RFC má více předchůdců (tj. vede na něj odkaz „`(Obsoleted by ...`“ z několika jiných dokumentů) anebo následníků (tj. v odkazu se vyskytuje více dokumentů oddělených čárkami, např.: „`(Obsoleted by RFC1034, RFC1035)`“). Upravte řešení tak, aby tyto případy také dokázalo zvládnout.
9. V kapitole 5.18 jsme v úvodu rozboru naznačili, jak by šlo řešení zvládnout bez triku s podsouváním souborů – spoluprací shellu a filtru **awk**. Zkuste si tuto verzi napsat.
10. Ve třetí části knihy (o editorech) jsme jako námět na procvičení uváděli formátování odstavců textu tak, aby délka jednotlivých řádek nepřesáhla určitou hranici. V jazyce **awk** můžete řešení ještě zdokonalit tak, aby skript kratší řádky doplňoval mezerami mezi slovy na přesnou délku, kterou navíc můžete mít volitelnou pomocí parametru.
11. V předchozí části knihy (o shellu) jsme jako námět na procvičení uváděli emulaci funkce příkazu **date**. Zkuste totéž pomocí **awk**, ale bez použití funkce **sub**.

12. Napište program pro kontrolu rejstříku této knihy. Jako zdroj máte textový soubor s definicemi rejstříkových odkazů (najdete ho na webových stránkách knihy [13]). Odkazy jsou dvou typů, nejprve „normální“ odkazy na čísla stránek a po nich následují odkazy, které namísto čísla stránky obsahují text „viz *jiná položka rejstříku*“. Název každé rejstříkové položky může navíc být „strukturovaný“ pomocí dvojteček – v takovém případě se v rejstříku vytvoří odsazení.

   Příklad definice:

```
XE "break (awk)"
XE "break (shell)"
XE "date"
XE \t "viz datum a čas" "čas"
XE \t "viz date" "datum a čas:aktuální"
XE \t "viz break" "ukončení cyklu"
```

   Pouze pro informaci – výsledný rejstřík vytvořený z této definice by vypadal zhruba takto (byl by samozřejmě abecedně setříděný, což soubor s definicí určitě nebude):

   break (awk) *stránka...*
   break (shell) *stránka...*
   čas viz datum a čas
   date *stránka...*
   datum a čas
   aktuální viz date
   ukončení cyklu viz break

   Kontrola musí odhalit případné nesprávné rejstříkové položky typu „viz ...“. Položka `XE \t "viz B" "A"` je nesprávná tehdy, pokud

   a) existuje stejnojmenná položka s přímým odkazem na číslo stránky (`XE "A"`)
   b) neexistuje položka se jménem „`B`“ (tedy `XE "B"`, resp. `XE \t "C" "B"`) ani žádná „odvozená“ položka (tj. se jmény `"B (C)"`, `"B:C"`, `"B (C):D"` atd.)
   c) položka se jménem „`B`“ sice existuje, ale sama je odkazem typu „viz ...“ (tj. např. `XE \t "viz C" "B"`).

Část 6 Případové studie

V poslední části knihy se pokusíme podrobněji rozebrat několik složitějších příkladů, abychom si ukázali použití stavebních kamenů, s nimiž jsme se v knize seznámili, pro budování trochu rozsáhlejšího projektu. Jedním z hlavních smyslů této části je předvést, že pomocí relativně jednoduchých nástrojů jsme v shellu schopni řešit netriviální praktické problémy.

6.1. Stahování webových stránek

Zadání:

Napište program, který dostane jako parametr adresu webové stránky zapsanou ve formátu URI („`http://`*server*/*cesta*/*stránka*“), stránku stáhne do aktuálního adresáře, přečte ji, najde v ní všechny odkazy vedoucí do stejného adresáře (resp. do podstromu pod tímto adresářem) a stáhne rovněž tyto soubory.

URI (Uniform Resource Identifier) je formát pro zápis (nejen) webových adres. V textu stránky je třeba zpracovat i neúplné a relativní cesty, tj. následující formáty URI:

//server/cesta/stránka	`http://`*server/cesta/stránka*
/cesta/stránka	`http:`*/cesta/stránka*
stránka	`http:`*stránka*

Skript musí umět rovněž rozpoznat i odkazy obsahující fragmenty (odkaz ve tvaru „*stránka*`#`*jméno*“ je z našeho hlediska totožný s odkazem „*stránka*“) a relativní odkazy vedoucí mimo základní adresář (např. „`../index.html`“). Musí být schopen odlišit odkazy jiných schémat (např. „`mailto:`…“).

Základní tvar zápisu odkazů v HTML je „`<a href="`*odkaz*`">`“, ale skript musí umět zpracovat značky („`A`“) nebo atributy („`HREF`“) zapsané velkými písmeny, s vloženými mezerami nebo konci řádek, s vynechanými uvozovkami, s přidanými atributy apod.

Pro vlastní stahování můžete použít program **`netcat`**, který se umí spojit na webový server zadaný v parametrech, přečte obsah svého standardního vstupu, pošle ho serveru a jeho odpověď zase vypíše na standardní výstup. Volá se:

```
sinek:~> netcat server 80
```

S webovým serverem je nutno komunikovat podle pravidel, kterým se říká *protokol*, konkrétně je to Hypertext Transfer Protocol (HTTP, [10]). Abyste od serveru dostali text stránky, musíte mu poslat požadavek, který může obsahovat například tyto dvě řádky:

```
GET http://server/cesta/stránka HTTP/1.0

```

Odpověd, kterou dostanete od serveru (a **`netcat`** ji vypíše), bude obsahovat kromě vlastního textu stránky ještě další informace – zhruba tyto:

```
HTTP/1.1 200 OK
... hlavičky odpovědi
                                        prázdná řádka oddělující hlavičku
... text stránky
```

Pro tuto chvíli můžete nadbytečné řádky ignorovat. Spravíme to v následující kapitole.

Rovněž můžete ignorovat další vlastnosti jazyka HTML, které jsme zde nezmiňovali, formuláře nebo zvláštní znaky (např. mezery) v URI.

Příklad vzorové stránky ke stažení najdete na webových stránkách knihy [13].

Rozbor:

Pokusíme se ukázat řešení vystavěná na třech různých základech (shell, **sed** a **awk**). V následující kapitole si z každého z nich vybereme to nejlepší.

Rozbor (shell):

Prvním úkolem je rozložit URI na schéma („`http:`“), jméno serveru, cestu a jméno stránky. Tuto operaci budeme provádět častěji, proto si napíšeme funkci (`rozdel`), která URI dostane jako parametr a výsledek uloží do proměnných `server`, `cesta` a `jmeno`.

Pokud se omezíme na prostředky shellu, pak samotný rozklad nejsnáze uděláme příkazem **set** s hodnotou proměnné **IFS** nastavenou na lomítko. Tím uložíme jednotlivé složky URI do pozičních parametrů. a ty pak postupně zpracujeme. Výhodou tohoto řešení je, že přes svoji zdánlivou složitost je poměrně rychlé (v rámci možností shellu) – vystačí si totiž s interními příkazy shellu a nebude nutné spouštět žádné další procesy.

Po rozdělení budeme mít v pozičních parametrech následující hodnoty: parametr `$1` bude obsahovat schéma (to už dále potřebovat nebudeme), `$2` bude prázdný, v `$3` bude jméno serveru (to uložíme do proměnné `server`), od `$4` počínaje budou jednotlivé složky cesty a nakonec jméno stránky. Abychom dostali zase celou cestu do proměnné, odstraníme první tři parametry (příkazem **shift**) a zbytek (až na poslední parametr) budeme postupně v cyklu přidávat do proměnné `cesta`.

Zkusme si ale rozmyslet, jaké chování funkce budeme potřebovat v dalších krocích až budeme zpracovávat další URI. To už budeme znát začátek cesty (k základnímu adresáři, kde leží první stahovaná stránka) a bude nás zajímat jen zbytek cesty (abychom věděli, kam novou stránku uložit). Potřebovali bychom tedy odstranit nikoliv první tři složky cesty, ale o něco více (o délku cesty k základnímu adresáři). To lze zařídit snadno – při prvním volání funkce si zjištěnou délku cesty uložíme do proměnné `hloubka` a při dalších voláních použijeme právě tuto hodnotu jako parametr příkazu **shift**.

Malou komplikaci nám budou působit odkazy vedoucí na adresář. Tyto odkazy končí lomítkem (naše proměnná `jmeno` obsahující poslední část cesty bude tedy prázdná), ale my musíme stránku, kterou od serveru dostaneme, uložit pod nějakým jménem souboru. Použijeme jméno `index.html` – tak se stránka doopravdy velmi často jmenuje. Jenže teď máme rozdílná jména pro URI, které potřebujeme poslat serveru, a pro soubor, který budeme vytvářet. Uložíme si je tedy do dvou proměnných: jméno z URI do proměnné `jmeno` a jméno souboru do proměnné `soubor`.

To ale ještě není všechno. Pokud URI bude opravdu končit lomítkem, shell toto lomítko (a prázdnou část cesty za ním) při přípravě parametrů příkazu **set** odstraní. Použijeme tedy jednoduchý trik. Za URI přidáme ještě jednu fiktivní složku cesty („`/.`“) a při zpracování ji budeme ignorovat (proto také máme v podmínce cyklu „**-gt 2**“ a nikoliv „**-gt 1**“).

Pro úplnost ještě dodejme, že ve skutečnosti mohou v odkazech na adresář koncová lomítka chybět, ale řešení tohoto problému také ještě odložíme. Museli bychom hlouběji proniknout do obsahu odpovědi serveru a komunikace s ním a to by nám výklad příliš zkomplikovalo, aniž by to přineslo nějaké zásadní nové informace.

Druhou pomocnou funkcí bude `stahni`. Dostane jako parametr URI, rozdělí ho (pomocí funkce `rozdel`) na cestu a vlastní název stránky, prověří existenci potřebné adresářové struktury v naší lokální kopii (cestu od kořene původní stránky), případné chybějící adresáře vytvoří („**`mkdir -p`**“), zavolá příkaz **`netcat`**, předá mu na vstup správný tvar požadavku a staženou stránku uloží.

Tuto funkci zavoláme poprvé na začátku práce skriptu, abychom stáhli základní stránku do souboru, ve kterém musíme hledat další odkazy. Pokud v shellu potřebujeme zpracovávat soubor, jehož struktura neodpovídá jednotlivým řádkám textu (už víme, že HTML má velmi volnou strukturu), je to poněkud obtížné. Jedním schůdným řešením je vytvořit proměnnou (`text`) a do ní postupně přidávat obsah dalších načtených řádek, dokud její hodnota neobsahuje celistvý úsek, který potřebujeme. V našem případě je tímto úsekem kompletní značka, proto v cyklu čekáme na výskyt koncového znaku „>“.

O něco menší potíže bude působit nutnost zpracovat na řádce postupně všechny odkazy. To zařídíme tak, že necháme řádku rozdělit na pole podle znaku „<“ a postupně tato pole obsahující jednotlivé značky zpracováváme. Jakmile dorazíme na konec řádky nebo na nekompletní značku, vnitřní cyklus ukončíme, vyplníme správně obsah proměnné `text` a pokračujeme ve vnějším cyklu (doplněním o text další řádky).

Vlastní zpracování odkazu (vyjmutí a kontrolu) řeší další funkce `odkaz`. Značku si znova rozdělíme pomocí příkazu **`set`** na slova (tentokrát „obyčejně“ podle bílých znaků), zkontrolujeme, že první slovo je „`A`“ a pak prohledáváme další slova, dokud nenarazíme na atribut „`href`“. Předpokládáme přitom, že rovnítko mezi atributem a jeho hodnotou má mezery buď z obou stran, nebo ani z jedné. Není to silné omezení a ošetření všech variant by bylo jen o něco více psaní.

Pokud atribut nebyl nalezen, funkci ukončíme. Pokud je URI uzavřeno do uvozovek, další postup je snadný – stačí značku rozdělit na slova podle uvozovek a vyjmout druhé slovo. V opačném případě potřebujeme vzít řetězec mezi rovnítkem a většítkem. Bylo by tedy možné tyto dva znaky dát do proměnné **`IFS`** a opět vyjmout druhé slovo.

Dalším krokem je kontrola a odstranění schématu. Poté už pouze zkontrolujeme, že odkaz obsažený v novém URI míří do podstromu, který stahujeme (zkontrolujeme začátek URI), doplníme případně potřebný prefix („`http://`*server*`/`*cesta*`/`“) a necháme novou stránku stáhnout.

Řešení (shell):

```
rozdel() {
    pom=$1/.                    ... přidání fiktivní složky kvůli adresářům
    IFS=/; set -- $pom          ... složky cesty v pozičních parametrech
    if [ -z "$hloubka" ]; then
        server=$3               ... nastavení jména serveru
        shift 2
        hloubka=$#              ... přiřazení hodnoty $# - 2
        shift
    else
        shift $hloubka          ... odstranění cesty po kořen základní stránky
    fi
    cesta=
    while [ $# -gt 2 ]; do      ... skládání cesty z jednotlivých složek
        cesta="$cesta/$1"
        shift
    done
    jmeno=`echo "$1" |          ... jméno souboru jako poslední složka
        cut -d# -f1`            ... odstranění jména fragmentu
    [ -n "$jmeno" ] && soubor=$jmeno || soubor=index.html
}
stahni() {
    rozdel "$1"
    [ -d ".$cesta" ] || mkdir -p ".$cesta"
    netcat $server 80 > ".$cesta/$soubor" << KONEC
GET $1 HTTP/1.0

KONEC
}
odkaz() {
    IFS=' '; set -- $1
    [ "$1" != "a" -a "$1" != "A" ] && return 1
    shift
    odkaz=
    while [ $# -gt 0 ]; do
        case $1 in
        [hH][rR][eE][fF]=* ) odkaz=$1; break;;
        [hH][rR][eE][fF] )   odkaz="href=$3"; break;;
        *=* ) shift; continue;;
        *   ) shift 3; continue;;
        esac
    done
```

```
    case $odkaz in
    '' )   return 2;;
    *\"* ) IFS='"';;
    * )    IFS='=>';;
    esac
    set -- $odkaz
    uri=$2
    IFS=:; set -- $uri
    if [ $# -gt 1 ]; then
        case $1 in
        http ) uri=$2;;
        '' | *[!-+.A-Za-z]* ) ;;
        * ) return 3;;
        esac
    fi
    case $uri in
    //$server$koren/* ) stahni "http:$uri";;
    $koren/* ) stahni "http://$server$uri";;
    /* | ..* | \#* ) return 4;;
    * ) stahni "http://$server$koren/$uri";;
    esac
}
# zacatek hlavniho programu
case $1 in
http://* ) ;;
* ) echo "Spatne URI $uri" >&2; exit 1;;
esac
rozdel "$1"                    ... rozdělení zadaného URI
koren=$cesta                   ... uložení cesty k základnímu adresáři
stahni "$1"                    ... stažení základní stránky
text=; while read nova; do # cteme ze souboru $soubor
    IFS='<'
    set -- $text $nova         ... doplnění a rozdělení řádky
    shift                      ... vypuštění textu před první značkou
    text=
    while [ $# -gt 0 ]; do     ... postupné zpracování značek
        case $1 in
        *\>* ) odkaz "$1"      ... zpracování (kompletní) značky
               shift;;         ... posun na další značku
        *    ) text="<$1"      ... uložení nekompletní značky do proměnné
               break;;         ... přechod na načtení další řádky
        esac
    done
done < $soubor
```

Poznámky (shell):

Možná vás napadlo, že by šlo ušetřit ve funkci `rozdel` jednu řádku a napsat rovnou:

```
set -- $1/.
```

To ale nepůjde. Říkali jsme si, že shell provádí druhé dělení řádky na slova pouze v té části řádky, kterou sám doplňoval v rámci substitucí. Proto by poslední lomítko v tomto příkazu za oddělovač už nepovažoval a celý trik s přidáním pole by nefungoval.

K nastavení hodnoty proměnné `hloubka` jsme použili malý trik. Potřebovali jsme do ní uložit hodnotu **`$#`** sníženou o 2. K tomu bychom potřebovali zavolat **`expr`**. Zároveň jsme ale potřebovali posunout parametry příkazem „**`shift 3`**“, který **`$#`** sníží o 3. Když tedy rozdělíme posouvání parametrů na dva kroky (nejprve „**`shift 2`**“ a pak „**`shift`**“), bude po prvním kroku v **`$#`** přesně ta hodnota, co potřebujeme.

Při programování podobných úloh je vždy velmi důležité si pečlivě rozmyslet, co přesně která proměnná bude obsahovat, obzvláště v krajních situacích. V našem případě je zásadní obsah proměnných `koren` a `cesta`: především to, zda lomítkem začínají nebo končí a jak vypadají, když v URI žádná cesta nebyla. Odpovězme si tedy na tyto otázky – při „prázdné“ cestě jsou proměnné prázdné a jinak lomítkem vždy začínají a nikdy nekončí. Nyní snadno poznáme, že například řetězec „**`.$cesta/$soubor`**“ funguje správně pro prázdný i neprázdný obsah proměnné `cesta`.

Třetí větev posledního **`case`** ve funkci `odkaz` by si asi zasloužila malý komentář. První vzor („**`/*`**“) reprezentuje odkaz ve tvaru absolutní cesty. Zdánlivě libovolné. Ale díky tomu, že v předchozí větvi jsme ošetřili případ, kdy absolutní cesta začíná správnou cestou ke kořeni, zbylé případy představují špatné absolutní odkazy. Druhý vzor („**`..*`**“) řeší cestu „vzhůru“ mimo náš strom. A třetí („**`\#*`**“) ošetřuje formu URI, která obsahuje pouze odkaz na fragment, tj. dovnitř právě zpracovávané stránky. Takový odkaz není nutné dále zpracovávat, protože odpovídající stránka je již stažena. A zpětné lomítko před znakem „**`#`**“ je obyčejný quoting v shellu, protože jinak by byl zbytek řádky považován za komentář.

Pokud budete psát skripty, které průběžně mění hodnotu proměnné **`IFS`**, buďte připraveni na to, že něco, co vypadá zcela neškodně, se občas bude chovat zdánlivě proti zdravému rozumu. Naučí vás to mj. důsledně používat uvozovky kolem parametrů.

Náš skript není odolný vůči léčkám, jako jsou např. odkazy ve tvaru „`x/../../y`“ (vede mimo základní strom, ačkoliv tak na první pohled nevypadá). Zatím to nebudeme řešit a spravíme to v následující kapitole.

Rozbor (`sed`):

Udělat řešení čistě pomocí editoru **`sed`** určitě nepůjde. Editor neumí spustit program a to my budeme potřebovat. Ovšem pokud budeme chtít opravdu téměř veškerou práci nechat na editoru, je možné pomocí něj připravit příkazy pro shell tak, aby je shell už pouze spustil. Někdy opravdu existují situace, kdy příprava dat pro shell tak, aby se jimi dokázal správně řídit, je natolik složitá, že je jednodušší mu rovnou připravit správné příkazy k vykonání. A nemusí jít nutně jen o shell. Klidně může editor připravit program pro filtr **`awk`** nebo dokonce jinou instanci editoru.

Vytvoříme opět funkci `stahni` a v ní editorem vyrobíme potřebné volání příkazu **`netcat`**, prakticky shodné s tím, které jsme viděli v minulém řešení. Vzhledem k tomu, že použijeme „stream" editor **`sed`**, bude výhodné mít URI ve vstupním proudu. Proto budeme naší funkci požadované URI posílat na vstup namísto předávání třeba pomocí parametru. Toho posléze s výhodou využijeme, až budeme potřebovat stáhnout zbytek souborů – jednotlivá URI prostě opět jen pošleme rourou naší funkci.

Přesněji řečeno, o něco jednodušší bude, když funkce bude dostávat pouze „lokální" část URI (bez schématu, serveru a cesty ke kořeni základní stránky), ale včetně úvodního lomítka. Jméno serveru a cestu ke kořeni (pro potřeby příkazu **`netcat`**) budeme mít uložené v proměnných shellu.

Skript pro **`sed`** ve funkci `stahni` bude vcelku jednoduchý. První příkaz odstraňuje z URI případný fragment. Další dva příkazy slouží pouze k ošetření problému stahování adresáře podobně, jako jsme to museli dělat v minulém řešení. Potřebujeme si opět připravit název souboru, pod nímž stránku uložíme. Zapíšeme si ho za text URI a oddělíme ho lomítkem (to se ve jméně souboru jistě vyskytovat nebude). Nejprve tedy opíšeme za URI ještě jednou název stránky a poté změníme tento název na „`index.html`", pokud URI končilo lomítkem. Poslední editační příkaz pak zamění řádku obsahující URI a název souboru za několik řádek s potřebnými příkazy shellu (pro vytvoření adresáře a stažení stránky). V regulárním výrazu ve vzoru si text rozdělíme na tři části: cestu od základní stránky k požadované stránce včetně úvodního a koncového lomítka („**`\1`**"), název stránky („**`\2`**") a název souboru („**`\3`**") a ty pak použijeme v řetězci náhrady.

Jako oddělovač řetězců u příkazů *substitute* zvolíme znak „**`#`**". Tradiční lomítko totiž nejde jednoduše použít, jelikož se v cestách hojně vyskytuje.

Podobnou techniku dělení URI a tvorby příkazů pro shell použijeme rovněž na samém začátku hlavního programu pro úvodní rozdělení zadaného parametru na jednotlivé složky a jejich přiřazení do odpovídajících proměnných shellu (zde ale musíme příkazy vykonat pomocí **`eval`**).

Složitější bude skript pro čtení HTML stránky, zvláště ošetření odkazů rozdělených na více řádek a výskytů více odkazů na jediné řádce. Tím začneme. Pokud se na řádce nevyskytuje menšítko, dál se s ní nebudeme zabývat (smažeme ji). V opačném případě zkontrolujeme, zda máme v pracovním prostoru i většítko, a dokud tomu tak nebude, musíme příkazem *Next* načítat další řádky. Konce řádek, které se nám tím do pracovního prostoru dostanou, budeme nahrazovat mezerami. Jakmile budeme mít text značky kompletní, nahradíme většítko znakem konce řádky a z textu před ním vymažeme vše kromě textu značky. Od té chvíle se budeme věnovat jen první „řádce" v pracovním prostoru (od začátku po znak konce řádky), která bude obsahovat celý text značky.

Začneme testem, zda se jedná o značku „`a`". Pokud tomu tak není, provedeme příkaz *Delete*. Tím se ukončí zpracování značky, první řádka se vymaže z pracovního prostoru, editor se vrátí na začátek skriptu a zkoumá další obsah pracovního prostoru.

V případě, že se naopak jednalo o značku „`a`", musíme postupně prozkoumat její atributy a pokusit se najít atribut „`href`". Podobný postup už jsme dělali v minulém řešení, změna bude v tom, že nyní budeme odmazávat nepotřebné atributy pomocí příkazu *substitute*. Pokud správný atribut nenajdeme, zavoláme *Delete*.

Jakmile atribut „`href`" najdeme, skočíme na návěští `odkaz`, a vyjmeme hodnotu atributu (cíl odkazu). Regulární výraz je poněkud složitější, tak si ho rozeberme. Vyjadřuje, že za (prvním) rovnítkem mohou být mezery a (jedny) uvozovky, potom následuje řetězec bez mezer a uvozovek (to je odkaz) a zbytek textu značky vymažeme. V seznamech v hranatých závorkách nesmíme zapomenout vyloučit znak konce řádky, abychom zajistili, že pracujeme stále jen s první řádkou pracovního prostoru.

Dále budeme zkoumat jednotlivé části URI a budeme přitom neustále používat stejný princip. Pokud určitá části URI vyhovuje, vymažeme ji a odskočíme. Pokud nevyhovuje, provedeme příkaz *Delete* a zpracování odkazu skončíme.

První na řadě bude schéma. Pokud je to „`http:`", odstraníme ho a odskočíme na návěští `schema_ok`. Pokud URI obsahuje jiné schéma, provedeme příkaz *Delete*.

Další kontrolovanou částí URI bude server. Kontrolujeme výskyt dvou lomítek a jména našeho serveru, v pozitivním případě tuto část URI vymažeme a skočíme na `server_ok`, v případě výskytu dvou lomítek s jiným serverem zpracování ukončíme.

Poté porovnáme začátek cesty s obsahem proměnné `koren`, při shodě opět úsek vymažeme a skočíme na `koren_ok`. Pokud se cesta neshoduje, ale začíná lomítkem nebo dvěma tečkami (odkaz vede mimo stahovaný strom) zpracování ukončíme.

Tímto postupem jsme v první řádce pracovního prostoru nechali pouze část URI od kořene stromu, který stahujeme, proto už jen přidáme na začátek lomítko (takto URI očekává funkce `stahni`) a provedeme příkazy *Print* (vytiskne první řádku pracovního prostoru) a *Delete* (ukončí zpracování odkazu).

Vypsané odkazy nakonec přesměrujeme jako vstup do funkce `stahni`.

Řešení (`sed`):

```
stahni() {
sed '
s/#.*//                          ... odstranění fragmentu
s#\([^/]*\)$#\1/\1#              ... připojení kopie názvu stránky na konec
s#/$#/index.html#                ... záměna prázdného jména za index.html
s#\(.*/\)\(.*\)/\(.*\)#\         ... příprava příkazů shellu
[ -d .\1 ] || mkdir -p .\1\
netcat '"$server"' 80 > .\1\3 << KONEC\
GET '"http://$server$koren"'\1\2 HTTP/1.0\
\
KONEC\
#' | sh
}
# zacatek hlavniho programu
eval `echo "$1" | sed -n '
s/#.*//
s#\([^/]*\)$#\1/\1#
s#/$#/index.html#
s#http://\([^/]*\)\(.*\)/\([^/]*\)/\([^/]*\)$#\
server=\1; koren=\2; stranka=\3; soubor=\4#p'`
echo /$stranka | stahni
sed '
/</ !d
: precti                         ... cyklus na kompletaci textu značky
/>/ !{
    N                            ... doplnění řádky
    s/\n/ /
    b precti
}
s/^[^<]*<\([^>]*\)>/\1\          ... separace textu  značky jazyka HTML
/
/^[aA]  */ !D                    ... test typu značky
s/^[aA]  *//
: hledani                        ... cyklus na hledání atributu HREF
    /^[hH][rR][eE][fF] *=/ b odkaz
    s/^[-+.A-Za-z0-9]* *= *[^ \n]* *//
    t hledani
    D
: odkaz                          ... vyjmutí odkazu
s/^[^=]*= *"\{0,1\}\([^ "\n]*\)[^\n]*/\1/
t zacatek                        ... vymazání příznaku úspěšné substituce
: zacatek
```

```
s/^http://
t schema_ok                    ... odskok při správném schématu
/^[-+.A-Za-z0-9]*:/ D          ... ukončení při špatném schématu
: schema_ok
/^#/ D                         ... ukončení pro URI ve tvaru fragmentu
s#^//'"$server"'/#/#
t server_ok                    ... odskok při správném serveru
/^\/\// D                      ... ukončení při špatném serveru
: server_ok
s#^'"$koren"'/##
t koren_ok                     ... odskok při správném kořeni
/^\// D                        ... ukončení při špatném kořeni
/^\.\./ D                      ... ukončení při cestě mimo strom
: koren_ok
s/^/\//                        ... přidání lomítka
P                              ... výpis upraveného odkazu
D
' $soubor | stahni
```

Poznámky (`sed`):

Podobně jako v kapitole 4.24 jsme zde museli pečlivě volit oddělovače řetězců pro příkaz *substitute*. Naštěstí se nám opět podařilo najít vhodný znak („**#**“), který se v URI nemůže vyskytovat (resp. alespoň tam, kde by nám vadil, tj. v cestě). Pokud by se nám to nepovedlo, museli bychom přistoupit k použití nové proměnné s upravenou hodnotou proměnné `koren` (se všemi lomítky uvozenými zpětným lomítkem).

Podmíněný skok na návěští `zacatek`, který bezprostředně předchází samotnému návěští slouží pouze k „vymazání“ příznaku úspěšné substituce, jak jsme o tom mluvili v kapitole 3.17. Skok samotný nic nedělá a k ničemu ho nepotřebujeme. Kdybychom ale příkaz vynechali, následující podmíněný skok na návěští `schema_ok` by se vykonal vždy (i kdyby schéma nebylo „`http:`“). Editor by si totiž stále pamatoval předchozí úspěšné substituce (separaci značky a vyjmutí odkazu).

K volání shellu ve funkci `stahni` jsme přidali ještě jeden parametr „**`-$-`**“. Speciální parametr shellu **`$-`** obsahuje písmenka všech přepínačů, které má nastavené aktuální shell. Pokud třeba nyní náš skript spustíme v „ladicím“ režimu s přepínačem „**`-x`**“, předá se tento přepínač shellu zavolanému ve funkci `stahni` a i on poběží ve stejném režimu.

Rozbor (awk):

Ve větvi **BEGIN** musíme provést rozklad URI zadaného jako parametr („**ARGV[1]**“). Použijeme funkci **split** a rozdělíme text podle lomítek na pole. Jméno serveru bude ve třetím poli (uložíme ho do proměnné `server`) a název stránky v posledním (`stranka`). Pak už budeme moci určit délku (`delka`) celého prefixu až ke kořeni základní stránky (`koren`). Poté zavoláme funkci `stahni`, která stáhne stránku a navíc do proměnné `soubor` uloží jméno výsledného souboru. Tím nahradíme původní hodnotu **ARGV[1]**, takže filtr **awk** začne číst stažený soubor, aniž by mezi parametry bylo jeho jméno.

Obsah stažené stránky budeme číst po jednotlivých značkách, ale oproti kapitole 5.12 poněkud změníme postup. Jako oddělovače záznamů a polí zvolíme „>“, resp. „<“, a tak z každého záznamu zpracujeme jeho druhé pole. Pokud se bude jednat o značku „`a`“, rozdělíme pomocí funkce **split** její text na slova (oddělená bílými znaky) a budeme hledat atribut „`href`“. Opět pro zjednodušení předpokládáme, že název atributu, rovnítko a hodnota tvoří buď jedno slovo, nebo tři, a proto se mezi kroky cyklu o jedno nebo o tři pole posunujeme. Jakmile najdeme pole se správným atributem, musíme nejprve izolovat hodnotu – bude-li délka pole rovna čtyřem („`href`“), nachází se hodnota o dvě pole dál, jinak získáme hodnotu odříznutím prvních pěti znaků z textu pole. Dále musíme z hodnoty vyjmout odkaz. Pokusíme se hodnotu rozdělit podle uvozovek. Pokud byl odkaz mezi uvozovkami, funkce **split** vrátí hodnotu 3 a odkaz bude ve druhém slově. Jinak bude odkazem celá hodnota atributu. Nakonec v této větvi už jen zkontrolujeme a případně vymažeme schéma.

Další postup už je velmi jednoduchý. Odkaz máme uložený v proměnné (`uri`) a v následujících větvích probereme pomocí vzorů jednotlivé případy, přičemž budeme volat funkci `stahni` s vhodně upraveným parametrem.

Ve funkci `stahni` začneme testem, zda zpracovávané URI patří do stahovaného stromu (tj. zda se shoduje jeho začátek s obsahem proměnné `koren`). Pro další činnost budeme opět potřebovat URI rozdělené funkcí **split** na složky. Jakmile budeme znát délku poslední části, spočítáme délku celé cesty od kořene základní stránky k nové stránce (tj. *délka URI* minus *délka prefixu* minus *délka názvu stránky*) a pomocí funkce **substr** cestu z URI vybereme. Nakonec z poslední části URI odřízneme případný fragment (pomocí funkce **sub**) a dostaneme řetězec se jménem souboru, pod nímž stránku uložíme. Pokud by tento řetězec byl prázdný, změníme ho na `index.html`.

Pro testování a případné vytváření adresářové struktury zavoláme funkci **system**.

Pro stahování stránky potřebujeme příkazu **netcat** předat dvě řádky na standardní vstup, proto bude vhodnější použít rouru. Pomocí funkce **sprintf** si připravíme řetězec s textem příkazu **netcat** a tento řetězec zapíšeme na pravou stranu roury u obou příkazů **print**. A také jako parametr funkce **close**, kterou rouru uzavřeme.

Řešení (awk):

```
#!/usr/bin/awk -f
function stahni( uri ) {
    if( substr( uri, 1, delka ) != koren )
        return
    n = split( uri, slova, "/" )
    cesta = substr( uri, delka,
        length( uri ) - delka - length( slova[n] ) )
    sub( /#.*/, "", slova[n] )
    soubor = slova[n] == "" ? "index.html" : slova[n]
    system( "[ -d ." cesta " ] || mkdir -p ." cesta )
    prikaz = sprintf( "netcat %s 80 > '.%s/%s'",
        server, cesta, soubor )
    print "GET " uri " HTTP/1.0" | prikaz
    print ""                    | prikaz
    close( prikaz )
}
BEGIN {
    uri = ARGV[1]             ... zpracování parametru (URI)
    pocet = split( uri, slova, "/" )
    if( pocet < 3 || slova[1] != "http:" ) {
        print "Spatne URI: " uri
        exit
    }
    server = slova[3]
    stranka = slova[pocet]
    delka = length( uri ) - length( stranka )
    koren = substr( uri, 1, delka )
    stahni( uri )             ... stažení stránky do souboru soubor
    ARGV[1] = soubor          ... záměna jména souboru pro čtení
    RS = ">"
    FS = "<"
}
$2 !~ /^[aA][ \n]/ { next }
{
    n = split( $2, pole, " " )
    i = 2
    while( i <= n && pole[i] !~ /^[hH][rR][eE][fF]/ )
        i += pole[i] ~ /^[-+.a-zA-Z0-9]*$/ ? 3 : 1
    if( i > n )
        next
    href = length( pole[i] ) == 4 ? \
        pole[i + 2] : substr( pole[i], 6 )
    uri = split( href, slova, "\"" ) == 3 ? slova[2] : href
```

```
    if( uri ~ /^http:/ )
        uri = substr( uri, 6 );
    else if( uri ~ /^[-+.a-zA-Z0-9]*:/ )
        next
}
uri ~ /^#/    { next }
uri ~ /^\.\./ { next }
uri ~ /^\/\// { stahni( "http:" uri ); next }
uri ~ /^\//   { stahni( "http://" server uri ); next }
{ stahni( koren uri ) }
```

Poznámky (awk):

Abychom nemuseli opustit prostředí **awk**, sáhli jsme tentokrát k řadě rozšíření. Hned v příští kapitole to zase trochu napravíme.

