IDIAP

Rapport technique

Une technique efficace de traitement en Prolog de la morphologie flexionnelle du français

Jean-Luc Cochard

Octobre 1992

INSTITUT DALLE MOLLE D'INTELLIGENCE ARTIFICIELLE PERCEPTIVE
CASE POSTALE 609 - 1920 MARTIGNY - VALAIS - SUISSE
TELEPHONE : ++41 26 22.76.64 - FAX : ++41 26 22.78.18
E-MAIL : IDIAP@IDIAP.CH

Numéro : 92-04

Une technique efficace de traitement en Prolog de la morphologie flexionnelle du français

Jean-Luc Cochard

Institut Dalle Molle d'Intelligence Artificielle Perceptive (IDIAP)
Case postale 609, CH-1920 Martigny, Suisse
Adresse électronique : cochard@idiap.ch

Résumé

Ce rapport décrit les différentes étapes qui ont conduit à la sélection d'une solution efficace et relativement peu encombrante pour un analyseur morphologique du français. Ce travail a été réalisé dans le cadre d'un projet de recherche financé par le Fonds National Suisse de la Recherche Scientifique, dans le cadre du Programme National de Recherche 23 : "Specification and prototyping of a system for the intelligent management of information". Chaque étape du processus de sélection est décrite précisément aussi bien par la forme du dictionnaire qui est construit que par l'analyseur morphologique qui gère ce dictionnaire. Les avantages et les faiblesses de chaque solution sont mises en évidence à l'aide de tests comparatifs portant sur l'analyse d'un texte fixé avec des dictionnaires de différentes tailles.

Mots-clé : linguistique informatique, traitement de la langue naturelle, morphologie, dictionnaires informatiques, programmation logique, Prolog.

Table des Matières

1 Introduction

Prolog est un langage de programmation énormément utilisé dans le traitement de la langue naturelle (TLN). Dans ce domaine particulier d'application on trouve une littérature très riche présentant en détail et de manière souvent très pédagogique des extensions de Prolog répondant aux exigences des linguistes en la matière ([9, 8, 7] parmi les plus connus). Cependant, l'effort principal de présentation dans cette littérature spécialisée porte sur *la composante syntaxique* d'un système de TLN, à la fois sur les formalismes comme DCG, MG, XG, etc. et sur les stratégies d'analyse. Force est de constater qu'une place très petite est faite aux techniques d'analyse morphologique, bien que cette composante figure généralement en première place dans la séquence des traitements qui aboutissent à une analyse d'un texte écrit. Certains documents réduisent même cette composante à la notion simpliste de *consultation de dictionnaire*. Une telle vision simplificatrice peut se justifier lorsqu'on traite une langue comme l'anglais qui est relativement peu fléchie. Par contre lorsqu'on travaille sur une langue plus riche en flexions comme le français ou bien l'italien ou l'allemand, l'analyse morphologique prend des proportions plus importantes et l'absence de littérature sur le sujet met chaque programmeur dans la situation de "réinventer la roue". Cet article tente donc modestement de combler partiellement cette lacune en décrivant quelques approches techniques de traitement de la morphologie flexionnelle du français dans le contexte d'un dictionnaire volumineux représentant largement plus de 100'000 formes fléchies.

L'analyseur morphologique décrit dans ce document ne traite qu'un sous-ensemble très simple de transformations morphologiques, connu sous le nom de *morphologie flexionnelle*, bien qu'il existe par ailleurs toute une gamme d'approches largement plus sophistiquées basées sur les travaux de K. Koskenniemi [6, 3]. Les travaux de Koskenniemi ont permis de concevoir des outils de description des phénomènes morphologiques à la fois extrêmement puissants mais aussi extrêmement délicats à utiliser. M. Simard dans [10] met en lumière certains de ces problèmes en relation avec une langue comme le français et dans le cadre d'une implantation en Prolog. Si bien que dans un contexte pratique où la morphologie n'est pas une fin en soi mais une étape dans un traitement linguistique complexe, une solution plus simple est justifiable.

Les différentes versions de l'analyseur morphologique décrit ci-dessous répondent à un cahier des charges très simple qui s'énonce comme suit :

> L'analyse morphologique doit, premièrement, isoler le premier "mot" d'un texte à analyser, la notion de mot n'étant pas forcément définissable par des critères purement typographiques.
>
> L'analyse morphologique doit, deuxièmement, fournir une description morpho-syntaxique du mot qui a été isolé, à l'aide d'une structure d'attributs qui code les valeurs pour les attributs classiques comme genre, nombre, personne, etc.

Ainsi "à bout portant" peut logiquement être considéré comme un mot puisqu'il s'agit d'une expression figée et qu'il apparaît comme locution adverbiale dans les dictionnaires conventionnels. C'est donc le contenu du dictionnaire qui définira implicitement la notion de mot.

Nous pouvons résumer ce cahier des charges à l'aide de la définition formelle suivante du prédicat `morpho/3`, prédicat qui définira l'interface entre les descriptions morphologiques et les niveaux supérieurs d'analyse linguistique (syntaxe, sémantique, etc) :

```
morpho(+String0, -FS, -String1)
```

`String0` est unifié avec le texte en entrée, constitué d'une liste de codes ASCII, sans aucune forme de segmentation préalable.

`FS` est unifié avec la structure d'attributs ("features structure") du mot dont la forme fléchie apparaît en tête de `String0`.

`String1` est unifié avec le texte résultat, c'est-à-dire avec le texte amputé du mot qui vient d'être reconnu.

La suite de cette présentation s'articule de la manière suivante : nous allons d'abord présenter la structure adoptée pour une définition d'une entrée lexicale[1] et des classes de flexion dans la section 2. Ces deux types de données forment le noyau de base sur lequel s'appuient toutes les versions d'analyseurs morphologiques. La section 3 décrit en détail un analyseur morphologique simple mais extrêmement inefficace qui utilise toutes ces données de base, d'où son intérêt, et qui permet aussi de fournir des valeurs de référence pour les résultats des tests comparatifs. La section 4 présente des solutions qui améliorent sensiblement les performances de l'analyseur morphologique mais au détriment d'autres critères mesurés dans les tests comparatifs. Finalement dans la section 5, nous proposons une solution intermédiaire qui combine les avantages réciproques des solutions précédentes, tout en diminuant la portée de leurs inconvénients. C'est sur cette solution finale qu'est construit l'analyseur morphologique du projet des Archives [2], moyennant quelques extensions qui sont décrites brièvement dans la section 6 et qui feront l'objet d'une publication ultérieure.

2 L'architecture d'un dictionnaire informatique de traitement du français

Les informations que doit contenir un dictionnaire informatique permettant le traitement du français sont de deux niveaux : premièrement, une description de traits, ou attributs, indépendants d'une quelconque flexion du mot et, deuxièmement, une description des règles de construction des formes fléchies avec les traits spécifiques à chacune des variantes possibles. Le dénominateur commun entre ces deux niveaux de description est la structure d'attributs. Il s'agit conceptuellement d'une liste de paires attribut-valeur. L'accès à la valeur de chacun des attributs se fait à l'aide de prédicats d'accès ce qui permet de cacher l'implantation réelle de cette structure (cf. Figure 1). Pour les besoins de ce document, nous avons aussi volontairement restreint les attributs à un ensemble minimal afin de diminuer la taille des programmes présentés.

```
cat(FS, Cat)                  % Cat ∈ {noun, adjective, verb, prep, adverb,
                              %        conjunction, pronoun, determiner}
gender(FS, Gender)            % Gender ∈ {masc, fem}
number(FS, Number)            % Number ∈ {sing, plur}
person(FS, Person)            % Person ∈ {1, 2, 3}
tense(FS, Tense)              % Tense ∈ {pres, imparf, p_simple, f_simple, passe}
v_mode(FS, Mode)              % Mode ∈ {infin, indic, subjonct, condit, imper, partic}
inflection_class(FS, Class) % Class est un atome, ensemble trop nombreux pour être
                              % décrit extensivement !
lex_form(FS, Lexical_form)    % Lexical_form est un atome
```

Figure 1 Liste complète des prédicats d'accès constituant une interface à la structure d'attributs avec en regard les valeurs possibles pour chaque attribut.

L'implantation de la structure d'attributs est volontairement très simple car un tel choix n'a pas de réelle influence sur l'efficacité du système. Comme ce point n'est pas central dans cet exposé, nous n'avons pas voulu entrer en matière sur des solutions sophistiquées — il y a toute une partie du chapitre 7 de [7] qui traite ce sujet. Une structure d'attributs est un terme Prolog `fs/4` dont l'organisation des arguments est décrite à la Figure 2.

[1] On utilisera aussi le terme d'*unité lexicale* pour recouvrir la même notion.

```
fs(Cat,
   agr(Gender, Number, Person),
   verbal(Tense, Mode),
   form(Class, Lexical_form))
```

Figure 2 Forme du terme qui code une structure d'attributs morphologiques et qui apparaît à la place de la variable `FS` dans la définition des prédicats d'accès (cf. Figure 1).

2.1 Les entrées lexicales

Le rôle d'une entrée lexicale est de grouper dans une définition toute une collection de traits morpho-syntaxiques, voire sémantiques, qui sont indépendants d'une forme fléchie particulière d'un mot. Une entrée lexicale sera indexée par une forme de citation ("lexical form", dans les programmes) qui suit les conventions des dictionnaires usuels : verbe à l'infinitif, adjectif au masculin singulier, etc.

```
dict_entry("maison", FS) :-
          cat(FS, noun),
          inflection_class(FS, n1),
          gender(FS, fem).
```

Figure 3 La forme de l'entrée de dictionnaire du mot "maison".

