Spécification et réalisation d'un formalisme générique pour la segmentation multiple de documents textuels multilingues

Julien Quint

▶ **To cite this version:**

Julien Quint. Spécification et réalisation d'un formalisme générique pour la segmentation multiple de documents textuels multilingues. Informatique [cs]. Université Joseph-Fourier - Grenoble I, 2002. Français. NNT : . tel-00521940

HAL Id: tel-00521940
https://theses.hal.science/tel-00521940

Submitted on 29 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE 1
UFR D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES

Numéro attribué par la bibliothèque

THÈSE
pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER
Discipline : INFORMATIQUE
présentée et soutenue publiquement

par

M. Julien QUINT

le 18 novembre 2002

SPÉCIFICATION ET RÉALISATION D'UN FORMALISME GÉNÉRIQUE POUR LA SEGMENTATION MULTIPLE DE DOCUMENTS TEXTUELS MULTILINGUES

JURY

Président	M. Yves LEDRU
Rapporteurs	M. Jacques CHAUCHÉ
	M. Jean VÉRONIS
Examinateur	M. Marc DYMETMAN
Directeur de thèse	M. Christian BOITET
Co-directeur	M. Jean-Pierre CHANOD

Thèse préparée au sein des laboratoires GETA-CLIPS (IMAG, CNRS & UJF) et XRCE

Remerciements

Je remercie toutes celles et tous ceux qui m'ont soutenu et aidé à terminer cette thèse, les équipes qui m'ont accueilli, et les membres du jury.

Table des matières

Table des figures

Liste des tableaux

Introduction

Une première étude de la morphologie du français à l'aide de la suite d'outils dits de calcul à états finis de Xerox (XFST[1]) montrait que ce type de langage spécialisé pour linguistes semblait adapté à une grande classe d'applications, allant de la segmentation en mots et en syllabes de textes écrits dans des systèmes d'écriture sans séparateurs à la (re-)structuration d'un document vers une forme XHTML propre, en passant par le balisage morphosyntaxique, l'analyse morphologique complète, et l'analyse syntaxique de surface (*shallow parsing*).

Des résultats sur l'augmentation de la puissance théorique des transducteurs finis obtenable par la « fermeture à point fixe », par les *flag diacritics* et par l'utilisation de poids, permettaient même de penser qu'il était possible d'aller jusqu'à l'analyse syntaxico-sémantique complète.

Toutes ces idées sont en fait connues depuis longtemps, et de nombreux outils, souvent gratuits et largement disponibles sur les plates-formes courantes, les implémentent, mais toujours de façon partielle. Mais d'autre part, après avoir étudié les systèmes réellement utilisés par les différentes sociétés et laboratoires, il a fallu constater qu'en réalité, il n'y avait aucune approche générale.

Par exemple, les nombreuses sociétés japonaises qui vendent des outils de manipulation de textes en japonais ont toutes besoin de segmenter ces textes, et ont toutes développé des outils propres, avec des approches voisines mais incompatibles, et cela sans s'appuyer sur aucun outil générique comme (par exemple) XFST.

Il était et il reste intéressant de chercher la raison de cet éclatement, et de proposer un formalisme implémentable, assez complet et simple, pour que des développeurs d'applications linguistiques puissent l'utiliser, en pouvant reprendre telles quelles leurs idées algorithmiques de base. Cela permettrait aussi la réutilisabilité des applications, grâce à la séparation plus claire entre les aspects de contrôle et les données linguistiques statiques, dont le coût de développement est considérable.

La première partie de cette discussion est une analyse détaillée de l'état de l'art dans le domaine de l'analyse présyntaxique, de façon à aboutir à une liste, assez longue, de fonctionnalités indispensables que devrait avoir un outil universel dans ce domaine. Cela permet d'arriver à définir des structures de données et des structures de contrôle qui généralisent celles que l'on trouve également dans des domaines connexes. Ainsi, la structure de données principale de Sumo, le formalisme proposé, est analogue à la partie monolingue d'une mémoire de traduction dite « à étages », dont la supériorité par rapport aux mémoires de traduction classiques a été montrée.

[1] `http://www.xrce.xerox.com/competencies/content-analysis/fst/`

La seconde partie commence par un exemple relativement complexe, et se poursuit par une présentation systématique de la syntaxe et de la sémantique de Sumo. Il n'a pas encore été possible de réaliser le système complet, mais un prototype qui généralise de plusieurs façons le traitement de base des transducteurs d'états finis pondérés a été implémenté et est décrit ici. Les choix d'implémentation, tant pour le prototype (actuellement écrit en Perl) que pour le système complet, sont également discutés.

Enfin, la troisième partie présente en détail la réalisation du prototype, et l'illustre par quelques exemples formels et linguistiques. La conclusion de la discussion présente des perspectives d'application du système complet, et envisage quelques extensions fonctionnelles intéressantes.

Première partie

De l'absence de solution universelle pour la segmentation de texte

Introduction

À moins que l'on ne se place dans le cadre de systèmes de gestion de collections de documents numériques, un document textuel n'est jamais considéré comme une seule unité, mais plutôt comme une hiérarchie, ou au moins une séquence d'unités plus petites : caractères, mots, phrases, paragraphes, etc. L'analyse présyntaxique regroupe les différents traitements subis par un texte écrit *avant* une éventuelle analyse syntaxique complète, qui peut elle-même être suivie d'une analyse sémantique, pragmatique, etc. Cela comprend aussi bien des étapes d'analyse complexe, comme l'analyse morphologique, que des tâches plus simples, comme la normalisation du texte. On peut même y ajouter des étapes d'analyse relativement poussées, comme l'extraction de syntagmes (groupes nominaux et groupes verbaux, noms propres, etc.) et l'analyse de surface.

Le point commun entre les différents traitements présyntaxiques est qu'ils proposent une transformation du texte : en entrée, le texte apparaît comme une suite d'unités d'un certain ordre ; en sortie, il apparaît comme une suite d'unités de nature différente. Ces unités peuvent également être enrichies par des informations extraites lors de l'analyse.

Plus précisément, on propose les étapes qui suivent comme fondements de l'analyse présyntaxique.

L'acquisition du texte. Comme tout document numérique, un texte électronique n'est constitué que d'une suite de bits. Le texte lui-même ne peut en être extrait qu'en connaissant l'encodage et le format du document. L'encodage établit la correspondance entre les caractères et leur représentation numérique. Le format définit la hiérarchie des éléments textuels. C'est une étape fondamentale pour toute application linguistique. Évidemment, pour que le texte extrait puisse être convenablement exploité, l'application doit connaître son encodage et son format.

> *Exemple* : un document en XML [Bray *et al.*, 2000] a un format particulier, qui consiste à baliser les éléments textuels pour expliciter la structure du document, et à définir des attributs pour les éléments. Il existe des éléments particuliers, notamment la *déclaration XML*, qui apparaît au début du document, et qui donne des indications sur la façon dont celui-ci doit être lu, et en particulier quel est son encodage.

La normalisation et le nettoyage du texte. Lorsque l'on s'intéresse à du texte courant, peu contraint, il faut considérer toutes les variations possibles. Espacements et typographie à l'intérieur d'un même texte ne sont pas toujours cohérents, et la phase d'acquisition peut introduire des éléments indésirables, quand ceux-ci ne font tout simplement pas partie du document d'origine. Nettoyage et normalisation consistent à supprimer

un maximum de bruit, puis à simplifier et à unifier les variations à l'intérieur du texte pour le rendre tout à fait cohérent.

Exemple : une application travaillant uniquement sur les éléments textuels contenus dans un document XML peut commencer par supprimer tout autre contenu (balises, attributs, commentaires...), puis normaliser les espacements en supprimant les espaces au début et à la fin de chaque élément, transformer retour à la ligne et tabulations en simple espace, et transformer tout le texte en minuscules. Le fragment de document suivant :

```
<p class="section">
  Un élément textuel
avec  des   blancs non
uniformes.
</p>
```

donne la chaîne

```
un élément textuel avec des blancs non uniformes.
```

une fois le texte nettoyé et normalisé de la sorte.

Ces deux opérations peuvent provoquer volontairement une perte d'information. Par exemple, un moteur de recherche d'information en français peut supprimer les accents dans une requête pour augmenter le rappel, éventuellement au détriment de la précision de la recherche (c'est ce que fait la version française du moteur de recherche Google[2]).

La segmentation du texte. Comme on va le voir plus bas, un type de segmentation important est la segmentation en mots, et elle fait partie intégrante de l'analyse morphologique. De même, l'acquisition du texte peut être considérée comme une segmentation en caractères. Cependant, de nombreuses autres segmentations sont possibles, de différentes natures : linguistique (syllabes, mots ou phrases), structurelle (sections, chapitres) ou artificielle (quantité de texte gérable confortablement par une application).

L'analyse morphologique. L'analyse morphologique est une étape capitale de l'analyse présyntaxique qui correspond clairement à un domaine de la linguistique traditionnelle, la morphologie. Comme il s'agit d'un processus qui peut être très complexe, on propose de le définir en séparant trois sous-processus qui sont :

1. la segmentation en mots : souvent appelée *tokenization* (ou *tokenisation*) dans la littérature, on propose d'employer le terme « itémisation » en français ; considérant qu'un item est une unité lexicale (mot, expression figée, etc.), soit un lexème ou un *token* en anglais.
2. la lemmatisation : c'est l'analyse de chaque lexème, qui extrait un lemme, soit la forme de base du mot analysé, et ses informations morpho-syntaxiques. Il est courant de désigner cette seule phase sous le terme d'analyse morphologique ; une application de ce type doit être précédée d'un itémiseur.
3. la désambiguïsation syntaxique : le découpage du texte en mots est parfois ambigu, comme peut l'être la lemmatisation d'un lexème donné. Il en résulte donc de nombreuses interprétations possibles, que l'on peut désambiguïser ici. Une application spécialisé dans ce type de désambiguïsation est appelée un étiqueteur syntaxique

[2] `http://www.google.fr/`

(*part-of-speech tagger*, abrégé en *POS tagger*) ; on utilise parfois un jeu d'étiquettes réduit par rapport aux informations du lemmatiseur.

L'extraction de syntagmes. Les résultats produits par l'analyse morphologique sont souvent exploitables tels quels par un analyseur syntaxique, qui peut voir le texte sous la forme d'une suite de lemmes convenablement étiquetés. Cependant, il est possible d'extraire plus d'information sans avoir recours à une analyse syntaxique complète, comme par exemple reconnaître des syntagmes et des entités dans le texte en se fondant sur le résultat de l'analyse morphologique.

Les deux principales applications sont d'une part la reconnaissance d'entités et d'autre part le morcelage (*chunking*). La reconnaissance d'entités permet de reconnaître des noms propres (lieux, personnes), des dates ou des termes techniques (groupes nominaux) dans un texte. Le morcelage est une nouvelle étape de segmentation, qui assemble les mots en groupes élémentaires[3] nominaux, verbaux, etc. Ces unités se combinent alors pour former les phrases.

L'analyse syntaxique de surface. On trouve dans [Hammerton *et al.*, 2002] une rapide introduction au *shallow parsing* (analyse syntaxique de surface), qui utilise les informations du « morceleur » et extrait les relations existantes entre les différents segments (sujets, objets, compléments, etc.) Il ne s'agit pas d'une analyse complète, dans le sens où l'on ne construit pas un arbre syntaxique mais seulement une énumération de ces liens entre différents syntagmes.

Cette première partie pose la plupart des questions auxquelles la suite de la discussion va devoir répondre. Dans le chapitre 1, ce sont les outils pour l'analyse syntaxique qui sont étudiés, des outils les plus rudimentaires (linguistiquement parlant), ne proposant aucune abstraction linguistique, mais paradoxalement une liberté de mouvement infinie, aux formalismes les plus sophistiqués. Le chapitre 2 revient sur des systèmes concrets d'analyse présyntaxique, notamment d'analyse morphologique. On y voit ici aussi des analyseurs très simples et des systèmes très perfectionnés. Le chapitre 3 fait donc une synthèse de cet état de l'art, et surtout dégage les principales caractéristiques que devrait présenter un formalisme complètement indépendant de la langue, dédié spécifiquement à la segmentation de documents textuels.

Cette étude confirme que, au contraire de la situation qui prévaut en analyse syntaxico-sémantique, il n'existe à ce jour aucun outil informatique général (langage spécialisé et environnement) permettant à des linguistes non-informaticiens de programmer toutes les applications citées, pour toutes les langues, quelle que soit la simplicité ou la complexité de leur morphologie et de leur système d'écriture.

[3] C'est-à-dire sans récursion sur des syntagmes ou classes de niveaux plus élevées dans une hiérarchie (de classes syntaxiques ou syntagmatiques) donnée [Vauquois et Chappuy, 1988].

Chapitre 1

Une étude de l'analyse présyntaxique, entre la complexité informatique et la félicité linguistique

Introduction

La présentation des approches actuelles de l'analyse présyntaxique peut s'aborder selon deux angles différents : d'une part, la complexité informatique de l'approche, c'est-à-dire les connaissances nécessaires en informatique pour pouvoir les mettre en œuvre ; et d'autre part, la félicité linguistique [Shieber, 1987], c'est-à-dire le confort et la puissance que la méthode donne à un linguiste pour exprimer les problèmes linguistiques abordés. La figure 1.1 montre la place occupée par les différentes approches décrites ici sur un graphe où la félicité linguistique figure en ordonnées et la complexité informatique figure en abscisses.

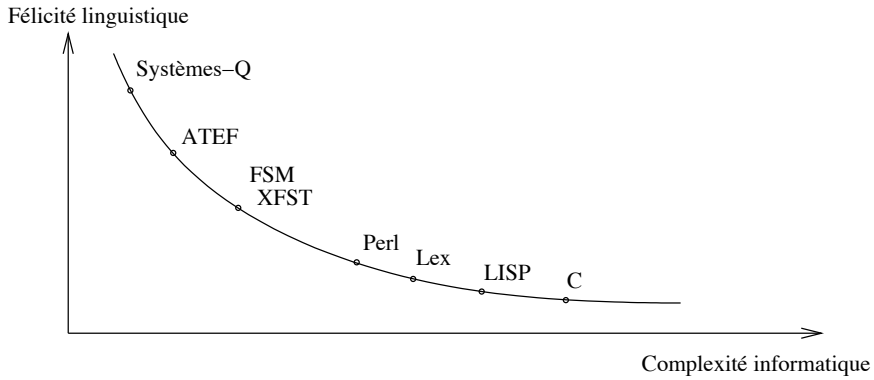

FIG. 1.1 – Techniques d'analyse morphologique

Ce chapitre est organisé selon cette courbe de progression, en suivant l'axe de complexité informatique de manière décroissante. On commence (section 1.1) par la programmation *ad hoc* d'analyseurs morphologiques, directement programmés dans un langage de programmation donné, ou à l'aide d'outils prévus à cet effet mais qui restent du domaine de la programmation et ne contiennent pas de notion linguistique spécifique. Viennent ensuite (section 1.2) les structures de données employées pour l'analyse morphologique, et plus particulièrement les techniques fondées sur les états finis. Enfin, de véritables formalismes pour l'analyse morphologique (ou utilisables à cette fin) sont vus dans la section 1.3.

1.1 Programmation ad hoc d'analyseurs morphologiques

1.1.1 Règles et heuristiques

On peut avancer deux raisons pour lesquelles une application présyntaxique est réalisée directement avec un langage de programmation classique et généraliste, comme C ou LISP. La première est que le problème à résoudre paraît très simple, et doit par conséquent être résolu simplement et efficacement. La seconde est, au contraire, que l'on ne dispose pas nécessairement de description linguistique satisfaisante des phénomènes concernés, et les formalismes disponibles ne sont alors pas considérés d'une grande utilité.

1.1.1.1 Applications présyntaxiques triviales ou analyse morphologique sophistiquée

Applications présyntaxiques triviales. L'application présyntaxique « triviale » par excellence est l'itémisation (« triviale » entre guillemets, car il apparaît très vite que cette vision des choses n'est pas la bonne). Un programme découpant du texte selon les blancs et les caractères non-alphanumériques s'écrit très simplement dans n'importe quel langage de programmation. Un tel programme en C ne fait que quelques lignes et produit un résultat correct pour environ 90 % des cas. Pour traiter les cas particuliers (« parce que », « aujourd'hui »), une liste d'exceptions peut s'ajouter.[1]

Mais certains analyseurs morphologiques sont aussi écrits de la sorte. En anglais, par exemple, le système de flexions est très simple : pluriel des noms en *-s* avec quelques exceptions, prétérit et participe passé des verbes en *-ed* sauf pour une liste connue d'exceptions, auxquelles on ajoute des altérations phonologiques faciles à décrire...

Si les règles morphologiques sont gérées par quelques fonctions, un lexique devient désormais nécessaire. Dans un cadre aussi simpliste, le lexique ne contient que peu d'informations et se résume souvent à une liste de mots. Les capacités de stockage disponibles actuellement permettent de gérer des listes de mots de très grande taille (des dizaines de milliers, voire des centaines de milliers d'entrées) sans problème particulier.

Il ne reste plus qu'à programmer les méthodes d'accès à ces lexiques, mais il ne s'agit là que d'une tâche dont tout étudiant en informatique est capable de s'acquitter. Représentation arborescente ou utilisation de bases de données ne sont que deux moyens de procéder ; la représentation d'une liste de mots par un automate d'états finis est également pertinente, mais il en sera largement question plus bas.

Analyse morphologique sophistiquée. L'autre raison invoquée pour la programmation complètement *ad hoc* d'un analyseur morphologique est l'absence de description linguistique utilisable, ou l'inadéquation des formalismes disponibles pour traiter les phénomènes morphologiques de certaines langues (voir le chapitre 3 à ce sujet). Chinois et japonais en sont de bons exemples.

Il est également possible de résoudre un problème complexe par la force brute, qui ne nécessite alors pas de technique de programmation particulière. Un lemmatiseur du français peut très bien s'envisager en créant (c'est là un travail de linguistique plus que d'informatique)

[1] Notons cependant que la liste peut atteindre des centaines d'entrées, *cf.* les prépositions composées de Gross comme « pour autant que », « à côté de », « au-dessus de », etc.

une liste exhaustive de mots fléchis (et en ajoutant quelques règles pour la gestion de mots inconnus). Si ce n'est pas une solution très excitante scientifiquement, dans un cadre bien délimité elle peut s'avérer tout à fait adéquate. C'est par exemple l'approche de SMorph (section 2.3.1).

En chinois, des analyseurs morphologiques ou segmenteurs très complexes sont ainsi réalisés, où la solution de chaque problème (traitement de classes de mots particulières, formation de nouveaux mots, etc.) est programmée au cas par cas par un informaticien à partir de données fournies par un linguiste. Un bel exemple est SLex [Swen et Yu, 1999], développé à l'Université de Pékin, qui est un segmenteur du chinois très performant programmé en C.

1.1.1.2 Heuristiques de « plus longue forme »

Aller plus loin que le simple découpage en suivant les séparateurs nécessite un principe directeur. L'étape suivante la plus logique est de faire appel à une ou des heuristiques pour décider, lors du parcours d'une séquence de caractères, si l'on a atteint la limite d'un mot ou non. La catégorie d'heuristiques la plus employée est sans conteste celle de « plus longue forme ».

Heuristique de la plus longue chaîne. L'heuristique de la plus longue chaîne semble très naturelle et est employée aussi bien dans des applications linguistiques que dans des systèmes de transduction de chaînes plus généraux. Son origine remonterait aux premières tentatives de systèmes de traduction automatique russes. Son principe de fonctionnement est des plus simples : soit un dictionnaire contenant toutes les formes de la langue et une chaîne à analyser. Parmi tous les préfixes de la chaîne à analyser correspondant à une entrée du dictionnaire, c'est le plus long qui est choisi comme un mot, et le découpage continue avec la chaîne restante.

Heuristique de la plus longue chaîne en parcours inverse. Variante de l'heuristique de la plus longue chaîne, elle est employée pour trouver des découpages possibles qui auraient été négligés par l'approche « gloutonne » de l'heuristique de plus longue chaîne. Son principe de fonctionnement est le même, mais en parcourant le texte en sens inverse (de droite à gauche).

Heuristique du plus petit nombre d'items. Cette heuristique est proche des deux précédentes mais cherche à maximiser la taille des segments globalement, et non localement. La longueur moyenne des segments doit être la plus grande possible, et donc leur nombre total le plus bas possible, d'où le nom fréquemment relevé dans la littérature de *shortest tokenization* (« plus courte itémisation »).

> *Exemples* : soit la chaîne `ABCDEFG` à segmenter avec un dictionnaire contenant les mots `A`, `AB`, `B`, `BC`, `BCDE`, `BCDEF`, `C`, `CD`, `D`, `DE`, `E`, `F`, `FG` et `G`. Les découpages possibles selon les trois heuristiques présentées ici sont :
> – plus longue chaîne, parcours de gauche à droite : `AB/CD/E/FG` ;
> – plus longue chaîne, parcours de droite à gauche : `A/BC/DE/FG` ;
> – plus petit nombre d'items : il y a deux possibilités différentes, `A/BCDEF/G` ou `A/BCDE/FG`.

Comme le montre cet exemple, cette heuristique peut tout de même mener à plusieurs découpages différents. Les trois heuristiques peuvent aussi être combinées pour générer efficacement un ensemble de découpages plausibles selon un dictionnaire de segmentation.

Variantes. Ces trois heuristiques ne sont que quelques exemples possibles. Un itémiseur du chinois [Chen et Liu, 1992] raffine la technique en proposant des choix plus fins, comme les séquences de trois mots de longueur maximum, la minimisation de l'écart-type de la longueur des mots, la minimisation du nombre de morphèmes liés, etc.

1.1.1.3 Segmentation critique

Selon [Guo, 1997], les heuristiques de « plus longue forme » sont les plus largement utilisées pour l'itémisation dans différentes langues mais ne permettent pas toujours de reconnaître tous les découpages suivant le principe de la plus longue forme. Il propose alors un modèle de segmentation qui caractérise mieux ces heuristiques : ce principe plus général est celui de la segmentation critique.

La segmentation critique se fonde sur les points et les fragments critiques. En se plaçant dans l'hypothèse d'un dictionnaire fermé, les points critiques d'une phrase sont exactement les limites non-ambiguës entre deux items. Un fragment critique est alors la plus longue sous-chaîne dans laquelle les limites entre items sont ambiguës. Les fragments critiques d'une chaîne sont délimités par ses points critiques.

Outre le début et la fin d'une phrase, qui sont des points critiques ordinaires, une phrase peut contenir zéro ou plusieurs points critiques extraordinaires, et par conséquent un ou plusieurs fragments critiques. La segmentation, ou itémisation critique, consiste alors à identifier les points et fragments critiques de la phrase, qui permettent ainsi un morcelage non-ambigu. Une segmentation complète de la phrase s'obtient en segmentant chaque fragment. Comme l'identification des points critiques se fait en un temps linéaire par rapport à la longueur de la chaîne à analyser, cette stratégie de morcelage peut s'avérer payante.

Hypothèse du dictionnaire fermé. Cette hypothèse est intéressante car elle garantit, selon la technique de segmentation critique, d'énumérer toutes les segmentations possibles. Ainsi, à défaut de pouvoir toujours identifier la bonne segmentation (et d'atteindre une précision parfaite), un rappel parfait serait théoriquement envisageable. De même, un dictionnaire incomplet peut mener à l'impossibilité de segmenter une chaîne lorsqu'une forme inconnue est rencontrée.

Si l'exhaustivité d'un dictionnaire est un but impossible à atteindre dans le traitement des langues naturelles, la segmentation critique reste applicable. Une « fermeture » du dictionnaire est faisable en ajoutant au dictionnaire chaque caractère de la langue considérée comme un mot. Cette modification du dictionnaire justifie également les heuristiques de plus longue forme : en chinois, par exemple, l'utilisation d'une heuristique de plus courte chaîne mènerait systématiquement au découpage du texte caractère par caractère.

1.1.2 Analyse lexicale à l'aide d'expressions régulières

1.1.2.1 L'analyse lexicale et l'analyse morphologique

Il existe un parallèle évident entre l'analyse des langues naturelles et l'analyse des langages de programmation (la compilation). Dans un compilateur, on retrouve toutes les étapes d'analyse décrites dans l'introduction de cette première partie : normalisation des espacements,

suppression des commentaires, analyse lexicale, suivies ensuite de l'analyse syntaxique, puis d'étapes sémantiques (vérification des types, génération de code, etc.)

L'analyse lexicale d'un langage de programmation correspond à l'analyse morphologique telle qu'elle est décrite plus haut. La tâche de l'analyseur lexical au sein d'un compilateur est de produire à partir du code source la liste des lexèmes (nommés *tokens* ou items dans [Aho *et al.*, 1986]) qui sera passée à l'analyseur syntaxique. Les lexèmes sont également étiquetés : identificateur, opérateur, littéral, etc. On voit donc le parallèle avec l'analyse morphologique : on a bien les tâches de segmentation d'une part, et de lemmatisation et désambiguïsation d'autre part (qui concernent l'étiquetage). Une différence notable est que les langages artificiels sont conçus pour être interprétés par des machines et présentent un minimum d'ambiguïtés, ce qui fait de l'analyse lexicale une tâche purement mécanique. De plus, les unités lexicales sont généralement atomiques et ne se décomposent pas en morphèmes libres et liés : les langages de programmation ont rarement recours aux flexions ou à la composition, sauf exceptions aberrantes (comme Perligata[2] de Damian Conway, permettant d'écrire des programmes Perl en latin).

Comme l'écriture d'un analyseur lexical est un travail méthodique et fastidieux, des outils comme `lex`, `flex` et autres ont été développés pour le faciliter. `lex` est choisi ici comme exemple, car c'est un programme classique, dont les principes sont largement repris par les autres. `lex` est un programme qui génère des programmes à partir d'une description de chaque classe de lexèmes (*e.g.* identificateurs, chaînes de caractères, mots-clés, commentaires, etc.) et des actions particulières à effectuer une fois qu'un lexème est identifié.

Le programme produit par `lex` à partir de cette description fournit en particulier une fonction d'analyse nommée `yylex()` qui balaie le texte en entrée, de gauche à droite, et avance à chaque appel de lexème en lexème, exécutant les actions prévues pour ce type de lexème et retournant un pointeur vers le lexème ainsi lu. Il s'agit ensuite d'exploiter cette fonction pour analyser le programme lexème par lexème ; par exemple, l'analyseur syntaxique accède ainsi au code source lexème par lexème par appels successifs à cette fonction.

On peut noter pour terminer qu'un analyseur lexical correspond plus précisément à un analyseur présyntaxique qu'à un analyseur morphologique. Dans un compilateur, c'est la première étape de l'analyse, et le traitement des commentaires, la gestion des espacements et la normalisation des items échoient également à l'analyseur lexical, qui effectue ces prétraitements à la volée.

1.1.2.2 Description des lexèmes par expressions régulières

Les classes de lexèmes sont définies à l'aide de motifs qui décrivent les formes possibles d'un lexème. Un identificateur en C, par exemple, se définit par : « un caractère alphabétique, suivi de zéro ou plusieurs caractères alphanumériques » ; un nombre décimal par « une séquence de zéro ou plusieurs chiffres, qui peut être suivie d'un point et d'une séquence de chiffres ». Pour exprimer ces motifs clairement et sans ambiguïté, `lex` emploie des expressions régulières qui définissent des langages rationnels.

La classe des langages rationnels sur un alphabet fini Σ est construite inductivement à partir de la base formé du langage vide, du langage réduit à ϵ et des langages singletons

[2] `http://www.csse.monash.edu/~damian/papers/HTML/Perligata.html`

réduits à une lettre par les trois opérations suivantes :

1. concaténation ;
2. union ;
3. itération (ou étoile de Kleene).

Une expression régulière est alors une description d'un langage rationnel et est définie de la même manière :

1. Le langage vide est dénoté par le symbole $\emptyset$;
2. le langage $\{\epsilon\}$ est dénoté par le symbole ϵ ;
3. si a est un symbole de l'alphabet Σ , le langage $\{a\}$ est dénoté par a ;
4. si r dénote le langage L alors (r) désigne également le langage L ;
5. si r et s sont deux expressions régulières décrivant les langages rationnels $L(r)$ et $L(s)$, alors $(r)(s)$ est une expression régulière qui décrit $L(r).L(s)$;
6. si r et s sont deux expressions régulières décrivant les langages rationnels $L(r)$ et $L(s)$, alors $(r)|(s)$ est une expression régulière qui décrit $L(r) \cup L(s)$;
7. si r dénote le langage L alors $(r)*$ désigne le langage $L*$;
8. les parenthèses peuvent être omises si l'on convient de la priorité usuelle des trois opérateurs, où l'étoile est prioritaire par rapport à la concaténation, qui est prioritaire par rapport à l'union.

   Exemple : les deux exemples de cette section peuvent s'exprimer à l'aide d'une expression régulière. Un identificateur est décrit par l'expression :

   `(a|A|b|B|...|z|Z|_)(a|...|Z|_|0|1|...|9)*`

   qui autorise des identificateurs comme `i`, `pointeur_liste`, `var3`, mais interdit `2eme` ou `avec des blancs`. De même, un nombre décimal est décrit par l'expression :

   `(0|...|9)*(.|ϵ )(0|...|9)(0|...|9)*`

   Cela autorise des nombres comme `"0.123"`, `".43"`, mais interdit `"12."` ou `"."`.

Si tous les langages rationnels peuvent être décrits par des expressions régulières, la présence de seulement trois opérateurs rend la tâche parfois difficile. C'est pourquoi de nombreux autres opérateurs ont été introduits, qui peuvent tous être réduits aux trois opérateurs classiques, pour rendre les expressions régulières plus faciles à écrire. Une liste non-exhaustive de certains de ces opérateurs et notations suit. La syntaxe particulière de de `lex` est reprise, car il n'y a malheureusement pas de syntaxe universelle pour la plupart des ces opérateurs.

- le caractère supplémentaire `"."` décrit une occurrence de n'importe quel caractère ;
- l'écriture d'expressions comme `a|A|b|B...` est laborieuse et peut mener à des erreurs si l'on veut énumérer des ensembles de caractères plus compliqués (par exemple les lettres accentuées en français). On peut disposer d'intervalles de caractères, comme `[0-9a-fA-F]`, qui représente l'ensemble des chiffres possibles en hexadécimal. Un intervalle peut aussi être défini par complémentation, comme `[^ \t\n]`, qui est un ensemble contenant tous les caractères *sauf* les caractères d'espacement (blanc, tabulation, nouvelle ligne) ;

– quand on travaille avec des jeux de caractères grands et compliqués, comme Unicode, les classes de caractères nommées sont très utiles [Davis, 2000]. Par exemple, `[:alnum:]` contient tous les caractères alphanumériques pour la *locale* donnée (chiffres, lettres et souligné, et donc, en français, les caractères accentués), ou `[:blank:]` tous les caractères d'espacement. Cette définition est plus claire et plus complète que l'énumération explicite des caractères ;
– Outre l'étoile de Kleene, qui signifie « zéro ou plusieurs occurrences », il est pratique de définir « au moins une occurrence » (`"+"`), « zéro ou une occurrence » (`"?"`), « exactement n occurrences » (`{n}`), « au moins n occurrences » (`{n,}`), « au plus n occurrences » (`{,n}`) ou encore « entre n et m occurrences » (`{n,m}`).

Toutes ces opérations et notations ne sont en réalité que des raccourcis qui facilitent la tâche de l'auteur de l'analyseur lexical. Les expressions régulières ainsi écrites peuvent toutes se réécrire à l'aide des seuls opérateurs de concaténation, d'union et d'itération.

Exemple : les deux expressions régulières vues plus haut s'écrivent plus facilement avec les opérateurs étendus de `lex` en :

```
[[:alpha:]][[:alnum:]]*
```

pour les identificateurs (permettant les caractères accentués ; on peut les interdire en donnant des intervalles de caractères explicites), et :

```
[0-9]*\.?[0-9]+
```

pour les nombres littéraux (le \ sert à « protéger » le caractère `"."` qui est normalement un caractères spécial).

Les langages rationnels permettent un bon compromis entre la puissance d'expressivité et l'efficacité de la mise en œuvre. Si l'on connaît les limites de l'expressivité des langages rationnels (on ne peut pas décrire de langages parenthésés, par exemple), les lexèmes des langages de programmation n'ont jamais une structure trop complexe pour être exprimée de la sorte. Ensuite, il est désormais bien connu que les langages rationnels sont équivalents aux automates d'états finis. Chaque motif est reconnu par un automate d'états finis ; cette reconnaissance peut se faire en un temps proportionnel à la longueur de la chaîne en entrée dans le cas d'automates déterministes (voir plus loin).

1.1.2.3 Découpage et analyse avec `lex`

L'analyse avec `lex` fonctionne en associant une action à chaque motif. Ces actions sont des blocs de code en C qui sont exécutés dans le contexte particulier de l'analyse.

Le processus d'analyse. L'algorithme d'analyse du texte est un algorithme glouton qui parcourt le texte de gauche à droite sans retour arrière (même si l'utilisateur peut légèrement modifier ce comportement). À partir de la position courante, chaque motif est essayé dans l'ordre pour chercher à identifier le lexème courant. Les ambiguïtés sont résolues de deux manières : premièrement, si plusieurs motifs correspondent au texte courant, c'est le premier dans l'ordre du fichier qui est choisi. Ainsi, l'ordre de définition des lexèmes dans le fichier source de `lex` a son importance. Deuxièmement, si un même motif s'applique à plusieurs préfixes possibles de la chaîne en entrée, c'est le lexème le plus long qui est choisi.

On note également que la chaîne vide ne doit jamais être reconnue, afin que le processus puisse toujours terminer. Ces règles permettent de résoudre toutes les ambiguïtés qui pour-

raient survenir au cours de l'analyse, et l'auteur de l'analyseur lexical doit ordonner et définir les motifs de chaque lexème correctement pour obtenir le résultat escompté.

Les actions lexicales. Une fois le lexème courant déterminé avec certitude, le parcours du texte s'arrête jusqu'au prochain appel à `yyget()`, mais le traitement du morphème courant n'est pas terminé : l'action correspondant au motif est exécutée. L'action par défaut est d'afficher le texte reconnu.

Une action classique, quand le résultat de l'analyse lexicale est utilisée par un analyseur syntaxique, est de retourner une étiquette, plutôt que de renvoyer la chaîne rencontrée, et de placer cette chaîne dans une variable accessible par l'analyseur syntaxique. Ainsi, quand un littéral est identifié, une étiquette « littéral » est renvoyée, et la valeur du littéral en question (et non plus la chaîne lue) est placée dans une variable. La chaîne reconnue peut ainsi être modifiée à volonté ; par exemple, quand l'item courant est une chaîne de caractères délimitée par des apostrophes, on peut supprimer les apostrophes et ne conserver que le contenu de la chaîne.

Les actions lexicales permettent également de modifier le comportement classique de l'analyseur. Ainsi, l'action spéciale `REJECT` permet de refuser le lexème reconnu, pour que la règle suivante s'applique. De même, on peut agir sur le texte en entrée en remettant une partie des caractères du lexème vus dans le texte à analyser, ou au contraire, en en consommant plus que ce à quoi correspond le lexème actuel.

Liens avec l'analyse morphologique. Comme on l'a vu dans la section 1.1.1.1, des applications morphologiques relativement simples sont programmées directement en C ; `lex` permet alors d'automatiser certaines tâches rébarbatives pour l'écriture d'un itémiseur ou d'un analyseur morphologique fondé sur les états finis. La description des motifs guide la reconnaissance des formes de surface et constitue, si l'on se contente des actions par défaut, une itémisation du texte.

La lemmatisation elle-même a lieu à l'intérieur des actions lexicales : on dispose de la chaîne de surface, à partir de laquelle il s'agit d'extraire la forme lexicale, ainsi que les informations morphologiques (*e.g.* catégorie syntaxique, genre, nombre, temps, etc.). On se retrouve alors exactement dans le même cas de figure que dans la section 1.1.1.1.

Enfin, l'utilisation d'actions permet de reconnaître des formes qui ne sont pas représentables par une expression régulière. Par exemple, le pluriel avec reduplication peut être traité ainsi.

Exemple : l'extrait de fichier `lex` ci-dessous traite le pluriel avec reduplication. La colonne de gauche contient les différents motifs, la colonne de droite les actions associées.

```
"nana"          ECHO; printf("\n");
[[:alpha:]]+    { char *w;
                  if (test_ww(yytext, &w)) {
                    printf("%s = %s+%s\n", yytext, w, w);
                  } else {
                    REJECT;
                  }
                }
[[:alpha:]]+    ECHO; printf("\n");
```

Le motif `[[:alpha:]]+` désigne toute séquence de un ou plusieurs caractères alphabétiques, soit un « mot » au sens le plus simple du terme. Le prédicat `test_ww`, étant donné une chaîne de caractère, est vrai si cette chaîne est de la forme *ww* et affecte la chaîne *w* à son deuxième paramètre par effet de bord.
Si le mot est bien de la forme *ww*, le résultat est conservé ; sinon, il ne s'agit pas d'un pluriel, aussi on rejette le lexème et l'on essaie la règle suivante qui reconnaît un mot quelconque. Cependant, s'il existe des mots de la forme *ww* mais qui ne sont pas des pluriels, ceux-ci doivent être traités *avant* la règle du pluriel. Dans tous les cas, les autres mots sont simplement recopiés par l'action `ECHO`.

Si la programmation d'actions en C permet une très grande flexibilité, celle-ci a un coût, et c'est l'absence quasi-totale d'abstraction linguistique, d'où sa position (et celle d'outils de génération de code comme `lex`) dans la figure 1.1. `lex` est tout de même cité comme plateforme pour l'itémisation dans des systèmes comme Satz [Palmer et Hearst, 1997], l'étiqueteur syntaxique PARTS [Church, 1988], ou pour des expérimentations de segmentation de phrases [Grefenstette et Tapanainen, 1994].

1.1.3 Outils de traitement textuel fondés sur les expressions régulières

Sans être spécialisés pour le traitement linguistique, d'autres outils développés spécifiquement pour le traitement du texte existent et peuvent rendre de nombreux services à un linguiste travaillant sur des corpus de texte. Quelques-uns des outils les plus courants sous Unix sont présentés, ainsi que le langage Perl, qui inclue un moteur d'expressions régulières très puissant qui le rend apte à des traitements présyntaxiques et même à la morphologie, au même titre que `lex`.

Perl est désormais un langage de programmation bien établi et utilisé pour une multitude de tâches, ce qui en fait un langage de programmation généraliste au même titre que ceux vus plus haut dans ce chapitre. Un aspect remarquable de Perl est que, contrairement à C, C++ ou Java, ou tout autre langage pour lesquels il existe des logiciels de génération de code tels que `lex`, les expressions régulières font partie intégrante du langage, et en constituent même un élément essentiel. Le programmeur a donc sous la main un moteur de reconnaissance de chaînes relativement puissant ; on verra d'ailleurs dans le chapitre 2 que l'on peut ainsi mettre en œuvre un itémiseur en seulement quelques lignes de code.

Il est intéressant de noter que d'autres langages, comme Python, Ruby, ECMAScript, etc. adoptent un mécanisme très similaire, inspiré du succès de Perl, et que des bibliothèques d'expressions régulières sont également disponibles pour C, C++, Java, et la liste n'est pas exhaustive.

1.1.3.1 Expressions régulières étendues

Cependant, les expressions régulières au sens où on l'entend en Perl vont beaucoup plus loin que la seule description de langages rationnels, aussi l'utilisation de ce terme peut sembler un peu exagérée. Perl emploie une syntaxe proche de celle de `lex` mais l'enrichit considérablement et procure plus de flexibilité, au prix parfois d'une syntaxe quelque peu absconse.

Si la plupart des extensions sont surtout du sucre syntaxique, une extension en particulier est particulièrement puissante, c'est celle de la « capture ». Il est possible de délimiter une partie d'un motif à l'aide de parenthèses ; on peut ensuite faire référence non plus à tout le texte reconnu par ce motif, mais seulement à la partie capturée.

Exemple : la syntaxe de la reconnaissance d'un motif dans une chaîne (ici contenue dans la variable `$chaine`) est de la forme

```
$chaine =~ m/"([^\"]*)"/;
```

où le motif est délimité par `m//`. Ce motif reconnaît une chaîne de caractères délimitée par des guillemets (*double quotes*). Une chaîne valide est constituée d'un guillemet, suivi d'une séquence de caractères autres que le guillemet, et suivi d'un dernier guillemet. Cependant, les guillemets ne sont là que pour délimiter le texte de la chaîne de son contexte et c'est seulement le contenu de la chaîne reconnu qui est intéressant. C'est pourquoi on le délimite par des parenthèses, pour pouvoir y accéder ensuite. Ici, la sous-chaîne capturée est contenue dans la variable spéciale nommée `$1` :

```
if ($chaine =~ m/"([^\"]*)"/) {
  print "contenu = $1\n";
}
```

Il est en réalité possible de faire référence à une sous-chaîne ainsi capturée à l'intérieur même de la description du motif. Les séquences spéciales `\1`, `\2`, `\3`, etc. permettent de faire référence au texte précédemment reconnu, pendant même la reconnaissance du texte. On peut donc écrire des motifs comme :

```
([[:alpha:]]+)\1
```

pour désigner une forme de reduplication. La première partie de l'expression, entre parenthèses, reconnaît une suite quelconque de caractères alphabétiques (longue d'au moins un caractère), et la seconde partie, `\1`, doit reconnaître exactement la même suite de caractères : ce motif décrit donc des mots ayant la forme *ww*.

1.1.3.2 Substitutions

Les substitutions sont un mécanisme hérité de `sed`, et qui bénéficie de la syntaxe étendue des expressions régulières de Perl. Là où une expression régulière se contente de décrire un motif à reconnaître, une substitution transforme ce motif. La syntaxe typique d'une substitution est :

`$chaine =~ s/`*motif*`/`*remplacement*`/;`

Si *motif* est identifié dans la chaîne contenue dans la variable `$chaine`, alors la partie de texte identifiée est remplacée par le membre droit de la substitution.

Exemple : si une variable contient la forme de surface d'un mot au pluriel, on peut chercher à le lemmatiser en appliquant à l'envers les règles de formation du pluriel. En anglais, les noms en *-y* ont un pluriel en *-ies* (*e.g.lorry*, camion, devient *lorries* au pluriel), ce qui donne :

```
$mot =~ s/ies$/y/;
```

1.1.3.3 Interprétation des expressions régulières

Contrairement à `lex`, les expressions régulières étendues de Perl ne sont pas compilées sous la forme d'un automate d'états finis. En effet, Kleene a prouvé dans son fameux théorème du milieu des années 1950 qu'un langage était rationnel si et seulement si il pouvait être reconnu par un automate d'états finis. Cette équivalence d'expressivité entre langages rationnels et automates d'états finis explique qu'un automate fini ne peut reconnaître un langage qui n'est pas rationnel. Or, les expression régulières étendues de Perl permettent d'exprimer des langages qui ne sont pas rationnels (comme le langage *ww*). Perl emploie donc un moteur de reconnaissance non-déterministe spécialisé accessible par les fonctions `m//` et `s///`.

L'intégration parfaite des expressions régulières étendues dans Perl permet également d'augmenter leur puissance, ou plutôt de les utiliser de manière plus souple. `m//` (*match*) et `s///` (*substitute*) sont des fonctions qui renvoient « vrai » si une correspondance est trouvée, et « faux » sinon. Elles s'appliquent toutes deux à une chaîne de caractère par l'opérateur `=~`. `s///` permet également, par effet de bord, de modifier la chaîne à laquelle elle s'applique.

En conjonction avec les structures de contrôle de Perl, on peut donc écrire simplement des « super expressions » qui sortent du cadre des langages rationnels.

> *Exemple :* les langages parenthésés sont des langages hors-contexte et donc ne peuvent être décrits par des expressions régulières. Par contre, par itération, on peut tester si une chaîne est bien parenthésée :
>
> ```
> while ($chaine =~ s/\([^()]*\)//) {}
> ```
>
> La substitution s'applique tant qu'un couple de parenthèses peut être identifié et consommé. Ainsi, après une substitution, la chaîne
>
> ```
> (a (b c) (d (e f)))
> ```
>
> devient
>
> ```
> (a (d (e f)))
> ```
>
> car `(b c)` a été reconnu par l'expression `\([^\)]*\)`. Quand l'itération s'arrête, la chaîne testée est soit vide (auquel cas elle était bien parenthésée), soit non vide, ce qui indique qu'elle n'était pas bien formée.

1.2 Des structures de données pour l'analyse présyntaxique

Une progression vers un plus grand confort d'expression linguistique consiste à employer des structures de données pour représenter les composantes linguistiques du traitement, en particulier les dictionnaires, mais également le texte en cours d'analyse ou analysé. Certaines de ces structures (DAG, treilles, machines d'états finis) sont relativement classiques et sont décrites ci-dessous.

1.2.1 Représentation des ambiguïtés dans un texte

Un problème majeur qui va revenir tout au long de la discussion est le traitement des ambiguïtés. Dès qu'un découpage ou qu'une analyse est entreprise, diverses possibilités se

présentent, parmi lesquelles il est parfois très difficile de choisir. Si l'on s'en tient aux informations disponibles au niveau présyntaxique, certains choix sont même impossibles. Il est donc nécessaire de travailler avec ces ambiguïtés, d'où la question de leur représentation.

1.2.1.1 De la nature des ambiguïtés

Deux types différents d'ambiguïtés se présentent au cours de l'analyse présyntaxique, et plus particulièrement en analyse morphologique. Premièrement, les ambiguïtés de découpage surviennent quand l'identification des limites entre segments n'est pas certaine. Deuxièmement, au cours de la lemmatisation, un même mot peut être analysé de plusieurs manières différentes, correspondant à un ou plusieurs lemmes, et à une ou plusieurs formes pour chaque lemme.

Dans l'analyse présyntaxique, il faut prendre en compte d'autres types d'ambiguïtés relatives à l'acquisition du texte :

- si le texte à analyser provient d'un programme de reconnaissance optique de caractères (OCR), certains caractères sont ambigus, du fait d'erreurs du processus d'OCR ;
- de même, si le texte à analyser provient d'un reconnaisseur de parole, certains phonèmes sont ambigus et donc plusieurs transcriptions sont possibles ;
- En français, un texte peut être mal accentué, voire ne pas être accentué du tout. Chaque caractère devra être considéré selon ses « potentiels » : un *e* pourrait en réalité être un *é*, un *è*, un *ê*, ou bel et bien un *e*.

Et ainsi de suite. Il est donc intéressant de noter que le texte peut être ambigu dès son acquisition, et que les mécanismes de représentation de l'ambiguïté sont valables non seulement pour la sortie et la représentation interne, mais également pour l'entrée.

1.2.1.2 Graphes orientés sans circuits

La stratégie de représentation la plus simple est de lister simplement toutes les analyses possibles, une par une. Lorsqu'il s'agit de présenter un résultat à la sortie d'un processus, des astuces de présentation peuvent être trouvées pour rendre la sortie gérable par l'utilisateur. Mais cette approche peut se révéler très coûteuse dans le cas de longs textes, ou le nombre d'interprétations possibles croit exponentiellement avec le nombre d'ambiguïtés.

La factorisation des ambiguïtés est un moyen de représenter efficacement toutes les interprétations possibles d'un texte. Une telle forme de factorisation est le graphe orienté sans circuits, ou DAG (pour *directed acyclic graph*) [Zaysser, 1996], dont la figure 1.2 montre un exemple.

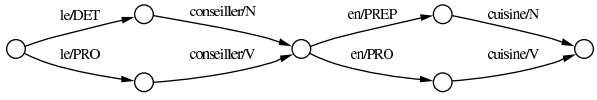

Fig. 1.2 – Représentation des ambiguïtés avec un DAG

Chaque analyse possible correspond alors à un chemin dans le DAG, depuis le nœud

initial jusqu'à atteindre un nœud final (qui n'a aucune transition sortante). Le graphe est évidemment orienté, orientation qui correspond au sens de lecture du texte. Enfin, il n'admet aucun circuit car le texte est obligatoirement fini (l'introduction d'un circuit créerait des chemins, et donc des analyses, de longueur infinie).

1.2.1.3 Treilles

Une autre structure similaire à celle du DAG est populaire en TALN, par exemple en analyse de la parole. Une treille peut aussi être vue comme un graphe orienté sans circuits, à la différence que ce sont les nœuds qui sont décorés, et plus les transitions, à l'opposé des DAG. La figure 1.3 montre un exemple tiré de [Seligman *et al.*, 1998]. Deux nœuds sont particulier et représentent les extrémités du texte.

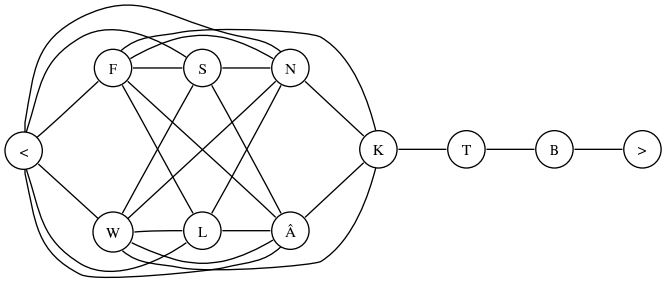

FIG. 1.3 – Représentation des ambiguïtés avec une treille

Les treilles sont souvent utilisées dans les systèmes stochastiques prenant les probabilités de transitions entre deux nœuds. Chaque transition est ainsi pondérée, et un chemin entre les deux extrémités de la treille a alors un poids. Il sera question plus loin de méthodes stochastiques, et donc de représentation de poids d'items et de transition entre deux items, comme dans les treilles de mots utilisées en reconnaissance de parole [Amtrup *et al.*, 1996].

1.2.2 Machines d'états finis

Les automates d'états finis sont des constructions mathématiques désormais tout à fait classiques en informatique, et leur application va bien au delà du TALN. Dans ce domaine, ils se montrent particulièrement utiles du fait de leur lien étroit avec les langages rationnels et les expressions régulières. Sous le terme de « machines d'états finis », on inclue également les transducteurs d'états finis, qui sont aux relations rationnelles ce que les automates d'états finis sont aux langages rationnels.

1.2.2.1 Langages rationnels et automates d'états finis

Compilation d'automates d'états finis. Les langages rationnels sont d'ores et déjà une composante familière du traitement présyntaxique, ainsi que les automates d'états finis qui sont des structures informatiques exactement équivalentes : pour chaque langage rationnel, il existe au moins un automate d'états finis qui reconnaisse ce langage [Autebert, 1994].

Différentes méthodes de compilation d'un automate d'états finis à partir d'une expression régulière existent. La plus simple est la construction de Thompson qui établit une correspondance directe entre la définition des langages rationnels et la construction d'un automate d'états finis ; à chaque opérateur rationnel correspond une opération sur les automates. La figure 1.4 montre un automate d'état fini ainsi construit à partir d'une expression régulière, puis minimisé.

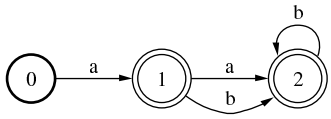

FIG. 1.4 – Automate d'états finis reconnaissant le langage `a(b*|a)b*`

Formellement, un automate d'états finis A est défini par un quintuplet (Σ, Q, I, F, E) avec :

- Σ un alphabet fini (le même que celui du langage reconnu) ;
- Q un ensemble fini d'états ;
- $I \subseteq Q$ l'ensemble des états initiaux ;
- $F \subseteq Q$ l'ensemble des états finals ;
- $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ l'ensemble des transitions.

Dans la représentation graphique classique, un automate est figuré par un graphe orienté dont les nœuds sont les états et les arcs sont les transitions. Les états sont traditionnellement numérotés ; les états initiaux sont distingués des autres états par un trait gras, et les états finals par un double trait (un état peut évidemment être à la fois initial et final).

Reconnaissance d'un mot par un automate d'états finis. Dans un automate d'états finis, on définit un sous-chemin entre deux états q et q' comme une suite de transitions successives menant de q à q' . Un chemin est un sous-chemin dont l'état origine a est un état initial et l'état destination q' un état final. Un chemin correspond à un mot du langage reconnu par l'automate : ce mot est constitué de la concaténation des symboles portés par chaque transition du chemin.

La reconnaissance d'un mot du langage est un prédicat qui a pour argument une chaîne de Σ^* . Le prédicat est vérifié si et seulement si il existe un chemin dans l'automate pour ce mot. Ainsi, dans la figure 1.4, il y a bien un chemin pour le mot `aabb`, par contre il n'y en a pas pour le mot `abab` qui ne fait effectivement pas partie du langage reconnu.

La recherche d'un chemin amène à se poser la question de l'ambiguïté et du déterminisme de l'automate. Un automate d'états finis est ambigu si le même mot peut être reconnu par plusieurs chemins différents. Un automate d'états finis est déterministe si et seulement si pour chaque état de l'automate et pour chaque symbole de l'alphabet, il n'est possible d'une suivre qu'une seule transition ou aucune (il faut ici prendre en compte les epsilon-transitions). Un automate non-déterministe n'est cependant pas nécessairement ambigu.

Dans un automate non-déterministe, la reconnaissance d'un mot se fait par un parcours de graphe traditionnel. À chaque état, il est possible que plusieurs transitions portant le même symbole atteignent des états différents, aussi toutes les possibilités doivent être essayées pour

trouver un chemin. Quand il est impossible de suivre une transition à partir d'un état donné, il faut effectuer un retour arrière. La recherche se termine quand un état final est atteint et que tous les caractères de la chaîne d'entrée ont été reconnus, auquel cas le mot est bien reconnu ; mais elle s'arrête également quand toutes les transitions ont été essayées sans succès (et la reconnaissance est un échec).

Dans un automate déterministe, il n'y a aucune ambiguïté : à chaque état, il suffit de suivre la transition qui porte le symbole à reconnaître. Dès qu'il devient impossible de suivre une transition, il y a échec de la reconnaissance. La déterminisation d'un automate est très pratique puisqu'elle permet ainsi de décider de l'appartenance d'un mot à un langage rationnel en un temps linéairement proportionnel à la longueur du mot. Mais cette déterminisation a un prix : si tout automate d'états finis peut être déterminisé, la taille de l'automate peut exploser exponentiellement. Dans le pire des cas, un automate non-déterministe ayant n états peut produire un automate déterministe minimal de 2^{n-1} états, comme l'automate reconnaissant l'expression `(a|b)*a(a|b)...` de la figure 1.5. Dans la pratique, les automates utilisés en TALN sont très souvent déterminisables sans explosion notable.

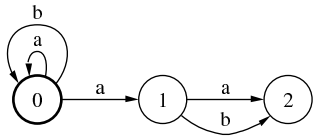

FIG. 1.5 – Automate d'états finis reconnaissant le langage `(a|b)*a(a|b)...`

Minimisation du nombre d'états d'un automate. Pour un langage rationnel donné, on peut en réalité construire une infinité d'automates reconnaisseurs, déterministes ou non. En pratique, les langages rationnels utiles peuvent être reconnus par des automates ayant un grand nombre d'états et de transitions. Par exemple, un dictionnaire contenant des dizaines de milliers de formes pourra être compilé sous la forme d'un automate contenant des milliers d'états et de transitions.

Il est toujours possible d'obtenir un automate minimal à partir d'un automate d'états finis déterministe [Watson, 1995]. Par automate minimal, on entend un automate reconnaissant strictement le même langage mais dont le nombre d'états est le minimum. Cette opération de minimisation, qui peut être coûteuse en mémoire et en temps de calcul, garantit cependant que les données linguistiques représentées sous la forme d'automates d'états finis peuvent l'être de façon efficace.

1.2.2.2 Relations rationnelles et transducteurs d'états finis

Définitions. Une relation est un ensemble de couples de mots qui appartiennent à deux langages, comme par exemple {(`ab`, `xy`), (`c`, `c`), (`d`, ϵ)}. Une relation rationnelle est alors une relation entre deux langages qui peut être reconnue par un transducteur d'états finis (figure 1.6).

Dans le cadre d'une relation rationnelle, on parle souvent de deux niveaux de langage : par exemple, dans la morphologie à deux niveaux, c'est la mise en correspondance de formes de

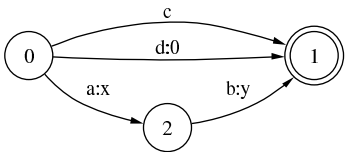

FIG. 1.6 – Un transducteur d'états finis

surface et de formes lexicales qui est intéressante [Karttunen *et al.*, 1992]. On peut voir aussi un langage comme le langage d'entrée, pour chaque mot duquel on veut connaître les mots du langage de sortie correspondant. En général, on parlera de langage inférieur et langage supérieur.

Formellement, un transducteur d'états finis T est un sextuplet $(\Sigma, \Omega, Q, I, F, E)$ avec :

- Σ l'alphabet du langage supérieur, ou alphabet d'entrée ;
- Ω l'alphabet du langage inférieur, ou alphabet de sortie ;
- Q l'ensemble fini des états du transducteur ;
- $I \subseteq Q$ l'ensemble des états initiaux du transducteur ;
- $F \subseteq Q$ l'ensemble des états finals du transducteur ;
- $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times Q$ est l'ensemble des transitions du transducteur.

Un chemin dans un transducteur est défini de la même façon que dans un automate. On peut cependant voir un chemin soit comme la reconnaissance d'une correspondance entre deux chaînes, soit comme une transduction qui transforme une chaîne d'un langage en une ou plusieurs chaînes de l'autre langage de la relation.

Déterminisation et minimisation. Un transducteur peut être déterministe ou non. Si considère l'automate sous-jacent, dont les symboles sont en réalité des couples de symboles, il peut être déterminisé exactement comme un automate d'états finis. Par contre, même déterministe, un transducteur n'est pas forcément fonctionnel. Un transducteur fonctionnel reconnaît une relation qui n'est pas ambiguë, c'est-à-dire que pour toute chaîne du langage supérieur ne peut être mise en correspondance qu'avec une seule chaîne du langage inférieur. Dans le cas contraire, le transducteur n'est pas fonctionnel. Et contrairement à la déterminisation, il n'est pas toujours possible de rendre un transducteur fonctionnel, car cette propriété dépend de l'ambiguïté de la relation reconnue.

De plus, un transducteur peut être globalement non ambigu tout en admettant toujours une forme d'ambiguïté locale si le l'automate obtenu par projection supérieure (l'automate reconnaissant le langage supérieur) est lui-même non-déterministe. Un transducteur est dit séquentiel s'il n'admet aucune ambiguïté locale, donc si l'automate supérieur est déterministe. L'avantage d'un transducteur séquentiel est, comme pour les automates déterministes, que l'application du transducteur à une chaîne se fait en un temps linéaire (proportionnel à la longueur de la chaîne), car il n'y a pas de retour arrière.

Composition de transducteurs. Plusieurs opérations nouvelles (en plus de l'union, la concaténation et la fermeture) sont possibles sur les transducteurs, comme la projection évoquée plus haut (la projection supérieure ayant pour résultat l'automate reconnaissant le langage supérieur, et la projection inférieure l'automate inférieur), mais surtout la composition. Soit deux transducteurs d'états finis T_1 et T_2 définis respectivement sur les langages

(Σ, Φ) et (Φ, Ω) : il existe toujours un transducteur d'état fini T défini sur (Σ, Ω) dont l'application est équivalente à l'application successive de T_1 puis T_2 . Cette opération de composition est très utilisée en TALN, car elle permet de composer plusieurs étapes successives d'un traitement en un seul et même transducteur qui s'applique cette fois-ci directement à l'entrée (on aura l'occasion d'illustrer cette opération à plus d'une reprise).

1.2.2.3 Réalisations

De nombreuses bibliothèques de fonctions de calcul à états finis existent, permettant de créer et de manipuler automates et transducteurs. Les deux plus remarquables sont XFST (la suite d'outils de Xerox, décrite plus en détail section 1.3.1), et INTEX [Silberztein, 1993]. Dans les deux cas, ils s'agit de systèmes de traitement de la langue complets fondés sur les automates et transducteurs d'états finis.

D'autres réalisations existent, qui ne sont pas forcément destinées uniquement à l'analyse linguistique. Beaucoup sont disponibles gratuitement sur le Web, voici quelques liens qui montrent la diversité des outils actuellement disponibles :

- les utilitaires FSA[3] de Gertjan van Noord, réalisés en Prolog ;
- Grail+[4], du *Department of Computer Science, University of Western Ontario* au Canada ;
- FIRE Lite, de Bruce Watson ;
- *The Automaton Standard Template Library*[5] de Vincent Le Maoult (Université de Marne-la-Vallée), une bibliothèque pour la manipulation d'automates en C++ ;
- etc.

1.2.3 Machines d'états finis pondérées

L'utilisation de statistiques est un moyen de traiter les ambiguïtés d'analyse en l'absence d'informations linguistiques suffisantes. En étudiant des corpus de millions de mots, on calcule des probabilités sur des unigrammes, des bigrammes et des trigrammes pour avoir un modèle plus ou moins réaliste de la langue, permettant de décider si une forme est probablement correcte ou probablement fausse. Or, les transducteurs d'états finis sont au contraire des machines binaires : une relation est reconnue où elle ne l'est pas. Étant donné une forme de surface, toutes les formes lexicales correspondantes seront trouvées, sans ordre particulier.

Les machines d'états finis pondérées sont donc une évolution des machines d'états finis « classiques ». Pour simplifier, un automate d'état finis pondéré ne se contente pas de déterminer si un mot appartient ou non à un langage (réponse binaire, oui ou non), mais associe également un poids (généralement en TALN une probabilité, ou une mesure de fréquence) à toute chaîne reconnue.

[3] `http://odur.let.rug.nl/~vannoord/papers/fsa/fsa.html`
[4] `http://www.csd.uwo.ca/research/grail/`
[5] `http://www-igm.univ-mlv.fr/~dr/ASTL/main.html`

1.2.3.1 Séries formelles rationnelles et automates d'états finis pondérés

Tout comme les automates d'états finis classiques correspondent aux langages rationnels, les automates d'états finis pondérés correspondent aux séries formelles rationnelles. Une série formelle est une fonction définie sur un langage vers un demi-anneau fermé ; informellement, on peut dire qu'elle associe à chaque mot du langage un poids appartenant au demi-anneau.

On reprend les définitions de [Berstel et Reutenauer, 1984] : un monoïde est un ensemble muni d'une loi de composition interne associative avec un élément neutre pour cette loi. Un demi-anneau est un ensemble $\mathbb{K}$ muni de deux lois de composition interne :

1. une loi additive notée $\oplus$ et dont l'élément neutre est noté $\overline{0}$ et
2. une loi multiplicative notée $\otimes$ et dont l'élément neutre est noté $\overline{1}$.

Ces lois ont les propriétés suivantes :

1. $(K, \oplus)$ est un monoïde commutatif ;
2. $(K, \otimes)$ est un monoïde ;
3. le produit $(\otimes\ )$ est distributif par rapport à la somme $(\oplus\ )$;
4. $\forall a \in K$, on a $\overline{0} \otimes a = a \otimes \overline{0} = \overline{0}$.

Trois exemples de demi-anneaux sont l'algèbre de Boole $(\{0,1\}, \vee, \wedge, 0, 1)$, les langages rationnels $(\Sigma^*, \cup, ., \emptyset, \{\epsilon\})$ et le demi-anneau tropical $(\mathbb{R} \cup \{\infty\}, min, +, \infty, 0)$ qui est fréquemment employé dans les applications de TALN.