Ve funkci `stahni` jsme použili funkci **sub**, o níž jsme se zmiňovali v kapitole 5.10. Sice bychom si opět vystačili se standardní funkcí **split**, ale na tomto místě je volání funkce **sub** opravdu jednodušší. Tentokrát dostává funkce ještě třetí parametr a tím je proměnná, v níž chceme substituci provést. Pozor, funkce nevrací výsledek substituce, jak by se bylo možno domnívat. Chceme-li upravit hodnotu nějaké proměnné a výsledek uložit do jiné proměnné, je třeba si nejprve celou hodnotu zkopírovat do nové proměnné a na ni teprve zavolat funkci **sub**.

Ze syntaktické kuchyně jazyka **awk** jsme také vytáhli *podmíněný výraz*. Má tvar „*podmínka* **?** *výraz1* **:** *výraz2*“ a jeho podstatou je, že se vyhodnotí podmínka, a je-li platná, vypočítá se a použije *výraz1*, zatímco v opačném případě se vypočítá a použije *výraz2*. Je to rovněž dědictví po jazyce C a stejně jako tam je nutné ho užívat uvážlivě. Pokud se podmíněný výraz použije tak jako v našem příkladu, lze s tím jistě souhlasit. Ale asi si dovedete představit, že když do sebe zamícháte několik podmíněných výrazů, snadno vyrobíte něco, o čem bude málokdo (včetně vás) schopen říci, co to vlastně dělá. A navíc podmíněný výraz patří také mezi rozšíření jazyka **awk**.

K vytvoření textu příkazu pro použití v rouře jsme použili funkci **sprintf**. Dělá tutéž práci jako příkaz **printf**, ale výsledný text pouze *vrací* jako řetězec. V našem příkladu jsme sice žádné složité formátování nepoužili, ale i tak je vytvoření správného řetězce tímto způsobem asi o něco čitelnější než:

```
prikaz = "netcat " server " 80 > '." cesta "/" soubor "'"
```

Namísto dvou příkazů **print** jsme mohli použít jeden **printf**:

```
printf "GET %s HTTP/1.0\n\n", uri | prikaz
```

Udělali jsme to pomocí posloupnosti příkazů jenom proto, abychom ukázali, jak by se muselo postupovat, pokud by skladba řádek posílaných do roury byla ještě složitější.

Pokud náš program zapíšete do souboru (a nastavíte mu práva), bude ho možné volat přímo jeho jménem, nebudete muset explicitně vyvolávat program **awk**. Důvodem je první řádka skriptu („**#!**...“). Stejný princip jsme viděli už v kapitole 4.34. Shell z řádky zjistí, jakým interpretem má soubor zpracovat, jaké přepínače má nastavit, a daný program zavolá. Trochu nepříjemné ale je, že cesta uvedená za vykřičníkem musí být absolutní, takže pokud na jiném stroji bude **awk** jinde, skript nepoběží.

Tento princip funguje pro libovolný interpret, který má *komentáře* ve stejném formátu jako shell, a to jazyk **awk** má (od znaku „**#**“ až do konce řádky).

Na řádce „**#!**...“ je nutno uvést ještě přepínač „**-f**“. Shell totiž interpret spouští tak, že do textu příkazové řádky přidá jméno souboru. Výsledná řádka bude mít tedy tvar:

awk -f *jméno_souboru*

Přepínač „**-f**“ slouží při volání **awk** pro zadání jména souboru s programem, pokud program nechceme psát jako první parametr na příkazové řádce. Funguje to stejně jako v editoru **sed** a stejně jako tam se tento parametr může opakovat, což umožňuje vytvořit si knihovnu pomocných funkcí a používat ji v dalších programech, například takto:

```
awk -f knihovna.awk -f program.awk soubor1 soubor2
```

Nevýhodou umístění programu do souboru je to, že si nemůžete nechat kód textově upravit podle aktuálních hodnot proměnných v shellu, jako jsme to dělali v kapitole 5.6. A pokud bude volání **awk** součástí nějakého skriptu, budete muset udržovat kód uložený ve dvou souborech (skript pro shell a program pro **awk**) místo jednoho.

6.2. Stahování do hloubky

Zadání:

Upravte program z minulé kapitoly, aby ve stahování odkazů pokračoval na dalších stránkách, a to až do hloubky zadané druhým parametrem.

Rozbor:

Pokusme se z řešení předvedených v minulé kapitole odvodit, které nástroje (shell, **`sed`**, **`awk`** a příp. jiné) se hodí pro kterou podúlohu.

Vlastní stahování stránek vždycky nakonec skončilo u shellu, není tedy důvod to tak nenechat i zde. Uděláme jen dvě změny. Rozdělení URI na části přece jen šlo nejlépe editoru, a proto s jeho pomocí budeme počítat i teď. A na rozdíl od minulé kapitoly budeme kontrolovat výsledek stahování. Získáme tím řadu výhod. Jednak budeme schopni reagovat na neúspěšné pokusy o stažení stránek (doposud jsme při chybě prostě uložili text odpovědi do souboru, jako by to byl správný text stránky). Za druhé budeme moci odstranit ze stahovaných souborů úvodní část s odpovědí serveru a hlavičkami stránky. A hlavně zvládneme i případy, kdy v odkazu na adresář bude chybět koncové lomítko. V takovém případě totiž server vrátí odpověď zhruba tohoto tvaru:

```
HTTP/1.1 301 Moved Permanently
... začátek hlaviček odpovědi
Location: správné kompletní URI
... začátek hlaviček odpovědi

... text chybové stránky
```

Když tuto odpověď dokážeme zpracovat a znovu zavolat **`netcat`** s opraveným URI, nebudou nám chybějící lomítka vadit.

Dalším důležitým úkolem je přečtení stránek a vyhledání odkazů. Vyzkoušeli jsme si různé varianty. Shellové řešení ponechme stranou, složitost HTML je poněkud nad jeho síly. Zbylá dvě řešení na tom byla se složitostí zhruba stejně, ale řešení v jazyce **`awk`** je asi přece jen běžným programátorským zvyklostem bližší a díky tomu snáze čitelné. Na editoru necháme drobnější práce, kde převažuje editační složitost nad složitostí logickou (dělení URI na části, náhrada prázdného jména souboru za `index.html` apod.).

Jedinou zásadní výhodou pro editor je krok, kdy z URI ve tvaru relativní cesty budeme tvořit URI s cestou absolutní. Totiž pokud například na stránce `/a/b/xy.html` bude odkaz „`http:../z/`“, budeme muset zjistit skutečnou absolutní cestu pro tento odkaz. Složíme cestu k adresáři („`/a/b/`“) a text vlastního odkazu, dostaneme cestu „`/a/b/../z/`“ a tu budeme muset „normalizovat“ na tvar „`/a/z/`“. Přesně to jsme už dělali v kapitole 3.10, takže řešení v editoru už máme hotové. To je sice určitá výhoda,

ale přesto raději nebudeme kód tříštit a dáme přednost čtení v jazyce **awk**. Normalizaci budeme sice muset napsat znova, ale není to zase tak složité.

Zbývá vyřešit, jak celé rekurzivní stahování řídit. Po každém stažení stránky (anebo určité skupiny stránek) musí přijít fáze jejich čtení a vyhledávání odkazů, než budeme mít k dispozici další odkaz nebo odkazy ke stažení.

Zároveň budeme muset kontrolovat, zda jsme nový odkaz už nestahovali (odkazy mohou vytvářet cyklus). Jedno z možných řešení nabízí jazyk **awk** a jeho asociativní pole – URI každé stažené stránky bychom ukládali do pole a u každého nalezeného URI bychom testovali, zda ho v poli už nemáme. Háček je v tom, že výhody tohoto přístupu bychom plně využili jen tehdy, pokud bychom celé řešení napsali v jazyce **awk**. Pokud budeme chtít kombinovat výhody různých nástrojů, bude lepší nechat jejich volání na shellu a v takovém případě už na něm můžeme nechat i řízení celé operace. Oproti řešení kompletně pomocí **awk** tím sice dojde k mírnému nárůstu počtu procesů, ale vzhledem k rozsahu naší úlohy si to za cenu lepší přenositelnosti a čitelnějšího kódu jistě můžeme dovolit. Navíc zásadní vliv na rychlost programu budou mít stejně přenosy dat po síti.

Pojďme si nyní rozebrat možnosti, jak na to. Potřebujeme nějaké místo pro uložení adres stránek, které už jsme stáhli, a stránek připravených ke čtení. V úvahu přicházejí dvě varianty (proměnná nebo soubor), které se od sebe ovšem nijak zásadně neliší. Vybereme řešení se souborem, protože je nepatrně čitelnější. Vytvoříme soubor, z nějž budeme číst stránky, které máme stáhnout, a do nějž budeme současně přidávat nové odkazy nalezené v průběhu práce. Mohlo by to vypadat zhruba takto:

```
while read dalsi; do
    stahni $dalsi | precti >> ke_cteni
done < ke_cteni
```

Ale podobná řešení, kdy máme soubor otevřený pro čtení a zároveň do něj zapisujeme, představují určité riziko, především z hlediska přenositelnosti. I kdybyste tohle řešení na jednom počítači odladili, nebudete mít záruku, že se na jiném nebude chovat jinak.

Navíc v našem případě by se v takovéhle smyčce obtížněji kontrolovalo omezení na požadovanou hloubku zanoření. Jednodušší bude mít soubory tři – z jednoho souboru budeme URI určená ke stažení číst (`stary`), do druhého (`novy`) budeme nově nalezená zapisovat a třetí (`kontrola`) bude obsahovat všechny navštívené stránky pro kontrolu cyklických odkazů. Hlavní smyčku programu pak pustíme tolikrát, kolik „pater“ odkazů máme za úkol projít podle hodnoty druhého parametru skriptu, a na konci každého kroku soubory `stary` a `novy` vyměníme.

Použitý mechanismus se dá velmi snadno „nastartovat“: URI z parametru zapíšeme (jako jediný záznam) do seznamu pro první krok cyklu (do souboru `stary`).

Přesněji řečeno, skutečná jména souborů budou `/tmp/wget.$$.stary` atd., abychom se vyhnuli problémům s kolizemi jmen.

Hlavní rysy programu bychom tedy měli a můžeme se věnovat detailům.

Už jsme se zmiňovali, že dělení URI na složky se nejlépe dělalo pomocí editoru.

Úvodní rozdělení zadaného URI, ze kterého vzejde jméno serveru a cesta ke kořeni základní stránky, proto uděláme téměř stejně jako v editorové variantě v minulé kapitole – editor **`sed`** připraví pro shell příkazy pro nastavení hodnot proměnných a ty provedeme pomocí příkazu **`eval`**. V rámci zkvalitňování služeb pouze navíc ošetříme případ, že v zadaném URI nebylo vůbec lomítko za jménem serveru.

Rozdělení odkazu v parametru funkce `stahni` vyřešíme podobně. Nejprve oddělíme prefix – známe jeho délku, takže jen použijeme přesný počet opakování („**`\{`***délka***`\}`**“) libovolného znaku („**`.`**“). Vyhneme se tím problémům s ošetřováním lomítek nebo metaznaků uvnitř řetězce s prefixem. Pak odstraníme z URI případný fragment. Třetí příkaz nahrazuje v textu URI konstrukt, který se používá pro zápis mezery („`%20`“), skutečnou mezerou. Správně bychom asi měli ošetřit tento konstrukt v plné obecnosti (lze takto zapsat jakýkoliv znak, pokud se za procento napíše jeho hexadecimální hodnota), ale ošetříme pouze mezery – jednak je mezera asi nejčastější případ a jednak jsme chtěli vyzkoušet, zda nám mezery ve jménech adresářů a souborů nebudou dělat nějakou neplechu. Předposlední příkaz *substitute* rozdělí URI na cestu a jméno stránky. Obě části přitom oddělí nejspolehlivějším oddělovačem – znakem konce řádky. Posledním krokem je doplnění jména souboru `index.html`, pokud v URI jméno stránky chybělo (neboli pokud druhá řádka pracovního prostoru je prázdná).

Výstup z editoru potřebujeme přečíst dvěma příkazy **`read`**, a proto ho přesměrujeme do složeného příkazu, který bude obsahovat zbytek kódu funkce. Nejprve zkontrolujeme „zacyklení“ odkazů. Musíme projít soubor `/tmp/wget.$$.kontrola`, zda se v něm nové URI („**`$cesta$soubor`**“) už nevyskytuje. Opět se budeme snažit vyhnout se problémům s metaznaky regulárních výrazů uvnitř URI (přinejmenším tečky tam jistě budou). Proto k vyhledávání použijeme příkaz **`grep`** s přepínačem „**`-F`**“ (*fixed string*). Díky němu se nebude zadaný vzor chápat jako regulární výraz, ale jako přesný řetězec, který se má hledat. Jenže tím jsme přišli o možnost říci (pomocí metaznaků pro ukotvení regulárních výrazů, „**`^`**“ a „**`$`**“), že požadujeme shodu na celé délce řádky. Proto musíme přidat ještě přepínač „**`-x`**“ (*exact*), který má přesně tento význam.

Odpověď serveru musíme podrobit pečlivému zkoumání. Opět by se nám hodilo použít několikrát příkaz **`read`**, a tak ještě jednou sáhneme k možnosti uzavřít další část kódu do složeného příkazu. Je to už naposledy, a tak to snad čitelnost programu nijak zásadně nezhorší.

První příkaz **read** přečte tzv. *stavovou řádku*, která obsahuje (jako druhé slovo) číselný kód odpovědi. Pro nás je důležité rozlišit tři případy:

200 Tento kód znamená, že server našel požadovanou stránku a její text je součástí odpovědi. V tom případě uložíme do souboru zbytek vstupu (resp. ještě z něj vymažeme hlavičky – od začátku po prázdnou řádku). Pokud má soubor příponu `htm` nebo `html`, budeme ho dále zpracovávat. Problém je ale v tom, že název stránky jsme z URI odvodili až ve funkci `stahni`, a to ve složeném příkazu za rourou a tedy v jiném procesu. Nemůžeme proto použít zdejší proměnné k tomu, abychom po návratu z funkce zjistili, jaký soubor máme vlastně číst. Proto celý text souboru na konci funkce vypíšeme na výstup a volající si ho převezme na svém standardním vstupu (rourou).

301 Tento kód znamená, že server sice zná požadované URI, ale správné URI pro danou stránku je ve skutečnosti jiné. A toto správné URI také server v odpovědi pošle, a to v hodnotě hlavičky `Location`. Spustíme tedy cyklus, kterým hlavičku najdeme, a pokud URI patří do našeho stromu, s touto novou hodnotou znova zavoláme funkci `stahni`.

Všechny ostatní kódy budeme považovat za chyby a na standardní chybový výstup ohlásíme URI a kód chyby.

Jádro programu v jazyce **awk** , který čte webovou stránku, bude stejné jako v minulé kapitole. Podstatný rozdíl je ale v tom, že při zpracování relativních odkazů potřebujeme znát cestu k aktuální stránce. Jenže s předáním jakýchkoliv informací z funkce `stahni` bude problém – to už víme. Proto bude nejsnazší i zde využít rouru. Funkce `stahni` ještě před vlastním textem stránky vypíše na standardní výstup cestu k adresáři, kde stránka leží, a program si cestu na první řádce (díky testu „**NR == 1**“) přečte a uloží do proměnné `cesta`. Poté si nastaví správné oddělovače záznamů a polí pro čtení stránky.

Lišit se bude také závěr skriptu, protože potřebujeme upravovat relativní odkazy. Odkazy začínající dvěma lomítky zkontrolujeme, zda obsahují správné jméno serveru, a toto jméno odstraníme. Před relativní odkazy naopak musíme doplnit obsah proměnné `cesta`. Nyní provedeme normalizaci celé cesty. Rozdělíme si ji na slova podle lomítek do pole `polozky` a budeme toto pole postupně procházet a přepisovat. V proměnné `posun` budeme uchovávat rozdíl mezi indexem prvku pole, který právě čteme, a indexem prvku, do něhož zapisujeme. Na začátku práce bude tento rozdíl nulový (tj. k žádnému přepisování nedochází). Při nalezení položky „.“ zvýšíme `posun` o jedničku, další zapisované položky se budou jakoby o jedničku „zpožďovat“, čímž položku „.“ ve výsledku ze seznamu vynecháme. Úplně stejně (jen s posunem o dvě položky) vyřešíme i výskyty položky „..“. Po skončení cyklu vypíšeme upravený obsah pole na výstup pomocí cyklu s příkazem **printf**.

Řešení:

```
stahni() {
    echo "$1" | sed '
        s/^.\{'${#prefix}'\}//
        s/#.*//
        s/%20/ /g
        s/\(.*\/\)*/\1\
/
        /\n$/s/$/index.html/
    ' | {
        read cesta
        read soubor
        grep -qxF "$cesta$soubor" /tmp/wget.$$.kontrola &&
            return 1
        echo "$cesta$soubor" >> /tmp/wget.$$.kontrola

        [ -d "./$cesta" ] || mkdir -p "./$cesta"
        netcat $server 80 << EOF | tr -d '\r' | {
GET $1 HTTP/1.0

EOF
            read x rc y
            case $rc in
            200 )
                sed '1,/^$/d' > "$cesta$soubor"
                case $soubor in
                *.[hH][tT][mM] | *.[hH][tT][mM][lL] )
                    echo "$koren/$cesta"
                    cat "$cesta$soubor";;
                esac;;
            301 )
                while read nazev hodnota; do
                    case "$nazev$hodnota" in
                    "Location:$prefix"* )
                        stahni "$hodnota"
                        return;;
                    esac
                done;;
            * )
                echo "Stranka '$1' se nestahla ($rc)" >&2;;
            esac
        }
    }
}
```

```
precti() {
    awk '
NR == 1 {
    cesta = $0
    server = "'$server'"
    delka = length( server )
    RS = ">"
    FS = "<"
    next
}
$2 !~ /^[aA][ \n]/ { next }
{
    n = split( $2, pole, " " )
    i = 2
    while( i <= n && pole[i] !~ /^[hH][rR][eE][fF]/ )
        i += pole[i] ~ /^[-+.a-zA-Z0-9]*$/ ? 3 : 1
    if( i > n )
        next
    href = length( pole[i] ) == 4 ? \
        pole[i + 2] : substr( pole[i], 6 )
    uri = split( href, slova, "\"" ) == 3 ? slova[2] : href
    if( uri ~ /^http:/ )
        uri = substr( uri, 6 );
    else if( uri ~ /^[-+.a-zA-Z0-9]*:/ )
        next

    if( uri ~ /^#/ )
        next
    if( uri ~ /^\/\// ) {
        if( substr( uri "/", 3, delka + 1 ) == server "/" )
            uri = substr( uri, delka + 3 )
        else
            next
    }
    if( uri !~ /^\// )
        uri = cesta uri

    n = split( uri, polozky, "/" )
    posun = 0
    for( i = 2; i <= n; i++ ) {
        if( polozky[i] == "." )
            posun++
        else
        if( polozky[i] == ".." )
            posun += 2
```

```
        else
        if( posun < i - 1 )
            polozky[i - posun] = polozky[i]
        else
            next
    }
    printf "http://%s", server
    for( i = 2; i <= n - posun; i++ )
        printf "/%s", polozky[i]
    printf "\n"
}' | grep -F "$prefix"
}

zpracuj() {
    while read uri; do
        stahni "$uri" | precti
    done
}

# zacatek hlavniho programu
eval `echo "$1" | sed -n '
s/#.*//
s#http://\([^/]*\)\(.*\)/\([^/]*\)$#\
server="\1"; koren="\2"; stranka="\3"#p
s#http://\([^/]*\)$#\
server="\1"; koren=; stranka=#p'`
[ -z "$server" ] && echo "`basename $0`: Spatne URI" && exit
prefix="http://$server$koren/"

if ! [ "$2" -gt 0 ] 2> /dev/null; then
    echo "`basename $0`: Spatna hloubka"
    exit
fi

trap 'rm -f /tmp/wget.$$.*; exit' 0 2 3 15
: > /tmp/wget.$$.kontrola
hloubka=
echo "$prefix$stranka" > /tmp/wget.$$.stary

while [ ${#hloubka} -le $2 -a -s /tmp/wget.$$.stary ]; do
    hloubka=@$hloubka
    zpracuj < /tmp/wget.$$.stary > /tmp/wget.$$.novy
    mv /tmp/wget.$$.novy /tmp/wget.$$.stary
done
```

Poznámky:

K otestování, zda druhý parametr obsahuje kladné číslo, jsme použili fintu. Nechali jsme rovnou příkaz **`test`** zkontrolovat, zda hodnota **`$2`** je větší než nula, aniž bychom napřed zjišťovali, zda tento řetězec vůbec obsahuje číslo. Pokud tam číslo nebude, příkaz **`test`** skončí chybou stejně jako v případě záporného čísla. Z hlediska (nenulovosti) návratové hodnoty se tedy budou oba případy chovat úplně stejně a nemusíme si lámat hlavu s jejich rozlišováním. Jenom jsme museli zahodit chybovou hlášku.

Napsali jsme cyklus, který postupně připisuje řádky k souboru (použili jsme operátor „**`>>`**“). Před začátkem cyklu musíme ovšem takový soubor „vyprázdnit“. Už jsme si o tom povídali – nejjednodušší je soubor prostě smazat. Jenomže u souboru `kontrola` musíme použít jiné řešení, protože na neexistující soubor bychom nemohli zavolat příkaz **`grep`**. Tady musíme soubor opravdu vyprázdnit (přesměrujeme do něj výstup příkazu „**`:`**“).

Použili jsme rovněž nový přepínač „**`-d`**“ (*delete*) příkazu **`tr`**. Způsobí to, že znaky uvedené v (jediné) tabulce se vymažou. Z řádek odpovědi jsme tak vymazali znaky CR, které server zapisuje na konec řádky před znak LF. K zadání znaku CR jsme použili escape-sekvenci „**`\r`**“, kterou už také známe.

Při analýze jsme poněkud příkře odsoudili shell a jeho možnosti při čtení textu stránek. Pravdou je, že kdybychom nelpěli na prostředcích shellu tak sveřepě a stránku nejprve pomocí několika utilit upravili k obrazu svému, bylo by i čtení v shellu poměrně snesitelné. Například bychom mohli všechny značky vypsat do samostatných řádek:

```
tr '>\n' '\n ' | cut -s -d'<' -f2
```

Nic to ovšem nemění na faktu, že editor či filtr **`awk`** mají ve zpracování textu navrch.

V minulé kapitole jsme slíbili přenositelnou verzi skriptu v jazyce **`awk`**. Nedodrželi jsme to jen v jediném bodě, a tím je použití podmíněných výrazů. Udělali jsme to ovšem pouze proto, aby se vám snáze navazovalo na řešení z minulé kapitoly. Jistě si sami umíte tyto konstrukty nahradit obyčejným příkazem **`if`**.

6.3. WWW News

Zadání:

Napište program, který spravuje seznam novinek na webovém serveru.

Předpokládejte, že máte vyhrazený adresář a v něm jednak hlavní stránku (soubor `index.html`) a jednak soubory se stránkami obsahujícími texty novinek (návrh formátu jmen souborů je součástí řešení). Indexový soubor má tvar:

```
...                                    text neobsahující značku <ol>
<ol>                                   začátek seznamu novinek
<li>den.měsíc. <a href=soubor_s_textem>titulek novinky</a></li>
...                                    další novinky
</ol>                                  konec seznamu novinek
...                                    zbytek textu stránky
```

Správa novinek se odehrává pomocí elektronické pošty. Předpokládejte, že pro každý doručený dopis se spustí váš skript a dostane text dopisu na svém standardním vstupu. Skript akceptuje pouze dopisy od odesilatelů uvedených v souboru `spravci`. Rozeznává dva druhy dopisů:

- Žádost o přidání novinky:

```
...začátek hlavičky dopisu
From: odesilatel
Subject: titulek novinky
...další řádky hlavičky
                                       prázdná řádka oddělující hlavičku a tělo
...text stránky s novinkou
```

- Žádost o smazání novinky číslo *n* z určitého data:

```
...začátek hlavičky dopisu
From: odesilatel
Subject: SMAZAT den.měsíc. n
...další řádky hlavičky
                                       prázdná řádka oddělující hlavičku a tělo
...libovolný text
```

Pokud je dopis v pořádku, skript opraví indexový soubor, uloží nebo smaže soubor s textem zprávy a pošle odesilateli odpověď.

Řádky s odesilatelem a předmětem dopisu mohou přijít v libovolném pořadí a nikoliv pouze bezprostředně za sebou. Pokud některá z nich chybí, dopis se ignoruje.

Skript musí fungovat korektně, i když přijdou dva dopisy současně a poběží dvě instance skriptu paralelně. Je třeba navrhnout ochranu před paralelním přístupem.

Rozbor:

Prvním krokem analýzy musí být rozhodnutí o struktuře a uložení potřebných dat, v našem případě to znamená především způsob, jakým budeme pojmenovávat soubory se stránkami. Systém jmen musí umožnit snadno zaručit jejich jednoznačnost a musí se nám s ním dobře pracovat. Požadavek jedinečnosti jména dobře splňuje například číselná řada. Splňuje zároveň i druhou podmínku – při hledání nového jména stačí vybrat současné maximum (můžeme ho mít třeba uložené v souboru `maximum`) a zvýšit ho o jedničku. Jako formát jmen souborů tedy zvolíme „`stranka.`*číslo*`.html`“.

Dalším krokem bude volba vhodných programátorských prostředků pro jednotlivé kroky. Využijeme přitom zkušeností získaných z analýzy v předchozí kapitole.

Pro čtení dopisu se určitě hodí jazyk **`awk`**. V našem zjednodušeném zadání se to tolik neprojeví, ale reálně fungující systém by z dopisu určitě potřeboval načerpat mnohem více informací, a pak by se výhoda tohoto řešení projevila podstatně.

Manipulace se soubory na základě takto získaných dat je naopak práce spíše pro shell. Vytvoření nového souboru lze provést prostě příkazem **`cat`**, přičemž pořadové číslo pro jméno souboru zjistíme z pomocného souboru `maximum`. Mazání souboru se starou stránkou zařídí příkaz **`rm`**, pouze pro něj nejprve musíme ze souboru `index.html` zjistit její jméno (pomocí příkazů **`grep`**, **`sed`** a **`read`**).

Editace indexového souboru, jak už název napovídá, bude úkolem pro editor **`ed`**. Ovšem i jemu musíme nejprve připravit potřebná data – pro přidání nové stránky to bude její titulek, jméno souboru a aktuální datum (to získáme příkazem **`date`**), pro smazání stačí například zjistit pořadové číslo rušené řádky v souboru `index.html`.

Posledním úkolem je ochrana před paralelním přístupem. Indexový soubor a soubor `maximum` bude třeba „zamykat“. V shellu je nejsnazším způsobem použít test existence nějakého souboru (tzv. *zámku*). Pokud soubor existuje, někdo v adresáři pracuje, jinak je adresář volný. Problém je v tom, že operace testu a případného nastavení zámku musí být *nepřerušitelná*. Jinak by se mohlo stát, že jeden proces provede test zámku, zjistí, že soubor neexistuje, a chystá se ho vytvořit. Mezitím ho operační systém přeruší a spustí druhý proces. Ten také provede test zámku, také zjistí, že má cestu volnou, a zámek vytvoří. Pokud v té chvíli operační systém zase pustí první proces, ten už má kontrolu zámku za sebou, takže také vytvoří (znova, to ale netuší) zámek a dostáváme se do situace, které jsme se poctivě snažili vyhnout. Nejlepší je vynechat fázi testování a vymyslet takový princip zámku, kdy pokus o jeho vytvoření selže, pokud zámek už existuje. Jednou z možností je zapnout přepínačem „**`-C`**“ režim *noclobber*, ve kterém nelze pomocí přesměrování přepsat existující soubor, a pokusit se přesměrovat nějaký výstup do souboru `zamek` (jinou variantou je příkaz **`mkdir`**). To se může podařit současně jen jedinému procesu, a dokud se vytvoření souboru nezdaří, budeme pokusy opakovat. Mezi jednotlivými pokusy budeme muset náš proces na nějakou dobu

suspendovat (uspat) příkazem **sleep**, abychom umožnili procesu, na nějž čekáme, dokončit svou práci a odemknout zámek. Celkový počet pokusů budeme muset nějak omezit. Mohlo by se totiž stát, že by zámek mohl být trvale zamčený – pokud by proces, který zámek zamknul, nějak zhavaroval. Abychom my sami podobné riziko omezili, po celou dobu úprav, kdy máme zámek zamčený, zakážeme přerušení našeho skriptu (zablokujeme signály).

Po této globální rozvaze můžeme přistoupit k detailům.

Elektronický dopis se skládá z hlaviček a těla. Program pro **awk** musí při čtení dopisu tyto dva stavy (čtení hlaviček a čtení těla) rozlišit. K tomu použijeme proměnnou `telo`. Ve fázi čtení hlaviček hledáme řádky „`From: ...`“ a „`Subject: ...`“. E-mailovou adresu odesilatele si uložíme do proměnné `odesilatel`. Pokud řádka s předmětem začíná slovem „`SMAZAT`“, uložíme si ho do proměnné `ukol` jako název operace a do proměnné `param` přiřadíme třetí a čtvrté pole řádky (bude tedy obsahovat text „*den*.*měsíc*.*číslo*“). Jakýkoliv jiný předmět znamená žádost o přidání stránky, do proměnné `ukol` dáme hodnotu „`PRIDAT`“ a do proměnné `param` uložíme celý text řádky (slova „`Subject:`“ se později snadno zbavíme). Hranicí hlaviček a těla dopisu je prázdná řádka. Jakmile na ni narazíme, vypíšeme hodnoty proměnných `ukol`, `odesilatel` a `param` a pak buď práci **awk** ukončíme (při mazání), nebo změníme stavovou proměnnou `telo`. Ve fázi čtení těla už pouze opisujeme řádky.

Musíme si rozmyslet, na kterém místě a jakými prostředky budeme provádět kontrolu odesilatele. Můžeme jeho adresu vypsat a následně ji kontrolovat v shellu proti obsahu souboru `spravci`. Ale také můžeme tento soubor přidat jako druhý (resp. spíše první) vstup programu, který bude číst dopis, uložit seznam správců do asociativního pole a rovnou kontrolovat, zda je odesilatel v pořádku.

Informace získané programem na čtení dopisu je třeba předat do shellu. Vzhledem k tomu, že struktura našeho programu není složitá, můžeme zbytek programu za voláním **awk** vykonat ve složeném příkazu. V tom případě je nejsnazší vypsat pevnou skladbu dat na pevný počet řádek a ty v shellu přečíst sérií příkazů **read**.

První příkaz **read** přečte kód operace. Otestujeme nejprve, zda náš program vůbec nějakou operaci vypsal – pokud ne, znamená to, že se dopisem nemáme dále zabývat.

Pokud operací bude „`PRIDAT`“, musí druhý příkaz **read** načíst řádku s předmětem dopisu. Dáme mu tedy dva parametry, do první proměnné se načte nepotřebný text „`Subject:`“ a do druhé titulek stránky. Poté musíme zjistit ze souboru `maximum` nejvyšší pořadové číslo stránky, zvýšit ho a zapsat ho do souboru zpátky. Na tomto místě není třeba příliš optimalizovat, postačí kombinace příkazů **read**, **expr** a **echo**. Soubor s textem stránky pak vytvoříme příkazem **cat**. Jeho výstup přesměrujeme do souboru se správným jménem a jeho vstupem bude zbytek textu v rouře – vypsali jsme do ní totiž přesně dvě řádky (ty už jsme přečetli) následované tělem dopisu.

Pokud bude operací „SMAZAT“, necháme druhý příkaz **read** číst s oddělovačem nastaveným na tečku, abychom jednoduše dostali hodnoty rovnou do proměnných **den**, **mes** a **cislo**. Pak musíme najít požadovanou řádku v indexovém souboru a zjistit její číslo. Pro hledání se nejlépe hodí příkaz **grep**, a to s přepínačem „**-n**“, který zařídí výpis včetně čísla řádky. Výstup příkazu budeme filtrovat editorem **sed** a ten vybere správnou řádku podle pořadí (pokud tam taková bude). Tuto řádku načteme příkazem **read** s oddělovačem polí výhodně nastaveným na tečku a dvojtečku, abychom snadno načetli jak pořadové číslo řádky v indexovém souboru (první slovo), tak číslo souboru, který máme mazat (páté slovo). Pro jistotu si ukažme, jak bude řádka vypadat:

číslo_řádky:<li>*den*.*měsíc*. <a href=stranka.*číslo_souboru*.html>*titulek...*

Zámek budeme zamykat a odemykat na několika místech kódu, vytvoříme si tedy funkce `zamkni` a `odemkni`. Pro ověřování stavu zámku použijeme tentokrát jiný cyklus, a to **until**. Chová se podobně jako cyklus **while**, jen vyhodnocuje podmínku obráceně, tj. probíhá tak dlouho, dokud se volaný příkaz (u nás vytvoření souboru) nepodaří.

Ještě potřebujeme pečlivě rozmyslet, kdy přesně se budou volat příkazy **trap** pro zablokování a odblokování signálů. Zablokovat signály musíme bezprostředně poté, co zámek zamkneme. Jenže i v případě, že příkaz **trap** napíšeme bezprostředně za příkaz **echo**, který soubor vytváří, nebude to dostatečně „bezprostředně“. Zablokování tedy musí předcházet pokusu o vytvoření souboru. Pro přehlednost si na kombinaci příkazů **trap** a **echo** napíšeme ještě jednu funkci (`cvak`). A v případě, že „cvaknutí“ neprojde, signály hned odblokujeme (tady drobné zpoždění nevadí).

Odesílání elektronických dopisů se v UNIXu zařídí jednoduše pomocí utility **mailx** (nebude-li váš systém znát toto jméno, zkuste třeba samotné „**mail**“). Na vstupu má text dopisu, v parametrech adresáty a případný předmět zadáme přepínačem „**-s**“.

