\d{0,5}

Regulární výrazy

Pavel Satrapa

Vytvořeno pro on-line magazín root
www.root.cz

Část 1: Znaky

Unix bez regulárních výrazů je jako sex bez partnera/partnerky. Dá se to používat, ale člověk o cosi zásadního přichází. Znalost regulárních výrazů vám dá do rukou mimořádně silný nástroj pro práci s textem. Jejich prostřednictvím můžete:

- vytahovat z textových dat údaje, které vás zajímají
- přetvářet je do podoby, kterou potřebujete
- vyhledávat a nahrazovat v textových editorech a dalších programech

Zkrátka regulární výraz je univerzální pomocník při práci s textem. Používá jej celá řada programů v Unixu. Umožňují prohledávat soubory (grep, egrep), editovat je (sed, vi), analyzovat a vypočítávát zajímavé údaje (awk) či nabízejí plnohodnotný programovací jazyk, kde si můžete dělat, co vás napadne (Perl, Tk). Ale nebudeme se dlouho zdržovat propagandou a vzhůru do díla.

Jednoduché výrazy

Nejjednodušším regulárním výrazem je obyčejné písmeno – třeba **r**. Když se v textu hledá řetězec, který by tomuto regulárnímu výrazu vyhověl, hledá se jednoduše písmeno „r". Implicitně se (jak bývá v Unixu zvykem) rozlišují malá a velká písmena. Ve většině nástrojů však můžete tuto vlastnost vypnout.

Jelikož i v těch nejjednodušších případech člověk zpravidla hledá slovo a ne jediné písmeno, lze regulární výrazy řetězit. Použijete-li regulární výraz **root**, představuje vlastně zřetězení čtyř elementárních jednopísmenných regulárních výrazů. Výsledkem je chování, které byste očekávali – v textu se bude hledat slovo „root".

Vyhledávání jednoduchých slov je tou nejprimitivnější, ale zároveň nejčastější aplikací regulárních výrazů.

> **Příklad:**
>
> Řekněme, že hledáte nejčerstvější soubory v aktuálním adresáři. Nevím jak vy, ale já v hlavě nenosím, jak se jmenuje volba příkazu `ls`, která zajistí uspořádání podle času. Takže zadám `man ls` a následně si pomocí /time nechám vyhledat první výskyt slova „time". Nebudu-li spokojen, stisknu klávesu n a poskočím tak na další výskyt.

Popsané hledání založené na regulárních výrazech dovedou oba programy používané obvykle pro zobrazování manuálových stránek (a řady dalších textů): `more` i `less`. Tyto programy zároveň ilustrují jeden obecný princip: regulární výraz se typicky vyhledává jako podřetězec v jednotlivých řádcích textu.

Libovolný znak

Poměrně často dochází k situacím, kdy vám na určité části hledaného řetězce nezáleží. Například chcete ve zdrojovém textu HTML stránky vyhledávat začátky buněk v tabulkách – čili značky `<TD>` a `<TH>`. Až na třetí znak jsou oba řetězce shodné, takže je lze vyhledávat jediným regulárním výrazem. Pouze je třeba říci, že na jeho třetím znaku nezáleží.

Tuto činnost obstará znak tečka (**.**). Při hledání jí vyhoví libovolný znak kromě konce řádku. Nelze ji však ignorovat – nějaký znak jí program vždy musí přiřadit.

> **Příklad:**
>
> Výše zmíněné hledání řetězců „<TD>“ či „<TH>“ obstará regulární výraz **<T.>**. Přesněji řečeno mu vyhoví libovolný čtyřznakový řetězec, který začíná „<T“ a končí znakem „>“.

Ne až tak libovolný znak

Použitím tečky zcela rezignujete na hodnotu příslušného znaku. V některých případech se to hodí, jindy byste však potřebovali výběr omezit přísněji. Pak můžete sáhnout po hranatých závorkách.

Zapíšete-li do hranatých závorek skupinu znaků, bude tomuto regulárnímu výrazu vyhovovat právě jeden z těchto znaků. Například výrazu **[xyz]** vyhoví buď znak „x“ nebo „y“ nebo „z“. Jestliže povolené znaky tvoří interval, můžete si ušetřit práci a v hranatých závorkách uvést pouze jeho meze, které spojíte pomlčkou.

> **Příklad:**
>
> Pro vyhledání libovolné číslice poslouží regulární výraz **[0–9]**. Předchozí hledání <TD> a <TH> bylo příliš benevolentní, protože za T povolovalo libovolný znak. Lepší je regulární výraz **<T[DH]>**, který se skutečně omezí jen na uvedené dvě značky.

Jednotlivých znaků a jejich intervalů můžete do hranatých závorek napsat, co hrdlo ráčí. Například výrazu **[a0–9zl–nt]** vyhoví libovolné z písmen a, l, m, n, t, z nebo libovolná číslice.

Kromě pomlčky se v hranatých závorkách vyskytuje ještě jeden speciální znak. Pokud hned za otevírací hranatou závorkou zapíšete stříšku (**^**), bude celá skupina negována. To znamená, že regulárnímu výrazu vyhoví libovolný znak odlišný od těch, které jsou uvedeny ve skupině. Například **[^0–9]** vyhoví cokoli kromě číslice.

Intervaly znaků vycházejí z kódování ASCII. To znamená, že například výrazu **[a–z]** vyhoví libovolné *malé* písmeno *anglické* abecedy. Doplnit velká písmena není žádný velký problém (**[a–zA–Z]**), ale s českými znaky je potíž. V některých programech najdete konstrukce, kterým vyhoví i znaky české abecedy, univerzálně platné elegantní řešení však neexistuje.

Speciální znaky

Možná už vás napadlo „ale co když potřebuji vyhledat tečku?“ Tedy obecněji: jak vyřadit speciální význam některých znaků. Obecná odpověď na tuto otázku zní

„zpětným lomítkem“. V Unixu bývá zvykem, že pokud speciálnímu znaku předřadíte zpětné lomítko, vypnete tak jeho speciální chování (a v některých případech právě naopak, jak uvidíte později).

Příklad:

Celkem pohledný regulární výraz **\.\.\.** hledá tři tečky. Chcete-li vyhledat písmeno uzavřené v hranatých závorkách (tedy cosi jako „[x]“), použijte **\[[a–z]\]**.

Uvnitř hranatých závorek panuje specifické prostředí. Tečka zde představuje obyčejnou tečku a význam ostatních dvou speciálních znaků lze potlačit prostým pořadím. Stříška představuje negaci jen pokud je uvedena na samotném začátku a pomlčka slouží jako oddělovač intervalu jen pokud má z obou stran jeho meze. Takže například výrazu **[.^az–]** vyhoví pouze jeden ze znaků „.“, „^“, „–“, „a“ nebo „z“.

Pokud má být jedním z povolených znaků pravá hranatá závorka, uveďte ji hned za otevírací. Takže například regulárnímu výrazu **[][]** vyhoví levá nebo pravá hranatá závorka. Pokud byste znaky uvnitř vnějších hranatých závorek zapsali v opačném pořadí (**[[]]**), význam by se radikálně změnil: byl by interpretován jako **[[]** bezprostředně následované **]**. Čili vyhověl by mu jedině řetězec „[]“.

Část 2: Základy opakování

V předchozí části jsem popsal základní prvky regulárních výrazů. Konstrukce, které jsem z nich vytvářel, měly jednu společnou nevýhodu: pevně daný počet znaků hledaného řetězce. V dnešní části se budu věnovat mechanismu opakování. Díky němu lze zajistit, že řetězec odpovídající regulárnímu výrazu může mít proměnlivou délku.

Opakování výrazu

Základní konstrukcí pro opakování regulárních výrazů je hvězdička (*****). Znamená, že regulární výraz bezprostředně před ní se může zopakovat, kolikrát to jenom jde.

