Rapport du projet d'Informatique Théorique II

Jérémy TURON & Salomé COAVOUX

28 mars 2013

Table des matières

1 Les fonctionnalités du code

1.1 Fonction : completer(Aut)

Cette fonction renumérote les états de 1 à N, puis choisit pour puits l'état N+1. On parcourt ensuite tous les sommets de l'automate, et on rajoute ses transitions manquantes vers le puits. Si l'on n'a rien rajouté, l'automate est déjà complet : on supprime l'état puits, grâce à la fonction intermédiaire *delete_state()* créée pour la fonction *completer()*, puis réutilisée plus tard. Sinon, on rajoute les transitions du puits vers lui-même pour toutes les lettres de l'alphabet. S'il existe des epsilons-transitions, on les ignore. On renvoie l'automate construit ainsi.

1.2 Fonctions : union(Aut1, Aut2), intersection(Aut1, Aut2)

Ces fonctions utilisent les définitions données en cours sur l'union et l'intersection, à savoir :

$A_1 = (Q_1, \text{A}, I_1, F_1, \delta_1)$
$A_2 = (Q_2, \text{A}, I_2, F_2, \delta_2)$
$A_\cup = (Q_1 \times Q_2, \text{A}, I_1 \times I_2, F_1 \times Q_2 \cup Q_1 \times F_2, \delta)$
$A_\cap = (Q_1 \times Q_2, \text{A}, I_1 \times I_2, F_1 \times F_2, \delta)$

Il reste donc à calculer les nouveaux états et les nouvelles transitions. Les états sont calculés à l'aide d'un fonction intermédiaire *produit_cartesien()*, qui se contente de parcourir deux listes, et de créer des tuples pour chaque couple d'éléments dans une liste différente. Les nouvelles transitions, elles, sont traitées dans la fonction *nouvelles_transitions_IU()*. Pour unir ou intersecter deux automates, on a besoin qu'ils soient complets, sinon l'algorithme peut avoir des problèmes pour créer les transitions, on les complète donc au tout début de la fonction. Toutefois, il n'a pas besoin d'être déterministe.

1.3 Fonction : miroir(Aut)

Pour écrire la fonction *miroir()*, il suffit de parcourir les transitions et d'inverser l'état de départ et l'état d'arrivée de chaque transition. C'est une fonction triviale.

1.4 Fonction : determinisation(Aut)

Pour déterminiser, on suit la méthode vue en cours et en TD : on crée le nouvel état initial à partir de tous les états initiaux de l'automate, ensuite on le rajoute à la liste des nouveaux états, et on l'enfile dans celle des états à traiter. Depuis l'état au sommet de la file, on regarde tous les états que l'on peut atteindre par une lettre, et on les enfile à leur tour, en tant qu'un ensemble d'états. Quand on a appliqué cette méthode à chaque lettre de l'alphabet, l'état du sommet de la file a été traité : on peut le défiler.

1.5 Fonction : complement(Aut)

La fonction *complement()* consiste simplement à récupérer tous les états non finaux dans un automate et à les transformer en états finaux, tandis que les états finaux deviennent non finaux.

1.6 Fonction : minimiser(Aut)

La fonction *minimiser()* a été implémentée en suivant l'algorithme de Moore. Pour cela, une fonction intermédiaire *moore()* a été créée. Un état est codé par une liste de chiffres. Le premier élément correspond au groupe de l'état, le reste au groupe que ce même état atteint par une lettre donnée. *moore()* a besoin de la liste générée par l'algorithme de Moore au tour précédent. Il se sert ensuite de cette liste pour refaire les groupes, puis rajoute les codes correspondant aux transitions en parcourant les états et l'alphabet au fur et à mesure. Il faut donc initialiser une liste une première fois avant de l'utiliser. C'est ce que l'on a fait ici en numérotant les états finaux par '1' et les autres états par '0'. On s'arrête quand la liste passée en paramètre de *moore()* et celle retournée par lui sont identiques. On construit ensuite l'automate minimal en fonction des groupes obtenus.

1.7 Fonction : expression_vers_automate(expr)

Pour implémenter cette fonction, nous avons choisi d'utiliser l'algorithme de Thomson. Nous avons utilisé pour cela quatre fonctions intermédiaires. Trois d'entre elles définissent les différentes opérations, à savoir l'étoile '*', l'union '+' ou la concaténation '.', chacune s'appliquant à une ou deux expressions, suivant si l'opération est binaire ('+', '.') ou unaire('*'). La quatrième fonction défini un switch entre les différentes opérations possibles, et, dans le cas où l'expression est réduite à une lettre, renvoie l'automate correspondant. Cette entité fonctionne récursivement, la fonction principale appelant le switch qui redirigera ensuite vers l'opération désirée, qui elle-même utilisera le switch pour récupérer les automates correspondants aux expressions passées en arguments.

1.8 Fonctions : expressions_rationnelles_vers_liste(expr)

Cette fonction a été implémentée de deux façons différentes : La première à l'aide des analyseurs syntaxiques et lexicaux, comme vu en Analyse Syntaxique. On utilise donc *lex* et *yacc* pour construire la grammaire des expressions rationnelles, vérifier cette grammaire et la transformer en liste *python*.
La seconde implémentation est entièrement codée en *python*. Tout d'abord, on récupère l'expression et on rend la concaténation explicite avec la fonction *expr_concat()*. Par exemple, l'expression 'a+b.c(ef)*' deviendra 'a+b.c.(e.f)*'. Ensuite *est_correcte_expr()* vérifie la grammaire, cette fonctionnalité étant gérée par *yacc* dans la première version. A l'aide d'un pseudo-switch/case, on regarde :

- Si le premier symbole de l'expression est correct : une parenthèse ouvrante ou une lettre.
- Pour tous les caractères sauf le dernier, si le caractère suivant est légal.
- Pour le dernier caractère, s'il est légal.

