GESTION DE FICHIERS DE CONFIGURATION PAR UNE VUE ABSTRAITE MODIFIABLE

par

Francis Giraldeau

Mémoire présenté au Département d'informatique
en vue de l'obtention de la maitrise en informatique

FACULTÉ DES SCIENCES

Sherbrooke, Québec, Canada, octobre 2010

Sommaire

La gestion de fichiers de configuration sous Linux est complexe et propice aux erreurs étant donné le grand nombre de fichiers de formats différents. Toutes les techniques couramment utilisées pour modifier ces fichiers sont insatisfaisantes. Nous proposons d'abstraire la syntaxe variée des fichiers de configuration par une structure de données unique et modifiable. Nous nous intéressons aux algorithmes permettant de transformer un fichier de configuration - une chaine de caractères - en une représentation abstraite, y effectuer des modifications et reporter ces modifications dans le fichier d'origine, ce qui doit se traduire par une modification minimale du fichier. Deux logiciels qui permettent d'effectuer des transformations bidirectionnelles sur des chaines de caractères sont modifiés pour nos besoins, soit XSugar et Augeas.

XSugar fait en sorte que certains caractères peuvent être perdus lors d'une transformation aller-retour. Dans le contexte de la gestion de fichiers de configuration, il est essentiel de préserver tous les caractères présents dans la représentation concrète, mais exclus de la représentation abstraite, de manière à les restituer dans la version concrète modifiée. Nous proposons deux techniques permettant de surmonter ces limitations. Cependant, les résultats ont été partiellement atteints.

Augeas est limité dans le traitement de certains types de fichiers balisés, comme les fichiers XML. Une extension au langage adaptée à ce problème a été développée. Cette extension permet de transformer efficacement tout type de fichiers balisés. Le développement d'un module pour la configuration du serveur Web Apache démontre le succès dans l'application pratique de cette extension et constitue une première dans le domaine.

Mots clés : *gestion de configuration, programmation bidirectionnelle, problème de mise à jour par la vue, ambigüité, analyse statique, XSugar, Augeas*

Sommaire

Préface

La réalisation de cette étude a été rendue possible grâce au soutien de mon directeur de maitrise, le professeur Gabriel Girard, ainsi qu'à mon codirecteur, le professeur Richard St-Denis.

Je remercie également les créateurs des logiciels libres utilisés pour ce projet de maitrise. Premièrement, Anders Møller, professeur à l'Université d'Aarhus au Danemark, pour ses recherches sur l'ambigüité des grammaires et le logiciel XSugar. Deuxièmement, David Lutterkort, employé de RedHat à San Francisco et auteur d'Augeas, pour ses conseils techniques et théoriques.

Merci également à ma famille et mes amis qui m'ont soutenu dans cette aventure, dont Benoît des Ligneris, président de Révolution Linux, Michel Dagenais, professeur à la Polytechnique de Montréal, mes parents Denis Giraldeau et Gisèle Deschamps et mon conjoint Dominic Gauthier.

Ce mémoire est écrit selon l'orthographe rectifiée.

Abréviations

ACLA *Ambiguity Checking with Language Approximation*

API *Application Programming Interface* (interface de programmation)

AST *Abstract Syntax Tree* (arbre syntaxique abstrait)

DFA *Deterministic Finite State Automaton* (machine à états finie déterministe)

DOM *Document Object Model*

DTD *Document Type Definition*

GNU *Gnu's Not Unix*

NFA *Nondeterministic Finite State Automaton* (machine à états finie non déterministe)

PCP *Post Correspondence Problem* (problème de postcorrespondance)

PDA *Pushdown Automaton* (automate à pile)

RTN *Recursive Transition Network* (réseau de transition récursif)

XML *eXtensible Markup Language*

XSLT *XML Stylesheet Language Transformation*

Table des matières

Liste des figures

Liste des tableaux

Introduction

Un fichier de configuration sert à stocker les paramètres d'un logiciel. Le *Linux Standard Base*[1] spécifie que les fichiers de configuration système doivent se situer dans le répertoire `/etc`. Ces fichiers sont lus lors du démarrage d'une application afin de modifier son comportement sans modifier le code source, permettant ainsi d'adapter un logiciel pour un contexte d'affaires particulier.

Un exemple de fichier de configuration du serveur Web Apache apparait à la figure 1. Avec ses centaines de directives et une structure récursive, la configuration d'Apache témoigne de l'ampleur de l'espace de configuration. Sur une station de travail Linux, il peut y avoir des centaines de fichiers de configuration déterminant le comportement du système. Il existe de nombreuses syntaxes utilisées. Ces syntaxes sont le fruit d'une évolution organique dans le temps selon les besoins rencontrés pour consigner les paramètres du logiciel.

Dans ce contexte, un fichier de configuration est un type d'interface homme-machine, par lequel l'analyste entre en interaction avec le système. Or, à cause du nombre et de la diversité des formats en jeu, pour l'analyste responsable de gérer ces fichiers, il s'agit d'une tâche très complexe, propice aux erreurs [12].

Notre but est d'abstraire la syntaxe des fichiers de configuration par une vue modifiable. Le principe de fonctionnement consiste à effectuer une transformation bidirectionnelle entre le fichier d'origine (représentation concrète) et une représentation abstraite de celui-ci, par une certaine fonction $f(x)$. La vue sert d'interface et peut être modifiée par l'utilisateur. Ensuite, une fonction $g(x)$ effectue une transformation inverse de la représentation abstraite modifiée, dont les modifications se répercutent

1. http ://www.linuxfoundation.org/collaborate/workgroups/lsb

```
<IfModule !mpm_winnt.c>
    <IfModule !mpm_netware.c>
         LockFile /var/lock/apache2/accept.lock
    </IfModule>
</IfModule>

<VirtualHost *:80>
        ServerAdmin webmaster@localhost

        DocumentRoot /var/www
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        <Directory /var/www/>
                Options Indexes FollowSymLinks MultiViewsper
                AllowOverride None
                Order allow,deny
                allow from all
        </Directory>

</VirtualHost>
```

Figure 1 – Extrait de la configuration du serveur Web d'Apache

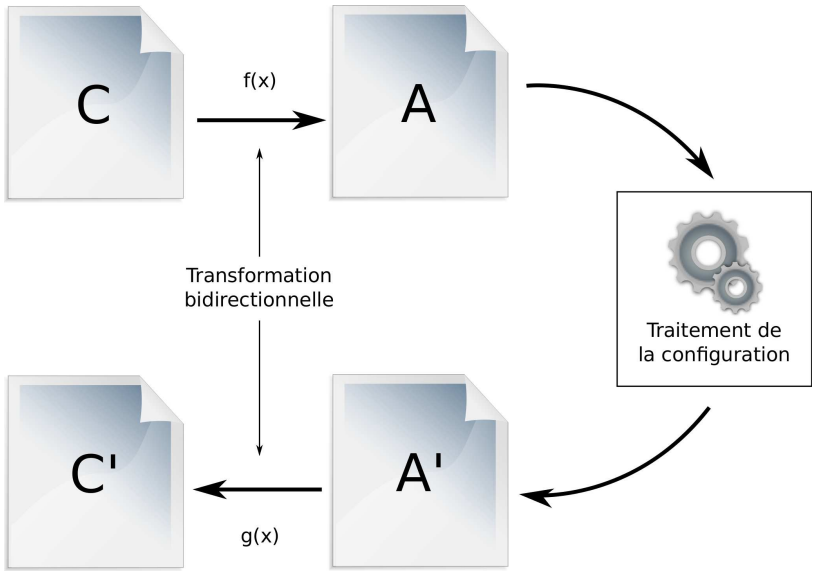


FIGURE 2 – Étapes de transformation d'un fichier de configuration

dans la représentation concrète par des modifications minimales. Les étapes du processus sont représentées à la figure 2.

Un aller-retour entre les deux représentations, sans modification, doit produire la fonction identité. Si la représentation abstraite est obtenue par $A = f(C)$ et que la représentation concrète est obtenue par $C = g(A)$, alors l'aller-retour sans modification doit correspondre à $g(f(C)) = I(C) = C$.

On cherche à minimiser la différence entre le fichier de départ et sa version modifiée obtenue après transformation. Le résultat obtenu doit être très près de celui qui serait obtenu en modifiant manuellement le fichier. Il est primordial de préserver les éléments exclus de la forme abstraite, comme l'indentation et les commentaires. Si l'indentation est réinitialisée, le résultat ne correspond pas à une modification minimale. Une modification de configuration est obscurcie par les changements liés à l'indentation. Quant aux commentaires, ils servent à documenter la configuration au même titre que ceux retrouvés dans le code source. Ils sont d'une grande valeur pour les personnes qui modifient les paramètres de configuration, d'où la nécessité de les conserver.

Une erreur syntaxique dans un fichier de configuration doit être évitée absolument, car celle-ci peut conduire à une erreur fatale causant une panne. Le formalisme

décrivant la syntaxe des fichiers de configuration doit permettre de valider statiquement que les transformations soient toujours valides, dans le but d'empêcher des transformations erronnées lors du fonctionnement.

La représentation abstraite de la configuration sous forme d'arbre d'arité et de profondeur indéfinies est la plus générale des structures de configuration. Cette structure de données garantit qu'il est possible de transformer tous les types de fichiers de configuration vers une représentation abstraite commune. En dernier lieu, il doit être facile de créer des grammaires de manière à supporter un grand nombre de formats de fichiers de configuration actuels et à venir.

Ce problème de transformation bidirectionnelle se rapporte à celui de la mise à jour par la vue (*view-update problem*) [6]. Deux logiciels libres permettant de faire des transformations bidirectionnelles sur des chaines de caractères sont étudiés, soit XSugar et Augeas.

XSugar [3] permet d'effectuer des transformations bidirectionnelles entre un format non XML et un format XML. La forme concrète est la représentation non XML, tandis que la représentation abstraite est le document XML sous forme de DOM en mémoire. Les modifications sont effectuées sur le document XML représentant la configuration. En pratique, les bibliothèques permettant d'analyser et de modifier le XML sont largement répandues. Il est à noter que le fonctionnement de XSugar utilise la représentation sérialisée du XML dans son fonctionnement. La correspondance entre les représentations se fait par un arbre syntaxique abstrait commun. L'arbre syntaxique est commun parce qu'il existe une association entre les règles de production des grammaires des représentations non XML et XML. XSugar permet de valider statiquement la relation bidirectionnelle par l'analyse de l'ambigüité des grammaires.

La relation bidirectionnelle de XSugar correspond à la fonction identité, *modulo* certains caractères. Ceci signifie qu'il existe des situations où il se produit une perte d'information pendant une transformation, ce qui doit être évité dans le contexte de la gestion des fichiers de configuration. Nous avons développé des techniques permettant d'assurer la préservation de tous les caractères pour une feuille de style quelconque, ce que nous appellons la bidirectionnalité stricte.

Quant à Augeas, il s'agit d'un logiciel basé sur un principe de programmation bidirectionnel [7]. La représentation abstraite modifiable se présente sous forme d'arbre

en mémoire, dont chaque élément est composé d'un identifiant et d'une valeur optionnelle. Les opérations élémentaires sont réalisées par des lentilles primitives. Chaque lentille primitive définit le comportement de la transformation dans les deux directions, soit de la représentation concrète vers l'abstraite et inversement. Les opérateurs d'union, de concaténation et de répétition permettent de combiner des lentilles tout en préservant leurs propriétés. Les lentilles sont vérifiées statiquement pour la présence d'ambigüité, de telle sorte que leur comportement est garanti lors de leur utilisation.

La principale limite d'Augeas concerne le support de certains langages balisés, comme le XML et la configuration d'Apache. Nous proposons une extension au langage pour ajouter une nouvelle lentille ayant les fonctions de transformation adaptées pour traiter ces fichiers.

Le mémoire débute par l'étude de la problématique au chapitre 1, suivi des préliminaires au chapitre 2, portant sur la théorie requise pour aborder les analyses subséquentes. L'analyse, les développements et les résultats obtenus pour XSugar et Augeas sont présentés au chapitre 3 et au chapitre 4 respectivement. Le mémoire se termine par une discussion sur les travaux futurs et la conclusion.

Chapitre 1

État de l'art

Les quatre techniques courantes pour la gestion automatique de la configuration sont revues dans ce chapitre. Le principe de fonctionnement de ces techniques, leurs avantages et leurs inconvénients sont présentés. Il est montré qu'aucune de ces techniques ne répond aux objectifs d'automatisation recherchés. Les techniques de ce chapitre ne font pas appel à la mise à jour par une vue abstraite, comme XSugar et Augeas, qui font l'objet des chapitres subséquents.

1.1 Base de données

Une représentation standard de la configuration peut être obtenue par une base de données de paramètres. Le registre sous Microsoft Windows^{TM}remplit cette fonction. Il s'agit d'un arbre de clés et de valeurs accompagné d'un API pour y effectuer des opérations. Sous Linux, Elektra [1] est un exemple d'une implémentation semblable au registre Windows. Cependant, pour des raisons historiques et à cause du mode de développement distribué des logiciels libres, où les standards ont de la difficulté à s'imposer, les fichiers de configuration demeurent la méthode privilégiée pour consigner les paramètres des applications sous Linux. Même sous Windows, rien n'empêche une application de créer son propre fichier de configuration plutôt que d'utiliser le registre.

1. http ://elektra.g4ii.com

Ainsi, nous visons à obtenir la même vue unifiée que celle d'une base de données, sans forcer la structure de la couche de persistance des données.

1.2 Gabarits

Un gabarit est un document permettant de séparer le contenu de sa présentation. Le gabarit contient des variables qui sont substituées par des valeurs concrètes. Le schéma de la figure 1.2 montre un exemple de gabarit *Django* [2]. Le gabarit est constitué d'une boucle `for` qui itère sur chaque item de la variable `params`. La sortie est une chaine comportant un couple clé-valeur par ligne.

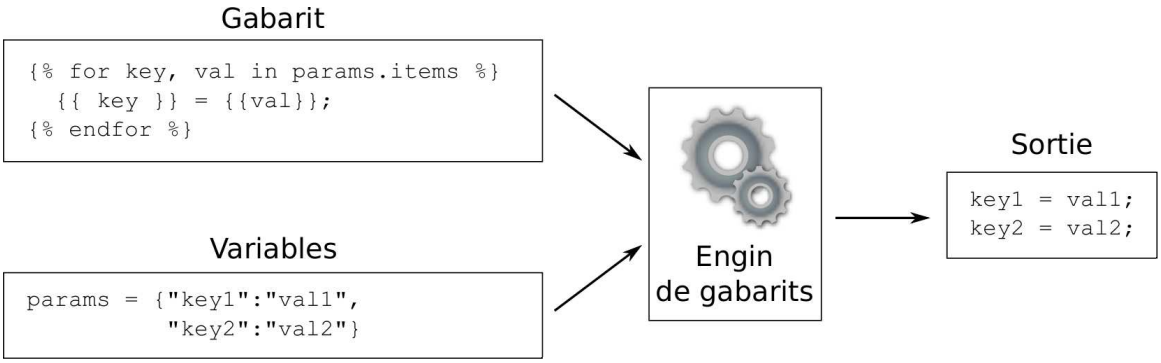


Figure 1.1 – Principe de fonctionnement d'un gabarit

Les variables peuvent provenir d'une base de données, et la plupart des engins de gabarits permettent de créer des boucles et des embranchements conditionnels et supportent même l'héritage et la surcharge de blocs de texte entre plusieurs gabarits, ce qui rend cette solution très flexible.

Cependant, la modification d'une valeur au fichier nécessite de régénérer le fichier en entier et de remplacer la version courante. Cette technique est donc à sens unique. Si une modification manuelle est effectuée au fichier en place, elle est perdue lorsque le fichier est de nouveau généré à partir de l'engin de gabrits. Ainsi, tout autre modification à la configuration entre en conflit avec cette technique.

2. http ://www.djangoproject.com

1.3 Scripts

L'utilisation de scripts représente une manière courante de manipuler des fichiers de configuration. L'utilisation d'outils tels que `perl`, `grep`, `awk`, `sed`, font partie de la culture Unix depuis son invention.

Un exemple d'un script `Shell` modifiant un fichier de configuration est présenté à la figure 1.2. Son exécution ajoute le paramètre `key1 = value1` à la fin d'un fichier de configuration simple de type clé-valeur.

Or, ce script ne teste pas si la ligne est déjà présente avant de l'ajouter. En conséquence, une ligne redondante est ajoutée à chaque exécution, ce qui n'est pas approprié. Il ne permet pas non plus d'annuler son action. Une version plus complète du script comblant ces lacunes est représentée à la figure 1.3.

```
#!/bin/sh
echo "key1 = value1" >> gazou.conf
```

Figure 1.2 – Script minimaliste modifiant la configuration

Cette nouvelle version prend en paramètre l'option `enable` ou `disable`. La fonction d'activation ajoute la ligne au fichier. Si l'option est trouvée par `grep`, le script ne fait rien. La fonction de désactivation retire la ligne et ne fait rien si elle n'est pas trouvée par `grep`.

Toutefois, ce script produit une erreur si une clé contient la sous-chaine de recherche. Par exemple, si la clé `somekey123`, qui contient la sous-chaine `key1`, est présente dans le fichier, alors la fonction `enable` n'ajoute pas le paramètre, tandis que la fonction `disable` supprime un autre paramètre que celui désiré.

Cet exemple est délibérément simple pour mettre en évidence un effet de bord que peut occasionner l'utilisation de scripts. Il serait possible de corriger à nouveau le script pour gérer ces cas. Ceci met en évidence qu'il est difficile de tester un tel script pour toutes les entrées auxquelles il pourrait être soumis. Un script se résume en un analyseur syntaxique *ad hoc* qui fonctionne uniquement dans certains cas particuliers. Le fait que le script soit destiné à un contexte particulier diminue son potentiel de réutilisation.

```
#!/bin/sh
conf="gazou.conf"
key="key1"
value="value1"
test -f $conf || touch $conf
case "$1" in
    enable)
        if [ -z "$(grep $key $conf)" ]; then
            echo "$key = $value" >> $conf
        fi
    ;;
    disable)
        cat $conf | grep -v $key > tmpfile.conf
        mv tmpfile.conf $conf
    ;;
    *)
        echo "$0 [enable | disable]"
        exit 1
    ;;
esac
exit 0
```

Figure 1.3 – Script amélioré modifiant la configuration

1.4 Bibliothèques spécialisées

Il existe des bibliothèques permettant de lire et modifier des fichiers dans certains formats utilisés pour les fichiers de configuration. Le cas du format `INI` est présenté, puisqu'il s'agit d'un format courant pour lequel il existe plusieurs implémentations de bibliothèques.

Le format `INI` est composé de sections débutant par un identifiant entre crochets, tel que `[section]`. Sous ces en-têtes se trouvent des paires de clé-valeur séparées par un caractère d'égalité, tel que `key=value`. Les commentaires sont des lignes débutant par le caractère dièse (`#`) ou le point virgule (`;`). Un exemple de cette syntaxe est représenté à la figure 1.4.

