UTC - CNRS

# Méthode B - Expressions régulières et automates

## Spécification formelle d'un nouvel opérateur

**Laurent WOUTERS**

**21/03/2008**

**Stage suivi par Jean-Louis BOULANGER**

Résumé :

On cherche ici à introduire un nouvel opérateur dans le langage des expressions régulières afin de simplifier l'écriture de certaines expressions. Dans ce cadre, la méthode B a été utilisée pour spécifier les structures de données et algorithmes qui permettent de définir la sémantique de cet opérateur ainsi que sa possible implémentation dans des outils tels que les générateurs de lexer. En particulier, on s'aperçoit que le point critique de la définition de l'opérateur dans de tels outils est la transformation de l'automate non déterministe contenant une représentation de celui-ci en un automate déterministe équivalent. On se penche donc spécialement dans la spécification sur ces deux structures de données et leurs algorithmes associés. Finalement, une adaptation des algorithmes implémentant cette spécification est donnée pour être directement intégré dans un outil utilisant la représentation des expressions régulières sous forme d'automates.

Summary:

This document introduces a new operator into the regular expressions language in order to simplify the writing of otherwise complicated expressions. Doing so, the formal B method is used to specify the data structures and algorithms which define this operator's semantic and implementation in software tools such as lexer generators. It has to be noted that the critical point for this operator's definition in such tools is the transformation of the non-deterministic automaton containing a representation of the operator into an equivalent deterministic automaton. So the specifications of those two structures and their associated algorithms are particularly taken care of. And last but not least, a solution adapting the standard algorithms to meet the specification is given so that it could be directly used in tools that use regular expressions represented with automata.

# Sommaire

# Introduction

L'objectif du projet est d'étendre le langage des expressions régulières en implémentant un nouvel opérateur. La méthode B étant utilisée pour spécifier et définir la sémantique de cet opérateur ainsi que son implémentation.

On sait que les langages réguliers peuvent être exprimés sous forme d'expressions régulières ainsi que sous forme d'automates à état finis. Dans un outil informatique tel qu'un générateur de lexer les opérations de bases sont la transformation d'une expression régulière en automate fini non déterministe puis la transformation de ce dernier en automate fini déterministe. (1) Le générateur de lexer part de l'ensemble des expressions régulières pour en former une générale avec plusieurs états finaux capable de reconnaitre tous les terminaux d'une grammaire d'un langage par exemple.

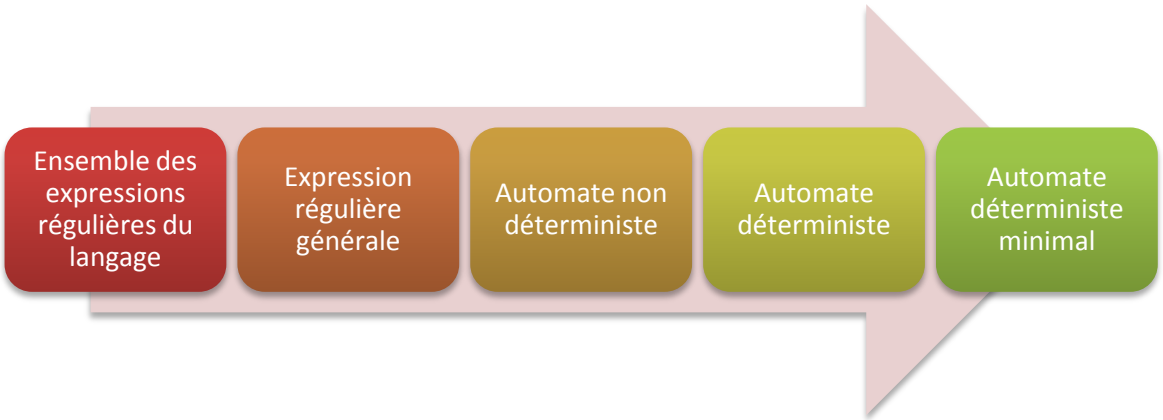


Figure 1 - Générateur de lexer : transformation des données

L'automate déterministe minimal peut alors avec un peu de liant faire office de lexer.

On propose ici d'étendre le langage des expressions régulières pour introduire un nouvel opérateur, l'opérateur de restriction. Un opérateur équivalent existe déjà dans la théorie des ensembles : la différence d'ensembles. On peut donc en particulier écrire $L_1 \setminus L_2$ si $L_1$ et $L_2$ sont respectivement des langages. $L_1 \setminus L_2$ désigne l'ensemble des éléments de $L_1$, excepté ceux appartenant aussi à $L_2$.

