Using regular tree grammars to optimise surface realisation

A dissertation submitted to
the Faculty of Computer Science,
Free University of Bozen-Bolzano
in fulfillment of the degree of

Masters of Science in Computer Science
as part of the
European Masters Program in Language and
Communicaton Technologies

Submitted by:
Laura Haide Perez

Supervisors:
Dr. Rafaella Bernardi
Dr. Claire Gardent

July 2009

Acknowledgements

First of all, I would like to thank my supervisors Raffaella Bernardi and Claire Gardent for their great support and expert guidance. Not only in this thesis work but also along this two years of studies.

I am immensely grateful to my parents and brother who have always encouraged me in every single thing I took up. Without their unconditional support and love I could not have made this far. I want to thank Yann because of staying by my side supporting me and making me happy.

Laura

Abstract

Surface realisation is the task within Natural Language Generation responsible for mapping an *abstract linguistic structure* (e.g., a logical formula) into one or more sentence(s). Particularly, reversible surface realisers, i.e. surface realisers that use the same reversible grammar for both parsing and generation, generally take as input a flat logical formulae. This kind of realisers present a number of important advantages but at the same time require some optimisations to overcome the complexity introduced due to the generation from flat semantics. To remedy this shortcoming, various optimisations have been proposed. In particular, filtering techniques have been developed which, on some given criterion, filter out from the initial search space all candidate solutions which cannot possibly lead to valid ones.

This thesis work aims to optimise such a realiser, specifically, we aim to reduce the initial search space by reducing the effects of one of the sources of complexity namely, *lexical ambiguity.* The surface realiser we focus on, is based on a wide-coverage *Feature Based Lexicalized Tree Adjoining Grammar* extended with a compositional semantics. The realiser furthermore integrates a polarity filter. Although it was shown to drastically reduce the initial search space, polariy filtering still has some limitations. In this thesis, we investigate an alternative filtering technique based on a novel formal framework for modeling polarity counts, namely *Feature Based Regular Tree Grammar.* The underlying motivation for this work is that the derivation trees of a Tree Adjoining Grammar can be encoded as the language of a Regular Tree Grammar. Thus, we can model polarities (the syntactic resources and requirements associated with a given TAG tree) based on all the information provided by the elementary trees. Furthermore, detecting the initial candidate combinations of trees that can not possibly yield a full derivation tree boils down to parsing with an RTG.

In this thesis, we make the RTG filtering idea precise. Next, we provide it with an implementation by developing an RTG based module for generating a parse forest from the logical formulae input to the surface realiser. Third, we compare the impact of RTG-based filtering with polarity filtering on a number of carefully selected test cases and using different metrics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Natural language generation (NLG) is seen in general as the sequence of operations needed to map information from some *non-linguistic* (e.g. raw data) into *linguistic* form (either oral or written). NLG system architectures involve different subtasks [Reiter and Dale, 2000] to accomplish this. Surface Realisation (SR) is the subtask that maps an *abstract linguistic structure* (e.g., a logical formula) into one or more sentence(s).

In this thesis, we focus on surface realisation and on how to make this generation step more efficient. More specifically, we aim to optimise a surface realiser that is based on a reversible Tree Adjoining Grammar (TAG)[Gardent and Kow, 2005, Gardent and Kow, 2007a, Gardent and Kow, 2007b].

It is known [Kay, 1996, Carroll and Oepen, 2005] that for such surface realisers and more generally, for grammar-based surface realisers whose input is a flat semantic formula, the initial search space is exponential in the number of literals contained in the input formula. One reason for this is the lack of ordering information. Contrary to parsing where the input is a string, i.e. an ordered list of words, the input to surface realisation is a set of literals. This lack of constraint on the input results in an unguided exploration of all possible combinations which in effect is exponential in the number of literals present in the input semantics. Another ground for the exponential complexity of the task is the high degree of lexical ambiguity generally allowed by the grammar. Because the grammar associates one literal with many lexical and/or grammatical structures, the number of possibilities to be explored is very high also in practice.

To remedy this shortcoming, various optimisations have been proposed. In particular, filtering techniques have been developped which, on some given criterion, filter out from the initial search space all candidate solutions which cannot possibly lead to valid ones.

One such approach is developed in [Gardent and Kow, 2005] which shows how so-called polarity filtering can be used to exclude from the initial search space all candidate solutions that cannot possibly yield a valid grammatical structure. The basic idea underlying this approach is that the primitives of the grammar (i.e., grammar rules or in the case of a Tree Adjoining Grammar, elementary trees) can be associated with a "polarity count" indicating their respective syntactic resources and requirements. This polarity count can then be used to filter out from the initial search space all initial combinations of grammatical units which cannot possibly yield a complete S tree either because a requirement remains unsatisfied (a syntactic requirement cannot be fulfilled) or because a resource cannot be used (some of the grammatical units selected by the realiser cannot be integrated into the final S tree). [Gardent and Kow, 2005] show that integrating such a polarity filtering technique into a TAG based surface realiser dramatically increases its efficiency. Similar results are provided in [Koller and Striegnitz, 2002] also for TAG and in [Carroll et al., 1999] for a surface realiser based on a Head Driven Phrase Structure Grammar.

In this thesis, we pursue the polarity filtering idea using a novel formal framework for modelling polarity counts, namely Regular Tree Grammar. As shown in [Schmitz and Le Roux, 2008], the derivation trees of a TAG are RTG (Regular Tree Grammar) trees and there exists a well defined translation of feature-based TAG to RTG. In the present work, we use this encoding to determine whether a given set of RTG-encoded TAG trees can be combined to yield a well formed TAG derivation tree. Any set of trees that does not can be filtered out (if there is no derivation tree, there can be no derived tree generated from this tree set and therefore no sentence). More specifically, we explore different types of filtering depending on how much feature (linguistic) information is preserved in the RTG trees. A first level is where only the syntactic category is preserved -this is the level at which polarity filtering was applied. A second level includes both syntactic category and semantic indices -this should a priori provide much better guidance for the filtering process since semantics information is what guides generation.

Our approach also departs from previous filtering approaches in that all TAG elementary trees are taken into account. This is in contrast to [Gardent and Kow, 2005] where auxiliary trees are not considered.

In practice, the work carried out in this thesis revolves around two main issues:

- implementing a parsing algorithm for RTG that takes into account the specific features of surface realisation (guided by semantic indices, unordered input)
- experimenting with various levels of filtering, analysing the results and comparing the results with those obtained with the polarity filtering technique presented in [Gardent and Kow, 2005].

The thesis is structured as follows. In Chapter 2 we briefly introduce the generation task, the surface realisation subtask and the main types of existing surface realisers. Furthermore, we discuss in more details the grounds underlying the computational complexity of surface realisation and summarise the various heuristics devised presented in the literature for optimising existing algorithms.

Chapter 3 introduces the background material needed to understand the rest of the thesis and presents the RTG based filtering technique we use to filter the initial search space. We start by describing the grammar formalism used by our surface realiser, namely *Feature Based Lexicalized Tree Adjoining Grammar* (FB-LTAG) enriched with a compositional semantics. We then sketch the algorithm used to generate sentences from semantic formulae together with the various optimisations already devised to increase its efficiency. In particular, we explain how *Polarity Filtering* filtering works. Finally, we introduce our RTG based proposal.

In Chapter 4, we present the RTG based parsing algorithm we will use for filtering.

Chapter 5 goes on to present the results obtained using the RTG based polarity filtering and compare these both qualitatively and quantitatively, with the results previously obtained by [Gardent and Kow, 2005].

Chapter 6 concludes with pointers for further research.

Chapter 2

Surface Realisation

This chapter gives a brief introduction to Natural Language Generation (NLG) and situates surface realisation within this broader task (section 2.1). In section 2.2, we identify two main approaches to surface realisation and give some motivation for choosing the approach we adopted. Section 2.3 then goes on to describe in more detail the type of input our realiser expects namely flat semantics formulae. Finally, section 2.4 focuses on the computational complexity of the surface realisation task. First, we give an intuitive explanation of why the task is exponential in the length of the input. Next, we summarise existing approaches that were developed in an attempt to reduce the practical complexity of the task by means of clever heuristics.

2.1 Task definition

Natural Language Generation (NLG) is usually seen as consisting of a basic pipeline of tasks as illustrated in the Figure 2.1. These tasks can roughly be divided into two parts, a strategic part which determines "what to say" with domain knowledge, and a tactical one which determines "how to say it" with linguistic knowledge. Finer-grained distinctions can also be made. Some years ago, it has become clearer that some tasks require both domain and linguistic knowledge and a new midway component was born, the microplanner. In this thesis, we assume that the surface realiser sits at the end of this pipeline (as it usually does) and receives its input from a microplanner.

The function of the Surface Realiser (SR) component is to take the text specification produced by the microplanner and convert it into text. Linguistic realisation of abstract representations, such as abstract syntax or lexicalized case frames [Reiter and Dale, 2006] generally requires the use of a *grammar*, this being a formal description of the syntactic and morphological resources available in the output language.

2.2 Two types of surface realisers

The present work contributes to the syntactic aspect of realisation, that is, making sure that the words come out in the right order in an efficient manner.

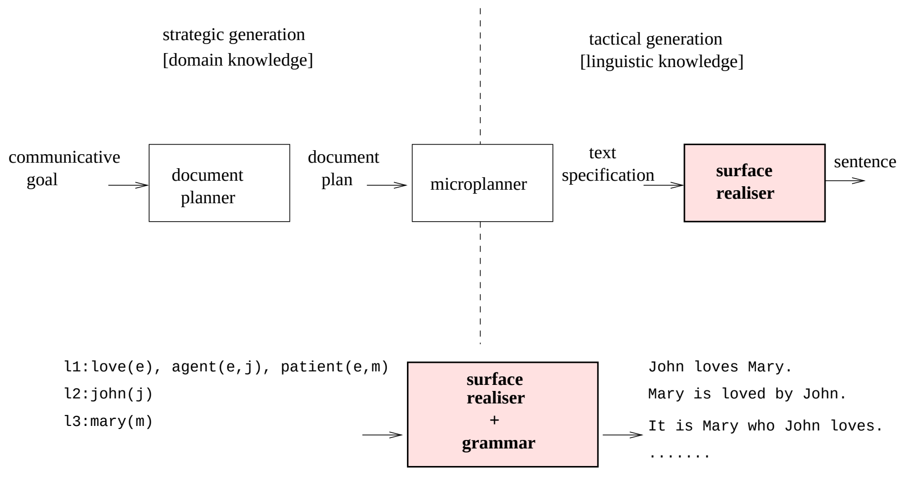


Figure 2.1: Typical Natural Language Generation pipeline

There are numerous linguistic formalisms and theories which can be incorporated into an NLG realiser. Furthermore, depending on their use, on their degree of non-determinism and on the type of grammar they assume, existing surface realisers can be divided into two main categories, namely NLG *geared realisers* and *reversible realisers*.

NLG geared realisers are meant as modules in a full-blown generation system and as such, they are constrained to be deterministic: a generation system must output exactly one text, no less, no more. In order to ensure this determinism, NLG geared realisers generally rely on theories of grammar which systematically link form to function such as systemic functional grammar (SFG, [Matthiessen et al., 1991]). In these theories, a sentence is associated not just with a semantic representation but with a semantic representation enriched with additional syntactic, pragmatic and/or discourse information. This additional information is then used to constrain the realiser output. One drawback of these NLG geared realisers however, is that the grammar used is not usually reversible i.e., cannot be used both for parsing and for generation. Given the time and expertise involved in developing a grammar, this is a non-trivial drawback. Prominent general purpose NLG geared realisers include REALPRO, SURGE, KPML, NITROGEN and HALOGEN.

Reversible realisers on the other hand, are meant to mirror the parsing process. They are used on a grammar developed for parsing and equipped with a compositional semantics. Given a string and such a grammar, a parser will assign the input string all the semantic representations associated with that string by the grammar. Conversely, given a semantic representation and the same grammar, a realiser will assign the input semantics all the strings associated with that semantics by the grammar. In such approaches, non-determinism is usually handled by statistical filtering: treebank induced probabilities are used to select from among the possible paraphrases, the most probable one. Since the most probable paraphrase is not necessarily the most appropriate one in a given context, it is unclear however, how such realisers could be integrated into a generation system.

In this work, we focus on this second type of realiser as it presents a number of important advantages. First, using a reversible grammar means that one and the same grammar and lexicon can be used both for parsing and for generation. Given the complexity involved in developing such resources, this is an important feature. Second, there are some engineering and computational motivations within the design of natural language systems such as consistency and non-redundancy. Thus, in dialogue systems [Benotti, 2009] it is important that the system produces the same language that it understands. Furthermore, reversibility makes it easy to rapidly create very large evaluation suites: it suffices to parse a set of sentences and select from the parser output the correct semantics. In contrast, NLG geared realisers either work on evaluation sets of restricted size (500 input for SURGE, 210 for KPML) or require the time expensive of developing each suite. In addition, a reversible grammar can be exploited to support not only realisation but also its reverse, namely semantic construction. Indeed, reversibility is ensured through a compositional semantics that is, through a tight coupling between syntax and semantics. Finally, the grammar can be used both to generate and to detect paraphrases.

In summary, reversibility has important advantages. If it is possible to develop reversible surface realisers which have comparable efficiency and functionality to non-reversible systems, then such systems would be preferred.

2.3 Surface realisation from flat semantics

A number of wide-coverage reversible computational grammars of Natural Language (NL) have been developed over the past few years. These grammars are used for generation from logical form input. The semantic input to the surface realiser for such grammars often is a flat semantic formulae which has a direct and unambiguous translation to first order logic [Copestake et al., 1999]. More specifically, a flat semantic formula is a bag of literals with semantic relationships (scope) between formulae captured by the appropriate instantiation of variable arguments (See Figure 2.1 for an illustration of such formulae). For a detailed description of two flat semantics frameworks, we refer the reader e.g., to [Gardent and Kallmeyer, 2003, Copestake et al., 1999]. For our purpose however, it suffices to say that a flat semantic formula is a set of literals where each literal consists of a predicate and some indices. There exist several flat semantic formalisms, but they have at least this much in common. We show for the sentence in (1) an example of a flat semantics input.

(1) *John seems to leave*
$\{leave(b), agent(b,c), john(c), seems(b)\}$

As argued in [Copestake et al., 1999], there are good reasons why Flat semantic representations were adopted in Natural Language Processing (NLP). Briefly, such representations fulfil the following criteria: (i) expressive adequacy, (ii) underspecification, (iii) computational tractability, and (iv) grammatical compatibility.

2.4 Complexity in surface realisation

As is well known [Kay, 1996], surface realisation is exponential in the length of the input. We here give a brief and intuitive summary of why this is so.

lack of ordering information

One first reason for the exponential complexity of surface realisation is the lack of ordering information . Contrary to parsing where the input is a string i.e., an ordered list of words, the input to surface realisation is a set of literals. Supposing each literal selects exactly one constituent in the lexicon, there could in the worst case be 2^n possible combinations between these constituents (the number of subsets obtainable from a set of size n).

intersective modifiers

In practice of course, there are possible restrictions on constituent combination. In particular, most existing realisers impose the constraint that only constituents with non overlapping semantics and compatible indices can be combined. Because of this restriction, the core of the complexity stems in practise from *intersective modiers* ([Kay, 1996, Brew, 1992]). Given a set of n modifiers all modifying the same structure, all possible intermediate structures will be constructed i.e. 2^n. For instance, there are $2^3 = 8$ possible subsets of modifiers in *fierce little black cat*.

(2) *cat,*
fierce cat,
little cat,
black cat,
fierce little cat,
fierce black cat,
little black cat,
fierce little black cat

lexical ambiguity

A second reason for the exponential complexity of surface realisation is lexical ambiguity . In surface realisation from flat semantics, the input is used to select a set of lexical entries, that is, all lexical entries whose semantics subsumes one or more of the input literals. In a wide coverage grammar, one literal will be associated with more than one lexical entries (as shown in Example 3). So, if Lex_i is the number of lexical entries associated with literal l_i, then, for an input semantics comprising n literals, the number of sets of lexical items covering the input semantics is: $\prod^{i=n}{}_{i=1} Lex_i$

(3) the case of *fast* and *quickly*

In addition, lexical ambiguity can also come from the many uses of a single lemma. For instance, different uses of verbs transitive and intransitive, or the same lemma with different functions as noun or verb (e.g. 4 and 5).

(4) *love* is transitive and intransitive

(5) the case of *place* that is a noun and a transitive verb.

Finally, lexical ambiguity can increase further if the grammar is paraphrastic i.e., associates several syntactic structures with the same semantics (e.g., *John destroyed the castle quickly/the destruction of the castle by John was quick*).

The two sources of complexity (lexical ambiguity and lack of input ordering) interact by multiplying out so that the potential number of combinations of constituents is: $2^n * \prod_{i=1}{}^{i=n} Lex_i$.

2.4.1 Related work on efficient realisation

We now review different approaches and discuss how they deal with the different sources of computational complexity in surface realisation.

[Kay, 1996]'s chart generation and indexing

Kay takes up again a previously introduced idea of using parsing charts in generation and defines a flexible approach for indexing.

Kay introduced an approach for indexing that relies on the use of flat semantics. He argues that its notation can be analysed as free word-order languages. Recall that a flat semantic formula is a set of literals where each literal consists of a predicate and some indices. For instance, (6) is the representation of the logical form of the sentences *John run fast* and *John run quickly*.

(6) $run(r), past(r), fast(r), arg1(r, j), name(j, john)$

It consist of a distinguished index r and a list of predicates whose relative order is inmaterial.

The basic chart algorithm schema proposed by Kay uses a bottom-up tree traversal. It assumes a grammar with binary rules and a lexicon that associates a sequence of words with a category and a semantic representation. The category and the semantic representation are linked with unification variables. The chart is initialised with those entries from the lexicon whose semantics subsumes the input semantics.

From this simple version of the algorithm Kay devised some key concepts. The use of a bit vector is introduced to check the *semantic coverage* before combining two edges , i.e. check whether two edges cover overlapping literals from the input semantics. Two flaws regarding efficiency are highlighted: (i) the fact that interactions must be considered explicitly between new edges and all edges currently in the chart, because no *indexing* is used to identify the existing edges that could interact with a new one; (ii) the process is exponential in the worst case, if a sentence contains k *modifiers* then a version with each of the 2^k subsets of those modifiers would be generated, all but one of them being rejected when it is finally discover that their semantics does not subsume the entire input.

