UNIVERZITA PARDUBICE
ÚSTAV ELEKTROTECHNIKY A INFORMATIKY

VYUŽITÍ REGULÁRNÍCH VÝRAZŮ PŘI OPRAVÁCH TEXTU, ÚPRAVÁCH A GENEROVÁNÍ SOUBORŮ

BAKALÁŘSKÁ PRÁCE

2007 Miloš Kukla

UNIVERZITA PARDUBICE
ÚSTAV ELEKTROTECHNIKY A INFORMATIKY

VYUŽITÍ REGULÁRNÍCH VÝRAZŮ PŘI OPRAVÁCH TEXTU, ÚPRAVÁCH A GENEROVÁNÍ SOUBORŮ

BAKALÁŘSKÁ PRÁCE

AUTOR PRÁCE: Miloš Kukla
VEDOUCÍ PRÁCE: RNDr. Josef Rak

2007

UNIVERSITY OF PARDUBICE
INSTITUTE OF ELECTRICAL ENGINEERING
AND INFORMATICS

USING OF REGULAR EXPRESSIONS FOR REPAIRING TEXT, MODIFYING AND GENERATING FILES

BACHELOR WORK

AUTHOR: Miloš Kukla
SUPERVISOR: RNDr. Josef Rak

2007

Univerzita
Pardubice
Ústav elektrotechniky
a informatiky

Vysokoškolský ústav: Ústav elektrotechniky a informatiky
Katedra/Ústav: Ústav elektrotechniky a informatiky
Akademický rok: 2006/2007

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Pro: Kukla Miloš

Studijní program: Informační technologie

Název tématu: Využití regulárních výrazů při opravách textu, úpravách a generování souborů

Zásady pro zpracování:
Regulární výrazy mají díky široké škále zástupných znaků a možností ukládání nalezených výrazů velké možnosti použití při korekci textu, textových a jiných (html, tex,...) souborů a jejich generování. Úkolem bude studium regulárních výrazů a jejich aplikací na úpravy v textu (základní opravy pravopisu), v korekce v různých typech souborů a generování různých souborů. Výsledkem bude také program (programy), který bude text (textové soubory) zpracovávat.

Seznam odborné literatury:
Pavel Satrapa – reulární výrazy:
http://www.kit.vslib.cz/~satrapa/docs/regvyr/

Rozsah: 30 stran

Vedoucí práce: Rak Josef

Vedoucí katedry (ústavu): prof. Ing. Pavel Bezoušek, CSc.

Datum zadání práce: 31. 11. 2006

Termín odevzdání práce: 18. 5. 2007

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně Univerzity Pardubice.

V Pardubicích dne 1. 3. 2007

ABSTRAKT

Tato práce se zabývá problematikou regulárních výrazů. Teoretická část obsahuje informace o vzniku regulárních výrazů, o jejich současném rozšíření v různých nástrojích a programovacích jazycích. Popisuje složení regulárních výrazů, principy jejich tvorby a využití v praxi demonstrované na příkladech. V praktické části jsem se zaměřil na opravu kontaktních údajů (e-mail, telefon, PSČ, titul, jméno a příjmení) v databázích osob. Jako vstupní data používám textové csv soubory a pomocí nástroje napsaného v jazyce Perl kontroluji a opravuji údaje.

Obsah

1. ÚVOD

Regulární výraz je množina znaků, která představuje určitý vzor. Pomocí regulárních výrazů můžeme vyhledávat v textu řádky obsahující žádaný vzor, tím však nemusí být jen konkrétní slovo. Můžeme si nadefinovat masku odpovídající např. struktuře e-mailové adresy nebo telefonního čísla a pomocí jediného regulárního výrazu tak vyhledat všechny e-mailové adresy nebo telefonní čísla v textu. Vyhledané údaje pak můžeme pomocí nahrazování přetvořit do podoby kterou potřebujeme, např. doplnit k e-mailové adrese html tagy a vytvořit tak odkazy do webové prezentace.

V současné době podporuje regulární výrazy velké množství programovacích jazyků (*C++*, *Delphi*, *Java*, *JavaScript*, *.NET*, *PCRE*, *Perl*, *PHP*, *Python*, *Ruby*, *VBScript*, *Visual Basic 6*) a základní znalost regulárních výrazů nám může ušetřit mnoho řádků kódu (především různé if-then-else konstrukce). Kromě programovacích jazyků najdeme regulární výrazy v některých textových editorech (jak pro Linux tak i pro Windows). Pro tvorbu složitých regulárních výrazů existují speciální programy a online testery.

2. HISTORIE

Původ regulárních výrazů leží v teorii automatů a formálních jazyků. Oba obory jsou součástí teoretické informatiky a zkoumají způsob jak popsat a ohodnotit jazyk. V 50tých letech dvacátého století popsal matematik Stephen Kleene tyto modely pomocí jeho matematického zápisu nazvaného regulární sady. Tento zápis použil Ken Thomson v interaktivním textovém editoru *QED* pro hledání vzorů v textových souborech. Později přidal tuto schopnost také do Unixového editoru *ed*, z kterého se posléze vyvinul populární vyhledávací nástroj *grep* používající regulární výrazy. "grep" je slovo odvozené z příkazu pro vyhledávání regulárního výrazu v editoru *ed* (g/re/p, kde re zastupuje regulární výraz). Od té doby vzniklo mnoho variací odvozených z Thompsonovi definice a tyto variace jsou široce využívány především v Unixových nástrojích (*expr*, *awk*,

Emacs, *vi*, *lex*, *Perl*).

3. METAZNAKY

Regulární výraz sestavíme pomocí speciálních řídicích znaků. Kdo zatím o regulárních výrazech neslyšel může si pod těmito znaky přestavit např. `*` nebo `?`, kterých se využívá především v operačním systému DOS nebo v Unixovém shellu jako zástupných znaků při selekci souborů. V regulárních výrazech nejsme omezeni jen na `*` a `?` ale máme k dispozici mnohem více znaků, říká se jim **metaznaky**.

Tečka

Tečka (`.`) zastupuje právě jeden libovolný znak. Tedy regulárnímu výrazu `Iv.na` vyhoví jména `Ivana` a `Ivona`.

Kvantifikátory

Tyto znaky určují, kolikrát se smí opakovat znak předcházející kvantifikátoru. Do této skupiny patří `?`, `*`, `+`, `{}`. Tabulka 1 popisuje význam těchto znaků.

Tabulka 1: Kvantifikárory

Kvantifikátor	***Počet opakování***
`?`	Minimálně 0krát, maximálně 1krát
`*`	Minimálně 0krát, maximálně neomezeno
`+`	Minimálně 1krát, maximálně neomezeno
`{n}`	Právě n-krát
`{m,n}`	Minimálně m-krát, maximálně n-krát
`{m,}`	Minimálně m-krát, maximálně neomezeno
`{,m}`	Maximálně m-krát

Pokud chceme kvantifikátor uplatnit na více než jeden předcházející znak tak požadovanou sekvenci znaků uzavřeme do kulatých závorek a kvantifikátor umístíme hned za pravou závorku. Například výrazu `(pra)*rodiče` vyhoví slova `rodiče`, `prarodiče`, `praprarodiče` atd.

Všechny výše uvedené kvantifikátory jsou nenasytné, to znamená že se snaží zachytit maximální možný počet znaků, který vyhovuje regulárnímu výrazu. Pokud se nám toto nehodí a potřebujeme zachytit minimální počet znaků, který vyhoví výrazu, tak musíme použít tzv. **líné kvantifikátory**. Ty vytvoříme z výše uvedených tak, že je zprava doplníme o otazník (`??`, `*?`, `+?`, `{m,n}?`, `{m,}?`).

Regulární výraz `(po)+` užitý na slovo `popokatepetl` vrátí řetězec `popo`, kdežto výraz `(po)+?` vrátí pouze `po`.