## Approche intuitive

La sémantique de l'opérateur de restriction est définie comme suit :

- ➔ Soit $exp_1$ une expression régulière et $L(exp_1)$ le langage décrit par $exp_1$.
- ➔ Soit $exp_2$ une expression régulière et $L(exp_2)$ le langage décrit par $exp_2$.

Alors $L(exp_1 – exp_2) = L(exp_1) \setminus L(exp_2)$ est l'ensemble des chaînes de $L(exp_1)$ privé de l'ensemble des chaînes de $L(exp_2)$ (zone bleue sur la figure ci-dessous). C'est aussi le complémentaire de $L(exp_1) \cap L(exp_2)$ dans $L(exp_1)$.

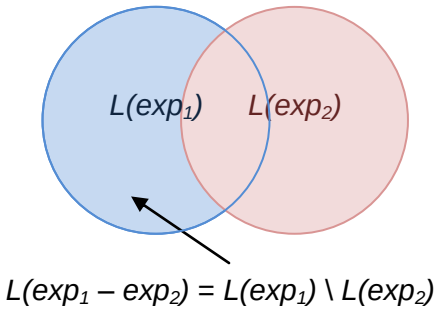


Figure 2 - Opérateur de restriction sur les ensembles

On sait construire un automate non déterministe à partir de l'arbre syntaxique de l'expression régulière associée. En particulier, l'opérateur d'union | peut se représenter ainsi : pour l'expression $exp_1$ | $exp_2$ on obtient l'automate suivant :

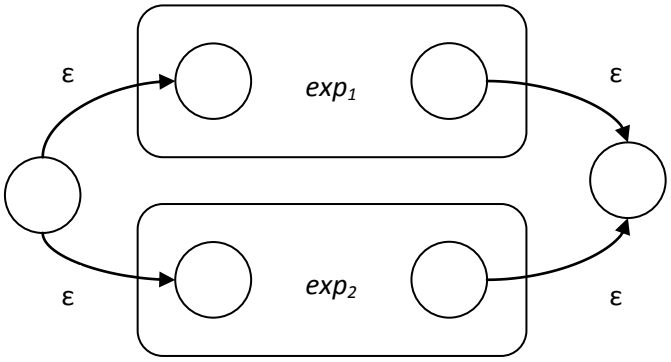


Figure 3 - Représentation de l'opérateur d'union

Pour représenter l'opérateur de restriction, on utilise quasiment la même méthode que pour l'union, mais on utilise des marques comme suit. Une marque positive est ajoutée sur l'état final du sous-automate représentant $exp_1$ et une marque négative est ajoutée sur l'état final du sous automate représentant $exp_2$. Pour l'expression $exp_1$ - $exp_2$ on définie que l'automate non déterministe associé est le suivant :

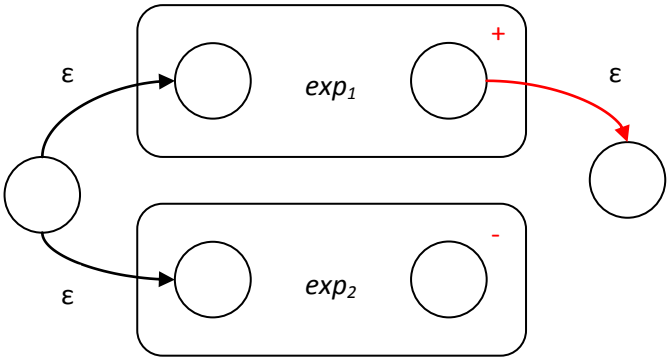


Figure 4 - Représentation de l'opérateur de restriction

En ce qui concerne la transformation en automate déterministe, intuitivement, lors de construction de l'automate équivalent on déterminera les chaînes reconnues à la fois par $exp_1$ et $exp_2$. Lorsqu'une telle chaîne est découverte il suffira d'interdire à l'algorithme l'epsilon-transition (en rouge) vers l'état final. On autorisera cette transition que si on ne peut accéder qu'à l'état marqué positivement, c'est-à-dire que seule $exp_1$ est reconnue. A l'inverse si seule $exp_2$ *est* reconnue, l'algorithme ne fera rien de toute façon puisque l'état final de $exp_2$ n'a pas de transitions sortantes.

Pour éviter les conflits entre les marques représentant différents opérateurs de restriction dans une même expression régulière, on peut associer à chacun un couple de valeurs opposées différent.