Techniquement, une entrée lexicale est réalisée par une clause Prolog `dict_entry/2` qui définit une relation entre la forme de citation — codée à l'aide d'une chaîne de caractères — et une structure d'attributs `FS` partiellement instanciée. Nous verrons dans les différentes versions d'analyseurs morphologiques comment une telle relation, définie strictement pour une seule forme, est généralisée pour s'appliquer à toutes les formes fléchies d'*un* mot, ce qui finalement lui confère bien le rôle que nous voulons lui assigner.

2.2 Les classes de flexions

Le rôle des classes de flexions est de spécifier précisément les variations orthographiques d'un mot. Dans ce but, nous avons adopté un principe extrêmement simple de décomposition des formes fléchies d'un mot :

> Chaque mot est caractérisé par une *racine* qui est la partie initiale du mot. Cette racine a la particularité d'être présente dans toutes les formes fléchies. La construction de la forme fléchie se fait alors par l'adjonction d'une *terminaison*, qui constitue l'autre partie de la décomposition.

Cette solution n'offre pas la possibilité de construire des formes par adjonction de préfixes. C'est une limitation qui n'est pas réellement contraignante en français, puisque les formes fléchies sont construites par adjonction de suffixes. Cette solution est assez puissante pour décrire tous les cas de figure de mots simples si nous admettons que la racine ou certaines terminaisons sont "vides".

Les classes de flexions sont donc définies par des *tables de terminaisons*. Pour la classification des verbes, nous avons adopté un standard en la matière pour le français : Le Bescherelles [4], qui fournit une description exhaustive de toutes les classes de flexions verbales. Pour les noms, les adjectifs et les pronoms, il n'existe rien de semblable. Nous avons créé des classes à partir de zéro au gré des découvertes de nouvelles configurations de terminaisons.

```
endings_entry(number_only, n1, % eg MAISON-S
              ["", % singular
               "s" % plural
              ]).

endings_entry(verb, vb6, % comme AIM ER
        ["er",                                                    % infinitif
         "e", "es", "e", "ons", "ez", "ent",                      % présent, indic.
         "ais", "ais", "ait", "ions", "iez", "aient",              % imparfait, indic.
         "ai", "as", "a", "âmes", "âtes", "èrent",                % passé simple, indic.
         "erai", "eras", "era", "erons", "erez", "eront",         % futur, indic.
         "e", "es", "e", "ions", "iez", "ent",                    % présent, subj.
         "asse", "asses", "ât", "assions", "assiez", "assent",    % imparfait, subj.
         "erais", "erais", "erait", "erions", "eriez", "eraient", % présent, condit.
         "e", "ons", "ez",                                        % impératif présent
         "ant", "é", "és", "ée", "ées"                            % participes
        ]).
```

Figure 4 Deux définitions de classes flexionnelles avec une table de terminaisons pour des noms qui nécessitent simplement l'adjonction d'un “s” au pluriel et une autre table pour des verbes qui se conjuguent comme “aimer”.

Une table de terminaisons qui définit une nouvelle classe de flexion est réalisée par une clause Prolog `endings_entry/3` :

`entry_endings( ?Type, +Class, -Endings)`

- `Type` est un atome qui permet de rattacher cette clause de `endings_entry/3` à une clause de `endings_pattern/3`.
- `Class` est un atome qui est le nom de la classe de flexion décrite par cette clause.
- `Endings` est une liste dont les éléments sont des terminaisons codées sous forme de chaînes de caractères. L'ordre d'énumération des terminaisons est déterminé précisément par la clause de `endings_pattern/3` à laquelle se réfère cette définition.

Pour qu'une définition de classe flexionnelle soit complète, il est nécessaire que chaque terminaison soit caractérisée par une structure d'attributs partiellement instanciée. C'est le rôle dédié à un prédicat auxiliaire `endings_pattern/3` qui permet de simplifier la définition des classes en évitant de reproduire, pour chacune d'elles, la structure d'attributs associée à chaque terminaison.

```
endings_pattern(number_only, FS,
                [number(FS, sing),
                 number(FS, plur)
                 ]).

endings_pattern(verb, FS,
   [%infinitif présent
    (tense(FS, pres), v_mode(FS, infin)),

    % présent de l'indicatif
    (tense(FS, pres), v_mode(FS, indic), person(FS, 1), number(FS, sing)),
    (tense(FS, pres), v_mode(FS, indic), person(FS, 2), number(FS, sing)),
```

```
    (tense(FS, pres), v_mode(FS, indic), person(FS, 3), number(FS, sing)),
    (tense(FS, pres), v_mode(FS, indic), person(FS, 1), number(FS, plur)),
    (tense(FS, pres), v_mode(FS, indic), person(FS, 2), number(FS, plur)),
    (tense(FS, pres), v_mode(FS, indic), person(FS, 3), number(FS, plur)),

    % imparfait de l'indicatif
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 1), number(FS, sing)),
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 2), number(FS, sing)),
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 3), number(FS, sing)),
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 1), number(FS, plur)),
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 2), number(FS, plur)),
    (tense(FS, imparf), v_mode(FS, indic), person(FS, 3), number(FS, plur)),

    % passé simple de l'indicatif
    ...

    % participes présents et passés
    (tense(FS, pres), v_mode(FS, partic)),
    (tense(FS, passe), v_mode(FS, partic), gender(FS, masc), number(FS, sing)),
    (tense(FS, passe), v_mode(FS, partic), gender(FS, masc), number(FS, plur)),
    (tense(FS, passe), v_mode(FS, partic), gender(FS, fem), number(FS, sing)),
    (tense(FS, passe), v_mode(FS, partic), gender(FS, fem), number(FS, plur))
   ]).
```

Figure 5 Deux définitions du prédicat auxiliaire `endings_pattern/3` permettant de compléter l'information des classes flexionnelles de la Figure 4.

Le prédicat `endings_pattern/3` illustré par des exemples dans la Figure 5, est défini de la manière suivante :

`endings_pattern(+Type, ?FS, -FS_descriptions)`

`Type` est un atome qui sert à nommer la clause, est utilisé comme référence par `endings_entry/3`.

`FS` est une structure d'attributs sur laquelle vont s'appliquer les descriptions d'attributs contenues dans `FS_descriptions`.

`FS_descriptions` est une liste de descriptions de structures d'attributs, via des prédicats d'accès. Chaque élément de la liste est une conjonction de prédicats d'accès dont l'exécution permet de compléter `FS`.

`endings_pattern/3` définit un ordre pour des tables de terminaisons car les listes qui figurent comme $3^{\text{ème}}$ argument de `endings_entry/3` et de `endings_pattern/3` sont traitées en parallèle (cf. la Figure 6 dans la section 3 pour une première illustration de ce traitement). L'ordre des terminaisons, bien qu'assez logique en soi, pourrait subir des permutations selon les désirs de chacun. Il y a cependant une contrainte très importante à respecter :

Contrainte d'ordre des terminaisons – La première terminaison dans la liste doit être celle de la forme de citation adoptée pour l'indexation des unités lexicales.

3 Un analyseur morphologique simpliste

Nous sommes en mesure maintenant de présenter un analyseur morphologique rudimentaire mais répondant aux exigences du cahier des charges et utilisant les différents prédicats introduits ci-dessus. À la suite de cette description, nous décrivons la méthodologie adoptée pour les tests comparatifs et fournissons les premiers résultats relatifs à cet analyseur morphologique.

3.1 Le corps de l'analyseur morphologique

L'analyseur morphologique est défini par le prédicat `morpho/3` ainsi que par une série de prédicats auxiliaires (cf. Figure 6).

```
morpho(String0, FS, String3) :-
        skip_spaces(String0, String1),
        optional_lowercase(String1, String2),
        word_selection(String2, FS, String3).

word_selection(String0, FS, String2) :-
        % Algorithme :
        % - sélection non-déterministe d'une entrée du dictionnaire
        % - extraction de la racine de l'entrée
        % - première comparaison : la racine et la partie initiale de String0
        % - deuxième comparaison : la partie initiale de String1 et une des
        %   terminaisons possibles.
        dict_entry(Lexical_form, FS),
        atom_chars(Lexical_atom, Lexical_form),
        lex_form(FS, Lexical_atom), % affectation automatique de FS !
        inflection_class(FS, Class),
        endings_entry(Pattern_name, Class, Endings),
        endings_pattern(Pattern_name, FS, FS_descriptions),
        root_extraction(Lexical_form, Root, Endings),
        root_elimination(String0, Root, String1),
        ending_identification(String1, Endings, FS_descriptions, String2),
        end_of_word(String2).

optional_lowercase(String, String).
optional_lowercase([Ch_upper|String], [Ch_lower|String]) :-
        lowercase(Ch_upper, Ch_lower). % définition de lowercase/2 absente !

skip_spaces([Ch | String0], String1) :-
        % Élimination des caractères de ponctuation, pour cette application
        punctuation(Ch), % définition of punctuation/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces([Ch | String0], String1) :-
        % Élimination des caractères TAB, "espace" et CR
        effective_space(Ch), % définition of effective_space/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces(String, String).

end_of_word([]).% En fin de texte, la condition est vérifiée
end_of_word([Ch | _String]) :-
        % sur un caractère de ponctuation, aussi !
        punctuation(Ch).
end_of_word([Ch | _String]) :-
```

```
        % sur une marque d'espacement, aussi !
        effective_space(Ch).

root_extraction(Lexical_form, Root, Endings) :-
        % La première terminaison est celle de la forme de citation
        % utilisée pour indexer les entrées.
        Endings = [Lexical_ending | _],
        append(Root, Lexical_ending, Lexical_form).

root_elimination(String0, Root, String1) :-
        append(Root, String1, String0).

ending_identification(String0, [Ending | _Endings], [Descr | _Descrs], String1) :-
        % Lorsque String0 débute par Ending !, nous sommes en
        % présence d'un candidat valide. Dès lors, nous
        % devons exécuter Descr qui complètera FS en y
        % introduisant les valeurs associées à la terminaison.
        append(Ending, String1, String0),
        call(Descr).
ending_identification(String0, [_ | Endings], [_ | Descrs], String1) :-
        % tentative avec une autre terminaison
        ending_identification(String0, Endings, Descrs, String1).
```

Figure 6 Un analyseur morphologique simple mais inefficace.