Skupiny znaků

Někdy potřebujeme omezit výběr znaků jen na určitou skupinu, např. na číslice nebo na velká písmena. Pro definování skupiny nám slouží hranaté závorky (`[` a `]`). Pokud chceme ze seznamu lidí vybrat jen jména začínající jedním z prvních třech písmen abecedy tak použijeme skupinu `[ABC]`, celý regulární výraz pak může vypadat takto:

`[ABC].+` - skupina říká že slovo musí začínat velkým písmenem A, B nebo C a za ním musí být minimálně jeden libovolný znak nebo libovolný počet libovolných znaků.

Abychom nemuseli vypisovat všechny znaky, které chceme do skupiny zahrnout, můžeme definovat jejich rozsah pomocí pomlčky (`-`), například pro všechny číslice definujeme skupinu `[0-9]`, pro všechna písmena malé abecedy `[a-z]`, pro velká i malá písmena abecedy a číslice `[0-9a-zA-Z]`. Pomlčka označuje interval jen pokud má z obou stran jeho meze, výraz `[09-]` platí pouze pro číslo `0`, `9` a znak pomlčka.

Můžeme také požadovat, aby do výběru patřily jen ty znaky, které neuvedeme ve skupině, potom je nejjednodušší použít negaci. Negaci provedeme pomocí stříšky (`^`) umístěné hned za levou hranatou závorku. Pokud stříška nebude jako první znak za závorkou tak ztratí svůj význam negování a stane se z ní jeden ze znaků skupiny. Pro získání všech znaků kromě číslic použijeme skupinu `[^0-9]`, pokud bychom však napsali `[0-9^]` tak skupině vyhoví všechna čísla a znak `^`.

Problém nastává pokud chceme definovat interval zahrnující i české znaky s diakritikou, regulární výrazy totiž berou pořadí znaků podle ASCII tabulky, která obsahuje znaky anglické abecedy.

Pro často používané skupiny znaků existují speciální zkratky, ty se ale liší podle verze regulárních výrazů. Tabulka 2 obsahuje jejich seznam.

Tabulka 2: Předdefinované skupiny znaků

Význam	***Perl***	***Klasické***	***Odpovídá***
číslice	`\d`	`[[:digit:]]`	`[0-9]`
nečíslice	`\D`	`[^[:digit:]]`	`[^0-9]`
písmena	`x`	`[[:alpha:]]`	`[a-zA-Z]`
alfanumerický znak	`\w`	`[[:alnum:]]`	`[a-zA-Z_0-9]`
nealfanumerický znak	`\W`	`[^[:alnum:]]`	`[^a-zA-Z_0-9]`
prázdný (bílý) znak	`\s`	`[[:space:]]`	`[\ \t\n\r]`
neprázdný znak	`\S`	`[^[:space]]`	`[^\ \t\n\r]`

Hranice

Do této chvíle se mohl regulární výraz vyskytovat kdekoliv v prohledávaném textu a nebylo možné přesněji určit jeho pozici. I pro tento případ však existují speciální poziční znaky. Začátek zkoumaného řetězce označuje stříška (`^`), jeho konec označuje dolar (`$`). Výraz `^[A-Z]` najde řádky začínající velkým písmenem, výraz `,$` najde řádky, které mají na konci čárku.

Dále můžeme určit také hranici slova. Tady už však záleží na tom, jakou verzi regulárních výrazů používáme. V programech používajících klasické regulární výrazy (*awk*, *grep*) tvoří hranici slova znaky `\<` (začátek slova) a `\>` (konec slova). V *Perl-compatible* regulárních výrazech je pro začátek i konec slova použit znak `\b`, navíc je zde ještě znak `\B`, kterému vyhoví vše kromě hranice slova (je to tedy negace `\b`). Pro vyhledání spojky `ale` použijeme regulární výraz `\<ale\>`, v *Perlu* potom `\bale\b`. O dalších rozdílech mezi verzemi regulárních výrazů budu psát dále.

Oddělovač variant

Funkci logického operátoru `nebo` plní znak roura (`|`). Použijeme ho pokud chceme dát na výběr několik variant textu. Pro vyhledání všech sobot a nedělí napíšeme `sobota|neděle`.

Původní význam metaznaků

Někdy potřebujeme, aby speciální znaky ztratily svůj význam a my s nimi mohli pracovat jako s obyčejnými znaky. Toho dosáhneme pokud před speciální znak umístíme zpětné lomítko (`\`). Mezi metaznaky, které takto musíme zbavit jejich funkce, patří: `\`, `^`, `$`, `.`, `[`, `]`, `|`, `(`, `)`, `?`, `*`, `+`, `{`, `}`. Pro vyhledání všech příkladů na sčítání musíme použít znaménko + zbavené jeho speciálního významu. Příklad může vypadat takto:

```
\d+ \+ \d+ = \d+
```

Například řetězec `10 + 5 = 15` vyhoví výše uvedenému regulárnímu výrazu.

4. ALGORITMY

Jsou tři způsoby, kterými může program zjišťovat, zda v prohledávaném textu některý řetězec vyhovuje zadanému regulárnímu výrazu:

Klasický nedeterministický

Při hledání vzoru, který by vyhověl regulárnímu výrazu používá rekurzi a jakmile najde první vyhovující řetězec tak hledání ukončí. Začíná se na začátku řetězce a postupuje se znak po znaku tak dlouho, dokud tento znak nevyhoví začátku regulárního výrazu. Poté se vezme celá zbývající část řetězce a stejným způsobem se hledá znak odpovídající konci regulárního výrazu s tím rozdílem, že se posouváme opačně, tedy od konce řetězce k jeho počátku. Jakmile je nalezený začátek i konec regulárního výrazu tak se zjišťuje, zda vybraný řetězec odpovídá i dalším částem regulárního výrazu. Pokud ano, je vyhledávání ukončeno, pokud ne tak se začátek posune o další znak doprava a celý postup se opakuje.

V ukázkovém příkladu máme html text a chceme najít text psaný

tučně, například pro pozdější nahrazení kurzívou. Pro hledání značek označujících začátek a konec tučného textu (a) použijeme regulární výraz .*.

1) V textu je slovo1 a slovo2 tučně.

Začíná se na prvním znaku a postupujeme dokud nenarazíme na html tag .

2) V textu je slovo1 a slovo2 tučně.

Po nalezení začátku regulárního výrazu se vezme celý zbytek řetězce.

3) V textu je slovo1 a slovo2 tučně.

Nyní postupujeme od tečky směrem na začátek dokud nenarazíme na html tag uzavírající tučně psaný text.

4) V textu je slovo1 a slovo2 tučně.

Tímto vyhledávání končí a označený text vyhovuje regulárnímu výrazu .*.

V uvedeném příkladu je vidět jedna z vlastností regulárních výrazů a tou je **nenasytnost** – regulární výraz se vždy snaží roztáhnout na co největší délku. To se nám v tomto případě nehodí, protože výraz nám úplně ignoruje uzavírací tag u slovo1 a otevírací tag u slovo2. A navíc pokud bychom chtěli z tučného písma udělat kurzívu, tedy nahradit tagy , za tagy , , tak dostaneme:

V textu je slovo1 a slovo2 tučně.

A to je v html syntaxi špatně, protože nám zůstaly dva neuzavřené tagy pro tučné písmo (a). Pokud chceme získat text uzavřený právě jedním párem tagů tak musíme náš regulární výraz upravit. Máme dvě možnosti – můžeme napsat [^<], potom nám regulární

výraz označí pouze takový text ohraničený tagy `<b>` a `</b>`, který neobsahuje žádný další html tag. Druhá možnost je regulární výraz `<b>.*?</b>`. Otazník nám z nenasytného kvantifikátoru `*` udělá kvantifikátor líný a ten se místo co nejdelšího řetězce snaží najít ten nejkratší vyhovující. V obou případech bude výsledkem regulárního výrazu řetězec `<b>slovo1</b>`. Tady je vidět další vlastnost regulárních výrazů a tou je levostrannost. Ze dvou řetězců vyhovujících upravenému regulárnímu výrazu (`<b>slovo1</b>`, `<b>slovo2</b>`) je vybrán ten nejlevější.

Tento algoritmus používá většina programů (*grep*, *Perl* a další).