Il existe de nombreuses implémentations d'analyseurs syntaxiques pour ce type de fichier. La bibliothèque `ini4j`[3] a été choisie pour vérifier ses propriétés bidirection-

3. http ://ini4j.sourceforge.net

```
; commentaire
[section1]
    key1 = value1
```

Figure 1.4 – Exemple de fichier de configuration `INI`

nelles. Le code de la figure 1.5 teste la lecture et l'écriture du fichier sans effectuer de modification.

```
public class TestINI {
    @Test
    public void testIniLoad() throws BackingStoreException,
                    InvalidIniFormatException,
                    FileNotFoundException,
                    IOException {
        Ini ini = new Ini();
        ini.load(new FileReader("src/ca/udes/input.ini"));
        FileWriter w = new FileWriter("src/ca/udes/output.ini");
        ini.store(w);
    }
}
```

Figure 1.5 – Code testant la bibliothèque `ini4j`

La différence entre l'entrée et la sortie est calculée avec l'utilitaire `diff`, et est présentée à la figure 1.6. On constate que le contenu est préservé, puisque la section ainsi que la paire de clé-valeur apparaissent dans la sortie. Cependant, le commentaire est perdu et l'indentation de la paire de clé-valeur est réinitialisée. On conclut donc que cette bibliothèque est bidirectionnelle *modulo* le formatage et les commentaires, ce qui ne correspond pas à la fonction identité.

Dans le cas des fichiers de configuration, une bibliothèque est nécessaire pour chaque type de fichier. Or, l'interface varie selon le type de fichier de configuration. Par exemple, l'API pour modifier un fichier au format `INI` ne correspond pas à l'API utilisé pour modifier un fichier XML, ce qui complexifie la réalisation d'une abstraction de ces fichiers.

Finalement, le nombre de formats est limité aux bibliothèques disponibles, qui le

```
$ diff -ru input.ini output.ini
--- input.ini   2010-01-25 13:49:03.631307148 -0500
+++ output.ini  2010-01-25 13:50:03.844053328 -0500
@@ -1,4 +1,3 @@
-; commentaire
 [section1]
+key1 = value1

-    key1 = value1
```

Figure 1.6 – Différence entre le fichier d'origine et celui traité par `ini4j`

sont uniquement pour les formats majeurs ou normés. Écrire une telle bibliothèque requiert un important investissement. La version 0.5.1 de la bibliothèque `ini4j` à elle seule est constituée d'environ 10 000 lignes de code, tel que rapporté par `sloccount`. On compte des dizaines de formats différents sous Linux, les supporter tous représente un effort colossal.

Chapitre 2

Préliminaires

Le traitement des fichiers de configuration fait appel aux techniques d'analyse des chaines de caractères. Ces techniques varient en fonction de la classe de langages à laquelle appartient le langage de configuration. La hiérarchie des langages de Chomsky comprend quatre classes. On s'intéresse en particulier aux classes des langages réguliers et hors contexte. La classe des langages hors contexte est un surensemble propre de la classe des langages réguliers. L'ensemble des langages hors contexte contient tous les fichiers de configuration à traiter, tel que montré à la figure 2.1. La figure contient deux exemples de fichiers de configuration. Le premier est `smb.conf`, le fichier de configuration de Samba, qui appartient à la classe des langages réguliers. Le second est `httpd.conf` d'Apache qui appartient à la classe des langages hors contexte.

Pour débuter, la classe des langages réguliers et celle des langages hors contexte sont présentées. Pour ces deux classes, les algorithmes permettant de les traiter sont introduits. Le résumé des concepts présentés dans cette section provient du livre *Languages and Machines* [10], auquel le lecteur peut faire référence au besoin pour plus de détails. La section suivante porte sur la détection de l'ambigüité des grammaires hors contexte, notion essentielle à la vérification statique des transformations bidirectionnelles.

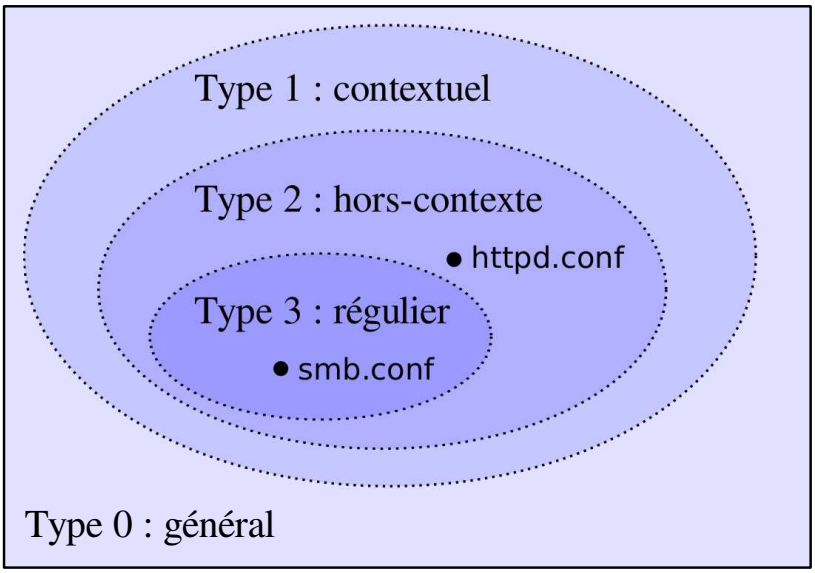


Figure 2.1 – Hiérarchie de Chomsky appliquée à la problématique

2.1 Langages réguliers

La classe des langages réguliers est la plus restreinte de la hiérarchie de Chomsky. Il s'agit de la classe de langages reconnus par une machine à états finie (aussi appelé automate), dont la définition apparait en 2.1.1.

Définition 2.1.1 (Machine à états finie déterministe (DFA))**.** Une machine à états finie déterministe (DFA) est un quintuple $M = (\Sigma, S, s_0, \delta, F)$, où Σ est un ensemble fini de symboles (alphabet), S un ensemble fini d'états, $s_0 \in S$ est l'état initial, δ est un ensemble de transitions $S \times \Sigma \rightarrow S$ et F est un ensemble d'états finaux (acceptants).

Une DFA peut être représentée sous la forme d'un graphe orienté, dont les états sont les noeuds et les transitions sont les arcs. Les états finaux sont dénotés avec un cercle double, tandis que les autres états ont seulement un trait simple. On peut également représenter une DFA avec une matrice $\Sigma \times S$, dans laquelle chaque colonne correspond à un symbole de l'alphabet et chaque ligne correspond à un état, et dont les éléments correspondent au prochain état.

Le fonctionnement pour reconnaitre une chaine d'entrée est le suivant. Au départ, la machine est dans l'état initial. Les symboles de la chaine d'entrée sont lus un à un. À chaque symbole, la transition qui accepte ce symbole est sélectionnée. La machine s'arrête lorsque tous les symboles de l'entrée sont lus ou si aucune transition ne correspond au symbole d'entrée courant. La chaine est acceptée si l'entrée est lue complètement et que la machine est dans un état final.

Afin de décrire un langage régulier, il est courant d'utiliser une expression régulière. La réalisation d'une DFA à partir d'une expression régulière peut s'effectuer avec la construction de Thompson [13]. Cependant, cette technique requiert une machine à états finie non déterministe.

Définition 2.1.2 (Machine à états finie non déterministe (NFA))**.** Une machine à états finie non déterministe (NFA) est un quintuple $M = (\Sigma, S, s_0, \delta, F)$, où Σ, S, s_0 et F sont les mêmes qu'une DFA. La fonction de transition δ est différente et est définie par $S \times (\Sigma \cup \epsilon) \to \mathcal{P}(S)$.

La première différence entre une DFA et une NFA est dans l'ajout du symbole spécial ϵ aux symboles de la machine. Ce symbole ne fait pas partie de l'alphabet, mais correspond plutôt à une transition spontanée, qui se produit sans consommer de symbole d'entrée. La deuxième différence est que, pour un état et un symbole, il peut exister plusieurs états cibles. Lorsqu'une telle alternative se présente dans l'exécution de la machine, toutes les possibilités doivent être considérées, soit de manière parallèle ou soit par une technique de retour en arrière.

Une DFA est déterministe si elle remplit trois conditions :

- L'alphabet ne contient pas le symbole ϵ.
- Pour chaque paire $S \times \Sigma$ il existe au plus une transition.
- Il n'existe qu'un seul état initial.

Pour chaque NFA, il existe un algorithme permettant d'obtenir une DFA acceptant le même langage. La version déterministe de la machine présente l'avantage qu'à chaque état ne correspond qu'une transition possible, ce qui évite d'explorer tous les chemins possibles. Le temps d'exécution résultant est linéaire par rapport à la longueur de la chaine d'entrée à reconnaitre.

Pour un langage régulier $\mathcal{L}$, il existe une machine déterministe minimale unique,

obtenue par une procédure de minimisation, qui consiste à fusionner les états équivalents.

2.1.1 Représentation d'un langage régulier

Une expression régulière est une représentation compacte d'un langage régulier. Dans ce mémoire, les expressions régulières sont décrites avec la notation POSIX [1].

- L'union de deux expressions régulières, dénotée par la barre verticale `|`, accepte l'une ou l'autre des alternatives.
- La concaténation de deux expressions régulières, dénotée par la juxtaposition, accepte la concaténation des chaines de la première expression régulière et de la seconde.
- La répétition d'une expression régulière accepte les chaines consécutives appartenant au langage, ce qui correspond à une forme de concaténation. L'étoile (`*`) signifie zéro, une ou plusieurs répétitions (aussi appelé étoile Kleene), le point d'interrogation (`?`) signifie zéro ou une répétition et le symbole plus (`+`) signifie une ou plusieurs répétitions.
- Les parenthèses sont utilisées pour faire des groupements et pour forcer la priorité des opérateurs.
- Un groupe de caractères est formé par une expression entre crochets. Par exemple, le groupe `[xy]` accepte le caractère `x` ou `y`.
- Une plage de caractères est résumée par une expression dont le premier et le dernier caractère de la suite sont séparés par un tiret. Par exemple, l'alphabet des lettres ASCII minuscules est dénoté par le groupe `[a-z]`.
- Le complément d'un groupe de caractères est dénoté par le circonflexe (`^`). Par exemple, le groupe `[^a]` accepte tous les caractères sauf un `a`.
- Le point correspond au caractère de remplacement (*wildcard*).

Exemple 2.1.1. L'expression régulière $(a|bc)^*$ accepte les chaines

$$\{\epsilon, a, bc, aa, abc, bca, bcbc, ...\}$$

1. http ://www.opengroup.org/onlinepubs/007908799/xbd/re.html

Le symbole ϵ dénote la chaine vide. L'expression régulière est représenté par l'automate de la figure 2.2.

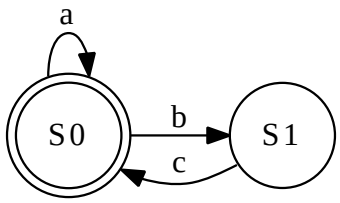


Figure 2.2 – Machine à états finie du langage $(a|bc)^*$

2.1.2 Construction de Thompson

Concrètement, une expression régulière doit être compilée sous forme d'un DFA pour reconnaitre une chaine. La construction de Thompson permet d'obtenir une DFA à partir d'une expression régulière. La procédure se décompose en trois étapes. La première consiste à décomposer l'expression régulière en automates élémentaires, puis à les assembler à l'aide de transitions ϵ. L'automate résultant est non déterministe.

La deuxième étape consiste à déterminiser l'automate. Le résultat n'est cependant pas nécessairement un automate minimal. La dernière étape consiste à appliquer l'algorithme de minimisation. Il est à noter que les deux dernières étapes ne sont pas obligatoires, puisqu'il est possible d'exécuter une NFA. Cependant, la déterminisation et la minimisation permettent de diminuer la complexité d'exécution de la DFA respectivement en temps et en espace.

2.1.3 Opérateur d'intersection

Le calcul d'intersection entre les langages réguliers $\mathcal{L}_1$ et $\mathcal{L}_2$ permet de déterminer s'il existe une chaine commune aux deux langages. Cette opération est utile pour déterminer si deux langages réguliers sont disjoints.

Les langages réguliers sont clos pour les opérateurs d'union, de concaténation, de répétition, de complément et d'intersection. Ceci signifie que ces opérateurs prennent en entrée un langage régulier et leur résultat est aussi un langage régulier. Cette

propriété permet de calculer l'intersection de deux langages à partir des opérateurs d'union et de complément.

Exemple 2.1.2. Soit un alphabet composé des symboles a,b et c et les langages réguliers $\mathcal{L}_1 = (a|b)^*$ et $\mathcal{L}_2 = (b|c)^*$. Le langage $\mathcal{L}_3$ correspondant à $\mathcal{L}_1 \cap \mathcal{L}_2 = \overline{(\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2})}$.

$$\begin{aligned}
\mathcal{L}_1 \cap \mathcal{L}_2 &= \overline{\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2}} \\
&= \overline{\overline{(a|b)^*} \cup \overline{(b|c)^*}} \\
&= \overline{(a|b)^*(c)(a|b|c)^* \cup (b|c)^*(a)(a|b|c)^*} \\
&= \overline{b^*(a|c)(a|b|c)^*} \\
&= b^*
\end{aligned}$$

Ainsi, les chaines $\{\epsilon, b, bb, bbb, ...\}$ font parties des deux langages. Ceci est représenté graphiquement à la figure 2.3.

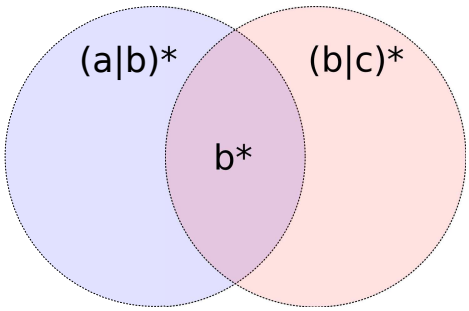


Figure 2.3 – Intersection des langages $(a|b)^*$ et $(b|c)^*$ pour l'alphabet $\{a, b, c\}$

Deux langages sont disjoints si leur intersection est nulle. Un exemple de chaine appartenant aux deux langages peut être trouvé en exécutant en parallèle les deux DFA qui les représentent. Aussitôt que les deux automates sont simultanément dans un état final, alors le chemin parcouru dans l'un ou l'autre des graphes représente la plus petite chaine commune acceptée par les deux automates.

2.2 Langages hors contexte

Soit le langage $\mathcal{L} = \{a^i b^i | n \geq 0\}$ générant les chaines $\{\epsilon, ab, aabb, aaabbb, ...\}$. Ce langage est particulier par le nombre égal de a et de b. La DFA de ce langage est représentée à la figure 2.4. Or, comme n n'est pas borné, il faut un nombre infini d'états pour compter le nombre de symboles rencontrés, ce qui entre en contradiction avec la définition d'une machine à états *finie*.

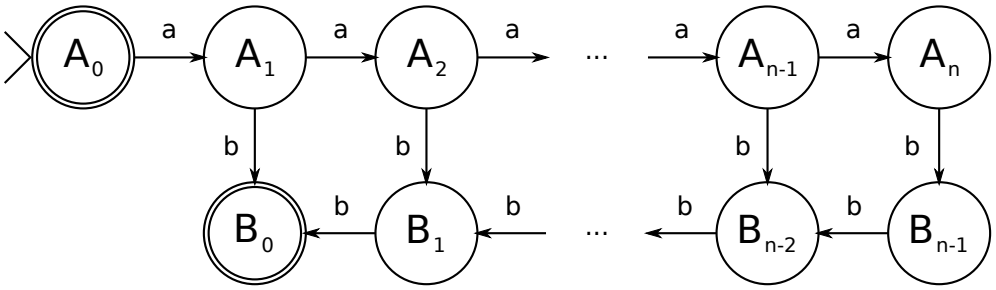


Figure 2.4 – DFA du langage $a^i b^i$

Le langage $a^i b^i$ appartient à la classe des langages hors contexte. Cette classe est plus générale que celle des langages réguliers. Elle correspond exactement aux langages reconnus par un automate à pile. Les symboles de la pile, ajoutés ou retirés lors des transitions de l'automate, conservent un état correspondant à l'historique des transitions. Cet état, combiné au symbole d'entrée, peut servir à déterminer quelle transition effectuer.

Définition 2.2.1. Un automate à pile est un sextuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, où Q est un ensemble fini d'états, Σ est un ensemble fini de symboles d'entrée (alphabet), Γ est un ensemble fini de symboles de la pile, q_0 l'état initial, $F \subseteq Q$ est un ensemble d'états finaux (acceptants) et δ est une fonction de transition $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$.

Un automate à pile acceptant le langage $a^i b^i$ est représenté à la figure 2.5. La différence avec une DFA est la présence des opérations de pile sur les transitions de l'automate. Pour la modification de la pile A/B, le numérateur et le dénominateur indiquent l'ajout et le retrait respectivement. Le symbole ϵ ne modifie pas la pile. La

chaine est acceptée si, lorsque tous les caractères sont traités, la machine est dans un état final et que la pile est vide.

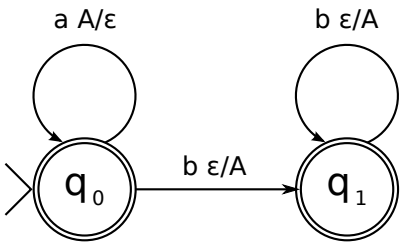


Figure 2.5 – Automate à pile du langage $a^i b^i$

Exemple 2.2.1. L'exécution de l'automate de la figure 2.5 pour la chaine *aaabbb* donne :

Pile	État	Entrée
$\{\}$	q_0	$aaabbb$
$\{A\}$	q_0	$aabbb$
$\{AA\}$	q_0	$abbb$
$\{AAA\}$	q_0	bbb
$\{AA\}$	q_1	bb
$\{A\}$	q_1	b
$\{\}$	q_1	ϵ

La machine est dans l'état final q_1 et la pile est vide, donc la chaine est acceptée.

2.2.1 Grammaires hors contexte

Une grammaire hors contexte permet de décrire les langages hors contexte, comme le langage $a^i b^i$, en plus des langages réguliers.

Les variables de la grammaire représentent les constructions syntaxiques. Celles-ci sont dérivées de manière itérative à partir du symbole de départ jusqu'à correspondre aux symboles terminaux de l'entrée. La séquence de dérivation produit un arbre de dérivation, duquel des traitements subséquents sont réalisés selon l'application.

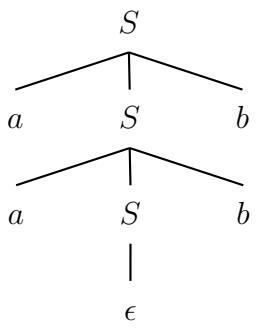
Définition 2.2.2 (Grammaire hors contexte)**.** Une grammaire hors contexte est un quadruplet $G = (V, \Sigma, P, S)$, où V est un ensemble fini de variables (symboles non terminaux), Σ est un ensemble fini de symboles terminaux, P est un ensemble fini de règles de production de la forme $V \to \mathcal{P}(E^*)$, où $E = \Sigma \cup V$ et $S \in V$ est le symbole de départ.

Une chaine $w \in (V \cup \Sigma)^*$ est une protophrase de G. On note l'application d'une dérivation $xAy \Rightarrow xwy$ pour une règle $A \to w \in P$. L'opérateur $\overset{*}{\Rightarrow}$ est la clôture transitive, indiquant que la protophrase est dérivable dans un nombre fini de dérivations. Le langage engendré par G est $\mathcal{L}(G) = \mathcal{L}_G(S)$.

Soit la chaine $x \in \mathcal{L}(G)$ telle que $S = \phi_0 \Rightarrow \phi_1 \Rightarrow ... \Rightarrow \phi_n = x$. Cette séquence de dérivation produit un arbre de dérivation, dont les noeuds internes sont étiquetés par les variables de V, le noeud racine est étiqueté par S et les feuilles sont étiquetées par les symboles de Σ. Dans une grammaire, le symbole spécial ϵ peut apparaitre seul dans le membre de droite. Il s'agit alors d'une règle d'effacement.

