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.