Nedeterministický podle POSIXu

Funguje stejně jako klasický, tj. používá rekurzi a hledá nejdelší a nejlevější posloupnost znaků, které vyhoví výrazu. Rozdíl nastane při použití oddělovače variant (`|`). Klasický algoritmus se zastaví při nalezení prvního z řetězců oddělených rourou a další možnosti již nezkoumá. Tento algoritmus nejprve vyhledá vyhovující řetězce pro všechny z variant oddělených rourou a teprve z nich vybere za výsledek nejdelší řetězec. Z toho plyne že u klasického algoritmu záleží na pořadí alternativ, u tohoto ne, vždy vybere tu nejdelší možnou. Nevýhodou tohoto algoritmu je jeho pomalost, protože musí hledat všechny varianty.

Deterministický

Tento algoritmus nepoužívá rekurzi ale paralelně sleduje všechny způsoby, které vyhoví regulárnímu výrazu. Najde všechny alternativy a vybere tu nejdelší a nejlevější a je nezávislý na pořadí. Tento algoritmus je ze tří uvedených nejrychlejší, protože celý řetězec zpracuje jedním průchodem. Oproti předchozím algoritmům má jednu nevýhodu, způsob jakým pracuje mu neumožňuje zapamatovat si části vyhovující jednotlivým podvýrazům.

Tento algoritmus používají programy *awk* a *egrep*.

Některé programy využívají dva algoritmy, klasický a deterministický. Běžně využívají rychlejší deterministický a když je potřeba zapa-

matovat si některou část regulárního výrazu tak použijí klasický algoritmus.

5. SPECIÁLNÍ KONSTRUKCE

Dále popsané konstrukce budou bez problémů fungovat v *Perlu* ale v některých jiných nástrojích pro práci s regulárními výrazy nemusejí být podporovány. O jejich podpoře se dočtete v manuálových stránkách nebo v nápovědě konkrétního programu.

Zapamatování

Pokud potřebujeme s řetězcem, který vyhověl některé části regulárního výrazu dále pracovat, máme k dispozici mechanismus zapamatování. Jeho použití je jednoduché, část kterou si chceme pamatovat uzavřeme do kulatých závorek a později se na ni odkážeme pomocí zpětného lomítka a indexu. Zapamatovaných částí můžeme mít libovolný počet, k jejich rozeznání slouží právě číselný index. Čísluje se od jedničky a pořadí levé závorky od začátku výrazu určuje hodnotu indexu. Zapamatované úseky lze navíc vzájemně vnořovat. Zapamatování se využije především při nahrazování řetězců, pro vyhledávání se příliš nehodí.

Nástroje které používají deterministický algoritmus zapamatování nepodporují. Patří mezi ně například *egrep* a *awk*.

Jako ukázku zapamatování použiji opět příklad s html tagy. Regulární výraz `<b>.*?</b>` doplníme kulatou závorkou, která nám umožní zapamatovat si text mezi tagy, takto: `<b>(.*?)</b>`.

```
V textu je <b>slovo1</b> a <b>slovo2</b> tučně.
```

Označené části textu odpovídají výrazu takto: `<b>(.*?)</b>`. Pro nahrazení kurzívou použijeme `<i>\1</i>`, kde `\1` vloží zapamatovanou část textu.

Uvedu ještě jeden příklad na vícenásobné zapamatování. Máme vstupní textový soubor, ve kterém je uloženo jméno, příjmení a rodné číslo a tyto údaje jsou oddělené středníkem. Z tohoto souboru chceme

vygenerovat nový soubor, ve kterém bude uloženo příjmení, jméno, rok narození, tyto pole budou také odděleny středníkem a navíc jako oddělovač textu budou použity uvozovky ("). Struktura souborů bude tedy vypadat takto:

Zdroj: `Bernau;Adam;710526/3131`

Cíl: `"Adam";"Bernau";"26. 05. 1971"`

Pro zjednodušení budeme počítat s tím, že rodné číslo patří muži narozenému do roku 2000 a že zdrojový text je bez diakritiky. Pokud by tomu bylo jinak museli bychom tyto případy ošetřit pomocí podmínek programovacího jazyka a zajistit podporu českých znaků v regulárních výrazech.

Regulární výraz pro vyhledání:

```
(\w+);(\w+);(\d{2})(\d{2})(\d{2})/\d{4}
```

```
Bernau;Adam;710526/3131
```

Regulární výraz pro nahrazení:

```
"\2";"\1";"\5. \4. 19\3"
```

```
"Adam";"Bernau";"26. 05. 1971"
```

Jak je vidět v příkladu, se zapamatovanými řetězci můžeme naložit libovolným způsobem, třeba měnit jejich pořadí nebo je doplňovat o další znaky a řetězce.

Vyhlížení

Vyhlížení je nástroj, který umožňuje podívat se na následující část řetězce a porovnat ji s regulárním výrazem použitým ve vyhlížení. Důležité je, že se tím nezmění aktuální pozice v řetězci, takže výraz před vyhlížením je ovlivněn ale samotné vyhlížení vrací prázdný řetězec. Existují dvě varianty – **pozitivní vyhlížení** se zapisuje `(?=výraz)` a je splněno, pokud následující část řetězce odpovídá výrazu. **Negativní vyhlížení** se zapisuje `(?!výraz)` a je splněno pokud následující část řetězce nevyhoví výrazu.

V následujícím příkladu chceme změnit hodnotu DPH z 21% na 19%. Pokud použijeme výraz `(\d+)(?=(% DPH))`, kterému díky vyhlížení odpovídá jen zvýrazněná hodnota `21`, stačí abychom celý regulární výraz nahradili hodnotou `19`.

```
Zboží stojí 549 Kč včetně 21% DPH.
```

Tento příklad můžeme řešit i bez vyhlížení použitím výrazu `(\d+)(% DPH)`, ten nám však vybere řetězec `21% DPH` a ten už nestačí nahradit hodnotou `19` ale musíme použít řetězec `19% DPH`.

Tvrzení o předcházejícím

Funguje na stejném principu jako vyhlížení, dívá se však na část řetězce předcházející regulárnímu výrazu. Stejně jako vyhlížení má dvě varianty, první je **kladné tvrzení o předcházejícím** a druhá **záporné tvrzení o předcházejícím**. Kladné tvrzení se zapisuje `(?<=výraz)` a záporné tvrzení se zapisuje `(?<!výraz)`.

V příkladu nahradíme v html souboru odkaz vedoucí na stránku `www.nekde.cz` adresou `www.jinde.cz`.

Regulární výraz: `(?<=<a href=")[^"]+`

Řetězec: `<a href="www.nekde.cz">Někde</a>`

Řetězec nahrazení: `www.jinde.cz`

Výsledek: `<a href="www.jinde.cz">Někde</a>`

Závorky bez zapamatování

Často použijeme několik párů kulatých závorek pro definici skupin v řetězci ale nepotřebujeme si tyto skupiny zapamatovat nebo nám stačí pamatovat si jen některé. Pro definování skupin, které si nechceme pamatovat a které tedy nebudou mít přiřazený index použijeme konstrukci `(?:výraz)`.

Na ukázku použiji příklad z kapitoly ***Zapamatování***. Tentokrát chci ze zdrojového souboru dostat pouze datum a rok narození. S použitím

původního regulárního výrazu musím počítat indexy všech šesti skupin i přesto, že pro získání data narození mi stačí skupiny `\3`, `\4`, `\5`. Při použití závorek bez zapamatování se regulární výraz změní takto:

```
(?:\w+);(?:\w+);(\d{2})(\d{2})(\d{2})/(?:\d{4})
```

Nyní mám zapamatované pouze tři skupiny, které obsahují datum narození a řetězec pro nahrazení bude:

```
\3. \2. 19\1
```

Modifikátory

Modifikátory jsou konstrukce, které nějakým způsobem změní řetězec následující po modifikátoru. Nejčastěji se používají pro změnu velikosti písmen při nahrazování zapamatovaných částí. Seznam a popis modifikátorů obsahuje Tabulka 3.

