Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Optimising a Surface Realiser

Mémoire de DEA

présenté et soutenu publiquement le 22 juin 2004

pour l'obtention du

DEA Informatique de Lorraine

Université Henri Poincaré – Nancy 1

par

Eric Kow

Composition du jury

Rapporteurs :	N. Carbonell
	D. Méry
	D. Galmiche
	J.Y. Marion
	J. Souquières
Encadrante:	C. Gardent

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Table of contents

Acknowledgments

I have had an immense amount of help on this DEA.

For starters, my supervisor Claire Gardent has gone out of her way for this little first step of a thesis. Working with Claire has been a fun and instructive experience, as she has this uncanny ability to always ask the right questions. This results in an endless supply of good ideas that are invariably of the simple and practical variety.

Carlos Areces has also contributed his fair share of elegant ideas. I would like to thank him as well for help with Haskell, the surface realiser GenI, and a million essential concepts. Carlos speaks with such pedagogical clarity that it almost seems like a bad habit; I wonder if he would even be capable of explaining anything poorly if he wanted to.

Many many thanks to Patrick Blackburn and Hélène Manuelian and especially Claire and Carlos for reading the many awkward drafts of this thesis and red-penning its way into passability. I hope to someday master the basic respect and consideration they have for their reader.

The good strangers who frequent the channel #haskell on irc.freenode.net deserve at least a grateful nod for their kindness to newbies.

Finally, the following people have helped in a more lateral fashion: Hélène with her fine Franglais-French translation of my DEA letter of motivation; office mates Yannick and Joseph for puttting up with much whining, and the occasional, inexpliquable “moo!”. Thank-you all for your friendship. And now I restrain myself from tearful speeches of friends and family, and hope to do these people justice at a later time.

Chapter 1

Introduction

The surface realisation task consists in using a grammar to produce the sentence(s) associated by this grammar with some input meaning representation. The meaning representation is written using some logical language. For example, a surface realiser might receive an **input semantics** of {`loves(l,john,mary)`}, for which it outputs the **surface realisation** *John loves Mary.*

Surface realisation, also referred to as **tactical generation**, is similar to the better known task of natural language parsing. When doing generation, we construct syntactic trees which are associated with the input semantics by the language's grammar, and trivially read the tree leaves to reconstruct the output string. Polynomial-time algorithms for parsing have long been developed, so it would be tempting to assert that generation is parsing in reverse and simply reuse these algorithms. But [KS02] demonstrates that the Hamiltonian Path problem can be reduced to a surface realisation task and so the problem is NP-complete. The intuitions for this can be found in [Kay96]. Both parsing and generation have the potential to be computationally expensive because natural language has a great deal of lexical ambiguity; however, parsers have the luxury of string positions. Using string positions, one can pack an arbitrary number of alternative structures into a small and fixed number of edges, so that parsing can be done in cubic time no matter what the ambiguity. The situation for surface realisers is more delicate because the input semantics have no natural order, and so there are as many potential edges as there are subsets of literals. This number is exponential with respect to input length, and as a result, we are forced to deal with the full effect of lexical ambiguity.

This thesis explores the problem of optimisation for a TAG-based surface realiser. We begin in this chapter with the essentials of surface realisation and continue in section 1.2 with a description of the realiser developed within the INRIA ARC GenI by Carlos Areces and used here as an implementation platform [Are03]. In chapter 2, we propose a set of optimisations, implement them into the GenI realiser, and discuss their effectiveness. We conclude in chapter 3 with proposals for further optimisation and pointers for further research.

1.1 Background

Building a surface realiser requires three ingredients: a semantic representation, a formal grammar for natural language syntax augmented with a semantic dimension, and an algorithm for building syntactic structures. The GenI generator uses respectively a flat semantics, the Tree Adjoining Grammar (TAG) formalism, and a bottom-up/head-driven algorithm.

1.1.1 Flat semantics

The semantics of a natural language expression can be approximated by a logical form. In this thesis, we write the semantics as a conjunction of **literals**, where each literal is a predicate with some arguments. The arguments may not themselves be literals, so there is no recursion, and we call this a **flat semantic representation** [CFM+95]. For example, to represent the meaning of the sentence *John loves Mary intensely* we could write {`loves(l, j, m), intense(l), name(j, john), name(m, mary)`}, meaning that there is an act of loving `l` between the individuals named *John* and `Mary` and that this act is intense. For a more detailed introduction to flat semantics, we refer the reader to [CFM+95].

The various approaches to surface realisation work by successively combining expressions which verbalise different parts of the input semantics. This aspect is particularly well served by a flat representation. In particular, note that in order to compare recursive semantic representations, something like higher order unification is required, whereas by using a flat semantics, this comparison can be carried out using set intersection. Similarly, we can combine two semantic expressions by taking their union.

1.1.2 Tree-Adjoining Grammar

Tree-Adjoining Grammar (TAG) is a formalism that generates the "mildly context-sensitive" class of languages. As defined in [JS97], a tree-adjoining grammar consists of a quintuple $\langle \Sigma, NT, I, A, S \rangle$ 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 labeled by non-terminal symbols;
   - frontier nodes are labeled 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 labeled by non-terminal symbols;
   - frontier nodes are labeled 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 label of the foot node must be identical to the label of 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. These trees can be composed pairwise to build more complex structures, called **derived trees**, through the operations of substitution and adjunction.

The **substitution** operation (figure 1.1) 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 **closed**.

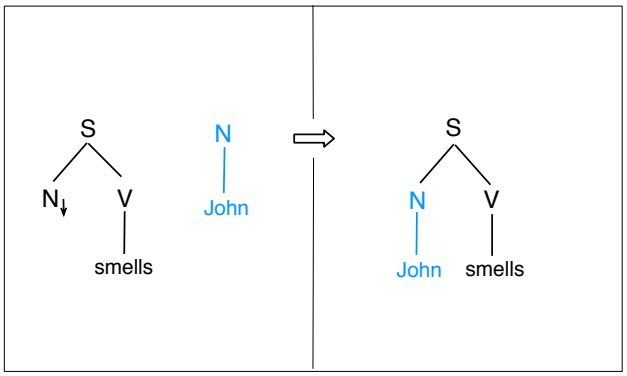

Figure 1.1: Substitution

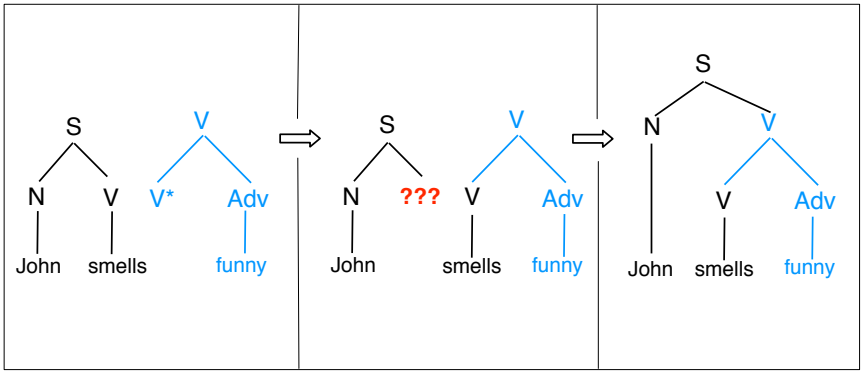

Figure 1.2: Adjunction

The **adjunction** operation 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 labeled by X, and β be an auxiliary tree whose root node is also labeled 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**.

1.1.2.1 TAG with flat semantics

We augment the TAG formalism by associating each tree in the grammar with a semantics using the flat representation. Whenever two trees are combined (via substitution or adjunction), the semantics of the resulting tree is calculated by taking the set union of the source tree semantics. In figure 1.3, we show the result of the substitution and adjunction from figures 1.1 and 1.2 once semantics is taken into account.

Note that it is not realistically feasible to implement the syntax-semantics interface without introducing a notion of variable unification. Without such a mechanism, we would have to create a separate entry for every possible combination of tree semantic indices, one for `smells(s,j)`, for `smells(s,k)`, for `smells(s,l)` and so forth. Realistic grammars have tree semantic literals

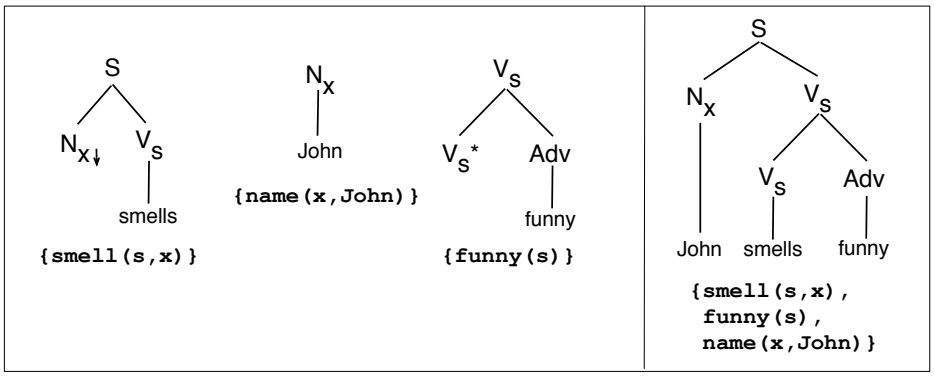

Figure 1.3: TAG with flat semantics

where the arguments are not necessarily constants, but variables, which simultaneously occur in the nodes of the tree. Before they are used for generation, trees have to be **instantiated**, meaning that unification must be performed between these variables and the arguments contained in the input semantics representation. Variable unification on node variables also occurs during the substitution and adjunction operations, which prevents incorrect realisations like *John hates Mary* when the intended output was *Mary hates John*.