Dans [Schützenberger, 1961], on trouve une caractérisation des séries formelles rationnelles similaire à celle que Kleene donne des langages rationnels :

> Une série formelle $S : \Sigma^* \rightarrow \mathbb{K}$ est rationnelle si et seulement si elle est reconnue par un automate d'états finis pondérés sur le demi-anneau fermé $\mathbb{K}$, l'ensemble des poids.

On trouve dans la littérature le terme de transducteur au sujet des automates d'états finis pondérés : en effet, ceux-ci effectuent une transduction entre les mots d'un langage rationnel et leur poids dans l'ensemble des poids. Cette transduction est impossible avec un transducteur classique, car l'ensemble de poids est possiblement infini, et un langage rationnel est défini sur un alphabet fini.

Concrètement, un automate pondéré diffère d'un automate classique par l'ajout d'un poids aux états initiaux, finals, et aux transitions. La fonction de pondération de l'automate associe un poids à chaque mot du langage, c'est-à-dire à chaque chemin dans l'automate (figure 1.7). Le poids d'un chemin donné est le produit du poids de l'état initial, de l'état final, et de chaque transition empruntée. Si dans l'automate plusieurs chemins correspondent à un même mot, le poids de ce mot est alors la somme des poids de tous les chemins possibles. Les transducteurs d'états finis pondérés sont définis de la même manière, associant à chaque relation un poids différent.

Il reste à noter une différence importante entre les automates d'états finis et leurs homologues pondérés : si les premiers sont toujours déterminisables, ce n'est pas le cas des seconds. [Mohri, 1997] revient en détail sur cette question et cherche à caractériser les classes d'automates déterminisables ou non. Cela a une influence sur la minimisation du nombre d'états car un automate doit être déterminisé avant de pouvoir être minimisé.

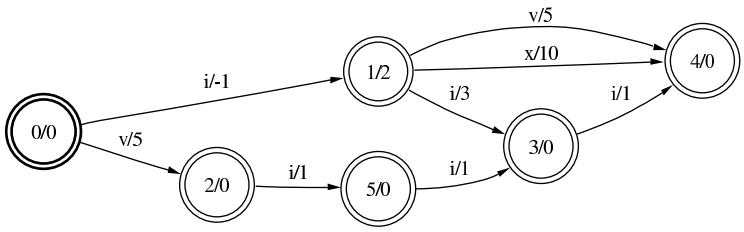

FIG. 1.7 – Automate d'états finis pondérés reconnaissant les chiffres romains de 0 à 9

1.2.3.2 Ensembles de poids

Les automates d'états finis pondérés ont des applications au-delà du domaine de la linguistique informatique, mais dans ce domaine particulier, l'ensemble de poids le plus fréquemment utilisé est le demi-anneau tropical. Dans l'exemple de la figure 1.7, c'est ce demi-anneau qui est employé : ainsi, le poids du mot `viii` est 0+5+1+1+1+0 (le poids de l'état initial, 0, le poids des quatre transitions empruntées, et le poids de l'état final, 4) soit 8. Pour `iv`, c'est 0-1+5+0, soit 4.

1.2.3.3 Réalisations

AT&T propose des outils de compilation et de manipulation d'automates et de transducteurs d'états finis pondérés. Cette suite logicielle, appelée FSM[6], ne propose pas d'interface ou d'environnement particulier comme le font XFST, XeLDA ou INTEX, mais une collection de programmes suivant la philosophie de conception d'Unix. Ainsi, sous Unix, ces programmes sont utilisables principalement en utilisant des scripts *shell* ; ou en programmant en C/C++ en appelant une bibliothèque de fonctions équivalente à ces programmes. Une liste non exhaustive des programmes disponibles révèle :

- Des programmes de compilation, comme `fsmcompile`, qui compile un automate ou un transducteur à partir d'une description textuelle (liste des états et des transitions), et de manipulations diverses de transducteurs : `fsmprint` pour afficher un transducteur, `fsmdraw` pour le dessiner, `fsminfo` pour obtenir une description ;
- Des programmes pour le calcul rationnel, chacun correspondant à un opérateur : `fsmclosure` (fermeture transitive), `fsmcompose` (composition), `fsmconcat` (concaténation), `fsmdifference` (différence), `fsmintersect` (intersection), `fsminvert` (inversion des niveaux d'un transducteurs), `fsmproject` (projection), `fsmreverse` (renversement), `fsmunion` (union).
- Des programmes modifiant ou testant les propriétés d'un automate : `fsmdeterminize` (déterminisation), `fsmminimize` (minimisation), `fsmrmepsilon` (suppression des epsilon-transitions), `fsmprune` (suppression des transitions et des états ne faisant partie d'aucun chemin, donc inutiles), etc.
- `fsmbestpath` et autres fonctions de consultation.

On retrouve donc les fonctions de calcul habituelles, même si certaines ont un effet légèrement différent : on reviendra par exemple sur le cas de la composition, qui est com-

[6] `http://www.research.att.com/sw/tools/fsm/`

pliquée par l'ajout de poids. (c'est également le cas pour le produit cartésien, une opération qui n'existe pas dans FSM ; il en sera question dans le chapitre 6). On trouve également des fonctions liées à l'apparition des poids qui n'existaient pas pour les automates et transducteurs non pondérés : trouver le chemin de meilleur coût dans un automate, trier les chemins par coût, etc.

D'autres réalisations existent, en particulier [Adant, 2000] qui est une extension de la bibliothèque en C++ de l'Université de Marne-la-Vallée.

1.3 Des formalismes pour l'analyse morphologique

Enfin, la progression vers la félicité linguistique totale se poursuit avec des formalismes linguistiques puissants, spécialisés ou non pour l'analyse morphologique. Encore une fois, les états finis sont prépondérants, que ce dans le formalisme de morphologie à deux niveaux de Xerox, où des transducteurs d'états finis sont explicitement construits, ou dans le système ATEF où l'automate est au contraire implicite. Enfin, on termine avec un formalisme nettement plus puissant - puisqu'il va bien au-delà des langages rationnels reconnaissables par une machine d'états finis, et même au delà du hors contexte - avec les systèmes-Q, dont le niveau d'abstraction linguistique est bien sûr le plus élevé parmi tous les systèmes présentés dans ce chapitre.

1.3.1 La morphologie à deux niveaux

La morphologie à deux niveaux tire son origine de la remise au goût du jour, au début des années 1980, de [Johnson, 1972]. L'idée principale de Johnson est que les règles phonologiques, malgré leur apparente puissance d'expressivité, peuvent être modélisées par des relations rationnelles en posant une simple contrainte sur leur contexte d'application, contrainte qui permet de compiler des transducteurs d'états finis représentant des jeux de règles phonologiques [Karttunen, 1991].

D'après [Kaplan et Kay, 1994] tout système de règles phonologiques ordonnées en cascade peut être représenté par un unique transducteur, car les relations rationnelles sont fermées sous l'opération de composition. Il est donc possible de définir un nombre arbitraire de niveaux intermédiaires pour décomposer le traitement, et de composer ensemble dans l'ordre d'application des règles tous les transducteurs pour n'en faire finalement qu'un, effectuant la correspondance directe entre niveau lexical et niveau syntaxique. Toute représentation intermédiaire a alors disparu.

C'est cette technique qui est mise en œuvre pour la première fois par [Koskenniemi, 1983] qui propose une description « à deux niveaux » de la morphologie du finnois. Le formalisme de Koskenniemi décompose cette description en deux parties :

1. le lexique, décrivant les morphèmes et les propriétés morphosyntaxiques de la langue, et
2. les règles à deux niveaux.

Le logiciel PC-KIMMO est une première mise en œuvre de ce formalisme à deux niveaux. Ses nombreuses restrictions (pas de compilateur pour les règles, et approche purement à deux

niveaux interdisant toute représentation intermédiaire) lui ont valu des critiques [Gazdar, 1985], mais ont surtout motivé le développement et le raffinement de cette approche. Des systèmes plus perfectionnés comme [Black *et al.*, 1987], mais surtout le système XFST de Xerox [Beesley et Karttunen, 2002] ont vu le jour.

1.3.1.1 Analyse morphologique à deux niveaux avec XFST

Sous la dénomination d'ensemble XFST, se trouvent la bibliothèque de calcul à état finis de Xerox, c'est-à-dire une bibliothèque de primitives de fonctions pour la création et la manipulation de machines d'états finis (automates, transducteurs, bimachines), et les divers outils et interfaces l'utilisant. Développée au PARC puis au sein de l'équipe MLTT à Grenoble, c'est une bibliothèque qui est particulièrement destinée aux applications linguistiques. La principale application a longtemps été la morphologie à deux niveaux, aussi est-il intéressant de s'y attarder en présentant deux outils qui s'y rattachent ; l'interface plus générale de manipulation de transducteurs est présentée ensuite.

Dans ce modèle, lexiques et règles sont compilés sous formes de transducteurs d'états finis, pour former par composition un transducteur lexical permettant à la fois l'analyse et la génération. En effet, une fois le transducteur compilé, on peut l'appliquer à une chaîne dans un sens ou dans l'autre. L'application du transducteur à une chaîne de surface réalise l'analyse de la chaîne pour obtenir sa forme lexicale ; et l'application dans l'autre sens à une chaîne de surface permet de générer la forme de surface correspondante.

> *Exemple*s : l'analyse de la forme de surface `heureuse` par un transducteur lexical donne la forme lexicale `heureux+Adj+Fem` par la correspondance (`0` représente en réalité ϵ) :
>
> ```
> h e u r e u x +Adj +Fem
> | | | | | | |  |    |
> h e u r e u s  0    e
> ```
>
> La génération de la forme de surface `jumelle` étant donné la forme lexicale `jumeau+N+Fem` est effectuée par la correspondance :
>
> ```
> j u m e a u +N +Fem
> | | | | | | |   |
> j u m e l l 0   e
> ```
>
> on a simplement inversé le sens d'application.

1.3.1.2 Compilation du transducteur lexical

Les deux outils principaux pour la compilation du transducteur lexical sont un compilateur de lexique et un compilateur de règles. Le lexique est composé avec les règles pour créer le transducteur. Ces deux compilateurs sont des formalismes destinés à des tâches très précises ; si au bout du compte ils produisent des transducteurs d'états finis classiques, leur spécialisation rend les descriptions beaucoup plus pratiques.

Compilation des lexiques. Le premier de ces outils est `lexc` (pour *lexicon compiler*), qui compile des transducteurs lexicaux définis par formalisme spécialisé, proche de celui de Koskenniemi, destiné aux lexicographes. `lexc` dispose également d'un mode de composition spécial pour composer un transducteur lexical avec des règles à deux niveaux compilées par `twolc`.

Un intérêt de `lexc` est de permettre la création de lexiques à deux niveaux. Dans l'écriture d'un lemmatiseur, par exemple, on peut directement spécifier dans le fichier lexique la plupart des formes possibles ; les règles à deux niveaux servent alors pour gérer les altérations graphiques ou phonologiques.

Un lexique est en réalité un ensemble de sous-lexiques. Les sous-lexiques aident à organiser un lexique de manière plus ou moins structurée en créant des classes de morphèmes. Un lexique minimal contient un unique sous-lexique (toujours nommé `Root`), un lexique plus complexe contiendra des dizaines voir des centaines de sous-lexiques. Un sous-lexique est à son tour composé d'une ou de plusieurs entrées lexicales. Une entrée lexicale est composée :

1. d'une forme lexicale ou a deux niveaux. Une forme lexicale peut être vide. Par exemple :
   - `hund` (racine de « chien » en esperanto) est une forme lexicale simple ;
   - `+Fem:in` est une forme à deux niveaux ; le premier est le niveau lexical, le second est le niveau de surface. Ils sont séparés par `:`.
2. d'une classe de continuation, soit une référence à un sous-lexique. Il y a trois cas possibles :
   (a) la classe de continuation est un sous-lexique différent ;
   (b) la classe de continuation est le même sous-lexique, ce qui permet de créer des boucles ;
   (c) la classe de continuation est la classe spéciale `#` marquant la fin d'une entrée.

Grosso modo, un lexique est composé de deux sortes d'entrées, les racines et les affixes. Une fois le lexique structuré, il ne reste plus qu'au lexicographe à ajouter les entrées dans les bonnes catégories et à considérer les exceptions séparément.

Exemple :

```
LEXICON Root
        Nouns ;

LEXICON Nouns
bird    Nmf ;
hund    Nmf ;
kat     Nmf ;
elefant Nmf ;

LEXICON Nmf
+Masc:0 N ;
+Fem:in N ;

LEXICON N
+N:o    # ;
```

Ce lexique tiré de [Beesley et Karttunen, 2002] reconnaît les formes masculines et féminines de quatre noms d'animaux en espéranto. Par exemple, `hundino` (« chienne ») sera analysé `hund+Fem+N`. De même, `elefant+Masc+N` génère la forme de surface `elefanto` (« éléphant »).

Compilation des règles à deux niveaux. Les règles à deux niveaux sont directement héritées de Koskenniemi, et ressemble aux règles phonologiques. Il existe quatre types de règles, qui sont toutes équivalentes à une relation rationnelle. Dans la description ci-dessous,

`a` et `b` représentent un symbole quelconque de l'alphabet, et c et d des expressions régulières quelconques (il peut s'agir de langages rationnels comme de relations).

1. La restriction de contexte :

   `a:b =>` c `_` d

   signifie que `a` apparaîtra comme `b` en surface uniquement dans le contexte c_d (*i.e.* si `a` est précédé d'une occurrence de c et suivi d'une occurrence de d). L'expression régulière correspondante est :

   `~[[~[?*` c`] a:b ?*] | [?* a:b ~[`d `?*]]]`

   On note que dans la syntaxe de XFST, `~` est la négation et `?` correspond à n'importe quel symbole (comme le `.` dans `lex`).

2. La contrainte de surface :

   `a:b <=` c `_` d

   signifie que `a` apparaîtra toujours comme `b` en surface dans le contexte c_d. L'expression régulière correspondante est :

   `~[?*` c `[a: - a:b]` d `?*]`

   `[a: - a:b]` signifie que `a` peut être réalisé par n'importe quel symbole en surface, sauf par `b`.

3. La conjonction des contraintes 1 et 2 :

   `a:b <=>` c `_` d

   signifie que `a` ne sera réalisé par `b` en surface que dans le contexte c_d.

4. Enfin, la règle d'interdiction :

   `a:b /<=` c `_` d

   signifie que `a` ne sera jamais réalisé par `b` dans le contexte c_d. L'expression régulière correspondante est :

   `~[?*`c `a:b` d`?*]`

Par construction, il apparaît que les règles à deux niveaux sont compilables sous la forme de transducteurs, et c'est justement ce que fait le compilateur `twolc` à partir d'un fichier de règles. Un avantage particulièrement intéressant des règles à deux niveaux est qu'elles permettent de spécifier le contexte d'application de la règle sur les deux niveaux en même temps.

Génération des transducteurs lexicaux. Le transducteur lexical final est compilé par `lexc` à partir de la description du lexique et des transducteurs de règles compilées par `twolc`. L'opération de composition de `lexc` est particulière car son résultat est que toutes les règles à deux niveaux s'appliquent parallèlement, et non pas en cascade (l'ordre des règles dans un jeu de règles n'est donc pas important). Par contre, il est possible de composer le transducteurs lexical successivement avec plusieurs jeux de règles différents, auquel cas l'ordre d'application des jeux de règles devient important.

1.3.1.3 Manipulation de transducteurs

Outre ces deux outils très spécialisés, qui facilitent des tâches courantes mais ont une expressivité plus limitée que celle des expressions régulières, XFST inclue l'interface `xfst` pour manipuler les automates et transducteurs d'états finis. En effet, le concept de classes de continuation de `lexc` est particulièrement adapté aux affixations, par contre des règles de formation plus complexes comme l'infixation sont difficiles, voire impossibles à exprimer. De même, les règles à deux niveaux ont des défauts ; le principal est sans doute qu'une règle ne peut concerner qu'un seul symbole à la fois. Si l'on veut écrire une règle pour les noms en *-al* qui ont leur pluriel en *-aux*, il faut en fait écrire plusieurs règles (transformer le l en u, insérer un x) qui en prenant soin de bien coordonner leurs conditions d'application. Elles ont par contre l'avantage de permettre l'expression du contexte d'application sur les deux niveaux simultanément ; on peut écrire une règle de la forme :

```
a:b => c:d _ e:f
```

Expressions régulières et règles de réécriture. Outre les outils de compilation de lexiques et de règles, XSFT dispose évidemment d'expressions régulières sur les automates et les transducteurs. On en a vu quelques exemples dans la présentation des règles à deux niveaux ; les opérations classiques sont définies (concaténation, union, répétition, intersection, complémentation, renversement, etc.) ainsi que les opérations portant sur les transducteurs (projection, composition) ou sur les automates mais créant des transducteurs (le produit cartésien).

Une extension puissante des expressions régulières est celle des règles de réécriture, similaires aux règles phonologiques, qui définissent des relations rationnelles [Karttunen, 1995]. La forme générale d'une règle de réécriture est

$$\phi \rightarrow \psi / \lambda \_ \rho$$

qui se lit ϕ se réécrit en ψ dans le contexte $\lambda \_ \rho$. Ces règles de réécriture sont plus générales que les règles à deux niveaux (ϕ et ψ sont des expressions régulières et non plus de simples symboles), mais les restrictions de contexte ne s'appliquent pas sur deux niveaux à la fois. Comme les règles à deux niveaux, il est possible de spécifier plusieurs règles de réécriture s'appliquant en parallèle (donc simultanément) ou en cascade (donc séquentiellement).

L'interface xfst. `xfst` est l'outil qui permet de coordonner les opérations sur les transducteurs. Des transducteurs créés par `lexc` peuvent être lus, de nouveaux transducteurs peuvent être compilés en utilisant des expressions régulières, et peuvent ensuite être sauvés pour une utilisation ultérieure.

`xfst` sert aussi à l'analyse ou la génération en permettant d'appliquer un transducteur à du texte pour observer le résultat. Mais en réalité, cette utilisation est plutôt réservée aux phases de développement, pour effectuer des tests précis plutôt que pour des analyses de texte en vraie grandeur. C'est cependant l'interface de consultation la plus souple et la plus puissante.

L'interface lookup. `lookup` est comme son nom l'indique dédié à l'analyse. *To lookup* signifie « consulter » (un dictionnaire), et cet outil permet de consulter un transducteur, avec pour entrée une chaîne de surface, et propose en sortie les lexèmes trouvés.

L'intérêt principal de `lookup` est de permettre d'utiliser plusieurs transducteurs pour la consultation. Ceux-ci peuvent être utilisés en cascade, la sortie de l'un servant d'entrée à l'autre. C'est un point intéressant car la composition de transducteurs est une opération qui peut être très coûteuse, et en temps de calcul et en mémoire, car la taille du transducteur résultat est dans le pire des cas le produit de la taille des deux transducteurs composés. De plus, les transducteurs produits par XFST sont systématiquement déterminisés et minimisés ; au cours de cette opération, leur taille peut exploser de manière exponentielle. Il ne faut remarquer que pour couvrir largement une langue donnée, il faut compter sur des transducteurs ayant des centaines de milliers d'états et de transitions ; une cascade de transducteurs peut être parfois plus économique.

Un autre intérêt de disposer de plusieurs transducteurs est la définition de stratégies. Que faire si un mot donné n'est pas défini par le transducteur ? Dans le cas de mots inconnus, l'analyse est un échec. Une solution est alors de fournir un autre transducteur, appelé « devineur », qui cherche à analyser le mot inconnu selon sa forme. Ainsi, si sa terminaison est proche d'une conjugaison (*e.g.* un mot qui terminerait en *-ez* pourrait vraisemblablement être la conjugaison à la deuxième personne du pluriel d'un verbe inconnu du dictionnaire), ou de la forme d'un adverbe, ou d'un nom propre, etc. le devineur propose une analyse alternative ; au pire, un comportement par défaut (donner l'étiquette « mot inconnu » par exemple) est alors défini, à moins que d'autres stratégies ne soient mises en œuvre par le biais d'un troisième transducteur, et ainsi de suite.

1.3.2 Le système ATEF

ATEF est un langage spécialisé pour linguiste (LSPL) défini et implémenté entre 1971 et 1973 par Jacques Chauché, Pierre Guillaume et Maurice Quézel-Ambrunaz [Chauché, 1974]. Il a depuis subi quelques modifications et a été utilisé pour écrire des analyseurs morphologiques du français, du russe, du portugais, de l'anglais, de l'allemand [Guilbaud, 1980] et du japonais [Laurent, 1977]. Cette section est une courte introduction à ATEF ; il en sera de nouveau question dans le chapitre suivant, où l'on revient sur l'analyse morphologique du japonais avec ATEF.

Le système ATEF est fondé sur un modèle de transduction d'états finis non-déterministe, mais se place à un niveau d'abstraction plus élevé que XFST (par exemple) : ici, l'automate d'analyse est « simulé », et le linguiste n'utilise pas d'expressions régulières ou des règles de réécriture, mais décrit dictionnaires, formats et règles morphologiques. Le résultat de l'analyse est un graphe quadrichrome arrière dont les nœuds sont les décorations associées aux solutions trouvées, et où les arcs indiquent la compatibilité des analyses avec celle des quatre occurrences précédentes. Il est généralement transformé en une structure arborescente, avec ou sans homophrases (conservant ou non la source des ambiguïtés) pour produire un résultat plus manipulable.

1.3.2.1 La composante linguistique

Les variables. Les variables permettent de coder les informations grammaticales, sémantiques ou logiques sur les mots analysés. Ces variables sont divisées, à la discrétion du linguiste, entre variables morphologiques et variables syntaxiques (il n'y a pas de différence formelle entre les deux catégories de variables), et entre variables exclusives (qui ne peuvent prendre qu'une valeur à la fois) et non-exclusives (qui peuvent prendre plusieurs valeurs simultanément). Les variables peuvent également avoir un rôle purement stratégique, sans signification linguistique particulière.

Exemple (les exemples sont tirés de l'analyseur morphologique de l'allemand) :

```
KMS := (VB, NM, ADJ, ADIP, DR, COP, PC, NA).
SUBCOP := (COOR, CJS, PRP).
```

`KMS` désigne la catégorie morpho-syntaxique (verbe, nom, adjectif, etc.). La valeur `COP` correspond à un lexème d'hypotaxe ou de parataxe, et est précisée par la variable `SUBCOP`, qui peut désigner une conjonction de coordination ou de subordination, ou une préposition.

Les formats morphologiques et syntaxiques. Les formats définissent des classes de morphèmes qui partagent des mêmes valeurs de variable. Deux fichiers de formats sont définis : les formats morphologiques et les formats syntaxiques. Un format morphologique déclenche l'appel d'au moins une règle (voir plus bas).

Exemple : `FMINV` est un format morphologique vide :

```
FMINV 01 == .** format vide.
```

`FSCOOR` est un format morphologique décrivant les conjonctions de coordination :

```
FSCOOR 01 == .** KMS -E- COP, SUBCOP -E- COOR.
```

Les dictionnaires. Les dictionnaires de bases, d'affixes et de tournures (expressions à mots multiples plus ou moins figées) sont des collections d'entrées qui contiennent : un morphe (le segment à reconnaître lors du découpage), ses formats, morphologique et syntaxique, et éventuellement une unité lexicale qui lui est liée. Différents types de morphes sont définis dans différents types de dictionnaires (bases, affixes et tournures ne sont pas mélangées) ; mais pour des raisons stratégiques, plusieurs dictionnaires de morphes de même type peuvent être définis.

Exemple : un article de dictionnaire pour la base *und* (« et » en allemand) :

```
UND == FMINV (FSCOOR, UND).
```

`FMINV` est le format morphologique qui va déclencher la ou les règles correspondantes ; `FSCOOR` est le format syntaxique, et l'unité lexicale associée est `UND`.

La grammaire. La grammaire contient les règles qui s'appliquent quand une forme est reconnue. Une règle est déclenchée quand une forme appartenant à un format morphologique donné est reconnue ; son application peut être soumise à des conditions (sur le texte restant à analyser, sur les quatre formes précédemment analysées, ou sur la forme en train d'être analysée). Les conditions peuvent aussi porter sur l'application d'au moins une sous-règle parmi une liste.

Il y a trois sortes d'actions possibles : premièrement, l'affectation de valeurs au masque `C` qui contient le résultat de l'analyse en cours. Deuxièmement, la chaîne restant à analyser peut subir des transformations. Troisièmement, une fonction spéciale d'ATEF peut être appelée pour :

- contrôler la recherche (*cf.* la composante algorithmique) ;
- stocker le résultat courant (utile pour les mots composés) ;
- créer de nouvelles valeurs d'unités lexicales à partir de la forme en cours ;
- décider qu'une borne de phrase est atteinte.

Exemple : la règle `RINV` donne au masque courant les valeurs du masque du morphe analysé, issues du dictionnaire. Elle s'applique aux morphes de format `FMINV` ou `PP14`, si le caractère suivant la forme courante est un blanc :

```
RINV : FMINV - PP14 == VAR(C) := VAR(A)
                     / SCHAINE(A, 0, 1) -E- ' '.
```

1.3.2.2 La composante algorithmique

Le transducteur défini par ATEF transforme une chaîne de caractères en entrée en combinaison de valeurs et de variables par un processus d'analyse non-déterministe. Le texte est segmenté en formes, et chaque forme est analysée. Dans le cas d'un échec de la segmentation, un retour arrière est opéré dans l'arborescence des choix. Le graphe quadrichrome arrière est construit au fur et à mesure de l'analyse, conservant toutes les ambiguïtés de segmentation et d'analyse rencontrées.

Chaque pas d'analyse correspond à trois étapes successives :

1. choix d'un dictionnaire parmi les dictionnaires ouverts. La variable spéciale `DICT` contrôle quels sont les dictionnaires ouverts à tout moment ;
2. choix d'un morphe dans le dictionnaire ;
3. application d'une règle correspondant à ce morphe.

On dispose dans le système de fonctions de contrôle sur l'arborescence des choix : `-FINAL-`, `-STOP-`, `-ARRET-`, `-ARD-` et `-ARF-`. Ces fonctions permettent d'éliminer des choix déjà faits ou à faire en coupant des branches de l'arborescence. Par exemple, `-FINAL-` coupe toutes les branches de l'arborescence hormis celle passant par `-FINAL-`, et continue sur le premier tronçon qui mène à une solution pour l'occurrence analysée ; `-STOP-` arrête le traitement si la règle en question mène à une solution, mais conserve toutes les solutions déjà trouvées, etc.

1.3.2.3 Le traitement des mots inconnus

Lorsqu'aucune occurrence ne peut être reconnue, c'est la « règle du mot inconnu » qui s'applique en simulant l'analyse d'une chaîne vide au format morphologique spécial `MODINC` qui appelle obligatoirement la règle `MOTINC`, dont le rôle est de produire une unité lexicale. L'appel à des sous-règles définies par le linguistique permet de définir une grammaire du mot inconnu, à l'instar du devineur évoqué dans XFST [Guilbaud et Boitet, 1997].

Exemple : cette règle est tirée d'un analyseur du français. Ici, toute l'occurrence est « consommée » par `TCHAINE`, qui transforme le reste de l'entrée, et une unité lexicale correspondante est créée par la fonction `-TRANS-`.

```
MOTINC : MODINC == CAT(C) := INC, -TRANS-
                 / / TCHAINE(0, *, '').
```

1.3.3 Transduction de graphes de chaînes avec les systèmes-Q

Les systèmes-Q [Colmerauer, 1970] constituent un système de règles de transduction de graphes de chaînes. Contrairement aux autres formalismes décrits dans ce chapitre, son champ d'application n'est pas limité à la morphologie, et les systèmes-Q n'ont pas de lien particulier avec les théories morphologiques vues ici, et sont beaucoup plus abstraits.

Le principe de ces systèmes-Q est de représenter les différentes analyses possibles d'un texte sous la forme d'un graphe de chaînes, et de se doter d'un système de règles pour manipuler ces graphes. Un traitement-Q est alors l'application successive d'un ou de plusieurs systèmes-Q manipulant un texte représenté sous cette forme.

1.3.3.1 Les graphes de chaînes

La factorisation choisie pour représenter les ambiguïtés d'analyse du texte dans les systèmes-Q est le graphe de chaîne. Ces graphes sont équivalents à des DAG, car ils n'admettent pas de circuit. Ils possèdent en outre la particularité, comme les treilles, d'avoir un unique nœud « initial » et un unique nœud « final ». Les nœuds sont numérotés pour pouvoir y faire référence dans la représentation textuelle d'un graphe. La figure 1.8 montre un graphe de chaînes.

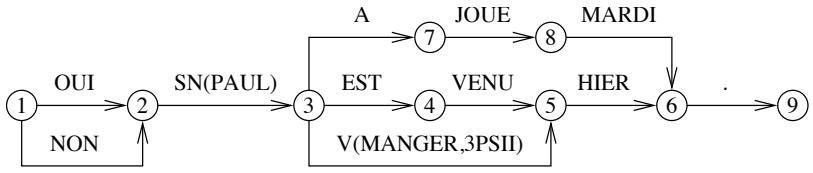

FIG. 1.8 – Un graphe de chaînes

Les graphes de chaînes représentent les données d'un traitement-Q. Les étiquettes des arcs ne sont cependant pas de simples chaînes, mais une représentation textuelle d'arbre. En effet, les systèmes-Q définissent trois types de données :

1. les étiquettes, qui sont de simples chaînes de caractères (*e.g.*`PAUL`) ;
2. les listes, dont les éléments sont des étiquettes ou des arbres, séparés par une virgule (*e.g.*`MANGER, 3PSII`) ;
3. les arbres, représentés sous forme parenthésée (*e.g.*`V(MANGER, 3PSII)`).

Une chaîne est une concaténation dont les symboles sont des arbres ou des étiquettes, séparés par le caractère `+`. Dans la figure 1.8, le graphe porte par exemple la chaîne `OUI + SN(PAUL) + A + JOUE + MARDI + ..`

1.3.3.2 Les systèmes-Q simplifiés

Un système-Q est un ensemble de règles de transduction de chaînes. Dans leur forme la plus simple, sans paramètres, les règles d'un système-Q sont de la forme :

$a_1 + a_2 + \ldots + a_m == b_1 + b_2 + \ldots + b_n$.

Cette règle indique que ce système-Q doit transformer toute occurrence de la chaîne de gauche en la chaîne de droite.

Exemple : le système-Q suivant reconnaît les chaînes des la forme $A^n B^n C^n$ en les réécrivant en S . Il est composé des quatre règles :

```
A + B + C == S.
A + S + B(*) + C == S.
B(*) + C == C + B(*).
B + B == B + B(*).
```

L'application des règles s'effectue en deux étapes. La première est l'ajout de nouvelles chaînes. Lorsqu'une chaîne dans le graphe correspond au membre gauche d'une règle, alors une nouvelle chaîne est ajoutée dans le graphe, commençant et terminant à la même position. Ce processus est répété jusqu'à la convergence éventuelle en prenant simplement soin de ne pas appliquer deux fois la même règle à la même chaîne. La figure 1.9 représente deux applications successives de règle, la première à s'appliquer est la règle `B + B == B + B(*)`, la seconde est `B(*) + C == C + B(*)`, dont l'application est rendue possible par l'ajout d'une chaîne par la règle précédente.

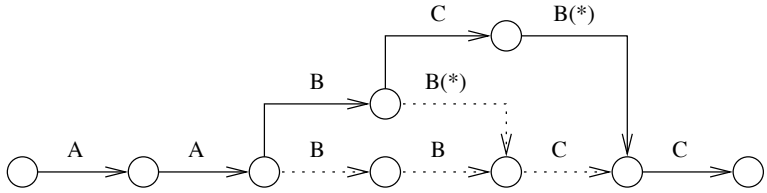

FIG. 1.9 – Ajouts de chaînes

La deuxième phase consiste ensuite à effacer chaque transition qui a été utilisée pour au moins une règle (les transitions utilisées sont représentées en pointillés). Une fois ces transitions éliminées, toutes les transitions qui ne font pas partie d'un chemin reliant les deux extrémités du graphe sont elles aussi éliminées (ce qui peut avoir pour effet de produire un graphe vide). Dans la figure 1.10, la seule transition qui subsiste est celle marquée `S` et qui témoigne du succès de la reconnaissance de la chaîne `AABBCC`.

1.3.3.3 Les systèmes-Q génériques et les traitements-Q

Dans un système-Q « général », les règles admettent paramètres et conditions. Les symboles d'une chaîne peuvent en fait être des étiquettes, listes et arbres paramétrés ; et une règle peut être complétée par une condition d'application.

Exemple : le système-Q précédent peut être réalisé plus simplement :

```
A* + A*(U*) == A*(1, U*) / A* -DANS- A, B, C.
A(U*) + B(U*) + C(U*) == S(1, U*).
```

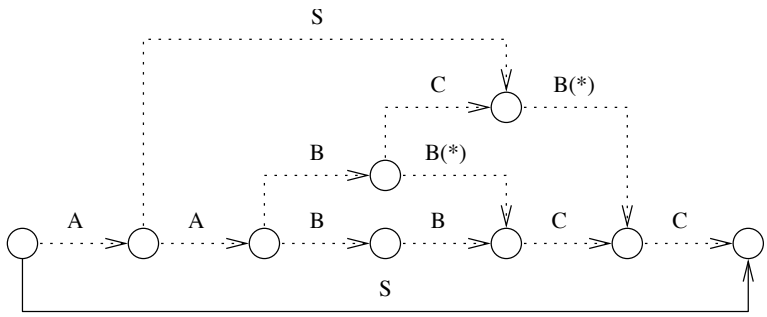

FIG. 1.10 – Suppression des arcs utilisés

Les paramètres de ces deux règles sont `A*` (étiquette paramétrée) et `U*` (liste paramétrée). La première règle s'applique si le paramètre étiquette `A*` est `A`, `B`, ou `C` ; la deuxième règle s'applique sans condition. La figure 1.11 montre l'application de ces règles à la chaîne `A+A+A+B+B+B+C+C+C`, juste avant la suppression des arcs ; le dernier arc restant est l'arc `S(1,1)`. La liste dominée par S « compte » donc le nombre d'occurrence de chacun des A, B et C dans l'expression reconnue.

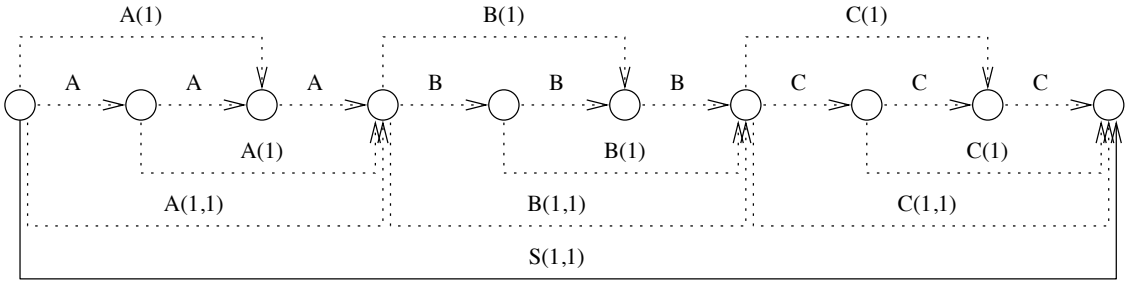

FIG. 1.11 – Système-Q générique

Un traitement-Q consiste en une série de systèmes-Q qui s'appliquent en cascade. Les données en entrée sont présentées sous la forme de graphes de chaînes, et chaque système-Q est appliqué sur le résultat du système-Q précédent pour obtenir le résultat final. Par exemple, Colmerauer a construit un paraphraseur « jouet » du français en quatre phases : une phase d'analyse, suivie d'une phase de synthèse (la génération des paraphrases) et de deux phases de normalisation où les arbres sont « aplatis » pour obtenir un résultat lisible.

Conclusion

La vue de l'analyse morphologique offerte par ce chapitre est nécessairement partielle et ne prétend absolument pas être exhaustive. Les formalismes et les structures présentés sont surtout du niveau trois selon la classification de Chomsky, c'est-à-dire de l'ordre des langages rationnels. Un avantage certain est que l'on dispose pour ces langages d'outils d'analyse efficaces, mais que leur puissance expressive est relativement limitée.

Des outils de plus haut niveau existent : les systèmes-Q, par exemple, même si ceux-ci ne sont pas spécifiquement destinés à la morphologie. Des approches hors-contexte ont été employées dans les années 1970, comme par exemple dans le système METAL, ou dans les ATN (*Augmented Transition*

Networks, réseaux de transition augmentés : les arc portent des conditions sur l'entrée et sur des « registres » de travail, ainsi que des actions sur ces registres et sur le contrôle de haut niveau du système).

Parmi les langages de programmation, on aurait pu également citer comme exemple intéressant Prolog, dont le formalisme de DCG se prête à toutes sortes d'analyse linguistiques, y compris en ce qui concerne l'analyse morphologique.

Chapitre 2

Des approches spécifiques de l'analyse morphologique en général, et de l'itémisation en particulier

Introduction

Si le chapitre 1 s'est attaché à montrer la variété des techniques pour l'analyse morphologique, il est temps de s'attacher à des analyseurs spécifiques actuellement utilisés. C'est l'occasion de voir à l'œuvre les machines d'états finis, pondérées ou non, et le formalisme ATEF, pour une variété de langues qui va du français au chinois.

Plutôt que de faire la distinction classique « langue avec séparateurs » / « langue sans séparateurs », il est plus intéressant de faire une distinction selon l'utilisation ou non de données statistiques pour l'itémisation et l'analyse. On trouvera des exemples de chacune de ces deux approches dans les sections 2.1 (pour le français et le japonais en particulier) et 2.2 (pour le chinois et le japonais).

Afin de ne pas se focaliser uniquement sur l'analyse morphologique, la section 2.3 met ces méthodes stochastiques ou non en perspective en voyant leur application à d'autres applications présyntaxiques, comme la segmentation en phrases ou le prétraitement du texte, ainsi qu'à des langues à la morphologie plus exotique, comme le thaï.

2.1 Méthodes non-stochastiques : règles et heuristiques

Tout d'abord, ce sont les méthodes non-stochastiques qui retiennent notre attention, car hormis les transducteurs d'états finis (sur lesquels on revient plus bas), ce sont celles qui sont le plus proches des outils et formalismes vus dans le chapitre précédent. Cette section suit d'ailleurs le même genre de progression, puisqu'elle montre l'application de trois outils distincts : premièrement, on montre un moteur de segmentation générique en Perl (avec une application au français) ; deuxièmement, la construction et l'utilisation d'un transducteur lexical avec les outils du calcul d'état fini de Xerox ; et troisièmement, un analyseur morphologique du japonais réalisé avec ATEF.

2.1.1 Segdict : un moteur d'itémisation générique en Perl

2.1.1.1 Le moteur d'itémisation

Segdict (le nom n'est pas très original, mais adéquat : SEGmentation avec DICTionnaire) est une application d'itémisation très simple que l'on a conçu pour illustrer le principe de séparation entre données et processus. Il sert d'ailleurs de support de cours en licence d'informatique.

Il s'articule autour d'un moteur, qui est le programme Perl (`segdict.pl`), et d'un dictionnaire, qui est un fichier décrivant les entités à segmenter par les expressions régulières étendues de Perl. Le programme accepte en paramètre un nom de fichier, celui du dictionnaire, et segmente tous les fichiers passés en argument, ou l'entrée standard si aucun fichier n'est spécifié. La sortie est on ne peut plus simple : les items identifiés sont affichés sur la sortie standard, un item par ligne.

Prétraitement du texte. Le texte subit deux étapes de prétraitement : il est tout d'abord segmenté en paragraphes, chaque paragraphe étant itémisé à son tour. Dans un paragraphe, les espacements sont normalisés.

Perl définit deux unités de texte : l'enregistrement (*record*) et le caractère. L'accès à un fichier texte se fait enregistrement par enregistrement, à l'aide de l'opérateur `<>`. Chaque appel à cet opérateur renvoie l'enregistrement suivant, sous la forme d'une chaîne de caractères. Une variable spéciale, appelée `$INPUT_RECORD_SEPARATOR` et abrégée `$/`, permet de définir le séparateur d'enregistrement, et par conséquent de définir les enregistrements eux-mêmes. Par défaut, cette variable a pour valeur le caractère ASCII de nouvelle ligne, et le comportement habituel de Perl est de lire un fichier ligne par ligne. Une valeur spéciale, `"\n\n"` (c'est-à-dire deux caractères ASCII de nouvelle ligne), permet de lire le texte paragraphe par paragraphe : les enregistrements doivent être séparés par deux nouvelles lignes ou plus.

Changer la valeur de `$/` permet une première segmentation du texte en paragraphes. Même si cette notion de « paragraphe » est rudimentaire, elle définit une bonne unité de traitement, car il est fort peu probable de rencontrer des items contenant plusieurs nouvelles lignes. Une fois un paragraphe lu, une normalisation des espacements dans le texte a lieu à l'aide de la construction classique en Perl

```
$paragraphe =~ s/^\s*//;
$paragraphe =~ s/\s*$//;
$paragraphe =~ s/\s+/ /g;
```

On a ici trois substitutions successives de la variable `$paragraphe`, qui contient le paragraphe courant. La première supprime la séquence de zéro ou plusieurs caractères d'espacement (`\s*`) au début du paragraphe (représenté par le méta-caractère `^`) en la remplaçant par la chaîne vide. La deuxième substitution opère de la même manière en fin de chaîne (représentée par le méta-caractère `$`). Enfin, la troisième substitution achève la normalisation en remplaçant toute séquence d'au moins un espace (`\s+`) par un unique blanc.

Au cours de la segmentation, l'invariant suivant doit être maintenu : un paragraphe de texte ne commence ni ne finit par un blanc, et chaque séquence de caractères qui ne sont pas des espacements est séparé par un seul et unique blanc. Cette convention facilite grandement l'écriture des motifs du dictionnaire de segmentation. Par contre, d'autres normalisations classiques, comme la normalisation typographique (*e.g.* transformer toutes les lettres en majuscules ou minuscules, supprimer les accents si besoin, etc.) ne sont pas effectuées, et les variantes possibles doivent être prises en compte dans l'écriture du dictionnaire.

Le segmenteur pas à pas. La figure 2.1 montre la partie essentielle de l'application. Les tâches subalternes comme l'acquisition du dictionnaire ne sont pas pertinentes ; le programme complet est présenté dans l'annexe B. Il suffit pour comprendre le principe de fonctionnement de l'itémiseur de supposer que le dictionnaire est contenu dans un tableau nommé `@dict`. Chaque élément de ce tableau a deux champs numérotés : le premier (numéro 0) est le motif à reconnaître ; le second (numéro 1) est l'item à afficher si le motif est reconnu. Le dictionnaire est décrit en détail plus bas.

L'algorithme de segmentation accède au texte paragraphe par paragraphe (lignes 1 et 3). L'appel à `<>` sans paramètres stocke automatiquement le paragraphe lu dans la variable `$ARG`, abrégée `$_` qui est la variable par défaut de nombreuses opérations. La ligne 4 concerne la normalisation des espaces décrite plus haut (condensée sur une seule ligne).

La segmentation elle-même a lieu lignes 5-16. L'itémiseur procède en consommant le texte contenu dans `$_` : à chaque itération, le texte est réduit, jusqu'à ce que `$_` ne soit plus qu'une chaîne vide,

après quoi la segmentation du paragraphe est terminée. Chaque motif du dictionnaire est essayé dans l'ordre (lignes 6-13) ; si le motif correspond à un préfixe du paragraphe à segmenter, alors la partie reconnue est consommée (ligne 7), ainsi que l'éventuel espacement qui suit (ligne 11). Lignes 8-10, l'item correspondant au préfixe reconnu est affiché. Le `last` de la ligne 12 permet de sortir de cette boucle et de recommencer le processus avec la chaîne restante.

On revient un instant sur les lignes 8 à 10 : la fonction `eval` permet de spécifier dans le dictionnaire une action en Perl pour chaque motif, ce qui ressemble au mode de fonctionnement de `lex`. Cette action doit avoir pour résultat une chaîne de caractères, qui est ensuite affichée. Pas de structure de données complexe, ici on a juste à afficher l'item reconnu sur une ligne, ou une chaîne vide s'il n'y a pas d'item à afficher (ce qui permet de supprimer des formes).

```
 1 $/ = "\n\n";
 2
 3 while (<>) {
 4   s/^\s*//; s/\s*$//; s/\s+/ /g;
 5   while ($_) {
 6     for my $motif (@dict) {
 7       if (s/^$motif->[0]//) {
 8         if (my $item = eval($motif->[1])) {
 9           print "$item\n";
10         }
11         s/^ ?//;
12         last;
13       }
14     }
15   }
16 }
```

Fig. 2.1 – Le moteur d'itémisation en Perl

2.1.1.2 Le dictionnaire d'itémisation

Le dictionnaire est un fichier texte qui comprend une entrée par ligne (on peut laisser des lignes blanches, et ajouter des commentaires précédés du symbole `#`). Chaque entrée comporte deux champs : le motif à reconnaître et la sortie à produire. Par exemple, l'entrée suivante :

```
du      "de\nle"
```

reconnaît le mot *du* et produit les deux items *de* et *le* (séparés par une nouvelle ligne).

L'action par défaut est tout simplement `$&`, qui est une variable spéciale contenant la chaîne reconnue par la dernière substitution. Si aucune action n'est spécifiée dans le dictionnaire, c'est celle-ci qui est entreprise.

La règle de segmentation principale est très simple :

```
\w+(\-\w+)*
```

et permet de reconnaître une séquence de caractères alphanumériques (`\w`), contenant possiblement des tirets pour les mots composés (comme *bleu*, *vert*, *bleu-vert*, *vert-de-gris*). On ajoutera donc des règles pour reconnaître des formes figées contenant des espaces (*a priori*, *parce que*),

Enfin, une dernière entrée est ajoutée automatiquement par le segmenteur :

```
.        $&
```

C'est un dernier recours : si aucune entrée n'a été reconnue, celle-ci permet de reconnaître n'importe quel caractère (`.`) et de l'afficher. Cette règle permet de toujours consommer du texte, afin que l'algorithme termine toujours. On peut évidemment outrepasser cette règle en spécifiant une entrée dans le dictionnaire qui est aussi générale, et qui sera nécessairement reconnue avant (celle-ci étant la toute dernière).

2.1.1.3 Extensibilité et généricité

Le dictionnaire complet est présenté dans l'annexe B. Il s'agit en fait d'un squelette proposé aux étudiants, qui le complètent en travaillant sur différents textes, en ajoutant des entités à reconnaître qui ne sont pas des mots : nombres, dates, sommes monétaires, titres, etc.

Cet itémiseur, bien que très simple, est adaptable à n'importe quelle langue où l'heuristique de la plus longue chaîne est raisonnable. Selon le sens donné au mot « raisonnable », cela inclut à peu près toutes les langues écrites. Il semble que ce programme soit biaisé, en étant plus adapté aux langues qui utilisent des séparateurs entre les mots, puisque l'on peut décrire l'immense majorité des mots par une simple expression régulière. Mais un segmenteur du chinois utilisant l'heuristique de la plus longue chaîne (décrite en détail plus bas) est réalisable avec Segdict en utilisant un dictionnaire de mots du chinois ordonnés selon leur longueur (les plus longs en premier), et quelques règles simples pour gérer les mots inconnus :

- si un caractère chinois est rencontré sans pouvoir reconnaître un mot, ce caractère est choisi comme item, et l'itémisation peut continuer ;
- si un autre type de caractère est rencontré (par exemple un symbole ou un caractère latin), c'est la chaîne de caractères du même type la plus longue possible qui est choisie comme item.

Une expérience a été réalisée pour tester ce segmenteur ; si les résultats sont conformes à ce qu'on peut en attendre, avec un dictionnaire de taille importante (de quelques dizaines de milliers de formes), l'efficacité n'est pas vraiment au rendez-vous, car des dizaines de milliers d'expressions sont testées en séquence pour chaque item.

2.1.2 Lemmatisation des noms et adjectifs du français avec XFST

La construction d'un lemmatiseur du français avec les outils de Xerox est une occasion de détailler et préciser certains aspects déjà évoqués dans la section 1.3.1. Atteindre une large couverture du français est un travail énorme, qui peut heureusement être divisé en tâches relativement indépendantes : par exemple, la morphologie des verbes peut se faire indépendamment de celle des noms et des adjectifs. Les transducteurs alors créés sont ensuite unis en un seul grand transducteur.

2.1.2.1 La définition des lexiques avec `lexc`

Noms et adjectifs partagent le même système de flexion et sont regroupé dans le même lexique. Le cœur de ce lexique est constitué de la liste des lemmes. Les lemmes sont organisés par catégories qui gouvernent les différents genres, nombres et catégorie syntaxique qu'un mot peut prendre. Par exemple, un « adjectif régulier » est un adjectif qui se trouve au masculin comme au féminin, au singulier comme au pluriel. Certains mots peuvent être à la fois nom et adjectif, d'autres sont invariables, etc.

Le lexique est au format `lexc` et divisé en sous-lexiques. Le sous-lexique principal, `nom-adj`, contient tous les lemmes ; chaque « catégorie » de lemme se voit attribuer un lexique particulier qui définit les traits morphologiques des mots de ces catégories.

Exemple : *ajusté* est un adjectif régulier, il est défini par l'entrée suivante dans le sous-lexique `nom-adj` :

```
ajusté reg-adj ;
```

Le sous-lexique `reg-adj` est la classe de continuation d'adjectifs réguliers comme *ajusté* :

```
LEXICON reg-adj
  ++Masc   reg-adj-num;
  ++Fem    reg-adj-num;

LEXICON reg-adj-num
  +SG+Adj # ;
  +PL+Adj # ;
```

Ainsi, *ajusté* peut prendre quatre formes lexicales différentes (masculin ou féminin, singulier ou pluriel). Les étiquettes morphosyntaxiques sont de la forme `+Etiq`, et constituent chacune un seul symbole dans l'automate. Le `+` qui précède l'étiquette du genre est en fait un séparateur entre le lemme et ses étiquettes.

Pour assurer une large couverture et analyser certains néologismes, des préfixes couramment employés et très productifs sont optionnellement ajoutés à chaque racine (*demi-*, *pseudo-*, *re-*, *anti-*, *euro-*, etc.) Le lexique est en fait séparé en deux parties : les mots commençant par un son vocalique (une voyelle ou un *h* non aspiré) et les autres.

Traitement des exceptions. Le traitement des flexions régulières est l'objet de la section suivante, mais il faut déjà prendre en compte les exceptions. Deux catégories d'exceptions se présentent : d'une part, celles qui sont suffisamment nombreuses pour faire l'objet d'une règle, et d'autre part, celles qui doivent être traitées individuellement. Le premier cas est évoqué plus bas.

Dans le deuxième cas on trouve les exceptions uniques en leur genre (*œil*/*yeux*, *ail*/*aulx*, etc.) ou des mots empruntés aux langues étrangères, ou des ensembles de mots qui suivent une forme régulière mais sont peu nombreux (comme *hibou*, *chou*, *genou* etc. qui ont un pluriel en *-x*). Pour celles-ci, les exceptions sont prises en compte directement par le lexique.

Exemple : *ciel* a un pluriel irrégulier en *cieux*, mais également en *ciels* (selon la signification, mais celle-ci n'est pas considérée ici). Le pluriel de *ciel* apparaît alors deux fois dans le lexique :

```
ciel++Masc++PL+Noun:cieux # ;
ciel++Masc++PL+Noun ;
```

La première forme aura donc un pluriel irrégulier, alors que la seconde suivra le modèle de flexion habituel.

2.1.2.2 Flexions et règles morphologiques à deux niveaux avec `twolc`

Les règles à deux niveaux écrites pour traiter le pluriel et le féminin des noms et adjectif du français consistent à mettre en correspondance les formes lexicales et les formes de surface. Les règles à deux niveaux sont des règles gouvernant la correspondance de deux symboles, un symbole de surface et un symbole lexical. Dans la situation présente, ce sont les étiquettes qui vont être réalisée par un symbole de surface : ainsi, `+PL` sera réalisé par `s` en surface ; `+SG` par `0` (la notation de `twolc` pour epsilon).

Exemple : la correspondance entre la forme de surface de l'adjectif *débutantes* et sa forme lexicale donne

```
d é b u t a n t + +Fem +PL +Adj
| | | | | | | | |   |   |    |
d é b u t a n t 0   e   s    0
```

Le système de règles défini doit prendre en compte différents cas de figure. Pour le pluriel, il faut prendre en compte les mots se terminant par une sifflante (*x*, *z* ou *s*) qui ne sont pas marqués au pluriel, la plupart des mots se terminant en *-al*, *-ail*, *-au* qui ont un pluriel en *-aux*, etc. Pour le féminin, les mots se terminant en *-e* qui n'ont généralement pas de marque supplémentaire, ou qui ont leur pluriel en *-esse*, etc. Un jeu d'une quinzaine de règles suffit à traiter la plupart des cas, les exceptions étant prises en compte dans le lexique.

Il reste cependant le problème des noms composés, comme *porte-fenêtre*, *cul-de-lampe*, *carte bleue*, *chair à saucisse*, etc. Si certains se comportent bien pour le système de flexion décrit jusqu'ici (*ie.* seule la dernière partie du mot composé est fléchie), d'autres subissent des flexions à l'intérieur du composé. En partant du principe qu'un mot composé est toujours composé de deux mots parties, au début et à la fin du composé, on distingue quatre catégories de noms composés qui nécessitent un traitement particulier :

1. les deux parties du composé varient et en nombre ;
2. seule la première partie du composé varie en nombre ;
3. les deux parties varient en genre et en nombre ;
4. seule la première partie varie en genre et en nombre.

Comme la première partie du composé varie comme n'importe quel mot, on définit un jeu de règles de « prétraitement » s'appliquant avec les règles sur les flexions proprement dites, qui copient les étiquettes morphologiques à l'intérieur du composé selon la nature du mot composé (chacune des quatre catégorie correspondant à un symbole diacritique spécial).

Exemple : *chef-d'œuvre* appartient à la deuxième catégorie. L'analyse de *chefs-d'œuvre* est alors :

```
c h e f 0   0    0  - d ' o e u v r e + ^pls +Masc +PL +Noun
        |   |    |                     |         |    |
c h e f + +Masc +PL - d ' o e u v r e +   0  +Masc  0    0
        |   |    |                   |         |
c h e f 0   0    s  - d ' o e u v r e 0   0    0    0    0
```

Le symbole `^pls` dénote l'appartenance de *chef d'œuvre* à la deuxième catégorie de mot composé, et déclenche les règles copiant `+PL` et `+Masc` à la fin de `chef` et la suppression de `+PL` dans la première étape.

2.1.2.3 La compilation du transducteur lexical avec `xfst`

Le transducteur lexical final est obtenu en plusieurs étapes :

1. compilation avec `lexc` des deux lexiques, et avec `twolc` des deux jeux de règles ;
2. composition avec `lexc` de chacun des deux lexiques avec les deux jeux de règles ;
3. union des deux transducteurs obtenus pour obtenir un lexique unique ;
4. composition avec `xfst` du lexique avec des règles de réécriture de nettoyage qui suppriment les divers signes diacritiques (`+`, `^è`, etc.) pour donner le lexique final.

Les règles de nettoyage sont spécifiées par des expressions régulières et compilées avec `xfst`. L'automatisation du processus peut se faire avec des outils comme `make`, comme lors de la compilation de programmes complexes en C.

Finalement, le transducteur obtenu réalise directement la transduction entre niveau lexical et niveau de surface ; toutes les étapes intermédiaires ont été « absorbées » par le composition des transducteurs successifs (nettoyage et jeux de règles).

Exemple : la correspondance entre *poétesse* et sa forme lexicale se fait réellement en trois étapes : les deux jeux de règles morphologiques, et le nettoyage des symboles intermédiaires (l'analyse se fait du bas vers le haut) :

```
p o è t e 0  0   +Fem +SG +Noun
| | | | | |  |    |    |    |
p o è t e + ^sse +Fem +SG +Noun
| | | | | |  |    |    |    |
p o è t e + ^sse +Fem +SG   0
| | | | | |  |    |    |    |
p o é t e s  s    e    0    0
```

Mais le transducteur final réalise cette correspondance directement :

```
p o è t e 0 0 +Fem +SG +Noun
| | | | | | |  |    |    |
p o é t e s s  e    0    0
```

L'exemple ci-dessus illustre également la réversibilité du processus : si l'analyse se fait « de bas en haut » à partir d'une forme de surface, la génération se fait identiquement « de haut en bas ».

2.1.3 Analyse morphologique du japonais avec ATEF

Bien qu'il existe de nombreux analyseurs morphologiques réalisés avec ATEF, celui du japonais [Laurent, 1977] est intéressant à (au moins) trois titres. Premièrement, il prouve l'universalité du formalisme en l'appliquant à une langue bien différente de celles qui sont habituellement traitées avec. Deuxièmement, comme il a déjà été dit, les analyseurs morphologiques du japonais ou du chinois sont quasiment toujours réalisés selon un modèle de développement fermé, et il s'agit là d'un beau contre-exemple. Troisièmement, en plus de résoudre des problèmes linguistiques, l'auteur du système (construit il y a plus de 25 ans !) a dû également s'affranchir des problèmes de représentation et d'impression des caractères japonais sur une machine IBM utilisant l'encodage EBCDIC (*cf.* Annexe A).

Commençons par ce dernier point. La translittération des caractères japonais utilisée provient du système de segmentation développé précédemment à Tokyo par Toshio Ishiwata. Ainsi, le début d'un texte analysé a cette forme :

```
KON  NICHI  ,    KI  KAI  SETU  KEI  KEI  SAN  , (...)
A.   #,     ,'   7E  N3   4R    01   01   D6   ,'
```

(la ligne supérieure est une romanisation des caractères japonais, *kanji* et *kana*). Chaque caractère japonais est codé par deux caractères imprimables de l'EBCDIC ; cela permet l'encodage de plus de 3 500 caractères différents et est largement suffisant pour du texte courant (cet encodage est annonciateur des standards actuels, qui n'encodent pas tellement plus de caractères). Les virgules (il y en a dans cet exemple) et les autres signes de ponctuation comme la marque de fin de phrase sont suivies d'un blanc.

Dans ATEF, une forme est une occurrence entre deux espaces, ce qui donne en français ou en allemand par exemple une première segmentation. Comme les mots ne sont généralement pas séparés par des blancs en japonais, l'analyse d'une forme consiste en réalité à analyser un fragment de texte de l'ordre de la phrase (même si les fragments sont généralement moins grands puisque tout signe de ponctuation est un séparateur). Bien que la taille d'une forme soit limitée à 256 caractères dans ATEF (soit 128 caractères japonais), il est peu probable qu'un fragment atteigne cette taille qui représenterait un bon tiers de page sans aucun signe de ponctuation.

L'analyse en ATEF consiste donc à segmenter chacune de ces formes et à analyser chaque mot qu'elle contient. Par exemple, l'analyse du fragment « *konnichi, kikaisetukeikeisan, ...* » donne :

```
UL(ULTXT)
  UL(ULFRA)
    UL(ULOCC)
      A.#, : UL(KONNICHI), SEPA(BLANC), KAT(TPS), K(SUB)
    UL(ULOCC)
      ,' : UL(VIRGULE)
    UL(ULOCC)
      UL(ULMCP)
      ...
       7EN34R0101D6 : UL(KIKAI), ...
       7EN34R0101D6 : UL(SETUKEI), ...
       7EN34R0101D6 : UL(KEISAN), ...
       ...
```

2.2 Méthodes stochastiques

La différence principale entre les méthodes stochastiques ou non est que les méthodes stochastiques se fondent sur des modèles statistiques de la langue (modèles d'unigrammes ou de bigrammes, par exemple), en plus des données classiques comme les lexiques. Certaines méthodes n'emploient d'ailleurs même pas de lexique ou de dictionnaire, comme on le voit dans la première sous-section.

2.2.1 Utilisation de statistiques sans dictionnaire

Même si elle est assez marginale, cette technique existe. L'intérêt de ne pas utiliser de ressources créées à la main comme des dictionnaires ou des systèmes de règles (qui sont longues et coûteuses à produire), mais d'utiliser des corpus, qui sont plus facilement disponibles, pour extraire des modèles statistiques de la langue considérée.

Un des premiers travaux en chinois sur l'utilisation de statistiques seules consiste à mesurer l'information mutuelle (*mutual information*) entre deux caractères [Sproat et Shih, 1990]. L'information mutuelle entre deux caractères x et y est donnée par :

$$I(x : y) = log_2 \frac{p(x,y)}{p(x)p(y)}$$

où $p(x)$ est la probabilité d'occurrence de x , $p(y)$ la probabilité d'occurrence de y et $p(x, y)$ la probabilité d'occurrence des deux caractères à la suite dans un texte [Church *et al.*, 1991]. Appliquée au chinois, cette mesure détermine si deux caractères doivent être séparés (leur information mutuelle est faible, voire négative) ou groupés (leur information mutuelle est forte). Cependant, seuls les mots de deux caractères étaient ainsi reconnus, ce qui est une vision trop simpliste de la langue.

Dans [Kashioka *et al.*, 1998], l'information mutuelle est utilisée pour créer des arbres de décision guidant la segmentation. Dans [Sun *et al.*, 1998], une extension à cette méthode est proposée, introduisant de nouvelles mesures qui prennent en compte le contexte des caractères. En effet, quand l'information mutuelle n'est pas assez élevée, il est difficile de décider si les deux caractères doivent être séparés ou groupés. Au lieu de calculer uniquement l'information mutuelle entre deux caractères dans l'absolu, celle-ci est calculée sur une fenêtre de quatre caractères (soit un caractère avant x et un caractère après y). La segmentation se fait en parcourant le texte de gauche à droite ; à chaque limite entre deux caractères, différentes mesures de plus en plus complexes (en commençant par l'information mutuelle) sont essayées jusqu'à ce qu'un résultat clair permette de trancher sur la séparation ou non des deux caractères. Cette approche a l'avantage d'être purement automatique une fois qu'un corpus est disponible.

Une autre méthode plus complexe [Li et Wang, 1995] propose de considérer toutes les sous-chaînes

possibles d'une chaîne de caractères à traiter et d'extraire incrémentalement les mots les plus « probables ». La pondération d'un mot w est donnée par la formule :

$$P(w) = \begin{cases} F(W)L(w)^c & si\ F(w) > min\ et\ L(w) > D \\ 0 & sinon \end{cases}$$

où $F(w)$ est la fréquence d'occurrence de w dans le corpus, $L(w)$ sa longueur, et c , min et D sont des constantes entières positives qui permettent de paramétrer le comportement du système. Le découpage d'une chaîne se fait en identifiant les sous-chaînes ayant la meilleure pondération comme des mots ; les mots les moins fréquents sont identifiés « par défaut », une fois que les mots les plus courants l'ont été.

Bien que peu utilisées dans la pratique, les méthodes purement statistiques s'avèrent payantes pour le traitement des mots inconnus [Hu et Yu, 2000; Shinnou et Ikeya, 2000] (voir aussi section 2.2.2), ou l'enrichissement de lexiques existants à partir d'un corpus, justement car elles ne reposent pas sur des dictionnaires.

2.2.2 Analyse morphologique du chinois par transducteurs d'états finis pondérés

Dans [Sproat *et al.*, 1996], l'analyse morphologique du chinois est vue comme « un problème de transduction stochastique. » Ils se proposent naturellement de le résoudre à l'aide de transducteurs d'états finis pondérés. À l'aide d'un dictionnaire représenté par un tel transducteur, on peut identifier les mots dans une suite de caractères et les lemmatiser, associer un poids à chaque mot analysé, et obtenir finalement la séquence de mots ayant le meilleur coût total.