Tabulka 3: Modifikátory

Modifikátor	***Název***	***Funkce***
`\u`	upper case	převede následující znak na velké písmeno
`\U`	upper case	převede na velká písmena všechny následující znaky až dokonce řetězce nebo do výskytu prvního `\E`
`\l`	lower case	převede následující znak na malé písmeno
`\L`	lower case	převede na malá písmena všechny následující znaky až do konce řetězce nebo do výskytu prvního `\E`
`\E`	end	ukončuje modifikátory `\U` a `\L`
`i`	ignore case	ignoruje velikost písmen
`s`	single line	tečka (`.`) odpovídá i znaku konce řádku `\n`
`m`	multiple lines	`^` a `$` odpovídají i začátku a konci každého řádku
`x`	extended	bílé znaky a komentáře jsou ignorovány
`g`	global match	hledají se všechny části řetězce, které odpovídají regulárnímu výrazu a ne jen první vyhovující řetězec

Modifikátory oficiálně nepatří mezi regulární výrazy, proto jejich podporu najdeme jen v několika nástrojích, například v *Perlu* a v editoru

vim.

Komentáře

I v regulárních výrazech můžeme používat komentáře, zapisují se jako `(?#komentář)` a kulaté závorky ani jejich obsah nemají vliv na vyhodnocení regulárního výrazu.

Uvolněná syntax

V regulárním výrazu normálně nemůžeme používat bílé znaky (odsazení, odřádkování) jako v programovacích jazycích, to činí regulární výrazy velmi obtížně čitelnými. Tuto užitečnou věc nám zpřístupní modifikátor `x`, v regulárním výrazu jsou pak ignorovány mezery, konce řádků a komentáře, pokud jim nepředchází zpětné lomítko (`\`).

6. PODPORA V PROGRAMOVACÍCH JAZYCÍCH

Regulární výrazy najdeme v mnoha vývojových prostředích pro různé programovací jazyky. Někde jsou nativně podporovány, jinde jejich použití umožní různé moduly.

AWK

- nativní podpora regulárních výrazů
- výhody – silný nástroj pro práci se soubory s pevně danou strukturou (csv), rozšíření funkčnosti pomocí knihoven
- nevýhody – jednostrannost, neobvyklá syntaxe
- v prostředí Unixu i Windows, šířen jako GNU

C++

- podporu přinášejí knihovny **Boost.Regex** a **GRETA** v podobě sady šablon
- **Boost.Regex** – lze použít pro většinu kompilátorů (*Borland C++ Builder*, *Microsoft Visual C++*, *GNU gcc* a další)

– **GRETA** – pro *Visual Studio 7* a *gcc 3.2*

– obě jsou volně ke stažení pro komerční i nekomerční využití na adresách http://www.boost.org/libs/regex/doc/index.html a http://research.microsoft.com/projects/greta

Delphi

– neexistuje žádná knihovna, je nutné použít wrapper

– wrapper **TperlRegEx** – využívá volně šiřitelnou *PCRE* knihovnu

– ke stažení na http://www.regular-expressions.info/delphi.html

Java

– od verze *JDK 1.4.0* obsahuje balíček **java.util.regex** vydaný přímo firmou Sun

– existují i další neoficiální rozšíření, kvalita oficiálního balíčku je však na takové úrovni, že jejich používání je zbytečné

JavaScript

– od verze 1.2 podporuje regulární výrazy

– tato verze *JavaScriptu* je v prohlížečích *Internet Explorer* od verze 4, *Netscape* od verze 4, ve všech verzích *Firefoxu* a v řadě dalších moderních prohlížečů

.NET

– funkce pro práci s regulárními výrazy obsahuje knihovna **System.Text.RegularExpressions**

PCRE

– open source knihovna napsaná v jazyce *C*

– je kompatibilní s většinou operačních systémů a kompilátorů jazyka *C*

– název je zkratka z *Perl Compatible Regular Expressions*

– obsahuje téměř kompletní podporu regulárních výrazů jaká je v *Perlu* verze 5

Perl

– nativní podpora regulárních výrazů

– výhody – mnohostrannost, nejširší sortiment regulárních výrazů, řada rozšiřujících knihoven, open source, standard pro tvorbu *CGI* skriptů

– verze pro *Unix*, *Windows* a *Macintosh*

PHP

– dvě sady funkcí pro práci s regulárními výrazy, **ereg** a **preg**

– **ereg** – standardní součást všech verzí *PHP*, podporuje pouze základní sadu regulárních výrazů

– **preg** – vychází z *PCRE*, nabízí tedy kompletní sadu regulárních výrazů, vyžaduje však, aby na serveru byla nainstalována knihovna *PCRE* a *PHP* zkompilované s podporou této knihovny

Python

– regulární výrazy obsahuje vestavěný modul **re**

Ruby

– nativní podpora, regulární výrazy jsou kompatibilní s *Perlem*

Visual Basic 6

– regulární výrazy doplní knihovna **VBScript**, ta je přítomná na každém počítači se systémem *Windows* a internetovým prohlížečem *IE 5.5* a novějším

7. PROGRAMY PRO TVORBU A TESTOVÁNÍ REGULÁRNÍCH VÝRAZŮ

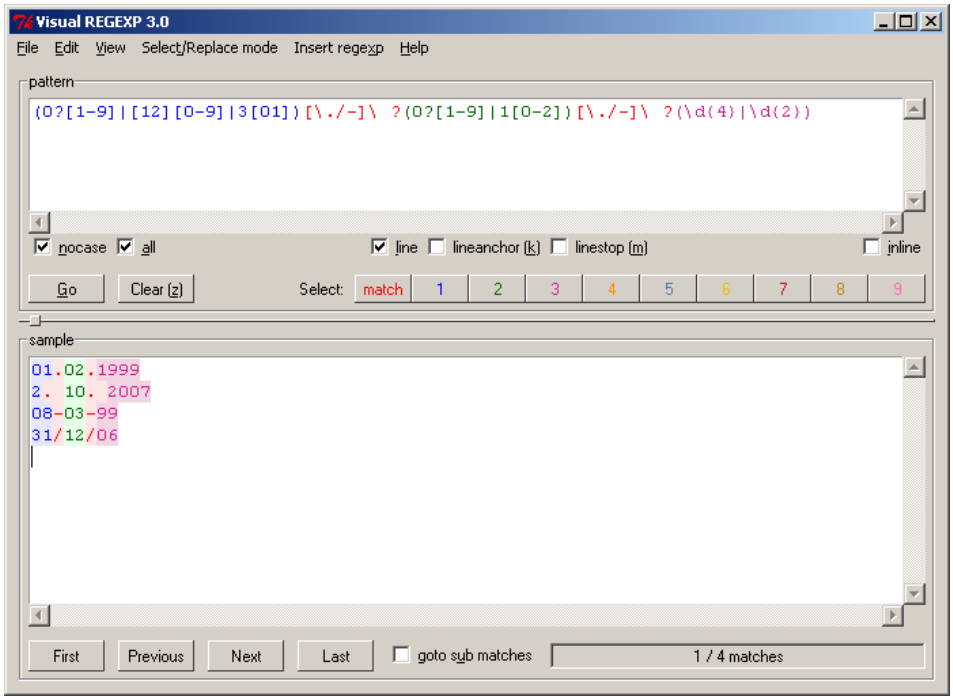
Vytvořit komplexní regulární výraz je obtížné a poznat jak nějaký již hotový výraz pracuje je často ještě obtížnější. Existují však speciální programy, které slouží k ladění a testování regulárních výrazů. Dokážou testovaný výraz a text barevně zvýraznit podle jednotlivých částí výrazu, umožňují testovat nahrazování v řetězci a dokonce také krokovat vyhodnocování výrazu.

Visual REGEXP

Tento tester regulárních výrazů je šířen jako freeware. K dispozici je pouze pro operační systém *Windows*. Má velice dobře provedené zvýraznění výrazu a textu na který je aplikován. Nabízí také několik předdefinovaných výrazů, například pro html tagy, emailovou adresu, IP adresu a další. Bohužel má nepřehledně provedené nahrazování a nenabízí žádné další věci jako je krokování nebo náhled rozkladu textu podle použitého výrazu. Obrázek 1 zobrazuje program Visual REGEXP při řešení regulárního výrazu pro datum.