Řešení:

```
zprava() {
    echo "$@" | mailx -s "WWW News" $odesilatel
}
odemkni() {
    rm zamek                    ... zrušení zámku
    trap - 2 3 15               ... odblokování signálů
}
cvak() {
    trap "" 2 3 15              ... zablokování signálů
    echo $$ > zamek             ... pokus o zamčení zámku
}
```

```
zamkni() {
    set -C                      ... nastavení režimu noclobber
    pokusy=                     ... počitadlo pokusů
    until cvak 2> /dev/null; do
        trap - 2 3 15           ... odblokování signálů pro zbytek těla
        if [ ${#pokusy} -ge 5 ]; then
            echo "Databaze je uzamcena" |
                mailx -s "WWW News Problem" root
            zprava "Zaslete pokyn pozdeji"
            exit
        fi
        sleep 1                 ... uspání procesu
        pokusy=@$pokusy
    done
    set +C                      ... vypnutí režimu noclobber
}
awk '
BEGIN { telo = 0; odesilatel = ""; ukol = "CHYBA" }
FILENAME == "spravci" { spravci[$0] = 1; next }
telo == 1 { print; next }
/^From: / { odesilatel = $2; next }
/^Subject: SMAZAT / {
    ukol = $2
    param = $3 $4               ... parametry: den.měsíc.číslo
    next
}
/^Subject: / {
    ukol = "PRIDAT"
    param = $0                  ... parametry: hlavička titulek
    next
}
/^$/ {                          ... konec hlavičky dopisu
    if( ! spravci[odesilatel] )
        exit                    ... konec (chybějící nebo špatný odesilatel)
    print ukol, odesilatel
    print param
    if( ukol != "PRIDAT" )
        exit
    telo = 1
}' spravci - | {
    read ukol odesilatel ||
        exit                    ... konec (chybějící nebo špatný odesilatel)
```

```
    case $ukol in
    PRIDAT )
        zamkni
        read x tit              ... načtení titulku z řádky Subject:
        read max < maximum
        max=`expr $max + 1`     ... nová hodnota maxima pro číslo stránky
        echo $max > maximum
        href=stranka.$max.html
        cat > $href             ... dočtení a uložení textu stránky
        ed -s index.html <<- KONEC
            /<ol>/a             ... vložení nové řádky za první řádku <OL>
            <li>`date +%d.%m.` <a href=$href>$tit</a></li>
            .
            w
            KONEC
        odemkni
        zprava "Novinka stranka.$max.html pridana";;
    SMAZAT )
        zamkni
        IFS=. read den mes n
        grep -n "^<li>0*$den\.0*$mes. <a href=" index.html |
            sed -n ${n}p |
            if IFS=:. read radka d m x cislo x; then
                ed -s index.html <<- KONEC
                    ${radka}d
                    w
                    KONEC
                rm stranka.$cislo.html
                odemkni
                zprava "Novinka stranka.$cislo.html smazana"
            else
                odemkni
                zprava "Novinka $den.$mes. $n nenalezena"
            fi
        ;;
    * ) zprava "Spatny format dopisu";;
    esac
}
```

Poznámky:

Abych se přiznal, ve skutečnosti jsem osobně příkaz **`until`** asi nikdy nepoužil, protože je zcela ekvivalentní příkazu „**`while !`** ...“ a cyklus **`while`** je daleko běžnější v jiných jazycích. Jeho použití v tomto řešení poměrně dobře vystihuje, proč se takový příkaz mohl autorům zdát užitečný. Snad jsme tím zpravodajské povinnosti dostáli se ctí.

Pokud se vám zdá, že jsme při manipulaci s počitadlem souborů dostatečně nevyužili potenciál bezmála tří set nastudovaných stránek, tak zůstaňte v klidu. Naše „obyčejné“ řešení je pro daný účel určitě perfektně vhodné a zcela srozumitelné. Ale abychom se nezpronevěřili zvyklostem, podívejme se na některá jiná.

Poměrně elegantní metodu, jak zvýšit hodnotu počitadla představuje příkaz:

```
sinek:~/wwwnews> expr `cat maximum` + 1 > maximum
```

Možná vás zarazí, že v rámci jednoho příkazu soubor čteme a přepisujeme. To jsme si přece ukazovali jako odstrašující případ. Ale tady nejprve proběhne příkaz ve zpětných apostrofech a ten najde soubor v původním stavu. Až po jeho skončení shell dokončí zpracování parametrů pro příkaz **expr** a v té chvíli už může soubor klidně přepsat.

Hodnotu nového maxima však musíme později dostat ještě do proměnné shellu, takže touto cestou na počtu příkazů ani procesů nic nezískáme.

Bylo by také možné využít jazyk **awk**. V něm umíme soubor s hodnotou maxima přečíst, hodnotu zvýšit a zase ji do souboru zapsat. Dokonce ji můžeme zároveň vypsat i na standardní výstup a pak přiřadit přes zpětné apostrofy do proměnné:

```
sinek:~/wwwnews> max=`awk '{
> max = $0 + 1
> print max > "maximum"
> print max
> }' maximum`
```

I zde platí, že filtr nejprve řádku ze souboru přečte a teprve potom se soubor otevírá pro zápis (jako důsledek volání příkazu **print**), a tak konstrukce bude fungovat. Tentokrát dokonce opravdu ušetříme nějaké procesy. Ale s čitelností je tohle řešení hodně na štíru. Něco jiného by bylo, kdybychom úplně celý program napsali v jazyce **awk** (což by pochopitelně bylo možné), pak by se nám podobné postupy hodily.

Všimněte si, jak jsme se vypořádali s problémem čtení z roury v shellu ve větvi `SMAZAT`. Díky použití příkazu **if** za rourou, běží celý příkaz **if** včetně větve **then** ve stejném procesu, a proto jsou proměnné `radka` a `cislo` nastaveny správně.

Připomeňme, že proměnné v programu pro **awk** nemají žádný vztah k proměnným shellu. Stejná jména (např. `ukol` a `odesilatel`) jsme zvolili pouze pro přehlednost.

Pokud chcete příklad ladit, nemusíte si kvůli tomu samozřejmě instalovat webový server. Stačí vytvořit adresář, do něj uložit konfigurační soubory `spravci` a `maximum`, indexový soubor `index.html` s obsahem:

```
<ol>
</ol>
```

a poté už pouze vyrábět „dopisy“ (v nich stačí řádky „`From:...`“, „`Subject:...`“ a tělo) a posílat je skriptu na standardní vstup.

6.4. Prohledávání pošty

Zadání:

Napište program na vyhledávání řádek se zadaným textem (1. parametr) v souboru s příchozí elektronickou poštou (2. parametr). O každém dopisu, v němž program najde nějakou řádku, je třeba vypsat do záhlaví základní informace (adresu odesilatele, datum a čas doručení a předmět dopisu) a poté text všech nalezených řádek.

Příchozí elektronické dopisy se v UNIXu ukládají každému uživateli chronologicky do jednoho souboru, jehož jméno bývá uloženo v systémové proměnné **`MAIL`**. Jednotlivé dopisy jsou od sebe odděleny tak, že dopis začíná zvláštní řádkou ve tvaru „`From ...`“, za níž následují hlavičky a tělo dopisu:

```
From adresa_odesilatele  datum_a_čas_doručení
...začátek hlavičky dopisu
Subject:  předmět dopisu
...další řádky hlavičky
                                        prázdná řádka oddělující hlavičku
...text dopisu
```

Požadovaný tvar výpisu je:

```
==== From adresa_odesilatele  datum_a_čas_doručení  předmět_dopisu
...nalezené řádky
==== From adresa_odesilatele  datum_a_čas_doručení  předmět_dopisu
...nalezené řádky
```

Počítejte s tím, že řádka hlavičky s předmětem dopisu může chybět. Dejte pozor také na případné výskyty řádek ve tvaru „`Subject:...`“ v těle dopisu. Takové řádky se do vypisovaného záhlaví dopisu nesmí promítnout – tam se smí zobrazit pouze obsah řádky „`Subject:...`“, která byla v záhlaví (byla-li tam). Naproti tomu výskyty řádek ve tvaru „`From ...`“ v těle dopisu nemusíte uvažovat.

Zadaný text se hledá i v řádkách s hlavičkami dopisů, může se tedy stát, že nějaké hledané řádky objevíte dříve než předmět dopisu. V úvodní oddělovací řádce dopisu („`From ...`“ se naopak text nehledá).

Pokud nemáte k dispozici rozumná data pro ladění, příklady mailboxů můžete najít na webových stránkách knihy [13].

Rozbor:

Opět se pokusíme ukázat řešení využívající tři známé nástroje (shell, **`sed`** a **`awk`**), ale tentokrát se zaměříme na rychlost.

Rozbor (shell):

Řešení v shellu bude poměrně přímočaré. Budeme postupně číst řádky souboru a pomocí příkazu **`case`** rozlišíme jednotlivé případy řádek, jež nás budou zajímat.

Pokud bychom neměli za úkol přidávat do záhlaví předmět dopisu, byli bychom prakticky hotovi – záhlaví bychom si zapamatovali v proměnné `zahlavi` a u prvního výskytu textu v rámci dopisu bychom ho vypsali. Dále už by se jen vypisovaly nalezené řádky. Museli bychom si pouze někde pamatovat, zda už jsme záhlaví vypisovali, a to lze udělat jednoduše tak, že proměnnou po vypsání záhlaví vymažeme.

Pokud ale na hledané řádky narazíme ještě před tím, než zjistíme předmět dopisu, nemůžeme v té chvíli ještě záhlaví vypsat. Nalezené řádky budeme tedy ukládat také do nějaké proměnné a počkáme na případný výskyt předmětu. Vyhodnocení dopisu (tj. ověření, zda se v něm nalezly nějaké řádky a budeme ho vypisovat) můžeme odložit na okamžik načtení záhlaví dalšího dopisu. A totéž nás samozřejmě bude čekat ještě jednou na úplném konci souboru – pro vyhodnocení posledního dopisu.

Abychom odlišili výskyty řádek ve tvaru „`Subject:...`“ v hlavičce a v těle dopisu, musíme v každém okamžiku vědět, zda čteme hlavičku nebo tělo dopisu. Nejjednodušší bude zavést ještě jednu proměnnou `telo`, kterou budeme nastavovat na hodnotu 0 při nalezení řádky „`From ...`“ a hodnotu 1 při nalezení prázdné řádky.

Jenže když už jsme přidali režii s detekcí těla dopisu, mohli bychom ještě mírně upravit náš původní algoritmus. Shromažďování nalezených řádek je totiž nutné dělat pouze tak dlouho, dokud čteme hlavičku. Na jejím konci už víme, zda a jaký byl předmět dopisu, můžeme tedy provést vyhodnocení a přejít do režimu, kdy nalezené řádky už pouze vypisujeme (s eventuálním vypsáním záhlaví, pokud vypisujeme první výskyt v celém dopisu).

Rámcové obrysy algoritmu jsou jasné, a tak můžeme programovat. Pokud to zkusíte a ověříte funkčnost programu na nějakých malých datech, budete nejspíš spokojeni. Věc má ale jeden háček. Pokud budete prohledávat velký mailbox, nedočkáte se. Přestože jsme se snažili o maximální efektivitu a během celého cyklu vlastně nespustíme nový proces (v současných implementacích jsou všechny námi použité příkazy zabudovány jako interní), režie shellu při zpracování takových objemů dat bude příliš vysoká.

Pokud chceme dosáhnout alespoň trochu rozumné použitelnosti, musíme program radikálně zrychlit. Hlavním problémem je velký objem dat zpracovávaných shellem. Ke zlepšení by proto mohl pomoci trik zmiňovaný v kapitole 3.6 – nejprve z mailboxu příkazem **`grep`** vybereme „správné“ řádky a pouze ty necháme shell číst. Míra zrychlení samozřejmě závisí na mnoha okolnostech (např. na velikosti mailboxu a na množství nalezených řádek), ale může jít o řádové rozdíly – podíváme se na to podrobněji na konci kapitoly.

Řešení (shell):

```
uloz_vyskyt() {
    vyskyty="$vyskyty
$radka"
}
uloz_predmet() {
    [ -z "$predmet" ] &&
        predmet=${radka#Subject:}
}
vyhodnot() {
    [ -n "$vyskyty" ] &&
        echo "==== $zahlavi$predmet$vyskyty"
    zahlavi=
}
vypis_vyskyt() {
    [ -n "$zahlavi" ] &&
        echo "==== $zahlavi$predmet"
    zahlavi=
    echo "$radka"
}
novy_dopis() {
    predmet=; vyskyty=; zahlavi=$1; telo=0
}
grep -e '^From ' -e '^Subject:' -e '^$' -e "$1" $2 | {
    novy_dopis
    telo=1
    while IFS= read -r radka; do
        if [ $telo = 1 ]; then
            case $radka in
            From' '*' '?* )  novy_dopis "$radka";;
            *"$1"* )         vypis_vyskyt;;
            esac
        else
            case $radka in
            Subject:*"$1"* ) uloz_vyskyt; uloz_predmet;;
            Subject:* )      uloz_predmet;;
            *"$1"* )         uloz_vyskyt;;
            '' )             vyhodnot; telo=1;;
            esac
        fi
    done
}
```

Poznámky (shell):

Vzhledem k tomu, že usilujeme o co nejvyšší rychlost, použili jsme další konstrukt, který se v shellu objevil až později, takže ho nenajdeme v nejstarších implementacích. Norma ho ale obsahuje a většina dnešních systémů rovněž. Jedná se o tzv. *substituci podřetězců*. Pomocí ní je možné expandovat hodnotu nějaké proměnné nebo parametru zkrácenou o určitou část (ze začátku nebo konce).

Konstrukty jsou celkem čtyři („**${***jméno***%***vzor***}**“, „**${***jméno***%%***vzor***}**“, „**${***jméno***#***vzor***}**“ a „**${***jméno***##***vzor***}**“), přičemž *vzor* popisuje vynechávaný řetězec a zadává se pomocí metaznaků porovnávacích vzorů („*****“, „**?**“ a „**[]**“). Varianty s jedním operátorem vypouštějí minimální podřetězec odpovídající vzoru, varianty se dvěma operátory naopak maximální. Varianty s operátorem „**%**“, resp. „**%%**“ vypouštějí řetězec zprava, varianty s operátorem „**#**“, resp. „**##**“ zase zleva (jednoduchá mnemotechnická pomůcka spočívá v tom, že znak „**%**“ je na klávesnici napravo od „**#**“).

Poněkud překvapivá je možná inicializace proměnné `telo` na hodnotu 1, ale když si situaci na začátku mailboxu představíte, náš program je vlastně opravdu v úplně stejném stavu, jako by byl na konci (těla) nějakého dopisu.

V řádkách dopisů lze určitě očekávat výskyty zpětných lomítek, stejně tak je třeba počítat s mezerami či tabulátory na začátku řádek. Proto jsme příkaz **`read`** volali jednak s přepínačem „**`-r`**“ a jednak s vyprázdněnou proměnnou **`IFS`**, abychom řádky dostali opravdu v přesném tvaru, jak byly uvedeny v dopisu.

Rozbor (`sed`):

Oproti předešlým úlohám budeme zde schopni v editoru **`sed`** napsat celé řešení. Základní myšlenka bude pochopitelně stejná, jako byla v shellu. Rozdíl ale spočívá v tom, že v editoru nemáme proměnné. Na podobný problém jsme narazili už v kapitole 3.16 a i zde bude řešením využití odkládacího prostoru. Drobnou komplikací ale je, že zde bychom potřebovali proměnných více a odkládací prostor je jen jeden. Budeme si tedy v něm muset data dobře zorganizovat, abychom s nimi mohli snadno pracovat. Potřebujeme ukládat řádky tří typů: záhlaví, předmět a nalezené řádky. Tyto typy řádek musíme být schopni v odkládacím prostoru jednoduše rozlišit. První dva typy řádek začínají vždy pevným znakem („`F`“, resp. „`S`“), stačilo by tedy před řádky textu přidat jiné písmeno (třeba „`T`“). Navrhovaná struktura odkládacího prostoru je:

```
From odesilatel  datum a čas
Třádka...
Subject: předmět dopisu
Třádka...
```

Za poslední řádkou už nebude znak konce řádky (jak to v editoru obvykle bývá).

Nový dopis začíná oddělovací řádkou, kterou stačí uložit do odkládacího prostoru. Tam ovšem v té chvíli ještě máme data předešlého dopisu, která potřebujeme zpracovat. Proto použijeme příkaz *exchange* – tím oddělovací řádku uložíme a zároveň připravíme do pracovního prostoru data starého dopisu, která následně zpracujeme.

Řádky, jež obsahují hledaný text, můžeme přidávat na konec odkládacího prostoru (to umí příkaz *Hold*). Pouze jim na začátek nejprve přidáme písmeno `T`.

Řádky s předmětem bychom mohli zpracovat stejně. Ale musíme se ještě zamyslet nad ochranou před nechtěným zpracováním řádek ve tvaru „`Subject:…`“ v těle dopisu. Řešení je jednoduché. Prvním krokem je přidání kontroly, že předmět ukládáme nejvýše jednou, a druhým je ošetření dopisů, které předmět v hlavičce nemají.

Začneme opět příkazem *exchange*, po jehož skončení budeme mít v pracovním prostoru uložené řádky aktuálního dopisu od začátku až do této chvíle. Zjistíme, zda se mezi nimi vyskytuje předmět (řetězec „`\nS`“), a pokud ne, přidáme aktuální řádku k datům dopisu. Aktuální řádka je ale nyní v odkládacím prostoru, musíme proto použít příkaz *Get*. Nakonec obsahy obou prostorů zase vyměníme.

A podobné kroky uplatníme i na prázdnou řádku oddělující hlavičky dopisu od jeho těla – chováme se vlastně stejně, jako bychom narazili na prázdný předmět.

Pokud bloky příkazů napíšeme ve správném pořadí (nejprve zpracování předmětu, pak hledaného textu), bude nám dobře fungovat i hledání textu v řádkách s předmětem.

Na konci dopisu musíme zpracovat nastřádaná data. Už jsme říkali, že zavoláme příkaz *exchange* a budeme probírat obsah pracovního prostoru. Nejprve zkontrolujeme, zda tam máme alespoň jednu nalezenou řádku (neboli řetězec „`\nT`“). Pokud ne, dopisem se dále nezabýváme. Jinak musíme nejprve zaměnit pořadí řádek tak, aby řádka s předmětem byla první hned za záhlavím, a poté vymažeme text „`\nSubject:`“ a na začátek přidáme rovnítka. Tím nám vznikne řádka se záhlavím dopisu v požadovaném tvaru a za ní nalezené řádky. Nyní obsah pracovního prostoru vypíšeme (a samozřejmě ještě vymažeme přidaná písmena `T` ze začátků řádek).

Posledním problémem zůstává vyhodnocení posledního dopisu. To ale naštěstí půjde vyřešit triviálně. Po dokončení všech operací s řádkou se podíváme, zda nejsme na konci souboru (tj. na poslední řádce). Pokud tomu tak bude, nahradíme jednoduše obsah pracovního prostoru textem, který bude simulovat, že přišlo nové záhlaví dopisu. Díky tomu proběhne potřebný blok příkazů pro zpracování konce dopisu.

Dospěli jsme k následujícímu řešení:

```
sed -n '
/^$/ {                          ... konec hlaviček, začátek těla
    x
    /\nS/ !s/$/\                ... předmět zatím není, přidáme prázdný
Subject:/
    x
}
/^Subject: / {
    x
    /\nS/ !G                    ... připojení řádky, pokud ještě nebyl předmět
    x
}
'/$1/' {                        ... nalezena řádka
    s/^/T/
    H
}
$ s/^/From x x/                 ... simulace záhlaví na poslední řádce
/^From .* ./ {                  ... záhlaví nového dopisu
    x                           ... uložení záhlaví, zpracování starého dopisu
    /\nT/ {                     ... přesun předmětu před nalezené řádky
        s/\(\n.*\)*\(\nS[^\n]*\)/\2\1/
        s/\nSubject:/ /         ... spojení záhlaví a předmětu
        s/\nT/\                 ... vypuštění přidaných písmenek
/g
        s/^/==== /
        p
    }
}' $2
```

Jenže opět vyvstává podobný problém jako v minulém řešení. Velká data bude náš program zpracovávat velmi pomalu. Můžeme sice použít podobnou fintu jako minule, ale zjistíme, že úspora tentokrát nebude tak významná. Co editoru tak dlouho trvá? Kamenem úrazu je jeden z posledních příkazů *substitute* ve skriptu – ten, který přesouvá řádku s předmětem před případné nalezené řádky, jež předmětu předcházely. Používá totiž zpětné reference, s nimiž se sice dají dělat různá kouzla, ale je třeba si přitom hlídat míru. A pro zpracování velkých dat to platí dvojnásob. Zkusíme se poohlédnout, zda bychom nedokázali zpětné reference ze závěrečných úprav vypustit.

Kdybychom výpis dělali nadvakrát – nejprve záhlaví a potom teprve nalezené řádky, stačilo by určité úseky z pracovního prostoru pouze odmazávat. Zcela bychom se obešli bez zpětných referencí a složitých regulárních výrazů. Jenže jakmile připravíme k tisku řádku se záhlavím a předmětem, přijdeme o nalezené řádky, které jsme úpravami odmazali. Potřebovali bychom si je někam schovat, ale není kam! Odkládací prostor

máme už obsazený! V okamžiku čtení úvodní řádky nového dopisu obsahuje totiž právě tuto řádku (protože jsme prohodili pracovní a odkládací prostor). A jinam si žádná data neuložíme. Nezbude nám tedy nic jiného než úvodní řádku nového dopisu přidat na konec odkládacího prostoru, za data předchozího dopisu, při přípravě výstupu tuto řádku ignorovat a po ukončení zpracování dopisu smazat vše kromě této řádky (tj. vše až po poslední znak konce řádky). Naše „paměť“ tak bude mít tvar:

```
From odesilatel  datum a čas
Třádka...
Subject:  předmět dopisu
Třádka...
From  záhlaví 2. dopisu
```

Z tohoto textu v prvním kroku vymažeme vše mezi prvním záhlavím a textem předmětu (zvlášť pro případ, kdy tam nějaké řádky byly, a zvlášť pro ten, kdy předmět následoval bezprostředně za hlavičkou) a poté můžeme použít příkaz *Print*, neboť vlastně chceme vypsat jen první řádku:

```
        s/\n.*\nSubject:/ /
        s/\nSubject:/ /
        s/^/==== /
        P
```

Nyní se potřebujeme dostat zase zpátky k původnímu obsahu. To však není problém, máme ho přece v odkládacím prostoru, stačí provést příkaz *get* a můžeme pokračovat. Nyní pro změnu vymažeme řádku s předmětem, obě záhlaví a přidaná písmena „T“:

```
        s/S[^\n]*\n//
        s/^[^\n]*\nT//
        s/\nF.*$//
        s/\nT/\
/g
```

Obsah pracovního prostoru opět vypíšeme a do třetice obnovíme obsah z odkládacího prostoru. Nyní už ale jen vymažeme vše kromě poslední řádky a tuto řádku pak vrátíme do odkládacího prostoru.

Řešení (sed):

```
sed -n '
/^$/ {                          ... konec hlaviček, přidání prázdného předmětu
    x
    /\nS/ !s/$/\                ... předmět zatím není, přidáme prázdný
Subject:/
    x
}
```

```
/^Subject: / {
    x
    /\nS/ !G                    ... připojení řádky, pokud ještě nebyl předmět
    x
}
'/$1/' {                        ... nalezena řádka
    s/^/T/
    H
}
$ s/^/From x x/                 ... na poslední řádce simulujeme záhlaví
/^From .* ./ {                  ... záhlaví nového dopisu
    H
    g
    /\nT/ {
        s/\n.*\nSubject:/ /
        s/\nSubject:/ /
        s/^/==== /
        P
        g
        s/S[^\n]*\n//
        s/^[^\n]*\nT//
        s/\nF.*$//
        s/\nT/\
/g
        p
        g
    }
    s/.*\n//
    h
}' $2
```

Poznámky (`sed`):

Poměrně složité operace s odkládacím prostorem jsme museli podstoupit proto, že máme k dispozici vlastně jen jedinou paměť, a tu si musíme pečlivě strukturovat. Velmi se to podobá programování Turingova stroje. To je matematický model počítače, který se používá pro dokazování různých vlastností výpočetních systémů. Takový model musí být velmi jednoduchý, jinak by se s ním podobné důkazy nedaly provádět. A zrovna s pamětí je na tom Turingův stroj dosti podobně jako editor **`sed`**. Daní za jednoduchost modelu je přirozeně to, že jakákoliv složitější operace se provádí velmi komplikovaně – a to je opět velmi podobné našemu případu.

V praxi je proto nutné vždy pořádně vyzkoušet, zda takové řešení není jen zajímavou logickou hříčkou, která díky své složitosti nemá odpovídající výkonové parametry, a zda tedy není nutno hledat jiné řešení.

Rozbor (awk):

Logiku řešení pro program v jazyce **awk** můžeme úplně opsat podle řešení v shellu.

Řešení (awk):

```
grep -e '^From ' -e '^Subject:' -e '^$' -e "$1" $2 | awk '
/^From .* ./ {
    zahlavi = $0
    predmet = ""
    vyskyty = ""
    telo = 0
    next
}
/^$/ && ! telo {
    if( vyskyty > "" ) {
        print "==== " zahlavi predmet vyskyty
        zahlavi = ""
    }
    telo = 1
    next
}
/^Subject:/ {
    if( predmet == "" )
        predmet = substr( $0, 9 )
}
/'"$1"'/ {
    if( ! telo )
        vyskyty = vyskyty "\n" $0
    else {
        if( zahlavi > "" )
            print "==== " zahlavi predmet
        zahlavi = ""
        print
    }
}'
```

Poznámky:

Všechny varianty řešení této úlohy jsme zkoušeli pouštět na různě silných počítačích s různými verzemi operačních systémů. Kvalita počítače nijak výrazně neovlivňovala vzájemné poměry výsledků, proto jsme ji z dalšího zkoumání vypustili. Finální testy jsme pouštěli pouze na třech různých typech operačních systémů, aniž bychom se snažili vyrovnat hardwarové rozdíly mezi jednotlivými stroji.

V případě shellu a **awk** jsme vyzkoušeli nejen variantu uvedenou v textu jako řešení (přímý výpis výskytů v těle), ale rovněž verzi s ukládáním nalezených řádek po celou dobu čtení hlaviček i těla dopisu a vyhodnocením a výpisem až na jeho konci.

U editoru jsme testovali obě řešení uvedená v textu – tj. se závěrečnými substitucemi pomocí zpětných referenci i s úpravou výpisu pomocí několika mazání příkazem *delete*.

Programy jsme zkoušeli na souboru o velikosti cca 5MB, a to pro čtyři kvantitativně odlišné výsledky hledání: žádný výskyt, výskyt na cca 3 % řádek, cca 20 % řádek a cca 70 % řádek. U všech variant jsme přidali i řešení bez úvodní probírky příkazem **grep**. Uvedené časy jsou ve vteřinách.

Prvním zkušebním strojem byl můj postarší notebook s FreeBSD:

			0 %	3 %	20 %	70 %
shell	přímý výpis	probírka	0,3	0,7	3,9	11,0
		plná data	17,1	17,1	17,9	19,2
	ukládání	probírka	0,3	0,7	5,4	85,0
		plná data	16,5	16,5	18,6	91,9
sed	*delete*	probírka	0,1	0,6	2,6	6,6
		plná data	0,9	1,4	3,1	6,7
	zpětné reference	probírka	0,1	8,6	10,4	9,2
		plná data	0,9	9,3	11,0	9,5
awk	přímý výpis	probírka	0,1	0,1	0,4	0,9
		plná data	0,6	0,6	0,7	1,0
	ukládání	probírka	0,1	0,1	2,2	74,2
		plná data	0,6	0,6	2,6	74,2

Druhým byl server s Linuxem, hardwarově sice lepší, ale trochu vytíženější:

			0 %	3 %	20 %	70 %
shell	přímý výpis	probírka	4,4	8,9	36,3	105,6
		plná data	29,6	31,5	38,4	64,2
	ukládání	probírka	4,4	8,5	108,9	2831,8
		plná data	27,4	28,4	108,4	2806,0
sed	*delete*	probírka	1,5	5,1	16,6	43,0
		plná data	28,9	31,6	36,2	49,1
	zpětné reference	probírka	1,4	5,5	14,9	35,9
		plná data	28,5	31,8	33,5	40,8
awk	přímý výpis	probírka	0,7	1,1	3,4	9,1
		plná data	10,3	10,5	10,7	10,9
	ukládání	probírka	0,5	0,9	4,0	30,8
		plná data	10,2	10,4	10,9	31,0

Posledním strojem byla pracovní stanice Sparc se Solarisem:

			0 %	3 %	20 %	70 %
shell	přímý výpis	probírka	1,9	3,4	11,9	38,8
		plná data	19,0	20,0	23,9	38,8
	ukládání	probírka	1,6	3,6	103,6	3219,6
		plná data	15,9	17,4	113,5	3219,3
sed	*delete*	probírka	0,4	0,6	12,5	1215,0
		plná data	0,9	1,4	13,2	1215,0
	zpětné reference	probírka	0,4	> 2 h	> 2 h	> 2 h
		plná data	0,9	> 2 h	> 2 h	> 2 h
awk	přímý výpis	probírka	0,4	0,4	0,9	2,8
		plná data	2,1	2,2	2,5	3,2
	ukládání	probírka	0,4	0,4	3,3	79,0
		plná data	2,0	2,1	4,8	79,4

Jaký závěr si lze odnést z pokusů v této kapitole?

Shell určitě není adept na zpracování velkých objemů dat. Jeho doménou je řízení procesů a to jsme v této úloze vůbec nepotřebovali. Při úvahách o jeho nasazení je třeba pečlivě posoudit možné objemy dat a shell zvolit pouze tehdy, pokud jejich rozsah nebude velký. A případně jejich množství snížit pomocí nějakého předzpracování.

Na zpracování textu je určitě lepší editor. Běží jako jediný proces, má zabudovanou práci s textem, s regulárními výrazy, díky tomu často není nutné ani předzpracování dat. Je však velmi citlivý na konkrétní implementaci, zejména u složitějších operací. A to je prvek, který se dá při tvorbě programu jen velmi obtížně zohlednit, zvláště pokud plánujeme program používat v různých systémech. A především: pokud potřebujeme do programu vložit netriviální logiku, je řešení v editoru obtížně čitelné.

Ze srovnání nám tedy vychází pro tuto třídu úloh nejlépe jazyk **awk**. Algoritmus se zapisuje velmi přirozeně a rychlost provádění je většinou dostačující. Ovšem i zde se vyplatí přemýšlet nad vylepšeními, která – nám nebo počítači – zjednoduší práci.

300

Aby váš UNIX skvěle shell...

...budete se muset naučit hodně triků a fíglů. K nim si budete muset nejprve trochu prošlapat cestičku.

V této knize jsme se snažili vám na té cestě trochu pomoci. Vůbec nechceme tvrdit, že jsme ukazovali vždy jedinou a správnou možnost. A určitě narazíte na problémy, které pomocí prostředků ukázaných v této knize řešit nepůjde. Nebo to nepůjde snadno. Pak asi sáhnete po jiném prostředku, kterým může být třeba některý z „opravdových" programovacích jazyků. Ale pokud si osvojíte techniky popsané v této knize, uvidíte, že vám překvapivě často budou postačovat. A mají neocenitelnou přednost, kterou je srozumitelnost řešení a rychlost vývoje vaší aplikace.

Tak, šťastnou cestu...

Příloha A: Stručný přehled příkazů

V této části uvádíme stručný popis příkazů, s nimiž jsme se v knize seznámili. Slouží pouze pro rychlejší a snazší orientaci, je omezen na nejdůležitější fakta a pro získání úplné informace je třeba vždy přečíst ještě odpovídající manuálovou stránku nebo příslušnou část normy [1].

Při popisu možností používáme následující konvence:

`command` (text zapsaný neproporcionálním fontem)

Tuto část příkazu nebo výrazu je třeba zapsat přesně tak, jak je uvedena.

argument (text zapsaný kurzívou, proporcionálním fontem)

Tuto část text je třeba nahradit odpovídajícím řetězcem (číslem, jménem, názvem souboru, příkazem apod.).

[] (text zapsaný mezi hranaté závorky)

Volitelná část příkazu nebo výrazu.
Pozor na záměnu se skutečnými hranatými závorkami, které je třeba opravdu zapsat.

... (tři tečky)

Bezprostředně předcházející část zápisu je možno libovolně opakovat.

Přehled nejdůležitějších utilit a příkazů shellu:

awk	Programovatelný filtr pro zpracování textových souborů	A.2
basename	Extrakce jména souboru	A.1
cat	Kopírování obsahu vstupu nebo souborů na standardní výstup	A.1
cd	Změna aktuálního pracovního adresáře shellu	A.6
chmod	Nastavení přístupových práv k souborům	A.1
comm	Výpis společných a rozdílných řádek dvou souborů	A.1
command	Zavolání příkazu	A.6
cp	Kopírování souborů	A.1
cut	Výběr sloupců ze vstupních souborů	A.1
date	Výpis aktuálního data a času	A.1
dd	Kopírování daného rozsahu dat a konverze formátu souborů	A.1
df	Výpis informace o připojených souborových systémech	A.1
diff	Porovnání dvou souborů, výpis nalezených rozdílů	A.1
dirname	Extrakce cesty k adresáři obsahujícímu soubor	A.1
ed	Editace souboru pomocí příkazů ze standardního vstupu	A.3
echo	Výpis parametrů na standardní výstup	A.1
eval	Vyvolání opakovaného zpracování příkazové řádky shellu	A.6
exec	Ukončení shellu a spuštění jiné úlohy	A.6
exit	Ukončení shellu	A.6
expr	Vyhodnocení celočíselného nebo logického výrazu	A.1
file	Výpis typu (formátu) obsahu souboru (souborů)	A.1
find	Hledání souborů podle zadaných kritérií	A.4
getopts	Zpracování přepínačů ve skriptu	A.6
grep	Výpis řádek vyhovujících regulárnímu výrazu	A.1
head	Výpis začátku standardního vstupu nebo zadaných souborů	A.1
id	Výpis UID a GID uživatele	A.1
join	Databázové spojení souborů (tabulek)	A.1
ln	Vytváření linků	A.1
ls	Výpis seznamu souborů (s případnými dalšími informacemi)	A.1
mailx	Zaslání elektronického dopisu	A.1
man	Zobrazení manuálové stránky příkazu	A.1
mkdir	Vytváření adresářů	A.1
mv	Přesouvání a přejmenování souborů	A.1
od	Výpis obsahu souboru v zadaném (implicitně oktalovém) formátu	A.1

paste	Spojení řádek vstupních souborů vedle sebe (zřetězení)	A.1
printf	Výpis formátovaného textu na standardní výstup	A.1
ps	Výpis informace o procesech	A.1
pwd	Výpis cesty k aktuálnímu pracovnímu adresáři	A.1
read	Čtení řádek ze vstupu	A.6
rm	Mazání souborů	A.1
rmdir	Mazání adresářů	A.1
sed	Výpis řádek vstupu nebo souborů upravených editačními příkazy	A.5
set	Výpis proměnných, nastavování parametrů a přepínačů shellu	A.6
sh	Spuštění shellu	A.6
shift	Posun pozičních parametrů shellu	A.6
sleep	Pozastavení běhu procesu	A.1
sort	Výpis setříděných souborů na výstup	A.1
split	Rozdělení souboru na několik souborů dané velikosti	A.1
tail	Výpis konce vstupu nebo souboru počínaje určeným místem	A.1
tar	Práce s archivy souborů	A.1
tee	Kopírování vstupu na výstup a zároveň do souboru	A.1
test, **[ ]**	Testování zadaných výrazů	A.1
touch	Změna časových údajů o souborech	A.1
tr	Konverze a mazání znaků	A.1
trap	Obsluha signálů v shellu	A.6
umask	Nastavení uživatelské masky shellu	A.6
uname	Výpis jména systému a dalších atributů	A.1
uniq	Výpis unikátních, resp. duplicitních řádek souboru	A.1
unset	Zrušení nastavení proměnných shellu	A.6
wc	Zjištění počtu znaků, slov a řádek v souborech	A.1
who	Seznam přihlášených uživatelů	A.1
xargs	Spouštění příkazů s parametry přečtenými ze standardního vstupu	A.1

A.1 Vybrané utility

basename *cesta* [*suffix*]

Příkaz odřízne z plné cesty k souboru název adresáře, takže zůstane pouze vlastní jméno souboru. Pokud je zadán druhý parametr, odřízne ještě příponu *suffix*.

cat [*soubor*]...