2.2.2.1 Une analyse pour la synthèse de parole est différente d'une analyse pour la traduction automatique

Ce système développé chez AT&T, et donc fondé sur les transducteurs d'états finis vus dans le chapitre précédent, est destiné à la génération de parole, qui est ici une sorte de translittération : à partir du texte source en chinois (composé d'une suite de *hanzi*), il s'agit de produire une séquence en *pinyin* qui sera ensuite « lue » par un système de génération de parole. L'itémisation est ici importante pour deux raisons : la première est qu'il est nécessaire d'identifier les limites phonologiques de chaque mot car elles influent sur les accentuations, les tons et la prosodie ; la seconde est que la prononciation exacte de chaque caractère peut différer selon le mot dans lequel ce caractère apparaît. La lecture du texte caractère par caractère ne pourrait pas prendre en compte ces informations et produirait un résultat inadapté.

Ces critères illustrent bien pourquoi le domaine d'application d'un analyseur est important : dans le cadre de la génération de texte, en présence d'une construction complexe, il faut bien segmenter les différents composants de l'expression pour la prononcer de manière intelligible. En revanche, dans le cadre de la traduction automatique, une traduction littérale serait jugée maladroite ou incorrecte car elle ne correspondrait pas à la traduction naturelle de l'expression. L'article cite l'exemple *zhong1-hua2 ren2-min2 gong4-he2-guo2*, littéralement *Chine peuple république*, qui devra être lu en séparant les trois mots, mais être traduit « République Populaire de Chine » : une seule unité pour la traduction [Wu, 1997] ou la recherche d'information [Wu et Tseng, 1993], mais trois en synthèse de parole. Cependant, pour des systèmes ayant un plus large champ d'application, des « guides » de segmentation existent [Liu *et al.*, 1994; Huang *et al.*, 1997].

2.2.2.2 L'algorithme d'analyse morphologique par unigrammes

Le système repose sur le dictionnaire, qui comporte pour chaque entrée sa prononciation, une étiquette syntaxique et une estimation du coût de l'entrée. Cette estimation C dépend du logarithme de la fréquence d'apparition du mot f dans un corpus de N mots :

$C = -log(\frac{f}{N})$

La représentation du dictionnaire par un WFST associe un chemin du transducteur à chaque entrée. Une transition correspond en surface à un *hanzi* et au niveau lexical (qui sert ici à la prononciation) à sa transcription en *pinyin* (qui est fonction de l'entrée). Le coût d'une telle transition est nul. Enfin, la dernière transition de chaque entrée est vide au niveau de surface mais contient l'étiquette syntaxique au niveau lexical et est pondérée par le coût de l'entrée calculé plus haut. Cette représentation illustrée par la figure 2.2 est proche de celle de Xerox produite par `lexc`, la pondération en plus.

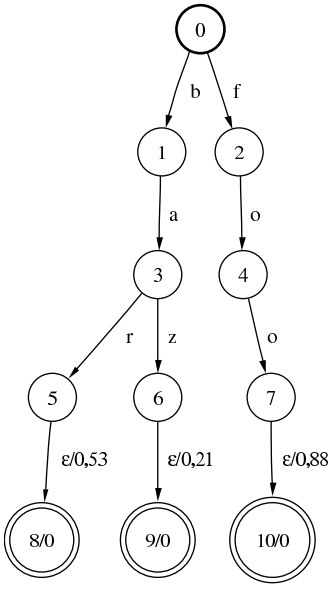

FIG. 2.2 – Représentation du dictionnaire de segmentation par un transducteur d'états finis pondérés

Le transducteur dictionnairique final est complété par tous les *hanzi* disponibles en Big5 (avec leur prononciation) ainsi que par les autres caractères susceptibles d'apparaître dans un texte en chinois. Chacune de ces entrée est identifiée comme « mot inconnu ».

L'algorithme d'analyse est le suivant : soit D le dictionnaire construit ci-dessus. Soit un texte en entrée constitué d'une suite de caractères. Le texte est converti en un automate d'états finis I où chaque transition correspond à un caractère. I contient un unique chemin qui est le texte en entrée. L'identification de tous les mots du texte s'effectue simplement par composition de I avec $D*$. Le WFST ainsi obtenu contient toutes les segmentations possibles selon le dictionnaire ; la meilleure analyse est donnée par le meilleur chemin (celui de coût total inférieur) dans ce transducteur.

2.2.2.3 La gestion des mots inconnus

Le système d'AT&T distingue trois sortes de mots inconnus : la première est celle des mots dérivés (analyse morphologique dérivationnelle), la deuxième celle des noms propres chinois, et la troisième celle des mots étrangers retranscrits à l'aide de caractères chinois.

Tout comme le lexique et le texte sont représentés par des machines d'états finis pondérés, les ressources supplémentaires nécessaires pour l'analyse des mots absents du lexique sont elles aussi compilées sous forme de transducteurs d'états finis suivant le modèle du lexique : aux mots sont associés leur prononciation et leur coût pour la désambiguïsation. Les nouveaux transducteurs ainsi

créés sont ajoutés au transducteur lexical D ; puis la fermeture transitive du résultat est calculée et c'est ce transducteur qui est effectivement appliqué lors de l'analyse (cf. section 2.2.2.2).

Analyse des mots inconnus. La première catégorie de mots n'apparaissant pas dans le dictionnaire est celle des mots qui peuvent être générés par quelques règles morphologiques très simples. Le chinois n'a pas un système de flexions aussi développé que le français et le système ne considère que des règles comme celle de la pluralisation des noms avec le suffixe *men2*. Ces règles morphologiques sont semblables à celles décrites dans le chapitre précédent dans le cadre de l'analyse morphologique à deux niveaux.

Il y a cependant une difficulté supplémentaire : quel est le coût des nouveaux mots ainsi formés ? Dans le cas des mots nouvellement formés présents dans le corpus (mais absents du dictionnaire), on peut simplement calculer le coût à partir du nombre d'occurrences dans le corpus comme pour un mot du dictionnaire. Dans le cas d'un mot absent du corpus, c'est l'estimation Good-Turing qui donne une mesure au mot nouvellement formé [Gale et Sampson, 1995].

Les noms de personnes en chinois. Évidemment, le dictionnaire ne peut contenir tous les noms de personnes chinois qui puissent être rencontrés dans les textes (même si des célébrités peuvent être présentes). De même que les règles de dérivations employées par l'analyse morphologique sont simples, des règles de formation de nom de personnes chinois sont définies.

Un nom chinois est de la forme *nom de famille + prénom*. Le nom de famille se compose d'un ou deux caractères, mais il n'existe que très peu de noms de famille en Chine : seulement quelques centaines longs de un caractère, et une dizaine longs de deux caractères. Le prénom est généralement long de deux caractères, plus rarement de un ; contrairement au nom de famille, ils peuvent être composés de n'importes quels *hanzi*. Si un nom de famille est facile à détecter (même s'il peut y avoir des ambiguïtés avec un nom commun), le cas des prénoms est plus délicat.

La méthode de Sproat et al., fondée sur [Chang *et al.*, 1992], utilise une liste de noms chinois (longue d'environ 17 000 noms différents) pour bâtir un modèle statistique similaire au dictionnaire de mots courants. On peut ainsi donner un coût plus ou moins élevé à un prénom potentiel selon que les caractères qui le composent sont adaptés. Des améliorations sont apportées au modèle initial, par exemple en prenant en compte la signification des caractères du prénom : un caractère qui fait partie d'une classe sémantique « favorable » (*e.g.or*, *jade* ou *herbe*) sera nettement plus probable qu'un caractères d'une classe défavorable (comme *rat* ou *mort*).

La transcription des noms étrangers. Les noms étrangers sont retranscris plus ou moins phonétiquement en choisissant des caractères dont la prononciation est proche du son du mot original. Comme pour les noms propres vus plus haut, n'importe quelle séquence de caractère pourrait *a priori* représenter la transcription d'un mot étranger ; mais comme pour les prénoms, quelques centaines de caractères sont très fréquemment employés et peuvent indiquer qu'une séquence de *hanzi* les contenant a une forte probabilité d'être la transcription d'un mot ou d'une expression d'origine étrangère.

2.2.3 Analyse morphologique du japonais par bigrammes

Deux systèmes d'analyse morphologiques très importants du japonais sont Juman[1] et Chasen[2]. Le premier a été développé à l'Université de Kyoto sous l'égide de Yuji Matsumoto et Sadao Kurohashi, mais ne semble plus évoluer depuis 1999. Juman est en quelque sorte son successeur, développé par Matsumoto entre autres au Nara Institute of Science and Technology. On peut voir Chasen comme une évolution de Juman, et une étude [Tomokiyo et Quint, 2000] montre que Chasen est clairement le plus performant des deux systèmes (beaucoup plus rapide, plus flexible et avec une meilleure couverture de la langue), aussi on s'intéressera ici à Chasen.

[1] `http://www-nagao.kuee.kyoto-u.ac.jp/nl-resource/juman-e.html`
[2] `http://chasen.aist-nara.ac.jp/`

2.2.3.1 Présentation et utilisation de Chasen

Chasen [Matsumoto *et al.*, 1999] est un analyseur morphologique qui segmente un texte en japonais en une séquence de morphèmes et étiquette chaque morphème avec ses parties du discours et sa prononciation. Chasen est fondé sur un modèle hybride qui utilise d'une part des informations linguistiques, sous la forme de dictionnaires et de grammaires ; et d'autre part, des informations statistiques. Ces informations statistiques concernent les morphèmes (chaque morphème se voit attribuer un coût propre) et la probabilité que deux morphèmes se suivent dans un texte. Ces coûts sont calculés automatiquement à partir d'un corpus d'articles de journaux japonais. On décrit l'algorithme d'analyse morphologique et les ressources utilisées plus bas.

La figure 2.3 montre l'analyse d'une phrase extraite d'un corpus du laboratoire NTT destiné à l'évaluation de systèmes de traduction automatique avec Chasen. La phrase se lit « *watashi wa ringo o taberu* », ce qui signifie « je mange une pomme. » La segmentation et l'analyse proposée est la suivante :

```
houdini chasen <1> cat phrase
私はりんごを食べる。
houdini chasen <2> chasen phrase
私      ワタシ  私      名詞-代名詞-一般
は      ハ      は      助詞-係助詞
りんご  リンゴ  りんご  名詞-一般
を      ヲ      を      助詞-格助詞-一般
食べる  タベル  食べる  動詞-自立        一段    基本形
。      。      。      記号-句点
EOS
houdini chasen <3>
```

Fig. 2.3 – Exemple d'analyse morphologique avec Chasen

Le format de sortie est pour chaque ligne : forme de surface (les mots tels qu'ils apparaissent dans le texte original), prononciation (en *katakana*), forme de base du morphème, partie du discours, et éventuellement informations supplémentaires (ici, pour le verbe). La solution proposée (correcte) est reprise dans la table 2.1.

watashi	je	nom-pronom-général
wa		particule-connexion
ringo	pomme	nom-général
o		particule-cas-général
taberu	mange	verbe-indépendant, général, forme de base
.	.	symbole-ponctuation

Tab. 2.1 – Détails de l'analyse morphologique

Le système est très souple : il est possible de modifier et d'ajouter des dictionnaires (on peut ajouter des lexiques pour augmenter la couverture dans un domaine spécialisé) et d'adapter le format de sortie à ses besoins. Si le comportement habituel de Chasen est de choisir, pour chaque phrase, la solution dont le coût total est optimal (le plus faible possible), donc la plus probable, il est possible

d'afficher plusieurs résultats s'il y a plus d'une analyse de coût minimum, mais également d'afficher toutes les analyses, quel que soit leur coût. On dispose alors d'une segmentation exhaustive du texte. Des options de nature plus cosmétique permettent de configurer la présentation du résultat, en ajoutant ou omettant des informations, ou en choisissant un format de présentation différent (par exemple sous forme Prolog).

2.2.3.2 Dictionnaires et grammaire

On reprend la documentation de Chasen pour décrire les différentes ressources utilisées, qui sont résumées dans la table 2.2.

	Dictionnaires	Grammaires
Fichiers externes	Définitions des morphèmes Dictionnaires de morphèmes	Définitions grammaticales Parties du discours Types de conjugaison Formes de conjugaison Règles de connectivité
Fichiers système	Dictionnaires systèmes Fichiers d'index	Table de connectivité Matrice de connectivité

Tab. 2.2 – Ressources linguistiques de Chasen

Les fichiers de ressources les plus intéressants sont :

- les dictionnaires de morphèmes, qui définissent les morphèmes de chaque partie du discours (par exemple, `Noun.dic` est le dictionnaire des noms). Les informations pour un morphème sont :
  - sa forme de surface (forme de base dans le cas d'un verbe) ;
  - sa prononciation (en *hiragana*) ;
  - son type de conjugaison (le cas échéant) ;
  - des informations sémantiques ;
  - un coût pour la forme de surface.
- les fichiers de définition des parties du discours décrivent les différentes parties du discours disponibles, qui sont hiérarchiques (ainsi, les catégories nom, adjectif, verbes, etc. peuvent être spécialisées) ;
- les fichiers de règles de connectivité définissent le coût de connectivité pour chaque couple de parties du discours.

Pour plus de détails, se reporter à la documentation de Chasen et consulter les fichiers en question qui sont fournis avec la distribution du logiciel (on peut d'ailleurs noter que sous sa forme actuelle, Chasen et ses dictionnaires sont disponibles séparément).

2.2.3.3 L'algorithme d'analyse morphologique par bigrammes

La première étape d'analyse de Chasen est une segmentation de l'entrée en phrases, qui s'opère simplement selon les marques de fin de phrase (point final, etc.) La phrase, vue comme une chaîne de caractères, est l'unité d'analyse de Chasen.

Ensuite, c'est une segmentation exhaustive de la phrase qui a lieu, c'est-à-dire que toutes les sous-chaînes correspondant à des morphèmes valides selon les dictionnaires sont identifiées. Pour chaque morphème, on calcule un coût qui est le produit :

- du coût de la forme de surface du morphème (cette mesure provient du dictionnaire) ;
- du coût de la partie du discours du morphème (cette mesure est définie pour chaque partie du discours dans un fichier de ressource global de Chasen) ;
- et du poids relatif des coûts des morphèmes (définie également dans le fichier de ressource global).

Pour chaque bigramme des parties du discours des morphèmes, on calcule également le coût de connectivité qui est le produit :

- du coût de connectivité (défini dans le fichier des règles de connectivité) ;
- et du poids relatif des coûts de connectivité (défini dans le fichier de ressource global).

Chaque morphème a donc un coût qui lui est propre, tout comme la connectivité de deux morphèmes, c'est-à-dire la probabilité qu'ils se suivent dans un texte linguistiquement correct, a également un coût. Le coût de connectivité est établi selon les parties du discours et non pas selon les morphèmes eux-mêmes, afin d'éviter le problème de la masse de données nécessaire pour établir ces coûts. Le poids total d'une analyse est simplement la somme des coûts de tous les morphèmes qui la composent et des coûts de connectivité pour chaque couple de morphèmes. La meilleure solution est alors celle dont le coût total est le moins élevé, et qui est probablement la plus correcte linguistiquement. Comme on l'a vu dans les options d'utilisation de Chasen, plusieurs solutions de coût minimal peuvent être proposées, mais aussi toutes les solutions possibles.

La gestion des mots inconnus. Encore une fois, le problème de la couverture du dictionnaire se pose : comment gérer les mots inconnus, qui non seulement n'apparaissent pas dans le dictionnaire (et ne sont pas reconnus lors de la phase d'analyse), mais surtout qui n'ont pas de coûts associé ? Il y a donc deux problèmes à résoudre : le premier est de trouver, lorsque l'on rencontre une sous-chaîne inconnue, quel est le découpage le plus probable ; reste ensuite à assigner un coût à chaque morphème ainsi obtenu, et à chaque bigramme.

Le découpage est considéré différemment selon la nature des caractères de la sous-chaîne. Ainsi, les caractères japonais comme les *kanji* et les *hiragana* ainsi que les symboles seront segmentés caractère par caractère ; par contre, les *katakana* (employés principalement pour les mots empruntés aux langues étrangères) ou les caractères romains seront eux segmentés en choisissant la plus longue séquence possible. Chaque séquence ainsi reconnue est ensuite munie de l'étiquette « mot inconnu » (on ne cherche pas à deviner quelle pourrait être la catégorie morphosyntaxique). Enfin, on donne également un coût très élevé à un morphème inconnu ; quant au coût de connectivité entre deux morphèmes dont au moins un est inconnu, celui-ci est également unique et très élevé.

2.3 Autres pratiques de l'analyse présyntaxique

2.3.1 Prétraitement et analyse morphologique en une seule étape

Il a jusqu'à présent beaucoup été question de segmentation ou de lemmatisation, mais il ne faut pas oublier que l'analyse présyntaxique englobe également toutes les opérations de prétraitement préalables à l'analyse, comme le nettoyage ou la normalisation du texte. Le système SMorph [Aït-Mokhtar, 1998] propose de traiter toutes les phases de l'analyse présyntaxique, jusqu'à la lemmatisation, en une seule étape.

2.3.1.1 La spécification des données linguistiques

L'originalité des dictionnaires de SMorph est qu'ils contiennent, en plus des informations linguistiques classiques, les informations typographiques nécessaires pour la normalisation du texte et le traitement des différentes variations que du texte courant peut présenter. Aussi, du point de vue la

morphologie, le dictionnaire se contente de lister exhaustivement les formes fléchies des mots lexicalisés, SMorph ne définissant pas de modèle morphologique.

Les entrées sont spécifiées sous la forme d'une expression régulière, suivie du lemme et des traits morphosyntaxiques correspondants. Certaines formes peuvent n'être reconnues que si elles sont toujours suivies ou précédées d'un caractère non-séparateur (préfixes, suffixes), si elles commencent par une majuscule, etc. (informations typographiques).

Exemple : une entrée classique est de la forme :

```
chantions chanter /v/2/pl/sub/pre .
```

qui associe au mot « chantions » le lemme « chanter » et divers traits morphosyntaxiques (verbe, deuxième personne, pluriel, subjonctif, présent).

Plus intéressant, en plus de lister des formes, le dictionnaire définit des classes de séparateurs et les potentiels typographiques des caractères de l'alphabet. Les classes de séparateurs définissent les caractères d'espacement (blanc, tabulation, retour à la ligne, blanc dur) et les autres types de séparateurs (ponctuation, parenthèses, symboles divers). Les potentiels des caractères sont, pour chaque caractère, les différentes formes que celui-ci peut prendre, en prenant en compte la casse (majuscule ou minuscule) et des accents.

Exemple : le potentiel de `A` est `{À, Â, a, à, â}`, celui de `a` est vide (un `a` dans le texte n'est toujours qu'un `a`).

2.3.1.2 La segmentation et la lemmatisation

La reconnaissance du texte se fait de gauche à droite, en cherchant à reconnaître les formes définies dans le dictionnaire. Pour qu'un préfixe du texte analysé soit reconnu, celui-ci doit être une variante typographique possible d'une forme du dictionnaire et doit satisfaire les contraintes typographiques de cette forme (pour les préfixes, suffixes, etc.) Une forme peut contenir des espacements ; dans le dictionnaire, ceux-ci sont spécifiés par le caractère _, dans le texte, toute séquence d'au moins un caractère d'espacement est reconnue comme un blanc dans la forme. Enfin, les césures (traits d'union en fin de ligne) sont également prises en compte.

Le découpage s'effectue en suivant le principe de la plus longue séquence : comme dans plusieurs systèmes vus plus haut, c'est cette heuristique qui permet de résoudre d'éventuelles ambiguïtés de découpage. Il est cependant possible de déclarer une forme comme étant ambiguë, auquel cas plusieurs découpages sont finalement proposés.

Exemple : la chaîne « Rendez-vous à Marseille » peut être segmentée en « Rendez-vous / à / Marseille » ou « Rendez / -vous / à / Marseille », si la forme « -vous » est définie dans le dictionnaire (comme pronom clitique), et « rendez-vous » est déclaré comme une forme ambiguë (sinon, c'est la première forme seule qui est choisie).

Vu la forme des entrées de dictionnaires, la lemmatisation pour Smorph consiste simplement à extraire du dictionnaire les lemmes et informations morphosyntaxiques de chaque forme segmentée. L'analyse produit un résultat sous forme d'une liste de mots, avec parfois plusieurs segmentations possibles, et pour chaque mot, la ou les analyses possibles.

2.3.1.3 Les modèles de formes inconnues

Les modèles de formes inconnues sont des patrons beaucoup plus généraux que les formes lexicales définies dans le dictionnaire. Ces patrons ont la même forme que les entrées du dictionnaires mais sont définis séparément.

Exemple :

```
@Min+ âge /n/s/m .
```

> est une forme inconnue composée d'au moins une lettre minuscule (l'expression `@Min+`) et se terminant par *-age*. Tout mot de cette forme est considéré comme un nom masculin singulier.

En cas d'échec du découpage, c'est-à-dire si aucune forme du dictionnaire n'est reconnue, l'analyse reprend au même point en utilisant cette fois-ci les modèles de formes inconnues (principe du devineur). En cas de nouvel échec, tout dépend du premier caractère de la chaîne analysée : si c'est un espacement, tous les espacements suivants sont sautés pour reprendre l'analyse au premier caractère non-blanc de la chaîne. Si ce n'est pas un espacement, le plus long préfixe ne contenant pas de séparateur est reconnu comme un mot et hérite de l'étiquette particulière des formes non reconnues.

2.3.2 La segmentation en phrases

De même que la segmentation en mots ne peut pas se faire uniquement en suivant les séparateurs apparents entre chaque mot, la segmentation en phrases ne peut pas non plus se faire uniquement en suivant la ponctuation. Le point en particulier est fortement ambigu aussi bien en français qu'en anglais ; la section suivante montre que le problème existe aussi dans des langues comme le thaï.

2.3.2.1 Traitement des abréviations avec des expressions régulières

Dans [Grefenstette et Tapanainen, 1994], les coupables sont clairement les abréviations (en anglais, en tout cas). En étudiant le corpus de Brown [Francis et Kucera, 1982], on se rend compte qu'environ 10 % des points ne marquent pas une fin de phrase. Il est donc proposé plusieurs solutions pour traiter les ambiguïtés de segmentation causées par les abréviations : premièrement, en définissant quelques expressions régulières avec `lex`, comme :

```
[A-Za-z]\.
[A-Za-z]\.([A-Za-z0-9]\.)+
[A-Z][bcdfghj-np-tvxz]+\.
```

qui prennent en compte un très grand nombre d'abréviations, introduisent quelques erreurs, et échouent pour certaines catégories (notamment les titres, comme *Sen.*, *Gov.*, ou les unités de mesure, comme *400-lb.* ou *10-yr.*) La deuxième approche est d'utiliser un filtre sur le corpus pour valider les expressions décrivant des abréviations potentielles. La troisième approche utilise un lexique, mais qui ne contient pas d'abréviations, celles-ci étant détectées par une cascade de règles comme

> si un mot suivi d'un point est en minuscules et si le même mot existe dans le lexique, il ne s'agit pas d'une abréviation.

Et enfin, la quatrième approche utilise un lexique avec nombre d'abréviations empruntées au corpus de référence.

2.3.2.2 Reconnaissance par automates d'états finis avec INTEX

La reconnaissance des phrases dans le système INTEX est un constituant de l'analyse morphologique du français. Celle-ci se fait évidemment par transducteurs d'états finis ; pour aller au plus simple, le segmenteur en phrases a pour rôle d'insérer un symbole {S} de fin de phrase à la fin de chaque phrase. Dans [Friburger *et al.*, 2000], deux types d'erreurs sont répertoriées : le bruit et le silence. Le bruit se produit si trop de séparateurs de phrases sont introduits (donc les phrases sont trop courtes), et le silence quand trop peu de séparateurs sont introduits (les phrases sont trop longues). Le découpage favorise le silence par rapport au bruit, car des phrases trop courtes compromettent l'identification de certains motifs qui ne sont reconnus qu'à l'intérieur d'une phrase.

En français, le point aussi est ambigu : il apparaît dans les motifs anthroponymiques (*M.* pour *monsieur*, abréviation des prénoms), dans les sigles (même si désormais on n'utilise plus les points

dans les sigles et acronymes en français), dans des symboles et noms composés avec des majuscules et dans diverses abréviations. Mais le point n'est pas le seul signe de ponctuation considéré dans le découpage en phrases :

- les parenthèses, crochets et tirets : aucun séparateur n'est introduit à l'intérieur de parenthèses pour ne pas séparer la phrase qui les contient ;
- les points de suspension, qui terminent une énumération, mais pas forcément une phrase ;
- les guillemets ne sont pas traités spécialement (INTEX utilise des *double quotes* anglais en lieu et place de guillemets français, il n'y a donc pas de différence entre ouverture ou fermeture des guillemets) ;
- les deux points qui peuvent précéder une citation et constituent alors une fin de phrase.

Pour chaque catégorie de problème, un transducteur particulier est défini, comme pour le traitement du point ou des parenthèses.

2.3.2.3 Des systèmes de plus en plus perfectionnés

Dans [Palmer et Hearst, 1997] est décrit un système d'analyse présyntaxique assez complet, Satz, dans le but de segmenter correctement le texte en phrases. Il s'agit là d'un véritable analyseur présyntaxique, qui comprend quatre étapes :

1. l'itémisation, réalisée avec `lex` ;
2. l'étiquetage syntaxique ;
3. la construction des tableaux de descripteurs ;
4. la classification par algorithme d'apprentissage.

L'étape de l'étiquetage syntaxique suit celle d'itémisation et associe à chaque item une ou plusieurs étiquettes syntaxiques provenant d'un lexique. Celles-ci s'accompagnent d'une probabilité dont la somme pour un item donné est 1. Satz peut également fonctionner sans pondération, chaque étiquette ayant alors le même poids. L'itémisation produit des formes qui n'appartiennent pas au lexique, et pour lesquelles des étiquettes doivent tout de même être appliquées. Un devineur utilisant quelques heuristiques (présence de chiffres, de signes de ponctuation, de majuscules, etc.) est alors employé.

La construction des tableaux de descripteurs simplifie les catégories syntaxiques (*e.g.*, une étiquette complexe telle que « participe passé » ou « verbe modal » est simplifiée en « verbe »), puis enfin la classification de chaque item est effectuée en prenant en compte un contexte de $k + 1$ items ($k/2$ items avant et après l'item considéré), appelé le k-contexte. Dans le cas d'un signe de ponctuation, l'algorithme d'apprentissage utilise le k-contexte pour déterminer son rôle exact. Deux algorithmes d'apprentissage sont présentés, l'un utilisant un réseau neuronal, et l'autre un arbre de décision.

Enfin, le principe d'apprentissage de Satz étant indépendant de la langue, d'autres langues (allemand, français) ont été traitées en plus de l'anglais, avec des résultats similaires à ceux de l'anglais.

2.3.3 La segmentation du thaï

On a vu la segmentation en phrases dans la section précédente, mais il y a des langues ou la distinction des phrases est beaucoup moins marquées. Le thaï en est un excellent exemple.

Dans la plupart des systèmes d'analyse présyntaxique en thaï, la distinction entre phrases ne se fait pas. Le texte est traité par paragraphes entiers, ou par segments qui se situent entre la phrase et le syntagme, délimité par des espacements. Ces espacements ne délimitent pas toujours des phrases, mais également des nombres, des clauses, les éléments d'une énumération, etc.

L'approche la plus courante est donc l'itémisation directe, sans passer par une segmentation en phrases. Elle n'est en réalité pas surprenante, car c'est également cette approche qui est choisie majoritairement même dans les langues ou la segmentation en phrases n'est pas aussi problématique.

Les multiples solutions pour l'itémisation en thaï sont désormais familières [Sornlertlamvanich *et al.*, 2000b] : heuristiques de plus longue chaîne, heuristique d'itémisation minimale, techniques probabilistes, etc. Des systèmes assez sophistiqués ont été développés, comme par exemple [Meknavin *et al.*, 1997], où des méthodes d'apprentissage de règles de résolution d'ambiguïtés de segmentation sont appliquées pour améliorer le résultat des méthodes précédentes. Les méthodes d'apprentissage telles que l'algorithme C4.5 sont également employées par [Sornlertlamvanich *et al.*, 2000a].

La segmentation en phrases. L'espace comme séparateur est ambigu au même titre que la ponctuation en français ou en anglais. Dans [Mittrapiyanuruk et Sornlertlamvanich, 2000], un algorithme de classification des espacements entre séparateur de phrases ou non est proposé. Le problème est reformulé en terme de l'étiquetage syntaxique (*part-of-speech tagging*) : si un blanc peut avoir deux étiquettes, SBS (*sentence-breaking space*, séparateur de phrases) ou *NSBS* (*non-sentence-breaking space*, non-séparateur de phrases), alors la distinction des phrases consiste simplement à identifier correctement la catégorie syntaxique d'un blanc dans le texte.

La segmentation en phrases repose sur une première étape d'itémisation et de lemmatisation. Ensuite, pour chaque couple d'items, un espace virtuel est inséré ; celui-ci peut avoir pour catégorie SBS ou NSBS. C'est au processus d'étiquetage syntaxique (donc de désambiguïsation des étiquettes) de trancher selon un modèle classique de trigrammes. Chaque espace virtuel ou non de catégorie SBS constitue alors un séparateur de phrases.

Conclusion

Encore une fois, les systèmes exposés ici ne sont qu'une partie la plus représentative possible de tout ce qui existe dans le domaine, mais on commence maintenant à avoir une idée plus claire des pratiques actuelles de l'analyse présyntaxique, même si la première idée semble être qu'il y a autant d'approches que de systèmes existants... Cela n'est pas totalement faux, car même si deux analyseurs morphologiques sont réalisés avec le même formalisme, des stratégies parfois fort différentes sont nécessaires selon la langue analysée. Par exemple, le traitement des entités composées dans la morphologie à deux niveaux n'est pas forcément la même en français et en allemand.

Chapitre 3

Une synthèse des travaux actuels en analyse présyntaxique

Introduction

Après avoir essayé de présenter un panorama assez large des travaux actuels dans le domaine de l'analyse présyntaxique, il est utile de donner une version plus synthétique des avancées réalisées jusqu'à présent et surtout du chemin restant à parcourir. Dans cette synthèse, on propose de remettre en perspective les idées les plus intéressantes vues précédemment par un comparatif systématique (section 3.1), ainsi que les problèmes qui sont encore loin d'être résolus (section 3.2).

De cette étude synthétique, on retire des points clés qui se dégagent d'approches d'apparences différentes. L'unification de tous ces outils n'est sans doute pas la meilleure solution pour la construction d'un outil générique destiné à la segmentation et aux applications présyntaxiques, mais en définissant un sous-ensemble de caractéristiques et de fonctionnalités propres à de nombreux outils (mécanismes fondés sur les états finis, possibilité d'utiliser des statistiques, mécanismes d'extension, etc.), on est en mesure de définir une sorte de cahier des charges pour un formalisme réellement universel, donc dont les fondements soient complètement indépendants d'une langue ou d'une autre (section 3.3).

3.1 Comparatif systématique des approches présentées

Cette section consiste en un comparatif systématique de `lex`, Segdict, SMorph, Satz, XFST, ATEF, les systèmes-Q, FSM et Chasen selon les critère suivants.

Niveaux de segmentation. Quel est le niveau des unités traitées ? Y a-t-il plusieurs niveaux de traitements ?

Traitement des ambiguïtés. Comment sont résolues les ambiguïtés de segmentation et d'analyse ? Sont-elles toutes conservées ou sont-elles résolues ? Si oui, comment se fait cette résolution ?

Formats d'entrée et de sortie. Sous quelle forme est attendu le texte à évaluer ? Et surtout, quelle est la forme du résultat de l'analyse ?

Modèle sous-jacent. Quelles sont les techniques qui gouvernent l'analyse ?

Programmabilité. Quelle est la latitude de l'auteur pour modifier le comportement du système avec lequel il écrit un analyseur ?

Réversibilité. Si l'on a surtout parlé d'analyse jusqu'à présent, la génération (dans le domaine de la morphologie par exemple) n'est pas à négliger. Un système est-il utilisable dans les deux sens ? Y a-t-il beaucoup de modifications à faire pour que ce soit le cas ?

Traitement des mots inconnus. Comme les ressources linguistiques disponibles sont forcément insuffisantes pour des applications robustes, comment gérer la présence de formes inconnues dans le texte à analyser ?

Langues couvertes. Pour quelle(s) langue(s) ont été écrits des analyseurs selon cette méthode ? Y a-t-il des aspects techniques, théoriques ou pratiques, qui empêcheraient ou faciliteraient l'écriture d'analyseurs dans d'autres langues ?

Ces critères sont appliqués aussi bien aux outils pour programmeur (`lex`, section 3.1.1), qu'aux systèmes génériques (comme Segdict, section 3.1.2 ou ATEF, section 3.1.6), qu'aux analyseurs spécifiques (comme l'analyseur du chinois par unigrammes section 3.1.8 ou l'analyseur du japonais par bigrammes Chasen, section 3.1.9). Ainsi, une comparaison devient plus aisée.

Cependant, il semble encore manquer un point important : il n'a encore pratiquement pas été question d'évaluation. Quelles sont les performances de ces analyseurs ? Quel est celui qui donne les meilleurs résultats ? Quelle est la technique la plus précise ? L'utilisation de statistiques donne-t-elle de meilleurs résultats que des heuristiques ? Comme on le verra un peu plus bas dans la section 3.2.3, l'évaluation d'un segmenteur ou d'un analyseur morphologique est une tâche particulièrement ardue, et la comparaison quantitative de plusieurs systèmes est quasiment impossible. Aussi, ce sont uniquement des critères qualitatifs qui sont appliqués ici.

3.1.1 Les caractéristiques de lex

Niveaux de segmentation. Un seul, celui des lexèmes.

Traitement des ambiguïtés. Tous les motifs sont essayés ; parmi tous les motifs qui reconnaissent un préfixe de la chaîne analysée, c'est celui qui reconnaît la plus longue chaîne qui est choisi. Si plusieurs motifs reconnaissent une chaîne de même longueur, la priorité est donné à celui qui est défini en premier dans le fichier source.

Formats d'entrée et de sortie. `lex` n'a pas de format de sortie à proprement parler puisqu'il ne s'agit que d'un outil pour programmer un analyseur. C'est donc l'application cible qui définit la sortie de l'analyse. Idem pour l'entrée ; c'est à l'application de se charger de l'acquisition du texte.

Modèle sous-jacent. Chaque motif est compilé sous la forme d'un automate d'états finis. Les automates sont appliqués successivement au texte à analyser. Les ajouts de `lex` à la syntaxe des expressions régulières est d'ordre purement pratique ; l'expressivité d'un motif n'en est pas changée.

Programmabilité. Une fois un motif reconnu, une action quelconque en C est entreprise. Ces actions peuvent modifier la reconnaissance, par exemple en rejetant un motif qui a été reconnu, ou en modifiant le texte qui reste à analyser. Une grande latitude est ainsi laissée au programmeur.

Réversibilité. Néant.

Traitement des formes inconnues. Un préfixe impossible à reconnaître cause un arrêt immédiat du reconnaisseur. Des motifs généraux doivent être prévus pour éviter cette situation.

Langues couvertes. En réalité, il n'y a que peu d'exemples d'utilisation de `lex` pour l'analyse de langues naturelles, et `lex` est essentiellement utilisé pour l'analyse de langages informatiques et artificiels. [Grefenstette et Tapanainen, 1994] montre cependant quelques exemples d'utilisation pour des étapes préliminaires à l'analyse morphologique.

Références. Manuels de programmation, comme celui de `flex`[1].

[1] `http://www.gnu.org/software/flex/`

3.1.2 Caractéristiques de Segdict

Niveaux de segmentation. Segdict effectue une première segmentation du texte en paragraphes, puis en mots, mais cette segmentation ne se retrouve pas dans le résultat final. Le traitement du texte paragraphe par paragraphe évite simplement d'avoir à stocker tout le texte en mémoire, et d'obtenir un résultat dès qu'un morceau de texte est disponible. Il n'y a donc au final qu'un unique niveau de segmentation.

Traitement des ambiguïtés. La sélection du motif appliqué dépend de l'ordre de définition dans le dictionnaire de segmentation. Pour un motif donné, c'est normalement la plus longue chaîne reconnaissable qui est identifiée (voir plus bas pour les exceptions).

Formats d'entrée et de sortie. L'entrée est un ou plusieurs fichiers de texte brut. La sortie est simplement constituée d'un item par ligne.

Modèle sous-jacent. Le moteur d'expressions régulières de Perl sert à une reconnaissance non-déterministe avec retour arrière. Il permet aussi le *look-ahead* pour consulter la suite de la chaîne à segmenter sans la consommer afin de décider des limites de l'item courant.

Programmabilité. Les extensions de Perl aux expressions régulières permettent de faire référence à une sous-chaîne précédemment reconnue dans un motif, ainsi que d'employer une heuristique de plus courte chaîne à l'intérieur d'un motif. Les actions associées à chaque motif sont totalement arbitraires ce qui donne une plus grande souplesse. L'action de reconnaissance stocke le mot reconnu (et/ou des parties de ce mot) dans les variables spéciales `$&`, `$1`, `$2`, etc. utilisables pour créer le ou les items à afficher en sortie.

Réversibilité. Néant.

Traitement des formes inconnues. Le dictionnaire de segmentation est automatiquement complété par une règle qui identifie un caractère quelconque comme un mot. L'auteur d'un dictionnaire peut le concevoir de sorte que le dictionnaire soit complet, et que cette règle ne s'applique jamais, afin de traiter les mots inconnus à sa guise.

Langues couvertes. Jusqu'à la version 5.6 de Perl, l'écriture d'expressions régulières pour des textes dans un encodage utilisant plus d'un octet par caractère est assez malaisée. La situation s'améliore avec une meilleure intégration d'Unicode dans le langage, qui devrait devenir parfaitement utilisable dans la version 6 du langage.

Un autre obstacle au traitement des langues asiatiques est qu'une simple règle générique pour reconnaître la plupart des mots doit laisser place à une véritable liste de dizaines de milliers de mots. L'approche relativement naïve de Segdict rend alors l'analyse dans ce contexte extrêmement lente.

Références. Section 2.1.1.

3.1.3 Caractéristiques de SMorph

Niveaux de segmentation. Un seul, celui des mots.

Traitement des ambiguïtés. Les ambiguïtés de segmentation sont résolues selon l'heuristique de la plus longue forme, même si certaines ambiguïtés peuvent être volontairement conservées. Les ambiguïtés de lemmatisation pour un mot donné sont toutes conservées.

Formats d'entrée et de sortie. Les textes en entrée sont des fichiers de texte brut en ASCII. La sortie est textuelle, avec un mot et son analyse par ligne. Des symboles particuliers indiquent les ambiguïtés de lemmatisation ou de segmentation.

Modèle sous-jacent. Le dictionnaire et les modèles de formes inconnues sont compilés en un transducteur d'états finis. Ces transducteurs sont utilisés par le moteur d'analyse pour la reconnaissance des formes.

Programmabilité. Certaines informations influencent le comportement du segmenteur en posant des conditions sur la reconnaissance d'une forme (notions de préfixes et suffixes) ou en admettant des ambiguïtés de segmentation. Peu de programmabilité à proprement parler.

Réversibilité. SMorph peut être utilisé aussi bien en analyse qu'en génération, seul change le sens d'application des transducteurs.

Traitement des formes inconnues. Le traitement des mots inconnus se fait en deux étapes, premièrement grâce à un devineur qui prend le relais si un mot est absent du dictionnaire principal, et deuxièmement, en traitant toute suite de caractères non-séparateurs non reconnaissable comme un mot inconnu.

Langues couvertes. Principalement le français, mais des exemples de traitement de phénomènes divers comme les clitiques infixés en portugais, l'accent tonique en espagnol et les césures en allemand sont évoqués.

Références. Décrit par [Aït-Mokhtar, 1998].

3.1.4 Caractéristiques de Satz

Niveaux de segmentation. Un seul, celui des mots. Les signes de ponctuation sont des mots particuliers qui peuvent délimiter des phrases.

Traitement des ambiguïtés. La plupart des ambiguïtés sont résolues statistiquement ou binairement, en suivant des règles décision. Cependant dans certains cas le système décide de ne pas prendre de décision.

Modèle sous-jacent. L'itémisation utilise `lex` ; l'étiquetage syntaxique et la segmentation en phrases utilisent un réseau neuronal ou un arbre de décision dont les règles de segmentation sont générées par apprentissage sur un corpus préparé à cet effet.

Programmabilité. Néant.

Réversibilité. Néant.

Traitement des mots inconnus. À partir des formes reconnues par l'analyse lexicale, le devineur affecte des étiquettes syntaxiques aux mots absents du lexique en utilisant une série d'heuristiques : une séquence de chiffres est un nombre, les mots contenant des tirets sont considérés comme des adjectifs, noms communs ou noms propres, etc.

Langues couvertes. Anglais, allemand et français.

Références. Décrit dans [Palmer et Hearst, 1997].

3.1.5 Caractéristiques de XFST

Niveaux de segmentation. Un seul, celui des lexèmes.

Traitement des ambiguïtés. Le lemmatiseur traite du texte déjà itémisé et conserve toutes les ambiguïtés d'analyse. Il produit pour chaque mot un ou plusieurs lemmes ; c'est ensuite à un processus séparé de désambiguïsation syntaxique de sélectionner la meilleure analyse par le biais de modèles de Markov cachés.

Formats d'entrée et de sortie. Avec `lookup`, l'entrée est constituée d'un texte déjà itémisé, contenant un item par ligne. En sortie, chaque lexème est présenté sur une ou plusieurs lignes selon le nombre d'étiquettes syntaxiques et de lemmes différents trouvés.

Avec `xfst`, ce sont des réseaux d'états finis qui sont directement manipulés. Entrée et sortie de la lemmatisation sont donc des réseaux d'états finis.

Modèle sous-jacent. Machines d'états finis.

Programmabilité. Il y a deux extensions au modèle d'états finis strict. La première est celle des « stratégies » de `lookup`, où plusieurs transducteurs peuvent être essayés successivement jusqu'au succès de l'un d'eux (de la même manière que Segdict). Ce comportement ne peut s'exprimer par un unique transducteur d'états finis.

La seconde extension est celle des *flag diacritics*, que l'on détaille plus bas. Ce sont des symboles spéciaux dont l'interprétation modifie le comportement du reconnaisseur.

Réversibilité. Un même transducteur d'états finis s'applique dans deux directions : de la surface au niveau lexical pour l'analyse, et du niveau lexical à la surface pour la génération.

Traitement des formes inconnues. Dans `xfst`, un mot inconnu est un échec pur et simple du reconnaisseur ; le résultat est nul. Dans `lookup`, l'échec d'un transducteur entraîne l'application du suivant dans la cascade. Dans cette cascade se trouve généralement un « devineur » dont la tâche est justement de proposer une analyse probable du mot inconnu. En désespoir de cause, le mot n'est pas lemmatisé et est retourné tel quel (ou muni d'un étiquette « mot inconnu »).

Langues couvertes. Les langues pour lesquelles il existe des analyseurs morphologiques, développés par Xerox ou par des universités utilisant les outils de Xerox, se comptent par dizaines. La plupart des langues traitées (langues d'Europe occidentale, d'Europe de l'Est, langues scandinaves, etc.) se prêtent particulièrement bien au modèle concaténatif de la morphologie à deux niveaux ; mais l'indonésien ou l'arabe font aussi l'objet de représentations à états finis.

Les langues ayant un encodage sur plusieurs octets sont plus difficiles à traiter ; un problème en particulier est le nombre limité d'étiquettes possibles. Dans un transducteur, chaque couple de symbole qui peut étiqueter un arc a un identificateur qui est propre. Quand les alphabets traités comportent des milliers de caractères, le nombre de couples possibles peut exploser ; on en voit une illustration plus bas.

Références principales. Décrit en détails dans [Beesley et Karttunen, 2002].

3.1.6 Caractéristiques d'ATEF

Niveaux de segmentation. L'analyse morphologique elle-même distingue phrases (ULFRA) et occurrences (ULOCC) ; mais aussi mots-composés (ULCMP), tournures (ULTOURN) et homophrases (ULSOL). Des fonctions d'analyse présyntaxique permettent un découpage plus fin des occurrences, mais a été peu utilisé dans la pratique.

Traitement des ambiguïtés. Les règles permettent de traiter les ambiguïtés d'analyse d'une forme en fonction de son contexte (en prenant en compte les quatre occurrences précédemment analysées). Les ambiguïtés qui ne peuvent pas être résolues par les règles sont conservées.

Formats d'entrée et de sortie. L'entrée est un fichier de texte brut. Plusieurs formes de sorties existent : toutes les ambiguïtés sont conservées dans le graphe quadrichrome arrière, ainsi que dans l'arbre avec homophrases. Les autres formes de sorties (arbre sans homophrases, graphe-Q, etc.) perdent de l'information par rapport au graphe avec homophrases.

Modèle sous-jacent. ATEF réalise une transduction d'états finis non-déterministe. Contrairement aux autres systèmes fondés sur les états finis comme INTEX ou XFST, le transducteur reste virtuel et est simulé par le moteur d'ATEF. En théorie, son nombre d'états (position dans la chaîne analysées, valeurs des différentes variables) et de transitions est immense.

Programmabilité. Les actions des règles de grammaire amènent une grande souplesse, en contrôlant l'arborescence des choix (actions de type -FINAL-, -STOP-, -ARRET-, -ARD-, -ARF- qui éliminent des choix faits ou à faire) et en transformant le texte à analyser (avec TCHAINE).

Réversibilité. Non. Cependant, les données statistiques (variables, formats, désinences, condition) peuvent être aisément inversés pour la génération en SYGMOR (une grammaire exhaustive du français en Sygmor a ainsi été écrite en une page).

Traitement des formes inconnues. La définition de la sous-grammaire du mot inconnu associé au format morphologique obligatoire MODINC permet de définir une grammaire plus ou moins sophistiquée du mot inconnu [Guilbaud et Boitet, 1997]. Cette sous-grammaire contient obligatoirement la règle MOTINC qui garantit la production d'une solution, mais le linguiste peut faire en sorte qu'elle ne soit jamais utilisée en traitant tous les cas par des règles de MODINC placées avant MOTINC.

Langues couvertes. Au moins : français, russe, anglais, allemand [Guilbaud, 1980], anglais, portugais, japonais [Laurent, 1977].

Références principales. En plus de celles citées ci-dessus, [Chauché, 1974].

3.1.7 Caractéristiques des systèmes-Q

Niveaux de segmentation. L'application séquentielle de plusieurs systèmes-Q au sein d'un même traitement-Q et l'utilisation de chaînes d'arbres permettent de représenter un nombre arbitraire de niveaux de segmentation.

Traitement des ambiguïtés. Le principe des systèmes-Q est de représenter les ambiguïtés d'analyse de texte dans des graphes de chaînes d'arbres étiquetés. C'est à l'auteur du traitement-Q de gérer les ambiguïtés à l'aide de la structure disponible.

Formats d'entrée et de sortie. L'entrée d'un traitement-Q est un graphe de chaînes que l'on peut construire trivialement à partir d'un texte source.

Le format de sortie d'un système-Q est un graphe de chaînes. Ces graphes sont donnés sous forme textuelle. Au cours d'un traitement, toutes les étapes d'analyse sont détaillées (c'est-à-dire que l'état d'un graphe est affiché après l'application de chaque système).

Modèle sous-jacent. Le système d'application des règles est itératif ; les règles s'appliquent tant qu'une règle peut s'appliquer à une instance de sa partie gauche où elle ne l'a pas encore été. Chaque application de règle bénéficie de l'application de règles précédentes.

Programmabilité. Néant.

Réversibilité. Une règle est réversible en échangeant tout ses membres gauche et droit. Le mot-clé `-INV-` est utilisé pour inverser les règles qui suivent. Par exemple,

```
A*(1, U) == A* + A*(U*) / A* -DANS- A, B, C.
S(1, U*) == A(U*) + B(U*) + C(U*).
```

génère la chaîne `A+A+A+B+B+B+C+C+C` à partir de la chaîne d'un seul arbre `S(1,1,1)` et s'obtient en précédant de `-INV-` le système-Q déjà vu (section 1.3.3.3).

Traitement des formes inconnues. Il n'y en a pas à proprement parler. Si aucune règle ne peut s'appliquer, le graphe de chaîne reste inchangé. Il se peut également que l'application d'un système de règles entraîne l'élimination de tout ou partie des chaînes d'un graphe.

Langues couvertes. Français, anglais, russe.

Références principales. Décrit dans [Colmerauer, 1970].

3.1.8 Caractéristiques de l'analyseur du chinois avec FSM

Niveaux de segmentation. Un seul, celui des mots.

Traitement des ambiguïtés. Les ambiguïtés peuvent être représentées efficacement dans l'automate ou le transducteur, et être toutes conservées. Le modèle statistique par unigramme permet d'obtenir le meilleur chemin, donc l'analyse la plus probable.

Modèle sous-jacent. Transducteur d'états finis pondérés et modèles statistiques d'unigrammes.

Programmabilité. Néant (pour l'analyseur du chinois). [Mohri, 2001] montre tout de même des applications des états finis pondérés qui permettent de reconnaître des langages qui ne sont pas rationnels (par exemple, des expressions régulières parenthésées).

Réversibilité. Comme pour XFST, le transducteur est applicable dans les deux directions. Le système chinois proposé ici permettrait donc d'opérer une translittération entre caractères chinois et *pinyin*.

Traitement des mots inconnus. Des classes de mots inconnus sont définies : mots étrangers, noms propres, etc. Des informations de nature phonétique (pour les mots étrangers) ou sémantiques (pour les noms propres) sont utilisées pour calculer le coût de ces mots inconnus.

Langues couvertes. Chinois (Chine continentale et Taïwan, pour ce système en particulier), évidemment la construction de transducteurs pour d'autres langues comme avec XFST est également envisageable.

Références. Décrit dans [Sproat *et al.*, 1996].

3.1.9 Caractéristiques de Chasen

Niveaux de segmentation. Un seul, celui des mots.

Traitement des ambiguïtés. Les ambiguïtés sont factorisées efficacement dans une treille. Le modèle statistique par bigrammes permet ensuite d'obtenir le meilleur chemin, donc l'analyse la plus probable, ou de classer les chemins par probabilité.

Modèle sous-jacent. Les analyses sont représentées sous la forme d'une treille, les dictionnaires par une structure intermédiaire.

Programmabilité. Pas de programmabilité à proprement parler, mais quelques options de configuration sont proposées : poids relatif du coût d'un mot par rapport à son coût de connexion, coût d'un mot inconnu, etc. Comme tout système statistique, le réglage fin des paramètres peut influencer les résultats.

Réversibilité. Néant.

Traitement des mots inconnus. Deux sortes d'heuristiques sont employées selon la nature des caractères rencontrés : dans le cas de mots étrangers, écrits en alphabet latin ou katakana, on choisit la plus longue chaîne possible. Au contraire, dans le cas de kanji ou d'hiragana, le texte est segmenté caractère par caractère. Une étiquette « mot inconnu » et un poids particulier sont attribués.

Langues couvertes. Japonais (développements vers le chinois et le coréen).

Références. Manuel d'utilisation en anglais [Matsumoto *et al.*, 1999].

3.2 Problèmes ouverts en analyse présyntaxique

Hormis l'évaluation, dont on a parlé - ou plus précisément, soigneusement évité de parler - plus haut, deux autres questions pertinentes restent en suspens : la première est de connaître précisément le rôle que joue le lexique dans l'analyse présyntaxique, puisque l'on a vu qu'il pouvait être aussi bien inexistant (remplacé par des modèles statistiques ou par une approche purement symbolique) que déterminant. Un corollaire à cette première question est le traitement des mots inconnus, déjà abordé dans la section précédente.

La deuxième question est plus liée à une méthode particulière, l'utilisation de machines d'états finis. Cette pratique étant très répandue, on s'intéresse à ses limitations théoriques et aux façons d'en contourner certaines dans la section 3.2.2.

3.2.1 L'importance du lexique dans l'analyse présyntaxique

Si l'on excepte les quelques approches purement statistiques ou purement symboliques, l'analyse morphologique repose toujours sur des lexiques et des dictionnaires, contenant les formes de la langue et les traits morphologiques liés. Mais quelle que soit la taille et la qualité du dictionnaire, les formes inconnues sont une fatalité : néologismes, fautes d'orthographe ou fautes de frappe, noms propres, etc. abondent dans le texte courant (articles de journaux ou pages trouvées sur le Web...)

3.2.1.1 La taille des lexiques

Il est très difficile d'évaluer précisément l'impact de la taille d'un lexique sur la qualité d'un analyseur morphologique, tout d'abord parce qu'il est difficile d'évaluer la qualité d'un analyseur syntaxique (voir plus bas), mais aussi parce que d'autres facteurs difficilement quantifiables entrent en jeu, comme le choix et la qualité des entrées du lexique.

D'après [Chen et Liu, 1992], « le lexique doit contenir autant de mots que possible. » Par conséquent, le lexique du segmenteur chinois simplifié présenté comprend plus de 90 000 entrées. Les auteurs suivent le principe directeur suivant : toute séquence de caractères pouvant raisonnablement être traitée comme un mot doit être ajoutée au lexique. Dans la littérature, pour la même catégorie de systèmes, on trouve ainsi des tailles de lexique allant de 30 000 à 90 000 mots. Ces lexiques sont spécifiques aux projets qui les utilisent. Un dictionnaire « classique » et librement disponible, celui de l'Academia Sinica, compte environ 78 000 entrées [kwong Wong et Chan, 1996], ce qui est du même ordre de grandeur.

Outre sa taille, l'adéquation d'un lexique au domaine étudiée joue un rôle important sur la qualité des résultats, en tout cas sur le rappel (voir section 3.2.3). Un test de Chasen sur différents corpus [Tomokiyo et Quint, 2000] montre que, dans un corpus général, seuls 0,30 % des caractères totaux ne sont pas reconnus, contre près de 6 % dans un corpus médical.

Dans [Wu et Fung, 1994], une expérience est menée sur l'amélioration d'un lexique déjà imposant (plus de 89 000 entrées). L'application d'un lexique préparé à Hong Kong à un corpus provenant de Taïwan permet de repérer de nombreux manques et une précision décevante (inférieure à 90 %). L'extraction de mots inconnus permet d'ajouter plus de 4 600 mots (d'après un corpus de plus de deux millions de mots) ; le nouveau lexique produisant de meilleurs résultats. Mais une meilleure précision est atteinte si les ajouts au lexique sont filtrés : après avoir filtré plus de 1 100 mots, la précision augmente encore pour finalement dépasser les 90 %.

3.2.1.2 Les devineurs et grammaires de mots inconnus

Il a déjà été question des méthodes d'identification de mots inconnus : méthodes statistiques d'extraction de mots (héritées des segmenteurs sans dictionnaires), heuristiques de découpage de formes, et devineurs. Le problème posé par un mot inconnu en analyse morphologique est double : quelle est sa forme de base (le lemme) et quelles sont ses informations morphologiques (étiquettes syntaxiques possibles, etc.) ?

Une bonne introduction aux devineurs est donnée par [Daciuk, 1999], exposant quelques essais précédents et développant une méthode encore une fois fondée sur les états finis. En utilisant un lexique existant et en le renversant (c'est-à-dire en écrivant les mots de droite à gauche), un ensemble fini de terminaisons des mots est extrait à partir des formes fléchies. Chaque terminaison est annotée : d'une part, avec les informations que l'on veut associer à cette terminaison (traits morphologiques, par exemple), d'autre part avec des informations pour obtenir un lemme (nombre de caractères à effacer, et séquence de caractères en remplacement). Comme l'ensemble de chaînes obtenues ainsi est fini, cette liste est compilée en un automate d'états finis acyclique, qui subit quelques transformations pour le

rendre le plus petit possible. Cet automate s'applique ensuite à chaque forme inconnue, qui doit être renversée au préalable.

Dans [Guilbaud et Boitet, 1997], on découvre une méthode plus sophistiquée puisqu'il s'agit d'une véritable grammaire pour les mots inconnus, dont le comportement s'efforce de s'approcher le plus possible d'une analyse normale (où la forme aurait été présente dans le dictionnaire). Dans le système ATEF, la grammaire du mot inconnu peut être développée autant qu'on le souhaite pour obtenir un traitement plus précis.

> *Exemple.* Soit la forme à analyser `LISPIFIONS`. Le dictionnaire de désinences contient des désinences de la forme °°`ASSIONS`, °°`IONS`, °°`ONS`, etc. chacune correspondant à un format morphologique et syntaxique donnés. La chaîne est analysée en introduisant le préfixe °°, et un suffixe `$` (soit °°`LISPIFIONS$`) Ensuite, tant que la forme ne correspond pas à une désinence connue, on supprime la première lettre pour la remettre à la fin. Pour notre exemple, trois formes seront reconnues : °°`IONS$LISPIF` et °°`ONS$LISPIF` et °°`S$LISPIFION`, qui donnent comme lemmes `LISPIFIER`, `LISPIFER` et le nom `LISPIFION`.

3.2.1.3 Des approches itératives pour un lexique mieux adapté

Comme l'on montré Wu et Fung, l'enrichissement d'un lexique, et surtout l'utilisation d'un lexique adapté au texte à analyser, permet d'augmenter sensiblement la qualité de l'analyse. Idéalement, si l'on dispose d'une analyse correcte du texte, alors on peut construire les ressources nécessaires pour faire une analyse correcte de ce texte. [Luo et Roukos, 1996] propose de construire itérativement un modèle statistique, en partant d'une première segmentation, puis en enrichissant le modèle avant d'effectuer une nouvelle segmentation, et ainsi de suite.

On peut appliquer cette idée à la reconnaissance d'entités : ainsi, si un nom propre est introduit dans un certain contexte (par exemple, sous une forme complète, précédée d'un titre ou d'indices contextuels forts), il peut être ajouté au lexique pour une nouvelle analyse, où il pourra donc être reconnu dans des contextes moins favorables (par exemple si seulement une partie du nom est employée). En appliquant cette idée à toutes formes de mots inconnus, on peut faire une analyse en plusieurs passes, un peu comme un compilateur qui ferait une première passe pour construire une table des symboles et une deuxième passe utilisant cette information.

Une autre idée similaire est d'utiliser « une segmentation par source » [Guo, 1998] : si une expression est segmentée correctement dans une partie du texte, toutes ses occurrences devraient être segmentées de la même façon dans toute cette partie du texte (une partie du texte pertinente est par exemple un fragment critique, au sens de la segmentation critique). Ainsi, une méthode de segmentation par mémorisation, apparentée aux mémoires de traduction, est proposée et semble donner de très bons résultats.

3.2.2 Limitations des états finis pour la morphologie

Les techniques fondées sur les états finis (avec ou sans pondération), jouent un rôle prépondérant dans l'analyse présyntaxique, mais des problèmes résistent à ces méthodes. Cette section présente trois catégories fréquemment citées, ainsi que quelques solutions.

3.2.2.1 Les phénomènes non-concaténatifs

Sous ce nom barbare, on trouve des phénomènes tels que le « peignage » en arabe ou en syriaque [Beesley, 1998]. Une analyse classique des langues sémitiques considère d'une part les racines, qui sont des radicaux de deux à quatre consonnes (exemple classique : *ktb*, qui a trait à l'écriture). Les bases

sont formées en insérant dans les racines les lettres d'un patron contenant des « emplacements », notés *C*, pour les consonnes de la racine.

Exemple : la racine *ktb* et le patron *CaCaC* se combinent pour former la forme verbale *katab*.

Les opérations rationnelles étant de nature concaténative (concaténation, répétition), un traitement particulier est nécessaire. Dans [Beesley et Karttunen, 2000], une nouvelle opération rationnelle est introduite : la fusion (les tentatives précédentes étaient fondées sur l'intersection ; la fusion est une évolution de l'intersection).

Le patron est divisé en deux parties, chacune représentée par un automate d'états finis. Le modèle abstrait alterne consonnes et voyelles (par exemple `CVCVC`), chaque symbole devant être réalisé par une lettre de la racine ou du patron. Le reste du patron est un réseau dit « de remplissage » (*filler network*) contenant les consonnes et voyelles précises. La fusion s'opère alors en deux temps : la racine est fusionnée avec le modèle, puis le résultat est fusionné avec le réseau de remplissage.

Exemple : soit la racine `ktb`, le modèle `CVVCVC` et le réseau de remplissage `u*i` (chaque voyelle est réalisée par `u`, sauf la dernière qui est réalisée par `i`). Dans XFST, l'opération de fusion s'écrit :

```
k t b .m>. C V V C V C .<m. u* i
```

(`.m>. et .<m.` sont les deux étapes de la fusion, *merge* en anglais). La première étape permet la réalisation des consonnes et produit la forme intermédiaire `kVVtVb`. La deuxième étape distribue les voyelles manquantes et produit la forme finale `kuutib`.

Dans [Kiraz, 2000], c'est une autre extension qui est proposée. Le patron est également décomposé en trois parties (appelées ici vocalisation, racine et patron), mais au lieu d'utiliser de simples transducteurs d'états finis, on utilise des automates à plus de bandes. Trois bandes sont ainsi dévolues au niveau lexical (pour la vocalisation, la racine et le patron), et une au niveau de surface. La génération des formes de surface se fait à l'aide de règles de réécriture mettant en correspondance les trois niveaux lexicaux avec le niveau de surface. Comme avec les transducteurs traditionnels, l'analyse comme la génération se font avec le même automate.

3.2.2.2 Les phénomènes de reduplication

Le pluriel des noms en malais ou en indonésien est sans doute l'exemple le plus connu de phénomène de reduplication [Beesley et Karttunen, 2000]. Cette fois-ci, le problème sort clairement du contexte des états finis car il est connu que le langage

$L = \{ww | w \in \Sigma^*\}$

est un langage sous contexte « strict » et qu'il ne peut donc être reconnu par un automate d'états finis si la longueur de w n'est pas bornée. En malais, le pluriel se fait par reduplication : *bagi* (« sac ») donne *bagibagi* au pluriel ; *pelabuhan* (« port »), un nom composé (qui peut être subdivisé en plusieurs morphèmes) donne *pelabuhanpelabuhan* : les racines peuvent être très complexes du fait des règles de formation des mots en malais.

En morphologie à deux niveaux, le pluriel est généralement géré par une relation comme

```
%+PL:s
```

c'est-à-dire qu'un symbole lexical `+PL` est réalisé par `s` en surface (voir section 2.1.2.2). Cette méthode est impraticable dans le cadre de la reduplication. La solution fondée sur les états finis apportée par Beesley et Karttunen consiste à générer tous les pluriels possibles des noms malais du lexique (*ie* les racines, mais également tous les noms dérivés comme *pelabuhan*) en ajoutant une étape d'évaluation : plutôt que de générer directement la forme de surface, on génère une nouvelle expression régulière qui devra être elle-même évaluée pour générer ensuite la forme de surface, par exemple :

```
     b a g i   +N +PL
^[ { b a g i }    ^ 2 ^]
```

les crochets `^[` et `^]` sont des *flag diacritics*, symboles à l'interprétation particulière, indiquant que l'expression ainsi parenthésée est une expression régulière à compiler. L'opérateur `^` est un opérateur de répétition. Cette solution permet au linguiste de traiter le pluriel pour tous les noms du dictionnaire. Par contre, un mot inconnu de la forme *ww* ne sera pas reconnu comme un pluriel.