Exemple 2.2.2. Soit la grammaire G en a) engendrant le langage $a^n b^n$ et pour $n = 2$ l'arbre de dérivation obtenu suite à la substitution en b).

a) $S \to aSb \mid \epsilon$ b)



2.2.2 Propriétés de clôture

L'ensemble des grammaires hors contexte est clos pour les opérations d'union, de concaténation et de répétition Kleene. Les grammaires hors contexte ne sont cependant pas closes pour l'opération d'intersection et de complément.

Soit une grammaire $G_1 = (V_1, \Sigma_1, P_1, S_1)$ et $G_2 = (V_2, \Sigma_2, P_2, S_2)$.

- Union : $G_1 \cup G_2 = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}, S)$
- Concaténation : $G_1 G_2 = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$

– Répétition Kleene : $G_1^* = (V_1, \Sigma_1, P_1 \cup \{S \to S_1 S | \epsilon\}, S)$

Théorème 2.2.1 (Clôture de l'intersection des grammaires hors contexte)**.** L'ensemble des grammaires hors contexte n'est pas clos pour l'opération d'intersection.
Preuve. Soit les deux grammaires hors contexte $L_1 = \{a^i b^i c^j | i, j \geq 0\}$ et $L_2 = \{a^j b^i c^i | i, j \geq 0\}$. L'intersection $L_1 \cap L_2 = \{a^i b^i c^i | i \geq 0\}$. Le résultat n'appartient pas à l'ensemble des langages hors contexte, mais plutôt à celui des langages contextuels. La figure 2.1 montre que la classe des langages contextuels de la hiérarchie de Chomsky est un surensemble de la classe des langages hors contexte. En conséquence, les langages hors contexte ne sont pas clos pour l'opération d'intersection.

2.3 Approximation régulière

Nederhof [9] a proposé plusieurs techniques pour obtenir un langage régulier approximant le langage d'une grammaire hors contexte. La technique basée sur les réseaux de transition récursifs est présentée ici.

Définition 2.3.1 (Réseau de transition récursif (RTN))**.** La définition 2.1.2 d'un NFA s'applique à un réseau de transition récursif, à l'exception de la fonction de transition, dont l'ensemble de symboles de transition est augmenté par un ensemble de symbole $\mathcal{N}$ ne faisant pas partie de l'alphabet.

Un RTN est construit à partir d'une grammaire hors contexte de la manière suivante. Pour chaque variable A, deux états q_A et q'_A sont ajoutés. Ceux-ci sont l'état initial et final de l'automate. Un nombre $m + 1$ d'états $q_0, ..., q_m$ sont créés pour chaque symbole terminal et variable du membre de droite de la règle $A \to X_1, ...X_m$. Pour le membre de gauche de A, la transition (q_A, ϵ, q_0) est ajoutée. Pour le membre de droite, une transition (q_{i-1}, X_i, q_i) est créée pour chaque i tel que $1 \leq i \leq m$. Pour compléter l'automate, la transition (q_m, ϵ, q'_A) est ajoutée. Le réseau complet est obtenu par l'ensemble des automates de chaque règle de production.

Le NFA approximatif est obtenu en remplaçant les transitions des variables (q, A, q') par deux transitions (q, ϵ, q_A) et (q'_A, ϵ, q'), ce qui a pour effet d'éliminer les règles récursives de la grammaire.

Exemple 2.3.1. Soit la grammaire des palindromes

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$$

Le RTN représentant S est présenté à la figure 2.6 en a). Les transitions ϵ triviales ont été simplifiées et le remplacement des transitions correspondant aux variables de la grammaire a été effectué. L'automate déterministe minimal généré par S est présenté en b), et correspond à l'expression régulière $(a|b)^*$. En substituant cette dernière dans le membre de droite de la règle $S \rightarrow aSa$, l'expression régulière $a(a|b)^*a$ est obtenue. L'expression régulière $b(a|b)^*b$ correspondant à règle $S \rightarrow bSb$ est obtenu de manière similaire.

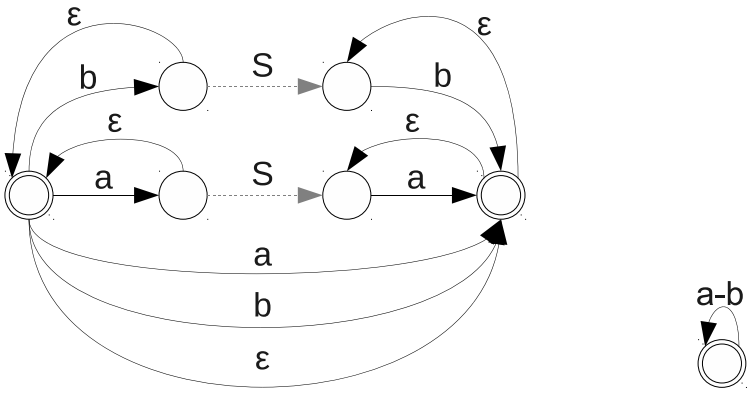
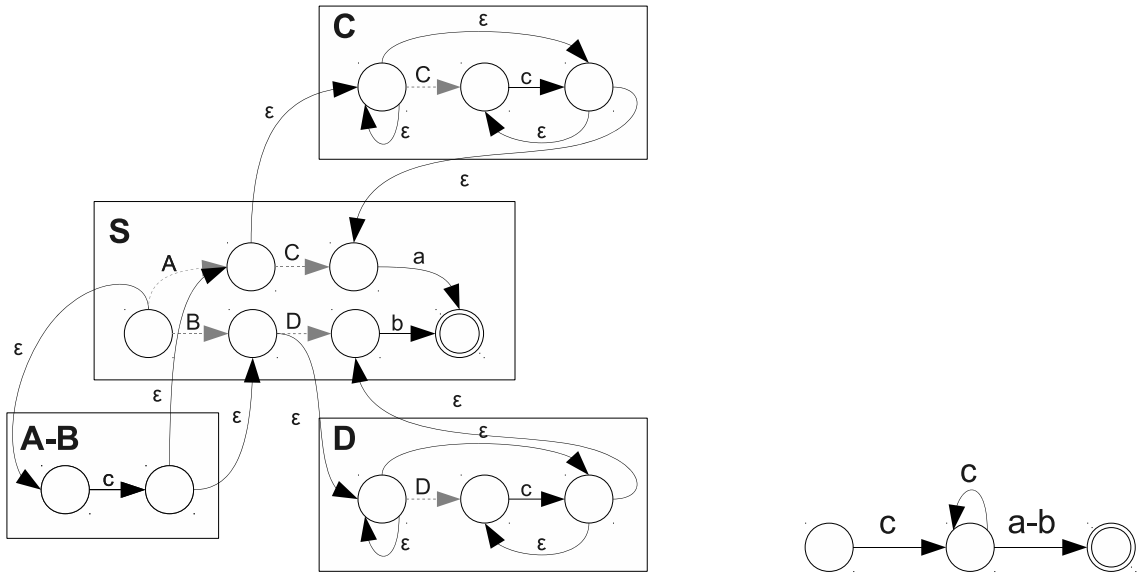


Figure 2.6 – RTN de la grammaire de l'exemple 2.3.1

Exemple 2.3.2. Soit la grammaire 2.1. Le réseau de transition récursif et l'automate résultant sont montrés à la figure 2.7 en a) et b) respectivement. On trouve donc que cette grammaire produit le langage régulier approximatif $\mathcal{L}_{approx} = c^+a \mid c^+b$, car les règles A et B forcent le symbole c, les règles C et D correspondent à une répétition Kleene de symboles c et les symboles a ou b doivent terminer la chaine.

$$
\begin{array}{ll}
(1) & S \rightarrow ACa \\
(2) & S \rightarrow BDb \\
(3) & A \rightarrow c \\
(4) & B \rightarrow c \\
(5) & C \rightarrow Cc \\
(6) & C \rightarrow \epsilon \\
(7) & D \rightarrow Dc \\
(8) & D \rightarrow \epsilon
\end{array}
\tag{2.1}
$$



a) b)

Figure 2.7 – RTN de la grammaire de l'exemple 2.3.2

L'analyse syntaxique consiste à retrouver la structure en arbre à partir d'une chaine de symboles. On distingue deux types d'analyse syntaxique, selon qu'elle soit déterministe ou non déterministe.

L'analyse déterministe est plus simple que l'analyse non déterministe, puisqu'à chaque symbole d'entrée correspond uniquement une transition possible. Les grammaires pouvant être analysées de la sorte font partie de l'ensemble des grammaires hors contexte déterministes.

Donald E. Knuth a introduit l'ensemble des grammaires déterministes $LR(k)$ pouvant être analysées de manière déterministe avec k symboles d'anticipation [8]. Un analyseur syntaxique $LR(k)$ traite les symboles d'entrée de gauche à droite *(Left to right)* par des dérivations à droite *(Rightmost derivation)* en temps linéaire. En augmentant le nombre de symboles d'anticipation k, un plus grand ensemble de grammaires peuvent être analysées de manière déterministe. Cependant, il n'existe aucune valeur de k permettant d'analyser l'ensemble complet des grammaires hors contexte.

La dérivation d'une chaine peut également se faire de manière non déterministe. L'algorithme Earley [5] appartient à cette catégorie et traite l'ensemble des grammaires hors contexte. L'analyseur syntaxique Earley retourne toutes les dérivations possibles d'une chaine par rapport à une grammaire hors contexte. Il fonctionne en développant une forêt représentant chaque arbre syntaxique possible selon les entrées lues. L'algorithme s'exécute en temps linéaire pour les grammaires hors contexte déterministes et est borné à $O(n^3)$ pour une grammaire quelconque.

Réaliser en pratique une grammaire déterministe pour un langage particulier est plus difficile, étant donné qu'il y a plus de restrictions à respecter pour préserver la propriété déterministe. L'analyse non déterministe permet plus de flexibilité dans la réalisation de la grammaire, et donc cette approche est à privilégier. L'inconvénient est que, contrairement à un analyseur déterministe, les grammaires ambigües sont acceptées, ce qui doit être évité. Ce problème est discuté dans la prochaine section.

2.4 Ambigüité

Certaines grammaires ne possèdent pas de relation bijective entre le langage accepté et l'arbre syntaxique obtenu. Bien que chaque arbre syntaxique produise une

chaine dans le langage, il se peut qu'une chaine donne lieu à plusieurs arbres syntaxiques.

Définition 2.4.1 (Ambigüité d'une grammaire hors contexte)**.** Une grammaire hors contexte $G = (V, \Sigma, P, S)$ est ambigüe s'il existe une chaine $w \in \Sigma^*$ qui est dérivable à l'aide de plusieurs dérivations à gauche (L) distinctes $S \overset{*}{\Rightarrow}_L w$.

Le but de la validation statique des transformations bidirectionnelles consiste à prouver que les transformations produiront un résultat valide pour l'ensemble des chaines du langage engendré par une grammaire. Cette étape de vérification apporte une garantie de comportement à l'exécution, ce qui s'apparente à la vérification des types d'un langage de programmation pour détecter les erreurs avant même la première exécution du programme.

Intuitivement, si l'entrée peut être interprétée de plusieurs manières, alors il peut exister plusieurs transformations applicables. En présence d'ambigüité dans la description du langage, il n'est pas possible de garantir la bijectivité des transformations bidirectionnelles. Le défi de la validation statique des transformations consiste donc à détecter les ambigüités de la grammaire. Seuls les langages non ambigus sont considérés dans cette analyse.

2.4.1 Détection de l'ambigüité

Le problème de l'ambigüité des grammaires hors contexte consiste à déterminer si une grammaire est ambigüe, et est non décidable selon le théorème 2.4.1.

Théorème 2.4.1 (Non décidabilité du problème de l'ambigüité des grammaires hors contexte)**.** Il n'existe pas d'algorithme capable de déterminer si une grammaire est ambigüe ou non ambigüe.

Le théorème 2.4.1 est appuyé par une preuve reposant sur la réduction du problème de postcorrespondance (PCP) au problème de l'ambigüité des grammaires hors contexte. S'il existe un algorithme pour le problème de l'ambigüité des grammaires hors contexte, alors un tel algorithme s'applique pour le problème de postcorrespondance. Or, il est établi que le problème de postcorrespondance est non décidable, ce qui

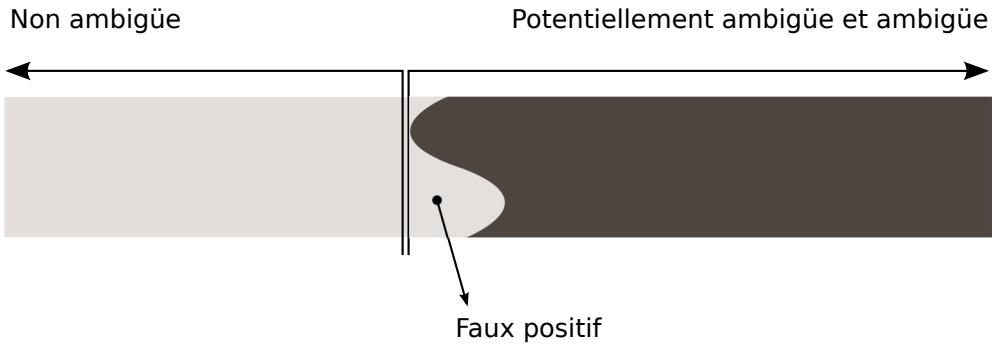


Figure 2.8 – Schéma de l'espace d'ambigüité

conduit à une contradiction. On en conclut que le problème de l'ambigüité des grammaires hors contexte est aussi non décidable.

D'autre part, pour statuer sur l'ambigüité du résultat, il doit être possible de vérifier, pour deux grammaires quelconques G_1 et G_2 si $G_1 \cap G_2 = \emptyset$. Il a été démontré en 2.2.1 que l'ensemble des grammaires hors contexte n'est pas clos sur l'opération d'intersection.

La détection de l'ambigüité de certaines grammaires hors contexte peut cependant se faire avec des techniques d'approximation. Un tel algorithme peut retourner trois résultats : ambigüe, non ambigüe ou impossible à déterminer. On s'intéresse à des approximations sures, ce qui veut dire que si l'algorithme détermine que la grammaire est ambigüe ou non ambigüe, alors il est certain que c'est véritablement le cas. On apprécie la précision de l'approximation par la taille relative de l'ensemble des grammaires dont l'ambigüité est impossible à déterminer. La figure 2.8 montre l'espace d'ambigüité pour l'ensemble des grammaires hors contexte et les frontières pour d'une approximation sure. Dans cet exemple, s'il est déterminé qu'une grammaire est dans l'ensemble de gauche, il est certain qu'elle est non ambigüe, tandis que si elle se retrouve dans l'ensemble de droite, on ne peut statuer sur son ambigüité.

2.4.2 Condition LR(k)

Une méthode de détection d'ambigüité consiste à réaliser un analyseur $LR(k)$ pour la grammaire. L'absence de conflits dans la table de l'analyseur indique que la

grammaire est non ambigüe [8]. Il s'agit de la condition $LR(k)$. Si la grammaire n'est pas $LR(k)$, alors on ne peut statuer sur son ambigüité.

Exemple 2.4.1. Soit la grammaire 2.1. Cette grammaire n'est pas $LR(k)$ peu importe k. La différence entre les règles (1) et (2) réside dans le dernier caractère. Il peut y avoir un préfixe arbitraire de c avant la réduction de A ou B, nécessitant un nombre de symboles d'anticipation non borné. La grammaire est non ambigüe, mais ne répond pas au critère $LR(k)$. Un algorithme non déterministe est donc nécessaire pour traiter directement cette grammaire.

Exemple 2.4.2. Soit la grammaire d'un palindrome $S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$. Elle ne répond pas à la condition $LR(k)$, malgré qu'elle soit non ambigüe et nécessite un analyseur non déterministe.

La condition $LR(k)$ est limitée à un ensemble de grammaires déterministes. Or, les exemples 2.4.1 et 2.4.2 montrent qu'il existe des grammaires non déterministes et non ambigües, pour lesquelles d'autres techniques sont nécessaires pour les identifier comme étant non ambigües.

2.4.3 Détection de l'ambigüité par technique d'approximation

La technique *Ambiguity Checking with Language Approximation* (ACLA) [2] consiste à détecter les ambigüités par des techniques d'approximation du langage généré par une grammaire hors contexte, tel que

$$\mathcal{L}_G(\alpha) \subseteq \mathcal{A}_G(\alpha) \ \forall \ \alpha \in \{V, \Sigma\}^*$$

L'approximation $\mathcal{A}_G$ est un surensemble du langage $\mathcal{L}_G$ pour toutes les protophrases α. En réalisant une approximation de la grammaire hors contexte dans le domaine des langages réguliers, le problème de l'ambigüité devient décidable.

La précision de la détection de l'ambigüité varie en fonction de la technique d'approximation utilisée. Une stratégie triviale consiste à approximer chaque variable par l'ensemble Σ^*. Cette stratégie n'est cependant pas précise et conduit à un très grand

nombre de faux positifs. À l'autre bout du spectre, on retrouve le langage hors contexte exact généré par la grammaire, pour lequel on obtient une précision parfaite, mais dont le problème n'est pas décidable. L'approximation régulière par réseau de transition récursif discuté à la section 2.3 est une approximation efficace pour détecter l'ambigüité.

Pour la suite de l'analyse, on décompose l'ambigüité d'une grammaire hors contexte en ambigüités verticale et horizontale.

Définition 2.4.2 (Ambigüité verticale)**.** Une grammaire G est verticalement ambigüe $\|\!\vdash_G$ ssi elle possède deux règles de production différentes $A \to \alpha$ et $A \to \beta$ qui peuvent dériver la même chaine, représenté à la figure 2.9.

$$\exists A \in V, (A \to \alpha), (A \to \beta) \in P, \alpha \neq \beta : \mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\beta) \neq \emptyset \tag{2.2}$$

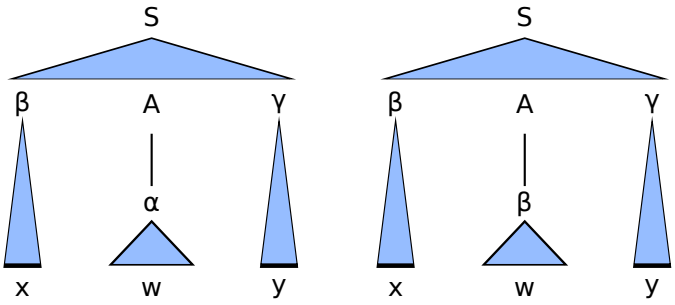


Figure 2.9 – Arbres syntaxiques résultant d'une ambigüité verticale

Définition 2.4.3 (Ambigüité horizontale)**.** Une grammaire G est horizontalement ambigüe $\models_G$ ssi elle possède une règle de production $A \to \gamma$ comportant une décomposition $\gamma = \alpha\beta$ telle que les langages $\mathcal{L}(\alpha)$ et $\mathcal{L}(\beta)$ se chevauchent, c'est-à-dire qu'il existe une séquence de terminaux pouvant finir $\mathcal{L}(\alpha)$ ou commencer $\mathcal{L}(\beta)$, représenté à la figure 2.10.

$$\exists A \in V, (A \to \alpha\beta) \in P : \mathcal{L}_G(\alpha) \bowtie \mathcal{L}_G(\beta) \neq \emptyset \tag{2.3}$$

où l'opérateur de chevauchement $\bowtie$ de deux langages X et Y est défini par

$$X \bowtie Y = xvy | x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X \wedge vy, y \in Y \tag{2.4}$$

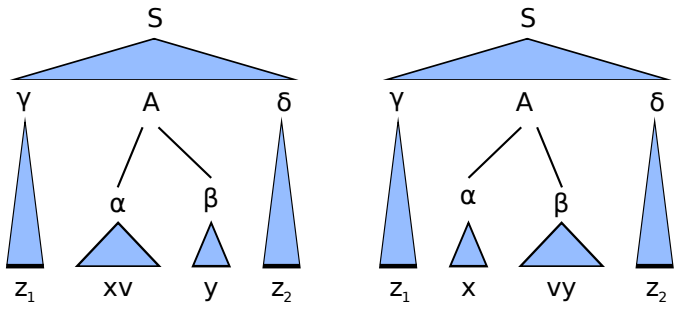


Figure 2.10 – Arbres syntaxiques résultant d'une ambigüité horizontale

Théorème 2.4.2 (Équivalence des définitions d'ambigüité)**.** Les deux définitions suivantes sont équivalentes :

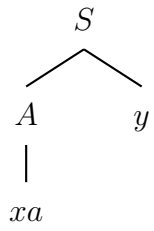
1. Une grammaire G est non ambigüe selon la définition 2.4.1.
2. Une grammaire G est verticalement et horizontalement non ambigüe.