Příkaz čte obsah standardního vstupu nebo zadaných souborů a vypisuje ho na standardní výstup.

chmod [**-R**] *nastavení soubor...*

Příkaz nastaví zadaným souborům přístupová práva.

Přepínače:

-R Příkaz pracuje rekurzivně (nastaví práva pro celý podstrom).

Symbolický formát zadání parametru *nastavení*: jedna nebo více definic ve tvaru „[*množina...*]*operátor*[*právo...*]“ oddělených čárkami.

Množiny uživatelů: **u** (vlastník), **g** (skupina vlastníka), **o** (ostatní), **a** (všichni).

Vynechaná hodnota znamená nastavení všem s následnou aplikací masky (viz příkaz **umask**).

Operátory: **=** (nastavit), **+** (přidat), **-** (odebrat).

Práva: **r** (read), **w** (write), **x** (execute/search)

X (=právo **x** jen pro adresáře a spustitelné soubory),

u, **g**, **o** (=kopie práv podle dané množiny uživatelů).

Numerický formát zadání parametru *nastavení*: „[[[*u*]*g*]*o*]“

Ostatní číslice odpovídají právům pro jednotlivé množiny uživatelů a skládají se z hodnot: **4** (read), **2** (write), **1** (execute, resp. search).

comm [**-123**] *soubor1 soubor2*

Příkaz čte (setříděné) soubory a vypisuje tři sloupce oddělené tabulátory: řádky obsažené pouze v prvním souboru, pouze ve druhém souboru, v obou souborech.

Přepínače:

-*n* Program nevypisuje sloupec číslo *n*.

command ...

Příkaz vykoná příkaz zadaný v parametrech, ignoruje případné stejnojmenné funkce.

Norma umožňuje implementovat příkaz jako externí utilitu, většinou je to ale interní příkaz shellu a my jej tak také v přehledu vedeme (viz str. 343).

cp [**-pf**] *soubor1 soubor2*

cp [**-Rpf**] *soubor... adresář*

První forma příkazu kopíruje soubor pod novým jménem.

Druhá forma kopíruje jeden nebo více souborů do existujícího adresáře.

Přepínače:

- **-f** Pokud uživatel nemá práva k cílovému souboru, příkaz se soubor pokusí nejprve smazat. [*force*]
- **-p** Nové soubory přebírají datum a čas poslední modifikace. [*preserve*]
- **-R** Pokud je zadán adresář, příkaz kopíruje celý podstrom. [*recursive*]

cut -c *seznam* [*soubor*]...

cut -f *seznam* [**-d** *oddělovač*] [**-s**] [*soubor*]...

Příkaz vystřihuje z řádek zadaných souborů určené úseky a vypisuje je na výstup. Výběr se provádí buď podle pořadových čísel znaků v řádce (čísluje se od jedničky), nebo podle čísel polí, oddělovač polí lze zadat (implicitně to je tabulátor). Seznam tvoří řada čísel nebo intervalů („[*x*]**-**[*y*]“) oddělených čárkami. Úseky se vypisují v tom pořadí, v jakém byly na vstupu, bez ohledu na pořadí v seznamu.

Přepínače:

- **-c** Zadání výřezu pomocí seznamu pořadových pozic znaků. [*chars*]
- **-d** Zadání znaku oddělovače polí místo tabulátoru. [*delimiter*]
- **-f** Zadání výřezu pomocí seznamu pořadových pozic polí. [*fields*]
- **-s** Pokud na řádce není dost polí, řádka se na výstup neopisuje. [*suppress*]

date [**-u**] [**+***formát*]

Příkaz vypíše aktuální datum a čas. Pokud není uveden parametr *formát*, použije implicitní formát „**+%a %b %e %H:%M:%S %Z %Y**“.

Přepínače:

- **-u** Výpis v UTC místo v lokálním čase.

Vybrané formátovací direktivy:

%a, **%A**	název dne v týdnu (zkrácený, resp. celý)
%b, **%B**	název měsíce (zkrácený, resp. celý)
%d, **%e**	číslo dne v měsíci (01 až 31, resp. *mezera*1 až 31)
%m, **%j**	číslo měsíce (01 až 12), číslo dne v roce
%H, **%M**, **%S**	hodiny (00 až 23), minuty, sekundy (00 až 59)
%u, **%W**	číslo dne v týdnu (Po = 1, Ne = 7), číslo týdne
%y, **%Y**	rok (dvojmístný, resp. čtyřmístný)
%%, **%n**	znak procento, znak konce řádky

dd [*parametr*]...

Příkaz kopíruje na základě údajů zadaných pomocí parametrů bloky dat ze souboru nebo standardního vstupu, provádí na nich požadované konverze a zapisuje je do souboru nebo na standardní výstup. Parametry mají tvar „*klíčové_slovo=hodnota*“.

Nejdůležitější parametry:

if=*soubor* Vstupní soubor (implicitně standardní vstup).

of=*soubor* Výstupní soubor (implicitně standardní výstup).

bs=*výraz* Velikost bloku.
Výraz tvoří celé kladné číslo nebo více čísel spojených znakem „**x**“ vyjadřujícím násobení uvedených hodnot. Ke každému číslu může být připojeno písmeno „**k**“ znamenající násobek 1024.

count=*n* Počet kopírovaných bloků.

skip=*n* Počet úvodních bloků, jež se mají přeskočit.

conv=*c*[**,***c*]... Požadované konverze.

Pomocí konverzí je možno provádět takové operace jako záměnu znakových sad ASCII a EBCDIC nebo změnu formátu textového souboru z běžného unixového (konec řádky určuje znak LF) na soubor s pevnou délkou řádky (doplněnou zprava mezerami).

df [**-k**] [*soubor*]...

Příkaz vypíše informace o kapacitě připojených souborových systémů. V případě použití bez parametrů vypíše informace o všech systémech, jinak vypíše pro každý uvedený soubor informaci o odpovídajícím systému, na němž se nachází.

Příkaz nemá standardizovaný tvar výstupu, takže jeho automatické zpracování je velmi obtížně přenositelné.

Přepínače:

-k Výpis kapacity v kB namísto v 512B blocích.

diff [*formát*] [**-br**] *soubor1 soubor2*

Příkaz porovná zadané soubory a vypíše nalezené rozdíly. Pomocí přepínačů je možné volit způsoby porovnávání a formát výpisu.

Implicitní formou výpisu je posloupnost bloků, z nichž každý začíná řádkou, která udává charakter nalezeného rozdílu (append, delete, resp. change) a pořadová čísla dotčených řádek v obou souborech, a pokračuje výpisem textu lišících se řádek prefixovaných řetězci „**<** “, resp. „**>** “ podle toho, k jakému souboru patří.

Alternativou je několik druhů kontextového výpisu, kdy je každý nalezený rozdíl vypsán společně s několika (implicitně třemi) okolními shodnými řádkami. Blíže viz manuálové stránky.

Poslední formou je výpis ve tvaru příkazů pro editor **ed**. Pokud by se tyto příkazy aplikovaly na první soubor, změnily by jeho obsah tak, že by se rovnal obsahu druhého souboru.

Přepínače ovlivňující styl porovnávání:

-b Při porovnávání se ignorují rozdíly ve skupinách bílých znaků. [*blank*]

-r Rekurzivní porovnávání pro adresáře. [*recursive*]

Přepínače ovlivňující formát výpisu:

-e Výstup ve tvaru příkazů pro editor **ed**.

-c, -C*n*, **-u, -U***n*

Různé druhy kontextového formátu výstupu (viz manuálové stránky).

dirname *cesta*

Příkaz odřízne ze zadané plné cesty k souboru poslední lomítko a jméno souboru, takže zůstane pouze cesta k adresáři, kde se soubor nachází. V případě souboru ležícího v kořeni příkaz vypíše lomítko.

echo [*parametr*]...

Příkaz vypíše na standardní výstup svoje parametry oddělené jednou mezerou a ukončené znakem konce řádky.

Příkaz se historicky vyskytuje ve dvou zcela nekompatibilních variantách:

- Verze ze systémů odvozených od BSD rozpoznává první parametr ve tvaru „**-n**“ jako přepínač, který způsobí vynechání závěrečného odřádkování.
- System V umožňuje v řetězci zapisovat znakové escape-sekvence jako v jazyce C (např. „**\t**“ nebo „**\n**“) a sekvence „**\c**“ má tentýž efekt jako „**-n**“ v BSD.

Norma ani jednu z těchto verzí nekodifikuje a doporučuje používat příkaz **printf**.

expr *parametr*...

Příkaz vyhodnotí celočíselný nebo logický výraz zapsaný pomocí samostatných parametrů příkazové řádky, vypíše výsledek na standardní výstup a podle něj vrátí návratovou hodnotu (0=nenulový výsledek, 1=nula, 2=chyba).

Výraz lze tvořit pomocí závorek a následujících operátorů:

&, | logické spojky „a zároveň“ a „nebo“

=, !=, <, <=, >, >=

aritmetické, resp. řetězcové porovnání

+, -, *, /, %

aritmetické operace, (**%**=zbytek po dělení)

: shoda s regulárním výrazem (automaticky ukotveným na začátek)

file *soubor...*

Pro každý zadaný soubor se příkaz pokusí rozpoznat jeho typ a odpovídající textový popis vypíše na standardní výstup.

getopts...

Příkaz postupně zpracovává přepínače ze seznamu parametrů.
Norma umožňuje implementovat příkaz jako externí utilitu, většinou je to ale interní příkaz shellu a my jej tak také v přehledu vedeme (viz str. 343).

grep [**-EFcilnqv**] *vzor* [*soubor*]...
grep [**-EFcilnqv**] [**-e** *vzor*]... [**-f** *soubor_vzorů*]... [*soubor*]...

Příkaz prohledává zadané soubory a hledá řádky, jejichž obsah (bez znaku konce řádky) odpovídá alespoň jednomu zadanému vzoru. Implicitní formou vzoru je BRE (viz str. 346). Implicitní formou výstupu je seznam nalezených řádek, kterým případně předchází jméno zdrojového souboru, bylo-li v parametrech uvedeno více souborů. Alespoň jeden vzor nebo soubor vzorů musí být vždy uveden.
Přepínače ovlivňující styl porovnávání:

[**-e**] *vzor*
: Zadání vyhledávacího vzoru; vzor může tvořit několik regulárních výrazů nebo řetězců oddělených znakem konce řádky. [*expression*]

-E Porovnávání podle vzorů chápaných jako ERE (viz str. 346).

-f *soubor_vzorů*
: Zadání souboru obsahujícího vyhledávacího vzory; soubor obsahuje řádky s regulárními výrazy nebo hledanými řetězci. [*file*]

-F Porovnávání přesné shody řetězců, nikoliv regulárních výrazů. [*fixed*]

-i Ignorují se rozdíly mezi malými a velkými písmeny. [*ignorecase*]

-v Vyhledávají se řádky neobsahující žádný ze zadaných vzorů. [*invert*]

-x Vyhledávají se řádky shodující se se vzorem na celé délce. [*exact*]

Přepínače ovlivňující styl výstupu:

-c Program vypisuje pouze počet nalezených řádek pro každý soubor. [*count*]

-l Program vypisuje pouze jména souborů obsahujících hledané řádky. [*list*]

-n Program vypisuje k nalezeným řádkám navíc jejich čísla. [*number*]

-q Program pouze vrací návratovou hodnotu (0=nalezeno), neprodukuje žádný výstup. [*quiet*]

head [**-n** *počet*] [*soubor*]...

Příkaz vypisuje na standardní výstup z každého zadaného souboru úvodní úsek o délce *počet* řádek (implicitně 10).

id [*uživatel*]

Příkaz vypisuje UID a GID aktuálního, resp. zadaného uživatele.

join [*přepínače*] *soubor1 soubor2*

Příkaz provádí spojení (join) dvou souborů na základě rovnosti hodnot v určených sloupcích (polích). Program vypíše na výstup jednu řádku pro každou dvojici řádek ze vstupních souborů, jež se shodují v hodnotě spojovacího sloupce. Implicitní formát výstupní řádky je: spojovací pole, všechna ostatní pole řádky prvního souboru, všechna ostatní pole řádky druhého souboru. Soubory musejí být setříděné podle spojovacího pole.

Přepínače:

-a *číslo*
: Zahrnutí výpisu všech (i nespárovaných) řádek souboru *číslo*. [*all*]

-e *text*
: Definice textu, který se má použít namísto prázdného řetězce při výpisu chybějících polí u nespárovaných řádek. [*empty*]

-o *formát*
: Definice výstupního formátu; hodnotou je čárkami oddělený seznam polí ve tvaru *číslo_souboru*.*číslo_pole*, resp. 0 pro spojovací pole. [*output*]

-t *znak*
: Definice oddělovače polí; implicitně je jím posloupnost bílých znaků.

-v *číslo*
: Výpis pouze nespárovaných řádek souboru *číslo*. [*invert*]

-1 *pole*, **-2** *pole*
: Zadání čísla spojovacího pole v prvním, resp. druhém souboru.

ln [**-fs**] *soubor1 soubor2*

ln [**-fs**] *soubor... adresář*

První forma příkazu vytváří *soubor2* jako link na *soubor1*.

Druhá forma příkazu vytváří linky na uvedené soubory v existujícím adresáři.

Přepínače:

-f Pokud nové jméno souboru již existuje, program ho nejprve smaže.

-s Příkaz nevytváří hardlink, ale symbolický link.

ls [**-ALRSadfilnqrtu1**] [*soubor...*]

Příkaz vypisuje informace o souborech, resp. adresářích zadaných jako parametry (implicitně o aktuálním adresáři). Bez použití přepínačů nejprve vypíše jména souborů, které nejsou adresáře, a poté postupně obsah všech zadaných adresářů. Pomocí přepínačů lze nechat vypisovat nejen jména souborů, ale i další informace o nich. Lze rovněž potlačit nebo naopak rozšířit zanořování do podadresářů. Seznam vypisovaných souborů, resp. souborů vypisovaných v rámci jednoho adresáře je implicitně setříděn abecedně, případně podle lokálního nastavení. Skryté soubory (začínající tečkou) se implicitně nevypisují.

Přepínače:

- **-a** Program vypisuje všechny soubory včetně skrytých. [*all*]
- **-A** Program vypisuje všechny soubory včetně skrytých s výjimkou „**.**“a „**..**“.
- **-d** Pro adresáře zadané v parametrech program vypisuje informace o adresáři samotném namísto výpisu jeho obsahu. [*directory*]
- **-f** Program netřídí výpis, vypisuje obsah adresářů tak, jak je uložen. [*force*]
- **-i** Program vypisuje ke každému souboru číslo i-node. [*inode*]
- **-l** Program vypisuje ke každému souboru plnou informaci (typ, práva, počet linků, vlastníka, skupinového vlastníka, velikost, datum a čas poslední modifikace a jméno), tzv. *dlouhý* výpis. [*long*]
- **-L** Pro symbolické linky program vypisuje informace o cílovém souboru namísto symbolického linku samotného. [*link-follow*]
- **-n** Program nevypisuje vlastníky souborů jménem, ale číslem (UID/GID). [*number*]
- **-q** Program vypisuje namísto netisknutelných znaků ve jménech souborů otazník. [*quote*]
- **-r** Program třídí výstup dle zadaných kritérií, ale vypisuje ho v obráceném pořadí. [*reverse*]
- **-R** Program postupuje rekurzivně a vypisuje celý podstrom. [*recursive*]
- **-S** Program třídí výstup sestupně podle velikosti souborů. [*size*]
- **-t** Program třídí výstup podle data a času poslední modifikace souboru. [*time*]
- **-u** Program pracuje s časem posledního přístupu místo času modifikace (platí pro výpis i pro třídění). [*usage*]
- **-1** Program vypisuje jeden soubor na jednu řádku výstupu.

mailx [**-s** *předmět*] *adresa...*

Příkaz pošle obsah standardního vstupu elektronickou poštou na zadané adresy.

Přepínače:

- **-s** Zadání předmětu dopisu. [*subject*]

man [**-k**] *jméno...*

Příkaz zobrazí informaci (manuálovou stránku) k danému jménu (příkazu), resp. vyhledá seznam stránek vztahujících se k zadanému jménu (při použití „**-k**“).

mkdir [**-p**] [**-m** *mód*] *cesta...*

Příkaz vytvoří zadané adresáře.

Přepínače:

-m Zadání požadovaných přístupových práv pro cílové adresáře; formát a význam parametru je shodný s parametrem příkazu **chmod**. [*mode*]

-p Vytvoření plné cesty; pokud zadaná cesta obsahuje mezilehlé adresáře, které neexistují, program vytvoří i je. [*parent*]

mv *soubor1 soubor2*

mv *soubor... adresář*

První forma příkazu přejmenuje nebo přesune soubor pod novým jménem.

Druhá forma příkazu přesune jeden či více souborů do existujícího adresáře.

od [**-v**] [**-A** *báze*] [**-j** *posun*] [**-N** *rozsah*] [**-t** *formát*]... [*soubor*]...

Příkaz vypíše obsah zadaných souborů v uživatelsky definovaném formátu.

Přepínače:

-A *báze*

Způsob výpisu adresy. Na začátku výstupních řádek program vypisuje relativní posun vůči začátku souboru. Je možné volit, zda tento výpis bude oktalový („**o**“, defaultní hodnota), dekadický („**d**“), hexadecimální („**x**“) nebo bude vypuštěn („**n**“). [*address radix*]

-j *posun*

Posun začátku výpisu. Výpis bude začínat zadaný počet bytů od počátku souboru. Velikost posunu lze zadat jako celé číslo v jazyce C (v libovolné soustavě), navíc lze připojit písmeno „**k**“ jako vyjádření násobku 1024 bytů nebo „**m**“ jako vyjádření násobku 1048576. [*jump*]

-N *rozsah*

Omezení rozsahu výpisu (počtem bytů zdrojového souboru). [*number*]

-t *formát*

Zadání formátu výpisu. Pokud je uvedeno více specifikátorů formátu, bude se každá řádka ve výstupu opakovat postupně ve všech formátech. [*type*]

-v Výpis veškerých dat. Implicitně program sdružuje shodné bloky vstupních souborů tak, že opakující se řádky výstupu nahradí hvězdičkou. [*verbatim*]

Specifikátory formátu (implicitní formát je „**o2**“):

- **a** Výpis znaků jejich symbolickými jmény.
- **c** Výpis znaků.
- **d**[*s*] Výpis dekadických čísel (*s* značí počet bytů).
- **f**[*s*] Výpis čísel v plovoucí čárce (*s* značí počet bytů).
- **o**[*s*] Výpis oktalových čísel (*s* značí počet bytů).
- **u**[*s*] Výpis dekadických čísel bez znaménka (*s* značí počet bytů).
- **x**[*s*] Výpis hexadecimálních čísel (*s* značí počet bytů).

paste [**-d** *seznam*] *soubor...*
paste -s [**-d** *seznam*] *soubor...*

V první formě příkaz spojí odpovídající řádky vstupních souborů do jedné řádky a oddělí je zadanými oddělovači. Pokud je některý soubor kratší, na místě jeho řádek se používají prázdné řetězce. Ve druhé formě naopak vytvoří z každého zadaného souboru jedinou řádku tak, že jednotlivé řádky souboru spojí za sebe.

Implicitním oddělovačem je tabulátor, pomocí přepínače „**-d**“ je možné zadat alternativní seznam oddělovačů. Seznam může tvořit několik různých znaků, které příkaz postupně střídá. Pro každou novou řádku výstupu se seznam začne brát znovu od začátku. V seznamu znaků je možné používat zvláštní escape-sekvence: „**\n**“ (znak konce řádky), „**\t**“ (tabulátor), „****“ (zpětné lomítko), „**\0**“ (prázdný řetězec).

printf *formát* [*parametr*]...

Příkaz vypíše na standardní výstup text daný formátovacím řetězcem, doplněný na místě formátovacích direktiv hodnotami parametrů. Formátovací řetězec může obsahovat běžné escape-sekvence („**\n**“, „**\r**“, „**\t**“ apod.). Seznam formátovacích direktiv je uveden na str. 348.

Pokud je parametrem číslo, lze použít i hexadecimální zápis („**0x***číslo*“).

V řetězci, který je parametrem pro „**%b**“, je možno používat escape-sekvence.

ps [*přepínače*] [**-o** *formát*]...

Příkaz vypíše informace o procesech. Implicitně vypisuje základní informace o procesech aktuálního uživatele.

Přesná skladba vypisovaných sloupců se na různých systémech liší, pro automatické zpracování výstupu přenositelným způsobem je vhodné používat přepínač „**-o**“.

Přepínače (hodnoty přepínačů mají formát seznamů, položky jsou odděleny čárkami nebo mezerami):

-A Výpis všech procesů v systému. [*all*]

-G *skupiny*
Výpis procesů patřících některé skupině ze seznamu. [*group*]

-o *formát*
Seznam identifikátorů sloupců (viz níže), jejichž výpis je požadován. Je přípustné seznam zadat i pomocí více výskytů přepínače. [*output*]

-p *procesy*
Výpis procesů se zadanými čísly (PID). [*process*]

-t *terminály*
Výpis procesů běžících na některém terminálu ze seznamu. [*terminal*]

-U *uživatelé*
Výpis procesů patřících některému uživateli ze seznamu. [*user*]

Nejdůležitější identifikátory sloupců:

user	vlastník procesu
pid	číslo (PID) procesu
ppid	PID rodičovského procesu
tty	terminál, na němž proces běží
comm	jméno příkazu
args	jméno příkazu a parametry
time	spotřebovaný čas CPU
etime	uplynulý čas od startu procesu

pwd [**-P**]

Příkaz vypíše cestu k aktuálnímu pracovnímu adresáři.

Přepínače:

-P Vypisuje se skutečná cesta, žádná položka není symbolický link. [*physical*]

read ...

Příkaz přečte jednu řádku vstupu a obsah přiřadí proměnným.

Norma umožňuje implementovat příkaz jako externí utilitu, většinou je to ale interní příkaz shellu a my jej tak také v přehledu vedeme (viz str. 343).

rm [**-rf**] *soubor...*

Příkaz smaže zadané soubory.

Přepínače:

-f Program nevyžaduje žádné potvrzení operace (např. v případě mazání souboru, k němuž uživatel nemá práva, ačkoliv má práva k adresáři) a nepíše žádné chyby v případě, že soubory neexistují. [*force*]

-r Je-li mezi parametry adresář, program se pokusí smazat celý podstrom (bez přepínače se mazání adresáře nepovede). [*recursive*]

rmdir *cesta...*

Příkaz smaže adresáře zadaných jmen, pokud jsou prázdné.

sleep *sekundy*

Příkaz pozastaví běh procesu na zadaný počet sekund.

sort [**-m**] [**-o** *výstup*] [**-bdfnru**] [**-t** *znak*] [**-k** *klíč*]... [*soubor*]...
sort -c [**-bdfnru**] [**-t** *znak*] [**-k** *klíč*] [*soubor*]

Příkaz třídí řádky vstupních souborů dle zadaných kritérií do výstupního proudu.

Přepínače:

-b Program ignoruje mezery na začátku třídicích polí. [*blank*]

-c Program pouze kontroluje uspořádání a nastavuje návratovou hodnotu. [*check*]

-d Program ignoruje nemezerové nealfanumerické znaky. [*dictionary*]

-f Program ignoruje rozdíly mezi malými a velkými písmeny. [*fold*]

-k Definice třídicího klíče. Pokud je klíčů více, mají sestupnou prioritu. [*key*]

-m Program soubory pouze slévá (musejí již být setříděné). [*merge*]

-n Program bere v třídicím klíči do úvahy pouze úvodní část, kterou lze interpretovat jako číslo, a třídí numericky. [*number*]

-o Výstup se ukládá do souboru zadaného jména (smí to přitom být jeden ze vstupních souborů). [*output*]

-r Program třídí řádky sestupně. [*reverse*]

-t Jako oddělovač polí se bere *znak* (implicitně je oddělovačem posloupnost bílých znaků).

-u Program třídí unikátně – ze všech řádek se shodným klíčem vypisuje jen jednu, v případě zadání přepínače „**-c**“ kontroluje, zda soubor neobsahuje duplicitní klíče. [*unique*]

Definice třídicího klíče: *pole_zač*[**.***prv_znak*][*typ*][**,***pole_konec*[**.***posl_znak*][*typ*]]

pole_zač	Pořadové číslo pole, kde začíná třídicí klíč.
prv_znak	Pořadové číslo znaku (v rámci pole), kde začíná třídicí klíč.
pole_konec	Pořadové číslo pole, kde končí třídicí klíč.
posl_znak	Pořadové číslo znaku (v rámci pole), kde končí třídicí klíč.
typ	Typ klíče (viz přepínače: **b**, **d**, **f**, **n** a **r**).

Pořadová čísla se počítají od jedničky.

split [**-l** *počet*] [**-a** *délka*] [*soubor* [*jméno*]]
split -b *velikost* [**-a** *délka*] [*soubor* [*jméno*]]

Příkaz rozdělí *soubor* na potřebný počet souborů dané velikosti (implicitně 1000 řádek). Jména nových souborů jsou tvořena připojením posloupnosti písmen malé anglické abecedy (o implicitní délce dva znaky) k řetězci *jméno* (tj. „*jméno*`aa`“, „*jméno*`ab`“ atd.). Pokud program dokončí zápis posledního souboru („*jméno*`zz`“), aniž by dočetl zdroj, skončí předčasně. Implicitní hodnotou pro *jméno* je řetězec „`x`“.

Přepínače:

-a	Zadání délky přípony.	[*appendix*]
-b	Zadání velikosti cílových souborů pomocí počtu bytů. Parametr může mít příponu „**k**“ (zadání v KB, násobcích 1 024 B) nebo „**m**“ (zadání v MB, násobcích 1 048 576 B).	[*bytes*]
-l	Zadání počtu řádek pro určení velikosti cílových souborů.	[*line*]

tail [**-f**] [**-n** *počet*] [*soubor*]
tail [**-f**] **-c** *počet* [*soubor*]

Příkaz vypisuje konec souboru, resp. standardního vstupu počínaje určeným místem. Implicitně vypisuje posledních 10 řádek.

Přepínače:

-c	Zadání začátku výpisu pomocí počtu znaků místo řádek.	[*chars*]
-f	Program neskončí výpis na konci souboru, ale pokračuje v běhu a vypisuje další znaky ze souboru, jakmile se tam objeví.	[*follow*]
-n	Zadání začátku výpisu pomocí počtu řádek.	[*number*]

Parametr *počet* udává počet řádek či znaků od konce souboru, kde má začít výpis. Může být zadán se znaménkem „**+**“, v tom případě se bod zahájení výpisu počítá od začátku souboru.

tar *operace*[*modifikátor*] [*parametr*] [*soubor*]...

Příkaz provádí operace s archivy souborů. Příkaz není obsažen v poslední verzi (č. 3) normy [1] (je nahrazen příkazem **pax**); popis pochází z předchozí verze (č. 2).

Operace:

- **c** Vytváří nový archiv z vyjmenovaných souborů. [*create*]
- **r** Přidává vyjmenované soubory na konec archivu. [*refresh*]
- **t** Vypisuje obsah archivu. [*table*]
- **u** Přidává vyjmenované soubory na konec archivu, pokud už v něm nejsou nebo byly modifikovány. [*update*]
- **x** Rozbaluje (implicitně všechny, resp. jen vyjmenované) soubory a adresáře z archivu. [*extract*]

Modifikátory:

- **f** Jako *parametr* bude zadán archiv, hodnota „**-**“ znamená standardní vstup nebo výstup. [*file*]
- **v** Vypisuje více informací o prováděné operaci. [*verbose*]

tee [**-a**] [*soubor*]...

Příkaz opisuje standardní vstup na standardní výstup a současně pořizuje kopii do jednoho souboru (příp. jiného počtu souborů).

Přepínače:

- **-a** Výstupní soubory nejsou přepisovány od začátku, ale data se připisují na jejich konec. [*append*]

test [*výraz*]
[[*výraz*] **]**

Příkaz vyhodnotí zadaný výraz a vrátí návratovou hodnotu (0=pravda, 1=nepravda).

Elementární výrazy pro testování souborů:

- **-e** *soubor* [*expression*]
  Výraz se vyhodnotí jako pravda, pokud existuje soubor zadaného jména.
- **-d** *soubor*, **-f** *soubor*, **-L** *soubor*
  Výraz se vyhodnotí jako pravda, pokud existuje soubor zadaného jména a typu (**d**=adresář, **f**=obyčejný soubor, **L**=symbolický link).
- **-s** *soubor* [*size*]
  Výraz se vyhodnotí jako pravda, pokud existuje soubor zadaného jména a jeho velikost je větší než nula.
- **-r** *soubor*, **-w** *soubor*, **-x** *soubor*
  Výraz se vyhodnotí jako pravda, pokud existuje soubor zadaného jména a uživatel má k danému souboru zvolené právo.

Elementární výrazy pro testování řetězců:

-z *řetězec* [*zero*]

Výraz se vyhodnotí jako pravda, pokud zadaný řetězec má délku nula.

[**-n**] *řetězec* [*nonzero*]

Výraz se vyhodnotí jako pravda, pokud zadaný řetězec je neprázdný.

řetězec **=** *řetězec*

Výraz se vyhodnotí jako pravda, pokud zadané řetězce jsou zcela shodné.

řetězec **!=** *řetězec*

Výraz se vyhodnotí jako pravda, pokud se zadané řetězce liší.

číslo **-eq** *číslo* [*equal*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a tato čísla mají stejnou hodnotu.

číslo **-ne** *číslo* [*not-equal*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a tato čísla nemají stejnou hodnotu.

číslo **-lt** *číslo* [*less-than*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a první číslo je menší než druhé.

číslo **-le** *číslo* [*less-than-or-equal*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a tato čísla jsou buď stejná, nebo první je menší než druhé.

číslo **-gt** *číslo* [*greater-than*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a první číslo je větší než druhé.

číslo **-ge** *číslo* [*greater-than-or-equal*]

Výraz se vyhodnotí jako pravda, pokud oba řetězce lze interpretovat jako celá čísla a tato čísla jsou buď stejná, nebo první je větší než druhé.

Operátory pro logické operace:

výraz **-a** *výraz*, *výraz* **-o** *výraz*

Logické spojky „a zároveň“ a „nebo“.

! *výraz*

Logická negace výrazu.

(*výraz* **)**

Uzavření podvýrazu do závorek.

time *utilita* [*parametry...*]

Příkaz vykoná zadaný program a vypíše jím spotřebovaný čas.

touch [**-acm**] [**-t** *čas*] *soubor...*
touch [**-acm**] **-r** *ref_soubor soubor...*

Příkaz změní časové údaje o souborech. Implicitně mění datum a čas poslední modifikace (není-li zadáno „**-a**“ ani „**-m**“) a nastavuje ho na aktuální. Pokud zadané soubory neexistují, program je vytvoří (prázdné).

Přepínače:

-a	Mění se rovněž datum a čas posledního přístupu.	[*access*]
-c	Neexistující soubory se nevytvářejí.	[*create*]
-m	Mění se rovněž datum a čas poslední modifikace.	[*modify*]
-r	Hodnota nastavovaného času se přebírá z referenčního souboru.	[*refer*]
-t	Čas se nastavuje ve formátu [[*RR*]*RR*]*MMDDhhmm*[**.***ss*].	[*time*]

tr [**-cs**] *tabulka1 tabulka2*
tr [**-c**] **-s** *tabulka1*
tr [**-c**] **-d** *tabulka1*
tr [**-c**] **-ds** *tabulka1 tabulka2*

Příkaz kopíruje data ze standardního vstupu na standardní výstup, přičemž provádí konverzi a mazání znaků dle zadaných parametrů. Implicitně provádí konverzi jakéhokoliv znaku uvedeného v první tabulce na znak se stejnou relativní pozicí ve druhé tabulce. V tabulkách lze kromě obyčejných znaků používat osmičkovou formu zápisu znaku (tj. „*****ooo*“), běžné escape-sekvence (např. „****“, „**\t**“, „**\n**“, nebo „**\r**“) a třídy znaků (viz str. 347, např. „**[:lower:]**“). Ve druhé tabulce lze použít konstrukci „**[***x******[*n*]**]**“, jež reprezentuje *n* opakování znaku *x* (resp. opakování až do konce tabulky).

Přepínače:

-c	Komplementární zápis první tabulky. První tabulku tvoří všechny znaky kromě těch, jež jsou uvedeny v parametru.	[*complement*]
-d	Namísto konverze program pouze maže znaky uvedené v první tabulce.	[*delete*]
-s	Kromě ostatních operací utilita ve výstupu slučuje posloupnosti stejných znaků obsažených v posledním parametru (první či druhé tabulce) tak, že na výstup vypisuje pouze jeden.	[*sequence*]

uniq [**-cdu**] [**-f** *počet*] [**-s** *znaky*] [*vstup* [*výstup*]]

Příkaz kopíruje řádky standardního vstupu nebo vstupního souboru na standardní výstup (nebo do zadaného výstupního souboru), přičemž z několika po sobě jdoucích shodných řádek vypisuje jen jednu.

Přepínače:

- **-c** Před každou řádkou je uvedeno číslo, kolikrát se daná řádka ve vstupu vyskytovala. [*count*]
- **-d** Program vypisuje pouze duplicitní řádky. [*duplicite*]
- **-f** Při porovnávání řádek se ignoruje prvních několik polí (oddělovačem polí je posloupnost bílých znaků). [*fields*]
- **-s** Při porovnávání řádek se ignoruje prvních několik znaků (po případných polích přeskočených díky přepínači „**-f**“). [*skip*]
- **-u** Program vypisuje pouze unikátní řádky. [*unique*]

uname [**-amnsrv**]

Příkaz vypisuje jméno operačního systému, při zadání přepínačů i další atributy.

Přepínače:

- **-a** Program vypíše všechny atributy. [*all*]
- **-m** Program vypisuje typ hardware. [*machine*]
- **-n** Program vypisuje jméno počítače. [*name*]
- **-s** Program vypisuje typ operačního systému (implicitní výstup). [*system*]
- **-v** Program vypisuje verzi operačního systému. [*version*]

wc [**-cwl**] [*soubor*]...