Le prédicat `morpho/3` de la Figure 6 définit un traitement morphologique en trois étapes :

`skip_spaces(String0, String1)` – *Élimination des "espaces"* – Cette opération permet de lire sans les traiter les caractères ASCII qui séparent des mots entre eux. Dans cette catégorie, nous plaçons les espaces, codes de tabulation, retour de ligne et caractères de ponctuation. La ponctuation pourrait très bien être traitée comme une unité syntaxique mais, pour les besoins de cette expérience, nous avons privilégié la simplicité.

`optional_lowercase(String1, String2)` – *Transformation orthographique* – Nous avons introduit ici un cas simple de transformation orthographique qui permet de traiter les mots en début de phrase qui commencent par une majuscule. La mise en oeuvre de cette transformation est indéterministe, mais elle privilégie la forme originale écrite avant d'appliquer la transformation si c'est possible (cf. l'ordre des clauses `optional_lowercase/2`).

`word_selection(String2, FS, String3)` – *Sélection d'un mot* – Il s'agit de l'opération centrale de cette version de l'analyseur morphologique. Son objectif est de sélectionner une entrée lexicale dont la *racine* suivie d'une *terminaison* possible de cette entrée constituent la partie initiale du texte à analyser.

Le prédicat `word_selection/3` de la Figure 6 définit une sélection d'un mot de la manière suivante :

`dict_entry(Lexical_form, FS)` – *Sélection non-déterministe d'une entrée lexicale* – En analyse morphologique, les deux arguments `Lexical_form` et `FS` sont des variables libres, ce qui rend la recherche tout à fait non déterministe et particulièrement inefficace. Mais comme la forme fléchie d'un mot peut être totalement distincte de la forme de citation (cas de figure avec une racine "vide"), on ne peut faire aucune hypothèse sur la forme de citation pour améliorer la recherche.

`atom_chars(...), lex_form(FS, Lexical_atom)` – *Instanciation partielle de la structure d'attributs* – Pour des raisons de lisibilité — les chaînes de caractères sont particulièrement illisibles en Prolog si elles ne sont pas affichées selon un format spécifique — nous avons choisi de mémoriser la forme de citation au sein de `FS` sous forme atomique.

`inflection_class(FS, Class)` – *Détermination de la classe de flexion* – En choisissant une entrée lexicale, l'attribut "classe flexionnelle" est unifié à une valeur précise. On extrait donc cette valeur pour sélectionner la bonne table de terminaisons.

`endings_entry(Pattern_name, Class, Endings)` – *Sélection de la table de terminaisons* – En plus de nous fournir la table de terminaisons, ce prédicat nous donne, via `Pattern_name`, le nom de la table auxiliaire de description des structures d'attributs de chaque terminaison.

`endings_pattern(Pattern_name, FS, FS_descriptions)` – *Sélection de la table auxiliaire* – Toutes les informations pertinentes en relation les unes avec les autres sont donc disponibles pour effectuer les manipulations. Cette unification dans laquelle apparaît `FS` comme 2$^{\text{ème}}$ argument, engendre une unification avec la structure d'attributs courante, dans tous les éléments de `FS_descriptions`.

`root_extraction(Lexical_form, Root, Endings)` – *Détermination de la racine* – Ce traitement s'appuie sur la *Contrainte d'ordre des terminaisons* émise à la page 7.

`root_elimination(String0, Root, String1)` – *Test sur la racine* – Il s'agit de la première partie de l'identification d'un mot qui met en correspondance la racine calculée avec le début du texte à analyser.

`ending_identification(String1, Endings, FS_descriptions, String2)` – *Test sur la terminaison* – Cette vérification termine l'identification d'un mot en complétant la structure d'attributs sur la base de la terminaison choisie.

`end_of_word(String2)` – *Test de la fin d'un mot* – Comme les mots de la langue naturelle ne constituent pas un "langage minimal" au sens de la théorie des langages de programmation [1], il est indispensable de tester la présence d'un caractère qui marque une fin de mot.

Les autres prédicats auxiliaires décrits dans la Figure 6 sont suffisamment simples pour ne pas les détailler ici.

3.2 Les tests comparatifs

Nous avons choisi de comparer les différentes versions d'analyseurs morphologiques selon deux critères : un critère de rapidité avec le temps d'exécution d'une analyse morphologique et un critère d'encombrement avec la taille nécessaire au stockage des dictionnaires. Tous les tests ont été réalisés en SICStus-Prolog 0.7 #9, sur une SUN SPARCstation IPX avec 16MB de mémoire vive.

Les tests comparatifs de rapidité que nous avons mis en place consistent à effectuer une analyse morphologique du texte suivant :

> *Il fut possible de survoler de nombreuses surfaces endommagées par les tempêtes dans des forêts directement protectrices mais certains cantons ne disposent actuellement que de vues aériennes partielles malgré le grand engagement de tous les participants. Vous trouverez un bilan des survols par canton dans une annexe de la présente circulaire.*

Les temps de traitement de deux stratégies d'analyse ont été retenus :

Une analyse sérielle, dans laquelle on se contente de la première solution fournie par l'analyseur. Ce résultat fournit le temps minimum de traitement morphologique du texte.

Une analyse exhaustive, dans laquelle toutes les analyses possibles des mots qui composent le texte sont fournies. On obtient ainsi la borne supérieure du temps de traitement morphologique.

L'encombrement produit par chaque solution envisagée est caractérisé par deux valeurs :

Le nombre de clauses Prolog nécessaires au stockage du dictionnaire.

La taille du dictionnaire sous forme compilée[2].

Tous les tests comparatifs ont été effectués dans la perspective de saisir l'influence de la taille du dictionnaire sur les valeurs obtenues. Notre dictionnaire actuel comporte grosso modo 20'000 entrées lexicales. Nous avons donc choisi de créer aléatoirement dix "sous-dictionnaires" contenant respectivement 2'000, 4'000, 6'000, etc. entrées lexicales en prenant bien garde que les mots du texte test figurent dans chacun d'entre eux.

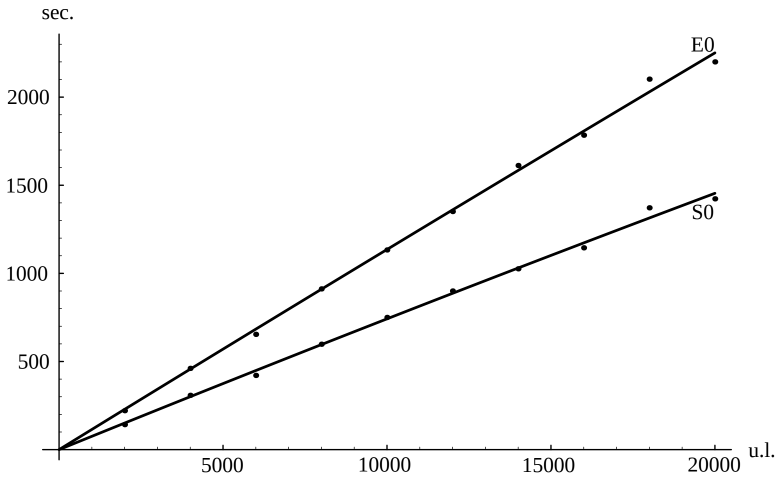

Figure 7 Courbes des durées en secondes, de l'analyse sérielle **S0** et exhaustive **E0** en fonction de la taille du dictionnaire en unités lexicales (**u.l.**).

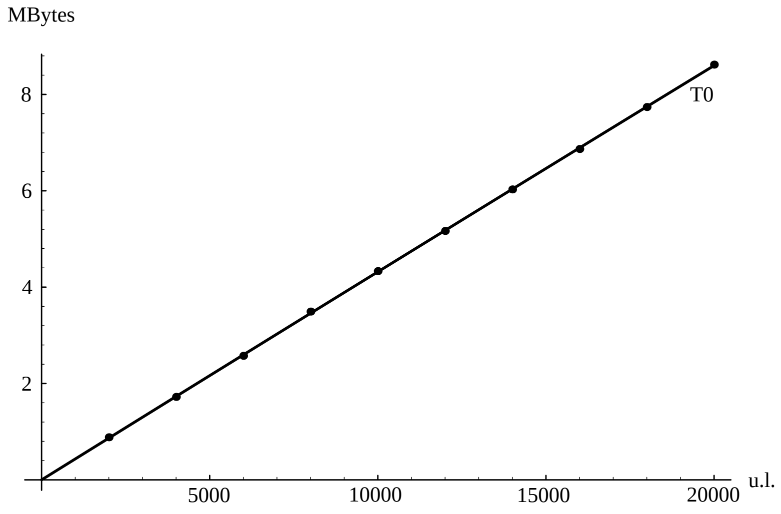

Figure 8 Courbe **T0** de la taille du dictionnaire compilé en MBytes, en fonction de la taille du dictionnaire en unités lexicales (**u.l.**).

[2] La version compilée d'un programme Prolog avec `fcompile/1` est moins sensible aux choix de dénomination des prédicats locaux et fournit donc un résultat plus objectif.