Příklad:

Výrazu **A*** tedy vyhoví libovolný počet písmen „A“, zatímco **[0–9]*** ztělesňuje libovolně dlouhou posloupnost číslic (opakovaným regulárním výrazem je zde **[0–9]**, tedy libovolná číslice).

V řadě programovacích jazyků je identifikátor definován jako libovolně dlouhá posloupnost písmen a číslic začínající písmenem. Pomocí regulárního výrazu bychom jej zapsali jako **[a–zA–Z][a–zA–Z0–9]***. Zdůrazňuji, že opakování se týká jen regulárního výrazu, který je uveden bezprostředně před hvězdičkou. Uvedený výraz tedy znamená ”právě jeden výskyt **[a–zA–Z]** (písmeno), za nímž následuje libovolný počet výskytů **[a–zA–Z0–9]** (písmeno nebo číslice)”.

Snad nejběžnějším opakovaným výrazem je tečka, která v kombinaci s hvězdičkou (**.***) znamená „libovolný řetězec znaků“. V souvislosti s opakováním si dobře zapamatujte tři důležité skutečnosti:

- do libovolného počtu opakování se počítá i nula
- opakování se týká regulárního výrazu, nikoli řetězce, který je s ním porovnáván
- opakování je hladové – snaží se „pozřít“ co nejvíc znaků

Přípustnost nulového počtu opakování znamená, že opakovanému regulárnímu výrazu vždy může vyhovět i prázdný řetězec. Praktickým důsledkem je, že jen vzácně dává smysl vyhledávat samotný opakovaný výraz. Zpravidla je třeba jej alespoň z jedné strany ohraničit něčím povinným.

> **Příklad:**
>
> Chcete-li vyhledat v textu všechna čísla, nemá smysl hledat "libovolně dlouhou posloupnost číslic" (**[0–9]***), protože posloupnost číslic nulové délky obsahuje každý řádek (vyzkoušejte grep '[0–9]*' soubor na libovolný soubor – uvidíte, že „najde“ všechny jeho řádky). Správně je třeba hledat „alespoň jednu číslici“, tedy použít regulární výraz **[0–9][0–9]***.

Skutečnost, že opakování se týká regulárního výrazu, nikoli srovnávaného řetězce, je velmi důležitá. Zapíšete-li **.***, spojujete dva prvky: symbol pro libovolný znak a symbol opakování. Výslednou konstrukci lze chápat dvěma způsoby. Buď jako libovolný počet libovolných znaků (opakování regulárního výrazu) nebo že v textu může být libovolný znak a ten se pak může opakovat, kolikrát chce (opakování ve zkoumaném řetězci). Správný je první výklad, jinak bychom se z toho nejspíš zbláznili.

Hladovost opakování se projevuje tím, že opakovaný regulární výraz se vždy snaží roztáhnout na co největší délku – zahrnout do sebe co největší počet znaků zkoumaného řetězce. Proto když například řetězec „brambora“ srovnáte s regulárním výrazem **r.*a** (libovolný řetězec znaků začínající „r“ a končící „a“), bude vyhovujícím řetězcem „rambora“ (od prvního „r“ až po poslední „a“).

Regulární výrazy versus žolíkové znaky

Začátečníci někdy zaměňují regulární výrazy s žolíkovými znaky. Jistá podobnost tu skutečně je – oba prostředky umožňují vytvářet jakési vzory, které jsou porovnávány se skutečnými daty. Existují mezi nimi dva zásadní rozdíly. Žolíkové znaky se týkají názvů souborů a zpracovává je interpret příkazů (shell). Naproti tomu regulární výrazy se zaobírají obsahem (textových) souborů a jejich interpretaci mají na starosti jednotlivé programy (editory, grep a podobně).

Případným omylům ještě nahrává podobnost některých speciálních znaků mezi oběma konstrukcemi. V tomto směru je záhodno především mít na paměti, že zatímco v žolíkových znacích ***** představuje libovolný řetězec, v regulárních výrazech se libovolný řetězec zapisuje pomocí **.***.

Mechanika srovnávání

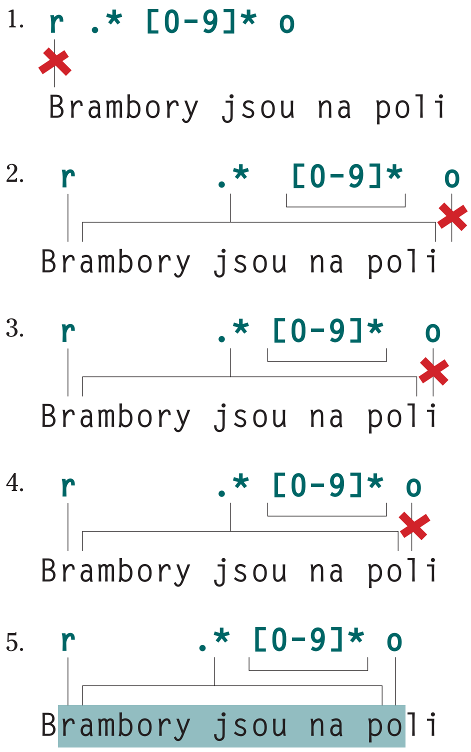
Pojmem srovnávání (matching) se označuje proces, kdy program hledá, zda předložený řetězec znaků odpovídá regulárnímu výrazu či nikoli. Zároveň se program snaží stanovit, které části řetězce odpovídají jednotlivým částem regulárního výrazu.

Dokud se zabýváte pouze hledáním, je pro vás v podstatě nezajímavé, kde *přesně* byl daný vzor nalezen. Ovšem když používáte regulární výrazy k nahrazování (a právě v tom je jejich největší síla), je tato informace velmi důležitá.

Základní princip srovnávání je následující: začne se od začátku řetězce. Každému prvku regulárního výrazu se snaží přiřadit vždy co nejdelší posloupnost znaků ze zkoumaného textu a teprve pak pokračuje srovnáváním dalších částí. Pokud to později nevyjde, vrátí se zpět a zkusí přidělený řetězec o jeden znak zkrátit. Jestliže již zkrátil na minimum vše, co zkrátit šlo, a přesto se nepodařilo najít shodu, posune se na další znak zkoumaného textu a vše se rozjede znovu.

Příklad:

Podívejme se, jak by vypadalo srovnávání regulárního výrazu **r.*[0–9]*o** s řetězcem „Brambory jsou na poli“. Při hledání by postupoval takto:



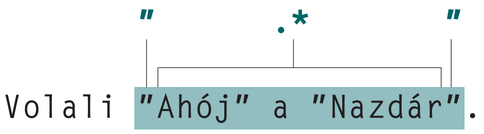
Z příkladu je vidět, že díky své hladovosti regulární výraz zabere co největší část řetězce – od prvního „r“ až po poslední „o“. Výrazu **.*** je nakonec přiřazen řetězec „ambory jsou na p“. Výrazu **[0–9]*** srovnávání přidělí prázdný řetězec. Zdánlivě proto, že zkoumaný text neobsahuje žádné číslice. Ve skutečnosti však řetězec odpovídající tomuto výrazu zůstane prázdný vždy, protože je umístěn nesmyslně. Předchází mu totiž *obecnější* regulární výraz s opakováním. Jakýkoli znak, který by mohl vyhovovat **[0–9]*** vyhovuje také předchozímu **.***, které jej díky své hladovosti sežere a na **[0–9]*** nezbyde nic.

Ze stejného důvodu když libovolný řetězec znaků srovnáte s regulárním výrazem **.*.***, bude prvnímu **.*** přiřazen celý zkoumaný řetězec a druhému **.*** prázdný řetězec.

