Expressions rationnelles
Mémo

Tahina Ramananandro
ramanana@clipper.ens.fr
http://www.eleves.ens.fr/~ramanana/enseigne/ifips

IFIPS – TP 1 et 2
4 et 5 mars 2008

Il est vivement recommandé de faire tourner « en vrai » les exemples indiqués dans cette fiche ainsi que dans les fiches de TP.

Prenez votre temps. La machine est rapide, ne vous adaptez pas à sa vitesse. N'hésitez pas à me poser des questions.

1 `grep` : recherche de motifs dans un texte

`http://www.gnu.org/software/grep/doc/grep.html`

Usage :

- `grep` *'motif' fichier1 fichier2...*
  Recherche le motif dans les fichiers spécifiés. **Attention,** bien encadrer le motif entre apostrophes droites car sinon le shell interprète ce qui sera tapé.
  Si aucun fichier n'est spécifié, lit sur l'entrée standard. Ceci permet d'utiliser le pipe |, ou, à défaut, de donner la main à l'utilisateur pour entrer des lignes de texte, où `grep` recherchera le motif. Dans ce dernier cas, l'utilisateur tape ses données, et finit par CTRL+D.

Résultats :

- Code de retour à la UNIX (0 = succès, autre = erreur) :
  - 0 : le motif a été trouvé
  - 1 : le motif n'a pas été trouvé
  - autre : erreur système
- Produit les lignes (lues dans les fichiers, tapées par l'utilisateur ou produites par une commande) comportant au moins une occurrence du motif et les affiche à l'écran

Options :

- `-i` (insensitive) : pas de différences majuscules-minuscules
- `-o` (only) : n'affiche que les occurrences de motifs elles-mêmes, et non toute la ligne qui les contient
- `-v` (inVert) : recherche les lignes qui NE contiennent PAS d'occurrences du motif
- `-q` (quiet) : n'affiche rien. Ceci permet d'utiliser `grep` en traitement scripting « transparent pour l'utilisateur », n'utilisant que le code de retour.

Syntaxe des motifs : (dans l'ordre décroissant de la priorité opératoire)

Motif `grep`	Expression rationnelle
`a`	a
`[abc]`	a+b+c
`[^abc]`	A*\(a+b+c)
`[a-c0-2]`	(a+b+c+0+1+2)
`.`	A
`\(`e`\)`	e
e`*`	e^*
e`+`	$e(e^*)$
f `?`	$\varepsilon + f$
ef	ef
e`|`f	e+f

Motif `grep`	Signification
^	Début de ligne
$	Fin de ligne

( ) | ne sont pas des caractères spéciaux et doivent être précédés de \ pour qu'ils soient interprétés comme tels. Au contraire, [ ] ^ - * + ? $ sont des caractères spéciaux.

Exemples :

- `grep -o '[A-Z][a-z]*\( \|$\)'`
  Affiche tous les mots constitués d'une majuscule suivie par des minuscules et d'une espace ou de la fin de ligne.
- `echo -n You are using "; \`[1]
  `ps -o args= | grep -o '^[^ ]*emacs\( \|$\)' || \`
  `echo no text editor`
  Ce petit programme shell produit sans l'afficher la liste des programmes actuellement lancés (`ps -o args=`), et y cherche des programmes dont le nom d'exécutable (avant la première espace) se termine par `emacs`. S'il n'en trouve pas, il affiche « You are using no text editor ». Sinon, il affiche « You are using *programme* » sur la première ligne, et chacun des autres programmes trouvés sur les autres lignes.

2 `sed` : remplacements

`http://www.gnu.org/software/sed/manual/`

Usages :

- `sed` *'clauses' fichier1 fichier2...*
  Affiche les lignes de chaque fichier en y effectuant les remplacements demandés dans les *clauses*. **Attention,** bien spécifier l'ensemble de clauses entre apostrophes droites.
  Si aucun fichier n'est spécifié, lit sur l'entrée standard. Ceci permet d'utiliser le pipe |, ou, à défaut, de donner la main à l'utilisateur pour entrer des lignes de texte, où `sed` effectuera ses remplacements. Dans ce dernier cas, l'utilisateur tape ses données, et finit par CTRL+D.

Résultats :

- Code de retour à la UNIX (0 = succès, autre = erreur), indépendamment du nombre de remplacements effectués.
- Produit les lignes (lues dans les fichiers, tapées par l'utilisateur ou produites par une commande) après leur avoir fait subir les remplacements demandés. Si aucune partie de la ligne en entrée ne correspond au motif, la ligne demeure inchangée.