Consider a grammar which has a tree *smells* with the semantics `smells(V,X)` (we indicate variables with capital letters). Given the input semantics {`smells(s,j), name(j,john), funny(s)`}, we must first instantiate *smells* by unifying `V` with `s` and `X` with `j`. If the generator were to use a different input semantics, like {`smells(s,m), name(m,mary), nice(s)`}, then *smells* would be instantiated differently.

1.1.3 Generation algorithms

We can see generation as the process of constructing a syntactic tree [SvNPM90]. There are two basic approaches to this process, building the tree from the top-down, or from the bottom-up. There is also a hybrid head-driven approach which combines the two basic strategies.

These approaches were originally developed in the framework of context free grammars with recursive semantics. For consistency with this thesis, we will maintain a TAG perspective: context free rules are viewed as shallow TAG trees, and non-terminal child nodes are substitution sites. The approaches also assume a recursive semantics, but rather than risking errors by converting them to a flat semantics, we will momentarily adopt a recursive semantics. The relationship between syntax and semantics is somewhat different here. Instead of using the union operation to combine tree semantics, we allow the use of complex variables and let unification do the work. Grammars in this section are presented as Prolog DCGs with the lexicon omitted.

1.1.3.1 Top-down generation

The top-down approach consists in working from the root to the leaves. We select any tree from the grammar with root category `s` (for sentences) and perform unification between its and the input semantics. Then we loop around; if the tree has any substitution sites, we select a tree with the same root as the site, perform substitution (and unification). Hopefully doing this

closes off all the substitution sites, but it may very well not, and worse, it may introduce its own share of sites. The loop continues until the tree is closed.

Non-termination This approach can fail to terminate if it is applied on the wrong grammar, namely, a left-recursive one. For instance, the following grammar should be able to produce *John smells funny* as a realisation for the input semantics `funny(smell(john))`. [1]

```
s(S)                --> np(NP), vp(NP,S).      % t1
vp(NP,S)            --> vp(NP,VP), adv(VP,S). % t2
vp(NP,S)            --> v(NP,S).               % t3
v(NP,smell(NP))     --> smells.                % t4
adv(VP,funny(VP))   --> funny.                 % t5
adv(VP,nice(VP))    --> nice.                  % t6
np(john)            --> john.                  % t7
```

But top-down generation on this grammar might not terminate:

1. We begin with t1. Ignoring semantic information, execution of t1 yields a tree `s(np↓, vp↓)` where, as noted above, the rule is represented by a local tree and the non-terminals by substitution sites labeled with the rule's right hand side syntactic categories.
2. We substitute t7 to produce `s(np(john), vp↓)`
3. The derived tree still has a substitution site to fill, this being `vp↓`, so we process t2.
4. t2 introduces its own share of substitution sites. If we attempt to fill the first one (`vp↓`) by substituting t2 into it, this brings us back to the same problem and traps us in an infinite loop.

1.1.3.2 Bottom-up generation

Bottom-up generation builds trees from the lexical items to the root. First, the **lexical selection step** selects from the grammar the trees whose semantics subsumes part of the input semantics, and places them in a pool. Then the generator loops: it arbitrarily selects a pair of trees from the pool and attempts substitution between the two trees. If the resulting derived tree is closed and if its semantics matches the input semantics, then it is added to the output. Otherwise, it is put into the pool for future consideration. The loop continues in this fashion until there are no new pairs to select.

Given the grammar from the previous section (1.1.3.1), a bottom-up generator would find the surface realisation *John smells funny* from the input semantics `funny(smell(john))` as follows:

1. Every tree in the grammar except t5 is selected, resulting in a pool that consists of t1 `s(np↓, vp↓)`, t2 `vp(vp↓, adv↓)`, t3 `vp(v↓)`, t4 `v(smells)`, t5 `adv(funny)` and t7 `np(john)`.
2. t4 is substituted into t3 and the resulting `vp(v(smells))` is added to the pool.

[1] In this grammar, categories encode both syntactic and semantic information. Specifically, a term of the form `Synt(Sem)` denotes a category with syntactic category `Synt` and semantic information `Sem`. Shared variables are used to adequately bind semantic arguments.

3. This is substituted into t2, and the resulting `vp(vp(v(smells)), adv↓)` is added to the pool.

4. t5 is then substituted into this, and the resulting `vp(vp(v(smells)), adv(funny))` is added to the pool.

5. This is substituted into t1 to produce `s(np↓, vp(vp(v(smells)), adv(funny)))`

6. t7 is substituted into this and the resulting `s(np(John), vp(vp(v(smells)), adv(funny)))` is added to the output.

7. This tree is syntactically complete and matches the input semantics. If we read the leaves of this tree, we extract the output sentence *John smells funny*

Termination and non-determinism A useful constraint for bottom-up algorithms is to only allow trees to be combined if they do not have any lexically selected trees in common. This prevents infinitely long strings like *smells funny funny funny...* from being generated.

Once this filter is in place, bottom-up algorithms are guaranteed to terminate, because the lexical selection step gives them a finite number of trees to start with. We can think of the algorithm loop as always consuming a lexically selected tree until it eventually runs out and has to stop.

This guarantee of termination comes at the price of highly non-deterministic behaviour. The generation loop selects at every iteration, a pairs of trees to be combined. If these choices do not lead to a result, then the generator must make a new set of selections, either by backtracking, or by having simultaneously considering the different choices that it can make.

Semantic monotonicity One potential disadvantage of this approach is that it requires grammars to be semantically monotonic. A **semantically monotonic** grammar only has rules in which the semantics of the daughter nodes is contained in the semantics of the mother node. For example, any grammar which includes the following fragment is not semantically monotonic:

```
vp(X,Y,callup(X,Y)) --> v(X,Y,callup(X,Y)), np(Y), pp(up).
v(X,Y,callup(X,Y))  --> call.
pp(up)              --> up.
```

The intention behind these rules is to generate sentences like *John calls Mary up* from the input semantics `callup(john,mary)`. A top-down generator would simply instantiate the rule, ignoring the semantics for *up*. But a bottom-up generator could never use this rule, because `up` is not in the input semantics, so that there is no reason to select the `pp(up)` rule.

Intersective modifiers Bottom-up generators are also affected with the non-determinism triggered by **intersective modifiers**. Consider the semantics `dirty(smelly(unpleasant(man(x))))`, whose predicates all modify the individual `x`. This should be realised with the following grammar as the fragment *dirty smelly unpleasant man*.

```
n(N,dirty(N))       --> dirty, n(N).
n(N,smelly(N))      --> smelly, n(N).
n(N,unpleasant(N))  --> unpleasant, n(N).
n(man)              --> man.
```

To get the desired result, however, the generator will try out each and every one of the following combinations:

dirty smelly unpleasant man,
dirty smelly man, *dirty unpleasant man*, *smelly unpleasant man*,
dirty man, *smelly man*, *unpleasant man*,
man

That is, it creates $n!$ possible sequences (where n is the number of modifiers) just to find a single correct answer. We could even be generous and impose a sort of word order constraint, so that *smelly dirty man* would not be allowed, whereas *dirty smelly man* would. In this case, the number of possible substrings drops to 2^n.

1.1.3.3 Head-driven generation

Head-driven generators are based on the observation that every syntactic tree necessarily has a **semantic head**, the lowest possible node whose semantics is that of the tree. If we revisit the example in section 1.1.3.1 (repeated below), we can identify the verb *smells* as the semantic head for the verb phrase *smells funny* and consequently for the sentence *John smells funny*. The semantic head can be used to guide the generation process. In [SvNPM90], the grammar is preprocessed so that trees are divided among **chain rules** whose semantic head is a child node, and **non-chain rules** whose semantic head is its root node. In the example, only t1, t2 and t3 are chain rules.

```
s(S)                --> np(NP), vp(NP,S).      % t1
vp(NP,S)            --> vp(NP,VP), adv(VP,S). % t2
vp(NP,S)            --> v(NP,S).               % t3
v(NP,smell(NP))     --> smells.                % t4
adv(VP,funny(VP))   --> funny.                 % t5
adv(VP,nice(VP))    --> nice.                  % t6
np(john)            --> john.                  % t7
```

The generator works with what is called a **target node**. The first target node has category `s` (for sentence) and the semantics equal to the input semantics. The basic approach is to use top-down processing of the non-chain rules to identify the semantic head of the target and use bottom-up processing of the chain rules to connect the two nodes. Any new substitution sites encountered along the way are recursively processed as targets. We can think of the generator as cycling between three behaviours: leaping (top-down), connecting (bottom-up) and recursion (top-down). When tracing the algorithm, it is useful to imagine a stack of target nodes. The algorithm does not actually use such a stack because it is implicit in the recursive behaviour.