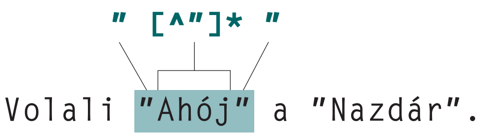
Příliš hladové opakování

Na hladovost opakování je třeba si dávat pozor. Díky ní se snadno může stát, že opakující se výraz pohltí i ty znaky, se kterými jste nepočítali. Klasickým příkladem tohoto chování je regulární výraz pro řetězec znaků v uvozovkách.

Začátečníci mají často tendenci uvažovat následovně: řetězec v uvozovkách, to jsou otevírací uvozovky, pak cokoli a na konci druhé uvozovky. To vyjádříme regulárním výrazem **".*"**. Problém je, že díky hladovosti hvězdičky se tento regulární výraz roztáhne od prvních uvozovek ve zkoumaném řetězci až po poslední. Takže například když jej vypustíte na řetězec 'Volali "Ahój" a "Nazdár".', dopadne to takto:



Řešením je nepřipouštět v uzavřeném řetězci libovolné znaky, ale pouze znaky jiné než koncový. V našem případě cokoli kromě uvozovek, takže tím správným regulárním výrazem bude **"[^"]*"**:



Mimochodem – kdybyste na řetězec aplikovali výraz **.*".*"**, roztáhne se první **.*** na „Volali "Ahój" a " a druhé **.*** na „Nazdár". Jelikož jsou v regulárním výrazu povinné uvozovky, nemůže první **.*** schramstnout všechno. Ukousne si však co nejvíc, čímž druhé **.*** omezí na minimum.

Část 3: Nástroje

První dvě části našeho krátkého seriálu se zabývaly těmi nejzákladnějšími prvky regulárních výrazů. Přestože jsem například v oblasti opakování ještě zůstal dost dlužen, rozhodl jsem se udělat malou přestávku a věnovat se programům, které regulární výrazy používají. Dnes se tedy pokusím popsat, co a jak se s nimi dá provádět.

grep a spol.

Rodina programů *grep slouží k vyhledávání v souborech. Typické použití: hledáte určitý identifikátor v haldě zdrojových kódů nebo chcete zjistit, odkud se spouští určitý program. Spuštění je prosté:

grep *vzor* *seznam_souborů*

Vzorem je regulární výraz. Výstup programu tvoří řádky, které vyhovují zadanému vzoru (což nejčastěji znamená, že obsahují zadané slovo). Pokud program zkoumá více než jeden soubor, vypíše zároveň před každý řádek název souboru, ze kterého pochází. Prostřednictvím voleb lze ovlivnit jeho chování. Těmi nejběžnějšími jsou:

–i	nerozlišovat malá písmena od velkých
–w	vybírat jen řádky, na nichž vzoru vyhovuje celé slovo
–v	negovat výsledek (vypisovat řádky, které *nevyhovují* vzoru)
–l	vypisovat jen jména souborů
–r	rekurzivně procházet adresáře (umí jen některé verze grepu)

grep je představitelem celé rodinky programů. Mají podobná jména i funkci, liší se jen v detailech. Jejími standardními členy jsou tyto tři programy:

grep	klasický grep, vzorem může být obyčejný regulární výraz
egrep	vzorem je rozšířený regulární výraz (viz příště), používá rychlejší vyhledávací algoritmus
fgrep	vzorem je jen obyčejný řetězec znaků; teoreticky nejrychlejší, ale praktická měření ukazují opak; zapomeňte na něj

Kromě nich existují ještě některé další pozoruhodné alternativy. Asi nejzajímavější je agrep (approximate grep) vyhledávající řetězce, které se zadanému vzoru pouze podobají. Najde například nejpodobnější nebo všechny takové, které se od vzoru liší jen v daném počtu znaků.

Vřele doporučuji používat grep a spol. pocházející z GNU projektu. Ve srovnání s klasickými implementacemi je rychlejší (používá lepší algoritmy) a navíc umí některé příjemnosti (třeba rekurzivní hledání). Zejména komerční verze Unixu však mají tendenci trvat na originálních verzích. Proto vyzkoušejte

grep ––version

Dostanete-li chybové hlášení nebo se ohlásí někdo jiný než GNU grep, máte co instalovat.

sed

Program sed je neinteraktivní editor. Zadáte mu sadu příkazů a on podle nich zpracuje vstupní text. De facto se jedná o nástroj pro vytváření editačních filtrů.

Regulární výrazy se v sedu vyskytují hned ve dvojí roli. Lze je použít k vyhledání řádků, na které se má vztahovat určitý příkaz. Druhým místem výskytu regulárních výrazů je příkaz pro nahrazování, který má tvar **s/vzor/náhrada/**. Vyhledává regulárním výrazem zadaný vzor a pokud jej najde, vloží na jeho místo náhradu. Za závěrečné lomítko můžete připojit ještě volby. Tou nejpoužívanější je **g** (global), která zajistí nahrazení všech výskytů vzoru na řádku. Standardně se totiž nahrazuje jen první.

Příkazy sedu mají obecně tvar

```
řádky příkaz
```

Počáteční definice řádků určuje, na které řádky vstupního textu se příkaz bude vztahovat (chybí-li, znamená to všechny řádky).

> **Příklad:**
>
> Řekněme, že jste změnili doménu z *kdesi.cz* na *jinde.cz*. Navíc chcete ze svých WWW stránek odstranit pracovní texty – tedy veškeré úseky začínající `<DIV CLASS="pracovni">` a končící `</DIV>`. Zajistí to následující dvojice příkazů:
>
> ```
> s/kdesi\.cz/jinde\.cz/g/<DIV CLASS="pracovni">/,/<\/DIV>/d
> ```
>
> První je klasické nahrazení, které se bez určení řádků vztahuje na celý vstupní text. Druhým příkazem je **d** (delete), kterému podlehnou všechny skupiny řádků od řádku vyhovujícího prvnímu regulárnímu výrazu až po nejbližší následující, který vyhovuje druhému. Všimněte si, že lomítko v </DIV> je třeba chránit zpětným lomítkem, protože v příkazu **s** má speciální funkci oddělovače. Tuto dvojici uložíte řekněme do souboru *zmena* a každou stránku pak podrobíte příkazu
>
> ```
> sed –f zmena
> ```
>
> Příklad tiše předpokládá, že své HTML stránky píšete stejně jako já – tedy že značky <DIV> a </DIV> jsou na samostatném řádku. Pokud ne, vezme při mazání za své celý řádek, který některou z nich obsahuje.

vi

Unixové textové editory, jejichž je *vi* nejklasičtějším představitelem, typicky používají regulární výrazy k vyhledávání textu a také k jeho nahrazování.

Konkrétně v případě *vi* zahajujete hledání stisknutím klávesy **/**, případně **?** (pokud chcete hledat směrem k začátku dokumentu). Následně napište regulární výraz a stiskněte [Enter]. Kurzor poskočí na nejbližší následující/předchozí řetězec vyhovující zadanému výrazu. Chcete-li se přesunout na další, stiskněte **n** (next) pokud chcete hledat stejným směrem či **N** pro hledání směrem opačným. Čili úplně stejně jako v programech more a less.

V kombinaci s tečkou (opakování poslední editační operace) tvoří **n** pořádně silnou dvojku. Pomocí **n** si poskakujete na další a další výskyty hledaného řetězce a tu a tam na některý zopakujete editační operaci stisknutím tečky.

Příkaz pro nahrazení má stejnou podobu, jako u editoru sed. Jediným rozdílem je, že pokud neuvedete řádky, implicitně se týká pouze aktuálního řádku. Chcete-li nahradit vzor v celém textu, použijte jako definici řádků znak **%**. Nahrazování patří do příkazového režimu, takže celý příkaz musíte zahájit dvojtečkou.