Le théorème 2.4.2 signifie que si une grammaire est verticalement et horizontalement non ambigüe, alors la grammaire G est non ambigüe. Si l'une ou l'autre des dimensions présente une ambigüité, alors la grammaire est ambigüe. La preuve du théorème est disponible dans la présentation de ACLA [2].

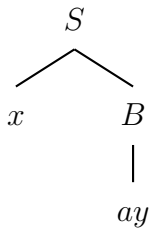
Exemple 2.4.3 (Ambigüité verticale)**.** Soit la grammaire 2.5, produisant le langage $\mathcal{L} = xay$. Cette grammaire comporte une ambigüité verticale, ce qui est démontré par les arbres syntaxiques obtenus en a) et b), puisque les règles de production S peuvent dériver toutes deux la chaine xay.

$$
\begin{array}{ll}
(1) & S \rightarrow Ay \\
(2) & S \rightarrow xB \\
(3) & A \rightarrow xa \\
(4) & B \rightarrow ay
\end{array}
\tag{2.5}
$$

a)

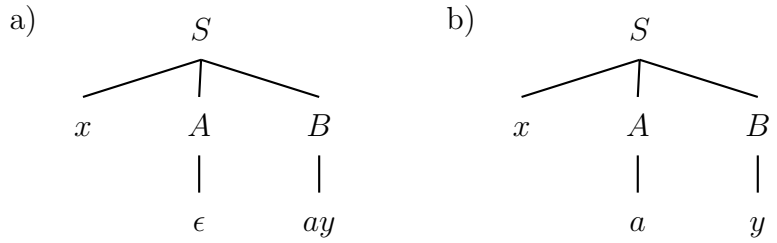


b)



Exemple 2.4.4 (Ambigüité horizontale)**.** Soit la grammaire 2.6, produisant le langage $\mathcal{L} = xay$. Cette grammaire comporte une ambigüité horizontale, démontrée par les arbres syntaxiques obtenus en a) et b), puisque le chevauchement des termes xA et B de la règle $S \rightarrow xAB$ contient le caractère a, ce qui est mis en évidence par l'expression $x[a]? \bowtie [a]?y$, dans laquelle le caractère a peut se retrouver à gauche ou à droite de l'opérateur.

$$
\begin{array}{ll}
(1) & S \rightarrow xAB \\
(2) & A \rightarrow a \\
(2) & A \rightarrow \epsilon \\
(3) & B \rightarrow ay \\
(4) & B \rightarrow y
\end{array}
\tag{2.6}
$$



L'approximation d'un langage doit être un surensemble du langage de départ, de manière à préserver les ambigüités.

Théorème 2.4.3 (Surensemble approximatif)**.** Soit X et Y, deux ensembles de chaines de symboles terminaux, et les surensembles $X' \supseteq X$ et $Y' \supseteq Y$. Alors les relations suivantes sont valides.

$$X' \cap Y' \supseteq X \cap Y \tag{2.7}$$

$$X' \between Y' \supseteq X \between Y \tag{2.8}$$

Preuve. L'équation 2.7 est une équation de base de la théorie des ensembles. Pour l'équation 2.8, $X \between Y$ vaut par définition

$$\{xvy | x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X \wedge vy, y \in Y\}$$

Soit xvy un élément de $X \between Y$, alors, $x, xv \in X \subseteq X'$ et $y, vy \in Y \subseteq Y'$, ce qui valide la condition de l'opérateur de chevauchement, donc xvy est un élément de $X' \between Y'$.

Exemple 2.4.5. Soit la grammaire de l'exemple 2.3.1 pour laquelle on désire prouver qu'elle est non ambigüe par la méthode ACLA. Les approximations régulières pour les règles de production de la grammaire sont détaillées en 2.9.

Règle	Approximation
(1) $S \to aSa$	$a(a\|b)^*a$
(2) $S \to bSb$	$b(a\|b)^*b$
(3) $S \to a$	a
(4) $S \to b$	b
(5) $S \to \epsilon$	ϵ

(2.9)

Le calcul de l'ambigüité horizontale est effectué en 2.10.

Production	Préfixe	Suffixe	Chevauchement
(1)	a	Sa	$a \bowtie (a\|b)^*a = \emptyset$
(1)	aS	a	$a(a\|b)^* \bowtie a = \emptyset$
(2)	b	Sb	$b \bowtie (a\|b)^*b = \emptyset$
(2)	bS	b	$b(a\|b)^* \bowtie b = \emptyset$

(2.10)

Le calcul de l'ambiguité verticale est effectué en 2.11.

Productions	Intersection
(1,2)	$a(a\|b)^*a \cap b(a\|b)^*b = \emptyset$
(1,3)	$a(a\|b)^*a \cap a = \emptyset$
(1,4)	$a(a\|b)^*a \cap b = \emptyset$
(1,5)	$a(a\|b)^*a \cap \epsilon = \emptyset$
(2,3)	$b(a\|b)^*b \cap a = \emptyset$
(2,4)	$b(a\|b)^*b \cap b = \emptyset$
(2,5)	$b(a\|b)^*b \cap \epsilon = \emptyset$
(3,4)	$a \cap b = \emptyset$
(3,5)	$a \cap \epsilon = \emptyset$
(4,5)	$b \cap \epsilon = \emptyset$

(2.11)

Étant donné qu'aucune ambigüité horizontale ni verticale n'a été trouvée, la grammaire est non ambigüe.

2.4.4 Ambiguïté de groupement d'une expression régulière

Les langages réguliers sont ceux reconnus par une machine à états finie. Comme toute expression régulière peut être réduite à une machine à états finie équivalente déterministe, il ne peut y avoir d'ambigüité dans une expression régulière.

Le traitement considéré ici ne consiste pas uniquement dans le test d'appartenance d'une chaine au langage, mais dans la segmentation de la chaine pour en capturer des parties destinées à un traitement ultérieur. Une expression régulière constituée de groupements peut conduire à plusieurs segmentations différentes de la

chaine d'entrée [4]. Par exemple, l'expression régulière `(a)*(a)*` est formée de deux groupes. Les segmentations possibles de la chaine *aaa* sont :

`(a)*...`	`...(a)*`
ϵ	`aaa`
`a`	`aa`
`aa`	`a`
`aaa`	ϵ

On constate que, même si l'expression régulière `(a)*(a)*` n'est pas intrinsèquement ambigüe, le groupement résultant est ambigu.

Dans le cas de transformations bidirectionnelles, la bijectivité des relations est assurée uniquement si la segmentation de la chaine d'entrée ne peut se faire que d'une manière. Contrairement à l'ambigüité des grammaires hors contexte, la détection de l'ambigüité des groupements d'une expression régulière est un problème décidable. L'ambigüité des groupes d'une expression régulière se détecte par l'opérateur d'intersection dans le cas de l'union de deux groupes et l'opérateur de chevauchement dans le cas de la concaténation de deux groupes. Aux fins de cette analyse, la répétition Kleene d'un groupe est un cas particulier de la concaténation de ce groupe à lui-même.

Chapitre 3

XSugar

XSugar [1] est un projet issu d'une recherche menée dans le laboratoire *Basic Research In Computer Science* (BRICS) de l'Université d'Aarhus au Danemark à propos de l'étude des transformations bidirectionnelles entre formats non XML et XML [3].

Les transformations sont écrites dans une feuille de style, qui contient la grammaire du fichier et sa correspondance XML. La forme abstraite correspond au format XML, qui est modifiable sous forme de *Document Object Model* (DOM).

En premier lieu, le principe de fonctionnement de XSugar est expliqué, les limites identifiées sont exposées et finalement les travaux réalisés sont détaillés.

3.1 Principe d'opération

XSugar fait la correspondance entre les formats par l'entremise de l'arbre syntaxique obtenu à partir de l'analyse du texte de l'entrée. Qu'ils soient obtenus à partir du texte d'origine ou du document XML, les arbres syntaxiques sont isomorphes. La feuille de style décrit la correspondance entre les grammaires des deux formats :

$$\mathcal{N} \rightarrow \alpha = \beta \tag{3.1}$$

La portion α rerésente la grammaire du format non XML, aussi appelée la grammaire de gauche, tandis que la portion β est l'équivalent XML, soit la grammaire de droite.

1. http ://www.brics.dk/xsugar/

Voici comment se produit une transformation aller-retour *non* $XML \rightarrow XML \rightarrow$ *non* XML. La transformation *non* $XML \rightarrow XML$ consiste à construire l'arbre syntaxique à l'aide de la grammaire de gauche. La chaine correspondant au XML est reconstruite à l'aide de la grammaire de droite. La transformation inverse $XML \rightarrow$ *non* XML se produit de la même manière, en inversant les grammaires.

Puisque le principe de fonctionnement repose sur des arbres syntaxiques isomorphes, la bijectivité des transformations est garantie si les grammaires sont non ambigües. XSugar analyse l'ambigüité horizontale et verticale des grammaires afin de vérifier statiquement la bidirectionnalité de la feuille de style, comme présenté à la section 2.4.3.

Exemple 3.1.1. Soit la feuille de style `n.xsg` de la figure 3.1, acceptant le langage

$$x^m y^n \text{ avec } m > 0 \text{ et } n > 0$$

Cette feuille de style montre l'énumération d'élément `<X>` et l'imbrication d'éléments `<Y>`. La variable `n` est le symbole de départ. La grammaire non XML est composée de la variable `xs` suivi de la variable `ys`. Du côté XML, l'élément racine `<A>` est défini et contient les deux variables dans l'ordre inverse. Cette règle montre que les variables peuvent apparaitre dans un ordre quelconque. La première règle `xs` comprend le terminal `x` suivi de la variable `xs`, ce qui correspond à une suite. La seconde règle, composée uniquement d'un terminal, permet de mettre fin à la suite. Les règles `ys` sont semblables, à l'exception du symbole non terminal `ys`, qui se retrouve à l'intérieur de l'élément `<Y>`, ce qui a pour effet de les imbriquer. L'entrée `xxyyy` produit le document XML de la figure 3.2.

3.1.1 Format d'une feuille de style

L'explication détaillée du fonctionnement de XSugar est réalisée à l'aide de la feuille de style `students.xsg` de la figure 3.3. Cette feuille de style prend une liste de nom d'étudiants, avec leur courriel et un numéro de matricule, et la convertie en XML.

Le format de la feuille de style comprend la déclaration de l'espace de nom à la ligne 1. Cet espace de nom est ajouté au résultat de la transformation XML et peut être utilisé pour la validation d'un document lorsque la DTD est disponible.

```
n : [xs x_s] [ys y_s] = <A> [ys y_s] [xs x_s]</>

xs : "x" [xs x_s] = <X></> [xs x_s]
   : "x"          = <X></>

ys : "y" [ys y_s] = <Y> [ys y_s] </>
   : "y"          = <Y></>
```

Figure 3.1 – Feuille de style `n.xsg`

```
<?xml version="1.0" encoding="UTF-8"?>
<A>
  <Y>
    <Y>
      <Y/>
    </Y>
  </Y>
  <X/>
  <X/>
</A>
```

Figure 3.2 – Résultat de la transformation de `xxyyy`

```
1  xmlns = "http://studentsRus.org/"
2
3  Name   = [a-zA-Z]+(\ [a-zA-Z]+)*
4  Email  = [a-zA-Z._]+\@[a-zA-Z._]+
5  Id     = [0-9]{8}
6  NL     = \r\n|\r|\n
7
8  file : [persons p] = <students> [persons p] </>
9
10 persons : [person p] [NL] [persons more] =
11                [person p] [persons more]
12         : =
13
14 person : [Name name] __ "(" [Email email] ")" _ [Id id] =
15          <student sid=[Id id]>
16              <name> [Name name] </>
17              <email> [Email email] </>
18          </>
```

Figure 3.3 – Feuille de style `students.xsg`

Entre les lignes 3 et 6, les expressions régulières utilisées sont déclarées. Le format est `ident = regexp (MAX)`, soit un identifiant, l'expression régulière et l'attribut optionnel `(MAX)`, indiquant que l'expression régulière doit capturer seulement la plus grande chaine possible, ce qui la rend avide (*greedy*).

Le reste du fichier contient les règles de production des grammaires de gauche et de droite. La règle à la ligne 8 déclare la règle `file`, incluant une variable `[persons p]` et son équivalent XML, un élément `<students>` contenant aussi la variable `[persons p]`. La balise fermante est normalisée sous la forme `</>`. Les lignes 10 à 12 déclarent la variable `persons` sous la forme d'une énumération $P \rightarrow S\ P \mid \epsilon$. La variable `persons` est identifiée par l'étiquette `p` dans l'arbre produit par l'analyse syntaxique. Chaque enregistrement est séparé par un terminal consistant en un retour de ligne `[NL]`. Fait à noter, ce dernier n'apparait pas du côté XML. Ceci signifie que la valeur exacte capturée lors de l'analyse du fichier non XML ne sera pas présente dans le document XML résultant de la transformation. La règle vide de la ligne 12 agit au même titre que ϵ pour terminer la récursion. Chaque ligne supplémentaire débutant par un deux points (`:`) est une règle alternative pour la même variable. Les lignes 14 à 18 déclarent

Table 3.1 – Résumé des items d'une grammaire XSugar

Type	Format	Description
Règle	`N : l = r`	`N` est un identifiant d'une variable, `l` est une suite de terminaux et de variables définissant la règle de la grammaire non XML et `r` est une suite de terminaux, de variables et d'éléments définissant la règle de la grammaire XML.
Variable	[Ident label]	Identifiant et étiquette entre crochet. Chaque variable présente dans la grammaire non XML doit apparaître dans la grammaire XML.
Terminal regexp	[Ident label]	Identifiant et étiquette facultative entre crochet. Le terminal peut apparaitre uniquement dans la grammaire non XML si l'étiquette n'est pas définie. Les symboles __ et _ sont respectivement des espaces obligatoires et facultatifs et n'ont pas d'étiquette définie.
Terminal fixe	`"string"`	Chaine de caractères fixe entre guillemets devant être capturée.
Élément	`<QName></>`	Déclaration des éléments XML, incluant la possibilité de définir des attributs. La balise de fermeture est normalisée.

la variable `person`. La partie de gauche comprend le terminal `[Name name]`. Le double souligné (__) est un symbole de convention indiquant un caractère blanc obligatoire, correspondant à l'expression régulière `/[ \r\n\t]+/`, tandis que le souligné simple (_) est un espace facultatif décrit par `/[ \r\n\t]*/`. La parenthèse entre guillemets correspond à un terminal. En résumé, l'expression régulière d'un enregistrement est la concaténation de `Name __ (Email) _ Id`. Dans cette règle, les terminaux `Name`, `Email` et `Id` ont une étiquette associée et sont reportés dans la grammaire XML. Lors de la transformation, les valeurs capturées apparaitront dans le document XML. Les items définissant la grammaire sont résumés à la table 3.1

```
file    : persons[p]
persons : person[p] <NL> persons[more] | ""
person  : <Name>[name] <__>[" "]
          "(" <Email>[email] ")"
          <_>[""] <Id>[id]
```

Figure 3.4 – Grammaire non XML de la feuille de style `students.xsg`

```
file    : "<" "students" ">" persons[p] "</>"
persons : person[p] persons[more] | ""
person  : "<" "student" " " "sid" "=" "\"" <Id>[id] "\"" ">"
              "<" "name" ">" <Name>[name] "</>"
              "<" "email" ">" <Email>[email] "</>"
          "</>"
```

Figure 3.5 – Grammaire XML de la feuille de style `students.xsg`

3.1.2 Compilation des grammaires

La feuille de style définit deux grammaires simultanément, soit celle non XML et sa contrepartie XML. La grammaire non XML est utilisée directement par l'analyseur syntaxique, tandis que la grammaire XML nécessite une étape de compilation.

Les éléments et attributs XML déclarés dans la feuille de style le sont avec un haut niveau d'abstraction. Pour analyser un document, ils sont compilés sous forme d'items de grammaire de bas niveau. Les éléments sont directement convertis sous forme de terminal fixe. Des règles prédéfinies sont générées pour les attributs, et ces règles ont la propriété de ne pas être ordonnées pour correspondre au standard XML 1.1, qui stipule que les attributs peuvent apparaitre dans un ordre quelconque. Les règles des grammaires non XML et XML résultantes sont montrées respectivement aux figures 3.4 et 3.5

3.1.3 Analyseur syntaxique

L'analyseur syntaxique de XSugar est un analyseur non déterministe basé sur l'algorithme Earley. Sa particularité est d'analyser l'entrée sans étape d'analyse lexicale.

L'arbre syntaxique produit par l'analyseur est formé de noeuds étiquetés par les variables et les terminaux de la grammaire, sauf pour les terminaux de la grammaire qui n'ont pas d'étiquettes, qui ne sont pas ajoutés à l'arbre.

L'option `(MAX)` pour les expressions régulières est généralement le comportement désiré. Si cette option n'est pas spécifiée et que l'entrée contient une erreur syntaxique, alors toutes les possibilités constituées des sous-chaines capturées par l'expression régulière seront testées, ce qui augmente drastiquement l'espace de recherche de l'analyse syntaxique. Ainsi, l'option `(MAX)` interrompt rapidement l'analyse dans le cas d'une erreur syntaxique.

3.1.4 Transformations

La première étape de la transformation *non* $XML \rightarrow XML$ consiste à obtenir l'arbre syntaxique correspondant à l'entrée selon la grammaire non XML. Ensuite, il s'agit d'effectuer l'opération d'*unparsing*, s'effectuant en parcourant l'arbre syntaxique en postordre, et en associant chaque noeud à sa règle de production. Les terminaux qui n'existent pas dans l'arbre syntaxique, mais qui sont présents dans la grammaire utilisée pour effectuer l'*unparsing* sont générés. Les terminaux fixes sont copiés. Pour les terminaux décrits par une expression régulière, l'automate représentant l'expression régulière est exécuté pour produire une chaine du langage.