## Spécification formelle

Afin de réaliser plus tard l'implémentation de ce nouvel opérateur dans un générateur de lexer on commence par spécifier de manière formelle les objets et méthodes. Ainsi il convient en particulier de spécifier les automates non déterministe et déterministe. Enfin comme le point crucial du processus ici est la conversion du l'automate non-déterministe en automate déterministe, on écrit une spécification pour cet algorithme.

Les automates spécifiés ici sont un peu particuliers comme les valeurs associées aux transitions sont un peu plus complexes. Au lieu de n'utiliser que de simples symboles comme valeurs pour les transitions des automates on utilise ici des ensembles de symboles. Ces ensembles seront en fait des intervalles. En effet, en considérant que chaque symbole est associé à un nombre entier différent (qui est en fait la représentation binaire du symbole dans un encodage particulier) il est possible d'ordonner l'ensemble des symboles.

L'utilisation d'intervalles amène un certain nombre de problème, dont la définition de ce qu'est un automate déterministe avec ce type de transition. On considère que dans un automate déterministe, pour chaque état les valeurs des transitions sortantes ont des intersections nulles deux à deux. Cela implique qu'au cours de la transformation, d'un automate non déterministe vers un automate déterministe une « normalisation » des transitions des états interviennent afin de correspondre à cette description.

Les spécifications des éléments communs aux deux types d'automates ont été regroupées dans un autre composant. Les spécifications sont en annexes.

- ➔ FA.mch contient les spécifications des ensembles communs aux 2 automates.
- ➔ NFA.mch contient la spécification de l'automate non déterministe
- ➔ DFA.mch contient la spécification de l'automate déterministe.

## Spécifications communes

Les spécifications communes contiennent les définitions des ensembles qui seront réutilisés dans les autres machines. Ainsi, on définie l'ensemble des intervalles utilisable comme valeurs pour une transition d'un automate (déterministe ou non). Un intervalle est donc un couple de valeur dans Z tel qu'un intervalle de Z+ est définie par ces valeurs.

De manière similaire, on définie l'ensemble des epsilons utilisable dans un automate non déterministe comme tous les intervalles de Z ayant une longueur nulle ou négative.

On définie également l'ensemble des entrées possible pour un automate comme une suite d'entier positifs ou nuls.

## Spécification de l'automate non déterministe

Une machine abstraite est utilisée pour spécifier un automate non déterministe et l'ensemble des opérations qu'il est possible de lui appliquer. La première chose à faire est de définir l'ensemble des valeurs possibles pour une transition dans cet automate comme l'union de l'ensemble des valeurs « normales » et de l'ensemble des epsilons. Les variables abstraites de la machine sont explicités ci-dessous.

| Nom | Commentaires |
|---|---|
| **nfa_states** | C'est la variable contenant l'ensemble des états de l'automate. Les états sont numérotés de 0 à n de manière automatique donc nfa_states peut également être considéré comme un intervalle de N commencant à 0. |
| **nfa_finals** | C'est une relation représentant les états finaux. L'appartenance du couple (s, n) à cette relation permet de représenter le fait que l'état s est final et que l'automate reconnait à cet état le symbole n. |
| **nfa_transitions** | C'est une relation représentant les transitions entre les états. Si la couple ((e, v), f) appartient à la relation alors il existe une transition de valeur v partant de e vers f. |
| **nfa_marks** | C'est l'injection partielle représentant le marquage des états. Si le couple (s, m) appartient à la fonction, l'état s est marqué par la marque m. |
| **eps_closure_simple** | C'est la fonction partielle associant à chaque sous-ensemble d'états l'epsilon-fermeture de celui-ci. |
| **eps_closure** | C'est la fonction partielle associant à chaque sous-ensemble d'états l'epsilon-fermeture de celui-ci en tenant compte du marquage des états. En particulier, deux états marqués de manière opposée ne peuvent pas appartenir au résultat de ce type de fermeture. |
| **nfa_end_trans** | Cette fonction associe à un ensemble d'état et à une valeur, l'ensemble des états accessibles depuis l'ensemble par une transition de la valeur donnée. L'ensemble des états d'arrivé peut éventuellement être vide. |
| **nfa_end_path** | Cette fonction associe à un ensemble d'état et à une entrée de l'automate (suite de valeur) l'ensemble des états accessibles avec cette suite de valeurs. Cela revient à simuler l'automate avec les valeurs données. L'ensemble des états passé en paramètre sert de point d'entré dans l'automate. L'ensemble des états d'arrivé peut éventuellement être vide. |
| **nfa_end** | Cette fonction associe à un ensemble d'état et à une entrée de l'automate l'ensemble des symboles reconnue par l'automate. (L'ensemble des états est en fait l'ensemble des points d'entré dans l'automate). |