Příklad:

Výše zmiňovanou náhradu *kdesi.cz* na *jinde.cz* by ve *vi* zajistil příkaz

:%s/kdesi\.cz/jinde\.cz/g

Některé verze *vi* určené pro grafické prostředí (například *gvim*) nabízejí i uživatelsky přítulnou verzi tohoto příkazu – viz obrázek. Ortodoxní uživatel o ni samozřejmě nezavadí ani pohledem, protože

1. Musí-li pracovat v textovém režimu nebo na jiném počítači, nebude tato lahůdka k dispozici.
2. **:s** je rychlejší a u novějších implementací *vi* máte navíc historii příkazů.
3. Jen při používání **:s** vypadáte jako *skutečný borec*.

Pokud používáte *vim*, můžete k vyznačení řádků s výhodou využít vizuální režim. Stiskněte klávesu **v** a zvýrazněte požadované rozmezí řádků. Když potom stisknete **:**, editor automaticky vloží podivné rozmezí řádků '<,'>, což znamená od prvního vyznačeného řádku po poslední. Pozor, nahrazení se týká celých řádků, ne jen vyznačeného textu v nich.

Ostatní editory nabízejí zpravidla podobné služby, liší se jen příkazy, kterými je vyvoláváte.

awk

Program awk realizuje specializovaný programovací jazyk, který umožňuje zpracovávat textové soubory s pevnou strukturou (např. */etc/passwd*, výstup příkazu *ls -l* a podobně).

Regulární výrazy opět vystupují ve dvou známých rolích: určují řádky, kterých se operace týká, a vyskytují se v nahrazování.

Základním problémem awku je jeho jednostrannost. Syntaktický tvar programů je velmi nezvyklý a držet jej v hlavě kvůli úzce vymezené skupině úloh, které dokáže řešit, se podle mého soudu nevyplatí.

Perl

Perl je pozoruhodný programovací jazyk, který z hlediska regulárních výrazů nabízí dvě vysoce zajímavé vlastnosti:

1. Má nejbohatší a nejpřehlednější sortiment regulárních výrazů ze všech mně známých nástrojů.
2. Spojuje regulární výrazy s běžnými programátorskými konstrukcemi a la C.

Ve srovnání s Perlem vám sed či awk nemají mnoho co nabídnout. V poslední době pro neinteraktivní úpravy textů nepoužívám nic jiného.

Základní konstrukcí pro uplatnění regulárních výrazů v Perlu je operátor srovnání (`=~`). Jeho levým operandem je řetězec znaků (typicky proměnná). Pravý operand závisí na tom, co s dotyčným řetězcem hodláte provádět. Pokud vám jde o prosté srovnání, zda řetězec vyhovuje regulárnímu výrazu, zapíšete sem obvyklý vzor uzavřený mezi dvě lomítka.

Příklad:

Perl je velmi košatý a lze jej používat mnoha různými způsoby. Začnu s minimalistickou variantou, kdy prováděný „program“ píšete přímo do příkazového řádku. V tomto případě se Perl používá s volbou `-e`, která způsobí, že následující text bude interpretován rovnou jako příkazy jazyka. Zpravidla bývá kombinována s volbou `-n`, díky níž budou příkazy prováděny pro každý řádek vstupního textu. Takže třeba

```
perl -ne 'print if /Pepa/'
```

vypíše ty řádky vstupního textu, které obsahují řetězec „Pepa“. Zároveň zde Perl demonstruje skutečnost, že si dokáže hodně věcí domyslet (chybí zde proměnná i operátor srovnání). Já osobně se však této domýšlivosti raději straním.

Následuje serióznější úryvek z programu. Vypíše hlášení, zda proměnná *$radek* obsahuje podřetězec „Pepa“.

```
if ( $radek =~ /Pepa/ ) {
  print "Je tam!";
} else {
  print "Není tam...";
}
```

Chcete-li použít regulární výraz k nahrazení řetězce jiným, uveďte na pravé straně srovnání obvyklou substituční konstrukci **s/vzor/náhrada/**.

Příklad:

A ještě jeden kondenzát. Následující volání nahradí ve vstupním textu všechna čísla (celá čísla, nikoli jednotlivé číslice!) znakem „X“:

```
perl -pe 's/[0-9][0-9]*/X/g'
```

Použitá volba -p se podobá -n, ale po provedení příkazů navíc pošle aktuální řádek na standardní výstup. Všimněte si apostrofů, kterými jsou obklopeny regulární výrazy a příkazy na příkazovém řádku. Zajišťují, že je interpret příkazů předá v nezměněné podobě volanému programu a nebude se snažit interpetovat je jako své speciální znaky.

V programech se substituce zpravidla používá takto:

```
$radek =~ s/[0-9][0-9]*/X/g;
```

Zde se nahrazuje v obsahu proměnné $radek.

Jak vidíte, použití regulárních výrazů na proměnné obsahující řetězce znaků je v Perlu velmi snadné. Zajímavé služby nabízí také funkce **split(/vzor/,řetězec)**, která řetězec znaků rozdělí na části ohraničené výskyty zadaného regulárního výrazu. Ideální pro zpracování souborů, jako je */etc/passwd*.

Část 4: Pokročilejší opakování a pozice

Po krátké odbočce se vracíme k opakování. Onu přestávku věnovanou programům podporujícím regulární výrazy jsem zařadil jednak abych předvedl nějaké praktické využití, jednak proto, že již nebude možné skrývat existenci různých dialektů. Různé programy totiž podporují lehce odlišné varianty regulárních výrazů. Ty, o kterých jsem mluvil až dosud, byly společné pro všechny. Dnes už narazíme na některé odlišnosti.

Omezený počet opakování

Základním problémem klasické opakovací hvězdičky je, že je nekontrolovatelná. Pro některé situace potřebujete přesnější vyjadřování.

Vaše touhy uspokojí konstrukce \{**min,max**\}. Opět se vztahuje na bezprostředně předcházející regulární výraz a říká, že se má opakovat alespoň *min*-krát, nanejvýš však *max*-krát. Jako každé opakování i tohle je hladové, takže se snaží uplatnit vždy co největší z povoleného počtu opakování.

Příklad:

Regulárnímu výrazu **i**\{**1,3**\} vyhoví řetězce „i“, „ii“ nebo „iii“.

Tvar tohoto opakovátka je velmi variabilní. Pokud chybí horní mez (**\{min,\}**), znamená to, že maximální počet opakování je neomezený. Jestliže v konstrukci použijete jen samotné číslo (**\{počet\}**), musí se regulární výraz opakovat přesně daný počet-krát.

> **Příklad:**
>
> Regulární výraz pro rodné číslo by vypadal takto:
>
> **[0–9]\{6\}/[0–9]\{3,4\}**
>
> Šest číslic, lomítko a ještě tři nebo čtyři číslice.

A již tu máme první odlišnost. V některých verzích regulárních výrazů (například v Perlu) se při omezování počtu opakování před složenými závorkami nepíší zpětná lomítka. Zapíšete-li zde **\{**, znamená to, že zkoumaný text má obsahovat znak „{“. Perl má pro tuto specialitu dobrou omluvu: zavedl pravidlo, že kombinace zpětného lomítka a znaku odlišného od písmena či číslice nikdy nemá speciální význam a vždy představuje dotyčný znak.

Nejpopulárnější opakovačky

Dva velmi populární případy opakování si vysloužily svůj vlastní speciální znak. Prvním je „alespoň jeden výskyt“ – tedy cosi velmi podobného klasické opakovací hvězdičce, až na to, že opakovaný regulární výraz nelze vynechat. Stejného efektu dosáhnete konstrukcí **\{1,\}**, ale to je příliš složité psaní. Proto se alespoň jeden výskyt předchozího regulárního výrazu zapisuje znakem plus (**+**).

Druhou populární situací je nepovinný (čili nanejvýš jeden) výskyt. Opět jej lze zapsat pomocí **\{0,1\}**, ale kratší je otazník (**?**).