Exemple 3.1.2. Soit le fichier d'entrée de la figure 3.6 comprenant deux enregistrements, et la feuille de style de la figure 3.3. L'arbre syntaxique obtenu est présenté à la figure 3.7. Les variables sont numérotées selon leur règle de production correspondante, numéro qui est affiché entre crochet après le nom de la variable dans l'arbre. Par exemple, l'étiquette du noeud `persons[#2]` correspond à la deuxième règle de `persons`, soit la règle d'effacement. L'arbre syntaxique ne contient pas les terminaux sans étiquette. Lors de l'étape du *unparsing*, les noeuds de l'arbre sont associés aux règles de production de la grammaire complémentaire pour générer la sortie, tel que montré à la figure 3.8.

```
John Doe (john_doe@notmail.org) 19701234
Jane Dow (dow@bmail.org) 19785678
```

Figure 3.6 – Fichier d'entrée pour la feuille de style `students`

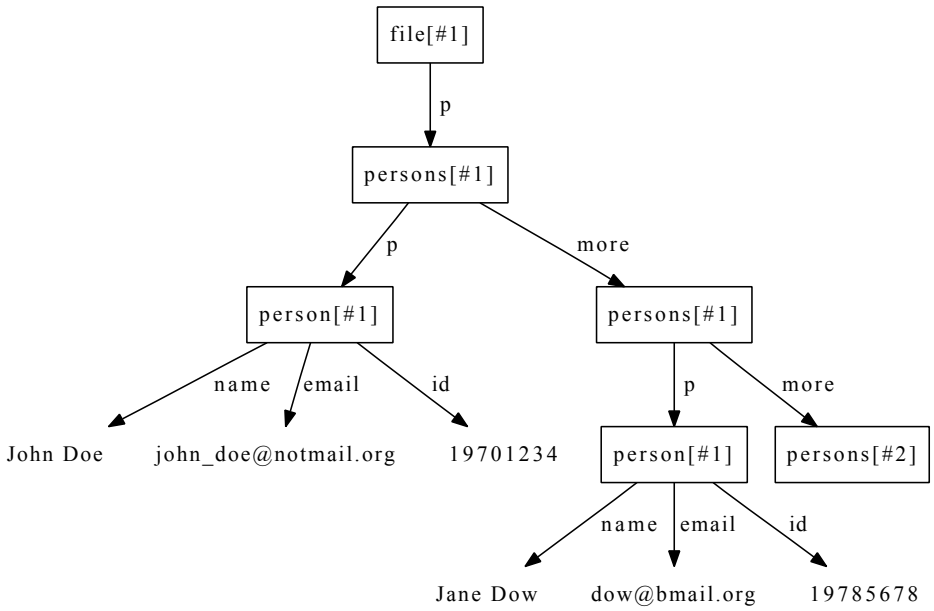


Figure 3.7 – Arbre syntaxique du fichier d'entrée

3.1.5 Analyse statique

L'analyse statique de la feuille de style, pour valider la propriété bidirectionnelle, s'effectue en trois étapes. La première consiste à vérifier que toutes les variables utilisées dans les règles de production de la grammaire non XML ont une correspondance dans la grammaire XML. On s'assure de cette manière que les arbres syntaxiques soient isomorphes. Ensuite, l'ambigüité horizontale et verticale des deux grammaires sont analysées indépendamment. Si aucune ambigüité n'est détectée, alors il est garanti que la relation est bijective et donc bidirectionnelle.

XSugar permet également de valider une feuille de style par rapport à un schéma XML de type DTD, RelaxNG ou XML Schema. Ceci permet de valider statiquement que tout document XML produit par la transformation est conforme au schéma.

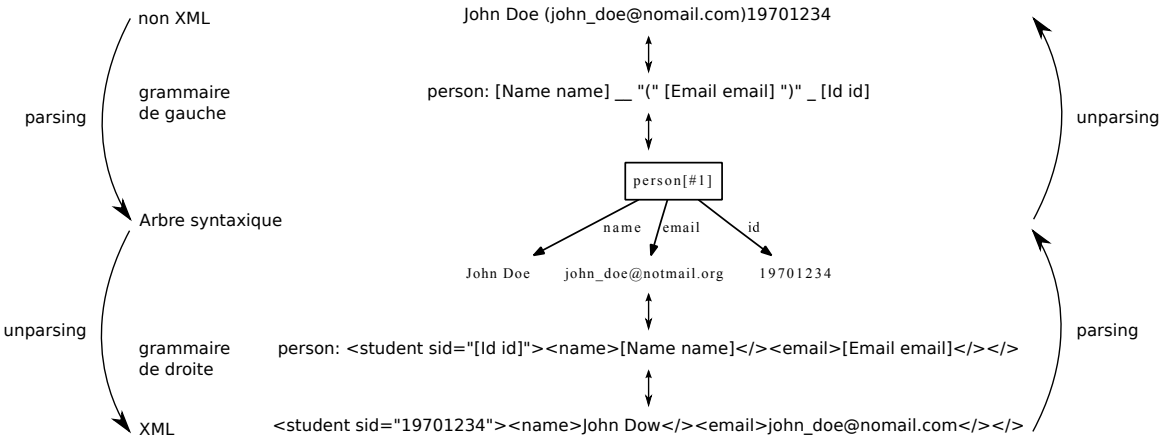


Figure 3.8 – Exemple de correspondance par l'arbre syntaxique

3.2 Limites

Dans certaines situations, une transformation aller-retour d'un document avec XSugar produit la fonction identité *modulo* la perte de certains caractères. Pour la gestion des fichiers de configuration, il est nécessaire que les caractères exclus de la représentation abstraite soient restitués lors de la transformation inverse. Dans cette section, les conditions conduisant à la perte d'information pendant un aller-retour sont présentées.

La première source de perte d'information se produit sur certains éléments de texte. Les figures 3.9 et 3.10 montrent la représentation brute et indentée respectivement du même document XML. L'élément `<root>` contient les deux éléments `<a>` et `<b>`, l'élément `<a>` contient seulement un noeud texte et l'élément `<b>` contient un noeud texte et l'élément `<c>`. Le contenu de ce dernier est mixte, contenant à la fois un noeud texte et un élément. Le noeud texte de l'élément `<b>` est modifié par l'indentation du fichier XML, procédure qui ajoute un retour de chariot et des espaces avant et après le texte d'origine, ce qui constitue une première distorsion. Cependant, cette modification est annulée par l'étape de normalisation du document XML, qui a lieu lors de la préparation du document pour la transformation inverse. La normalisation fait appel à la fonction `getTextTrim()` sur les noeuds texte lorsque l'élément contient d'autres éléments et des noeuds textes, de manière à retrouver la représentation XML brute. Ainsi, le texte de la balise `<a>` est préservé, mais celui de

```
<root><a> x y z </a><b> p q r <c></c></b></root>
```

Figure 3.9 – Document XML non indenté

```
<root>
  <a> x y z </a>
  <b>
    p q r
    <c></c>
  </b>
</root>
```

Figure 3.10 – Document XML indenté

la balise `<b>` ne l'est pas, puisque les espaces au début et à la fin sont enlevés par la fonction `getTextTrim()`, parce que son contenu est mixte. En résumé, cette perte d'information se produit pour les caractères blancs au début et à la fin d'un noeud texte lorsque celui-ci possède au moins un noeud élément frère.

L'autre perte d'information est liée aux terminaux d'expressions régulières présents dans une grammaire pour lesquels il n'existe pas de correspondance dans l'autre grammaire. Ces terminaux sont perdus dans une direction et une valeur par défaut est calculée dans l'autre. Cette situation est illustrée à l'exemple 3.2.1

Exemple 3.2.1. Soit la règle de production `P: [B] = <x></>` d'une feuille de style quelconque et l'expression régulière `B = [b]+ (MAX)`. Le terminal `[B]` de la grammaire de gauche n'est pas étiqueté et ne se retrouve pas dans la grammaire de droite. Puisque l'expression régulière `B` est avide (*greedy*), la transformation du document `bbb` en XML produit un seul élément `<x></>`. Lors de la transformation inverse, la première chaine appartenant au langage de `[b]+` est générée, soit `b`. En conséquence, le nombre exact de `b` est perdu lors d'un aller-retour.

Ces deux conditions provoquent la perte de caractères lors d'une transformation aller-retour. Les vérifications effectuées sur la feuille de style ne détectent pas ces situations. Dans la section 3.3, une technique permettant de détecter les situations pouvant conduire à une perte d'information dans le cas de contenu XML mixte est

présentée. Dans la section 3.4, deux stratégies préservant les chaines de caractères des terminaux sans étiquette sont exposées, dans le but d'atteindre une relation bidirectionnelle stricte et éviter toute perte d'information pendant un aller-retour.

3.3 Détection statique de contenu mixte

Dans cette section, on s'intéresse à des méthodes d'analyse de la feuille de style détectant les situations pouvant mener à la perte d'information à cause des propriétés du XML. Deux conditions sont nécessaires, soit la présence de contenu mixte d'éléments et de texte, et le texte doit pouvoir débuter ou terminer par des caractères blancs `/[ \r\n\t]*/`. Si tel est le cas, ces caractères peuvent être perdus par la fonction `getTextTrim()`, et la feuille de style doit être modifiée manuellement pour éviter cette situation.

Le problème central de l'analyse est de déterminer quel type, entre élément ou texte ou les deux, peut générer une variable. Une technique basée sur une approximation régulière de la feuille de style est présentée. Ensuite, la détection du chevauchement avec les caractères blancs est exposée. Toutes ces analyses portent sur la grammaire XML. Chaque section couvre les hypothèses de départ, les travaux réalisés, les résultats obtenus et une conclusion.

3.3.1 Approximation régulière du type de contenu

Le calcul du type est facilement effectué pour les terminaux de type expression régulière, chaine fixe et élément. Les deux premiers génèrent un noeud de type texte, tandis que le second, trivialement, génère un noeud de type élément. Pour les variables, il existe une correspondance directe pour des règles non récursives, telle que présentée dans la table 3.2.

En calculant de la sorte le type des variables, alors on peut propager leur type dans la grammaire. Cependant, cette technique ne fonctionne pas avec des variables mutuellement récursives

$$\texttt{S}: \alpha \texttt{P} \beta$$
$$\texttt{P}: \sigma \texttt{S} \omega$$

Table 3.2 – Types possibles d'une variable

Règle	Type
`S: <a>...</>`	élément
`S: [Email email]`	texte
`S: <a>...</> [Email email]`	mixte (élément et texte)

puisque pour déterminer le type de `S`, le type de `P` doit être déterminé, ce qui nécessite lui-même de connaitre le type inconnu de `S`. Pour y arriver, il faut briser le cycle récursif.

Hypothèse

Il doit être possible de déterminer le type de n'importe quelle variable par une technique de réseau de transition récursif introduite à la section 2.3.

Travaux

Un algorithme compilant un automate à partir de la grammaire XML a été réalisé. La compilation de l'automate pour une variable se fait en concaténant les états $q_1, ...q_n$ avec des transitions (q_{x-1}, X_i, q_x) pour chaque item X_i (terminal, variable ou élément) de la règle de production. L'union de chaque alternative pour une même règle est réalisée avec des règles ϵ. La substitution des transitions correspondant aux variables s'effectue de la même manière que la technique RTN d'origine. L'automate résultant est déterminisé et minimisé. Le type des items de l'alphabet de l'automate résultant représente une approximation supérieure des types que peut générer cette variable.

Exemple 3.3.1. Soit la feuille de style `root.xsg` de la figure 3.11, pour lequel on tente de déterminer le type de la variable `s`. Les automates résultants sont présentés à la figure 3.12. L'alphabet de l'automate est {`<a>`,`[B b]`,`[C c]`}, dont les types sont {`élément`,`texte`,`texte`}. Selon cette approximation, le contenu de `<root>` peut être mixte, conduisant potentiellement à une perte d'information.

```
xmlns = "http://example.com/"

B = [b]+
C = [c]+

file : [s ss] = <root> [s ss] </>

s : [p pp] = <a> [p pp] </>
  : [B b] [p pp] = [B b] [p pp]
  : =

p : [C c] [s ss] = [C c] [s ss]
```

Figure 3.11 – Feuille de style `root.xsg`

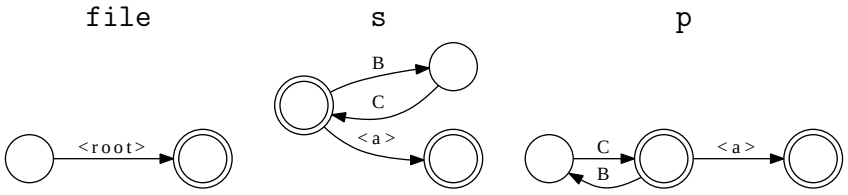


Figure 3.12 – Automates approximatifs des variables de `root.xsg`

3.3.2 Calcul de chevauchement

S'il est déterminé que le contenu d'un élément est mixte, cela ne conduit pas nécessairement à une perte d'information. Pour que ce comportement survienne, il faut que le langage accepté par le contenu puisse débuter ou terminer par des caractères blancs.

La technique utilisée consiste à obtenir l'approximation régulière du contenu de l'élément, obtenue de la même manière que lors de l'analyse de l'ambigüité. Les transitions depuis l'état initial et vers les états finaux sont analysées pour déterminer si ces transitions acceptent un caractère blanc. Si tel est le cas, alors la feuille de style peut conduire à une perte d'information.

Table 3.3 – Résultats de la détection du contenu mixte

Feuille de style	Items mixtes	Perte d'information aller-retour
`bcard.xsg`	Non	Non
`bibxml.xsg`	Non	Non
`empty.xsg`	Non	Non
`json.xsg`	Non	Non
`multy.xsg`	Non	Non
`nice.xsg`	Non	Non
`relax.xsg`	Non	Non
`students.xsg`	Non	Non
`wiki.xsg`	Oui	Oui
`xflat.xsg`	Non	Non

Résultats

L'algorithme a été utilisé pour analyser les 11 feuilles de style d'exemples de XSugar. Les résultats sont comparés à ceux obtenus expérimentalement par la transformation bidirectionnelle de documents représentatifs pour chaque feuille de style. Les résultats sont présentés dans la table 3.3.

On observe que la feuille de style `wiki.xsg` cause une dégradation du document lors d'un aller-retour pour le document d'exemple. L'algorithme détecte adéquatement que cette feuille de style permet une perte d'information à cause de contenu mixte.

Conclusion

L'algorithme proposé d'approximation régulière du type de contenu pouvant être généré par une variable de la grammaire fonctionne adéquatement. Cette technique a permis de détecter toutes les pertes d'information relevées de manière expérimentale.

3.4 Bidirectionalité stricte

Une feuille de style contenant des terminaux non étiquetés peut provoquer une perte d'information lors d'une transformation aller-retour. Dans cette section, deux techniques permettant de préserver ces terminaux sont présentées.

3.4.1 Modification dynamique de la feuille de style

Le principe de fonctionnement de cette première technique consiste à inclure les éléments autrement perdus dans le document XML produit suite à une transformation. Le document XML produit est donc un surensemble strict du document d'origine.

Hypothèse

Il doit être possible de modifier une feuille de style quelconque pour éviter toute perte d'information en définissant une étiquette pour chaque terminal non étiqueté.

Travaux

Un algorithme analysant chaque production de la feuille de style a été réalisé. Il recherche dans les items de la grammaire non XML des terminaux d'expressions régulières sans étiquette. Si un tel terminal est trouvé, alors la règle est copiée et une priorité plus élevée lui est assignée. La grammaire de droite de cette nouvelle règle de production est modifiée. Un élément contenant le terminal est ajouté à l'intérieur d'un élément existant s'il en existe un, sinon l'élément est ajouté à la fin de la règle. Une étiquette unique générée est assignée à tous les terminaux d'expressions régulières de manière à capturer leur valeur dans l'arbre syntaxique. Un exemple du résultat de l'algorithme est montré pour la feuille de style `students.xsg` à la figure 3.13. Le symbole « `>:` » désigne une priorisation des règles selon leur ordre de déclaration.

Préserver la règle d'origine, plutôt que de la modifier sur place, permet de rendre facultatifs les éléments liés aux terminaux sans étiquette. S'ils ne sont pas fournis, alors la règle d'origine est sélectionnée et des valeurs par défaut sont générées. S'ils sont disponibles, alors ces terminaux sont utilisés. La priorité permet d'éviter une ambigüité verticale, puisque la règle d'origine et sa copie génèrent le même langage dans la grammaire non XML.

Résultats

Les données de la table 3.4 montrent que cet algorithme permet de modifier les 10 feuilles de style étudiées pour qu'elles répondent au critère de bidirectionnalité

```
xmlns = "http://studentsRus.org/"

Name   = [a-zA-Z]+(\ [a-zA-Z]+)*
Email  = [a-zA-Z._]+\@[a-zA-Z._]+
Id     = [0-9]{8}
NL     = \r\n|\r|\n
MWS    = [ \t]+
OWS    = [ \t]*

file : [persons p] = <students> [persons p] </>

persons >: [person p] [NL nl] [persons more] =
              [person p] [persons more] <blank1> [NL nl] </>
        >: [person p] [NL] [persons more] =
              [person p] [persons more]
        >: =

person >: [Name name] [MWS mws]
            "(" [Email email] ")" [OWS ows] [Id id] =
         <student sid=[Id id]>
             <name> [Name name] </>
             <email> [Email email] </>
         </>
         <blank2> [MWS mws] </>
         <blank3> [OWS ows] </>
      >: [Name name] __ "(" [Email email] ")" _ [Id id] =
         <student sid=[Id id]>
             <name> [Name name] </>
             <email> [Email email] </>
         </>
```

Figure 3.13 – Feuille de style `students.xsg` stricte

stricte. Seule la feuille de style `wiki.xsg` ne fonctionne pas à cause de la présence de contenu mixte. Cette feuille a été retirée de l'analyse. Cependant, la préservation de ces éléments a un cout, puisque le document XML à partir des feuilles de style modifiées est jusqu'à 496% plus volumineux que le document XML obtenu avec les feuilles de style d'origine.

Afin de vérifier en pratique la technique, un scénario de modification du fichier XML a été réalisé avec la feuille de style `students.xsg`. Chaque scénario modifie le fichier XML et on vérifie la représentation non XML obtenue. Les opérations ont

Table 3.4 – Résultats de la détection du contenu mixte

Feuille de style	Aller-retour Original	Aller-retour Modifiée	Ratio de taille XML augmenté
`bcard.xsg`	Oui	Oui	138%
`bibxml.xsg`	Non	Oui	496%
`empty.xsg`	Oui	Oui	97%
`json.xsg`	Non	Oui	441%
`multy.xsg`	Non	Oui	223%
`nice.xsg`	Oui	Oui	75%
`relax.xsg`	Non	Oui	277%
`students.xsg`	Non	Oui	109%
`xflat.xsg`	Oui	Oui	45%

été effectuées sans tenir compte des éléments contenant les terminaux sans étiquette, puisque ceux-ci ne font pas partie de l'abstraction. La table 3.5 contient un résumé des tests effectués.

Deux types d'erreurs sont observées, soit les erreurs syntaxiques et le problème d'alignement. Les erreurs syntaxiques sont causées par des modifications au fichier XML d'une manière telle qu'il ne correspond plus à la grammaire XML. Pour éviter les erreurs syntaxiques, l'utilisateur doit également tenir compte des éléments de terminaux sans étiquette, ce qui diminue le degré d'abstraction obtenu.

Quant au problème d'alignement, il provient de l'ordre dans lequel sont associés les éléments de terminaux sans étiquette avec les autres éléments auxquels ils s'appliquent. Lorsqu'un élément est ajouté ou déplacé, mais que les éléments de terminaux sans étiquette associés restent fixes, alors ces éléments seront associés à un autre élément. L'exemple de la table 3.6 comprend une liste de lettres et de chiffres. Soit une vue abstraite de cette liste, excluant le chiffre, sur laquelle les lettres sont classées par ordre alphabétique. Lors de la transformation inverse, l'ordre des noms est modifié, mais celui des chiffres reste fixe, provoquant la perte de l'association d'origine.