La machine contient un ensemble de propriétés invariantes permettant en partie de spécifier son comportement. Ainsi les algorithmes d'epsilon-fermetures sont spécifiés avec des propriétés liant leurs entrés à leurs résultats. De même, les algorithmes simulant l'automate sont spécifiés par les propriétés sur les trois variables nfa_end_trans, nfa_end_path et nfa_end. De cette manière, on peut utiliser ces trois variables/fonctions dans des propriétés d'ordre supérieur notamment sur le résultat de l'automate, ce qui permettra d'écrire la propriété fondamentale d'équivalence entre les automates non-déterministe et les automates déterministe associés.

Les opérations exposées par la machine sont les suivantes :

| Nom | Commentaire |
|---|---|
| **NFA_AddState** | Cette fonction permet d'ajouter à l'automate un nouvel état qui est alors renvoyé. Cet état est automatiquement numéroté à la suite des autres. |
| **NFA_AddFinal** | Cette fonction permet de spécifier un état déjà existant de l'automate comme final. De plus, la fonction permet de spécifier un identifiant pour le symbole reconnu à cet état de l'automate. |
| **NFA_AddTransition** | Cette fonction permet d'ajouter une transition entre deux états de l'automate. Comme l'automate est non-déterministe, les contraintes sur cette nouvelle transition sont assez faibles : les deux états à chaque bout de la transition ne doivent pas être marqués. |
| **NFA_AddMark** | Cette fonction permet d'ajouter une paire de marques opposées sur deux états de l'automate. Des contraintes assez fortes pèsent sur les deux états à marquer comme la structure donnée dans la partie Approche intuitive doit être respectée. |

## Spécification de l'automate déterministe

La structure de l'automate déterministe est directement calquée sur celle de l'automate non-déterministe. Ainsi, on retrouve les même variables abstraites (exceptées bien sûre toutes celles relatives aux epsilon-fermeture et aux marques). Mais on dispose toujours de la variable dfa_end représentant la simulation globale de l'automate.

De même, les fonctions exposées par la machine sont les même, il est possible d'ajouter des états et des transitions (mais pas de marques). La différence notable vient du fait qu'il est possible de construire l'automate déterministe à partir de l'automate non-déterministe. L'automate déterministe sera alors équivalent au non-déterministe. C'est la fonction DFA_ConstructFromNFA qui réalise ceci. La spécification impose outre les invariants de l'automate déterministe que l'automate construit sera équivalent à celui de départ non-déterministe. Pour imposer cette contrainte, la propriété d'équivalence demande que pour chaque entrée possible dans un automate, les deux automates reconnaissent les mêmes symboles. L'écriture de cette propriété se base sur l'utilisation des 2 variables abstraites simulant les automates nfa_end et dfa_end.

## Preuves

Les spécifications des automates ont été complètement prouvées de manière automatique. Cependant certaines difficultés ont été rencontrées :

- ➔ Au départ, les variables représentant des algorithmes comme eps_closure pour l'epsilon-fermeture ont été directement définies comme des lambda-expressions. Cette forme était pratique mais ne laissait pas beaucoup de possibilités en terme d'implémentation. Les lambda-expressions ont donc été remplacées par des groupes de propriétés se rapprochant ainsi plus d'une spécification. Mais le problème rencontré lors de la preuve avec des lambdas est que le prouveur ne possède aucune règle dessus.
- ➔ Une autre difficulté rapidement rencontrée est que le prouveur automatique ne sait pas abstraire automatiquement une expression. Il essaye toujours de décomposer les expressions rajoutant des détails alors qu'il suffirait parfois d'abstraire une partie pour retrouver une propriété connue.

