Expressions rationnelles

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.

À l'inverse, vous pouvez même aller encore plus loin que ne le suggère ce TP.

Travaillez et coopérez : montrez votre code à vos voisins, demandez-leur de le tester, de trouver des cas qui feraient bugguer le programme obtenu, etc.

1 `grep` : recherche de motifs dans un texte

1.1 Recherche de fichiers

La commande `find` *dossier* permet de rechercher et d'afficher la liste des dossiers et fichiers récursivement contenus dans le *dossier*.

Écrire de deux façons différentes une commande UNIX permettant d'afficher les fichiers et dossiers à partir du 3ème niveau.

1.2 Affichage du nom de fichier « nu »

Comment afficher le nom d'un fichier sans le dossier qui le contient ? (utiliser `grep -o`)[1]

1.3 Extraction de données

Examiner la variable PATH (`echo $PATH`). Elle est constituée d'une liste de dossiers séparée par des deux-points :

Écrire une commande pour extraire le 3ème dossier en partant de la gauche, le 3ème dossier en partant de la droite. Peut-on déduire une formule générale ?

Écrire une commande UNIX permettant de séparer les dossiers du PATH en n'en affichant qu'un par ligne.

Si la commande `ypcat passwd` fonctionne (censée afficher la liste de tous les comptes UNIX de la machine), alors écrire une commande UNIX pour afficher les prénom et nom de tous les utilisateurs.

2 `sed` : remplacements

2.1 Extraction de données (bis)

Reprendre les commandes précédentes de la séparation de données en utilisant sed au lieu de grep.

Remarque : sed possède une fonction interne de séparation de champs. Ne pas l'utiliser. Utiliser seulement la construction s présentée dans le mémo.

[1] Ceci n'est pas un smiley :-)

2.2 Échappement de caractères

On peut constater sur les exemples précédents que, dès que l'on veut utiliser des données avec un séparateur, il se pose un problème : on ne peut pas *a priori* définir de valeurs utilisant elles-mêmes le séparateur. En d'autres termes, il est impossible de déclarer dans le PATH un dossier dont le nom contient un deux-points.

Pour contourner ce problème, on a recours à l'échappement de caractères.

2.2.1 Échappement d'un caractère

Imaginons que nous vivions dans un monde idéal où le PATH serait codé avec la règle suivante :

– pour chaque caractère deux-points présent dans un nom de dossier, il doit être écrit \ : au lieu de :

Cette règle seule est insuffisante. Exhiber un contre-exemple, et rajouter la règle manquante.

Réécrire alors la commande UNIX qu'il aurait fallu utiliser dans ce monde idéal.

Écrire également une commande UNIX qui permet à partir de la liste de dossiers, de constituer un PATH valide selon les règles ci-dessus (on négligera les sauts de ligne ; cependant, cette négligence peut être contournée en perl)

En utilisant les commandes `find` *dossier* et la construction `for`, construire une commande UNIX permettant d'afficher tous les fichiers accessibles par le PATH. Penser à échapper le caractère espace. Raffinement : on fera la liste de telles commandes sous la forme *commande # chemin/d/accès/complet*

Remarque : find peut se voir spécifier plusieurs dossiers. Mais dans la question ci-dessus, se restreindre à un seul dossier à la fois.

Pour améliorer la lisibilité, on pourra faire suivre cette commande de `| sort`[2] `| less`[3]

2.2.2 Chaînes de caractères

Imaginons que nous vivions dans un monde idéal ou le PATH serait codé avec la règle suivante :

– les dossiers peuvent contenir des chaînes de caractères codées avec des guillemets doubles "*chaîne*", ou des apostrophes '*chaîne*'.
Exemple : `ceci'est'une"representation"valide` correspondant au dossier `ceciestunerepresentationvalide`

Comment représenter un nom de dossier contenant à la fois les caractères ' " :

Réécrire les deux commandes UNIX voulues (séparation et reformation, pour cette dernière on négligera les sauts de ligne).