Given the example grammar above and the input semantics `funny(smell(john))`:

1. We push the desired result `s(funny(smell(john)))` on the stack.
2. (td) We leap down to t5 `adv(funny)` because its semantics unifies with `funny(smell(john))`. There are no child nodes, so we try to connect upwards.
3. (bu) t5 can serve as the semantic head of chain-rule t2, so we perform substitution to produce `vp(vp↓, adv(funny))`. We push the child node `vp(NP,smell(john))` on the stack.

   a (td) We leap down to t4 `v(smells)` because its semantics unifies with `smell(john)`. There are no child nodes, so we try to connect upwards.

   b (bu) t4 can serve as the semantic head of chain-rule t3, so we perform substitution to produce `vp(v(smells))`. There are no child nodes to fill, so we try to connect upwards.

   c (bu) The root node of the above tree unifies with the target `vp(NP,smell(john))`, so we perform substitution to produce `vp(vp(v(smells)), adv(funny))`, and bottom-up processing is complete. We pop the stack and try to connect upwards to the new target.

4. (bu) `vp(vp(v(smells)), adv(funny))` can serve as the semantic head of chain-rule t1, so we perform substitution to produce `s(np↓, vp(vp(v(smells)), adv(funny)))`. We push the child node `np(john)` on the stack.

   a (td) we leap down to t7 because its semantics unifies with `john`. There are no child nodes, so we now try to connect upwards.

   b (bu) The t7 root unifies with the target, so we perform substitution to produce `s(np(John), vp(vp(v(smells)), adv(funny)))`. We pop the stack and try to connect upwards.

5. (bu) The t1 root unifies with the target, so bottom-up processing is complete. The stack is empty and we have a semantically complete result, from which we extract the sentence *John smells funny*.

The head-driven approach combines the most attractive properties of bottom-up and top-down processing, namely termination and efficiency. Like bottom-up algorithms, the head driven algorithms lexically select and then consume a finite set of trees on the basis of their semantics. The main difference is that the lexical selection (leaping) is interspersed with the generation loop as opposed to being set in the very beginning, but the essential property of using the semantic information is retained. The efficiency of these approaches comes from the top-down ability to guide the selection items by syntactic criteria. This is most evident with intersective modifiers; given the semantics `dirty(smelly(unpleasant(man(x))))` and the grammar fragment in section 1.1.3.2, the head-driven algorithm would zip down the rules, *dirty*, *smelly*, *unpleasant* in a similar fashion to a top-down algorithm, except that this zipping would consist of trivial leaping and connecting steps.

1.2 The GenI surface realiser

Within the INRIA ARC GenI, Carlos Areces developed a TAG-based surface realiser in Haskell. This is simultaneously a bottom-up and head-driven generator, bottom-up in its basic tree-building strategy and head-driven by virtue of TAG and the use of entire trees as basic linguistic units.

The TAG formalism enjoys what [GT01] call an "extended domain of locality" in that elementary trees can be of arbitrary depth. This makes the requirement of semantic monotonicity much more palatable. In the case of *John calls Mary up*, the word *up* would simply be incorporated into an elementary tree `s(np↓, v(calls), np↓, pp(up))`. Furthermore the trees with their extra depth and complexity contain syntactic information that would normally be built in

initial trees		auxiliary trees	
`np(John)`	`{name(X,john)}`	`v(v*,adv(funny))`	`{funny(X)}`
`s(np↓, v(smells))`	`{smell(S,X)}`		

Figure 1.4: TAG grammar for *John smells funny*

the top-down phase of head-driven generation. Using TAG thus allows us to imbue a bottom-up generator with head-driven characteristics.

The GenI generator also includes a number of optimisations, including chart generation, ordered substitution and delayed adjunction, which we now discuss.

1.2.1 Chart generation

Some algorithms for natural language parsing use a tabulation method to improve efficiency. These **chart algorithms** can also be adapted or generalised to do generation [Shi88, Kay96]. We examine chart methods from the perspective of our thesis, namely, with bottom-up processing and with trees as basic linguistic units.

1.2.1.1 Basic architecture

The essence of bottom-up processing is a loop that compares two trees and when possible, combines them to make a new tree, until it runs out of trees to compare. Chart algorithms will modify this process by avoiding recomputation of redundant intermediate structures.

We introduce two data structures called an **agenda** and a **chart**. The agenda is a form of to-do list and it contains the trees that need to be processed. The chart contains the trees that have already been processed. In the beginning of the processing, the agenda is initialised with those trees whose semantics subsumes the input semantics. During the course of the processing, an item is removed from the agenda, placed on the chart and combined with items from the chart. Any tree that results from the combination is placed on the agenda, unless it is syntactically complete (closed) and its semantics exactly matches the input semantics, in which case it is added to the results. We loop in this fashion until the agenda is completely empty, which means that all comparisons have been made.

1.2.1.2 An example

Given the grammar in figure 1.4 and the input semantics `{smell(s,x), funny(s), name(x,john)}`:

1. We initialise the agenda with the trees τ_{smells} , τ_{funny} , and τ_{John} , as in figure 1.3 (items on agenda: 3, chart: 0).

2. Item τ_{smells} is removed from the agenda. There are no items to compare it with, so it is simply placed on the chart (items on agenda: 2, chart: 1).

3. Item τ_{funny} is removed from the agenda. It is compared with τ_{smells} which results (by adjunction) in the tree $\tau_{smells\ funny}$. Item τ_{funny} is placed on the chart, and $\tau_{smells\ funny}$ is placed on the agenda. (items on agenda: 2, chart: 2).

4. Item $\tau_{smells\ funny}$ is removed from the agenda. It is unsuccessfully compared with τ_{smells} and τ_{funny} and is placed on the chart. (items on agenda: 1, chart: 3).

5. Item τ_{John} is removed from the agenda. It is unsuccessfully compared with τ_{funny} and successfully compared with τ_{smell} and with $\tau_{smells\ funny}$. τ_{John} is placed on the chart. The result $\tau_{John\ smells}$ is placed on the agenda; on the other hand the other result $\tau_{John\ smells\ funny}$ is both syntactically and semantically complete, so it is added to the output. (items on agenda: 1, chart: 4).

6. Item $\tau_{John\ smells}$ is removed from the agenda. It is unsuccessfully compared with τ_{John}, τ_{smells}, and $\tau_{smells\ funny}$; and successfully compared with τ_{funny}. The resulting tree $\tau_{John\ smells\ funny}$ is syntactically and semantically complete, so it is added to the output. Item $\tau_{John\ smells}$ is placed on the chart. (items on agenda: 0, chart: 5).

7. Since the agenda is empty, the processing ends.

We can observe that there are two distinct derivations for the derived tree $\tau_{John\ smells\ funny}$, one which begins by creating $\tau_{smells\ funny}$ and the other by creating $\tau_{John\ smells}$. This is not an artifact of chart processing but of using a naive bottom-up strategy. The GenI surface realiser employs a number of mitigating tactics which we will discuss in sections 1.2.2 and 1.2.3.

1.2.1.3 Intermediate storage

The chart data structure acts as a form of storage for intermediate data structures and thus eliminates a great number of redundant recalculations. For example, the trees for *unpleasant smelly man* and *dirty smelly man* share the subtree for *smelly man*. Using a chart algorithm means that the latter subtree is only computed once and reused as many times as needed.

1.2.1.4 Chart indexing

The algorithm as we have presented it does not yet exploit the full potential of tabulation. Doing so would depend on the availability of a good **indexing scheme** which would allow us to reduce the search space.

Indexing schemes are more evident for natural language parsing than for generation; we assign every tree an index for its initial and final string position. So, to parse *John saw Mary with Peter* the agenda would be initialised with (0,1,τ_{John}), (1,2,τ_{saw}), (2,3,τ_{mary}), (3,4,τ_{with}), (4,5,τ_{Peter}). This prevents some clearly unnecessary comparisons from taking place. The tree for (1,2,τ_{saw}) is compared with that for (2,3,τ_{Mary}) because they articulate around the common index 2. But it is not compared with (4,5,τ_{Peter}) because their indices have nothing to do with each other. The use of indices partitions the search space between parts that are clearly avoidable and those that need to be explored.

Numerical indices serve as the "natural points of articulation" for a parsing algorithm [Kay96], but they cannot be used for generation because linear order of literals is not natural to semantic expression. The semantics {`name(j,john), smells(s,j)`} is equivalent to {`smells(s,j), name(j,john)`}, whereas the English phrases "John smells" and "smells John" are not. Put another way, the surface realisation task is equivalent to that of parsing a completely free word order language. GenI does not implement an indexing scheme, but we will propose one in section 3.1.3.

1.2.2 Ordered substitution

We observed in section 1.2.1.2 that it is possible to produce the same derived tree in different ways. In the GenI generator, some of these redundant derivations are prevented by using or-

initial trees	
s(np↓, v(hates), np↓)	{hate(X,Y,Z)}
np(mary)	{name(X,mary)}
np(john)	{name(X,john)}

Figure 1.5: TAG grammar for *Mary hates John*

.

initial trees		auxiliary trees	
np(man)	{man(X)}	np(det(the),np*)	{def(X)}
s(np↓, v(rejects), np↓)	{reject(X,Y,Z)}	np(adj(beautiful),np*)	{beautiful(X)}
np(woman)	{woman(X)}	np(adj(heartless),np*)	{heartless(X)}
		np(adj(smelly),np*)	{smelly(X)}
		np(adj(dirty),np*)	{dirty(X)}

Figure 1.6: TAG grammar for *the beautiful heartless woman rejects the dirty smelly man*

dered substitution. Using the grammar in figure 1.5, the derived tree s(np(Mary), v(hates), np(John)) can be built either by first substituting np(Mary) into the tree s(np↓, v(hates), np↓) to get s(np(Mary), v(hates), np↓) and then substituting np(John) into that tree. Alternatively, it can be built by first substituting np(John) to get s(np↓,v(hates), np(John)), into which np(Mary) is substituted. If we perform substitution in a fixed order, for example left-to-right, then the second derivation is eliminated. Since the trees np(Mary) and np(John) are inserted into two different substitution sites, we know that they do not interact, and that rejecting the redundant derivation will not prevent any legitimate results from being produced.