To deal with the complexity issue of intersective modifiers Kay proposes a strategy based on the definition of the concept of *internal and external indices.* The basic idea is that edges should not be combined if the result of doing so would be to make internal (i.e. not more accesible) an index occurring in part of the input semantics that the new phrase does not subsume. This approach does not solve the generation of an exponential number of variants of phrases containing modifiers. However, it limits the spill out of ill effects by allowing only the maximal one to be incorporated in larger phrases.

Efficient wide coverage realisation

[Carroll and Oepen, 2005] present an algorithm for efficient tactical generation from underspecified logical-form semantics. They build on top of the chart generation proposed by Martin Kay and propose some refinements to the algorithm as well as two techniques namely, the integration of subsumption-based local ambiguity factoring, and a procedure to selectively unpack the generation forest according to a probability distribution given by a conditional, discriminative model.

This modified realisation algorithm is based on the Minimal Recursion Semantics (MRS) formalism, a member of the family of flat, underspecified, eventbased semantic frameworks for computational semantics, and an HPSG grammar. The basic chart generation procedure works as follows. In order to initialise the chart, lexical entries are retrieved from the lexicon checking against the input semantics. When a lexical entry is retrieved, the variable positions in its relations are instantiated in one-to-one correspondence with the variables in the input semantics. After initialising the chart (with inactive edges), active edges are created from inactive ones by instantiating the head daughter of a rule; the resulting edges are then combined with other inactive edges. Before combining two edges a check is made to ensure that edges do not overlap, i.e. that they do not cover the same relation(s).

In this approach, the generator operates exclusively on typed feature structures which are associated with chart edges. To guide the search from the input semantics two techniques are employed that relate components of the logical form to corresponding sub-structures in the feature structures. One is such that all feature structure correspondences to logical variables from the input semantics are made ground, namely *skolemization of variables.* The other is indexing by *Elementary Predications (EPs) coverage.*

Kay's coverage approach (Section 2.4.1) is implemented and it is used not only to combine active and inactive edges, but also to (i) determine which intersective modifier(s) can be adjoined into a partially incomplete subtree in the second phase of chart generation, (ii) and for a subsumption check when implementing local ambiguity factoring.

The authors argue that for a non trivial semantic input and a wide coverage grammar hundreds or thousands of edges may be produced. Then, indexing edges with a relatively small number of logical variables from the input semantics involves bookkeeping that turns out to be worthless in practise. In contrast, they suggest ruling out a priori edge combinations with incompatible indices by edge coverage. Further, filtering out the reminder by checking before unification which rules dominate which others and applying the quick-check as developed for unification-based parsing.

On the basis of [Kay, 1996]'s proposal they implement the *index accessibility filtering* a more general mechanism aiming at dealing with modifiers. To implement this, two sets of semantic indices are added for each chart edge. One contains the semantic variables in the feature structure that are accessible (could be picked up by another edge when it is combined). The other is the set of inaccessible variables, those ones that were once accessible but no longer are. The key usage of this sets to filter is on creating an inactive edge, each EP in the input semantics that the edge does not (yet) cover is inspected, and if the EPs index is in the edges inaccessible set then the edge is discarded (since there is no way in the future that the EP could be integrated with any extension of the edges semantics).

They review and compare their proposal with [Carroll et al., 1999]'s technique to deal with the modifiers problem, namely *delayed modifier insertion.* The processing of modifiers is delayed to a second phase where they are inserted in the generation forest at appropriate locations before the forest is unpacked.

To reduce the search space in generation the authors incorporate two techniques which work together. The *local ambiguity factoring* technique is based on a parsing strategy to compute the parse forest in polynomial time. It consist in packing sub-analysis dominated by the same non terminal and covering the same segment of the input string. In generation, the category equality test is replaced by feature structure subsumption and the input span is expressed as coverage of the input semantics. They found that packing is crucial to improve realisation time only if the subsumption operation between feature structures is used rather than feature structure equality.

Exhaustive unpacking results in an exponential time, then, to keep the total generation time polynomial they propose a *selective unpacking* procedure. A small set of the n-best realisations is extracted from the generation forest at a minimal cost.

The evaluation of the generator's efficiency was carried out under different configurations, i.e. different combinations of the techniques described above. The best-performing one was the one phase-generation with packing and index accessibility filtering. For cases with low- to medium-ambiguity, filtering gives rise to a bigger improvement than packing.

Chapter 3

Optimising surface realisation

TAG-based reversible surface realisation naturally admits the implementation of different optimisation techniques. The strategy we investigate and implement in this thesis work aims to optimise such a realiser. More precisely, we aim to reduce the initial search space, that is to reduce the effects of one of the sources of complexity: *lexical ambiguity.*

In this chapter, we first introduce the specific realiser we are working with (Section 3.1). In Section 3.2.2, we then summarise the polarity filtering optimisation proposed in[Gardent and Kow, 2005, Kow, 2007]. To deal with lexical ambiguity, polarity filtering permits eliminating combinations of lexical items which cannot possibly lead to a syntactically valid sentence. Finally, in Section 3.3.2, we present a novel approach for reducing the initial search space. Based on a conversion of the TAG grammar of elementary trees to an RTG (Regular Tree Grammar) of TAG derivation trees, this approach aims to filter out from the initial search space all combinations of elementary trees that cannot yield a derivation tree covering the input semantics.

3.1 Generating with Tree Adjoining Grammars

Before describing the surface realiser algorithm, we review the grammatical formalism it uses and recap some concepts such as TAG *derivation trees*, which are central in our optimising approach.

The algorithm uses a *Feature Based lexicalized* TAG (FB-LTAG) extended with semantic information as described in [Gardent and Kallmeyer, 2003]. We present the formalism as it is used by the surface realiser. We first present the core formalism and then introduce the two extensions which yield FB-LTAG. We also describe the association of the grammar, the syntactic formalism, with the flat semantic formalism. Finally, we emphasise the notion of TAG derivation tree, which is a central concept in the approach we are proposing.

3.1.1 The grammar: FB-LTAG integrating semantic information

Tree-adjoining grammars: The core formalism

Tree-Adjoining grammar is a grammar formalism defined by [Joshi and Schabes, 1997]. The elementary objects manipulated by TAG are trees, i.e., structured objects and not strings. Thus, it is a tree-generating system rather than string generating system. As defined in [Joshi and Schabes, 1997] a TAG consist of a tupple (Σ, NT, I, A, S) where

1. Σ is a finite set of terminal symbols.
2. NT is a finite set of non-terminal symbols: $\Sigma \cap NT = \emptyset$.
3. S is a distinguished non-terminal symbol: $S \in NT$
4. I is a finite set of finite trees, called **initial** trees where
   - interior nodes are labelled by non-terminal symbols;
   - frontier nodes are labelled by terminals or non-terminals; non-terminal symbols on the frontier are called substitution sites and are marked for substitution, by convention, annotated with a down arrow ($\downarrow$).
5. A is a finite set of finite trees, called **auxiliary** trees where
   - interior nodes are labelled by non-terminal symbols;
   - frontier nodes are labelled by terminal or non-terminal symbols; non-terminal symbols on the frontier are marked for substitution except for one node, called the foot node, by convention, annotated with an asterisk ($*$); the symbol labelling the foot node must be identical to that labelling the root node.

The trees in $I \cup A$ are called elementary trees and describe the syntactic structure of the basic components of a language, namely words or collocations. A tree built by composition of two other trees is called *derived tree.* The two composition operations that TAG uses are *substitution* and *adjunction.*

The *substitution operation* (Figure 3.1) replaces one substitution site of one tree by the tree to be substituted. The tree to be substituted must be derived from an initial tree. When a tree does not have substitution sites, we say that it is syntactically complete.

The *adjunction operation* (Figure 3.2) can be understood as splicing an auxiliary tree into another tree (which can be of any type, initial, auxiliary or derived.) Let α be a tree containing a non-substitution node n labelled by X, and β be an auxiliary tree whose root node is also labelled by X. Adjoining β into α is built by (1) excising the sub-tree of α dominated by n (call it t) (2) replacing the foot node of β with t to produce an intermediary structure β' and (3) replacing the excised tree in α with the augmented auxiliary tree β'. Nodes on which adjunction may be performed are called adjunction sites. By definition, any adjunction on a node marked for substitution is disallowed.

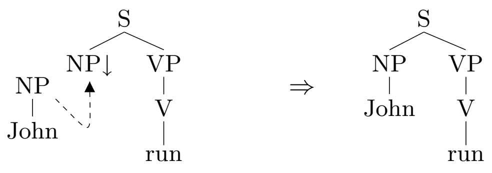


Figure 3.1: TAG substitution.

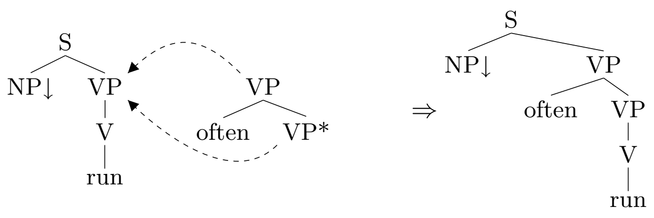


Figure 3.2: TAG Adjunction

Feature structure based TAG

A Feature Structure consists of a set of attribute-value pairs, where a value may be either atomic or another feature structure. The main operation for combination of feature structures is unification. A Feature Based TAG is a TAG where features structures are associated with the nodes of the elementary trees. That is, the tree nodes are decorated with two feature structures, called ***top*** and ***bottom***). The operations of substitution and adjunction are then defined in terms of unification of appropriate feature structures, thus allowing the constraints on substitution and adjunction to be modeled by the success or failure of unifications. On substitution, the top of the substitution node is unified with the top of the root node of the tree being substituted in. On adjunction, the top of the root of the auxiliary tree is unified with the top of the node where adjunction takes place; and the bottom features of the foot node are unified with the bottom features of this node. At the end of the derivation, a validation step takes place wherein the top and bottom of all nodes in the derived tree are unified. If unification fails, the derived tree is invalid.

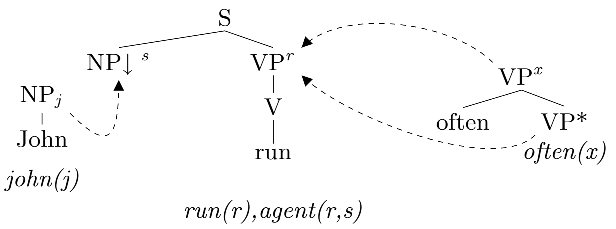
Lexicalized TAG

According to the definition given in [Joshi and Schabes, 1997], a grammar is *lexicalized* if it consist of: (i) a finite set of structures each associated with a lexical item; each lexical item will be called the *anchor* of the corresponding structure; (ii) an operation or operations for composing the structures. In lexicalized TAG at least one terminal symbol, namely the *anchor*, must appear at the frontier of all initial or auxiliary tree.

As mentioned at the beginning of this section, the surface realiser we work with uses a FB-LTAG, which is a straightforward combination of FB-TAG feature structures with LTAG lexicalization.

TAG **equipped with compositional semantics**

To associate semantic representations with natural language representations, the FB-LTAG is modified as proposed in [Gardent and Kallmeyer, 2003]. Each elementary tree is associated with a flat semantic representation [1]. For instance, in Figure 3.3, the trees[2] for *John*, *run* and *often* are associated with the semantic *john(j)*, *run(r), agent(r,s)*, and *often(x)* respectively.



⇒ *john(j), run(r), agent(r,j), often(r)*

Figure 3.3: Flat semantics for "John often runs"

Importantly, the arguments of a semantic functor are represented by unification variables which occur both in the semantic representation of this functor and on some nodes of the associated syntactic tree. More precisely, the substitution nodes of the tree associated with a semantic functor will be associated with with semantic parameters, that is, unification variables, while root nodes and certain adjunction nodes will be labelled with semantic indices. For instance, in Figure 3.3, the ***semantic index*** s occurring in the semantic representation of *run* also occurs in the subject substitution node of the associated elementary tree.

The value of semantic arguments is determined by the unification resulting from adjunction and substitution. For instance, the semantic index s in the tree for *run* is unified during substitution with the semantic indices labelling the root nodes of the tree for *john*. As a result, the semantic for *John often runs* is:

(7) {*john(j), run(r), agent(r,j), often(r)*}

Generally, the idea is that the association between tree nodes and unification variables encodes the **syntax/semantics interface**, i.e. it specifies which node in the tree provides the value for which semantic parameter in the semantic representation of a semantic functor.

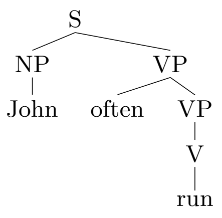
TAG **Derivations**

The tree obtained by derivation, the *derived tree*, does not give enough information to determine how it was constructed. On the other hand, the *derivation tree* is an object that specifies uniquely how a derived tree was constructed. Both operations, adjunction and substitution, are considered in a TAG derivation.

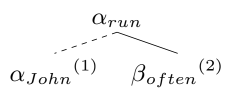
[1]The examples given actually show a simplified version of the flat semantics used in the surface realisation algorithm proposed in [Gardent and Kallmeyer, 2003] where in particular, so-called labels are omitted. A full specification is given in [Gardent and Kallmeyer, 2003].

[2]C^x/C_x abbreviate a node with category C and a top/bottom feature structure including the feature-value pair { **index** : x}.

The definition of a derivation tree is best explained by means of an example. Figure 3.4 shows the derived tree for the sentence *John often runs*. It has being built with the elementary trees shown in Figure 3.3. The root of a derivation tree for TAG s is labelled after the name of an initial tree of category S. All other nodes in the derivation tree are labelled by auxiliary trees' name in the case of adjunction or initial trees's name in the case of substitution. A tree address is associated with each node in the derivation tree. This tree address is the address of the node in the parent tree to which the adjunction or substitution has been performed. That is, back to the example, the derivation tree in Figure 3.4 shows how the derived tree was obtained. It tells as (i) what elementary trees it is made of (ii) and how they were put together. This derivation tree should be read as follows: the tree α_{john} is substituted in the tree α_{run} at address 1 (NP) and the tree α_{often} is adjoined in the tree α_{run} at address 2 (VP). The order in which the derivation tree is interpreted has no impact on the resulting derived tree.



derived tree



derivation tree

Figure 3.4: Derived and derivation trees for "John often runs"

3.1.2 A TAG based surface realiser

In this work we develop a strategy for optimising surface realisation within the context of the TAG-based surface realiser described in [Gardent and Kow, 2005, Gardent and Kow, 2007b, Gardent and Kow, 2007a]. The basic surface realisation algorithm used is a bottom up, tabular realisation algorithm optimised for TAGs. It follows a three step strategy that can be summarised as follows. Given an empty agenda, an empty chart and an input semantics ϕ:

Lexical Selection. Select all elementary trees whose semantics subsumes (part of) ϕ. Store these trees in the agenda. Auxiliary trees devoid of substitution nodes are stored in a separated agenda called the auxiliary agenda.

Substitution phase. Retrieve a tree from the agenda, add it to the chart and try to combine it by substitution with trees present in the chart. Stop when the agenda is empty.

Adjunction phase. Move the chart trees to the agenda and the auxiliary agenda trees to the chart. Retrieve a tree from the agenda, add it to the chart and try to combine it by adjunction with trees present in the chart. Add any resulting derived tree to the agenda. Stop when the agenda is empty.

When processing stops, the yield of any syntactically complete tree whose semantics is ϕ yields an output, that is, a grammatical sentence.

The workings of this algorithm can be illustrated by the following example. Suppose that the input semantics that given in example (7). In a first step (lexical selection), the elementary trees selected are the ones for *john*, *runs*, *often*. Their semantics subsumes part of the input semantics. The trees for *john* and *runs* are placed on the agenda, the one for *often* is placed on the auxiliary agenda.

The second step (the substitution phase) consists in systematically exploring the possibility of combining two trees by substitution. Here, the tree for *john* is substituted into the one for *runs*, and the resulting derived tree for *john runs* is placed on the agenda. Trees on the agenda are processed one by one in this fashion. When the agenda is empty, indicating that all combinations have been tried, we prepare for the next phase. All items containing an empty substitution node are erased from the chart (here, the tree anchored by run). The agenda is then reinitialised to the content of the chart and the chart to the content of the auxiliary agenda (here often). The adjunction phase proceeds much like the previous phase, except that now all possible adjunctions are performed. When the agenda is empty once more, the items in the chart whose semantics matches the input semantics are selected, and their strings printed out, yielding in this case the sentence *John often runs*.

3.2 TAG naturally derived optimisations

In this section we start by summarising how TAG naturally supports the introduction of different proposals aiming at reducing the surface realisation complexity, specially, in light of the complexity issues we introduce in Chapter 2 ([Gardent and Kow, 2006]). We then go on to describe the proposal put forward in [Gardent and Kow, 2006] for optimising the surface realiser introduced in previous Section 3.1.2.

3.2.1 Two composition operations: Two phase generation

The design of the surface realisation algorithm (Section 3.1.2) goes along with the two composition operations of TAG. This supports the integration of a mechanism to deal with intersective modifiers and to eliminate those ill trees obtained after the first phase before the second phase starts. Both optimisations prevent the proliferation of ill-combinations.

Reducing the impact of intersective modifiers

To deal with the lack of ordering information, more precisely, the problem of intersective modifiers a technique which enforces the delayed adjunction of modifiers has been introduced in [Carroll et al., 1999], and further evaluated in [Carroll and Oepen, 2005] besides proposing another approach. These proposals either handle modifiers after a complete syntactic tree is built (i.e., after all syntactic requirements are fulfilled) or before the modifiee is combined with other items (e.g., before the head noun has combined with a determiner). Although the number of intermediate structures generated is still 2^n for n modiers, both strategies have the effect of blocking these 2^n structures from multiplying out with other structures in the chart.

There are two specificities in TAG that support the implementation of a delayed adjunction of intersective modifiers. Due to the fact that in TAG substitution and adjunction are applied independently of each other a two-phase generation strategy is naturally implemented [Gardent and Kow, 2006]. Thus, generating all syntactically complete trees in a first phase. Second, in TAG intersective modifiers are introduced by adjunction trees then they will be processed in the second phase adding the modifiers to the complete trees obtained in the previous phase. As a result, proliferation of intermediate structures syntactically correct but semantically incomplete induced by intersective modifiers is restricted.

An important difference with other approaches to delayed modifiers adjunction is that with TAG it is not necessary to specially write or modify rules of the grammar to account for the delayed adjunction strategy.