The Regex Coach

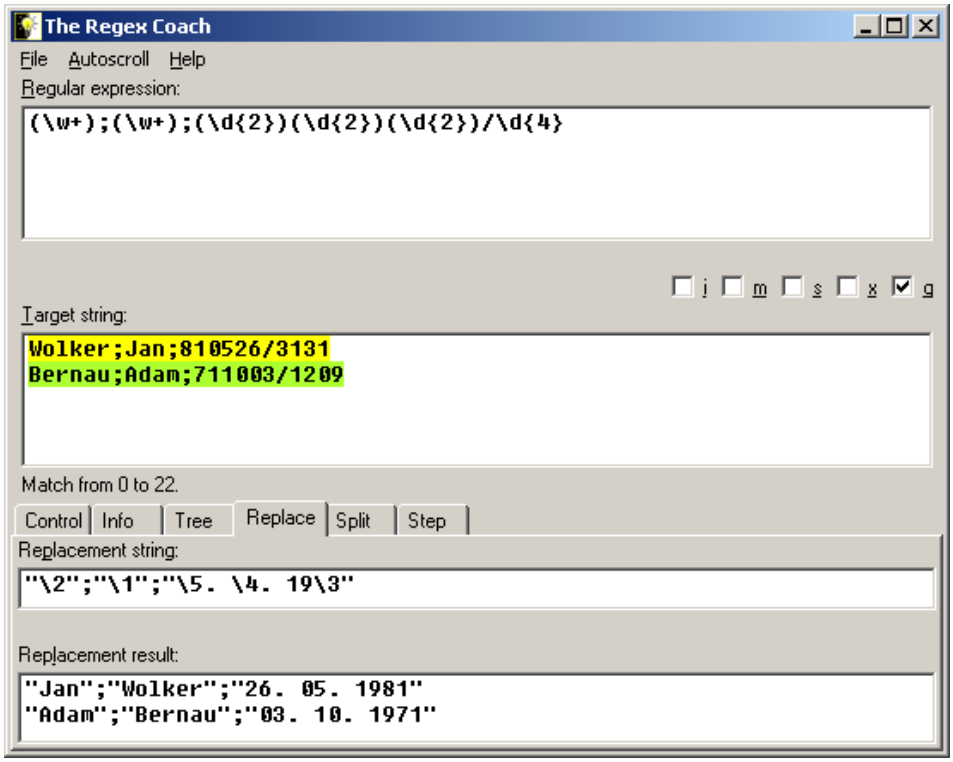
Další freewarový tester, také pouze pro Windows. Oproti *Visual REGEXP* nemá tak propracované zvýrazňování, nabízí však jiné výhody. Jednou je to, že změny na testovaném textu se projevují automaticky po každé změně v regulárním výrazu. Má také velmi vydařené nahrazování, ukázka nahrazovaného textu se také mění s každou úpravou nahrazovacího výrazu. Dokáže také graficky zobrazit regulární výraz rozložený na jednotlivé podvýrazy. Další užitečná věc je krokování s postupným zvýrazňováním textu, což velmi usnadní pochopení regulárního výrazu. Obrázek 2 ukazuje řešení příkladu z kapitoly *Zapamatování* v programu The Regex Coach včetně požadovaného nahrazení. Rozklad použitého výrazu na podvýrazy ukazuje Obrázek 3.



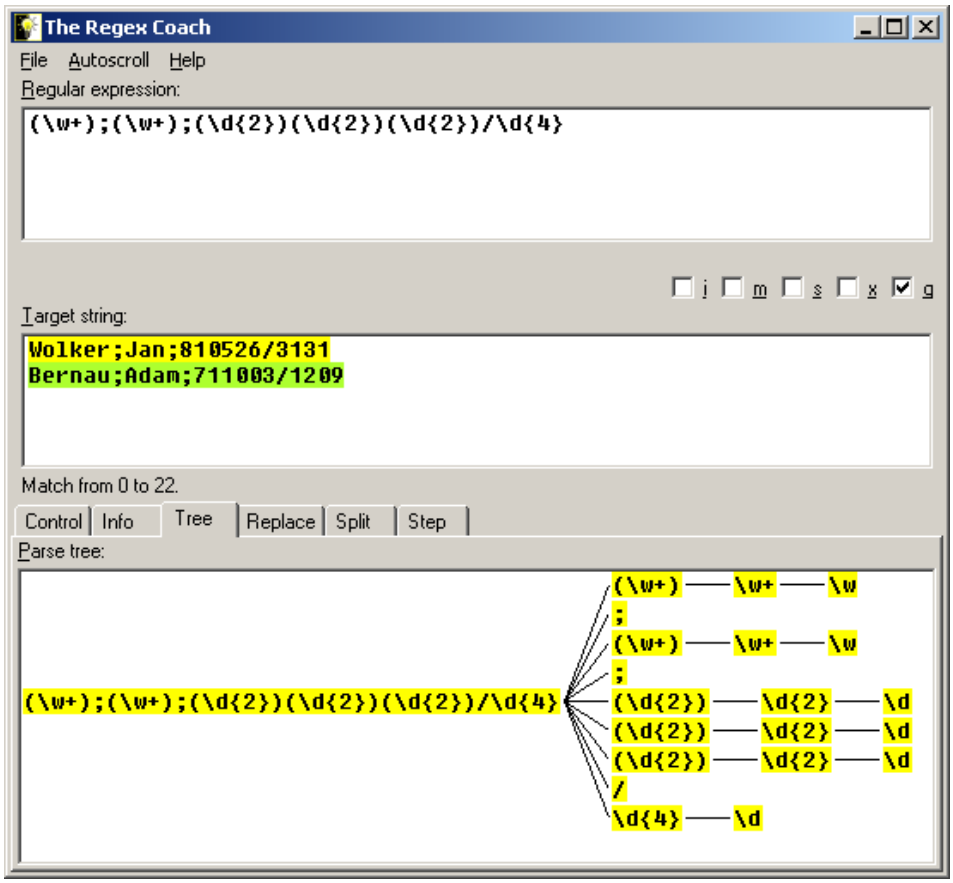
Obrázek 1: Visual REGEXP

8. DATABÁZE REGULÁRNÍCH VÝRAZŮ

Pokud potřebujeme rychle použít nějaký regulární výraz tak není nutné ho vždy vymýšlet, na internetu existuje řada databází kde najdeme již hotové výrazy. Mezi nejobsáhlejší patří databáze *RegExLib.com*, která se nachází na adrese http://regexlib.com. Tato databáze obsahuje více než 1600 regulárních výrazů rozdělených do osmi kategorií. Regulární výrazy je možné vyhledat podle klíčových slov, mezi další kritéria patří právě kategorie výrazu nebo hodnocení kvality. Každý výraz obsahuje popis, vyhovující i nevyhovující příklady, jméno autora, hodnocení a komentáře uživatelů. Výraz je také možné okamžitě otestovat přímo v prohlížeči zadáním libovolného řetězce.



Obrázek 2: The Regex Coach - nahrazování



Obrázek 3: The Regex Coach - rozklad výrazu

9. PROGRAM NA OPRAVU ÚDAJŮ V DATABÁZI

Při sběru dat od většího množství uživatelů je nutné počítat s nejednotným formátem zápisu, protože každý z nich může různé typy dat zapisovat různými způsoby. Například PSČ je možné zapsat jako sled pěti číslic nebo jako trojici a dvojici číslic oddělených mezerou. V získaných datech se také můžou vyskytovat různé chyby nebo nežádoucí kombinace typů dat, například pokud bude u telefonního čísla zapsané také jméno. Před importem do databáze SQL je vhodné sjednotit formát a rozdělit data na základní typy.

Pro zautomatizování této činnosti jsem vytvořil program *Piecemaker* (kouskovač). Ten zpracovává databázové soubory uložené v textové podobě ve formátu *csv* a na základě klíčových slov v hlavičce aplikuje jednotlivé regulární výrazy na úpravu dat ve sloupcích označených klíčovým slovem. Program je napsaný v jazyce *Perl*, pro přehlednost je kód rozdělen do množství krátkých funkcí, které jsou postupně volány z hlavní části programu. V následující části popíšu ovládání a práci programu, jednotlivé funkce programu a vysvětlím použité regulární výrazy.