La reduplication n'apparaît pas seulement en malais et en indonésien ; en chinois, ce phénomène est également courant [Chen et Liu, 1992]. Deux types de constructions sont la reduplication du verbe (forme *A-A* où *A* est un verbe), qui changent sa sémantique et modifient son comportement syntaxique ; et la construction interrogative courante *A-non-A* (où *A* est également un verbe). La méthode de reconnaissance préconisée est de procéder d'abord à l'itémisation du texte, puis d'appliquer des règles particulières pour l'identification *ad hoc* de composés de la forme *A-A* ou *A-non-A*[2] .

3.2.2.3 Les langages hors-contexte

Dans le cadre de l'analyse présyntaxique, par exemple dans la segmentation en phrases, on rencontre également des phénomènes hors-contexte. Par exemple, la segmentation des phrases nécessite de reconnaître des expressions parenthésées (parenthèses, guillemets, crochets, etc.) Or les langages parenthésés sont hors-contexte strict. [Roche, 1996] montre que l'on peut transformer des grammaires hors-contexte en transducteurs d'états finis à condition d'appliquer ceux-ci itérativement, jusqu'à arriver à un point fixe. Il sera à nouveau question de ce principe d'itération plus tard dans la discussion.

Une autre solution est l'approximation [Nederhof, 2000]. Si l'on reprend l'exemple des expressions parenthésées, il n'est pas aberrant de limiter la profondeur d'imbrication de parenthèses dans une phrase en français ou en anglais (alors qu'une telle limitation serait nettement plus gênante pour une langue artificielle) ; la reconnaissance de ces structures par des automates d'états finis, comme elle se fait avec INTEX par exemple, ne pose donc plus de problèmes.

3.2.3 L'évaluation des analyseurs

Dans tout ce qui a précédé, il a été maintes fois question de la difficulté de définir précisément ce qu'est un mot. Chaque segmenteur fonctionne avec un objectif précis, et produit un résultat qui lui est propre. Dans le cadre de l'analyse syntaxique, il est courant de trouver des ambiguïtés impossibles à résoudre - au niveau syntaxique en tout cas. La même situation existe pour l'itémisation, car idéalement il faudrait des informations d'ordre sémantique et une connaissance parfaite du contexte pour pouvoir choisir un découpage avec certitude [Habert *et al.*, 1998].

Tout cela fait que l'évaluation de la qualité d'un analyseur syntaxique ou simplement d'un itémiseur est une question particulièrement épineuse. Deux questions se posent en particulier :

1. comment quantifier la qualité de l'analyseur ?
2. comment comparer plusieurs analyseurs ?

Si plusieurs réponses sont développées ci-dessous pour la première question, il est déjà possible de répondre à la seconde. Une fois une mesure adoptée (ou plusieurs), une référence est choisie. En chinois, c'est souvent une heuristique de plus longue forme qui sert ainsi de référence ; en japonais, ce sont Juman et Chasen qui se retrouvent le plus souvent ; en thaï, on a l'exemple de [Meknavin *et al.*, 1997] où la référence est un analyseur à base de trigrammes, soit un modèle déjà relativement sophistiqué.

Mais la précision de la segmentation n'est pas la seule information pertinente pour l'évaluation, et différentes parties d'un système peuvent être évaluées séparément. Par exemple, la composante

[2]Notons qu'en ATEF, il est très facile de reconnaître ce genre de construction grâce à la fonction `SCHAINE` dans une fenêtre de 2 à 5 mots.

d'identification de mots inconnus d'un analyseur lexical peut être évaluée séparément de l'ensemble du système.

3.2.3.1 Simples mesures de précision

L'attitude la plus simple adoptée est de proposer une seule mesure synthétique, généralement appelée « précision ». Cela permet d'annoncer des résultats ronflants, comme 99,77 % de précision (référence exacte ?) Cette mesure de précision correspond à la comparaison de la sortie de l'itémiseur par rapport à un unique résultat attendu. On la retrouve aussi bien en chinois [Wang *et al.*, 1991; Webster et Kit, 1992; Nie *et al.*, 1995; Sun *et al.*, 1998] qu'en japonais [Kashioka *et al.*, 1998] ou en thaï [Sornlertlamvanich *et al.*, 2000b].

3.2.3.2 Précision et rappel

En recherche d'information, ce sont deux mesures qui sont généralement utilisées : la précision, qui est la justesse des résultats trouvés par rapport à la requête originale, et le rappel, qui est le nombre de résultats trouvés par rapport au nombre de résultats trouvables. Pour l'itémisation, qui peut être considérée comme la recherche des mots dans un texte, la précision (P) et le rappel (R) peuvent être définis de la sorte (prenant exemple sur [Li *et al.*, 1998]) :

$$P = \frac{nombre\ de\ mots\ correctement\ reconnus}{nombre\ de\ mots\ dans\ le\ texte}\ R = \frac{nombre\ de\ mots\ reconnus}{nombre\ de\ mots\ dans\ le\ texte}$$

On peut bien sûr regrouper précision et rappel en un seul nombre qui serait la précision générale du système considéré, mais la présence de plusieurs mesures permet de mieux comprendre les qualités et les défauts d'un itémiseur. Encore une fois, cette méthode d'évaluation se retrouve aussi bien en chinois [Sun *et al.*, 1997; Fuchi et Takagi, 1998], en japonais [Nagata, 1994; Mochizuki *et al.*, 1998] ou en thaï [Sornlertlamvanich *et al.*, 2000a].

3.2.3.3 Accord entre juges

Dans les deux cas précédents, il faut pouvoir disposer d'un unique résultat idéal, comme un corpus segmenté ou des résultats validés manuellement. Mais une phrase donnée ne peut-elle pas avoir plusieurs lectures ? Il est alors possible que deux résultats différents soient également valides, ce que ne peuvent pas prendre en compte les techniques exposées précédemment.

Une intervention manuelle devient alors nécessaire. On parle de « juges », qui proposent leur propre segmentation d'un texte, et déclarent s'ils sont d'accord ou non avec une autre segmentation qu'on leur propose [Wu et Fung, 1994]. La méthode des nk juges demande que n juges parmi k acceptent une segmentation pour que celle-ci soit considérée comme correcte.

Dans [Sproat *et al.*, 1996], un processus d'évaluation plus complet est proposé ([Xia *et al.*, 2000] reprend un procédé similaire). Le chinois écrit étant une langue partagée par des centaines de millions de personnes, des divergences régionales sont inévitables. Ainsi, un locuteur natif de Pékin n'aura pas la même vision de l'itémisation qu'un taïwanais : six juges, trois de Chine continentale, et trois de Taïwan, participent donc à l'évaluation, en plus de deux itémiseurs « témoin », l'un utilisant l'heuristique de plus longue forme, et le second celle de plus courte forme.

Le résultat n'est plus une question de précision ou de rappel, mais d'accord avec les juges. Par exemple, le segmenteur utilisant l'heuristique de plus longue forme s'accorde avec le premier juge chinois dans 43 % des cas, et avec le premier juge taïwanais dans 60 % des cas. Le modèle d'unigrammes réalisé avec un transducteur d'états finis pondérés améliore ce résultat d'un vingtaine de pour-cent, obtenant des scores allant de 64 à 84 % avec les juges. Si ces résultats semblent moins impressionnants que les 95 à 99 % typiques des autres systèmes, on peut tout de même remarquer que les juges sont très souvent en désaccord entre eux !

3.3 Vers un formalisme universel pour la segmentation

Parmi tous les éléments réunis jusqu'à présent, il convient désormais de faire un choix sur la façon dont on compte unifier les différentes approches pour proposer un formalisme universel pour la segmentation. Les choix réalisés sont présentés de la même manière et suivant les mêmes critères que dans la section 3.1.

Niveaux de segmentation. Une réelle indépendance de la langue exige qu'il n'y ait aucun *a priori* sur les unités linguistiques à traiter. Si la plupart des articles sur l'itémisation du chinois commencent par : « Qu'est-ce qu'un mot en chinois ? », ce n'est pas à l'auteur du formalisme de trancher, mais au linguiste qui l'utilise. Plusieurs niveaux de segmentation sont alors nécessaires.

Un autre avantage de l'introduction de niveaux de segmentation est qu'il est possible de segmenter un même texte en plusieurs types d'unités (*e.g.* mots, phrases, etc.) avec le même outil, chaque segmentation pouvant se faire en utilisant des informations sur la segmentation des autres niveaux.

Traitement des ambiguïtés. Une segmentation parfaite est sans doute impossible en n'utilisant que des informations de nature présyntaxique. Afin de ne pas faire de mauvais choix à cette étape, il est nécessaire de conserver des ambiguïtés. Un moyen habituel de représenter les ambiguïtés résultant de la segmentation et de l'analyse des segments est une forme de graphe ou de treille.

Cependant, il est tout de même nécessaire de supprimer un maximum de résultats qui sont clairement incorrects, et de réduire ainsi l'ambiguïté de la sortie. Dans les approches observées ici, cela peut se faire par des règles, des heuristiques et/ou l'utilisation de statistiques. L'utilisation de statistiques doit être rendue possible par l'emploi d'une structure pondérée.

Formats d'entrée et de sortie. Les formats d'entrée du formalisme doivent accepter au moins les trois formats suivant :

1. du texte brut dans un encodage quelconque ;
2. du texte au format XML, qui est désormais un format d'échange de donnés très répandu. Des corpus entiers se constituent à partir de documents provenant du Web : ces documents sont principalement en HTML (facilement transformables sous forme XML) ou en XHTML, qui est une application de XML ;
3. du texte au format « interne », qui peut provenir d'une analyse précédente, ou d'un processus générant du texte ambigu comme la reconnaissance de la parole ou de l'écriture (bien que dans ce cas là, une conversion de format soit probablement nécessaire).

La sortie d'un analyseur réalisé avec le formalisme doit être versatile pour s'adapter à différentes applications : il doit être possible de la formater pour qu'elle soit informative (comme Chasen, par exemple), mais elle doit aussi correspondre aux différents formats d'entrée possibles.

Modèle sous-jacent. Un modèle fondé sur les états finis est un choix logique au vu de l'état de l'art dans le domaine de la segmentation. Les documents en cours d'analyse peuvent être représentés par des automates d'états finis pondérés (permettant l'utilisation de statistiques), qui sont ensuite manipulés par des transducteurs d'états finis pondérés, définis par des règles de réécriture pondérées.

Programmabilité. Le modèle d'états finis est une fondation solide, mais certaines limites d'expressivité persistent. Un contrôle plus fin sur l'application d'un transducteur (application conditionnelle, comme dans `lookup`, ou itérative, jusqu'à un point fixe) permet de pallier certains manques. De même, l'addition de symboles particuliers (comme les *flag diacritics* de XFST) permet des extensions intéressantes.

Réversibilité. Il est facile d'intervertir les niveaux d'un transducteur d'états finis, et donc de transformer un analyseur en générateur. Il faut cependant veiller à ce que les extensions au modèle d'états finis conservent cette réversibilité.

Traitement des mots inconnus. Le traitement des mots inconnus dans ce formalisme sera laissé à la discrétion du linguiste. Les stratégies vues précédemment (grammaire de mot inconnu, reconnaissance de la plus longue ou plus courte chaîne de caractères inconnus, etc.) doivent être permises par cet outil.

Langues couvertes. Par définition, le formalisme doit permettre le traitement de n'importe quelle langue. Cela recouvre donc plusieurs points :

1. la gestion de différents encodages. Le formalisme ne doit pas se fonder sur l'hypothèse classique « un caractère = un octet » mais prendre en compte tout type d'encodages. Dans la pratique, il est illusoire de gérer tous les encodages existants, mais une bonne utilisation d'Unicode permet une couverture déjà très importante ;
2. la définition de niveaux de segmentation ;
3. la séparation claires des données et du processus de segmentation ;
4. l'expressibilité de phénomènes présyntaxiques par le modèle computationnel.

Conclusion

Au terme de l'étude de l'état de l'art en analyse présyntaxique, on arrive à la conclusion qu'il est possible de créer un formalisme permettant aux linguistes et informaticiens d'attaquer les problèmes de segmentation, et par extension de morphologie et de présyntaxe, dans une langue donnée de la même manière qu'ils le feraient pour la syntaxe où existent de nombreux formalismes. Les *desiderata* pour un tel formalisme ont été exprimés : une séparation claire entre les données et les processus d'analyse, la possibilité de pouvoir travailler à n'importe quel niveau de segmentation, et même sur plusieurs niveaux en même temps, la possibilité d'utiliser des statistiques ou non, etc. Le formalisme sera fondé sur les états finis, qui sont centraux dans les travaux actuels sur la présyntaxe, mais disposera également de mécanismes d'extension qui sont nécessaires pour une plus grande souplesse

Deuxième partie

Le formalisme Sumo

Introduction

Le formalisme Sumo (Segmentation Universelle Multiple par Ordinateur) est une première réponse aux questions soulevées par la première partie, sous la forme d'un outil destiné à l'écriture de segmenteurs, et ce quel que soit le ou les niveaux de segmentation considérés.

L'ambition principale de Sumo est d'être un outil utile aussi bien pour la conception d'itémiseurs, que pour celles de segmenteurs plus spécifiques, à l'image de Satz ou des segmenteurs pour le thaï. Et puisque Sumo n'impose pas de restriction sur le nombre et la nature des niveaux de segmentation, toutes sortes d'autres applications liées à la segmentation et à la présyntaxe sont envisageables, comme on le verra dans le chapitre 9.

Après la première partie, le lecteur devrait être également convaincu que la segmentation ne peut généralement se concevoir qu'en fonction des autres tâches présyntaxiques. Aussi Sumo doit-il répondre à ce besoin en proposant un modèle linguistique et calculatoire puissant et versatile. Ce modèle comprend une structure dédiée à la segmentation multiple d'un même document (c'est-à-dire selon plusieurs niveaux en simultané), et les moyens de manipuler cette structure à l'aide d'opérateurs et de fonctions clairement définis.

Alors, l'écriture d'un analyseur avec Sumo consiste à décider des niveaux de segmentations à considérer pour les documents que l'application doit traiter, et comment chaque niveau est lié aux niveaux qui lui sont adjacents. Il faut également prendre en compte la manière dont chaque niveau est construit en fonction des autres niveaux du document.

Cette deuxième partie de la discussion décrit le formalisme Sumo pour en donner une vue d'ensemble. Le chapitre 4 présente les structures de données du formalisme et les principales opérations et fonction en donnant un exemple détaillé de moteur de segmentation écrit avec Sumo. Le chapitre 5 approfondit la description en détaillant les aspects précis de Sumo. Enfin, dans le chapitre 6, les problèmes relatifs au prototypage et à l'implémentation de ce formalisme sont discutés.

Chapitre 4

Introduction au formalisme Sumo

Introduction

Dans ce chapitre sont exposés les principes fondamentaux de Sumo : premièrement, la structure de données pour la représentation de documents et la segmentation sur laquelle repose le formalisme, et deuxièmement, les autres composantes de Sumo pour la réalisation de moteurs de segmentation utilisant cette structure de représentation complexe et originale. Le chapitre 5 détaille les spécifications précises de ces composantes ; mais pour que la discussion ne soit pas trop abstraite et plus facilement compréhensible, un exemple complet et illustré est donné dans la troisième section de ce chapitre pour comprendre les principes fondamentaux de fonctionnement de Sumo.

Des versions préliminaires de Sumo ont été présentées dans [Quint, 1999; Quint, 2000b; Quint, 2000a].

4.1 Documents Sumo

La structure de représentation du document au cours de la segmentation est appelée « document Sumo ». Il peut s'agir aussi bien d'un document en cours de segmentation que de données linguistiques pour la segmentation à partir desquels on extrait les règles d'identification et de liaison utilisées pour la segmentation. On peut faire un parallèle avec la morphologie à états finis, où le texte analysé peut être lui-même un automate d'états finis, et où l'analyseur est un transducteur que l'on compose avec cet automate pour l'analyse ou la génération.

C'est une structure étagée, ou multi-couches, qui peut représenter le même document simultanément à différents niveaux de la segmentation. La taille de ce document est arbitraire : il peut aussi bien s'agir d'un court fragment que d'un document composé de plusieurs milliers de mots.

On décrit cette structure de document en commençant par la plus petite unité qu'elle peut contenir : l'unité de segmentation (on appelle ces unités les « items »). Une segmentation possible à un niveau donné est une séquence d'items de ce niveau ; toutes les segmentations possibles au même niveau sont factorisées sous la forme d'un graphe d'items. Ainsi, pour chaque niveau de segmentation, on aura un graphe d'items différent, qui trouvera sa place dans un étage de la structure Sumo.

4.1.1 Unités de segmentation

Le choix de l'unité de segmentation est déterminant pour la réalisation d'un segmenteur car il définit en grande partie le résultat de l'analyse. Dans Sumo, une telle unité de segmentation est appelée item (ce terme correspond à la traduction française de *token*, mais on le généralise pour toute

étape de segmentation). L'item est un concept central de Sumo : la segmentation du texte y est vue comme la transformation d'une séquence d'items en une autre séquence d'items de nature différente. Par exemple, la segmentation en mots est une transformation d'une séquences de caractères en une ou plusieurs séquences de mots.

C'est la nature exacte des items d'un niveau de segmentation qui permet de définir la nature même du niveau de segmentation en question.

4.1.1.1 Définition formelle

Un item est défini par un couple (i, A) où i est l'intitulé de l'item, et A un ensemble fini d'attributs. L'intitulé de l'item est une forme correspondant à cet item : forme de surface, forme lexicale, étiquette, etc. L'intitulé est une chaîne de caractères définie sur un alphabet donné Σ , et peut être une chaîne vide.

Exemples d'intitulés :

- une forme de surface : `chevaux` ;
- une forme lexicale : `cheval` ;
- un radical : `ktb` ;
- une étiquette : `NP` ;
- une forme vide : ϵ .

Les attributs sont des informations supplémentaires sur l'item, qui complètent l'intitulé. Chaque attribut est un couple (a, V) où a est le nom de l'attribut et V un ensemble fini de valeurs. Les valeurs n'ont pas de type déclaré et sont donc des chaînes sur Σ au même titre que l'intitulé. Un attribut peut ne pas avoir de valeur, ou avoir pour valeur l'ensemble vide.

Exemples d'attributs :

- une simple étiquette : (Fem, {Fem}) qui correspondrait à un symbole comme `+Fem` dans la morphologie à deux niveaux de Xerox ;
- un trait syntaxique pouvant avoir plusieurs valeurs : (catégorie, {nom}) ;
- des valeurs potentielles pour un item donné : (nombre, {singulier, pluriel}).

Les attributs contiennent toutes sortes d'information, linguistique ou autre. La sémantique exacte des attributs est laissée à la discrétion de l'auteur, de même que la manière de traiter différentes valeurs pour un même attribut. On reviendra précisément sur les types dans le chapitre 5.

4.1.1.2 Exemples d'items

Au niveau des mots, un item contient l'analyse morphologique complète d'une forme. L'intitulé est la forme lexicale du mot (résultat de la lemmatisation), les attributs sont les traits morphologiques qui lui sont associés. Soit la forme « relations », deux des analyses possibles donnent les deux items :

1. (*relation*, {(catégorie, {nom}), (genre, {féminin}), (nombre, {pluriel})}) ;
2. (*relater*, {(catégorie, {verbe}), (personne, {2}), (nombre, {pluriel}), (temps, {présent}), (mode, {subjonctif})}).

Si l'on procède à une segmentation du document en éléments structurels, par exemple en séparant chapitres, sections, sous-sections et sous-sous-sections, la présente sous-section correspond à l'item suivant :

(*Les unités de segmentation*, {(partie, {2}), (chapitre, {4}), (section, {1}), (sous-section, {1})})

4.1.1.3 Relations entre items

D'après les exemples vus plus haut, il semble manquer nombre d'informations sur les items considérés. Quelle est la forme de surface de (relation, {(catégorie, {nom}), ...}) ? Quel est le contenu de la sous-section « Les unités de segmentation » ? Un item ne contient que des informations pertinentes pour le niveau de segmentation considéré. La forme de surface de *relation...* est constituée d'un ou plusieurs items de surface ; la sous-section *Les unités de segmentation* contient des paragraphes et des sous-sous-sections qui sont des unités de niveau inférieur. Les relations entre items de différents niveaux sont explicitées plus bas.

4.1.2 Graphes d'items

Si les items définissent la nature des niveaux de segmentation, ce sont les graphes d'items qui représentent le document à un niveau de segmentation donné. Un graphe d'items contient toutes les segmentations possibles du document reconnues à un moment donné du traitement.

4.1.2.1 Définition formelle

Un graphe d'items est défini par un couple (S, A) , où S est l'ensemble fini des sommets du graphe et A est l'ensemble fini des arcs du graphe (on reprend ici la terminologie de [Xuong, 1992]). Parmi les sommets du graphe d'items, on distingue le sommet initial s_i qui est source de tous les chemins dans la graphe, et le sommet terminal s_t qui est la destination de tous les chemins dans le graphe. Les sommets ne portent pas d'information particulière, mais sont numérotés. Les graphes d'items représentant la segmentation d'un document sont des graphes sans circuits (DAG) ; on verra cependant plus loin que Sumo manipule également des graphes d'items plus généraux qui peuvent contenir des circuits.

Un arc est une arète orientée reliant deux sommets. On définit un arc par un quadruplet (q, i, p, r) où q et $r \in S$ sont les extrémités de l'arc (respectivement le sommet d'adjacence et le sommet d'incidence), i est un item (c'est l'étiquette de l'arc), et p est le poids de l'arc (on y revient plus bas).

Un premier exemple de graphe d'items est donné par la figure 4.1 ; c'est un graphe très simple car il représente un texte segmenté en caractères, sans ambiguïté. Comme pour les automates d'états finis, on indique le sommet initial par un trait gras, et le sommet final par un double trait.

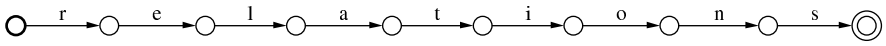

FIG. 4.1 – Un graphe d'item simple

La figure 4.2 montre un graphe d'items plus complexe ; cette fois-ci, le texte a été segmenté en mots et le graphe représente le niveau lexical du document. Les ambiguïtés de segmentation et d'analyse ont été conservées.

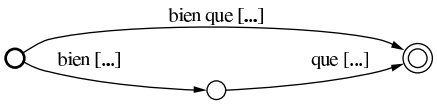

FIG. 4.2 – Un graphe d'item avec deux chemins

4.1.2.2 Propriétés des graphes d'items

La définition des automate d'états finis donnée plus haut est très générale, et les graphes que l'on sera amené à manipuler ont des propriétés plus spécifiques. Cependant, on ne restreint pas la définition, car on sera amené plus tard à manipuler des graphes d'items un peu particuliers.

Les extrémités du graphe. Les deux extrémités du graphe (le sommet initial et le sommet final) sont uniques. Un graphe d'items représentant un document a donc un unique sommet initial et un unique sommet final. De plus, ces deux sommets ont un poids nul.

Chemins et sous-chemins dans un graphe. Un chemin est une succession d'arcs dans le graphe reliant l'état initial à l'état final. Dans le cadre classique d'un automate d'états finis, la concaténation des étiquettes des arcs d'un chemin donne un mot du langage reconnu ; le langage lui-même est l'ensemble de tous les chemins possibles. Dans le cadre d'un graphe d'items, un chemin est une segmentation possible du document.

Un sous-chemin est une succession d'arcs dans le graphe entre deux états quelconques. Il s'agit donc juste d'un sous-chaîne, et non d'un mot du langage, ou de la segmentation d'une partie du document (et non de tout le document). Contrairement à la terminologie plus stricte de la littérature sur les états finis, on peut parler indifféremment de chemins et de sous-chemins dans un graphe d'items.

Définition d'un sous-graphe. Étant donné un graphe d'items G on peut définir un sous-graphe par deux sous-ensembles finis de sommets de G : l'ensemble des sommets initiaux I et celui des sommets finals F . Le sous-graphe lui-même contient tous les sommets et tous les arcs de G qui se trouvent sur au moins un sous-chemin entre un sommet initial $i \in s_i$ et un sommet final $f \in s_f$.

4.1.2.3 Graphes d'items et automates d'états finis pondérés

Dans Sumo, un graphe d'items est en réalité un automate d'états finis pondérés. Le parallèle entre un graphe d'items tel que décrit ci-dessus et un automate d'états finis pondérés est trivial. On peut donc donner une définition complète d'un graphe d'items au sens de Sumo, c'est-à-dire d'un automate d'états finis pondérés.

Les ensembles de pondération sont définis comme on l'a vu dans la section 1.2.3.1 :$\mathbb{K} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ est défini comme un anneau à ceci près que la loi $\oplus$ fait de S un demi-groupe et l'on demande souvent que :

- $\overline{0}$ soit un élément neutre pour $\oplus$ (comme pour $\mathbb{N}$) ;
- $\oplus$ soit idempotente ($\forall a \in S, a \oplus a = a$) ;
- si $a_1, a_2, \ldots, a_i, \ldots$ est une séquence dénombrable d'éléments de S , alors $a_1 \oplus a_2 \oplus \ldots \oplus a_i \oplus \ldots$ existe et soit unique ;
- $\otimes$ soit distributif sur les sommes infiniment dénombrables comme sur les sommes finies.

On appellera simplement poids un élément $k \in \mathbb{K}$ étant donné qu'un demi-anneau permet de définir une structure de poids pour une machine d'états finis pondérée. Le demi-anneau le plus courant dans le domaine du TALN est le demi-anneau tropical, $(\mathbb{R} \cup \{\infty\}, min, +, \infty, 0)$, soit l'ensemble des réels où min (minimum de deux réels) est l'opération de sommation et $+$ (addition de deux entiers) l'opération d'extension. Pour la suite de la discussion, il est admis (sauf mention contraire) que l'ensemble de pondération choisi est le demi-anneau tropical.

Un automate d'états finis pondérés est alors défini sur un demi-anneau fermé $\mathbb{K}$ par un 7-uplet $A = (\Sigma, Q, I, F, E, \lambda, \rho)$ avec :

- Σ l'alphabet fini de l'automate ;
- Q un ensemble fini d'états ;
- $I \subseteq Q$ l'ensemble des états initiaux ;
- $F \subseteq Q$ l'ensemble des états finals ;

– $E \subseteq Q \times \Sigma \cup \{\epsilon\} \times K \times Q$ l'ensemble des transitions ;
– $\lambda : I \to K$ est la fonction de pondération initiale ;
– $\rho : F \to K$ est la fonction de pondération finale.

4.1.3 La structure de représentation de documents

Tous les éléments sont maintenant réunis pour une description complète de la structure de représentation de documents proprement dite. L'objectif premier de cette structure est de servir de conteneur efficace pour toutes les segmentations possibles ou envisagées d'un document à différents niveaux.

Les niveaux sont des étages du graphe, il existe donc une relation d'ordre entre eux. Des relations existent entre ces niveaux pour exprimer la décomposition d'un item ou d'une séquence d'items d'un niveau n en items de niveau $n-1$ (le niveau $n-1$ est inférieur au niveau n). Un niveau de segmentation correspond à un ou plusieurs graphes d'items et est nommé (on parlera donc du niveau des mots, des caractères, etc.)

4.1.3.1 Relations entre chemins

Le dernier élément restant à définir est la relation entre deux chemins de niveaux différents. Il s'agit de relations entre deux niveaux du langage, tout comme les relations rationnelles sont des relations entre deux langages rationnels. Ainsi, un mot se décompose en morphèmes ou en caractères ; une phrase en mots, etc.

La figure 4.3 montre un exemple de relation entre deux chemins ; par convention, on place le graphe inférieur en bas et le graphe supérieur en haut de la figure ; les relations sont orientées du haut vers le bas.

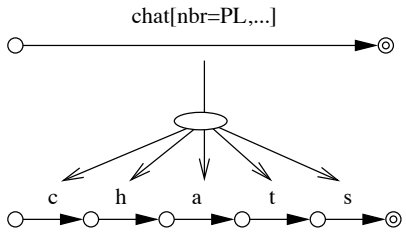

FIG. 4.3 – Une relation de chemins

Formellement, une relation entre deux chemins est un couple (c_s, c_i) avec c_s (respectivement c_i) un sous-chemin (une liste d'arcs consécutifs) du graphe supérieur (respectivement inférieur) qui participe à la relation.

Un item peut participer à plusieurs relations simultanément ; par exemple, une phrase peut être représentée au niveau n par un item « phrase », et ses différentes segmentations en mots par des chemins de niveau $n-1$. L'item phrase est en relation avec plusieurs chemins différents, ce qui s'exprime par plusieurs relations différentes.

4.1.3.2 Documents Sumo

Un document Sumo n'est autre qu'un ensemble de graphes d'items. Chaque graphe d'item prend place dans un niveau de segmentation. Les graphes sont relié entre eux par les relations entre chemins ;

il n'y a pas d'autre lien explicite entre les niveaux de segmentation. Par contre, étant donné un graphe d'items g de niveau n , toutes les relations ayant pour origine un chemin $c \in g$ doivent avoir pour destination un chemin appartenant à un graphe g' de niveau $n-1$.

Niveaux de segmentation. Informellement, un niveau de segmentation n'est autre qu'un ensemble de graphes dont les items sont tous de même nature. Les niveaux sont nommés par l'auteur du système est sont strictement ordonnés, et l'on a un niveau inférieur à tous les autres niveaux et un niveau supérieur à tous les autres.

La construction d'un document Sumo se fait pendant l'analyse du texte d'une manière généralement ascendante : le niveau le plus bas est une segmentation automatique du document source en caractères ; les niveaux successifs sont constitués d'unités plus grandes, chacun étant obtenu par la segmentation du niveau inférieur. La figure 4.4 montre un résultat typique de document Sumo.

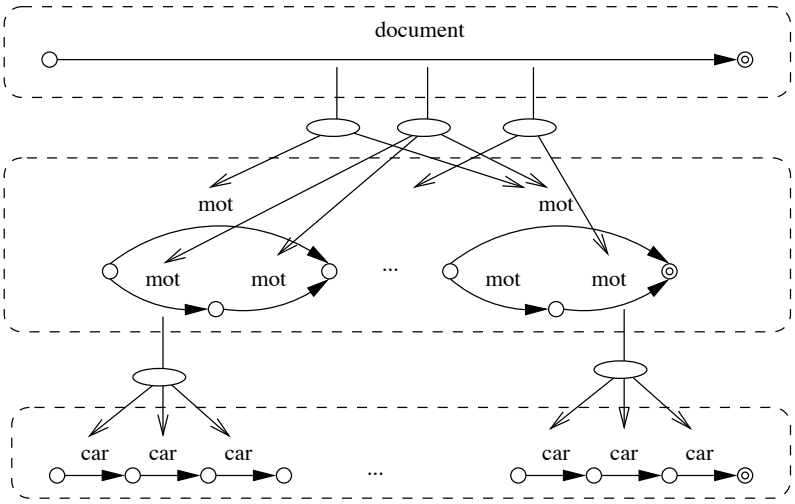

FIG. 4.4 – Un document Sumo à trois niveaux

Le niveau le plus bas est celui des caractères (on a illustré uniquement le début et la fin du texte). Au niveau directement supérieur, on trouve les mots résultant de l'analyse morphologique (ou d'une simple itémisation) ; on a montré quelques relations entre les deux niveaux. Le graphe montre les ambiguïtés de segmentation à ce niveau. Enfin, au dernier niveau, on a le document dans son entier (un seul item), en relation avec toutes les itémisations possibles (on en a esquissé trois différentes dans la figure).

La structure est souple : on peut ne sélectionner que certaines segmentations en n'exprimant que quelques relations entre l'item document et le niveau des mots ; mais on peut aussi supprimer les segmentations non-désirées du niveau des mots.

Formellement, un niveau de segmentation est un couple $n = (n, G)$ où n est le nom du niveau (une simple chaîne de caractères qui sert d'identifiant pour le niveau) et G est un ensemble fini de graphes d'items. Sumo ne définit pas de « types » d'items précis : c'est le fait qu'un graphe appartienne à un niveau de segmentation qui fait que les items dont il est composé sont de ce niveau. Ainsi, tous les graphes d'un même niveau ont *de facto* des items de même nature. Il découle également de cela qu'un graphe ne peut appartenir qu'à un seul niveau de segmentation.

Définition d'un document. Tous les éléments composants un document Sumo ayant été définis, la définition d'un document est extrêmement simple. Il s'agit simplement d'un couple (N, R) où N est la liste des niveaux de segmentation, et R est l'ensemble fini des relations entre les différents graphes constituant les niveaux de segmentation.

Dans l'implémentation de Sumo, on associera d'autres informations au document : le ou les fichiers « source » du document, son encodage, etc.

4.2 Composantes principales de Sumo

Sumo comprend les outils nécessaires à la manipulation des deux principales structures de données que sont les transducteurs d'états finis et les documents Sumo. Pour ces deux types de constructions, un calcul spécifique est défini : pour les transducteurs, ce sont les opérations rationnelles classiques ; pour les documents Sumo, ce sont des opérations similaires, ainsi que des opérations plus spécifiques. Enfin, la composante de contrôle de haut niveau permet de construire les applications de segmentation exploitant ces structures.

4.2.1 Calcul d'états finis pondéré

La première des composantes algorithmiques de Sumo est la partie calcul d'états finis pondéré. Cette composante permet de décrire des transducteurs d'états finis pondérés par des expressions régulières, de charger et de sauver des transducteurs existants, tout comme le permet le formalisme XFST de Xerox, mais avec des transducteurs pondérés.

Cette composante existe sous deux formes. Sous sa première forme, elle est totalement intégrée dans Sumo, alors que sous sa seconde forme, elle en est indépendante et s'utilise toute seule à la manière de l'interface `xfst` de Xerox : c'est l'interface `wfst`. On reviendra sur cette interface dans la troisième partie, et on s'attardera donc pour l'instant sur la première forme.

Dans le cadre de Sumo, le rôle de la composante d'états finis pondérés est de manipuler individuellement le texte à chaque niveau de segmentation. La segmentation à chaque niveau étant représentée par un graphe d'items, lui-même équivalent à un automate d'états finis, les transducteurs d'états finis permettent de transformer les graphes d'items de manière très naturelle.

Sumo définit donc une syntaxe particulière pour les expressions régulières qui présentent la particularité de définir non pas de simples transducteurs, mais des transducteurs pondérés. [Mohri et Sproat, 1996] propose une syntaxe pour les règles de réécriture pondérées. Sumo s'inspire de cette syntaxe pour ses règles de réécriture pondérées et l'étend aux expressions régulières dans leur ensemble. La syntaxe précise est exposée en détail dans le chapitre 5 ; on en donne quelques exemples ici pour s'en faire une idée.

Exemple. Cette expression régulière définit le transducteur de la figure 4.5 :

```
a:A/2 (b:B/3 | c:C/1)+
```

Ici chaque symbole se décompose en trois parties : un symbole du langage supérieur, un symbole du langage inférieur (séparés par `:`, qui est l'opérateur du produit cartésien), et un poids (précédé du séparateur `/`).

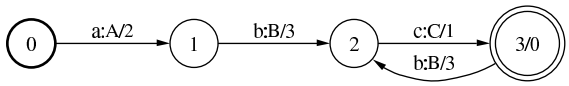

Fig. 4.5 – Transducteur reconnaissant `a:A/2 (b:B/3 c:C/1)+`

4.2.2 Opérations analogues sur les documents Sumo

Sumo est doté d'expressions analogues aux expressions régulières pour décrire les structures complexes. Comme les expressions régulières, celles-ci sont fondées sur trois opérateurs fondamentaux, qui

sont la concaténation, l'union et l'itération. Informellement, on peut dire qu'il s'agit de décrire des structures complexes en composant des structures plus simples. Une opération donnée s'effectue sur toute la structure Sumo en la répercutant à chaque niveau : ainsi, l'union de deux structures Sumo se fait en faisant l'union des deux niveaux les plus bas de chaque structure, puis des deux niveaux supérieurs, etc.

On dispose également d'expressions de chemins pour décrire des chemins et leurs relations dans une structure Sumo.

Exemple. L'expression de chemin

```
(d u) ^: (de le)
```

illustre la relation entre deux chemins : au niveau le plus bas, les deux caractères `d` et `u` sont liés au niveau supérieur (par l'opérateur `"^:"`) aux deux mots `de` et `le`. Cette relation est illustrée par la figure 4.6.

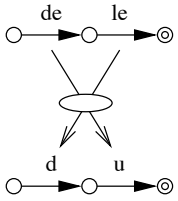

FIG. 4.6 – Structure Sumo décrivant la relation `(d u) ^: (de le)`

4.2.3 Contrôle de haut niveau

La composante de contrôle de haut niveau de Sumo est ce qui en fait un outil de construction de moteurs de segmentation. Elle permet la description des stratégies de segmentation en exploitant des ressources existantes, ainsi que la création de nouvelles ressources.

Dans un formalisme même puissant comme ATEF ou XFST, l'analyse se fait toujours de la même manière et l'auteur n'a que peu de latitude quant à un traitement différent : c'est le critère appelé « programmabilité » dans la comparaison systématique du chapitre 3. Par exemple, dans XFST, il existe des embryons d'un langage de contrôle : `lookup` permet de définir des stratégies de recherche avec des parcours différents du graphe des solutions (largeur ou profondeur d'abord, par exemple) ; `xfst` peut appeler des scripts pour le traitement répétitif ou systématique de certaines tâches, mais on ne peut pas encore parler de véritable langage de haut niveau.

Les approches de bas niveau (au sens cette fois-ci de l'abstraction linguistique) permettaient une bien meilleure programmabilité, mais en s'adressant à des informaticiens plutôt qu'à des linguistes. Les LSPL, langages spécialisés pour la linguistique [Lafourcade, 1994], apportent un début de solution. Un excellent exemple est le système KEAL [Quinton, 1980], qui propose un langage pour la description de stratégies de recherche dans un espace de solutions.

Sumo permet de faire de même à l'aide de sa composante de contrôle, même si celle-ci est sans doute d'un peu plus bas niveau, et se rapproche plus des langages de programmation plus génériques, pour un maximum de souplesse.

4.3 Un exemple détaillé d'analyseur morphologique

Le meilleur moyen de comprendre le fonctionnement de Sumo est de montrer un exemple complet et détaillé d'analyseur morphologique. L'exemple présenté ci-dessous est lui-même générique ; on montre

un cadre de segmentation et d'analyse précis, mais on ne précise pas de données particulières, même si la section 4.3.3 décrit sommairement la manière dont celles-ci peuvent être spécifiées pour obtenir un analyseur complet.

La mise en place du système est la suivante. Le texte une fois analysé est représenté par une structure à deux niveaux de segmentation :

1. Le niveau le plus bas est le niveau de surface et correspond exactement au fichier source à analyser (en se plaçant dans l'optique d'analyse de texte brut dans un encodage donné). Les items de ce niveau sont donc des caractères au sens de l'encodage du document.
2. Le niveau le plus élevé est le niveau lexical. Les items de ce niveau sont des formes lexicales annotées résultant de l'analyse morphologique des formes de surface correspondantes. Itémisation et lemmatisation se font de concert ; un item est ici équivalent à un mot. Les ambiguïtés de segmentation et de lemmatisation sont conservées dans le graphe d'items de ce niveau.

Le lexique employé pour la segmentation est lui-même un document Sumo qui a la même forme : un niveau de surface et un niveau lexical. Le texte en entrée permet de construire le niveau de surface du document analysé ; l'application du lexique permet l'identification des items lexicaux pour créer le niveau lexical dans le document analysé. Cette application est exhaustive et produit toutes les analyse possibles comme dans l'exemple de la section 4.3.1.

Cependant, il n'est souvent pas judicieux de conserver *toutes* les analyses possibles. Une phase supplémentaire de désambiguïsation peut donc être nécessaire. Dans la section 4.3.2, on verra comment réaliser cette désambiguïsation selon des heuristiques de plus longue forme, un modèle d'unigrammes ou un modèle de bigrammes.

4.3.1 Le moteur d'analyse

Chacun des différents objets manipulés dans Sumo correspond à un type. On a ainsi des booléens (`bool`), des nombres réels ou entiers (`number`), des chaînes de caractères (`string`) qui sont des chaînes Unicode ; mais aussi des transducteurs d'états finis (`wfst`), des graphes d'items (`graph`), etc. Enfin, le type d'un document Sumo est simplement appelé `sumo`.

La première étape consiste à déclarer la structure principale qui va contenir le document à analyser.

```
S : sumo ;
```

La deuxième étape est la récupération du texte en entrée. Celle-ci se fait à partir d'un fichier à l'aide d'une primitive `read_text` qui lit un fichier en texte brut et le convertit en un graphe de caractères. Les deux paramètres de cette fonction sont le fichier à lire (`"-"` représente l'entrée standard) et son encodage (ici l'ISO-8859-1, autrement dit l'ISO-Latin1).

On affecte le graphe résultat directement à la liste (en fait un tableau) des niveaux de segmentation de `S` (le tableau `S.level`) et on donne un nom à ce graphe, « Caractères ».

```
S.level[1].graph = read_text("-", "iso-8859-1") ;
S.level[1].name = "Caractères" ;
```

La troisième étape est l'analyse morphologique proprement dite à l'aide d'un lexique. La segmentation se fait à l'aide de règles dites d'identification (décrites plus loin). On peut représenter ces règles sous la forme d'une structure Sumo, mais elles doivent être « extraites » de la structure pour être appliquées.

La fonction `read_sumo` créé une structure Sumo à partir d'un fichier contenant une structure précédemment enregistrée. La fonction `extract` en extrait les règles d'identification (le second paramètre indique quel est le niveau des règles à extraire).

```
lexique : grammar ;
lexique = extract(read_sumo("lexique.sumo"), 1) ;
```

La suite de l'analyse morphologique est la phase d'identification : l'opération `identify` est une opération au cœur de Sumo qui crée un nouveau graphe d'items à partir d'un graphe existant et d'une grammaire de règles d'identification. On donne donc comme arguments à la fonction `identify` le niveau des caractères et les règles extraites du lexique pour créer un deuxième niveau que l'on appelle « Mots ».

```
S.level[2].graph = apply(S.level[1], lexique) ;
S.level[2].name = "Mots" ;
```

Notons que l'exemple présenté ici omet quelques détails, comme par exemple la normalisation et le nettoyage du texte en entrée, ou les interactions avec l'utilisateur (par exemple, que faut-il faire du document une fois l'analyse terminée ?) Le premier point est une simple affaire de prétraitement à l'aide de transducteurs d'états finis pondérés ; on verra dans la section 4.3.3 des exemples de transformation de graphes d'items par des transducteurs pondérés. Quant au deuxième point, on espère que le chapitre 5, qui détaille entre autres les questions d'entrée/sortie, donnera assez d'éléments pour imaginer différents scénarios possibles.

4.3.2 Résolution des ambiguïtés

La résolution des ambiguïtés consiste à choisir une seule segmentation parmi toutes les segmentations en mots possibles. Ainsi, si le texte en entrée est « ABCDEFG » et si le lexique contient les mots A, AB, B, BC, BCDEF, C, CD, D, DE, E, F, FG et G, alors la segmentation exhaustive produira le document Sumo de la figure 4.7 (toutes les relations ne sont pas indiquées afin de ne pas surcharger la figure).

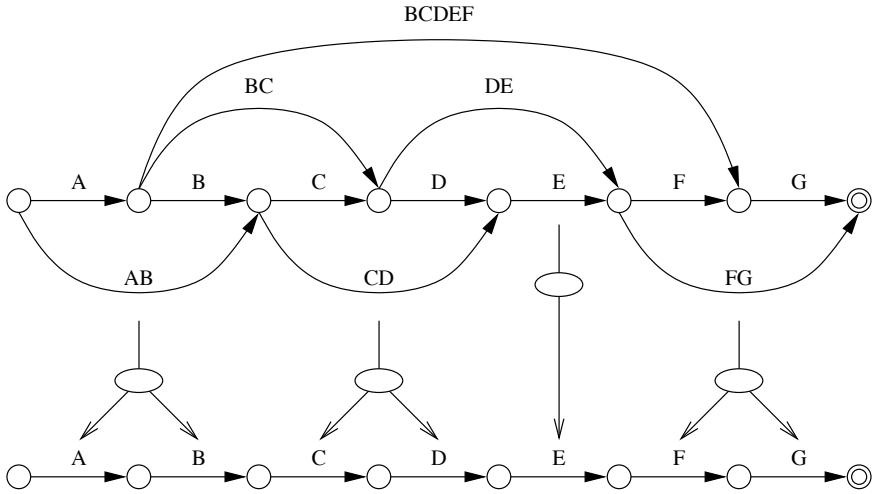

Fig. 4.7 – Segmentation exhaustive de « ABCDEFG »

Une désambiguïsation totale ou partielle peut se faire de plusieurs façons dans Sumo. En général, on définit une fonction de désambiguïsation ; celle-ci, appliquée à un graphe donné, produit un nouveau graphe ne contenant que la ou les segmentations désirées. Cette fonction est ensuite appliquée au graphe d'items dans le document pour ne garder que la forme désambiguïsée, mais on peut vouloir conserver toutes les ambiguïtés et appliquer cette fonction à la demande.

On détaille ici plusieurs types de fonctions de désambiguïsation correspondant à des techniques vues dans la partie précédente, mais exprimées désormais en Sumo. Ce sont les heuristiques de plus longue forme, ainsi que les modèles statistiques fondés sur le unigrammes et les bigrammes (qui peuvent

se généraliser aux -grammes, même s'il semble que l'on rencontre peu de trigrammes ou autres dans la pratique).

4.3.2.1 Heuristiques de plus longue forme

Si jusqu'à présent on a travaillé avec des graphes entiers, la désambiguïsation nécessite que l'on s'intéresse aux constituant de ces graphes : sommets (`node`), items (`item`) et chemins (`path`), qui sont des séquences d'items.

Heuristique de la plus longue chaîne. Cette heuristique est programmée de la manière suivante : partant d'un sommet donné (on commencera évidemment par le sommet initial du graphe), on choisit parmi tous les items qui ont ce sommet pour origine celui qui est le plus long.

On voit ci-dessous la définition dans le langage de Sumo d'une fonction récursive `ft` (pour *forward tokenization*) qui, pour un sommet donné, `n`, construit un chemin (un tableau d'items, `array item`) en sélectionnant l'item le plus long (lignes 5 à 12) et renvoie le chemin résultant de la concaténation de l'item le plus long, `longest`, et du chemin partant du sommet destination de cet item.

Si le sommet est final, on a trouvé une segmentation : le résultat renvoyé est un chemin vide, `[]`. Le chemin est construit récursivement. Cet algorithme est simplifié et suppose que le graphe est connecté, qu'il n'a qu'un état initial et qu'un état final.

```
 1  defun ft (n : node) returns array item {
 2    if (n.final) do {
 3      return [] ;
 4    } else {
 5      longest : item ;
 6      longest = n.items[1] ;
 7      foreach a in n.items[2..] do {
 8        it : item ;
 9        if (it.length > longest.length) do {
10          longest = it ;
11        }
12      }
13      return [longest, ft(longest.dest)] ;
14    }
15  }
```

Heuristique de plus longue chaîne en parcours inverse. Cette heuristique se programme de la même façon, en partant cette fois-ci du sommet final du graphe pour atteindre le sommet initial. On peut également renverser le graphe (en échangeant états initiaux et états finals, et en inversant les extrémités de chaque arc), appliquer la fonction `ft` vue plus haut, et renverser le chemin obtenu pour avoir le même résultat.

Heuristique du plus petit nombre d'items. Cette heuristique favorise les chemins comportant le plus petit nombre d'arcs, donc d'items : en moyenne, ceux-ci sont donc de longueur maximale. L'algorithme présenté ici est un algorithme classique de parcours de graphe et montre une autre construction de Sumo : la liste (une généralisation des chemins, car une liste contient n'importe quel type d'élément).

On montre ci-dessous la définition de la fonction `st` (pour *shortest tokenization*) en Sumo ; celle-ci repose sur une fonction de tri topologique (`t_sort`, ligne 7) que l'on ne détaille pas ici. À vrai dire, il s'agit ici d'un simple algorithme cherchant le plus court chemin dans un graphe en maintenant un ensemble de distances (nombre d'items entre un sommet et l'origine) et de prédécesseur pour chaque sommet (l'ensemble `d`).

`g.start` et `g.end` représentent le sommet initial et le sommet final d'un graphe `g`.

```
 1  defun st (g : graph) returns array item {
 2    d : set (nd : node, dist : number, prec : node) ;
 3    foreach n in g.nodes do {
 4      add(d, {nd = n, dist = number.max}) ;
 5    }
 6    foreach n in t_sort(g.nodes) do {
 7      foreach it in n.items do {
 8        if (d{nd == it.dest}).dist > (d{nd == n}).dist + 1 {
 9          (d{nd == it.dest}).dist = (d{nd == n}).dist + 1 ;
10          (d{nd == it.dest}).prec = n ;
11        }
12      }
13    }
14    n = g.end ;
15    sp : array item;
16    sp = [n] ;
17    while (n != g.start) do {
18      n = (d{nd == n}).prec ;
19      unshift(sp, n) ;
20    }
21    return sp ;
22  }
```

4.3.2.2 Modèle d'unigrammes

Outre les heuristiques de plus longue forme, on peut employer un modèle statistique semblable à celui de [Sproat *et al.*, 1996], c'est-à-dire un modèle d'unigrammes : à chaque entrée du dictionnaire est associé un poids dans le demi-anneau tropical, et la meilleure segmentation est alors le chemin dont la somme des poids est optimale, donc la plus petite.

La réalisation de ce modèle dans Sumo est des plus simples grâce aux graphes d'items pondérés. Ainsi, dans le dictionnaire de segmentation, chacune des formes lexicales (chaque item au niveau des mots) se voit ainsi attribuer un poids. Lors de la phase d'itémisation, les items ainsi créés dans le graphe de mots auront pour poids le poids de l'item lexical provenant du dictionnaire.

Dans Sumo, il est prévu d'extraire le chemin de poids optimal dans un graphe d'item grâce à un algorithme de Viterbi. Il est également possible de trier les chemins selon leur poids ; ainsi, plutôt que de désambiguïser totalement un graphe d'items, on peut choisir les n meilleurs chemins, ou tous les chemins dont le poids est inférieur à un seuil donné, ou toute autre combinaison.

4.3.2.3 Modèles de bigrammes

Si les automates d'états finis pondérés se prêtent particulièrement bien au modèle d'unigrammes vu plus haut, ce n'est pas le cas pour un modèle de bigrammes, où une structure de treille est plus adaptée (comme c'est le cas pour Chasen). Ainsi, le « coût de connectivité » est exprimé entre par un arc entre deux items, qui sont des sommets dans la treille.

Dans Sumo, il faut procéder de manière légèrement différente ; cependant, la méthode proposée ici est généralisable à n'importe quel type de n-grammes. Soit un item X pouvant être suivi de deux items, Y et Z. On note $k_{X,Y}$ le coût de connectivité entre X et Y et $k_{X,Z}$ le coût de connectivité entre X et Z. Entre deux items, on peut alors introduire une epsilon-transition dont le poids est exactement le coût de connectivité entre les deux items (figure 4.8). Ces epsilon-transitions peuvent ensuite être

supprimées (comme on le verra dans le chapitre 7) ; les poids individuels des items n'auront alors plus réellement de signification, mais les chemins auront toujours un poids correct.

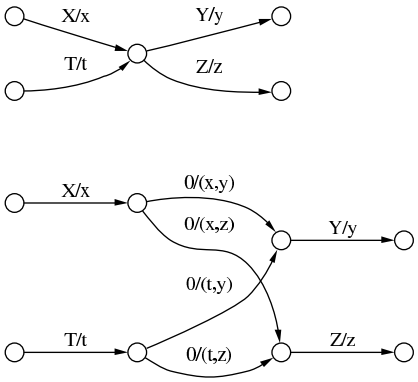

FIG. 4.8 – Coût de connectivité entre deux items

Le coût de connectivité de deux items est une fonction de ces deux items. Dans Chasen, il dépend de la catégorie syntaxique de chaque item et est donné par une matrice de coût. Quelle que soit la manière dont le coût de connectivité est calculé, on peut programmer une fonction `cout` (qui consulte une table par exemple) qui prend pour argument deux items et retourne un item ϵ avec le bon coût. Il reste maintenant à parcourir le graphe et à insérer un nouvel item entre chaque couple d'item.

L'introduction de ces epsilon-transitions se fait à un nouveau niveau, encore une fois à l'aide de règles d'identification. La règle utilisée est celle-ci :

```
0 => 0/$k || $x _ $y
  { $x = . ;
    $y = . ;
    $k = cout($x@cat, $y@cat); } ;
```

Cette règle est une règle d'insertion. `"=>"` est l'opérateur de règle (règle d'identification avec variables), `"||"` est le séparateur de contexte, et la partie entre accolades est la partie d'affectation des variables. Il y a donc quatre parties dans cette règle :

1. le chemin à identifier dans le graphe source : epsilon ;
2. le chemin correspondant dans le graphe cible : un epsilon-item avec le poids `$k` ;
3. le contexte dans le graphe source : `$x _ $y`) ;
4. les affectations : `$x` et `$y` correspondent à un item quelconque ; `$k` est le résultat de l'application de la fonction `cout` qui prend comme argument l'attribut `cat` (catégorie syntaxique) de l'item de gauche et de l'item de droite et retourne le coût correspondant (en consultant une matrice par exemple).

4.3.3 La construction des ressources

L'exemple que l'on étudie ici repose sur un dictionnaire de segmentation au format Sumo. On montre ici deux variantes : tout d'abord, on part d'un transducteur lexical existant ; ensuite, si un tel transducteur n'existe pas, on le crée directement avec Sumo.

4.3.3.1 La récupération d'un transducteur lexical existant

Première étape: lecture du transducteur lexical. La première étape consiste à lire le transducteur lexical existant (créé avec XFST ou les outils AT&T par exemple) à partir d'un fichier. On introduit une variable de type `wfst` que l'on appelle `translex` que l'on initialise par la fonction `read_wfst`.

```
translex : wfst ;
translex = read_wfst("lexique.wfst") ;
```

Deuxième étape: transformation du transducteur lexical en structure Sumo. Le transducteur est « explosé » par application de l'opérateur `"<>"` qui transforme un `wfst` en une structure Sumo à deux niveaux. Cette conversion est illustrée par la figure 4.9.

```
lexique : sumo ;
lexique = < $translex > ;
lexique.level[1].name = "surface" ;
```

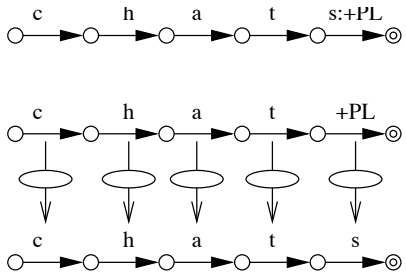

FIG. 4.9 – Conversion d'un transducteur en structure Sumo

Troisième étape: création du niveau des mots. On obtient le niveau des mots (où chaque item est un lemme muni de ses attributs morphologiques) à partir du niveau lexical de la structure Sumo (le niveau 2) par deux groupes de règles appliqués en séquence.

Le premier groupe de règles transforme les symboles représentant les étiquettes morphologiques comme `+PL` en items à l'intitulé vide portant un attribut correspondant à cette étiquette.

Le second groupe est réduit à une règle qui identifie tous les chemins du dernier niveau et crée pour chacun un nouvel item obtenu par compression de ce chemin.

```
g : grammar ;
g = (+PL -> 0@nbr=pl ,,
     +SG -> 0@nbr=sg ,,
     ... ) %
    $x  => [$x] || # _ # { $x = .+ } ;
apply(lexique.level[2], g) ;
```

Suppression du niveau intermédiaire. On a maintenant quatre niveaux dans notre lexique, mais seul les niveaux 1 et 4 sont intéressants. On supprime ces deux niveaux intermédiaire par l'opérateur de suppression `",-"`.

```
lexique = $lexique,-2 ;
lexique = $lexique,-2 ;
```

Après la première suppression, le niveau 3 est devenu le niveau 2 et le niveau 4 est devenu le niveau 3. Le lexique final contient des entrées de la forme illustrée par la figure 4.10.

Sauvegarde du lexique Sumo. Enfin, il ne reste plus qu'à sauver le lexique pour pouvoir le réutiliser dans d'autres applications Sumo.

```
write_sumo(lexique, "lexique.sumo");
```

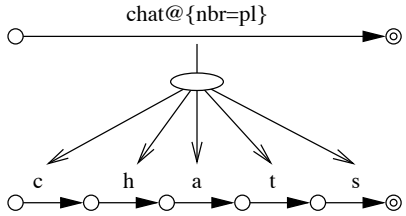

FIG. 4.10 – Un mot extrait du lexique Sumo

4.3.3.2 Création du transducteur lexical dans Sumo

Plutôt que de récupérer un transducteur existant, il est bien entendu possible de le créer à partir de zéro dans Sumo. Contrairement aux outils XFST, Sumo ne propose pas de raccourcis particuliers pour la description de lexiques, aussi utilise-t-on la syntaxe d'expressions régulières de Sumo.

Listes de mots et morphosyntaxique. La définition d'un lexique passe entre autres par la définition des différentes formes - morphes, affixes, tournures et expressions à mots multiples, entités -, que l'on regroupe habituellement en sous-lexiques de taille plus pratique.

Dans Sumo, un sous-lexique n'est rien d'autre qu'un transducteur que l'on affecte à une variable. Ainsi, un lexique où chaque mot est formé d'une racine précédée de zéro ou plusieurs préfixes et suivie d'un ou plusieurs suffixes a la forme suivante :

```
prefixes, suffixes, racines, lexique : wfst ;
prefixes = <pré> | <post> | ... ;
suffixes = <ation> | <é> | ... ;
racines  = <compil> | <déterminis> | ... ;
lexique  = $prefixes* $racines $suffixes* ;
```

Les trois sous-lexiques sont chacun décrits par une expression régulière et le lexique lui-même est décrit en fonction de ces sous-lexiques. La compilation de chaque expression se fait normalement pour produire les transducteurs pondérés.

Il est courant, comme le suggère d'ailleurs l'exemple ci-dessus, qu'un lexique ne soit guère plus qu'une longue liste de formes (séparées par `|` dans une expression régulière). Un outil comme `lexc` dispose d'un compilateur qui traite ce genre d'expressions beaucoup plus efficacement qu'un compilateur général. Dans Sumo, comme on le verra dans la section 7.1.3.3 une méthode simple de compilation de « dictionnaires » est disponible pour compiler plus rapidement de tels lexiques.

Altérations phonologiques. Les altérations phonologiques qui ne sont pas directement prises en compte par le lexique peuvent ensuite être exprimées sous forme de règles de réécriture pondérées, qui donnent naturellement des transducteurs [Mohri et Sproat, 1996; Sproat et Riley, 1996; Adant, 2000] qui sont composés avec les lexiques pour obtenir le transducteur lexical final.

Sumo ne propose pas de règles à deux niveaux du style PC-KIMMO ou `twolc`, c'est-à-dire de règles qui permettent d'exprimer le contexte sur les deux niveaux en même temps ; cependant, c'est une extension envisageable. [Adant, 2000] décrit d'ailleurs brièvement l'inclusion de règles à deux niveaux dans un système d'états finis pondérés.

Gestion des attributs. Traditionnellement, dans les modèles d'états finis « purs », les attributs morphologiques sont encodés sous formes de symboles spéciaux (*e.g.*`+Fem`, qui constitue un unique symbole). Dans Sumo, on peut pour cela des items vides, ayant chacun un attribut correspondant à un de ces symboles particuliers, comme on l'a vu dans la section 4.3.3.1.

Conclusion

Ce chapitre a donné une vue d'ensemble, partielle mais représentative, du formalisme Sumo et de son utilisation pour la création d'un moteur d'analyse morphologique. Les principales composantes de Sumo ont été décrites : ce sont les structures de données pour la segmentation (principalement, les transducteurs d'états finis pondérés et les structures Sumo elles-mêmes) et les moyens dont on dispose pour travailler avec ces structures. Il y a donc un véritable formalisme d'expressions régulières décrivant les transducteurs pondérés, des opérations « rationnelles » et d'autres plus spécifiques sur les structures Sumo, et enfin le langage de description de moteurs d'analyse lui-même.

Comme on l'a vu dans ce chapitre, et ce que confirmera le chapitre suivant, les différentes composantes de Sumo permettent de travailler avec la structure Sumo selon plusieurs points de vue différents. Ainsi, les expressions régulières et les règles d'identification et de réécriture sont des outils de nature « linguistique », qui agissent sur des symboles, des mots et des langages, qui ne nécessitent pas de connaissance approfondie des structures de représentation sous-jacentes. Cela rejoint la plupart des outils de morphologie à états finis. Mais Sumo dispose de structures plus complexes et plus ouvertes qui sont construites et utilisées par des outils informatiques, qui eux agissent sur des arcs, des relations ou des graphes.

Chapitre 5

Syntaxe et sémantique formelle de Sumo

Introduction

Après avoir donné au chapitre précédent une spécification générale de Sumo, nous nous proposons de définir plus formellement les différents aspects du formalisme, à la fois syntaxiques et sémantiques, pour les deux structures principalement manipulées par Sumo : les transducteurs d'états finis pondérés et les structures Sumo, que ce soient des documents ou des ressources. Les deux premières sections de ce chapitre sont consacrées à ces deux structures (sections 5.1 et 5.2) ; on revient ensuite sur les structures de contrôle qu'offre Sumo aux linguistes et programmeurs pour la réalisation d'une application présyntaxique (section 5.3).

Dans ce chapitre, on utilise un méta-langage pour la description de la syntaxe des divers composants de Sumo où :

- une expression en *italique* indique un non-terminal ;
- une expression entre `"guillemets"` indique un terminal du langage Sumo ;
- les parenthèses () délimitent une sous-expression ;
- le point d'interrogation ? indique une partie optionnelle ;
- l'étoile * indique une partie qui peut être répétée 0, 1 ou plusieurs fois ;
- le symbole | indique une alternance.

5.1 Le calcul d'états finis pondéré

Dans Sumo, toute machine d'états finis est considérée comme un transducteur d'états finis pondérés (WFST, pour *Weighted Finite-State Transducer*). La définition d'un WFST est très proche de celle d'un graphe d'items (section 4.1.2.3) et repose toujours sur un ensemble de pondération $\mathbb{K}$ (comme d'habitude, on suppose que $\mathbb{K}$ est le demi-anneau tropical ; mais il peut s'agir de n'importe quel demi-anneau fermé).

Définition. Un transducteur d'états finis pondéré (WFST) T est un octuplet $(\Sigma, \Omega, Q, I, F, E, \lambda, \rho)$ avec :

- Σ l'alphabet du langage supérieur ;
- Ω l'alphabet du langage inférieur ;
- Q un ensemble fini d'états ;
- $I \subseteq Q$ l'ensemble des états initiaux ;
- $F \subseteq Q$ l'ensemble des états finals ;
- $E \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Omega \cup \{\epsilon\} \times \mathbb{K} \times Q$ un ensemble fini de transitions ;
- $\lambda : I \to \mathbb{K}$ la fonction de pondération initiale ;
- $\rho : F \to \mathbb{K}$ la fonction de pondération finale.