Ordered substitution has the secondary benefit of preventing certain intermediary structures from being built. Specifically, no derived tree can be built in which a substitution site is filled whose predecessor (in the chosen substitution order) is not. For instance assuming a left to right substitution order, the tree s(np↓,v(hates), np(John)) in the above example will not be produced.

1.2.3 Delayed adjunction

TAG adjunction provides us with means for coping with the problem of intersective modifiers (section 1.1.3.2). Following [CCFP99], we break generation into two phases. In the first phase, we perform generation but using only TAG substitution. Then we discard any trees which have substitution nodes that remain to be filled, because we know that these could never lead to a result. Then we switch to the second phase, which consists of generation using only adjunction.

We can examine the procedure in more detail in an example. Consider the target semantics {def(m), dirty(m), smelly(m), man(m), def(w), woman(w), beautiful(w), heartless(w), reject(r,w,m)}. Using the grammar in figure 1.6, this should be realised as the sentence *the beautiful heartless woman rejects the dirty smelly man*.

1. In the first phase, TAG substitution is performed between $\tau_{rejects}$ and τ_{man} resulting in $\tau_{rejects\ man}$. This derived tree is then substituted with τ_{woman} to get the result $\tau_{woman\ rejects\ man}$.

2. Now trees that still have substitution sites, i.e. $\tau_{rejects}$ are discarded. This leaves us

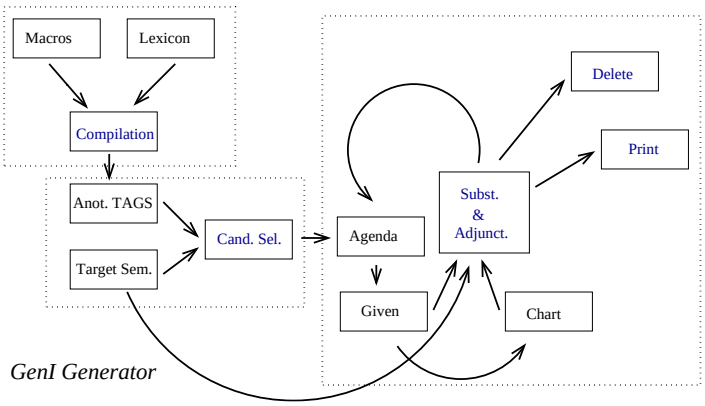

Figure 1.7: The GenI algorithm

with the trees τ_{man}, τ_{woman}, $\tau_{woman\ rejects\ man}$; and a chart containing τ_{the}, τ_{dirty}, τ_{smelly}, $\tau_{beautiful}$ and $\tau_{heartless}$.

3. In the second phase, TAG adjunction is performed between the auxiliary and the non-auxiliary trees. This eventually produces a derived tree that covers the input semantics, the leaf nodes of which can be read to produce the string *the beautiful heartless woman rejects the dirty smelly man*.

Delayed adjunction does not solve the problem of intersective modifiers but largely contains the number of intermediate structures being built. The above example still produces an exponential number of redundant substrings (such as *the beautiful woman rejects the man*). However by getting substitution over with in the beginning, we prevent a large number of useless interactions such as trying to substitute $\tau_{dirty\ man}$ into $\tau_{woman\ rejects}$, or into $\tau_{the\ woman\ rejects}$, or $\tau_{beautiful\ woman\ rejects}$, and so forth.

1.2.4 The GenI algorithm

We present the basic GenI generator in figure 1.7, as well as algorithms 1 and 2. This includes a lexical selection step (section 1.1.3.2), delayed adjunction via separate substitution and adjunction phases, a chart-processing algorithm (section 1.2.1) with a trivial implementation of the chart data structure. We assume that the functions SUBSTITUTION and ADJUNCTION have already been defined and that they return a set of trees (an empty set if the operation fails). Ordered substitution is assumed to be implemented in the SUBSTITUTION function. For simplicity's sake, we completely ignore variable unification.

Throughout this thesis, we assume a language and/or libraries with the following features:

- complex assignments: $\langle foo, bar \rangle \leftarrow baz$ which assumes that *baz* is a tuple with compatible type, and foo gets assigned to the first value of the tuple, and *bar* to the second.
- lists: ":" indicates list concatenation, and "[]" indicates the empty list.
- complex types with named components (indicated by the operator "."): for example, *tree.semantics* should return the component *semantics* of the complex object *tree*

- associative arrays: If arr is an associative array, then $arr[k]$ should retrieve the item associated with key k or return the value $NULL$ if there is none. $arr.keys$ should always return the set of keys used in arr. $\emptyset$ should be overloaded to also mean the empty associative array, and $foo \in bar$ to mean the item foo in array bar, regardless of its key.

Algorithm 1 Basic GenI algorithm

```
 1: function GENERATOR(Grammar,InputSem)
 2:     Agenda ← LEXSELECTION(Grammar,InputSem)
 3:     Chart ← ∅
 4:     output ← ∅
 5:     step ← substitution
 6:     while step ≠ done do
 7:         aTree ← any tree ∈ Agenda
 8:         Agenda ← Agenda \ {aTree}
 9:         Chart ← CHARTINSERTION(Chart,aTree)
10:         for all cTree ∈ CHARTRETRIEVAL(Chart,aTree) do
11:             if step = substitution then
12:                 derived ← SUBSTITUTION(cTree, aTree) ∪ SUBSTITUTION(aTree, cTree)
13:             else
14:                 derived ← ADJUNCTION(cTree, aTree)
15:             end if
16:             ⟨complete, incomplete⟩ ← SUCCESSCHECK(derived,InputSem)
17:             Agenda ← Agenda ∪ incomplete
18:             output ← output ∪ complete
19:         end for                                          # end cTree loop
20:         if Agenda = ∅ then
21:             if step = substitution then
22:                 step ← adjunction
23:                 ⟨Chart, Agenda⟩ ← PARTITIONTREES(Chart)
24:             else
25:                 step ← done
26:             end if
27:         end if
28:     end while                                            # end step loop
29:     return output
30: end function
```

Algorithm 2 Helper functions for GenI algorithm

```
1: function LexSelection(Grammar,InputSem)
2:     selected ← ∅
3:     for all tree ∈ Grammar do
4:         if tree.semantics ⊆ InputSem then
5:             selected ← selected ∪ {tree}
6:         end if
7:     end for
8:     return selected
9: end function

10: function SuccessCheck(results,InputSem)
11:     complete ← ∅
12:     incomplete ← ∅
13:     for all tree ∈ results do
14:         if tree.substitutionSites = ∅ and tree.semantics = InputSem then
15:             complete ← complete ∪ {tree}
16:         else
17:             incomplete ← incomplete ∪ {tree}
18:         end if
19:     end for
20:     return ⟨complete, incomplete⟩
21: end function

22: function PartitionTrees(Chart)
23:     initial ← ∅
24:     auxiliary ← ∅
25:     for all tree ∈ Chart do
26:         if tree.substitutionSites = ∅ then          # only take complete trees
27:             if tree is an auxiliary tree then
28:                 auxiliary ← initial ∪ {tree}
29:             else
30:                 initial ← initial ∪ {tree}
31:             end if
32:         end if
33:     end for
34:     return ⟨initial, auxiliary⟩
35: end function

36: function ChartRetrieval(Chart,tree)    # these will be reimplemented in section 2.1.4
37:     return Chart
38: end function

39: function ChartInsertion(Chart,tree)
40:     return Chart ∪ {tree}
41: end function
```

Chapter 2

Optimising the GenI surface realiser

This chapter presents the work carried out during the DEA stage. In section 2.1 we describe four optimisations we introduced in the GenI generator. We detail in 2.2 the implementation of these optimisations and various other extensions to the generator. Finally, section 2.3 discusses the effect of the optimisations on the overall performance of the generator.

2.1 Optimisations

This thesis is mainly about four optimisations for the surface realiser that we designed and implemented. The two simplest deal with the combinatorics of adjunction. **Ordered adjunction** works similarly to ordered substitution (section 1.2.2) and imposes an order on the evaluation of adjunction sites in the tree. **Semantic filtering** on the other hand improves on the delayed adjunction mechanism by detecting and removing intermediary structures that cannot lead to a semantically complete result. The next optimisation, **polarity automata**, represents the bulk of this thesis. We use linguistic information (which has to be added to grammars) to identify and filter out the sequences of lexical items which cannot lead to a successful realisation. Finally, we use **chart sharing** to refine the polarity automata, thereby reducing the overhead it introduces.

