Kočírujeme smršť experimentů aneb Unix pro (mírně) programující lingvisty

Ondřej Bojar
bojar@ufal.mff.cuni.cz

25. leden 2006

Přehled

- Motivace a metodika
- Výhody řádkového rozhraní (bash)
- Unix je nástroj na práci s textovými soubory (textutils)
- Make pro lepší přehled procesech výroby souborů
- CVS pro lepší přehled o historii i lepší pořádek v současnosti

Motivace a metodika

Proč se učit něco nového:

- automat šetří čas (naprogramuju jednou, použiju mockrát)
- program je dobrá dokumentace (mám rovnou pečlivě zapsáno, jak jsem pokus prováděl)

Jak se učit Unix:

- po kapkách, praxí, najednou je toho nezvládnutelně moc
- postupným obohacováním repertoáru známých příkazů a přepínačů

Výhody řádkového rozhraní (bash)

- příkazy mohou být přesnější a pestřejší než příkazy udílené myší
- snadno srozumitelné pro počítač $\Rightarrow$ snadno strojově zpracovatelné
  - pohodlně reprodukovatelné pouhým copy-paste
  - pohodlně zobecnitelné (nahraď konkrétní jména proměnnými)
- plná síla programovacích jazyků (větvení, cykly)
- pohodlné propojování programů, navazující výpočty

bash = Bourne-Again Shell

$\Rightarrow$ **man bash**

Základem Unixu je roura

cat defined-vallex-frames.txt

```
absolvovat  0  0  1  phase verb  absolvovat-1  ACTobl(#nom) PATobl(#aku)
absolvovat  0  0  2  -  absolvovat-2  ACTobl(#nom) PATobl(#aku)
akceptovat  0  0  1  -  akceptovat-1  ACTobl(#nom) PATobl(#aku,ze)
```

. . .

cat defined-vallex-frames.txt | head -2

```
absolvovat  0  0  1  phase verb  absolvovat-1  ACTobl(#nom) PATobl(#aku)
absolvovat  0  0  2  -  absolvovat-2  ACTobl(#nom) PATobl(#aku)
```

cat defined-vallex-frames.txt | cut -f 1

```
absolvovat
absolvovat
akceptovat
```

Ještě roura a ještě roura a přesměrování

cat defined-vallex-frames.txt | cut -f 1 | sort | uniq -c

```
      2 absolvovat
      1 akceptovat
      1 analyzovat
```

Přesměrování na standardní vstup programu a do souboru:
cat vstup | head a **cat vstup > head**

Přesměrování obecně:
program < standardní-vstup > standardní-výstup 2> chybový-výstup
program < standardní-vstup >> prodloužení-existujícího-souboru
program1 2> &1 | program2-dostane-na-vstup-spojený-std-a-chyb-výstup
program1 `program2-výstup-vygeneruje-přímo-na-příkazovou-řádku`

Náš první skript

- Z užitečného úseku roury si udělej skript:
  echo "sort | uniq -c" > suc
- Nastav mu příznak spustitelnosti:
  chmod +x suc
- Umísti skript do svého (nově vyrobeného) osobního adresáře skriptů:
  mkdir -p ~/bin; mv suc ~/bin
- Řekni bashi, ať programy (skripty) hledá *přednostně* v tom osobním adresáři:
  export PATH=~/bin:$PATH

cat defined-vallex-frames.txt | cut -f 1 | suc | head -2

```
2 absolvovat
1 akceptovat
```

Proměnné v bashi a v prostředí

- Nastavit proměnnou pouze v bashi:
  pozdrav=ahoj
- Nastavit proměnnou v bashi a prostředí pro všechny *dále* spuštěné programy:
  export pozdrav=ahoj

Otestujme to:

```
echo 'echo "Zdravim: $pozdrav"' > test-pozdravu
cat test-pozdravu
echo "Zdravim:  $pozdrav"
sh test-pozdravu
Zdravim:
pozdrav=ahoj; sh test-pozdravu
Zdravim:
export pozdrav=nazdar; sh test-pozdravu
Zdravim:  nazdar
```

Souhrn operátorů pro řazení programů v bashi

pgm1 | pgm2	výstup pgm1 bude předán jako vstup do pgm2
pgm1 ; pgm2	po ukončení pgm1 bude spuštěn (nezávisle) pgm2, oběma lze přisměrovat vstup, má stejný význam jako nový řádek ve skriptu
pgm1 && pgm2	pokud spuštění pgm1 skončí úspěchem, bude spuštěn i pgm2
pgm1 || pgm2	pokud pgm1 skončí neúspěchem, bude spuštěn pgm2

Kulaté závorky znamenají: spusť sekvenci v podshellu.