Ainsi, dans un WFST, les arcs entre deux états sont étiquetés par deux symboles (un symbole du langage supérieur, et un du langage inférieur) et possèdent un poids. Un chemin est une séquence d'arcs consécutifs reliant un état initial $i \in I$ à un état $f \in F$; un sous-chemin est une séquence d'arcs consécutifs reliant deux états quelconques de Q . Un chemin définit ainsi une transduction pondérée entre un mot du langage supérieur et un mot du langage inférieur ; le poids de la relation est le produit des poids des arcs, de l'état initial et de l'état final (on rappelle que dans le demi-anneau tropical, l'opération de multiplication est + : ainsi, les poids sont en réalité ajoutés). Si plusieurs chemins correspondent à la même relation dans le transducteur, le poids de la relation est la somme de tous les poids dans le transducteur (donc, le poids minimum dans le demi-anneau tropical).

Un automate d'états finis est alors un cas particulier d'un transducteur ; il a la propriété d'être simple (c'est-à-dire que toutes ses étiquettes sont simples). De même, un transducteur peut ne pas être pondéré : par défaut, un arc a un poids de $\overline{1}$ (poids unitaire), et un transducteur pondéré dont tous les poids sont $\overline{1}$ est équivalent à un transducteur classique.

On présente ici les principales opérations rationnelles sur les transducteurs, qui sont directement héritées des opérations sur les transducteurs « traditionnels » ; la section 5.1.2 s'intéresse cependant aux différences qu'amènent ces pondérations. Enfin, on dote le formalisme de règles de réécriture pondérées, présentées dans la section 5.1.3.

5.1.1 Les opérations rationnelles classiques

On donne ici la syntaxe et la sémantique des opérations classiques sur les WFST.

5.1.1.1 Symboles des WFST

La table 5.1 présente la syntaxe des symboles d'un WFST dans Sumo. Ce sont les symboles terminaux (avec les opérateurs, décrits plus bas) des expressions régulières dans Sumo.

Expression	Symbole	Notes
`a`	`a`	Symbole simple, poids unitaire
`0`	ϵ	Le symbole vide, noté `0`
`.`	$\perp$	Le symbole par défaut, noté `.`
`a\ b`	`a b`	Le caractère \ est un caractère d'échappement
`\0`	`0`	Le symbole 0
`\.`	`.`	Le symbole .
`\\`	`\`	Le symbole \
`"a b"`	`a b`	On peut également écrire `'abc'` (guillemets simples)
`\u00e6`	æ	\u*xxxx* désigne un caractère Unicode par son code UCS-2
`a:b`	`a:b`	Symbole complexe, poids unitaire
`a:`	`a:.`	Raccourci
`:a`	`.:a`	Idem
`a/.5`	`a`	Symbole `a` avec un poids de 0,5

TAB. 5.1 – Syntaxe des symboles

La syntaxe normale d'un symbole est une simple chaîne de caractères (par exemple, `a`). Cette chaîne ne doit pas comporter de métacaractère, c'est-à-dire de caractère à l'interprétation particulière : espacement (qui est un séparateur), opérateur, etc.

Deux symboles sont particuliers : "`0`" est le symbole vide (noté ϵ dans les manuels), et le symbole "`.`" est le symbole « universel » qui correspond à n'importe quel symbole de Σ .

Pour rendre l'écriture de symboles particuliers possible, il est possible d'utiliser le caractère d'échappement "`\`" ou d'employer les *quotes*, simples ou doubles. Enfin, le métacaractère "`\u`" permet de spécifier n'importe quel caractère Unicode par son code UCS-2 (en hexadécimal).

Le séparateur "`:`" permet de définir un symbole complexe, c'est-à-dire un couple (symbole du langage supérieur, symbole du langage inférieur). Les deux symboles sont définis selon les règles ci-dessus.

Enfin, le séparateur "`/`" suit un symbole simple ou composé et donne un poids à l'arc qui porte ce symbole. La syntaxe exacte du poids dépend de l'ensemble de pondération choisi : pour le demi-anneau tropical ou les réels, c'est un réel ; pour le demi-anneau booléen c'est uniquement 0 ou 1.

5.1.1.2 Les opérations sur les WFST

La table 5.2 décrit les opérateurs rationnels de Sumo. Certains sont détaillés dans la section suivante ; les autres sont classiques. Dans cette table, `A` et `B` sont deux expressions régulières quelconques.

Opérateur	Nom	Automates	Transducteurs
`A B`	Concaténation	oui	oui
`!A`	Complément	oui	non
`A % B`	Composition	oui	oui
`A %+ B`	Composition itérative	oui	oui
`A & B`	Intersection	oui	non
`A*`	Fermeture transitive	oui	oui
`A+`	Fermeture positive	oui	oui
`A,1`	Projection inférieure	oui (identité)	oui
`A,u`	Projection supérieure	oui (identité)	oui
`A - B`	Différence	oui	non
`A : B`	Produit cartésien	oui	non
`A?`	Optionnalité	oui	oui
`^A`	Inversion	oui (identité)	oui
`A | B`	Union	oui	oui
`~A`	Renversement	oui	oui

Tab. 5.2 – Opérateurs rationnels

Donnons quelques détails sur ces opérateurs :

Concaténation. La concaténation de `A` et de `B`.

Complément. Le langage complément de celui reconnu par `A` : $L(!A) = \Sigma^* - L(A)$. Le complément n'est défini que sur les transducteurs simples.

Composition, composition itérative. Voir plus loin.

Intersection. L'intersection des langages reconnus par `A` et `B`, c'est-à-dire tous les mots reconnus à la fois par `A` et par `B`. L'intersection n'est définie que pour les transducteurs simples.

Répétition. Il y a trois opérations de répétition : `A*` (fermeture transitive, 0 ou plusieurs occurrences), `A+` (fermeture positive, 1 ou plusieurs occurrences) et `A?` (optionnalité, 0 ou 1 occurrences).

Projection. Il y a deux sortes de projection : la projection inférieure, `A,l` (l pour *lower*), qui donne l'automate reconnaissant le langage inférieur du transducteur, et la projection supérieure, `A,u` (u pour *upper*), qui donne l'automate reconnaissant le langage supérieur du transducteur. La projection est définie pour les transducteurs simples, mais est évidemment sans effet.

Différence. La différence des langages reconnus par `A` et `B`. L'égalité suivante est toujours vraie : `A - B` = `A &!B`.

Produit cartésien. Voir plus loin.

Inversion. Échange les niveaux inférieurs et supérieurs du transducteur. Sans effet sur les automates.

Union. L'union des langages reconnus par `A` et `B`.

Renversement. Reconnaît le langage miroir de `A`.

5.1.1.3 Les expressions régulières

Pour définir une expression régulière dans Sumo, on ajoute simplement deux constructions :

- les parenthèses, `()`, qui permettent de marquer des sous-expressions ;
- la référence vers une variable de Sumo, introduite par `"$"`.

Deux sortes d'éléments peuvent être introduits dans une expression régulière par le biais de `"$"` : une chaîne de caractères (qui est traitée comme un symbole), et un autre WFST, lui-même compilé à l'aide d'une expression régulière. On peut ainsi décomposer une expression en expressions plus simples, réutiliser des éléments communs à plusieurs expressions, ou encore paramétrer une expression.

Début et fin d'une chaîne. Certaines formes d'expressions régulières, par exemple celles de `grep` ou de Perl, servent à des décrire des motifs qui apparaissent à l'intérieur de chaînes. Dans ce cas, il est nécessaire d'avoir un moyen d'indiquer que l'on cherche à reconnaître le début ou la fin d'une chaîne. Dans Sumo, les expressions régulières décrivent exactement les chaînes reconnues ; ces caractères spéciaux ne sont alors pas nécessaires (ils le deviendront cependant dans les règles de réécriture).

Exemple. Le motif `/foo/` en Perl correspond à `.* f o o .*` en Sumo.

5.1.2 Opérations spécifiques

5.1.2.1 Déterminisation et minimisation

Dans le cas simple des automates d'états finis (sans pondération), il est connu qu'un automate non-déterministe est toujours déterminisable : il existe au moins un automate déterministe qui lui est exactement équivalent, et l'on dispose d'algorithmes efficaces pour construire cet automate déterministe. Le cas des transducteurs est plus complexe, car un transducteur est déterminisable si l'on considère l'automate sous-jacent, mais peut rester ambigu si la relation qu'il reconnaît l'est.

On dit d'un transducteur qu'il est fonctionnel, ou non-ambigu, si une chaîne du langage supérieur ne peut être mise en relation qu'avec une unique chaîne du langage inférieur. Dans le cas contraire, le transducteur est ambigu. Un transducteur peut être globalement non-ambigu tout en admettant toujours une forme d'ambiguïté locale si l'automate obtenu par projection supérieure n'est lui-même pas déterministe.

Un transducteur est dit séquentiel s'il n'admet aucune ambiguïté locale, c'est-à-dire si l'automate obtenu par projection supérieure est déterministe. L'avantage d'un transducteur séquentiel est, comme pour les automates déterministes, que l'application du transducteur à une chaîne se fait en un temps linéaire (proportionnel à la longueur de la chaîne), car il n'y a pas de retour arrière (voir section 5.1.2.3).

Un algorithme de déterminisation de WFST découle simplement de l'algorithme classique fondé sur les sous-ensembles dans le cas des automates d'états finis classiques [Aho *et al.*, 1986]. La seule modification majeure réside dans le calcul du poids des transitions et des poids résiduels pour chaque transition de l'automate non-déterministe [Mohri, 1997]. Cette modification a cependant un impact déterminant, car l'algorithme peut continuer indéfiniment pour certains types de transducteurs. Ainsi, tous les transducteurs d'états finis pondérés ne sont pas déterminisables. [Mohri, 1997] donne un critère de déterminisabilité pour les transducteurs non-ambigus ; mais certains transducteurs ambigus peuvent également être déterminisés.

La minimisation du nombre d'états d'un transducteur ne pose pas de problème particulier ; Sumo emploie l'algorithme de Brzozowsky, qui repose sur l'égalité suivante, où « Rev » dénote le renversement du transducteur, et « Det » sa déterminisation.

$$Min(T) = Rev(Det(Rev(Det(T))))$$

De cette égalité, il découle qu'un transducteur d'états finis pondéré est minimisable si et seulement si il est déterminisable. L'algorithme lui-même est alors trivial puisque les opérations de renversement et de déterminisation sont connues.

5.1.2.2 Composition et produit cartésien

La composition de deux transducteurs est une opération fondamentale du calcul à états finis. Soient deux transducteurs T_1 défini sur les alphabets Σ et Ψ , et T_2 défini sur les alphabets Ψ et Ω ; alors la composition de ces deux transducteurs donne un nouveau transducteur T défini sur les alphabets Σ et Ω , mettant en relation directement le langage supérieur de T_1 et le langage inférieur de T_2 . Par composition, une séquence de transducteurs peut se réduire à un unique transducteur effectuant toutes les mises en relation en une seule étape. Un inconvénient est que, dans certains cas, le résultat de la composition peut être beaucoup plus gros que les deux transducteurs composés.

La composition de deux transducteurs est une opération rationnelle classique, mais dans le cas de transducteurs pondérés, les epsilon-transitions sont source d'ennuis, créant des chemins inutiles et surtout menant à des poids incorrects. [Mohri *et al.*, 2000] propose un algorithme de composition pour des transducteurs ne comportant aucune epsilon-transition. Pour que cet algorithme soit utilisable pour n'importe quel transducteur, y compris ceux comprenant des epsilon-transitions, le symbole ϵ est remplacé par deux symboles spéciaux, ϵ_u et ϵ_l représentant un ϵ du côté supérieur ou inférieur d'une transition. Un filtre de composition est alors introduit pour effectuer la conversion de ces symboles spéciaux entre les deux transducteurs. La figure 5.1 montre une version légèrement modifiée de ce filtre.

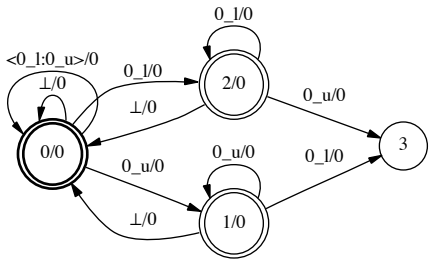

FIG. 5.1 – Filtre de composition

L'utilisation de ce filtre nécessite un prétraitement des opérandes :

1. suppression des epsilon-transitions dans les deux transducteurs ;

2. remplacement de toutes les occurrences de ϵ du côté *inférieur* du transducteur supérieur par le symbole spécial ϵ_l ;
3. ajout d'une transition étiquetée par ϵ_u depuis chaque état du transducteur supérieur vers lui-même ;
4. remplacement de toutes les occurrences de ϵ du côté *supérieur* du transducteur inférieur par le symboles spécial ϵ_u ;
5. ajout d'une transition étiquetée par ϵ_l depuis chaque état du transducteur inférieur vers lui-même.

Dans l'exemple de Mohri et al., on compose le transducteur reconnaissant la relation `a b:`ϵ `c:`ϵ `d` avec la relation `a:d` ϵ `:e d:a`. La figure 5.2 montre les deux transducteurs après l'étape de prétraitement, prêts à être composés *via* le filtre de composition.

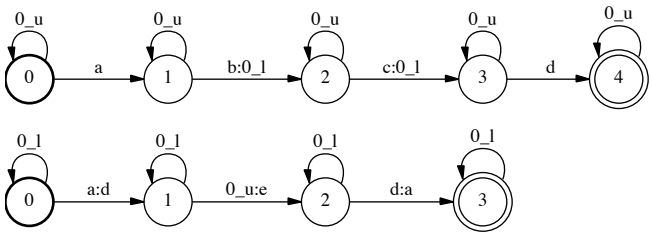

FIG. 5.2 – Prétraitement des automates avant la composition

La composition s'effectue en deux étapes : premièrement, composition du transducteur supérieur avec le filtre, puis composition du résultat avec l'automate inférieur. Le résultat doit ensuite être post-traité en supprimant toute occurrence des symboles spéciaux introduits durant le prétraitement.

Un algorithme de composition avec filtre. Sumo utilise un algorithme générique de composition, utilisé pour quatre opérations : la composition elle-même, la composition itérative (décrite plus bas), le produit cartésien, et l'intersection. Il s'agit d'un algorithme de composition ignorant les epsilon-transitions et mettant en jeu un filtre. La composition s'opère en deux étapes : premièrement, le transducteur supérieur est composé avec le filtre ; deuxièmement, le résultat est composé avec le transducteur inférieur.

L'algorithme prend bien sûr en compte les poids des transitions : lorsque trois transitions sont composées, le poids de la nouvelle transition est le « produit » du poids des trois transitions, et de même, le poids des nouveaux états initiaux et finals créés est le « produit » des poids des états composés. Il est intéressant de noter que cet algorithme est tout aussi utilisable avec des transducteurs non-pondérés, car le filtre permet d'éviter la création de chemins tels que `a:`$\epsilon\epsilon$ `:b` pour obtenir directement `a:b`.

L'intersection de deux automates est exactement la composition de ces deux automates.

L'implémentation de cet algorithme est présentée chapitre 7. Pour deux transducteurs donnés T_1 et T_2 et un filtre T_f , un nouveau transducteur R est créé. Les états de R correspondent à des triplets d'états de chaque automate ; on maintient une queue Q pour le parcours de tous les états en l'initialisant avec les états initiaux de R qui sont les états initiaux des trois automates. De même, un état « composite » est final si les trois états auquel il correspond le sont.

On note $e \approx f$ la correspondance de deux arcs. Deux arcs se correspondent si l'étiquette inférieure de l'un correspond à l'étiquette supérieure de l'autre. Notons que si l'une des étiquettes est $\perp$, la correspondance est toujours vraie. Le deuxième test de correspondance, ligne 13, est légèrement différent : si l'étiquette inférieure de e_f est $\perp$, alors il faut en réalité tester $e_1 \approx e_2$.

Enfin, ligne 21, on crée un nouvel arc entre q et q' ; son poids est le produit des poids des trois arcs, et son étiquette est construite à partir de celles des trois arcs : l'étiquette supérieure est l'étiquette supérieure de e_1 , et l'étiquette inférieure est l'étiquette inférieure de e_2 .

Le produit cartésien. Le produit cartésien de deux automates est également proche de la composition, et peut même être réalisé, comme l'intersection, avec cet algorithme général. La seule différence est que l'on utilise un nouveau filtre (figure 5.3) et un nouveau symbole particulier, ϵ_x .

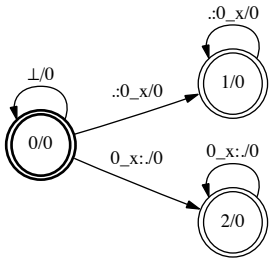

FIG. 5.3 – Filtre pour le produit cartésien

La figure 5.4 montre le prétraitement de deux automates reconnaissant les langages `a*b` et `cd*`, afin de réaliser le produit cartésien `ab* : c*d`. L'automate supérieur est transformé en transducteur dont toutes les transitions ont un ϵ_l au niveau inférieur ; de la même manière, l'automate inférieur est transformé en transducteur dont toutes les transitions ont un ϵ_u au niveau supérieur. Le filtre permet donc d'effectuer la relation entre n'importe quel couple de transitions. On remarquera également qu'une transition étiquetée par ϵ_x est ajoutée à chaque état final, afin de pouvoir apparier des chaînes de longueur différente.

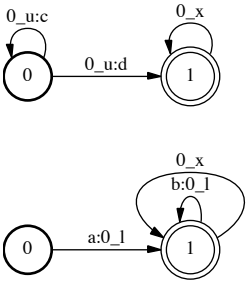

FIG. 5.4 – Prétraitement des automates avant le produit cartésien

La figure 5.5 montre le résultat du produit cartésien, une fois le transducteur « nettoyé ».

La composition itérative et l'équivalence des transducteurs. La composition itérative est une opération qui sort un peu du cadre habituel des états finis, car comme son nom l'indique, c'est une opération qui s'applique itérativement jusqu'à atteindre un point fixe. Soit deux transducteurs T_1 et T_2 ; les deux transducteurs sont composés, puis le résultat est de nouveau composé avec T_2 , puis le résultat est de nouveau composé avec T_2 et ainsi de suite tant que deux compositions successives donnent un résultat différent. Si les deux dernières opérations donnent deux transducteurs équivalents, alors c'est le résultat final de l'opération.

Nous n'avons pas trouvé dans la littérature d'algorithme permettant de déterminer l'équivalence de deux transducteurs pondérés. Nous avons résolu ce problème en définissant l'équivalence entre états de transducteurs pondérés. Deux états sont équivalents s'ils sont origine des mêmes chemins, avec le

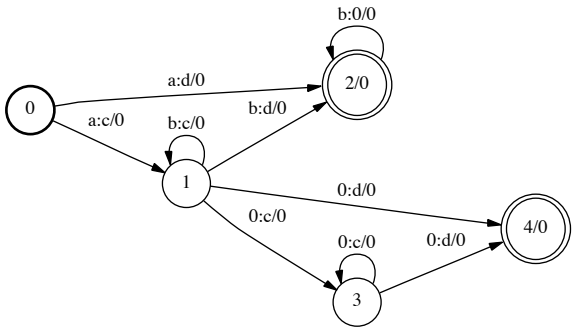

FIG. 5.5 – Produit cartésien de ab* et c*d

même poids, à un poids résiduel près (deux chemins peuvent avoir le même poids sans pour autant que les arcs eux-mêmes aient tous le même poids). L'algorithme utilisé est présenté chapitre 7.

5.1.2.3 Application d'un transducteur à une chaîne et poids d'un chemin

L'application d'un transducteur d'états finis pondéré à une chaîne se fait dans les deux directions : *lookup* pour l'application au niveau inférieur, donnant les correspondants au niveau supérieur, et inversement pour *lookdown*. Cette opération produit zéro, une ou plusieurs chaînes.

Si le transducteur est pondéré, on produit alors chaque chaîne une seule fois, avec son meilleur poids (par exemple le minimum dans le demi-anneau tropical). Mais, en général, on ne veut obtenir que la meilleure chaîne. C'est pourquoi on utilise alors une technique de programmation dynamique à la Viterbi. Ainsi, l'application d'un transducteur pondéré est en général plus rapide que celle du même transducteur non pondéré, car les calculs sur les poids sont beaucoup plus rapides que l'examen combinatoire de toutes les chaînes possibles.

5.1.3 Règles de réécriture pondérées

5.1.3.1 Syntaxe

Les règles de réécriture pondérées sont une extension des règles de réécriture classiques dans les formalismes d'états finis. Ces règles de réécriture décrivent une transduction rationnelle et effectuent donc une mise en correspondance entre des chaînes de deux langages rationnels. Chacun de ces deux langages est défini sur un alphabet qui lui est propre, mais ces deux alphabets sont tous deux compris dans un alphabet « universel ».

On considère les règles de la forme :

$\phi \rightarrow \psi / \lambda \_ \rho$

ce qui se lit : « une occurrence de ϕ précédée d'une occurrence de λ et suivie d'une occurrence de ρ se réécrit en ψ », avec ϕ , ψ , λ et ρ des expressions régulières pondérées. Ces règles sont directement inspirées des règles de réécriture en phonologie. Plusieurs de ces règles peuvent être appliquées en parallèle ou en séquence. On a donc une notion de grammaire décrites par des expressions dont les symboles sont des règles de réécriture et les opérateurs sont `",,"` pour l'application parallèle et `"%"` pour l'application séquentielle.

La syntaxe d'une règle dans Sumo est la suivante :

ϕ `"->"` ψ `"||"` λ `"_"` ρ

Les quatre expressions en paramètres suivent évidemment la syntaxe d'expressions régulières de Sumo et ne doivent décrire que des automates, et non des transducteurs. Le contexte gauche ou droit peut être vide, et dans le cas des règles inconditionnelles (qui s'appliquent quelque soit le contexte), toute la partie de description du contexte de la règle peut être omise (par exemple, la règle `a -> b` est plus simple à écrire que `a -> b || _`).

5.1.3.2 Sémantique

L'application d'un ensemble de règles parallèles à un automate peut se comprendre en considérant indépendamment chaque chemin. L'automate résultat est l'union du résultat de l'application des règles à chacun de ces chemins.

Pour un chemin donné, on applique les règles de gauche à droite. Une règle donnée peut s'appliquer si le membre gauche de la règle est bien présent dans le chemin dans le contexte décrit par la règle, auquel cas la partie non-contextuelle est « consommée » et mise en correspondance avec le membre droit dans la chaîne résultat. Tous les symboles non consommés sont recopiés tels quels. Plusieurs chemins peuvent être produits pour un seul chemin donné.

Par exemple, étant données les deux règles

```
a b -> e f || _ c ,, b c -> f g || a _ d;
```

et la chaîne en entrée `babcd`, on aura les deux résultats suivants :

```
b a b c d   b a b c d
| | | | |   | | | | |
b e f c d   b a f g d
```

Dans le premier cas, c'est la première règle qui s'applique pour réécrire `ab` en `ef` ; dans le deuxième cas, c'est la deuxième qui réécrit `bc` en `fg`.

5.1.3.3 Exemple de compilation et d'application

Un jeu de règles est compilé par Sumo sous la forme d'un transducteur. L'application de ce jeu de règles se fait par le biais de l'opérateur de composition `%` comme le montre cet exemple. Soit les deux règles parallèles suivantes décrivant le transducteur `$R` :

```
$R = a b -> e f ||   _ c ,,
     b c -> f g || a _ d
```

On voit le résultat de la compilation de ces règles dans la figure 5.6. Même si le transducteur est quelque peu complexe, on peut comprendre son principe de fonctionnement. Par exemple, dans l'état 0, quand on lit un `a`, il y a deux possibilités : soit ce `a` est le début de la séquence `abc`... qui doit être remplacé par `efc`..., auquel cas on va dans l'état 2 où il faut impérativement lire un `b` puis un `c` sous peine d'être bloqué et de devoir faire un retour arrière ; soit ce `a` ne fait pas partie d'une telle séquence et l'on passe dans l'état 1 (il se peut par contre que `a` soit le début d'une séquence `abcd` qui doit se réécrire en `afgd`).

Étant donné un ensemble de chaînes en entrée sous la forme d'un automate (figure 5.7), l'application du transducteur de règles produit la sortie attendue (figure 5.8) et s'écrit dans Sumo :

```
$entree = a b c d   | a n d   |
          b a b c d | b a n d |
          c b c d   | c n d   |
          m c d ;
$resultat = ($entree % $R),l ;
```

Alphabet universel pour l'application des règles

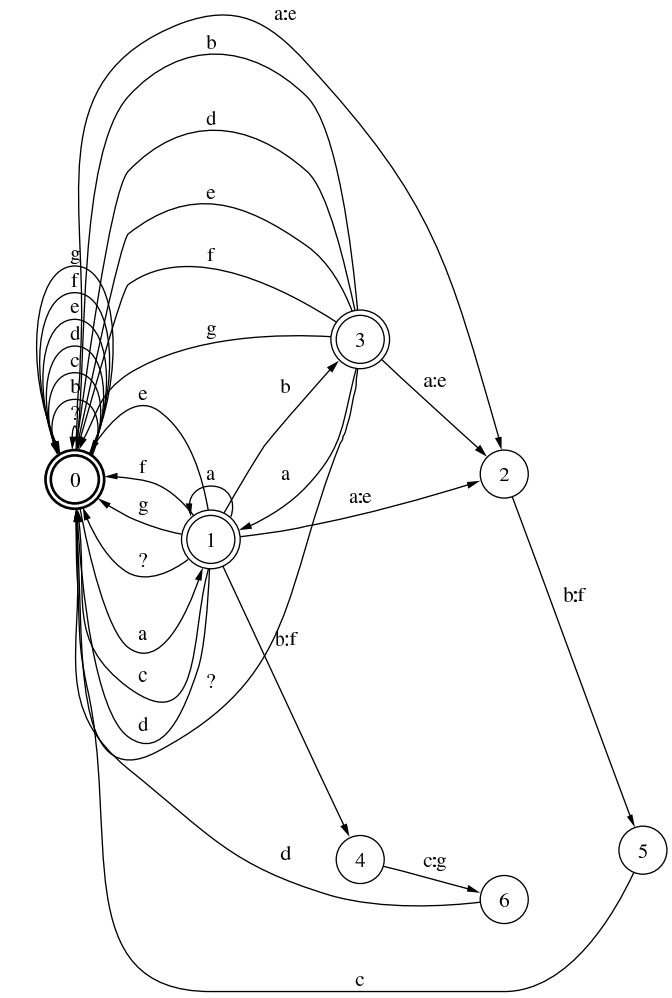

FIG. 5.6 – Le transducteur de règles R

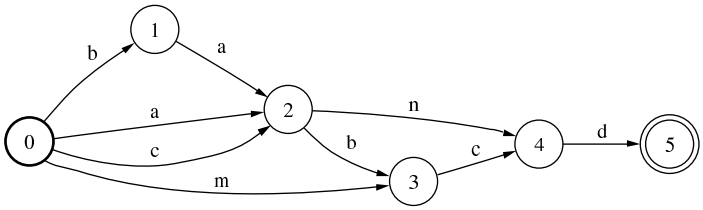

FIG. 5.7 – Automate d'entrée

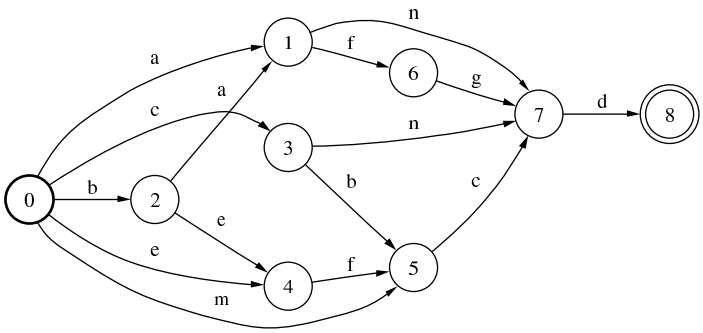

FIG. 5.8 – Automate résultat

5.1.3.4 Réalisation et variantes

On dispose d'algorithmes efficaces pour la compilation de jeux de règles de réécriture parallèles [Kaplan et Kay, 1994; Mohri et Sproat, 1996], pondérées ou non. Les règles séquentielles se compilent grâce à l'opération de composition déjà vue.

Nous avons ajouté des variantes existant dans la littérature et dans les autres systèmes d'états finis. Elles sont compatibles entre elles.

- l'opérateur `"->"` dénote, comme dit plus haut, une règle s'appliquant de gauche à droite ; cependant, une règle peut tout à fait s'appliquer de droite à gauche (`"<-"`) ou simultanément dans les deux directions (`"<->"`) ;
- une règle peut être optionnelle, c'est-à-dire qu'elle s'applique ou non à une chaîne. La règle

  ```
  b ->? d || a _ c
  ```

  appliquée à la chaîne `abc` produira donc deux chaînes, `adc` et `abc`..
- lorsque le membre gauche d'une règle décrit plusieurs chaînes, la règle s'applique à toutes les occurrences de cette chaîne. Par exemple, la règle

  ```
  a+ -> b
  ```

  appliquée à la chaîne `aaac` produira les trois chaînes `bc`, `abc` et `aabc`. [Karttunen, 1996] propose des règles « dirigées » qui remplacent uniquement l'occurrence la plus longue ou la plus courte. Ainsi, la règle

  ```
  a+ ->@ b
  ```

  appliquée à la chaîne `aaac` produira uniquement `bc`. Ces règles décrivent toujours des transducteurs d'états finis, mais la taille de ces transducteurs peut très vite exploser, aussi ne sont-elles pas toujours utilisables en pratique ;
- il est également possible de « recopier » dans le membre droit la partie reconnue par le membre gauche lorsque celle-ci peut varier à l'aide du symbole `"..."`. Par exemple,

  ```
  a | b | c -> ... d || _ f
  ```

  ajoute un `d` après un `a`, un `b` ou un `c` suivi d'un `f` (par exemple, `afbf` donnera `adfbdf`) ;
- enfin, des raccourcis syntaxiques permettent de spécifier plusieurs contextes possibles pour une même règle, par exemple

  ```
  a b -> c d || f _ g, h _ i
  ```

  ou encore de spécifier plusieurs règles partageant un même contexte, comme

  ```
  a b -> c d, f -> g || h _ i
  ```

5.2 L'algèbre des structures Sumo

Les structures Sumo forment une algèbre. On présente ici la syntaxe des expressions permettant de décrire des structures Sumo et des chemins dans ces structures ; on voit ensuite un mode de manipulation fondamental que sont les règles d'identification et de liaison. Enfin, on revient sur les attributs, qui sont une partie importante des items.

5.2.1 Description des items

La syntaxe d'un item est donnée par l'expression :

intitulé (`"@{"` (*attribut* (`","` *attribut*)*)? `"}"`)? (`"/"` *poids*)?

On distingue donc trois parties : l'intitulé, la liste d'attributs et le poids de l'item. L'intitulé est un symbole qui suit la même syntaxe que les symboles des transducteurs pondérés, et ne peut être qu'un symbole simple (donc pas de symbole de la forme `a:b`).

Le poids est comme pour les symboles une expression de poids dont la syntaxe varie selon le type de poids employé ; dans la plupart des cas il s'agit de réels, mais il peut aussi s'agir de booléens ou de chaînes quelconques.

Si l'ensemble d'attributs est omis, alors l'item n'a aucun attribut. Au contraire, le symbole spécial `"..."` dénote n'importe quel ensemble de valeurs. Enfin, un item n'ayant qu'un seul attribut peut s'exprimer plus simplement par :

intitulé `"@"` *attribut* (`"/"` *poids*)?

Une expression *attribut* est de la forme

nom `"="` *valeur*

Le *nom* d'un item est un symbole de la même forme qu'un intitulé. La *valeur* peut être positive, négative ou potentielle. Une valeur positive est un symbole de la même forme qu'un intitulé ; on peut utiliser "`0`" pour une valeur vide et `"."` pour une valeur quelconque.

Exemples :

`chat@{cat=N,gnr=Masc}/2`

L'item intitulé `chat` a un poids de 2 et deux attributs, `cat` et `gnr`, qui valent respectivement `N` et `Masc`.

`chat`

L'item intitulé `chat` n'a aucun attribut.

`chat@{cat=N,gnr=.,nbr=.,...}`

L'item intitulé `chat` a pour catégorie `N` et peut avoir n'importe quelle valeur pour genre et nombre. Il peut s'agir ici d'une forme lexicale représentant des formes comme *chat*, *chats*, *chatte* ou *chattes*. Il peut également avoir n'importe quel autre attribut.

`chat@{...}`

L'item intitulé `chat` peut avoir n'importe quels attributs.

Une valeur négative est définie de la même façon, mais est précédée d'un `!`.

Exemple :

`chat@{mark=!.,time!=.,...}`

L'item intitulé `chat` peut avoir n'importe quel attribut, mais ne peut pas avoir de valeur pour l'attribut `mark`, et ne peut pas avoir l'attribut `time`.

Une valeur potentielle est une liste de valeurs séparées par `|` (signifiant une alternance), ou, pour les nombres (réels ou entiers), un intervalle délimité par des crochets ouverts ou fermés. Une valeur potentielle peut également être négative si elle est précédée d'un "`!`".

Exemples

`.@{gnr=Masc|Fem}`

Un item quelconque qui peut être masculin ou féminin.

`a@x=]0,1]`

Un item intitulé `a` ayant un attribut `x` dont la valeur est comprise entre 0 (non-inclus) et 1.

5.2.2 Opérations concernant les structures

5.2.2.1 Expressions de chemins

Une expression de chemins décrit un ou plusieurs chemins dans une structure Sumo. Il s'agit d'une extension de la notation des expressions régulières sur les chaînes :

- les éléments terminaux sont des items ;
- on utilise les deux opérateurs `":"` et `"^:"` pour relier deux niveaux.

Syntaxe. *expression-de-chemin* `::=` *expression-régulière-d'items-de-chemins*

Les opérateurs réguliers sont ici limités à : la concaténation (ϵ), l'union (`"|"`), l'étoile (`"*"`), le plus (`"+"`) et l'optionnalité (`"?"`).

item-de-chemin `::=` *item-usuel* `|` `"["` *chemin-plat* `"]"` `|`
item-de-chemin opref item-de-chemin
item-usuel `::=` *item*
chemin-plat `::=` *expression-régulière-d'items-usuels* `|` `"<"` *item-usuel* `">"`
opref `::=` `"^:"` `|` `":"`

Sémantique. `"<"` *item* `">"` éclate l'*item* en une liste d'items dont les intitulés sont les caractères de l'intitulé de *item* et les attributs sont ceux de *item*.

`"["` *chemin-plat* `"]"` comprime le *chemin-plat* en un seul item dont l'intitulé est la concaténation des intitulés et les attributs l'union des attributs.

α `":"` β décrit une relation entre les deux chemins plats α (situé au niveau courant n) et β (situé au niveau $n-1$).

α `"^:"` β décrit une relation entre les deux chemins plats α (situé au niveau courant n) et β (situé au niveau $n+1$).

Exemples :

`chat : <chat>`

décrit la correspondance entre un item (intitulé `chat`) au niveau n et sa décomposition en caractères (`c`, `h`, `a`, `t`) au niveau $n-1$.

`(a : b c) (d e ^: f)`

décrit un chemin `a d e` au niveau n tel que `a` soit en relation avec `b c` au niveau $n-1$ et `d e` soit en relation avec `f` au niveau $n+1$.

5.2.2.2 Opérations sur les structures Sumo

La table 5.3 présente ces opérations. La colonne « Struct » précise si une opération est possible sur les structures Sumo ; la mention « MN » est mise quand une opération binaire n'est définie que pour deux structures de même niveau. `A` et `B` désignent deux structures Sumo quelconques.

Opérateur	Nom	Struct	Interprétation spécifique
`A B`	Concaténation	oui (MN)	concaténation de chaque niveau, liaisons gardée telles quelles
`!A`	Complément	oui	complément à chaque niveau, aucune liaison sauf entre les `"."` rajoutés
`A % B`	Composition	non	
`A %+ B`	Composition itérative	non	
`A & B`	Intersection	oui (MN)	intersection à chaque niveau, de haut en bas, liaisons restantes gardées telles quelles
`A*`	Fermeture transitive	oui	fermeture de chaque niveau
`A+`	Fermeture positive	oui	idem
`A,l`	Projection inférieure	oui	étage inférieur (raccourci)
`A,u`	Projection supérieure	oui	étage supérieur (raccourci)
`A,`*n*	*n*ème projection		extraction du *n*ème niveau
`A,-`*n*	*n*ème suppression		suppression du *n*ème niveau
`A,"`*nom*`"`	projection nommée		extraction du niveau nommé *nom*
`A,-"`*nom*`"`	suppression nommée		suppression du niveau nommé *nom*
`A - B`	Différence	oui	il reste à chaque niveau les chemins de A non chemins de B et liaisons restées intactes
`A : B`	Produit cartésien	oui	A se met au-dessus de B, aucune liaison nouvelle
`A?`	Optionnalité	oui	Optionnalité de chaque niveau (ajout d'un arc epsilon à chaque niveau)
`^A`	Inversion	oui	l'ordre des étages et l'orientation des liaisons sont inversés
`A | B`	Union	oui (MN)	union à chaque niveau, de haut en bas, liaisons restantes gardées telles quelles
`~A`	Renversement	oui	renversement (horizontal) à chaque niveau et modification correspondante des liaisons

Tab. 5.3 – Opérateurs sur les structures

5.2.2.3 Fonctions reliant transducteurs et structures Sumo

Les deux fonctions de la table 5.4 sont utiles. L'explosion crée de nouvelles relations entre les deux

Opérateur	Nom	Interprétation
`<T>`	Explosion	Explosion d'un transducteur vers une structure Sumo à deux niveaux
`[A]`	Compression	Compression d'une structure Sumo de deux niveaux vers un transducteur

TAB. 5.4 – Passage de transducteurs à structures Sumo et réciproquement

niveaux ; la compression perd les relations entre les deux niveaux ainsi que les attributs des items.

5.2.3 Règles d'identification et de liaison

Les règles d'identification créent un nouveau niveau de segmentation à partir d'un niveau existant, et les règles de liaison créent des relations entre deux niveaux adjacents. Ces deux types de règles ont une syntaxe similaire et existent en deux versions : une version simple et une version paramétrée.

5.2.3.1 Syntaxe

Une règle simple est de la forme suivante :

ϕ *opérateur* ψ (`"||"` λ `"_"` ρ)?

où ϕ , ψ , λ et ρ sont des expressions de chemin et *opérateur* est l'opérateur d'identification (`->`) ou de liaison (`<->`). Le contexte peut être omis.

Une règle avec variables est de la forme

ϕ *opérateur* ψ (`"||"` λ `"_"` ρ)? (`"?{"` *conditions* `"}"`)? (`"{"` *actions* `"}"`)?

où ϕ , ψ , λ et ρ sont des expressions de chemin pouvant contenir des variables, et *opérateur* est l'opérateur d'identification (`=>`) ou de liaison (`<=>`). Le contexte ainsi que le parties *conditions* et *actions* peuvent être omis. Les conditions portent sur les variables de ϕ et peuvent appeler des fonctions provoquant des effets de bord, comme des messages envoyés à l'utilisateur ou conservés dans un fichier. Les actions consistent en des affectations aux variables de la partie droite et en appels à des fonctions arbitraires.

Exemple :

```
0 => 0/$p || $x _ $y { $x = . ; $y = . ;
                       $p = f($x@CAT, $y@CAT) }
```

Cette règle insère un symbole epsilon entre deux items ; son poids `$p` est calculé à partir d'une matrice de poids par la fonction `f` qui prend la catégorie grammaticale de deux items comme arguments.

Exemple :

```
$a $b => [$a $b] || ?{$a eq $b} {$a = (a|b)* ; $b = (a|b)* }
```

Cette règle reconnaît des mots définis sur l'alphabet {a, b} de la forme *ww*.

Les expressions de contexte λ et ρ peuvent contenir le symbole spécial `#`, qui indique une extrémité (le début d'un chemin en contexte gauche, et la fin en contexte droit).

Grammaires. Une grammaire élémentaire consiste en une liste de règles combinées par le même opérateur :

- *règle* (`","` *règle)** pour une liste de règles avec priorité
- *règle* (`",,"` *règle)** pour la mise en parallèle
- *règle* (`"%"` *règle)** pour la mise en séquence

Une grammaire générale est une grammaire élémentaire ou une expression formée de grammaires combinées avec ces mêmes opérateurs.

L'opérateur `"%"` est prioritaire par rapport à l'opérateur `",,"` qui est lui-même prioritaire par rapport à `","` : `a%b,,c,d` équivaut à `(((a%b),,c),d)`.

5.2.3.2 Sémantique

Règles d'identification. Une règle d'identification s'applique à un niveau existant pour créer des chemins dans un nouveau niveau, ainsi que des relations entre ces deux niveaux. L'application d'une règle simple crée, pour chaque chemin correspondant à l'expression ϕ , tous les chemins décrits par ρ et une relation de chemins entre chaque couple de chemins.

Une règle avec variables a la même sémantique, mais trois aspects altère ce comportement :

1. chaque expression de chemin peut contenir des variables. ϕ , λ et ρ permettent de capturer certaines parties du chemin reconnu pour le stocker dans une variable, tandis que ψ est lui-même décrit à l'aide de variables ;
2. le bloc *conditions* spécifie des conditions que le chemin reconnu doit remplir, et qui ne sont pas exprimables par l'expression de chemin. Ces conditions portent sur les variables introduites par les expressions ϕ , λ et ρ ;
3. le bloc *actions* affecte une valeur aux variables de l'expression ψ à partir des variables capturées dans le chemin reconnu et son contexte (ϕ , λ et ρ).

Règles de liaison. Les règles de liaison sont similaires aux règles d'interprétation mais s'appliquent entre deux niveaux existants afin d'ajouter de nouvelles relations. Ainsi, la partie droite de la règle (notée ψ) doit représenter un chemin existant dans la structure considérée.

Restrictions sur les chemins. On pose une restriction sur les expressions de chemins dans ces deux types de règles.

Dans les règles d'identification, les expressions de chemins ϕ , λ et ρ ne peuvent faire référence qu'à des chemins dans des niveaux inférieurs à celui considéré ; alors que ψ ne peut décrire qu'un chemin plat.

Dans les règles de liaison, les expressions de chemins ϕ , λ et ρ ne peuvent faire référence qu'à des chemins dans des niveaux inférieurs à celui considéré ; et ψ ne peut faire référence qu'à des niveaux supérieurs à celui considéré.

5.2.3.3 Extraction de règles à partir d'une structure Sumo

L'extraction de règles à partir d'une structure Sumo est un moyen de produire des dictionnaires de segmentation à partir de documents Sumo. C'est particulièrement intéressant dans le cas où un ensemble de structures Sumo a été créé ou amélioré par édition manuelle (comme on l'a vu dans la section 4.3.3).

5.3 Les structures de contrôle

5.3.1 Types, constructeurs de types et fonctions

5.3.1.1 Types de données

La table 5.5 décrit les types de données fondamentaux de Sumo. On distingue les types du langage du contrôle des types des structures Sumo (structure à étages et transducteurs pondérés).

`number`	nombre réel
`boolean`	booléen
`string`	chaîne de caractères
`sumo`	une structure Sumo
`weight`	le poids d'un item ou d'un symbole
`graph`	un graphe d'items
`item`	un item Sumo
`attribute`	un attribut d'un item
`handle`	un intitulé Sumo
`relation`	une relation
`trans`	une transition dans un graphe d'items
`node`	un nœud dans un graphe d'items
`wfst`	un transducteur d'états finis pondérés
`symbol`	un symbole d'un transducteur
`arc`	un arc d'un transducteur
`state`	un état d'un transducteur
`grammar`	une grammaire de règles d'identification ou de liaison
`rule`	une règle d'identification ou de liaison

Tab. 5.5 – Les types de donnée

Chacun de ces types a des attributs qui le définissent. Par exemple, un item `I` a les attributs suivants :

1. `"I.str"` est l'intitulé de l'item (`string`) ;
2. `"I.w"` est le poids de l'item (`weight`) ;
3. `"I.attrs"` est la liste des attributs de l'item (`set of (attr : attribute)`) ;
4. `"I@`*attr*`"` désigne l'attribut *attr* de l'item `I` ; c'est un raccourci pour `I.attrs{attr=`*attr*`}`.

5.3.1.2 Constructeurs de types

Les constructeurs de la table 5.6 permettent d'introduire de nouveaux types de données.

Tableaux. Les tableaux sont indexés par des entiers, à partir de 1. `T[n]` désigne le $n^{ème}$ élément du tableau `T`. Un tableau possède trois attributs :

1. `T.size` est le nombre d'éléments du tableau. Un tableau vide a pour taille 0 ;
2. `T.last` est l'indice du dernier élément du tableau.

`array` *type*	tableau d'éléments de type *type*
`struct (`*champs*`)`	enregistrement
`set (`*champs*`)`	ensemble d'éléments

TAB. 5.6 – Les constructeurs

Structures. La liste des champs pour un enregistrement est une expression de la forme

champ (`":"` *type*)? (`","` *champ* (`":"` *type*)?)*

qui définit pour chaque champ son type (le type peut être omis, auquel cas le type choisi est `string`). `S.field` désigne le champ `field` de la structure `S`.

Ensembles. La liste des champs pour un ensemble est de la même forme que pour les enregistrements.

`S{`*condition sur les champs*`}` désigne les éléments de l'ensemble `S` dont les champs vérifient les condition.

`S.size` désigne le nombre d'éléments de l'ensemble `S` et vaut 0 pour un ensemble vide.

5.3.1.3 Déclarations de variables, de types et de fonctions

Une variable est déclarée par :

variable `":"` *type*

Un nouveau type peut être créé par l'un des trois constructeurs `array`, `struct` et `set` et est nommé à l'aide du mot-clé `deftype`.

`"deftype"` *type constructeur*

Exemple :

```
deftype entree struct (morphe, ftm, fts, ul) ;
deftype dict set (entree) ;
```

Les deux lignes précédentes décrivent un dictionnaire ATEF comme un ensemble d'entrées, qui sont des enregistrements comprenant les quatre champs *morphe*, *ftm*, *fts* et *ul*, qui sont chacun des chaînes.

```
entrees : array entree ;
entrees = dict{morphe=="chat"} ;
```

La variable `entrees` est un tableau d'entrées, à laquelle on affecte la liste des entrées du dictionnaire `dict` ayant pour morphe `chat`.

Une fonction est définie par le mot-clé `defun` :

`"defun"` *f* `"("`*arguments*`")"` (`"returns"` *type*)? `"{"`*actions*`"}"`

où la liste d'arguments est définie par :

(*var* (`":"` *type*)?)? (`","` *var* (`":"` *type*)?)*

On définit ainsi la fonction *f* de zéro, un ou plusieurs arguments (dont le type est soit `string`, soit le type déclaré), qui retourne une valeur ou non (si la partie `returns` est omise, on a une procédure) et dont le corps est constitué d'un bloc d'*actions* (voir plus loin).

L'instruction `return` renvoie une valeur et arrête l'exécution d'une fonction. Si une fonction arrive à terme sans se terminer par un `return`, alors elle renvoie la valeur spéciale `nil` pour indéfini.

Exemple :

```
defun max (l : array number) returns number
{
  max : number ;
  i   : number ;

  if (l.size > 0) do {
    max = l[1] ;
    for (i = 2; i <= l.last ; i = i + 1) do {
      if (l[i] > max) then {
        max = l[i] ;
      }
    }
    return max ;
  } else {
    return nil ;
  }
}
```

définit la fonction `max` retournant pour une liste de nombres la plus grande valeur ; et `nil` si la liste de nombres est vide.

5.3.2 Structures de contrôle

5.3.2.1 Instructions et blocs

Une instruction est constituée d'une seule action. Un bloc est une suite de 0 ou plusieurs instructions séparées par un point-virgule `";"` et est compris entre deux accolades `"{}"`. C'est le type de construction que l'on note `"{"`*actions*`"}"`. Le point-virgule suivant la dernière action est facultatif.

Une action ou un bloc vide n'a aucun effet.

5.3.2.2 Traitement conditionnel

Une construction de type *si-alors-sinon* a pour syntaxe :

`"if" "("` *condition* `")" "then" "{"` *actions* `"}"` (`"else" "{"` *actions* `"}"`)?

La *condition* est une expression booléenne quelconque. Si cette condition est vraie, alors les *actions* correspondant à la partie `then` sont évaluées ; sinon, c'est la partie `else` qui évaluée. Si celle-ci est omise, alors il ne se passe rien.

5.3.2.3 Itérations

Il existe deux types d'itérations : la boucle `for` et la boucle `while`. Un `for` est de la forme :

`"for" "("` *initialisations* `";"` *condition* `";"` *post-actions* `")" "do" "{"` *actions* `"}"`

La partie *initialisations* est évaluée, puis, tant que la *condition* (une expression booléenne) est vraie, la partie *actions* puis la partie *post-actions* sont évaluées.

Une variante de `for` est `foreach` :

`"foreach"` *variable* `"in"` *liste* `"do" "{"` *actions* `"}"`

variable est instanciée successivement par chacun des éléments de *liste* qui peut désigner un tableau ou un ensemble.

Un `while` peut prendre deux formes. La première forme est la suivante :

`"while" "(" condition ")" "do" "{" actions "}"`

La partie *actions* est évaluée tant que la *condition* (une expression booléenne) est vraie. Cette condition est évaluée avant d'évaluer la partie *actions*. Inversement, la forme

`"do" "{" actions "}" "while" "(" condition ")"`

inverse l'ordre d'évaluation des actions et de la condition.

5.3.3 Fonctions prédéfinies

On prévoit pour Sumo complet une large bibliothèque de fonctions prédéfinies pour la manipulation des données primitives et des données du formalisme, ainsi que pour d'autres tâches comme les entrées/sorties (lecture et écriture de fichiers, interactions avec l'utilisateur...).

5.3.3.1 Opérations sur les types

La table 5.7 présente les opérations sur les nombres.

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
x `%` y	modulo (reste de la division entière de x par y)
x `**` y	exponentiation (x^y)

TAB. 5.7 – Opérations sur les nombres

La table 5.8 présente les opérations sur les chaînes.

`#`	concaténation	`"abc" # "def" == "abcdef"`
\	division à gauche*	`"abcdef" \ "abc" == "def"`
`/`	division à droite*	`"abcdef" / "def" == "abc"`
`~`	renversement (mirroir)	`~"abc"` == ”`cba"`

TAB. 5.8 – Opérations sur les chaînes

Note (*) : la division n'est pas totale et peut produire `nil` si le résultat n'est pas défini (par exemple `"abcdef" \ "def"`).

La table 5.9 présente les opérations sur les booléens.

Enfin, on définit des opérations de comparaison entre nombres (`"=="`, `">"`, etc.) `"=="` est également défini pour tous les autres types, mais pour les types complexes, il ne teste pas le type en profondeur (*eg.* si `A` et `B` sont deux `sumo`, `A == B` est vrai si et seulement si `A` et `B` font référence à la même structure).

5.3.3.2 La bibliothèque de fonctions Sumo et autres

On retient un certain nombre de fonctions héritées de Perl et d'autres langages. La table 5.10 les présente. La table 5.11 présente les fonctions prédéfinies pour les structures Sumo.

`!`	négation
`&`	conjonction
`|`	disjonction non-exclusive
`^`	disjonction exclusive
`!=`	différence (logiquement équivalent à `^`)
`=>`	implication
`<=>`	équivalence
`==`	égalité (logiquement équivalent à `<=>`)

TAB. 5.9 – Opérations sur les booléens

Mathématiques	`abs`, `cos`, `int`, `log`, `exp`, `sqrt`, `rand`, `sin`, `min`, `max`
Tableaux	`pop`, `push`, `shift`, `unshift`, `splice`, `reverse`, `sort`, `join`, `map`, `filter`, `reduce` (vers un accumulateur)
Ensembles	`add`, `delete`, `each` (donne un tableau)
Entrées/Sorties	`print`, `prompt`, `read`, `write`, `open`, `close`
Contrôle	`return`, `abort`, `exit`

TAB. 5.10 – Principales fonctions prédéfinies

`apply`	appliquer une grammaire de règles d'identification
`read_text`	lire un fichier texte vers un niveau Sumo
`read_sumo`, `write_sumo`	lire/écrire une structure Sumo depuis/vers un fichier
`read_wfst`, `write_wfst`	lire/écrire un transducteur depuis/vers un fichier

TAB. 5.11 – Fonctions Sumo

Chapitre 6

Choix pour l'implémentation d'un prototype

Introduction

L'implémentation de Sumo passe tout d'abord par la réalisation d'un prototype, qui permet de tester les principes du formalisme ainsi que de définir un cahier des charges précis pour une réalisation plus robuste du système complet. Ici, c'est la partie sur les états finis pondérés qui a fait l'objet d'un prototypage approfondi.

Le choix du cadre d'implémentation n'est pas une question triviale car le prototype doit pouvoir être réalisé rapidement tout en étant assez complet et efficace pour permettre des tests pertinents.

Tout d'abord, on passe en revue certaines implémentations déjà existantes (XFST, FSM et ASTL) pour étudier les techniques d'implémentation communes qui pourront servir de base pour le prototype de Sumo.

Dans un deuxième temps, on explicite les critères choisis pour déterminer le langage d'implémentation, car aucune des approches citées ne peut convenir telle quelle pour le prototype de Sumo. Dans la troisième, on verra pourquoi le prototype est finalement réalisé en Perl et quels sont les avantages supplémentaires qu'apporte ce langage dans cette situation particulière, tout en gardant à l'esprit que les critères définis ici sont valables pour un prototype, et pas forcément pour la réalisation du système complet.

6.1 Techniques d'implémentation classiques

La plupart des implémentations de bibliothèques d'états finis (avec ou sans pondération) disponibles actuellement sont écrites en C ou en C++. Certaines applications de moindre importance sont écrites en Java, par exemple des *applets* pour manipuler graphiquement de petits automates et transducteurs sur le Web.

6.1.1 Une revue de trois implémentations existantes

6.1.1.1 XFST

La bibliothèque de calcul d'états finis et les outils de compilation de transducteurs de Xerox sont écrits entièrement en C, après une première implémentation écrite à l'origine en Interlisp. Une des raisons de cette migration était sans doute la portabilité, et peut-être dans une moindre mesure les performances du système. Cette bibliothèque est en développement depuis longtemps et est particulièrement optimisée, pour offrir de bonnes performances en temps de calcul et en taille en mémoire

des transducteurs. Elle comporte également beaucoup d'aspects originaux et novateurs, comme par exemple la compression des transducteurs, les bi-machines ou encore les *flag diacritics*[Beesley et Karttunen, 2002].

La bibliothèque de calcul d'états finis peut être exploitée directement par une application écrite en C (ou qui peut être liée à une bibliothèque de fonctions en C), mais on l'utilise plutôt par le biais d'interfaces telles que `xsft`.

6.1.1.2 FSM

Les grandes lignes de l'implémentation des outils FSM de AT&T sont donnés par [Mohri *et al.*, 1997]. À l'époque, ces outils étaient réalisés en C mais leurs auteurs comptaient passer à C++ pour utiliser les *templates* et généraliser ainsi les étiquettes de transition et les structures de poids utilisées (jusqu'à présent, les poids étaient gérés par des macros en C). Outre la pondération, qui est un point essentiel, un aspect original des outils AT&T est l'utilisation d'algorithmes « à la demande ».

Un exemple d'un tel algorithme est la composition paresseuse. La composition de plusieurs transducteurs volumineux est toujours possible mais peut avoir pour résultat un transducteur gigantesque, bien plus grand que la somme de la taille des transducteurs qui ont été composés pour le produire. Dans le même ordre d'idée que l'application de cascades de transducteurs des outils de Xerox, il est ici possible de calculer la composition de manière locale, car d'état en état, seule l'information sur les transitions quittant cet état est nécessaire. Ainsi, on peut manipuler un transducteur virtuel dont on découvre les chemins au fur et à mesure de son application.

Cette technique n'est pas généralisable à toutes les opérations. Si les opérations d'union, de concaténation ou de fermeture transitive ou même la déterminisation sont toutes calculables localement, d'autres qui nécessitent de parcourir le transducteur dans les deux directions, comme la minimisation, la connexion ou le calcul du meilleur chemin, et ne peuvent donc pas être implémentées de la sorte.

FSM n'a pas d'interface comparable à `xfst`, du moins disponible publiquement, mais une collection de filtres. Le compilateur `fsmcompile` ne compile pas un transducteur à partir d'une expression régulière, mais à partir d'une description des symboles, états et transitions du transducteur. La compilation d'un transducteur est donc soit manuelle, soit le résultat de l'application successives de différents filtres (opérations rationnelles, suppression des epsilon-transitions, déterminisation, minimisation, connexion, etc.) ; mais on ne dispose pas d'un formalisme d'expressions régulières semblable à celui de la section 5.1.

6.1.1.3 ASTL

La bibliothèque ASTL de l'Université de Marne-la-Vallée [Maout, 1997] est une bibliothèque écrite en C++. Il s'agit en réalité d'une extension de la Standard Template Library pour les automates et transducteurs d'états finis. [Adant, 2000] présente une extension de cette bibliothèque aux automates et transducteurs d'états finis pondérés.

Cette bibliothèque est tout à fait comparable à FSM et propose le même genre de fonctionnalités, hormis les méthodes de compilation à la demande. Par contre, on retrouve un même type d'interface fondée sur une série de filtres, ainsi que le même format textuel de description de transducteurs. ASTL emprunte également aux outils de Xerox les règles morphologiques à deux niveaux, mais les étend au modèle pondéré.

6.1.2 Points communs des implémentations existantes

Dès la première partie, on a donné une définition formelle des transducteurs d'états finis ; dans le chapitre précédent et dans la littérature on trouve de nombreux algorithmes pour la compilation de transducteurs et les opérations rationnelles sur les transducteurs d'états finis, pondérés ou non.

Dans [Aho *et al.*, 1986], diverses représentations possibles d'automates d'états finis sont évoquées ; il semblerait qu'une représentation particulière soit la plus répandue et partagée par les implémentations vues ici. Le format de description des transducteurs d'AT&T permet de se faire une idée précise de cette représentation ; elle est partagée par celle d'ASTL et similaire à celle de XFST (et sans doute d'autres réalisations existantes).

Un transducteur est essentiellement constitué de deux alphabets (chaque alphabet étant un ensemble de symboles), d'un ensemble d'états (parmi lesquels certains sont initiaux et d'autres sont finals) et d'un ensemble de transitions. Dans XFST, il n'y a en réalité qu'un seul alphabet contenant les symboles de chacun des deux langages de la relation.

Les éléments de l'alphabet sont des chaînes de caractères arbitrairement longues. Un symbole est donc encodé par un entier (par exemple, `x` est encodé par 1, `y` par 2, `z` par 3 et `w` par 4) qui servira d'étiquette à une transition. Les états sont eux aussi numérotés ; une transition n'est plus définie que par quatre entiers (le numéro de l'état d'origine, le numéro de l'état destination, le numéro du symbole supérieur, le numéro du symbole inférieur) et un poids.

Ainsi, le format de description de FSM consiste en trois parties :

1. la définition des symboles
2. une liste de transitions
3. la liste des états finals

L'état initial est toujours l'état 0 ; il peut y avoir un nombre arbitraire d'états finals. La description est séparée en plusieurs fichiers : un fichier par alphabet, et un fichier par automate ou transducteur. Par exemple, on voit ci-dessous un fichier d'automate AT&T, où les transitions ne portent qu'un seul symbole :

```
0   0   1   .5
0   1   2   .3
1   2   3   .6
1   2   4   .6
2
```

Cet automate (que l'on voit également figure 6.1) peut également s'exprimer par l'expression régulière pondérée

```
x/0.5* y/0.3 z/0.6 | w/0.6
```

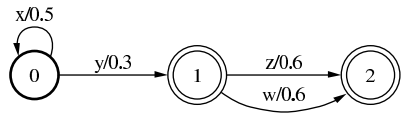

FIG. 6.1 – Automate d'états finis pondéré `x/0.5* y/0.3 z/0.6 | w/0.6`

Dans la représentation interne, il est pertinent d'associer à chaque état l'ensemble des transitions qui ont cet état pour origine.

6.2 Caractéristiques souhaitées des outils d'implémentation

Deux questions se posent désormais : peut-on (ou veut-on) bâtir le prototype de Sumo sur une bibliothèque d'états finis pondérés existante ? Dans le cas d'une réponse négative, il faut créer une nouvelle bibliothèque dont les fonctionnalités répondent mieux aux besoins de Sumo ; et pour cela il faut redéfinir complètement le cadre d'implémentation.

Si l'on commence par répondre à la deuxième question, on aura également une réponse pour la première. Le prototype doit comporter une bibliothèque de calcul à états finis pondérés, ainsi qu'une ou plusieurs interfaces pour utiliser celle-ci. Parmi les types d'interface existants, il y a les filtres (FSM et ASTL), la ligne de commande (XFST) ou une interface graphique pour « dessiner » les transducteurs.

D'un point de vue pratique, c'est le premier type qui est le plus simple à réaliser ; d'un point de vue linguistique, c'est sans doute le deuxième qui est le plus satisfaisant car un linguiste s'intéressera sans doute plus volontiers à la définition d'une relation qu'à sa représentation concrète par un transducteur. Mais l'interface graphique a tout de même des avantages, et l'on reviendra sur cette question d'interface dans le chapitre 9.

6.2.1 Unicité des outils d'implémentation

Le choix d'un outil unique pour l'implémentation, c'est-à-dire le même pour la bibliothèque est l'interface, permet de simplifier les choses au maximum. Utiliser plusieurs langages (par exemple, une bibliothèque en C avec une interface graphique en Tcl/TK) nécessite l'écriture de couches supplémentaires, avec tous les problèmes que cela peut comporter. Notre premier critère est donc de choisir un langage qui dispose de fonctionnalités permettant d'écrire une interface de quelque nature qu'elle soit tirant le meilleur partie de la bibliothèque de calcul d'états finis pondérés écrite dans ce même langage.

Entre autres, il faut aussi pouvoir disposer dans un même environnement de librairies pour la construction d'interfaces graphiques (comme par exemple TK, le *toolkit* de Tcl), ainsi que d'analyseurs syntaxiques pour l'interface en ligne de commande (qui doit analyser aussi bien les expressions régulières pondérées que les commandes entrées par l'utilisateur).

6.2.2 Rapidité du prototypage

C'est un point important car il faut pouvoir produire rapidement un prototype utilisable. Les langages interprétés, comme Perl, Ruby, Python, Rexx ou Tcl, Common Lisp Object System (CLOS), Scheme encore Caml sont de bons candidats car ils permettent un cycle de développement rapide (pas de compilation) et déchargent le programmeur de nombreuses tâches de bas niveau comme la gestion de la mémoire.

On préférera tout de même un langage qui inclut une phase de compilation (*e.g.* Perl ou Python) à un langage purement interprété (*e.g.* Tcl) pour une plus grande sécurité. Par exemple Perl, dans son mode de compilation et d'exécution le plus strict permet de repérer beaucoup d'erreur dès la compilation, malgré le typage très faible du langage. Au contraire, Tcl, qui est seulement interprété, ne détecte les erreurs qu'à l'exécution. Un autre avantage est en général un gain de performance grâce à une étape intermédiaire de représentation du programme.

On élimine donc ici les langages tels que C/C++ ou Java car ils ne satisfont pas ce critère. Par conséquent, cela élimine les bibliothèques existantes comme ASTL, programmée C++. L'étude du code d'ASTL ou de XFST (en C) est cependant riche d'enseignement pour la réalisation du prototype, quel que soit le langage finalement choisi.

