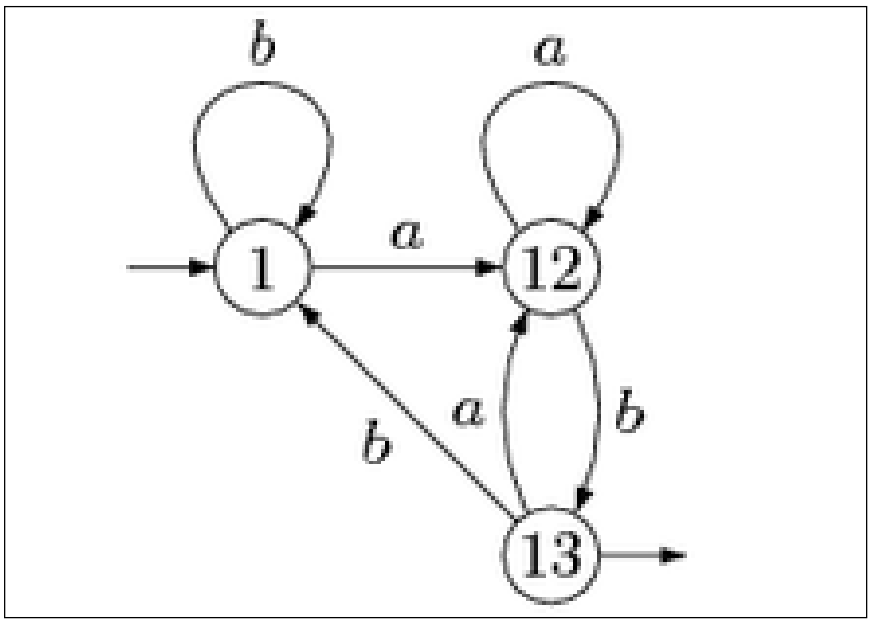
Rapport Projet EGREP

Implanter la commande EGREP

AFFES Mohamed Amin «3262731» et KOBROSLI Hassan « 3001993 » - 19 avril 2015



Introduction

Le but de ce projet est d'implanter la commande grep (sans options bien entendu ^^). Nous avons effectué ceci en suivant les étapes suivantes :

- Ecrire un Lexer, et un Parser pour les expressions régulières étendues en langage Java.
- Ecrire Les classes de base représentant un Automate (Automate, Etat, Transition).
- Modifier le Parser afin qu'il retourne un Automate représentant l'expression régulière donnée en suivant l'algorithme de Thompson pour la construction de l'automate qui est à la base non déterministe.
- Ecrire les différents algorithmes servant dans la construction de Thompson (concaténation, union, modification de l'automate suivi par un caractère « * » ou « + » ou « ? » …).
- Ecrire les algorithmes de déterminisation et minimisation.

Les algorithmes cités se trouvent dans le package « cpa.egrep.automate », les Lexer (fichier egrep.l) et Parser (fichier egrep.y) dans le dossier «LexParse» se trouvant à la racine du projet, les classes de base représentant un automate dans le package « cpa.egrep.automate » et enfin les classes représentant le lexer et le parser dans le package « cpa.egrep.parserlexer ».

Pour exécuter egrep, il faut executer l'archive jar fournie ou compiler le projet puis exécuter la classe Parser :

java –jar egrep.jar <regex> <inputFile>

ou

java cpa.egrep.parserlexer.Parser <regex> <inputFile>

Description des différentes Etapes du projet.

Descriptions des classes de bases de l'automate :

Tout d'abord on commence par définir ce qu'est un automate :

Un automate est constitué d'états et de transitions étiquetées par un caractère. Son comportement est dirigé par une ligne fournie en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée, si le mot fourni mène à un état final de l'automate on dit que l'automate accepte cette ligne ; sinon il ne l'accepte pas.

Nous avons donc 3 classes principales décrivant un automate puis une classe Factory avec des méthodes statiques pour créer des nouvelles instances de ces 3 classes et des classes ou autres méthodes statiques pour les algorithmes faits sur les automates, les 3 classes principales décrivant un automate sont :

- La classe « Etat » représente un état de l'automate, elle possède une liste de transitions menant vers d'autres états et une méthode statique donnant les états suivants de cet automate pour un caractère donnée ainsi que les epsilon transitions. Il existe aussi deux champs pour savoir si cet état représente le début d'une ligne (regex commençant par '^') ou bien la fin d'une ligne (regex terminant par '$').

La liste nommé « etatsDet » sert dans l'algorithme de déterminisation, son rôle est donc expliqué dans la partie décrivant cet algorithme.