(cat vstup1.txt ; zcat vstup2.txt.gz) | sem-potečou-data-z-obou-souborů

Unix je nástroj pro práci s texty

- Přehled základních užitečných programů:
  ⇒ **info coreutil**

`cat tac nl wc`	výpis, obrácený výpis, číslování a počítání řádek
`less`	prohlížení dlouhého výpisu: **cat dlouhy | less**
`cut paste join`	vyřezávání sloupců a spojování souborů po sloupcích
`tee`	rozvětvení roury: **cat vstup | tee stranou | . . .**
`grep`	výběr zajímavých řádek ze vstupu
`sed tr awk`	jednodušší i složitější úpravy textů

- Některé užitečnosti ale chybí:
  ⇒ `http://www.cuni.cz/~obo/textutils`
  ⇒ **ls /home/bojar/tools/{vimtext,shell}/**
- A nezapomeňte si vyrábět vaše vlastní!

Make: recept, co z čeho vzniká

⇒ **info make**

- Make je původem nástroj pro programátory: z jakých zdrojových souborů kompilovat pomocí jakých kompilátorů jaké programy.
- Make pro lingvisty poslouží stejně: z jakých dat jakými postupy generovat jaké přehledy, souhrny či jiná data.

Pravidla zapisujte do souboru `Makefile`. Struktura pravidla pro make:

```
cilovy-soubor: zdrojovy-soubor1 zdrojovy-soubor2
    prikaz1 < zdrojovy-soubor1 > /tmp/pomocny-soubor
    prikaz2 zdrojovy-soubor2 < /tmp/pomocny-soubor < cilovy-soubor
```

↑ tabulátor, nikoli mezery!

Každý řádek spuštěn v novém shellu ⇒ nelze předat proměnné, ani přes prostředí, pokud řádky nespojím.

Příklad: Makefile pro počítání frekvencí

Takto vypadá připravený Makefile
cat Makefile

```
entries.freq: defined-vallex-frames.txt
    cat $< | suc > $@
```

Takto jej použijeme – necháme vyrobit výstupní soubor a prohlédneme si ho:
make entries.freq
head -2 entries.freq

```
2 absolvovat
1 akceptovat
```

Po změně vstupů:
touch defined-vallex-frames.txt
příkaz **make entries.freq** přegeneruje i výstupy, bez změn by nedělal nic.

Proměnné v pravidlech

Pravidlo pracující pro více různých vstupů:

```
entries-version-%.freq: defined-vallex-%-frames.txt
    cat $< | suc > $@
```

make entries-version-1.0.freq zpracuje defined-vallex-1.0-frames.txt
make entries-version-1.5.freq zpracuje defined-vallex-1.5-frames.txt

Proměnná	Význam
`$<`	zastupuje název prvního ze vstupních souborů
`$^`	zastupuje názvy všech vstupních souborů
`$@`	zastupuje název výstupního souboru
`$*`	zastupuje tu část, která odpovídala % (např. `1.0`)
`$(HOME)`	proměnná z prostředí (na rozdíl od bashe nutné závorky)
`$$`	zastupuje prostě znak dolar, $

Složitější příklad: maximální dosažitelné pokrytí při učení se VALLEXových hesel

Úkol je vyhodnotit maximální teoreticky dostupný recall algoritmu:

- Vstup: VALLEXová hesla pro množinu trénovacích sloves, množina testovacích sloves
- Výstup: Algoritmem navržené rámce pro testovací slovesa.
  Algoritmus přitom musí brát trénovací rámce jako nedělitelné celky a jen je pro testovací slovesa použít (nebo nepoužít).
- Dostupný recall říká, jak často sloveso z principu nemůže dostat nějaký svůj rámec, protože takový rámec nebyl u žádného trénovacího slovesa vidět.