Ovládání programu

Program se spouští z příkazové řádky, bez zadání parametrů nebo s přepínačem `-h` se na obrazovku vypíše nápověda. Jinak očekává dva povinné parametry, prvním z nich je název souboru, který se bude zpracovávat. Soubor musí být uložen v podobě formátovaného textu, jako oddělovač záznamů může sloužit libovolný znak. Jako druhý parametr musí být programu předán znak, který byl použit právě jako oddělovač záznamů ve vstupním souboru. Poté již program zpracuje celý vstupní soubor a po skončení vypíše název nového vygenerovaného souboru. Nový soubor obsahuje původní i vyčištěná data aby bylo možné porovnat výsledek čištění. Data, která chceme vyčistit označíme již ve vstupním souboru zapsáním jednoho z klíčových slov do hlavičky a označíme tak sloupce, které program zpracuje. Mezi klíčová slova patří `FULLNAME, PHONE,`

`ZIPCODE`, `CITY`, `ADDRESS`, `EMAIL`, `DATE`, `URL`. Jejich význam je upřesněn v dalších kapitolách nebo je popsán v nápovědě programu.

Program je možné spustit jak v operačním systému Linux tak v operačním systému Windows, pokud je v nich nainstalovaný *Perl* (ve Windows *ActivePerl*). Pro Windows je případně k dispozici také zkompilovaný exe soubor.

Příklady spuštění programu:

- `piecemaker.pl "databaze.csv" "|"`
- `piecemaker.exe "databaze2.csv" ";"`

Kontrola vstupních parametrů

Program očekává zadání dvou vstupních parametrů, jedním je jméno vstupního souboru a druhým je oddělovač sloupců v souboru. Zda byly parametry správně zadány ověří funkce `check_arguments`, která při úspěchu vrací zadané parametry a při neúspěchu ukončí program a vypíše nápovědu a chybové hlášení.

Otevření vstupního souboru

O otevření vstupního souboru a jeho zamčení pro čtení se stará funkce `open_source_file`, která jako parametr obdrží jméno souboru. Kontroluje také existenci vstupního souboru, v případě neúspěchu ukončí program a vypíše chybové hlášení.

Otevření cílového souboru

Funkce `open_destination_file` dostane jako parametr jméno vstupního souboru a z něho odvodí jméno cílového souboru. Na vstupní soubor použije regulární výraz `(.*)(\..*)`, ten zjistí zda jeho jméno obsahuje tečku a příponu. Pokud ano, sestaví jméno cílového souboru z původního vložením řetězce `_OK` mezi jméno a příponu. Pokud vstupní soubor příponu nemá tak se za jeho jméno pouze připojí řetězec `_OK`. Nakonec se ověří, zda takový soubor již neexistuje, aby se zabráni-

lo nechtěnému přepsání. Pokud existuje, tak do názvu souboru ještě přidáme jedinečný identifikátor generovaný systémem z čísla běžícího procesu, v Perlu ho získáme použitím proměnné `$$`. Poté cílový soubor otevřeme a zamkneme pro zápis.

Čtení vstupního souboru

Funkce `read_source_file` načte celý vstupní soubor do paměti do proměnné `@source_data`, což je pole ve kterém každá položka obsahuje jeden řádek vstupního souboru. Funkce také kontroluje zda vstupní soubor není prázdný, v tomto případě ukončí program chybovou hláškou.

Čištění dat

Funkce `clean_data` je nejobsáhlejší funkcí programu. Řídí zpracování zdrojových dat a generování nového výstupního souboru. Nejprve vyjme ze zdrojových dat hlavičku, tu rozdělí podle oddělovače záznamů do pole `@header`. Každá položka pole reprezentuje název jednoho sloupce ve vstupním souboru. Poté funkce v cyklu projde všechny řádky vstupních dat. Každý zpracovávaný řádek si rozdělí podle oddělovače záznamů do pole `@column`. V dalším vnořeném cyklu porovná každý ze sloupců hlavičky s definovanými klíčovými slovy a při úspěchu zavolá čistící funkci na takový sloupeček, který má v poli vstupních dat stejný index jako v poli s hlavičkou. Jednotlivé čistící funkce budou popsány v dalších kapitolách. Klíčová slova, která označují sloupce vstupního souboru pro čištění, jsou:

- `FULLNAME` – hledá titul před a za jménem, křestní jméno a příjmení
- `PHONE` – hledá telefonní čísla
- `ZIPCODE` – hledá poštovní směrovací číslo
- `CITY` – hledá jméno města, to může obsahovat i číslo městského obvodu
- `ADDRESS` – hledá jméno ulice následované číslem popisným,

následovat může i číslo orientační

- `EMAIL` – hledá řetězec vyhovující zápisu e-mailové adresy
- `DATE` – hledá den, měsíc a rok
- `URL` – hledá řetězec odpovídající adrese URL, akceptuje i číselný zápis IP adresy

Vyčištěná data jsou vracena funkcemi v podobě textových řetězců, ty jsou ukládány do polí. Každý typ vyčištěných dat má svoje vlastní pole. Když je zpracováno celé vstupní pole tak se sestaví nová hlavička pro výstupní soubor, ta má tolik nových sloupců kolik je maximální počet vyčištěných dat na řádek. Nová hlavička se zapíše do výstupního souboru a následuje zápis vyčištěných dat. Ta obsahují i původní sloupce aby bylo možné porovnat výsledek čištění. O sestavení každého řádku se stará funkce `make_row`, ta dostane jako parametry původní a vyčištěná data a jejich počet. Vrácený řetězec je zapsán jako nový řádek do výstupního souboru.

Čištění jmen

Volaná funkce se jmenuje `clean_fullname`. Při zpracování údajů obsahujících jméno, příjmení a titul nejprve najdu všechny tituly, uložím si je do další proměnné a poté je z původního řetězce odstraním. Předpokládám, že zbývající část řetězce již obsahuje pouze jméno a příjmení. Na rozdělení jména od příjmení použiji regulární výraz `([^\s\d,.-]+) ([^\d,.-]+)`. Ten se sestává ze dvou subvýrazů, prvnímu z nich vyhoví právě jedno slovo, druhému více slov oddělených mezerou.

Pokud je řetězec `Miloš Kukla`, pak odpovídá

- 1. subvýrazu řetězec `Miloš`
- 2. subvýrazu řetězec `Kukla`

Pokud je řetězec `Radek Alexandr Dimitrov`, pak odpovídá

- 1. subvýrazu řetězec `Radek`

- 2. subvýrazu řetězec `Alexandr Dimitrov`

Čištění telefonních čísel

Pro zpracování telefonních čísel se volá funkce `clean_phone`. Regulární výraz `\+?(?:420)?[ \/-]?([1-9]\d{2})[ \/-]?(\d{3})[ \/-]?(\d{3})` se snaží pokrýt co nejvíce možných forem zápisu českého telefonního čísla. Součástí čísla mohou být mezery, závorky, lomítka, může ale nemusí obsahovat mezinárodní předvolbu. Výraz si zapamatuje pouze tři trojice čísel bez mezinárodní předvolby a složí je do výsledného řetězce o devíti číslicích bez oddělovačů. Telefonních čísel může být v jednom sloupci zapsáno více, každé nalezené z řetězce vyjmu a hledám další. Pokud již žádné nenajdu tak cyklus ukončím. Možné kombinace zpracovatelných zápisů telefonního čísla ukážu na následujících příkladech. Barevně vyznačené části označují zapamatované části řetězce.

- `+420774369544`
- `495-522-343`
- `775 234 444, 466 282 498`
- `+420 777 233 685; reklamace`
- `obchod - 466-756-122,servis - 466-756-123`
- `604 / 43 67 23` – tento zápis nevyhoví regulárnímu výrazu

Čištění poštovních směrovacích čísel

Poštovní směrovací čísla upravuji funkcí `clean_zipcode`. Předpokládám, že řetězec obsahuje pouze jedno PSČ, to může být zapsané jako souvislá pětice číslic nebo trojice a dvojice oddělená mezerou, pomlčkou nebo podtržítkem. Výsledný řetězec zapisuji jako souvislou pětici číslic. Používám regulární výraz `(\d{3})[ _-]?(\d{2})`.