Dialekty regulárních výrazů se u této dvojice znaků opět silně rozcházejí. Programy používající klasické regulární výrazy (grep, sed, vi) jim předřazují zpětné lomítko (**\+** a **\?**). Generace, která implementuje rozšířené regulární výrazy, (egrep, awk, Perl) je píše bez něj (**+** a **?**).

> **Příklad:**
>
> Alespoň jedna číslice se tedy v grep, sed a vi vyjádří pomocí **[0–9]\+**, zatímco egrep, awk či Perl nabízí kratší **[0–9]+**. Zápis můžeme lehce rozšířit na regulární výraz pro celé číslo: nepovinné znaménko následované alespoň jednou číslici. V klasických regulárních výrazech to bude **[–+]\?[0–9]\+**, zatímco v rozšířených **[–+]?[0–9]+**.

Pozice

Výrok „dejte mi pevný bod a pohnu zeměkoulí“ jistě znáte. Nahlíženo jeho optikou byly všechny naše dosavadní regulární výrazy poněkud neukotvené. Řetězec, který jim vyhovuje, se mohl vyskytovat kdekoli ve zkoumaném textu. Občas však člověk musí být přísnější.

Proto regulární výrazy nabízejí několik speciálních pozičních znaků. Těmi nejznámějšími jsou stříška (**^**), která ztělesňuje začátek řádku (resp. zkoumaného řetězce znaků), a dolar (**$**) označující jeho konec.

> **Příklad:**
>
> `grep '^#'` vám tedy najde řádky začínající znakem '#', `grep '[0–9]$'` řádky končící číslicí a konečně `grep '^–\+$'` řádky složené pouze z pomlček (nikoli však prázdné).

Dalším významným místem je hranice slova. Ve většině regulárních dialektů máte k dispozici konstrukci `\<`, která označuje začátek slova, a `\>`, které vyhoví pouze jeho konec.

Perl nerozlišuje začátek slova od konce, má pouze speciální znak **\b** pro „hranici slova“ (tedy začátek nebo konec). Ovšem dlužno přiznat, že z okolního kontextu bývá zřejmé, zda **\b** může vyhovět začátek nebo konec slova. Jako cenu útěchy získáváte v Perlu ještě **\B**, kterému vyhoví libovolné místo ve zkoumaném řetězci kromě hranice slova. Jedná se tedy o negaci **\b**.

> **Příklad:**
>
> Zajímají-li vás všechny řádky, na nichž se písmeno 'a' vyskytuje v roli jednopísmenné spojky, nasaďte
>
> ```
> grep '\<a\>'
> ```
>
> Totéž v Perlu (až na to, že se nevypisují jména souborů) by zajistil příkaz
>
> ```
> perl –ne 'print if /\ba\b/'
> ```
>
> Pokud se názvů souborů nehodláte vzdát, použijte
>
> ```
> perl –ne 'print "$ARGV:$_" if /\ba\b/'
> ```

Část 5: Zapamatování

Na pátou část seriálu jsem si pošetřil snad nejsilnější prvek regulárních výrazů – jejich paměť. Regulární výraz si totiž dokáže zapamatovat řetězec, který vyhověl jeho části, a později jej použít. Největší služby tento mechanismus odvede při nahrazování.

Zapamatuj a vzpomeň si

Prostředky pro zapamatování jsou směšně jednoduché. Část, kterou si má regulární výraz podržet v paměti, prostě ohraničíte konstrukcemi **\(** a **\)**. Jistě si vzpomínáte, že v Perlu nemá nealfanumerický znak předcházený závorkou nikdy speciální význam, takže tam se ke stejnému účelu používají obyčejné závorky. Takových úseků můžete v regulárním výrazu mít hned několik. Dokonce je lze i vzájemně vnořovat.

Když později chcete použít zapamatovaný řetězec, napište **\číslo**, kde *číslo* je pořadové číslo zapamatovaného úseku. Pořadová čísla začínají jedničkou a rozhoduje o nich pořadí levé (otevírací) závorky zapamatovávané sekvence.

Příklad:

Podrobíte-li řádek ze souboru */etc/passwd* regulárnímu výrazu **^\([^:]*\):[^:]*:\([^:]*\)**, bude **\1** obsahovat přihlašovací jméno a **\2** jemu odpovídající identifikátor (UID).

O dobrodiní paměti jste ochuzeni u programů egrep a awk. Jejich algoritmus pro srovnávání regulárních výrazů nepodporuje zapamatování (a z principiálních důvodů ani podporovat nemůže). V příští části se k této otázce vrátím podrobněji.

Použití při hledání

Použití zapamatovaných částí při vyhledávání je poměrně vzácné. Řečeno mluvou politiků: společenská potřeba takové služby je celkem malá. Příklady sice existují, nicméně bývají takové školometské. Tak si nějaké zkusíme.

Příklad:

Hezkou a zcela neužitečnou ilustrací zapamatování je hledání palindromů (slov, která se nezmění, když je čtete pozadu). Tak například výrazu

```
\<\(.\)\(.\).\2\1\>
```

vyhoví právě a pouze pětiznakový palindrom (např. radar či rotor). První dva znaky slova si zapamatuje, za nimi následuje třetí libovolný znak, čtvrtý musí být stejný jako zapamatovaný druhý a pátý znak se musí shodovat s prvním.

Příklad:

Zkusím něco, co by alespoň vzdáleně připomínalo reálný život. Řekněme, že máte výstupy z jakéhosi algoritmu – na každém řádku sadu čísel oddělených mezerami. Každý řádek zároveň končí správným výsledkem. Pokud algoritmus pracuje správně, poslední dvě čísla na řádku jsou totožná. Hledáme tedy řádky, ve kterých se poslední číslo liší od předposledního. K řešení poslouží grep s negovanou podmínkou (volba –v):

```
grep –v ' \([^ ]\+\) \+\1$'
```

Vzor začíná mezerou před předposledním číslem. Za ní následuje neprázdná posloupnost nemezerových znaků ([^]\+), která se zapamatuje. Po ní následuje alespoň jedna mezera (\+) a znovu stejná posloupnost, za kterou už je jen konec řádku.

Použití při nahrazování

Daleko častěji se zapamatované řetězce vyskytují v příkazech pro nahrazování. Díky nim si můžete ze vstupních dat vytáhnout informace, které vás zajímají, a poskládat si je do tvaru, který potřebujete.

Příklad:

Běžný problém všedního dne: potřebujete u skupiny souborů změnit příponu z *.htm* na *.html*. Pro podobné účely sice existují různá udělátka, ale je třeba si je doinstalovávat a práce s nimi nebývá úplně snadná. takže se podívejme, jak poslouží standardní nástroje, které najdete v každém Unixu.

Postup je jednoduchý: obstaráte si seznam jmen souborů, každé jméno pak změníte na příkaz mv staré nové a tyto příkazy provedete. Popsaný postup lze realizovat třeba takto:

```
ls *.htm > seznam
sed 's/\(.*\)/mv \1 \1l/' seznam > akce
chmod a+x akce
./akce
rm seznam akce
```

Uznávám, že prosté připojení „l“ na konec jména souboru je dosti snadnou modifikací. Složitější věci však znamenají jen úpravu příkazu **s**. Například změnu přípony z *.doc* na *.txt* by zajistilo **s/\(.*\)\.doc/mv \1.doc \1.txt/** – zapamatuje si jen vlastní jméno souboru a přípony jsou explicitně vyjmenovány.