Postup řešení

- defined-vallex-frames.txt převedeme tak, aby na každé řádce byly všechny rámce daného slovesa, každá řádka ať odpovídá právě jednomu slovesu
- soubor pak rozdělíme na trénovací a testovací řádky
- napíšeme malý perlový skript, který vypočítá dostupný recall pro daný trénovací a testovací soubor

```
joined: defined-vallex-frames.txt
    # tady nejak spoj radky odpovidajici jednomu slovesu
training: joined
    skip 100 < $< > $@
test: joined
    head -100 < $< > $@
vysledek: training testing zmerit-recall.pl
    ./zmerit-recall.pl training testing > $@
```

Nakonec zavoláme **make vysledek**.

Jak spojit řádky pro jedno sloveso

Vstup:

```
absolvovat  0  0  1  phase verb  absolvovat-1  ACTobl(#nom) PATobl(#aku)
absolvovat  0  0  2  -  absolvovat-2  ACTobl(#nom) PATobl(#aku)
akceptovat  0  0  1  -  akceptovat-1  ACTobl(#nom) PATobl(#aku,ze)
```

```
joined:  defined-vallex-frames.txt
  cut -f1,7 $< \                  vyseknout lemma slovesa a rámec
  | split_at_colchange 1 \        můj jednoduchý nástroj přidá volné řádky
  | tr '\n' '|' \                 nahraď konce řádků speciálním znakem
  | sed 's/||/@/g' \              dvojici konců řádků nahraď dalším speciálním znakem
  | tr '@' '\n' \                 ze speciálního znaku udělej zpátky konec řádku
  > $@                            ulož výsledek
```

Výstup:

```
absolvovat ACTobl(#nom) PATobl(#aku)|absolvovat ACTobl(#nom) PATobl(#aku)
akceptovat ACTobl(#nom) PATobl(#aku,ze)
argumentovat ACTobl(#nom) MEANStyp(#instr,ze) PATopt(proti#dat,pro#aku)
```

Jak spočítat dosažitelný recall?

. . . to je cvičení z Perlu, za domácí úkol ☺

- Načíst dva soubory po řádcích, druhý sloupec vždy rozdělit (split) na znaku '|'.
- Z prvního souboru se naučit množinu známých rámců.
- U druhého souboru spočítat, kolik rámců bylo spatřeno v množině známých rámců.
- Recall = podíl těch spatřených ke všem testovacím.

Než se skript zmerit-recall.pl podaří odladit, budeme opakovaně zadávat **make vysledek**, abychom skript spustili.

Nakonec se výsledek dozvíme takto:
cat vysledek $\Rightarrow$ 67.5 %

Lepší práce s daty: n-fold evaluation

- Výsledek předešlého pokusu je nestabilní, záleží na konkrétních vybraných slovesech (vzali jsme slovesa od začátku abecedy!)
- Stabilnější výsledek dostaneme tak, že pokus budeme opakovat desetkrát, vždy jednu desetinu sloves použijeme jako test a ostatní k trénování.

Můj skript `nfold` právě tohle řeší. Obohaťme Makefile:

```
stabilni: joined zmerit-recall.pl
    nfold "./zmerit-recall.pl %train %test" < $< > $@
```

- `nfold` načte všechny řádky ze vstupu, zamíchá pořadí, rozdělí na desetiny
- pak spustí daný shellový skript (program s parametry) celkem desetkrát, přičemž vždy připraví trénovací a testovací soubor a speciální řetězec `%train` a `%test` nahradí jmény příslušných souborů

Jak to dopadlo

make stabilni ; cat stabilni

```
1 85.43
2 91.58
3 90.00
4 93.24
5 90.09
6 88.01
7 94.25
8 92.34
9 92.29
10 90.95
```

- Už prosté promíchání sloves vedlo k podstatně jinému výsledku: 85 místo 67 %
- Je vidět, jak rozptýlené hodnoty jsou: 85 až 94 %

/home/bojar/tools/shell/grp –keys= –items=AVG2,STDDEV2 < stabilni
$\Rightarrow 90.82 \pm 2.60$