Syntaxe des clauses de remplacement : *clause1* ; *clause2* ;...

Les clauses sont évaluées successivement de gauche à droite.

Une clause s'écrit : s/*motif*/*remplace*/*opt*

Le *motif* est une expression régulière comme avec grep, à ceci près que les obliques droits / doivent être échappés (\/ au lieu de /).

La chaîne de remplacement est une chaîne de caractères, qui peut comprendre :

- `&` pour désigner la portion de l'entrée qui correspond au motif
- `\1` à `\9` pour désigner, dans le texte qui correspond au motif, la partie correspondant à l'une des 9 premières sous-expressions (entre parenthèses).

L'option peut être une combinaison de :

- rien : un remplacement par ligne
- `i` : pas de différences majuscules-minuscules
- `g` (global) : tous les remplacements possibles doivent être effectués pour chaque ligne, de gauche à droite.

Remarque : on peut indifféremment remplacer / par un autre caractère, par exemple `!`, pour que / n'ait plus besoin d'être échappé, mais alors l'échappement devient nécessaire pour le caractère choisi.

Exemples :

- `sed 's/\([0-9]*\)\.\([0-9]*\)/\1,\2/g'`
  Remplace tous les points décimaux par des virgules, lorsqu'on considère qu'un point décimal est situé immédiatement entre deux blocs de chiffres, et qu'il peut y avoir plusieurs nombres données sur une ligne.
- `sed 's!^[^ :]* ://\([^/ :]*\).*!\1!'`
  Extrait le nom du serveur (comme étant la partie de texte correspondant à la première sous-expression entre parenthèses) à partir d'une URL complète (de la forme *protocole ://serveur :port/suite...* ou *protocole ://serveur/suite...*) donnée en entrée. Comme on veut désigner le caractère / dans le motif, on utilise ! comme délimiteur dans la clause.

`sed` (anciennement Stream EDitor) propose d'autres fonctionnalités, mais elles sont moins utilisées (en tous cas, pas couramment), car `perl` s'est substitué à `sed` pour ces tâches avancées.

[1] Ce \ signifie que l'utilisateur n'a pas fini de taper la commande mais veut quand même passer à la ligne.

3 `perl` : traitements avancés

perl est un langage de script multi-usages qui tient à la fois du C et du shell. L'intégration massive de fonctionnalités liées aux expressions régulières est une de ses grandes forces. Elles dépassent de beaucoup le cadre de ce mémo.

`man perlre`

4 `flex` : analyse lexicale

`http://www.gnu.org/software/flex/manual/`

flex est utilisé par les programmeurs pour générer des parseurs pour des langages de programmation ou autres. flex est la première étape qui permet de détecter les *tokens*, ou lexèmes, élémentaires du langage.

Usage :

`flex -o` *code_cible.c fichier_source.l*

Le programme lit le *fichier source* décrivant les *règles* de formation des lexèmes, et produit en sortie un *code cible* C destiné à être compilé avec la librairie `fl` :

`gcc -lfl` *code_cible.c*

Une règle de formation est un couple qui à une expression régulière associe une action. Alors, le code C produit par lex donnera, une fois compilé, un programme qui traitera l'entrée standard, vérifiera si une partie du texte d'entrée correspond à l'expression régulière d'une des règles et exécutera l'action associée.

Un fichier source flex se présente en trois parties sous la forme suivante :

préambule
%%
règles
%%
postface

Le préambule est une suite de déclarations chacune de la forme :

- *nom motif*
  Définit une sorte d'alias pour le motif, destiné à être réutilisé dans les règles.
- `%{`
  code C
  `%}`
  Définit un morceau de code C à ajouter avant le code correspondant au lexeur proprement dit (par exemple, une primitive de calcul)
- `%top{`
  code C
  `}`
  Définit un morceau de code C à ajouter *impérativement au début* du fichier C produit. Notamment : les diverses directives du préprocesseur telles que `#include` ou `#define`

La postface est un morceau de code C que l'on voudrait rajouter à la suite du code correspondant au lexeur proprement dit (par exemple, le point d'entrée `main`). Ce code peut appeler le lexeur avec `yylex();`

L'ensemble des règles est présenté sous la forme suivante :

règle1
règle2
...