Příkaz vypíše na standardní výstup počet znaků, slov (oddělených mezerami) a řádek ve standardním vstupu nebo v jednotlivých souborech (a také souhrnné počty).

Přepínače:

- **-c** Program vypisuje počet znaků. [*chars*]
- **-w** Program vypisuje počet slov. [*words*]
- **-l** Program vypisuje počet řádek. [*lines*]

Pokud je uveden alespoň jeden přepínač, vypisují se pouze údaje vybrané pomocí přepínačů, jinak se vypisují všechny tři.

who
who -q
who am i

Příkaz vypíše na standardní výstup informace o přihlášených uživatelích. Implicitně vypisuje jednu řádku s informacemi pro každého uživatele, S přepínačem „**-q**“ vypisuje pouze jména a počet uživatelů. Poslední varianta vypisuje informace pouze o uživateli, který příkaz vyvolal.

xargs [**-t**] [**-L** *počet*] [**-n** *počet*] [**-s** *délka*] [*program* [*parametr*]...]
xargs [**-t**] **-I** *řetězec* [*program* [*parametr*]...]

Příkaz připravuje příkazovou řádku pro volání utility *program* a utilitu poté zavolá. Jako parametry používá jednak svoje vlastní parametry (následující za názvem utility) a jednak obsah svého standardního vstupu. Implicitně konstruuje maximálně tak dlouhou řádku s tolika parametry, aby šla spustit, ale rozsah příkazové řádky lze pomocí přepínačů omezit. Příkaz volá utilitu opakovaně, dokud nedosáhne konce standardního vstupu. Konec příkazu nastane rovněž tehdy, když volaná utilita skončí s návratovým kódem 255. Pokud není uveden název utility, spouští se příkaz **echo**.

Přepínače:

- **-I** Program spouští příkazy po jednotlivých řádkách. Text vstupní řádky je přitom dosazen do vytvářené řádky na místa označená v parametrech pomocí vybraného řetězce. [*insert*]
- **-L** Program spouští příkazy po skupinách řádek. [*line*]
- **-n** Program spouští příkazy po skupinách slov přečtených ze vstupu. [*number*]
- **-s** Program omezuje tvorbu řádky tak, aby nepřesáhla danou velikost. [*size*]
- **-t** Program vypisuje text prováděných příkazů. [*trace*]

A.2 Příkaz awk

awk [**-F** *FS*] [**-v** *přiřazení*]... *program* [*parametr*]...
awk [**-F** *FS*] [**-v** *přiřazení*]... **-f** *soubor* [**-f** *soubor*]... [*parametr*]...

Příkaz čte soubory zadané v parametrech (implicitně standardní vstup) a vykonává program (skript) napsaný v jazyce **awk**. Skript lze uvést jako první parametr na příkazové řádce nebo uložit do jednoho či více souborů a ty pak zadat pomocí přepínače „**-f**“). Parametry příkazu mohou být jednak jména vstupních souborů a jednak požadavky na *přiřazení* hodnoty proměnné ve tvaru „*proměnná*=*hodnota*“. Hodnota se proměnné nastaví bezprostředně před otevřením vstupního souboru, který v parametrech za přiřazením následuje.

Přepínače:

-f	Zadání souboru s programem.	[*file*]
-F	Nastavení oddělovače polí (proměnné **FS**).	[*field separator*]
-v	Inicializace proměnné (s platností už ve větvi **BEGIN**).	[*variable*]

Vstup je chápán jako posloupnost záznamů oddělených definovaným oddělovačem (v proměnné **RS**), implicitně znakem konce řádky. Každý záznam je rozdělen na pole podle definovaného oddělovače polí (proměnná **FS**), implicitně posloupností bílých znaků.

Program se skládá z větví ve tvaru „[*vzor*] [**{** *akce* **}**]“, pro každý záznam se procházejí všechny větve, a pokud je vzor větve platný, provede se akce. Vynechaný vzor znamená, že se akce provádí vždy, vynechaná akce znamená, že se na výstup opíše celý záznam.

Vzory:

BEGIN
: Vzor je platný právě jednou, na začátku práce programu.

END
: Vzor je platný právě jednou, na konce práce programu.

výraz
: Vzor je platný, pokud se výraz vyhodnotí jako pravda.

výraz,*výraz*
: Vzor je platný vždy počínaje okamžikem splnění prvního výrazu a konče okamžikem splnění druhého výrazu.

Výrazy v jazyce **awk** mají vždy textovou hodnotu, ale případě použití v numerickém kontextu se řetězec převede na celé, resp. reálné číslo (s desetinnou tečkou). Řetězce jako konstanty (literály) se zapisují mezi uvozovky a dají se v nich použít běžné escape-sekvence (např. „**\t**“, „**\n**“, „****“ nebo „**\"**“).

Proměnné se označují identifikátorem tvořeným posloupností alfanumerických znaků a podtržítek („_“), začínající písmenem. Malá a velká písmena nejsou totožná, identifikátory „a“ a „A“ označují různé proměnné.

Rozlišují se proměnné dvou typů – *skalární*, které mohou obsahovat jednu textovou hodnotu (řetězec), a *asociativní pole*, obsahující sérii hodnot. Indexem pole může být libovolný řetězec, prvek pole se zapisuje: „*var*`[`*index*`]`“.

Znak konce řádky je jazyce **awk** významový, rozdělit řádku nelze kdekoliv. Možná místa jsou: na konci příkazu, za složenými závorkami, za čárkou mezi parametry funkce či příkazu, za operátory „**&&**“ a „**||**“, za středníky v závorce příkazu „**for**“, za závorkou u příkazů „**if**“, „**for**“ a „**while**“ a za příkazy „**else**“ a „**do**“. Rozdělit řádku na jiném místě lze pomocí pokračovací řádky – na konec řádky se napíše zpětné lomítko a pokračuje se na další řádce.

Přehled nejdůležitějších operátorů (zkratka *lv* označuje výraz, který může stát na levé straně přiřazovacího příkazu, např. proměnnou, prvek pole, odkaz na vstupní pole):

(*výraz***)**

Uzavření výrazu do závorek.

$*výraz*

Odkaz na pole vstupního záznamu. Výraz **$1** představuje první pole atd. Výraz **$0** představuje celý vstupní záznam.

*lv***++**, *lv***--**

Postinkrement, postdekrement. Hodnotou výrazu je obsah *lv*, vedlejším efektem je zvýšení (snížení) hodnoty *lv* o jedničku.

++*lv*, **--***lv*

Preinkrement, predekrement. Výraz způsobí zvýšení (snížení) hodnoty *lv* o jedničku, hodnotou výrazu je nový obsah *lv*.

! *výraz*

Negace výrazu.

výraz ***** *výraz*, **/**, **%**, **+**, **-**

Binární aritmetické operátory (**%**=zbytek po dělení).

výraz *výraz*

Zřetězení výrazů (spojení jejich obsahu bezprostředně za sebe).

výraz **==** *výraz*, **!=**, **<**, **<=**, **>**, **>=**

Porovnání výrazů na rovnost, nerovnost atd.

[*výraz* **~**] **/***ERE***/**, **!~**

Porovnání výrazu na shodu (resp. neshodu) s regulárním výrazem (ERE, viz str. 346). Samostatně stojící „**/***ERE***/**“ se chápe jako „**$0 ~ /***ERE***/**“.

výraz **&&** *výraz*, **||**

Logické spojky „a zároveň“ a „nebo“.

lv **=** *výraz*

Přiřazovací operátor. Hodnota výrazu uvedeného na pravé straně se přiřadí do *lv*. Hodnotou celého výrazu je přiřazovaná hodnota.

lv ***=** *výraz*, **/=**, **%=**, **+=**, **-=**

Multiplikativní a aditivní operátory přiřazení. S hodnotami *lv* a *výraz* se provede naznačená aritmetická operace a výsledek je jednak hodnotou celého výrazu, jednak se přiřadí do *lv*.

Přehled nejdůležitějších vestavěných proměnných:

ARGC Počet parametrů příkazové řádky (resp. pole **ARGV**). [*arg count*]

ARGV Pole odkazů na parametry příkazové řádky (číslované od 1). [*arg values*]

ENVIRON [*environment*]

Pole proměnných prostředí (indexované jmény proměnných).

FS Oddělovač polí vstupního záznamu (implicitně mezera). [*field separator*]

- Pokud je **FS** rovno mezeře, oddělovačem je posloupnost bílých znaků.
- Pokud je ve **FS** jiný znak, oddělovačem je každý výskyt tohoto znaku.
- Pokud je **FS** delší, chápe se jako ERE a oddělovačem je každý výskyt regulárního výrazu v záznamu.

NF Počet polí nalezených v aktuálním vstupním záznamu. [*number of fields*]

NR Číslo aktuálně zpracovávaného záznamu. [*number of record*]

OFS Oddělovač parametrů příkazu **print** (implicitně mezera). [*output FS*]

ORS Oddělovač záznamů příkazu **print** (impl. znak konce řádky). [*output RS*]

RS Oddělovač vstupních záznamů (jeden znak, implicitně znak konce řádky). Pokud je hodnotou proměnné **RS** prázdný řetězec, za záznam se považuje blok řádek ukončený (alespoň jednou) prázdnou řádkou.[*record separator*]

Řídicí příkazy:

{*příkaz...***}**

Složený příkaz.

if(*výraz***)** *příkaz*

if(*výraz***)** *příkaz***; else** *příkaz*

Podmíněný příkaz.

for(*inicializace***;** *podmínka***;** *krok***)** *příkaz*

Příkaz cyklu. Vyhodnotí se výraz *inicializace*, a dokud platí *podmínka*, opakuje se provádění těla cyklu a vyhodnocení výrazu *krok*.

for(*proměnná* **in** *pole***)** *příkaz*

Příkaz cyklu. Všechny hodnoty indexů v poli se postupně dosadí do řídicí proměnné a pro každý se provede tělo cyklu.

while(*podmínka***)** *příkaz*

Příkaz cyklu. Dokud platí *podmínka*, provádí se tělo cyklu.

do *příkaz*; **while**(*podmínka*)

Příkaz cyklu. Dokud platí podmínka, provádí se příkaz v těle cyklu. Na rozdíl od cyklu **while** se příkaz provede vždy alespoň jednou.

break

Ukončení cyklu.

continue

Ukončení aktuálního kroku cyklu, zahájení nového kroku.

next

Ukončení práce se záznamem a zahájení nového kroku, tj. načtení dalšího záznamu a procházení seznamem větví od počátku programu.

exit [*výraz*]

Ukončení běhu programu. Pokud je příkaz vyvolán ve větvi **END**, ukončí se činnost programu, jinak se pokračuje větví **END** (pokud existuje). Má-li příkaz parametr, použije se k nastavení návratové hodnoty.

Výstupní příkazy:

print [*parametr* [**,** *parametr*]...]

Příkaz vypisuje hodnoty parametrů, odděluje je hodnotou proměnné **OFS**. Bez parametrů vypisuje obsah **$0**. Na závěr vypíše obsah proměnné **ORS**.

printf *formát* [**,** *parametr*]...

Příkaz vypisuje obsah formátovacího řetězce a doplňuje postupně na místa formátovacích direktiv hodnoty parametrů. Přehled direktiv viz str. 348.

příkaz **>** *výraz*

Příkaz (**print** nebo **printf**) vypisuje text do souboru, jehož jméno je hodnotou výrazu. Soubor se před provedením prvního zápisu vytvoří, resp. vymaže, další volání již zapisují na konec souboru.

příkaz **>>** *výraz*

Příkaz vypisuje text na konec souboru, jehož jméno je hodnotou výrazu.

příkaz **|** *výraz*

Příkaz vypisuje text do roury. Proces konzumenta je nastartován pomocí příkazu a parametrů, které jsou obsahem výrazu.

Vstupní příkazy:

getline

Příkaz načte ze vstupu nový záznam do proměnné **$0**, rozdělí ho na pole, patřičně nastaví hodnotu proměnné **NF** a zvýší hodnotu proměnné **NR**.

getline *proměnná*

Příkaz načte do proměnné ze vstupu nový záznam. Proměnné svázané se vstupním záznamem (**NF**, **$0**, **$1** atd.) se nemění, zvýší se jen hodnota **NR**.

getline ... **<** *výraz*

Příkaz **getline** přečte záznam ze souboru, jehož jméno je hodnotou výrazu, soubor se před provedením prvního čtení otevře.

výraz **|** **getline** ...

Příkaz **getline** přečte záznam z roury, před provedením prvního čtení se pro danou rouru spustí proces s příkazem daným výrazem.

Vybrané funkce:

index(*s***,** *t***)**

Funkce vrací pozici prvního výskytu řetězce *t* v řetězci *s* (číslováno od jedničky), resp. nulu, pokud se v něm řetězce *t* nevyskytuje.

length([*x*]**)**

Funkce vrací délku řetězce *x* (implicitně délku **$0**).

substr(*text***,** *začátek*[**,** *délka*]**)**

Funkce vrací podřetězec vybraný z řetězce *text* od pozice *začátek* (číslováno od jedničky) o délce *délka* (implicitně až do konce).

split(*text***,** *pole*[**,** *fs*]**)**

Funkce rozdělí řetězec *text* na pole podle oddělovače *fs* (implicitně se použije hodnota proměnné **FS**) a vrací jejich počet. Jednotlivá pole se uloží do prvků *pole* (*pole***[1]**, *pole***[2]**,...).

sprintf(*formát***,** *parametr***,**...**)**

Funkce vrací řetězec vzniklý doplněním formátovacího řetězce o hodnoty parametrů podobně, jako to dělá příkaz **printf**.

sub(*vzor***,** *text*[**,** *lv*]**)**

Funkce mění obsah proměnné *lv* (implicitně **$0**) tak, že nahradí řetězcem *text* první výskyt podřetězce vyhovujícího regulárnímu výrazu *vzor*.

gsub(*vzor***,** *text*[**,** *lv*]**)**

Funkce mění obsah proměnné *lv* (implicitně **$0**) tak, že nahradí řetězcem *text* každý výskyt podřetězce vyhovujícího regulárnímu výrazu *vzor*.

match(*text***,** *vzor***)**

Funkce vrací pozici prvního výskytu podřetězce vyhovujícího regulárnímu výrazu *vzor* v řetězci *text*, resp. nulu, pokud se v něm takový nevyskytuje.

int(*x***)**

Funkce vrací celou část desetinného čísla.

sin(*x***)**, **cos(***x***)**, **exp(***x***)**, **log(***x***)**, **sqrt(***x***)**, **atan2(***x***)**

Matematické funkce s plovoucí čárkou (s reálnými čísly).

rand()

Generátor náhodných čísel, funkce vrací číslo *n* z rozsahu $0 <= n < 1$.

srand([*x*]**)**

Funkce inicializuje generátor hodnotou *x* nebo hodnotou aktuálního času.

Kromě vestavěných funkcí lze definovat a používat vlastní funkce. Definice:

function *jméno***(**[*param***,**...]**) { ***příkazy***; return** *výraz***; }**

A.3 Příkaz `ed`

`ed` [**`-s`**] [*soubor*]

Příkaz si vytvoří dočasnou kopii zadaného souboru a na ní provádí editační příkazy, které čte ze standardního vstupu. Pokud se výsledek provedené editace má uložit zpět do souboru, je třeba použít příkaz *write* („**`w`**“).

Přepínače:

`-s` Potlačení pomocných informativních výpisů. [*suppress*]

Formát příkazů:

[*adresa*[**`,`***adresa*]]*příkaz*[*parametr*]

Každé složce příkazu může předcházet libovolně mnoho mezer. Adresa udává řádku, se kterou příkaz pracuje. V případě udání dvou adres příkaz pracuje s intervalem řádek (včetně obou mezí). V případě vynechání adresy je implicitní hodnotou obvykle aktuální řádka (viz níže), výjimky jsou uvedeny v přehledu příkazů.

Editor si pamatuje polohu v rámci souboru – tzv. *aktuální řádku*. Obvykle je to poslední řádka, se kterou pracoval poslední příkaz, po otevření souboru je to poslední řádka souboru.

Formát adresy:

`.` aktuální řádka

`$` poslední řádka souboru

n řádka číslo *n*

`/`*BRE***`/`**

následující řádka, na níž se nachází text vyhovující *BRE* (viz str. 346)

`?`*BRE***`?`**

předcházející řádka, na níž se nachází text vyhovující *BRE*

`'`*z* řádka s nastavenou značkou *z* (viz příkaz **`k`**)

`+`[*n*], **`-`**[*n*]

řádka s relativní vzdáleností *n* (impl. 1) vůči aktuální řádce

Za adresou může následovat (příp. vícekrát) relativní posun ve tvaru **`+`**[*n*], **`-`**[*n*].

Konvence pro popis některých parametrů v následujícím přehledu příkazů:

soubor Název souboru, s nímž mají pracovat příkazy jako jsou **`E`**, **`f`**, **`r`** nebo **`w`**. Pokud u takového příkazu hodnotu parametru vynecháme, použije se jméno právě editovaného souboru.

text Nový text vkládaný příkazy **`a`**, **`c`** a **`i`**. Blok nových řádek se ukončuje řádkou se samotnou tečkou. Žádná z vkládaných řádek nesmí obsahovat samotnou tečku.

Přehled nejdůležitějších příkazů:

[*x*]**a** [*append*]

text

. Vložení textu za řádku s adresou *x*.

[*x*[**,***y*]]**c** [*change*]

text

. Nahrazení řádek daného intervalu novým textem.

[*x*[**,***y*]]**d** [*delete*]

Smazání řádek (aktuální řádkou se stane následující řádka).

E[*soubor*] [*Edit*]

Zrušení editované kopie, zahájení editace nového souboru.

f[*soubor*] [*file*]

Změna jména editovaného souboru uloženého v paměti editoru. Tato nová hodnota se bude používat jako implicitní hodnota pro příkazy pracující se jménem souboru (např. **w**). Vedlejším efektem je vypsání (nového) jména souboru na standardní výstup. Vykonání příkazu se nijak neprojeví ani na obsahu editované kopie souboru, ani na původním souboru na disku.

[*x*[**,***y*]]**g/***BRE***/**[*příkazy*] [*global*]

Provedení sady příkazů na řádky, které vyhovují *BRE*.

Implicitní rozsah řádek je „**1,$**“, oddělovačem *BRE* místo lomítka může být libovolný znak, který není bílý, všechny *příkazy* kromě posledního musejí být ukončeny zpětným lomítkem.

[*x*]**i** [*insert*]

text

. Vložení textu před řádku s adresou *x*.

[*x*[**,***y*]]**j** [*join*]

Spojení intervalu řádek do jediné řádky. Implicitně spojí aktuální řádku s následující.

[*x*]**k***z* [*mark*]

Nastavení značky *z* na řádku *x*. Značka je reprezentovaná písmenem malé anglické abecedy.

[*x*[**,***y*]]**l** [*list*]

Výpis řádek na standardní výstup, netisknutelné znaky se vypisují buď obvyklými escape-sekvencemi, nebo oktalovou hodnotou („*oct*“, pokud patřičná sekvence neexistuje).

[*x*[**,***y*]]**m***z* [*move*]

Přesun řádek za řádku s adresou *z*.

[*x*[**,***y*]]**n** [*number*]

Výpis řádek doplněných o číslo řádky na standardní výstup.

[*x*[**,***y*]]**p** [*print*]

Výpis řádek na standardní výstup.

Q [*Quit*]

Ukončení editace.

[*x*]**r**[*soubor*] [*read*]

Načtení obsahu zadaného souboru a vložení za řádku *x* (implicitně za poslední řádku).

[*x*[**,***y*]]**s/***BRE***/***náhrada***/***příznaky* [*substitute*]

Náhrada textu vyhovujícího *BRE* za nový text.

V náhradě je možno použít zpětné reference regulárních výrazů a také metaznak **&** reprezentující celý text pokrytý *BRE*.

Příznaky: *n* (náhrada *n*-tého výskytu místo prvního), **g** (*global*, náhrada všech výskytů), **p**, **l**, **n** (výpis řádky po provedení náhrady příkazem **p**, **l**, resp. **n**).

[*x*[**,***y*]]**t***z* [*transfer*]

Vložení kopie řádek za řádku s adresou *z*.

u [*undo*]

Zrušení efektu posledního provedeného editačního příkazu.

[*x*[**,***y*]]**v/***BRE***/**[*příkazy*] [*invert*]

Provedení sady příkazů na řádky, které nevyhovují *BRE*.

Implicitní rozsah řádek je „**1,$**“, oddělovačem *BRE* místo lomítka může být libovolný znak, který není bílý, všechny *příkazy* kromě posledního musejí být ukončeny zpětným lomítkem.

[*x*[**,***y*]]**w**[*soubor*] [*write*]

Zápis vybraného rozsahu řádek (implicitně celého souboru) do souboru daného jména.

[*x*]**=**

Výpis čísla řádky (implicitně poslední).

A.4 Příkaz `find`

`find` [**`-HL`**] *kořen... parametr...*

Příkaz prochází podstrom (resp. podstromy) souborového systému zadaný kořenem a hledá soubory vyhovující kritériím zadaným jako posloupnost parametrů tvořící logický výraz.

Přepínače:

- **`-H`** Pokud je některý *kořen* symbolický link, začíná se cílovým souborem.
- **`-L`** Pokud program narazí při práci na symbolický link, pracuje místo něj jeho s cílovým souborem.

Logické spojky:

- **`(`**, **`)`** závorky
- **`!`** negace
- **`-a`** konjunkce (a zároveň), spojku lze vynechat
- **`-o`** disjunkce (nebo)

Konvence pro popis některých parametrů v přehledu elementárních výrazů:

- *vzor* se zapisuje pomocí expanzních znaků shellu („**`*`**“, „**`?`**“, „**`[]`**“), expanzní znaky přitom zastupují libovolný znak (včetně teček a lomítek)
- *n* zápis vyžadující přesnou shodu číselné hodnoty; před hodnotu je možno zapsat znak „**`+`**“, pak znamená „více než“ nebo znak „**`-`**“ („méně než“)

Vybrané elementární výrazy (termín „cesta“ označuje cestu od konkrétního kořene až po jméno souboru včetně):

`-xdev`
: vždy pravda; způsobí, že program nepokračuje v prohledávání, pokud by měl opustit souborový systém, na němž leží *kořen*

`-prune`
: vždy pravda; způsobí, že program nepokračuje v prohledávání, pokud soubor je adresář

`-depth`
: vždy pravda; způsobí, že v případě nalezení adresáře program nejprve zpracuje veškeré soubory v adresáři a teprve poté vlastní adresář (probíhá prohledávání do hloubky)

`-name` *vzor*
: vyhodnocen jako pravda, pokud jméno souboru vyhovuje vzoru

`-path` *vzor*
: vyhodnocen jako pravda, pokud cesta k souboru vyhovuje vzoru

-perm *nastavení_práv*

vyhodnocen jako pravda, pokud se nastavení práv souboru přesně shoduje s argumentem; formát argumentu je shodný s formátem odpovídajícího parametru příkazu **chmod**

-perm **-***nastavení_práv*

vyhodnocen jako pravda, pokud soubor má nastavena alespoň ta práva, která jsou uvedena v argumentu; formát argumentu je shodný s formátem odpovídajícího parametru příkazu **chmod**

-type [**bcdflps**]

vyhodnocen jako pravda, pokud písmeno zadaného typu odpovídá typu souboru (**f**=obyč. soubor, ostatní písmena odpovídají hodnotě vypisované příkazem **ls -l**)

-links *n*

vyhodnocen jako pravda, pokud soubor má *n* hardlinků

-user *uživatel*

vyhodnocen jako pravda, pokud vlastníkem souboru je *uživatel*; argument může být UID nebo jméno

-group *skupina*

vyhodnocen jako pravda, pokud skupinovým vlastníkem souboru je *skupina*; argument může být GID nebo jméno

-size *n*[**c**]

vyhodnocen jako pravda, pokud velikost souboru je *n*; velikost se zadává v blocích o velikosti 512 B, resp. v bytech (je-li použita přípona „**c**“)

-mtime *n*

vyhodnocen jako pravda, pokud se okamžik startu programu liší od data a času poslední modifikace souboru o *n* (celých) dní

-atime *n*

dtto pro čas posledního přístupu k souboru

-newer *soubor*

vyhodnocen jako pravda, pokud soubor je „mladší“ (má novější datum a čas poslední modifikace) než *soubor*

-exec *příkaz* [*parametr*]... **;**

výraz způsobí spuštění zadaného příkazu (pro každý nalezený soubor) a je vyhodnocen jako pravda, pokud příkaz skončí návratovým kódem nula; pokud některý parametr má tvar „**{}**“, bude nahrazen cestou k souboru

-print

vždy pravda; způsobí vypsání cesty k souboru

A.5 Příkaz `sed`

`sed` [**`-n`**] *program* [*soubor*]...
`sed` [**`-n`**] [**`-e`** *příkazy*]... [**`-f`** *soubor*]... [*soubor*]...

Utilita („stream editor“) čte postupně řádky standardního vstupního proudu nebo zadaných souborů, provádí operace uvedené v programu (skriptu) a výsledek editace vypisuje na svůj standardní výstup (nebyl-li zadán přepínač „**`-n`**“). Skript se zadává buď jako první parametr na příkazové řádce, nebo pomocí jednoho či více přepínačů „**`-e`**“, resp. „**`-f`**“.

Jeden krok pracovního cyklu vypadá tak, že editor načte řádku ze vstupu, uloží ji do *pracovního prostoru* (bez znaku konce řádky), poté prochází jednotlivé příkazy programu a modifikuje obsah pracovního prostoru. Po dosažení konce skriptu vypíše výsledek editace (obsah pracovního prostoru) na standardní výstup (nebyl-li zadán přepínač „**`-n`**“). Tím je krok ukončen a editor se vrací se na začátek dalšího kroku cyklu (načte novou řádku a provádí příkazy od začátku skriptu).

Kromě pracovního prostoru má editor ještě k dispozici tzv. *odkládací prostor* a může přesouvat data mezi oběma prostory.

Přepínače:

- **`-e`** Zadání textu programu [*expression*]
- **`-f`** Zadání souboru s programem [*file*]
- **`-n`** Potlačení výpisu výsledku editace; na výstup se vypisuje pouze takový text, pro nějž to bylo požadováno explicitním příkazem [*nooutput*]

Formát příkazů:

[*adresa*[**`,`***adresa*]]*příkaz*[*parametr*]

Každé složce příkazu může předcházet libovolně mnoho mezer. Adresa udává řádku, se kterou příkaz pracuje. V případě udání dvou adres příkaz pracuje s intervalem řádek (včetně obou mezí). V případě vynechání adres příkaz platí pro každou řádku. Příkazy se standardně oddělují znakem konce řádky, za jednoduššími příkazy (těmi, které nemají parametry nebo je mají jasně omezené) lze jako oddělovač příkazů použít středník.

Formát adresy:

- **`$`** příkaz se provede na poslední řádce souboru
- *n* příkaz se provede na řádce číslo *n*
- **`/`***BRE***`/`**
  příkaz se provede, pokud obsah pracovního prostoru vyhovuje *BRE*

Konvence pro popis některých parametrů v následujícím přehledu příkazů:

návěští Symbolické označení místa v programu, na něž lze skákat pomocí příkazů **b** a **t**. Doporučovaná maximální délka návěští je 8 znaků. Znaková sada pro volbu názvu není nijak omezena, obecně lze asi doporučit používání pouze alfanumerických znaků a rozhodně je vhodné se vyhnout znakům jako mezera, středník apod.

text Nový text vkládaný příkazy **a**, **c** a **i**. Pokud je řádek více, zapisují se tak, že všechny kromě poslední musejí končit zpětným lomítkem.

Přehled příkazů:

*komentář*

Komentář.

: *návěští*

Definice návěští. Na takto označená místa programu je možno skákat pomocí příkazů skoku *branch* a *test*.

[*x*[**,***y*]]**{** [*příkaz*]
příkaz...
}

Složený příkaz. Na řádkách, které vyhovují zadanému adresnímu rozsahu, se postupně provádějí příkazy, které jsou součástí složeného příkazu.

[*x*]**a** [*append*]

text Vložení textu za řádku s adresou *x*. Text se do výstupu zapíše těsně před pokusem o načtení další řádky.

[*x*[**,***y*]]**b** [*návěští*] [*branch*]

Skok na návěští. Pokud návěští není uvedeno, skočí se na konec skriptu.

[*x*[**,***y*]]**c** [*change*]

text Nahrazení řádek novým textem. Zadaný rozsah řádek se vymaže a poté se na standardní výstup vypíše nový text a editor zahájí nový cyklus.

[*x*[**,***y*]]**d** [*delete*]

Smazání řádek. Editor vymaže obsah pracovního prostoru a zahájí nový krok cyklu.

[*x*[**,***y*]]**D** [*Delete*]

Smazání první řádky prostoru. Editor vymaže obsah pracovního prostoru až po první znak konce řádky (včetně) a zahájí nový krok cyklu.

[*x*[**,***y*]]**g** [*get*]

Do pracovního prostoru se uloží obsah odkládacího prostoru.

[*x*[**,***y*]]**G** [*Get*]

K obsahu pracovního prostoru se připojí znak konce řádky a obsah odkládacího prostoru.

[*x*[**,***y*]]**h** [*hold*]

Do odkládacího prostoru se uloží obsah pracovního prostoru.

[*x*[**,***y*]]**H** [*Hold*]

K obsahu odkládacího prostoru se připojí znak konce řádky a obsah pracovního prostoru.

[*x*]**i** [*insert*]

text Vložení textu před řádku s adresou *x*. Text se do výstupu zapíše, jakmile **sed** narazí na příkaz *insert*.

[*x*[**,***y*]]**l** [*list*]

Výpis řádek na standardní výstup, netisknutelné znaky se vypisují buď obvyklými escape-sekvencemi, nebo oktalovou hodnotou („*oct*“, pokud patřičná sekvence neexistuje).

[*x*[**,***y*]]**n** [*next*]

Výpis obsahu pracovního prostoru na standardní výstup (nebyl-li zadán přepínač „**-n**“), načtení nové řádky do pracovního prostoru a pokračování následujícím příkazem skriptu.

[*x*[**,***y*]]**N** [*Next*]

Načtení nové řádky a připojení na konec pracovního prostoru. Nová řádka bude od dosavadního obsahu oddělena znakem konce řádky.

[*x*[**,***y*]]**p** [*print*]

Výpis obsahu pracovního prostoru na standardní výstup.

[*x*[**,***y*]]**P** [*Print*]

Výpis obsahu pracovního prostoru až po první znak konce řádky (včetně) na standardní výstup.

[*x*]**q** [*quit*]

Ukončení editoru. Pokud nebyl zadán přepínač „**-n**“, vypíše se ještě obsah pracovního prostoru. Pokud se na aktuální řádce uplatnil nějaký příkaz **a**, provede se rovněž výpis vloženého textu.

[*x*]**r** *soubor* [*read*]

Načtení obsahu zadaného souboru a vložení za řádku *x*. Text se do výstupu zapíše těsně před načtením následující řádky.

[*x*[**,***y*]]**s/***BRE***/***náhrada***/***příznaky* [*substitute*]

Náhrada textu vyhovujícímu *BRE* za nový text.

V náhradě je možno použít zpětné reference regulárních výrazů a také metaznak **&** reprezentující celý text pokrytý BRE.

Příznaky: *n* (náhrada *n*-tého výskytu místo prvního), **g** (náhrada všech výskytů), **p** (výpis řádky po provedení náhrady), **w** *soubor* (výpis řádky po provedení náhrady do souboru).

[*x*[**,***y*]]**t** [*návěští*] [*test*]

Podmíněný skok na návěští. Pokud návěští není uvedeno, skočí se na konec skriptu. Podmínkou pro provedení skoku je to, zda se od načtení poslední vstupní řádky nebo od provedení posledního příkazu *test* vykonala nějaká náhrada příkazem *substitute*.

[*x*[**,***y*]]**w** *soubor* [*write*]

Výpis obsahu pracovního prostoru na konec zadaného souboru.

[*x*[**,***y*]]**x** [*exchange*]

Záměna obsahů pracovního a odkládacího prostoru.

[*x*]**=**

Výpis čísla řádky na standardní výstup.

A.6 Příkaz `sh` a jazyk shellu

sh [*přepínače*] **-c** *řetězec* [*jméno* [*parametr*]...]

sh [*přepínače*] *skript* [*parametr*]...

sh [*přepínače*] [**-s** [*parametr*]...]

Příkaz čte ze zadaného zdroje (řetězce, skriptu nebo svého standardního vstupu) text, provádí s ním úpravy podle pravidel jazyka shellu a vykonává nalezené příkazy.

Shell lze zavolat třemi způsoby:

První varianta (s přepínačem „**-c**“) očekává příkazy v prvním parametru, další případné parametry se dosadí do **$0**, **$1**, **$2** atd. [*command*]

Druhá varianta čte příkazy ze souboru zadaného prvním parametrem, jeho jméno dosadí do **$0**, další případné parametry do **$1**, **$2** atd.

Poslední varianta čte příkazy ze vstupu. Přepínač „**-s**“ není nutno uvádět, pokud nejsou zadány další parametry (hodnoty pro **$1**, **$2** atd.). [*standard input*]

Nejdůležitější přepínače režimů práce:

-a Všechny proměnné jsou automaticky exportovány. [*allexport*]

-C Shell nedovolí přepsat existující soubor operátorem přesměrování, je třeba použít speciální operátor „**>|**“. [*noclobber*]

-e Pokud jakýkoliv příkaz (vyjma těch, jež jsou testovány v nějaké řídicí konstrukci) vrátí nenulový návratový kód, shell okamžitě skončí. [*errexit*]

-f Shell neprovádí expanzi jmen souborů, tzv. režim *noglob*. [*filenames*]

-n Shell pouze čte příkazy a neprovádí je. Slouží pro syntaktickou kontrolu skriptů. [*noexec*]

-u Pokus o substituci nenastavené proměnné ukončí běh shellu. [*nounset*]

-v Shell vypisuje na standardní chybový výstup řádky, které přečte ze vstupu a chystá se je interpretovat. [*verbose*]

-x Shell vypisuje na standardní chybový výstup příkazy, ve tvaru, v jakém je bude provádět (*execute trace*). [*xtrace*]

Nejdůležitější proměnné prostředí:

HOME Domovský adresář uživatele.

IFS Seznam oddělovačů polí na příkazové řádce, resp. na vstupní řádce čtené příkazem **read**. [*internal field separator*]

LC_ALL Nastavení národního prostředí.

LOGNAME Uživatelské jméno přihlášeného uživatele.

PATH Dvojtečkami oddělený seznam adresářů, kde shell hledá spustitelné programy zadané jako příkaz pouze jménem.