Ce problème d'alignement est présent dans Boomerang [1]. L'alignement par po-

Table 3.5 – Résultats obtenus avec la feuille de style `students.xsg` stricte

Scénario	Résultat
Modification d'une valeur `<name>`	La nouvelle valeur est reportée dans le document non XML.
Ajout d'un enregistrement `<student>`	Le nouvel enregistrement est reporté dans le document non XML, mais risque de causer une erreur syntaxique selon l'emplacement de l'ajout. Aussi, ceci peut causer un problème d'alignement des terminaux produisant une différence non minimale entre le document d'origine et celui modifié.
Déplacement d'un enregistrement `<student>`	L'ordre dans le document d'origine est modifié adéquatement, mais risque de causer une erreur de syntaxe selon l'emplacement de destination.
Suppression d'un enregistrement `<student>`	Cause une erreur de syntaxe si les éléments des terminaux sans étiquette ne sont pas supprimés aussi.

Table 3.6 – Exemple du problème d'alignement

Entrée	Vue abstraite	Vue abstraite modifiée	Sortie
C 1	C	A	A 1
B 2	B	B	B 2
A 3	A	C	C 3

sition résultant n'est pas satisfaisant, puisqu'il peut conduire à une corruption des données de départ.

Conclusion

La technique qui consiste à ajouter l'ensemble des terminaux sans étiquette dans le document XML permet de préserver lors d'un aller-retour tous les caractères du document original. Il est possible d'utiliser cette technique pour accéder au document en lecture seulement. Toute modification au document peut provoquer une erreur syntaxique ou un problème d'alignement entre la version abstraite et concrète du document, ce qui ne satisfait pas aux objectifs poursuivis.

3.4.2 Recouvrement par fusion des arbres syntaxiques abstraits

La seconde technique étudiée ne repose pas sur le principe d'ajouter les terminaux sans étiquette dans le document XML. Plutôt, au moment de convertir le document XML en un document non XML, les terminaux sans étiquette provenant de l'arbre syntaxique du document d'origine sont réintégrés à la place des valeurs par défaut.

Hypothèse

Il doit être possible de réinjecter les caractères capturés par les terminaux sans étiquette du document d'origine dans le nouvel arbre syntaxique, de manière à prévenir les problèmes d'erreurs syntaxiques lors de modifications du document et les problèmes d'alignement.

Travaux

Pour une transformation, deux versions de la même feuille de style sont utilisées. La première est la version originale et l'autre est sa version stricte obtenue par l'algorithme présenté dans la section 3.4.1, dans laquelle tous les terminaux d'expressions régulières sont étiquetés. Quatre grammaires sont définies, soit la grammaire de gauche et de droite originale G_O, D_O et leur contrepartie stricte G_S, D_S.

Les transformations produisant les arbres syntaxiques sur lesquels la fusion s'effectue sont les suivantes. Le document XML est obtenu à partir de la feuille de style d'origine et modifié par l'utilisateur produisant une nouvelle version XML' (3.2). L'arbre syntaxique représentant le document modifié est obtenu, mais dont les terminaux sans étiquette ont tous une valeur par défaut (3.3). Le document d'origine est également chargé avec la feuille de style stricte (3.4). À ce moment, l'arbre syntaxique du document modifié AST' et celui d'origine AST peuvent être fusionnés, puisqu'ils sont issus de la même grammaire.

$$non\ XML \overset{G_O}{\rightarrow} AST_O \overset{D_O}{\rightarrow} XML \overset{mod}{\rightarrow} XML' \tag{3.2}$$

$$XML' \overset{D_O}{\rightarrow} AST'_O \overset{G_O}{\rightarrow} non\ XML' \overset{G_S}{\rightarrow} AST'_S \tag{3.3}$$

$$non\ XML \overset{G_S}{\rightarrow} AST_S \tag{3.4}$$

Un algorithme parcourant l'arbre par niveau a été utilisé. Pour chaque noeud de l'arbre AST', on recherche dans l'arbre d'origine AST un noeud similaire. L'association entre deux noeuds se fait en comparant les noeuds enfants, excluant ceux qui représentent des terminaux sans étiquette dans la feuille de style d'origine. Si une association est trouvée, les chaines des terminaux sans étiquette de la feuille d'origine de ce noeud sont copiées et remplacent les valeurs par défaut dans AST'. Si aucun noeud ne correspond, alors les valeurs par défaut sont conservées. Pour les noeuds ne possédant pas de terminal avec étiquette dans la feuille de style d'origine, ils sont associés par ordre de parcours de l'arbre. Un noeud n'est associé qu'une seule fois. La figure 3.14 montre le résultat de l'association pour la feuille de style `students.xsg` simplifié de la figure 3.15 dans le cas de la suppression d'un enregistrement du scénario 5 à la table 3.7.

Résultats

Pour évaluer le comportement de cette technique, des scénarios d'utilisation ont été réalisés avec la feuille de style `ids.xsg`, de la figure 3.15. Il s'agit d'une version simplifiée de `students.xsg`. Les caractères blancs ont été remplacés par des caractères imprimables pour faciliter la lecture. Le séparateur `[SEP]` tient lieu des retours de ligne et `[SPC]` représente des espaces. Le document de départ est la chaine

```
1zz1aa22zzz22aaa333zzzz333aaaa
```

qui produit le fichier XML de la figure 3.16. L'entrée a été choisie pour mettre en évidence l'alignement. Le succès de la conversion et la qualité du résultat obtenu sont observés et sont présentés à la table 3.7.

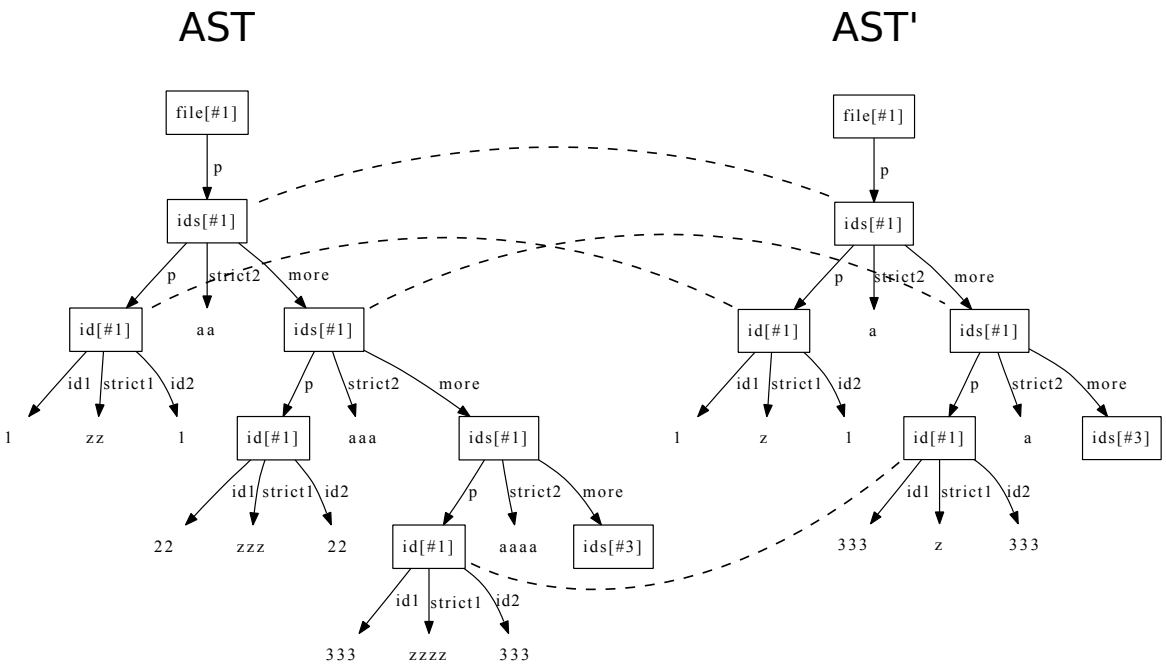


Figure 3.14 – Exemple de l'association entre arbres syntaxiques

```
xmlns = "http://udes.ca/"

Id      = [0-9]*
SEP      = [a]+
SPC      = [z]+

file : [ids p] = <ids> [ids p] </>

ids : [id p] [SEP] [ids more] = [id p] [ids more]
        : =

id : [Id id1] [SPC] [Id id2]  =
         <id>
             <id1> [Id id1] </>
             <id2> [Id id2] </>
         </>
```

Figure 3.15 – Feuille de style `students.xsg` simplifiée

```
<?xml version="1.0" encoding="UTF-8"?>
<ids xmlns="http://udes.ca/">
  <id>
    <id1>1</id1>
    <id2>1</id2>
  </id>
  <id>
    <id1>22</id1>
    <id2>22</id2>
  </id>
  <id>
    <id1>333</id1>
    <id2>333</id2>
  </id>
</ids>
```

Figure 3.16 – Document XML produit par `students.xsg` simplifié

Table 3.7: Tests de la fusion d'arbres syntaxiques

Scénario 1 :	Aller-retour sans modification
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie par défaut :	`1z1a22z22a333z333a`
Sortie fusionnée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Observations :	Le scénario 1 montre que tous les caractères ont été récupérés et replacés au bon endroit. En conséquence, le document de sortie est identique au document de départ.
Scénario 2	Modification de l'enregistrement `22` pour `55`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie par défaut :	`1z1a55z55a333z333a`
Sortie obtenue :	`1zz1aa55z55aaa333zzzz333aaaa`
Observations :	Le scénario 2 montre que l'élément modifié `22zzz22aaa` est modifié en `55z55aaa`, ce qui représente la valeur par défaut pour `[SPC]` et la valeur d'origine pour `[SEP]`. L'algorithme utilisé recherche des valeurs exactes, une modification d'un enregistrement est traitée comme un ajout.
Scénario 3	Déplacement de l'enregistrement `22` à la fin
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie par défaut :	`1z1a333z333a22z22a`
Sortie obtenue :	`1zz1aa333zzzz333aaa22zzz22aaaa`
Observations :	Le scénario 3 montre que les séparateurs `[SPC]` ont suivi correctement les enregistrements, tandis que la position des séparateurs `[SEP]` est demeurée fixe. Ce comportement est causé par la règle `ids` qui ne contient aucun terminal étiqueté. L'association des noeuds est faite alors séquentiellement.

Table 3.7: Tests de la fusion d'arbres syntaxiques (suite)

Scénario 4	Ajout de l'enregistrement `99` après l'enregistrement `1`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie par défaut :	`1z1a99z99a22z22a333z333a`
Sortie obtenue :	`1zz1aa99z99aaa22zzz22aaaa333zzzz333a`
Observations :	Le scénario 4 montre que le nouvel enregistrement `99` obtient un séparateur `[SPC]` par défaut. Les séparateurs `[SEP]` restent fixes, ce qui a pour conséquence de créer un décalage. Le dernier enregistrement `333` obtient la valeur par défaut pour `[SEP]`, toujours à cause de la règle `ids`.
Scénario 5	Suppression de l'enregistrement `22`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie par défaut :	`1z1a333z333a`
Sortie obtenue :	`1zz1aa333zzzz333aaa`
Observations :	Le scénario 5 montre que le séparateur `[SPC]` est préservé pour l'enregistrement `333`, mais que le séparateur `[SEP]` est celui de l'enregistrement supprimé.

Conclusion

L'algorithme suggéré pour la fusion des arbres syntaxiques offre plusieurs avantages par rapport à la technique d'inclusion des terminaux sans étiquette dans le document XML. En évitant à l'utilisateur de se préoccuper des terminaux sans étiquette, l'abstraction est maintenue et rend plus sures les modifications.

Cependant, le problème d'alignement demeure. L'algorithme utilisé actuellement est limité par son fonctionnement par niveaux.

Chapitre 4

Augeas

Augeas est un langage de programmation bidirectionnel dont le type de données traité est essentiellement des chaines de caractères. Il a été conçu pour l'application en gestion de fichiers de configuration. Les relations bidirectionnelles sont établies grâce à des lentilles, dont le principe de fonctionnement s'inspire des recherches sur Boomrang [1]. Les lentilles sont vérifiées statiquement pour s'assurer de leur comportement bidirectionnel. Grâce à une lentille décrivant le format d'un fichier de configuration, Augeas produit un arbre en mémoire pouvant être lu et modifié. L'arbre résultant peut ensuite être utilisé pour réécrire le fichier, tout en préservant le formatage du fichier et les commentaires du fichier d'origine. Plus de 50 lentilles permettent de traiter autant de syntaxes différentes.

La section débute par la présentation du fonctionnement interne d'Augeas. Ensuite, ses limites sont exposées. Les solutions proposées pour remédier aux limites sont présentées et évaluées. La section se termine par les travaux futurs.

4.1 Principe d'opération

Les programmes Augeas effectuent des transformations bidirectionnelles entre des chaines de caractères et une représentation abstraite de la chaine sous forme d'arbre. La transformation de la chaine vers l'arbre est la direction `get` et la transformation inverse est la direction `put`.

La programmation se fait par l'assemblage de lentilles primitives combinées avec

des opérateurs. Chaque lentille primitive est complètement définie dans les deux directions, `get` et `put`, et forment ensemble la fonction identité. Les opérateurs préservent le comportement bidirectionnel des lentilles primitives. L'arbre Augeas est formé de noeuds possédant une clé et une valeur optionnelle. Un noeud peut avoir un nombre indéfini d'enfants et de descendants.

Par exemple, la lentille `[ key "a" . store "b" ]`, dans la direction `get` accepte la chaine « ab » et produit le noeud { `"a" = "b"` }, tandis que la chaine « ab » est régénéré à partir de ce noeud par la fonction `put`. La section suivante présente en détail le langage de programmation d'Augeas.

4.1.1 Lentilles primitives

Une lentille primitive l définit quelle partie de la chaine doit être exclue de l'arbre ainsi que quelle partie de la chaine constitue les clés et les valeurs des noeuds de l'arbre. Chaque lentille comprend trois fonctions liant une représentation concrète de C à une représentation abstraite de A :

$$\begin{aligned} l.get &\in C \to A \\ l.put &\in A \times C \to C \\ l.create &\in A \to C \end{aligned}$$

Chaque fonction doit correspondre aux comportements aller-retour pour tout $c \in C$ et tout $a \in A$ tel que

$$\begin{aligned} l.put(l.get(c)) &= c \\ l.get(l.put(a, c)) &= a \\ l.get(l.create(a)) &= a \end{aligned}$$

La fonction `get` prend une chaine c en entrée et calcul sa version abstraite a sous forme d'arbre de clés et de valeurs. La fonction `put` prend une représentation abstraite a et une chaine concrète c et produit une nouvelle chaine concrète c. Finalement, la fonction `create` prend une représentation abstraite a pour laquelle il n'existe pas de chaine d'entrée c correspondante et produit une chaine c par défaut.

Augeas définit sept lentilles primitives ayant des comportements bidirectionnels distincts et trois types de données, soit *regexp* pour une expression régulière, *string* pour une chaine de caractères et `lens` pour une lentille.

- `key regexp` : La valeur capturée par `get` sert de clé dans le noeud courant de l'arbre. En direction `put`, la clé est retournée. La fonction `create` n'est pas définie.
- `store regexp` : Le comportement est similaire à la lentille `key`, sauf qu'elle s'applique à la valeur du noeud courant plutôt qu'à la clé.
- `label string` : La direction `get` définit la chaine `string` pour la clé dans le noeud courant de l'arbre. Cette lentille ne consomme pas de caractères dans la direction `get`. Les fonctions `put` et `create` ne sont pas définies.
- `value string` : Le comportement est similaire à la lentille `label`, sauf qu'elle s'applique à la valeur du noeud courant plutôt qu'à la clé.
- `del regexp string` : La valeur capturée par `get` n'apparait pas dans l'arbre. En direction `put`, la valeur capturée est restituée. La fonction `create` retourne la chaine `string`.
- `counter string` : Déclare un compteur identifié par `string` et possédant la valeur initiale 1.
- `seq string` : Dans la direction `get`, utilise la valeur du compteur identifié par `string` comme clé du noeud courant. Les fonctions `put` et `create` ne sont pas définies.

Les commentaires débutent par `(*` et terminent par `*)`. Les identifiants sont déclarés par le mot clé `let`. La fonction `test` permet d'exécuter un test unitaire sur les fonctions `get` et `put`. Ces directives sont réunies dans un seul fichier sous le mot clé `module`.

La définition d'une lentille peut inclure des paramètres de type `string`, `regexp` et `lens`. Par exemple, il est courant d'utiliser la lentille

```
let dels (s:string) = del s s
```

pour supprimer une chaine fixe dans la direction `get` et la régénérer dans la direction `put`. Dans cet exemple, le premier argument passé à `del` est de type `string`, tandis que le type `regexp` est attendu. Dans cette situation, une expression régulière acceptant la chaine est compilée automatiquement.

4.1.2 Combinaison des lentilles

Soit les lentilles l, l_1 et l_2. Quatre opérateurs sont définis afin de combiner les lentilles et définir la structure de l'arbre produit.

- Concaténation $l_1.l_2$: La lentille l_1 et la lentille l_2 doivent s'appliquer sur la chaine et l'arbre. Pour assurer que la chaine ne puisse se scinder que d'une seule manière entre l_1 et l_2, il est vérifié que $l_1 \bowtie l_2 = \emptyset$.
- Union $l_1 \mid l_2$: La lentille l_1 ou la lentille l_2 s'applique. Pour éviter toute ambigüité, il est vérifié que $l_1 \cap l_2 = \emptyset$
- Répétition $l?$,$l+$,$l*$: La lentille $l?$ doit s'appliquer zéro ou une fois, la lentille $l+$ doit s'appliquer une fois ou plus et la lentille $l*$ doit s'appliquer zéro ou plusieurs fois. Pour éviter toute ambigüité, il est vérifié pour $l+$ et $l*$ que $l \bowtie l* = \emptyset$.
- Sous-arbre [l] : La lentille entre crochets [l] définit la structure d'un noeud de l'arbre. Les sous-arbres peuvent être imbriqués pour créer une hiérarchie.

Une règle générale à respecter est qu'une seule lentille affectant la clé et la valeur peut être définie pour un noeud de l'arbre, tandis qu'un nombre illimité de lentilles `del` peuvent être utilisées.

Quatre expressions régulières sont établies pour chaque lentille primitive, construites selon leur combinaison. Les expressions régulières sont celles de la clé (`ktype`), de la valeur (`vtype`), du type concret dans la direction `get` (`ctype`) et du type abstrait dans la direction `put` (`atype`).

```
127.0.0.1       francis-laptop  localhost.localdomain   localhost
127.0.1.1       fg
```

Figure 4.1 – Fichier d'exemple `/etc/hosts`

```
{ "1"
  { "ipaddr" = "127.0.0.1" }
  { "canonical" = "francis-laptop" }
  { "alias" = "localhost.localdomain" }
  { "alias" = "localhost" }
}
{ "2"
  { "ipaddr" = "127.0.1.1" }
  { "canonical" = "fg" }
}
```

Figure 4.2 – Arbre abstrait du fichier `/etc/hosts`

4.1.3 Sélecteur de noeud

La recherche et la sélection de noeuds dans l'arbre Augeas se fait par une expression *XPath*[1]. Le chemin est constitué des clés des noeuds de l'arbre séparées par une barre oblique. Ce standard permet de sélectionner un chemin dans l'arbre à la manière des chemins d'un système de fichier Unix. Une fois un noeud sélectionné, il est possible de vérifier son existance, de consulter sa valeur ou de la modifier. La documentation complète du sélecteur est disponible sur le Web[2].

Exemple 4.1.1. Soit le fichier `/etc/hosts` de la figure 4.1 et l'arbre résultant à la figure 4.2, obtenu par la lentille `hosts.aug`. Le détail de la lentille n'est pas nécessaire pour l'exemple et n'est pas recopiée par mesure de concision.