6.2.3 Puissance du langage d'implémentation

Un transducteur d'états finis pondérés est une structure complexe, et la bibliothèques de calcul d'états finis doit manipuler des listes de symboles, d'états ou de transitions ; les algorithmes tels que la déterminisation d'un transducteur pondérés sont relativement complexes. Il faut donc que les structures de données proposées par le langage soient assez puissantes pour cela. On posera comme condition minimum la présence de listes ou de tableaux non bornés, ainsi que des tables de hachage qui permettront entre autre une gestion plus simple des symboles.

Un transducteur d'états finis pondérés est défini sur une structure de poids et des alphabets arbitrairement complexes. Les poids ne sont pas forcément des entiers ou des réels, on peut utiliser des expressions régulières par exemple. De même pour les symboles qui étiquettent les transitions : dans Sumo, on sait que les items sont des unités de segmentation qui contiennent plus d'information qu'une simple chaîne de caractères. Quant aux transitions elles-mêmes, elles doivent prendre en compte les relations qui existent entre les items de différents niveaux. Suivant l'exemple d'ASTL et de FSM, il semble qu'une approche fondée sur les objets soit une solution satisfaisante : ainsi, on définit une classe « poids » présentant une interface commune à toutes les structures de poids possibles (comme les opérations additives et multiplicatives) ; cette classe peut être raffinée en poids tropicaux, réels, booléens, etc. Et il en est de même pour les items ou les arcs.

Enfin, Sumo est un formalisme multilingue. Le langage d'implémentation choisi doit pouvoir traiter au moins du texte en Unicode, et si possible d'autres encodages. Encore une fois, il vaut mieux que cet aspect soit intégré au langage et ne nécessite pas de modules supplémentaires.

6.2.4 Portabilité

La portabilité est un dernier critère important : le même programme doit pouvoir fonctionner avec un minimum (si possible, aucune) de modifications sur des plate-formes diverses ; aussi le langage choisi doit être porté sur les systèmes visés. Les bibliothèques et boîtes à outil graphiques sont parfois un frein à la portabilité, mais TK par exemple existe sur de nombreuses plates-formes différentes sur lesquelles il se comporte de façon cohérente, et est disponible pour une grande variété de langages.

De ces quatre critères, le choix de langages se réduit à quelques candidats comme Perl, Ruby, Python ou Common Lisp. On choisira finalement Perl qui possède des avantages supplémentaires décrits ci-dessous.

6.3 Avantages du prototypage en Perl

Perl est un langage fort répandu (disponible en standard dans la plupart des systèmes Unix, et librement sur la quasi-totalité des autres systèmes d'exploitation actuels), qui dispose d'un modèle d'objets simple et original, ainsi que des structures de données nécessaires à l'implémentation (les tables de hachages sont au cœur du langage). Les versions récentes, notamment la nouvelle version 5.8, intègrent Unicode de manière quasiment transparente. Et la version 6 du langage, prévue pour les années à venir, devrait garantir la pérennité du langage et apporter des améliorations appréciables.

La communauté Perl est particulièrement active depuis des années, et est peut-être la plus importante parmi celle des quatre langages cités ci-dessus. Sumo pourra mettre à profit certains modules développés par cette communauté, notamment les différentes bibliothèques graphiques (dont TK) ainsi que le module `Parse::RecDescent` qui est un générateur d'analyseurs syntaxiques, au même titre que `yacc` mais nettement mieux intégré au langage (voir section 7.3.2). Celui-ci sera utile pour l'analyse des expressions régulières pondérées et la création d'une interface similaire à `xfst`.

Un autre avantage non négligeable est la compacité du code ; on est loin de la verbosité de Java ou Python.

On a aussi vu dans la première partie que Perl était déjà un langage intéressant pour l'analyse présyntaxique ; cependant, pour la tâche à accomplir ici, ce n'est pas forcément un avantage. En effet, les expressions régulières de Perl sont un atout majeur du langage (et facilitent énormément les opérations sur les chaînes de caractères), mais sont trop différentes des langages rationnels et des séries formelles rationnelles qui nous intéressent ici pour être directement utilisables. En effet, malgré la profusion de modules existant pour Perl et disponibles gratuitement sur le Web, dont un nombre de ressources linguistiques, il n'y a rien sur le calcul à états finis.

Conclusion

Le prototype actuel de Sumo sera présenté en détails dans la troisième partie. On y présentera la réalisation en Perl de la bibliothèque de calcul d'états finis pondérés et d'une interface en ligne de commande.

Le prototype présenté utilise une version récente de Perl ainsi qu'un unique module externe à la distribution classique, le module `Parse::RecDescent` mentionné plus haut. Celui-ci est disponible sur le site de CPAN[1] (*Comprehensive Perl Archive Network*). Pour plus d'informations sur le langage Perl lui-même, on peut se reporter à [Wall *et al.*, 2000] et au site Web `http://www.perl.com/`.

On notera pour finir que le choix de Perl convient pour la réalisation du prototype de Sumo et de la partie sur les états finis pondérés en particulier ; mais les critères de choix de langage auraient été différents si l'on avait voulu réaliser directement le système Sumo complet, de manière la plus efficace possible et en réutilisant des bibliothèques existantes. Cependant, nous n'avons trouvé aucune bibliothèque pour le traitement de structures à étages coordonnées.

[1] `http://cpan.perl.org/`

Troisième partie

Premières expérimentations

Introduction

Dans cette partie, on présente enfin la réalisation du prototype, les premières expérimentations et l'étude de la façon dont on pourra utiliser cette implémentation pour la réalisation du système complet.

Chapitre 7

Une bibliothèque expérimentale de calcul à états finis pondérés

Introduction

La première étape de la réalisation de Sumo est la mise en place d'une bibliothèque expérimentale de calcul à états finis pondérés. Bien qu'il existe des bibliothèques librement disponibles avec leurs avantages et leurs inconvénients, celles-ci sont plutôt destinées à être utilisées telles quelles et n'offrent pas l'ouverture nécessaire à l'expérimentation d'algorithmes et à l'inclusion dans une structure de données originale comme l'est celle de Sumo.

La plate-forme d'expérimentation choisie est prototypée en Perl. Les unités composant les transducteurs (poids, items, états, arcs) et les transducteurs eux-mêmes sont des objets. L'intérêt d'utiliser des objets est double : premièrement, grâce entre autres à la surcharge des opérateurs, on peut traiter les poids de manière assez naturelle en utilisant simplement `"+"` et `"*"` pour les opérations multiplicatives et additives. Deuxièmement, il est facile de remplacer une structure de poids ou de symboles en créant de nouvelles classes plus spécifiques héritant des classes qu'elles remplacent.

Notons avant de décrire plus avant cette bibliothèque que les exemples de programmes présentés sont simplifiés pour une meilleure lisibilité ; commentaires, messages d'informations optionnels et surtout traitement des erreurs ont généralement été omis. Enfin, les messages émis par le système (hormis les messages pour la mise au point) sont disponibles en plusieurs langues (actuellement, français et anglais) grâce à un dispositif transparent de localisation des messages développé pour l'occasion et permettant la localisation dans n'importe quelle langue.

7.1 Définition des structures de données

La bibliothèque d'états finis elle-même se présente sous la forme du module Perl `WFST.pm`, qui définit une classe `WFST` dont hériteront toutes les autres classes. Cette classe définit entre autres les méthodes d'accès aux différents attributs des objets, ainsi que des paramètres globaux que l'on verra plus en détails dans la section 7.3.

La hiérarchie des classes du module `WFST.pm` est représentée par la figure 7.1.

7.1.1 Structures de poids

On commence par décrire la structure de poids car c'est sans doute le composant le plus susceptible d'être modifié. Si le demi-anneau tropical est le plus courant en TALN, on a bien vu que tout demi-anneau pouvait servir de poids dans une série rationnelle. On définit donc une classe de poids générique

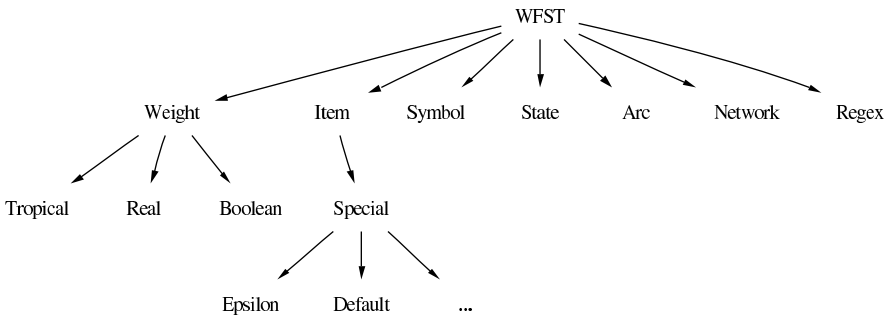

FIG. 7.1 – Hiérarchie des classes du module WFST

(qui serait plus ou moins l'équivalent d'une classe virtuelle en C++ ou d'une interface en Java) dont les classes de poids spécifiques (demi-anneau tropical, réels, booléens) héritent.

La devise de Perl étant « *there is more than one way to do it* » (« il y a plus d'une façon de le faire »), la façon exacte de définir une classe est très libre. Pour toutes les classes, on suit le même modèle que l'on illustre ci-dessous avec la classe de poids `WFST::Weight`.

```
package WFST::Weight;
use overload "+"   => "wadd",
             "*"   => "wmul",
             "/"   => "wdiv",
             "<=>" => "wcmp",
             "0+"  => "wval";
use vars qw(@ISA);
@ISA = qw(WFST);
```

La classe `WFST::Weight` est définie comme héritant de `WFST` grâce à la variable de classe `@ISA` (pour « *is a* », « est un(e) »). Différents opérateurs sont surchargés grâce au *pragma*`overload`, chacun correspondant à une fonction sur les poids définie plus loin. Les deux plus intéressants sont l'addition et la multiplication, qui correspondent respectivement aux méthodes `wadd` et `wmul` définies pour chaque sous-classe.

La définition d'une classe passe ensuite par la définition de sa représentation et des méthodes qui lui sont associées. Un poids est simplement un scalaire (qui peut aussi bien être un nombre entier ou réel qu'une chaîne de caractères). Dans Perl, un objet est toujours une référence (vers un scalaire ou vers un tableau) qui est ensuite bénie par la fonction `bless`. C'est la fonction `new` qui crée un nouvel objet poids, en prenant pour argument la classe ou une instance de cette classe (paramètre `$self`) et le poids (paramètre `$weight`, qui est soit un scalaire, soit un autre poids).

```
sub new {
  my ($self, $weight) = @_;
  if (ref $self) {
    $weight = $$self if !defined $weight;
    bless \$weight, ref $self;
  } else {
    $weight = $self->one if !defined $weight;
    bless \$weight, $self;
  }
}
```

Comme pour toutes les autres classes de `WFST.pm`, la méthode `new` est une méthode d'instance *et* de classe, c'est-à-dire qu'un poids peut être créé de deux façons possibles :

```
$w = new WFST::Weight(1);
$w2 = $w->new(2);
```

Trois variables de classe sont également définies. Il faut tout d'abord définir deux poids spéciaux, qui sont le poids nul et le poids unitaire (les éléments $\overline{0}$ et $\overline{1}$), représentés respectivement par les variables `$zero` et `$one`. Enfin, une variable supplémentaire contient la définition de la syntaxe d'un poids qui se greffe sur la grammaire décrivant la syntaxe d'une expression régulière. Cette variable s'appelle `$grammar`. Trois méthodes, `zero`, `one` et `grammar`, permettent d'accéder à ces trois variables de classes. On les définit de manière un peu « magique », de façon à pouvoir les redéfinir automatiquement dans les classes qui héritent de `WFST::Weight` :

```
do {
  no strict "refs";
  for my $var qw(one zero grammar) {
    *$var = sub { ${(ref $_[0] || $_[0]) . "::$var"} };
  }
};
```

Si les fonctions `zero`, `one` et `grammar` sont définies et renvoient la valeur correspondant à leur nom, les variables elles-mêmes ne sont pas définies.

Enfin, les différentes opérations sont définies par les fonctions `wadd` (addition), `wmul` (multiplication), `wdiv` (division), `wcmp` (comparaison) et `wval` (qui renvoie simplement la valeur scalaire du poids). Les deux dernières sont déjà définies, par contre les trois premières doivent obligatoirement être redéfinies par les classes qui héritent de `WFST::Weight` comme on va le voir plus bas.

```
sub wadd {}
sub wmul {}
sub wdiv {}
sub wcmp {
  (${$_[0]} <=> (ref $_[1] ? ${$_[1]} : $_[1])) * ($_[2] ? -1 : 1)
}
sub wval { ${$_[0]} }
```

On note pour finir que le premier paramètre de chacune de ces fonctions est toujours une instance de poids, mais que le second peut être soit un poids, soit un scalaire.

7.1.1.1 Le demi-anneau tropical

La définition du demi-anneau tropical est la suivante :

```
package WFST::Weight::Tropical;
use vars qw(@ISA $one $zero $grammar);
@ISA  = qw(WFST::Weight);

$zero = new WFST::Weight::Tropical "inf";
$one  = new WFST::Weight::Tropical 0;
$grammar = q{
  weight   : /\-?\d*\.?\d+/
           { new WFST::Weight::Tropical($item[1]) }
};

sub wadd { $ {$_[0]} < (ref $_[1] ? $ {$_[1]} : $_[1]) ? $_[0] : $_[1] }
sub wmul { $_[0]->new($ {$_[0]} + (ref $_[1] ? $ {$_[1]} : $_[1])) }
sub wdiv { $_[0]->new($ {$_[0]} - (ref $_[1] ? $ {$_[1]} : $_[1])) }
```

L'addition est en réalité la fonction *min* qui renvoie le minimum de deux valeurs ; la multiplication est en fait l'addition, et la division n'est autre que la soustraction. Logiquement, les variables `zero`

et `one` sont définies à ∞ et 0 respectivement. La variable grammaire contient une règle qui définit un poids comme un nombre réel.

7.1.1.2 Le demi-anneau réel

Dans le même ordre d'idée, le demi-anneau réel est défini comme suit :

```
package WFST::Weight::Real;
use vars qw(@ISA $one $zero $grammar);
@ISA  = qw(WFST::Weight);

$zero = new WFST::Weight::Real 0;
$one  = new WFST::Weight::Real 1;
$grammar = q{
  weight   : /\-?\d*\.?\d+/
           { new WFST::Weight::Real($item[1]) }
};

sub wadd { $_[0]->new($ {$_[0]} + (ref $_[1] ? $ {$_[1]} : $_[1])) }
sub wmul { $_[0]->new($ {$_[0]} * (ref $_[1] ? $ {$_[1]} : $_[1])) }
sub wdiv { $_[0]->new($ {$_[0]} / (ref $_[1] ? $ {$_[1]} : $_[1])) }
```

7.1.1.3 Le demi-anneau booléen

Enfin, le demi-anneau booléen qui utilise des entiers (Perl ne définit pas de vrais booléens : comme en C, une valeur nulle est fausse et une valeur non-nulle est vraie) :

```
package WFST::Weight::Boolean;
use vars qw(@ISA $one $zero $grammar);
@ISA = qw(WFST::Weight);

$zero = new WFST::Weight::Boolean 0;
$one  = new WFST::Weight::Boolean 1;
$grammar = q{
  weight   : /[01]/
           { new WFST::Weight::Boolean($item[1]) }
};

sub wadd { $_[0]->new($ {$_[0]} || (ref $_[1] ? $ {$_[1]} : $_[1])) }
sub wmul { $_[0]->new($ {$_[0]} && (ref $_[1] ? $ {$_[1]} : $_[1])) }
sub wdiv { $_[0]->new($ {$_[0]} && !(ref $_[1] ? $ {$_[1]} : $_[1])) }
```

Ici, l'opération additive est le « ou » logique (dont l'élément neutre est 0) et l'opération multiplicative est le « et » logique (dont l'élément neutre est 1). On a défini la division comme la multiplication par l'inverse, soit $a \wedge \overline{b}$.

7.1.2 Items et symboles

Un transducteur d'états finis pondérés est défini sur un ensemble de poids et sur deux alphabet. Dans cette bibliothèque, les alphabets ne sont pas définis précisément : les arcs sont étiquetés par des symboles, qui se décomposent en deux items, un pour chaque niveau de l'arc.

7.1.2.1 Items

Dans les bibliothèques d'états finis (pondérés ou non) existantes, les items sont généralement des chaînes de caractères, ce qui est tout à fait logique pour le TALN. Cependant, dans Sumo un item de segmentation peut être beaucoup plus complexe que cela.

La classe `WFST::Item` définit un item comme une chaîne de caractères (en réalité, comme un scalaire Perl, mais la différence n'est pas significative). Cependant, comme pour les poids, il est très facile de redéfinir ce qui constitue un item en spécifiant une sous-classe de `WFST::Item` (par exemple, `WFST::Item::Sumo`). Ainsi, la « définition » d'un item se décompose en deux parties : le constructeur de l'item, `new`, et la ou les règles de grammaire décrivant la syntaxe de l'item.

La définition de base est la suivante :

```
%STASH = ();

sub new {
  my ($class, $string) = @_;
  if (defined $string && $string ne "") {
    my $qstring = quote($string);
    if (exists $STASH{$qstring}) {
      $STASH{$qstring};
    } else {
      my $item = bless \$string, ref $class || $class;
      $STASH{$qstring} = $item;
      $item;
    }
  } else {
    new WFST::Item::Epsilon;
  }
}

$GRAMMAR = q{
  item     : "0"             {new WFST::Item::Epsilon}
           | "."             {new WFST::Item::Default}
           | /([1-9a-zA-Z_\x80-\xff]|\\\\u[\da-f]{1,4}|\\\\.)+/i
                             {$item[1] =~ s/\\\\u([\da-f]{1,4})/
                                WFST::Regex::ucs2utf8(hex $1)/gexi;
                              $item[1] =~ s/\\\\(.)/$1/g;
                              new WFST::Item($item[1])}
           | /\"((?:[^\\\\\"]|(?:\\\\.))+)\"/
                             {(my$str=$1) =~ s/\\\\(.)/$1/g;
                              new WFST::Item($str)}
};
```

Le tableau associatif `STASH` est la réserve contenant les items déjà créés, afin de ne pas créer de multiples copies d'un même item. Les items sont indexés par leur forme « protégée », où les caractères ayant une signification particulière sont précédés d'un \. La méthode `quote` transforme également les caractères non-imprimables et les caractères d'espacement pour une meilleure lisibilité.

```
sub quote {
  local $_ = shift;
  s/[.0:<>\\ ]/\\$&/g;
  s/\t/\\t/g;
  s/\n/\\n/g;
  s/\r/\\r/g;
```

```
  s/[\x00-\x1f\x7f]/sprintf"\\x%02x",ord($&)/ge;
  $_;
}
```

Les deux méthodes principales des items sont le test d'égalité entre deux items, et le test de correspondance entre un item et une chaîne de caractères.

```
sub equal { $_[0]->key eq $_[1]->key }

sub match_str {
  my ($self, $string) = @_;
  my $candidate = substr($string, 0, length $$self);
  $candidate eq $$self ? $candidate : undef;
}
```

L'égalité est triviale, c'est l'identité des objets (ce qui découle de l'utilisation de la réserve). La méthode `key` renvoie simplement un identificateur de l'objet. Par exemple, la clé d'un item ressemble à `WFST::Item=SCALAR(0x814a274)`. Deux objets identiques ont donc la même clé.

La méthode `match_str` renvoie le plus long préfixe de la chaîne de caractères donnée en paramètre correspondant à l'item. Si aucun préfixe ne peut être trouvé, une valeur indéfinie est renvoyée. Le préfixe retourné peut être vide si l'item est epsilon, comme on le voit ci-dessous.

7.1.2.2 Items spéciaux

Les méthodes suivantes sont des prédicats sur les propriétés d'un item :

```
sub regular { 1 }
sub epsilon { 0 }
sub default { 0 }
```

Il existe deux sortes d'items spéciaux : premièrement, ceux qui ont une signification différente des items normaux et qui sont disponibles dans les expressions régulières. Ces items sont epsilon (noté `0`) et le symbole universel (noté `"."`). Deuxièmement, on a des items spéciaux qui sont utilisés en interne, par exemple les symboles epsilon utilisés par l'algorithme de composition ou pour les règles de réécriture. On ne décrit pas ceux-ci plus avant ; ce sont simplement des « coquilles vides » qui permettent de les différencier des autres items par les algorithmes qui les utilisent.

Évidemment, pour les items spéciaux, le prédicat `regular` est redéfini :

```
sub regular { 0 }
```

Epsilon. La définition de l'item epsilon est :

```
package WFST::Item::Epsilon;
our @ISA = qw(WFST::Item::Special);
sub as_string { "0" }
sub epsilon   { 1 }
match_str     { "" }
```

La représentation d'un epsilon est `0` dans Sumo (dans une version purement Unicode, on pourrait utiliser un véritable caractère ϵ). La méthode `match_str` renvoie toujours une valeur définie, qui est la chaîne vide.

Item universel. La définition de l'item universel est :

```
package WFST::Item::Default;
our @ISA = qw(WFST::Item::Special);
sub as_string { "." }
sub default() { 1 }
sub match_str { undef }
```

La représentation du symbole univeresl est "`.`" dans Sumo. La méthode `match_str` renvoie toujours une valeur indéfinie ; c'est un symbole particulier qui doit être traité séparément (comme on le verra dans la section 7.2).

7.1.2.3 Symboles

Un symbole a deux côtés, le côté supérieur et le côté inférieur. Il contient un item supérieur et un item inférieur optionnel (s'il n'est pas défini, l'item inférieur est considéré comme étant le même que l'item supérieur ; le symbole est alors dit « simple »). On maintient une liste de symboles déjà créés dans une réserve (`STASH`).

Les items dans la réserve sont indexés par leur représentation sous forme d'une chaîne de caractères, qui est définie par la fonction `as_string`. Un symbole a deux formes différentes : s'il est simple, il s'agit simplement de l'item supérieur (par exemple, `a`). Sinon, les deux symboles sont séparés par `":"` et le symbole est délimité par des chevrons (par exemple, `<a:b>`).

```
%STASH = ();

sub new {
  my ($class, $upper, $lower) = @_;
  $upper = ref $upper ? $upper : new WFST::Item($upper);
  $lower = defined $lower ? ref $lower ?
    $lower : new WFST::Item($lower) : undef;
  my $sym = bless [$upper], ref $class || $class;
  push @$sym, $lower if (defined $lower) && ($upper ne $lower);
  my $key = $sym->as_string;
  if (exists $STASH{$key}) {
    $STASH{$key};
  } else {
    $STASH{$key} = $sym;
    $sym;
  }
}

sub as_string { defined $_[0]->[1] ? "<$_[0]->[0]:$_[0]->[1]>" : $_[0]->[0]; }
```

Les méthodes principales des symboles sont les suivantes :

- `simple` teste si un symbole est simple :

  ```
  sub simple    { !defined $_[0]->[1] }
  ```

- `upper` et `lower` accèdent à l'item supérieur et inférieur du symbole :

  ```
  sub upper     { $_[0]->[0] }
  sub lower     { $_[0]->[defined $_[0]->[1]] }
  ```

- `epsilon` teste si un symbole est epsilon, c'est-à-dire s'il est simple et que son item est epsilon :

  ```
  sub epsilon   { !defined $_[0]->[1] && $_[0]->[0]->epsilon }
  ```

- `default` teste de la même manière si un symbole est universel :

  ```
  sub default   { !defined $_[0]->[1] && $_[0]->[0]->default }
  ```

- `invert` intervertit les deux côtés du symbole :

  ```
  sub invert { $_[0]->new(reverse @{$_[0]}) }
  ```

- `equal` teste l'égalité entre deux symboles (ils ont la représentation sous forme de chaîne) :

  ```
  sub equal     { $_[0]->key eq $_[1]->key }
  ```

– `match_str` teste si le symbole correspond à une chaîne de caractère, c'est-à-dire si l'item supérieur correspond à cette chaîne (voir plus bas) :

```
sub match_str { $_[0]->[0]->match_str($_[1])}
```

– `match` teste si deux symboles correspondent, c'est-à-dire si l'item inférieur du premier symbole correspond à l'item supérieur du deuxième symbole :

```
sub match {
  $_[0]->[0]->equal($_[1]->[0]) &&
    $_[0]->[defined $_[0]->[1]?1:0]->equal($_[1]->[defined $_[1]->[1]?1:0]);
}
```

7.1.3 États, arcs et transducteurs

7.1.3.1 États

Bien que ne portant pas réellement d'information, un état comporte pourtant une multitude d'informations. Pour représenter des structures complexes, on utilise régulièrement des tableaux associatifs.

La méthode `new` crée un nouvel état, avec pour paramètres optionnels un poids initial et/ou un état final. Le seul fait qu'un de ces poids soit défini indique si l'état est final et/ou initial (d'où les deux fonctions qui testent si un état est initial ou final plus bas). L'état est alors créé avec les paramètres suivants :

– `id` : un entier servant d'identificateur (voir la méthode `name` plus bas) ;
– `iweight`, `fweight` : le poids initial et le poids final ;
– `arcs`, `epsilon`, et `default` : les listes d'arcs normaux, epsilon et universel dont l'état est origine. Les arcs normaux sont classés par étiquette, puis par destination ; les autres arcs, qui ont tous la même étiquette, sont classés par destination ;
– `deterministic` : une propriété de l'état qui indique si celui-ci est déterministe.

```
sub new {
  my ($class, $fweight, $iweight) = @_;
  $fweight = $WFST::WREF->new($fweight) if defined $fweight && !ref $fweight;
  $iweight = $WFST::WREF->new($iweight) if defined $iweight && !ref $iweight;
  $class = ref $class || $class;
  bless { _id            => $ID++,
          _iweight       => $iweight,
          _fweight       => $fweight,
          _arcs          => {},
          _epsilon       => {},
          _default       => {},
          _deterministic => 1,
        }, $class;
}
```

Par convention, on nomme un item sn , où n est l'identificateur de l'item. De plus, un le nom d'un état initial est préfixé par `i` et celui d'un état final par `f` ; on numérote les états à partir de zéro. Dans la représentation graphique d'un automate, les états initiaux et finals montrent également leur poids initial et/ou final. Par exemple, l'état initial zéro d'un automate avec un poids initial de `1.2` sera noté `1.2/is0`. Les deux méthodes `name` et `wname` (avec l'affichage des poids) sont définies ainsi :

```
sub name {
  (defined $_[0]->{_iweight} ? "i" : "") .
    (defined $_[0]->{_fweight} ? "f" : "") . "s" . $_[0]->{_id};
```

```
}

sub wname {
  (defined $_[0]->{_iweight} ? "$_[0]->{_iweight}/i" : "") .
    (defined $_[0]->{_fweight} ? "f" : "") . "s" . $_[0]->{_id} .
      (defined $_[0]->{_fweight} ? "/$_[0]->{_fweight}" : "");
}
```

Les méthodes suivantes rendent un état initial ou non (`make_initial` et `make_non_inital`), et final ou non (`make_final` et `make_non_final`). Pour cela, il suffit de modifier ou de supprimer la valeur du poids correspondant. La méthode `make_initial` (et symétriquement, `make_final`) permet des raccourcis, comme par exemple de ne pas spécifier le poids initial (ou final), auquel cas c'est un poids nul qui est choisi.

```
sub make_initial {
  my ($self, $iweight, $keep) = @_;
  $iweight = $WFST::WREF->one if !defined $iweight;
  $iweight = $WFST::WREF->new($iweight) if !ref $iweight;
  my $prevw = defined $keep && $keep && defined $self->{_iweight} ?
    $self->{_iweight} : $WFST::WREF->zero;
  $self->{_iweight} = $prevw + $iweight;
  $self;
}

sub make_non_initial { $_[0]->{_iweight} = undef; $_[0] }

sub make_final {
  my ($self, $fweight, $keep) = @_;
  $fweight = $WFST::WREF->one if !defined $fweight;
  $fweight = $WFST::WREF->new($fweight) if !ref $fweight;
  my $prevw = defined $keep && $keep && defined $self->{_fweight} ?
    $self->{_fweight} : $WFST::WREF->zero;
  $self->{_fweight} = $prevw + $fweight;
  $self;
}

sub make_non_final   { $_[0]->{_fweight} = undef; $_[0] }
```

Enfin, deux méthodes importantes sont celles qui permettent d'ajouter et de supprimer un arc partant de cet état. La méthode `add_arc` ajoute un arc passé en paramètre :

```
sub add_arc {
  my ($self, $arc) = @_;
  my $added = 0;
  $arc->set(from => $self) if !defined $arc->from;
  if ($arc->epsilon) {
    if (my $parc = $self->{_epsilon}->{$arc->to->key}) {
      $parc->set(weight => $parc->weight + $arc->weight);
    } else {
      $self->{_epsilon}->{$arc->to->key} = $arc;
      $self->{_deterministic} = 0;
      ++$added;
    }
  } elsif ($arc->default) {
    if (my $parc = $self->{_default}->{$arc->to->key}) {
      $parc->set(weight => $parc->weight + $arc->weight);
```

```
      } else {
        $self->{_default}->{$arc->to->key} = $arc;
        $self->{_deterministic} = 0 if %{$self->{_arcs}} || %{$self->{_default}};
        ++$added;
      }
    } else {
      if (exists $self->{_arcs}->{$arc->label}) {
        if (my $parc = $self->{_arcs}->{$arc->label}->{$arc->to->key}) {
          $parc->set(weight => $parc->weight + $arc->weight);
        } else {
          $self->{_deterministic} = 0;
          $self->{_arcs}->{$arc->label}->{$arc->to->key} = $arc;
          ++$added;
        }
      } else {
        $self->{_arcs}->{$arc->label}->{$arc->to->key} = $arc;
        ++$added;
      }
    }
    $added;
  }
```

On distingue trois cas, selon que l'arc porte un symbole normal, epsilon, ou universel (car il y a trois listes d'arcs différentes). Dans chaque cas, si un arc vers la même destination existe déjà, son poids est ajusté ; le nouveau poids de l'arc est la somme du poids précédent et du poids de l'arc ajouté. Si aucun arc n'existait précédemment, il est simplement ajouté à la liste. Enfin, une valeur booléenne est renvoyée (`$added`) indiquant si un nouvel arc a effectivement été ajouté.

La méthode `rm_arc` supprime un arc donné depuis un état et retourne l'arc supprimé :

```
  sub rm_arc {
    my ($self, $arc) = @_;
    if ($arc->epsilon) {
      delete $self->{_epsilon}->{$arc->to->key};
    } elsif ($arc->default) {
      delete $self->{_default}->{$arc->to->key};
    } else {
      my $a = delete $self->{_arcs}->{$arc->label}->{$arc->to->key};
      if (!%{$self->{_arcs}->{$arc->label}}) {
        delete $self->{_arcs}->{$arc->label};
      }
      $a;
    }
  }
```

7.1.3.2 Arcs

Un arc est une structure plus simple, définie par quatre attributs :

- une étiquette (`label`), qui est un symbole ;
- un poids (`weight`) ;
- un sommet origine (`from`), et
- un sommet destination (`to`).

Cela donne le constructeur suivant :

```
sub new {
  my ($class, %args) = @_;
  my $label = exists $args{label} ? $args{label} : new WFST::Item::Epsilon;
  $label = new WFST::Symbol($args{label}) if ref $label ne "WFST::Symbol";
  my $weight = exists $args{weight} ? $args{weight} : $WFST::WREF->one;
  $weight = $WFST::WREF->new($weight) if !ref $weight;
  my $arc = { _label  => $label,
      _weight => $weight,
      _from   => $args{from},
      _to     => $args{to},
    };
  bless $arc, ref $class || $class;
  $arc;
}
```

Les attributs sont passés en paramètres par un tableau associatif ; la plupart sont facultatifs. Un arc portant un symbole, les méthodes et prédicats définis sur les symboles sont définis de même sur les arcs et permettent un accès transparent à leur étiquette (revoir la section 7.1.2.3).

7.1.3.3 Transducteurs

Reprenant la terminologie de Xerox, les transducteurs de `WFST.pm` sont appelés réseaux (*networks*), et sont aussi bien simples (automates) que complexes (transducteurs). C'est évidemment la structure la plus complexe, définie par les attributs suivants :

- un nom (`name`), qui permet d'identifier le réseau ;
- la liste des états initiaux (`initial`) ;
- la liste des états finals (`final`) ;
- la liste de tous les états (`state`) ;
- la liste des arcs (`arcs`) ;
- l'alphabet (`sigma`) ;
- le nombre de chemins dans le réseaux (`paths`). Ce nombre peut être positif ou nul si le réseau est acyclique, et vaut par convention `-1` si le réseau est cyclique, auquel cas le nombre de chemins est infini ;
- une liste de propriétés booléennes (`props`) :
  - `deterministic` : vrai si le réseau est déterministe (s'il a été déterminisé) ;
  - `has_epsilon` : vrai si le réseau contient des epsilon-arcs ;
  - `has_cycle` : vrai si le réseau contient un cycle ;
  - `minimal` : vrai si le réseau a un nombre d'états minimal ;
  - `trim` : vrai si le réseau est connecté ;
  - `simple` : vrai pour un automate, faux pour un transducteur ;
  - `complete` : vrai si le réseau est complet.

Le constructeur permet de créer plusieurs types de réseaux fondamentaux et est donc un peu complexe :

```
sub new {
  my ($class, %args) = @_;
  my $start = new WFST::State(undef, $WFST::WREF->one);
  my $net = bless
    { _name    => $args{name},
      _initial => {$start->key => $start},
      _final   => {},
      _states  => {$start->key => $start},
      _arcs    => {},
```

```
      _sigma   => {},
      _paths   => 0,
      _prop    => { deterministic => 1,
                    has_epsilon   => 0,
                    has_cycle     => 0,
                    minimal       => 1,
                    trim          => $start->final,
                    simple        => 1,
                    complete      => 1,
                  },
    }, ref $class || $class;

  if (defined $args{arc}) {
    $net->{_name} = $args{arc}->label if !defined $net->{_name};
    if ($args{arc}->epsilon) {
      $start->make_final($args{arc}->weight);
    } else {
      my $weight = exists $args{weight} ? $args{weight} : $WFST::WREF->one;
      my $fs = new WFST::State($weight);
      $net->_add_arc($args{arc}->set(from => $start, to => $fs));
    }
    ++$net->{_paths};
  }
  $net->{_name} = "net" . $ID++ if !defined $net->{_name};
  $net;
}
```

On peut dénombrer trois types de réseaux fondamentaux ; les deux premiers peuvent être construits directement par le constructeur `new`, le troisième demande une étape supplémentaire.

Premièrement, le réseau qui reconnaît le langage $\{\epsilon\}$, qui ne contient qu'un état, qui est initial et final à la fois. Deuxièmement, le réseau qui reconnaît le langage $\{a\}$ pour un symbole a donné. Cet automate est constitué d'un état initial, d'un état final et d'un unique arc entre ces deux états. Troisièmement, le réseau qui reconnaît le langage vide, qui ne contient aucun chemin. Il est constitué d'un unique état initial qui n'est pas final. Cet automate particulier est généralement le résultat d'une opération (par exemple, une intersection vide) et est rarement créé de manière volontaire.

Les algorithmes complexes sur les réseaux d'états finis pondérés sont l'objet de la section suivante ; il reste à présenter les méthodes primitives que ces algorithmes utilisent. Les méthodes correspondantes sont précédées d'un _, qui est une manière d'indiquer qu'elles sont « privées » (toutes les méthodes sont publiques en Perl ; cette convention sert à décourager l'utilisation de ces méthodes à usage interne depuis l'extérieur). Les deux principales sont `_add_state` (ajout d'un état) et `_add_arc` (ajout d'un arc), qui mettent à jour la liste des états et des arcs, ainsi que les propriétés associées et l'alphabet.

```
sub _add_state {
  my ($self, $state) = @_;
  if (!$self->{_states}->{$state->key}) {
    $self->{_states}->{$state->key} = $state;
    $self->{_initial}->{$state->key} = $state if $state->initial;
    $self->{_final}->{$state->key} = $state if $state->final;
    $self->{_prop}->{minimal} = 0;
    $self->{_prop}->{trim} = 0;
  }
  $state;
}
```

```
sub _add_arc {
  my ($self, $arc) = @_;
  if (!exists $self->{_arcs}->{$arc->key}) {
    my ($source, $dest) = ($arc->from, $arc->to);
    $self->_add_state($source);
    $self->_add_state($dest);
    $self->{_arcs}->{$arc->key} = $arc;
    $source->add_arc($arc);
    if ($arc->epsilon) {
      $self->{_prop}->{has_epsilon}++;
    } else {
      $self->{_sigma}->{$arc->label}++;
    }
    $self->{_prop}->{deterministic} = 0 if !$source->deterministic;
    $self->{_prop}->{has_cycle}     = 1 if $source == $dest;
    $self->{_prop}->{simple}        = 0 if !$arc->simple;
    $self->{_prop}->{minimal}       = 0;
    $self->{_prop}->{complete}      = 0;
  }
}
```

On note que `_add_state` se contente d'ajouter un état au réseau, mais pas les arcs pour cet état, alors que `_add_arc` ajoute (si nécessaire) les états extémités de l'arc au réseau.

Un premier pas vers l'optimisation de la compilation de lexiques (c'est-à-dire de listes de mots) est la fonction `add_path` qui ajoute un chemin entier. Le chemin est constitué d'une liste de couples (symbole, poids), et des arcs sont créés automatiquement depuis l'état initial du transducteur jusqu'à un nouvel état final (de poids unitaire par défaut).

```
sub add_path {
  my $self = shift;
  my $current = $self->_unique_start_state;
  for (@_) {
    my ($sym, $weight) = @$_;
    my $arc = new WFST::Arc(from   => $current,
                            to     => new WFST::State,
                            label  => $sym,
                            weight => $weight,
                            add    => 0);
    $self->_add_arc($arc);
    $current = $arc->to;
  }
  $current->make_final;
  $self;
}
```

Deux autres primitives « internes » fréquemment utilisées concernent l'état initial. Celui-ci est toujours unique, sauf durant certaines phases de traitement ; la méthode `_unique_start_state` permet de s'en assurer : elle renvoie l'état initial du réseau ou génère une erreur s'il n'y a pas d'état initial, ou s'il y en a plus d'un. La méthode `_new_start_state`, quant à elle, crée un nouvel état initial et renvoie l'ancien (et unique) état initial, ainsi que le nouveau. Le nouvel état initial est créé avec le même poids initial que son prédécesseur, ainsi qu'un éventuel poids initial. L'ancien état initial reste dans le graphe, mais se voit retirer son poids initial.

```
sub _unique_start_state {
```

```
  my $self = shift;
  my @start = values %{$self->{_initial}};
  if (@start == 0) {
    my $caller = (caller(1))[3];
    confess "Network $self->{_name} has no start state";
  } elsif (@start > 1) {
    my $caller = (caller(1))[3];
    confess "Network $self->{_name} has several start states";
  }
  $start[0];
}

sub _new_start_state {
  my ($self, $fweight) = @_;
  my $prev = $self->_unique_start_state;
  my $new = new WFST::State($fweight, $prev->iweight);
  $self->_add_arc(new WFST::Arc(from   => $new,
                                to     => $prev,
                                weight => $WFST::WREF->one));
  (delete $self->{_initial}->{$prev->key})->make_non_initial;
  $self->{_final} = {$new->key => $new} if $new->final;
  ($new, $prev);
}
```

Enfin, l'affichage d'un réseau d'états finis peut se faire soit textuellement, soit graphiquement. La partie graphique est déléguée au programme `dot`[1] de AT&T qui convient particulièrement à cette tâche (la plupart des transducteurs d'états finis pondérés illustrés ici ont été produits par `dot` et `WFST.pm`). La représentation textuelle est elle similaire à celle de Xerox et est illustrée ci-dessous (on reconnaîtra l'automate de la figure 1.7) :

```
Network "Chiffres romains":
Size: 6 states, 8 arcs, 10 paths.
Sigma: i (5), v (2), x (1).
Properties: not complete, deterministic, no cycle, no epsilon, minimal,
  simple, trim.
States:
0/ifs0/0: i/-1 -> fs1, v/5 -> fs2.
fs1/2: i/3 -> fs3, v/5 -> fs4, x/10 -> fs4.
fs2/0: i/1 -> fs5.
fs3/0: i/1 -> fs4.
fs4/0: no arcs.
fs5/0: i/1 -> fs3.
```

La taille du réseau est exprimée en fonction du nombre d'états, d'arcs et de chemins. Chaque symbole de l'alphabet (« sigma ») est suivi de son nombre d'occurrences dans le réseau. Ensuite, les propriétés du réseau sont énumérées (celui-ci est un automate déterministe, acyclique, sans epsilon-transition, minimal, connecté mais pas complet). Enfin, on affiche la liste des états de l'automate et pour chaque état, la liste des arcs ayant cet état pour origine. Pour chaque arc, on affiche son étiquette, son poids et son état destination.

Pour obtenir une information rapide sur un transducteur, on peut se contenter de sa signature, qui reprend les deux premières lignes de l'affichage ci-dessus : le nom du transducteur et sa taille en états, arcs et chemins.

```
sub signature {
```

[1] `http://www.research.att.com/sw/tools/graphviz/`

```
  my $self = shift;
  my $states = keys %{$self->{_states}};
  my $narcs = keys %{$self->{_arcs}};
  my $npaths = $self->{_paths};
  "$self->{_name}, $states state" . ($states > 1 ? "s" : "") .
    ", $narcs arc" . ($narcs > 1 ? "s" : "") . ", " .
      ($npaths < 0 ? "circular" :
       ("$npaths path". ($npaths > 1 ? "s" : "")));
}
```

7.2 Algorithmes sur les transducteurs d'états finis pondérés

7.2.1 Propriétés des transducteurs d'états finis pondérés

7.2.1.1 Suppression des epsilon-transitions

Il est toujours possible de supprimer les epsilon-transitions dans un transducteur. Ceci concerne les epsilon-transitions « pures », et évidemment pas les transitions dont le symbole contient un epsilon inférieur ou supérieur. On adapte ici la technique classique qui consiste à calculer l'epsilon-fermeture du transducteur, c'est-à-dire à déterminer l'ensemble des états accessibles depuis un état donné en suivant un nombre quelconque d'epsilon-transitions.

La fonction `_epsilon_closure` calcule une matrice de transitions de manière itérative. Cette matrice est remplie avec le poids des epsilon-transitions entre chaque couple d'états, ou une valeur indéfinie s'il n'y a pas d'epsilon-transition entre les deux états (initialisation de la matrice, lignes 2 à 17). Ensuite, pour chaque triplet d'états, s'il y a deux epsilon-transitions successives, on crée une nouvelle epsilon-transition entre le premier et le troisième état (lignes 18 à 27).

La matrice `@t` ainsi créée est retournée par cette fonction qui prend comme paramètres la liste des états du transducteur (dans le tableau `@_`, comme d'habitude en Perl).

```
 1  sub _epsilon_closure {
 2    my @t = ();
 3    for my $i (0 .. $#_) {
 4      for my $j (0 .. $#_) {
 5        if ($i == $j) {
 6          $t[$i]->[$j] = $WFST::WREF->one;
 7        } else {
 8          my @e = grep { $_->to == $_[$j] } values %{$_[$i]->epsilon};
 9          if (@e) {
10            $t[$i]->[$j] = $WFST::WREF->zero;
11            $t[$i]->[$j] = $t[$i]->[$j] + $_->weight for @e;
12          } else {
13            $t[$i]->[$j] = undef;
14          }
15        }
16      }
17    }
18    for my $k (0 .. $#_) {
19      for my $i (0 .. $#_) {
20        for my $j (0 .. $#_) {
21          if (defined $t[$i]->[$k] && defined $t[$k]->[$j]) {
22            my $tt = $t[$i]->[$k] * $t[$k]->[$j];
```

```
23            $t[$i]->[$j] = $tt if !defined $t[$i]->[$j] || $t[$i]->[$j] < $tt;
24          }
25        }
26      }
27    }
28    @t;
29  }
```

La suppression des epsilon-transitions se fait alors en deux étapes. Premièrement (lignes 8 à 33), on ajoute de nouvelles transitions. Pour chaque couple d'états distincts (u, v) pour lesquels il existe une epsilon-transition (comme on l'a calculé dans la fermeture, lignes 6-7), toutes les transitions ayant v pour origine sont copiées pour l'état u (lignes 11-30). Deuxièmement, les epsilon-transitions sont retirées (lignes 34-37, et voir plus bas).

```
 1  our %visited;
 2  sub rm_epsilon {
 3    my ($self, $copy) = @_;
 4    if ($self->{_prop}->{has_epsilon}) {
 5      my $net = defined $copy && $copy ? $self->clone : $self;
 6      my @V = sort { $a->id <=> $b->id } values %{$net->{_states}};
 7      my @closure = _epsilon_closure(@V);
 8      for my $i (0 .. $#V) {
 9        for my $j (0 .. $#V) {
10          if ($i != $j && defined $closure[$i]->[$j]) {
11            for ((map { values %$_ } values %{$V[$j]->arcs}),
12                 values %{$V[$j]->default}) {
13              my $we = $closure[$i]->[$j];
14              my $arc = undef;
15              if (exists $V[$i]->arcs->{$_->label}) {
16                for my $a (values %{$V[$i]->arcs->{$_->label}}) {
17                  if ($a->to == $_->to) {
18                    $arc = $a;
19                    last;
20                  }
21                }
22              }
23              if (defined $arc) {
24                $arc->set(weight => $_->weight * $we + $arc->weight);
25              } else {
26                my $arc = $_->clone(from => $V[$i]);
27                $arc->set(weight => $_->weight * $we);
28                $net->_add_arc($arc);
29              }
30            }
31          }
32        }
33      }
34      %visited = ();
35      $net->_remove_epsilon_arcs($_) for values %{$net->{_initial}};
36      $net->{_prop}->{has_epsilon} = 0;
37      $net->{_prop}->{trim} = 0;
38      $net;
39    } else {
40      $self;
```

```
41    }
42  }
```

La suppression des transitions est effectuée en parcourant le transducteur en profondeur d'abord (de manière récursive), pour supprimer d'abord les transitions les plus « profondes ». En effet, si l'on supprime une epsilon-transition menant à un état final, alors l'état qui était à l'origine de cette transition devient lui-même final, et son poids est le produit du poids de l'état final et de celui de la transition qui y mène.

```
sub _remove_epsilon_arcs {
  my ($self, $state) = @_;
  if (!$visited{$state}) {
    ++$visited{$state};
    for ($state->all_arcs) {
      $self->_remove_epsilon_arcs($_->to);
    }
    for (values %{$state->epsilon}) {
      my $arc = $state->rm_arc($_);
      if ($arc->to->final) {
        $arc->from->make_final($arc->weight * $arc->to->fweight);
        $self->{_final}->{$arc->from->key} = $arc->from;
      }
      delete $self->{_arcs}->{$arc->key};
    }
  }
}
```

7.2.1.2 Connexion

La connexion d'un transducteur consiste à retirer tous les états et transitions qui ne font pas partie d'un chemin. On commence par déterminer quels sont les états accessibles (pour lesquels il existe un sous-chemin ayant pour origine un état initial), puis quels sont les états co-accessibles (pour lesquels il existe un sous-chemin ayant pour destination un état final). Les états inaccesssibles sont éliminés à chaque étape.

```
sub trim {
  my ($self, $copy) = @_;
  if (!$self->{_prop}->{trim}) {
    my $net = defined $copy && $copy ? $self->clone : $self;
    my %accessible = ();
    my @queue = values %{$net->{_initial}};
    while (@queue) {
      my $state = shift @queue;
      ++$accessible{$state->key};
      push @queue, $_->to
      for grep { !$accessible{$_->to->key} } $state->all_arcs;
    }
    $net->_remove_states(%accessible);
    %accessible = ();
    @queue = values %{$net->{_final}};
    while (@queue) {
      my $state = shift @queue;
      ++$accessible{$state->key};
      push @queue, $_->from
```

```
        for grep { $_->to == $state && !$accessible{$_->from->key} }
          values %{$net->{_arcs}};
    }
    $net->_remove_states(%accessible);
    $net = new WFST::Network if !keys %{$net->{_states}};
    $net->{_prop}->{trim} = 1;
    $net;
  } else {
    $self;
  }
}
```

La suppression des états qui ne sont pas accessibles ou co-accessibles s'accompagne évidemment de la suppression des transitions de et vers ces états. La fonction `_remove_states` teste pour chaque état s'il est (co-)accessible ou non. S'il ne l'est pas (lignes 5 à 12), on le supprime de la liste des états du transducteur, ainsi que les transitions dont il est l'origine. Sinon (lignes 14 à 18), on teste chacune des transitions quittant l'état pour supprimer celles qui mènent à un état inaccessible.

```
 1  sub _remove_states {
 2    my ($self, %accessible) = @_;
 3    for (values %{$self->{_states}}) {
 4      if (!$accessible{$_->key}) {
 5        for ($_->all_arcs) {
 6          my $arc = delete $self->{_arcs}->{$_->key};
 7          delete $self->{_sigma}->{$_->label} if
 8            --$self->{_sigma}->{$_->label} == 0;
 9        }
10        delete $self->{_states}->{$_->key};
11        delete $self->{_initial}->{$_->key} if $_->initial;
12        delete $self->{_final}->{$_->key} if $_->final;
13      } else {
14        for my $arc (grep { !$accessible{$_->to->key} } $_->all_arcs) {
15          delete $self->{_arcs}->{$arc->key};
16          delete $self->{_sigma}->{$arc->label} if
17            --$self->{_sigma}->{$arc->label} == 0;
18          my $arc = $_->rm_arc($arc);
19        }
20      }
21    }
22    $self;
23  }
```

7.2.1.3 Déterminisation et minimisation

L'algorithme de déterminisation est celui vu dans la section 5.1.2, et on présente ici une réalisation en Perl. Un nouveau transducteur est créé à partir de zéro, et on le remplit avec les états et les transitions correspondant à la déterminisation.

```
sub determinize {
  my $self = shift;
  if (!$self->{_prop}->{deterministic}) {
    $self = $self->clone->rm_epsilon->trim;
    my $net = new WFST::Network(name => $self->{_name});
    my %states = ();
```

```
my %cstate = ();
%cstate = map { $_->name => [$_, $_->iweight] }
  values %{$self->{_initial}};
$states{join ";", map { "$_,$cstate{$_}->[1]" } keys %cstate} =
  (values %{$net->{_initial}})[0];
my @queue = (\%cstate);
while (@queue) {
  my $cstate = shift @queue;
  my $key = join ";", map { "$_,$cstate->{$_}->[1]" } keys %$cstate;
  my $current = $states{$key};
  if (my @finals = grep { $_->[0]->final } values %$cstate) {
    my $w = $WFST::WREF->zero;
    $w = $w + ($_->[0]->fweight * $_->[1]) for @finals;
    $current->make_final($w);
    $net->{_final}->{$current->key} = $current;
  }
  my %arcs = ();
  for my $cs (values %$cstate) {
    for my $arc ($cs->[0]->all_arcs) {
      $arcs{$arc->label->key}->{sym} = $arc->label;
      my $w = $arc->weight * $cstate->{$arc->from->name}->[1];
      if (exists $arcs{$arc->label->key}->{$arc->to->name}) {
        $arcs{$arc->label->key}->{$arc->to->name}->[1] = $w +
          $arcs{$arc->label->key}->{$arc->to->name}->[1];
      } else {
        $arcs{$arc->label->key}->{$arc->to->name} = [$arc, $w];
      }
    }
  }
  for my $symkey (sort keys %arcs) {
    my $sym = delete $arcs{$symkey}->{sym};
    my($key, $wmin, %ncstate);
    if ((my @arcs = values %{$arcs{$symkey}}) > 1) {
      $wmin = $arcs[0]->[1];
      for (@arcs[1 .. $#arcs]) {
        $wmin = $_->[1] if $_->[1] < $wmin;
      }
      %ncstate = ();
      for (@arcs) {
        $ncstate{$_->[0]->to->name} = [$_->[0]->to, $_->[1] / $wmin];
      }
      $key = join ";", map { "$_,$ncstate{$_}->[1]" } keys %ncstate;
    } else {
      $wmin = $arcs[0]->[1];
      $key = $arcs[0]->[0]->to->name . "," . $WFST::WREF->one;
      %ncstate = ($arcs[0]->[0]->to->name =>
                  [$arcs[0]->[0]->to, $WFST::WREF->one]);
    }
    my $ns = $states{$key};
    if (!defined $ns) {
      $ns = $states{$key} = new WFST::State;
      push @queue, \%ncstate;
    }
```

```
        my $arc = new WFST::Arc(from   => $current,
                                to     => $ns,
                                label  => WFST::Symbol->get_from_stash($sym),
                                weight => $wmin);
        $net->_add_arc($arc);
      }
    }
    $net->{_prop}->{deterministic} = 1;
    $net->{_prop}->{minimal} = 0;
    $net->renumber_states;
  } else {
    $self;
  }
}
```

La minimisation du nombre d'états d'un transducteur par l'algorithme de Brzozowsky est alors un jeu d'enfant :

```
sub minimize {
  my ($self, $copy) = @_;
  if (!$self->{_prop}->{minimal}) {
    my $net = defined $copy && $copy ? $self->clone : $self;
    $net =
      $net->rm_epsilon->trim->_reverse->determinize->_reverse->determinize;
    $net->{_prop}->{minimal} = 1;
    $net;
  } else {
    $self;
  }
}
```

7.2.1.4 Complétion

La complétion d'un transducteur est réalisée à l'aide d'un état puits, créé ligne 5. Ensuite, pour chaque état, on ajoute un arc défaut vers cet état puits, ce qui permet de capturer toutes les transitions « manquantes » pour un état donné.

```
 1  sub complete {
 2    my ($self, $copy) = @_;
 3    if (!$self->{_prop}->{complete}) {
 4      my $net = defined $copy && $copy ? $self->clone : $self;
 5      my $sink = new WFST::State;
 6      for my $state (values %{$net->{_states}}) {
 7        $net->_add_arc(new WFST::Arc(from => $state, to => $sink,
 8                                     label => new WFST::Item::Default))
 9          if values %{$state->{_default}} == 0;
10      }
11      if (exists $net->{_states}->{$sink->key}) {
12        $net->_add_arc(new WFST::Arc(from => $sink, to => $sink,
13                                     label => new WFST::Item::Default))
14      }
15      $net->{_prop}->{trim} = 0;
16      $net->{_prop}->{minimal} = 0;
17      $net->{_prop}->{complete} = 1;
```

```
18      $net;
19    } else {
20      $self;
21    }
22  }
```

7.2.1.5 Nombre de chemins et détection de cycles

Le décompte du nombre de chemins se fait par un parcours en profondeur d'abord du transducteur, en additionnant le nombre de chemins ayant pour origine chacun des états initiaux. La fonction de parcours est récursive. Comme le nombre de chemins est inifini dans un transducteur qui comporte au moins un circuit, cette propriété est testée en même temps que les chemins sont dénombrés. Le dénombrement s'arrête dès qu'un circuit est détecté.

```
sub _count_paths {
  my $self = shift;
  my $paths = 0;
  for (values %{$self->{_initial}}) {
    my $p = $self->_count_paths_r($_, ());
    if ($p < 0) {
      $paths = -1;
      last;
    } else {
      $paths += $p;
    }
  }
  $self->{_paths} = $paths;
  $self->{_prop}->{has_cycle} = $paths < 0;
  $self;
}

sub _count_paths_r {
  my ($self, $state, %seen) = @_;
  return -1 if $seen{$state->key}++;
  my $paths = 0;
  for ($state->all_arcs()) {
    my $p = $self->_count_paths_r($_->to, %seen);
    return -1 if $p < 0;
    $paths += $p;
  }
  ++$paths if $state->final();
  $paths;
}
```

7.2.1.6 Renumérotation des états

La renumérotation des états est une étape cosmétique mais pratique. Les états sont normalement numérotés à l'aide d'un compteur global, aussi au fur et à mesure des opérations la numérotation des états paraît confuse lorsque l'on affiche un transducteur. On renumérote ici les états en parcourant le transducteur en largeur d'abord, en partant de l'état initial qui est numéroté 0.

```
sub renumber_states {
  my $self = shift;
```

```
  my @stack = (values %{$self->{_initial}});
  my %visited = ();
  my @states = ();
  while (@stack) {
    my $s = shift @stack;
    if (!$visited{$s->key}++) {
      push @states, $s;
      push @stack, $_->to for $s->all_arcs;
    }
  }
  my $i;
  for ($i = 0; $i < @states; $i++) {
    $states[$i]->set(id => $i);
  }
  for (sort { $a->id <=> $b->id }
       grep { !$visited{$_->key} } values %{$self->{_states}}) {
    $_->set(id => $i++);
  }
  $self;
}
```

7.2.2 Opérations fondamentales

Cette section présente les opérateurs du calcul d'états finis pondérés. Il y manque seulement la compilation des règles de réécriture, qui ne fonctionnent pas correctement dans le prototype actuel.

7.2.2.1 Union de deux transducteurs

L'union de deux transducteurs se fait selon la construction de Thompson. Lignes 7 et 8, on crée un nouvel état initial pour le transducteur `$net_a` (le premier opérande) et l'on garde une référence vers les états initiaux des deux transducteurs. C'est le nouvel état initial de `$net_a` qui sera l'état initial du transducteur résultat ; pour réaliser l'union, une epsilon-transition est ajoutée du nouvel état initial à l'ancien état initial de `$net_b` (lignes 9 à 12), en utilisant le poids initial de cet état comme poids de la transition. La suite consiste à mettre à jour les propriétés du transducteur résultat en fonction des propriétés des deux opérandes (lignes 14 à 22).

```
 1  sub union {
 2    my ($net_a, $net_b, $copy) = @_;
 3    if (defined $copy && $copy) {
 4      $net_a = $net_a->clone;
 5      $net_b = $net_b->clone;
 6    }
 7    my ($sa, $ss) = $net_a->_new_start_state;
 8    my $sb = $net_b->_unique_start_state;
 9    $net_a->_add_arc(new WFST::Arc(from   => $sa,
10                                   to     => $sb,
11                                   weight => $sb->iweight));
12    (delete $net_a->{_initial}->{$sb->key})->make_non_initial;
13    $net_a->_add_arc($_) for (values %{$net_b->{_arcs}});
14    $net_a->{_final}->{$_->key} = $_ for (values %{$net_b->{_final}});
15    $net_a->{_prop}->{has_cycle} ||= $net_b->{_prop}->{has_cycle};
16    $net_a->{_prop}->{trim} &&= $net_b->{_prop}->{trim};
```

```
17    $net_a->{_prop}->{simple} &&= $net_b->{_prop}->{simple};
18    $net_a->{_prop}->{minimal} = 0;
19    if ($net_a->{_paths} >= 0) {
20      $net_a->{_paths} = $net_b->{_paths} > 0 ?
21      $net_a->{_paths} + $net_b->{_paths} : -1;
22    }
25    $net_a;
26  }
```

7.2.2.2 Concaténation de deux transducteurs

La concaténation suit également le modèle de Thompson. Cette fois-ci, on relie chaque état final de `$net_a` à l'ancien état initial de `$net_b` (lignes 7 à 14).

```
 1  sub concat {
 2    my ($net_a, $net_b, $copy) = @_;
 3    if (defined $copy && $copy) {
 4      $net_a = $net_a->clone;
 5      $net_b = $net_b->clone;
 6    }
 7    my $sb = $net_b->_unique_start_state;
 8    for (values %{$net_a->{_final}}) {
 9      $net_a->_add_arc(new WFST::Arc(weight => $_->fweight * $sb->iweight,
10                                     from   => $_,
11                                     to     => $sb));
12      (delete $net_a->{_final}->{$_->key})->make_non_final;
13    }
14    (delete $net_a->{_initial}->{$sb->key})->make_non_initial;
15    $net_a->_add_arc($_) for (values %{$net_b->{_arcs}});
16    $net_a->{_final}->{$_->key} = $_ for (values %{$net_b->{_final}});
17    $net_a->{_prop}->{has_cycle} ||= $net_b->{_prop}->{has_cycle};
18    $net_a->{_prop}->{trim} &&= $net_b->{_prop}->{trim};
19    $net_a->{_prop}->{simple} &&= $net_b->{_prop}->{simple};
20    $net_a->{_prop}->{minimal} = 0;
21    $net_a->{_paths} *= $net_b->{_paths} if $net_a->{_paths} >= 0;
22    $net_a->{_paths} = 0 if !$net_b->{_paths};
23    $net_a;
24  }
```

7.2.2.3 Itération et optionnalité (A*, A+ et A?)

Ces trois opérations sont déduites de la première (`closure`). Encore une fois, seuls les états inital et finals sont concernés.

```
sub closure {
  my ($self, $copy) = @_;
  my $net = defined $copy && $copy ? $self->clone : $self;
  my $start = $self->_unique_start_state;
  for (grep { $_->final } values %{$net->{_states}}) {
    $net->_add_arc(new WFST::Arc(weight => $_->fweight,
                                 from   => $_,
                                 to     => $start));
  }
```

```
  $start->make_final;
  $net->{_final}->{$start->key} = $start;
  $net->{_prop}->{has_cycle} = 1;
  $net->{_prop}->{minimal} = 0;
  $net->{_paths} = -1;
  $net;
}

sub positive_closure {
  my ($self, $copy) = @_;
  my $net = defined $copy && $copy ? $self->clone : $self;
  my $start = $self->_unique_start_state;
  for (grep { $_->final } values %{$net->{_states}}) {
    $net->_add_arc(new WFST::Arc(weight => $_->fweight,
                                 from   => $_,
                                 to     => $start));
  }
  $net->{_prop}->{has_cycle} = 1;
  $net->{_prop}->{minimal} = 0;
  $net->{_paths} = -1;
  $net;
}

sub optional {
  my ($self, $copy) = @_;
  my $net = defined $copy && $copy ? $self->clone : $self;
  my $start = $self->_unique_start_state;
  my $was_final = $start->final;
  $start->make_final;
  $net->{_final}->{$start->key} = $start;
  ++$net->{_paths} if $net->{_paths} >= 0 &&  !$was_final;
  $net;
}
```

7.2.2.4 Renversement

Le renversement (« miroir ») existe en deux versions selon que l'on maintient un unique état initial (dans le cas de `reverse`) ou non (dans le cas de `_reverse`). En effet, pour renverser un transducteur, il suffit d'échanger états initiaux et finals et d'échanger les extrémités de chacun des arcs. Comme les transducteurs dans `WFST.pm` ont la propriété de toujours avoir un unique état initial mais plusieurs états finals, il faut créer un nouvel état initial.