Možné zápisy PSČ mohou vypadat takto (barevně vyznačené části označují zapamatované části řetězce):

- `530 12 - Pardubice, Dubina`

- `Pardubice, 53012`

Čištění názvů měst

Volaná funkce má název `clean_city`. Předpokládám, že ve zkoumaném řetězci se může kromě názvu města vyskytovat také PSČ, proto nejprve pomocí výše uvedeného regulárního výrazu vyhledám a odstraním případné PSČ. Regulární výraz pro název města má podobu `([^0-9,-]+[^\s\d,.-])[ ,.-]*([1-9][0-9]*)?[ ,.-]*([\w ,.-]+[^\s\d,.-])?`. Snaží se najít a zapamatovat jeden až tři údaje – název města, číslo městského obvodu a název městské části. Tyto údaje mohou být odděleny různou kombinací mezer, čárek, pomlček a teček. Podle počtu nalezených a zapamatovaných řetězců se vyhodnotí podoba výsledného řetězce. Můžou nastat čtyři kombinace, podle kterých se výsledný řetězec sestaví následovně:

1. `&trim($1)`
2. `&trim($1)." - ".&trim($3)`
3. `&trim($1)." ".&trim($2)`
4. `&trim($1)." ".&trim($2)." - ".&trim($3)`

Volání funkce `trim` ořeze z jednotlivých subřetězců mezery na začátku a na konci, barevné zvýraznění označuje zapamatovanou část řetězce jak odpovídá jednotlivým subvýrazům v následujících příkladech:

1. zdroj: `Bystřice pod Hostýnem`
   cíl: `Bystřice pod Hostýnem`
2. zdroj: `Uherské Hradiště-Mařatice`
   cíl: `Uherské Hradiště - Mařatice`
3. zdroj: `Praha 7`
   cíl: `Praha 7`
4. zdroj: `Praha 9, Černý Most`
   cíl: `Praha 9 - Černý Most`

Čištění adres

Funkce se jmenuje `clean_address`. Hledá zápis jména ulice následovaný číslem popisným, případně i číslem orientačním, regulární výraz se proto skládá ze tří subvýrazů. První slouží pro nalezení a zapamatování jména ulice, druhý pro získání čísla popisného a třetí získá číslo orientační pokud je přítomno. Celý regulární výraz má tvar `(.+) (?:([1-9][0-9]*)\/)?([1-9][0-9]*[a-cA-C]?)`. V následujících příkladech jsou barevně znázorněny jednotlivé části regulárního výrazu, které funkce vrací jako vyčištěná data:

- Makovského náměstí 2
- Nedbalova 18/467
- Varnsdorfská 339/10 Střižkov

Čištění emailových adres

Emailové adresy kontroluje funkce `clean_email`. Vstupní řetězec může obsahovat více adres, zpracovávány jsou postupně všechny. Ze vstupního řetězce je vždy vyjmut právě jeden podřetězec, který tvarem odpovídá zápisu emailové adresy. Ten je považován za vyčištěný a celý postup se pomocí while cyklu opakuje dokud je ve vstupním řetězci obsažena alespoň jedna emailová adresa. Pro vyhledání emailové adresy v řetězci používám regulární vyraz `([A-Za-z0-9_.-]{2,})@([A-Za-z0-9-]{2,62})\.([A-Za-z]{2,4})`. Skládá se ze tří částí, část před zavináčem může být složena z velkých a malých písmen anglické abecedy, čísel, podtržítka, tečky a pomlčky. Také musí mít minimální délku dva znaky. Část za zavináčem se skládá z doménového jména a přípony. Doménové jméno může z nealfanumerických znaků obsahovat pouze pomlčku a jeho délka může být maximálně 64 znaků včetně přípony (proto je v regulárním výrazu maximální délka 62, počítám s příponou o minimálně dvou znacích). Samotná přípona se může skládat pouze z písmen anglické abecedy a její délka může být od dvou do čtyř písmen.

Čištění data

Funkce pro zpracování data má název `clean_date`. Předpokládám, že vstupní údaj je zapsán v pořadí den, měsíc a rok. Rok může být zapsán dvěma i čtyřmi číslicemi. Jako oddělovač údajů může sloužit tečka, lomítko, podtržítko nebo pomlčka, vše může být nepovinně následováno mezerou. Pro výstupní formát používám jako oddělovač tečku následovanou mezerou. Použitý regulární výraz má tvar `(0?[1-9]|[12][0-9]|3[01])[.\/_-] ?(0?[1-9]|1[0-2])[.\/_-] ?((19|20)?[0-9]{2})`. Jako den může být zapsán pouze existující den v měsíci, tj. od 1 do 31. Číslům menším než 10 může předcházet nula. To samé platí pro zápis měsíce, zde jsou povolené hodnoty od 1 do 12. Pokud je letopočet zapsán jako čtveřice, tak první dvojice může být tvořena pouze čísly 19 a 20. Dále uvedu několik příkladů, barevně vyznačené části odpovídají zapamatovaným částem regulárního výrazu.

- `13.05.1997`
- `24. 1. 2002`
- `12/02/99`
- `04-09-1987`
- `12-13-99` – nevyhoví regulárnímu výrazu

Čištění URL

Pro kontrolu internetových adres slouží funkce `clean_url`. Použitý regulární výraz akceptuje zápis v podobě doménového jména i číselné IP adresy, zde navíc kontroluje zda není použit nepřípustný rozsah. Adrese mohou předcházet protokoly http, https a ftp. Regulární výraz má podobu `((http|https|ftp)\:\/\/)?([a-zA-Z0-9\.\-]+(\:[a-zA-Z0-9\.&%\$\-]+)*@)?((25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-`

```
5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|
[0-9])|([a-zA-Z0-9\-]+\.)*[a-zA-Z0-9\-]+\.[a-zA-
Z]{2,4})(\:[0-9]+)?(\/[^\/][a-zA-Z0-
9\.\,\?\'\\\/\+&%\$#\=~_\-@]*)*).
```

Níže uvádím několik zápisů, které vyhovují použitému regulárnímu výrazu.

- `www.google.com`
- `ftp.czilla.cz`
- `https://student.upce.cz/webmail`
- `http://212.80.112.54/search.php?name=Milos`

Sestavení nové hlavičky

Funkce `make_header` dostane jako parametry původní hlavičku a počet všech typů dat, které se podařilo vyčistit. Novou hlavičku sestaví z původní hlavičky ke které připojí názvy nových sloupců podle typu a počtu vyčištěných dat. Název nového sloupce se skládá z klíčového slova, indexu a řetězce „_OK“ (např. `PHONE1_OK`). Jako výsledek vrací řetězec, který je poté zapsán jako první řádek do výstupního souboru.

Sestavení nového řádku dat

Funkce `make_row` dostane jako parametry původní data, všechna nová data získaná čištěním a navíc maximální počet vyčištěných dat od každého typu. To proto, aby všechny řádky cílového souboru měly stejný počet sloupců bez ohledu na to, kolik vyčištěných údajů se podařilo získat. Funkce vrací jeden řádek vyčištěných dat, ten je zapsán do cílového souboru.

Zápis do cílového souboru

Funkce write_destination_file přebírá jako parametr řetězec, který zapíše na konec cílového souboru.

Zavření cílového souboru

Funkce `close_destination_file` odemkne a zavře cílový soubor.

Zavření vstupního souboru

Funkce `close_source_file` odemkne a zavře zdrojový soubor.

Ořezání mezer na začátku a na konci řetězce

Protože Perl neobsahuje předefinovanou funkci pro ořezání mezer na začátku a na konci řetězce musel jsem napsat vlastní funkci `trim`. Ta na řetězec, který ji byl předán jako parametr použije dvě nahrazení pomocí regulárních výrazů. První (`s/^\s+//`) ořeže bílé znaky na začátku řetězce, druhé (`s/\s+$//`) ořeže bílé znaky na konci řetězce. Funkce vrací upravený řetězec.

Vypsání nápovědy

Pokud programu nezadáme žádný parametr nebo zadáme parametr `-h` nebo nastane nějaká chyba tak je pomocí funkce `show_help` vypsána nápověda jak program používat.