### Récapitulatif des preuves du projet

| Component | TC | POG | Obv | nPO | nUn | %PR |
|---|---|---|---|---|---|---|
| **DFA** | OK | OK | 54 | 20 | 0 | 100 |
| **FA** | OK | OK | 1 | 0 | 0 | 100 |
| **NFA** | OK | OK | 91 | 29 | 0 | 100 |
| **Total** | OK | OK | 146 | 49 | 0 | 100 |

### Statut des preuves par composant

| Composant | Elément | NbObv | NbPO | NbPRi | NbPRa | NbUn | %PR |
|---|---|---|---|---|---|---|---|
| **FA** | Initialisation | 1 | 0 | 0 | 0 | 0 | 100 |
| **NFA** | Initialisation | 12 | 12 | 0 | 12 | 0 | 100 |
| **NFA** | NFA_AddState | 18 | 6 | 0 | 6 | 0 | 100 |
| **NFA** | NFA_AddFinal | 21 | 3 | 0 | 3 | 0 | 100 |
| **NFA** | NFA_AddTransition | 20 | 4 | 0 | 4 | 0 | 100 |
| **NFA** | NFA_AddMark | 20 | 4 | 0 | 4 | 0 | 100 |
| **DFA** | Initialisation | 7 | 7 | 0 | 7 | 0 | 100 |
| **DFA** | DFA_AddState | 9 | 6 | 0 | 6 | 0 | 100 |
| **DFA** | DFA_AddFinal | 12 | 3 | 0 | 3 | 0 | 100 |
| **DFA** | DFA_AddTransition | 11 | 4 | 0 | 4 | 0 | 100 |
| **DFA** | DFA_ConstructFromNFA | 15 | 0 | 0 | 0 | 0 | 100 |

## Raffinement

Le raffinement des algorithmes n'a pas été terminé mais un exemple d'implémentation est donné dans la partie suivante.

## Implémentation

On suppose ici qu'un générateur de lexer converti d'abord un ensemble d'expressions régulières vers un automate fini non-déterministe à epsilon-transition puis ce dernier vers une automate fini déterministe. On suppose également que le nouvel opérateur de restriction est représenté dans l'automate non déterministe par deux marques opposées sur deux états différents, comme décrit précédemment.

Le point important de l'implémentation concerne la conversion d'un automate non-déterministe en automate déterministe. On peut ici utiliser les algorithmes standards à l'exception de celui de l'epsilon-fermeture d'un ensemble d'état. Pour implémenter le nouvel opérateur il faut alors utiliser l'algorithme 2 utilisant les marques au lieu de l'algorithme 1. De plus, l'algorithme 2 utilise l'algorithme standard d'epsilon-fermeture rappelé dans l'algorithme 1. Dans une implémentation dans un langage particulier il faut donc retranscrire les 2 algorithmes.

Algorithme 1 : **ε-fermeture(Q)** (2 p. 35)

```
Pour tout e ∈ Q
     Empiler e dans la pile P
Initialiser ε-fermeture(Q) avec Q
Tant que P est non vide
     Dépiler P dans s
     Pour chaque état q tel qu'il existe un arc ε entre s et q
          Si q n'est pas dans l'ε-fermeture(Q)
               Ajouter q dans l'ε-fermeture(Q)
               Empiler q
```

Algorithme 2 : **ε-fermeture avec marques(Q)**

```
Initialiser ε-fermeture avec marques(Q) avec ε-fermeture(Q)
Tant qu'il existe un couple d'états (p, n) de l'ε-fermeture avec
marques(Q) tel que p est marqué par une marque positive et n est
marqué par la marque opposée.
     Calculer C l'ε-fermeture({p, n})
     Enlever les états de C à l'ε-fermeture avec marques(Q)
```

## Exemple d'application

Soit l'expression régulière suivante :

```
RES         = ('a' | 'b')* - ('aaaa' | 'bbbb') ;
```

Cette expression régulière reconnait toutes les chaînes composées de a et de b sauf les chaînes de 4 'a' ou de 4 'b' consécutifs. L'expression régulière reconnaît bien les chaînes de 5 'a' ou de 5 'b' ou plus consécutifs.

On voit ici qu'il serait complexe d'écrire une expression régulière équivalente sans utiliser l'opérateur de restriction (bien que cela soit faisable). Si les opérandes deviennent un peu plus complexe il devient rapidement impossible en pratique d'écrire de réécrire l'expression sans l'aide de l'opérateur.

```
DATA        = [a-zA-Z]* - ([a-zA-Z]* 'this' [a-zA-Z]*) ;
```

L'expression régulière DATA reconnait toutes les expressions constituées de lettres latines majuscules ou minuscules, d'une longueur quelconque éventuellement nulle, à l'exception celles contenant l'expression 'this'. Ecrire une expression équivalente serait difficile en pratique.