Cependant, pour l'opération de minimisation (vue dans la section précédente), il est crucial de garder plusieurs états initiaux pour que les poids soient corrects dans le transducteur minimisé.

```
sub reverse {
  my ($self, $copy, $paths) = @_;
  my $net = defined $copy && $copy ? $self->clone : $self;
  my ($start, $ss) = $net->_new_start_state;
  $_->reverse for (values %{$net->{_arcs}});
  for (values %{$self->{_final}}) {
    $net->_add_arc(new WFST::Arc(weight => $_->fweight,
                                 to     => $_,
```

```
                                    from   => $start));
    (delete $net->{_final}->{$_->key})->make_non_final;
  }
  $ss->make_final;
  $net->{_final}->{$ss->key} = $ss;
  $net->{_prop}->{minimal} = 0;
  $net->{_prop}->{deterministic} = 0;
  $net->{_name} = "(~ $net->{_name})" if WFST::debug_tag("alg");
  $net;
}

sub _reverse {
  my ($self, $copy) = @_;
  my $net = defined $copy && $copy ? $self->clone : $self;
  $net->{_initial} = {};
  $net->{_final} = {};
  for (values %{$net->{_states}}) {
    $_->set(iweight => $_->fweight, fweight => $_->iweight);
    $net->{_initial}->{$_->key} = $_ if $_->initial;
    $net->{_final}->{$_->key} = $_ if $_->final;
  }
  $_->reverse for (values %{$net->{_arcs}});
  $net->{_prop}->{deterministic} = 0;
  $net->{_prop}->{minimal} = 0;
  $net->{_name} = "(~~ $net->{_name})" if WFST::debug_tag("alg");
  $net;
}
```

7.2.2.5 Inversion

L'inversion est triviale, puisqu'il s'agit juste d'inverser chacun des arcs du transducteur. Dans le cas d'un automate (transducteur « simple »), c'est encore plus facile : c'est l'opération identité.

```
sub invert {
  my ($self, $copy) = @_;
  if (!$self->{_prop}->{simple}) {
    my $net = defined $copy && $copy ? $self->clone : $self;
    $_->invert for (values %{$net->{_arcs}});
    $net;
  } else {
    $self;
  }
}
```

7.2.2.6 Projection supérieure et inférieure

De la même manière, la projection supérieure ou inférieure est réalisée en modifiant chaque arc pour ne conserver qu'un seul des items du symbole étiquetant l'arc ; en fait chaque arc est supprimé, puis réintroduit dans l'automate résultat, ce qui permet de maintenir facilement un « sigma » correct.

```
sub upper {
  my ($self, $copy) = @_;
  if (!$self->{_prop}->{simple}) {
```

```
    my $net = defined $copy && $copy ? $self->clone : $self;
    $net->{_sigma} = ();
    for (values %{$net->{_arcs}}) {
      $_->from->rm_arc($_);
      $_->upper;
      $_->from->add_arc($_);
      if ($_->epsilon) {
        $net->{_prop}->{has_epsilon}++;
      } else {
        $net->{_sigma}->{$_->label}++;
      }
    }
    $net->{_prop}->{simple} = 1;
    $net;
  } else {
    $self;
  }
}

sub lower {
  my ($self, $copy) = @_;
  if (!$self->{_prop}->{simple}) {
    my $net = defined $copy && $copy ? $self->clone : $self;
    $net->{_sigma} = ();
    for (values %{$net->{_arcs}}) {
      $_->from->rm_arc($_);
      $_->lower;
      $_->from->add_arc($_);
      if ($_->epsilon) {
        $net->{_prop}->{has_epsilon}++;
      } else {
        $net->{_sigma}->{$_->label}++;
      }
    }
    $net->{_prop}->{simple} = 1;
    $net;
  } else {
    $self;
  }
}
```

7.2.2.7 Complément

Le complément d'un automate se calcule à partir de la déterminisation et de la complétion de l'automate source.

```
sub complement {
  my ($self, $copy) = @_;
  if ($self->{_prop}->{simple}) {
    my $net = defined $copy && $copy ? $self->clone : $self;
    $net = $net->determinize->complete;
    for (values %{$net->{_states}}) {
      if ($_->final) {
```

```
        (delete $net->{_final}->{$_->key})->make_non_final;
      } else {
        $_->make_final;
        $net->{_final}->{$_->key} = $_;
      }
    }
    $net->{_name} = _simple_name("!($net->{_name})");
    $net;
  } else {
    confess "complement: operation defined on automata only";
  }
}
```

7.2.2.8 Composition

L'algorithme de composition générique avec filtre (sans epsilon-transitions) est central, car il permet de réaliser aussi bien la composition que l'intersection, le produit cartésien, la soustraction, ainsi que la composition itérative. L'algorithme peut sembler complexe, car il effectue en réalité deux compositions successives (le premier opérande avec le filtre, puis le résultat de cette première composition avec le second opérande), et doit traiter les symboles universels qui sont particuliers.

```
sub _compose {
  my ($net_a, $net_b, $filter) = @_;
  my $net = new WFST::Network;
  my $sa = $net_a->_unique_start_state;
  my $sf = $filter->_unique_start_state;
  my $sb = $net_b->_unique_start_state;
  my $ss = (values %{$net->{_initial}})[0];
  $ss->make_initial($sa->iweight * $sf->iweight * $sb->iweight);
  if ($sa->final && $sf->final && $sb->final) {
    $ss->make_final($sa->fweight * $sf->fweight * $sb->fweight);
    $net->{_final}->{$ss->key} = $ss;
  }
  my $cstate = [$sa, $sf, $sb, $ss];
  my %states = (join("", map{ $_->name } @$cstate[0..2]) => $cstate);
  my @queue = ($cstate);
  while (@queue) {
    $cstate = shift @queue;
    for my $e1 ($cstate->[0]->all_arcs) {
      my @match = ();
      for my $ef ($cstate->[1]->all_arcs) {
        my $match = $e1->match($ef);
        push @match, [$e1, $ef] if $match;
      }
      my @match2 = ();
      for my $ec (@match) {
        my $to_match = $ec->[1]->default ? $ec->[0] : $ec->[1];
        for my $e2 ($cstate->[2]->all_arcs) {
          my $match = $to_match->match($e2);
          push @match2, [$ec->[0], $ec->[1], $e2] if $match;
        }
      }
      for my $ec (@match2) {
```

```
        my $ncstate = $states{join "", map { $_->to->name } @$ec};
        if (!defined $ncstate) {
          my $state = new WFST::State;
           $state->make_final($ec->[0]->to->fweight * $ec->[1]->to->fweight *
                              $ec->[2]->to->fweight)
             if $ec->[0]->to->final && $ec->[1]->to->final &&
               $ec->[2]->to->final;
          $ncstate = [(map { $_->to } @$ec), $state];
          push @queue, $ncstate;
          $states{join("", map{ $_->to->name } @$ec)} = $ncstate;
          $net->{_state}->{$state->key} = $state;
          $net->{_final}->{$state->key} = $state if $state->final;
        }
        my $usym = ref $ec->[0]->label->upper eq "WFST::Item::Upper" ||
                    ref $ec->[0]->label->upper eq "WFST::Item::X" ?
                      "" : $ec->[0]->label->upper;
        my $lsym = ref $ec->[2]->label->lower eq "WFST::Item::Lower" ||
                    ref $ec->[2]->label->lower eq "WFST::Item::X" ?
                      "" : $ec->[2]->label->lower;
        my $arc = new WFST::Arc
                   (from   => $cstate->[3],
                    to     => $ncstate->[3],
                    label  => new WFST::Symbol($usym, $lsym),
                    weight => $ec->[0]->weight * $ec->[1]->weight *
                                $ec->[2]->weight);
        $net->_add_arc($arc);
      }
    }
  }
  $net->{_prop}->{trim} = 0;
  $net->{_prop}->{minimal} = 0;
  $net->rm_epsilon->trim->renumber_states->_count_paths;
}
```

La composition (`compose`) et le produit cartésien (`cross_product`) sont deux applications de cet algorithme. Dans les deux cas, les opérandes doivent être prétraités pour être compatibles avec le filtre. Les deux transducteurs filtres sont créés la première fois que l'on fait appel à ces fonctions, puis sont mémorisés (variables de classe) pour les utilisations ultérieures.

```
sub compose {
  my ($net_a, $net_b) = @_;
  $net_a = $net_a->clone->rm_epsilon->trim;
  my $label = new WFST::Symbol(new WFST::Item::Upper);
  for (values %{$net_a->{_states}}) {
    $net_a->_add_arc(new WFST::Arc(from   => $_,
                                   to     => $_,
                                   label  => $label));
    $_->set(label => $_->label->lower_epsilon) for $_->all_arcs;
  }
  $net_b = $net_b->clone->rm_epsilon->trim;
  $label = new WFST::Symbol(new WFST::Item::Lower);
  for (values %{$net_b->{_states}}) {
    $net_b->_add_arc(new WFST::Arc(from   => $_,
                                   to     => $_,
```

```
                                    label  => $label));
    $_->set(label => $_->label->upper_epsilon) for $_->all_arcs;
  }
  if (!defined $FILTER) {
    _make_filter();
  }
  my $net = _compose($net_a, $net_b, $FILTER);
  $net;
}

sub cross_product {
  my ($net_a, $net_b) = @_;
  if ($net_a->{_prop}->{simple} && $net_b->{_prop}->{simple}) {
    my $epsilon = new WFST::Symbol(new WFST::Item::X);
    $net_a = $net_a->clone->rm_epsilon->trim;
    for (values %{$net_a->{_states}}) {
      $net_a->_add_arc(new WFST::Arc(from   => $_,
                                     to     => $_,
                                     label  => $epsilon)) if $_->final;
      $_->set(label => $_->label->add_lower_epsilon) for $_->all_arcs;
    }
    $net_b = $net_b->clone->rm_epsilon->trim;
    for (values %{$net_b->{_states}}) {
      $net_b->_add_arc(new WFST::Arc(from   => $_,
                                     to     => $_,
                                     label  => $epsilon)) if $_->final;
      $_->set(label => $_->label->add_upper_epsilon) for $_->all_arcs;
    }
    WFST::debug(xpr => split /\n/, $net_b);
    if (!defined $X_FILTER) {
      _make_x_filter();
    }
    my $net = _compose($net_a, $net_b, $X_FILTER);
    $net;
  } else {
    cluck "cross product: operation defined on automata only";
  }
}
```

La différence (`subtract`) est en réalité une simple intersection entre son premier opérande et le complément du second.

```
sub subtract {
  my ($net_a, $net_b, $copy) = @_;
  if ($net_a->{_prop}->{simple} && $net_b->{_prop}->{simple}) {
    if (defined $copy && $copy) {
      $net_a = $net_a->clone;
      $net_b = $net_b->clone;
    }
    my $net = $net_a->intersect($net_b->complement);
    $net->{_name} = $name;
    $net;
  } else {
    confess "complement: operation defined on automata only";
```

```
  }
}
```

7.2.2.9 Composition itérative

Cette composition itérative (`composep`) s'écrit en fonction de la composition classique. On distingue deux cas, selon que le transducteur est simple ou complexe : dans le cas d'un transducteur complexe, on effectue une projection inférieure du transducteur résultat après chaque application.

```
sub composep {
  my ($net_a, $net_b) = @_;
  my $prev = $net_a;
  my $curr;
  if ($net_a->{_prop}->{simple}) {
    while (defined $prev) {
      $curr = $prev->compose($net_b)->lower;
      if ($curr->equivalent($prev)) {
        $prev = undef;
      } else {
        $prev = $curr;
      }
    }
  } else {
    while (defined $prev) {
      $curr = $prev->compose($net_b);
      if ($curr->equivalent($prev)) {
        $prev = undef;
      } else {
        $prev = $curr;
      }
    }
  }
  $curr;
}
```

7.2.2.10 Équivalence de deux transducteurs

L'algorithme de composition itérative ci-dessus doit pouvoir comparer deux transducteurs pour vérifier si un point fixe est atteint. On propose ici la fonction `equivalent` qui repose sur une structure Union-Find (que l'on ne détaille pas ici, mais qui est réalisée par un module Perl de quelques dizaines de lignes seulement).

```
sub equivalent {
  my ($net_a, $net_b) = map { $_->rm_epsilon(1) } @_;
  my $uf = new UnionFind::PathCompression(keys %{$net_a->{_states}},
                                          keys %{$net_b->{_states}});
  my $sa = [$net_a->_unique_start_state, $WFST::WREF->one];
  my $sb = [$net_b->_unique_start_state, $WFST::WREF->one];
  _merge($uf, $sa, $sb);
}
```

Évidemment, l'essentiel du travail est effectué par la fonction de fusion de deux états (`_merge`). On retourne une valeur vraie si les deux états peuvent être fusionnés, et fausse sinon. Difficulté supplémentaire, il faut tenir compte des poids résiduels : aussi, les paramètres sont en réalité des

couples (état, poids résiduel), ce qui est proche de la situation qui se présente pour la désambiguïsation.

```
sub _merge {
  my ($uf, $u, $v) = @_;
  if (!$uf->find($u->[0]->key, $v->[0]->key)) {
    $uf->union($u->[0]->key, $v->[0]->key);
    return 0 if ($u->[0]->initial && !$v->[0]->initial) ||
      ($u->[0]->final && !$v->[0]->final) ||
        (!$u->[0]->initial && !$u->[0]->final &&
          ($v->[0]->initial || $v->[0]->final));
    if ($u->[0]->initial) {
      return 0 if $u->[0]->iweight * $u->[1] != $v->[0]->iweight * $v->[1];
    }
    if ($u->[0]->final) {
      return 0 if $u->[0]->fweight * $u->[1] != $v->[0]->fweight * $v->[1];
    }
    my %uarcs = %{$u->[0]->{_arcs}};
    my %varcs = %{$v->[0]->{_arcs}};
    for (keys %uarcs) {
      return 0 if !exists $varcs{$_};
      for my $uarc (values %{$uarcs{$_}}) {
        for my $varc (values %{$varcs{$_}}) {
          my $wu = $uarc->weight * $u->[1];
          my $wv = $varc->weight * $v->[1];
          my $wm = $wu < $wv ? $wu : $wv;
          return 0 if !_merge($uf, [$uarc->to, $wu / $wm],
                                   [$varc->to, $wv / $wm]);
        }
      }
      delete $varcs{$_};
    }
    return 0 if keys %varcs > 0;
  }
  1;
}
```

7.2.3 Application

7.2.3.1 Liste exhaustive de tous les chemins

La liste exhaustive de tous les chemins est un parcours (ici récursif) des arcs du transducteur pour générer chaque chemin avec son poids. Le traitement des circuits est simple et ressemble à celui de XFST ; un état ne doit être visité qu'une seule fois.

```
sub all_paths {
  my $self = shift;
  map { $self->_all_paths($_, {}, $_->iweight) } values %{$self->{_initial}};
}

sub _all_paths {
  my ($self, $state, $seen, $iweight) = @_;
  $iweight = defined $iweight ? $iweight : $WFST::WREF->one;
```

```
  my @paths = ();
  my %seen = %$seen;
  return () if $seen{$state->key}++;
  push @paths, ["", $state->fweight * $iweight] if $state->final;
  for my $arc ($state->all_arcs) {
    push @paths, map { [($arc->epsilon ? "" : $arc->label) . $_->[0],
      $arc->weight * $_->[1] * $iweight] }
      $self->_all_paths($arc->to, \%seen);
  }
  @paths;
}
```

7.2.3.2 Application d'un transducteur à une chaîne

Il existe deux directions d'application d'un transducteur à une chaîne qui sont parfaitement symétriques. Dans un transducteur normal, tous les chemins reconnus sont retournés, alors que dans un transducteur pondérés, on produit soit tous les chemins avec leur poids, soit un sous-ensemble où les chemins sont optimaux. Dans cette implémentation, tous les chemins sont simplement retournés.

On montre ici la fonction « lookup », qui produit tous les chemins du langage supérieur correspondant à une chaîne du langage inférieur. La fonction « lookdown » est analogue.

```
sub all_paths_u {
  my ($self, $str) = @_;
  map { $self->_all_paths_u($_, $str, {}, $_->iweight) }
    values %{$self->{_initial}};
}

sub _all_paths_u {
  my ($self, $state, $str, $seen, $iweight) = @_;
  if ($seen->{$state->key . $str}++) {
    warn "Epsilon loop detected in network.\n";
    return ();
  }
  $iweight = defined $iweight ? $iweight : $WFST::WREF->one;
  my @paths = ();
  push @paths, ["", $state->fweight * $iweight] if $state->final && $str eq "";
  my $match = 0;
  for my $arc ($state->all_arcs) {
    if (defined(my $match = $arc->label->lower->match_str($str))) {
      my $out = $arc->label->upper->epsilon ? "" : $arc->label->upper;
      my @p = map { [$out . $_->[0], $arc->weight * $_->[1] * $iweight] }
      $self->_all_paths_u($arc->to, substr($str, length $match), $seen);
      ++$match if @p;
      push @paths, @p;
    }
  }
  if ((!$self->{_prop}->{deterministic} || !$match) && $str =~ /^./) {
    my $match = $&;
    my $str = $';
    for my $arc (values %{$state->default}) {
      my $out = $arc->label->upper->epsilon ? "" :
        $arc->label->default ? $match : $arc->label->upper;
```

```
        push @paths, map { [$out . $_->[0], $arc->weight * $_->[1] * $iweight] }
          $self->_all_paths_u($arc->to, $str, $seen);
      }
    }
    @paths;
}
```

L'application dans l'un des deux sens d'un automate à une chaîne a pour résultat cette même chaîne avec son poids dans l'automate, ou une chaîne vide si la chaîne en entrée n'est pas reconnue.

7.2.3.3 Génération aléatoire de chemins

Un transducteur « utile » pour une véritable application linguistique contient un grand nombre de chemins, et très souvent des circuits. Pour avoir une idée des mots du langage reconnu par un transducteur, on propose une fonction générant aléatoirement des chemins.

L'algorithme ressemble à l'algorithme d'application ; il n'est cependant plus guidé par une chaîne en entrée, mais par le hasard. On suppose que le transducteur contient au moins un chemin et qu'il est connecté, ce qui évite d'avoir faire des retours arrière. À chaque état, on choisit une transition au hasard, ou, si l'état est final, on peut également choisir de s'arrêter (comme s'il existait une epsilon-transition vers cet état). Chaque choix possible est pondéré par le nombre de fois que l'état destination a été visité ; il est de moins en moins probable de visiter à nouveau un état qui l'a déjà été. Cela permet de suivre « raisonnablement » un chemin dans un circuit. Un chemin a été découvert une fois qu'un état final a été choisi.

```
sub random_path {
  my $self = shift;
  my $path = undef;
  my $weight = undef;
  if ($self->{_paths} && $self->{_prop}->{trim}) {
    my @start = values %{$self->{_initial}};
    my $start = $start[rand@start];
    $path = "";
    $weight = $start->iweight;
    my %seen = map { $_ => 0 } keys %{$self->{_states}};
    my $state = $start;
    while (defined $state) {
      ++$seen{$state->key};
      my $max_seen = 0;
      for ($state->all_arcs) {
        $max_seen = $seen{$_->to->key} if $seen{$_->to->key} > $max_seen;
      }
      my @choice = ();
      if ($state->final) {
        $max_seen = $seen{$state->key} if $seen{$state->key} > $max_seen;
        push @choice, ($state) x ($max_seen - $seen{$state->key} + 1);
      }
      push @choice, ($_) x ($max_seen - $seen{$_->to->key} + 1)
        for $state->all_arcs;
      my $choice = $choice[rand@choice];
      if (ref $choice eq "WFST::State") {
        $choice->name . ", now we're done!");
        $weight *= $state->fweight;
        $state = undef;
```

```
        } else {
          $path .= $choice->label;
          $weight *= $choice->weight;
          $state = $choice->to;
        }
      }
    }
    defined $path ? [$path, $weight] : undef;
  }
```

7.3 L'interface wfst.pl

L'interface `wfst.pl` est une application permettant la manipulation directe de transducteurs d'états finis pondérés. Ceux-ci sont décrits par des expressions régulières pondérées, stockés dans des variables, sauvés dans des fichiers, etc.

7.3.1 Utilisation du module WFST.pm

L'utilisation du module `WFST.pm` dans un programme en Perl nécessite d'inclure le module en définissant le type de poids et d'items que l'on va utiliser.

`wfst.pl` permet de manipuler des transducteurs définis sur n'importe quelle structure de poids d'une classe héritant de `WFST::Weight`. Par contre, seuls des items de la classe `WFST::Item` sont disponibles.

```
1  use WFST;
2  my $weight = "tropical";
3  my $debug  = 0;
4  GetOptions("weights=s" => \$weight, debug => \$debug);
5  $weight = "WFST::Weight::" . ucfirst $weight;
6  WFST::set_weight($weight);
7  WFST::set_debug($debug);
```

Par défaut, les poids sont du type « tropical » (ligne 2). L'appel à la fonction `GetOptions` ligne 4 permet de choisir le type de poids en utilisant l'option `--weight` sur la ligne de commande ; par exemple

```
wfst.pl --weight boolean
```

Le type de poids est construit en préfixant le nom passé en paramètre par `WFST::Weight::` et en ajoutant une majuscule (`boolean` devient donc `WFST::Weight::Boolean`; ligne 4). Le module `WFST.pm` est inclu ligne 1, et ses paramètres sont modifiés par un appel aux fonctions `set_weight` et `set_debug`, en relation avec l'opération `--debug`.

7.3.2 Une grammaire d'états finis pondérés

La grammaire des expressions régulières a été vue dans le chapitre 5 ; c'est évidemment le rôle du module `WFST.pm` d'analyser une expression régulière et de compiler le transducteur d'états finis pondérés correspondant. Cette fonctionnalité est fournie par un objet de la classe `WFST::Regex`, qui est fondé sur le module Perl `Parse::RecDescent`.

Ce module, très populaire, est un outil similaire à `yacc` qui réalise un analyseur syntaxique descendant à partir d'une grammaire. Comme dans `yacc`, une grammaire est décrite par des règles de

production auxquelles sont associées des actions écrites en Perl, qui permettent par exemple de générer un arbre syntaxique abstrait.

La grammaire complète pour une expression régulière Sumo est donnée ci-dessous, dans le format utilisé par `Parse::RecDescent` :

```
rx       : regex /\Z/
regex    : intersect(s /\%/)       {WFST::Regex::tree(compose=>$item[1],1)}
intersect: subtract(s /\&/)        {WFST::Regex::tree(intersect=>$item[1],1)}
subtract : union(s /\-/)           {WFST::Regex::tree(subtract=>$item[1],1)}
union    : concat(s /\|/)          {WFST::Regex::tree(union=>$item[1],1)}
concat   : closure(s)              {WFST::Regex::tree(concat=>$item[1],1)}
closure  : unary "*"               {WFST::Regex::tree(closure=>[$item[1]])}
         | unary "+"               {WFST::Regex::tree(pclosure=>[$item[1]])}
         | unary "?"               {WFST::Regex::tree(optional=>[$item[1]])}
         | unary
unary    : "^" unit                {WFST::Regex::tree(invert=>[$item[2]])}
         | "~" unit                {WFST::Regex::tree(reverse=>[$item[2]])}
         | "!" unit                {WFST::Regex::tree(complmnt=>[$item[2]])}
         | unit ",u"               {WFST::Regex::tree(upper=>[$item[1]])}
         | unit ",l"               {WFST::Regex::tree(lower=>[$item[1]])}
         | unit
unit     : "(" regex ")"           {$item[2]}
         | symbol                  {WFST::Regex::tree(item=>$item[1])}
         | net                     {WFST::Regex::tree(net=>$item[1])}
net      : "stub"
symbol   : sym "/" weight          {new WFST::Arc(label  => $item[1],
                                    weight => $item[3])}
         | sym                     {new WFST::Arc(label  => $item[1])}
sym      : ":" <skip:""> item      {new WFST::Symbol("", $item[3])}
         | item <skip:""> ":" item {new WFST::Symbol($item[1], $item[4])}
         | item <skip:""> ":"      {new WFST::Symbol($item[1], "")}
         | item                    {new WFST::Symbol($item[1])}
item     : "stub"
weight   : "stub"
```

`Parse::RecDescent` propose quelques raccourcis intéressants. Premièrement, les terminaux dans la grammaire sont aussi bien décrits par des chaînes de caractères que par des expressions régulières Perl. Deuxièmement, une règle comme

```
regex : intersect(s /%/)
```

permet de noter une liste d'un ou plusieurs `intersect` séparés par le symbole `%` (l'ordre des premières règles permet de définir l'ordre de priorité des différents opérateurs). On remarque enfin que les trois règles `net`, `item` et `weight` sont en fait des « moignons » de règles (*stub* en anglais), qui sont appelés à être remplacés (voir plus bas).

La partie droite de chaque règle est l'action exécutée lorsque la règle s'applique. Ici, il s'agit simplement de créer un arbre abstrait correspondant à l'expression en passant par la fonction `tree` suivante :

```
sub tree {
  my ($type, $children, $skip) = @_;
  if (ref $children eq "ARRAY") {
    if (@$children == 1 && defined $skip && $skip) {
      $children->[0];
    } else {
      [$type, @$children];
```

```
    }
  } else {
    [$type, $children];
  }
}
```

Les nœuds de l'arbre sont des opérateurs (premier paramètre de la fonction) et les feuilles de l'arbre sont des arcs, construits au fur et à mesure de l'analyse des symboles et des poids de l'expression. La fonction `tree` permet d'obtenir un arbre compact tout en maintenant une description simple de la grammaire.

On a vu dans la section 7.1 que les poids aussi bien que les items pouvaient être redéfinis ; et leur syntaxe dans une expression régulière dépend de leur nature exacte. C'est pourquoi les classes `WFST::Weight` et `WFST::Item` définissent leur propre règle de grammaire `weight` et `item`, qui est utilisée par la suite pour remplacer le moignon laissé dans la grammaire principale. Ainsi, si l'on utilise des poids du demi-anneau tropical et les items par défaut (les seuls disponibles actuellement), ce qui est la situation vue dans le chapitre 5, on a les règles de grammaire suivantes :

```
item     : "0"                      {new WFST::Item::Epsilon}
         | "."                      {new WFST::Item::Default}
         | /([1-9a-zA-Z\x80-\xff]|\\\\u[\da-f]{1,4}|\\\\.)+/i
                                    {$item[1] =~ s/\\\\u([\da-f]{1,4})/
                                     WFST::Regex::ucs2utf8(hex $1)/gexi;
                                     $item[1] =~ s/\\\\(.)/$1/g;
                                     new WFST::Item($item[1])}
         | /\"((?:[^\\\\\"]|(?:\\\\.))+)\"/
                                    {(my$str=$1) =~ s/\\\\(.)/$1/g;
                                     new WFST::Item($str)}

weight   : /\-?\d*\.?\d+/           {$WFST::wref->new(0+$item[1])}
```

7.3.3 L'interface

Il reste un dernier « moignon » dans la grammaire d'expressions régulières : la règle `net`. Celle-ci indique la syntaxe permettant de faire référence à un transducteur précédemment défini. Dans `wfst.pl`, une expression régulière peut être affectée à une variable (de la forme `$var`) ; cette variable fait ensuite référence au transducteur compilé pour cette expression régulière. La partie de la grammaire correspondante est alors :

```
net: var              { ::fetch_var($item[1]) }
var: '$' <skip:""> id { $item[3] }
id : /[^\W\d][\w]*/
```

L'utilisation première de l'interface est la construction de transducteurs en les décrivant par des expressions régulières pondérées en suivant la syntaxe du chapitre 5. Les transducteurs ainsi créés peuvent être enregistrés dans un fichier pour être lus et ultérieurement par l'interface ou par un programme d'analyse qui s'en servira comme donnée. Une autre utilisation est l'application de ces transducteurs à d'autres transducteurs ou à des chaînes de caractères pour l'analyse ou la génération.

L'interface est interactive et se présente comme un shell de commandes classique. On peut ainsi expérimenter, corriger des erreurs, faire des tests, etc. comme dans l'exemple de session suivant :

```
[wfst] $romain_1 = regex i/1 | i i 0/2 | i i i 0/3 | i v 0/4 | v/5
"net0", 4 états, 5 arcs, 5 chemins.
[wfst] $romain_2 = read "romain2.wfst"
"net1", 6 états, 7 arcs, 5 chemins.
[wfst] $romain = ($romain_1 | $romain_2) ?
```

```
"net2", 6 états, 9 arcs, 10 chemins.
[wfst] name $romain "Chiffres romains"
"Chiffres romains", 6 états, 9 arcs, 10 chemins.
[wfst] save $romain "romain.wfst"
"Chiffres romains", 6 états, 9 arcs, 10 chemins.
[wfst] lookup "iv" in $romain
iv/4
[wfst] lookup $romain
look up "Chiffres romains" > viii
viii/8
look up "Chiffres romains" >
/0
look up "Chiffres romains" > xix
look up "Chiffres romains" > ^D
[wfst] quit
```

Chaque commande est précédée d'une invite, qui est par défaut `[wfst]`. La première commande crée un nouveau transducteur pour les chiffres 1 à 5 ; le résultat produit est un transducteur nommé `net0` ayant 4 états, 5 arcs et 5 chemins. On n'affiche que la « signature » du transducteur résultat, qui permet de se rendre compte rapidement si le résultat obtenu semble conforme à ce que l'on attend. La suite de la construction se fait en lisant un transducteur préalablement enregistré, contenant les nombres 6 à 10, puis on construit l'automate final en faisant l'union des deux, on donne un nom à cet automate et on l'enregistre dans un fichier.

La suite consiste à tester cet automate en le consultant (`lookup`). On peut consulter une chaîne à la fois, ou entrer dans une boucle de reconnaissance. L'invite change alors pour indiquer un mode d'interaction différent. Pour chaque chaîne de caractères, les chemins trouvés sont affichés suivis de leur poids.

`wfst.pl` accepte un argument optionnel `--script`, qui lit un fichier et non l'entrée standard pour ses commandes. On peut ainsi automatiser certaines tâches comme la construction d'un transducteur. Une fois le script entièrement lu, le contrôle repasse à l'utilisateur, à moins que le script comporte un commande `quit` qui termine l'application.

Les commandes de l'interface sont résumées dans la table 7.1.

On revient brièvement sur les commandes `regex` et `_regex`. La commande `regex` compile un transducteur et le minimise, le nettoie et renumérote les états. Cependant, tous les transducteurs pondérés ne sont pas minimisables (car non déterminisables), et pour l'instant, cette propriété n'est pas testée *a priori* : la création du transducteur peut ne pas terminer. Une commande qui se contente de compiler le transducteur est nécessaire, c'est la commande `_regex`.

Une fois le transducteur ainsi compilé, on peut étudier l'effet des différentes opérations sur le transducteur, normalement effectuées automatiquement : suppression des epsilon-transitions, renumérotation des états, suppression des états et transitions non connexes, etc.

Conclusion

Au total, la programmation de cette bibliothèque en Perl est divisée en une dizaine de fichiers pour un total de 3600 lignes, y compris l'interface `wfst.pl`. La compilation des transducteurs peut être assez lente à partir d'une certaine taille (de l'ordre de plusieurs secondes pour quelques milliers de lignes), mais l'exécution est relativement rapide.

Un inconvénient de l'utilisation de Perl est que la taille mémoire utilisée pour stocker un transducteur sous forme compilée est environ dix fois plus importante que pour l'implémentation d'AT&T

Commande	Effets
`vars`	liste toutes les variables définies
`delete`	supprime les variables données en argument
`weights`	rappelle le type de poids des transducteurs
`=`	affecte une valeur à une variable (chaîne de caractères ou transducteur)
`load`	charge un transducteur depuis un fichier
`import wfsa`	importe un automate au format AT&T
`import wfst`	importe un transducteur au format AT&T
`export`	exporte un transducteur au format AT&T (poids « tropicaux »)
`save`	enregistre un transducteur
`draw`	enregistre un transducteur au format `dot`
`dump`	affiche le transducteur au format textuel vu dans la section 7.1.3
`regex`	compile une expression régulière et renvoie un transducteur minimal
`_regex`	compile une expression régulière et renvoir un transducteur « brut »
`trim`	nettoie un transducteur pour qu'il soit connecté
`determinize`	déterminise un transducteur
`minimize`	minimise un transducteur
`reverse`	renverse un transducteur (un seul état initial)
`preverse`	renverse un transducteur (plusieurs états initiaux)
`epsilon`	supprime les epsilon-transitions dans un transducteur
`renum`	renumérote les états dans un transducteur
`name`	donne un nom à un transducteur
`lookup`	applique le transducteur de bas en haut (analyse)
`lookdown`	applique le transducteur de haut en bas (génération)
`random`	génère des chemins aléatoires
`quit`	quitte l'interface

Tab. 7.1 – Les commandes de wfst.pl

du fait de la représentation des variables en Perl. Un bon exemple est le format de sauvegarde des transducteurs compilés. En utilisant le module de sérialisation `Storable` (disponible sur CPAN, et en standard depuis la version 5.8 du langage), on peut sauver un objet de la classe `WFST::Network` dans un fichier binaire. La taille d'un tel fichier est plus de dix fois plus importante que le même fichier sauvé par XFST ou FSM. Ce n'est pas rédhibitoire pour un prototype de test mais on perd tout de même l'avantage de la compacité des ressources linguistiques exprimées sous cette forme.

Par contre, un avantage est la facilité de développer une interface dans le même langage et d'ajouter dynamiquement de nouvelles classes d'objets et de poids.

Chapitre 8

Expérimentation avec les transducteurs d'états finis pondérés

Introduction

Pour illustrer l'utilisation du prototype vu dans le chapitre précédent, on donne ici deux exemples de reconnaissance de langages : un langage de parenthèses et un langage d'expressions régulières.

8.1 Expressions parenthésées

Note : dans ce premier exemple, on a omis les poids des transitions et des états finals dans les figures.

8.1.1 Le langage à reconnaître

Dans cet exemple formel, on étudie un langage qui contient trois symboles : la parenthèse ouvrante `(`, la parenthèse fermante `)`et `S` (qui représente un symbole autre qu'une parenthèse ouvrante ou fermant). Il est défini formellement par

$L = \{w \in \{(,),S\}^* | w$ est correctement parenthésé $\}$

8.1.2 Principe

On a vu dans la section 1.1.3.3 comment reconnaître des expressions parenthésées à l'aide de substitutions fondées sur les expressions régulières. Une substitution effaçait chaque sous-expression parenthésée ne contenant elle-même pas de parenthèse, et répétait l'opération jusqu'à obtenir une chaîne vide.

En lieu et place du `while` alors employé pour répéter la substitution, on utilise l'opérateur de composition itératif `%+` de WFST (noté T^∞ par Roche), remplaçant successivement chaque sous-expression parenthésée par `S`. Au final, l'expression peut être réduite au langage S^* .

8.1.3 Source

La réécriture d'une expression parenthésée élémentaire, qui ne contient pas de sous-expression parenthésée (donc uniquement des `S`) en un unique symbole `S`est exprimée par la règle :

```
\( S* \) -> S
```

La compilation de cette règle donne le transducteur de la figure 8.1. L'application de cette règle se fait par l'opérateur `%+` à un automate représentant une expression dont on veut vérifier si elle appartient à L . Le résultat final de l'application itérative, R doit être un mot du langage S^* . On teste l'appartenance du résultat à ce langage en faisant l'intersection de R avec A_S , automate reconnaissant le langage S^* . Cette intersection doit être non-nulle pour que la chaîne soit acceptée.

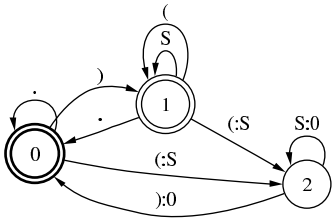

FIG. 8.1 – Un reconnaisseur d'expressions parenthésées

8.1.4 Exemple

On se propose de tester l'appartenance de deux chaînes à L avec `wfst.pl`.

```
[wfst] $paren = regex \( S* \) -> S
"net0", 3 états, 9 arcs, circulaire.
[wfst] $expr1 = regex \( S \) S \( S \( S \) \)
"net1", 11 états, 10 arcs, 1 chemin.
[wfst] $result1 = regex ($expr1 %+ $paren),1 & S*
"net2", 4 états, 3 arcs, 1 chemin.
[wfst] draw $result1 in "result1.dot"
```

La première expression est `(S)S(S(S))`. On compile le transducteur de règle `$paren` utilisé pour les tests, puis celui de la première expression (`$expr1`). `$result1` contient le résultat de l'analyse ; pour visualiser ce résultat, on le dessine dans le fichier `"result1.dot"`, qui produit directement la figure 8.2.

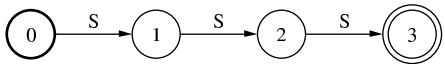

FIG. 8.2 – Résultat de la reconnaissance de l'expression `(S)S(S(S))`

Bien que les étapes intermédiaires ne soient pas visibles, la figure fig:etapes1 représentent la forme du résultat après chaque itération. Les deux dernières formes sont identiques, ce qui cause l'arrêt de l'itération, et donne le résultat final conforme à la figure 8.2.

Un second essai, cette fois-ci avec la forme `(S)S(S(S)` qui n'est pas correctement parenthésée (il manque la dernière parenthèse fermante) donne un résultat vide. Les étapes de la composition sont illustrés par la figure 8.4 ; le dernier automate a bien une intersection vide avec `S*`.

```
[wfst] $expr2 = regex \( S \) S \( S \( S \)
"net3", 10 états, 9 arcs, 1 chemin.
[wfst] $result2 = regex ($expr2 %+ $paren),1 & S*
"net4", 1 état, 0 arc, 0 chemin.
[wfst] quit
```

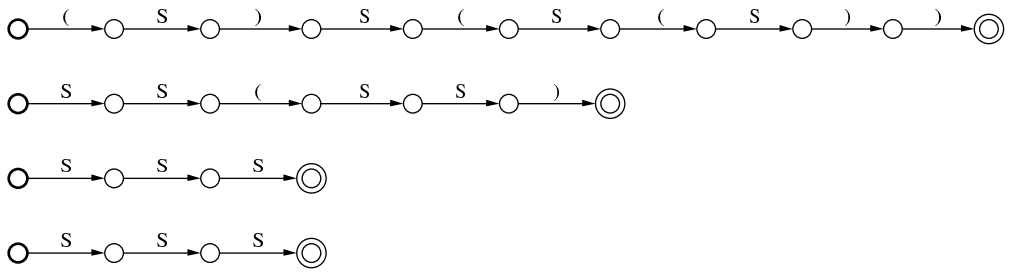

FIG. 8.3 – Étapes de la reconnaissance de l'expression `(S)S(S(S))`

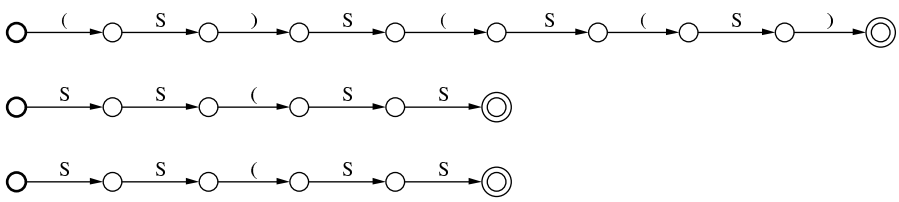

FIG. 8.4 – Étapes de la reconnaissance de l'expression `(S)S(S(S)`

8.2 Expressions régulières

On reprend ici un exemple exposé dans [Mohri, 2001] en utilisant d'ailleurs la fonctionnalité d'importation des transducteurs de FSM offerte par Sumo. Cet exemple met à profit la pondération

8.2.1 Le langage à reconnaître

Un langage d'expressions régulières sur l'alphabet $\{a, b\}$, où l'on dispose des opérateurs de concaténation, d'union `"+"` et d'itération `"*"`, ainsi que de parenthèses pour grouper des sous-expressions. La chaîne vide est notée `e` dans ce langage ; pour simplifier l'exemple, le reconnaisseur ne permet pas d'exprimer le langage vide $\emptyset$.

8.2.2 Principe

Le langage des expressions régulières est hors-contexte, mais peut pourtant être reconnu par un automate d'états finis pondéré. Le critère permettant d'accepter ou de rejeter une chaîne est cependant modifié par rapport au critère habituel : non seulement la chaîne doit correspondre à un chemin dans l'automate, mais également la somme des poids de tous les chemins pour cette chaîne dans l'automate doit appartenir à un sous-ensemble de l'ensemble de poids. Généralement, ce sous-ensemble $\mathbb{J} \in \mathbb{K}$ est un singleton, ce qui permet de tester l'appartenance des chemins trouvés à cet ensemble en temps constant.

Dans ce cas particulier, $\mathbb{K} = (\mathbb{R} \cup \{\infty\}, min, +, \infty, 0)$ (soit le désormais habituel demi-anneau tropical) et $\mathbb{J} = \{0\}$. Une expression est donc acceptée si et seulement il existe au moins un chemin correspondant à cette expression, et si le chemin de poids minimum a un poids de 0.

Dans l'automate de la figure 8.5, on voit que les poids sont utilisés pour « compter » les parenthèses, afin de vérifier que l'expression est correctement parenthésée. Ici, on ne se base pas sur l'itération : le test d'appartenance au langage se fait par une simple application de l'automate à une chaîne donnée.

C'est par contre une méthode moins générale que celle de l'itération vue plus haut, car tous les langages hors-contexte ne peuvent pas forcément être reconnus de la sorte (voir [Roche, 1996] pour des exemples plus complexes).

8.2.3 Source

Dans ce cas particulier, on ne dispose pas d'une expression régulière mais du reconnaisseur de la figure 8.5, dont la description au format FSM (état source, état destination, symbole, poids) est la suivante :

```
0 1 1 0
0 1 2 0
0 1 3 0
0 0 4 1
0 2 5 -1
1 1 1 0
1 1 2 0
1 1 3 0
1 0 4 1
1 2 4 -1
1 1 5 -1
1 3 5 1
1 1 6 0
1 1 7 0
2 3 1 0
2 3 2 0
2 3 3 0
2 2 4 -1
3 3 1 0
3 3 2 0
3 3 3 0
3 2 4 -1
3 3 5 1
3 2 6 0
1 0
3 0
```

Avec comme fichier de symboles :

```
a 1
b 2
e 3
( 4
) 5
+ 6
* 7
```

8.2.4 Exemple

Le premier exemple est une expression qui est reconnue : `(abb*)*(e+(a*b))` correspond bien à un unique chemin de poids 0 (notons que l'automate n'est pas déterministe).

```
[wfst] $rx = import wfsa "regex.fsa" "regex.sym"
"net0", 4 états, 25 arcs, circulaire.
```

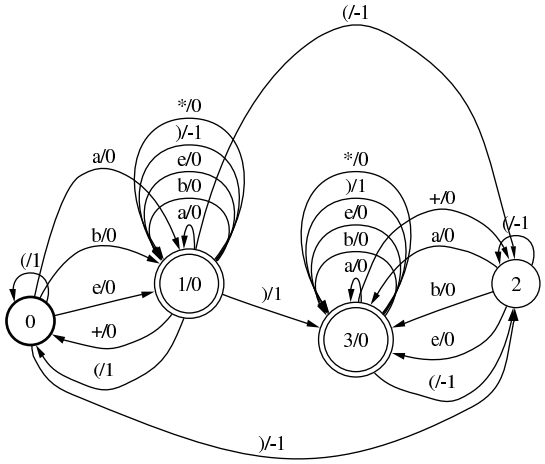

FIG. 8.5 – Un reconnaisseur d'expressions régulières

```
[wfst] lookup $rx
look up "net0" > (abb*)*(e+(a*b))
(abb*)*(e+(a*b))/0
```

Une expression peut être rejetée pour deux raisons : la première est simplement qu'il n'y a aucun chemin dans l'automate ; et la deuxième est que le ou les chemins trouvés n'ont pas un poids nul. Dans le premier cas, on a par exemple une expression commençant par une parenthèse fermante qui est immédiatement rejetée.

Dans le deuxième cas, si l'on prend l'expression `(abb**` à laquelle manque une parenthèse fermante, on trouve deux chemins dans l'automate : le premier est de poids 1, le second de poids -1. L'expression est bien rejetée.

```
look up "net0" > )(ab
look up "net0" > (abb**
(abb**/-1
(abb**/1
look up "net0" > ^D
[wfst] quit
```

Conclusion

On a donné deux exemples d'utilisation d'automates d'états finis pondérés à l'aide de `wfst.pl` illustrant l'apport de l'itération et de la pondération à l'aide de langages formels simples. L'exemple de la section 8.1 est particulièrement instructif car il introduit le principe que l'on utilisera pour créer des structures arborescentes à partir d'expressions parenthésées, comme un texte en XML.

Chapitre 9

Applications potentielles et perspectives

Introduction

L'introduction de la première partie a présenté différentes tâches qui constituent ce que l'on a appelé l'analyse présyntaxique. Bien que ce niveau d'analyse n'atteigne pas la sophistication des niveaux supérieurs, de puissantes applications linguistiques sont déjà réalisables à ce stade d'analyse.

Sumo a pour ambition de permettre de réaliser de façon aisée et uniforme de l'analyse présyntaxique. En définissant la segmentation comme un problème très large afin de pouvoir traiter des documents de taille conséquente dans n'importe quelle langue, on a rendu le champ d'application potentiel de Sumo très large lui aussi. Il couvre des applications des états finis pondérés (section 9.1.1) ou de la structure Sumo (section 9.1.2), mais également des applications qui s'éloignent quelque peu des objectifs initiaux de Sumo (section 9.1.3).

Les expériences menées et les applications esquissées ici permettent d'établir les principaux développements à venir de Sumo : premièrement, la réalisation complète du système ; deuxièmement, les interfaces potentielles ; et troisièmement, les extensions fonctionnelles pour de futures versions de Sumo.

9.1 Applications potentielles

9.1.1 Applications potentielles du prototype existant

Le prototype actuel de Sumo permet la réalisation de maquettes d'applications utilisant des transducteurs d'états finis pondérés. La première des applications est en réalité l'interface `wfst.pl`, accompagnés de programmes utilitaires pour compiler des dictionnaires ou dessiner des automates en utilisant `dot`.

La première partie a montré combien les automates d'états finis étaient utilisés en TALN ; aussi propose-t-on ici trois applications potentielles des transducteurs d'états finis pondérés de Sumo.

9.1.1.1 Génération aléatoire sous contraintes

La génération aléatoire de mots d'un langage reconnu par un automate d'états finis pondérés est une fonctionnalité de Sumo (section 7.2.3.3) pour la mise au point d'un reconnaisseur ou d'un transducteur : en générant aléatoirement des mots du langage, on peut détecter rapidement d'éventuels problèmes qui sinon resteraient longtemps non détectés.

Une autre application est la génération de corpus de taille importante à partir d'un langage. Si l'on a un reconnaisseur de phrases, on peut générer des dizaines, des centaines, ou des milliers de phrases de ce langage aléatoirement afin de créer un corpus qui pourra servir pour la validation du reconnaisseur, ou l'entraînement de systèmes statistiques.

De plus, cette génération peut être contrainte : intervalle de longueur souhaitée, éléments à inclure ou à exclure, etc. Le langage contraint peut s'exprimer par l'intersection de l'automate reconnaissant ce langage avec l'automate représentant la contrainte. Un nombre arbitraire de contraintes expressible par une expression régulière, peuvent ainsi être posées. On obtient ainsi un nouvel automate pour le langage contraint, à partir duquel on peut générer aléatoirement un nombre arbitraire d'occurrences peuvent être générées aléatoirement.

9.1.1.2 Traduction de livres de phrases

La traduction de livres de phrases est un sujet qui a surgi récemment, dans le contexte de recherches en traduction de dialogues oraux finalisés. NEC a ainsi utilisé vers 1998-2000 un corpus de 150000 phrases en japonais et en anglais, alignées, pour régler les paramètres de son système de conversation face à face. ATR et le consortium CSTAR ont aussi construit un corpus aligné de 250000 phrases à partir de 120 livres de phrases anglais-japonais, et sont en train de l'étendre à d'autres langues. Le travail est presque fini pour l'italien, et a demandé plusieurs hommes-années. D'autres groupes commencent la traduction vers leurs langues respectives (chinois, français, allemand, coréen).

Il est donc intéressant de chercher des méthodes permettant de réaliser ces traductions de façon plus efficace, et si possible en se servant de dictionnaires existants, et de traductions existantes. Par exemple, si on a un livre de phrases anglais-français, et des traductions alignées japonais-français, on devrait pouvoir au moins identifier les phrases anglaises égales ou similaires et travailler par transitivité. Une autre idée est de factoriser les sous-ensembles de termes qui apparaissent en regard, et de les réutiliser.

Par exemple, on trouvera dans la section "au restaurant" la construction suivante :

Pourriez-vous me donner ... à la place du dessert?	Instead of dessert, may I have ... ?
un café	a coffee
un thé	a tea
une tisane	a herbal tea
du fromage sec	some cheese
un fromage blanc	some cottage cheese

On peut alors automatiquement créer le transducteur du minidictionnaire local bilingue, et l'intégrer à celui de la phrase (figure 9.1).

Grâce à des logiciels d'alignement de phrases, on peut aussi retrouver dans un autre corpus de phrases parallèles (anglais-japonais) une phrase identique ou proche avec même partie variable, la transformer en transducteur, et composer les deux, ce qui donnera le japonais-anglais pour la partie commune.

9.1.1.3 Traduction pidgin pour lecture active

La lecture active consiste, pour un utilisateur qui ne maîtrise pas, ou mal, la langue du texte qu'il veut lire, à consulter rapidement un dictionnaire ou, encore mieux, une version du même texte dans une langue qu'il connaît.

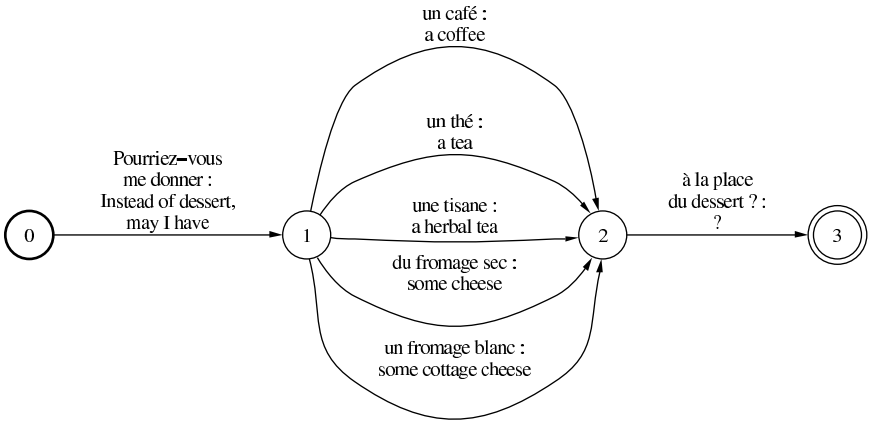

FIG. 9.1 – Transducteur d'une phrase avec variantes

Les aides informatisée à la lecture active sont en général dynamiques et consistent soit à préparer le « dictionnaire de la phrase ou de la fenêtre » (voir par exemple le prototype Compass de Xerox), soit à préparer une traduction complète.

Pour certains couples de langues, il existe des systèmes de TA. En général, ils ne font cependant pas l'affaire, car les traductions sont non seulement mauvaises (ce qui n'est guère étonnant puisqu'il ne s'agit pas de systèmes spécialisés à un contexte donné), mais encore et surtout elles changent trop l'ordre des mots pour que le lecteur trouve facilement les correspondants qu'il cherche. Le genre de traduction souhaitée dans le cas de la lecture active semble plutôt la « traduction pidgin », telle qu'on la trouve dans les manuels de type Assimil après la « bonne traduction » : il s'agit d'une traduction par morceaux, en suivant l'ordre du texte source.

Qu'il s'agisse de construire un dictionnaire local ou une traduction pidgin, il faut toujours :

- segmenter en mots si nécessaire
- faire une lemmatisation (cas du dictionnaire) ou une analyse morphologique complète (cas de la traduction pidgin)
- traduire, en consultant un dictionnaire bilingue, et, dans le cas de la traduction pidgin, en tentant une génération morphologique produisant des formes parallèles (par exemple, traduire un pluriel par un pluriel, un passé par un passé, etc.)

Chacune de ces étapes peut se faire avec un transducteur fini. L'avantage de travailler sur une structure à étages est de pouvoir montrer les différents niveaux, et par exemple de rendre disponible en même temps une traduction purement mot à mot et une traduction pidgin.

9.1.2 Esquisses de réalisation en Sumo des applications initialement prévues

Bien que l'on ne dispose pas pour l'instant d'implémentation complète du système, et notamment pas d'outil pour travailler avec les structures étagées, on peut donner ici les principes de réalisation des applications pour lesquelles Sumo a été conçu dès le départ.

La segmentation à plusieurs niveaux est l'application principale envisagée pour Sumo, et a fait l'objet d'une présentation dans le chapitre 4 : segmentation des mots, puis des segments (*chunks*), des phrases, des sections, etc.

On pourrait aller plus loin, en itérant à certains niveaux. Par exemple, après la segmentation en mots, on peut reconnaître certains types d'unités (noms de personnes, de sociétés, de lieux, de

marques...) à l'aide de motifs réguliers, puis intégrer ces unités au dictionnaire de segmentation, et produire une segmentation de meilleure qualité, qui permettra certains traitements, comme la recherche d'entités nommées.

Une autre application est la construction des étages « bas » d'une mémoire de traduction « à étages » (structures TELA [Planas, 1998]) : flot d'entrée tel quel, puis lexèmes de base comme les caractères dénotés par des entités XML, puis niveau des balises (XML, HTML ou autres), puis niveau des lemmes et des termes techniques.

9.1.3 Autres applications potentielles

Il y a aussi des applications dont on n'a pas parlé plus haut, comme l'analyse multi-niveaux, la construction de banques d'arbres ou la fabrication d'annotations structurées complexes.

L'analyse multi-niveau consiste à représenter sur un même graphe plusieurs niveaux d'interprétation linguistique. Il ne s'agit donc pas ici des différents étages d'une structure TELA ou d'une structure Sumo ; il s'agit de représenter plusieurs niveaux d'information linguistique sur un même étage d'une structure Sumo.

Par exemple, on se propose de représenter sur un graphe de segmentation obtenu par analyse morphématique et morphologique des informations de type syntaxique et sémantique. Cela est très facile en composant l'automate représentant le texte analysé au premier niveau avec des transducteurs implémentant des dictionnaires associant des catégories syntaxiques et sémantiques aux lemmes.

En second lieu, on pourrait aussi représenter une banque d'arbres en Sumo, bien que la structure d'arbre ne soit pas présente en tant que telle. Il suffit d'introduire des balises correspondant aux parenthèses ouvrantes et fermantes et d'utiliser l'écriture linéaire standard des arbres. Par exemple, l'arbre de dépendance projectif *V:*:court(N:suj:Jean, *, Adv:man:vite)* peut être noté par la chaîne

```
V:*:court <(> N:suj:Jean * Adv:man:vite <)>
```

La figure 9.2 illustre la correspondance arbre et chaîne.

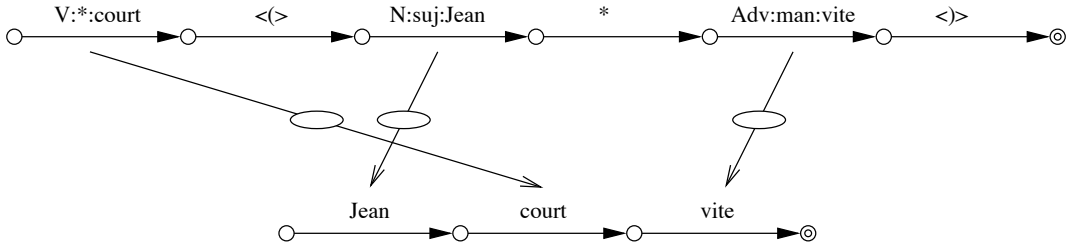

FIG. 9.2 – Correspondance chaîne-arbre

Cela permet de représenter une banque d'arbres par une collection de structures Sumo à deux niveaux (chaînes et arbres), et d'écrire des expressions en Sumo pour rechercher les couples (chaîne, arbre) vérifiant telle ou telle condition, grâce à des transducteurs pouvant exprimer des schémas sur les chaînes aussi bien que des schémas sur cette représentation des arbres. Une possibilité supplémentaire, absente des banques d'arbres actuelles, est de représenter tout ou partie des liaisons entre les sous-chaînes et les sous-arbres ainsi codés.

Enfin, on peut tout à fait imaginer d'utiliser Sumo pour enrichir des textes. Supposons par exemple qu'on veuille annoter les mots pleins d'un texte par des primitives sémantiques, et, pour être plus précis, par des UW d'UNL. On segmente le texte en phrases, et on construit, pour chaque phrase, une structure Sumo contenant le niveau du texte d'entrée (1) et le niveau des lemmes (2), qui comporte

souvent des alternants. On compose ensuite avec un dictionnaire dans 0, 1 ou plusieurs UW pour chaque lemme, tout en construisant la correspondance entre chaque arc contenant un lemme et les arcs du niveau supérieur (3) contenant les UW associées à ce lemme.

On construit ensuite un quatrième niveau à l'aide de la règle d'identification

```
$x => [ $x "<unl uw=\"" $c "\">" ]
  { $x = . ; $c = join(" ", $x.children()); } ;
```

Dans cette règle, la notation `$x.children()` désigne le tableau contenant tous les items du niveau 2 avec lesquels l'item `$x` (de niveau 3) est en relation. Il suffit finalement d'extraire ce quatrième niveau et de le transformer en une forme linéaire pour obtenir le texte annoté.

Exemple : dans la phrase « J'ai vu le chien. », « vu » donne

```
vu<unl uw="see(icl>do,ag>animate)">
```

et « chien »

```
chien<unl uw="dog(icl>animal) dog(icl>thing)">
```

9.2 Perspectives

9.2.1 Approches possibles pour une réalisation complète

On se propose d'abord de poursuivre la méthode employée jusqu'à présent pour le prototype et de réaliser une implémentation du système complet en Perl. Cette implémentation utiliserait en l'étendant la bibliothèque d'états finis pondérés déjà réalisée (chapitre 7), et en suivant le même modèle, c'est-à-dire en écrivant une bibliothèque de fonctions agrémentée d'une ou plusieurs interfaces. Une nouvelle classe d'items dérivant de `WFST::Item` pour représenter les unités de segmentation doit être définie, ainsi qu'une classe pour la structure étagée et ses opérations (sur le modèle de `WFST::Network`).

Un aspect important de Sumo est son langage de contrôle. Celui-ci est syntaxiquement proche de Perl, et peut être facilement traduit en Perl, à l'aide du module `Parse::RecDescent` déjà utilisé pour l'analyse des expressions régulières. À partir de n'importe quel fichier source Sumo, on obtiendrait un programme Perl exécutable normalement et employant les bibliothèques de fonctions de Sumo. L'utilisation de filtres de source (*source filters*) rend une telle opération de traduction transparente.

Même si Perl se révèle tout à fait adapté à la réalisation d'une plate-forme expérimentale pour le système complet, permettant de faire de nouvelles expériences et de corriger d'éventuels défaut dans les spécifications de Sumo, il est probable que la question des performances se posera, du fait notamment de l'utilisation dispendieuse de la mémoire (phénomène déjà observé dans le prototype actuel).

Dans ce cas, on sera amené à choisir un autre langage d'implémentation, ne répondant plus nécessairement aux critères du chapitre 6 mais plutôt à ceux de robustesse, de rapidité d'exécution, d'utilisation judicieuse de la mémoire (pour le traitement de gros documents, qui est tout de même un objectif de Sumo), et de typage plus fort. Le fait d'utiliser un langage à objets est intéressant car la conception de l'implémentation en Perl pourra alors être reprise.

Parmi les candidats qui se présentent alors (CLOS, C/C++, Java, etc.), C++ semble le plus intéressant pour plusieurs raisons : la syntaxe de Perl est proche de celle de C++, ce qui permet une traduction rapide ; la STL de C++ propose des classes de tables de hachage (abondamment utilisées dans la version Perl) ; il peut être possible de réutiliser une bibliothèque d'états finis pondérés existante et disponible librement comme ASTL ; et enfin de nombreux outils de génération d'analyseurs syntaxiques (Yacc, Bison, ANTLR, etc.) sont disponibles pour remplacer `Parse::RecDescent`.

9.2.2 Interfaces spécifiques envisageables

L'interface déjà réalisée pour la manipulation de transducteurs pondérés suit le modèle de la ligne de commande ; les transducteurs eux-mêmes peuvent être décrits indirectement par des expressions régulières et directement exhaustivement en utilisant le format de FSM. L'extension logique pour l'interface du système complet est un interprète Sumo disposant d'une ligne de commande (en mode REPL, ou *read-eval-print loop* comme en LISP) et d'un mode d'exécution direct des applications écrites en Sumo.

Deux développements ultérieurs paraissent intéressants. Le premier est la création de langages spécialisés, à l'instar du langage de description de lexiques de Xerox, pour faciliter la tâche du linguiste pendant l'édition de dictionnaires ou de grammaires Sumo.

Le deuxième développement s'inspire plutôt de INTEX, où les transducteurs sont éditables graphiquement. Ici aussi, l'édition graphique de transducteurs, ainsi que de structures Sumo complètes (qui formeraient un niveau de visualisation au dessus des transducteurs) offrirait de nouvelles possibilités d'interaction avec le linguiste développant des données linguistiques pour une application Sumo. Des outils existent pour cela, depuis les interfaces graphiques à la TK jusqu'aux outils spécialisés comme `dotty` (AT&T), qui permet d'éditer des graphes généraux.

La présence d'interfaces graphiques dans Sumo permettrait alors de nombreux développements. Par exemple, les applications Sumo pourraient à leur tour bénéficier d'une interface graphique, rendant les interactions avec l'utilisateur plus « amicales » et ergonomiques. De la même manière, l'édition graphique d'une structure complexe comme celle de Sumo où les différents niveaux sont en relation pose le problème de la synchronisation des niveaux, qui est un sujet de réflexion sans doute intéressant.

9.2.3 Extensions fonctionnelles

On termine ici par la présentation succincte de quelques extensions fonctionnelles au formalisme que l'on peut déjà prévoir.

9.2.3.1 Relations de sous-graphes

Les relations de chemins mettent en correspondance deux chemins de niveau différent ; si un item ou un chemin d'un niveau donné peut être lié à plusieurs chemins différents à un niveau inférieur, alors on crée autant de relations différentes. Une relation de sous-graphe permettrait de mettre en relation directement un chemin avec un sous-graphe, et donc un nombre arbitraire de chemins différents. De telles relations seraient un outil pratique pour réaliser la segmentation en fragments critiques d'un texte.

9.2.3.2 Relations horizontales

En plus des relations verticales déjà présentes dans Sumo, une nouvelle sorte de relation pourrait être créée pour exprimer des liens horizontaux comme il en existe dans les mémoires de traduction (*eg.* les structures TELA). La figure 9.3 schématise le genre d'application que pourraient avoir ces relation dans un projet tel que Papillon, utilisant différentes ressources monolingues liées entre elles par une structure pivot.

9.2.3.3 Compilation des règles d'identification et de liaison

Sumo emprunte énormément aux méthodes fondées sur les états finis. Les règles de réécriture que l'on emploie (qu'elles soient pondérées ou non) sont parfaitement inscrites dans ce formalisme et

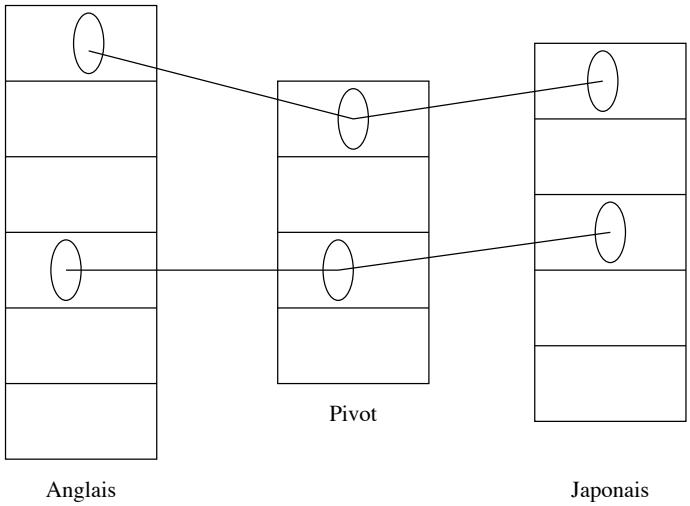

FIG. 9.3 – Alignement de structures Sumo par un pivot

peuvent se compiler sous la forme de transducteurs. L'application des règles s'effectue alors par une composition de transducteurs classique.