PS1 Obsah promptu příkazové řádky. [*prompt string*]

PS2 Obsah sekundárního promptu.

TERM Typ terminálu.

Metaznaky pro rušení významu metaznaků (quoting):

**** (zpětné lomítko, backslash)

Následující znak ztrácí zvláštní význam, stává se obyčejným znakem.

' (apostrof)

Všechny znaky až po následující párový apostrof ztrácejí zvláštní význam.

" (uvozovka)

Všechny znaky až po následující párovou uvozovku ztrácejí zvláštní význam, s výjimkou znaku dolaru, zpětného apostrofu a zpětného lomítka.

Metaznaky porovnávacích vzorů:

Používají se pro substituci podřetězců, výběr větve příkazu **case** a pro expanzi jmen souborů (v tom případě nezahrnují znak lomítko a tečku na začátku jména souboru).

? (otazník)

Metaznak zastupuje libovolný znak.

[*seznam***]** (hranaté závorky)
[!*seznam***]**

Metaznak zastupuje jeden libovolný znak ze seznamu (první forma), resp. z doplňku znaků v seznamu (druhá forma). Do seznamu je možno zapsat libovolné znaky nebo třídy znaků (viz str. 347). Pravidla pro zápis metaznaků do seznamu:

znak „**]**“ musí být uveden jako první (po případném metaznaku „**!**“)

znak „**!**“ nesmí být uveden jako první

znak „**-**“ musí být uveden na takovém místě, aby nemohl být považován za metaznak pro interval, tj. buď jako první znak, jako poslední znak, nebo jako první znak bezprostředně za nějakým intervalem

***** (hvězdička)

Metaznak zastupuje libovolný podřetězec.

Řídicí operátory:

; (středník) *příkaz1***;** *příkaz2*

Odděluje příkazy, které se mají provést sekvenčně.

& (ampersand) *příkaz***&**

Ukončuje zápis příkazu, který se má provést na *pozadí* (vstup příkazu je odpojen od terminálu, shell nečeká na skončení příkazu).

() (závorky) **(***příkaz***)**

Oddělují příkaz nebo skupinu příkazů, která se má provést v subshellu.

| (svislítko) *příkaz1* **|** *příkaz2*

Odděluje příkazy, které se mají provést paralelně, standardní výstup prvního je přesměrován na standardní vstup druhého (tzv. *roura*).

&&, **||** *příkaz1* **&&** *příkaz2*

Odděluje příkazy, které se mají provést sekvenčně, a to podmíněně v závislosti na návratovém kódu prvního příkazu (tj. pokud první příkaz uspěje, resp. neuspěje).

Operátory přesměrování:

Pokud operátorům bezprostředně předchází číslo, znamenají přesměrování proudu s daným číslem deskriptoru namísto standardního vstupu, resp. výstupu.

< (menšítko) *příkaz* **<** *soubor*

Přesměrování standardního vstupu příkazu ze souboru, jehož jméno je operandem operátoru (první slovo uvedené za operátorem).

<< *příkaz* **<<** *slovo*

Přesměrování standardního vstupu příkazu z here-dokumentu. Vstupní řádky shellu, které za příkazem následují, až po řádku obsahující slovo uvedené jako operand operátoru (nevčetně), jsou předány na vstup příkazu.

> (většítko) *příkaz* **>** *soubor*

Přesměrování standardního výstupu příkazu do souboru, jehož jméno je operandem operátoru (první slovo za operátorem). Pokud soubor neexistuje, bude vytvořen, pokud existuje (a není přepínačem „**-C**“ zapnut režim noclobber), bude přepsán.

>> *příkaz* **>>** *soubor*

Přesměrování standardního výstupu příkazu do souboru, jehož jméno je operandem operátoru (první slovo uvedené za operátorem). Pokud soubor neexistuje, bude vytvořen, pokud existuje, bude výstup připsán na jeho konec.

>| *příkaz* **>|** *soubor*

Přesměrování standardního výstupu příkazu do souboru. Soubor bude přepsán i v případě, že je přepínačem „**-C**“ zapnut režim noclobber.

>&, **<&** *příkaz* **>&** *n*

Duplikace deskriptoru – přesměrování standardního vstupu, resp. výstupu příkazu do proudu s daným číslem deskriptoru.

>&-, **<&-** *příkaz* **>&-**

Uzavření výstupního/vstupního deskriptoru.

Další metaznaky:

$** (dolar) **$*jméno*, resp. **${***výraz***}**

Náhrada proměnné nebo parametru.

` (zpětný apostrof) `*příkaz*`

Náhrada výstupu příkazu; *příkaz* (uzavřený mezi pár zpětných apostrofů nebo do závorek) se spustí a obsah jeho standardního výstupního proudu se vloží zpět do příkazové řádky, kde nahradí původní zápis příkazu. V novějších shellech a v normě existuje alternativní zápis „**$(***příkaz***)**“.

(hash-kříž)

Komentář (do konce řádky).

Substituce parametrů:

Parametrem může být buď proměnná (jako označení se použije její jméno), poziční parametr (označuje se pořadovým číslem), nebo speciální parametr (viz dále).

${*param***}**

Substituce hodnoty parametru. Pokud znak následující v textu bezprostředně za místem substituce není alfanumerický, lze v zápisu vynechat složené závorky.

${#*param***}**

Substituce délky hodnoty parametru.

${*param***-***slovo***}**

Podmíněná substituce hodnoty. Pokud hodnota není nastavena, použije se *slovo*.

${*param***=***slovo***}**

Podmíněná substituce hodnoty s přiřazením. Pokud hodnota není nastavena, použije se *slovo* a zároveň se nastaví do parametru jako jeho hodnota.

${*param***?***slovo***}**

Podmíněná substituce s testem. Pokud hodnota není nastavena, skript vypíše *slovo* na standardní chybový výstup a skončí.

${*param***+***slovo***}**

Substituce alternativní hodnoty. Pokud hodnota parametru je nastavena, použije se *slovo*, jinak se použije prázdný řetězec.

Poslední čtyři substituce v uvedené formě chápou hodnotu parametru za „nastavenou“, i když obsahuje prázdný řetězec. Lze je ovšem použít také ve formě s dvojtečkou za označením parametru (např. „**${***param***:-***slovo***}**“), a pak se prázdná hodnota chápe, jako by hodnota nebyla nastavena.

${*param***%***vzor***}**

Substituce hodnoty parametru zkrácené zprava o nejkratší možný řetězec vyhovující porovnávacímu vzoru (viz výše metaznaky porovnávacích vzorů).

${*param***%%***vzor***}**

Substituce hodnoty zkrácené zprava o nejdelší možný řetězec vyhovující vzoru.

${*param***#***vzor***}**

Substituce hodnoty zkrácené zleva o nejkratší možný řetězec vyhovující vzoru.

${*param***##***vzor***}**

Substituce hodnoty zkrácené zleva o nejdelší možný řetězec vyhovující vzoru.

Speciální parametry:

0 (nula) **$0**

Název shellu, resp. shellového skriptu.

n (nenulová číslice) **$1**, **$2** až **$9**

Poziční parametr číslo *n*.

(hash-kříž) **$#**

Počet aktuálně nastavených pozičních parametrů.

***** (hvězdička) **$***

Zřetězení všech pozičních parametrů.
Po nahrazení se ztrácí informace o hranicích původních parametrů (pokud mezi parametry byly prázdné řetězce nebo řetězce obsahující mezery).

@ (zavináč) **$@**

Zřetězení všech pozičních parametrů.
V případě uvedení mezi uvozovkami („**"$@"**“) vznikne přesně stejná struktura parametrů, jakou měly poziční parametry (zachovávají se i prázdné parametry nebo parametry s mezerami).

$** (dolar) **$$

Číslo procesu aktuálního shellu.

? (otazník) **$?**

Návratová hodnota posledního provedeného příkazu.

- (minus) **$-**

Aktuální nastavení přepínačů shellu (seznam písmen platných přepínačů).

Řídicí struktury:

! *příkaz*

Negace výsledku příkazu. Shell provede příkaz a převrátí návratovou hodnotu.

{ *příkaz***;**... **}**

Složený příkaz. Za otevírací složenou závorkou a místo středníků lze odřádkovat.

case *slovo* **in**
vzor1 [| *vzor2*]... **)**
[*příkaz* [**;***příkaz*]...]**;;**
...
esac

Příkaz větvení. Testované *slovo* se postupně porovná s jednotlivými vzory (ve vzorech se dají používat shellové metaznaky pro porovnávání řetězců) a první větev, jejíž vzor uspěje, se vykoná. Odřádkování zobrazená ve schematickém popisu nejsou povinná, ale jsou doporučená kvůli čitelnosti.

for *proměnná* [**in** [*slovo*...]] **; do**
příkazy...
done

Příkaz cyklu. V každém kroku se jedno ze slov dosadí jako hodnota řídicí proměnné. V případě vynechání části s klíčovým slovem **in** se dosazují poziční parametry.

if *příkaz1***; then**
příkazy...
[**elif** *příkaz2***; then**
příkazy...]
[**else**
příkazy...]
fi

Podmíněný příkaz. Spustí se příkaz *příkaz1* a otestuje se jeho návratová hodnota. V případě úspěchu (tj. hodnota nula) se pokračuje příkazy za klíčovým slovem **then** (větev **then**), jinak se pokračování liší podle použitého formátu příkazu. Pokud následuje klíčové slovo **else**, provedou se příkazy ve větvi **else**. Pokud následuje klíčové slovo **elif**, spustí se příkaz *příkaz2* a podle jeho výsledku se pokračuje analogicky jako na začátku při testování výsledku příkazu *příkaz1*.

until *příkaz*; **do**
    příkazy...
done

Příkaz cyklu. V každém kroku se spustí řídicí *příkaz* a v případě nenulové návratové hodnoty (neúspěchu) se vykoná tělo cyklu, jinak cyklus skončí.

while *příkaz*; **do**
    příkazy...
done

Příkaz cyklu. V každém kroku se spustí řídicí *příkaz* a v případě nulové návratové hodnoty (úspěchu) se vykoná tělo cyklu, jinak cyklus skončí.

Interní příkazy:

:

Prázdný (null) příkaz.

break [*n*]

Příkaz ukončí provádění cyklu. Implicitně ukončí nejvnitřnější cyklus obklopujícího volání příkazu, v případě zadání parametru vyskočí z *n* zanořených cyklů.

cd [*adresář*]

Příkaz změní aktuální adresář běžící instance shellu na zadaný *adresář* (implicitně domovský adresář uživatele).

command [*příkaz* [*parametr...*]]

Příkaz vykoná zadaný příkaz, ignoruje případné stejnojmenné funkce.

continue [*n*]

Příkaz ukončí provádění těla cyklu a skočí na začátek, tj. pokračuje novým krokem (u cyklů **while** a **until** to znamená nové provedení řídicího příkazu).
Implicitně skočí na začátek nejvnitřnějšího cyklu obklopujícího volání příkazu, v případě zadání parametru skočí na začátek *n*-tého nejvnitřnějšího cyklu.

eval [*parametr...*]

Příkaz vytvoří ze svých parametrů novou příkazovou řádku (oddělí je mezerami) a tento příkaz nechá znova přečíst a zpracovat shell.

exec [*příkaz* [*parametr...*]] [*přesměrování...*]

Příkaz vykoná zadaná přesměrování a poté spustí *příkaz*, kterým nahradí aktuální instanci shellu. Pokud příkaz nebyl zadán, uplatní se přesměrování v běžícím shellu.

exit [*n*]

Příkaz ukončí běh shellu. V případě zadání parametru nastaví návratovou hodnotu.

getopts *popis jméno* [*parametr*]...

Příkaz postupně čte přepínače z pozičních parametrů nebo ze zadaných parametrů. Seznam známých přepínačů udává parametr *popis*, písmena přepínačů s hodnotu jsou v seznamu následována dvojtečkou. Při každém vyvolání příkaz uloží jeden nalezený přepínač do proměnné *jméno* a jeho případnou hodnotu do proměnné **OPTARG**. V případě nalezení neplatného přepínače se chování liší podle zadání řetězce *popis*. Pokud *popis* nezačíná dvojtečkou, bude proměnná *jméno* obsahovat otazník a na chybový výstup se vypíše chybová zpráva. Pokud řetězec *popis* dvojtečkou začíná, v proměnné **OPTARG** bude uložen znak nalezeného přepínače a proměnná *jméno* bude obsahovat otazník (pro neznámý přepínač) nebo dvojtečku (při chybějící hodnotě u přepínače s povinnou hodnotou).

Po vyčerpání seznamu přepínačů (konec parametrů, nalezení parametru „**--**“ nebo nalezení parametru, který nezačíná pomlčkou) příkaz vrací nenulovou návratovou hodnotu (dá se tedy testovat např. v rámci příkazu **while**) a v proměnné **OPTIND** je číslo parametru, který se má dále zpracovat.

Příkaz je možno vyvolávat opakovaně, je ale nutné inicializovat proměnnou **OPTIND**.

read [**-r**] *proměnná...*

Příkaz přečte jednu řádku vstupu, rozdělí ji podle oddělovačů polí uvedených v proměnné **IFS** na jednotlivá pole a ta postupně přiřadí proměnným (poslední proměnná obsahuje zbytek řádky). Zpětná lomítka v řádce fungují jako quoting pro oddělovače polí, pro zpětné lomítko samo a pro znak konce řádky (pokud zpětným lomítkem řádka končí, přečte se i následující řádka a obě se spojí). Toto chování lze potlačit přepínačem „**-r**“ (*raw*).

return [*rc*]

Příkaz ukončí běh funkce. V případě zadání parametru nastaví návratovou hodnotu.

set

Příkaz vypíše aktuální nastavení všech proměnných shellu.

set -o

Příkaz vypíše aktuální nastavení všech přepínačů shellu.

set [*přepínač...*] [[**--**] [*parametr...*]]

Příkaz nastavuje přepínače (viz str. 337) nebo poziční parametry shellu příp. obojí. Pokud je přepínač uveden znakem „**+**“ místo znaku „**-**“, znamená to vypnutí patřičného režimu.

shift [*n*]

Příkaz posune poziční parametry shellu o *n* pozic (implicitně jednu) doleva.

trap *akce podmínka...*
trap - *podmínka...*
trap "" *podmínka...*
trap

Příkaz nastaví ošetření podmínky (číslo nebo jméno signálu pro příchod signálu, resp. „**0**“ nebo „**EXIT**“ pro ukončení shellu). V případě zadání akce „**-**“ se ošetření podmínky vrací do defaultního stavu. V případě zadání prázdné akce se příchod signálu ignoruje (a v případných subshellech pak není možné tento signál jakkoli ošetřovat). Bez parametrů vypíše příkaz aktuální stav ošetření podmínek.

Nastavení platí v aktuálním shellu, do okamžiku zadání nového příkazu **trap**.

umask [*maska*]

Bez parametru příkaz vypíše aktuální nastavení uživatelské masky.

Při volání s parametrem nastaví aktuální uživatelskou masku na danou hodnotu.

Maska ovlivňuje implicitní přístupová práva pro nově vzniklé soubory – po nastavení práv dle aplikace se zamaskují (odeberou) práva daná jednotlivými bity masky.

unset [**-fv**] *jméno...*

Příkaz zruší nastavení hodnoty proměnné (**-v**), resp. platnost definice funkce (**-f**).

A.7 Regulární výrazy

Norma rozlišuje dva dialekty regulárních výrazů, *základní (BRE)* a *rozšířené (ERE)*. BRE se používají v editorech **sed**, **ed** a programech od nich odvozených (**grep**). ERE se používají v jazyce **awk**, ne vždy ovšem v úplné implementaci (např. často chybí konstrukt „**{}**"). Volitelně také třeba v programu **grep**, pokud to při volání zadáme pomocí odpovídajícího přepínače („**-E**").

Konstrukt	Význam	Příklad	Shoda s	Neshoda s
znak	výskyt znaku	**abc**	**abc**, **xabcy**	**abbc**, **cba**
.	jakýkoliv znak	**ab.c**	**ab1c**, **abXc**	**abc**, **ab10c**
[*seznam***]**	znak ze seznamu	**a[0-9]**	**a0**, **a1**, **a99**	**a-**, **a**
[^*seznam***]**	znak z doplňku	**a[^0-9]**	**aa**, **a-**, **a^**	**a0**, **a**
^ (na začátku)	začátek řetězce	**^abc**	**abc**, **abcde**	**xabc**, **Abc**
$ (na konci)	konec řetězce	**c$**	**xyz.c**	**xyz.core**
*prvek******	libovolný počet výskytu prvku	**ab*c**	**ac**, **abbbc**	**bc**, **abab**
*prvek***+**	alespoň jeden výskyt (*ERE*)	**ab+c**	**abc**, **abbbc**	**ac**, **abab**
*prvek***?**	nejvýše jeden výskyt (*ERE*)	**ab?c**	**ac**, **abc**	**abbc**
*prvek***\{***m***,***n***\}**	výskyt prvku *m*- až *n*-krát *	**ab\{2,3\}c**	**abbc**, **abbbc**	**abc**, **abbbbc**
*prvek***\{***m***,\}**		**ab\{1,\}c**	**abc**, **abbbbc**	**ac**
*prvek***\{***m***\}**		**ab\{2\}c**	**abbc**	**abc**, **abbbc**
\(*výraz***\)**	podvýraz *	**^\(ab\)*$**	**ab**, **abab**	**aab**, **abb**
*****i	zpětná reference (*BRE*)	**a\(.\)\1a**	**abba**, **adda**	**aba**, **alda**
*v1***|***v2*	alternativa (*ERE*)	**abc|xyz**	**abc**, **xyz**	**abz**, **xy**

* Konstrukt se vyskytuje v BRE i v ERE, v ERE se zapisuje bez zpětných lomítek.

Do seznamu v hranatých závorkách je možno zapsat libovolné znaky nebo třídy znaků (viz str. 347). Pravidla pro zápis metaznaků do seznamu:

znak „**]**" musí být uveden jako první (po případném metaznaku „**^**")

znak „**^**" nesmí být uveden jako první

znak „**-**" musí být uveden na takovém místě, aby nemohl být považován za metaznak pro interval, tj. buď jako první znak, jako poslední znak, nebo jako první znak bezprostředně za nějakým intervalem

A.8 Třídy znaků

x-*y*

Interval znaků mezi hodnotami *x* a *y* (včetně). Přesný význam ovlivňuje nastavení národního prostředí, za implicitních podmínek lze jako rozhodující brát pozici v tabulce znaků ASCII.

[:*třída***:]**

Pojmenovaná třída znaků, nejdůležitější třídy:

digit – číslice;

xdigit – hexadecimální číslice;

lower – malá písmena;

upper – velká písmena;

alpha – písmena;

alnum – alfanumerické znaky (písmena a číslice);

blank – mezera nebo tabulátor;

space – bílé znaky (mezera, tabulátor, znak konce řádky, backspace...);

print – tisknutelné znaky včetně mezery;

graph – tisknutelné znaky bez mezery;

punct – interpunkce, tj. tisknutelné znaky bez **space** a **alnum**;

cntrl – znaky mimo **print**.

[.*znak***.]**
[=*znak***=]**

Třídy ekvivalence národních znaků. Speciální konstrukt umožňující popisovat skupiny národních znaků pomocí jejich příbuznosti.

Závisí na nastavení systému. Pozor především na neúmyslné náhodné použití takového pořadí znaků v seznamech – vyvolá neočekávané chování.

A.9 Formátovací direktivy pro `printf`

Popis platí pro utilitu **`printf`** a příkaz **`printf`** jazyka **`awk`**.

direktiva	znamená
`%c`	výpis znaku
`%s`	výpis řetězce
`%b`	výpis řetězce, ve kterém se správně interpretují escape-sekvence *(pouze u utility **`printf`**, neplatí pro funkci jazyka **`awk`**)*
`%o`	oktalový výpis čísla
`%d`	dekadický výpis (celého) čísla (se znaménkem)
`%u`	dekadický výpis (přirozeného) čísla (bez znaménka)
`%x`	hexadecimální výpis čísla (pomocí znaků 0123456789abcdef)
`%X`	hexadecimální výpis čísla (pomocí znaků 0123456789ABCDEF)
`%e`	výpis reálného čísla v semilogaritmickém tvaru ([**`-`**]*d.dd*[**`e`**][**`-`**]*dd*)
`%f`	výpis reálného čísla v plovoucí čárce ([**`-`**]*ddd.dddd*)
`%g`	výpis reálného čísla v **`e`** nebo **`f`** formátu podle velikosti čísla
`%%`	výpis znaku procento

modifikátor	znamená
`%`*MIN***`s`**, **`%`***MIN***`d`**, ...	výpis v délce *MIN* znaků (doplnění mezer zleva)
`%-`*MIN***`s`**, **`%-`***MIN***`d`**, ...	výpis v délce *MIN* znaků (doplnění mezer zprava)
`%[-][`*MIN***`].`***MAX***`s`**	výpis nejvýše *MAX* znaků z řetězce
`%0`*MIN***`d`**, ...	výpis čísla v délce *MIN* znaků (doplnění nulami)
`%+[`*MIN***`]d`**, ...	vynucení výpisu znaménka
`%[+][`*MIN***`].`***PREC***`g`**	výpis reálných čísel s požadovanou přesností

Příloha B: Formáty používaných zdrojů dat

B.1 Popis vybraných souborů

Soubor `/etc/passwd` :

```
root:*:0:0:Super User:/root:/bin/bash
forst:*:1004:1004:Libor Forst:/home/forst:/bin/bash
```

Každému uživateli přísluší jedna řádka se sloupci:

1. přihlašovací jméno (login)
2. zakódované heslo, příp. hvězdička
3. číslo uživatele (UID)
4. číslo (primární) skupiny uživatele (GID)
5. plné jméno uživatele
6. domovský adresář uživatele
7. login-shell.

Soubor může obsahovat komentáře – řádky začínající znakem „#“.

Soubor `/etc/group` :

```
wheel:*:0:root,forst,kernun
forst:*:1004:
```

Každé skupině přísluší jedna řádka se sloupci:

1. jméno skupiny
2. bezvýznamové pole
3. číslo skupiny (GID)
4. seznam uživatelů (kromě těch, pro něž je tato skupina primární).

Soubor může obsahovat komentáře – řádky začínající znakem „#“.

Soubor `rfc-index.txt` :

Soubor se skládá z bloků popisujících jednotlivé dokumenty a jejich atributy. Na začátku souboru je cca jedna stránka úvodních informací s jiným formátem.

Ukázka bloku:

```
0172 The File Transfer Protocol. A. Bhushan, B. Braden,
     W. Crowther, E. Harslem, J. Heafner, A. McKenzie,
     J. Melvin, B. Sundberg, D. Watson, J. White. June 1971.
     (Format: TXT=21328 bytes) (Obsoleted by RFC0265)
     (Updates RFC0114) (Updated by RFC0238)
     (Status: UNKNOWN)

0173 ...
```

Soubor s uloženou elektronickou poštou:

Příchozí elektronické dopisy se v UNIXu ukládají každému uživateli chronologicky do jednoho souboru, jehož jméno bývá uloženo v systémové proměnné **`MAIL`**. Jednotlivé dopisy jsou od sebe odděleny zvláštní řádkou ve tvaru „`From ...`“, za níž následují hlavičky a tělo dopisu:

```
From adresa_odesilatele  datum_a_čas_doručení
...začátek hlavičky dopisu
Subject:  předmět dopisu
...další řádky hlavičky
                                          prázdná řádka oddělující hlavičku
...text dopisu
From ...
```

B.2 Popis vybraných souborů

Příkaz `date` :

```
sinek:~> date
Tue Oct  1 03:30:00 CET 1968
```

Implicitní výstup příkazu obsahuje: název (zkratku) dne v týdnu, název (zkratku) měsíce, číslo dne v měsíci, čas, označení časové zóny a (čtyřciferný) rok.

Příkaz `diff` :

Předpokládejme následující obsah souborů `old` a `new`:

old	new
o-1 o-2 o-3 o-4 o-5 o-6	o-3 n-x n-y o-6 n-z

Standardní výstup příkazu:

```
sinek:~> diff old new
1,2d0     ... ř. 1 až 2 ze souboru old v souboru new chybějí, patřily by za ř. 0
< o-1           ... řádka číslo 1 ze souboru old
< o-2           ... řádka číslo 2 ze souboru old
4,5c2,3   ... ř. 4 až 5 z old se liší od odpovídajících ř. 2 až 3 z new
< o-4           ... řádka číslo 4 ze souboru old
< o-5           ... řádka číslo 5 ze souboru old
---       ... oddělovací řádka výpisu
> n-x           ... řádka číslo 2 ze souboru new
> n-y           ... řádka číslo 3 ze souboru new
6a5       ... ř. 5 ze souboru new v souboru old chybí, patřila by za ř. 6
> n-z           ... řádka číslo 5 ze souboru new
```

Výstup ve formě editačních příkazů:

```
sinek:~> diff -e old new
6a
n-z
.
4,5c
n-x
n-y
.
1,2d
```

Příkaz `id` :

```
sinek:~> id
uid=1004(forst) gid=1004(forst) groups=1004(forst),0(wheel),
1(daemon),999(kernun),1028(j28)
```

Obvykle výstup příkazu obsahuje tři pole: číslo a jméno uživatele, číslo a jméno primární skupiny uživatele a čísla a jména všech skupin, jichž je uživatel členem.

Příkaz `ls -l` :

```
drwxr-xr-x  2 forst  forst 512 Nov 27 02:05 bin
-rwxr-xr-x  2 forst  forst   8 Nov 27 02:05 ll
-rwxr-xr-x  2 forst  forst   8 Nov 27 02:05 ltr.h
lrwxr-xr-x  1 forst  forst   2 Nov 27 02:06 ltr.s -> ll
```

Každému souboru přísluší jedna řádka se sloupci:

1. typ souboru a přístupová práva
2. počet existujících hardlinků pro daný soubor (resp. i-node)
3. vlastník souboru
4. skupinový vlastník souboru
5. velikost souboru
6. datum poslední modifikace souboru (měsíc)
7. datum poslední modifikace souboru (číslo dne v měsíci)
8. čas poslední modifikace (ve tvaru *hod*:*min*) nebo rok (u starších souborů)
9. jméno souboru a v případě symbolického linku ještě šipka a cíl odkazu.

Příkaz `ps` :

```
sinek:~> ps
  PID  TT  STAT      TIME COMMAND
 9732  p9  Ss     0:00.00 bash
27263  p9  R+     0:00.00 ps -l
```

Každému procesu přísluší jedna řádka, ovšem skladbu sloupců norma neupravuje. Přenositelné programy proto musejí používat přepínač „`-o`“ pro pevné pořadí sloupců.

Příkaz `who` :

```
sinek:~> who
forst            ttyv1    Nov 27 02:05
forst            ttyv1    Nov 27 02:05
forst            ttyp1    Nov 27 02:05 (sinek)
```

Každému aktuálně přihlášenému uživateli přísluší jedna řádka s minimálním obsahem: přihlašovací jméno, označení terminálu a datum a čas přihlášení.

Literatura

[1] The Single UNIX® Specification, Version 3 (POSIX.1-2008),
The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008
`http://www.opengroup.org/onlinepubs/9699919799`

[2] A. Frisch: Essential System Administration; O'Reilly & Associates Inc., 2002;
ISBN 978-0-596-00343-2

[3] D. Dougherty, A. Robbins: sed & awk; O'Reilly & Associates Inc., 1997;
ISBN 978-1-56592-225-9

[4] A. Robbins, N. Beebe: Classic Shell Scripting; O'Reilly & Associates Inc., 2005;
ISBN 978-0-596-00595-5

[5] C. Albing, J. Vossen, C. Newham: bash Cookbook; O'Reilly & Associates Inc.,
2007; ISBN 978-0-596-52678-8

[6] E. Quigley: UNIX Shells by Example; Pearson Education Inc. (Prentice-Hall),
2005; ISBN 0-13-147572-X

[7] S. Kochan, P. Wood: Unix Shell Programming; SAMS, 2003; ISBN 0-672-32390-3

[8] E. Raymond: The Art of UNIX Programming; Addison Wesley; 2004;
ISBN 0131429019

[9] HTML 4.01 Specification; `http://www.w3.org/TR/html4/`

[10] R. Fielding et al.: Hypertext Transfer Protocol -- HTTP/1.1; RFC2616

[11] T. Berners-Lee, R. Fielding, L. Masinter: Uniform Resource Identifier (URI):
Generic Syntax; RFC3986

[12] P. Resnick, Ed.: Internet Message Format; RFC5322

[13] Domovská stránka knihy: `http://www.yq.cz/SvP/`
(obsahuje řešení úloh na procvičení a vzory souborů)

Rejstřík

Pokud se u nějakého pojmu vyskytuje více druhů písma v odkazech na stránky, pak **tučné** odkazy znamenají stránky se zásadními informacemi, zatímco odkazy *kurzívou* představují pouze zmínky v textu nebo významnější použití v kódech programů.

& (ampersand)

& (expr) 167
& (sed) **93**, *115*, *132*
& (shell) 338
&& (awk) **219**, 222, *223*
&& (shell) **155**, *157*, *164*, *171*
& (HTML) 127
> (HTML) 127
< (HTML) 127

' (apostrof)

'...' (shell) **30**, 32, *40*, *45*, *48*, *60*, *68*, *88*, *135*, **136**

, (čárka)

, (awk) 239
, (ed) 119
, (sed) 89, 100

$ (dolar)

$- **270**, 341
$ (ed) **119**, *121*, *126*
$ (regexpy) **40**, *93*, **99**, 112, 240
$ (sed) **89**, *108*, *110*
$ (shell) **133**, **135**, 136, 148
$# **148**, *171*, *174*, *182*, *197*
$$ **191**, *196*, *199*, *276*
$(()) 167
$() 138
$? **155**, *160*, *197*
$* **151**, *197*, *244*
$@ **150**, *172*, *182*, *185*, *244*
${*#jméno*} **166**, *279*, *281*
${*jméno##vzor*} 293, 341
${*jméno#vzor*} 293, 341
${*jméno%%vzor*} 293, 341
${*jméno%vzor*} 293, 341
${*jméno*:=*slovo*} 152
${*jméno*:*pozice*:*délka*} *210*
${*jméno*:-*slovo*} 152, *206*
${*jméno*} **148**, *152*
$0 (awk) **221**, 236, *238*, *253*, *256*
$0 (shell) **149**, 151, *171*, 194, *197*
$*číslo* (awk) **215**, 222, 236
$*číslo* (shell) **148**, 151, *152*, 172, 181
$HOME viz HOME
$IFS viz IFS
$LOGNAME viz LOGNAME
$MAIL viz MAIL
$OPTARG viz getopts
$OPTIND viz getopts
$PATH viz PATH
$PPID viz PPID
$PS1 viz PS1
$PS2 viz PS2
$TERM viz TERM

: (dvojtečka)

: (awk) 273
: (expr) 167
: (sed) 102
: (shell) viz prázdný příkaz

(hash kříž)

(/etc/group) *53*
(/etc/passwd) 25, *84*
(awk) 274
(sed) 99
(shell) 169, *265*
#! 198, 274
#define 255
#include 255

* (hvězdička)

* (expr) 167
* (porovnávací vzory)
  (case) **159**
  (expanzní znaky) **31**, *41*, *131*, *167*
  (find) 71
* (regexpy) **40**, **91**, *97*

/ (lomítko)

/
  kořen stromu adresářů 23, *69*
  oddělovač adresářů 21
  ve jméně souboru 39
/ (awk)
  dělení 217
  regulární výrazy **216**, *218*, 224
/ (expr) 167
/ (sed) 88, 96
/bin **59**, *84*
/dev 180
/dev/null **37**, **51**, *68*, *142*, *156*, *167*, *180*
/dev/stderr **180**
/dev/stdin **180**
/dev/stdout **180**
/dev/tty **180**, **186**, *232*
/etc **34**
/etc/group **45**, 50, *80*, *97*, *201*, **349**
/etc/passwd **24**, *43*, 50, *75*, *79*, *80*, *87*, *127*, *164*, *201*, **349**
/tmp **28**
/usr/include *131*, 255

< (menšítko)

< (awk) 219
< (diff) 123
< (expr) 167
< (shell) **24**, *76*, *122*, *145*, *164*, *186*
<& **187**, *203*
<< **152**
<<- **153**
<= (awk) 219
<= (expr) 167
<a> (HTML) 239, 261
 (HTML) 283
 (HTML) 240, 283
<p> (HTML) 107
<table> (HTML) 127
<td> (HTML) 127
<tr> (HTML) 127

- (minus)

- (ed) 118
- (parametr) **49**, *51*, *79*, *227*
- (printf) 237
- jako typ souboru 38
- na začátku jmen souborů 32
-- (awk) **221**, **231**
-- (parametr) **32**, *173*

? (otazník)

? (awk) 273
? (ed) **119**
? (porovnávací vzory)
  (case) **159**, *175*
  (expanzní znaky) **131**
? (regexpy) **218**

+ (plus)

+ (ed) 118, *125*
+ (regexpy) **218**, *236*
++ (awk) **221**, **231**
+= (awk) **217**

_ (podtržítko)

_ (podtržítko) 137, 153, 217

% (procento)

% (awk) 217
% (expr) 167
% (URI) 277
%% (date) *212*, **307**
%% (printf) **348**
%d (printf) *232*, **233**, **237**
%n (date) 27, **307**
%s (printf) **196**, *232*, 237, *273*
%*spec* (date) 27, **307**
%*spec* (printf) **348**

= (rovnítko)

= (awk) **217**, **219**
= (expr) 167
= (shell) 134
= (test) **158**, *160*, *164*, *171*
== (awk) **219**

; (středník)

; (awk, sed, shell)
  viz oddělovač příkazů
; (find) 69
;; (shell) 160

^ (stříška)

^ (awk) 217
^ (regexpy) **38**, *40*, *77*, *93*, **99**, **112**
^ mezi [] viz [^...]

| (svislítko)

| (case) **160**, *185*, *198*
| (expr) 167
| (regexpy) **218**, **240**
| (shell) **34**, 35
|| (awk) **219**, 222
|| (shell) **155**, *157*

. (tečka)

. (adresář) viz adresář:.
. (ed) 125
. (regexpy) 40
. (shell) 199
. ve jméně souboru 36, 41
.. (adresář) viz adresář:..
./program 59, *60*, *149*, 172, 194
.bak (přípona) *132*
.bash_history 53
.c (přípona) *162*
.h (přípona) *131*, *132*, *255*
.profile 199

" (uvozovky)

"..." (awk) viz awk:řetězce:zápis
"..." (shell) **136**, *139*, *150*, *158*, *161*, *167*, *170*, *179*, *183*

> (většítko)

> (awk)
  operátor porovnání 219
  operátor přesměrování **233**, *249*
> (diff) 123
> (expr) 167
> (prompt) 21, 45
> (shell) **26**, 27, 35, **37**, *60*, 68, 87, 116, *167*, *281*
>& (shell) 142, 156
>| (shell) *337*, **339**