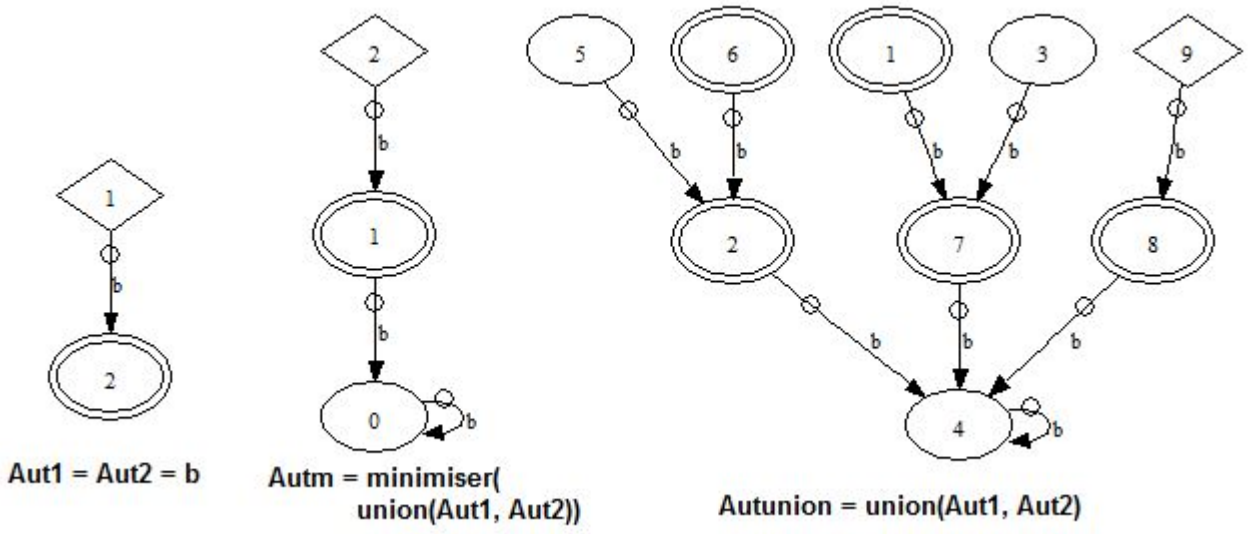
On utilise cette fonction dans la fonction principale pour vérifier que l'argument passé en paramètre est correct. Ensuite, on construit la liste à l'aide d'une pile similaire à l'automate à pile vu en Analyse Syntaxique. On refait un switch/case qui, en fonction du caractère lu, pourra :

- Empiler si l'on ne sait pas encore quelle règle appliquer.
- Dépiler une fois et rempiler l'élément construit pour l'opérateur unaire : '*'.
- Dépiler trois fois et rempiler l'élément construit pour les opérateurs binaires : '+' et '.'. On regarde l'opérateur suivant pour connaître l'ordre de priorité empilement/réduction.
- Réduire jusqu'à la première parenthèse ouvrante lue si l'on voit une parenthèse fermante.

2 Les problèmes rencontrés et les choix dans le code

2.1 L'union et l'intersection

Ces deux fonctions ont été implémentées très simplement et en suivant les définitions données en cours et rappelées plus haut. Le problème que nous avons et que nous n'avons pas pu résoudre de manière efficace est le nombre d'états. En effet, l'automate obtenu est juste, mais génère souvent un ensemble d'états inutiles puisqu'on ne peut pas les atteindre à partir des états initiaux. Ce problème vient de la façon dont nous avons généré les états : avec le calcul $Q_1 \times Q_2$, on crée tous les couples d'états possibles, et on en fait les nouveaux états. Or, il est tout à fait possible qu'un couple d'état ne soit pas utile pour une union ou une intersection. La différence est qu'en cours, nous construisions les couples d'états au fur et à mesure de la construction de l'automate, tandis qu'ici, on les construit tous d'un coup sans chercher véritablement à savoir si l'on en aura besoin ou non.



2.2 La minimisation

Pour minimiser un automate, nous avons décidé d'utiliser l'algorithme de Moore. En effet, il nous paraissait vraiment inintéressant d'écrire la fonction avec le double renversement, fonction qui pouvait tenir sur une seule ligne, et qui utilisait les autres fonctions que l'on devait implémenter. Le principal problème pour cette implémentation a été de construire la liste pour appliquer Moore. En effet, cet algorithme doit tout d'abord parcourir la liste précédente pour déterminer le groupe de chaque état, puis parcourir les états et l'alphabet pour déterminer dans que groupe va chaque état par chaque lettre de l'alphabet. Il a donc fallut utiliser une liste de tuple, la bibliothèque *automaton* fournie ne permettant pas de nommer des états à partir d'objets non "hashable".

2.3 De l'expression vers la liste préfixe

La fonction qui transforme une expression en liste préfixe a été pour nous une vraie source de problèmes. Il nous paraissait vraiment difficile d'avoir à gérer les opérateurs, les parenthèses qui peuvent être sur plusieurs niveaux, les priorités entre les différents opérateurs, la concaténation implicite, etc... Nous avons donc exploité l'idée d'utiliser un analyseur syntaxique, tout en continuant nos recherches sur l'implémentation *python*. Jérémy s'est donc occupé de l'implémentation en *lex/yacc*, tandis que Salomé a cherché l'algorithme en *python*. Au final, les deux implémentations étaient correctes, nous les avons donc gardées toutes les deux. Cependant, le programme de test utilise exclusivement la fonction utilisant *lex/yacc*. Il est toutefois possible d'accéder à la seconde version à la main sous le nom de textitexpr_vers_liste_bis().