Pour notre analyseur morphologique simpliste nous obtenons les résultats suivants :

- La Figure 7 fournit une interpolation quadratique sur la base des résultats ponctuels d'analyses sérielles et exhaustives. Cette solution apparaît d'emblée comme impraticable. L'unité de temps étant la seconde, on mesure à quel point l'indéterminisme de la sélection des entrées lexicales pèse sur l'efficacité de l'analyse.

- Le nombre de clauses nécessaires au stockage du dictionnaire est strictement égal au nombre d'unités lexicales de ce dictionnaire dans cette version puisqu'elle manipule directement les entrées lexicales.

- La taille du dictionnaire compilé est fournie dans la Figure 8. Ce résultat n'a pas d'intérêt en soi, mais il servira de point de comparaison avec les autres solutions.

4 Deux versions performantes d'analyseurs morphologiques

Pour rendre l'analyseur morphologique largement plus efficace que celui qui est présenté ci-dessus et ainsi obtenir une solution utilisable, nous avons adopté une technique qui consiste à transcrire le dictionnaire et les descriptions des classes flexionnelles en structures d'*arbres discriminatoires* (cf. section 4.1). Le problème essentiel auquel nous devons faire face ici est celui de la technique d'implantation d'un arbre discriminatoire en Prolog. Deux techniques sont proposées : la première, *l'arbre discriminatoire actif* (cf. section 4.2), génère un programme Prolog et la deuxième, *l'arbre discriminatoire passif* (cf. section 4.4), génère des faits Prolog. Dans le premier cas, l'analyseur morphologique devient trivial. Dans le deuxième cas, l'analyseur morphologique est un interprète de faits, un peu moins trivial mais malgré tout très simple et très efficace.

4.1 La structure d'arbre discriminatoire

La notion d'*arbre discriminatoire* a été introduite par D.E. Knuth dans [5] sous le terme de "trie". Dans le cadre de notre application, un tel arbre fournit une solution pour mémoriser des chaînes de caractères en factorisant les parties initiales. Chaque arc de l'arbre est étiqueté par un caractère composant la chaîne et les chemins dans l'arbre, en partant de la racine, constituent des représentations des chaînes que l'on désire mémoriser.

Pour définir les arbres discriminatoires dont nous avons besoin dans cette application, nous avons repris la distinction entre racine et terminaison introduite dans la section 2.2. Nous constituons donc deux arbres discriminatoires : un pour les racines des mots et un autre pour les terminaisons. Le premier est construit essentiellement à l'aide des entrées lexicales. Une consultation des tables de terminaisons est cependant nécessaire pour déterminer la sous-chaîne racine à partir de la forme de citation. Le deuxième arbre discriminatoire est construit exclusivement sur la base des tables de terminaisons et des tables auxiliaires.

Dans la suite de ce document, nous allons surtout mettre l'accent sur la description et l'utilisation de l'arbre discriminatoire des racines. Comme il est aisé d'appliquer les mêmes techniques à l'arbre des terminaisons, une explication détaillée de ce dernier serait redondante et inutile.

Ainsi, par exemple, la Figure 9 représente graphiquement l'arbre discriminatoire des racines des mots "aimer", "abri", "cheval" et "maison". Les racines sont respectivement "aim", "abri", "cheva" et "maison". Dans notre conception d'un arbre discriminatoire, nous permettons aussi de mettre des étiquettes sur les noeuds. En particulier, la Figure 9 a des étiquettes sur les noeuds feuille. Il s'agit ici d'une description d'une structure d'attributs présentée sous forme de prédicats d'accès tels qu'ils apparaissent dans le corps de la clause d'une entrée lexicale.

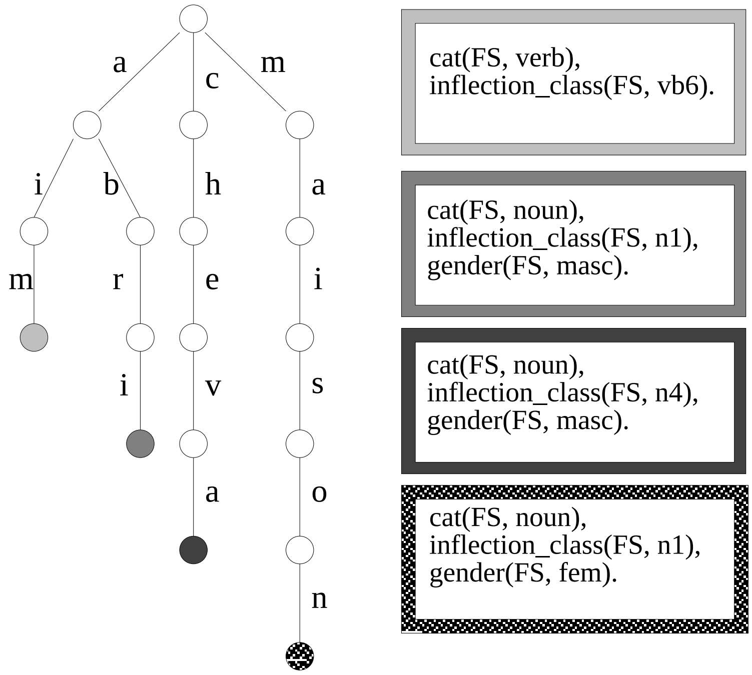

Figure 9 Représentation graphique de l'arbre discriminatoire des racines des mots "aimer", "abri", "cheval" et "maison".

La génération de l'arbre discriminatoire des racines et de celui des terminaisons est réalisée automatiquement à l'aide d'un *compilateur de dictionnaires*. Le code de ce programme ne sera pas présenté ici car cela sort du sujet de ce document. Nous allons par contre étudier les résultats produits par ce compilateur, ou plutôt ces compilateurs, puisque chaque version d'analyseur morphologique dépend d'une version compilée différente du dictionnaire donc produit par un compilateur distinct. Une raison supplémentaire de ne pas présenter ici le code de ces programmes est qu'il est très aisé de le construire à partir de la spécification qu'en constituent les données (le dictionnaire source) et les résultats (le dictionnaire compilé).

4.2 L'implantation active d'un arbre discriminatoire

Cette implantation d'un arbre discriminatoire adopte une solution dans laquelle chaque clause décrit un arc en mettant en relation un noeud "père" avec le sous-arbre situé sous le noeud "fils". La dénomination "active" a été choisie pour mettre en valeur le fait que l'implantation est réalisée à l'aide de clauses Prolog complètes et que c'est l'exécution d'une de ces clauses qui conduit à l'exécution des suivantes.

```
'$root'(1, [0'a|S0]-S1, FS) :-
        '$root'(2, S0-S1, FS).
'$root'(2, [0'i|S0]-S1, FS) :-
        '$root'(3, S0-S1, FS).
'$root'(3, [0'm|S0]-S1, fs(verb, AGR, VERBAL, form(vb6,aimer))) :-
        '$ending'(1, S0-S1, fs(verb, AGR, VERBAL, form(vb6,aimer))).
'$root'(2, [0'b|S0]-S1, FS) :-
        '$root'(4, S0-S1, FS).
'$root'(4, [0'r|S0]-S1, FS) :-
        '$root'(5, S0-S1, FS).
'$root'(5, [0'i|S0]-S1, fs(noun, agr(masc,N,P), VERBAL, form(n1,abri))) :-
```

```
        '$ending'(1, S0-S1, fs(noun, agr(masc,N,P), VERBAL, form(n1,abri))).
'$root'(1, [0'c|S0]-S1, FS) :-
        '$root'(6, S0-S1, FS).
'$root'(6, [0'h|S0]-S1, FS) :-
        '$root'(7, S0-S1, FS).
'$root'(7, [0'e|S0]-S1, FS) :-
        '$root'(8, S0-S1, FS).
'$root'(8, [0'v|S0]-S1, FS) :-
        '$root'(9, S0-S1, FS).
'$root'(9, [0'a|S0]-S1, fs(noun, agr(masc,N,P), VERBAL, form(n4,cheval))) :-
        '$ending'(1, S0-S1, fs(noun, agr(masc,N,P), VERBAL, form(n4,cheval))).
'$root'(1, [0'm|S0]-S1, FS) :-
        '$root'(10, S0-S1, FS).
'$root'(10, [0'a|S0]-S1, FS) :-
        '$root'(11, S0-S1, FS).
'$root'(11, [0'i|S0]-S1, FS) :-
        '$root'(12, S0-S1, FS).
'$root'(12, [0's|S0]-S1, FS) :-
        '$root'(13, S0-S1, FS).
'$root'(13, [0'o|S0]-S1, FS) :-
        '$root'(14, S0-S1, FS).
'$root'(14, [0'n|S0]-S1, fs(noun, agr(fem,N,P), VERBAL, form(n1,maison))) :-
        '$ending'(1, S0-S1, fs(noun, agr(fem,N,P), VERBAL, form(n1,maison))).
```

Figure 10 Implantation de l'arbre discriminatoire de la Figure 9 sous la forme de clauses Prolog.

Le prédicat `'$root'/3` de la Figure 10 est défini en précisant dans la tête de la clause quel caractère doit se trouver en tête de la chaîne de caractères (la première partie du 2ème argument) à partir d'un noeud donné dans l'arbre (le 1er argument). La deuxième partie du 2ème argument, ``-S1'', s'unifie avec la partie de la chaîne de caractères qui ne sera pas traitée dans ce parcours d'arbre, c'est-à-dire, si le traitement est correct, avec les mots suivants le mot que nous sommes en train de reconnaître. Le 3ème argument s'unifie avec la structure d'attributs construite par l'aboutissement à une feuille de l'arbre des racines. Le corps de `'$root'/3` spécifie à quel sous-arbre via le noeud qui le domine, est délégué le traitement de la suite de la chaîne.