>= (awk) 219
>= (expr) 167
>> (awk) **233**
>> (shell) **27**, 68, 282

~ (vlnka)

~ (awk) **223**
~ (domovský adresář) **21**, **133**

! (vykřičník)

! (awk) **219**, *229*
! (find) 71
! (sed) **96**, *105*, *269*
! (shell) **157**, *281*
! (test) **158**, *159*, *182*
! mezi [] viz [!...]
!~ (awk) **224**
!= (awk) **219**, *249*, *272*
!= (expr) 167
!= (test) **158**, *198*

[] (závorky hranaté)

[...] viz test
[!...] (porovnávací vzory) **131**
[...] (awk) 228
[...] (porovnávací vzory)
  (case) **159**, *174*
  (expanzní znaky) **131**
  doplněk seznamu (!) **131**, *159*, *169*
  metaznaky v seznamu 131, *192*
  rozsah znaků (-) **131**, *159*
[...] (regexpy) **90**, *97*, **346**
  doplněk seznamu (^) **91**, *100*, *103*
  metaznaky v seznamu 91, *93*, 110, 115, 170
  rozsah znaků (-) **91**
[^...] (regexpy) **91**
[.*znak*.] 347
[=*znak*=] 347
[:alnum:] 347
[:alpha:] 347
[:blank:] 347
[:cntrl:] 347
[:digit:] 91, 347
[:graph:] 347
[:lower:] 44, 347
[:print:] 347
[:punct:] 347
[:space:] 347
[:*třída*:] 44, 347
[:upper:] 44, 347
[:xdigit:] 347
[^...] (regexpy) **91**
[=*znak*=] 347

() (závorky kulaté)

(...) (case) **160**
(...) (expr) 167
(...) (find) **71**, *72*
(...) (regexpy) **219**, 240
(...) (shell) **146**
(...) (test) 158

{} (závorky složené)

{...} (awk) **216**, **221**
{...} (regexpy)
  viz regulární výraz:počet výskytů
{...} (sed) **98**
{...} (shell) **143**, *204*
{} (find) 69

\ (zpětné lomítko)

\ (awk)
  pokračovací řádka 222
  v řetězci 233
\ (regexpy) **40**, 92, *170*, 219
\ (sed) 96, 101
\ (shell) **47**, **136**, 139, 170

\ mezi [...] 170
\ na konci řádky (awk) 222
\ na konci řádky (sed) 101
\ na konci řádky (shell) 47
\ ve vstupu (read) 165
\(...\) (regexpy) **92**, **93**, 94, *97*, *110*
\{...\} (regexpy) *40*, **94**, *95*, *115*, *279*
\c (echo) 68
číslo (regexpy) **92**, **94**, **110**
\n (awk) 233, *249*
\n (paste) 76
\n (sed) **107**, *109*, **110**, 112, *113*, *269*
\n (tr) **45**, *46*
\r (CR) viz escape-sekvence
\t (tabulátor) viz escape-sekvence

` (zpětný apostrof)

`...` (shell) 136, **138**, *140*, *146*

A

-a (find) 71
-a (shell) 196
-a (test) **158**, *245*
A (HTML) 239, 261
a zároveň (awk, shell) viz &&
a zároveň (expr) viz &
a zároveň (find, test) viz -a
absolutní cesta viz cesta:absolutní
access mode viz přístupová práva
adresa IP 95
adresář **21**, 22
  . 23, 36, 59, *102*, *134*, *275*
  .. 23, 36, *102*, *188*, *266*, *275*
  aktuální 21
    odkaz 23, *59*, *102*, *275*
    v PATH 134
    výpis cesty 21
    výpis obsahu 22
    změna 21, 146
  bin 59, 65
  domovský 21, 24, 36, 133, 349
  implementace 57
  jako poštovní schránka *57*
  kořenový 21
  nadřazený 23, *102*, *275*
  počítání souborů *34*, *36*, *38*, *39*
  podadresář 21, *23*, *38*, *67*, *70*, *188*
  podstrom *39*
  pracovní viz adresář:aktuální
  přístupová práva 57, 72
  rodičovský 23, *102*, *275*
  stromová struktura 21
  testování existence 158
  výpis atributů (ls -d) 23
  výpis cesty viz pwd
  výpis obsahu viz ls
  vytvoření viz mkdir
  změna viz cd
adresářová struktura 21
alfanumerické znaky 137, 217, *227*
ALGOL 163
allexport režim 337
alnum (třída znaků) 347
alpha (třída znaků) 347
alternace viz | (regexpy)
ampersand viz &
API (Application Programming Interface) 22, *211*
apostrof viz '...'
ARGC (awk) **255**
argument přepínače **28**, 41
argumenty funkce (shell) viz poziční parametry
argumenty příkazu viz parametry příkazu
argumenty shellskriptu viz poziční parametry
ARGV[] (awk) **255**, *272*
archivace (tar) 66

aritmetická expanze viz $(())
aritmetické porovnání (awk) 218
aritmetické porovnání (expr) 167
aritmetické porovnání (test) 158
aritmetické třídění viz sort
aritmetika (awk) 217
aritmetika (expr) 167
aritmetika (shell) 148, 166, 167, *193*
ASCII **74**, *91*, *308*
asociativní pole
    viz awk:indexované proměnné
atribut značky (HTML) 261
awk *52*, **213**, **323**
  akce **216**, **219**, 224
  analýza *236*, *271*, *275*, *284*, *298*
  aritmetika 217
  asociativní pole
      viz awk:indexované proměnné
  cyklus **325**
  čísla
    náhodná viz rand()
    převod na celá viz int()
    řádek 227
    výpis viz printf (awk)
  dekrementace viz --
  editace vstupu 236
  efektivita 213
  formátování textu viz sprintf()
  formátování výstupu viz printf (awk)
  funkce
    knihovny funkcí 274
    předávání parametrů 230
    vestavěné **217**, **327**
    vlastní **228**, *272*, **327**
  implicitní akce 219
  implicitní vzor 217
  indexované proměnné **228**
    inicializace prvků 229, 231
    použití jako mapa *232*, *235*, *241*, *245*, *255*, *276*
    procházení prvků viz for in
    řetězec jako index 228, *231*, *245*, *255*, *276*
    uložení prvků 231
    vícerozměrné 230
inicializační akce viz BEGIN
inkrementace viz ++
interval ve vzoru 239
komentáře 274
konec programu viz exit (awk)
konfigurace programu 246, 278
ladění 244
mezera 215, 223
návratová hodnota viz exit (awk)
oddělovač polí
  implicitní **215**
  platnost **235**
  proměnná FS **233**, *246*
  sémantika **236**
  zadání přepínačem (-F) **218**, *220*
oddělovač příkazů 222
oddělovač záznamů
  nastavení 239
  prázdný řetězec 241
parametrizace programu 225
parametry funkce
  předávání hodnotou 230
  předávání referencí 230
parametry příkazové řádky
  inicializace proměnných 226, *233*
  přístup k hodnotám 255
  vstupní soubory 223, 255
počet otevřených rour 254
počet otevřených souborů 249
podmíněný příkaz viz if (awk)
podmíněný výraz 273
pokračovací řádka 222
pole (indexovaná proměnná) 228

pole (rozdělení řetězce na) viz split()
pole (vstupního záznamu)
  oddělovač viz awk:oddělovač polí
  odkaz **215**, 217, **222**
  počet viz NF
  poslední pole 222
  rozdělení záznamu **215**
  změna obsahu **236**
program v souboru
  vs. v parametru 274
  zadání (-f) 274
proměnné
  datový typ 217
  indexované 228
  inicializace **226**, **229**, *233*
  jména 217
  konverze 217
  nenastavená hodnota 229
  přiřazení **217**
  skalární **216**, 230, **324**
  ve funkcích 230
proměnné prostředí 248
proměnné vestavěné 325
přenositelnost 213
přepínače **323**
  -f (jméno skriptu) 274
  -F (oddělovač polí) 218
  -v (inicializace proměnné) 226
přesměrování výstupu
    viz awk:výstupní příkazy
přiřazovací operátory 217
regulární výrazy
  dialekt **218**, *240*
  oddělovač polí **236**, 256
  operátor shody **223**
  ve vzoru **216**
  vs. hledání podřetězce 226
roura **252**, *271*
rozdělení řádky 222
rozsah ve vzoru 239
řádka (vstupu) viz awk:záznam
řetězce
  délka viz length()
  hledání podřetězce viz index()
  náhrada podřetězce viz sub()
  náhrada všech výskytů viz gsub()
  rozdělení na slova viz split()
  testování regulárního výrazu viz ~
  výběr podřetězce viz substr()
  vypsání řetězců z programu 127
  zápis *127*, **222**, 233, **323**
  zřetězení **223**
skript
  spouštění jako příkaz 274
  uložený v souboru (-f) 274
slova viz awk:pole
složený příkaz **221**, 222
spolupráce s shellem **250**, 255
spouštění příkazů 251, 252
standardní vstup 223, 227
stav programu 221, *223*
syntaxe 222
testování shody s regexpem viz ~
ukončení cyklu viz break (awk)
ukončení funkce viz return (awk)
ukončení práce se záznamem
    viz next (awk)
ukončení programu viz exit (awk)
ukončení provádění těla cyklu
    viz continue (awk)
větve **216**, **221**, 224, *230*, *243*
vstupní soubory
  rozlišení viz FILENAME
  zadání 223
  změna seznamu 255
výraz 217
  aritmetický **217**
  logický **219**, *227*, *229*, *256*
  porovnávací **219**
  řetězcový **223**

awk *(pokračování)*
výstup do roury **254**
výstup do souboru **233**, **249**
výstupní příkazy **326**
vzor **216**, 224
BEGIN **216**, *240*
END **216**, *231*
implicitní **217**, *231*
logický výraz **217**, *218*
regulární výraz **216**, *220*
rozsah vzorů **239**
závěrečná akce viz END
záznam
číslo viz NR
obsah $0 a RS 240
oddělovač
viz awk:oddělovač záznamů
odkaz **221**, *238*, *240*, *253*, *256*
změna obsahu 236
zřetězení řetězců 223

B

backquote viz `...`
backslash viz \
bak (přípona) *132*
basename 149, *171*, *211*
bash 133
BEGIN (awk) **216**, 226, 229, *232*, *233*, *242*, *244*, *256*, *272*
Berkeley Software Distribution
viz BSD systém
bezpečnost 134, 179, 226, 227, 234
bílý znak 75, *80*, *141*, *255*
bin (adresář) 65
binární soubor 24, *59*, *207*
blank (třída znaků) 347
bloky řádek
jako záznamy (awk) 241
přesun (ed) 125
přidání zarážky 124
příkazy pro blok řádek (ed) 119
příkazy pro blok řádek (sed) 100
ve výstupu diff 123
větvení kódu (awk) 220
vkládání (ed) 117
vkládání (sed) 105
vyhledávání (sed) 111
výpis (sed) *127*
Bourne Again Shell (bash) 133
Bourne shell 133
brace viz {...}
bracket viz [...]
BRE viz regulární výraz:základní
break (awk) **254**
break (shell) **164**
BSD systém 253
built-in příkazy (shell)
viz interní příkazy (shell)

C

-C (shell) 337
c (přípona) *162*
cache *112*, **143**
carriage return viz CR
case **159**, *173*, *291*
dynamické návěští 161
implicitní větev 160
porovnávací vzory 159, *176*
pořadí větví 160, 173
prázdný řetězec 185
case senzitivita 316
(sort) 74
jména proměnných (awk) 217
jména proměnných (shell) 137, 141
jména souborů 40
parametry 27
přepínače 63
příkazy editoru 107

cat **26**, *29*, *144*
  kopírování vstupu na výstup 144
  standardní vstup v parametrech 51
  výpis čísel řádek (-n) 211, *257*
  záměna s cut 44
  záměna s echo 26
  zřetězení vstupů 51
cd 21, *23*, *37*, 146, *181*, *188*, *212*
cesta **21**
  absolutní **23**, 67, *168*
  extrakce adresáře viz dirname
  extrakce jména souboru
    viz basename
  normalizace *102*, *275*
  relativní **23**, 67, *102*, *168*, *275*
  zadání příkazu 59
cesta (URI) 262
close() (awk) **249**, *272*
cntrl (třída znaků) 347
comm **79**
command (shell) **181**
continue (awk) **254**
continue (shell) **163**, *173*, **174**, *193*
cp
  kopírování linků 65
  kopírování souboru **59**, **66**, *68*, *132*
  kopírování více souborů **66**, *131*
  rekurzivní kopírování (-r, -R) 66
  zachování data a času (-p) 132
CR (carriage return) 210, *282*
  zápis viz escape-sekvence
cross-device link 62
Ctrl+\ 193
Ctrl+C 191
Ctrl+D 25
Ctrl+V 141
cut **43**, *50*, *76*, *77*
  dva různé oddělovače 46
  jako SQL SELECT 43
  oddělovač sloupců (-d) **43**
  pořadí sloupců ve výstupu **44**, *92*
  slučování oddělovačů **52**, *75*
  výběr podle pozice (-c) **43**, *52*, *160*, *200*, *209*
  výběr podle sloupce (-f) **43**, *45*, *46*, *52*, *75*
  vynechávání neúplných řádek (-s) 79, *282*
  zadání seznamu sloupců **44**, *52*
  zpracování souboru v parametru 43
  zpracování standardního vstupu 46
cyklus
  (awk) **325**
  (ed) viz ed:opakování příkazu
  (sed) 102
  (shell) **342**
  přerušení cyklu viz break
  přeskočení zbytku těla viz continue
  přesměrování 164, *185*, *208*
  ukončení cyklu viz break
  ukončení provádění těla viz continue

Č

čárka viz ,
čas viz datum a čas
čísla
  generování řady čísel 166
  kontrola zápisu 159
  porovnávání (awk) 219
  porovnávání (expr) 167
  porovnávání (test) 158
  procesů viz proces:číslo
  řádek viz řádky:číslování
  signálů viz signály
  skupin viz skupina:číslo
  uživatelů viz uživatel:číslo
  výpis (awk) 232, 237
  výpis (printf) 233
čitelnost kódu viz srozumitelnost kódu

členství ve skupině viz skupina:členství
čtecí přístupové právo 57
čtení odpovědi uživatele
viz interakce s uživatelem
čtení proudu dat (shell) 164
čtení přesného textu 165, 293
čtení souboru (shell) 164
čtení z here-dokumentu *154*
čtení z roury 145, *154*, *186*, *277*, *289*

D

-d (test) **158**, *182*, *264*
d jako typ souboru 38
databáze
spojení tabulek 80
v souboru DBF 207
v textovém souboru 43
date **27**, *53*, *212*, *257*, **307**, **351**
datové typy proměnných
viz typ proměnné
datum a čas
aktuální viz date
formát uložení 30
poslední modifikace souboru
porovnání 82, 332
vyhledávání viz find
výpis viz ls
zachování 29, 132
změna viz touch
DBF 207
DD (Data Definition) 208
dd (utilita) **208**
dekrementace (awk) viz --
dělení viz aritmetika
deskriptor 37
0 viz standardní vstup
1 viz standardní výstup
2 viz standardní chybový výstup
3 a vyšší 203
duplikace vstupu 187, 203
duplikace výstupu 142
uzavření 187
dev (adresář) viz /dev
df **63**
diff **79**, **308**, **351**
výstup ve tvaru ed-skriptu (-e) 123
digit (třída znaků) 91, 347
dirname 309
disk viz souborový systém
divide et impera 35, 199
do (awk) 326
do (shell) 163, 164
dokumenty RFC viz RFC
dolar viz $
done (shell) 163, 164, 165
dosazení viz substituce
duplicity
uvnitř řádek *94*
vyhledávání viz uniq
vylučování viz sort
duplikace deskriptoru 142, 156, 187, *203*
dvojtečka viz :

E

-e (shell) 337
-e (test) **158**, *169*
-eq (test) **158**, *204*
EBCDIC 308
ed **85**, **116**, **328**
adresa **328**
aktuální řádka 125
číslo řádky 117, 124
hledání vpřed 118
hledání vzad 119
implicitní rozsah 121
offset 118
poslední řádka viz $ (ed)

posun 118
rozsah řádek 119
vzor (regulární výraz) 118
aktuální řádka **118**, *119*, 120, *125*
analýza *284*
append (a) *116*, **117**, *123*, *288*
delete (d) **118**, *119*, *123*, *152*, *288*
formát příkazu 117
global (g) **119**, *121*, 125
hledání řádky s řetězcem
další řádka viz ed:adresa
reakce na chybu 163
všechny řádky viz ed:global
change (c) **117**, *123*
implicitní regulární výraz 126
insert (i) **117**, 125
interaktivní editace 117, *149*
invert (v) **120**
join (j) 329
kopírování řádek viz ed:transfer
ladění skriptů 126
mazání řádek viz ed:delete
mazání řetězce viz ed:substitute
move (m) **121**, 125
náhrada řádek viz ed:change
náhrada řetězců viz ed:substitute
number (n) 329
opakování příkazu 119
potlačení statistik (-s) 116
print (p) **121**
přepínače
-s (potlačení statistik) 116
přesun řádek viz ed:move
quit (q) **117**, 126
Quit (Q) **121**
skript 122, 124
spojení řádek viz ed:join
substitute (s) **117,**
viz též sed:substitute
implicitní vzor 126
transfer (t) 330
ukončení editoru 126
uložení souboru viz ed:write
v (invert) **120**
vložení před řádku viz ed:insert
vložení řetězce viz ed:substitute
vložení za řádku viz ed:append
výpis řádek viz ed:print
write (w) **117**, *152*, *288*
zadání příkazů ze skriptu 152, *162*
zápis do souboru viz ed:write
změna čísel řádek při editaci 124
editace proudu dat viz sed
editace souboru viz ed
editace výstupu příkazu *49*, 87
efektivita 73, 74, 80, 96, 112, 117, 122, 146, 148, 163, 169, 199, 201, 203, 209, 210, 213, 228, 230, 246, 251, 253, 262, 290
echo **26**
efektivita 146
escape-sekvence 309
kontrola parametrů příkazu 33
odřádkování 26, *33*, *68*
seznam souborů 38
více parametrů 136
výpis bez odřádkování **68**, *187*, *196*
výpis hodnoty proměnné 135, *138*, *160*, *161*, *184*, *226*
výpis textu do roury 138, 160, *226*
výpis textu na terminál 135, *144*, *185*
výstup do chybového proudu 156
vytvoření souboru 26, *29*, *32*, *60*
záměna s cat 26
elektronická pošta 283, 290, 350
elif (shell) viz if
else (shell) viz if
emulace *127*, *212*
END (awk) **216**, 231, 233, *238*, *243*
Enter 26, 187

ENVIRON[] (awk) **248**
environment viz proměnné prostředí
epocha 30
ERE viz regulární výraz:rozšířený
errexit režim 337
esac viz case
escape-sekvence (awk) 233
escape-sekvence (echo) 309
escape-sekvence (paste) 314
escape-sekvence (printf) 196, 210, 314
escape-sekvence (sed) 101
escape-sekvence (tr) 45, 320
etc (adresář) viz /etc
eval **177**, *182*, *184*, *201*, 234, *269*
exec 187, *206*
execute (právo) 57, *59*, *60*, *274*
exit (awk) **233**, *242*, **244**, *256*, *272*
  návratová hodnota 244
exit (shell) **160**, *173*, *176*, *190*
  návratová hodnota 160, *197*
  za rourou 186
EXIT (trap) 193
expanze aritmetická
    viz substituce:aritmetická
expanze jmen souborů **31**
  adresáře 38, 70
  jako etapa zpracování řádky shellu
    31, *33*, *73*, *131*, *132*, *148*, *244*
  pořadí při zpracování řádky 137, 178
  skryté soubory 36
  soubory nenalezeny 37
  uspořádání jmen 33
  výhodná volba jmen souborů 76
  zákaz expanze viz -f (shell)
expanze parametrů
    viz substituce:proměnných
expanze proměnných
    viz substituce:proměnných
expanze výrazu viz substituce:výrazu
expanzní znaky **31**, 41, **131**, **137**, 196
export **196**
export všech proměnných viz -a (shell)
expr **166**, *183*, *192*, *195*, *243*
  návratová hodnota 167, 198, 309
  parametry s metaznaky shellu 167
  regulární výrazy 167
  standardní výstup 167, 309

F

-f (shell) 196
-f (test) **158**, *159*, *174*
false 198
fi viz if
FIFO (First In First Out) 212
file (utilita) **73**
FILENAME (awk) **223**, 227, *232*, *235*,
    238, *256*
filesystem viz souborový systém
filtr 43, 213
find *41*, **69**, *142*, *185*, **331**
  ! (negace) 71
  -a (logické „a“) 71
  -exec 69
  -exec a expanze 73
  -exec jako podmínka 72
  -name *69*, **71**, *131*, *140*
  -o (logické „nebo“) 71
  -path 71, *131*
  -prune 71
  -type 70, *72*
  nevhodné použití 70
  skryté soubory 71
for (awk) **221**, 222, *228*, *238*, *242*
for (shell) **162**
  cyklus nad parametry **171**, *183*
  cyklus nad soubory **162**, *188*
  čtení souboru 165
  počet opakování 166
for in (awk) **231**

fork 145
formátování odstavců *107*, *128*, *257*
formátování textu (awk) 273
formátování výstupu (awk) 232, 237
formátování výstupu (printf) 196, 348
fragment (URI) 261
fronta 212
FS (awk) **233**, **236**, *246*, *272*
funkce (awk) viz awk:funkce
funkce (shell) **176**, *262*
  definice 176
  parametry 181, 189
  proměnné 189
  rekurzivní volání 188
  volání 181
  vs. subshell 191

G

-ge (test) **158**
-gt (test) **158**, *171*, *174*, *182*, *197*, *205*, *264*, *281*
generování skriptu *123*, 132, *267*
getopts **174**, *176*, *178*
GID **24**, **45**, 57, *81*, *94*, *104*, *118*, *201*, *203*, *218*, *228*, *249*, 349
globbing viz noglob režim
graph (třída znaků) 347
grep **38**, *46*, *73*, *90*, *159*, *161*
  efektivita 96, 112, 163
  etymologie názvu 122
  hledání bloků řádek 111
  hledání celých slov (-w) 50, *161*
  inverzní výběr (-v) 39, *120*
  jako SQL SELECT 43
  počet nalezených výskytů (-c) 38
  pouze kontrola shody (-q) 160
  pouze test výskytu (-q) 72, *161*, *279*
  rozšířené regulární výrazy (-E) 218
  shoda na celé délce (-x) 277
  výpis čísel řádek (-n) 77, *286*, *288*
  výstup při více souborech 51
  vzor jako řetězec (-F) 277
  vzor začínající minusem (-e) 41
  zadání více vzorů (-e) 39
group (konfigurační soubor) viz /etc/group
gsub() (awk) **236**

H

h (přípona) *131*, *132*, *255*
hardlink **62**, *68*
  kopírování 65
  přesun 67
  vytvoření viz ln
hardware 321
hash kříž viz #
head **42**, *82*, *140*, *144*, *209*
here-dokument **153**, *264*
hexadecimální výpis *207*, 348
hexadecimální zápis 314
hierarchický systém adresářů 21
hlavička HTTP odpovědi 261
hlavičkový soubor *131*, *132*, *255*
hledání celých slov 50, 97, 161
hledání čísel *95*
hledání maxima *171*
hledání minima *171*
hledání na celé délce
  v porovnávacích vzorech **131**, 161
  v regulárních výrazech 40, 93, 131
hledání podřetězců
  v porovnávacích vzorech **161**
  v regulárních výrazech **38**, *40*, *90*, *93*, *131*
hledání přesných řetězců 277
hledání řádek viz řádky:vyhledávání
hledání souborů viz find

hledání umístění příkazu
viz příkaz:hledání umístění
hledání více řetězců 39, *98*, 218
hodnota návratová
viz návratová hodnota
hodnota proměnné (awk)
viz awk:proměnné
hodnota proměnné (shell)
viz proměnné (shell)
HOME 133, **337**
hostname (utilita) 93
hranaté závorky viz [...]
HREF (HTML) 261
HTML (Hypertext Markup Language)
107, *127*, *211*, *239*, *257*, *261*, 353
HTTP (Hypertext Transfer Protocol)
261, 353
odpověď 261, 275
požadavek 261
stavová řádka 278
hvězdička viz *

Ch

chmod **58**, *59*, *60*, **72**, *149*
chybové zprávy
potlačení 37
výpis 156, *195*
chybový výstup
viz standardní chybový výstup

I

IBM 208
id **46**, *49*, *50*, *100*, *118*, **352**
if (awk) **222**
if (shell) **156**, 161
vs. podmíněné spuštění 155, 156
IFS **136**, 234, **337**
dělení řádky shellu
postup 136, 178
výjimky 137, 140, 160, 178
oddělovač polí (read) **147**, **164**
oddělovač polí (shell) **136**, *140*, **178**,
180, *262*
slučování oddělovačů **141**, 165
indentace 99, 153
index() (awk) **220**, *226*, *227*, *237*, *247*
index.html *262*, *283*
indexový uzel viz i-node
inkrementace (awk) viz ++
i-node 62, 64
INT (signál) 191
int() (awk) 217
interakce s uživatelem 144, *257*
internetové normy viz RFC
interní příkazy (shell) 59, 144, 145,
148, 262, 343
interpret příkazů 1, 133, 274
interpretovaný jazyk 74, 132, 161, 172,
177, 201, 238, 246
IP adresa 95
ISO-8859-2 74

J

JCL (Job Control Language) 208
jméno počítače viz hostname
jméno skriptu ($0) 149
jméno souboru viz soubor:jméno
jméno uživatele viz uživatel:jméno
join **80**, *84*, *205*, *206*

K

kill **193**
knihovny funkcí (awk) 274
knihovny funkcí (sed) 96
knihovny funkcí (shell) 199

kolona 48
komentáře viz #
konec řádky
  (echo) 26, *33*, *68*
  (shell) 26, 47, 155
  připsání za 99
  souborů v MS Windows 210
  souborů v UNIXu 24
  vložení (sed) 101, *115*
  vymazání (paste) 49
  vymazání (tr) 45
konec shellskriptu viz exit
konec souboru
  připsání za viz >> (shell)
  výpis viz tail
konec vstupu
  detekce (read) 164
  výpis viz tail
  vyvolání (Ctrl+D) 25
konečný automat 221, 243
kontrola dat 227, 234
kontrola parametrů *33*, *159*, *161*, *169*, *173*, *176*, *177*, *182*, *211*
kontrola rejstříku *258*
kontrola uzávorkování *128*
kontrola výrazu *243*
konverze znaků viz tr
kopírování adresářů 66
kopírování dat (dd) 208
kopírování deskriptoru
    viz duplikace deskriptoru
kopírování řádek (sed) 106
kopírování souborů 59
kopírování vstupu
  do souboru 48
  na výstup 48, 144
Korn shell 167
kořen systému souborů 21
kvadratický algoritmus 201, *203*

L

-L (test) **158**, *169*, *190*
-le (test) **158**, *166*, *183*, *281*
-lt (test) **158**, *159*, *171*, *204*
l jako typ souboru 63
ladění shellskriptů 172
ladění skriptů (awk) 244
ladění skriptů (ed, sed) 126
LC_ALL 337
length() (awk) **237**, *272*
LF (line feed) viz znak konce řádky:LF
libovolný znak (case, find, shell) viz ?
libovolný znak (regexpy) viz .
libovolný znak ze seznamu viz [...]
LIFO (Last In First Out) 212
line feed viz znak konce řádky:LF
link
  hardlink viz hardlink
  symbolický viz symlink
ln **62**, *149*
logická negace viz !
logická spojka „a“ viz a zároveň
logická spojka „nebo“ viz nebo
login viz uživatel:jméno
login shell 24
LOGNAME **337**
lomítko viz /
lower (třída znaků) 44, 347
ls **22**, *34*, *60*, *70*, *84*, *132*, *148*
  COLUMNS 200
  dlouhý výpis (-l) **22**, *29*, *52*, *191*, *215*, **352**
  dlouhý výpis u symlinku 63
  formát dlouhého výpisu 352
  formát výpisu data a času 30
  implicitní chování 23
  numerický výpis (-n) *84*, 201, *235*
  obrácení pořadí výpisu (-r) 30
  potlačení třídění výstupu (-f) 33

ls *(pokračování)*
potlačení výpisu obsahu adresáře (-d) 23, *36*, *169*, *215*, *245*
rekurzivní výpis (-R) 39
skryté soubory 36
sleduj link (-L) 63, *168*
třídění výstupu 30
podle času (-t) 30, 82, *147*
potlačení (-f) **33**
pozpátku (-r) 30
výpis detailních informací 22
výpis i-node (-i) 62
výpis obsahu adresáře 23
výpis přístupových práv 58, *245*
výpis skrytých souborů (-a) 37, *193*
výpis typu souboru 38
závislost formátu na typu výstupu 34
zpracování výstupu *52*, *75*, *80*, *144*, *147*, *191*, *215*, *216*

M

MAIL 290, 350
mailx 286
makro 255
man **22**
manuálové stránky 22
maskování přístupových práv 61
match() (awk) 224
mazání
duplicitních řádek 78
linků 63
obsahu řádky 93
obsahu souboru 27, 29, 33, 68
proměnných 345
řádek *28*, 89, 93, 118, *152*
souborů viz rm
znaků (sed) 88
znaků (tr) viz tr
menšítko viz <
metaznaky (regexpy) 40, 90, **346**
metaznaky (shell) **26**, 30, 32, **133**, **338**
metaznaky ve jménech souborů 33
mezera (awk) 215, 223
mezera (HTML) 239
mezera (shell) **22**, 26, 30, 135
mezera ve jménech souborů 30, 33, *277*
minus viz -
mkdir **37**, *58*, *67*, *156*
jako zámek 284
vytvoření plné struktury (-p) 263
mód souboru viz přístupová práva
modifikace sloupců souboru *43*, *109*
modifikace souboru 28, 87
Multics 23
mv
přejmenování souboru **28**, **66**, *281*
přesun více souborů **66**, *67*, *150*, *182*
změna linku vs. přesun dat 67

N

-n (shell) 172
-n (test) **158**, *176*
-ne (test) **158**, *204*
náhodná čísla viz rand()
náhrada znaků viz záměna znaků
nápověda 22
národní prostředí *74*, *91*, *112*, **337**
následující řádka 100, 118, 119, 125
násobení viz aritmetika
návrat vozíku viz CR
návratová hodnota *72*, **155**, *156*
funkce (shell) 190
negace výsledku 157
po substituci příkazu 155, 198
podmíněného příkazu 157
předání výsledku 191
příkazu null (:) 198
přiřazení do proměnné 198

přístup k hodnotě viz $?
roury 155
skriptu 160, *195*
složeného příkazu 155
volání funkce system (awk) 252
nawk 224
nebo (awk, shell) viz ||
nebo (expr) viz |
nebo (find, test) viz -o
negace viz !
nepodmíněné vyhodnocování výrazu
viz vyhodnocení výrazu
nepřerušitelnost 284
nepřímé odkazy na proměnné 177
nepřiřazené proměnné (awk)
viz awk:proměnné
nepřiřazené proměnné (shell)
viz proměnné (shell)
netcat *261*
next (awk) **221**, *232*, *235*, *238*, *244*, *247*
NF (awk) **220**, *222*, *224*, *247*
noclobber režim 284, 337, 339
noexec režim 337
noglob režim 196, 337
norma **1**, 30, 32, 50, 51, 66, 68, 71, 83, 88, 105, 133, 138, 146, 160, 167, 174, 180, 187, 190, 194, 196, 198, 201, 209, 210, 218, 224, 230, 235, 253, 256
nounset režim 337
NR (awk) **227**, *247*, *278*
NUL (znak) *39*, **74**, *207*
null (pseudosoubor) viz /dev/null
null příkaz viz prázdný příkaz
numerické třídění viz sort

O

-o (find) 71
-o (test) **158**, *192*
obracení pořadí řádek souboru *121*
obrácení pořadí třídění (sort) 74
obrácení pořadí výpisu (ls) 30
obracení textu řádky *113*
obsluha signálu 191
od (utilita) **207**
oddělovač adresářů 21
oddělovač parametrů 22, 136
oddělovač polí (awk)
viz awk:oddělovač polí
oddělovač polí (cut) 43
oddělovač polí (join) 80
oddělovač polí (paste) 44
oddělovač polí (shell) viz IFS
oddělovač polí (sort) 75
oddělovač příkazů (awk) 222
oddělovač příkazů (sed) **89**, **96**, *103*
oddělovač příkazů (shell) **50**, *69*, *143*, *157*, *163*
odečítání viz aritmetika
odpověď HTTP 261
odstavce
formátování *107*, *128*, *257*
převod do HTML *107*
odstranění viz mazání
OFS (awk) **234**, *236*
oktalový výpis *207*, 348
OL (HTML) 240
omezení délky výpisu řetězce 348
opakování podvýrazu (regexpy) 40, 93, 94, 218
opakování znaků (tr) 47
operační systém (typ) 254
operand operátoru přesměrování **24**, 152, 178
operand příkazu **22**
operátory (awk) **324**
operátory (expr) **167**
operátory (find) **71**

operátory (shell)
detekce při čtení řádky 135, 178
priorita 142
přesměrování *24*, *26*, *35*, **339**
řídicí *34*, *146*, *155*, **338**
operátory (test) **158**
OPTARG viz getopts
OPTIND viz getopts
option viz přepínače
ordinální hodnota znaku 74
ORS (awk) **240**
ošetření signálu 192
otazník viz ?