- La classe « Transition » représente une transition. Elle contient un état de départ un autre d'arrivé et un caractère représentant l'étiquette de cette transition.

- La classe « Automate » représente un automate. Elle contient un état initial, une liste d'états finaux, et une liste de caractères représentant l'alphabet de cet automate. Cette classe contient une méthode statique retournant un boolean « accept(String line, Automate a) ». Cette méthode prend une ligne et un automate en paramètre, commence par vérifier si l'état initial de l'automate est un état de début de ligne « expression régulière commençant par ^ » si c'est la cas elle appelle la 2 ème méthode statique récursive continuant la vérification sinon on fait autant d'appel à la 2 ème méthode accept récursive que de caractères dans notre ligne en retirant à chaque fois un caractère au début de la dernière ligne vérifié. Dès que la 2ème méthode retourne un résultat true c'est à dire

que la ligne contient une chaine correspondante à la regex donnée on retourne true sinon si tous les appels ne donnent que des retours négatifs « false » on retourne false car aucune sous chaine dans notre ligne n'est accepté par cet automate et ne correspond donc à la regex donnée. Expliquons maintenant le rôle de la deuxième méthode accept prenant cette fois 3 arguments un état initial , une ligne et un automate. Voici le pseudo code de cette méthode:

```
Boolean accept (Etat init, String line, Automate a)
input: Etat init 'état courant', String line 'ligne à vérifier', Automate a
output: Boolean = true si ok false sinon
If etat_final(init)
        If fin_de_ligne(init)
                If (length(line) == 0) return true
                else return false.
                end If
        return true.
        end If
end If
If Not_empty(line)
        For all Transition t in init :
                If (caractere(t) == « eps » && accept(Etat_arrivee(t), line, a) == true)
                        return true
                else if (caractere(t) == line.charAt(0) &&
                        accept(Etat_arrivee(t), line.substring(1), a) == true)
                        return true
                end If
        end For
end If

return false
end accept
```

Pour les différents tests que nous avons effectué, nous avons réalisé une fonction qui compte le nombre total d'epsilon transitions, une autre le nombre de transitions.
Deux autres méthodes retournent respectivement tous les états et toutes les transitions d'un automate. Ces deux méthodes nous servent principalement à produire un fichier représentant un automate. En utilisant GraphViz, nous pouvons alors obtenir une représentation graphique de cet automate.

Lexer :

Le Lexer contient un espace de définition de règles, dans cet espace on définit les chaines de caractères ou expressions régulières que notre Lexer accepte, puis ce que le Lexer doit faire quand il les rencontre. Dans notre cas on veut que le Lexer retourne le token correspondant défini dans le Parser, afin que ce dernier puisse par la suite parser correctement le texte donnée ou déclencher une exception si la chaine donnée au Lexer est incorrecte. Notre Lexer attend donc une expression régulière et retourne une suite de tokens correspondant à cette expression au Parser qui se charge de générer l'automate correspondant.

Parser :

Le Parser contient tout d'abord un espace de déclaration, dans cet espace nous déclarons les tokens et la règle start retournant le résultat en fin de parser ici « extended_reg_exp » ensuite on donne les types de retour des règles définies dans la suite.
Ensuite on définit donc ces règles, on définit donc dans cet espace le comportement que doit avoir le Parser lors de la réception d'une suite de tokens spécifique de la part du Lexer.
Nous avons divisés ce Parser en deux grandes parties :
Bracket Expression et Extended Regular Expression.

Tout d'abord concernant les Bracket Expression :

- nous avons le cas [matching list] ou le cas [non matching list]. Concernant ce premier cas le parser retourne un automate à deux états ayant autant de transitions que de caractères possibles dans la liste de caractères matching list étiquetés par ces caractères. Pour le deuxième cas on retourne un automate à deux états contenant autant de transitions que de caractères ASCII étendues (code 0 à 255 excepté le caractère «\n ») étiquetés par ces caractères et qui ne sont pas contenues dans la liste de caractères contenues entre les brackets. Cette liste de caractères ASCII étendue est construite dans la règle « nonmatching_list ».
- les règles « matching_list » et « nonmatching_list » contiennent la règle « bracket_list » qui représente la liste des caractères à traiter se trouvant entre les brackets, Cette « bracket_list » contient une « follow_list » qui contient une ou plusieurs « expression_term ». Toutes ces règles retournent une liste de caractères assemblent les

différents listes venants des règles venants après (pour le cas de la règle « | follow_list expression_term »).

- Une « expression_term » est soit une « single_expression » qui est une liste contenant un caractère unique (dans le cas où le passer retourne un caractère précédé par un backslash ce dernier est supprimé). Ou alors, une « range_expression » qui retourne la liste des caractères contenues entre a et c par exemple pour le cas « a-c » on parcourt tous les caractères ASCII se trouvant entre les deux caractères (eux compris) se trouvant des deux côtés du séparateur '-', donc les caractères a, b et c.