Eliminating complete unusable trees

The two-phase generation strategy supports another optimisation which takes place in between the two-phases. After the first phase was completed, it is possible to filter out the ill formed trees before starting with the following phase. This other mechanism aims at filtering out those derived trees produced by the first phase for which there is no way to further combine them into a successfull tree. Two different filters are applied respectively for:

- all trees with an unfilled substitution site
- all saturated trees whose root node is not labelled with an S category

The first filter eliminates unsaturated trees. That is, eliminates those trees that were left with a not fulfilled substitution site after the first-phase. As the second phase only applies adjunction, and adjunction is not allowed on substitution nodes (Section 3.1.1), this trees would not be further completed within the rest of the generation algorithm.

The second, called *Root Node Filter* (RNF), is based on the property of auxiliary trees (described in Section 3.1.1) which insists that root and foot node should be labelled with the same category. Because of this property, adjunction cannot affect the category of the tree it adjoins to. In particular, a tree which after all possible substitutions have been performed, has root label C with $C \neq S$ can never lead to the creation by adjunction of a tree with root label S. Thus, this trees are also ruled out and not introduced into the following phase.

Both filters result in a reduction of the trees introduced in the second phase. Hence, they contribute to the carry out the adjunction of modifiers only within a reduce set of candidate trees.

3.2.2 Polarity filtering

The polarity filtering optimisation introduced in [Gardent and Kow, 2005] takes place between the lexical selection phase and the generation phases of substitution and adjunction. The basic idea is that after selection of all the lexical items whose semantics subsumes the input semantics, lexical combination with non-neutral polarities are filter out, and then perform realisation on the remaining items. This technique reduces the complexity introduced by lexical ambiguity. As explained in Section 2.4, the number of combinations that are a priori possible after the lexical selection phase is $\Pi_{1 \leq i \leq n} a_i$ with a_i the degree of lexical ambiguity of the i-th literal and n the number of literals in the input semantics.

The motive for the interest in reducing this combinatorics is based in the observation that not all the combinations of the lexical items would lead to a successful derivations but only some of them. That is, some combinations of lexical items that cover the input semantics turn out to be syntactically invalid either because a syntactic requirement is not fulfilled or because a syntactic resource is not used. For instance, given the semantics in (8), the set of lexically selected items for this input are those shown in Figure 3.5.

(8) $\{picture(p), cost(c), agent(c,p), patient(c,h), high(h)\}$

Lexical ambiguity

In this example we can see that more than one tree was selected for each literal in the semantics. For example, for the literal $picture(p)$ the trees $T_{picture} : picture(p)$ and $T_{painting} : picture(p)$, the same occurs for $cost(c)$ and $high(h)$. Although, it is not possible to successfully obtain all the combinations out of them as there may be some syntactic incompatibilities. Indeed, $T_{cost} : cost(c)$ may be combined with $T_{ishigh} : high(h)$ to form the sentence *The cost of N is high* while combining $T_{cost} : cost(c)$ with $T_{alot} : high(h)$ results in the phrase *The cost of a lot* which is an invalid sentence. The same occurs when combining $T_{costs} : cost(c)$ with $T_{alot} : high(h)$ and $T_{ishigh} : high(h)$.

Syntactic variations

The problem of synonymy in the lexicon and generating the compatible lexical combinations becomes even worse within the context of wide-coverage lexicalized grammar. As sketched in Section 3.1.2, the realisation algorithm for a wide-coverage lexicalized TAG consist of three phases, namely lexical selection, and two phase-generation: substitution and adjunction. In the first phase, the realiser selects a set of elementary TAG tree schemas that are associated with the lexical items steming from the lexical semantics in the input semantic formula. Once this lexical selection is done, the realiser works over the selected trees rather than over the whole grammar. However, for every lexical item hundreds of trees might be selected in a wide-coverage grammar, then the size of this set, and thus, the initial search space for the generation phases is not trivial.

To detect and eliminate unsuccessful combinations polarity based filtering proceeds as follows:

- assigns each lexical item a polarity signature reflecting its syntactic requirements and resources
- computes for each possible combination of lexical items the net sum of its syntactic requirements and resources and
- eliminates all combinations of lexical items that do not have a net sum of zero (because such combinations cannot possibly lead to a syntactically valid sentence)

Then, the filter has two main steps. The first one concerns calculating the polarity signature of each lexical item. Since in TAG, substitution nodes indicate syntactic requirements while an initial tree permits fulfilling a syntactic requirement, polarity signatures can be automatically computed as follows:

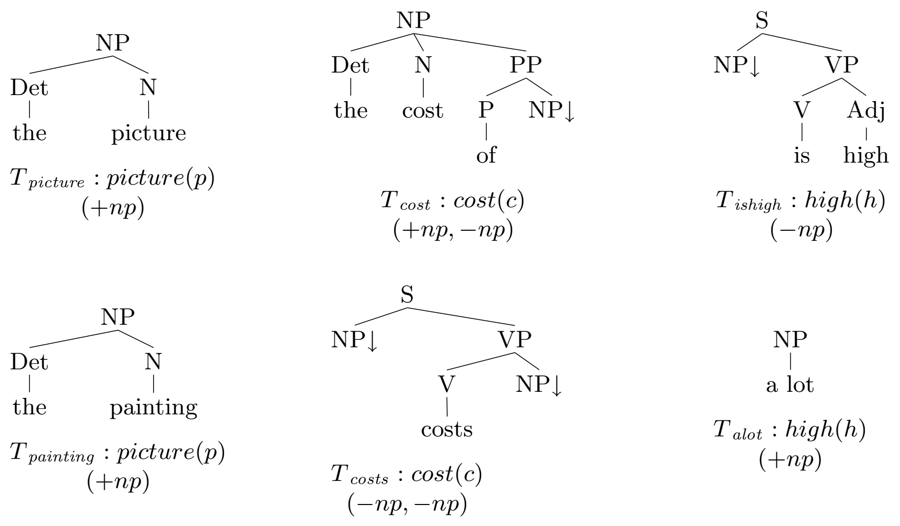


Figure 3.5: Small grammar for the input semantics in 8

- a polarity of the form $+Cat$ is added to the tree polarity signature of each initial tree with root node category *Cat*.
- a polarity of the form -*Cat* is added to the tree polarity signature of each initial tree for each substitution node with category *Cat* in that tree.

In Figure 3.5 the polarity charge for each tree is shown below the tree. For instance, the tree T_{costs} has polarity $(-np, -np)$ meaning that it requires two NPs while if we considered the category s the polarity would be $(+s)$ meaning that it provides a sentence.

The second step is to compute the polarity of all possible combinations of lexical items. This is done by:

- building a *polarity automaton* for each polarity category occurring in the set of possible combinations (in this case, np),
- computing the intersection of these automaton and
- minimising the resulting automaton.

For the example we have followed so far, the final automaton is that given in Figure 3.6 where each state is labelled with the accumulated polarity of the path(s) leading to that state and where the transitions are labelled with the lexical item covered. As discussed above, the combination of T_{cost} : $cost(c)$ with T_{alot} : $high(h)$ is useless as we get *the cost of a lot* which is also incompatible with *the painting*. This can be seen in the polarity automaton where the sum up of the polarities of $T_{painting}$: $picture(p)$, T_{cost} : $cost(c)$, and T_{alot} : $high(h)$ gives a $+2np$ charge and the combination is rejected (does not finish in a final state). In contrast, the combination of $T_{painting}$: $picture(p)$, T_{costs} : $cost(c)$, and T_{alot} : $high(h)$ has a charge of $0np$ which means that this combination will be worth trying in the surface realisation phase.

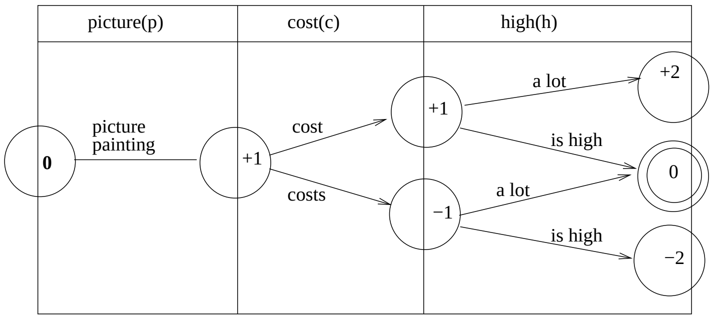


Figure 3.6: Automaton for the input semantics in 8

Polarity filtering is naturally applied to TAG based on definitory properties of TAG and it is proven that it drastrically reduces surface realisation search space. Still, there are some limitations in the strategy.

Limited number of polarity keys. The definition of polarity key assumed by the strategy is: a pair $a : k$ where a is an attribute and k is a possible value for that attribute. However, it is not possible in practise to take another attribute rather than the category. This is because the strategy assigns polarities to the lexical items based on the polarity key, that is based on the values of the attributes. Then, it is required that the values of the attributes are instantiated. For instance, the attribute associated to nodes in the tree which correspond to the semantic indice (Figure 3.3) cannot be used. As the requirement of an attribute value of being instantiated cannot be assured for most of the attributes present in the nodes of the trees, practically, the category feature is the only one that is can be taken into account by the filtering strategy.

Cost of automaton intersection. Using multiple polarity keys can be expensive because of the potential cost of computing their intersection. Although polarity filtering simplifies the automaton enough for intersection not to be a problem, it has being evaluated with a small set of polarity keys. The results obtained in the stragey evaluation show that the more keys are included in the set the better is the filtering done. But the improvements made by adding more polarity keys has a limit which is given by the cost benefit relation. In other words, it could be helpful to the extent that the overhead of building, minimising and intersecting these automaton does not offset the gains made.

Ignoring of auxiliary trees. The polarity filtering strategy, as discussed so far, is constructed based only on initial trees. In other words, the auxiliary trees are invisible to the polarity filtering. However, an extention to account for the auxiliary trees is forseen. Basically, it consist in defining differently the charges of auxiliary trees, indeed, to obtain the polarity signature of a given auxiliary tree, charges are multiplied instead of being added. An *adjunction automaton* is constructed in similar way to the polarity automaton and it can be further combined by the intersection operation with other adjunction or polarity automatons. Despite this possibility of using adjunction automatons the shortcoming comes down to the problem of using multiple polarity keys.

3.3 RTG-Based optimisation

In this section we present a more principled and empirically more complete filtering method for TAG based surface realisation. This approach is also naturally derived from TAG and is based on the fact that the derivation tree language of TAG can be generated by a *Regular Tree Grammar* (RTG). Moreover, there exist a well defined translation from Feature Based TAG to a Feature Based RTG [Schmitz and Le Roux, 2008]. Therefore, we can translate each TAG elementary tree, including as much linguistic information as convenient, into *Regular Tree Grammar* (RTG) rules and then generate the derivation trees (defined in Section 3.1.1). Using this TAG to RTG mapping, we can then identify those combinations of TAG elementary trees that can be combined into a derivation tree and therefore are worth taking into account for surface realisation.

We first introduce the grammatical formalism underlying our filtering method, and then proceed by describing how the strategy works.

3.3.1 From TAG to RTG

Feature Based RTG is a formalism defined by [Schmitz and Le Roux, 2008]. They argument that the motivation behind the definition of the regular tree grammar formalism extention was the fact that the derivation tree language of TAG is much simpler to manipulate than the corresponding derived language because of being a tree language. They observed that derivation trees are the base of many approaches to sentence generation as well as semantic computation. However, if the grammar includes feature structures they should be moved into the corresponding derivation trees, in order to account for all the information given by the grammar. Therefore, [Schmitz and Le Roux, 2008] defined the Feature Based variant of RTG and a translation from Feature Based TAG. The *Feature Based* RTG produces derivation trees that account for the feature structures found in a FB-TAG.

On one hand, TAG derivation trees encode two important pieces of information about a derivations: the elementary trees involved and in which nodes of the elementary trees the substitutions or adjunctions took place. Each node of the derivation tree is named after an elementary tree and its daughter nodes represent the elementary trees combined by substitution or adjunction in the mother node. On the other hand, a RTG (grammar formalism defined in [Comon et al., 2007]) provides the means for describing tree languages.

[Schmitz and Le Roux, 2008] start by defining the RTG of the TAG derivation trees. Briefly, each TAG elementary tree is converted to an RTG rule where, in relation to the notion mentioned in previous paragraph, the rule consists of a mother node as *left-hand side* (i.e. node carring the tree information needed for combination *into other trees*) and daughter nodes as *right-hand side* (i.e. indicating which operations can be applied and which trees can be used for combinations *into this tree*). These daughter nodes were called *active nodes* and the number of active nodes is the rank of the grammar rule. To name the nodes in the grammar rule, they defined a labelling function that considers the syntactic category and the type of the original TAG node. Considering a TAG elementary tree γ, its nodes are labelled as follows to give rise to the RTG non-terminals:

$$\begin{cases} X_A & \text{if } \gamma_i \text{ is an adjunction site} \\ X_S & \text{if } \gamma_i \text{ is a substitution site} \end{cases} \tag{3.1}$$

The labelling for the left-hand side of the RTG rule includes also the syntactic category, but it will be X_S if the rule is a map from an initial tree and X_A if the rule is a map from an auxiliary tree. For instance, the grammar rule corresponding to the elementary tree anchored by *the cost of* in Figure 3.5 is then $NP_S \rightarrow cost(NP_A, Det_A, N_A, PP_A, P_A, NP_S)$, meaning that this tree can be substituted into an NP substitution node, and furthermore that it expects adjunctions on its NP_A, Det, N, PP and P nodes and a substitution on its NP node.

To model the fact that adjunction is in fact optional, an epsilon rule, $X_A \rightarrow \varepsilon_A$, is added for each main category. Each epsilon rule describe the fact that a C_A category (with $C \in \{NP, Det, N, PP, P, VP, V\}$) can rewrite to the empty string.

The important point for our purpose, is that, given each set of elementary TAG trees, the TAG to RTG mapping makes it possible to determine whether this set can be combined into a derivation tree.

So far, we saw how RTG rules are built from the TAG elementary trees based on the type of trees and active nodes. To this first step of encoding, [Schmitz and Le Roux, 2008] add the encoding of the features structures and define the *feature based regular tree grammar* formalism.

In a *feature-based regular tree grammar* the rule categories become a tuple where each non-terminal is accompanied of a feature structure. That is, rules are of the form, $(A, d) \rightarrow a((B_1, d_1), ..., (B_n, d_n))$, where $A, B_1, ..., B_n$ are non-terminals, $d, d_1, ..., d_n$ are feature structures, and a is a terminal with rank n. The derivation relation for this kind of rules not only requires matching of the non-terminal symbol but also the the computation of the most general unifier mgu at each derivation step (i.e. an μ-substitution σ).

They propose two ways of encoding a FB-TAG. However, they argue and show that the *left corner transformation* results in more predictive derivations. Therefore, for our pourpose we have chosen to make use of this translation mode. This transformation enables the derivations in the following order: first apply the ε-rules of ranking 1, then apply the root adjunctions (i.e. auxiliary trees that adjoin into root nodes of a derived or initial trees) in reversed order, and end with the initial tree substitution.

As for the encoding of the feature structures within RTG rules, they propose that the left-hand side and the lef-corner of the rules should provide a sort of interface of the tree. That is, they should account for *root top* feature structure information of initial and auxiliary trees and *foot bottom* feature structures for the case of auxiliary trees. On the other hand, the right-hand side symbols should include the top and bottom feature structure from the active nodes.

Concretely, the following definitions are devised in [Schmitz and Le Roux, 2008]. The first equations proposed by [Schmitz and Le Roux, 2008] describe the map for the features structures, then, their definition below defines the map from TAG elementary trees to RTG rules. We will see along this work how this feature based RTG formalism is used by the RTG-based polarity filtering approach.

$$in(\alpha) = \begin{bmatrix} top : t \\ top : top(\alpha_r) \end{bmatrix}. \tag{3.2}$$

$$in(\beta) = \begin{bmatrix} top : t \\ top : top(\beta_r) \\ bot : bot(\beta_f) \end{bmatrix}. \tag{3.3}$$

$$feat(\gamma_i) = \begin{cases} \begin{bmatrix} top : t \\ bot : bot(\gamma_r) \end{bmatrix} & \text{if } \gamma_i = \gamma_r \\ \begin{bmatrix} top : top(\gamma_i) \\ bot : bot(\gamma_r) \end{bmatrix} & \text{otherwise} \end{cases} \tag{3.4}$$

$$in_{lc}(\beta) = \begin{bmatrix} top : t \\ bot : bot(\beta_f) \end{bmatrix}. \tag{3.5}$$

$$feat(\gamma_i) = \begin{cases} \begin{bmatrix} top : t \\ top : top(\gamma_r) \\ bot : bot(\gamma_r) \end{bmatrix} & \text{if } \gamma_i = \gamma_r \\ \begin{bmatrix} feats(\gamma_i) \end{bmatrix} & \text{otherwise} \end{cases} \tag{3.6}$$

$$in_{lc}(\beta) = \begin{bmatrix} tr_{lc}(\gamma_i) = (nt(\gamma_i), feats_{lc}(\gamma_i)) \end{bmatrix}. \tag{3.7}$$

Definition. The *left-corner transformed feature-based* RTG $G_{lc} = \langle S_S, N \cup N_S \cup N_A, F_{lc}, D, R_{lc} \rangle$ of a TAG $\langle \Sigma, N, I, A, S \rangle$ with feature structures in D has terminal alphabet $F_{lc} = I \cup A \cup \{\varepsilon_A, \varepsilon_S\}$ with respective ranks $rk(\alpha) - 1$, $rk(\beta)$, 0, and 1, and set of rules
$R_{lc} = \{X_S[top : t] \rightarrow \varepsilon_S(X[top : t; bot : t]) | X_S \in N_S\} \cup \{(X, feats(\alpha_1)) \rightarrow \alpha(tr_{lc}(\alpha_2), ..., tr_{lc}(\alpha_n)) | \alpha \in I, n = rk(\alpha), X = lab(\alpha_r)\}$
$\cup \{(X, feats_{lc}(\beta_1)) \rightarrow \beta((X, in_{lc}(\beta)), tr_{lc}(\beta_2), ..., tr_{lc}(\beta_n))$
$| \beta \in A, n = rk(\beta), X = lab(\beta_r)\}$
$\cup \{(X_A, in(\beta)) \rightarrow \beta(tr(\beta_1), tr_{lc}(\beta_2), ..., tr_{lc}(\beta_n)) | \beta \in A, n = rk(\beta), X = lab(\beta_r)\}$
$\cup \{X_A[top : t; bot : t] \rightarrow \varepsilon_A | X_A \in N_A\}$

Importantly for our purpose, is the note they made regarding computational complexity properties, that is that the translation can be computed in linear time, and results in a grammar with at worst twice the size of the original TAG.