P

P (HTML) 107
paralelní běh 283
parametry funkce (shell)
viz poziční parametry
parametry příkazu
oddělovač parametrů 22
omezení počtu 73, 83
operandy vs. přepínače 22
vs. vstup 82
zadání množiny souborů 31
parametry shellskriptu
viz poziční parametry
párování hodnot sloupečků *80*
parsování 147
passwd (konfigurační soubor)
viz /etc/passwd
paste **43**
oddělovač sloupců (-d) **44**, **49**, *76*
přidávání oddělovače k řádce 51
sekvenční spojování řádek (-s) **49**, *76*
spojování řádek více souborů za sebe **43**, *76*, *205*
PATH **134**, *140*, *149*, **337**
modifikace hodnoty 134
rozdělení na pole 140, *165*, *180*
pax *66*, *318*
Perl *213*
permission viz přístupová práva
PID viz proces:číslo
plovoucí čárka 348
plus viz +
počet
dnů v měsíci *202*, *211*
řádek (wc) viz wc
slov (shell) viz $#
slov (wc) viz wc
souborů 38
souborů v adresáři *34*, *36*, *38*, *39*
uživatelů systému 24
znaků (awk) viz length()
znaků (shell) viz ${*#jméno*}
znaků (wc) viz wc
podadresář viz adresář:podadresář
podmíněná substituce
viz ${*jméno*:-slovo}
podmíněné spuštění příkazu 155
podmíněné vyhodnocování výrazu
viz vyhodnocení výrazu
podmíněný příkaz (shell) 156
podřetězce
hledání v porovnávacích vzorech 159, 161
hledání v regulárních výrazech 38, 40, 90, 93, 131
podtržítko viz _
pointer 177
pokračovací řádka (shell) 47, 155
pole (indexovaná proměnná)
(awk) 228
(shell) 201

pole (textového souboru)
modifikace *43*, *109*
přehazování *92*
spojování viz join
stříhání viz cut
třídění viz sort
pole příkazové řádky
viz příkazová řádka:pole
pole vstupní řádky (awk) viz awk:pole
pole vstupní řádky (cut) 43, 52
pole vstupní řádky (shell)
načtení do proměnných 147, 164
popd 212
porovnání obsahu souborů 79
porovnání řetězců
viz řetězce znaků:porovnání
porovnávací vzory **31**, 41
(case) 159
(find) 71, 131
(shell) 131
* *41*, **338**
? **338**
[!...] **131**, **338**
[...] **131**, **338**
expanze jmen souborů 70
hledání podřetězce 161
libovolný řetězec viz *
libovolný znak viz ?
shoda na celé délce 131, 161
substituce podřetězců 293
znak z doplňku seznamu viz [!...]
znak ze seznamu viz [...]
pořadí jmen souborů při expanzi 33
pořadí substitucí
viz shell:postup zpracování řádky
POSIX 1, **353**
poslání signálu 193
poslední parametr 182
poslední řádka (ed, sed) viz $
poslední řádka (tail) 28
postup zpracování řádky
viz shell:postup zpracování řádky
posun parametrů viz shift
pozadí 338
poziční parametry (shell) 148
expanze všech 150
implicitní hodnota 152
nastavení 148
počet viz $#
posun viz shift
předání jinému příkazu 150
zpracování 152, 171
požadavek HTTP 261
PPID 194
práva viz přístupová práva
prázdná hodnota proměnné 137, 152
prázdná řádka
v regulárním výrazu 40, 108
ve vstupu (read) 187
prázdné pole (sloupec) 51, 52, 165
prázdný příkaz (:)
návratová hodnota 198
syntaktická vata 157, 183
založení souboru 68, *281*
preprocesor viz textový preprocesor
print (awk)
oddělovač parametrů 234
oddělovač záznamů 240
výpis parametrů **215**, *216*, *220*
výpis vstupního záznamu **219**, *235*
výstup do roury **254**, *272*
výstup do souboru **233**, *249*
print (třída znaků) 347
printf **348**
%d 232
%s 196, *232*
modifikátory direktiv 237
printf (awk) **232**, *240*
formátovací řetězec v proměnné 237
výstup do souboru **233**, *249*

printf (utilita) *68*, **196**
  escape-sekvence 196, 210
  rychlost 187
procento viz %
proces **145**, *176*
  číslo **191**, *196*, *252*
  informace 184
  rodičovský **145**, *194*, *252*
  synovský **145**, *154*, *186*, *196*
program pro interpret viz skript
procházení stromu
  (find) 69
  do hloubky 188, 331
  do šířky 189
proměnné (awk) viz awk:proměnné
proměnné (shell)
  datový typ 137
  délka hodnoty viz ${*#jméno*}
  expanze hodnoty 133, 153
  exportované proměnné 196
  globální 176, *183*, *189*
  identifikátor 137
  implicitní hodnota 137, 152
  indexované viz pole
  inicializace 137
  kopírování 140
  lokální 181, *189*, 190
  mazání 345
  modifikace hodnoty 134, 138
  načtení hodnoty viz read
  název 137
  nenastavená hodnota 137, 152, 337
  odebrání z prostředí viz unset
  odstranění viz unset
  podmíněná substituce 152
  podmíněné přiřazení
    viz ${*jméno*:=*slovo*}
  použití hodnoty 133
  použití v kódu jiných programů 225
  předávání subshellu 196
  přiřazení hodnoty 134, *146*, 152, 165
  přiřazení výstupu programu 138
  vložení do prostředí viz export
  vymazání z prostředí viz unset
  výpis hodnoty 135
  výpis všech proměnných 137
  zařazení do prostředí viz export
proměnné prostředí 133, 248
  nastavení pro příkaz 165
  předávání subshellu 196
  přístup v awk 248
prompt **21**
  nastavení 337
  sekundární 45, *47*, *337*
  výpis cesty 21, *23*, *37*
prostředí viz proměnné prostředí
protokol 261
proud dat 203
  čtení (shell) 164
  editace viz sed
  chybový
    viz standardní chybový výstup
  konec vstupu 25
  vstupní viz standardní vstup
  výstupní viz standardní výstup
provádění příkazů 26, 59
předávání výsledku 155, 190, 191, 195, 244, 267
předcházející řádka 111, 119, 164
přejmenování souborů viz mv
přenositelnost 1, 35, 47, 51, 66, 68, 83, 88, 105, 133, 148, 167, 174, 180, 193, 198, 210, 213, 224, 228, 230, 251, 253, 254, 276
přepínače **22**
  s hodnotou **28**, 41
  shellu 337
  slučování **23**, **41**, *174*
  ukončení seznamu 32
  zpracování 174

přerušení programu 191
přesměrování
  duplikace vstupního deskriptoru 187, *203*
  duplikace výstupního deskriptoru 142, *156*
  here-dokument 152
  chybového výstupu 37
  priorita 142
  pro celý cyklus 164, *209*
  přepisování souboru 26, 337
  přepsání vstupního souboru 28
  příkazem exec 187
  připisování za konec souboru 27
  vstupu 24
  výstupu (awk) 233
  výstupu (shell) 26, 27, *35*
  zavření vstupního deskriptoru 187
přesný počet opakování (regexpy) viz \{...\}
přesouvání řádek (sed) 106
přesouvání souborů viz mv
převod číselných soustav 314
převod na celá čísla 217
převod UID a GID na jména *80*, *201*, *235*, *237*
převrácení pořadí řádek souboru *121*
převrácení textu řádky *113*
přidávání řádek *29*, 105, 106
příkaz
  hledání umístění 59, 134, 140
  interní viz interní příkazy (shell)
  kontrola syntaxe viz -n (shell)
  nastavení prostředí 165
  návratová hodnota 155
  obrácení výsledku 157
  oddělovač viz oddělovač příkazů
  parametry viz parametry příkazu
  podmíněné spuštění 155
  složený **143**, *144*, *146*, *277*
  vestavěný viz interní příkazy (shell)
  volání programu 59
  výpis před provedením viz -x (shell)
  výsledek viz návratová hodnota
příkazová řádka **21**, 138, 139
  omezení počtu parametrů 73, 83
  pole 30, 135, 151, 178
  postup zpracování 135, 140, **178**
  rozdělení do více řádek **47**, *71*, **155**
  slova viz příkazová řádka:pole
  ukončení 26
příkazový soubor
  (awk) viz awk:skript
  (ed) viz ed:skript
  (sed) viz sed:skript
  práva viz přístupová práva
  spouštění 274
přípona jména souboru 41
připsání textu do souboru *27*
přístupová práva 57, *68*, *245*
  execute **57**, *59*, *60*, *274*
  příkazový soubor 60, *274*
  read **57**, *60*, *142*
  search **57**
  write **57**, *61*, *68*
ps *53*, **184**, *211*, *252*, **352**
PS1 337
PS2 337
pseudosoubor /dev/null 37, 51
pseudosoubor /dev/stderr 180
pseudosoubor /dev/stdin 180
pseudosoubor /dev/stdout 180
pseudosoubor /dev/tty 180, 186
punct (třída znaků) 347
pushd 212
pwd **21**, *169*, *188*

Q

QUIT (signál) 193
quoting **136**, *153*, **178**, *225*
  apostrofy **30**, *45*, *48*, *69*, *71*, *88*, *89*, **136**
  nesprávné použití 32, 136
  pokračovací řádka **47**, *51*, *71*
  uvozovky **136**, *170*
  ve vstupu (read) 165
  zpětné apostrofy 139
  zpětné lomítko **47**, **136**, *153*, *265*

R

-r (test) **158**, *159*
rand() (awk) 327
read **144**
  čtení celé řádky 165
  čtení po slovech 196
  čtení z jiného deskriptoru *203*
  čtení z roury 145, *154*
  návratová hodnota **164**
  potlačení quotingu (-r) 165
  s více parametry **147**, *184*
  slučování oddělovačů polí 147
  zpětná lomítka ve vstupu **165**
read (právo) 57, *60*
regexp viz regulární výraz
regulární výraz 38, **41**, 131, **346**
  (awk) 216, 218
  (ed) 118
  (expr) 167
  (grep) 38
  (sed) 88, 94
  BRE 94, 346
  ERE 218, 346
  jako adresa (ed) 118, 119
  jako adresa (sed) 96, 100
  libovolný znak (.) 40
  metaznaky 40, 91, *170*, **346**
  negace 96, 120
  počet výskytů
    alespoň jeden (+) 218
    libovolný (*) 40
    nejvýše jeden (?) 218
    zadaný intervalem (BRE) viz \{...\}
    zadaný intervalem (ERE) 218, 346
  quoting 40, *170*
  rozšířený 218, 346
  shoda na celé délce 40, 93
  shoda v podřetězci 38, *40*, *90*, *93*
  ukotvení na konec ($) 40
  ukotvení na začátek (^) 38
  uzávorkování podvýrazu (BRE) 93
  uzávorkování podvýrazu (ERE) 219
  výběr z více variant (|) 218
  vždy vyhovující 77, 100, 121
  základní 94, 346
  zpětné reference 94
rekurzivní algoritmus 188, 191, 195, 199, 276
rekurzivní kopírování podstromu 66
rekurzivní volání
  funkce v shellu 188
  shellskriptu 190
rekurzivní výpis podstromu 39
relativní cesta viz cesta:relativní
return (awk) **230**, *272*
return (shell) **190**
rev 113
RFC (Request for Comments) 111, 119, *127*, *212*, 241, *257*, **349**
rm **27**, *32*, *35*, *82*, *155*
  potlačení chyb (-f) **32**, *197*
rodičovský adresář viz adresář:rodičovský
rodičovský proces viz proces:rodičovský
rotace souborů 212

roura **34**
(awk) 252
čtení vstupu více příkazy 144
čtení z roury 145
implementace 145
jako vstup cyklu *185*, *279*
jako vstup pro editor 152
ladění 48
spojení výstupů do roury 143
v řídicí konstrukci if *157*
zpracování chybového výstupu 142
rovnítko viz =
rozděl a panuj viz divide et impera
rozdělení příkazové řádky na pole
viz shell:postup zpracování řádky
rozdělení řádek (sed) 101, *115*
rozdělení řádek (tr) 46, *282*
rozdělení řádky programu (awk) 222
rozdělení řádky skriptu (shell) 47, 155
rozdělení souboru (split) 49
rozdělení vstupní řádky na pole
viz pole vstupní řádky
rozdělení vstupu mezi příkazy 144
rozvinutí viz substituce
RS (awk) 239, *272*
rušení viz mazání
rušení významu metaznaků (regexpy)
viz regulární výraz:quoting
rušení významu metaznaků (shell)
viz quoting

Ř

řádky
bloky viz bloky řádek
číslování (awk) viz awk:čísla:řádek
číslování (cat)
viz cat:výpis čísel řádek (-n)
číslování (ed) viz ed:number
číslování (grep)
viz grep:výpis čísel řádek (-n)
kopírování (ed) viz ed:transfer
kopírování (sed) 106
mazání duplicit 78
náhrada (ed) viz ed:change
náhrada (sed) viz sed:change
porovnávání s regexpem
(grep) 160
konec řádky viz $ (regexpy)
začátek řádky viz ^ (regexpy)
porovnávání se vzorem viz case
přesun (ed) viz ed:move
přesun (sed) 106
převrácení textu *113*
rozdělení na pole 147, 164
smazání (ed) viz ed:delete
smazání (sed) viz sed:delete
spojování (ed) viz ed:join
spojování (sed) viz sed:Next
třídění viz sort
vložení
před řádku (ed) viz ed:insert
před řádku (sed) viz sed:insert
za řádku (ed) viz ed:append
za řádku (sed) viz sed:append
vyhledávání
(awk) viz awk:vzor
(ed) viz ed:adresa
(grep) viz grep
(sed) viz sed:adresa
výpis duplicit viz uniq
zjištění počtu (wc) 24
řetězce znaků
délka (awk) viz length()
délka (shell) viz ${*#jméno*}
formátování viz sprintf()
hledání podřetězce (awk) 220, 223
náhrada (awk) viz sub()

řetězce znaků *(pokračování)*
náhrada (ed) viz ed:substitute
náhrada (sed) viz sed:substitute
porovnání
abecední 167, 219
lexikografické 167, 219
na rovnost 158, 167, 219
s porovnávacím vzorem 159
s regexpem 159, 161, 167, 223
použití části řetězce v náhradě 92
rozdělení na pole viz split()
uložení do paměti (regexpy) 92
výběr podřetězce viz substr()
vyhledávání přes více řádek *110*
vyhledávání řádek viz grep
výpis (printf) 196
vytváření dle formátu viz sprintf()
zřetězení (awk) 223
zřetězení (shell) 134
řídicí konstrukce (shell) **342**
řídicí operátory (shell) **338**
řídicí proměnná cyklu (shell) 162
řídicí struktury (shell) 157

S

-s (test) **158**, *281*
sčítání viz aritmetika
sdružování přepínačů
viz přepínače:slučování
search (právo) 57
sed **85**, **87**, **333**
adresa 89, **333**
číslo řádky 89
doplněk viz ! (sed)
oddělovač řetězců 96
poslední řádka viz $ (sed)
rozsah čísel řádek 89
rozsah vzorů 100, *239*
vzor (regexp) 96
alternativní umístění skriptu (-f) 96
analýza *236*, *267*, *275*, *293*
append (a) **105**
branch (b) **102**, **107**, *110*, *111*, *269*
cyklus 102
delete (d) **89**, **96**, *98*, *104*, *269*
Delete (D) **115**, *269*
editace souboru *116*
exchange (x) **106**, *107*, *109*
formát příkazů 89, 333
formátování výstupu *202*
get (g) **106**, *112*
Get (G) **107**, *109*
hledání celých slov 97
hold (h) **106**, *107*, *109*
Hold (H) **107**
change (c) **105**
insert (i) **104**, **105**, *116*
komentáře 99
konec řádky
jako zarážka 113
v pracovním prostoru 99, 107, 110, 112, *113*
v řetězci náhrady 101, *114*, *269*
ve vzoru *101*, 107, *110*, 112, *114*
konverze znaků (y) **99**, 109
ladění skriptů 126
mazání řádek viz sed:delete
mazání řetězce viz sed:substitute
načtení další řádky viz sed:next
náhrada řádek viz sed:change
náhrada řetězců viz sed:substitute
návěští (:) 102
nepodmíněný skok viz sed:branch
next (n) **101**, 221
Next (N) **110**, *111*, *127*, *269*
oddělovač příkazů **89**, **96**, *103*
odkládací prostor **106**, *109*, *112*, *293*
paměť 112, 293
podmíněný skok viz sed:test

potlačení výstupu editace (-n) **89**, *97*, *99*, *104*, *288*
použití proměnných shellu 169, *202*
pracovní prostor **98**, 111
  konec 112, 115
  výpis 97
  začátek 112, 115
print (p) **89**, *104*
Print (P) **115**, *270*, *297*
první řádka pracovního prostoru 115
přehazování sloupců *138*
přepínače
  -f (název souboru se skriptem) 96
  -n (potlačení výstupu editace) 89
quit (q) **89**
skok 102
skript **96**, 101, *212*
složený příkaz **98**, *111*, *143*, *269*
smazání řádky vs. vymazání obsahu 93, 115
spojování řádek viz sed:Next
substitute (s) **87**
  konec řádky 99, 101, 107, 110, 112, 113
  mazání znaků 88, *93*, *98*, *112*
  metaznak & 93, *115*
  náhrada všech výskytů (parametr g) **100**, 101, **102**
  oddělovač řetězců 88, 103, *170*
  podmínka pro skok 102, 113, 270
  použití vzoru v náhradě viz & (sed)
  přehazování sloupců 92
  tisk výsledku substituce (parametr p) **97**, 101
  začátek a konec regexpu 93
  zápis výsledku do souboru (parametr w) **97**
  zpětné reference **92**, 110, *113*, *115*, **295**
  test (t) **102**, **107**, *113*, *115*, *269*
  ukončení práce viz sed:quit
  úprava následující řádky 100
  vložení před řádku viz sed:insert
  vložení řetězce viz sed:substitute
  vložení za řádku viz sed:append
  vymazání obsahu řádky vs. smazání 93, 115
  výpis řádky **97**, *98*, **115**
  vytváření skriptů *132*, *212*
  write (w) **117**
  x (exchange) **106**, *107*, *109*
  y (konverze znaků) **99**
  zadání skriptu (-f) 96
  zápis do souboru viz sed:write
seek 209
SELECT (SQL) 43
semilogaritmický formát 348
set
  nastavení pozičních parametrů **148**, *183*, *202*, *262*
  nastavení přepínačů 200
  výpis proměnných 137
  výpis přepínačů (-o) 344
sh 73, 132, **337**
shell 1, 24, 60, **337**
  analýza *262*, *275*, *284*, *291*
  číslo procesu viz $$
  postup zpracování řádky 135
  prázdný příkaz (:) 152
  proměnné viz proměnné (shell)
  přepínače **337**
    aktuální nastavení viz $-
    export všech proměnných (-a) 196
    ladění (-n, -v, -x) 172
    nastavení (#!) 198
    nastavení (set) viz set
    volání s jedním příkazem (-c) 73
    vypnutí expanzních znaků (-f) 196

shell *(pokračování)*
příkazy (interní)
viz interní příkazy (shell)
režimy práce 337
typ **133**
volání viz sh
výběr shellu (#!) 198
shellskript **60**, *73*
interakce 185, 187
jméno ($0) 149
ladění 172
parametry viz poziční parametry
provádění v aktuálním prostředí 199
přístupová práva 60
spuštění 60, 149
ukončení viz exit
vložené spuštění 199
vytvoření (echo) 60, 149
vytvoření (sed) 132
shift **150**, *171*, *182*, *189*
návratová hodnota 173
parametr $# 171
parametr $0 151
s parametrem **150**, *173*
schéma (URI) 262
signály 191, 345
Single UNIX Specification (SUS)
viz POSIX
skript (awk) viz awk:skript
skript (ed) viz ed:skript
skript (sed) viz sed:skript
skript (shell) viz shellskript
skupina
číslo **45**, *78*, *81*, *201*, *228*, **349**
členství **50**, *78*, *97*, *100*, *143*, *220*, *223*, *225*, *228*, *245*, **349**
jméno **45**, *80*, *201*, **349**
primární 24, *203*
seznam viz /etc/group
seznam členů **45**
sleep 285
slepování řádek viz spojování řádek
sloupce viz pole
slova
hledání celých slov 50, 97, 161
na příkazové řádce
viz příkazová řádka:pole
na vstupní řádce
viz pole vstupní řádky
počítání viz wc
složené závorky viz {...}
složka viz adresář
slučování bílých znaků 139
slučování přepínačů
viz přepínače:slučování
slučování znaků viz tr
sort **74**
abecední třídění 74
efektivita 74, 122
lexikografické třídění 74
mazání duplicit (-u) 78
numerické třídění (-n) 74
obrácení pořadí třídění (-r) 74
různé oddělovače polí 76, 87
slití souborů 204
třídění podle pomocného sloupce 77
třídění podle polí 75, 76, *87*, *206*
unikátní třídění (-u) 78, 79, *143*
zadání oddělovače polí (-t) 75, 76
zadání třídicího klíče (-k) 75, 76, *206*
soubor
binární **24**, *59*, *207*
čtení (shell) 164
datum a čas
jako vyhledávací kritérium 70
poslední modifikace 29, 70, 332
posledního přístupu 320, 332
testování 147
výpis 29, 352
změna 29

délka viz wc
dočasný **28**
  jako mezivýsledek *28*, *35*, *143*, *145*
  mazání 35, 191, *197*
  předávání výsledků 191
  volba jména 28, 191, 276
editace viz ed
hledání rozdílů 79
hledání řetězců viz grep
hledání ve stromě viz find
jméno
  case senzitivita 40
  další jméno viz hardlink
  implementace 62
  metaznaky ve jméně 31, 40
  povolené znaky 39
  přípona 41
  výpis 352
  změna viz mv
  znak - na začátku 32
  znak . na začátku 36
  znak / ve jméně 39
mód viz přístupová práva
obyčejný **22**, *23*, *38*, *57*, *62*
počet linků 64, 352
počet řádek viz wc
pomocný viz soubor:dočasný
porovnání obsahu 79
prázdný 29, 33, *68*
přejmenování viz mv
přepsání obsahu viz > (shell)
přesun viz mv
příkazový viz skript
připsání na konec viz >> (shell)
přiřazení deskriptoru 206
přístupová práva 57
  nastavení 58
  testování 158
  výpis 352
  regulární viz soubor:obyčejný
  rozdělení na menší 49
  rozpoznání typu obsahu 73
  skrytý **36**, *211*
  skupinový vlastník 352
  smazání 27, 57
  současné čtení a zápis 28, 87, *116*, 276, 289
  spustitelný **57**, *59*, *60*, *72*, *274*
  testování existence 23, 158, *159*, *174*
  testování typu 158
  testování vlastností 158
  textový **24**
  typ **22**, 38, 70, 158, *193*, *216*, 352
  velikost 70, *84*, 158, 191, *216*, 352
  vlastník 70, *201*, 352
  vlastnosti
    hledání 70
    testování 158
    výpis 22, 352
  vymazání obsahu 27, 29, 33, 68
  výpis obsahu 26, 207
  vytvoření 26, 60
souborový systém
  architektura 62
  hierarchická struktura 21
  kapacita viz df
  logický 62
  seznam viz df
současné čtení a zápis souboru 28, 87, *116*, 276, 289
současný přístup 283
space (třída znaků) 347
split (utilita) **49**
split() (awk) **228**, **256**, *272*
spojení tabulek viz join
spojení výstupních proudů (>&) 142
spojení výstupu příkazů 143

spojování podmínek 158, 181
spojování přepínačů
viz přepínače:slučování
spojování řádek (paste) 43, 49
spojování řádek (sed) 110
spojování řádek (tr) 45, *282*
spojování souborů (cat) 26
sprintf() (awk) **273**
spuštění příkazu na každý soubor
viz find:-exec
spuštění příkazu na pozadí 338
SQL (Structured Query Language) 43
srozumitelnost kódu 76, 99, 103, 139, 169, 172, 190, 198, 210, 224, 227, 230, 242, 246
standard viz norma
standardní chybový výstup **37**
/dev/stderr 180
deskriptor 37
přesměrování 37
do /dev/null 37, *142*
spolu s výstupem 142
výpis do 156
standardní vstup **25**
/dev/stdin 180
deskriptor 37
duplikace vstupního proudu 187
editace (sed) 87
jako implicitní zdroj dat 49
jako součást kódu skriptu 152
kopírování do souboru 285
kopírování na výstup 144
přesměrování 24
převzetí od jiného programu 34
roura 34
vs. parametry 82
z terminálu 186
zadání v parametrech viz - (parametr)
zpracování více příkazy 144
standardní výstup **25**, **37**
/dev/stdout 180
deskriptor 37
předání jinému programu 34
přesměrování 26
do /dev/null 37, *142*
do chybového výstupu 156
spolu s chybovým výstupem 142
příkazu volaného z awk 252
roura 34
uložení do proměnné 138
stáří souboru *82*
stavová řádka (HTTP) 278
stderr (pseudosoubor) viz /dev/stderr
stdin (pseudosoubor) viz /dev/stdin
stdout (pseudosoubor) viz /dev/stdout
stream viz proud dat
strom adresářů 21
prohledávání viz find
procházení viz procházení stromu
středník viz ;
stříška viz ^
sub() (awk) **236**, *272*
SUBSEP (awk) 230
subshell 145
čtení z roury 145, *154*
dědění funkcí 199
na pravé straně roury 145, *154*, 186
předávání proměnných 196
předávání výsledků 190, 191
spuštění příkazu viz (...) (shell)
vs. funkce 191
substituce
aritmetická 167
jmen souborů
viz expanze jmen souborů
parametrů viz substituce:proměnných
podmíněná viz ${*jméno*:-slovo}
podřetězců 293

proměnných **133**, *134*, *138*
jako jedno slovo 140, 158, 166
pořadí při zpracování řádky 178
v alfanumerickém řetězci 153
příkazu **138**, *140*
pořadí při zpracování řádky 178
zdvojení zpětných lomítek 170
výrazu 167
substr() (awk) **220**, *237*, *247*, *272*
SUS (Single UNIX Specification)
viz POSIX
svislítko viz |
symlink **62**, *68*, *168*
kopírování 65
neplatný odkaz 67
přesun 67
relativní vs. absolutní odkaz 67
testování 158, *168*, *169*
výpis odkazu 352
vytvoření viz ln
zacyklení 64, 168
systém souborů viz souborový systém
System V 253
system() (awk) **251**, *252*, *272*
systémové proměnné
viz proměnné prostředí

T

TABLE (HTML) 127
tabulátor
indentace 153
jako oddělovač polí 43, 44
na příkazové řádce 141
zápis viz escape-sekvence
tac 121
tail **28**, **36**, *42*, *77*, *173*, 209
tar **66**
TD (HTML) 127
tečka viz .
tee **48**
TERM (proměnná) 337
TERM (signál) 193
terminál **25**, *26*, *34*, *37*, *138*, *185*, *200*
přesměrování vstupu 186
pseudosoubor /dev/tty 180
typ 337
test (shell) **156**
logické výrazy **158**, *159*, *192*, *245*
negace podmínky **158**, *159*, *182*
test čísel **158**, *159*, *166*, *171*, *174*, *182*, *197*, *204*, *264*
test prázdného řetězce **158**, *176*, *181*, *197*, *264*
test rovnosti řetězců **158**, *164*, *171*, *188*, *192*, *205*, *244*
test shody se vzorem **160**
test více řetězců najednou 181
test vlastností souborů **158**, *159*, *169*, *174*, *182*, *188*, *245*, *264*
testování proměnných 158, 166
testování čísel (awk) 219
testování čísel (expr) 167
testování čísel (test) 158, 282
testování podmínek viz test
testování podřetězce 38, 161, 210, 247
testování prvního znaku 160, 210
testování shody řetězce se vzorem 160, 161, 167, 223
textový preprocesor 133, *137*, *161*, *171*, *184*, *225*
textový soubor 24
then viz if
tilda viz ~
time 257
tmp (adresář) viz /tmp
touch **29**, *31*, *33*, *166*
převzetí času z jiného souboru (-r) 29
zadání konkrétního času (-t) 30

tr **43**, *76*, *140*
doplněk znakové sady (-c) **95**
escape-sekvence **45**, *47*, *208*, **320**
interval znaků **44**, *184*
mazání znaků (-d) **282**
opakování znaků v sadě **47**, *95*, *282*
rozdělení řádek 46, *49*, *78*, *95*, *143*, *282*
slučování znaků (-s) **46**, **52**, *78*, *95*
speciální znaky 45
spojování řádek 45, *282*
třídy znaků 44
znak konce řádky 45
TR (HTML) 127
trap **192**, *197*, *281*, 286, **345**
true 198
třídění
jmen souborů při expanzi 33
řádek viz sort
unikátní viz uniq
výstupu ls 82
třídicí klíč
jako nový sloupec 77, 92, *206*
vícesložkový 76
zadání 75
třídy znaků **44**, *91*
tty (pseudosoubor) viz /dev/tty
Turingův stroj 297
typ proměnné (awk) 217
typ proměnné (shell) 137

U

-u (shell) 337
UID **24**, *49*, 57, *70*, *74*, *77*, *79*, *90*, *94*, *104*, *164*, *201*, *218*, *231*, *232*
ukazatel 177
ukončení cyklu viz break
ukončení editoru viz sed:quit
ukončení funkce viz return
ukončení provádění těla cyklu viz continue
ukončení skriptu viz exit
uložení podřetězce do paměti (regexpy) 92
uložení výstupu do proměnné 138
umask 61
uname 254
uniq **78**, 79, *84*
unset (příkaz) 345
until 286
upper (třída znaků) 44, 347
úprava souboru 28, 87
URI (Uniform Resource Identifier) *21*, 261, 353
UTC (Coordinated Universal Time) 307
utilita 1, 24, 35
uvozovky viz "..."
uzavření deskriptoru 187
uživatel
číslo **24**, *77*, *79*, *90*, *94*, *104*, *106*, *201*, *231*, *232*, **349**
členství ve skupině viz skupina:členství
domovský adresář **24**, 84, **349**
heslo 1, **24**, **349**
jméno 1, **24**, *201*, *232*, **349**
login jméno viz uživatel:jméno
login shell **24**, **349**
plné jméno **24**, *76*, *87*, *92*, **349**
primární skupina **24**, 50, 80, *94*, *104*, *118*, *249*, **349**
přihlašovací jméno viz uživatel:jméno
příjmení viz uživatel:plné jméno
seznam viz /etc/passwd
výpis informací viz id
výpis seznamu přihlášených viz who
uživatelská maska 61

V

-v (shell) 172
věčný cyklus 198
verbose režim 337
vestavěné příkazy (shell)
    viz interní příkazy (shell)
většítko viz >
vkládání konců řádek (sed) 101, *115*
vkládání řádek *29*, 105, 106
vlnka viz ~
vložené volání skriptu 199
vložený hlavičkový soubor 255
vstup příkazu viz standardní vstup
výběr interpretu viz #!
výběr polí viz cut
výběr sloupců viz cut
vyhledávání
  řádek viz řádky:vyhledávání
  souborů viz find
vyhodnocení výrazu
  (expr) 166
  (shell) 167
  podmíněné 71, 155, 158, 219
vykřičník viz !
výpis
  aktuálního adresáře viz pwd
  aktuálního data a času viz date
  bez odřádkování *187*
  duplicitních řádek viz uniq
  formátovaného textu viz printf
  informace o systémech souborů
      viz df
  informace o systému viz uname
  informace o uživateli viz id
  informací o souborech viz ls
  intervalu řádek *42*, *89*
  jména počítače viz hostname
  konce souboru viz tail
  obsahu adresáře viz ls
  obsahu binárního souboru viz od
  přihlášených uživatelů viz who
  přiřazených proměnných viz set
  rozdílů mezi soubory viz diff
  rozsahu řádek *42*, *89*
  seznamu procesů viz ps
  souboru viz cat
  textu bez odřádkování 68
  textu do tabulky 348
  textu na terminál 26
  typu obsahu souboru viz file
  unikátních řádek viz uniq
  začátku souboru viz head
výpočet výrazu 166, 167, *195*
výraz
  (awk) viz awk:výraz
  aritmetický viz aritmetika
  regulární viz regulární výraz
  testování viz test
  vyhodnocování
      viz vyhodnocení výrazu
výstup příkazu viz standardní výstup
vytvoření adresáře viz mkdir
vytvoření souboru 26, 60
vývojové diagramy 243
vzor
  porovnávací viz porovnávací vzory
  vyhledávací viz řádky:vyhledávání

W

-w (test) **158**, *159*
wc **24**, *34*, *148*, *159*
webová adresa 21, 261
webová stránka 21, 261, *275*
webový server 261, *283*
while (awk) **242**
while (shell) **164**, *166*, *169*, *171*, *173*, *182*, *189*, *264*
who *53*, *84*, **322**, **352**

wildcard 31
Windows-1250 74
write (právo) 57
WWW 21, 261, 283

X

-x (shell) 172
-x (test) **158**
xargs **82**, *132*, *139*
xdigit (třída znaků) 347
xtrace režim 337

Z

-z (test) **158**, *197*
začátek řádky
  připsání před 99
  v porovnávacích vzorech 41
  v regulárních výrazech 41
začátek souboru viz head
zadávání příkazů 22
zachycení signálu 191
zámek 284
záměna znaků
  (sed) viz sed:konverze znaků
  (tr) viz tr
zápisové přístupové právo 57
zarážka
  detekce konce souboru při čtení 206
  oddělení vstupů awk 234, 246
  pro úpravy řádky 113
  ukončení here-dokumentu 152
  vymezení bloku řádek 124
zarovnání výpisu 348
zásobník 212
zástupce 64
závorky hranaté viz [...]
závorky kulaté viz (...)
závorky složené viz {...}
záznam
  (awk) 239
  v textovém souboru 43
  víceřádkový *111*, *119*, 241
zbytek po dělení viz %
změna adresáře viz cd
změna jména souboru viz mv
značka (HTML) 107, 127, 239, 261
znak konce řádky
  jako zarážka 113
  LF (line feed) **24**, *210*, *282*
  metaznak (shell) 135, 136
  odstranění viz spojování řádek
  přidání viz rozdělení řádek
  v HTML 239
  v substituci příkazu 139
  ve výstupu (awk) 215, 240
  ve výstupu (echo) 26, 33, 68
  zápis v parametru (date) 27
  zápis v parametru (tr) 45
  zápis v programu (awk) 233
  zápis ve formátu (printf) 196
znaky
  konverze (sed)
    viz sed:konverze znaků
  konverze (tr) viz tr
  libovolné (porovnávací vzory)
    viz ? (porovnávací vzory)
  libovolné (regexpy) viz . (regexpy)
  mazání (cut) viz cut
  mazání (tr) viz tr
  počítání (awk) viz length()
  počítání (shell) viz ${*#jméno*}
  počítání (wc) 321
  se zvláštním významem
    viz metaznaky
  výběr podle pozice (cut) viz cut
  výběr ze seznamu viz [...]

zobrazení
data a času viz date
nápovědy viz man
obsahu adresáře viz ls
obsahu souboru viz cat
obsluhy signálů 345
platných přepínačů 344
proměnných 135, 137
přihlášených uživatelů viz who
zpětné lomítko viz \
zpětné reference 92, 94, 295
zpětný apostrof viz `...`
zrušení viz mazání
zrušení významu metaznaků (regexpy)
viz regulární výraz:quoting
zrušení významu metaznaků (shell)
viz quoting
zřetězení řetězců (shell) 134

Libor Forst

Shell v příkladech

aneb

ABY VÁŠ UNIX SKVĚLE SHELL

Vydal
MATFYZPRESS
vydavatelství
Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 346. publikaci

Obálku navrhl Petr Kubát

Z předloh připravených v MS Word
vytisklo Reprostředisko UK MFF
Sokolovská 83, 186 75 Praha 8

První vydání

Praha 2010

ISBN 978-80-7378-152-1