La deuxième partie concerne les Extended Regular Expression :

- Une « extended_reg_exp » est soit une « ERE_branch » ou l'union de plusieurs « ERE_branch » donc l'union de plusieurs automates. L'algorithme de l'union est expliqué dans la suite dans la partie décrivant les algorithmes. Une « ERE_branch » est un automate représentant une expression régulière ou une concaténation de plusieurs automates représentant une expression régulière.

- Une « ERE_expression » est un automate représentant une expression régulière. Cet automate peut être une automate de début ou de fin de ligne (Automate à un état représentant le début ou la fin de la ligne), ou une « extended_reg_exp » entre parenthèse (retourne l'automate contenant la regex étendue), ou une « one_char_or_coll_elem_ERE » qui est un automate représentant un caractère (un automate à une transition représentant le caractère). Un Quoted Char (un automate à une transition représentant le caractère après le back slash) ou un point, ce dernier est représenté par un automate contenant tous les caractères ASCII en transition entre deux états un initial et l'autre final.
- Une « ERE_expression » peut être aussi un automate « ERE_expression » suivi d'un « ERE_dupl_symbol » qui est l'un des symboles suivants (*, +, ?, {DIGIT}, {DIGIT,}, {DIGIT,DIGIT}). Pour chacun de ces cas on effectue la transformation correspondante. Les transformations sont expliquées dans la partie décrivant les algorithmes.

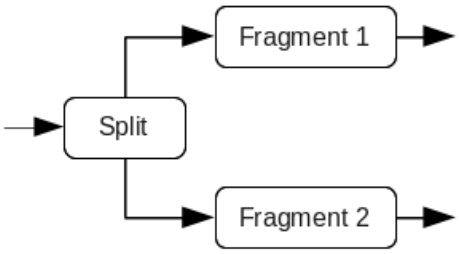
Descriptions des algorithmes utilisés :

Tel que mentionné précédemment, l'automate est composé d'un ensemble de fragments. Ces fragments sont assemblés suivant les opérations de base des expressions régulières.

Les figures ci-dessous montrent comment les fragments sont assemblés en fonction des opérations appliquées.

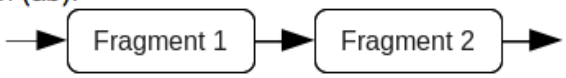
- Pour effectuer l'union entre deux automates a et b on crée un nouvel état initial Split qui devient l'état initial du nouvel automate puis ajoute une epsilon transition allant de cet état aux états initiaux de a et b, les états finaux du nouvel automate sont les états finaux de a et de b, l'alphabet du nouvel automate et l'union des alphabets des automates a et b.

- L'union (a|b):



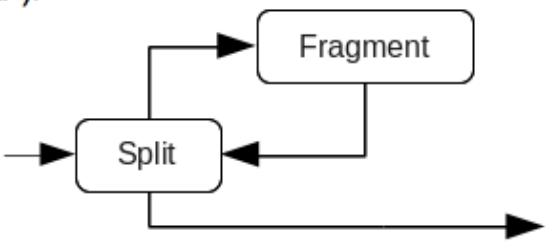
- Concernant la concaténation entre deux automates a et b, un nouvel automate est créé ayant pour état initial l'état initial de a et pour états finaux les états finaux de b, on ajoute une epsilon transition entre les états finaux de a et l'état initial de b.

- Concaténation (ab):



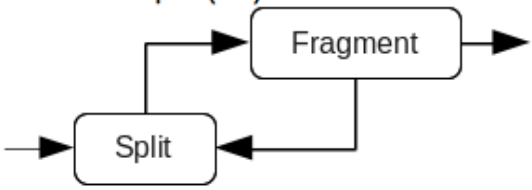
- Concernant la fermeture (a*) pour un automate a On crée un nouvel état initial pour l'automate init, on ajoute une epsilon transition entre init et l'ancien état initial de a, on crée aussi un nouvel état f et on ajoute une epsilon transition entre init et f, puis enfin on ajoute une epsilon transition entre chaque état final de a et sont nouvel état initial init. On ajoute f aux états finaux de a.

- La fermeture (a*):



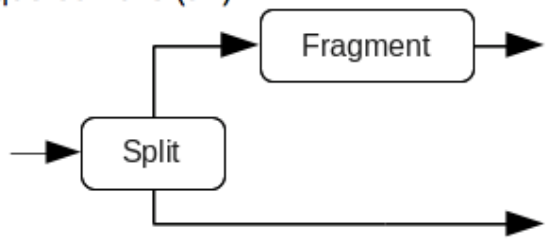
- Concernant la fermeture (a*) pour un automate a On crée juste une epsilon transition entre chaque état final et l'état initial.