La forme des clauses décrivant les arcs qui débouchent sur les feuilles de l'arbre est différente, comme l'illustre la 3ème définition de `'$root'/3` dans la Figure 10. Plutôt que d'introduire des buts comme ceux qui figurent dans les rectangles de la Figure 9, nous avons introduit une optimisation en effectuant une évaluation partielle de ces prédicats lors de la construction de l'arbre ce qui explique que le 3ème argument est un terme `fs/4` partiellement instancié. Cette technique permet un gain sur deux niveaux. Premièrement, la lisibilité des programmes et la séparation entre définition et utilisation d'une structure de données sont améliorées en passant par des prédicats d'accès. Deuxièmement, les performances du programme ne sont pas pénalisées puisque ces prédicats sont exécutés lors de la “compilation” des dictionnaires. Le corps d'une telle clause délègue le traitement de la suite de la chaîne à l'arbre des terminaisons `'$ending'/3`. La racine de cet arbre des terminaisons a par définition la valeur `1`, d'où la valeur du premier argument de but `'$ending'/3`.

Dans le but de faire le tour de la question, la Figure 11 résume très succintement la forme de l'arbre discriminatoire des terminaisons qui s'intègre avec l'arbre des racines de la Figure 10.

```
'$ending'(1, S-S, fs(_CAT, agr(_G,sing,_P), _VERBAL, form(n1,_))).
'$ending'(1, [0's|S]-S, fs(_CAT, agr(_G,plur,_P), _VERBAL, form(n1,_))).
```

Figure 11 Transcription en clauses Prolog du sous-arbre des terminaisons généré par la définition de la classe flexionnelle `n1` de la Figure 5.

Ce sous-arbre de terminaisons illustre la manière simple de garantir que la terminaison qui est analysée est rattachée à la classe flexionnelle déterminée lors de l'analyse de la racine.

4.3 La version active de l'analyseur morphologique

L'analyseur morphologique est défini par le prédicat `morpho/3` ainsi que par une série de prédicats auxiliaires.

```
morpho(String0, FS, String3) :-
        skip_spaces(String0, String1),
        optional_lowercase(String1, String2),
        word_selection(String2, FS, String3).

word_selection(String0, FS, String1) :-
        '$root'(1, String0-String1, FS),
        end_of_word(String1).

optional_lowercase(String, String).
optional_lowercase([Ch_upper|String], [Ch_lower|String]) :-
        lowercase(Ch_upper, Ch_lower). % définition de lowercase/2 absente !

skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères de ponctuation, pour cette application
        punctuation(Ch), % définition of punctuation/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères TAB, "espace" et CR
        effective_space(Ch), % définition of effective_space/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces(String, String).

end_of_word([]).
end_of_word([Ch | _String]) :-
        punctuation(Ch).
end_of_word([Ch | _String]) :-
        effective_space(Ch).
```

Figure 12 Un analyseur morphologique utilisant un arbre discriminatoire actif.

Seule la définition du prédicat `word_selection/3` de la Figure 12 change par rapport à la définition de la Figure 6 et mérite quelques explications :

`'$root'(1, String0-String1, FS)` – *Parcours de l'arbre discriminatoire des racines* – Le fait de spécifier la valeur `''1''` comme noeud de départ, assure que le parcours débute sur la racine de l'arbre. Comme l'arbre des terminaisons est automatiquement connecté à l'arbre des racines, il n'est pas nécessaire de mentionner explicitement un accès à cet arbre ici.

`end_of_word(String1)` – *Test de la fin d'un mot* – Comme les mots de la langue naturelle ne constituent pas un "langage minimal" au sens de la théorie des langages de programmation [1], il est indispensable de tester la présence d'un caractère qui marque une fin de mot.

Les résultats fournis par cette méthode sont extraordinairement plus rapides. La Figure 13 est révélatrice de cet état de faits. L'amélioration est d'un rapport supérieur à 500. La raison principale se trouve dans la conformité aux techniques d'indexation des prédicats adoptées par Prolog lors de la compilation des programmes, la clé d'indexation primaire étant le nom du prédicat et la clé secondaire le foncteur du premier argument. Ainsi lorsque les clauses d'un même prédicat se distinguent sur la valeur du premier argument et que le premier argument du but est instancié à une de ces valeurs, l'accès à la bonne clause est direct. C'est ce que nous garantissons en faisant figurer la valeur numérique du noeud comme premier argument de `'$root'/3` et de `'$ending'/3`.

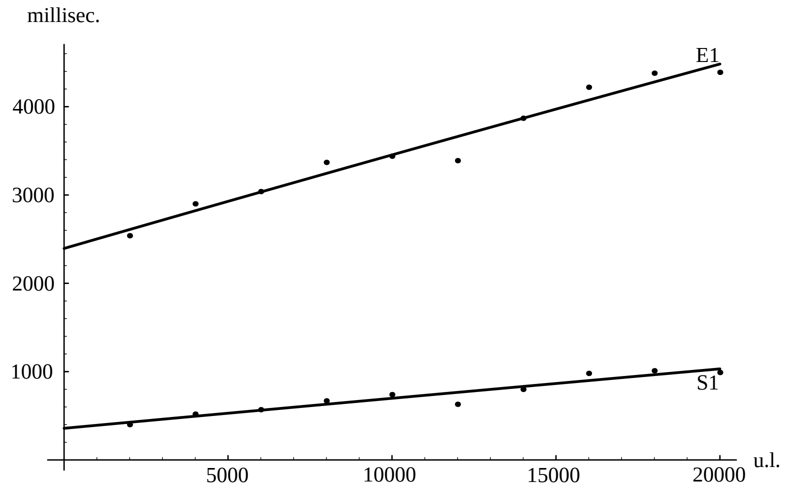

Figure 13 Courbes des durées en millisecondes, de l'analyse sérielle **S1** et exhaustive **E1** en fonction de la taille du dictionnaire en unités lexicales (**u.l.**).

Évidemment un tel progrès se paie. Le prix à payer est ici une augmentation sensible du nombre de clauses (cf. Figure 14) et une augmentation tout aussi importante de la taille du dictionnaire transcrit sous forme d'arbre discriminatoire.

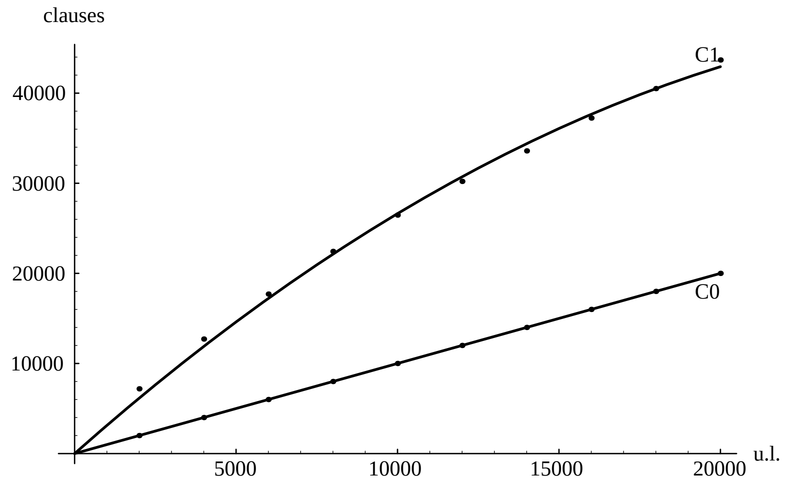

Figure 14 Courbes comparatives du nombre de clauses nécessaires au stockage du dictionnaire — **C0** pour la version simpliste, **C1** pour la version courante.

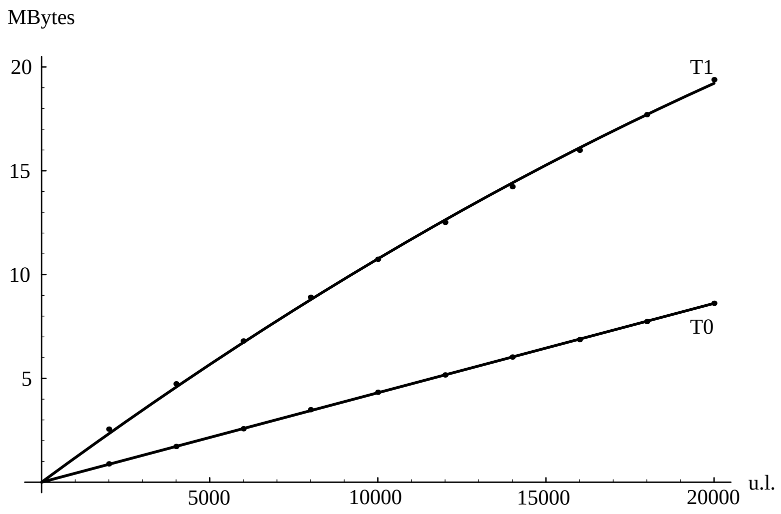

Figure 15 Courbes comparatives de la taille des dictionnaires — **T0** pour la version simpliste, **T1** pour la version courante.