2.1.1 Ordered adjunction

This optimisation has a similar effect to ordered substitution (section 1.2.2), that is, it prevents the generation of certain redundant derivations and intermediate structures for the same tree. As with ordered substitution, the adjunction sites of a given derived tree are tried in a given order. Note that in contrast to substitution sites, adjunction sites do not necessarily need to be used. Therefore, once all auxiliary trees have been considered for a given site, the next site will be considered for adjunction. Another difference with ordered substitution is that adjunction sites may be used more than once. To account for this, we repeat the adjunction attempts until no further adjunctions are possible.

2.1.2 Semantic filtering

One possible way to improve the delayed adjunction mechanism is to make better use of the semantics. As discussed in section 1.2.3, delayed adjunction means using separate substitution and adjunction phases, and only letting the trees without any remaining substitution sites into the adjunction phase. This filtering criterion could benefit from further tightening. To see this, consider again the example discussed in section 1.2.3. In this example, at the stage where

non-auxiliary trees	auxiliary trees
`np(man)`	`np(det(the),np*)`
`np(woman)`	`np(adj(beautiful),np*)`
`s(np(woman), v(rejects), np(man))`	`np(adj(heartless),np*)`
	`np(adj(smelly),np*)`
	`np(adj(dirty),np*)`

Figure 2.1: After the substitution phase in delayed adjunction

non-auxiliary trees	auxiliary trees
`s(np(woman), v(rejects), np(man))`	`np(det(the),np*)`
	`np(adj(beautiful),np*)`
	`np(adj(heartless),np*)`
	`np(adj(smelly),np*)`
	`np(adj(dirty),np*)`

Figure 2.2: Delayed adjunction with semantic filtering

adjunction starts to apply, the trees available to the generator are given in figure 2.1. Now note that of the three non-auxiliary trees, only one can yield a sentence covering the entire input semantics when combined with the auxiliary trees. The optimisation we now introduce aims at filtering the other two trees away; since their use cannot possibly lead to a successful realisation, such trees can be safely ignored. Concretely, we modify the function PartitionTrees in algorithm 2 to calculate the union of all auxiliary tree semantics, which we call ϕ_A, and delete from the agenda, any derived tree with semantics ϕ_s if $\phi_A \cup \phi_s \neq \phi_i$, where ϕ_i is the input semantics. This leaves behind only the trees which have a chance of fulfilling the input semantics via adjunction, in this case `s(np(woman), vp(rejects), np(man))`, as we see in figure 2.2. When combined with the auxiliary trees, this will indeed produce the sentence *the beautiful heartless woman rejects the dirty smelly man.*

2.1.3 Polarities

Natural language often permits many different ways to express the same idea. For example, we can write in the following table some possible expressions for the literals `gift(g)`, `cost(g,x)` and `high(x)`. [2]

`gift(g)`	`cost(g,x)`	`high(x)`
$\tau_{the\ gift}$	$\tau_{the\ cost\ of}$	$\tau_{is\ high}$
$\tau_{the\ present}$	τ_{costs}	τ_{much}

The table above shows some **lexical ambiguity** for each literal. As in the table, ambiguity can come from the availability of different lexical items to express the same literal, but it can also come from the many uses of a single word. In fact in a average-sized TAG, lexical ambiguity

[2] For simplicity reasons we here make the simplifying assumption that determiner and noun form a single lexical item. Note also that lexical items in a TAG can have multiple anchors so that for instance the initial tree for the predicative noun *costs* is anchored with both the noun *costs* and the preposition *of*.

is usually very high. For instance, the core TAG for French developed by Anne Abeille in Paris 7 counts an average of 150 lexical items for verbs.

We can think of generation partly as selecting a combination of lexical items, which when taken together, verbalise the entire input semantics. The number of possible combinations is $\prod_{1 \leq i \leq n} a_i$ with a_i the degree of lexical ambiguity for the ith literal, and n, the number of literals in the input semantics. So in the above example, we need to explore $2 \times 2 \times 2 = 8$ combinations. In other words, the size of the search space is exponential with respect to the number of literals.

The "polarity optimisation" we are now presenting is based on the observation that a large number of the initially possible combinations are syntactically invalid, i.e. cannot possibly lead to a successful realisation. For instance, although combinations like *the gift*, *the cost of*, *a lot* cover the entire semantics, they cannot be stitched together to form a complete expression. This leaves a considerable margin for an optimisation to detect and filter out such combinations ahead of time. Indeed, such an optimisation has been successfully put to work in a parser developed by Guillaume Bonfante, Bruno Guillaume and Guy Perrier for interaction grammars [Per02, BGP03]. The work we present here investigates how this idea can be transposed to the TAG-based surface realisation task.

The optimisation involves the addition to the grammar of some additional linguistic knowledge, the so called "polarities"; the construction and the pruning of a finite state automaton on the basis of this information; and the use of this automaton to filter out the necessarily invalid combinations. We now discuss each of these in turn.

2.1.3.1 Polarity keys

We augment our grammars with some explicit combinatoric knowledge. Each lexical item is associated with a set of **polarity keys** where each key is a label, and a positive or negative integer which we call a **charge**. Polarity keys allow trees to provide hints about which other trees they may combine with. Below, for instance, we assign to the trees τ_{costs} and $\tau_{is\ high}$ the keys $\{-1np\}$ meaning that these verb phrases "require" an np tree. To balance these, we assign $\{+1np\}$ to the noun phrase trees.

Any combination of lexical items which does not have a net charge of zero is necessarily syntactically invalid. For instance, *the cost of the gift much* has a charge of $+1np$, so it is clearly not a solution. In contrast, *the cost of the gift is high* has a charge of $0np$, which does not *guarantee* syntactic compatibility but suggests that the combination is worth exploring.

`gift(g)`	`cost(g,x)`	`high(x)`
$\tau_{the\ gift}$ +1np	$\tau_{the\ cost\ of}$	$\tau_{is\ high}$ -1np
$\tau_{the\ present}$ +1np	τ_{costs} -1np	τ_{much}

2.1.3.2 Automaton construction

Construction for a single polarity key The construction and use of the automata fits in between the lexical selection and the generation step. It requires a table like that in section 2.1.3.1 where columns are indexed by the literals of the input semantics, and rows contain the lexical items whose semantics includes the column index. Given such a table, a combination of lexical items covering the input semantics corresponds to a path across the table where we select exactly one row for every column. The finite-state automata we build is a way to represent the set of possible combinations and to associate with each of these combinations the polarity charge assigned to it by the grammar. We refer to these automata as **polarity automata**.

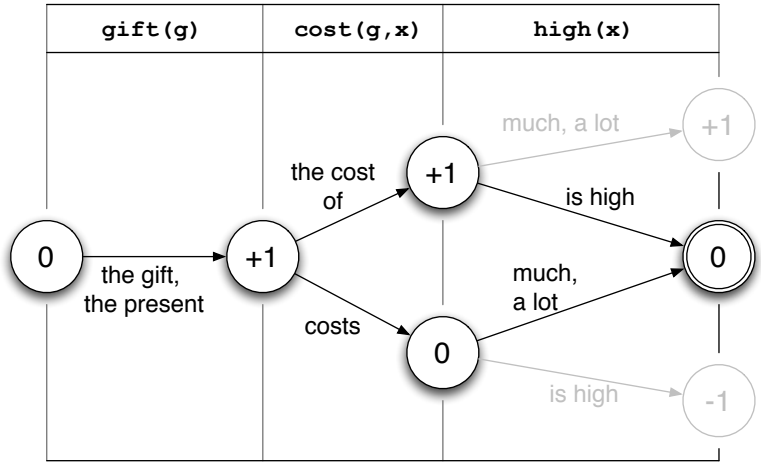

Figure 2.3: Simple polarity automaton (pruned)

name	type and use
N	the number of literals in the input semantics
$Labels$	the set of integers from 1 to N
LE	$(Labels \rightarrow Lex)$ a function mapping elements of $Labels$ to sets of lexical entries
Pol'	$(Lex \rightarrow Integers)$ a function mapping a lexical entry to a charge
$Automaton$	a polarity automaton $\langle States, InitialState, FinalStates, Transitions \rangle$
$PolKeys$	a set of polarity keys
Pol	$(Lex \times PolKeys \rightarrow Integers)$ a function mapping a lexical entry and a given polarity key to a charge
$LSem$	$(Lex \rightarrow \{Integers\})$ a function mapping a lexical entry to a subset of $Labels$

Figure 2.4: Special variables used in polarity automaton algorithms

Algorithm 3 presents a simple version of the automaton construction, in which we assume that we only have to deal with a single polarity key. Each state $\langle L, C \rangle$ in the automaton represents the set of paths up to the Lth literal that have a net charge of C. Any transitions in the automaton are from some state $\langle L, C_1 \rangle$ to $\langle L+1, C_2 \rangle$ and has as label, the name of one of the lexical items verbalising the literal L.

We initialise the automaton with a state $\langle 0, 0 \rangle$ that does not cover any literal. Then we augment the automaton one literal at time (lines 3.6 to 3.17) until reaching the last literal N. If we declare the final state of the automaton to be $\langle N, 0 \rangle$, and prune it (algorithm 4), we are left with a minimised automaton [HU79] that only represents the combinations with zero net charge. The final state automaton for the example discussed above (section 2.1.3.1) is given in figure 2.3.