- Répétition simple ou multiple (a+):



- Concernant la fermeture (a?) pour un automate a On crée un nouvel état initial pour l'automate init, on ajoute une epsilon transition entre init et l'ancien état initial de a, on crée aussi un nouvel

- Répétition unique ou nulle (a?):



état f et on ajoute une epsilon transition entre init et f puis on ajoute f aux états finaux de a.

- Concernant la méthode ERE_dupl_symbol dans le cas où la chaine s est égal à l'un des caractère suivant (*,?,+) l'une des transformations précédentes est appliqué à cet automate, sinon:

- Dans le cas {n} :

On concatène l'automate n fois avec une copie de lui-même.

- Dans le cas {n,} :

On concatène l'automate n fois avec une copie de lui-même. La dernière copie avant de la contacter à l'ensemble on lui applique la transformation (+).

- Dans le cas {n,m} :

On concatène l'automate m fois avec une copie de lui-même. Après (n-1) concaténations on ajoute chaque prochain résultat de concaténation à la liste d'états finaux du nouvel automate.

- Concernant la méthode getCharBetween(a,b), elle retourne un liste contenant les caractères ASCII se trouvant entre les deux bornes données eux inclues.

- Concernant la méthode getExtendedAsciiChar elle retourne une liste contenant les caractères ASCII étendues.

- Concernant la méthode getExtendedAsciiChar(HashSet<String> s) elle retourne une liste contenant les caractères ASCII étendues et qui ne sont pas contenues dans la liste s.

- Concernant la méthode estAutomateComplet elle vérifie si l'automate est complet c'est à dire si pour tout état e et toute lettre a, il existe au moins une transition qui part de e et qui est etiquetté par la lettre a.

- Concernant l'algorithme de déterminisation voici le pseudo code de la méthode:

Un automate fini déterministe est un quintuplé $(Q, \Sigma, \delta, q_0, F)$ constitué des éléments suivants

- un alphabet fini (Σ)
- un ensemble fini d'états (Q)
- une fonction de transition ($\delta : Q * \Sigma \rightarrow Q$)
- un état de départ ($q_0 \in Q$)
- un ensemble d'états finaux (ou acceptant) $F \subseteq Q$

Début algorithme :

- Q_d est initialisé à ∅ et soit E un ensemble d'états initialisé à E = {{q_0}}
- Tant que E est non vide,
  - choisir un élément S de E (S est donc un sous ensemble de Q_n),
  - ajouter S à Q_d,
  - pour tout symbole a ∈ Σ,
    - calculer l'état S' = $\cup_{q \in S} \delta_n(q, a)$
    - si S' n'est pas déjà dans Q_d, l'ajouter à E
    - ajouter un arc sur l'automate entre S et S' et la valuer par a

Pour tout état S' dans Qd,

Si S' contient un état final de l'automate alors :

F(nouvel automate)add(S')

Nous disposons aussi d'une méthode de suppression des Epsilon Transitions :

- En partant d'un état initial Ei, pour chaque transition, on appelle récursivement la méthode de suppression sur l'état d'arrivée de la transition.
- Si cette transition est une ε-transition menant vers E1, pour toute transition T1 sortant de E1, on crée une nouvelle transition sortant de Ei et allant vers la destination de T1.

Chaque état visité est ajouté dans une liste d'états, afin d'éviter de boucler plusieurs fois si l'automate contient un cycle.

L'algorithme de minimisation implémenté est celui de Brzozowski. Son principe est simple :

- On a en entrée un automate déterministe A.
- On crée l'automate miroir (transposé) A~ de cet automate.
- On déterminise cet automate A~.
- On crée l'automate miroir de A~.
- On déterminise ce dernier automate et on le retourne.

L'automate miroir A~ d'un automate A est l'automate tel que les états finaux de A sont les états initiaux de A~ et inversement.

Conclusion :

Dans le pire des cas, la méthode accept qui détermine si la regex est matchée par une ligne de taille n, la complexité est en $O(n^2)$: on prend comme point de départ chaque caractère de la ligne, puis on fait des appels récursifs sur la suite de la ligne. Pour le premier caractère, on fera n-1 appels récursifs, pour le deuxième caractère, n-2 appels récursifs, …

Cette complexité peut être encore plus grande si l'automate contient beaucoup d'epsilon transition, d'où l'intérêt de les supprimer, ce qui a un avantage non négligeable sur les performances.