Dans l'état actuel de Sumo, les règles doivent être interprétées et le lien entre données linguistiques et règles n'est pas aussi fort. Un objectif est de délimiter une classe de règles d'identification et de liaison qui soient compilables sous la forme de structures Sumo. Pour l'instant, le mécanisme d'extraction de règles permet de pallier à ce manque.

Il semble possible de compiler les règles « simples » (introduites par les opérateurs `->` et `<->`) et il est évident que les règles plus puissantes (`=>` et `<=>`) ne seront pas compilables sous la forme d'une structure Sumo telle qu'on l'a définie plus haut. Il faut également préciser le mode d'application des structures correspondant à ces règles.

9.2.3.4 Bibliothèques de fonctions

Il serait utile d'introduire dans Sumo le concept de bibliothèques génériques réutilisables. On peut envisager deux sortes d'extensions : premièrement, des bibliothèques de fonctions utiles pour la segmentation (telles que des bibliothèques d'algorithmes de désambiguïsation) écrite dans le langage de contrôle de Sumo, ou des extensions écrites dans le langage d'implémentation (donc de plus bas niveau) pour des extensions « fondamentales » (par exemple, l'ajout de nouveaux encodages ou formats de fichiers). Deuxièmement, des collections de ressources linguistiques pour la segmentation réutilisables dans différents contextes et applications.

9.2.3.5 Utilisabilité sur le Web

Sumo apparaît jusqu'à présent comme un programme centralisé dont le moteur et les données doivent se trouver sur la machine de l'utilisateur. Une utilisabilité à distance, via un protocole client/serveur ou sur le Web, est une extension souhaitable, et facilement réalisable. Ainsi, des serveurs Sumo pourraient centraliser des données importantes et être interrogés par des utilisateurs locaux ou distants par le biais d'une simple interface Web, comme il existe déjà de nombreux exemples.

Conclusion

Grâce à sa généricité, Sumo permettra, une fois la réalisation achevée, de traiter de nombreuses applications présyntaxiques - outre celles décrites plus haut, on peut ajouter l'extraction d'entités, la mise en forme de corpus, la conversion de format de documents, la détection de fautes d'orthographes et la correction orthographiques, etc.

Le cadre de segmentation est ouvert et se prête à de nombreuses extensions fonctionnelles, ainsi qu'à l'étude de questions intéressantes comme l'utilisabilité et les interactions avec linguistes, programmeurs et utilisateurs ; ainsi que la réutilisabilité des données mais aussi des algorithmes et des applications créées.

Conclusion

Le problème de la segmentation en mots, ou itémisation, est souvent considéré comme trivial grâce à la présence de séparateurs dans l'écriture. L'essor de l'Internet et surtout du Web a rendu disponibles des millions de documents dans une multitude de langues et généré un intérêt pour les applications multilingues, qui ont rapidement montré les limites des approches simplistes en vigueur jusqu'à présent.

L'étude, d'une part, des systèmes d'analyse morphologiques (en particulier les formalismes fondés sur les états finis), et d'autre part, des applications spécialisées pour l'itémisation dans différentes langues réputées difficiles (japonais, chinois, thaï) nous a mené à des observations contrastées. La notion même de mot, et donc le processus d'itémisation, varie grandement d'une langue à l'autre ; et s'il n'existe pas de méthode générique, surtout en l'absence de séparateurs entre les mots, des approches similaires sont employées par différents systèmes pour différentes langues.

Nous avons donc tenté de nous placer au dessus de l'itémisation et de parler de segmentation de texte en général. Nous avons introduit un langage spécialisé pour la segmentation nommé Sumo (Segmentation Universelle Multiple par Ordinateur) dont la principale caractéristique est d'offrir une séparation claire entre le processus de segmentation et la ou les langues considérées.

Pour cela, nous avons introduit une structure de données dédiée, qui représente un document simultanément à différents niveaux de segmentation (en mots, en phrases, etc.) À chaque niveau correspond un graphe d'items, les unités de segmentation à ce niveau. Cette structure à étages est définie de façon à pouvoir être bien traitée par les transducteurs d'états finis pondérés, et son implémentation étend directement celle de ces transducteurs.

Nous avons défini une algèbre pour la manipulation de cette structure, qui étend elle-aussi l'algèbre servant à la manipulation des automates et transducteurs d'états finis. Enfin, pour obtenir un langage complet, nous avons ajouté un langage de contrôle permettant de construire des applications complètes.

Au niveau théorique, nous avons montré que ce formalisme permet de réimplémenter la plupart des applications pour l'instant implémentées de manière hétérogène.

Au niveau de l'implémentation, nous nous sommes concentré sur la réalisation d'un prototype implémentant les expressions sur les transducteurs d'états finis (ce que ne fait pas FSM) et les poids (ce que ne font ni XFST ni INTEX), ainsi que l'itération au point fixe qui n'est ni dans FSM, ni dans XFST, ni dans INTEX.

Ce prototype nous a permis de traiter un certain nombre d'exemples. Il est pour l'instant écrit en Perl, et nous avons étudié les diverses façons de prolonger cette implémentation pour réaliser un système Sumo complet, robuste et efficace. À court terme, on propose d'utiliser le système dans son état courant pour développer un certain nombre d'applications utiles pour l'accès et la manipulation de la base lexicale Papillon.

Un objectif à plus long terme sera d'intégrer le langage Sumo avec d'autres outils utilisés en recherche d'information ou communication multilingue. Il s'agit d'outils comme des transducteurs d'arbres utilisant la reconnaissance de motifs dans les arbres, ou fondés sur des graphes sémantiques, qui représentent une autre classe de problèmes.

Bibliographie

[Adant, 2000] Arnaud Adant. Study and implementation of a weighted finite-state library - applications to speech synthesis. Rapport technique, TCTS Lab, Faculté Polytechnique de Mons, Belgique, 2000.

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, et Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachussetts, 1986.

[Amtrup *et al.*, 1996] Jan W. Amtrup, Henrik Heine, et Uwe Jost. What's in a Word Graph. Evaluation and Enhancement of Word Lattices. Rapport technique, Universität Hamburg, décembre 1996.

[André et Goossens, 1995] Jacques André et Michel Goossens. Codage des caractères et multilinguisme : de l'ASCII à l'Unicode et ISO/IEC-10646. *Cahiers GUTenberg*, (20):1–54, mai 1995.

[Aït-Mokhtar, 1998] Salah Aït-Mokhtar. *L'analyse présyntaxique en une seule étape*. Thèse de linguistique et informatique, Université Blaise Pascal, Clermont-Ferrand, février 1998.

[Autebert, 1994] Jean-Michel Autebert. *Théorie des langages et des automates*. Masson, Paris, 1994.

[Beesley et Karttunen, 2000] Kenneth R. Beesley et Lauri Karttunen. Finite-State Non-Concatenative Morphotactics. In *ACL 2000*, pages 191–198, Hong Kong, octobre 2000.

[Beesley et Karttunen, 2002] Kenneth R. Beesley et Lauri Karttunen. *Finite-State Morphology: Xerox Tools and Techniques*. Cambridge University Press, Cambrige, UK, 2002.

[Beesley, 1998] Kenneth R. Beesley. Consonant Spreading in Arabic Stems. In *Actes de COLING/ACL 98*, pages 117–123, Montréal, Canada, août 1998.

[Berstel et Reutenauer, 1984] Jean Berstel et Christophe Reutenauer. *Les séries rationnelles et leurs langages*. Masson, Paris, 1984.

[Black *et al.*, 1987] Alan Black, Graeme Ritchie, Steve Pulman, et Graham Russell. Formalisms for Morphographemic Description. In *Actes de EACL 87*, pages 11–18, Copenhague, Danemark, avril 1987.

[Bray *et al.*, 2000] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et Eve Maler. Extensible Markup Language version 1.0. Rapport technique, World Wide Web Consortium, octobre 2000.

[Chang *et al.*, 1992] Jyun-Shen Chang, Shun-De Chen, et Shu-Jin Ke. Large-corpus-based methods for Chinese personal name recognition. *Journal of Chinese Information Processing*, 6(3):7–15, 1992.

[Chauché, 1974] Jacques Chauché. *Transducteurs et arborescences. Études et réalisations de systèmes appliqués aux grammaires transformationnelles*. Thèse de mathématiques, Université Scientifique et Médicale, Grenoble, décembre 1974.

[Chen et Liu, 1992] Keh-Jiann Chen et Shing-Huan Liu. Word Identification for Mandarin Chinese Sentences. In *Actes de COLING 92*, pages 101–107, Nantes, août 1992.

[Church *et al.*, 1991] Kenneth Church, Patrick Hanks, Donald Hindle, et William Gale. Using Statistics in Lexical Analysis. In *Lexical Acquisition: Using On-line Resources to Build a Lexicon*, éditeur U. Zernik, pages 115–164. Lawrence Erlbaum, 1991.

[Church, 1988] Kenneth Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Second Conference on Applied Natural Language Processing*, pages 136–143, Austin, Texas, 1988.

[Colmerauer, 1970] Alain Colmerauer. Les Systèmes Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Rapport technique, Université de Montréal, septembre 1970.

[Daciuk, 1999] Jan Daciuk. Treatment of Unknown Words. In *Workshop on Implementing Automata (WIA)*, juillet 1999.

[Daniels et Bright, 1996] éditeurs Peter T. Daniels et William Bright. *The World's Writing Systems*. Oxford University Press, New York, 1996.

[Davis, 2000] Mark Davis. Unicode Regular Expression Guidelines. Rapport technique, Unicode Consortium, août 2000.

[Francis et Kucera, 1982] W. Francis et H. Kucera. *Frequency Analysis of English*. Houghton Mifflin Company, Boston, Massachussets, 1982.

[Friburger *et al.*, 2000] Nathalie Friburger, Anne Dister, et Denis Maurel. Améliorer le découpage des phrases sous INTEX. In *Journées INTEX 2000*, Lièges, Belgique, 2000.

[Fuchi et Takagi, 1998] Takeshi Fuchi et Shinichiro Takagi. Japanese Morphological Analyzer using Word Co-occurrence - JTAG. In *Actes de COLING/ACL 98*, pages 409–413, Montréal, Canada, août 1998.

[Gale et Sampson, 1995] William A. Gale et Geoffrey Sampson. Good-Turing Frequency Estimation Without Tears. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.

[Gazdar, 1985] Gerald Gazdar. Finite State Morphology: a Review of Koskenniemi (1983). Rapport technique, Center for the Study of Language and Information, Stanford, Californie, septembre 1985.

[Grefenstette et Tapanainen, 1994] Gregory Grefenstette et Pasi Tapanainen. What is a Word, What is a Sentence? Problems of Tokenization. In *COMPLEX 94*, pages 79–87, 1994.

[Guilbaud et Boitet, 1997] Jean-Philippe Guilbaud et Christian Boitet. Comment rendre une morphologie robuste du français encore plus robuste en traitant finement les mots inconnus avec les données disponibles. In *TALN 97*, pages 49–59, Grenoble, juin 1997.

[Guilbaud, 1980] Jean-Philippe Guilbaud. *Analyse morphologique de l'allemand en vue de la traduction par ordinateur de textes techniques spécialisés*. Thèse de linguistique appliquée, Université de la Sorbonne Nouvelle, Paris, juin 1980.

[Guo, 1997] Jin Guo. Critical Tokenization and its Properties. *Computational Linguistics*, 23(4):569–596, décembre 1997.

[Guo, 1998] Jin Guo. One Tokenization per Source. In *Actes de COLING/ACL 98*, pages 457–463, Montréal, Canada, août 1998.

[Habert *et al.*, 1998] B. Habert, G. Adda, M. Adda-Decker, P. Boula de Marëuil, S. Ferrari, O. Ferret, G. Illouz, et P. Paroubek. Towards Tokenization Evaluation. In *LREC 1998*, pages 427–431, Grenade, Espagne, mars 1998.

[Hammerton *et al.*, 2002] James Hammerton, Miles Osborne, Susan Armstrong, et Walter Daelemans. Introduction to Special Issue on Machine Learning Approaches to Shallow Parsing. *Journal of Machine Learning Research*, 2:551–558, mars 2002.

[Hasan, 1999] Md Maruf Hasan. The Encoding and Processing of Indian Languages - an Alternative Approach. In *MAL 99 Workshop*, pages 27–31, Pékin, Chine, novembre 1999.

[Hu et Yu, 2000] Junfeng Hu et Shiwen Yu. The Multi-layer Language Knowledge Base of Chinese NLP. In *Actes de LREC 2000*, Athènes, Grèce, mai 2000.

[Huang *et al.*, 1997] Chu-Ren Huang, Keh-Jiann Chen, Feng-Yi Chen, et Li-Li Chang. Segmentation Standard for Chinese Natural Language Processing. *Computational Linguistics and Chinese Language Processing*, 2(2):47–62, août 1997.

[Johnson, 1972] C. Douglas Johnson. *Formal Aspects of Phonological Description*. Mouton, La Hague, Pays Bas, 1972.

[Kaplan et Kay, 1994] Ronald M. Kaplan et Martin Kay. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3):331–378, septembre 1994.

[Karttunen *et al.*, 1992] Lauri Karttunen, Ronald M. Kaplan, et Annie Zaenen. Two-Level Morphology with Composition. In *Actes de COLING 92*, pages 141–148, Nantes, août 1992.

[Karttunen, 1991] Lauri Karttunen. Finite-state Constraints. In *International Conference on Current Issues in Computational Linguistics*, pages 1–18, Penang, Malaisie, juin 1991.

[Karttunen, 1995] Lauri Karttunen. The replace operator. In *ACL 95*, 1995.

[Karttunen, 1996] Lauri Karttunen. Directed Replacement. In *ACL 96*, pages 108–115, 1996.

[Kashioka *et al.*, 1998] Hideki Kashioka, Yasuhiro Kawata, Yumiko Kinjo, Andrew Finch, et Ezra W. Black. Use of Mutual Information Based Character Clusters in Dictionary-less Morphological Analysis of Japanese. In *Actes de COLING/ACL 98*, pages 658–662, Montréal, Canada, août 1998.

[Kiraz, 2000] George Anton Kiraz. Multitiered Nonlinear Morphology Using Multitape Finite Automata: A Case Study on Syriac and Arabic. *Computational Linguistics*, 26(1):61–76, mars 2000.

[Koskenniemi, 1983] Kimmo Koskenniemi. *Two-level Morphology: A General Computational Model for Word-form Recognition and Production.* Thèse d'informatique (?), University of Helsinki, Finlande, 1983.

[kwong Wong et Chan, 1996] Pak kwong Wong et Chorkin Chan. Chinese Word Segmentation based on Maximum Matching and Word Binding Force. In *COLING 96*, pages 200–203, 1996.

[Lafourcade, 1994] Mathieu Lafourcade. *Génie logiciel pour le génie linguiciel.* Thèse d'informatique, Université Joseph Fourier, Grenoble, France, décembre 1994.

[Laurent, 1977] Annick Laurent. *Détection des mots et de leurs correspondants en japonais écrit : un essai d'analyse morphologique automatique.* Thèse de linguistique appliquée, Université des langues et lettres, Grenoble, France, janvier 1977.

[Li *et al.*, 1998] Haizhou Li, Shuanhu Bai, et Zhiwei Lin. Chinese Tokenization using Viterbi Decoder. In *International Symposium on Chinese Spoken Language Processing*, Singapour, décembre 1998.

[Li et Wang, 1995] Junjie Li et Kaizhu Wang. Study and Implementation of Nondictionary Chinese Segmentation. In *NLPRS 95*, pages 266–271, Séoul, décembre 1995.

[Liu *et al.*, 1994] Y. Liu, Q. Tan, et X. Shen. *Segmentation Standard for Modern Chinese Information Processing and Automatic Segmentation Methodology.* Qinghua University Press, Pékin, 1994.

[Lunde, 1999] Ken Lunde. *CJKV Information Processing.* O'Reilly and Associates, Sebastopol, Californie, 1999.

[Luo et Roukos, 1996] Xiaoqiang Luo et Salim Roukos. An Iterative Algorithm to Build Chinese Language Models. In *ACL 96*, pages 139–143, 1996.

[Maout, 1997] Vincent Le Maout. Automaton Standard Template Library. Rapport technique, Université de Marne-la-Vallée, 1997.

[Matsumoto *et al.*, 1999] Yuji Matsumoto, Akira Kitauchi, Tatsuo Yamashita, et Yoshitaka Hirano. Japanese Morphological Analysis System ChaSen version 2.0 manual. Rapport technique, Nara Institute of Science and Technologie, Nara, Japon, avril 1999.

[Meknavin *et al.*, 1997] Surapant Meknavin, Paisarn Charoenpornsawat, et Boonserm Kijsirikul. Feature-based Thai Word Segmentation. In *NLPRS 97*, pages 41–46, 1997.

[Mittrapiyanuruk et Sornlertlamvanich, 2000] Pradit Mittrapiyanuruk et Virach Sornlertlamvanich. The Automatic Thai Sentence Extraction. In *Actes de SNLP 2000 (Fourth Symposium on Natural Language Processing)*, pages 23–28, Chiang Mai, Thaïlande, mai 2000.

[Mochizuki *et al.*, 1998] Hajime Mochizuki, Takeo Honda, et Manabu Okumura. Text Segmentation with Multiple Surface Linguistic Cues. In *Actes de COLING/ACL 98*, pages 881–885, Montréal, Canada, août 1998.

[Mohri *et al.*, 1997] Mehryar Mohri, Fernando C. N. Pereira, et Michael Riley. A Rational Design for a Weighted Finite-State Transducer Library. In *Workshop on Implementing Automata*, pages 144–158, 1997.

[Mohri *et al.*, 2000] Mehryar Mohri, Fernando C. N. Pereira, et Michael Riley. The Design Principles of a Weighted Finite-State Transducer Library. *Theoretical Computer Science*, 231(1):17–32, 2000.

[Mohri et Sproat, 1996] Mehryar Mohri et Richard Sproat. An Efficient Compiler for Weighted Rewrite Rules. In *ACL 96*, pages 231–238, 1996.

[Mohri, 1997] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, juin 1997.

[Mohri, 2001] Mehryar Mohri. Language Processing with Weighted Transducers. In *TALN 2001*, pages 5–14, Tours, juillet 2001.

[Nagata, 1994] Masaaki Nagata. A Stochastic Japanese Morphological Analyzer Using a Forward-DP Backward-A* N-Best Search Algorithm. In *COLING 94*, pages 201–207, Kyoto, 1994.

[Nederhof, 2000] Mark-Jan Nederhof. Practical Experiments with Regular Approximation of Context-Free Languages. *Computational Linguistics*, 26(1):61–76, mars 2000.

[Nie *et al.*, 1995] Jian-Yun Nie, Marie-Louise Hannan, et Wanying Jin. Unknown Word Detection and Segmentation of Chinese using Statistical and Heuristic Knowledge. *International Journal of the Chinese and Oriental Language Information Processing Society*, 5, 1995.

[Palmer et Hearst, 1997] David D. Palmer et Marti A. Hearst. Adaptative Multilingual Sentence Boundary Disambiguation. *Computational Linguistics*, 23(2):241–267, juin 1997.

[Planas, 1998] Emmanuel Planas. *TELA. Structures et Algorithmes pour la Traduction Fondée sur la Mémoire*. Thèse d'informatique, Université Joseph Fourier, Grenoble, juillet 1998.

[Quint, 1999] Julien Quint. Towards a Formalism for Language-Independent Text Segmentation. In *NLPRS 99*, pages 404–408, Pékin, Chine, novembre 1999.

[Quint, 2000a] Julien Quint. A Formalism for Language-Independent Text Segmentation. In *Actes de COLING 2000*, pages 656–662, Saarbrucke, Allemagne, août 2000.

[Quint, 2000b] Julien Quint. Universal Segmentation of Text with the Sumo Formalism. In *NLP 2000*, Patras, Grèce, juin 2000.

[Quinton, 1980] P. Quinton. *Le système KEAL*. PhD thesis, ???, 1980.

[Roche, 1996] Emmanuel Roche. Parsing with Finite-State Transducers. Rapport technique, Mitsubishi Electric Research Laboratory, Cambridge, Massachussets, novembre 1996.

[Schützenberger, 1961] Marcel Paul Schützenberger. A remark on finite-state transducers. *Information and Control*, 4(2-3):185–196, 1961.

[Seligman *et al.*, 1998] Mark Seligman, Christian Boitet, et Boubaker Meddeb-Hamrouni. Transforming Lattices into Non-deterministic Automata with Optional Null Arcs. In *Actes de COLING/ACL 98*, pages 1205–1211, Montréal, Canada, août 1998.

[Shieber, 1987] Stuart M. Shieber. Separating linguistic analyses from linguistic theories. In *Linguistic Theory and Computer Applications*, éditeurs Peter Whitelock, Mary McGee Wood, Harold L. Somers, Rod L. Johnson, et Paul Bennett, pages 1–36. Academic Press, Londes, 1987.

[Shinnou et Ikeya, 2000] Hiroyuki Shinnou et Masanori Ikeya. Extraction fo unkown words using the probability of accepting the kanji character sequence as one word. In *Actes de LREC 2000*, Athènes, Grèce, mai 2000.

[Silberztein, 1993] Max Silberztein. *Dictionnaires électroniques et analyse automatique de textes : le système INTEX*. Masson, Paris, 1993.

[Sornlertlamvanich *et al.*, 2000a] Virach Sornlertlamvanich, Tanapong Potipiti, et Thatsanee Charoenporn. Automatic Corpus-Based Thai Word Extraction with the C4.5 Learning Algorithm. In *Actes de COLING 2000*, Saarbrucke, Allemagne, août 2000.

[Sornlertlamvanich *et al.*, 2000b] Virach Sornlertlamvanich, Tanapong Potipiti, Chai Wutiwiwatchai, et Pradit Mittrapiyanuruk. The State of the Art in Thai Language Processing. In *ACL 2000*, Hong Kong, octobre 2000.

[Sproat *et al.*, 1996] Richard Sproat, Shilin Chih, William Gale, et Nancy Chang. A Stochastic Finite-State Word-Segmentation Algorithm for Chinese. *Computational Linguistics*, 22(3):377–404, 1996.

[Sproat et Riley, 1996] Richard Sproat et Michael Riley. Compilation of Weighted Finite-State Transducers from Decision Trees. In *ACL 96*, pages 215–222, 1996.

[Sproat et Shih, 1990] Richard Sproat et Chilin Shih Shih. A Statistical Method for Finding Word Boundaries in Chinese Text. *Computer Processing of Chinese and Oriental Languages*, (4):336–351, 1990.

[Sun *et al.*, 1997] Maosong Sun, Dayang Shen, et Changning Huang. CSeg&Tag 1.0: A Practical Word Segmenter and POS Tagger for Chinese Texts. In *Proceedings of Fifth Conference on Applied Natural Language Processing*, pages 119–126, 1997.

[Sun *et al.*, 1998] Maosong Sun, Dayang Shen, et Benjamin K. Tsou. Chinese Word Segmentation without Using Lexicon and Hand-crafted Training Data. In *Actes de COLING/ACL 98*, pages 1265–1271, Montréal, Canada, août 1998.

[Swen et Yu, 1999] Bing Swen et Shiwen Yu. A Graded Approach for the Efficient Resolution of Chinese Word Segmentation Ambiguities. In *NLPRS 99*, pages 19–24, Pékin, Chine, novembre 1999.

[Tomokiyo et Quint, 2000] Mutsuko Tomokiyo et Julien Quint. Comparaison des analyseurs morphologiques du japonais Juman et Chasen. Rapport technique, GETA-CLIPS-IMAG, Grenoble, juillet 2000.

[Vauquois et Chappuy, 1988] Bernard Vauquois et Sylviane Chappuy. Static Grammars. A Formalism For the Description of Linguistic Models. In *International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Colgate University, août 1988.

[Wall *et al.*, 2000] Larry Wall, Tom Christiansen, et Jon Orwant. *Programming Perl, 3rd Edition.* O'Reilly & Associates, Sebastopol, Californie, juillet 2000.

[Wang *et al.*, 1991] Liang-Jyh Wang, Tzusheng Pei, Wei-Chuan Li, et Lih-Ching R. Huang. A Parsing Method for Identifying Words in Mandarin Chinese Sentences. In *IJCAI 91*, pages 1018–1023, 1991.

[Watson, 1995] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Thèse d'informatique, Eindhoven University of Technology, 1995.

[Webster et Kit, 1992] Jonathan J. Webster et Chunyu Kit. Tokenization as the initial phase in NLP. In *Actes de COLING 92*, pages 1106–1110, Nantes, août 1992.

[Wu et Fung, 1994] Dekai Wu et Pascale Fung. Improving Chinese Tokenization with Linguistic Filters on Statistical Lexical Acquisition. In *ACL 94*, pages 180–181, 1994.

[Wu et Tseng, 1993] Zimin Wu et Gwyneth Tseng. Chinese Text Segmentation for Text Retrieval: Achievements and Problems. *Journal of the American Society for Information Science*, 44(9):532–542, 1993.

[Wu, 1997] Dekai Wu. Stochastic Inversion Transduction Grammars and Bilingual Parsing of Parallel Corpora. *Computational Linguistics*, 23(3):377–403, septembre 1997.

[Xia *et al.*, 2000] Fei Xia, Martha Palmer, Nianwen Xue, Mary Ellen Okurowski, John Kovarik, Fu-Dong Chiou, Shizhe Huang, Tony Kroch, et Mitch Marcus. Developping Guidelines and Ensuring Consistency for Chinese Text Annotation. In *Actes de LREC 2000*, Athènes, Grèce, mai 2000.

[Xuong, 1992] Nguyen Huy Xuong. *Mathématiques discrètes et informatique.* Masson, Paris, 1992.

[Zaysser, 1996] Laurence Zaysser. Representing Morpho-syntactic Ambiguity. In *MIDDIM 96 seminar*, pages 205–209, Le Col de Porte, France, août 1996.

Annexes

Annexe A

Systèmes d'écriture, jeux de caractères et encodages

L'encodage est l'opération de représentation numérique d'une suite de caractères. Un encodage particulier est en réalité la représentation numérique d'une table de caractères, dans laquelle les caractères sont ordonnés et ont tous une position précise. La constitution de ces tables de caractères dépend des jeux de caractères des langues considérées, qui dépendent eux du système d'écriture de ces langues.

Les facteurs régissant la conception d'un encodage sont :

- la langue et son système d'écriture. Le système d'écriture d'une langue exprime quels types de caractères sont employés - alphabets, syllabaires, ou idéogrammes, mais également chiffres, ponctuation et nombre d'autres symboles typographiques - et comment ils interagissent [Daniels et Bright, 1996]. On peut aussi vouloir encoder plusieurs langues et avoir à prendre en compte des systèmes d'écriture variés ;
- un ou des jeux de caractères. Un jeu de caractères est l'énumération d'un ensemble de caractères utilisés par un système d'écriture. Il est impossible de créer un jeu de caractère exhaustif, aussi différents jeux de caractères sont définis, correspondant chacun à des besoins spécifiques. Même pour un ensemble de caractères donnés, comme les caractères chinois, les jeux de caractères ne représentent qu'un sous-ensemble des caractères existant réellement. Il faut aussi faire un choix parmi les caractères « auxiliaires » que sont les signes de ponctuation, les symboles mathématiques, et autres signes divers ;
- une ou des table de caractères. Les caractères sont triés, les variantes (comme les graphies différentes d'un même caractère) sont rationalisées, etc. Chaque caractère dispose d'une position précise dans cette table, qui permet d'y faire référence sans ambiguïté.

L'encodage associe à chaque caractère une séquence d'octets, fonction de la table d'origine du caractère et de sa position (point de code) dans cette table.

L'étude des systèmes d'écriture existant dans le monde est un sujet passionnant qui dépasse largement le cadre de cette étude. Pour simplifier et présenter les principales méthodes d'encodages actuelles, les différents systèmes d'écriture sont classés en deux catégories, selon qu'ils reposent sur :

1. un alphabet ou un syllabaire : c'est le cas le plus simple : alphabets et syllabaires ne comportent jamais beaucoup de caractères, de l'ordre de quelques dizaines, auxquels il faut bien entendu ajouter chiffres, signes de ponctuation et autres symboles couramment utilisés dans l'écrit (pourcentage ou symboles monétaires par exemple) ;

2. des idéogrammes. Un exemple évident est le chinois et toutes les langues dont le système d'écriture s'est inspiré du chinois.

A.1 Le code ASCII et la norme ISO-646

Le code ASCII (*American Standard Code for Information Interchange*) est aujourd'hui omniprésent [André et Goossens, 1995]. Normalisé par l'ISO en 1967 (ISO X3.4), son origine remonte à la fin des années 1950 et repose sur des codes plus anciens utilisés pour la transmission d'information par télétype. L'ASCII fait la distinction entre, les caractères de commande (dits aussi de contrôle), destinés au pilotage des télétypes (tabulation, retour à la ligne, cloche, etc.) et d'autre part les caractères dits imprimables, c'est-à-dire les lettres, chiffres, et caractères de ponctuation.

Le jeu de caractères choisi en 1963 est très restreint (48 caractères imprimables, dont 26 lettres majuscules et 10 chiffres, plus 35 caractères de commande). 7 bits sont nécessaires pour encoder un caractère ; cette taille permettant d'encoder un maximum de 128 caractères différents, certaines valeurs sont donc indéfinies. Le standard X3.4 étend le jeu de caractères original en y ajoutant les lettres minuscules et modifie quelques caractères de commande. Les 128 caractères sont alors définis et se divisent entre 34 caractères de commande (dont l'espace, occupant les 33 premières positions ; la dernière étant réservée au caractère d'effacement `DEL`. D'autres caractères d'espacement sont définis comme les tabulations horizontales et verticales et les sauts de ligne) et 94 caractères imprimables :

```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

La norme ISO-646 distingue 12 caractères optionnels, permettant ainsi la création de variantes nationales et d'une version internationale de référence, dite IRV. Ces caractères optionnels sont :

```
# $ @ [ \ ] ^ ` { | } ~
```

En France, on a ajouté entre autres des caractères accentués (`à`, `ç`, `é`, `ù` et `è`) ; d'autres pays ont ajouté leur symbole monétaire. ASCII et ISO-646 correspondent depuis 1988 au même encodage, où la variante américaine (sans caractères accentués, mais avec crochets, accolades, et autres symboles utilisés par les informaticiens) a été choisie.

A.2 Les normes ISO-8859-n

Les variantes locales de l'ISO-646, largement insuffisantes, ont finalement mené à la création de jeux de caractères étendus utilisant les 8 bits d'un octet, soit deux fois plus de caractères encodables. Dans un premier temps, les efforts des différents pays, constructeurs d'ordinateurs et éditeurs de logiciels ont eu pour résultat de nombreux jeux de caractères incompatibles entre eux et liés à des systèmes particuliers.

Au milieu des années 1980, l'ECMA (*European Computer Manufacturer's Association*) a proposé d'unifier les encodages européens en se fondant sur l'ASCII pour créer une nouvelle norme. 256 caractères ne suffisant toujours pas à représenter tous les caractères européens (surtout si l'on inclut les autres alphabets utilisés en Europe comme le grec et le cyrillique, et même en dehors de l'Europe comme l'arabe, l'hébreu, le thaï ou le devanagari), plusieurs tables de caractères ont été proposées. Cela a mené à la création de la famille de normes ISO-8859-*n*, décrites par la table A.1.

La première séparation de l'alphabet latin en Latin-1 à 4 ayant créé de nombreux problèmes (en français par exemple la ligature « œ » est absente, ainsi que le *ÿ* majuscule), il a été nécessaire de procéder à des aménagements comme le montrent les autres ISO-Latin.

Tous ces encodages suivent le même principe : les 128 premiers caractères sont identiques à l'ASCII, ce qui assure une parfaite compatibilité avec tout texte encodé en ASCII. Les 128 caractères suivants

n	nom courant	langues couvertes
1	Latin-1	Europe occidentale et Amérique Latine (allemand, anglais, danois, espagnol, féroïen, finnois, français, islandais, italien, néerlandais, norvégien, portugais, suédois)
2	Latin-2	Europe orientale (albanais, allemand, anglais, croate, hongrois, polonais, roumain, slovaque, slovène, tchèque)
3	Latin-3	autres langues utilisant l'alphabet latin (afrikaans, anglais, allemand, catalan, espagnol, esperanto, italien, maltais, néerlandais, turc)
4	Latin-4	Europe du Nord (allemand, anglais, danois, estonien, finnois, groënlandais, letton, lituanien, norvégien, suédois, sami)
5	latin/cyrillique	
6	latin/arabe	
7	latin/grec	
8	latin/hébreu	
9	Latin-5	variante de Latin-1 pour le turc
10	Latin-6	sami, nordique et eskimo
11	latin/thaï	
12	latin/devanagari	
13	Latin-7	langues baltes (letton)
14	Latin-8	langues celtiques (gaëllique et gallois)
15	Latin-9 ou Latin-0	corrections du Latin-1 pour le français et le finnois

Tab. A.1 – les normes ISO-8859-n

(qui, codés sur 8 bits, ont tous leur octet de poids fort à 1), se partagent en 32 caractères de commande (ou « non-imprimables ») et 96 caractères imprimables.

A.3 Autres encodages « alphabétiques »

Même si les ISO-8859 se sont imposé largement, notamment l'ISO-Latin-1 en Europe occidentale avec le Web[1], il existe toujours des encodages spécifiques à une langue (pour le russe ou l'arabe, des standards nationaux sont encore préférés aux récents standards ISO) ou à un système (le système d'exploitation Windows de Microsoft a longtemps utilisé son propre encodage, et le MacOS d'Apple persiste au moins jusqu'à sa version 9). La plupart sont tout de même fondés sur l'ASCII, à l'exception de l'EBCDIC (*Extended Binary Coded Decimal Information Code*). Encore utilisé par IBM dans ses gros systèmes, il a été comme l'ASCII enrichi de variantes nationales similaires à l'ISO-646.

Certaines langues qui utilisent un alphabet sont tout de même à l'étroit dans les 96 caractères que la norme ISO-8859 met à leur disposition. Un exemple est le vietnamien : le système d'écriture actuel est bien fondé sur l'alphabet latin (qui a remplacé les caractères chinois utilisés jusqu'au XVII^{ème} siècle) mais l'usage intensif de diacritiques (accents, marques de ton) produit plus de 96 caractères. On a donc recours à des encodages différents, comme TCVN-5712 (recommandé par le gouvernement vietnamien), VISCII, ou des encodages propres à des systèmes comme Windows ou MacOS.

A.4 Encodages du chinois, du japonais et du coréen

L'utilisation de jeux de caractères réduits et d'encodages simples comme l'ASCII ont conduit à considérer qu'un caractère été toujours représenté par un octet. Les langages de programmation, C en tête, sont généralement fondés sur ce principe très simple. Mais l'espace qui suffisait pour encoder un alphabet (mais pas pour *plusieurs* alphabets, comme le montrent les différents ISO-Latin) ne peut contenir des dizaines de milliers de caractères, comme en chinois, japonais ou coréens.

Ces trois langues illustrent bien les méthodes d'encodage de jeux de caractères étendus [Lunde, 1999]. Elles ont aussi pour point commun d'utiliser dans leur écriture les mêmes caractères chinois : *hanzi* en Chine, *kanji* au Japon, et *hanja* en Corée. On s'intéressera dans un premier temps plus au chinois et au japonais pour revenir aux spécificités du coréen.

A.4.1 Jeux et tables de caractères

Il est impossible de dénombrer le nombre de caractères chinois existants, tout comme il est impossible de compter tous les mots d'une langue. Les grands dictionnaires de caractères en dénombrent des dizaines de milliers, dont certains extrêmement rares dans l'écriture quotidienne. Des efforts de classification ont été faits, d'abord dans un but pédagogique : en Chine, une liste de 7 000 *hanzi* appelée *tongyong hanzi*, qui définit quels sont les caractères enseignés dans les écoles aux élèves Chinois ; à Taïwan, des listes similaires, définissant *hanzi* à enseigner, caractères rares, etc. Au Japon, le même genre de liste existe pour définir quels sont les 1 945 *kanji* enseignés jusqu'au lycée.

Il existe également des définitions de jeux de caractères à but administratif : en Chine, 2 249 *hanzi* ont vu leur graphie simplifiée dans les années 1960. L'écriture de la République Populaire de Chine est désormais le chinois dit « simplifié » car il utilise systématiquement la nouvelle graphie pour ces caractères d'usage courant. Cette réforme, adoptée également par Singapour, a évidemment été ignorée par Taïwan et Hong Kong, où l'on utilise toujours l'écriture dite « traditionnelle ». Un autre exemple de liste administrative est la liste des 285 *kanji* acceptés par le gouvernement japonais dans l'écriture des patronymes.

[1] la spécification de HTML 4 utilise l'ISO-Latin-1 comme encodage par défaut.

Ces listes de caractères sont une première étape vers la définition de jeux de caractères pour l'échange d'information numérique, qui permettent l'encodage de textes. Chaque gouvernement a donc publié ses propres standards, qui organisent caractères chinois, alphabets nationaux et étrangers et symboles divers. Des standards industriels existent également, et sont parfois plus utilisés que les standards nationaux : c'est le cas à Taïwan où le Big Five domine.

Le principe des tables utilisés est cependant similaire. Un tel jeu de caractères comprend :

- des milliers de caractères chinois, répartis en plusieurs « niveaux ». Ces caractères peuvent exister sous plusieurs formes (par exemple, traditionnelle et simplifiée, ou avec des variantes) ;
- des alphabets et syllabaires : alphabet latin, grec, cyrillique ; *iraganaa* et *katakana* japonais, *zhuyin* chinois[2] ;
- chiffres arabes, ponctuation et symboles divers.

La plupart des caractères sont définis en« pleine largeur », c'est-à-dire qu'ils sont aussi larges que hauts. Les caractères romains, comme ceux définis par l'ASCII, sont dits en « demi largeur » car ils sont deux fois moins larges. Dans beaucoup d'encodages, d'ailleurs, les caractères en demi largeur sont encodés sur un octet et ceux en double largeur sur deux octets. Certains caractères asiatiques, comme les *kana* japonais, sont spécifiés à la fois en pleine largeur et en demi largeur, même si leur usage est marginal sous cette dernière forme.

À titre d'exemple, la table A.2 ci-dessous montre quelques tables de caractères et le nombre de caractères qu'elles contiennent ; dans toutes ces tables, les caractères chinois sont répartis en deux niveaux (les caractères des deux niveaux peuvent être classés différemment).

Jeu de caractères	Niveau 1	Niveau 2	Symboles
GB 2312-80 (Chine)	3 755	3 008	682
Big Five (Taïwan)	5 401	7 652	913
JIS X 0208:1997 (Japon)	2 965	3 384	524

Tab. A.2 – Exemples de jeux de caractères chinois et japonais

L'organisation de ces tables est quasiment identique. Les caractères sont tous disposés en matrices de 94×94 caractères (certaines tables ont plusieurs matrices, appelées plans) et l'on fait référence à la position d'un caractère par les coordonnées de la cellule qu'il occupe (notation ligne-cellule). Si l'on prend l'exemple de la table GB 2312-80, les 3 755 *hanzi* de niveau 1 occupent les lignes 16 à 55 de la table, les 3 008 *hanzi* de niveau 2 occupent les lignes 56 à 87, et les autres symboles sont classés dans les 10 premières lignes de la table.

Il existe bien plus de tables de caractères que celles présentées ici. Ces autres tables sont soit des évolutions des tables existantes (correction de la graphie ou de l'ordre de certains caractères, ajouts plus ou moins importants), soit des extensions introduisant des milliers de caractères supplémentaires, comme GB 6345.1-86 et GB 8565.2-88 en Chine, Big Five Plus à Taïwan ou encore JIS X 0212-1990 au Japon.

A.4.2 Méthodes d'encodage

Il faut faire attention à ne pas confondre méthode d'encodage et table de caractère encodée. Les tables de caractères vues ci-dessus existent indépendamment de toute forme d'encodage. La même table de caractère peut être encodée de plusieurs façons, et inversement, une même méthode d'encodage peut

[2]Les *zhuyin* sont, comme les *kana*, destinés à représenter la prononciation d'un mot mais ne sont pas utilisé seuls dans l'écriture courante. On les appelle aussi *bopomofo*, prononciation des quatre premiers caractères.

encoder plusieurs tables de caractères. Ces deux aspects sont très importants, car certaines méthodes d'encodage (ISO-2022 et EUC) peuvent ainsi être utilisés pour différentes langues ; de plus, encoder différentes tables de caractères par la même méthode permet d'inclure les caractères ASCII pour l'anglais, ainsi que les tables étendues pour disposer de plus de caractères.

On dresse ci-dessous une liste rapide des différentes méthodes d'encodage utilisées en pour le chinois, le japonais ou le coréen (certaines s'appliquent à d'autres langues, comme le vietnamien).

ISO-2022. Il s'agit d'une méthode d'encodage modale où des séquences de caractères particulières permettent de passer d'un mode à l'autre. La norme ISO-2022 est en réalité très complexe, aussi a-t-elle été réduite pour les besoins de chaque région (ISO-2022-CN en Chine, ISO-2022-JP au Japon, ISO-2022-KR en Corée, etc.) permettant l'encodage d'un nombre spécifique de tables de caractères. Dans la plupart des cas, il s'agit de l'ASCII, d'une table de caractères nationale, et d'une ou plusieurs extensions.

L'encodage d'un caractère provenant d'une table étendue se fait avec deux octets. On remarque qu'il existe en ASCII 94 caractères imprimables, et que les tables de caractères ont une dimension de 94×94 : on peut donc encoder chaque caractère par une paire (comme dans la notation ligne-cellule) de caractères ASCII. Les séquences de caractère de changement de mode, elles aussi exprimées à l'aide de caractères ASCII, permettent de savoir combien d'octets font les caractères qui suivent et à quelle table ils font référence.

Un inconvénient de cet encodage est le surcoût imposé par les séquences de changement de mode, qui ajoutent 4 octets supplémentaires au texte à chaque changement de mode. Mais comme tous les caractères sont codés sur 7 bits, cette méthode d'encodage est pratique pour la transmission de texte sur un réseau (et même nécessaire quand l'intégrité du huitième bit n'est pas garantie), et est surtout utilisé à cette fin (pour l'e-mail ou Usenet par exemple), plutôt que pour le stockage.

EUC. L'Extended Unix Coding, qui comme son nom l'indique s'est d'abord développé sous Unix, permet lui aussi d'encoder différentes langues et existe dans différentes versions, comme EUC-JP au Japon ou EUC-CN en Chine. Les caractères sont codés sur un ou plusieurs octets, mais il n'existe pas de mode : le premier octet d'un caractère indique le nombre total d'octets occupés par un caractère, et sa table d'origine. Contrairement à l'ISO-2022, les 8 bits de l'octet sont utilisés. Les avantages et les inconvénients sont donc inverses : plus compacte, cette forme est plus pratique pour le stockage de texte. Par contre, on peut encoder moins de tables de caractères différentes.

Prenons l'exemple de la variante japonaise de l'EUC, qui encode quatre tables de caractères : l'ASCII, le JIS X 0208:1997, les *katakana* en demi-largeur, ainsi que le JIS X 0212-1990 (cette extension contient environ 6 000 caractères supplémentaires). La table A.3 montre les différentes valeurs possibles des octets (les valeurs sont données en hexadécimal) d'un caractère :

Table de caractère	Octets	1^{er} octet	$2^{ème}$ octet	$3^{ème}$ octet
ASCII	1	`21-7E`		
JIS X 0208:1997	2	`A1-FE`	`A1-FE`	
Katakana	2	`8E`	`A1-DF`	
JIS X 0212-1990	3	`8F`	`A1-FE`	`A1-FE`

Tab. A.3 – Spécifications de l'encodage EUC-JP

L'encodage EUC-CN, qui permet d'encoder l'ASCII et le GB 2312-80, est très utilisé en Chine où son nom est souvent abrégé GB (à ne pas confondre avec les tables de caractères !) C'est sans doute le plus utilisé pour le chinois simplifié, même en dehors de Chine. Au Japon, on utilise aussi beaucoup l'EUC-JP, que l'on appelle aussi plus simplement EUC tout court.

Big Five (Taïwan). Big Five est le nom de la méthode d'encodage de la table éponyme, qui permet également d'encoder l'ASCII. En fait, cette méthode est proche de l'EUC (et de la version taïwannaise de l'EUC, qui n'encode pas un jeu de caractères aussi grand que Big Five, qui compte tout de même plus de 13 000 caractères) : un caractère ASCII est encodé sur un octet tandis qu'un caractère Big Five est encodé sur deux octets. Le premier octet est également toujours supérieur à 128 (son bit de poids fort est à 1). Big Five est sans doute l'encodage le plus utilisé pour le chinois traditionnel, même en dehors de Taïwan.

Shift-JIS (Japon). Alors que l'EUC-JP est plutôt utilisé sous Unix, Microsoft a défini son propre encodage qui est utilisé sur PC et Mac (mais également quelques Unix). Il est relativement proche de l'EUC-JP, car il encode les mêmes tables de caractères à l'exception du JIS X 0212-1990 (un caractère ne fait donc qu'un ou deux octets en Shift-JIS, mais jamais trois), mais sur des plages de valeurs différentes. Les *katakana* en demi largeur sont également codés sur un seul octet contre deux pour l'EUC-JP.

A.4.3 Le coréen

Le cas du coréen est intéressant, car contrairement au japonais qui a assimilé les caractères chinois dans son système d'écriture (auxquels se sont ajoutés *hiragana* et *katakana*), les coréens ont d'abord adopté les caractères chinois (*hanja*) puis ont créé leurs propres caractères, appelés *hangul*. Les caractères chinois n'ont plus qu'un usage traditionnel en Corée, bien qu'il existe des standards nationaux comparables à ceux vus plus haut pour le chinois.

Comme en chinois, il existe des milliers de *hangul*, mais ceux-ci sont en fait construits méthodiquement à partir d'éléments nommés *jamo*, qui sont proches des lettres de l'alphabets. Les *jamo* se combinent pour former des syllabes et créer des caractères *hangul* : il existe 19 *jamo* initiaux, 21 *jamo* intermédiaires, et 27 *jamo* finaux ; et en tout 11 172 combinaisons possibles (le dernier *jamo* pouvant être absent). La figure A.1 illustre la création d'un *hangul* à partir de trois *jamo*.

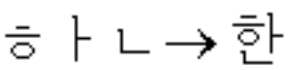

Fig. A.1 – Formation d'un *hangul* à partir de *jamo*

Ainsi, une première méthode d'encodage consistait à n'encoder que les *jamo*, et laisser au système d'affichage le soin de générer les caractères automatiquement (cette méthode est appelée *johab*). Cependant, une autre approche existe : chacun des différents *hangul* est considéré séparément, comme les différents caractères du chinois. On a alors des tables de caractères comprenant les caractères coréens (en plus des caractères chinois). Les méthodes d'encodages sont les mêmes que celles vues plus haut, car dans les deux cas il y a beaucoup de caractères à encoder. L'EUC-KR semble être très répandu, sur le Web par exemple.

Il existe d'autres langues où la combinaison de caractères rend la tâche de l'affichage difficile, même en ignorant les problèmes liés au sens d'écriture. En arabe, chaque lettre peut avoir une forme différente selon sa position dans un mot (en début, milieu, fin ou seule) ; dans certaines langues indiennes, le traitement des ligatures est également complexe et mériterait une solution similaire à celle adoptée en Corée [Hasan, 1999], passant d'un encodage sur un octet à un encodage sur plusieurs octets.

A.5 Unicode et ISO-10646

A.5.1 Présentation

Malgré la possibilité technique d'encoder plusieurs tables de caractères, donc des caractères de langues différentes (par exemple grâce à l'ISO-2022), il est toujours difficile de faire cohabiter plusieurs langues aux systèmes d'écritures différents dans un même document. Même dans le cas des langues européennes, il est impossible de mélanger deux langues qui ne sont pas couvertes par la même norme ISO-8859-*n*.

Il y a environ dix ans, l'ISO et un consortium formé par de grands industriels, le consortium Unicode, ont lancé deux initiatives parallèles de création d'un jeu de caractères universel, qui engloberait toutes les langues possibles. Ces deux efforts se sont rapidement rejoints, et évoluent depuis en parallèle. La norme de l'ISO, numérotée 10646, définit un espace d'encodage gigantesque, l'*Universal Character Set* : 256 groupes de 256 plans, chaque plan étant constitué de 256 lignes de 256 cellules (soit 2^{32} caractères possibles, un nombre supérieur à quatre milliards). Parmi ces plans, c'est au plan 0, le *Basic Multilingual Plane* (BMP) que cette discussion s'intéresse principalement.

L'un des premiers objectifs d'Unicode a été l'unification des caractères chinois (*Han unification*). En effet, si plusieurs langues utilisent des caractères chinois, les tables de caractères définies dans chaque pays sont différentes : les jeux de caractères varient et ne sont pas organisés de la même manière. Les positions des caractères communs ne correspondent pas. Selon la région, la graphie des caractères change aussi : ce peut être une différence assez radicale, comme les caractères simplifiés de Chine Populaire, ou une variante plus subtile du même caractère entre la Chine, le Japon, la Corée et le Vietnam. La norme Unicode prend tout cela en compte, en cherchant à unifier les caractères sans pour autant gommer leurs différences d'apparence. Un même caractère peut avoir plusieurs variantes qui partagent le même point de code.

Un autre aspect intéressant et le maintien de la compatibilité entre Unicode et les jeux de caractères nationaux ; ainsi on peut retrouver à partir d'un caractère Unicode son pendant dans une table japonaise ou chinoise, ce qui est utile pour le transcodage. De plus, les 256 premiers caractères (en terme de position dans la table) sont identiques à ceux de l'ISO-8859-1, par conséquent les 128 premiers caractères d'Unicode sont également ceux du code ASCII.

La version la plus récente d'Unicode (version 3.1.1) contient des dizaines de *scripts* différents pour un total de 94 140 caractères. C'est un standard vivant, qui évolue, auquel de nombreux caractères sont régulièrement ajoutés, mais la majeure partie des aspects intéressants sont stables.

A.5.2 Encodages

Le BMP est divisé en 256 lignes de 256 cellules. Cette organisation ressemble à celle des tables de caractères utilisées pour les caractères chinois ou japonais ; le plan est divisé en plusieurs zones contenant caractères alphabétiques, symboles divers, emplacements réservés pour des extensions privées, ainsi qu'une très large plage pour les caractères chinois unifiés.

Le point de code d'un caractère dans le plan est donné dans la notation ligne-cellule. Comme les lignes contiennent 256 cellules (soit 2^{16}) et que le plan contient 256 lignes, on indique traditionnellement la position du caractère en hexadécimal : les deux premiers chiffres indiquent la ligne, et les deux derniers la cellule. Par exemple, l'espace (qui a pour code ASCII 32) est noté `U+0020` (la première ligne du plan est numérotée 0), et un caractère chinois est par exemple `U+5F41`.

Il existe différentes méthodes d'encodage liés à l'Unicode, dont :

UCS-2. Cet encodage sur 16 bits (2 octets) correspond aux points de code dans le BMP. Ainsi, le caractère `U+0020` est codé `0020` et le caractère `U+5F41` est codé `5F41` en UCS-2.

UCS-4. C'est un encodage sur 32 bits (4 octets) qui permet d'encoder tout l'espace de l'ISO/IEC 10646 (l'UCS-2 ne permet d'encoder qu'un plan). Les quatre octets se présentent dans l'ordre suivant : groupe (octet-G), plan (octet-P), rang (octet-R) et cellule (octet-C). Le BMP étant le le plan 0 du groupe 0, les deux premiers octets sont toujours nuls pour un caractère du BMP. Le caractère `U+0020` est donc codé `00000020` et le caractère `U+5F41` est codé `00005F41`.

UTF-1. Première transformation de la famille UTF (pour *UCS Transformation Format*[3]), c'est une transformation sur 8 bits qui évite les caractères de commande de l'ISO-2022. Présentant de nombreux défaut, elle n'est désormais plus utilisée. Un caractère peut occuper de 1 à 5 octets ; par exemple `U+0020` est transformé en `20`, et `U+5F41` en `F64AFF`.

UTF-7. L'UTF-7 est une transformation à 7 bits s'apparentant aux transformations de type Base 64 et est utile, comme l'ISO-2022, pour la transmission de données, par exemple par e-mail à l'aide de MIME. Il utilise seulement 7 bits. Le caractère `U+0020` est donc transformé en un simple blanc, quant au caractère `U+5F41` il est transformé en la séquence `+X0E-`.

UTF-8. L'UTF-8 est une transformation sur 8 bits, version améliorée de l'UTF-1, où les caractères sont encodés sur 1 à 6 octets (les caractères du BMP prennent au maximum 3 octets). Le premier octet permet toujours de savoir combien d'octets va occuper un caractère et les octets suivants suivent tous le même format ; il est donc facile de savoir où l'on se situe dans un flux de caractères en UTF-8. Un autre avantage, et non des moindres, et que tous les caractères de l'ASCII ont exactement le même code en UTF-8, et qu'aucun caractère non-ASCII n'est représenté par un caractère ASCII (tous les octets d'un tel caractère ont un bit de poids fort à 1).

Sous cette forme, le caractère `U+0020` a simplement pour code `20`, tandis que `U+5F41` (un caractère du BMP) se code sur 3 octets par `E5BD81`.

UTF-16. L'UTF-16 est une transformation permettant d'obtenir UCS-4 à l'aide de codes UCS-2. Elle réserve deux fois 1024 codes UCS-2 (soit plus d'un million de positions) pour étendre le BMP afin de pouvoir inclure des caractères chinois supplémentaires, ainsi que des caractères historiques qui ne sont plus utilisés (par exemples les hiéroglyphes égyptiens, qui font partie du SMP, *Supplementary Multilingual Plane*, le plan 1).

Cette liste n'est pas exhaustive ! Cependant, les deux encodages de références sont l'UCS-2 et surtout l'UTF-8.

A.6 Dépendance du système

Normalement, l'encodage d'un fichier texte devrait être indépendant de tout système d'exploitation. Bien sûr, les encodages propriétaires (ceux définis par Microsoft, Apple, IBM, etc.) sont propres à un système d'exploitation donné ; cependant, si une application sur un autre système est capable de lire un fichier dans cet encodage ou de le transformer dans un autre encodage connu, le document est lisible normalement.

Il existe pourtant une exception à cette situation, qui est l'interprétation des caractères de commande. Les 32 premiers caractères du code ASCII (et donc de la plupart des autres encodages, car ils se fondent sur l'ASCII) sont des caractères de commande, non-imprimables, qui ont été définis du temps des télétypes. Ainsi, la sémantique de certains a évolué depuis les années 1950 ; l'exemple le plus pertinent pour notre discussion concerne les sauts de lignes.

À l'époque des télétypes, le saut de ligne était indiqué par deux caractères nommés CR (*carriage return*, ou retour de chariot, de code ASCII 13, aussi noté `\r`) et `LF` (*line feed*, de code ASCII 10, noté `\n`). Le premier indiquait au télétype de passer à la ligne suivante, mais avant de pouvoir commencer à imprimer sur cette nouvelle ligne, il fallait le temps à la tête d'impression de revenir au début de

[3]Pour plus de détails sur ces transformations, consulter `http://czyborra.com/utf/`

la ligne. Pour ne pas stopper la transmission de données, un caractère spécial était transmis pour marquer une pause pendant le déplacement de la tête d'impression.

Aujourd'hui, la signification de ces deux caractères a évolué différemment selon les systèmes. Pour montrer trois approches différentes, on peut citer :

- MS DOS, qui a conservé la séquence `CR-LF` pour indiquer un saut de ligne ;
- Unix, qui n'a conservé que le caractère `LF` ;
- Mac OS, qui n'a conservé que le caractère `CR`.

Ainsi, un simple document encodé en ASCII et écrit sur un Macintosh ne s'affichera pas correctement sur une station Unix, car les sauts de lignes auront mystérieusement disparu ; pire encore, venant de MS DOS, un document aura l'air normal sous Unix, mais il peut se produire des choses étranges lors de son traitement à cause d'un caractère `CR` parasite à la fin de chaque ligne !

Il reste un dernier problème lié à la transmission d'informations entre plates-formes différentes : selon les microprocesseurs et le protocole de transport sur le réseau, les octets ne sont pas forcément dans le même ordre. Unicode définit ainsi un caractère spécial dit *Byte Order Mask* (BOM), `U+FEFF`, qui permet à toute application de vérifier l'ordre dans lequel sont transmis les octets (ce caractère, s'il est présent, est typiquement le premier d'un flux, et permet à l'application de faire un choix immédiat).

Annexe B

Segdict

B.1 Le moteur d'itémisation

Le source complet (qui compte seulement une quarantaine de lignes) de `segdict.pl` est donné ci-dessous. Le programme dispose de deux options :

- `--dict` pour spécifier un fichier dictionnaire autre que celui par défaut, et
- `--defaut` qui permet de spécifier une règle par défaut différente.

```
#!/usr/bin/perl

use strict;
use warnings;
use locale;
use File::Basename;
use Getopt::Long;

my $dir = (fileparse $0)[1];
my $dict = "$dir/fr.dic";
my $defaut = '$&';
GetOptions("dict=s" => \$dict, "defaut=s" => \$defaut);

open DICT, $dict or die "Erreur d'ouverture du dictionnaire $dict : $!\n";
my @dict =  map  { my @F = split /\t/, $_, 2;
   [qr($F[0]), defined $F[1] ? $F[1] : $defaut ] }
            grep { /\S/ }
            map  { chomp; s/(?!\\)\#.*$//; s/\\\#/\#/g; $_ } <DICT>;
close DICT or die "Erreur de fermeture du dictionnaire $dict : $!\n";
push @dict, [".", '$&'];

#   Lit le texte en entrée "paragraphe par paragraphe"

$/ = "\n\n";

while (<>) {
  s/^\s*//; s/\s*$//; s/\s+/ /g;
  while ($_) {
    for my $motif (@dict) {
      if (s/^$motif->[0]//) {
```

```
        if (my $item = eval($motif->[1])) {
          print "$item\n";
        }
        s/^ ?//;
       last;
      }
    }
  }
}
```

B.2 Le lexique du français

C'est le lexique présenté dans le chapitre 2, ici en intégralité.

```
# Titres

M\.
Mr\.
Mrs\.
MM\.

# Numéros de section

\d+(\.\s*\d+)*

# Références bibliographiques

\[[^\]]+ \d+]

# Mots composés

(?i)a priori
(?i)aujourd'hui
(?i)bien qu[e'] "bien que"
(?i)chevals? d'arçons
(?i)de bonne heure
(?i)hic et nunc
(?i)parce qu[e'] "parce que"
(?i)pommes? de terre

# Élision (cas particulier : possessif anglais)

(?i)\w+'s\b
(?i)(c|d|j|l|m|n|(\w*qu)|s|t)'

(?i)du\b "de\nle"

# Mots avec des tirets, attention a : "va-t-il", "y a-t-il", "avait-il", etc.

(\w+(?:\-\w+)*)\-t\-(\w+) lc"$1\n$2"
\w+(\- ?\w+)* (my$i=$&)=~s/ //g;$i
```

```
# Ponctuation

\-\-*
\=\=*
\(\.\.\.\)
\.\.\.
!!!
\?\?\?
\W+
```

SPÉCIFICATION ET RÉALISATION D'UN FORMALISME GÉNÉRIQUE POUR LA SEGMENTATION MULTIPLE DE DOCUMENTS TEXTUELS MULTILINGUES

Résumé. Le problème de la segmentation en mots, ou itémisation, est souvent considéré comme trivial à cause de la présence de séparateurs dans l'écriture. L'essor de l'Internet et surtout du Web a rendu disponibles des millions de documents dans une multitude de langues et généré un intérêt pour les applications multilingues, qui ont rapidement montré les limites des approches simplistes en vigueur jusqu'à présent.

L'étude, d'une part, des systèmes d'analyse morphologiques (en particulier les formalismes fondés sur les états finis), et d'autre part, des applications spécialisées pour l'itémisation dans différentes langues réputées difficiles (japonais, chinois ou thaï) mène à des observations contrastées. La notion même de mot, et donc le processus d'itémisation, varie grandement d'une langue à l'autre ; et s'il n'existe pas de méthode générique, surtout en l'absence de séparateurs entre les mots, des approches similaires sont employées par différents systèmes pour différentes langues.

On propose de se placer au dessus de l'itémisation et de parler de segmentation de texte en général. On introduit un langage spécialisé pour la segmentation nommé Sumo (Segmentation Universelle Multiple par Ordinateur) dont la principale caractéristique est d'offrir une séparation claire entre le processus de segmentation et la ou les langues considérées. Sumo définit une structure de donnée dédiée, fondée sur les automates d'états finis pondérés, qui représente un document simultanément à différents niveaux de segmentation, ainsi qu'un ensemble d'opérations pour la manipulation de ces structures, similaire au calcul d'états finis. Enfin, un langage de contrôle permet de programmer des applications de segmentation sophistiquées.

Un prototype expérimental de calcul à états finis pondéré a été réalisé en Perl, et la réalisation d'un système complet, efficace et robuste est discutée. Les applications actuelles et potentielles de Sumo sont présentées, ainsi que les perspectives de développements à venir.

Mots-clés : segmentation, analyse morphologique, analyse présyntaxique, transducteurs d'états finis pondérés, langage spécialisé pour la linguistique, Perl.

SPECIFICATION AND IMPLEMENTATION OF A UNIVERSAL FORMALISM FOR MULTIPLE SEGMENTATION OF MULTILINGUAL TEXTUAL DOCUMENTS

Abstract. The issue of word segmentation, or tokenization, is often treated as a trivial matter because of the use of separators in writing. The rise of the Internet and the Web led to the availability of millions of documents in countless languages, which in turn led to a renewed interest for mutlingual applications. These applications rapidly showed the limitations of the simplistic approaches in use until now.

Studying morphological analyzers (especially the ones based on finite-state technology) on the one hand, and specific tokenization applications for "hard" languages (Chinese, Japanese or Thai) on the other hand yields contrasted observations. The very notion of word, and in turn of tokenization, varies widely from one language to the other; and if there is no universal method, especially when there are no written separators, similar approaches are used by different systems for different languages.

A proposal is made to consider any kind of text segmentation, rather than tokenization. A specialized language for segmentation is introduced, named Sumo. Its main feature is to offer a clear distinction between the segmentation process and the considered language(s). Sumo defines a dedicated data structure based on weighted finite-state automata, as well as a set of operations on this structure similar to finite-state calculus. Programming sophisticated segmentation applications is done using a specialized control language.

An experimental prototype for weighted finite-state calculus has been implemented in Perl, and the implementation of a full, efficient and robust system is discussed. Current and potential applications of Sumo are discussed, as well as future work on the formalism.

Keywords: segmentation, morphological analysis, presyntactic analysis, weighted finite-state transducers, specialized languages for linguistics, Perl.