Příklad:

A teď něco drsnějšího. Chtěl bych ze souboru */etc/passwd* vyrobit seznam domácích stránek uživatelů. Takže potřebuji řádky transformovat z původní podoby

```
uživatel:heslo:UID:GID:vlastní jméno:...
```

na

```
<A HREF="/~uživatel">vlastní jméno</A>
```

Kýženým substitučním příkazem, který to zařídí, je

```
s/\([^:]*\):\([^:]*:\)\{3\}\([^:]*\).*/<A
HREF="/~\1">\3<\/A>/
```

Jak vidíte, prostřednictvím závorek lze předepsat počet opakování i rozsáhlejšímu úseku regulárního výrazu – zde se třikrát opakuje skupina **[^:]*:**. V takovýchto situacích závorky většinou neslouží k zapamatování (i když si pochopitelně něco zapamatují), ale čistě k vymezení opakované části. Ta má – bez ohledu na skutečný počet opakování –

jen jediné pořadové číslo. Proto se závěrečné zapamatované jméno uloží jako **\3**.

Problémem regulárních výrazů je, že jsou velmi kompaktní. Vyznat se ve výše citovaném substitučním příkazu zabere chvíli času (zpravidla více, než jej vymyslet). V Perlu si můžete vytvoření seznamu rozložit: nejprve řádek rozkrájíte v místě výskytu dvojteček a z výsledného pole pak použijete první a pátý prvek (indexy 0 a 4):

```
while ( $radek = <> ) {
   @uzivatel = split(/:/, $radek);
   print "<A HREF=\"/~$uzivatel[0]\">";
   print "$uzivatel[4]</A>\n";
}
```

Program uložte třeba do souboru *htmlseznam* a spusťte

```
perl htmlseznam /etc/passwd
```

Část 6: Modifikátory, druhy regulárních výrazů

Od minula si umíte nalezené věci zapamatovat a později použít. To je v reálném životě šeredně nebezpečná vlastnost. Například jste teď nepoužitelní jako voliči. Abyste se opět stali politicky korektními, naučím vás pozměnit, co jste si zapamatovali.

Modifikátory

Kdybych chtěl být protivně korektní, vlastně bych o modifikátorech neměl mluvit, protože oficiálně nepatří mezi regulární výrazy. Používají se však v těsné součinnosti s nimi, takže zpravidla bývají házeny do jednoho pytle.

Modifikátory jsou konstrukce, které změní následující znaky. Jejich typické použití nese následující znaky:

- slouží k vzájemnému převodu malých/velkých písmen
- vyskytují se ve druhé (nahrazující) části příkazu pro nahrazování řetězců
- bývají aplikovány na zapamatované řetězce.

Těmi nejjednoduššími jsou **\u** a **\l**. První z nich převede následující znak z malého písmene na velké (upper case). Následuje-li cokoli jiného, než malé písmeno, zůstane znak beze změny. Konstrukce **\l** funguje právě opačně – převádí následující velké písmeno na malé (lower case).

Uvedené modifikátory se týkaly vždy jen jediného znaku, který následuje bezprostředně za nimi. Pokud použijete velké písmeno (**\U** či **\L**), zasáhne modifikátor všechny následující znaky až po konec řetězce nebo nejbližší následující výskyt **\E**, kterážto konstrukce slouží jako ukončující.

Bohužel jsou modifikátory poměrně vzácné. grep a jeho bratři nedovedou nahrazovat, takže u nich nemají smysl, nemá je sed ani awk. Z mnou běžně používaných nástrojů nabízí modifikátory jen vim (jak je na tom klasické vi nevím) a Perl. Pokud používáte něco jiného, je potřeba nahlédnout do dokumentace.

Příklad:

A zase něco ze života: vytvořili jste WWW stránky v prostředí MS Windows a následně je vystavili na Unixovém serveru. A nestačíte se divit. Část obrázků záhadně zmizela, nefungují dříve funkční odkazy a podobně. Příčina je v tom, že Unix a protokoly pohánějící WWW rozlišují malá písmena od velkých, zatímco MS Windows nikoli. Máte-li na stránku vložen obrázek se jménem *Image1.gif*, ale soubor se ve skutečnosti jmenuje *image1.gif*, ve Windows vypadá vše v pořádku, ačkoli ve skutečnosti není.

Řešení: je třeba sjednotit velikost znaků – nejlépe tak, že se vše převede na malá písmena. Takže si vypíšete seznam souborů v adresáři se stránkami (raději pomocí find, aby výpis obsahoval i cesty do podadresářů), vychytáte ta jména, která obsahují velká písmena, a převedete je na malá:

```
find . | grep ’[A–Z]’ > seznam
perl –pe ’s/(.*)/mv \1 \L\1/’ seznam > akce
source akce
rm seznam akce
```

Po provedení těchto příkazů máte soubory přejmenovány na malá písmena (druhý řádek není zcela korektní, vrátím se k němu na konci příkladu). Zbývá vypořádat se s odkazy na ně. Za nejschůdnější považuji vytvořit si konverzní program *maleodkazy*:

```
#!/bin/sh
for soubor in $*
do
perl –pei.bak ’s/(href=)([’”][^’”]+[’”])/\1\L\2/ig;
   s/(src=)([’”][^’”]+[’”])/\1\L\2/ig’ $soubor
done
```

V substitucích si všimněte, že kromě volby **g** (nahradit všechny výskyty) obsahuje i volbu **i** (ignorovat rozdíly mezi malými a velkými písmeny). Díky ní nezáleží na tom, jakými písmeny jsou psány názvy atributů SRC a HREF. Zároveň jsem si je zapamatoval zvlášť a vystrčil před **\L**, aby velikost jejich písmen zůstala zachována.

Volba –i.bak způsobí, že Perl soubor zpracuje „na místě“ a zálohu jeho původního obsahu uloží s příponou *.bak*.

Program pak vypustíte na odpovídající soubory

```
./maleodkazy *.html
```

Jsou-li stránky i v podadresářích, bude zřejmě lepší vložit jména souborů pomocí příkazu find uzavřeného ve zpětných apostrofech:

```
./maleodkazy `find . –name \*.html`
```

Avizoval jsem, že přejmenování souborů není zcela v pořádku. Bude zlobit, pokud se velká písmena vyskytnou ve jménech podadresářů, takže pokud například aktuální adresář bude obsahovat soubor *Abc/Def*, vygenerují se příkazy

```
mv Abc abc
mv Abc/Def abc/def
```

přičemž v okamžiku provádění druhého již adresář *Abc* neexistuje. Řešit by se to dalo tak, že oddělíte část cesty před lomítkem a část za ním. U části před lomítkem pak můžete předpokládat, že už je malými písmeny. Aby se navíc negenerovala hlášení o přesunu souborů na sebe samé, je záhodno oddělit zpracování adresářů a souborů:

```
#adresáře
find . –type d | grep '[A–Z]' > seznam
perl –pe 's/(.*\/)(.*)/mv \L\1\E\2 \L\1\2/' seznam > akce
source akce

#soubory – adresáře už nezlobí
find . | grep '[A–Z]' > seznam
perl –pe 's/(.*)/mv \1 \L\1/' seznam > akce
source akce
rm seznam akce
```

Klasické a rozšířené regulární výrazy

Programy grep a egrep rozštěpily regulární svět na dva proudy. Každý z nich používal poněkud jinou variantu regulárních výrazů a také jiný algoritmus pro jejich hledání. Regulární výrazy grepu představují klasickou variantu, zatímco odrůdě reprezentované egrepem se říká rozšířené regulární výrazy.

Základní rozdíly rozšířených regulárních výrazů proti klasickým byly následující: zavedly speciální znaky **+** a **?** pro alespoň jeden a nanejvýš jeden výskyt, znak | pro „nebo“ a nepodporovaly mechanismus zapamatování.

Postupem času se však původní rozdíly smazávaly a obě odrůdy implementovaly konstrukce, které byly původně k dispozici jen u konkurence. Takže mezi současným GNU grepem a GNU egrepem je rozdíl pouze v tom, kde se píší zpětná lomítka.