3.3.2 RTG filtering

The idea of RTG-based filtering is to use this TAG to RTG mapping to determine whether a given set of RTG-encoded TAG trees can be combined to yield a well formed TAG derivation tree. Any set of trees that does not can be filtered out. This filtering takes place after the lexical selection phase in the surface realisation algorithm (described in previous sections). This consist in,

- building the RTG grammar G_{RTG} using the TAG encoding and the results from the lexical selection phase,
- obtaining $L(G_{RTG})$, i.e. generating the corresponding parse forest

After applying the translation we get a grammar G_{RTG} as defined by the translation such that all TAG elementary trees (i.e. initial and auxiliary trees) are taken into account. The non-terminal symbols in the RTG grammar (i.e. X_A,X_S and X) are built by the labelling function of the encoding from the categories present in these trees. Thus, all the categories present in each node of each TAG elementary trees are also present in the new grammar and consequently can be used for filtering. In addition, the translation of the feature structures allows the transference of linguistic information from the TAG of derived trees to the RTG of derivation trees. We can, in principle, select different features and feature values. As the derivation relation encompasses unification, there are no difficulties in dealing with non-instantiated values for the features[3]. The filtering algorithm could be any that implements the derivation relation defined for RTG, the major requirement here being that the algorithm should be guided by the input semantics.

Each elementary tree is represented as the fragment it contributes to the derivation tree. Then, any set of TAG tree that cannot be combined into a derivation tree in $L(G_{RTG})$ should be ruled out. Further, the information provided by $L(G_{RTG})$, namely the set of successfull combinations and potential candidates, can be taken into account by the surface realiser to produce the TAG derived trees in a straightforward manner.

Different levels of filtering

We can define different types of filtering depending on how much linguistic information (i.e. features from the feature structures) is preserved in the RTG trees.

A *first level* of filtering would be where only the syntactic category is preserved. The categories constitute the non-terminal symbols in the RTG, then, at this level no feature structures would be transferred. In some cases, the successful combinations resulting from this filtering level would account for some permutations of the trees in the combination as the semantic index information is not included. This is the level at which polarity filtering was applied overcoming one difference. In this RTG filtering we cover all the categories involved in TAG derivations while in polarity filtering for each category considered one more automaton is involved in the intersection operation. Then, for the latter the complexity is growing exponentially for each new category that is added to the filter.

A *second level* includes both syntactic category and semantic index (the features presented in 3.1.1). This semantic information should a priori provide much better guidance for the filtering process since semantics information is what guides generation.

A *third level* would imply developing methods for automatically identifying the grammar features that are most used in generation and then using those for filtering.

[3]This is in contrast to the polarity filtering approach where only syntactic categories are taken into account and unification of feature values is not performed.

Chapter 4

Generation of the parse forest

The filtering technique we describe in the previous chapter, uses an RTG encoding of a Tree Adjoining Grammar to filter out from the initial search space, all sets of elementary trees which cover the input semantics but cannot yield a complete derivation tree.

To detect whether a given tree set can yield a full derivation tree thus boils down to parsing with an RTG. Given the set of RTG rules that is associated by the lexical selection module with the input semantics, such a parsing algorithm permits determining which set of RTG rules, if any, covers the input semantics.

In this chapter, we present the RTG based parsing algorithm we use for filtering and we discuss its integration in the surface realisation process.

We start by describing the main characteristics of this algorithm (Section 4.1). We then describe the Prolog implementation we wrote for testing our RTG filtering proposal (Section 4.2).

4.1 A tabular Earley-style RTG parser

The algorithm we designed for implementing our RTG filter integrates several ideas and techniques from the parsing literature. We draw on [Shieber et al., 1995]'s deductive parsing framework as a flexible and declarative way to represent different parsing algorithms. To avoid the repeated computation of intermediate structures common to several larger parse structures, we integrate [Kay, 1986]'s chart mechanism[1]. Finally, the specific parsing strategy adopted is the Earley's algorithm [Earley, 1970] adapted as described in [Kay, 1996] to support generation from a flat semantics.

4.1.1 Chart parsing and Earley algorithm

As it is well known, chart parsing, through the concept of *chart* data structure, provides a means to "store" intermediate parsing results which can be further used in different larger constructions along the parsing process. That

[1]The storing and reuse of intermediate results is also sometimes referred to as "tabulation" or "dynamic programming".

is, whenever the same intermediate result is used within different larger parsing constructions, it can be taken from the chart instead of being constructed each time it is needed. In addition, chart parsing allows for pursuing all alternative analyses in parallel. Particularly, in our problem, we are exploring not just one but *all* possible lexical combinations that the RTG grammar can generate. Indeed, in this process we generate several intermediate results which are used to build several distinct derivation trees. By storing those intermediate results in a chart we can avoid their re-computation. Hence, the implementation of the parser for the filtering method is based on chart parsing.

The specific parsing algorithm we implement is the Earley algorithm as such a search strategy provides a hybrid approach which integrates both top-down predictions and bottom-up completions.

Finally, the parsing as deduction framework, by stating parsing algorithms in terms of axioms, goals and set of inference rules, provides us with a useful tool for describing the algorithm. To account for the feature structures present in our grammar formalism, Feature Based RTG, and the fact that the parsing process is guided by the input semantics, we furthermore adapt this algorithm as summarized in Table 4.1.

Axiom	$\overline{[S' \rightarrow \bullet S_S, \emptyset]}$
Goal	$[S^{'} \rightarrow S_S \bullet, \phi]$ where ϕ is the input semantics.
Prediction	$\dfrac{[(A,d) \rightarrow a(\alpha \bullet (B,d_i)\beta), \varphi]}{[(B,\sigma(d')) \rightarrow b(\bullet(B_1,\sigma(d'_1)),...,(B_n,\sigma(d'_n))), \psi]}$ where $(B,d^{'}) \rightarrow b((B_1,d^{'}_1),...,(B_n,d^{'}_n))$ is a rule in the grammar with associated semantics ψ, $\sigma = mgu(d_i,d^{'})$ and $\varphi \cap \psi = \emptyset$
Completion	$\dfrac{[(A,d) \rightarrow a(\alpha \bullet (B,d_i)\beta), \varphi][(B,d') \rightarrow b(\beta)\bullet, \varphi]}{[(A,\sigma(d)) \rightarrow a(\alpha(B,\sigma(d_i)) \bullet (C,\sigma(d_{i+1}))\beta), \phi]}$ where $\sigma = mgu(d_i,d^{'})$, $\varphi \cap \psi = \emptyset$ and $\varphi \cup \psi = \phi$

Table 4.1: Earley deductive parsing system

The item standard representation is $[(A,d) \rightarrow (B_1,d_1),..,\bullet(B_i,d_i),...,(B_n,d_n),\psi]$ where the dot in the production marks the point reached by recognition or analysis of the derivation tree. The second component of the item is ψ representing a semantic formula. In the items we do not keep track of string positions, as usually done when parsing a string, but rather we keep the associated semantic formula. As we saw in Section 3.1.1, each TAG elementary tree is enriched with a flat semantic formula, accordingly, their counterpart Feature Based RTG rules also maintain associated the same flat semantic formula.

The algorithm starts from the initial fact, the axiom, $[S^{'} \rightarrow \bullet S_S, \emptyset]$. Note that in this item the non-terminal symbol S_S is the axiom in the grammar

G_{RTG} while the second component represents the empty semantics. In string parsing, we would have the string index equal to 0 meaning that the recognition of the string is at that point. In contrast, as we are parsing based on the input semantics the part of the input semantics analysed so far is empty. Note here that because of lexical ambiguity, there might be several axioms, steaming from the several initial trees. On the other hand, in the goal item $[S^{'} \rightarrow S_S \bullet, \phi]$ the dot at the end of the item production means that S_S has been analysed and that on reaching this position the semantics should be exactly the input semantics.

Both inference rules, *prediction* and *completion*, have certain preconditions. The semantic preconditions regarding the guidance of the input semantics represent the semantic coverage put forward in [Kay, 1996]. For the case of the *prediction* inference rule it is required that not only the non-terminals match but also that there exists $\sigma = mgu(d_i, d^{'})$ between the features structures associated to the non-terminal symbols. Moreover, a restriction on the semantic coverage of the items, imposes that the semantic coverage of the antecedent item does not overlap with the semantic coverage of the consequent item. In this way, we restrict predictions to those items that when becoming passive would be successfully applied by future applications of the *completion* inference rule. Specifically, in our grammar there might be many trees for each lexical item augmented and sometimes most of them potentially predictable in the prediction step. By checking for the semantic coverage we would rule out in advance at the prediction step those trees corresponding to lexical items that were already considered, thus, introducing less items in the parsing process. As for the *completion* inference rule, also known as fundamental rule, the first restriction requires that there exists a most general unifier among the feature structures present in the passive and active items. Further, the two semantic constraints lay down the criteria on item combination based on the semantic coverage. One of them establishes that the semantic coverage of the two items being combined should not overlap and the other that the semantics of the resulting item should include the semantics of the two items being combined.

Inference rules are applied repeatedly starting from the axioms, producing new facts (active or passive items), and working the way towards the goal item. Specifically, this processing is carried out by the chart parsing framework. When talking about this items or facts within the chart parsing implementation we would refer to them as edges. And as the items that we store in the chart are complete and incomplete ones, that is active and inactive items, the type of chart parser amounts to an active chart parser. Active edges are referred to as dotted rules.

4.1.2 Mechanisms included in the parser

As mentioned above, there are several desired parsing techniques that are embedded in the parsing approach we propose, namely chart parsing, Earley algorithm and deductive parsing. Additionnally, several other mechanisms are introduced which are specific to the problem at hand namely surface realisation from a flat semantics. We now discuss each of these points in more detail.

Tabulation of intermediate results. Tabulation techniques are motivated by considering problems which display a high degree of redundant computations such as in particular, Natural Language (NL) parsing. One major cause for redundancy in Natural Language parsing lies in the inherent ambiguity of natural language and thereby of its grammar. Additionally, in our case, because the grammar used is a wide-coverage lexicalized grammar, lexical ambiguity (the number of grammatical units associated with each word or lexical semantics) is very high. Indeed, for one lexical item there might be several family trees with several trees. Furthermore, each of these trees might give place to different derivations where each of these derivations may have sub-derivations in common. By using tabulation, such intermediate or partial results can be "stored" for future use thus avoiding their re-computation. One such approach for tabulation of intermediate results is chart parsing.

Sharing of intermediate results and computing derivations. Tabulation does not imply sharing, just storing. It is indeed possible to store intermediate results but to copy them whenever they need to be combined with several distinct structures. To avoid copying, subtree sharing can be implemented by means of pointers. This reduces the cost of storing items but adds a load in the post-processing step when recovering parse trees.

Chart parsing facilitates item sharing in that parse trees can be reconstructed out of chart items by some off-line technique (e.g. by following the trail of string position indices). Alternatively, the construction of the derivation information can be generated in parallel with the recognition process. Our parsing approach requires the construction of derivations. For this, the items would include another component whose value is a derivation tree, where nodes are labelled by the TAG tree names (i.e. each production of the grammar that in fact also has the tree name information) and the list of ordered daughter derivation trees, one for each non-terminal of the right-hand side of the item before the dot.

This may seem to make the storage of items rather pointless as items with different derivations will be regarded as different in the chart. The point thought is that for our problem, the heavy re-use of items comes from combining several times some intermediate structure (e.g. a given S intermediate structure) with all possible trees in all possible families. Thus, we are still taking advantage for this case and these intermediate results are not re-computed nor copied when deriving new items. To eliminate these item differentiations, introduced by keeping the derivation information as another component in the item representation, we should (i) write down the derivation information while the derivation process is being performed and (ii) implement a post-processing approach to reconstruct the derivations as suggested by [Shieber, 1988].

Agenda based control. As has been said (Section 4.1.1), chart parsing is a mechanism for applying the inference rules on items and provides a chart data structure for storing them. On applying the inference rules new items are generated. This items are not directly placed into the chart but stored in an auxiliary storage structure called *agenda*. Indeed, this agenda contains the items to be analysed. The general schema for chart parsing algorithms [Shieber, 1988] is the following:

- Move an item from the agenda into the chart.
- Apply inference rules and add the newly produced items into the agenda for later processing.

- If there are no more items in the agenda then stop, otherwise process the following item.

Note that the item taken from the agenda would be actually added into the chart if it is not already in it. As the RTG rules are non-ground an adequate existence checking requires subsumption checking, as we will discuss below.

The use of an agenda allows the customization of the search strategy. Specifically, implementing the agenda as a stack would provide a depth-first search strategy, whereas implementing it as a queue would result in a breadth-first search. The difference in the search strategy does not affect the results produced by the selected tree traversal.

In our implementation we add one more checking for redundancy in the agenda. That is, we prevent new derived items from being added more than once in the agenda. By checking this early, we avoid having to keep track of such redundant items.

Indexing. In string parsing, chart items standardly contain two indices, pointing at the start and end positions of the recognised span over the input string. These positions help not only in ensuring correctness but also in improving efficiency, as two edges are only considered for combination whenever they are adjacent. In this way, the number of non-productive attempts at applying the inference rules is reduced. In our problem, because the input to surface realisation is a flat semantic formula that is a set of literals, there are no string positions to use. Instead, we use the information given by the semantic indices in the semantic formulae as proposed in [Kay, 1996].

Based on Kay's proposal, we suggest the following mechanism for chart indexing in our problem. Each item is associated with a given semantic index. Moreover, we distinguish between active and passive items. For passive items, we require that the index (INDEX) be the distinguished semantic index associated with the lexical semantics or the semantics of the head of the item. For active items on the other hand, the index (DOTTED-INDEX) is determined by the semantic index requirement of the current active symbol on the right-hand side of the active item, that is, the `idx` feature in the top or bottom feature structure of the node. When an item becomes passive both associated items should be the same (INDEX and DOTTED-INDEX become the equal). Note that this condition could be also further exploited to implement an efficient access to passive items in the chart.

Semantic filter. A semantic filter is implemented based on the standard idea of edge coverage. In our algorithm, we use this as described in the previous section (preconditions regarding semantics in the inference rules). To begin with, each predicted edge is associated with the semantic formulae in the grammar rule that gives rise to this item. Next, when edges are combined their semantics are too. Then, the *semantic filter in edge combination* takes place allowing or not those edge combinations. That is, checking if their semantic does not cover the same lexical items. *Semantic filter in edge prediction* consists in permitting the prediction of new edges as long as the semantic of the newly predicted item is not subsumed by the semantic of the active one (i.e. the intersection is empty). This means that, in advance, we only predict edges which can combine with actives edges at least from the semantic coverage point of view.

Unification. In the completion or prediction steps, a unification mechanism is used, given that our grammar rules do not consist of atomic categories but rather of tuples (B, d). More precisely, as described in Section 3.3.1, B is a non-terminal expressing both the syntactic category and the kind of site it represents within the original TAG grammar (e.g. VP_A (i) as node of category VP being an adjunction site, if the non-terminal is present in the right hand-side of the RTG rule, or (ii) as an adjunction tree with root category VP, if the non-terminal is in the left-hand side of the RTG rule). d represents the top and bottom feature structure associated with the non-terminal symbol. Therefore, in our algorithm we use a specific unification procedure to match the grammar categories.

Subsumption based blocking of new edges. On adding a new edge into the chart, it is essential to check whether it already exists in the chart or not. This verification, in our algorithm, involves subsumption checking. The reason is that the RTG grammar rules contain feature structures, underspecified features values and sometimes underspecified syntactic category (as in the case of epsilon rules), therefore, newly derived edges may differ only in terms of instantiations. Instead of keeping every differently instantiated edge, the idea is to keep the most general form which would be available in the chart for further utilization in different operations. Thus, we have to check whether a more general edge has already been stored in the chart. In short, a new edge should be added to the chart only if no subsuming (more general) edge already exists in it.

In addition, to ensure general enough predictions we always introduce into the agenda the most general form of a given rule. The edges in its most general form would be later used and unified according to many different situations when completion steps are carryied out.

4.2 Implementation

We integrate the parse forest generation module within the context of the surface realisation algorithm presented in Section 3.1.2.

This algorithm has three major inputs: the input semantics, the grammar and the lexicon.

In the first so-called lexical selection (or lexical look-up) step, the realiser retrieves from the Tree Adjoining Grammar all the elementary trees whose semantics subsumes part of the input semantics. These trees are TAG elementary trees which are present as tree schemas in the input TAG but anchored into lexicalized trees after lexical selection.

Our filtering module works over this subset of the grammar output by the lexical selection, but mapped into RTG rules. After applying the parse forest generation algorithm, we obtain the language generated by the grammar, i.e. the set of TAG derivation trees. The TAG trees output by the lexical selection but not involved in any derivation tree output by our module are the ones that should be ruled out. In other words, we can define the output of our RTG-filtering module as the set of trees which should be the input to the next generation phases.

In the following sections, we present the Prolog predicates which implement the parse forest generation module [2]. We start by explaining the data structures. We then go on to describe the implementation of the parsing algorithm and of its main features.

4.2.1 Main data structures

RTG **grammar rules**

To translate the FB-TAG we use two tools within the open source tool set *Grammar Test Suite Generator* (gtsg) implemented by [Schmitz and Le Roux, 2008, Schmitz, 2008]. One of them is `tag2rtg`, a translation from TAG to RTG and the other is `rtg2pl` that converts the RTG into a set of Prolog files, from this set we use the one that contains the Prolog encoding of the grammar rules. In this section, we describe the data structures produced by this translation tools and which will be part of the input to our parsing module.

As has been said, Feature Based RTG rules have the following form $(A, d) \rightarrow a((B_1, d_1), ..., (B_n, d_n))$ where $A, B_1, .., B_n$ are non-terminal symbols and $d, d_1, ..., d_n$ their associated feature structures. In the FB-TAG mapping to RTG rules, the non-terminal symbols are obtained after node labelling function which assigns the label as a combination of two pieces of information: the type of tree or node and the syntactic category. Regarding the feature structures, there are two namely top and bottom which are obtained in the way described in 3.3.1.