Co když by VALLEX byl menší?

```
vyvoj: joined
    rm -f $@
    for i in 50 100 200 500 1080; do \
      cat $< \
      | shuffle \
      | head -$$(($$i+50)) \
      | nfold --testsize=50 "./zmerit-recall.pl %train %test" \
      | prefix --tab $$i \
      >> $@ ; \
    done
```

`for i in ...; do cmd; done`	cyklus v bashi, proměnná i
`$i`	hodnota *bashovské* proměnné `$i`
`$((1+2))`	výpočet výrazu v bashi
	dvojí dolar nutno psát kvůli make
shuffle, prefix	moje nástroje na promíchání řádek a přidání prefixu

Jak to dopadlo

make vyvoj ; head -5 vyvoj

↓ přidal `prefix`, velikost trénovací sady
↓ přidal `nfold`, číslo skupiny
↓ samotný výsledek skriptu `zmerit-recall.pl`

50	1	30.98
50	2	35.22
100	1	41.67
100	2	50.45
100	3	53.43

. . .

grp –keys=1 –items=AVG3,STDDEV3 vyvoj | sort -n

50	33.10	3.00
100	48.52	6.11
200	56.69	2.94
500	76.99	4.10
1080	90.92	2.62

. . . na generování tohoto přehlednějšího výstupu se samozřejmě hodí navazující pravidlo v Makefile.

Udržet pořádek, sdílet s kolegy / na víc počítačích

CVS = Concurrent Versions System

⇒ **man cvs**
⇒ `http://ufal.mff.cuni.cz/~semecky/vallex/documents/navod_cvs/`

- Pro všechny účastníky existuje jeden společný depozitář (repository).
- Jednou za život si každý vyrobí odštěpek: pracovní adresář (cvs check-out).
- Pravidelně se v pracovním adresáři:
  - zaregistrovávají nové soubory (cvs add),
  - vkládají platné verze do repozitáře, aby byly k dispozici všem (cvs commit),
  - aktualizují pracovní verze podle toho, co udělali jiní (cvs update).

Existují i modernější implementace (Subversion, svn) nebo přístupy (git).

Výhody CVS

Očividné (proto bylo CVS vyvinuto):

- pohodlné "kopírování" soustav pracovních souborů mezi lidmi či počítači
- pohodlné řešení případných konfliktů
- podrobný přehled o historii souborů (možnost kdykoli se vrátit)

Méně nápadné:

- vede k pečlivému oddělení citlivých souborů (náročné na výrobu, ruční práce ap.) a souborů odvozených pomocí Makefilů
  Do CVS přidávám (add) právě ty drahé soubory, neaktuální naopak odstraňuji (cvs remove).
- snadno se otestuje, že nic základního v CVS nechybí
  Udělám čistý check-out (hraju si na nového uživatele), nechám vše vybudovat od základu.

Slíbená smršť experimentů

- Každý experiment bude mít svůj podadresář: `exp-pokus1`, `exp-pokus2`. . .
- Budu mít jeden společný `Makefile.common` s obecnými pravidly a každý experiment bude mít ve svém podadresáři vlastní `Makefile`:

  Takto se jeden Makefile včleňuje do jiného:

  `include ../Makefile`
- Každý Makefile bude (po svém) reagovat na dohodnuté cíle:

  for e in exp-*; do cd $e; make results; cd ..; done
- Takto spojím výsledky všech experimentů pro komparativní vyhodnocení:

  (for e in exp-*; do prefix --tab $e < $e/results; done) > all-results
- Navíc si mohu připravit globální Makefile s cíli:

  make all-results

  make clone-pokus1-novypokus

  # nejen udělá kopii, ale též vloží standardní výbavu experimentu do CVS

Shrnutí

Na Unixu se pohodlně pracuje:

- s textovými soubory (roury, přesměrování)
- se stavebními kamínky, skripty všeho druhu
- s postupným vylepšováním procedury pro vyhodnocení nějakého experimentu včetně možnosti sledovat víc variant postupu (různé cíle v jednom Makefile)
- s více pracovními verzemi těchže experimentů, z více počítačů, s více spolupracovníky současně (CVS)

Existuje i pro Windows:

- bash a další nástroje – balík cygwin ⇒ `http://cygwin.com/`
- ikonkové CVS – TortoiseCVS ⇒ `http://www.tortoisecvs.org/`