L'expression `/2/ipaddr` sélectionne le noeud `ipaddr` du deuxième enregistrement. On constate que les clés `alias` du premier enregistrement ne sont pas uniques, ce qui n'est pas possible sur un système de fichier. La sélection du deuxième alias se fait en ajoutant l'index au chemin, par exemple avec l'expression `/1/alias[2]`. Finalement,

1. http ://www.w3.org/TR/xpath/
2. http ://augeas.net/page/Path_expressions

il est également possible de sélectionner un noeud basé sur sa valeur. L'expression

```
/*/ipaddr[../canonical='fg']
```

sélectionne le noeud `ipaddr` du deuxième enregistrement.

4.1.4 Exemple synthèse

Soit le module `Intro` de la figure 4.3, qui prend en entrée une liste de noms et d'années de naissance séparés par des espaces.

La lentille `record` de la ligne 4 à 7 est définie par un sous-arbre pouvant se répéter zéro ou plusieurs fois. Le contenu du noeud est la concaténation des lentilles

```
key . del . store . del
```

La lentille `key` capture le nom tandis que la lentille `store` capture l'année. Les espaces et le retour de fin de ligne sont exclus de la représentation abstraite par les lentilles `del`. La fonction `test` de la ligne 17 appelle la fonction `get` de la lentille `record` pour l'entrée `input`. Le résultat est l'arbre à droite de l'égalité, et comprend trois noeuds pour lesquels les clés et valeurs ont été définies. Le second test à la ligne 24 montre la modification de la liste. Le test consiste à l'appel de la fonction `put` de la lentille `record` pour l'entrée `input` après avoir altéré l'arbre en ajoutant l'enregistrement de Renaud et en modifiant l'année de naissance d'Albert. Le résultat suit le test et montre que les modifications ont été appliquées. La figure 4.4 montre la hiérarchie des lentilles qui composent la lentille `record`, incluant les expressions régulières les définissant. Les expressions régulières `ctype` et `atype` sont construites de bas en haut. Les feuilles de cet arbre sont les lentilles `key`, `store` et `del`. Leur parent est une lentille de concaténation, dont le type concret est la concaténation des expressions régulières des types concrets de chaque lentille enfant. Les types `ktype` et `vtype` qui servent à définir la clé et la valeur d'un noeud de l'arbre sont déterminés à partir des lentilles enfants. La lentille de sous-arbre utilise les types `ktype` et `vtype` de la lentille enfant afin de construire le type abstrait complet, tandis que le type concret de la lentille sous-arbre est copié tel quel depuis son enfant. Finalement, la lentille de répétition Kleene applique l'itération sur le type abstrait et concret.

```
1  module Intro =
2
3  (* definition de la lentille record *)
4  let record = [   key /[a-zA-Z]+/
5                 . del /[ ]+/ " "
6                 . store /[0-9]+/
7                 . del "\n" "\n" ]*
8
9  (* initialisation de la variable input avec la chaine de test *)
10 let input =
11 "Maxime 1982
12 Robert 1956
13 Albert 1942
14 "
15
16 (* test de la fonction get de la lentille record *)
17 test record get input =
18   { "Maxime" = "1982" }
19   { "Robert" = "1956" }
20   { "Albert" = "1942" }
21
22 (* test de la fonction put suite a l'ajout d'un enregistrement
23    et la modification d'un enregistrement *)
24 test record put input after set "/Renaud" "1985";
25                              set "/Albert" "1879"  =
26 "Maxime 1982
27 Robert 1956
28 Albert 1879
29 Renaud 1985
30 "
```

Figure 4.3 – Exemple synthèse d'Augeas

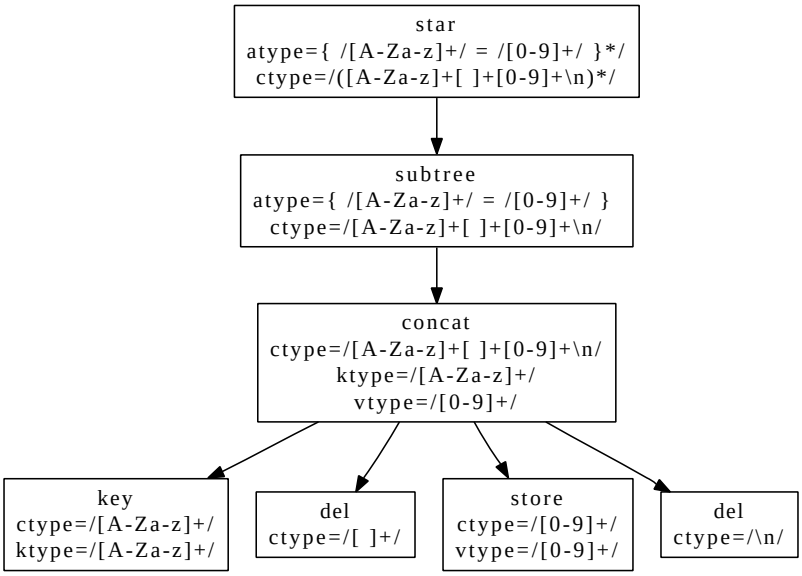


Figure 4.4 – Hiérarchie de la lentille `record`

4.1.5 Lentille récursive

Les versions d'Augeas antérieure à 0.7 étaient limitées à traiter des fichiers décrits par une grammaire régulière. Une grande proportion de fichiers de configuration sont modélisés par un langage régulier, pour laquelle les expressions régulières sont adaptées. Or, plusieurs fichiers de configuration sont composés de structures récursives, tels que les fichiers `dhcpd.conf` et `apache.conf`, et dans le cas général, un fichier XML, qui font partie des langages hors contexte. Dans ce cas, un automate à pile est absolument nécessaire.

La lentille primitive `rec` a été ajoutée pour pallier cette limite. Cette lentille permet de créer une grammaire hors contexte à l'aide des lentilles, et ainsi permettre la récursivité. Dans la direction `get`, l'analyseur syntaxique segmente la chaine d'entrée et les sous-chaines sont ensuite traitées par les lentilles régulières. Il n'existe pas de différence dans la direction `put`, puisque la sélection d'une lentille pour un noeud de l'arbre ne dépend pas de sa position relative dans l'arbre.

Exemple 4.1.2. Soit le module `Antipal` de la figure 4.5, qui accepte le langage $a^i b^i$, et qui produit un arbre abstrait de profondeur i.

```
1  module Antipal =
2
3  (* definition de la lentille recursive exp *)
4  let rec exp = [ key "a" . exp . store "b" ] *
5
6  (* test de la fonction get *)
7  test exp get "aaabbb" =
8    { "a" = "b"
9      { "a" = "b"
10       { "a" = "b" }
11     }
12   }
13
14 (* test de l'augmentation de la profondeur de l'arbre *)
15 test exp put "aaabbb" after set "/a/a/a/a" "b" = "aaaabbbb"
16
17 (* test de la repetition de la lentille exp *)
18 test exp put "aaabbb" after set "/a[2]" "b" = "aaabbbab"
```

Figure 4.5 – Lentille `Antipal`

La ligne 4 déclare la lentille récursive `exp`. La grammaire générée par cette lentille est

Règle	Item de lentille
$S \rightarrow A$	Déclaration rec exp
$A \rightarrow B*$	Répétition Kleene *
$B \rightarrow C$	Sous-arbre [ ]
$C \rightarrow a\ E\ b$	Concaténation key . exp . store
$E \rightarrow A$	Récusion exp

Analyse de l'ambigüité

L'ambigüité pour la direction `put` est facilement décidable à l'aide de la technique du réseau de transition récursif, permettant d'obtenir la liste des sous-arbres pouvant être générés par une variable. Dans le modèle actuel, la position d'un noeud dans l'arbre ne peut servir comme moyen pour éviter une ambigüité. Le type `atype` de chaque lentille doit donc être distinct globalement.

Par contre, il est plus complexe d'effectuer la vérification de l'ambigüité de la

```
module Align =

let ids = ( [ key /[0-9]+/
              . del /[z]+/ "z"
              . store /[0-9]+/
            ]
            . del /[a]+/ "a" )*

let idk = [ key /[0-9]+/
            . del /[z]+/ "z"
            . store /[0-9]+/
            . del /[a]+/ "a" ]*

let rec idr = idr? . [ key /[0-9]+/
                       . del /[z]+/ "z"
                       . store /[0-9]+/
                       . del /[a]+/ "a" ]
```

Figure 4.6 – Lentille `align.aug`

grammaire pour la direction `get`. La version 0.7.0 d'Augeas ne comporte pas de vérification de l'ambigüité de la grammaire. Le concepteur peut donc réaliser une grammaire ambigüe. Celle-ci est acceptée par l'analyseur syntaxique Earley. Dans cette situation, une erreur de conflit peut survenir lors de l'exécution pour une chaine d'entrée particulière. Il revient au concepteur de s'assurer que la grammaire résultante ne soit pas ambigüe. Par exemple, il est possible d'analyser la grammaire résultante avec l'utilitaire `dk.brics.grammar`, l'implémentation de référence des algorithmes ACLA.

4.1.6 Alignement

Le plan de test de l'alignement exécuté est le même que celui utilisé pour XSugar. Un test supplémentaire a été ajouté pour vérifier le comportement dans le cas d'une clé dupliquée. La figure 4.6 montre les trois lentilles utilisées pour effectuer les tests. Ces lentilles acceptent le même langage et produisent le même arbre abstrait. La lentille `ids` est régulière, dont le dernier terme `del` est à l'extérieur du sous-arbre.

Pour la lentille `idk`, le sous-arbre contient tous les termes. Finalement, la lentille `idr` utilise la récursivité.

Ces tests montrent que l'alignement par clé apporte plus de précision que l'alignement par séquence. Le concepteur peut grouper les lentilles dans un sous-arbre de manière à ce que l'alignement résultant minimise la différence ligne à ligne entre les fichiers. L'alignement obtenu par une lentille récursive est identique à celle obtenue par une lentille régulière pour tous les tests effectués.

Table 4.1: Tests d'alignement d'Augeas

Scénario 1 :	Aller-retour sans modification
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `idk` :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `idr` :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Observations :	Toutes les lentilles ont le comportement désiré.
Scénario 2	Modification de l'enregistrement `22` pour `55`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa55z55aaa333zzzz333aaaa`
Sortie `idk` :	`1zz1aa55z55a333zzzz333aaaa`
Sortie `idr` :	`1zz1aa55z55a333zzzz333aaaa`
Observations :	Le scénario 2 montre que pour la lentille `ids`, le dernier terme du nouvel enregistrement n'obtient pas une valeur par défaut, mais la valeur du document d'origine. Les lentilles `idk` et `idr` montrent que le nouvel enregistrement obtient la bonne valeur par défaut.

Table 4.1: Tests d'alignement d'Augeas (suite)

Scénario 3	Déplacement de l'enregistrement `22` à la fin
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa333zzzz333aaa22zzz22aaaa`
Sortie `idk` :	`1zz1aa333zzzz333aaaa22zzz22aaa`
Sortie `idr` :	`1zz1aa333zzzz333aaaa22zzz22aaa`
Observations :	Le scénario 3 montre que la lentille `ids` effectue un alignement par séquence, puisque l'enregistrement `333` se retrouve avec le séparateur de l'enregistrement `22`, tandis que les lentilles `idk` et `idr` font en sorte que les séparateurs suivent leur enregistrement.
Scénario 4	Ajout de l'enregistrement `99` après l'enregistrement `1`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa99z99aaa22zzz22aaaa333zzzz333a`
Sortie `idk` :	`1zz1aa99z99a22zzz22aaa333zzzz333aaaa`
Sortie `idr` :	`1zz1aa99z99a22zzz22aaa333zzzz333aaaa`
Observations :	Le scénario 4 montre que a lentille `ids` fait en sorte que le nouvel enregistrement `99` obtient le séparateur d'origine et le dernier enregistrement `333` obtient un séparateur par défaut. Les lentilles `idk` et `idr` font en sorte que le nouvel enregistrement obtient le séparateur par défaut.
Scénario 5	Suppression de l'enregistrement `22`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa333zzzz333aaa`
Sortie `idk` :	`1zz1aa333zzzz333aaaa`
Sortie `idr` :	`1zz1aa333zzzz333aaaa`
Observations :	Le scénario 5 montre que le séparateur pour l'enregistrement `333` appartient à l'enregistrement supprimé `22`, tandis que les deux autres lentilles préservent le bon séparateur.

Table 4.1: Tests d'alignement d'Augeas (suite)

Scénario 6	Duplication de l'enregistrement `333`
Entrée :	`1zz1aa22zzz22aaa333zzzz333aaaa`
Sortie `ids` :	`1zz1aa333zzzz333aaa22zzz22aaaa333z333a`
Sortie `idk` :	`1zz1aa333zzzz333aaaa22zzz22aaa333z333a`
Sortie `idr` :	`1zz1aa333zzzz333aaaa22zzz22aaa333z333a`
Observations :	Le scénario 6 montre que pour la lentille `ids` le premier enregistrement `333` dupliqué obtient le séparateur appartenant à l'enregistrement `22`, et cause un décalage jusqu'à la fin. Les deux autres lentilles produisent le même résultat, et associent correctement le séparateur. On constate cependant que le premier enregistrement rencontré obtient les séparateurs d'origine, tandis que le second obtient des valeurs par défaut.

4.2 Limites

Il a été montré qu'Augeas est en mesure de traiter un grand nombre de fichiers basés sur des langages réguliers et hors contexte. Or, nous avons constaté une limitation par rapport au traitement des langages balisés, comme le XML. La lentille `xml1` de la figure 4.7 accepte une suite d'éléments XML avec un contenu de texte. À la ligne 9, le test montre le fonctionnement normal de la lentille et l'arbre produit. Le test de la ligne 10 montre que la lentille accepte un élément erroné. Finalement, le test de la ligne 11 montre que la balise de fermeture d'un élément prend la valeur par défaut lors de la création indépendamment de la balise ouvrante, ce qui produit un mauvais résultat.

La lentille `xml2` de la figure 4.8 utilise une chaine fixe plutôt qu'une expression régulière pour le nom des balises. La seule balise acceptée est `<a>`. Ceci fait en sorte que le test de la ligne 11 rejette l'entrée et que le test de la ligne 12 crée l'élément correctement. Or, étant donné qu'il faut lister le nom des balises acceptées et qu'il

existe un nombre infini de balises possibles, alors il est impossible d'écrire une lentille générique capable de traiter un fichier XML quelconque.

Nous proposons une nouvelle lentille primitive *square* pour gérer cette situation. Les spécifications de cette lentille sont présentées en premier lieu. L'implémentation et les résultats obtenus avec cette lentille sont ensuite discutés.

```
1  module Xmlprob =
2
3  let dels (s:string) = del s s
4  let content = store /[a-z]*/
5  let xml1 = [   dels "<" . key /[a-z]+/ . dels ">"
6              . content
7              . dels "</" . del /[a-z]+/ "x" . dels ">" ] *
8
9  test xml1 get "<a>yyy</a>" = { "a" = "yyy" }
10 test xml1 get "<a>yyy</b>" = { "a" = "yyy" }
11 test xml1 put "" after set "/a" "yyy"   = "<a>yyy</x>"
```

Figure 4.7 – Lentille `Xmlprob`

```
1  module Xmlfix =
2
3  let dels (s:string) = del s s
4  let xml2 (tag:string) = [   dels "<" . key tag . dels ">"
5                            . content
6                            . dels "</" . del tag tag . dels ">" ]*
7
8  let a_tag = xml2 "a"
9
10 test a_tag get "<a>yyy</a>" = { "a" = "yyy" }
11 test a_tag get "<a>yyy</b>" = *
12 test a_tag put "" after set "/a" "yyy"   = "<a>yyy</a>"
```

Figure 4.8 – Lentille `Xmlfix`

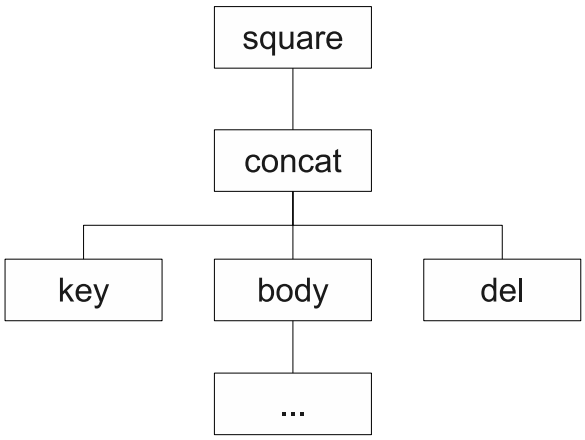


Figure 4.9 – Hiérarchie de la lentille `square`

4.3 Lentille primitive `square`

Hypothèse

Il doit être possible de traiter de manière générique un fichier XML quelconque avec une lentille ayant le comportement bidirectionnel suivant :

- La fonction `get` doit accepter un élément uniquement si la balise ouvrante est identique à la balise fermante. La balise ouvrante doit être utilisée comme clé et la balise fermante ne doit pas apparaitre dans la version abstraite.
- Les fonctions `put` et `create` doivent utiliser la clé pour générer la balise ouvrante et fermante.

Il est à noter que la fonction identité est obtenue pour la lentille `square` si la balise ouvrante est identique à la balise fermante. Si une telle vérification n'est pas effectuée, alors la balise fermante est normalisée pour correspondre à la balise ouvrante dans la direction `put`. Cette contrainte dépasse le cadre des grammaires hors contexte.

Travaux

La lentille `square` a été réalisée. Il s'agit d'une lentille composée, dont la hiérarchie est présentée à la figure 4.9.

La lentille prend deux paramètres, soit une expression régulière `re` et une lentille `body`. L'expression régulière `re` sert pour la clé `key` et la lentille `del`. La chaine par

défaut de `del` est nulle, puisque la clé courante est copiée. Finalement, la lentille quelconque `body` traite le contenu entre les balises.

La vérification de l'ambigüité suit le processus normal établi. Pour la direction `get`, la concaténation des lentilles est testée pour s'assurer que les langages n'aient pas de chevauchement. Dans la direction `put`, il est vérifié que les langages des noeuds sont disjoints.

Résultats

La première étape a été de vérifier le comportement pour le plan de test utilisé à la section 4.2, qui est un sous-ensemble du langage XML. La lentille et les résultats obtenus apparaissent à la figure 4.10. La lentille `body` à la ligne 5 contient les caractères internes de l'élément, équivalent à l'expression régulière `/>[a-z]*<\//`. La lentille `xml` de la ligne 6 définit une séquence de sous-arbre dont le contenu est exactement un élément XML. La lentille `xml` produit l'expression régulière

```
/<[a-z]+>[a-z]*<\/[a-z]+>/
```

ce qui correspond bien à un élément XML simplifié et utilisant une expression régulière plutôt qu'un nom fixe de balise. La ligne 10 montre le fonctionnement normal. La ligne 11 teste qu'une erreur est émise dans le cas d'une erreur syntaxique dans l'entrée. La ligne 12 teste le comportement lors de la création d'un nouveau noeud. Ceci se traduit correctement par la copie de la clé pour la balise fermante.

4.3.1 Conclusion

La lentille `square` permet de modéliser de manière générique un fichier XML simplifié, en obtenant exactement le bon comportement bidirectionnel pour des balises ouvrantes et fermantes.

```
1  module Simple_pass_square =
2
3  let dels (s:string) = del s s
4  let content         = store /[a-z]*/
5  let body            = dels ">" . content . dels "</"
6  let xml             = [   dels "<"
7                            . square /[a-z]+/ body
8                            . dels ">" ]*
9
10 test xml get "<a>yyy</a>" = { "a" = "yyy" }
11 test xml get "<a>yyy</b>" = *
12 test xml put "" after set "/a" "yyy"   = "<a>yyy</a>"
```

Figure 4.10 – Test de la lentille `square`

4.4 Lentille XML

La lentille de test utilisée pour valider la lentille primitive `square` présentée à la section 4.3 est un sous-ensemble du langage XML. Dans cette section, il est déterminé de quelle manière cette lentille peut être utilisée pour modéliser un fichier XML incluant des attributs sur les éléments et des noeuds de texte.