V předminulém odstavci jste možná zastříhali ušima, protože jsem se zmínil o dosud utajené konstrukci. Jedná se o „nebo“, které se zapisuje v podobě svislé čáry (|). U klasických regulárních výrazů se před ni zapisuje zpětné lomítko (pokud je vůbec k dispozici).

„Nebo“ má nižší prioritu než zřetězení, takže například egrep 'raz|dva' najde řádky obsahující řetězec „raz“ nebo „dva“. Pokud potřebujete účinek omezit, použijte závorky.

Příklad:

Následující příkaz vyhledá řádky, které začínají řetězcem „From:“ nebo „Subject:“ (může se hodit například pro prohledávání vaší poštovní schránky):

```
egrep '^(From|Subject):'
```

V podání grepu by příkaz vypadal takto:

```
grep '^\(From\|Subject\):'
```

Regulární stroje

Nastal čas podívat se alespoň orientačně na zoubek algoritmům, které se starají o interpretaci regulárních výrazů. Lze je rozdělit do tří kategorií:

Klasický nedeterministický Tento algoritmus jsem popsal ve druhé části seriálu. Je založen na rekurzi a své usilování skončí, jakmile narazí na první řetězec vyhovující srovnávanému vzoru. Používá jej valná většina programů – grep, vi, Perl a další.

Nedeterministický podle POSIXu Jedná se o variantu klasického algoritmu, jejíž chování se liší, pokud regulární výraz obsahuje „nebo“. POSIX varianta se nezastaví při prvním nálezu. Místo toho prozkoumá všechny možné vyhovující řetězce a z těch, které začínají nejdříve, pak vybere ten nejdelší. Důsledkem je, že výsledek nezávisí na pořadí alternativ.

> **Příklad:**
>
> Vezměme řetězec „12345“ a srovnejme jej s regulárními výrazy **1+|[0–9]+** a **[0–9]+|1+**. Pokud se používá klasický nedeterministický algoritmus, budou výsledky rozdílné: prvnímu výrazu vyhoví podřetězec „1“, zatímco při použití druhého vyhoví celý řetězec „12345“. Klasický algoritmus začne zkoumat první alternativu a jakmile najde vyhovující řetězec, skončí. Naproti tomu při POSIXovém algoritmu v obou případech vyhoví „12345“, protože obě alternativy začínají na stejné pozici a tato je delší.

Za svou nezávislost na pořadí platí POSIXový algoritmus rychlostí. Jelikož musí prozkoumat všechny varianty, je nejpomalejší. Upřímně řečeno představuje spíše teoretickou možnost, protože jsem se dočetl pouze o několika málo programech, které jej implementují (jediné obecněji rozšířené jsou lex a mawk).

Deterministický Deterministický algoritmus stojí na zcela odlišném principu. Nepoužívá rekurzi, ale paralelně sleduje všechny možné způsoby, jak vyhovět danému vzoru. Stejně jako předchozí tedy najde všechny alternativy a vybere nejdelší z nejlevějších, takže je také nezávislý na pořadí. Jelikož celý řetězec zpracuje jedním průchodem bez návratů, je nejrychlejší. Platí za to nemožností zapamatovat si části vyhovující jednotlivým podvýrazům.

Deterministický algoritmus používá egrep a awk.

Z hlediska taxonomie regulárních strojů mají velmi zvláštní pozici GNU nástroje. Jsou totiž dvoumotorové: obsahují implementaci jak deterministického, tak klasického nedeterministického algoritmu. Pokud to jde, používají rychlejší deterministický. Jakmile však přikážete něco si zapamatovat, sáhnou po nedeterministickém. Díky tomu například GNU egrep na rozdíl od klasického egrepu nabízí mechanismus zapamatování.

Část 7: Speciality Perlu

Na poslední díl našeho seriálu jsem si pošetřil speciality jazyka Perl. Pokud se vám při zahlédnutí jména Perl povážlivě zvedá hladina adrenalinu, podívejte se alespoň na konec článku. Najdete tam odkazy na literaturu a srovnávací tabulku speciálních znaků, kteréžto materiály by se vám mohly hodit.

Perl je do značné míry srdeční záležitost. Znám řadu jeho vášnivých zastánců a zrovna tak jeho zuřivé odpůrce. Ten jazyk je pohledem teoretika mimořádně odporný, ale zároveň neuvěřitelně mocný a praktický. Docela výstižně (a velmi vtipně) to myslím vyjádřil Jeffrey E. F. Friedl ve své knize o regulárních výrazech:

> Síla Perlu může být ničivou zbraní v rukách zkušeného uživatele, zdá se však, že v Perlu sbíráte zkušenosti tak, že se opakovaně střílíte do nohy.

Dnes se ale nehodlám věnovat jazyku jako takovému. Raději se soustředím na jeho speciality v oboru regulárních výrazů. A v tomto směru je v současnosti jasná jednička, ať se to jeho odpůrcům líbí nebo ne.

Uvolněná syntax

Regulární výrazy jsou velmi kondenzované, což je bohužel činí obtížně srozumitelnými. Obecně si troufám prohlásit, že regulární výraz je často snadnější vytvořit než jej sám po sobě pochopit.

Do jazyka Perl (přesněji řečeno do verze 5, ale ta už je na světě pět let, takže ji lze považovat za standard jazyka) proto přibyla možnost rozvolnění zápisu regulárních výrazů. Nepřidává žádné nové konstrukce, jen je umožňuje srozumitelně zapsat.

Uvolněnou syntax zapíná volba **x** v operátoru pro hledání či nahrazování (operátory **m** a **s**). Regulární výraz se pak zapisuje do složených závorek podobně jako blok programu, ignorují se v něm mezery a konce řádků (pokud jim nepředchází zpětné lomítko) a komentáře zahájené znakem #.

Příklad:

V části o zapamatování jste se mohli setkat s nechutnou substitucí

```
s/([^:]*):([^:]*:){3}([^:]*).*/<A HREF="/~\1">\3<\/A>/
```

která z řádků v */etc/passwd* vyráběla seznam domácích stránek uživatelů. S využitím uvolněné syntaxe bychom ji mohli přepsat následovně:

```
s{
  ([^:]*):         #přihlašovací jméno (1)
  ([^:]*:){3}      #přeskočíme heslo, UID a GID
  ([^:]*)          #vlastní jméno (3)
  .*               #přeskočíme zbytek
  }{<A HREF="/~\1">\3</A>}x
```

Nepřipadá vám to *o dost* srozumitelnější? Upozorňuji, že uvolněná syntax se týká pouze regulárního výrazu, nikoli nahrazujícího řetězce. V něm se projeví pouze tím, že je uzavřen do složených závorek.

Drobnosti, které potěší

Perl nabízí řadu konstrukcí, které sice nepřinášejí nějakou zásadní inovaci stran schopností regulárních výrazů, ale v běžném životě silně potěší. Asi nejčastěji používané jsou kategorie znaků. Z těch nejběžnějších lze jmenovat:

Zápis	*Význam*	*Odpovídá*
\d	číslice	[0–9]
\D	nečíslice	[^\d]
\w	alfanumerický znak	[a–zA–Z_0–9]
\W	nealfanumerický znak	[^\w]
\s	prázdný znak	[\ \t\n\r]
\S	neprázdný znak	[^\s]

Tyto speciální znaky celkem výrazně přispívají ke srozumitelnosti regulárních výrazů. Navíc pokud svůj program zahájíte příkazem use locale, budou mezi alfanumerické znaky zařazeny i akcentované znaky národní abecedy.