4.4 L'implantation statique d'un arbre discriminatoire

Afin de réduire l'encombrement de l'arbre discriminatoire généré par la version "active", nous proposons ici une solution que nous appelons "statique" car l'arbre discriminatoire n'est constitué que de *faits* Prolog. L'interprétation de ces faits est laissée à l'analyseur morphologique.

```
'$root'(1, 0'a, 2, _FS).
'$root'(2, 0'i, 3, _FS).
'$root'(3, 0'm, 0, fs(verb, _AGR, _VERBAL, form(vb6,aimer))).
'$root'(2, 0'b, 4, _FS).
'$root'(4, 0'r, 5, _FS).
'$root'(5, 0'i, 0, fs(noun, agr(masc,_N,_P), _VERBAL, form(n1,abri))).
'$root'(1, 0'c, 6, _FS).
'$root'(6, 0'h, 7, _FS).
'$root'(7, 0'e, 8, _FS).
'$root'(8, 0'v, 9, _FS).
'$root'(9, 0'a, 0, fs(noun, agr(masc,_N,_P), _VERBAL, form(n4,cheval))).
'$root'(1, 0'm, 10, _FS).
'$root'(10, 0'a, 11, _FS).
'$root'(11, 0'i, 12, _FS).
'$root'(12, 0's, 13, _FS).
'$root'(13, 0'o, 14, _FS).
'$root'(14, 0'n, 0, fs(noun, agr(fem,_N,_P), _VERBAL, form(n1,maison))).
```

Figure 16 Implantation de l'arbre discriminatoire de la Figure 9 sous la forme de faits Prolog.

Le prédicat `'$root'/4` de la Figure 16 définit une relation extrêmement simple entre deux noeuds — le noeud père comme 1$^{\text{er}}$ argument et le noeud fils comme 3$^{\text{ème}}$ argument — tout en spécifiant le caractère qui joue le rôle d'étiquette sur l'arc — le 2$^{\text{ème}}$ argument. Les noeuds feuille sont repérés par la valeur `''0''`. De nouveau, les buts définis par les prédicats d'accès ont été évalués à la compilation.

4.5 La version statique de l'analyseur morphologique

La version "statique" de l'analyseur morphologique reprend en grande partie l'organisation des deux premières versions. La seule différence notable est la définition d'un *interprète* des relations définies à l'aide de `'$root'/4`.

```
morpho(String0, FS, String3) :-
        skip_spaces(String0, String1),
        optional_lowercase(String1, String2),
        word_selection(String2, FS, String3).

word_selection(String0, FS, String1) :-
        root_trie_run(1, String0, FS, String1),
        end_of_word(String1).

root_trie_run(Father, [Char|Next], FS, String) :-
        % cas 1 : traitement d'un arc interne
        '$root'(Father, Char, Son, FS),
        root_trie_run(Son, Next, FS, String).
root_trie_run(Father, [Char|Ending], FS, String) :-
        % cas 2 : traitement d'un arc débouchant sur une feuille
        '$root'(Father, Char, 0, FS),
        ending_trie_run(1, Ending, FS, String).
root_trie_run(1, Ending, FS, String) :-
        % cas 3 : traitement d'une racine vide
        '$root'(1, [], 0, FS),
        ending_trie_run(1, Ending, FS, String).

ending_trie_run(0, String, _FS, String).
ending_trie_run(1, String, FS, String) :-
        '$ending'(FS).
ending_trie_run(Father, [Char|Next0], FS, String) :-
        '$ending'(Father, Char, Next0-Next1, Son, FS),
        ending_trie_run(Son, Next1, FS, String).

optional_lowercase(String, String).
optional_lowercase([Ch_upper|String], [Ch_lower|String]) :-
        lowercase(Ch_upper, Ch_lower). % définition de lowercase/2 absente !

skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères de ponctuation, pour cette application
        punctuation(Ch), % définition of punctuation/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères TAB, "espace" et CR
        effective_space(Ch), % définition of effective_space/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces(String, String).

end_of_word([]).
```

```
end_of_word([Ch | _String]) :-
        punctuation(Ch).
end_of_word([Ch | _String]) :-
        effective_space(Ch).
```

Figure 17 Un analyseur morphologique utilisant un arbre discriminatoire statique.

La définition du prédicat `word_selection/3` de la Figure 17 diffère très légèrement de celle de la version précédente :

`root_trie_run(1, String0, FS, String1)` – *Parcours de l'arbre discriminatoire des racines* – Le fait de spécifier la valeur `''1''` comme noeud de départ, assure que le parcours débute sur la racine de l'arbre. Comme l'arbre des terminaisons est automatiquement connecté à l'arbre des racines, il n'est pas nécessaire de mentionner explicitement un accès à cet arbre ici.

`end_of_word(String1)` – *Test de la fin d'un mot* – Comme les mots de la langue naturelle ne constituent pas un "langage minimal" au sens de la théorie des langages de programmation [1], il est indispensable de tester la présence d'un caractère qui marque une fin de mot.

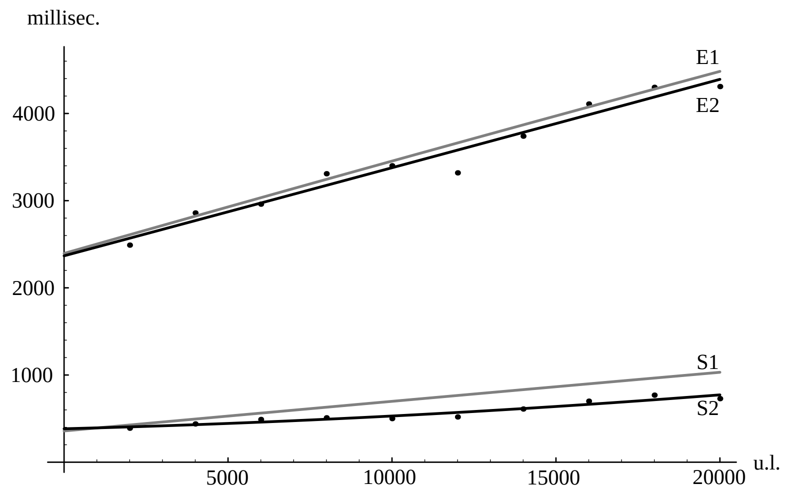

Figure 18 Courbes des durées en millisecondes, de l'analyse sérielle **S2** et exhaustive **E2** en fonction de la taille du dictionnaire en unités lexicales (**u.l.**), mises en perspective avec **S1** et **E1** de la Figure 13.

Le prédicat `root_trie_run/4` de la Figure 17 définit un parcours de l'arbre des racines en décrivant trois cas particuliers :

cas 1 : traitement d'un arc interne – Le caractère initial de la chaîne en entrée, `Char`, doit être l'étiquette d'un arc dont le noeud père s'unifie avec `Father`. Le résultat obtenu est le noeud fils `Son` et la procédure est appelée récursivement sur la chaîne `Next` avec `Son` comme noeud père.

cas 2 : traitement d'un arc débouchant sur une feuille – Lorsque l'arc débouche sur une feuille — valeur `''0''` pour le noeud fils — le traitement de la suite de la chaîne est délégué à `ending_trie_run/4` pour un parcours de l'arbre des terminaisons.

cas 3 : traitement d'une racine vide – Une racine vide est, tout comme une terminaison vide, un cas limite tout à fait acceptable[3]. Les solutions pour coder ces cas limites dans les arbres sont différentes

[3] Les verbes "avoir" et "être" sont des cas particuliers avec des racines vides, car leurs formes sont tellement irrégulières qu'aucune sous-chaîne initiale n'est stable.

mais absolument équivalentes. Pour les racines nous avons choisi de garder une uniformité avec les autres clauses `'$root'/4`. Pour les terminaisons, nous avons choisi de coder ce cas particulier à l'aide d'un autre prédicat `'$ending'/1`.

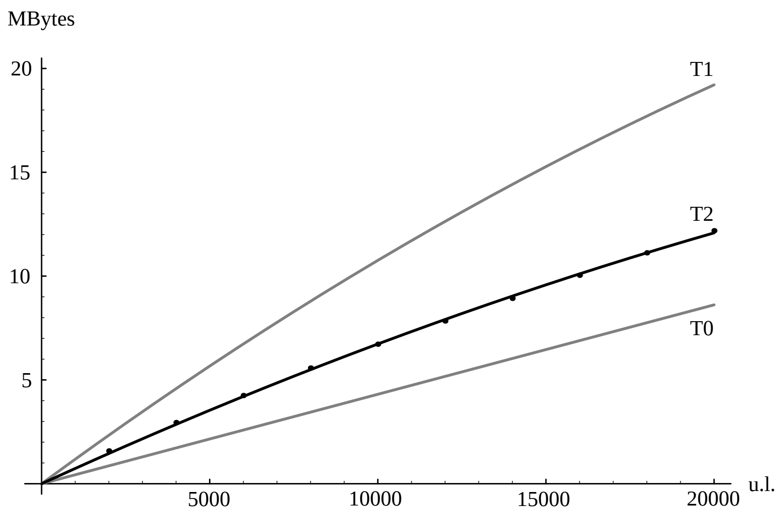

Figure 19 Courbes comparatives de la taille des dictionnaires — **T0** pour la version simpliste, **T1** pour la version "arbre actif", **T2** pour la version "arbre statique".

Les performances de cette méthode sont sensiblement équivalentes à celles de la solution active. Elles sont même légèrement meilleures (cf. Figure 18) sans qu'il soit aisé d'en expliquer les raisons, car l'arbre discriminatoire est tout à fait identique. La seule différence majeure, nous la verrons après, est la diminution de la taille du dictionnaire.

Cette solution d'arbre discriminatoire statique génère exactement le même nombre de clauses que la solution d'arbre discriminatoire actif puisque le principe de chacune des deux techniques est de construire une clause pour chaque arc de l'arbre. Par contre la taille du dictionnaire est bien moins importante avec cette solution comme le démontre la Figure 19.

5 Un analyseur morphologique mixte

Finalement la solution la plus intéressante qui a aussi l'avantage de diminuer considérablement le nombre de clauses du dictionnaire, consiste à adopter un moyen terme entre les deux approches précédentes, en introduisant une optimisation qui affecte la taille de l'arbre généré. Cette optimisation est illustrée par la Figure 20. Le principe est : toute portion de chemin dans l'arbre qui ne contient pas de "bifurcation" peut être réduite à un seul arc. C'est ainsi que la totalité du chemin qui code "maison" dans la Figure 9 est réduit à un seul arc.