Those non-terminal symbols and its feature structures are represented, by the translation tool, as *variables* which are a 4-tuple (as shown in Table 4.2). They are represented by a compound term, **var/4**, where the first argument (*Type*) represents the type of variable. In other words, these values, namely *aux* or *init* are derived form the type of tree (auxiliary or initial) and the type of node (adjunction or substitution site). More specifically, as we are producing the left-corner encoding the type *lc* stands for those non-terminal symbols introduced by rules coming from auxiliary trees that adjoin in root nodes. The other argument (*Cat*) represents the syntactic category of the non-terminal symbol. The last two arguments (*Bot*, *Top*) represent both bottom and top associated features structures.

var/4 **var(Type,Cat,Bot,Top)**		
Type	Prolog	Atom. Its possible atom values are **init**,**lc** or **aux**).
Cat	Prolog	Atom. Is a syntax category.
Top,Bot	Prolog	Term. The top and bottom feature structures **fs/n**.

Table 4.2: Variables (i.e. non-terminal symbols) in the RTG grammar, var/4

One important thing to point out is related to the data structure used to represent the *Top* and *Bot* features structures. As we can notice in the description

[2]In the description of some predicates we present a reduced version which includes the outstanding major parts, i.e. the core predicates. In fact, we will sometimes omit some parts including predicates that are related to secondary computation, such as write into text files or auxiliary calculation, which do not contribute to the algorithm explanation. Indeed, the included parts are the central ones required to explain the design and implementation. For a detailed description of the module and the predicates, the sources are included in one final appendix.

given in Table 4.2, the predicate specification **fs/n** suggests n as the number of arguments of predicate **fs**. This is because when using the tool (from the gtsg tool set) to translate the TAG grammar into RTG rules, it is possible to choose which features we want to keep in the resulting translation. This possibility facilitates us the implementation of the different levels of filtering discussed in Section 3.3.2.

Tree grammar rules are 5-tuples represented by the term **rule/5** (Table 4.3). The first argument is a rule number assigned by the encoding tool which will not be used by our parse forest generation module. The second argument contains the anchoring information. It is important to note that the translation from TAG to RTG is applied off-line and over the input TAG which contains the tree schematas. Therefore, this information contains the name of the tree family, the bottom feature structure of the anchor node and the *semantic schema*. Finally, the *RHS* argument represents the left-hand side non-terminal symbol and its associated features structures, i.e. a *variable* or var/4. The *RHS* argument contains a list of all right-hand side *variables*. Again, each of them is represented as argument of another term **rhs/2** generated by the translation, its first argument is a variable number not used in our implementation and the second argument is, in fact, the **var/4** structure.

rule/5
rule(RuleNumber,Tree,Anchor,LHS,RHS)

Tree	Prolog Atom. Is the name of the TAG elementary tree.
Anchor	Prolog Term. Denotes the anchoring constraints **anchor/3**.
LHS	Prolog Term. Is a *variable* (**var/4**).
RHS	Prolog Term. is a list of right-hand side *variables* of form **rhs/2**, rhs(Num,Var), where *Num* identifies the variable and *Var* provides its details.

Table 4.3: RTG grammar rules Prolog datastructure, rule/5

The parse forest generation module will take the RTG rules anchored with the *input semantics* after the lexical selection. The anchored version is loaded under the predicate **rtgRule/5**. This predicate is essentially the same as **rule/5** it just differs in the *Anchor* argument. Specifically, as the rules are anchored they keep the instantiated *semantic formula*, in contrast to the semantic schema associated with the rule schema, and omits the anchor's bottom feature structure.

Chart and agenda

The chart is represented as a Prolog knowledgebase under the term **edge/3** (Table 4.4). This predicate just represents the items as introduced in Section 4.1.1, that is, consisting of two components the dotted rule *DottedRule* and the semantic coverage (*Semantics*) plus a third argument for maintaining the current derivation tree (*DerivationTree*). In turn, the dotted rule is represented by the compound term **dottedRule/3**. The arguments in this predicate are the left-hand side of the RTG rule (*LHS*), the already covered part of the input semantics (*CompletedTree*) and the rest of the production that remains to be analysed. Graphically, that is,

$$LHS \rightarrow CompletedTree \bullet UnseeingTree$$

edge/5 **edge(?Index, ?DottedIndex, ?DottedRule, ?Semantics, ?DerivationTree)** **DottedRule= dottedRule(LHS,CompletedTree,UnseeingTree).**	
Index	Prolog Atom. Is the distinguished semantic indice associated to the tree's lexical semantics.
DottedIndex	Prolog Term. Is the distinguished index associated to the active element in the edge.
LHS	Prolog Term. Is a *variable* (**var/4**).
CompletedTree and UnseeingTree	Prolog List of terms. Lists of *variables*(**var/4**).
Semantics	Prolog List of terms. Represents the current semantic of the DottedRule, from the part of the input semantics analysed until the dot.
DerivationTree	Prolog Term. Contains the set of RTG rules or trees used in the derivation of the current edge represented by the predicate **dvTree/2** where the first argument is the name of the tree, i.e. RTG rule, and the second is a list of **dvTree/2** predicates.

Table 4.4: Chart edge Prolog representation: edge/5

Similarly, the agenda is implemented as a Prolog knowledgebase but under the predicate **taskAgenda/1** (Table 4.5). The idea is that all the edges that should be stored in the agenda will be stored as the sole data structure argument of this predicate.

taskAgenda/1 **taskAgenda(Edge)**	
Edge	Prolog Term. The Prolog predicate **edge/3** .

Table 4.5: Edges stored in the agenda are arguments of the Prolog predicate taskAgenda/1

4.2.2 The algorithm

generateParseForest/1 is the driver predicate we use in our implementation (Figure). It takes as input an *input semantics* formula. As the parse forest generation module requires the RTGgrammar, the first step is to load the grammar rules associated during the lexical selection phase with the *input semantics*, this job is carried out by the predicate **loadRTGGrammar/0**. Then, both chart and agenda structures are initialized by first removing data from previous runs and afterwards by the addition of edges done by **initializeAgenda/0** and **initializeChart**. Next, the predicate **processAgenda/0** is the core predicate implementing the Early-style chart based algorithm described in Section 4.1.1. when the processing of the agenda is finished the parsing results are extracted from the chart **successEdge/2**.

loadRTGGrammar/0 loads the knowledge base composed by the lexical selected and instantiated RTG rules, represented by **rtgRule/5**. This constitutes the grammar with which the parsing algorithm works.

Before the application of the inference rules takes place, the agenda should

```
%generateParseForest(+InputSemantics)
generateParseForest(IS):-
        /*Load RTG grammar rules and create structures*/
        loadRTGGrammar,
        agenda:emptyAgenda,
        cleanupChart,

        /*Parse*/
        initializeAgenda,
        initializeChart,
        processAgenda,

        /*Extract results*/
        successEdge(Complete,IS).
```

Figure 4.1: Parse forest generation module main predicate.

be initialized. Instead of initializing the agenda with the axiom $[S^{'} \rightarrow \bullet S_S, \emptyset]$ we start one step forward. Specifically, we initialize the agenda with the items $[S_S \rightarrow \bullet\alpha((P_1, d_1), ..., (P_n, d_n)), \psi]$ where ψ is the semantic formula associated to the RTG rule stemming from an initial tree. In Figure 4.2 we give the definition for the predicate **initializeAgenda/0** which carries out this initialization step. It goes through the grammar selects the rules with the axiom in the left-hand side and builds the edge which further stores in the agenda by means of **storeInAgenda/1** which we will explain hereafter.

```
%initializeAgenda
initializeAgenda:-
        rtgRule(Number,Tree,Anchor,var(lc,s,Bot,Top),RHS),
        Anchor =.. [anchor,_,Semantics],
        getDistinguishedIndex(Semantics,Index),
        DottedRule =.. [dottedRule,var(lc,s,Bot,Top),[],RHS],
        agenda:storeInAgenda(edge(Index,DottedRule,Semantics,dvTree(Tree,[]))),
        fail.

initializeAgenda:-true.
```

Figure 4.2: Initialization of the agenda: adding the edges corresponding to the grammar axiom.

The translation from TAG trees to RTG rules encompasses two grammar rules which are called by [Schmitz and Le Roux, 2008] *epsilon* rules. The epsilon rule ε_A with ranking 0 is mean to be used in adjunction nodes and meaning that there would be no adjunction applied. In the prolog RTG rules knowledge base it will be represented as the most general case. In addition, as a chart edge this rule is a passive edge, which we also want to keep in the chart in the most general form. As we observed that it will be often used in completion against active edges, instead of letting the prediction step to add it, we directly incorporate it into the chart. This is done by calling the predicate **initializeChart/0** that initializes the chart. The other epsilon rule ε_S with rank 1, which is mean to be used in the left-corner transformation for inverting the order of the derivations, is not explicitely translated as a chart edge. In contrast, it is implicitly applied by the predicate that verifies whether two *variables* **var/4** unify. This predicate is taken from the *gtsg* tool set implementation [Schmitz, 2008] where is used for matching the grammar *variables.* The predicate **varmatch/2** is shown in

Figure 4.3 and works as follows. In first place, we should consider that the tool for building the TAG to left-corner RTG mapping would generate the left-hand side of rules mapped from TAG initial trees and adjunction trees which adjoin in root nodes, as a **var/4** structure with the type *lc* and the corresponding category. On the other hand, the tree nodes which constitute a substitution site are translated also using the same structure but with type *init*. Then, the second clause of the predicate **varmatch/2** implements the ε_S rule $\{X_S[top : t] \rightarrow \varepsilon_S(X[top : t; bot : t]) | X_S \in N_S\}$ under the translation formalism in [Schmitz and Le Roux, 2008]. The first clause is meant for checking other cases. We will see later that this predicate is used by the deduction rules wen implementing the preconditions as described in Section 4.1.1.

```
varmatch(Var,Var) :- !.
varmatch(var(init,Cat,_,T),var(lc,Cat,T,T)) :- !.
```

Figure 4.3: Comparing RTG non-terminal symbols and their feature structures.

The agenda is manipulated by a set of operations defined over the data structure. In Figure 4.4 we show the Prolog predicates that implement this operations. The predicate **storeInAgenda/1**, which asserts the edge into the agenda, as its first action carries out the redundancy check as explained in Section 4.1.2. As the assertions are made with Prolog built-in predicate **asserta** the agenda is implemented as a stack.

```
% storeInAgenda(+Item). Push
storeInAgenda(Item):-
        ( ( taskAgenda(Item) ) -> fail ; asserta(taskAgenda(Item)) ).

% retrieveFromAgenda(-Item). Pop
retrieveFromAgenda(Item):- retract(taskAgenda(Item)).

%isEmptyAgenda.
isEmptyAgenda:-!, \+(taskAgenda(_)).

%topAgenda(+Item)
topAgenda(Item):- taskAgenda(Item),!.

%emptyAgenda
emptyAgenda:- retractall(taskAgenda(_)).
```

Figure 4.4: Agenda manipulation operations.

So far we have presented predicate's implementations that contribute to the working of the parse forest generation module. Hereafter, we describe the major predicates that implement the Earley chart based algorithm discussed in Section 4.1.1.

The *agenda based control* of chart parsing is defined by the predicate **processAgenda/0** shown in Figure 4.5. As described before, it takes one edge from the agenda (**retrieveFromAgenda/1**), process the edge by adding it into the chart and applying the inference rules (**enterEdge/1**), and process the following edge. These steps are repeated until there are no more edges in the agenda.

The predicate **enterEdge/1** (Figure 4.6) determines whether and Edge should be processed or not based on the *subsumption check* mechanism. This verification is based on Prolog 's unification mechanism and implements the following

```
%processAgenda.
processAgenda:-
        retrieveFromAgenda(Edge),
        enterEdge(Edge),
        processAgenda.
processAgenda:- true.
```

Figure 4.5: Checking before adding an item into the chart with subsumption check.

criteria. Assuming that the edges already in the chart are the most general ones, we said that there is already an edge in the chart that subsumes the current *Edge* if no variable instantiation is required in this coming *Edge*.

```
enterEdge(Edge):-
        term_variables(Edge, SVars),
        Edge,
        term_variables(SVars, SVars1),
        SVars == SVars1, !, fail.

enterEdge(Edge):-
        addToChart(Edge),
        processEdge(Edge).
```

Figure 4.6: Checking before adding an item into the chart with subsumption check.

In the first clause of enterEdge/1, the *subsumption check* is carried out. If *Edge* is already in the chart, then the clause fails without trying the second clause because of the “!” operator. Subsequently, the next edge in the agenda is evaluated. If the *Edge* is not already in the chart, the second clause of **enterEdge/1** is evaluated. What this second clause does is to call **addToChart/1** and **processEdge/1**. The first one, addToChart/1, asserts the *Edge* that has been taken from the agenda into the chart knowledge base.

The second predicate, processEdge/1 applies the *inference rules*. In Figure 4.7, we give the definition for this predicate. Its implementation is based on [Blackburn and Striegnitz, 2002] and varies in that it applies prediction only if no completion was carried out. The predicate is defined by two clauses, one of them considers the case when the *Edge* is a passive edge, the other when it is an active edge. This distinction derives from the inspection of the dot position in the edge. More precisely, this is done by looking at the third argument of the dottedRule/3 structure. As we have seen before, this argument represents the part of the derivation tree that remains to be analysed (i.e. $LHS \rightarrow CompletedTree \bullet UnseeingTree$ where $UnseeingTree$ corresponds to the third argument of the data structure). If the value is the empty list, then, this means that the entire right-hand side has been recognized and that the edge is passive, otherwise, the edge is active.

When *Edge* is a passive edge, the predicate **applyFundamentalRulePassive/1** is called. This predicate, as defined in Figure 4.8, goes through the chart by means of a fail driven loop. It looks for active edges whose active *variable* (i.e. the non-terminal/feature structures tuple after the dot) unifies with the head of the passive edge. Here, the preconditions described for the *completion inference rule* described in Section 4.1.1, $\sigma = mgu(d_i, d^{'})$, $\varphi \cap \psi = \emptyset$, are implemented

```
%We have a passive arc -> only step completion applies.
processEdge(Edge):-
        %checks whether is passive
        Edge = edge(_,dottedRule(_,_,[]),_,_),
        applyFundamentalRulePassive(Edge).

%We have an active arc -> step completion and prediction applies.
processEdge(Edge):-
        %checks whether is active
        Edge = edge(_,dottedRule(_,_,[NextNonterminal|_]),Sem,Tree),
        applyFundamentalRuleActive(Edge,Applied),
        %Only predict if it was not possible to make any completion.
        (Applied == 0 ->   predictNewEdges(NextNonterminal,Sem); true).
```

Figure 4.7: Application of the inference rules, processEdge/1 predicate's definition.

by calling to the predicate varmatch/2 and the Prolog built-in predicate intersection/3. If the precondition are applied successfully, that is both predicates succeed, then a new edge is produced and stored in the agenda. On constructing the new edge, the dot is moved one position forwards, the semantic coverage of the edge is augmented with the semantics of the combined passive edge (i.e. precondition in completion inference rule $\varphi \cup \psi = \phi$), and in the derivation tree the rule applied is added. The incorporation of the new edge in the agenda is done by calling the predicate storeInAgenda/1.

```
applyFundamentalRulePassive(edge(IndexPas,dottedRule(LHSPassive,CompletedTree,[]),
                                                        SemanticsPas,TreePas)):-

        edge(IndexActive,dottedRule(LHSActive,CompletedTreeActive,
        [rhs(_,ActiveSymbol)|UnseeingTree]),SemanticsAct,TreeAct),
        once(varmatch(ActiveSymbol,LHSPassive)),
        once(intersection(SemanticsPas,SemanticsAct,[])),
        once(flatten([CompletedTreeActive|ActiveSymbol],NewCompletedTree)),
        once(flatten([SemanticsPas|SemanticsAct],NewSemantics)),

        TreePas = dvTree(Tname,_),
        ( Tname == epsilon -> NewTree = TreeAct ;
        TreeAct = dvTree(TreeName,Descendants),
        append(Descendants,[TreePas],NewDescendants),
        NewTree = dvTree(TreeName,NewDescendants)
        ),

        once(storeInAgenda(edge(IndexActive,dottedRule(LHSActive,NewCompletedTree,
                                     UnseeingTree), NewSemantics,NewTree))),
        fail.
applyFundamentalRulePassive(_):-true.
```

Figure 4.8: Completion inference rule for passive edges, predicate's definition applyFundamentalRulePassive/1.

When *Edge* is an active edge, the second clause of processEdge/1 is applied. First, it applies the *completion inference rule* calling predicate **applyFundamentalRuleActive/2**, then, the *prediction inference rule* calling predicate **predictNewEdges/1**. The predicate regarding the application of the completion inference rule for active edges, works similarly to the one described previously for passive completion. The main difference is that this time an active edge is taken as input. Therefore, to applied the completion rule the predicate goes through the chart looking for passive edges. Again, for a passive edge to be combined the inference rule preconditions are verified. As we can see, the inference rule preconditions are applied and the new edge is constructed and stored into the agenda in the same way. A second difference is that the predicate outputs a kind of flag that indicates the number of successful combinations. The definition of this predicate [3] is shown in Figure 4.9.

```
applyFundamentalRuleActive(edge(IndexAct,dottedRule(LHSActive,CompletedTree,
          [rhs(N,ActiveSymbol)|UnseeingTree]),SemanticsAct,TreeAct),Applied):-

        Counter = counter(0),
( edge(IndexPas,dottedRule(LHSPassive,CompletedTreePassive,[]),
                                               SemanticsPas,TreePas),
        once(varmatch(ActiveSymbol,LHSPassive)),
        once(intersection(SemanticsPas,SemanticsAct,[])),
        once(flatten([CompletedTree|ActiveSymbol],NewCompletedTree)),
        once(flatten([SemanticsPas|SemanticsAct],NewSemantics)),

        TreePas = dvTree(Tname,_),
        ( Tname == epsilon -> NewTree = TreeAct ;
        TreeAct = dvTree(TreeName,Descendants),
        append(Descendants,[TreePas],NewDescendants),
        NewTree = dvTree(TreeName,NewDescendants)
        ),

        once(storeInAgenda(edge(IndexAct,dottedRule(LHSActive,NewCompletedTree,
                                        UnseeingTree), NewSemantics,NewTree))),

        TreePas = dvTree(NamePas,_),

        ( NamePas == [epsilon] -> true ;
                arg(1, Counter, N0),
                Next is N0 + 1,
                nb_setarg(1, Counter, Next)
        ),
        fail
;
arg(1, Counter, Applied)
).
```

Figure 4.9: Completion inference rule for active edges, predicate's definition applyFundamentalRuleActive/2.

In both cases, either active or passive completion application, active and passive edges are distinguished by the third argument of the data structure dottedRule/3.