Generalisation for multiple polarity keys We can generalise the optimisation to the use of several polarity keys. Each key acts as a linguistic filter, so the more keys we take into account, the more filtering we accomplish. One way to do this is to construct a polarity automaton for each polarity key and then intersect them. For greater efficiency, we instead perform the intersection

Algorithm 3 Construction of a simple polarity automaton

```
1: function SimpleAut(Key,LE,Pol,N)                          # Key ∈ PolKeys
2:     InitialState ← ⟨0, 0⟩
3:     FinalStates ← {⟨N, 0⟩}
4:     States ← ∅
5:     curStates ← {InitialState}
6:     for i, 0 < i < N do
7:         newStates ← ∅                                     # recently created items
8:         for all st ∈ newStates do
9:             ⟨_, _, c⟩ ← st
10:            for all lex ∈ LE(i) do                        # see figure 2.4
11:                newSt ← ⟨i, c + Pol′(lex)⟩
12:                newStates ← newStates ∪ {newSt}
13:                States ← States ∪ newSt
14:                Transitions ← Transitions ∪ {⟨st, lex, newState⟩}
15:            end for
16:        end for
17:        curStates ← newStates
18:    end for
19:    Automaton ← ⟨States, InitialState, FinalStates, Transitions⟩
20:    return PruneAut(Automaton)
21: end function
```

Algorithm 4 Automaton pruning

```
1: function PruneAut(Automaton)
2:     ⟨States, InitialState, FinalStates, Transitions⟩ ← Automaton
3:     nonfinal ← States \ FinalStates
4:     blacklist ← {st ∈ nonfinal s.t. HasNoTrans(st)}
5:     while blacklist ≠ ∅ do
6:         newBlacklist ← ∅
7:         for all s ∈ blacklist do
8:             for all ⟨p, l, s⟩ ∈ Transitions do
9:                 States ← States \ s
10:                Transitions ← Transitions \ {⟨p, l, s⟩}
11:                if HasNoTrans(p) then
12:                    newBlacklist ← newBlacklist ∪ {p}
13:                end if
14:            end for
15:        end for
16:        blacklist ← newBlacklist
17:    end while
18:    return ⟨States, InitialState, FinalStates, Transitions⟩
19: end function

20: function HasNoTrans(st)
21:    trans ← {⟨st, _, _⟩ ∈ Transitions}
22:    return trans = ∅
23: end function
```

incrementally, that is, we build each successive polarity automaton using the previous one as a skeleton.

This requires us to change the definition of states to a tuple $\langle L, C \rangle$ where C is no longer a single charge, but a list of them. (We actually augment it to the three-tuple $\langle L, V, C \rangle$ but this extra item can be ignored until section 2.1.3.3). The general architecture (algorithms 5 and 6) begins with a **seed automaton** that does not account for any polarity keys. We then iterate through the polarity keys, each time building an intersected automaton that accounts for every key up to the current one. Each iteration also includes a pruning step, which we saw in algorithm 4.

Algorithm 5 Polarity automaton construction with incremental intersection

```
 1: function PolAut(PolKeys,Lex,Pol,N,LE)
 2:     Automaton ← SeedAut(Lex,N,LE)
 3:     for all key ∈ PolKeys do
 4:         Automaton ← BuildAut(Automaton,key,Pol)
 5:         Automaton ← PruneAut(Automaton)
 6:     end for
 7:     return Automaton
 8: end function

 9: function BuildAut(Automaton,Key,Pol)                    # Key ∈ PolKeys
10:     ⟨_, oldInitial, oldFinal, oldTrans⟩ ← Automaton
11:     ⟨_, _, p⟩ ← oldInitial
12:     InitialState ← ⟨0, ∅, 0 : p⟩                  # ':' is a list concatenation operator
13:     FinalStates ← ∅
14:     for all oldst ∈ oldFinal do
15:         ⟨n, v, p⟩ ← oldst
16:         FinalStates ← FinalStates ∪ {⟨n, v, 0 : p⟩}
17:     end for
18:     cross ← {⟨InitialState, oldInitial, 0⟩}
19:     while cross ≠ ∅ do
20:         newCross ← ∅                                    # recently created items
21:         for all ⟨ost1, nst1, np1⟩ ∈ cross do
22:             for all ⟨ost1, lex, ost2⟩ ∈ oldTrans do
23:                 ⟨i, v, op2⟩ ← ost2
24:                 np2 ← np1 + Pol(lex, Key)          # calculate the polarity of the next state
25:                 nst2 ← ⟨i, v, np2 : op2⟩
26:                 newCross ← newCross ∪ {⟨ost2, nst2, np2⟩}
27:                 States ← States ∪ {nst2}
28:                 Transitions ← Transitions ∪ {⟨nst1, lex, nst2⟩}
29:             end for                                    # oldTrans loop
30:         end for                                        # cross loop
31:         cross ← newCross
32:     end while
33:     return ⟨States, InitialState, FinalStates, Transitions⟩
34: end function
```

Algorithm 6 Seed automaton construction algorithm

```
1: function SeedAut(Lex,N,LE,LSem)
2:     InitialState ← ⟨0, ∅, []⟩                          # '[]' indicates the empty list
3:     FinalStates ← {⟨N, ∅, []⟩}
4:     States ← {InitialState}
5:     Transitions ← ∅
6:     created ← {InitialState}
7:     for all lit, 1 < lit < N do
8:         newCreated ← ∅
9:         for all current ∈ created do
10:            ⟨newCreated, newTrans⟩ ← SeedAutHelper(current,Lex,LE,LSem)
11:            States ← States ∪ newCreated
12:            Transitions ← Transitions ∪ newTrans
13:        end for                                          # current loop
14:        created ← newCreated
15:    end for                                              # lit loop
16:    return ⟨States, InitialState, FinalStates, Transitions⟩
17: end function

18: function SeedAutHelper(current,Lex,LE,LSem)             # Simple version
19:    ⟨lit_cur, _, _⟩ ← current
20:    lit ← 1 + lit_cur
21:    next ← ⟨lit, ∅, []⟩
22:    newCreated ← {next}
23:    newTrans ← ∅
24:    for all lex ∈ LE(lit) do
25:        newTrans ← newTrans ∪ {⟨current, lex, next⟩}
26:    end for                                              # lex loop
27:    return ⟨newCreated, newTrans⟩
28: end function
```

2.1.3.3 Multiple literals semantics

One potential complication is that trees in realistic grammars may span more than one literal (consider $\tau_{is\ expensive}$, which covers both *cost(g,x)* and *high(x)*). Since these trees do not fit neatly into a single column of the automaton construction table, we repeat them for all the columns of the semantics they represent. But this introduces its own share of problems. Once we use a **multiliteral tree** for a literal, we must make sure that we use the same tree for the other literals that it covers, and make sure that its polarity charge is only taken into account once.

We do this by augmenting the states in the polarity automaton to be a three-tuple $\langle L, V, C\rangle$ where V contains any extra semantics information which is contributed by a multiliteral tree. Whenever we encounter a literal which is in the extra semantics of the state, we can remove that literal from the extra semantics and make an empty transition ϵ. This modification only requires a change to the construction of the seed automaton (algorithm 7) and lookup function which assigns no polarity keys to the empty transition.

Algorithm 7 Modified seed automaton construction algorithm

function SeedAutHelper($current$,Lex,LE,$LSem$) # Enhanced version
  $newCreated \leftarrow \emptyset$
  $newTrans \leftarrow \emptyset$
  $\langle lit_{cur}, v_{cur}, \_\rangle \leftarrow current$
  $lit \leftarrow 1 + lit_{cur}$
  for all $lex \in LE(lit)$ **do**
    $v_{lex} \leftarrow LSem(lex)$
    $v \leftarrow v_{lex} \cup v_{cur}$
    if $lit \in v_{lex}$ **then** # check extra semantics
      $v \leftarrow v \setminus \{lit\}$
      $lex \leftarrow \epsilon$
    end if
    $next \leftarrow \langle lit, v, []\rangle$
    $newTrans \leftarrow newTrans \cup \{\langle current, lex, next\rangle\}$
    $newCreated \leftarrow newCreated \cup \{next\}$
  end for
  return $\langle newCreated, newTrans\rangle$
end function

2.1.4 Chart sharing

The polarity automaton thus yields a compact representation of those combinations of lexical items that potentially lead to a successful realisation. Next it must be used to guide the surface realisation process. We explored two different ways to do this, which we now discuss.

The simplest approach is to collect a set of paths by walking the automaton, and to perform chart generation separately for each path, using the lexical items along that path as a basis for generation. The motivation behind the separation is to prevent mutually incompatible items from being compared with each other. However, this simple approach does not account for the fact that different paths may often have lexical items in common.

The second, more sophisticated approach consists in factorising these lexical items that are common to several paths. To this end, we annotate each tree with the set of paths on which it

appears. During generation, we only compare trees if they have some paths in common, that is if the intersection of their paths is non-empty. Any resulting derived tree is annotated with this intersection of paths. Since the number of paths is known, we can express these sets as bit vectors, using the bitwise-and operation to calculate the intersection. We can further improve on this concept by introducing the bit vector as a chart indexing scheme, where the chart is an associative array mapping a bit vector to a set of trees (algorithm 8). Retrieving trees from the chart consists of selecting the subset of keys which intersect a given tree's path set, and taking the union over the resulting sets of trees.