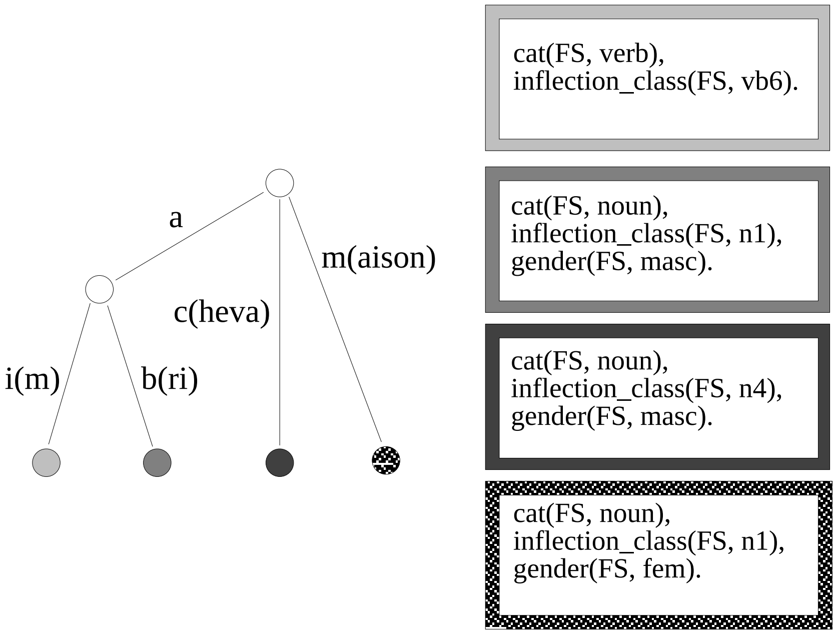

Figure 20 Représentation graphique de l'arbre discriminatoire optimisé des racines des mots "aimer", "abri", "cheval" et "maison".

Dans la Figure 20, l'arc est toujours étiqueté par un caractère, qui reste le critère de sélection de l'arc. Les autres caractères qui figuraient comme étiquettes des arcs qui ont été éliminés, sont mentionnés entre parenthèses et seront simplement retranchés de la chaîne de caractères durant l'étape d'identification de l'arc.

5.1 L'implantation de l'arbre discriminatoire

L'implantation de cet arbre discriminatoire a beaucoup de similitudes avec l'arbre statique de la solution précédente. Cette solution génère cependant des clauses `'$root'/5` avec un argument supplémentaire — le troisième — qui permet de traiter à l'aide d'une unification les caractères qui doivent être retranchés de la chaîne de caractères à analyser, selon la technique des "difference-lists" décrite dans [11].

```
'$root'(1, 0'a, S-S, 2, _FS).
'$root'(2, 0'i, [0'm|S]-S, 0, fs(verb, _AGR, _VERBAL, form(vb6,aimer))).
'$root'(2, 0'b, [0'r, 0'i|S]-S, 0, fs(noun, agr(masc,_N,_P), _VERBAL, form(n1,abri))).



'$root'(1, 0'c, [0'h, 0'e, 0'v, 0'a|S]-S, 0,
        fs(noun, agr(masc,_N,_P), _VERBAL, form(n4,cheval))).
'$root'(1, 0'm, [0'a, 0'i, 0's, 0'o, 0'n|S]-S, 0,
        fs(noun, agr(fem,_N,_P), _VERBAL, form(n1,maison))).
```

Figure 21 Implantation de l'arbre discriminatoire de la Figure 20 sous forme mixte

La définition de `'$root'/5` ne nécessite pas de commentaire particulier tant elle est proche de celle de `'$root'/4`. Notons cependant le rôle joué par le 3ème argument qui permet d'unifier "`S`" avec la suite d'une chaîne de caractères débutant par les caractères spécifiés dans la première partie de l'argument.

5.2 La version mixte de l'analyseur morphologique

Dans la mesure où la structure des relations `'$root'/5` est très similaire à celle des relations `'$root'/4`, il n'est pas surprenant que l'analyseur morphologique lié à cette nouvelle solution soit aussi proche de celui de la version précédente.

```
morpho(String0, FS, String3) :-
        skip_spaces(String0, String1),
        optional_lowercase(String1, String2),
        word_selection(String2, FS, String3).

word_selection(String0, FS, String1) :-
        root_trie_run(1, String0, FS, String1),
        end_of_word(String1).

root_trie_run(Father, [Char|Next0], FS, String) :-
        '$root'(Father, Char, Next0-Next1, Son, FS),
        root_trie_run(Son, Next1, FS, String).
root_trie_run(Father, [Char|Next], FS, String) :-
        '$root'(Father, Char, Next-Ending, 0, FS),
        ending_trie_run(1, Ending, FS, String).
root_trie_run(1, Ending, FS, String) :-
        '$root'(FS),
        ending_trie_run(1, Ending, FS, String).

ending_trie_run(0, String, _FS, String).
ending_trie_run(1, String, FS, String) :-
        '$ending'(FS).
ending_trie_run(Father, [Char|Next0], FS, String) :-
        '$ending'(Father, Char, Next0-Next1, Son, FS),
        ending_trie_run(Son, Next1, FS, String).

optional_lowercase(String, String).
optional_lowercase([Ch_upper|String], [Ch_lower|String]) :-
        lowercase(Ch_upper, Ch_lower). % définition de lowercase/2 absente !

skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères de ponctuation, pour cette application
        punctuation(Ch), % définition of punctuation/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces([Ch | String0], String1) :-
        % On élimine les caractères TAB, "espace" et CR
        effective_space(Ch), % définition of effective_space/1 absente !
        !,
        skip_spaces(String0, String1).
skip_spaces(String, String).
```

```
end_of_word([]).
end_of_word([Ch | _String]) :-
        punctuation(Ch).
end_of_word([Ch | _String]) :-
        effective_space(Ch).
```

Figure 22 Un analyseur morphologique utilisant un arbre discriminatoire mixte

Seule la définition de `root_trie_run/4` est différente afin de tenir compte du nouvel argument de `'$root'/5`.

Les performances de cette solution sont sensiblement meilleures en raison principalement du moins grand nombre d'étapes à parcourir pour atteindre la bonne feuille de l'arbre des racines (cf. Figure 23). Ce résultat est très intéressant car dans le cas de **S3** qui caractérise la première solution de l'analyseur morphologique on peut presque considérer que l'analyseur est insensible à la taille du dictionnaire.

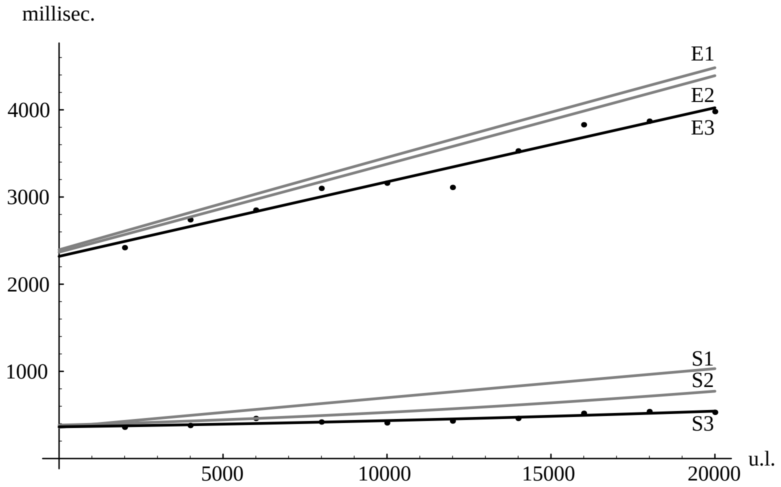

Figure 23 Courbes des durées en millisecondes, de l'analyse sérielle **S3** et exhaustive **E3** en fonction de la taille du dictionnaire en unités lexicales (**u.l.**), mises en perspective avec **S1**, **S2** et **E1**, **E2**.

La diminution du nombre de clauses est une autre particularité de cette solution (cf. Figure 24). Ce résultat reste encore, malgré tout, bien en dessus du nombre de clauses du dictionnaire source même si le saut quantitatif d'avec les deux autres arbres discriminatoires est important.

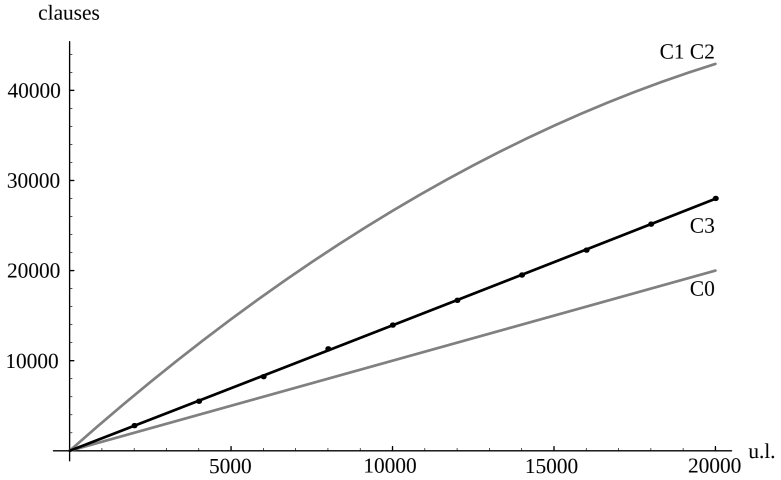

Figure 24 Courbes comparatives du nombre de clauses nécessaires au stockage du dictionnaire — **C0** pour la version simpliste, **C1 C2** pour les versions active et statique et **C3** pour la version mixte.