4.4.1 Hypothèse

Il doit être possible de traiter des fichiers XML de manière générique, incluant des attributs et l'indentation des balises.

4.4.2 Travaux

Une lentille permettant de traiter un fichier XML a été réalisée avec la lentille `square`. Cette lentille est représentée à la figure A.1 de l'annexe A. La structure de la représentation abstraite résultante, ainsi que l'écart avec le langage défini par la norme XML sont analysés.

La lentille XML conçue est récursive, permettant un nombre arbitraire d'imbrication des éléments. Le contenu d'un élément peut être du texte, des éléments ou les deux.

Table 4.2 – Solutions pour définir les attributs de la lentille `xml.aug`

a)

```
[  sep_spc
 . key word
 . sep_eq
 . sto_dquote ]*
```

⇒

```
{ "foo" = "bar" }
```

b)

```
[  label "#attribute"
 . sep_spc
 . [ label "#name" . store word ]
 . sep_eq
 . [ label "#value" . sto_dquote ] ]*
```

⇒

```
{ "#attribute"
  { "#name"  = "foo" }
  { "#value" = "bar" }
}
```

Lors de la réalisation de la lentille, il a été impossible de placer la définition d'un attribut dans un seul noeud, tel que présenté en a) à la table 4.2. Dans ce cas, une ambigüité dans la direction `put` se produit, parce qu'un attribut possède la même expression régulière qu'un nom de balise. Il n'est donc pas possible de déterminer si un noeud de l'arbre correspond à un attribut ou à un élément, qui ont deux représentations concrètes différentes. La solution présenté en b) utilise une clé fixe `#attribute` pour éviter une ambigüité dans la direction `put`. Une clé fixe débutant par le caractère dièse, qui ne fait pas partie de l'expression régulière d'une balise, permet d'obtenir deux clés disjointes et évite l'ambigüité.

Cette feuille de style a été utilisée pour tester du point de vue pratique la modification d'un fichier XML. Le détail des tests et des résultats sont à la figure A.2 de l'annexe A.

Ces tests démontrent que la modification de la représentation abstraite se traduit par une modification minimale. Les noeuds `#text` sur un même niveau sont alignés par séquence, étant donné qu'il ne s'agit pas d'une clé unique. Lors des tests, il a été aussi observé que l'ordre des noeuds a de l'importance. Par exemple, dans le cas du test d'ajout d'un attribut, pour que la commande s'exécute avec succès, le noeud `#name` doit absolument précéder `#value`. S'ils sont inversés, alors l'expression régulière pour ces noeuds de l'arbre, décrit par

```
{ /#name/ = /[A-Za-z][.0-9A-Z_a-z-]*/ }{ /#value/ = /[^"]*/ }
```

ne s'applique pas.

Conclusion

Il a été possible de traiter un fichier XML incluant des attributs et des noeuds de texte de manière arbitraire. Cette expérience a démontré qu'il était nécessaire d'utiliser des clés fixes pour éviter des ambigüités dans la direction `put`. Cette technique a pour désavantage de complexifier les requêtes et de causer un alignement par séquence.

4.5 Lentille Apache Httpd

Le serveur Apache Httpd est le serveur Web le plus utilisé sur Internet en 2010 [3]. Sa configuration possède une syntaxe proche de celle des fichiers XML. La syntaxe est essentiellement divisée en deux catégories, soit des sections et des directives. Les sections sont composées d'une balise ouvrante et fermante, contenant des directives ou d'autres sections, nécessitant une lentille récursive `square`. Environ 20 types de sections différentes existent. Quant aux directives, il s'agit de mots clés suivis d'options de configuration. Les directives acceptées par la configuration varient en fonction des modules chargés. Chaque module inscrit et gère ses propres paramètres, ce qui fait en sorte que la configuration est très extensible. Par défaut, plus de 200 directives sont disponibles. Un problème de performance a été observé lorsque toutes les directives et les sections sont listées, ce qui prévient le fonctionnement adéquat de la lentille.

4.5.1 Hypothèse

Il doit être possible de réaliser une lentille pour traiter la configuration d'Apache de manière générique, qui soit plus flexible et performante qu'une lentille exhaustive listant tous les noms de sections et de directives.

3. http ://news.netcraft.com/archives/2010/

4.5.2 Travaux

Trois versions de lentilles `httpd` ont été réalisées pour traiter la configuration d'Apache, soit une version générique, hybride et exacte. La version exacte consiste à lister toutes les directives et sections existantes, tandis que la version générique utilise plutôt des expressions régulières. La version hybride utilise une liste fixe de nom de section. La figure A.3 de l'annexe A contient les lentilles testées.

Les listes des directives et des sections ont été obtenues à partir de la documentation d'Apache. Un script a été réalisé pour générer ces listes automatiquement.

Lors de la réalisation de la lentille générique, il n'a pas été possible d'utiliser des expressions régulières générales pour les sections et les directives à cause des conflits dans la direction `put`. Si tel est le cas, un noeud { `/[a-zA-Z0-9]+/` } peut correspondre à une directive ou une section, dont chacune possède une représentation concrète différente. Pour éviter cette situation, une clé fixe pour les directives, à la manière des attributs XML de la section 4.4, a été utilisée. La lentille hybride utilise l'expression régulière pour les directives calculée par

```
let dname = word - Httpd_sections.sections
```

Cette soustraction rend les langages `dname` et `sections` disjoints. Ainsi, les lentilles hybride et exacte possèdent la même structure d'arbre, tandis que la lentille générique produit un arbre dont le nom des directives n'apparait pas directement sur le chemin XPath.

4.5.3 Résultats

Les lentilles sont évaluées par rapport à la facilité d'utilisation, l'écart entre le langage accepté par la lentille et celui de la configuration et les performances obtenues.

La lentille générique nécessite un niveau d'indirection supplémentaire dans les requêtes contrairement à la lentille hybride et exacte. La hiérarchie des lentilles hybride et exacte est plus simple et facilite les requêtes XPath. Aussi, éviter des clés génériques favorise un meilleur alignement dans la direction `put`.

Table 4.3 – Résultat des performances selon la version de la lentille `httpd`

Lentille	Temps (sec.)		Mémoire (Mo)	
	avec vérif.	sans vérif.	avec vérif.	sans vérif.
Exacte	5,314	0,336	1536	62
Hybride	3,257	0,077	125	6
Générique	0,090	0,048	3	1

La relation entre les ensembles des langages acceptés par chacune des lentilles est décrite par

$$generique \supset hybride \supset exact$$

Les lentilles génériques et hybride acceptent des directives n'existant pas dans le langage de configuration. L'utilisateur peut donc créer des configurations invalides. Cependant, il s'agit aussi d'un avantage, puisqu'il peut exister des directives ou des sections valides pour des extensions spéciales d'Apache qui ne sont pas listées dans la documentation officielle, comme le module `mod_macro` [4]. D'autre part, Apache ne requiert pas que les noms de section et de directive respectent la case. Les lentilles générique et hybride sont donc plus flexibles.

Les résultats des tests de performance sont présentés au tableau 4.3. La mémoire allouée est la somme de chaque `malloc`, et non la taille maximale de mémoire utilisée par le processus, tel que mesuré par `valgrind`. Le temps d'exécution est la moyenne du temps total d'exécution du processus de test tel que mesuré avec la commande Unix `time`. La moyenne du temps d'exécution a été effectuée avec 10 mesures. Les tests ont été réalisés sous Ubuntu 10.04 avec un processeur Intel 32 bits à double coeur de 1,8 GHz et 2 Go de mémoire vive.

La lentille exacte est la moins performante. Ceci s'explique par la taille relative de la lentille, puisque charger chaque directive et effectuer la vérification d'ambigüité sont couteux. Malgré que la lentille hybride ne liste que les 20 noms de section, l'automate des noms de directive obtenue par soustraction des langages est volumineux, ce qui explique sa faible performance. La version générique est la plus performante.

4. http ://cri.ensmp.fr/c̃oelho/mod_macro/

4.5.4 Conclusion

La lentille générique est la plus performante pour le traitement de la configuration d'Apache. Cependant, l'arbre produit comporte des clés fixes, qui complexifient les modifications de l'arbre. Le meilleur compromis est la lentille hybride, permettant d'avoir la même structure d'arbre que la version exacte, mais de manière beaucoup plus efficiente.

Conclusions et travaux futurs

La théorie sur laquelle s'appuient les transformations bidirectionnelles de chaines de caractères a été résumée. Pour assurer la propriété bidirectionnelle, il ne doit pas y avoir d'ambigüité dans la description des langages. Comme il s'agit d'un problème non décidable dans le cas des grammaires hors contexte, une technique faisant appel à l'approximation régulière permettant de détecter les ambigüités a été présentée.

Le fonctionnement de XSugar a été étudié afin de déterminer les techniques permettant d'atteindre une relation bidirectionnelle stricte. Inclure les symboles terminaux sans étiquette dans le document XML provoque des effets de bords indésirables rendant impossible son utilisation en pratique. La fusion d'arbre syntaxique fonctionne, mais ne produit pas nécessairement une modification minimale de la chaine pour une modification donnée, un problème lié à l'alignement.

Un algorithme plus puissant de détection des modifications entre les arbres syntaxiques pourrait être envisagé [11]. Un tel algorithme peut détecter l'ajout, la suppression ou le déplacement d'un groupe de noeuds, ce qui préviendrait l'association séquentielle utilisée par défaut. Une autre avenue d'amélioration consiste dans l'association des noeuds, qui se produit uniquement si les chaines des symboles terminaux correspondent exactement. Cette comparaison pourrait être remplacée par une fonction de similitude plus flexible permettant d'associer deux noeuds quasi identiques. Le résultat obtenu pourrait être différent selon le degré de modification du document. Du point de vue du client, le comportement pourrait apparaitre aléatoire, ce qui n'est pas souhaitable. Plus de recherche est nécessaire pour déterminer si ces changements peuvent être applicables pour régler le problème d'alignement.

Le logiciel Augeas a ensuite été présenté. La lentille `square` a été ajoutée au langage, ce qui permet de traiter efficacement les fichiers ayant des balises ouvrantes et

fermantes, comme les fichiers XML et la configuration du serveur Web Apache. Il a été démontré que le traitement générique est plus performant, mais il reste encore des problèmes d'ambigüité dans la direction `put` empêchant une lentille complètement générique dans le cas de la configuration d'Apache. D'autres moyens permettant d'éviter les ambigüités dans la direction `put` sont nécessaires, telle une version modifiée d'une lentille `key` qui ajoute un préfixe dans la direction `get` et le supprime dans la direction `put`, ce qui a pour effet de différencier les noeuds. Aussi, la détection de l'ambigüité pour les lentilles récursives n'est pas effectuée, ce qui peut causer des problèmes lors du traitement, ce qui serait évité en ajoutant les algorithmes d'approximations régulières à la vérification de la grammaire de la lentille. Finalement, il est à noter que le nom des sections et des directives de la configuration d'Apache ne sont pas sensibles à la case. Cette particularité du langage devrait être pris en considération, sans quoi des configurations valides ne pourront pas être chargées.

L'abstraction des fichiers de configuration offerte par Augeas ouvre la voie pour une nouvelle classe d'outils de gestion de configuration, qu'il s'agisse d'éditeurs graphiques, de suivi des modifications ou de systèmes automatisés de gestion, qui restent à être étudiés dans le futur.

Annexe A

Code source

```
(* XML lens for Augeas
   Author: Francis Giraldeau <francis.giraldeau@usherbrooke.ca>

   Reference: http://www.w3.org/TR/2006/REC-xml11-20060816/
*)

module Xml =

(************************************************************
 *                          Utilities lens
 ************************************************************)

let dels (s:string)   = del s s
let sep_spc           = del /[ \t\n]+/ " "
let sep_osp           = del /[ \t\n]*/ ""
let sep_eq            = del /[ \t]*=[ \t]*/ "="

let nmtoken             = /[a-zA-Z:_][a-zA-Z0-9:_\.-]*/
let word                = /[a-zA-Z][a-zA-Z0-9\._\-]*/
let sto_dquote          = dels "\"" . store /[^"]*/ . dels "\""
let sto_squote          = dels "'" . store /[^']*/ . dels "'"

let comment             = [ label "#comment"
```

```
                            . dels "<!--" . store /([^-]|-[^-])*/
                            . dels "-->" ]
let pi                    = nmtoken - /[Xx][Mm][Ll]/
let empty                 = Util.empty

(************************************************************************
 *                              Attributes
 ************************************************************************)
let attributes = [ label "#attribute"
                   . sep_spc
                   . [ label "#name" . store word ]
                   . sep_eq
                   . [ label "#value" . sto_dquote ] ]*

let prolog               = [ label "#declaration"
                           . dels "<?xml"
                           . attributes
                           . sep_osp
                           . dels "?>" ]


(************************************************************************
 *                              Tags
 ************************************************************************)

let text = [ label "#text" . store /[^<>]+/ ]

let element (body:lens) =
    let h = attributes . sep_osp . dels ">" . body* . dels "</" in
        [ dels "<" . square word h . del />[\n]?/ ">\n" ]

let rec content = element (text|comment|content)

let doc = (sep_osp . prolog)? . sep_osp . content . sep_osp
```

Figure A.1 – Lentille XML générique

```
(* Document de depart *)
let ul1 = "
<ul>
 <li>test1</li>
  <li>test2</li>
   <li>test3</li>
    <li>test4</li>
</ul>
"

(* Representation abstraite du document *)
test Xml.doc get ul1 =
  { "ul"
    { "#text" = "
 " }
    { "li"
      { "#text" = "test1" }
    }
    { "#text" = "  " }
    { "li"
      { "#text" = "test2" }
    }
    { "#text" = "   " }
    { "li"
      { "#text" = "test3" }
    }
    { "#text" = "    " }
    { "li"
      { "#text" = "test4" }
    }
  }

(* Modification d'un noeud texte *)
test Xml.doc put ul1 after set "/ul/li[3]/#text" "bidon" = "
<ul>
 <li>test1</li>
  <li>test2</li>
   <li>bidon</li>
```

```
    <li>test4</li>
</ul>
"

(* Suppression d'un element *)
test Xml.doc put ul1 after rm "/ul/li[2]" = "
<ul>
 <li>test1</li>
     <li>test3</li>
    <li>test4</li>
</ul>
"

(* Ajout d'un element avant le 2e element *)
test Xml.doc put ul1 after insb "a" "/ul/li[2]" = "
<ul>
 <li>test1</li>
  <a></a>
<li>test2</li>
   <li>test3</li>
    <li>test4</li>
</ul>
"

(* Ajout d'un element apres le 1er element *)
test Xml.doc put ul1 after insa "a" "/ul/li[1]" = "
<ul>
 <li>test1</li>
<a></a>
  <li>test2</li>
   <li>test3</li>
    <li>test4</li>
</ul>
"

(* Definition d'un attribut *)
test Xml.doc put ul1 after insb "#attribute" "/ul/li[2]/#text";
                    set "/ul/li[2]/#attribute/#name" "foo";
                    set "/ul/li[2]/#attribute/#value" "bar" = "
```

```
<ul>
 <li>test1</li>
  <li foo=\"bar\">test2</li>
   <li>test3</li>
    <li>test4</li>
</ul>
"
```

Figure A.2 – Résultat des tests pour la lentille XML générique

```
(* Apache HTTPD lens for Augeas
   Authors: Francis Giraldeau <francis.giraldeau@usherbrooke.ca>

   Reference: http://httpd.apache.org/docs/trunk/
*)
module Httpd =
(************************************************************************
 *                            Utilities lens
 ************************************************************************)
let dels (s:string)     = del s s
let sep_spc             = del /[ \t]+/ " "
let sep_osp             = del /[ \t]*/ ""
let sep_eq              = del /[ \t]*=[ \t]*/ "="

let nmtoken             = /[a-zA-Z:_][a-zA-Z0-9:_\.-]*/
let word                = /[a-zA-Z][a-zA-Z0-9\._\-]*/
let sto_dquote          = dels "\"" . store /[^"]*/ . dels "\""
let sto_squote          = dels "'" . store /[^']*/ . dels "'"
let comment             = Util.comment
let eol                 = Util.eol
let empty               = Util.empty
let indent              = Util.indent

(************************************************************************
 *                            Attributes
 ************************************************************************)
let secarg              = [ sep_spc . label "#secarg"
                            . store /[^ \t\n>]+/ ]*
let sto_arg             = sep_spc
                          . store /[^ \t\n>]+|[^ \t\n>].*[^ \t\n>]/
(************************************************************************
 *                            Tags
 ************************************************************************)
(* Exact:
 * let dname = Httpd_directives.directives
 * let sname = Httpd_sections.sections
 *
 * Hybrid:
```

```
 * let dname = word - Httpd_sections.sections
 * let sname = Httpd_sections.sections
 *)
(* Generic *)
let sname = word

let directive = [ indent . label "#directive" . store word
                  . [ label "#arg" . sto_arg ]? . eol ]

let element (body:lens) =
    let h = sto_arg . sep_osp . del ">" ">"
            . eol . body*
            . indent . del "</" "</" in
        [ del /[ \t]*</ " <"
          . square sname h
          . del ">" ">" . eol ]

let rec content = element (content|directive|comment|empty)

let lns = (content|directive|comment|empty)*
```

Figure A.3 – Module httpd

Bibliographie

[1] A. Bohannon, J.N. Foster, B.C. Pierce, A. Pilkiewicz et A. Schmitt.
« Boomerang : Resourceful lenses for string data ».
Dans *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 407–419. ACM, 2008.

[2] C. Brabrand, R. Giegerich et A. Møller.
« Analyzing ambiguity of context-free grammars ».
Science of Computer Programming, 75(3):176–191, 2010.

[3] C. Brabrand, A. Møller et M. I. Schwartzbach.
« Dual syntax for XML languages ».
Information Systems, 33(4-5):385–406, 2008.

[4] C. Brabrand et J. G. Thomsen.
« Typed and unambiguous pattern matching on strings using regular expressions ».
Dans *PPDP'10 : Proceedings of the 12th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 243–254. ACM, 2010.

[5] J. Earley.
« An efficient context-free parsing algorithm ».
Commun. ACM, 13(2):94–102, 1970.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, C. Pierce et A. Schmitt.
« Combinators for bidirectional tree transformations : A linguistic approach to the view-update problem ».
ACM Trans. Program. Lang. Syst., 29(3):1–65, 2007.

[7] J. N. Foster, A. Pilkiewicz et B. C. Pierce.
« Quotient lenses ».
SIGPLAN Not., 43(9):383–396, 2008.

[8] D. E. Knuth.
« On the translation of languages from left to right ».
Information and control, 8:607–639, 1965.

[9] M.-J. Nederhof.
« Practical experiments with regular approximation of context-free languages ».
Computational Linguistics, 26(1):17–44, 2000.

[10] T. A. Sudkamp.
Languages and Machines.
Addison Wesley, 1996.

[11] K.-C. Tai.
« The tree-to-tree correction problem ».
J. ACM, 26(3):422–433, 1979.

[12] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan et G. Jung.
« Approaches for service deployment ».
Internet Computing, IEEE, 9(2):70–80, 2005.

[13] K. Thompson.
« Programming Techniques : Regular expression search algorithm ».
Commun. ACM, 11(6):419–422, 1968.