2.3 Extensions de fichiers

Certains noms de fichiers font figurer une extension de fichier : partie du nom de fichier (ses dossiers contenants exclus) entre le dernier point et la fin du nom de fichier. Étant donné une liste de fichiers en entrée, écrire une commande UNIX qui produit la liste de ces fichiers sans leur extension.

Puis, écrire une commande UNIX qui produit la liste de ces fichiers dont tous ceux terminant par l'extension .jpg sont à renommer avec l'extension .jpeg

Enfin, adapter cette dernière commande pour produire la commande (sans utiliser `for`) qui renomme tous les fichiers .jpg en .jpeg (pour renommer un *fichier1* en son nouveau nom *fichier2*, la commande est `mv` *fichier1 fichier2*)

3 `perl` : traitements avancés

On ne traitera rien directement avec perl : cela nécessiterait trop de temps. Mais ceux qui sont intéressés peuvent poser des questions.

Éventuellement, ceux qui connaissent déjà perl peuvent traiter une partie des problèmes précédents avec perl au lieu de grep et sed.

[2] Trie les lignes de texte produites
[3] Affiche le résultat page par page

4 `flex` : analyse lexicale

4.1 `grep` et `sed` revisités

Choisir un (ou plusieurs, selon le temps qu'il reste) exemple(s) des parties précédentes (grep et sed) et essayer de le réécrire avec flex.

Commencer par choisir un exemple, le plus simple possible, et regarder le code C produit par flex. Que peut-on observer ?

4.2 Une calculatrice en notation polonaise inversée (ou postfixe)

Certaines calculatrices utilisaient (et utilisent encore de nos jours) une notation appelée *notation polonaise inversée*. Cette notation permettait d'écrire a+b sous la forme $a\ b\ +$

Formellement, elle correspond à la grammaire suivante :

opérande : := nombre | opérande opérande opérateur

Montrer que cette grammaire est non ambiguë : pour toute expression respectant cette grammaire, il existe une unique façon d'appliquer les règles. (On pourra dire qu'un mot est formé par une liste de nombres et d'opérateurs, et on pourra raisonner par récurrence sur la longueur de cette liste.)

Comment une machine pourrait-elle effectuer simplement un calcul en notation polonaise inversée ?

En s'inspirant de l'exemple fourni, écrire un source flex implémentant une calculatrice en notation polonaise inversée gérant l'addition, la soustraction, la multiplication. Commencer avec les entiers seulement. Puis, observer ce qu'il se passe si on rajoute les flottants. Enfin, on pourra rajouter des fonctions telles que le cosinus, etc. (cos x s'écrira alors x cos) (on admettra que la grammaire reste non ambiguë tant que les opérateurs binaires ont tous des notations distinctes des opérateurs unaires).

On pourra borner par une constante la mémoire utilisée par la calculatrice.

Que se serait-il passé si on avait choisi la notation polonaise « non inversée » (ou préfixe) ?

4.3 Vers une calculatrice en notation infixe ?

Supposons que l'on rajoute dans le préambule :

```
%{
int paren = 0;
%}
```

et que l'on rajoute les deux règles suivantes :

```
"(" { ++paren; }
")" { --paren; }
```

Que faudrait-il encore rajouter pour pouvoir passer à une calculatrice infixe avec parenthésage intégral ? Et avec parenthésage optionnel et priorités opératoires ?

5 Automates finis

Pour préparer les séances suivantes, s'il reste du temps, on pourra commencer à :

– modéliser les automates en C : trouver une représentation
– écrire les fonctions de reconnaissance pour un automate quelconque

D'abord pour un automate déterministe, puis en raffinant avec les automates quelconques : déterminisation à la volée.

Mais pour la chaîne complète des expressions rationnelles vers les automates, on a besoin des cours sur le *parsing*, à venir.