La taille physique sur disque, nécessaire au stockage du dictionnaire suit aussi une très nette tendance à la baisse. Le résultat dans ce cas (cf. Figure 25) est très intéressant parce que très proche de celui du dictionnaire source. Une raison probable de cette très faible différence, alors que le nombre de clauses reste malgré tout largement supérieur, est le fait que les structures d'attributs sont partiellement évaluées dans le cas présent alors qu'elles sont décrites sous forme de prédicats d'accès dans le dictionnaire source.

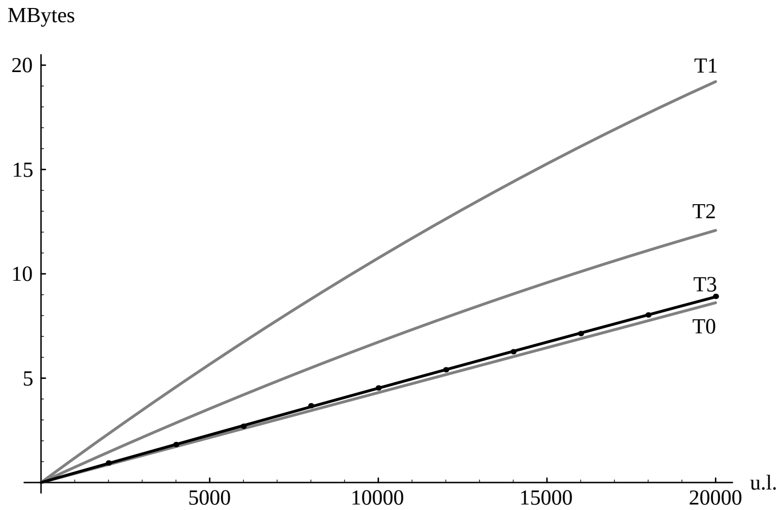

Figure 25 Courbes comparatives de la taille des dictionnaires — **T0** pour le dictionnaire original, **T1** pour la version "arbre actif", **T2** pour la version "arbre statique" et **T3** pour la version "arbre mixte".

6 Conclusion

Nous avons donc réusssi par cette solution "mixte" à améliorer considérablement l'efficacité de l'analyseur morphologique en réduisant au maximum les effets auxiliaires mis en évidence avec les premières implantations des arbres discriminatoires. Il faut cependant noter que cette solution n'est pas une fin en soi. Il s'agit plutôt d'un point de départ pour une réalisation concrète d'un analyseur morphologique. C'est dans cet esprit que ce travail a été complété dans le cadre du projet des Archives [2]. Nous pouvons donc affirmer que la solution "mixte" est suffisamment générale pour s'adapter aux problèmes concrets que l'on rencontre dans des textes réels.

6.1 Les variations orthographiques et typographiques

Un problème réel rencontré dans les textes est celui de l'initiale majuscule en début de phrase. Le traitement de ce phénomène orthographique fait déjà partie des exemples d'analyseurs morphologiques fournis ci-dessus (prédicat `optional_lowercase/2`). D'autres phénomènes typographiques comme la mise en majuscules de certains mots ou l'espacement des lettres d'un même mot — ce sont des techniques de mise en évidence héritées des cours de dactylographie — peuvent être traités par des prédicats ad hoc.

La césure, qui est le phénomène le plus courant et le plus ambigu dans les textes concrets, peut être traitée intégralement au niveau des deux arbres, celui des racines et celui des terminaisons. La solution est simple dans le cas d'un arbre statique. Elle consiste simplement pour les racines, par exemple, à introduire quelques définitions supplémentaires de `'$root'/4`, à savoir :

```
'$root'(N, 0'-, x+N, _FS).
'$root'(x+N, 10, N, _FS).
'$root'(N, 10, N, _FS).
```

La première et la deuxième définition de `'$root'/4`, qui vont ensemble, permettent de traiter la séquence ''-<CR>'' qui constitue la césure en restituant ensuite la position N dans l'arbre, c'est-à-dire la position d'avant le tiret. Ce petit "détour" permet ainsi de comparer la chaîne courante avec la chaîne de référence qui est forcément sans césure. La troisième définition permet de traiter le cas où la césure est placée sur un trait d'union, donc seul le <CR> doit être analysé séparément.

Pour le cas d'un arbre mixte qui est la solution que nous avons retenue, la transformation pour la gestion de la césure est un peu plus complexe car il ne suffit pas simplement d'ajouter trois définitions de `'$root'/5` comparables à celles de `'$root'/4`; cela ne règlerait qu'une partie des cas de figure. Comme la définition d'un arc permet de traiter plusieurs caractères à la fois, si la césure est faite au milieu d'un tel groupe de caractères, l'unification ne se fera pas et la chaîne ne sera pas reconnue. Il faut donc transformer la tête des prédicats `'$root'/5` en laissant le 3ème argument non instancié et en déléguant à un prédicat auxiliaire, `substring_unification/2` par exemple, la tâche de réduire partiellement la chaîne de caractères. La deuxième définition de `'$root'/5` dans la Figure 21 prend alors la forme suivante :

```
'$root'(2, 0'i, S0-S1, 0, fs(verb, _AGR, _VERBAL, form(vb6,aimer))) :-
        substring_unification([0'm|S]-S, S0-S1).
```

Et la définition de `substring_unification/2` est la suivante :

```
substring_unification(S0-S1, S0-S1) :- !.
        % reproduit l'unification sur l'argument
substring_unification([0'-, C|S0]-S1, [0'-, 10, C|T0]-T1) :-
        % la chaîne de référence contient un tiret et la
        % césure se fait sur ce tiret.
        substring_unification(S0-S1, T0-T1).
substring_unification([C|S0]-S1, [0'-, 10, C|T0]-T1) :-
        % pas de tiret dans la chaîne de référence
        substring_unification(S0-S1, T0-T1).
substring_unification([C|S0]-S1, [C|T0]-T1) :-
        % simule le traitement caractère par caractère
        substring_unification(S0-S1, T0-T1).
```

Évidemment cette solution a des incidences sur les performances de l'analyseur morphologique car elle introduit de l'indéterminisme et des étapes supplémentaires dans la vérification de la relation `morpho/3`. Mais la pénalisation n'est pas très importante et il est plus intéressant d'avoir une solution qui soit cohérente avec le principe de ne pas faire de segmentation à priori de la chaîne de caractères en mots.

6.2 Les extensions non décrites

Afin de compléter le panorama des améliorations que nous avons apportées au système "mixte" pour le rendre utilisable dans un environnement de textes réels, nous allons évoquer les problèmes que nous avons résolus. La description détaillée des techniques mises en oeuvre sera fournie dans un autre document en cours de rédaction sur l'analyseur morphologique du projet des Archives.

Deux problèmes linguistiques, l'élision et la contraction, ont été considérés et une solution satisfaisante a été apportée en traitant ces deux problèmes comme des cas particuliers de classes flexionnelles. Cette technique permet même de traiter de manière très générale des cas de contractions comme "duquel" ou "desquelles" simplement en précisant que "du" est la contraction de "de" et "le", que "des" peut être la contraction de "de" et "les" et que "lequel" est une unité connue du dictionnaire.

Un autre point nécessaire au bon fonctionnement d'un analyseur morphologique est sa capacité de traiter correctement les mots composés. À cet effet, nous avons construit des classes flexionnelles spécialisées qui décrivent les variations flexionnelles des mots simples qui constituent un mot composé. En pratique, les variations flexionnelles ne s'appliquent qu'aux noms composés; pour le reste, les formes composées adverbiales ou prépositionnelles, nous sommes en présence de formes invariables. Le cas des formes invariables est très simple à gérer puisque l'analyseur morphologique ne nécessite aucune segmentation préalable en mots simples. Il n'y a donc aucune limitation à la chaîne de caractères codée dans le dictionnaire. En particulier, elle peut très bien contenir des espaces qui sont normalement considérés comme des séparateurs de mots.

Références

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 2: Parsing. Prentice-Hall, 1972.

[2] Jean-Luc Cochard, Michael Hess, and Andreas Kellerhals. Specification and prototyping of a system for the intelligent management of information (status report). June 1991.

[3] M. Dalrymple, R.M. Kaplan, L. Karttunnen, K. Koskenniemi, S. Shaio, and M. Wescoat. Tools for morphological analysis. Report CSLI–87–108, Center for the Study of Language and Information, September 1987.

[4] Foma, Lausanne, editor. *Le Nouveau Bescherelle, 1. L'art de conjuguer.* Hatier, Paris, 1980.

[5] Donald E. Knuth. *The Art of Computer Programming*, volume 3, chapter Sorting and Searching. Addison-Wesley, Reading MA, 1973.

[6] Kimmo Koskenniemi. Two-level morphology: a general computational model for word-form recognition and production. Publication 11, Dept of General Linguistics, University of Helsinki, 1983.

[7] Gerard Mellish and Chris Gazdar. *Natural Language Processing in Prolog.* Addison-Wesley, 1989.

[8] Fernando C.N. Pereira and Stuart M. Shieber. Prolog and natural-language analysis. Lecture Notes 10, Center for the Study of Language and Information, 1987.

[9] Peter Ross. *Advanced Prolog, Techniques and Examples.* Addison-Wesley, 1989.

[10] Michel Simard. Integrating of finite-state morphological models to top-down syntactic parsers in Prolog. Master's thesis, School of Computer Science, McGill University, Montreal, March 1990.

[11] Leon Sterling and Ehud Shapiro. *The Art of Prolog, Advanced Programming Techniques*, chapter Incomplete Data Structures. The MIT Press, 1986.