The second predicate called by processEdge/1 is **predictNewEdges/2**, shown in Figure 4.10. This predicate takes two different arguments as input in order to make the predictions. The first one corresponds to the symbol after the dot in the current active edge, that is the *variable* structure var/4, and the semantic covered by the current edge. To predict new edges, it goes through the grammar

[3] There is a second clause in the definition of this predicate. This clause accounts for some non-terminal/feature structures present in the right-hand side of the RTG rules. Specifically, they are mapped from nodes in the TAG trees which account for a lemmanchor. Therefore, we give them a special treatment that is not part of the algorithm from the inference rule application point of view.

looking for a RTG rule whose head (or left-hand side) unifies with the current active symbol and whose semantics do not overlap with the semantic coverage of the current edge. In other words, applies the preconditions of the *prediction inference rule* mentioned in Section 4.1.1. Again, for doing this the predicate calls varmatch/2 and Prolog 's intersection/3. Finally, the successfully predicted edges are added into the agenda by calling storeInAgenda/1. One important thing to notice, is that those rules which have the terminal symbol in the right-hand side with ranking 0 are incorporated into the process as passive edges whenever they are predicted.

```
predictNewEdges(rhs(_,NonTerminalSymbol),Sem):-

        rtgRule(Number,Tree,Anchor,Var,RHS),
        %epsilon edges should not be introduced in any turn as
        the are already in the chart in its more general form.

        \+(Tree == epsilon),
        \+(\+ once(varmatch(NonTerminalSymbol,Var))),

        Anchor =.. [anchor,_,Semantics],
        getDistinguishedIndex(Semantics,Index),
        DottedRule =.. [dottedRule,Var,[],RHS],

        %Predict only those edges whose semantic coverage does not overlap
        with the current active edge that need to be completed.
        once(intersection(Semantics,Sem,[])),

        %Add the edge into the agenda.
        once(agenda:storeInAgenda(edge(Index,DottedRule,Semantics,dvTree(Tree,[])))),

        fail.

predictNewEdges(_,_):-true.
```

Figure 4.10: Prediction inference rule, predicate's definition predict-NewEdges/2.

Chapter 5

Results and evaluation

The RTGencoding of the grammar used for filtering encompasses almost all the information present in the grammar used by the surface realisation algorithm. Depending on how much of this information is included in RTG-filtering, different levels of filtering can be tested. In this chapter, we present an evaluation of the RTG-filtering introduced in Section 3.3.2 and implemented as described in the previous chapter. We show how this approach permits reducing the initial search space. Furthermore, we compare this approach with the polarity filtering previously proposed in [Gardent and Kow, 2005].

The structure of the chapter is as follows. We start (Section 5.1) by introducing the methodology used for the evaluation. We discuss the grammar being tested, the type of inputs we focus on and the evaluation metrics involved. In 5.2, we present the results obtained systematically comparing the RTG-filtering with the polarity filtering approach. Section 5.2 also identifies some current limitations and possibilities for further improvement.

5.1 Methodology

Grammar

As mentioned above, the TAG-based surface realiser takes as input a flat semantic formula, a grammar and lexicon. The grammar used to test our filtering approach is a wide-coverage FB-LTAGdeveloped in Nancy. Two such grammars are in fact currently available, one for French [Gardent, 2008] and one for English. However the grammar for English is still in its early development stages and has not been thoroughly tested. On the other hand, the grammar for French has already been used for several medium scale experiment. Moreover it comes with a test suite of semantic input derived from the "Test Suites for Natural language Processing" (TSNLP) which is not available for English. For these reasons, we decided to base our RTG-filtering experiments on the grammar for French (called SemFraG).

Test cases

The test suite consists of a set of sentences encompassing various language phenomena. Each sentence is associated with the *flat semantic formulae* assigned by the grammar to this sentence.

From this test suite, we selected some specific cases based on features that are relevant for evaluating the impact of our RTG-filtering method. Those features include the number of finite verbs present in the sentence verbalising the input semantics and similarly, the presence or not of adjectives (because they are modelled as auxiliary trees), the valency of the verbs (to vary the number of substitution nodes involved) and the number of relative clauses (to simultaneously increase the length of the input formula and the number of auxiliary trees involved).

Comparison and evaluation metrics

To evaluate our approach, we systematically compare the results obtained by the polarity filtering method on the input selected from the test suite with those obtained by the polarity filtering technique implemented in GenI [Gardent and Kow, 2005]. The GenI surface realiser (described in Section 3.1.2) is implemented in Haskell and incorporates the polarity filtering approach[1]. It includes a graphical interface as well as a debugger which we use to obtain the respective results. We run both tools with the same grammar and same test cases.

We compare both approaches with respect to different test cases and different metrics.

At a first coarser granularity level, we examine the impact of both filtering techniques on the total number of elementary trees involved in the processing of a given input. More precisely, we compare the number of trees output by the parse forest generation module, i.e. those trees that were involved in a successful derivation, with the number of trees retrieved by the lexical selection phase. We first show the amount of filtering done by the different levels in our RTG-filtering approach. Following what has been suggested when we introduce the approach, we filter at a *first level* considering only the category, at a *second level* considering the category and the semantic indices, and at last, as *third level*, we filter with a selection of features (i.e. linguistic information) present in the feature structures. Further, we also contrast these results with the number of trees left by the polarity filtering approach.

At a finer granularity level, we examine the impact of filtering on the number of elementary trees per input literal. In this way, we get a more precise estimation of the impact of filtering on the combinatorics of the initial search space. Since the initial search space includes all the possible combinations of the trees per input literal ($\Pi_{1 \leq i \leq n} a_i$ with a_i with a_i the degree of lexical ambiguity of the i-th literal and n the number of literals in the input semantics), counting the number of elementary trees remaining after filtering and comparing it with the same number before filtering, permits assessing the impact of filtering on the initial combinatorics. Again, the RTG-filtering approach is compared against the outputs of both the lexical selection phase (no filtering) and the polarity filtering.

Additionally, we compare the results of the two filtering strategies with regard to the following features:

- the number of initial and auxiliary trees ruled out

[1]The GenI surface realizer is available on: `http://trac.loria.fr/~geni/`

- the number of trees ruled out due to feature structure unification failure, i.e. top/bottom unification failure,
- the size of the input semantics (specifically in terms of verbal lexical items),
- root constraints

5.2 Analysis of the results

To carry out the comparisons we have chosen the following representative examples (9) from the test suite and the three different levels of filtering considered as shown in (10).

(9) a. test case: t20v
Semantics:{a:aimer(b),a:agent(b,c),a:patient(b,d),e:jean(c),f:marie(d)}
Sentence: *Jean aime Marie*
(*Jean loves Marie*)

b. test case: t290v
Semantics:{a:demander(b),a:agent(b,c),a:recipient(b,c),a:topic(b,d), e:venir(d),e:agent(d,f), g:paul(f),h:jean(c)}
Sentence: *Jean se demande si Paul viendra*
(*Jean asks himself whether Paul will come.*)

c. test case: t50p
Semantics: {a:avoir(b),a:agent(b,c),a:patient(b,d),e:ingénieur(c), f:le(c),g:décision(d),a:un(d),h:intelligent(d)}
Sentence: *L'ingénieur a une décision intelligente*
(*The ingeneer has an intelligent desicion*)

d. test case: t20rs2
Semantics:{a:partir(b),a:agent(b,c),d:homme(c),e:le(c),f:aimer(g), f:agent(g,c),f:patient(g,h),i:marie(h)}
Sentence: *L'homme qui aime Marie part*
(*The man who loves Marie leaves*)

e. test case: t20rs3
Semantics:{a:partir(b),a:agent(b,c),d:jean(c),e:dire(f),e:agent(f,g), e:patient(f,b),h:homme(g),i:le(g),j:aimer(k),j:agent(k,g),j:patient(k,l), m:marie(l)}
Sentence: *L'homme qui aime Marie dit que Jean part*
(*The man who loves Marie says that Jean leaves*)

f. Some examples addapted from the test suit to compare with examples from [Gardent and Kow, 2005]:

i. test case: s7a
Semantics:{a:auditionne(b),a:agent(b,c),a:patient(b,d),e:directeur(c), f:le(c), g:consultant(d),a:un(d),h:nouveau(d)}
Sentence: *Le directeur auditionne un nouveau consultant*
(*The director auditions a new consultant*)

ii. test case: s7b
Semantics:{a:auditionne(b),a:agent(b,c),a:patient(b,d),e:directeur(c), f:le(c),g:consultant(d),a:un(d),h:nouveau(d),i:religieux(d),p:chinois(d)}

Sentence: *Le directeur auditionne un nouveau consultant religieux d'origin chinois*
(*The director auditions a new religious consultant of chinese origin*)

(10) a. *rtg-Level1*: syntactic category, i.e. {feature `cat` }.

b. *rtg-Level2*: syntactic category and semantic indice, i.e. features { `cat` and `idx` }.

c. *rtg-Level3*: a selection of linguistic features { `aux, aux-pass, aux-refl, case, cest, cleft, control-gen, control-num, control-pers, cop, ctrl-gen, ctrl-num, ctrl-pers, def, det, func, gen, idx, invertedCliticIdx, label, lemanchor, loc, neg-adv, neg-nom, nomPropre, num, pp-gen, pp-num, princ, refl, wh, whIdx, zeroObjectIdx, zeroSubjectIdx` }.

Impact of filtering on the number of elementary trees in the initial search space. Table (5.1) shows the impact of filtering on the number of elementary trees present in the search space before filtering ("LS output" column) and after filtering using either polarity filtering or the RTG-based method. As can be seen both filtering approaches have a strong impact on this number thereby drastically reducing the size of the search space. Some additional comments are in order here.

Test case	LS output	*rtg-Level1*	*rtg-Level2*	*rtg-Level3*	Polarity filtering
t20v 9a	94	16	16	16	16
t290v 9b	178	48	21	14	48
t20rs2 9d	109	32	28	27	35

Table 5.1: Comparing the number of elementary trees left as input to the realisation phase by the different approaches.

First of all, we focus on the three columns regarding the RTG-filtering approach levels. We can see that the more linguistic information the filter includes the better are the results in term of trees ruled out. By increasing the numbers of features translated into the RTG grammar, the advantage is twofold. On one hand, because of the restrictions that the feature structures apply over the grammatical categories the number of combinations among edges decreases, and thus, the number of edges handled by the derivation process. On the other hand, the final successful combinations output by the derivation process is more precise, this means they are close to the set that the realisation phase might generate.

Regarding the results obtained by the two filtering approaches, there are some interesting differences. In the first place, we notice that for the *rtg-Level1* and the polarity filtering columns, the results are almost the same. Indeed, as we argue in Section 3.3.2, both consider only the syntactic categories.

Next, we look at the results obtained for each different test case (i.e. rows in the table). We can observe that for the first example (9a) all the levels of filtering have equal results to polarity filtering. From the experimentation we have carried out, cases such as the first one, show almost the same results. The reason is that these cases are quite simple cases, comprising only one tree for each proper noun and the set of trees for the tree families associated to the main verb. For the other two cases (9b,9d) however, the filtering impact increases together with the different levels of RTG-filtering. Specifically, *rtg-Level3* compares favourably to polarity filtering in both cases. On one hand, the second test case (9b) increases the difficulty by including two verbs from which one is di-transitive. Now, the combinations between the two sets of trees is evaluated. The difference in the results obtained for this case is due to the linguistic information included in *rtg-Level3* which is not present in polarity filtering. For instance, Figure (5.1) shows two edges extracted from the chart, the active edge associated to the verb *demander* fails to unify its top/bottom features structures with the passive edge for *Jean viendra* because of the feature `wh`. In polarity filtering, as the charges in terms of categories of these trees are balanced the trees are not filtered out. The third case (9d), on the other hand, adds a relative clause and contains two verbs. The difference in the result stems from the filtering of some auxiliary trees done by *r*tg-Level3 which are not ruled out by polarity filtering. During the derivation process done by RTG-filtering, the rules mapped from these auxiliary trees are even never predicted. In contrast, as the polarities of auxiliary trees are neutralized (the foot and root node of such trees being of the same category, they cancel out each other in terms of resources and requirement i.e., polarity), they survive the filtering and are not filtered out.

```
passive edge:
edge(d, dottedRule(var(lc, s, fs(..., idx:d, _, label:e, ..., wh:-, ...), _),
[], []), [g:literalSchema(paul, [f]), e:literalSchema(venir, [d]),
e:literalSchema(agent, [d, f])],
dvTree(Tn0V-1277d, [dvTree(Tpropername-6448f, [])]))

active edge:
edge(b, dottedRule(var(lc, s, fs(..., idx:b, ..., wh:-, ...), _),[],[rhs(26,
var(init,s,_,fs(..., princ:-, _, wh:+, zeroObjectIdx:d, _, zeroSubjectIdx:c)))]),
[h:literalSchema(jean, [c]), a:literalSchema(demander, [b]), a:literalSchema
(agent,[b,c]), a:literalSchema(topic,[b,d]), a:literalSchema(recipient,[b,c])],
dvTree(Tn0Vs1intan2-2267b, [dvTree(Tpropername-6448c, [])]))
```

Figure 5.1: Completion failure, top and bottom features structures do not unify.

Impact of filtering on the number of elementary trees per literal present in the input semantics. So far, we compared the total number of trees left by both approaches. We now examine how those numbers divide per lexical item to gain a clearer view of how filtering affects the combinatorics of the initial search space (not just the size).

In Table 5.2, we give the details of two test cases. For our analysis here, we will focus only on those lexical items whose lexical ambiguity is greater than 1. This is because we are considering the combinatorics of the trees per lexical item. Therefore, there are $162 \times 14 = 2268$ possible combinations for the first example (9b) and $94 \times 8 \times 10 = 7520$ for the other example (9c). Looking at the last column in the tables, we can see the effect of both polarity filtering and RTG-filtering. We see in particular, that RTG-filtering out-performs polarity filtering by a factor of 8 in the first case (example t290v). This could be further improved by adding a constraint that the GenI surface realiser imposes on the root nodes of successful derived trees namely that successful derived trees should describe finite (indicative or subjunctive mode), non inverted, non questioned sentences .

Literals t290v (9b)	demander(b)	venir(d)	$\Pi_{1 \leq i \leq n} a_i$
Lexical selection output	162	14	2268
Polarity filtering	40	6	240
RTG filtering	7	5	35

Literals t50p (9c)	avoir(b)	ingénieur(c)	intelligent(d)	$\Pi_{1 \leq i \leq n} a_i$
Lexical selection output	94	8	10	7520
Polarity filtering	18	1	2	36
RTG filtering	14	1	2	28

Table 5.2: Comparing reduction per lexical item and the effect in the combinatorics.

Impact of filtering on the number of auxiliary trees present in the search space. In Table 5.3, we show two examples where the difference in the results obtained by both approaches stems from the auxiliary tree filtering. While RTG-filtering rules out 100% of the auxiliary trees output by the lexical selection and not involved in derivations (i.e. 28 - 6), polarity filtering filter out only 68% and 50% of the auxiliary trees for the two cases respectively. The reason why polarity filtering filters out auxiliary trees at all is that some of these are hybrid trees involving both a foot node and a substitution site. Contrary to "pure" auxiliary trees i.e. trees not containing a substitution nodes, such trees can be ruled out by polarity filtering. Hence the numbers.

	LS Output	In derivation	RTG-filtering	Polarity filtering
t20rs2 9d	28	6	22	15
t20rs3 9e	52	4	48	24

Table 5.3: Comparing filtering by auxiliary trees.
The first two columns indicate the number of auxiliary trees present in the initial search space and in the successfully generated sentences. The last two columns indicate the number of auxiliary trees filtered by the two filtering methods.

Impact of filtering on the number of initial trees present in the search space. Table 5.4 shows two examples with the difference in the results obtained by both approaches. But in this case, the difference stems from initial tree filtering. RTG-filtering excludes the 100% of initial trees output by the lexical selection and not involved in derivations (i.e. 148 - 13) whereas polarity filtering filter out 68% and 54% of the initial trees for the two cases respectively.

Impact of filtering on the number of tree sets to be considered for combination. Further, we examine the impact of filtering on the number of tree sets that will be considered for combination. In polarity filtering, such tree

	LS Output	In derivation	RTG-filtering	Polarity filtering
t290v 9b	148	13	134	101
t20rs3 9e	143	5	138	80

Table 5.4: Comparing filtering by initial trees.
The first two columns indicate the number of auxiliary trees present in the initial search space and in the successfully generated sentences. The last two columns indicate the number of auxiliary trees filtered by the two filtering methods.

sets are defined by the polarity automaton : it is the set of elementary trees that label the transitions of a given path in the automaton. In the RTG-filtering approach, tree sets are defined by the derivation trees produced by the filtering. For the comparisons in Table 5.5, the grammar used for testing polarity filtering [Gardent and Kow, 2005] was not exactly the same as the version we are using for our testing. Therefore, we included their lexical selection results, that is, the lexical item combinations actually explored according to their grammar. Similarly, as our example for 9f-ii contains only 3 adjectives, we suppose the existence of another adjective to obtain the fourth adjective used in their second example. As adjectives will have the same set of associated trees, and adjectives in our grammar have 10 trees, we multiply the result by 10 for the fourth adjective. Accordingly, we estimate the reduction in $14 \times 2^4 = 224$ as we observed that for two and three modifiers the number of trees left equals 2.

Test case	LS output	*rtg-Level3*	LS output*	Polarity filtering*
s7a 9f-i	850	28	856	55
s7b 9f-ii	9350	48	n/a	n/a
s7b 9f-ii (3Adj)	93500	112	n/a	n/a
s7b 9f-ii (4Adj)	935000	224	438272	232

Table 5.5: Comparing the number of trees left as input to the realisation phase by the different approaches.

As can be seen the RTG-filtering out performs polarity filtering by decreasing the number of tree sets to be considered for combination more markedly (28 tree sets left against 55 in the first case and 224 against 232 in the second).

Impact of filtering on the number of intermediate structures built. Further, we analyse the effect of including more linguistic information in the RTG grammar regarding the parse process and the corresponding chart and agenda sizes. In Table 5.6, we can see that in *rtg-Level1* the fact that no feature structure information is included results in over generation. In particular, because of not including the semantic indices associated to the top/bottom features structures of the nodes which guide the tree combination. The size of the number of edges produced is much greater than for levels 2 and 3. Even though many different alternations of the possible combination are produced, the resulting set of succesfully combined trees correctly corresponds to the polarities over categories criteria. In the second level, *rtg-Level2*, we include the `idx` (i.e. semantic indice feature). In addition to get some trees filtered out, the size of both chart and agenda decreases. Again, both filtering percentage and chart agenda size decrease after the addition of a bigger set of features, that is, *rtg-Level3*.