Algorithm 8 Chart operations with automaton path indexing

```
1: function ChartRetrieval(Chart,tree)                    # returns a set of trees
2:     selection ← ∅
3:     for all k ∈ Chart.keys do                          # Chart is an associative array
4:         if k ∩ tree.paths ≠ ∅ then
5:             selection ← selection ∪ Chart[k]
6:         end if
7:     end for
8:     return selection
9: end function

10: function ChartInsertion(Chart,tree)                   # returns a chart
11:     samePath ← Chart[tree.path]
12:     if samePath = NULL then
13:         samePath ← ∅
14:     end if
15:     Chart[tree.path] ← samePath ∪ {tree}
16:     return Chart
17: end function
```

2.2 Implementation

The optimisations just described were implemented in Haskell. Since we did not know the potential effect of optimisations, we had to pay particular attention to modularity, so that any or all of the optimisations could be enabled, with special care to account for all possible combinations of optimisations. In addition to the optimisations, we also implemented a few supporting features:

1. Corrections to the generator as bugs were revealed. This included minor subtleties (havoc caused by neglecting to sort a list) and conceptual oversights (for example, it was not originally foreseen that auxiliary trees could have substitution sites, or that adjunction nodes could be used multiple times).

2. A rewritten graphical user interface (figures A.1 and A.2). This was mainly motivated by a need to test the generator and to visualise the results (automata, trees, etc). The visualisation of trees and automata was made possible by the open source Graphviz tool.

3. A graphical debugger (figure A.3). As grammars grew and optimisation work continued, it became apparent that some means of inspecting the intermediary structures built during

generation would be necessary. The graphical debugger allows the user to step through the generator, gaining an understanding of trees being built and their movement through the agenda, chart and results.

4. A batch-processing system. This made it possible to run a set of examples over all possible combinations of optimisations and automatically tabulate the results.

2.3 Results

2.3.1 Methodology

The criteria we used to measure performance were

agenda size - the number of items that get added to agenda; note that we do not decrement this counter when a tree is removed from the agenda

chart size - the number of items that get added to the chart

comparisons - the number of times any two trees are compared for substitution or adjunction

time - the time needed for generation, excluding the lexical selection phase but including the automaton construction time.

The generator was compiled using the Glasgow Haskell Compiler (6.2) with -O2 level optimisation. It was run on a single G4 866 Mhz processor with Mac OS X as the operating system. Timing results are averaged over 100 iterations of the generator.

The optimisations evaluated are abbreviated as follows: **oadj** - orderded adjunction (section 2.1.1); **sfilt** - semantic filtering (section 2.1.2); **pol** - polarity automata (section 2.1.3); **c-shr** - chart sharing (section 2.1.4). If the polarity optimisation is used without chart sharing, we sum the results over each generation subtask.

2.3.2 Tests performed

We evaluated the performance of the generator on two input semantics with a different grammar for each input.

2.3.2.1 Lexical ambiguity

The **promettre** example tests the usefulness of polarity automata on a grammar with high lexical ambiguity. The input semantics is {`promise(e1, a, b, e2), jean(a), marie(b), convince(e2, a, d, e3), give(e3, d, c, e), indef(c), book(c), paul(d), claire(e)`}, which is realised in the sentence *jean promettre a marie de persuader paul de donner un livre a claire* (note: the generator does not perform inflection, hence the infinitive *promettre* instead of *promets*). For this example, the degree of lexical ambiguity in the grammar is as follows: 14 lexical items for *promettre*, 6 for *persuader*, and 8 for *donner*. Thus, the number of possible combinations is 672. With the polarity optimisation, the number of combinations actually explored by the generator is brought down to 3.

optimisations	agenda size	chart size	comparisons	time (ms)
none	66	51	2639	20.0
pol	69	39	585	30.0
pol c-shr	37	23	447	30.0
sfilt	53	51	2569	20.0
sfilt pol	43	39	493	30.0
sfilt pol c-shr	25	23	385	30.0
oadj	137	51	2635	20.0
pol oadj	158	39	590	40.0
pol c-shr oadj	100	23	442	30.0
sfilt oadj	72	51	2570	20.0
sfilt pol oadj	62	39	494	30.0
sfilt pol c-shr oadj	44	23	386	30.0

Since this example is focused on lexical ambiguity, we only comment here on the effect of the polarity and chart sharing optimisations. Using the polarity optimisation reduces the number of tree comparisons by three quarters and the chart size by a quarter. Without chart sharing, however, it increases the agenda size due to the lack of factorisation. This is remedied by adding chart sharing, in which case, the agenda size also decreases by more than a quarter.

2.3.2.2 Intersective modifiers

The **chatnoir** example examines the generator's performance against the problem of intersective modifiers. It has an input semantics {`chase(e1,a,b), cat(a), black(a), fierce(a), def(a), def(b), mouse(b)`} which is realised in the corresponding grammar by the sentence *le mechant chat noir chasser le souris*.

In the grammar used for this example, the lexical items *le*, *noir*, *mechant* are all auxiliary trees adjoining on to the trees they modify.

optimisations	agenda size	chart size	comparisons	time (ms)
none	121	13	3660	50.0
pol	87	5	2572	40.0
pol c-shr	87	5	2572	40.0
sfilt	93	13	3076	40.0
sfilt pol	61	5	2020	30.0
sfilt pol c-shr	61	5	2020	30.0
oadj	472	13	1988	40.0
pol oadj	326	5	1300	30.0
pol c-shr oadj	326	5	1300	40.0
sfilt oadj	332	13	1428	50.0
sfilt pol oadj	196	5	780	20.0
sfilt pol c-shr oadj	196	5	780	20.0

This table shows the following results.

First, we observe that semantic filtering has a strong decreasing impact on agenda size, chart size and comparisons. Specifically, it reduces agenda size and chart size by a quarter with respect to the null optimisation baseline. When combined with the polarity optimisation, it decreases

the chart size by 40% with respect to the polarity optimisation baseline, and when combined with polarity and chart sharing, it decreases it by the same amount.

Second, note that without ordered adjunction, there are 48 possible derivations for the result *le mechant chat noir chasser le souris*. Ordered adjunction removes many of these redundant derivations by bringing this number down to 10 (The 10 that remain represent different orders for the auxiliary trees $\tau_{mechant}$, τ_{le} and τ_{noir} to be adjoined at the node *chat*).

A secondary benefit of this reduction in spurious realisation is an important decrease in the number of comparisons made. In general, the decreases is of approximately 50% with respect to the corresponding baseline (54% with respect to the null optimisation baseline, 50% with respect to the polarity baseline and 46% with respect to the semantic filtering baseline). Note further that combining all three optimisations reduces the number of comparisons by 80%.

Finally note that the increase in agenda size is misleadingly inflated because we increment the counter for every adjunction site instead of every tree. Future work on evaluation will include a better evaluation criterion which accounts for this factor.

Chapter 3

Conclusion and future work

We presented a series of optimisations for a natural language surface realiser using a Tree Adjoining Grammar and a flat semantics. These optimisations dealt with two major sources of complexity for the generation problem. With polarity automata and chart sharing, we provided a means for coping with lexical ambiguity. With semantic filtering and ordered adjunction, we reduced the number of intermediary and final structures produced via adjunction. A more general judgement of how these optimisations affect the efficiency of the generator would require a large-scale evaluation based on the use of a reasonable sized TAG.

3.1 Further optimisation

3.1.1 Polarity signatures

Constructing and walking a polarity automaton can involve a great deal of overhead, especially considering that a realistic automaton would be handling hundreds of trees at any time. One possible solution for reducing the overhead would be to pre-process the lexically selected trees, and group together trees with identical semantics and polarity keys. The groups are labeled with a tuple of $\langle S, K \rangle$ where S is the tree semantics and K is the set of polarity keys that the tree holds. These tuples, which we call **polarity signatures**, could then be used instead of trees as the labels of the polarity automaton, the reasoning being that though there maybe hundreds of trees for a given lexical item, the number of distinct polarity signatures is likely to be smaller.

3.1.2 Grouped adjunction

Despite delayed adjunction, semantic filtering, and ordered adjunction, the use of auxiliary trees remains a challenge for generation. In the **chatnoir** test, for example, the sentence *le mechant chat noir chasser le souris* was derived in 10 different ways, each of these representing different orders for the adjunction of auxiliary trees.

A possible option to deal directly with the intersective modifiers problem is to identify auxiliary trees that adjoin to the same site, then we can group them into sets and instead of performing adjunction with these trees, we would use a placeholder that represents the entire group. When generation is complete, we can post-process the results to properly insert the modifiers. Even if the post-processing were to use a bottom-up algorithm (like the rest of the generation), this modification would still be useful because it would isolate the generation of intermediate structures to a single item. In the case of *The beautiful heartless woman rejects the dirty smelly man*, this would mean that *beautiful heartless woman* would be dealt with

separately from *dirty smelly man*. Instead of $4! = 24$ intermediary structures, we would only be dealing with $2! \times 2! = 4$. It might be possible to use a post-processing strategy that incorporates word-order constraints, so that *dirty smelly man* is generated, but not *smelly dirty man* which is awkward to native English speakers. This would further reduce the number of intermediate structures, and as an added bonus, produce higher quality sentences.