Čištění dat při jejich zadávání

Popisovaný program čistí data až po jejich sesbírání. Druhá možnost čištění dat je kontrola jejich správnosti už při zadávání uživatelem. Pro tento účel je po menší úpravě možné program použít. Všechny čistící funkce zůstanou tak jak jsou, jediná změna proběhne ve funkci `clean_data`. Místo aby získávala data z textového souboru tak je získá z aplikace se kterou pracuje uživatel a nebude je ukládat do nového souboru ale rovnou do cílové databáze. Toto využití by bylo vhodné především při sběru dat z webových formulářů. Na webu máme totiž jen málo možností kontroly, *JavaScript* sice podporuje regulární výrazy, nemají ho však všechny prohlížeče a navíc si ho uživatel může libovolně vypnout.

Vzorová databáze

Na ukázku funkčnosti jsem připravil vzorovou databázi s nevhodně zapsanými daty a zpracoval ji programem. Obrázek 4 obsahuje data před čištěním. Obrázek 5 zobrazuje tato data po zpracování programem.

	A	B	C	D
1	FULLNAME	ADDRESS	CITY_ZIPCODE	PHONE
2	Irena Zítková	Hlavní 107	702 00 Ostrava	596513248,605273451,
3	Dana Černíková	Bránická 15/111	500 06 Hradec Králové	585220897
4	Ivo Pánek	U pohádky 400	625 00 Brno	585417590
5	Ing.Ivana Kubáková	Vrchlického sad 2	382 11 Větřní	553658551
6	Pavel Vybíral	Radiová 5	709 00 Ostrava-Mariánské hory	553657251
7	Jana Lieberzeitová	Fugnerovo nábr. 1796	669 02 Znojmo	úč.p.Adamcová606114787,,
8	Miroslav Sloup	Nový Šaldorf - Sedlešovice 126	602 00 Brno	728803842
9	Alena Všetečková	Rozdrojovice 71/e	702 00 Ostrava	605153569maminka
10	Drahomíra Šorfová, Judr.	Podolí 400	503 15 Hradec Králové	724191484
11	Pavel Cupák	Dominikánské náměstí 5	466 01 Jablonec nad Nisou	257322593,605451
12	Jaroslav Boháč	Nám. Přemysla Ot.II. 50/14	500 04 Hradec Králové - Kukleny	543251077

Obrázek 4: Vzorová databáze

	E	F	G	H	I	J	K	L	M	N
1	TITLE_BEFORE1_OK	FIRST_NAME1_OK	LAST_NAME1_OK	PHONE1_OK	PHONE2_OK	ZIPCODE1_OK	CITY1_OK	STREET1_OK	HOUSE_NO1_OK	HOUSE_NO_LOC1_OK
2		Irena	Zítková	596513248	605273451	70200	Ostrava	Hlavní	107	
3		Dana	Černíková	585220897		50006	Hradec Králové	Bránická	15	111
4		Ivo	Pánek	585417590		62500	Brno	U pohádky	400	
5	Ing.	Ivana	Kubáková			38211	Větřní	Vrchlického sad	2	
6		Pavel	Vybíral	553657251		70900	Ostrava - Mariánské hory	Radiová	5	
7		Jana	Lieberzeitová	606114787		66902	Znojmo	Fugnerovo nábr.	1796	
8		Miroslav	Sloup	728803842		60200	Brno	Nový Šaldorf - Sedlešovice	126	
9		Alena	Všetečková	605153569		70200	Ostrava	Rozdrojovice	71	
10	Judr.	Drahomíra	Šorfová	724191484		50315	Hradec Králové	Podolí	400	
11		Pavel	Cupák	257322593		46601	Jablonec nad Nisou	Dominikánské náměstí	5	
12		Jaroslav	Boháč	543251077		50004	Hradec Králové - Kukleny	Nám. Přemysla Ot.II.	50	14

Obrázek 5: Vzorová databáze po čištění

10. ZÁVĚR

Určité povědomí o regulárních výrazech jsem měl dříve, než jsem se rozhodl napsat na toto téma bakalářskou práci. Věděl jsem, že se s nimi setkám především v prostředí linuxu, v Perlu a ve speciálních nástrojích pro práci s textem jako jsou AWK nebo grep. Také jsem dokázal sestavit jednoduchý výraz, např. pro odstranění opakujících se mezer nebo pro rozeznání PSČ. Neměl jsem však dostatek znalostí a zkušeností abych věděl, že je možné využívat regulární výrazy i při rutinní činnosti, např. editaci zdrojového kódu v obyčejném textovém editoru. Jak jsem začal sbírat informace pro bakalářskou práci a zkoušel sestavovat různé regulární výrazy, zjistil jsem jak široká je možnost jejich použití. V současné době používám textový editor *Notepad++*, který umožňuje při nahrazování používat regulární výrazy a běžně si usnadňuji práci při psaní procedur v SQL nebo při generování nových loginů pro uživatele databáze. Další činnost, kterou je možné efektivně řešit pomocí regulárních výrazů, spočívá v převodu údajů z databáze v Excelu do druhé normální formy. Je to z důvodu importu do databáze SQL. Bez použití regulárních výrazů mi tato činnost zabrala běžně kolem dvou hodin, proto jsem se snažil najít způsob jak toto zautomatizovat. Využil jsem jednotlivé regulární výrazy pro zpracování vždy určitého typu dat a spojil je pomocí Perlu do jediného programu, který je popisovaný v této práci. S jeho pomocí dokážu zkrátit dobu zpracování jedné databáze na několik málo minut, které jsou potřeba ke kontrole vygenerovaných očištěných dat.

Další možnosti pro využití regulárních výrazů vidím u webových aplikací. Nejde jen o kontrolu údajů zadávaných do jednoduchých formulářů, jako např. správná syntaxe emailové adresy. Pomocí regulárních výrazů je možné také opravit pravopis při zadávání celých článků do různých redakčních systémů nebo kontrovat obsah příspěvků v diskusních fórech.

Znalosti získané při tvorbě bakalářské práce mám v úmyslu dále rozšiřovat, především mě zajímá programovací nástroj Perl pro jeho rozsáhlé možnosti při zpracování textů a tvorbě webových aplikací.

SEZNAM POUŽITÉ LITERATURY

1. *AWK*, URL: http://cs.wikipedia.org/wiki/AWK.
2. *GRETA: The GRETA Regular Expression Template Archive*, URL: http://research.microsoft.com/projects/greta/gretausergui-de.htm
3. *History of QED*, URL: http://plan9.bell-labs.com/who/dmr/qed.html.
4. *Perl regular expressions*, URL: http://www.perl.com/doc/manual/html/pod/perlre.html.
5. *Regular Expression Library*, URL: http://regexlib.com/.
6. *Regular expressions history*, URL: http://en.wikipedia.org/wiki/Regular_expression#History.
7. *Regular-Expressions.info*, URL: http://www.regular-expressi-ons.info/.
8. *Regulární výrazy*, URL: http://www.regularnivyrazy.info/.
9. SATRAPA, P.: *Perl pro zelenáče* 2. vydání, Praha: Neocortex, 2001. ISBN 80-86330-02-8.
10. SATRAPA, P.: *Regulární výrazy*, URL: http://www.kai.tul.cz/~satrapa/docs/regvyr/.

ÚDAJE PRO KNIHOVNICKOU DATABÁZI

Název práce	Využití regulárních výrazů při opravách textu, úpravách a generování souborů
Autor práce	Miloš Kukla
Obor	Informační technologie
Rok obhajoby	2007
Vedoucí práce	RNDr. Josef Rak
Anotace	Cílem práce bylo zjistit možnosti využití regulárních výrazů, naučit se tvořit vlastní regulární výrazy a tyto znalosti aplikovat při řešení konkrétních problémů. První část práce vysvětluje syntaxi výrazů a možnosti jejich využití. Druhá část popisuje strukturu a funkčnost programu, který využívá regulární výrazy pro selekci a opravu kontaktních údajů osob a firem v databázích ve formátu csv.
Klíčová slova	Regulární výraz, regular expression, regexp, Perl, grep, metaznak, oprava textu