# Retour d'expérience sur la méthode B

## La technologie B

La méthode B permet de construire des systèmes sûrs par la formalisation du cahier des charges techniques en spécification formelle puis par le raffinement progressif et prouvé de ces dernières vers le système exécutable. Incidemment, la méthode permet d'obtenir un système conforme aux spécifications et donc aux attentes du client dès lors que le passage des spécifications informelles aux spécifications formelles a été fait avec soin. La méthode B est donc un outil puissant dont on peut n'utiliser qu'une partie, par exemple la formalisation des spécifications.

Même si le bagage mathématique requis n'est pas très important la méthode demande un réel investissement pour être vraiment bien utilisée (ce qui peut passer par une formation). En effet, l'utilisation d'un langage très proche des mathématiques pour décrire des systèmes n'est que peu naturel, même si le langage B met à disposition une certaine couche de sucre syntaxique le rendant ainsi plus accessible pour les profanes.

De plus, le langage gagnerait à être modernisé. Il est ainsi peu logique de séparer la déclaration des variables de leurs contraintes de typage ; ce qui oblige à une certaine gymnastique, au moins musculaire, si ce n'est intellectuelle. De plus, les règles de raffinement ne sont pas toujours claires (et de loin).

D'une manière générale il est dommage que l'on ne trouve quasiment pas de documentation en ligne (quelques mauvais tutoriaux ou exemples). La documentation de Clearsy sur le langage est à l'image de la technologie : formelle.

## L'outil Atelier B

L'Atelier B est le logiciel produit par Clearsy permettant d'utiliser industriellement la méthode B. Il est indéniable que l'utilisation du logiciel ne laisse pas indifférent par son côté un peu « old school ».

Concernant la partie interface, on peut apprécier son efficacité/austérité dans la gestion des projets (les contrôles se comptent sur les doigts des deux mains). Mais on pourrait regretter le manque d'information que fournie le logiciel sur les composants des projets. De plus la manière de les présenter (une liste) peut rendre l'exploration assez difficile. A contrario lors d'une preuve interactive le fait d'avoir 3 fenêtres différentes à afficher et lire en même temps peut être difficile à gérer. On peut également regretter que le logiciel se limite à un composant par fichier physique. Dans le cas d'une petite machine il aurait été judicieux d'autoriser de mettre dans le même fichier la spécification, les raffinements et l'implémentation. Cela marche peut être chez les autres mais chez moi ca ne marche pas, bien que j'ai suivi ce qui était marqué dans la documentation.

Un gros point négatif est quand même à mon avis le générateur d'obligation de preuve qui génère des expressions illisibles avec des noms de variable incompréhensibles. De plus il n'est pas possible de visualiser les obligations de preuve dans les fichiers générés.

Quant au prouveur, il se révèle efficace dans l'ensemble. On regrette par contre les noms de commandes qui paraissent assez ésotériques au début (une fois que l'on a compris la logique c'est bon). Par contre, l'utilisation se fait parfois difficile, notamment comme il est impossible de copier-coller du texte dans l'invite de commande de l'interface de preuve interactive. La base de règle du prouveur pourrait également être étendue. Elle ne contient par exemple aucune règle concernant les lambda-expressions.

En fait on aperçoit ici les limites de ce que peut faire un industriel seul sur ce genre de produit. Il pourrait être bénéfique qu'un comité scientifique maintienne de manière régulière le prouveur et sa base de règle afin de les compléter. C'est ce qui est fait sur d'autres technologies/langages comme C++, Java, C#.

De même il serait je pense assez profitable de disposer d'un véritable éditeur de texte intégré à l'Atelier, avec une coloration syntaxique et pourquoi pas une auto-complétion, et des snippets pour des éléments de langage comme les opérateurs 'pour tous' et 'il existe'. Dans le même ordre d'idée, le langage pourrait être avantageusement modifié pour le clarifier.

# Conclusion

L'opérateur de restriction dans l'algèbre des expressions régulières a bien pu être défini avec la méthode B. Il a également été possible de l'implémenter dans un générateur de parser suivant les spécifications. Ce nouvel opérateur se révèle utile pour écrire des expressions complexes.

Concernant l'utilisation de la méthode B, l'expérience est intéressante mais je crois que la méthode est trop lourde à mettre en œuvre pour qu'elle soit rentable dans une utilisation complète dans un projet dont le produit est non critique. Toutefois, l'utilisation de la partie formalisation des spécifications pourrait être plus largement répandue.

# Bibliographie

1. **AHO, Alfred V., et al.** *Compilers: Principles, Techniques, and Tools (2nd Edition).* s.l. : Addison Wesley, 2006. 978-0321486813.

2. **MOULIN, Claude.** *Théorie des Langages.* Compiègne : BUTC, 2007.