More specifically, we observe that the precondition in the prediction rule is most effective when more that one verb is present in the semantics. Thus after adding this precondition, the agenda number of edges processed through the agenda was reduced from 56570 for the test case 9b to 4193.

Test case	*rtg-Level1*		*rtg-Level2*		*rtg-Level3*	
	chart	agenda	chart	agenda	chart	agenda
t50p 9c	34914	35762	3388	3634	1576	1707
t20rs2 9d	9253	9457	2895	2959	1638	1682

Table 5.6: Comparing the chart and agenda size through the different levels of RTG-filtering.

One important thing to note is that neither polarity filtering nor RTG-filtering will generate false positives results. That is, they do not filter out trees because of not finding a way to sucessfully combine them when they actually could have been used into a complete derivation.

Chapter 6

Conclusion

6.1 Summary

The leading motive of the work in this thesis was surface realisation optimisation. In particular, optimisation within the context of reversible realisers based on wide-coverage lexicalized grammar formalism and generation from flat semantics. Our work addressess the *lexical ambiguity* problem and its negative influence in the initial search space for a TAG-based surface realiser. We describe a *global* filtering approach based on [Schmitz and Le Roux, 2008]'s Feature Based RTG formalism which encodes TAG derivation trees as the language of the grammar. To implement this filtering approach, we develop a parse forest generation module. From the set of derivation trees obtained, we can identify which are the elementary structures which lead to successful derivations and which ones fail to be combined in any possible derivation. Then, those useless structures are ruled out reducing the initial search space for the realisation phase, that is we prevent the realisation from dealing with spurious structures.

The algorithm that generates the parse forest is based on well known parsing techniques: *chart parsing*, *Earley algorithm* and *deductive parsing*, but adapted for generation in the way suggested by Kay. As a result, the algorithm incorporates some mechanims inherent to these approaches which are crucial in our specific problem namely, *tabulation of intermediate results*, *agenda-based control*, *indexing*, and *semantic filter*. Furthermore, we incorporated additional mechanisms that were required by the specific type of grammar used namely, unification for non-atomic grammar categories and subsumption based blocking of new edges.

We have implemented in Prolog the parse forest generation module in the context of the TAG-based surface realiser presented in Section 3.1.2. Both the input grammar formalism and the Feature Based RTG formalism used for filtering (Sections 3.1.1, 3.3.1) make heavy use of both feature structures and unification variables shared across the semantic representations and the feature structures labelling the nodes of the syntactic trees. In this context, an advantage of a Prolog implementation is that it can take full advantage of Prolog 's built in unication mechanism.

Once the implementation was completed, we evaluated the impact of the filtering approach by analysing different cases taken from the test suite developed for the surface realiser. As was expected, we found that filtering with different levels of linguistic information yield different results whereby the more linguistic information is included in the filter, the more *precise* the reduction of the spurious trees and the more *efficient* the parse generation process.

Next, we compared the RTG-filtering approach with the polarity filtering proposed for the same grammar and surface realiser by [Gardent and Kow, 2005]. For the simpler cases, the amount of filtering was found to be the same for both strategies. For more complex cases (e.g., longer input formulae, cases involving adjunctions) however, RTG-filtering consistently out performs polarity filtering. This is explained by the following facts. First, while polarity filtering fails to filter auxiliary trees, our approach applies to initial as well as to auxiliary trees. Second, while our approach takes into account the syntactic categories, semantic information and feature structures present in the grammar, polarity filtering only considers polarity keys based on syntactic categories.

As explained in Chapter 3, the exponential complexity of surface realisation can be traced back in particular, to the semantics of intersective modifiers, lexical ambiguity and the lack of ordering in the input. Although the work presented here focuses on reducing lexical ambiguity, a side effect of the RTG filtering approach is that by not considering the information about the derivation trees in the edges it also helps to reduce the number of intermediate structures licensed by intersective modifiers as tabulation permits avoiding recomputing two intermediate structures with identical semantics (e.g., *small black cat* and *black small cat*).

6.2 Future work

Future work could investigate the following two main issues : (i) devising additional mechanisms to make the parse forest generation as efficient as possible from the design and implementation points of view; (ii) exploring strategies about how to integrate the RTG-filtering filtering approach with the surface realiser algorithm in oder to achieve an efficient surface realisation.

As for the mechanisms to incorporate in our parse forest generation algorithm, we are planning to analyse the benefits of:

- Reorder the variables in the left hand-side in such a way that uninstantiated variables are handled as late as possible thereby reducing non determinism.
- More sophisticated indexing schemes for the chart.
- Off-line production of the derivations.
- Packing.
- Intersective modifiers.
- Reduce the right-hand side of RTG rules to generate less chart items. For instance, processing sequences of right-hand side *variables* which are closed by epsilon rules because there is no other possible combination. The information that their features structures could provide is transfered through unification to the unrecognized part of the rule and the number of items would amount to those points where other combinations than epsilon ones can be performed.

Regarding Prolog 's implementation, an obvious next step would be to incorporate built-in clauses indexing on given arguments, based for instance on the semantic indices used to index chart edges. Another improvment would involve implementing a bit vector representation of the items semantic coverage instead of working with sets of literals represented as predicates **literal/2**

Bibliography

[Benotti, 2009] Benotti, L. (2009). Frolog: an accommodating text-adventure game. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 1–4, Athens, Greece. Association for Computational Linguistics.

[Blackburn and Striegnitz, 2002] Blackburn, P. and Striegnitz, K. (2002). Natural language processing techniques in prolog. `http://cs.union.edu/~striegnk/courses/nlp-with-prolog/html/index.html`.

[Brew, 1992] Brew, C. (1992). Letting the cat out of the bag: generation for shake-and-bake mt. In *Proceedings of the 14th conference on Computational linguistics*, pages 610–616, Morristown, NJ, USA. Association for Computational Linguistics.

[Carroll et al., 1999] Carroll, J., Copestake, A., Flickinger, D., and Paznański, V. (1999). An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of EWNLG '99*.

[Carroll and Oepen, 2005] Carroll, J. and Oepen, S. (2005). High efficiency realization for a wide-coverage unification grammar.

[Comon et al., 2007] Comon, H., Dauchet, M., Gilleron, R., Loding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

[Copestake et al., 1999] Copestake, A., Flickinger, D., Sag, I. A., and Pollard, C. (1999). Minimal recursion semantics: an introduction. Draft.

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

[Gardent, 2008] Gardent, C. (2008). Integrating a unification-based semantics in a large scale lexicalised tree adjoininig grammar for french. In *COLING'08*, Manchester.

[Gardent and Kallmeyer, 2003] Gardent, C. and Kallmeyer, L. (2003). Semantic construction in feature-based tag. In *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 123–130, Morristown, NJ, USA. Association for Computational Linguistics.

[Gardent and Kow, 2005] Gardent, C. and Kow, E. (2005). Generating and selecting grammatical paraphrases. In *Proceedings of the 10th European Workshop on Natural Language Generation*, Aberdeen, Scotland.

[Gardent and Kow, 2006] Gardent, C. and Kow, E. (2006). Three reasons to adopt tag-based surface realisation. In *TAG06*.

[Gardent and Kow, 2007a] Gardent, C. and Kow, E. (2007a). Spotting overgeneration suspects. In *Proceedings of the 12th European Workshop on Natural Language Generation*, Dagstuhl.

[Gardent and Kow, 2007b] Gardent, C. and Kow, E. (2007b). A symbolic approach to near-deterministic surface realisation using tree adjoining grammars. In *ACL07*, Prague.

[Joshi and Schabes, 1997] Joshi, A. and Schabes, Y. (1997). Tree-adjoining grammars.

[Kay, 1986] Kay, M. (1986). Algorithm schemata and data structures in syntactic processing. pages 35–70.

[Kay, 1996] Kay, M. (1996). Chart generation. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 200–204, Morristown, NJ, USA. Association for Computational Linguistics.

[Koller and Striegnitz, 2002] Koller, A. and Striegnitz, K. (2002). Generation as dependency parsing. In *Proceedings of the 40th ACL*, Philadelphia.

[Kow, 2007] Kow, E. (2007). *Surface realisation: ambiguity and determinism.* PhD thesis, Université Henri Poincaré.

[Matthiessen et al., 1991] Matthiessen, C. M. I. M., Bateman, J. A., and Patten, T. (1991). Text generation and systemic-functional linguistics: Experiences from english and japanese.

[Reiter and Dale, 2006] Reiter, E. and Dale, R. (2006). *Building Natural Language Generation Systems.* Studies in Natural Language Processing. Cambridge University Press, New York.

[Schmitz, 2008] Schmitz, S. (2008). Grammar test suite generator. Available on: `svn://scm.gforge.inria.fr/svn/paule/trunk/gtsg`.

[Schmitz and Le Roux, 2008] Schmitz, S. and Le Roux, J. (2008). Feature unification in TAG derivation trees. In Gardent, C. and Sarkar, A., editors, *Proceedings of the 9th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+'08)*, pages 141–148, Tubingen, Germany.

[Shieber, 1988] Shieber, S. M. (1988). A uniform architecture for parsing and generation. In *Proceedings of the 12th conference on Computational linguistics*, pages 614–619, Morristown, NJ, USA. Association for Computational Linguistics.

[Shieber et al., 1995] Shieber, S. M., Schabes, Y., and Pereira, F. C. N. (1995). Principles and implementation of deductive parsing. *J. Log. Program.*, 24(1&2):3–36.

Appendix A

Surface realization input resources

In this appendix we describe the format of the three major inputs to the realiser persented in 3.1.2: the *input semantics*, the *grammar* and the *lexicon*.

The input semantics

The semantic formula input to the realiser represents the meaning to be verbalized. As mentioned in Section 3.1.1, the surface realiser assumes a flat semantic representation so that a semantic formula is a set of literals where each literal consists of a predicate and some indices representing its arguments. More precisely, we refer to these indices as *semantic parameters* when the semantic index value is missing and furthermore, they will be represented by unification variables. When unification variables are present in a literal we may refer to it as *literal schema* or *semantic schema*. Notice here that there are no literals within literals: the formulae are "flat', this means non recursive.

We show for the sentence in (11) an example of a flat semantics input format:

(11) *"jean semble partir"*
{*partir(b), agent(b,c), jean(c), sembler(b)*}

And in (11) we give an example semantic schema, a formula with variable predicates:

(12) *Semantics for intransitive verbs*
{*P(e), Theta(e,x))*}

The grammar

As mentioned previously, the input grammar used for the evaluation is SemFraG(Section 5.1). This grammar is compiled from a higher-level XMG (eXtensible MetaGrammar) specification (a so-called "meta-grammar"). Briefly, the XMG formalism permits specifying basic classes and then combining them (by inheritance, conjunction and disjunction) to produce SemFraG elementary trees and their associated semantics [Gardent and Kallmeyer, 2003]. Then, a compiled TAG consists of a list of entries, each representing an unanchored tree (tree schema) of the grammar (Figure A.1). More precisely, each entry contains the following data:

- a unique name of the form *TFamilyName-Id*
- a family name determining to which tree-family it belongs,
- a trace, which corresponds to the list of the classes that have been accumulated to produce this entry,
- a syntactic description, giving the structure of the tree (a tree schema for TAG),
- a semantic description (possibly empty), expressed in flat semantics (set of literals),
- an interface consisting of an attributes-values matrix whose features are shared with features in the tree (used for anchoring).

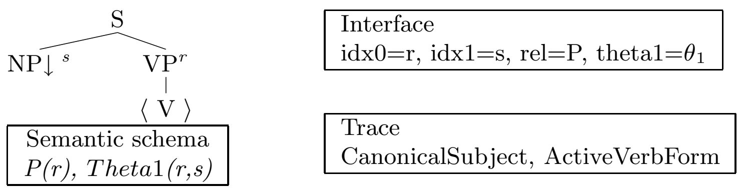

S
NP↓ s
VP r
⟨ V ⟩
Semantic schema
P(r), Theta1(r,s)
Interface
idx0=r, idx1=s, rel=P, theta1=θ1
Trace
CanonicalSubject, ActiveVerbForm

Figure A.1: TAG tree entry associated to intranditive verbs (e.g. “courir”)

The lexicon

The lexicon, SYNTACTIC LEXICON, associates semantic representations with lexical items and trees. The lexicon associated to SemFraG contains the following information:

***ENTRY:** used to store the lexical item (i.e. lemma),

***CAT:** syntactic category,

***SEM:** semantic information (at this time of writing a macro call, note that these macros are extracted automatically from the metagrammar by XMG),

***FAM:** the family (that is, subcategorization frame)

***EQUATIONS:** anchoring equations, of the form:
node → [top—bot.]feat = val

***COANCHORS:** coanchor equations, of the form:
node → lemma / category

These equations are used to specify a lexical item that has to be added in the tree. The format also includes two other fields which at the moment are unused, they are: *ACC:verb acceptance (for word having several meanings such as: parler, example: jean parle anglais ; jean parle à marie); and *EX: list of exceptions (in tagml this is a features list having the value ”-”).

Appendix B

Scripts summary

Test suite -input test file format

For testing our lexical selection algorithm we will use use an already developed test suite (as explained in Section 5.1). It consists of a set of sentences with their associated semantic reprsentation. The **testSuiteEntry/3** (Table B.1) fact will represent the test suite entries within our Prolog knowledge base (`gen.pl`). The facts listed below are taken from this knowledge base as example and in correspondace with the examples used in Section 5.2.

testSuiteEntry/3 **testSuiteEntry(identifier,semantics,sentence)**	
identifier	atom.
semantics	list of label:literal pairs.
label	prolog term: atom.
literal	prolog term: compound term, with atoms as arguments.
sentence	atom.

Table B.1: Test suite knowledge base format.

```
testSuiteEntry(t20v,[a:aimer(b),a:agent(b,c),a:patient(b,d),e:jean(c),f:marie(d)],
'jean aime marie').

testSuiteEntry(t290v,[a:demander(b),a:agent(b,c),a:recipient(b,c),a:topic(b,d),e:venir(d),
e:agent(d,f), g:paul(f),h:jean(c)],'jean se demande si paul viendra').

testSuiteEntry(t50p,[a:avoir(b),a:agent(b,c),a:patient(b,d),e:ingnieur(c),f:le(c),
g:dcision(d),a:un(d), h:intelligent(d)],'l ingnieur a une dcision intelligente').

testSuiteEntry(t20rs2,[a:partir(b),a:agent(b,c),d:homme(c),e:le(c),f:aimer(g),f:agent(g,c),
f:patient(g,h),i:marie(h)],'l homme qui aime marie part').

testSuiteEntry(t20rs3,[a:partir(b),a:agent(b,c),d:jean(c),e:dire(f),e:agent(f,g),
e:patient(f,b),h:homme(g),i:le(g),j:aimer(k),j:agent(k,g),j:patient(k,l),m:marie(l)],
'l homme qui aime marie dit que jean part').

testSuiteEntry(s7a,[a:auditionne(b),a:agent(b,c),a:patient(b,d),e:directeur(c),f:le(c),
g:consultant(d),a:un(d),h:nouveau(d)],'le directeur auditionne un nouveau consultant').

testSuiteEntry(s7b,[a:auditionne(b),a:agent(b,c),a:patient(b,d),e:directeur(c),f:le(c),
g:consultant(d),a:un(d),h:nouveau(d),i:religieux(d),p:chinois(d)],'le directeur
auditionne un nouveau consultant religieux d'origin chinois').
```

Agenda manipulation operations (agenda.pl)

```
:-module('agenda', [storeInAgenda/1,
                    retrieveFromAgenda/1,
                    isEmptyAgenda/0,
                    emptyAgenda/0,
                    topAgenda/1]).

:-use_module('rtgFilter',[edge/4]).

:- dynamic taskAgenda/1.

% storeInAgenda(+Item). Push
storeInAgenda(Item):-
        ( ( taskAgenda(Item) ) -> fail
        ;
        asserta(taskAgenda(Item)),
        /*Dump agenda item*/
        util:file(agendaHistory,_,_,Stream),
        write(Stream,Item),
        nl(Stream),nl(Stream)
        /**/
        ).

% retrieveFromAgenda(-Item). Pop
retrieveFromAgenda(Item):-
        retract(taskAgenda(Item)).

isEmptyAgenda:-!,
        \+(taskAgenda(_)).

topAgenda(Item):-
        taskAgenda(Item),!.


emptyAgenda:-
        retractall(taskAgenda(_)).
```