3.1.3 Distinguished chart indexing

[Kay96] proposes a scheme for using semantic indices as chart indices. Under this scheme, a **distinguished index** is arbitrarily selected for every tree among the semantic indices of that tree. The distinguished index is propagated during generation: whenever a tree is substituted or adjoined into another, the distinguished index of the derived tree is that of latter tree. If we consider a tree *John* with distinguished index `j` for semantics {`name(j,john)`} and a tree *smells* with the distinguished index `s` for semantics {`smells(s,j)`}. Substituting *John* into *smells* would result in the derived tree *John smells* with the distinguished index `s`.

Kay uses the distinguished index as a chart index. We had mentioned in section 1.1.2.1 that tree nodes are also associated variables which are unified against the arguments in the input semantics. We call these variables (after unification is done) **missing indices**. Given an agenda tree with a substitution site to fill, or an adjunction site to test, we only consider for combination the chart trees whose distinguished index is the same as the missing index of the agenda tree. This idea is not yet integrated into the GenI generator and would definitely be worth exploring.

3.2 Large scale testing

The generator has so far been tested on a small number of linguistic phenomena with a small grammar. For linguistic, applicative and evaluation purposes, it would be necessary to link the generator with a a full-sized grammar for French. Such a grammar can be produced with the help of the metagrammar compiler developed by Denys Duchier, Benoit Crabbe, Joseph Leroux and Yannick Parmentier. This metagrammar facilitates the semi-automated modular development of grammars. At present it is used to develop a core grammar for French and a grammar is already available which has a linguistic coverage comparable to Anne Abeille's TAG.

To integrate the GenI algorithm with the metagrammar compiler, we have developed a basic interface that reads the compiler output, but several open issues still need to be resolved, so as to fully support generation from the resulting grammars. In particular, the unification mechanism needs to be revised.

Once this grammar can be used by the GenI generator, full scale evaluation becomes possible. This evaluation can be achieved on the basis of a test suite, which can be written with the program *Annotator* developed by Marilisa Amoia. *Annotator* provides an environment in which semantic expressions may be associated with a set of natural language expressions. These can then be saved in a file, which could eventually be parsed and fed into the batch processing system for the surface realiser. This will require some consideration into the most appropriate means of displaying the test-suite results. For example, does a test fail if the generator does not produce all of the natural language expressions for an input semantics? What if it produces too many? Another issue for consideration is that the generator should identify the source of test failures whenever possible, for example, that a certain word is not in the grammar. Since GenI does not produce inflected forms, it would also be necessary to either get test-writers in the habit of writing uninflected expressions (*Marie detester Jean*), or more usefully, to pre-process

and de-inflect the Annotator files, or better still for the generator to produce inflected output (*Marie deteste Jean*).

3.3 Generation system

Once the generator reaches a satisfactory level of completeness and efficiency, it would become interesting to integrate it into a wider generation system. Such a system would perform not only surface realisation, but a fuller range of generation tasks from content planning (determining what the surface realiser should say), to applying the morphological finishing touches.

This system could itself be used and validated into user applications such as question-answering systems and interactive dialogue systems, the integration of which brings up a host of questions on its own. One topic of interest is how a generator should behave when producing a full text or a dialogue with the user. A sophisticated generator should be able to produce referring expressions in order to create more economical and natural sentences. In other words, it should be able to produce *Mary hates John. He smells funny.* instead of *Mary hates John. John smells funny.*

In short the tactical generation task that we explored in this thesis is only a small part of a vast pool of topics to be explored in natural language generation.

Appendix A

GenI generator screenshots

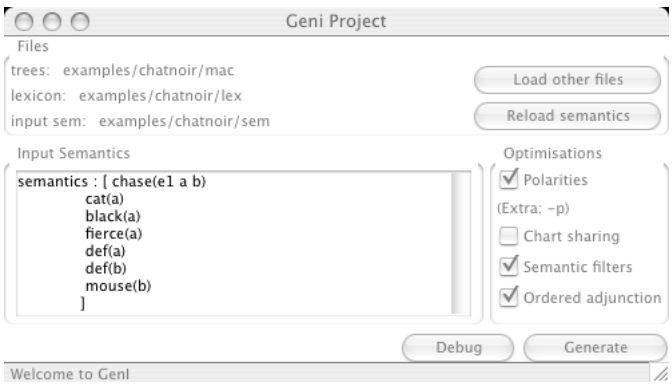

Figure A.1: Graphical interface for the GenI generator

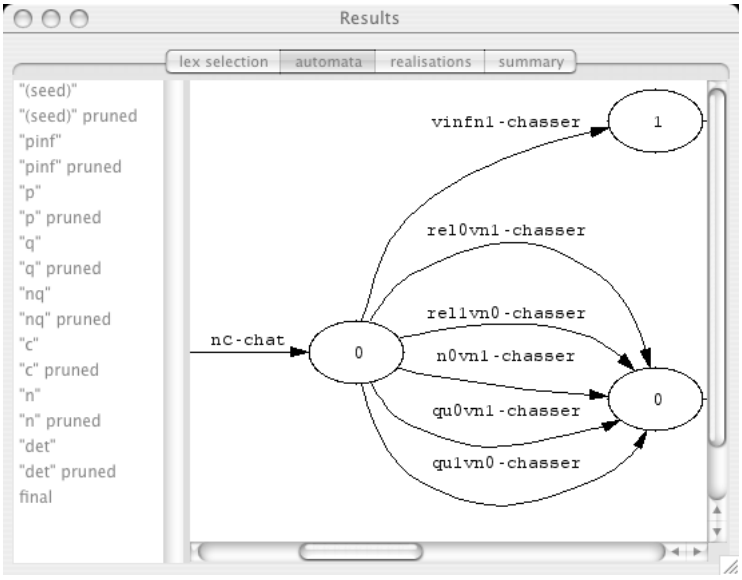

Figure A.2: Automaton visualisation

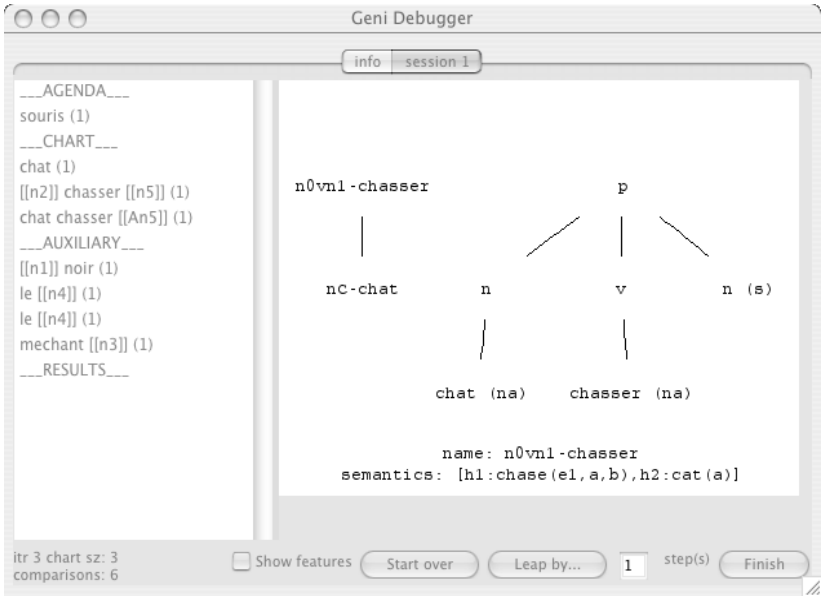

Figure A.3: Graphical debugger

Bibliography

[Are03] Carlos Areces. GeNI's NL Generator. `http://www.loria.fr/projets/geni/doc/30.09.03/clean-meta2.pdf`, 2003.

[BGP03] Guillaume Bonfante, Bruno Guillaume, and Guy Perrier. Analyse syntaxique électrostatique. *Évolutions en analyse syntaxique, Revue TAL (Traitement Automatique des Langues)*, 44(3), 2003.

[CCFP99] John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznanski. An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of the 7th European Workshop on Natural Language Generation (EWNLG'99)*, Toulouse, 1999.

[CFM+95] Ann Copestake, Dan Flickinger, Rob Malouf, Susanne Riehemann, and Ivan A. Sag. Translation using Minimal Recursion Semantics. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation (TMI95)*, Leuven, Belgium, 1995.

[GT01] Claire Gardent and Stefan Thater. Generating with a grammar based on tree descriptions: a constraint-based approach. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, Toulouse, 2001.

[HU79] John Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company, 1979.

[JS97] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

[Kay96] Martin Kay. Chart generation. In *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics*, pages 200–204, San Francisco, 1996. Morgan Kaufmann Publishers.

[KS02] Alexander Koller and Kristina Striegnitz. Generation as dependency parsing. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, 2002.

[Per02] Guy Perrier. Descriptions d'arbres avec polarités : les grammaires d'interaction. In *9ième Conférence annuelle sur le Traitement Automatique des Langues (TALN'02)*, Nancy, 2002.

[Shi88] Stuart M. Shieber. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619, Budapest, Hungary, 1988.

[SvNPM90] Stuart M. Shieber, Gertjan van Noord, Fernando C. N. Pereira, and Robert C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–41, 1990.

Abstract

The surface realisation task consists in using a grammar to produce the sentence(s) associated by this grammar with some input meaning representation. This task is known to be NP-complete, so that optimisation is an important aspect of any practical implementation. Starting from an existing surface realiser, this thesis proposes and implements four optimisation techniques, evaluates their effects and identifies directions for further improvement.