Une règle a pour forme :

motif { *code_C* }

où le code C correspond à l'action à effectuer. L'accolade ouvrante doit se situer sur la même ligne que le motif. Le code C peut inclure les variables suivantes :

- `char* yytext` correspondant à la chaîne de caractères trouvée respectant le *motif*. **Attention :** le pointeur contenu dans `yytext` n'est valable que dans le cadre de l'action effectuée et ne doit pas en fuir. Il faut donc faire un `strcpy` si on veut réutiliser la chaîne plus tard (notamment, identifiant). Problème : trous de mémoire.
- `int yyleng` correspondant à la longueur du texte correspondant
- `FILE* yyin` correspondant au canal d'où est lu le texte. Ne pas redéfinir cette variable à la main, plutôt utiliser `yyrestart`.
- `void yyrestart (FILE* newfile)` est une procédure permettant de réinitialiser la lecture (sans toutefois annuler les actions exécutées jusque-là, mais seulement la lecture) avec un canal, soit `yyin`, soit éventuellement un autre. Ceci permet de redéfinir `yyin`.

– `FILE* yyout` correspondant au canal où est écrit le texte avec la macro ECHO (cf. infra).

Dans les actions, on peut aussi utiliser les macros suivantes :

– `ECHO ;` écrit dans `yyout` la chaîne contenue à `yytext`
– `REJECT ;` force le lexeur à choisir une autre règle (et éventuellement donc essayer un autre motif)
– `yymore() ;` conserve la valeur de yytext pour la prochaine règle : lorsque la règle suivante sera utilisée, yytext contiendra la chaîne correspondant à la concaténation des motifs des *deux* règles. Ces effets peuvent se cumuler par un nouvel yymore(), etc.
– `yyless(`n`) ;` force le lexeur à n'avancer que de n caractères (au lieu de `yyleng`) : ceci permet de réutiliser tout ou partie de la chaîne matchée pour le choix de la règle suivante.
– `unput(`c`) ;` concatène c devant les caractères prévus pour le choix de la règle suivante
– `input()` consomme le premier caractère parmi ceux prévus pour le choix de la règle suivante, et le retourne.
– `yyterminate() ;` met définitivement fin à la lecture.

Une règle doit être prévue pour le motif `<<EOF>>` : lorsque la fin de la lecture est atteinte. Elle doit alors lancer `yyterminate() ;`

Syntaxe des motifs : (dans l'ordre décroissant de la priorité opératoire)

Motif `flex`	Expression rationnelle
`a`	a
`[abc]`	a+b+c
`[^abc]`	A*\(a+b+c)
`[a-c0-2]`	(a+b+c+0+1+2)
`(`e`)`	e
e`*`	e^*
e`+`	$e(e^*)$
f `?`	$\varepsilon + f$
ef	ef
e`|`f	e+f

Motif `flex`	Signification
`^`	Début de ligne
$	Fin de ligne
"*chaine*"	Chaîne de caractères verbatim
{*nom*}	Expression définie par l'alias dans le préambule

La construction de chaînes de caractères remplace l'échappement \ de `grep` et `sed`.

Petit exemple complet :

```
/* PREAMBULE */

/* Cette option peut être nécessaire pour éviter certains bugs */
%option noyywrap

%top{
/* Nécessaire pour atoi, atof */
#include <math.h>
}

entier     [0-9]+
flottant   ([0-9]+"."[0-9]*)|("."[0-9]+)
ident      [A-Za-z_][A-Za-z0-9_]*
espace     [ \t\n]+
autre      .

%%

{flottant} {
 printf ("Un flottant : %g\n", atof (yytext));
}

{entier}   {
  printf ("Un entier : %d\n", atoi (yytext));
}

{ident}    {
 printf ("Un identificateur : %s\n", yytext);
```

```
}

{espace}   {
}

{autre}    {
 printf ("Caractère non reconnu : %s\n", yytext);
 yyterminate ();
}

<<EOF>>    {
 printf ("Fin du fichier.\n");
 yyterminate ();
}

%%

/* Point d'entrée */

int main (int argc, char * argv[]) {
 ++argv, --argc;
 if (argc > 0) {
  for (; argc > 0; --argc) {
   yyin = fopen (argv[0], "r");
   yylex ();
   fclose (yyin);
  }
 } else {
  yyin = stdin;
  yylex ();
 }
 return 0;
}
```