Parse forest generation module (rtgFilteringWI.pl)

```
% Exported predicates:
:- module(rtgFilterWI,[generateParseForest/1]).

% Imported predicates:
:- use_module('Tools/util',[memo/1,
                openFile/3,
                closeFile/1,
                file/4]).

:- use_module('agenda', [storeInAgenda/1,
                        retrieveFromAgenda/1,
                        isEmptyAgenda/0,
                        emptyAgenda/0,
                        topAgenda/1]).

:- dynamic edge/4.

/*===========================================
generateParseForest(+InputSemantics)
Loads the rtg rules associated by the lexical selection with the input semantics,
initializes the chart and agenda data structures, starts the parsing process, and finally,
extracts the parsing results.
===========================================*/
generateParseForest(IS):-
        /*Load RTG grammar rules and create structures*/
        loadRTGGrammar,
        agenda:emptyAgenda,
        cleanupChart,

        util:openFile(chart,'chart.txt',write),
        util:openFile(agendaHistory,'agendaHistory.txt',write),

        /*Parse*/
        initializeAgenda,
        initializeChart,
        processAgenda,

        /*Extract results*/
        util:openFile(derivationTrees,'derivationTrees.txt',write),
        util:openFile(incompleteDerivationTrees,'incompleteDerivationTrees.txt',write),
        util:file(derivationTrees,_,_,Complete),
        util:file(incompleteDerivationTrees,_,_,Incomp),
        util:file(chart,_,_,StreamChart),
        successEdge(Complete,IS),
        incompleteEdge(Incomp,IS),
        util:file(agendaHistory,_,_,AgendaHistory),
        util:closeFile(AgendaHistory),
        util:closeFile(Complete),
        util:closeFile(Incomp),
        util:closeFile(StreamChart).

/*============================================
loadRTGGrammar
Loads the rtg rules associated by the lexical selection with the input semantics
============================================*/
loadRTGGrammar:-
        retractall(rtgRule(_,_,_,_,_)),
        /*Loads the RTG rules selected after the lexical selection phase according
        to the input semantics*/
        util:file(selectedTAGtreesAsRTG,FileName,Mode,Stream),
        load_files(FileName, [if(changed)]),
        clearFeaturesInfo,
        setFeaturesInfo.

/*============================================
cleanup
Cleans the database of chart entries.
============================================*/
cleanupChart :-
        retractall(edge(_,_,_,_)).

/*============================================
initializeChart
Cleans the database of chart entries.
============================================*/
initializeChart:-
        DottedRule =.. [dottedRule,var(aux, _, FS, FS),[],[]],
        addToChart(edge(Index,DottedRule,[],dvTree(epsilon,[]))).

/*============================================
initializeAgenda
Make initial agenda: i.e., an agenda that only contains those rules of the grammar whose
left non-terminal symbol is the grammar axiom S.
============================================*/
initializeAgenda:-
        rtgRule(Number,Tree,Anchor,var(lc,s,Bot,Top),RHS),
        Anchor =.. [anchor,_,Semantics],
        getDistinguishedIndex(Semantics,Index),
        DottedRule =.. [dottedRule,var(lc,s,Bot,Top),[],RHS],
        concat(Tree,Index,IDTree),
        agenda:storeInAgenda(edge(Index,DottedRule,Semantics,dvTree(IDTree,[]))),
        fail.

initializeAgenda:-true.

/*============================================
processAgenda
Produces an empty chart: i.e., a chart that only contains those rules of the grammar
whose left is the axiom.
============================================*/
processAgenda:-
```

```
        retrieveFromAgenda(Edge),
        enterEdge(Edge),
        processAgenda.
processAgenda:- true.
% aca probar en lugar tde true el predicado empty agenda para seguir la linea usando las
operaciones de agenda.

/*============================================
enterEdge(Edge)
Add the edge into the chart only if it does not yet exists.
============================================*/
enterEdge(Edge):-
        term_variables(Edge, SVars),
        Edge,
        term_variables(SVars, SVars1),
        SVars == SVars1, !, fail.

enterEdge(Edge):-
        addToChart(Edge),
        processEdge(Edge).

/*============================================
processEdge(Edge)
============================================*/
%We have a passive edge -> step completion applies.
processEdge(Edge):-
        %checks whether is passive
        Edge = edge(_,dottedRule(_,_,[]),_,_),
        applyFundamentalRulePassive(Edge).

%We have an active edge -> step completion and prediction applies.
processEdge(Edge):-
        %checks whether is active
        Edge = edge(_,dottedRule(_,_,[NextNonterminal|_]),Sem,Tree),
        applyFundamentalRuleActive(Edge,Applied),
        %Only predict if it was not possible to make any completion.
        (Applied == 0 ->predictNewEdges(NextNonterminal,Sem); true).

/*============================================
applyFundamentalRule(+arc)
Use the fundametal rule to combine this edge with edges from the chart.
Any newly edges obtained in this way should be added to the agenda.
============================================*/

%%% We have a passive edge; we are looking for an active one that precedes it.
applyFundamentalRulePassive(edge(IndexPas,dottedRule(LHSPassive,CompletedTree,[]),
SemanticsPas,TreePas)) :-
        edge(IndexActive,dottedRule(LHSActive,CompletedTreeActive,
        [rhs(_,ActiveSymbol)|UnseeingTree]),SemanticsAct,TreeAct),
        once(varmatch(ActiveSymbol,LHSPassive)),
        once(intersection(SemanticsPas,SemanticsAct,[])),
        %The following rule should be not commented for debbuging and seen the whole
        %edge information.
        %once(flatten([CompletedTreeActive|ActiveSymbol],NewCompletedTree)),
        NewCompletedTree=[],
        once(flatten([SemanticsPas|SemanticsAct],NewSemantics)),

        TreePas = dvTree(Tname,_),
        ( Tname == epsilon -> NewTree = TreeAct ;
        TreeAct = dvTree(TreeName,Descendants),
        append(Descendants,[TreePas],NewDescendants),
        NewTree = dvTree(TreeName,NewDescendants)
        ),

        once(storeInAgenda(edge(IndexActive,dottedRule(LHSActive,NewCompletedTree,UnseeingTree),
        NewSemantics,NewTree))),
        fail.
applyFundamentalRulePassive(_):-true.


%%% We have an active edge; we are looking for a passive one that follows it.
applyFundamentalRuleActive(edge(IndexAct,dottedRule(LHSActive,CompletedTree,
[rhs(_,ActiveSymbol)|UnseeingTree]),SemanticsAct,TreeAct),Applied) :-
        ActiveSymbol =.. [var,Aux,Cat,Bot,Top],
        %Skip lemanchor nodes! Do not use them for applaying the fundamental rule nor for
        predicting new edges
        \+var(Top),
        isLemanchor(Top),
        %The following rule should be not commented for debbuging and seen the whole edge
        %information.
        %once(flatten([CompletedTree|ActiveSymbol],NewCompletedTree)),
        NewCompletedTree=[],
        storeInAgenda(edge(IndexAct,dottedRule(LHSActive,NewCompletedTree,UnseeingTree),
        SemanticsAct,TreeAct)),
        Applied = 1.

/*Apply the fundamental rule when the edge is active*/
applyFundamentalRuleActive(edge(IndexAct,dottedRule(LHSActive,CompletedTree,
[rhs(N,ActiveSymbol)|UnseeingTree]),SemanticsAct,TreeAct),Applied) :-
        Counter = counter(0),

( edge(IndexPas,dottedRule(LHSPassive,CompletedTreePassive,[]),SemanticsPas,TreePas),
        once(varmatch(ActiveSymbol,LHSPassive)),
        once(intersection(SemanticsPas,SemanticsAct,[])),
        %The following rule should be not commented for debbuging and seen the whole
        %edge information.
        %once(flatten([CompletedTree|ActiveSymbol],NewCompletedTree)),
        NewCompletedTree=[],
        once(flatten([SemanticsPas|SemanticsAct],NewSemantics)),

        TreePas = dvTree(Tname,_),
        ( Tname == epsilon -> NewTree = TreeAct ;
        TreeAct = dvTree(TreeName,Descendants),
        append(Descendants,[TreePas],NewDescendants),
        NewTree = dvTree(TreeName,NewDescendants)
        ),

        once(storeInAgenda(edge(IndexAct,dottedRule(LHSActive,NewCompletedTree,UnseeingTree),
        NewSemantics,NewTree))),

        TreePas = dvTree(NamePas,_),

        ( NamePas == [epsilon] -> true ;
                arg(1, Counter, N0),
                Next is N0 + 1,
                nb_setarg(1, Counter, Next)
        ),
        fail
;
arg(1, Counter, Applied)
).

/*============================================
predictNewEdges(+NonTerminalSymbol)
This predicate implements the prediction step:
2.d. Make hypotheses (i.e., active edges) about new constituents based on the arc
and the rules of the grammar.
Add these new arcs to the agenda.
============================================*/
predictNewEdges(rhs(_,NonTerminalSymbol),Sem):-

        rtgRule(Number,Tree,Anchor,Var,RHS),

        %epsilon edges should not be introduced in any turn as the are already in the chart
        in its more general form.
        \+(Tree == epsilon),

        \+(\+ once(varmatch(NonTerminalSymbol,Var))),

        % checking for semantic coverage in the way of Carroll as a kind of indexing
        to reduce the number of predictions.
        Anchor =.. [anchor,_,Semantics],
        getDistinguishedIndex(Semantics,Index),
        DottedRule =.. [dottedRule,Var,[],RHS],

        %Predict only those edges whose semantic coverage does not overlap with the current
        active edge that need to be completed.
        once(intersection(Semantics,Sem,[])),
        concat(Tree,Index,IDTree),

        %Add the edge into the agenda.
        once(agenda:storeInAgenda(edge(Index,DottedRule,Semantics,dvTree(IDTree,[])) ) ),

        fail.

predictNewEdges(_,_):-true.

/*=============================================
addToChart
==============================================*/
addToChart(Edge):-
                assertz(Edge),
                util:file(chart,_,_,StreamChart),
                write(StreamChart,Edge),nl(StreamChart).

successEdge(Stream,InputSem):-
        Counter = counter(0),
        TreeSet = treeSet([]),
        (
        edge(Index,dottedRule(LHS,CompletedTree,[]),Semantics,Trees),
        once(varmatch(LHS,var(lc,s,_,_))),
        /*check whether Semantics is the input semantics*/
        once(permutation(InputSem,Semantics)),

        write(Stream,'% '),write(Stream,Trees),nl(Stream),nl(Stream),

        %Increment the number of trees
        arg(1, Counter, N0),
        Next is N0 + 1,
        nb_setarg(1, Counter, Next),

        %Add new trees to the set of successfully used trees.
        extractTreeNames(Trees,TreeNames),
        arg(1, TreeSet, S0),
        union(S0,TreeNames,Set),
        nb_setarg(1, TreeSet, Set),
        fail
;
arg(1, Counter, Number), write(Stream,'-- Number of derivations: '),
write(Stream,Number),nl(Stream),
arg(1, TreeSet, SuccessfulTrees), write(Stream,'-- Trees successfully used: '),
write(Stream,SuccessfulTrees),nl(Stream),
length(SuccessfulTrees,Len), write(Stream,'-- Nr. successfully used trees: '),
write(Stream,Len),nl(Stream)
).

incompleteEdge(Stream,InputSem):-
        edge(Index,dottedRule(LHS,CompletedTree,RHS),Semantics,Tree),
        ( \+(RHS == []) ->
                write(Stream,edge(Index,dottedRule(LHS,CompletedTree,RHS),Semantics,Tree)),
                nl(Stream),nl(Stream)
        ;
        %check if Semantics is not the input semantics
        (\+ permutation(InputSem,Semantics) ->
                write(Stream,edge(Index,dottedRule(LHS,CompletedTree,RHS),Semantics,Tree)),
                nl(Stream),nl(Stream) )
        ),
        fail.

incompleteEdge(_,_):-true.

/*=============================================
varmatch(+Variable,+Variable)
Verifies that the two variables match.
==============================================*/
/*gtsg transformed grammar */
% Copyright 2008 INRIA
% contributors: Sylvain Schmitz <Sylvain.Schmitz@loria.fr>
varmatch(Var,Var) :- !.
varmatch(var(init,Cat,_,T),var(lc,Cat,T,T)) :- !.
```

```
/*============================================
getDistinguishedIndex(+SemanticFormula,-DistinguishedIndex)
Obtains the index from the distinguished semantic indice from the semantic formula
============================================*/
getDistinguishedIndex([],_).
getDistinguishedIndex([_:literalSchema(Pred,[IDX])|RestSemantics],IDX):-!.
getDistinguishedIndex([Literal|RestSemantics],Index):-
                getDistinguishedIndex(RestSemantics,Index).

/*============================================
getActiveDistinguishedIndex(+ActiveSymbol,-DistinguishedIndex)
Obtains the index feature from the active category (the symbol after the dot)
from an active edge.
============================================*/
getActiveDistinguishedIndex(var(aux,Cat,Bot,Top),Idx):-
                getActiveDistinguishedIndexFeature(Bot,Idx),!.

getActiveDistinguishedIndex(var(Type,Cat,Bot,Top),Idx):-
                getActiveDistinguishedIndexFeature(Top,Idx),!.

/*gtsg transformed grammar*/
getActiveDistinguishedIndexFeature(FS,Idx):-
                idxPos(IdxPos),
                featuresCount(Nro),
                functor(FS,fs,Nro),!, arg(IdxPos,FS,Idx).

/*============================================
isLemanchor - checks whether the node is subst but lemmanchor type
============================================*/
/*gtsg transformed grammar*/
isLemanchor(FS):-
        lemanchorPos(LemanchorPos),
        featuresCount(Nro),
        functor(FS,fs,Nro),!, arg(LemanchorPos,FS,Value),term_variables(Value,SVars),SVars==[].


/*gtsg transformed grammar obtain features information*/
clearFeaturesInfo:-
        retractall(idxPos(_)),
        retractall(lemanchorPos(_)),
        retractall(featuresCount(_)).

setFeaturesInfo:-
        util:getFeatures(CompleteList),
        eliminateLast(CompleteList,List),
        write('Features considered: '),write(List),nl,
        length(List,L),
        write('number of features: '),write(L),nl,
        assert(featuresCount(L)),
        (nth1(IndexPos,List, 'idx') -> write('idx pos: '),write(IndexPos),nl
        ;
        IndexPos=0),
        assert(idxPos(IndexPos)),
        nth1(LemanchorPos,List, 'lemanchor'),write('lemanchor pos: '),write(LemanchorPos),nl,
        assert(lemanchorPos(LemanchorPos)).

%Required because of the way the translation is done for feature structures.
Adding to the list of features one more element that is the variable unknown
of prolog '_'. We eliminate it.
eliminateLast([_],[]).
eliminateLast([Elem|Rest],[Elem|NewList]):-
        eliminateLast(Rest,NewList).

%From the derivation tree extructure, extract the list of names.
extractTreeNames(dvTree(Name,[]),[Name]).
extractTreeNames(dvTree(Name,Trees),[Name|NewNames]):-
        extractTreeNamesDaughter(Trees,NewNames).

extractTreeNamesDaughter([],[]).
extractTreeNamesDaughter([FisrtTree|Sisters],NewListNames):-
        extractTreeNames(FisrtTree,FisrtNames),
        extractTreeNamesDaughter(Sisters,ListNames),
        append(FisrtNames,ListNames,NewListNames).
```

RTG rules selection (rtgSelection.pl)

```
/*
Exported predicates:
*/
:- module(rtgSelection,[selectRTGRules/1]).

:- use_module('Tools/util',[literalSchemaFormat/2,getFeatures/1,memo/1]).

:- use_module('Tools/unify',[unify/3]).

:- use_module('Tools/domain',[domain/2]).

selectRTGRules(Type):-!,
        style_check(-singleton),
        getGrammarTranslation(Type,Context),
        write('Filtering level: '),write(Type),nl,
        selectRTGRulel(Context),
        Context:features(List),
        retractall(getFeatures(_)),
        memo(getFeatures(List)).

selectRTGRulel(Context):-
util:file(selectedTAGtreesAsRTG,_,_,Stream),
write(Stream,'/*===================Substitution trees=====================*/'),nl(Stream),
Counter = counter(0),
(
        util:agenda(treeSchema(_,TreeName,Interface,_,_,Sem,_)),
        Context:rule(Number,TreeName,anchor(FamilyName,FS,SemanticSchema),LHS,RHS),
        once(util:literalSchemaFormat(SemanticSchema,SemanticSchemaFormated)),
        once(replacePredicates(Interface,SemanticSchemaFormated,SemanticSchemaWithPred)),
        once(unify_with_occurs_check(SemanticSchemaWithPred,Sem)), %swi-prolog

        Anchor =.. [anchor,Family,SemanticSchemaWithPred],

        format(Stream,'rtgRule(~w,', [Number]),
        format(Stream,'~q,', [TreeName]),
        write(Stream,'Anchor,LHS,RHS):-'),
        nl(Stream),
        format(Stream,'Anchor= ~w,',[Anchor]),
        nl(Stream),
        format(Stream,'LHS= ~w,', LHS),
        nl(Stream),
        format(Stream,'RHS= ~w. ', [RHS]),

        nl(Stream),nl(Stream),

        arg(1, Counter, N0),
        N is N0 + 1,
        nb_setarg(1, Counter, N),

        fail
;
        arg(1, Counter, Times),
        write('Initial trees traduced into RTG rules: '),write(Times),nl,fail
).

selectRTGRulel(Context):-
util:file(selectedTAGtreesAsRTG,_,_,Stream),
write(Stream,'/*===================Auxiliary trees=====================*/'),nl(Stream),
Counter = counter(0),
(
        util:auxiliaryAgenda(treeSchema(_,TreeName,Interface,_,_,Sem,_)),
        Context:rule(Number,TreeName,anchor(FamilyName,FS,SemanticSchema),LHS,RHS),
        once(util:literalSchemaFormat(SemanticSchema,SemanticSchemaFormated)),
        once(replacePredicates(Interface,SemanticSchemaFormated,SemanticSchemaWithPred)),
        once(unify_with_occurs_check(SemanticSchemaWithPred,Sem)), %swi-prolog

        Anchor =.. [anchor,Family,SemanticSchemaWithPred],

        format(Stream,'rtgRule(~w,', [Number]),
        format(Stream,'~q,', [TreeName]),
        write(Stream,'Anchor,LHS,RHS):-'),
        nl(Stream),
        format(Stream,'Anchor= ~w,',[Anchor]),
        nl(Stream),
        format(Stream,'LHS= ~w,', LHS),
        nl(Stream),
        format(Stream,'RHS= ~w. ', [RHS]),
        nl(Stream),nl(Stream),

        arg(1, Counter, N0),
        N is N0 + 1,
        nb_setarg(1, Counter, N),

        fail
;
        arg(1, Counter, Times),
        write('Auxiliary trees traduced into RTG rules: '),write(Times),nl,fail
).

selectRTGRulel(_):-
        util:file(selectedTAGtreesAsRTG,_,_,Stream),
        write(Stream,'rtgRule(0, epsilon, anchor(epsilon,[]), var(aux, _, FS, FS), []).'),
        nl(Stream),
        true.

%replacePredicates(+Interface,+SemanticSchema,-SemanticSchemaWithPred).
replacePredicates(_,[],[]).
replacePredicates(Interface,[L:literalSchema(J,G)|SemanticSchema],
        [L:literalSchema(PredicateName,G)|SemanticSchemaWithPred]):-
        obtainRelatedName(Interface,J,PredicateName),
        replacePredicates(Interface,SemanticSchema,SemanticSchemaWithPred).

% obtainRelatedName(+Interface,+J,-PredicateName)
obtainRelatedName([],_,_).
obtainRelatedName([J:Value|Interface],J,Value).
obtainRelatedName([Feature:Value|Interface],J,PredicateName):-
        obtainRelatedName(Interface,J,PredicateName).

%different translations of the grammar:
%Filtering Level1
getGrammarTranslation(1,File):-
        File = rtgGrammar0,
        load_files(File:['LinguisticResources/Grammar/rtgGrammar1.pl'],[rule/5,features/1]).

%Filtering Level2
getGrammarTranslation(2,File):-
        File = rtgGrammar2,
        load_files(File:['LinguisticResources/Grammar/rtgGrammar2.pl'],[rule/5,features/1]).

%Filtering Level3
getGrammarTranslation(3,File):-
        File = rtgGrammarVAlt,
        load_files(File:['LinguisticResources/Grammar/rtgGrammar-vAlt'],[rule/5,features/1]).
```