Příklad:

Výměnu prvních dvou slov na řádku pak zajistí celkem mírumilovný příkaz

```
s/(\w+)(\W+)(\w+)/\3\2\1/
```

Kromě Perlu už převzaly podobné znakové kategorie i současné verze dalších nástrojů (konzultujte s dokumentací). Kromě nich je leckde podporován i zápis kategorií podle POSIXu, kde se například číslice zapisuje jako **[:digit:]**, alfanumerický znak jako **[:alnum:]** a prázdné místo **[:space:]**. Tyto kategorie se však zapisují mezi **[]**, takže například řádek začínající číslicí se vyjádří pomocí **^[[:digit:]]**, což už má k eleganci poměrně daleko (totéž v Perlu: **^\d**). Perl POSIXové kategorie podporuje až od verze 5.6.

Syté (též líné) opakování pomůže řešit problémy s nadměrnou žravostí opakovacích konstrukcí. Zapisuje se jednoduše: za opakovací znak (*, +, ? či {min,max}) připojíte otazník. Přípustný počet opakování se tím nijak nezmění, ale na rozdíl od klasické varianty se snaží, aby opakování bylo co nejméně. Takže třeba oblíbený řetězec v uvozovkách lze hledat pomocí **".*?"**. Je to přehlednější, ale pomalejší než obvyklé řešení **"[^"]*"**.

Třetí užitečnou drobností jsou závorky bez zapamatování. Slouží výlučně k vymezení části regulárního výrazu (např. pro opakování nebo omezení působnosti „nebo“). Vyhovující řetězec se neukládá, takže vám ubyde špetka starostí při počítání indexů těch zapamatovaných. Tyto speciální závorky se zapisují ve tvaru **(?:...)**.

Příklad:

Ještě jednou přepracuji příklad pro transformaci řádku z */etc/passwd* na položku v seznamu domácích stránek. S využitím popsaných konstrukcí by mohl vypadat takto:

```
s{
  (.*?):          #přihlašovací jméno (1)
  (?:.*?:){3}     #přeskočíme heslo, UID a GID
  (.*?):          #vlastní jméno (2)
  .*              #přeskočíme zbytek
  }{<A HREF="/~\1">\2</A>}x
```

Tentokrát se v části přeskakující heslo, UID a GID vyhnu zapamatování, takže pod čísly 1 a 2 mám uloženy skutečně jen ty informace, které mne zajímají. Dvojtečka z konstrukce **(?:** se v tomto případě bohužel nepěkně plete s oddělovačem položek, ale s tím se nedá nic dělat. Líné opakování mi umožnilo poněkud zjednodušit výrazy pro jednotlivé části řádku.

Vyhlížení

Dost silný nástroj pro některé speciální případy nabízí tak zvané vyhlížení (anglicky lookahead). Existuje ve dvou odrůdách: pozitivní vyhlížení se zapisuje v podobě **(?=*výraz*)** a je splněno, pokud následující část řetězce vyhovuje *výrazu*. Negativní vyhlížení **(?!*výraz*)** naopak uspěje, pokud *výrazu* nevyhovuje. Důležité je, že vyhlížení se jen podívá, zda se vzor dá či nedá najít, ale neposouvá zkoumanou pozici dál (jemu odpovídající řetězec je vždy prázdný).

Příklad:

Vzoru **\d+(?= Kč)** vyhoví neprázdná posloupnost číslic, ale jen pokud za ní následuje řetězec „ Kč“. Ten však sám o sobě není zahrnut do vyhovujícího řetězce. Takže pokud příkaz

s/(\d+)(?= Kč)/???/g

vypustíte na řetězec

Párátka 20 CX Turbo za 25 Kč kus

obdržíte výsledek

Párátka 20 CX Turbo za ??? Kč kus

Za příklad negativního vyhlížení může posloužit **(?!000)\d\d\d**, kterému vyhoví libovolná trojice číslic s výjimkou 000.

Faktem je, že vyhlížení není principiálně nezbytné. Zpravidla je lze nahradit jinými konstrukcemi jazyka. Například zachování „ Kč“ za skupinou číslic by se dalo zařídit pomocí zapamatování. Test na trojici číslic, ne však 000 by se dal rozložit do dvou nezávislých testů a podobně. V některých případech však vyhlížení dává zajímavé možnosti.

Příklad:

Hezkou vychytávkou využívající vyhlížení je oddělování řádů ve velkých číslech. Mezi trojice číslic se mají vložit mezery, takže z „12345678“ vznikne „12 345 678“. Vtip je v tom, že se trojice počítají odzadu, na což regulární výrazy nejsou příliš zařízené. Vyhlížení umožňuje následující řešení:

```
s{
  (\d{1,3})          #za první skupinu patří mezera
  (?=                #ale jen když následuje
    (?:\d\d\d)+      #alespoň jedna trojice číslic
    (?!\d)           #a tím číslo končí
  )
  }{\1 }gx
```

Díky vyhlížení se můžete přesvědčit, že počet číslic, které následují za aktuální pozicí ve zpracovávaném řetězci, je násobkem tří. Vyhlížení zároveň zajistí, že se tato pozice nezmění a že se při příštím opakování (díky volbě **g** se provádí pro všechny výskyty) se bude pokračovat za naposledy vloženou mezerou.

Literatura

Pokud je mi známo, vyšla zatím jediná kniha věnovaná výlučně regulárním výrazům. Napsal ji Jeffrey E. F. Friedl a jmenuje se *Mastering Regular Expressions* (vydalo nakladatelství O'Reilly & Associates v roce 1997, ISBN 1–56592–257–3). Není to pochopitelně vyslovená oddechovka, ale v rámci možností vysvětluje vlastnosti regulárních výrazů, vhodné a nevhodné techniky pro jejich vytváření. Vzhledem k rozsahu přes 300 stran si může dovolit jít dost do hloubky.

Cenné služby odvede kniha *Unix Power Tools*, jejímiž autory jsou Jerry Peek, Tim O'Reilly a Mike Loukides. Vyšla také u O'Reilly & Associates v roce 1997 (2. vydání, ISBN 1–56592–260–3). Obsahuje velmi slušnou kapitolu o regulárních výrazech a kromě ní řadu tipů a triků na využívání programů, které s nimi pracují. Osobně ji považuji za jednu z nejpřínosnějších knížek, které jsem kdy měl v ruce.

Přehledová tabulka

Jako závěrečnou přílohu vám nabízím stručný přehled regulárních výrazů v nejběžnějších nástrojích. Konkrétně se jedná o *GNU grep* a *egrep* verze 2.3, *GNU awk* verze 3.0.4, *vim* verze 5.5 a *Perl* verze 5.005_03.

Regulární výrazy

	GNU grep	*GNU egrep*	*GNU awk*	*vim*	*Perl*
znaky					
libovolný znak (kromě \n)	.	.	.	.	.
jeden ze znaků	[ ]	[ ]	[ ]	[ ]	[ ]
kromě těchto znaků	[^]	[^]	[^]	[^]	[^]
skupiny \w|\W|\d|\D|\s|\S	●|●|○|○|○|○	●|●|○|○|○|○	○|○|○|○|○|○	●|●|○|○|○|○	●|●|●|●|●|●
POSIXové skupiny	●	●	●	●	○
opakování					
libovolný počet	*	*	*	*	* *?
alespoň jeden	\+	+	+	\+	+ +?
nanejvýš jeden	\?	?	?	\?	? ??
min až *max*	\{*min*,*max*\}	{*min*,*max*}	{*min*,*max*}	\{*min*,*max*\}	{*min*,*max*} {}?
pozice					
začátek řetězce	^	^	^	^	^
konec řetězce	$	$	$	$	$
hranice slova	\< \>	\< \>	\< \> \y	\< \>	\b
závorky a paměť					
skupina se zapamatováním	\(\)	( )	○	\(\)	( )
skupina bez paměti	○	○	( )	○	(?:)
třetí zapamatovaný	\3	\3	○	\3	\3 $3

Legenda: ● podporuje ○ nepodporuje

Obsah